Master's Thesis in Aerospace Engineering:

Development of an AI environment in Matlab/Simulink for AOCS applications.

Supervisors:

Prof. BRACCO Giovanni
Eng. LAURENT Lorenzo

Candidate:

CAVALIERI Vincenzo

Academic Year 2022/2023

A mio padre che mi ha insegnato la dignità,
a mia madre che mi ha insegnato l'amore.

Abstract

The present work is intended as the final product of the double master's degree program between Politecnico di Torino and ISAE-Supaero, result of the final internship at CNES, the French space agency. The objective of this research was first and foremost to provide the CNES AOCS service with a bibliographic reference tool regarding Neural Networks and all their possible applications by understanding their mathematics and theory. It is then demonstrated how it is possible to reproduce such networks and their different configurations in the Matlab/Simulink environment through the development of an entire AI library in which each type of layer is represented with its characteristics, having Python as a reference. Each layer of neural network studied in depth in the state of the art (MLP, RNN, CNN) is therefore reproduced in respective Simulink models in which it is possible to load weights from Python of the corresponding network already trained through specific functions able to read .h5 and .json files. Once obtained the structure of the net, the loading of the weights in the Simulink models is done through configuration files coded according the OCEANS simulation environment, that is the main simulator used in the service for AOCS control. The aim is therefore to have a structure in Simulink that can reproduce the networks and use them in feedforward to make predictions useful for AOCS analysis. The training phase is therefore carried out in Python and is not reproduced in Simulink. In order to make possible the integration of this library with the AOCS environment all the models are coded in the OCEANS environment too. Such a library thus gives the service the ability to use any type of network by knowing its structure via Python files. The final part lastly focuses on validating these models by comparing the predictions between Python and Simulink of concrete cases through the use of networks already validated internally in the CNES service and from previous case studies. The whole work thus allowed us to fully explore and understand how neural networks work and their possible applications, how it is possible to reproduce their architecture in an environment different from Python by applying their mathematics and respecting their structure.

Keywords - Neural Networks, AOCS, Python, AI library, Matlab/Simulink

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence
AOCS Attitude and Orbit Control System
CFD Computational Fluid Dynamics
CNES Centre National d'Etudes Spatiales
CNN Convolutional Neural Network
GNC Guidance Navigation and Control
GRU Gated Recurrent Unit
LSTM Long Short-Term Memory
ML Machine Learning
MLP MultiLayer Perceptron
NaN Not a Number
NLP Natural Language Processing
OCEANS Outil de Création, d'Etude et d'ANalyse SCAO (Matlab simulation tool for AOCS mission analyses)
RNN Recurrent Neural Network

Introduction

We are witnessing in today's society a real revolution that touches all aspects of our lives: the advent of artificial intelligence. In today's global scientific environment, as well as in all the domains of our culture, artificial intelligence is imposing itself at a rapid pace, radically changing the way we see and perceive things, the way we learn and work, by setting new paradigms and presenting new tools that until a few decades ago were only pure science fiction. The application of AI and Machine Learning algorithms are proving to be highly useful and incredibly more efficient than the classical methods used until now in almost all fields of knowledge: ranging from medicine, finance, art, engineering and agriculture. Let us just think of the ability to recognise a cancer based on image processing, voice recognition, financial predictions, autonomous driving and much more. In the purely aerospace field, the application and mastery of such techniques is increasingly becoming useful and essential, especially when compared to the classical calculation and simulation tools, which in many cases are slow and expensive, i.e. CFD or finite model calculations. Especially when dealing with complex phenomena for which a precise and reliable mathematical model is not available, the use of Artificial Neural Networks turns out to be of great help as they are able to clearly distinguish non-linearities and recurring patterns that the classic analytic methods are unable to do. A deep and important look at the application of artificial intelligence in the aerospace field is provided by Stefano Silvestrini and Michèle Lavagna's article, 'Deep Learning and Artificial Neural Networks for Spacecraft Dynamics, Navigation and Control' [1]. In the development of the present work, this article was of fundamental importance in order to understand the different types of machine learning algorithms, methodologies and characteristics, but above all their possible and concrete applications in the AOCS/GNC field: for example, the identification of spacecraft dynamics by means of a fully connected network, the use of convolution networks for navigation and landing in space ('Vision-Based Navigation'), Reinforcement Learning and Meta-Reinforcement Learning for Adaptive Guidance and Control. Other important perspectives are provided in the papers 'Modelling of the dynamics of a gyroscope using artificial neural networks' [2] and 'Deep learning for autonomous lunar landing' [3]. The former represents an excellent application of neural networks, specifically a fully connected network, to learn very complex body dynamics for which a real mathematical model would be too complex to implement. Important here is also the learning phase that sees the real dynamic system available and based on the results from the latter, by minimising an appropriate loss function between the real system and the prediction, sees the network finally reproduce the full dynamic model. In 'Deep learning for autonomous lunar landing', instead, a deep insight in using convolutional neural networks is provided. The goal of the paper is to design a set of deep neural networks, i.e. Convolutional Neural Networks (CNN) and Recurrent Neural Net-works (RNN) which are able to predict the fuel-optimal control actions to perform autonomous Moon landing,

using only raw images taken by on board optimal cameras. Such approach can be employed to directly select actions without the need of direct filters for state estimation. Indeed, the optimal guidance is determined processing the images only. For this purpose, Supervised Machine Learning algorithms are designed and tested.Two possible scenarios are considered, i.e. 1) a vertical 1-D Moon landing and 2) a planar 2-D Moon landing. For both cases, fuel-optimal trajectories are generated.

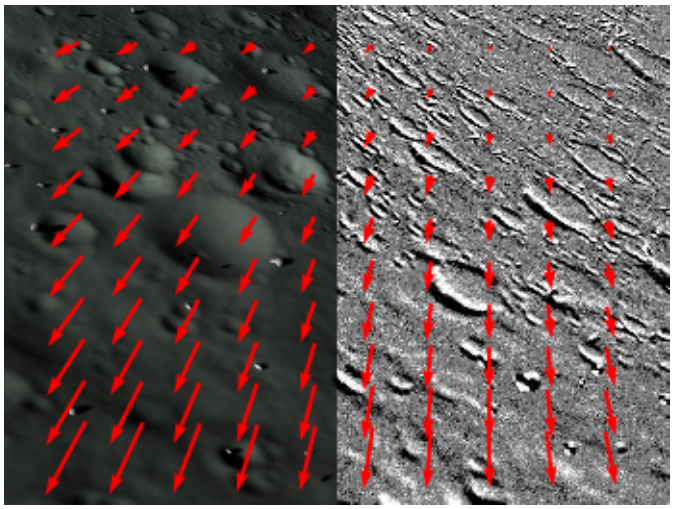


Figure 2: Vision-based navigation [4]

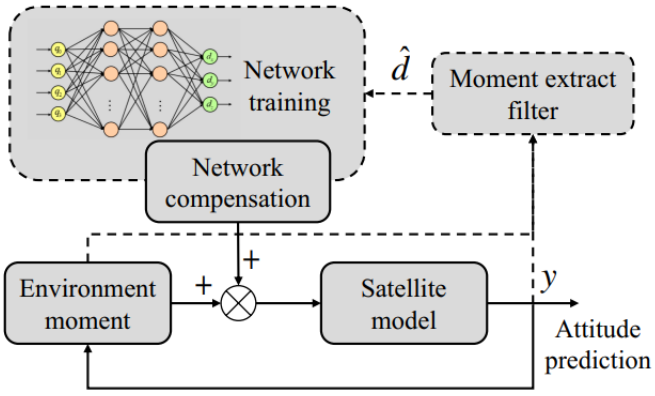


Figure 3: Attitude prediction system with Neural Networks compensation [5]

This work is therefore enscribed as part of CNES's program to create a Matlab/Simulink environment that could effectively represent an AI interface that can be used in the AOCS environment, especially for those cases in which the complexity of the system or the dynamics involved do not allow the use of classical calculation models. This study represents the continuation of a project previously conducted by the agency and aimed at analysing, by means of neural networks, the phenomena of propellant sloshing inside satellite tanks during agile manoeuvres [6]. The main objective was to use Machine Learning algorithms to predict torques and forces acting on the satellite structure and caused by the movement of the propellant, and subsequently compare this method with classical CFD. Once the torques and forces acting on the satellite are known, it is therefore possible to synthesise a controller that can alleviate the destabilisations caused by these

phenomena and reduce the relaxation time, improving pointing performance. The results coming out of the Machine Learning algorithms appropriately compared with those of CFD revealed a strong potential in terms of computation time and accuracy: this gave CNES the opportunity of developing its own AI environment in Matlab/Simulink that could be integrated into the AOCS simulators. In order to effectively be able to undertake the research work, it was initially necessary to carry out intensive state-of-the-art analysis in order to actually have a theoretical basis on which to develop the AI models in Simulink and then be able to compare their performance with Python results. The work therefore involved the creation of an entire AI library in Matlab/Simulink, which could be versatile and easy to use in order to be able to create the main types of neural networks for space applications using all the basic models.

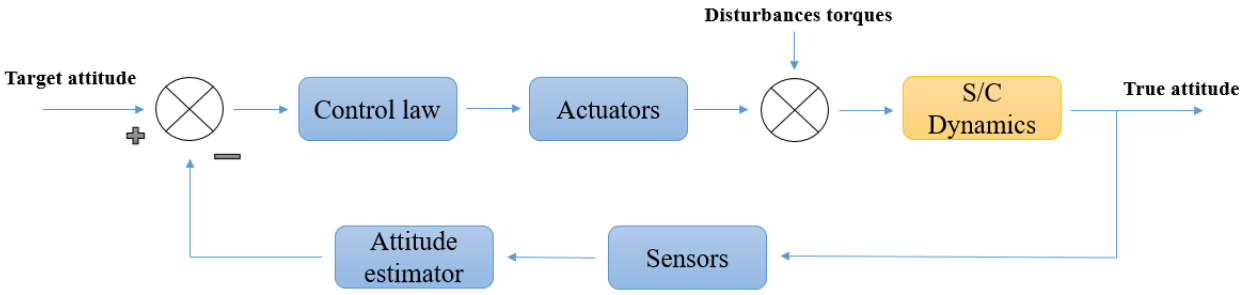


Figure 4: AOCS Control Loop

Context

This work was carried out at CNES, Centre National d'Etudes Spatiales, a public establishment whose role is to propose and develop a space programme for the French government. The establishment was founded in 1961 by President Charles de Gaulle in order to more effectively coordinate the country's space activities.

The various activities of CNES have led it to establish itself on the following sites:

- The agency's head office, located in Paris les Halles. This is where France's space policy is formulated and both national and international and international space programmes are coordinated. The site employs around 185 people.
- The Guiana Space Centre (CSG). The launch site in French Guiana employs 1,700 people to carry out lift-offs for launchers in the European range (Ariane range of launchers (Ariane 5, Vega and Ariane 6 in the near future), as well as the Russian Soyuz launcher.
- The launchers department at Paris Daumesnil, where 285 employees are responsible for future launchers.
- The Toulouse Space Centre (CST), the agency's largest technical and operational centre, with more than 1,700 CNES employees and 800 industrial partners. It is here that the design, production and operation of orbital systems takes place.

In particular, the present study was performed in the framework of the Toulouse Space Centre of CNES, more precisely in the AOCS Architecture department (DTN/DV/AS), belonging to the Flight Dynamics sub-directorate (DTN/DV), which is part of the Technical and Digital Directorate (DTN). Members of the department are involved in attitude control of orbital systems and atmospheric balloons during all phases of a space project. Their tasks are therefore very varied and include developing AOCS algorithms, maintaining and updating internal libraries and simulators, designing attitude control systems and, of course, operating orbital systems.

Chapter 1

State of the art

This academic project is centered around a meticulously crafted state of the art that stands as a forefront in both research and application within the extensive domain of Machine Learning, Artificial Intelligence, and Neural Networks. Its objective is to comprehensively showcase the key components comprising an Artificial Neural Network, including various existing network types and their practical implementations, alongside the diverse methodologies employed to deal with today's complex challenges. The project will cover conventional activation functions, followed by essential parameters delineating distinct layers and their initialization. Lastly, it will delve into prominent network types, exploring their attributes and architectures.

What is a Neural Network ?

Neural networks are computational frameworks that take cues from the structure and functioning of the human brain. Similar to the brain, they are comprised of interconnected units called **neurons**, organized into **layers**. These layers process input data, and each neuron undertakes a weighted calculation. Nevertheless, distinct differences exist. While the brain's neurons can handle diverse information concurrently, neural networks primarily excel at numerical data. Furthermore, brain neurons are intricate, utilizing electrochemical signals, whereas artificial neurons employ mathematical functions for computations. Learning within neural networks occurs through a technique called backpropagation. This involves adjusting internal weights based on the comparison between the network's output and the desired outcome. This mirrors the brain's process of strengthening or weakening connections between neurons. In essence, neural networks derive inspiration from the brain's architecture, yet they are simplified constructs tailored for efficient processing of specific data types. While backpropagation is based on the propagation of error from the output to the input of the network in order to optimise the network's calculation parameters, i.e. the weights, the process that actually allows predictions to be made is called feedforward. During this process, the network, already trained and optimised as a result of the backpropagation process, is thus able to obtain from a specific data input, the prediction output with an accuracy that typically depends on the training parameters. In fact, in the context of suprvised learning, there are various techniques for training a network and managing the data through which it is trained, in order to obtain accurate predictions on the basis of already verified data.A neural network, thus more or less long or complex, is today able to provide answers where exact mathematical

models are not available, to learn complex dynamics and to be in some cases even computationally lighter and easier to interpret.

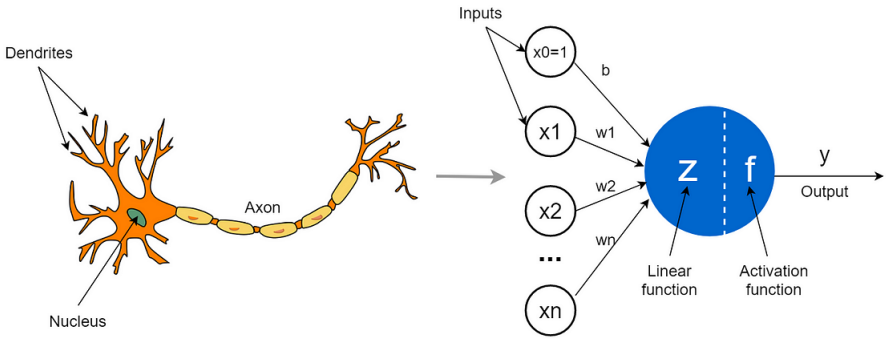


Figure 1.1: Neurons comparaison [7]

1.1 Activation functions

An activation function in AI is a mathematical function applied to the output of a neuron in an artificial neural network. It determines whether a neuron should be activated (send a signal) or not, based on the weighted sum of its inputs. Activation functions introduce non-linearity to the network, allowing it to learn and represent complex relationships in data. Activation functions are used according to their specific characteristics, which can be useful at different stages in the implementation of a neural network:

- Non-linearity
- Differentiable everywhere
- Identity in 0
- Monotonic
- Range
- Monotonic derivative

Each of these features makes it possible to exploit these functions appropriately to meet different criteria in terms of network's training speed, convergence, and overall performance.

1.1.1 Sigmoid and Tanh

All logistic functions have the property of representing the whole number line in a small interval, for example between 0 and 1, or between -1 and 1. One of the uses of a sigmoid function is therefore to convert a real value into a value that can be interpreted as a probability. The mathematical expression of a sigmoid function is : $f(x) = \frac{1}{1+e^{-x}}$. Logistic functions are an important elements of a logistic regression model. This is a modification of linear regression for two-class classification, and

converts one or more real-valued inputs into a probability, such as the likelihood of a customer buying a product. The final stage of a logistic regression model is often set to the sigmoid function, enabling the model to produce a probability.

• **Logistic regression** [8]: logistic regression is a statistical model used to determine whether an independent variable has an effect on a binary dependent variable. This means that there are only two potential outcomes for an input. Other forms of regression analysis, such as linear regression, require the definition of a threshold to distinguish binary classes. Linear regression allows a probability to be established, but this must then be applied to a logistic regression to establish a distinct classification. A commonly used model is a sigmoid function. In a sigmoid function, the outputs lie between the limits 0 and 1.

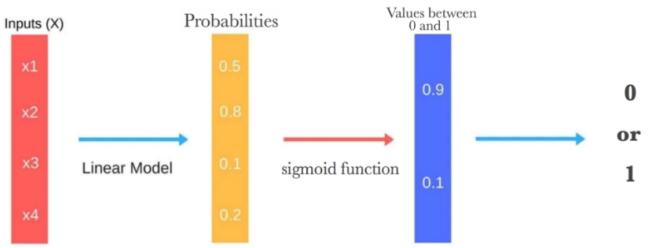


Figure 1.2: Logistic regression

• **Tanh**: the main difference between the sigmoid function and the tanh function is that the tanh function pushes the input values to 1 and -1 instead of 1 and 0. Both functions belong to the S-type functions that limit the input value to a certain band: this allows the network to keep its weights limited and avoid the problem of gradient explosion, where the value of the gradients becomes very large.

An important difference between the two functions is the behavior of their gradients: we observe that the gradient of the tanh is four times greater than the gradient of the sigmoid function. This means that using the tanh activation function leads to higher gradient values during training and larger updates of the network weights. So, if we want high gradients and large learning steps, we need to use the tanh activation function. Another difference is that the output of tanh is symmetrical around zero, enabling faster convergence.

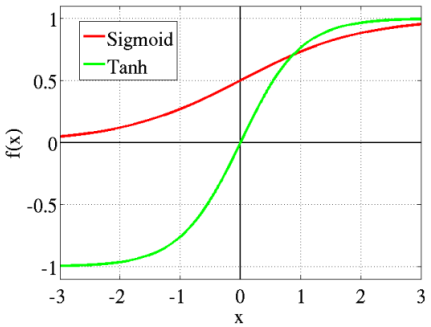


Figure 1.3: Sigmoid - Tanh [9]

1.1.2 Softmax

For multinomial logistic regression, i.e. logistic regression associated with multi-class problems where more than 2 outputs are possible, the softmax activation function is certainly more appropriate. The softmax function takes as input a vector z of K real numbers and normalises it to a probability distribution made up of K probabilities proportional to the exponentials of the input numbers. After applying softmax, each component of the input vector will be in the interval (0,1) so that they can be interpreted as probabilities. The softmax function is defined as follows:

$$\sigma(Z)_i = \frac{\exp^{Z_i}}{\sum_{j=1}^{K} \exp^{Z_j}},\ i = 1, ..., K\ et\ z = (z_1, ..., z_K) \in R^K \tag{1.1}$$

In other words, it applies the standard exponential function to each element z_i of the input vector z and normalises these values by dividing them by the sum of all these exponentials.

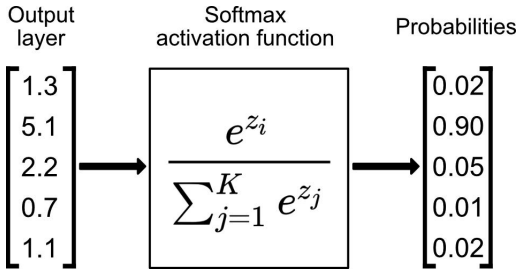


Figure 1.4: Softmax [10]

1.1.3 ReLu

It is frequently used as an activation function because it is easy to calculate, particularly its derivative: f(x)=max(0,x). A disadvantage of the ReLU function is that its derivative becomes zero when the input is negative, which can impede gradient back propagation. We can then introduce a version called Leaky ReLU defined by: $f(x) = max(\epsilon x, x)$ where $\epsilon \in]0, 1[$. The derivative is then equal to ϵ when x is strictly negative, which preserves the weight update of a perceptron using this activation function.

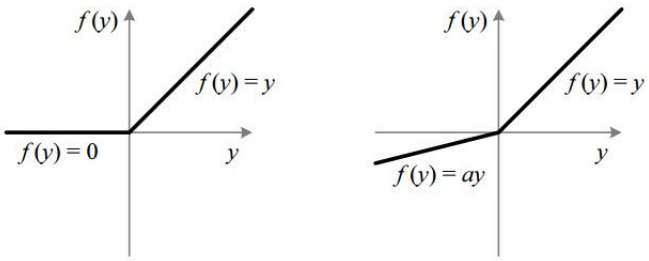


Figure 1.5: ReLu - Leaky ReLu [9]

1.1.4 PReLu

A parametric rectified linear unit, or PReLU, is an activation function which generalises the traditional rectified unit with a slope for negative values:

$$f(x) = \begin{cases} ax, x < 0 \\ x, x >= 0 \end{cases} \tag{1.2}$$

1.1.5 Elu

The Exponential Linear Unit is a derivative of the ReLU. It uses an exponential for the negative part instead of a linear function:

$$f(x) = \begin{cases} \alpha(\exp^x - 1), x < 0 \\ x, x >= 0 \end{cases} \tag{1.3}$$

1.1.6 Softplus

SoftPlus is a soft approximation of the ReLU function and can be used to force the output to always be positive: $f(x) = ln(1 + e^x)$

Summary

The following table summarises all the most frequently used activation functions in AI and their specific properties [11]:

Function	Derivative	Range	Continuity	Monotone	Smooth	Id.in 0
Sigmoid	$f' = f(x)(1 - f(x))$	[0,1]	C^∞	Yes	No	No
Tanh	$f' = 1 - f(x)^2$	[-1,1]	C^∞	Yes	No	Yes
ReLU	$f'(x) = \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases}$	R_+	C^∞	Yes	Yes	Yes
PReLU	$f'(x) = \begin{cases} \alpha, x < 0 \\ 1, x \geq 0 \end{cases}$	R	C^0	Yes	Yes	Yes
ELU	$f'(x) = \begin{cases} f(x) + \alpha, x < 0 \\ 1, x \geq 0 \end{cases}$	$[-\alpha, +\infty[$	$C^1 si\ \alpha = 1$	Yes	Yes	Yes if $\alpha \approx 1$
Softplus	$f'(x) = \frac{1}{1+e^{-x}}$	R_+	C^∞	Yes	Yes	No

Table 1.1: Activation funcions properties

The softmax function is different from the previous ones because it has as input and output a vector, $R^n \rightarrow R^n$, so its derivative corresponds to the following Jacobian matrix [12]:

$$J_{softmax} = \begin{pmatrix} \frac{\partial s_1}{\partial z_1} & \cdots & \frac{\partial s_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial s_n}{\partial z_1} & \cdots & \frac{\partial s_n}{\partial z_n} \end{pmatrix}$$

1.2 Weights initialisation: Xavier Glorot

Neural network models are configured using an optimisation algorithm called *Stochastic Gradient Descent*, which progressively changes the weights of the network in order to minimise a *Loss Function*, in the expectation of obtaining a set of weights capable of making useful predictions. This optimisation algorithm requires a starting point in the space of possible weights values from which to begin the optimisation process. Weights initialization is a procedure that sets the weights of a neural network to small random values that define the starting point for optimizing the neural network model.

- **Glorot**: the current standard approach for initializing the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called 'glorot' or 'xavier' initialization. The Glorot initialisation method is calculated as a random number with a uniform probability distribution (U) between $-\frac{1}{\sqrt{n}}$ and $\frac{1}{\sqrt{n}}$, where n is the number of inputs to the node.
- **Normalized Glorot**: the normalised Xavier initialisation method is calculated as a random number with a uniform probability distribution (U) between $-\sqrt{\frac{6}{n+m}}$ and $\sqrt{\frac{6}{n+m}}$, where n is the number of inputs to the node (for example, the number of nodes in the previous layer) and m is the number of outputs from the layer (for example, the number of nodes in the current layer).

1.3 Biases initialisation

It is possible and common to initialise the biases to zero, since the asymmetry breaking is provided by the small random numbers in the weights. For ReLu nonlinearities, it's liked by some to use a small constant value such as 0.01 for all biases, as this ensures that all ReLu units trigger at the beginning and therefore get and propagate a certain gradient. However, it's not clear if this provides a consistent improvement (in fact, some results seem to indicate that performance is worse) and it's more common to simply use a bias initialization of 0.

1.4 Types of Neural Networks

Neural networks have revolutionized the field of machine learning by drawing inspiration from the human brain's intricate neural connections. Among the diverse types of neural networks, three prominent architectures stand out:

- **Multi-Layer Perceptron (MLP)**: A foundational architecture, MLP consists of interconnected layers of nodes, each node applying a weighted sum and an activation function. This type is adept at solving a wide range of tasks, from simple regression to complex classification problems.
- **Recurrent Neural Network (RNN)**: Designed to capture sequential dependencies, RNNs utilize recurrent connections that allow information to loop back. They excel in tasks like natural language processing, time series analysis, and speech recognition, where context and order matter.
- **Convolutional Neural Network (CNN)**: Highly effective for image and spatial data analysis, CNNs employ specialized convolutional layers to automatically detect features and patterns within images. This type has significantly advanced object recognition, image generation, and computer vision applications.

These three types represent a fraction of the neural network landscape, each tailored to specific data structures and problem domains. As technology evolves, neural networks continue to drive innovations across various industries, showcasing their immense potential for creating intelligent systems.

1.4.1 MLP: Multilayer Perceptron

The multilayer perceptron is a type of artificial neural network structured in several dense layers. The dense layer is the basic layer for defining the architecture of a neural network: its role is to apply, using an appropriate activation function, a transformation to the input quantity, which can be a vector or a tensor. In a dense layer, all the neurons are connected to those in the previous layer, which is why it is also called 'Fully Connected' (FC).

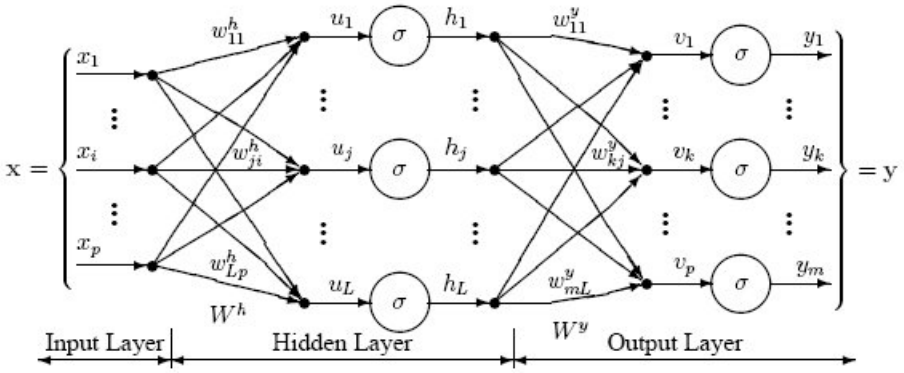


Figure 1.6: MLP [13]

From a broadly general point of view, a neural network can be used in two ways: **Feedforward** and **Backpropagation**. Feedforward in neural networks refers to the flow of data through the network in a single direction, from the input layer

to the output layer. In this type of architecture, each layer's output serves as the input to the next layer, without any loops or cycles. It's the fundamental architecture used in most neural networks, where information is processed sequentially through layers of interconnected nodes, with each node applying a set of weights and biases to the input it receives. This process eventually leads to the generation of predictions or classifications at the output layer based on the learned patterns in the data.

Backpropagation is a training algorithm used in neural networks to adjust the weights and biases of the connections between nodes in order to minimize the error between predicted and actual outputs. It involves two main phases: the forward pass and the backward pass. During the forward pass that we already discussed, input data is passed through the network layer by layer, producing predictions at the output layer. The error between these predictions and the actual target values is then calculated. In the backward pass, the error is propagated backward through the network from the output layer to the input layer. The algorithm calculates how much each weight and bias contributed to the error. This information is used to adjust the weights and biases in a way that reduces the error in subsequent iterations. This adjustment is typically done using **gradient descent**, which involves finding the direction and magnitude of the steepest descent in the error space. Backpropagation essentially fine-tunes the network's parameters to improve its accuracy over time. It iteratively updates the weights and biases based on the calculated gradients, gradually improving the network's ability to make accurate predictions on new data.

The behaviour of a Neural Network is determined by its **Hyperparameters**: these are parameters whose values control the learning process and determine the values of the model parameters that a learning algorithm eventually learns. Hyperparameters in a neural network can be classified according to the following criteria:

- Net structure
- Learning and Optimisation
- Regularization effect

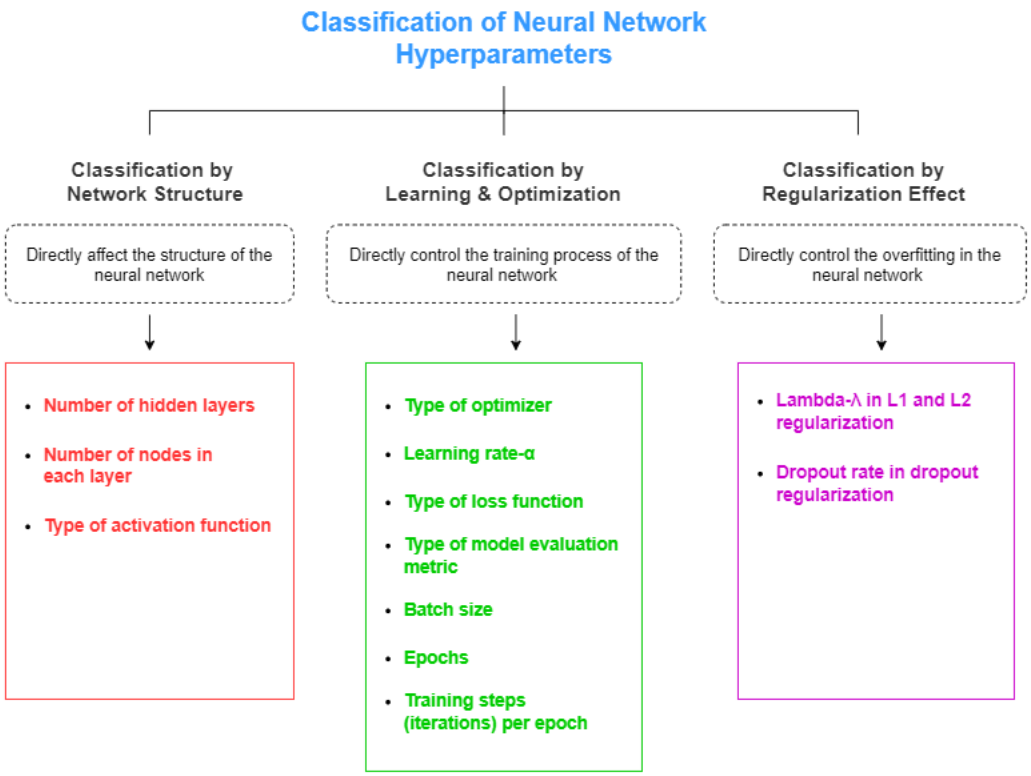


Figure 1.7: Neural Networks Hyperparameters [14]

1.4.1.1 Hyperparameters defining the network structure

- **Number of hidden layers**
  This is known as the depth of the network. The number of hidden layers determines the learning capacity of the network: in order to learn all the important non-linear patterns in the data, the neural network must have a sufficient number of hidden layers. A very small number of hidden layers generates a smaller network that may be under-suited to the training data. This type of network does not learn the complex patterns in the training data, nor does it perform well in predicting unknown data. Too many hidden layers result in a larger network that risks overfitting the training data. This type of network attempts to store the training data instead of learning patterns from the data. As a result, this type of network does not generalise well to new, unpublished inputs.

  Over-fitting is not as bad as under-fitting, as the former can be reduced or eliminated using an appropriate regularisation method.

- **Number of nodes (neurons/units) in each layer**
  This is also known as the width of the network. The number of hidden units is another factor that affects the learning capacity of the network. Too many hidden units create very large networks which can cause overfitting of the training data and a very small number of hidden units create smaller networks which can cause underfitting. In most cases, the number of neurons depends on the size of the input vector and the type of problem we want to solve. Still on the subject of the structure of the network, while the choice of the number of neurons and the activation function is arbitrary and depends

on the accuracy and computation time requirements of the output solution, the definition of the size of the weights and biases matrices requires particular attention because it depends on the parameters defined above. Thus, if we imagine that we are in the first layer of the network [1], the initialization will be of the type : $W^{[1]} = [N_{inputs}, N_{outputs}]$ and $B^{[1]} = [N_{outputs}]$, where $N_{outputs}$ corresponds exactly to the number of neurons in the layer. The operation in layer [1] will therefore be as follows:

$$U^{[1]} = W^{[1]} \cdot X + B^{[1]} = \sum_{i}^{n} w_i x_i + B^{[1]} \tag{1.4}$$

Ex: with $N_{inputs} = 2$ and 3 neurons in the layer: $(3,1) \leftarrow (3,2) \cdot (2,1) + (3,1)$
Once the input values have been correctly weighted using the matrices defined above, the activation function appropriately transforms the output by normalising it and improving its spatial visualisation: $H^{[1]} = f_{activation}(U^{[1]})$.

- **Type of activation function**

To sum up:

- The number of neurons in the input layer is equal to the number of input variables in the processed data.
- The number of neurons in the output layer is equal to the number of outputs associated with each input.
- In artificial neural networks, hidden layers are required if and only if the data must be separated non-linearly.

How many layers and neurons?

A single-layer neural network can only be used to represent linearly separable functions. This means very simple problems where, for example, the two classes of a classification problem can be clearly separated by a line. If your problem is relatively simple, a single-layer network may suffice. Most of the problems we want to solve are not linearly separable. It's easy to find in the literature how an MLP is able to overcome this limitation of linearity: in fact, a multi-layer perceptron can be used to represent convex regions. This means that, in effect, they can learn to draw shapes around examples in a high-dimensional space that can separate and classify them, overcoming the linear separability limitation. As there are no precise rules for determining the correct number of layers or units in a network to solve the problem posed, different methods can be more or less effective:

- Experimentation
- Intuition
- Literature on the problem to be solved
- Having several layers is, in general, always better

1.4.1.2 Hyperparameters defining training and optimisation

So far, we've described how the network works in feedforward, so we can now concentrate on training and after backpropagation [15] First, let's define the difference between sample, batch and epoch [16].

- A **sample** is a single line of data: it contains inputs that are fed into the algorithm and an output that is used to compare with the prediction and calculate an error. A training dataset is made up of several lines of data, i.e. several 'samples'.
- The **size of the batch** is a hyperparameter that defines the number of samples to be processed before updating the parameters of the internal model. A training dataset can be divided into one or more batches, made up of different samples. When all the training samples are used to create a batch, the learning algorithm is called batch gradient descent. When the batch is the size of a sample, the learning algorithm is called stochastic gradient descent. When the size of the batch is larger than a sample and smaller than the size of the training dataset, the learning algorithm is called mini-batch gradient descent.
  Batch size is another important hyperparameter: a larger size generally requires a lot of computing resources per epoch, but fewer epochs to converge. A smaller size does not require many computation resources per epoch, but does require many epochs to converge.
- The number of **epochs** is a hyperparameter that defines the number of times the learning algorithm runs through the training data set. An epoch means that each sample in the training data set has had the opportunity to update the internal parameters of the model. An epoch is made up of one or more batches. The number of epochs is traditionally high, often in the hundreds or thousands, allowing the learning algorithm to run until the model error has been sufficiently minimised.

For example, if we have 2000 examples of training data and the batch size is set at 20, it takes 100 iterations to complete 1 epoch.

- **Type of optimiser**
  The optimiser is also known as an optimisation algorithm. The task of the optimiser is to minimise the loss function by updating the network parameters. Gradient descent is one of the most popular optimisation algorithms and has three variants:batch gradient descent, stochastic gradient descent and mini-batch gradient descent. All these variants differ in the size of the batch we use to calculate the gradient of the loss function. Others are for example: gradient descent by momentum, Adam etc.
- **Learning rate** α
  This hyperparameter can be found in any optimisation algorithm. During optimisation, the optimiser takes small steps to move down the error curve. The learning rate refers to the size of the step. It determines the speed at which the optimiser moves down the error curve. The direction of the step is determined by the gradient (derivative). This is one of the most important hyperparameters in neural network training.

- **Type of loss function**
  The loss function is used to calculate the loss score (error) between predicted and actual values. Our aim is to minimise the loss function using an optimiser. This is what we do during the training course. The type of loss function to be used during training depends on the type of problem we have: MSE, MAE, MAPE and Huber Loss are used more for regression problems, while Log Loss, Categorical Cross-entropy and Sparse Categorical Cross-entropy are useful for binary classification and multi-classification problems.

1.4.1.3 Hyperparameters defining regularization effects

- **Regularization effect**
  Regularisation techniques reduce the risk of overfitting a neural network by limiting the range of values that the weights within the network can hold: more precisely, it modifies the loss function of the result, which in turn modifies the weight values produced.
  L1 regularization: the effect of L1 regularisation on the weight values of the neural network is that it penalises weight values close to 0 by making them equal to 0. Negative weight values are also set to 0. Thus, if a weight value is -2, under the effect of L1 regularisation, it becomes 0. The general intuition of L1 regularisation is that, if a weight value is close to 0 or very small, it is negligible in terms of the overall performance of the model, so making it equal to 0 does not affect the performance of the model and may reduce the memory capacity of the model. L1 penalises the sum of the absolute values of the weights: $MSE + \lambda \sum_{j=1}^{n} |w_i|$.
  L2 regularization: L2 regularisation also penalises weight values. For small values of the weights as well as for relatively large values, the L2 regularization transforms the values into a number close to 0, but not quite 0. L2 penalizes the sum of the square of the weights: $MSE + \lambda \sum_{j=1}^{n} |w_i^2|$.

- **Dropout**
  Dropout is a technique that involves ignoring randomly selected neurons during training. They are randomly 'dropped'. This means that their contribution to the activation of downstream neurons is temporally suppressed during the forward pass and weight updates are not applied to the neuron during the reverse pass. As a neural network learns, the weights of the neurons adapt to their context within the network. The weights of neurons are adapted to specific characteristics, giving them a degree of specialisation. Neighbouring neurons come to rely on this specialisation which, if taken too far, can result in a fragile model that is too specialised for the training data it was trained on. This dependence of a neuron on the context during training is called *complex coadaptation*. We can imagine that if neurons are randomly eliminated from the network during training, other neurons will have to step in and manage the representation needed to make predictions for the missing neurons. This is thought to result in the network learning multiple independent internal representations.The network thus becomes less sensitive to the specific weights of the neurons. The result is a network capable of better generalisation and less likely to overfit the training data.

Backpropagation: the backpropagation is a supervised learning method used by neural networks to update the parameters in order to make the network's predictions more correct. The parameters optimisation process is carried out using the gradient descent algorithm. The objective is to minimise the error:

$$E(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(y, \widehat{y}) \tag{1.5}$$

where **m** is the number of training examples (samples) and **L** is the error/loss incurred when the model predicts $\widehat{y}$ instead of the actual value y. A loss function must have two characteristics: it must be continuous and differentiable at each point. This is achieved by drifting E with respect to the parameters and adjusting weights and biases in the opposite direction of the gradient (this is why the optimisation algorithm is called 'gradient descent'). The loss function, L, is defined according to the task to be performed. In supervised learning, there are two main types of loss functions — these correlate to the 2 major types of neural networks: regression and classification loss functions [17].

- For classification problems, **Cross-Entropy** (also known as Log Loss) and **Hinge Loss** are appropriate loss functions. Specifically, in **Binary Classification** models, where the model takes in an input and has to classify it into one of two pre-set categories, the Cross-Entropy is used:

  $$CELoss = \frac{1}{n} \sum_{i=1}^{n} -(y_i \cdot log(p_i)) + (1 - y_i) \cdot log(1 - p_i) \tag{1.6}$$

  In binary classification, there are only two possible actual values of y, 0 or 1. Thus, to accurately determine loss between the actual and predicted values, it needs to compare the actual value (0 or 1) with the probability that the input aligns with that category (p_i = probability that the category is 1; 1 — p_i = probability that the category is 0). In cases where the number of classes is greater than two, we utilize *Categorical Cross-Entropy* which this follows a very similar process to binary cross-entropy.

  $$Categorical\ CELoss = -\frac{1}{n} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{ij} \cdot log(p_{ij}) \tag{1.7}$$

  Hinge Loss, instead, is a commonly used loss function for training in binary classification tasks. The hinge loss function encourages to maximize the margin between the decision boundary and the closest data points, while penalizing points that are misclassified or lie within the margin. The advantages include margin maximization, robustness to outliers, and sparsity of the resulting model. However, hinge loss is non-smooth and non-differentiable, which can make it difficult to optimize using some numerical optimization methods, and the choice of regularization parameter can have a significant impact on the performance of the model. For an intended output of $t = \pm 1$ and a classifier score y, the hinge loss of the prediction y is defined as

  $$L(y) = max(0.1 - t \cdot y) \tag{1.8}$$

- Regarding regression problems Mean Square Error (MSE) and Mean Absolute Error (MAE) are appropriate loss functions for these tasks. MSE finds the average of the squared differences between the target and the predicted outputs:

$$MSE = \frac{1}{n}\sum_{i=1}^{N}(y^i - \widehat{y}^i)^2 \tag{1.9}$$

This function has numerous properties that make it especially suited for calculating loss. The difference is squared, which means it does not matter whether the predicted value is above or below the target value; however, values with a large error are penalized. MSE is also a convex function with a clearly defined global minimum — this allows us to more easily utilize gradient descent optimization to set the weight values. However, one disadvantage of this loss function is that it is very sensitive to outliers; if a predicted value is significantly greater than or less than its target value, this will significantly increase the loss.

MAE finds the average of the absolute differences between the target and the predicted outputs.

$$MAE = \frac{1}{n}\sum_{i=1}^{N}|y^i - \widehat{y}^i| \tag{1.10}$$

This loss function is used as an alternative to MSE in some cases. As mentioned previously, MSE is highly sensitive to outliers, which can dramatically affect the loss because the distance is squared. MAE is used in cases when the training data has a large number of outliers to mitigate this.
To gather the advantages of both MAE and MSE another loss function has been created: the **Huber Loss** function

$$L_\delta = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\ |(y-\hat{y})| < \delta \\ \delta((y-\hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases} \tag{1.11}$$

If the absolute difference between the actual and predicted value is less than or equal to a threshold value, δ, then MSE is applied. Otherwise — if the error is sufficiently large — MAE is applied.

Moreover, during the backpropagation process the **learning rate** α is the hyperparameter that controls how much the model changes in response to the estimated error each time the model weights are updated. Choosing the learning rate is difficult, as too low a value may result in a long learning process that is likely to stall, while too high a value may result in learning a sub-optimal set of weights too quickly or in an unstable learning process.

Having defined the learning rate and the loss function, we can see how the parameters are updated in the network architecture:

$$w_{t+1} = w_t - \alpha\frac{\partial E(w)}{\partial w} = w_t - \alpha\frac{\partial L(w)}{\partial w} \rightarrow \Delta w_{t+1,t} = -\alpha\frac{\partial E(w)}{\partial w} \tag{1.12}$$

$$b_{t+1} = b_t - \alpha\frac{\partial E(b)}{\partial b} = b_t - \alpha\frac{\partial L(b)}{\partial b} \rightarrow \Delta b_{t+1,t} = -\alpha\frac{\partial E(b)}{\partial b} \tag{1.13}$$

where $\mathbf{t}$ is the learning step and α is the learning rate. The chain rule is then applied to calculate the derivative of the loss function with respect to the weights of the layer whose parameters are to be updated:

$$\frac{\partial L(w)}{\partial w} = \frac{\partial L(w)}{\partial H} * \frac{\partial H(w)}{\partial U} * \frac{\partial U(w)}{\partial w} \tag{1.14}$$

Let's better define this last step with a simple exemple. We need to figure out a way to change the weights so that the cost function improves. Any given path from an input neuron to an output neuron is essentially just a composition of functions; as such, we can use partial derivatives and the chain rule to define the relationship between any given weight and the cost function. We can use this knowledge to then leverage gradient descent in updating each of the weights. Let's define our cost function to simply be the squared error. In order to minimize the difference between our neural network's output and the target output, we need to know how the model performance changes with respect to each parameter in our model. In other words, we need to define the relationship (partial derivative) between our cost function and each weight:

$$\frac{\partial J(\theta)}{\partial \theta_1} \quad - \quad \frac{\partial J(\theta)}{\partial \theta_2} \tag{1.15}$$

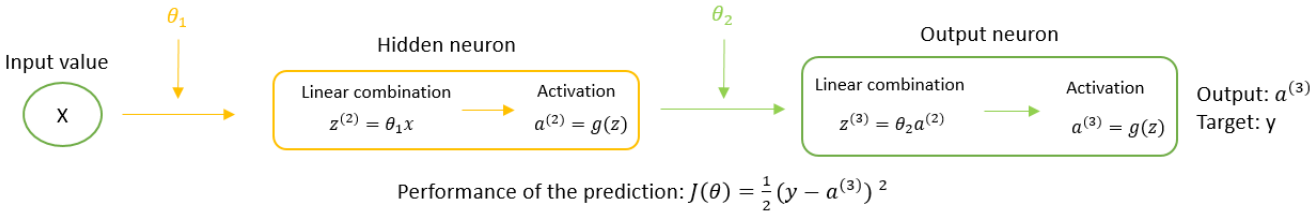


Figure 1.8: Parameters of the network to be updated

By easily applying the chain rule we can trace all intermediate partial derivatives by reconstructing the learning process from the input:

$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \theta_1} \tag{1.16}$$

$$\frac{\partial J(\theta)}{\partial \theta_2} = \frac{\partial J(\theta)}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z} \frac{\partial z}{\theta_2} \tag{1.17}$$

1.4.2 CNN: Convolutional Neural Network

A Convolutional Neural Network (ConvNet/CNN) is a deep learning algorithm capable of taking an image as input, assigning importance (weights and biases that can be learned) to various aspects/objects in the image, and differentiating between them. CNNs are based on the fact that the input data consists of images. One of the main differences is that the CNN layers are made up of neurons organised in three dimensions, the spatial dimension of the input (height and width) and the depth (channels). Colour images have several channels, generally one for each colour channel, such as red, green and blue. From a data point of view, this means that a single image provided as input to the model is actually made up of three images: if we wanted to use a classic neural network we would have to put all the pixels that make up each plane into a vector, but the images can even reach dimensions of 7680x4320. **The advantage of convolution networks is that they reduce images to a form that is easier to process, without losing the features that are essential for good prediction.**

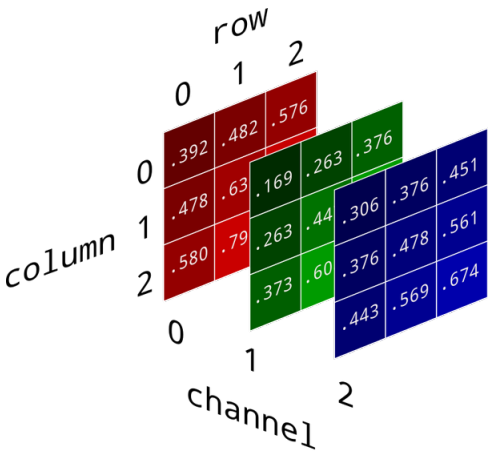


Figure 1.9: RGB image [18]

In their concrete applications Convolutional Neural Networks (CNNs) are widely used in various practical environments due to their ability to process and analyze visual data effectively. Some practical cases, other than image classification, where CNNs are commonly used include:

- Object Detection: CNNs can detect and locate multiple objects within an image, making them valuable in applications like self-driving cars, surveillance, and robotics.
- Facial Recognition: CNNs are employed in facial recognition systems for tasks like identifying individuals in photos or videos.
- Medical Imaging: CNNs aid in analyzing medical images like X-rays, MRIs, and CT scans, assisting doctors in diagnosing diseases and detecting anomalies.
- Natural Language Processing (NLP): While primarily used for computer vision, CNNs can also be combined with recurrent neural networks for processing text and sentiment analysis in NLP tasks.

- Autonomous Vehicles: CNNs play a crucial role in self-driving cars by processing visual data from cameras and sensors to understand the vehicle's surroundings.
- Art Generation: CNNs have been used to create artworks, apply artistic styles to photos, and generate realistic images from scratch.
- Video Analysis: CNNs are used in video analysis tasks like action recognition, object tracking, and video summarization.

The basic functionality of this network can be divided into four main parts [19]:

- **The Input layer** contains the pixel values of the image. E.g. 200x200x3, i.e. 200 pixels by 200 pixels with 3 colour channels, e.g. red, green and blue. The Input layer of a CNN expects, thus, a tensor x^l with $x^l \in \mathbb{R}^{H^l \times W^l \times D^l}$ where l is the $l-th$ layer. The triplet (i^l, j^l, d^l) refers to one element in x^l, which is in the $d^l - th$ channel, and at the spatial location (i^l, j^l) (at the $i^l - th$ row, and $j^l - th$ column).
- **The Convolution layer** will determine the output of neurons that are connected to local regions of the input by calculating the dot product between their weights and the region connected to the input volume. The element involved in the convolution operation in the first part of a convolution layer is called the kernel/filter, K. The kernel moves over the image each time it performs an element multiplication operation between K and the portion of the image over which the kernel is passing. The displacement is performed with a Stride value that indicates how much the filter moves over the underlying image after each matrix multiplication. If we choose K as a 3x3x1 matrix equal to:

$$\begin{matrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{matrix}$$

We'll have:

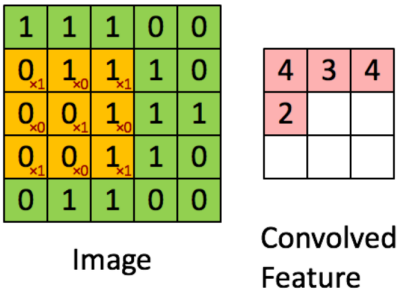


Figure 1.10: Convolution operation [20]

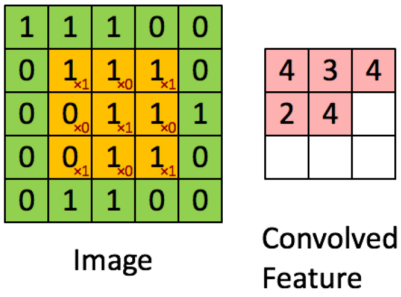


Figure 1.11: Kernel stride of 1 [20]

A filter of size F x F applied to an input containing C channels is a volume F x F x C that convolves an input of size I x I x C and produces an output feature map (also called an activation map) of size O x O x 1.
NB: the number of channels in the filter must be the same as the number of channels in the input image.

Size of convolution output: given I the length of the input volume, F the length of the filter, P the amount of zero padding, S the stride, the output size O of the feature map along this dimension is given by:

$$O = \frac{I - F + P_{start} + P_{end}}{S} + 1 \tag{1.18}$$

NB: in most cases $P_{start} = P_{end} = 2P$, where P_{start} and P_{end} designate the number of zeros at the beginning and end of the rows or columns of the matrix.

The purpose of the convolution operation is to extract high-level features, such as edges, from the input image. CNNs do not necessarily have to be limited to a single convolution layer. By convention, the first convolution layer is responsible for capturing low-level features such as edges, colour, gradient orientation etc. With additional layers, the architecture also adapts to high-level features, giving us a network that has a global understanding of the images in the dataset, just as we would. In precise mathematics, the convolution procedure can be expressed as an equation [21]:

$$y_{i^{l+1},j^{l+1},d} = \sum_{i=0}^{H} \sum_{j=0}^{W} \sum_{d^l=0}^{D^l} f_{i,j,d^l,d} \times x^l_{i^{l+1}+i,j^{l+1}+j,d^l} \tag{1.19}$$

Where l is the number of the layer; H, W, D^l are respectively the row, column and $l-th$ channel and the operation stride is equal to 1. The equation (1.19) is repeated for all $0 \leq d \leq D = D^{l+1}$, and for any spatial location (i^{l+1}, j^{l+1}) satisfying $0 \leq i^{l+1} \leq H^l - H + 1 = H^{l+1}$, $0 \leq j^{l+1} \leq W^l - W + 1 = W^{l+1}$. In the equation $x^l_{i^{l+1}+i,j^{l+1}+j,d^l}$ refers to the element of x^l indexed by the triplet $(i^{l+1} + i, j^{l+1} + j, d^l)$. A bias term b_d is usually added to $y_{i^{l+1},j^{l+1},d}$.

The operation generates two types of result: the dimensionality of the convolved feature is reduced relative to that of the input, while the dimensionality is increased or remains unchanged. To achieve this, a valid padding is applied in the first case, or an identical padding in the second.

Zero-padding: the process of adding zeros to each side of the input boundaries. Zero padding occurs when we add a border of pixels all having the value zero around the edges of the input images. This adds a sort of band of zeros around the outside of the image, hence the name 'zero padding'.

There are two categories of padding:
The first is referred to as **'valid'**. This simply means that there is no fill. If we specify a valid fill, this means that our convolutional layer will not fill at all and our input size will not be maintained.
The other type of fill is called **'same'**. This means that we want to fill the original input before convolving it so that the size of the output is the same as that of the input. The formula used in this case to calculate the number of layers of zeros to be added to the edges of the image to obtain a convolution output of the same size as the input is as follows: $P = \frac{F-1}{2}$ where F is the filter size.
NB: this formula guarantees that the input and output have the same size only if the stride value is S=1.

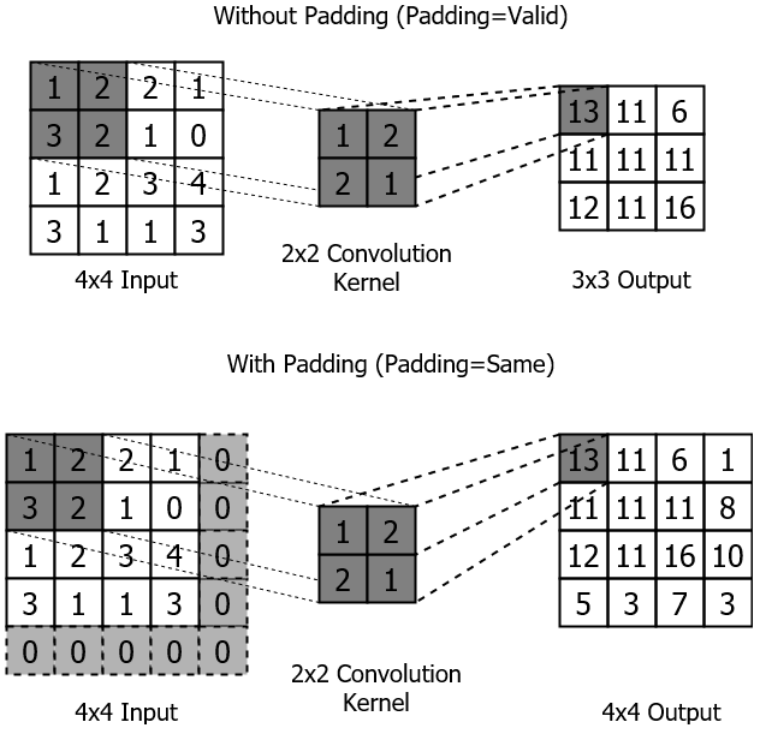


Figure 1.12: Padding operation

- **Pooling layer**: the pooling operation therefore consists of reducing the size of the images, while preserving their important features. As with convolution, a kernel of appropriately defined size scans the convolution output image at a defined step (always the stride) and retains the maximum (Max-Pooling) or average (AvPooling) value for each local window. If we assume an input $x^l \in \mathbb{R}^{H^l \times W^l \times D^l}$ to the $l-th$ layer, which is now a pooling layer, and also that H divides H^l and W divides W^l and the stride equals the pooling spatial extent, the output of pooling will be an order 3 tensor of size

$H^{l+1} \times W^{l+1} \times D^{l+1}$ [21], with:

$$H^{l+1} = \frac{H^l}{H}, \quad W^{l+1} = \frac{W^l}{W}, \quad D^{l+1} = D \tag{1.20}$$

A pooling layer operates upon x^l channel by channel independently. Within each channel, the matrix with $H^l \times W^l$ elements are divided into $H^{l+1} \times W^{l+1}$ nonoverlapping subregions, each subregion being $H \times W$. Each subregion is then transformed into a number. In precise mathematics,

$$Max : y_{i^{l+1},j^{l+1},d} = \max_{0 \le i < H, 0 \le j < W} x^l_{i^{l+1} \times H + i, j^{l+1} + \times W + j, d} \tag{1.21}$$

$$Average : y_{i^{l+1},j^{l+1},d} = \frac{1}{HW} \sum_{0 \le i < H, 0 \le j < W} x^l_{i^{l+1} \times H + i, j^{l+1} + \times W + j, d} \tag{1.22}$$

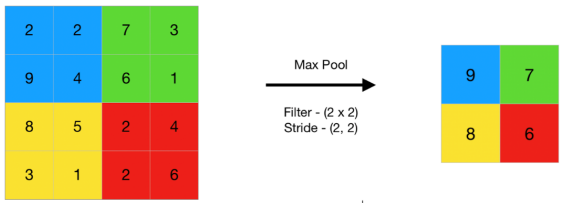


Figure 1.13: Max Pooling operation [22]

NB: there are other pooling methods than those mentioned above.

- **The fully connected layers** then perform the same tasks as standard ANNs and attempt to produce class scores from the activations, which will be used for classification. It is also suggested that ReLu be used between these layers to improve performance. With this simple transformation method, CNNs are able to transform the original input layer by layer using convolution and subsampling techniques to produce class scores for classification and regression purposes. First, we 'flatten' the output of the convolution layers. For example, if the final feature maps have a dimension of 4x4x512, we flatten them to an array of 8192 elements.

To sum up:

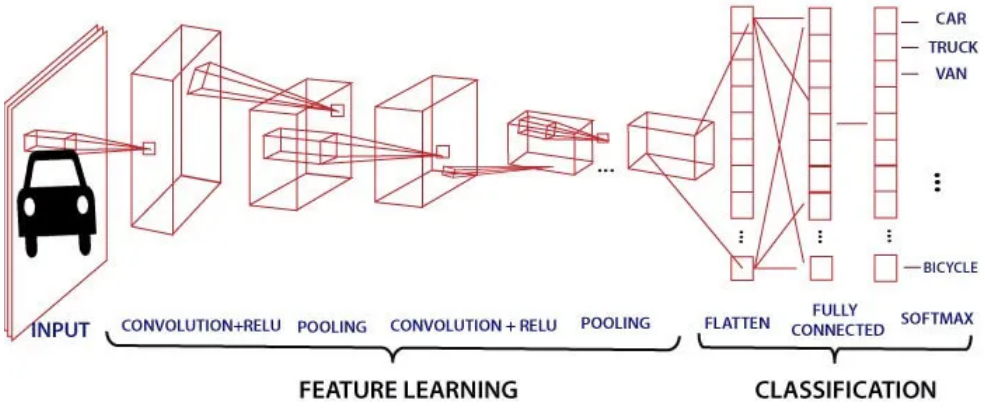


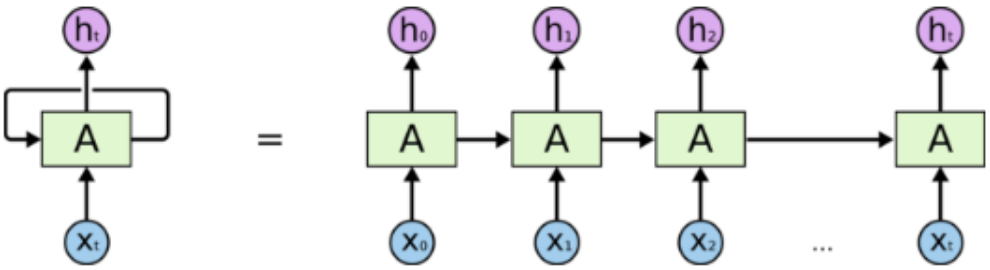
Figure 1.14: CNN network overview [23]

1.4.3 RNN: Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of artificial neural network that uses sequential data or time series. These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), speech recognition and image captioning. Like feedforward and convolutional neural networks (CNNs), recurrent neural networks use training data and are distinguished by their 'memory', as they use information from previous inputs to influence the current input and output. Whereas traditional deep neural networks assume that inputs and outputs are independent of each other, the outputs of recurrent neural networks depend on previous elements of the sequence. Recurrent networks are also distinguished by the fact that they share parameters between each layer of the network. Whereas feedforward networks have different weights for each node, recurrent neural networks share the same weight parameter in each layer of the network. That said, these weights are always adjusted by backpropagation and gradient descent processes to facilitate learning.

RNNs use the backpropagation over time (BPTT) algorithm to determine gradients, which is slightly different from traditional backpropagation because it is specific to sequential data. The principles of BPTT are the same as those of traditional backpropagation, but it differs in that it adds up the errors at each time step, whereas feedforward networks do not need to add up the errors since they do not share the parameters across each layer.

During this process, RNNs tend to encounter two problems, known as *Gradient Explosion* and *Vanishing Gradient*. These problems are defined by the size of the gradient, which is the slope of the loss function along the error curve. When the gradient is too small, it continues to decrease, updating the weight parameters until they become insignificant, i.e. 0. When this happens, the algorithm stops learning. Gradient explosion occurs when the gradient is too large, creating an unstable model. In this case, the model weights become too large and are eventually represented by NaN.



An unrolled recurrent neural network.

Figure 1.15: RNN [24]

Weights extraction for an RNN cell

Obtaining and extracting the weights for a recurrent cell as in the case of LSTMs and GRUs was not as immediate as in the case of the dense layer of a fully connected network. The python method **model.get_weights()**, in fact, in the case of RNNs returns a tensor that is not easy to interpret due to the complexity of the cell. Knowing in fact that each port behaves as a neural network in its own right, i.e. as a dense layer, it is necessary to understand which exact values of the weights correspond to the specific port in order to obtain the same prediction result in output. By analysing the source code of TensorFlow, in particular the Keras library in which the AI environments of neural networks are developed [25], it was possible to study the structure of this tensor and thus be able to break down its values in order to distribute them in the corresponding gates. As it is also mentioned in following Kaggle repository [26] the output tensor of the get_weights() in python is composed by:

$$\textbf{[[W:lstm\_kernel, U:lstm\_recurrent\_kernel, B:biases]]} \tag{1.23}$$

One of the two dimensions of these tensors is therefore 4*units, where units is the number of neurons, i.e. also the size of the cell output, and 4 is the number of gates in the cell:

- **W** is calculated as 4*features*LSTMoutputDimension
- **U** is calculated as 4*features*LSTMoutputDimension*LSTMoutputDimension
- **b** is calculated as 4*features*LSTMoutputDimension

Each of three tensor contains weights for the four gates in this order:

[[i (input), f (forget), c (cell state) and o (output)]]

$$W = [[W_i, W_f, W_c, W_o]]$$
$$U = [[U_i, U_f, U_c, U_o]]$$
$$B = [[B_i, B_f, B_c, B_o]]$$

Therefore, in order to extract each single weight the following Python code has been used:

```
W = model.layers[0].get_weights()[0]
U = model.layers[0].get_weights()[1]
b = model.layers[0].get_weights()[2]

W_i = W[:, :units]
W_f = W[:, units: units * 2]
W_c = W[:, units * 2: units * 3]
W_o = W[:, units * 3:]

U_i = U[:, :units]
U_f = U[:, units: units * 2]
U_c = U[:, units * 2: units * 3]
U_o = U[:, units * 3:]

b_i = b[:units]
b_f = b[units: units * 2]
b_c = b[units * 2: units * 3]
b_o = b[units * 3:]
```

Figure 1.16: Python code to extract recurrent weights

By analysing the source code of TensorFlow on GitHub [25], in particular the Keras library in which the AI environments of neural networks are developed, it was possible to analyse the structure of this tensor and thus be able to break down its values in order to distribute them in the corresponding gates.

Stacked RNN cells

The Stacked LSTM is an extension of the single layer model that has multiple hidden LSTM layers where each layer contains multiple memory cells: stacking LSTM hidden layers makes the model deeper, more accurately earning the description as a deep learning technique. By incorporating extra hidden layers into a Multilayer Perceptron (MLP) neural network, its depth is increased. This empowers the network to grasp intricate features and representations from input data in a hierarchical manner. This progression can be likened to moving from simple elements like lines to more intricate ones like shapes, and finally to advanced concepts such as objects. Moreover, a deeper network allows for more efficient function approximation compared to a shallower network with a single hidden layer. This advantage stems from the fact that fewer neurons are required in deeper networks, leading to quicker training and enhanced optimization of representations [27].

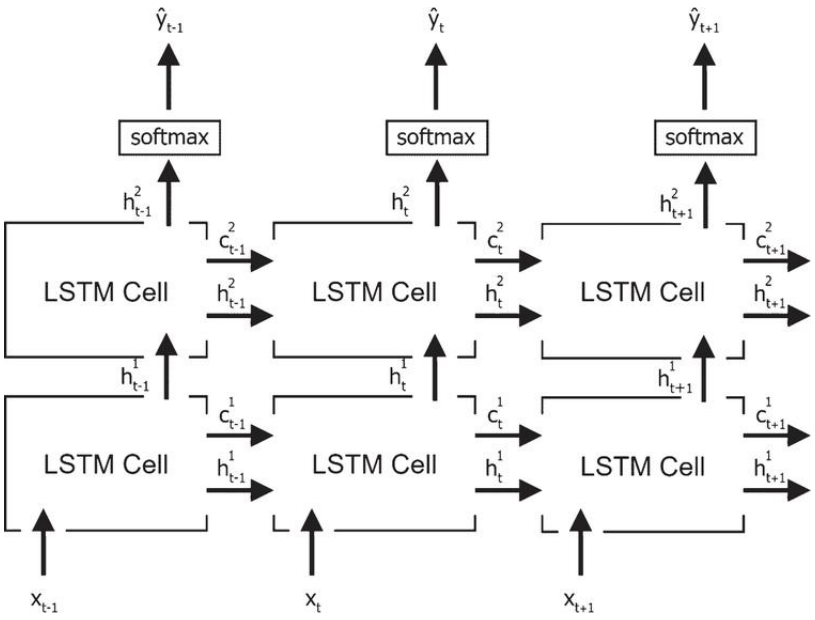


Figure 1.17: Stacked LSTM cells [28]

Given that LSTMs operate on sequence data, it means that the addition of layers adds levels of abstraction of input observations over time. In effect, chunking observations over time or representing the problem at different time scales. In the case of stacked cells the layer above provides a sequence output rather than a single value output to the layer below. Specifically, one output per input time step, rather than one output time step for all input time steps. For this, it is important to specify the option **'return_sequences=True'** for the above layers: in this case the layer produces the hidden state and cell state for every timestep in the input data, thus preserving the temporal relationships between the input timesteps. In the case of **'return_sequences=False'** the RNN layer will only return the last hidden state output h_t.

1.4.3.1 LSTM

LSTM networks were specifically developed to solve the long-term dependency problem faced by recurrent RNNs, in particular the 'vanishing gradient problem' mentioned earlier, through the intuition of creating an extra module in a neural network that learns when to remember and when to forget relevant information. In other words, the network effectively learns what information might be needed later in a sequence and when that information is no longer needed. [29]
Firstly, the output of an LSTM at a given time depends on three elements:

- The network's current long-term memory - known as the *'cell state'*.
- The output at the previous point in time - known as the *'previous hidden state'*.
- Input data at the current time step

LSTMs use a series of 'gates' that control how information in a data sequence enters, is stored and leaves the network. A typical LSTM has three gates: the **forget gate**, the **input gate** and the **new memory gate**, the **output gate**. These gates can be considered as filters and each constitutes its own neural network.

- **Forget gate**
  Here, we decide which bits of the cell state (the network's long-term memory) are useful given the previous hidden state and the new input data. To do this, the previous hidden state and the new input data are fed into a neural network. This network generates a vector, each element of which lies in the interval [0,1] (guaranteed by the use of sigmoid activation). It is trained to produce outputs close to 0 when an element of the input is deemed irrelevant and close to 1 when it is relevant. It is useful to think of each element of this vector as a kind of filter or sieve that lets more information through as the value approaches 1.

  These output values are then sent upwards and element-wise multiplication with the previous state of the cell is applied. This one-off multiplication means that components of the cell state that have been deemed irrelevant by the forget gate network will be multiplied by a number close to 0 and will therefore have less influence on subsequent steps. In short, the forget gate decides which elements of long-term memory should now be forgotten (given less weight) given the previous hidden state and the new data point in the sequence.

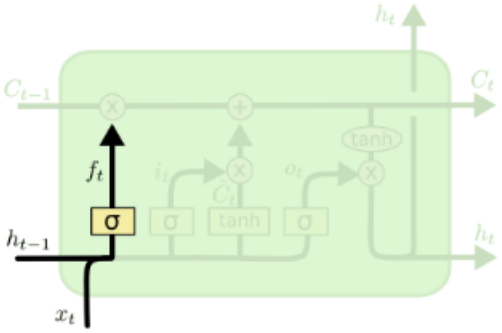


Figure 1.18: Forget Gate

- **Input gate**
  The next step concerns the new memory array and the input gate. The objective of this step is to determine what new information needs to be added to the network's long-term memory (cell state), given the previous hidden state and the new inputs. Both the new memory network and the input gate are neural networks that use the same inputs, namely the previous hidden state and the new input data. It should be noted that the inputs are in fact the same as those of the forgetting gate.

  The new memory network is a tanh-activated neural network that has learned to combine the previous hidden state and the new input data to generate a 'new memory update vector'. This tanh activation function constitutes the so-called **candidate gate** and the output vector essentially contains information from the new input data, given the context of the previous hidden state. This vector tells us how much to update each long-term memory component (cell state) in the network based on the new data. We use a tanh here because its values lie within [-1,1] and can therefore be negative. The possibility of negative values is necessary if we want to reduce the impact of a component on the cell state. However, the new memory vector doesn't actually check whether the new input data is worth storing. This is where the front door comes in. The input gate is a sigmoid-activated network that acts as a filter, identifying which components of the 'new memory vector' are worth retaining. This network will produce a vector of values in [0,1] (due to the sigmoid activation), allowing it to act as a point multiplication filter. As with the forgetting gate, an output close to zero tells us that we don't want to update that element of the cell state.

  Finally, the results of the new memory array gate and the input gate are multiplied in a pointwise way. The resulting combined vector is then added to the cell state, resulting in the network's long-term memory being updated.

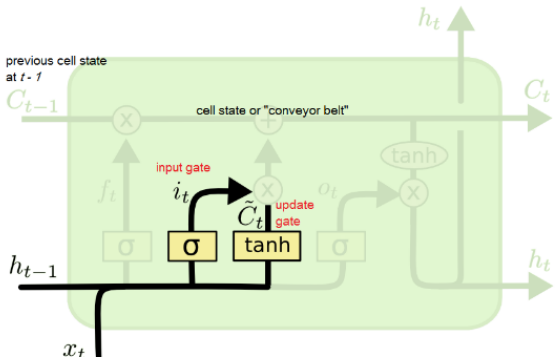


Figure 1.19: Input Gate

- **Output gate**
  Now that the network's long-term memory updates are complete, we can move on to the final stage, the exit gate, by deciding on the new hidden state. To do this, we'll use three elements: the recently updated cell state, the previous hidden state and the new input data.

  Although cell state captures long-term dependencies and carries information from memory, it may not be directly suitable as an output in many tasks.

The reason why hidden state is used as output instead of cell state lies in the architectural design and objectives of the LSTM model.

The cell state is regulated by the input gate, the forget gate and the output gate. The input gate determines the amount of new information to be stored in the cell state, while the forget gate determines the amount of information from the previous cell state to be discarded. The output gate, as mentioned earlier, determines which parts of the cell state are to be output. The output gate controls the flow of information from the cell state to the hidden state.

The hidden state is a filtered version of the cell state that is transformed using the output gate. The output gate selectively allows certain information from the cell state to influence the hidden state, making it more relevant and appropriate to the task in hand. By using the hidden state as an output, LSTM can focus on the most important features and remove less relevant or noisy information, resulting in a more efficient representation for the given task.

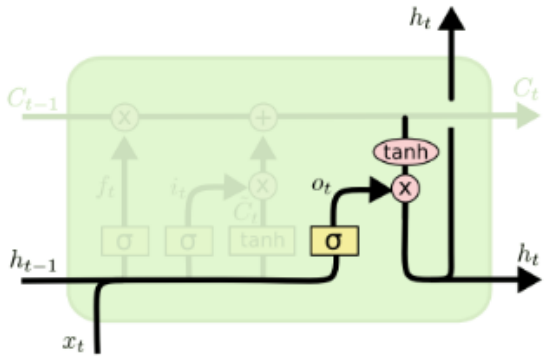
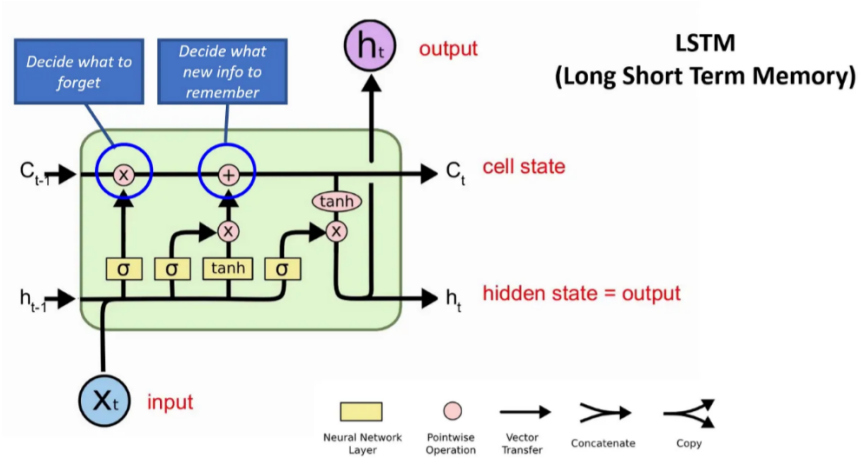


Figure 1.20: Output Gate

In summary, while the cell state captures long-term dependencies and carries memory information in an LSTM cell, the hidden state is generally chosen as the output because of its ability to filter and summarise relevant information from the cell state, making it more suitable for the specific task it is facing.



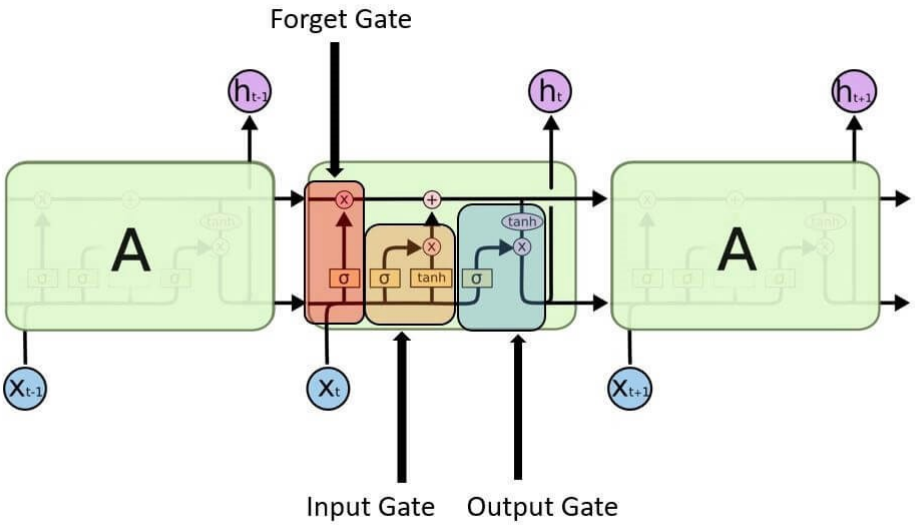


Figure 1.21: Summary of the LSTM cell

LSTM expects the input data to be a 3D tensor such as :

[batch-size, timesteps, feature]

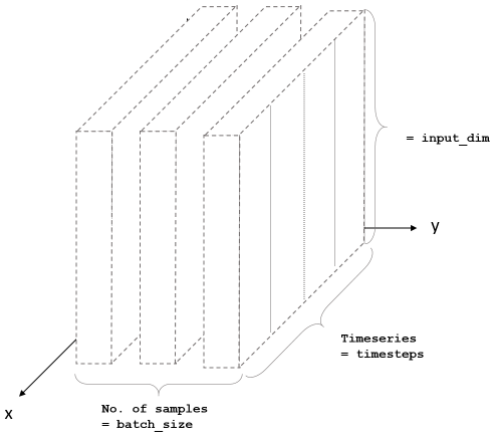


Figure 1.22: Input Size of Recurrent Neural Network [30]

- **Batch-size**: number of samples in each batch during training and testing.
- **Timesteps**:corresponds to the number of values present in a sequence or, more simply, the number of lines in each batch. For example, in [4, 7, 8, 4], there are 4 timesteps of feature = 1.
- **Features**:corresponds to the number of values present in a sequence or, more simply, the number of lines in each batch. For example, in [4, 7, 8, 4], there are 4 time steps of characteristic = 1.

When defining the input layer of the LSTM network, the network assumes that we have 1 or more samples and asks us to specify the number of time steps and

the number of features. We can do this by specifying a tuple to the 'input-shape' argument.
For example, the model below defines an input layer that expects 1 or more samples, 50 time steps and 2 features:

model.add(LSTM(32,input-shape=(50,2)))

In terms of structure, there are 3 different inputs for an LSTM cell:

- h_{t-1}: is the hidden state at the previous time step.
- c_{t-1}: is the cell state at the previous time step.
- x_t: is the current input at time t and is a vector whose dimension is the number of features specified in the input vector.

Four dense layers inside the cell:

- Forget gate
- Input gates = input + candidate
- Output gate

In the case of the figure above, it is important to note that the number of elements in the input vector is 3, while that of the hidden state and the cell is 2, for definition:

- The number of elements in the hidden state h and in the cell state c must be the same.
- The dimensions of h and c at times t_1 and t must be the same.
- Each entry in each sequence must have the same dimensions.

The outputs of the 4 different doors can be defined as follows:

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{1.24}$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \tag{1.25}$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \tag{1.26}$$
$$\tilde{c_t} = \sigma_h(W_c x_t + U_c h_{t-1} + b_c) \tag{1.27}$$
$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c_t} \tag{1.28}$$
$$h_t = o_t \cdot \sigma_h(c_t) \tag{1.29}$$

The following variables can be distinguished in these equations:

- $x_t \in R^d$: input vector of the LSTM cell.
- $f_t \in R^h$: 'forget gate' output vector.
- $i_t \in R^h$: 'input gate' output vector.
- $o_t \in R^h$: 'input gate' output vector.

- $\tilde{c}_t \in R^h$: 'candidate gate' output vector.
- h_t: 'hidden state' vector.
- c_t: 'cell state' vector.
- $W \in R^{h \times d}$, $U \in R^{h \times h}$ and $b \in R^h$: weights and biases matrices.

The upper indices **d** and **h** indicate respectively the number of input features and units hidden in the layers. We can now calculate the number of parameters in the cell. If we focus on the 'forget gate' in the figure above, we can make the following calculations:

- $W = h \times d$
- $U = h \times h$
- $B = h$

$$Parameters = W + U + B = (h \times d) + (h \times h) + h = (d + h) \times h + h \quad (1.30)$$

And since we have 4 doors defined in the same way, as it mentioned in the following paper [31], we simply have :

$$Total\ number\ of\ parameters = 4 \times ((d + h) \times h + h) \quad (1.31)$$

1.4.3.2 GRU

GRU stands for Gated Recurrent Unit and is a simplified version of the LSTM. It has just two gates: a reset gate and an update gate. The reset gate decides how much of the previous hidden state to keep, and the update gate decides how much of the new input to incorporate into the hidden state. The hidden state is also the cell state and the output, so there is no separate output gate. GRU is easier to implement and requires fewer parameters than LSTM.

The performance of LSTMs and GRUs depends on the task, the data and the hyperparameters. In general, LSTM is more powerful and flexible than GRU, but it is also more complex and subject to overfitting. GRU is faster and more efficient than LSTM, but it may not capture long-term dependencies as well as LSTM. Some empirical studies have shown that LSTM and GRU perform similarly in many natural language processing tasks, such as sentiment analysis, machine translation and text generation. However, some tasks can benefit from the specific features of LSTM or GRU, such as image captioning, speech recognition or video analysis. Despite their differences, LSTM and GRU share some common features that make them effective variants of RNN. They both use gates to control the flow of information and avoid the problem of disappearing or exploding gradients. They can both learn long-term dependencies and capture sequential patterns in the data. Both can be arranged in multiple layers to increase the depth and complexity of the network. Both can be combined with other neural network architectures, such as convolutional neural networks (CNNs) or attention mechanisms, to improve their performance.

The main differences between LSTM and GRU lie in their architectures and trade-offs. LSTM has more gates and more parameters than GRU, which gives it greater

flexibility and expressiveness, but also higher computational costs and a risk of overfitting. GRU has fewer gates and fewer parameters than LSTM, which makes it simpler and faster, but also less powerful and less adaptable. LSTM has a separate cell state and output, allowing it to store and output different information, whereas GRU has a single hidden state that serves both purposes, which can limit its capacity. LSTMs and GRUs may also have different sensitivities to hyperparameters, such as learning rate, dropout rate or sequence length.

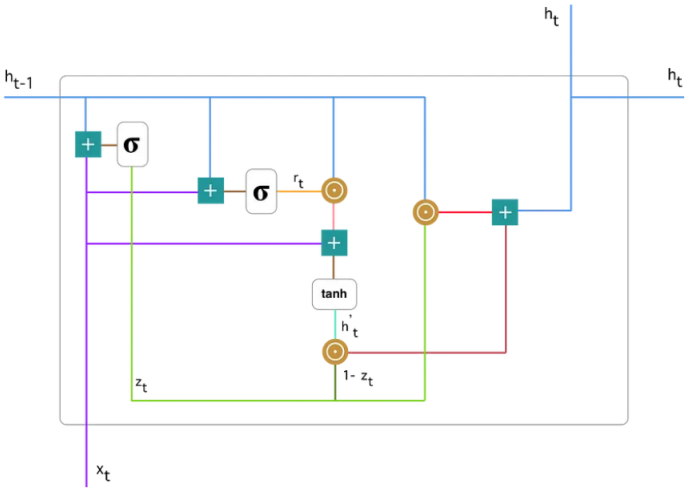


Figure 1.23: GRU structure

The equations governing the functioning of the cell are the following [32]:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{1.32}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{1.33}$$

$$\tilde{h_t} = tanh(W_h x_t + r_t * U_h h_{t-1} + b_z) \tag{1.34}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t} \tag{1.35}$$

The following variables can be distinguished in these equations:

- $x_t \in R^d$: input vector of the GRU cell.
- $\tilde{h_t} \in R^h$: candidate activation vector.
- $h_t \in R^h$: output vector.
- $z_t \in R^h$: update gate vector.
- $r_t \in R^h$: reset gate vector.
- $W \in R^{h \times d}$, $U \in R^{h \times h}$ and $b \in R^h$: weights and biases matrices.

As we saw in the case of the LSTM cell, it is possible to calculate the total number of parameters in the layer, knowing that in this case the number of gates is 3 [31]:

$$Total\ number\ of\ parameters = 3(n^2 + nm + 2n) \tag{1.36}$$

Where n is the dimension of the hidden state and m is the dimension of the input.

- **Update gate**
  When x_t is inserted into the network unit, it is multiplied by its own weights $W(z)$. The same applies to h_{t-1}, which contains information about the previous $t-1$ units and is multiplied by its own weights $U(z)$. The two results are added together and a sigmoid activation function is applied to overwrite the result between 0 and 1. The update gate helps the model to determine how much past information (from previous time steps) should be passed on to the next.

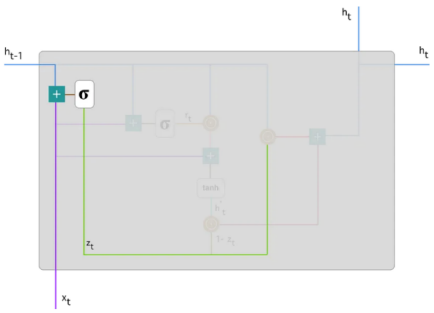


Figure 1.24: Update Gate - GRU Cell

- **Reset gate**
  Essentially, this gate is used by the model to decide how much past information to forget.

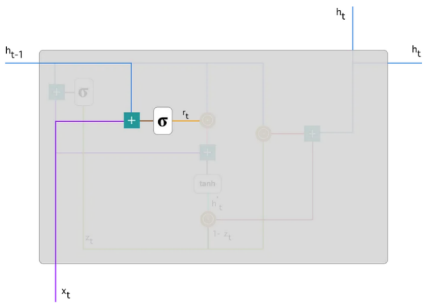


Figure 1.25: Reset Gate - GRU Cell

- **Current memory point**
  By calculating the product per element between r_t and h_{t-1} we can determine which elements to eliminate from the previous step.

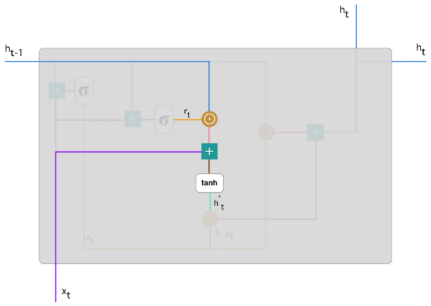


Figure 1.26: Current memory point - GRU Cell

- **Final memory point**
  In the last step, the layer calculates the vector h_t which contains the information for the current unit thanks to the update gate.

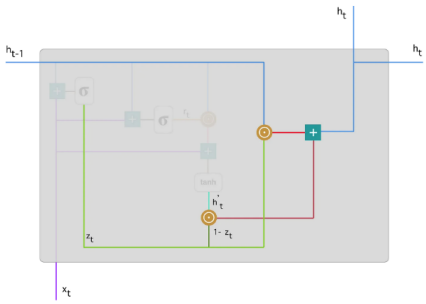


Figure 1.27: Final memory point - GRU Cell

The most common practical applications of GRU and LSTM cells could include the following:

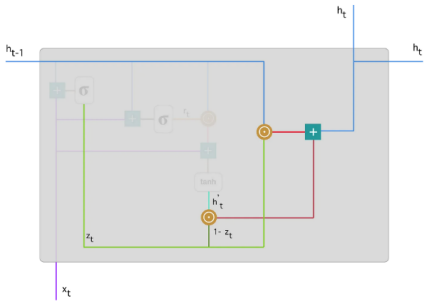


Figure 1.28: Final memory point - GRU Cell

- **Natural language processing** Natural language processing (NLP) refers to the AI field concerned with comprehending and generating human language, encompassing both text and speech. LSTM and GRU models are frequently employed in NLP tasks, including machine translation, text summarization, sentiment analysis, question answering, and chatbots. These models are adept at capturing the meaning and structure of language, accommodating inputs and outputs of varying lengths. For instance, an LSTM or GRU model can translate text between languages or create concise summaries of lengthy passages. This is achieved by encoding the input sequence into a concealed state and subsequently decoding it to produce an output sequence.

- **Speech recognition** Speech recognition involves the conversion of spoken language into text, and vice versa. LSTM and GRU models are valuable for speech recognition tasks due to their ability to understand the temporal and acoustic characteristics of speech signals, even in the presence of noise or incomplete input. For instance, an LSTM or GRU model can discern spoken words from a user or generate speech based on text. This is achieved through a sequence-to-sequence framework, akin to natural language processing. Moreover, these models can be enhanced by incorporating additional neural network components like convolutional or attention layers, which further enhance the accuracy and effectiveness of speech recognition.

- **Video analysis** Video analysis refers to the process of deriving insights or data from video content, which can encompass objects, actions, events, and emotions. LSTM and GRU models are also applicable in video analysis tasks due to their capacity to grasp both spatial and temporal aspects of video frames, enabling them to comprehend intricate and evolving scenes. For instance, an LSTM or GRU model can identify activities or emotions within a video, or produce descriptive captions or summaries by treating the video frames as a sequential series of images. This involves encoding them into a concealed state and subsequently decoding them into output labels or textual content.

- **Time series forecasting** Time series forecasting involves predicting future patterns or trends using previous data points from a time series, like stock prices, weather conditions, or traffic. LSTM and GRU models are also adept at time series forecasting due to their ability to capture time-related patterns and relationships within data. These models excel at handling complex, non-linear, and changing data. For instance, an LSTM or GRU model can anticipate forthcoming stock prices or weather situations by leveraging historical data, learning from the sequential context of the time series, and generating output values or ranges.

Chapter 2

Working environment and tools

This small chapter is concerned with the presentation of the technical working environment and the tools used, as well as the organisation of activities during the months of training.

2.1 OCEANS

Time-domain simulation of spacecraft has become crucial for studying Attitude and Orbit Control Systems (AOCS), ensuring the validation of attitude estimation and control algorithms. When these simulations accurately represent attitude and orbit dynamics, they enable the analysis, preparation, and evaluation of space mission performance. To address this requirement, the CNES AOCS Architecture department has created and consistently updates the OCEANS tool (known as "Outil de Création, d'Etude et d'ANalyse SCAO" in French), an environment within Matlab/Simulink for AOCS definition, study, and analysis.
An exportable variant, named LOCEANS (Light OCEANS), has been developed to allow industry and academic collaborators to leverage its capabilities. The OCEANS workbench encompasses all essential models to construct a mission scenario for time simulation, encompassing dynamics models, orbit propagators, environmental models, on-board and ground functions, and more.

This tool offers the following functionalities:

- Aiding in the creation of basic models.
- Building and modifying simulators.
- Generating and testing scenarios.
- Analyzing test outcomes.
- Managing technical assets.

The syntax and coding rules of the OCEANS environment were therefore taken into account in the definition of the different network models, particularly with regard to configuration files and their parameterization. In fact, when there are several parameters within a Simulink model that are subject to change, it is decided to opt for configurable models that make it possible to initialise all the constants in use in the model at first, and then, at a second stage, to obtain the rest of the variables dependent on these constants. This procedure thus makes it possible to

distinguish between independent and dependent variables in a simulation and to ensure time optimisation. Non-configurable models, on the other hand, are models whose parameters are not subject to change (e.g. elementary models of activation functions), so that no configuration file is needed. In both cases, Simulink models, once created, take the form of 'masked' models so that they cannot be modified, except with the administrator's consent: this is very useful to avoid accidental errors in modifying such elements that may prevent simulations from running correctly. In fact, once a Simulink block has been validated, it is not necessary to make any changes to it.

2.2 Utilities functions

Utilities functions are scripts created to facilitate the interface between Python and Matlab, especially with regard to the management of files in which network models already validated in Python are saved. In fact, such files, typically '.json' and '.h5' are often not easy to access as they necessarily require a Python interface, and therefore Matlab may be inappropriate for obtaining all the details relating to the network, such as the size of the weights, the number of units, any initialisations performed and the number of layers. A .json file and a .h5 file serve different purposes in the context of Keras or deep learning:

- **.json** file (JSON format): in Keras, a .json file is typically used to store the architecture or configuration of a neural network model. It contains information about the layers, their types, parameters, and the model's overall structure. However, it does not store the trained weights or model training history.
- **.h5** file (HDF5 format): an .h5 file, on the other hand, is used to store the trained weights and biases of a Keras model. It saves the learned parameters from the training process, allowing you to load a pre-trained model's weights and use it for predictions or fine-tuning. It does not store the model's architecture.

Typically, when working with Keras, it's common to save and load models in two parts: the model architecture (as .json or .yaml) and the model weights (as .h5). This separation allows to reuse or transfer models more flexibly. Importing the weights, moreover, was certainly one of the biggest challenges of this work as their storage by Python is never unique and their formats change according to each type of layer and network. At first, i.e. the first validation phase, they were obtained by direct saving as '.txt' from the Keras code of the network model. However, for more efficient use of the developed architecture, it was later necessary to develop a function that could analyse each layer of the network and output the weights for each layer in a Matlab-accessible format.

- **h5tojson.m** : this function is responsible for obtaining the structure of the network in **.json** format, knowing that the latter is easily retrievable from the attributes of the **.h5** file.

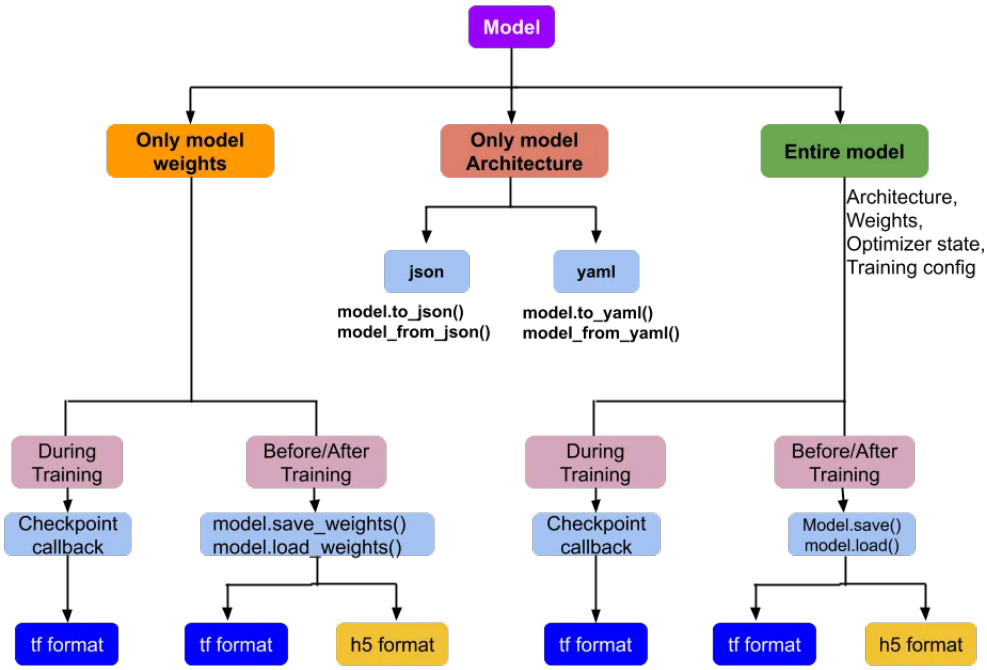


Figure 2.1: Saving and Loading of Keras Sequential and Functional Models [33]

- **read_struct.m** : is able to read the **.json** file and create a similar structure in matlab corresponding to the network. This structure in Matlab will therefore have several fields to which all the layers and their dimensions, the dimensions of the inputs and outputs, and all the other main characteristics will correspond in order to have an accurate description of the model from Python.
- **extractH5_weights.m** : the purpose of this function is instead that of creating a structure that could contain as many fields as there are layers in the network and in each of these have the different weight matrices corresponding to each layer with the correct dimensions. In this way all the weights corresponding to each layer are separately saved as Matlab variables and then easily loaded into the Simulink environment through the appropriate configuration files.

2.3 Internship Roadmap

In the second part, all models that were the subject of initial research were reproduced in Simulink with particular attention to the OCEANS simulation environment. The AI library was thus created with strong reference to the Python codes of the Keras library in which all the structures of the networks necessary for the implementations of this work are well documented.
The last period has been focused on the analysis for the validation of these modeling schemes and comparing the results with Python, checking for equivalence and thus applicability. Small concrete application cases were then studied, and attempts were made to interface the networks with in-service simulators for AOCS applications.

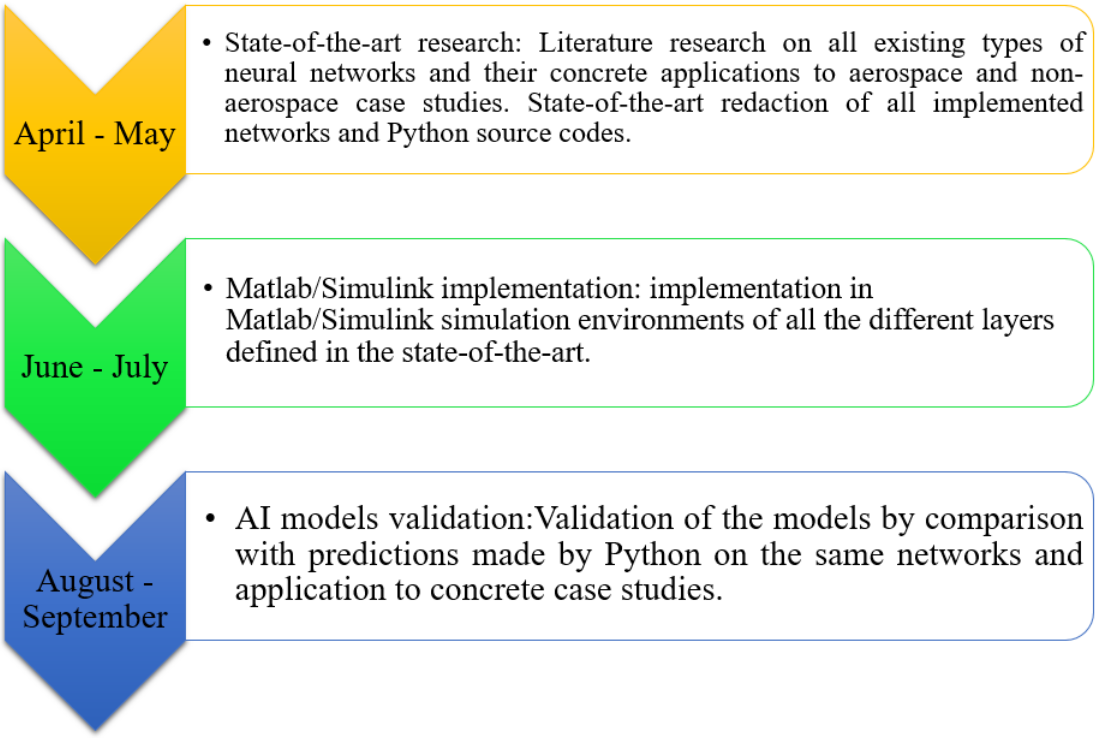


Figure 2.2: Internship Roadmap

Chapter 3

Matlab|Simulink implementation

This section is dedicated to the description of the Matlab/Simulink models of the implemented networks. It is important to note that these implementations were conceived for a feedforward use of the networks, starting from the principle of having already optimised weights obtainable from .h5 files from Python. The reasons for this choice lie in the fact that most of the industry and academia develops its AI tools in Phyton, and it is therefore not readily possible to interface the latter with the AOCS simulation environments which are fully developed in Matlab/Simulink. The aim was thus to create an AI environment that could be directly used to make predictions without proceeding through the training phase, having the weights of the optimised network already available and making the collaboration with industry and academy easier and more productive.

3.1 AI Folder

In the library, we can find all the Simulink models and associated configuration files for generating the architecture of the network. We then distinguish between the activation functions, initialising them as non-configurable models, and the various elementary layers that make up the neural network to be created. Each layer, on the other hand, is initialised as a configurable model in which the configuration file is used to initialise its parameters for simulation purposes.

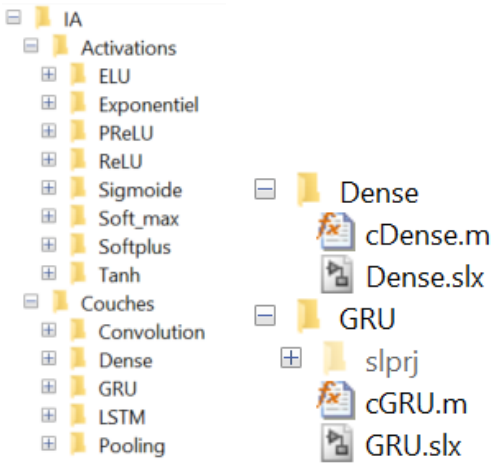


Figure 3.1: Library architecture

The goal of creating versatile, elementary, interchangeable Simulink models lies

in the ability to create, following individual validations, any network models by simply moving elementary blocks from the library to the Simulink of the network. Thus, this allows for a strong ability to create custom models that can satisfy all kinds of needs by taking advantage, above all, of Simulink's graphical interface that allows, compared to Python, a better overall understanding of the network structure.

3.1.1 Activation functions

In the following are the simple Simulink models of the main activation functions used. The models concerning the activation functions have been chosen as non-configurable as the parameters that compose them are mostly simple constants involved in as many simple operations.

- **ELU**

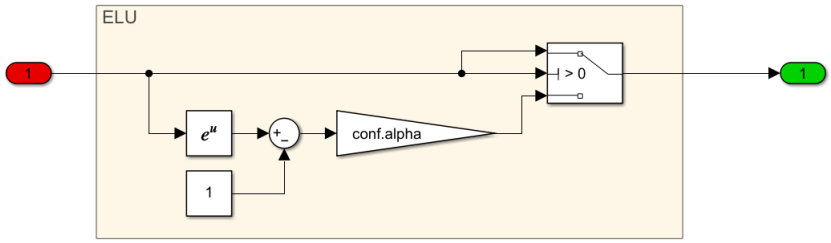


Figure 3.2: ELU activation function

- **Exponential**

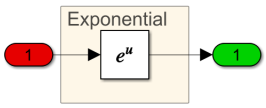


Figure 3.3: Exponential activation function

- **PReLu**

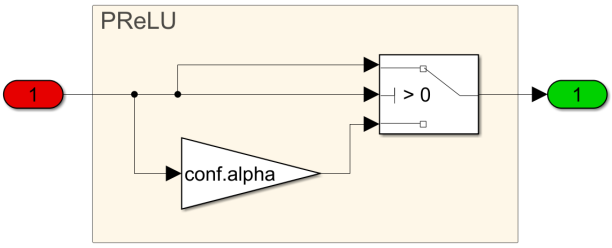


Figure 3.4: PReLu activation function

- **ReLu**

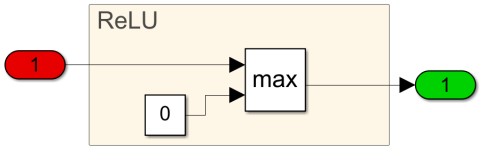


Figure 3.5: ReLu activation function

- **Sigmoid**

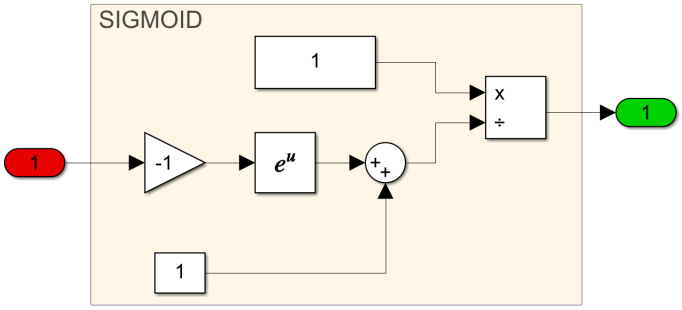


Figure 3.6: Sigmoid activation function

- **Softmax**

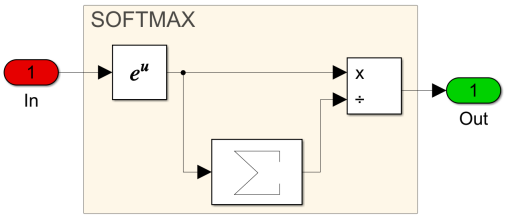


Figure 3.7: Softmax activation function

- **Softplus**

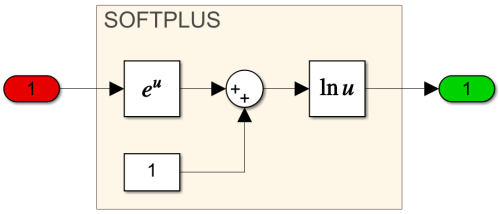


Figure 3.8: Softplus activation function

- **Tanh**

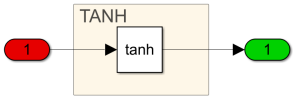


Figure 3.9: Tanh activation function

3.1.2 Layers

The Layers folder contains thus all the network models implemented according to the OCEANS language. Each of these will therefore contain a configurable Simulink model corresponding to the architecture of the layer and a configuration file in which the two phases of model calibration will be clearly distinguished: first the independent parameters are defined, then the dependent parameters are computed from them.

3.1.2.1 Dense

The library's 'Dense' folder contains the masked Simulink model representing the layer and the associated configuration file in OCEANS format. The model performs the simple operation of matrix multiplication and adding the bias to the input vector X:

$$U^{[1]} = W^{[1]} \cdot X + B^{[1]} = \sum_{i}^{n} w_i x_i + B^{[1]}$$

In the **first step** of the configuration the main parameters independent of the layer are initialized:

- Number of layer inputs
- Number of neurons
- The value 1 or 0 indicating whether or not to use the biases
- The names of the layer, the activation function and the weights and biases files

Then, **step two** of the config file allows us to retrieve the weights and biases of a network already trained in Matlab format and which can be loaded into the Simulink model.

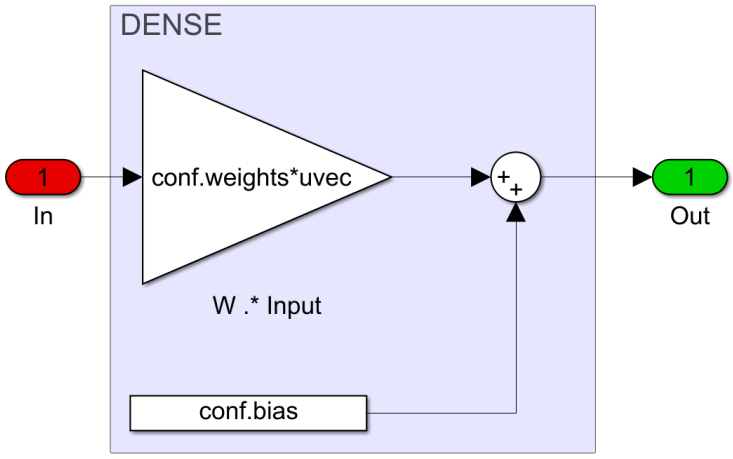


Figure 3.10: Simulink model of the dense layer

```
PDense =

  struct with fields:

      input_size: 6
            name: 'Dense1'
      activation: 'Linear'
           units: 10
        use_bias: 1
       bias_file: 'B1.txt'
    weights_file: 'W1.txt'
            bias: [10×1 double]
         weights: [10×6 double]
```

Figure 3.11: Output of the configuration file **cDense.m**

3.1.2.2 LSTM

Regarding the LSTM layer, we tried to reproduce its structure in Simulink by having all the weights of the different gates obtained previously in the python network. Since the structure of an LSTM consists of a single cell that repeats a number of times equal to the time steps specified in the input, only this cell has been modeled, which will then be repeated according to the data specification of entry.

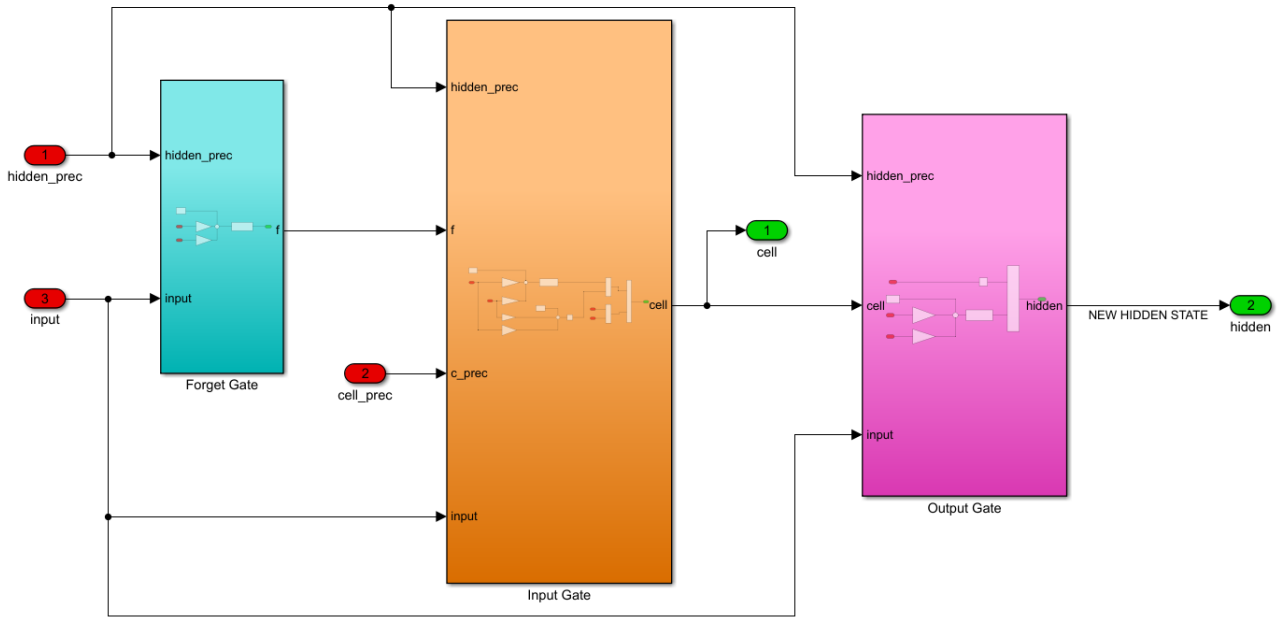


Figure 3.12: Simulink model of the LSTM layer

Entering each of the subsystems that make up each gate, the following structures can be observed:

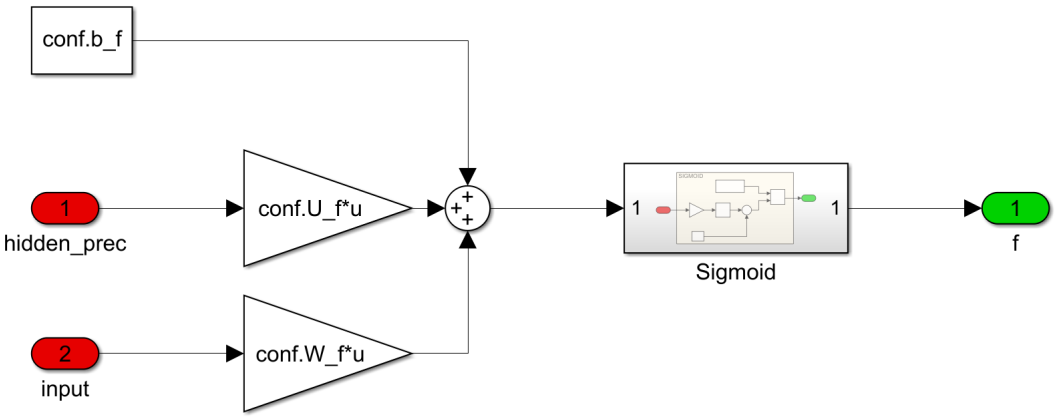


Figure 3.13: Forget gate of the LSTM layer

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

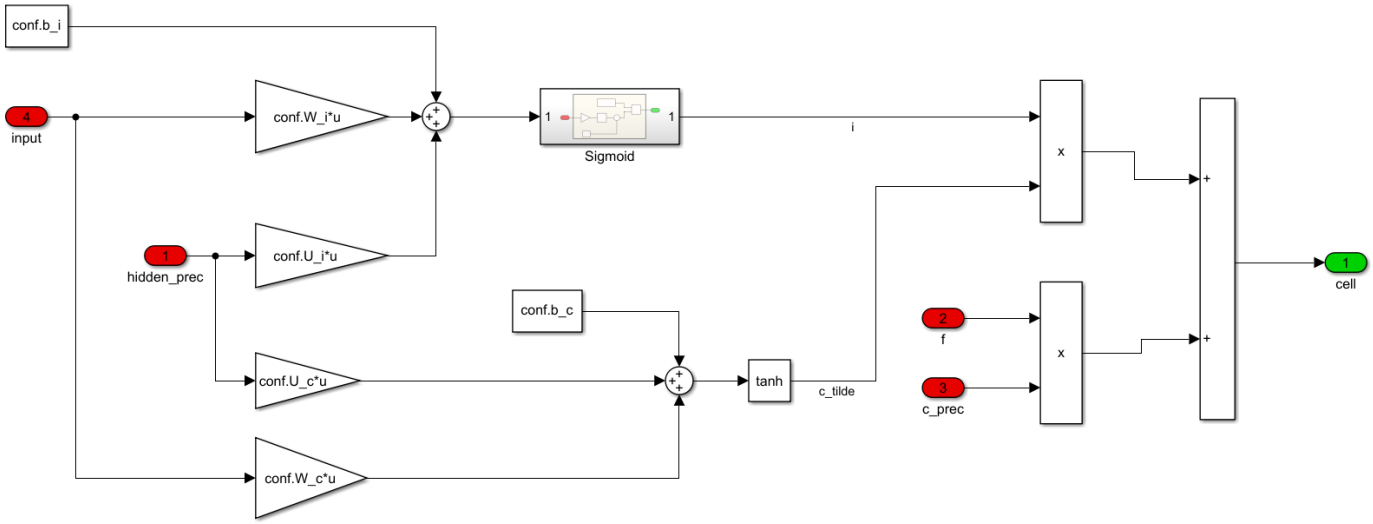


Figure 3.14: Input gate of the LSTM layer

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$\tilde{c_t} = \sigma_h(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c_t}$$

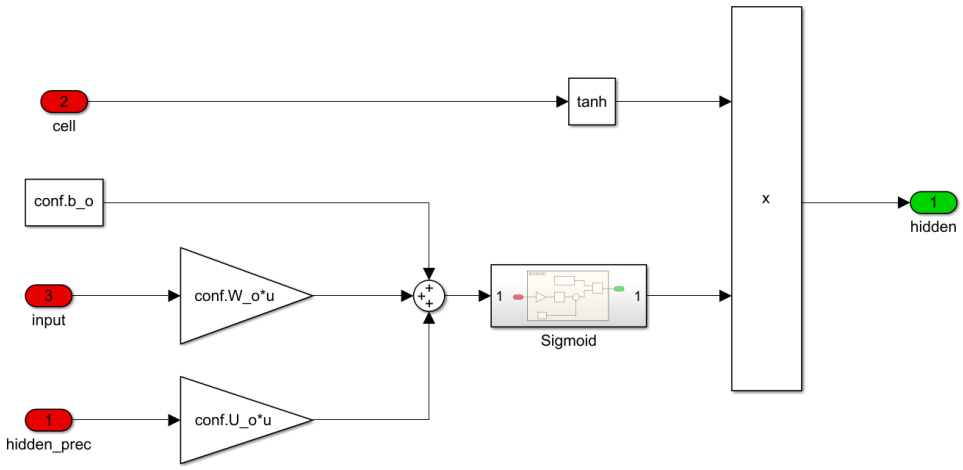


Figure 3.15: Output gate of the LSTM layer

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$h_t = o_t \cdot \sigma_h(c_t)$$

As far as the network entry is concerned, in the first part of the configuration file it is possible to specify the number of units, the time steps and the number of batches available. If we imagine each batch as a matrix of dimensions time step x characteristics our input can be represented as a 3D vector. The elementary cell will be repeated a number of times equal to the time step defined in the inputs and each cell will have as input a batch line with the specified number of characteristics.

```
PLSTM =

  struct with fields:

        units: 10
         name: 'LSTM'
    timesteps: 1
     features: 6
   batch_size: 1
     b_c_file: 'b_c.txt'
     b_f_file: 'b_f.txt'
     b_i_file: 'b_i.txt'
     b_o_file: 'b_o.txt'
     U_c_file: 'U_c.txt'
     U_f_file: 'U_f.txt'
     U_i_file: 'U_i.txt'
     U_o_file: 'U_o.txt'
     W_c_file: 'W_c.txt'
     W_f_file: 'W_f.txt'
     W_i_file: 'W_i.txt'
     W_o_file: 'W_o.txt'
```

```
b_c: [10×1 double]
b_f: [10×1 double]
b_i: [10×1 double]
b_o: [10×1 double]
U_c: [10×10 double]
U_f: [10×10 double]
U_i: [10×10 double]
U_o: [10×10 double]
W_c: [10×6 double]
W_f: [10×6 double]
W_i: [10×6 double]
W_o: [10×6 double]
```

Figure 3.16: Output of **cLSTM.m** file

3.1.2.3 GRU

Similar to the LSTM scheme, an attempt was made to reproduce the well-described state-of-the-art architecture in the case of the GRU by considering it possible to load weights via the configuration file. As in the previous cases, the first step in the configuration is therefore to define the general characteristics of the cell in terms of units, activation functions, and input dimensions. In the second part, however, the actual weights are loaded into the model.

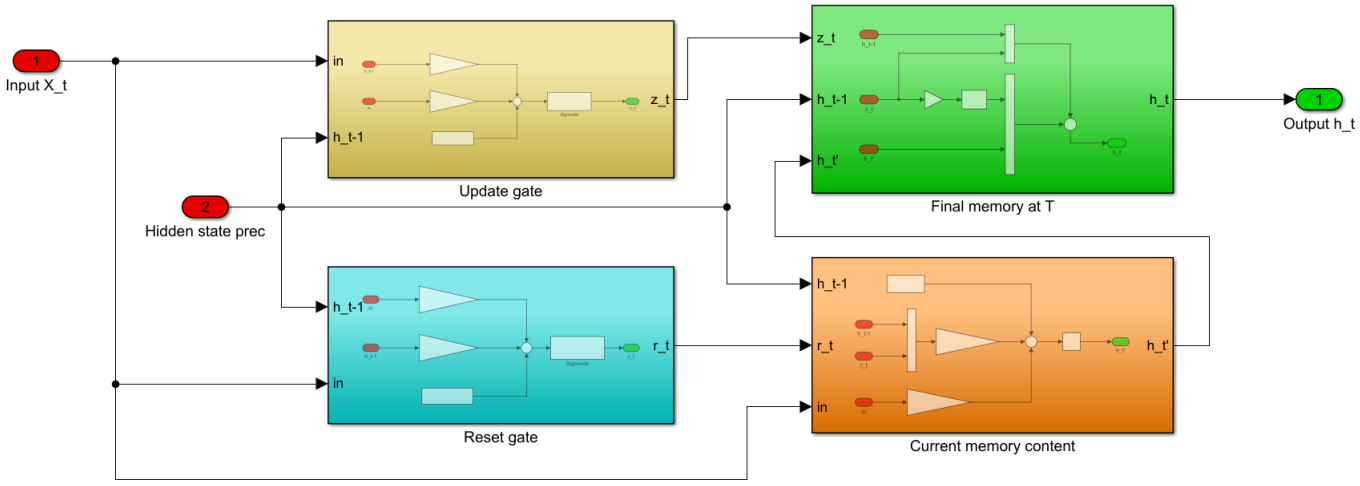


Figure 3.17: Simulink model of the GRU layer

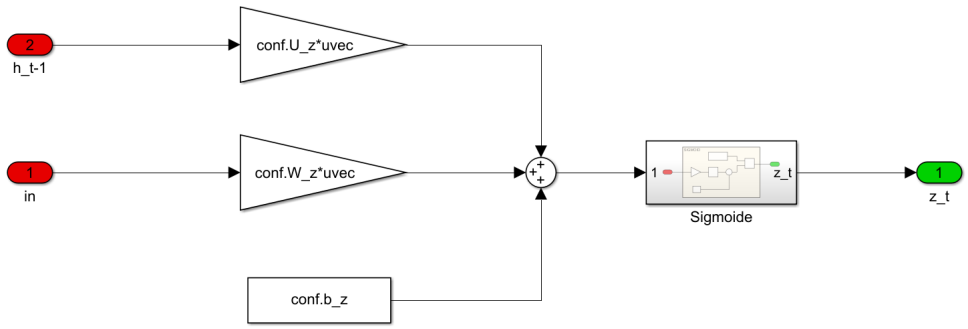


Figure 3.18: Update gate of the GRU layer

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

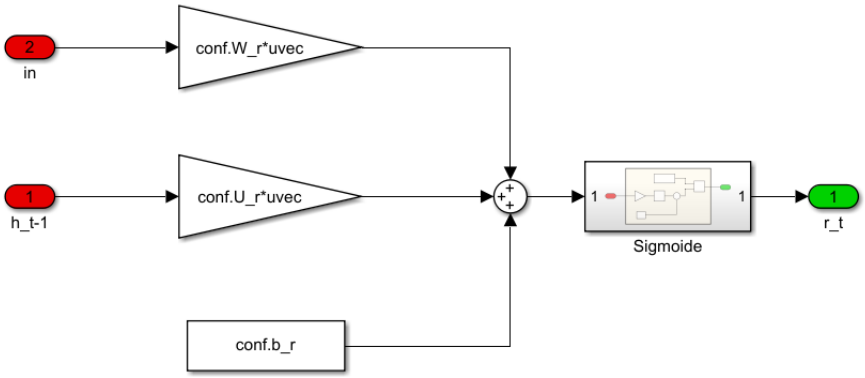


Figure 3.19: Reset gate of the GRU layer

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

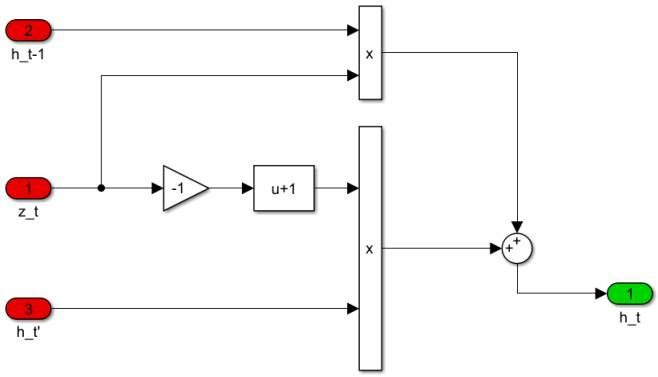


Figure 3.20: Final memory at time T

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

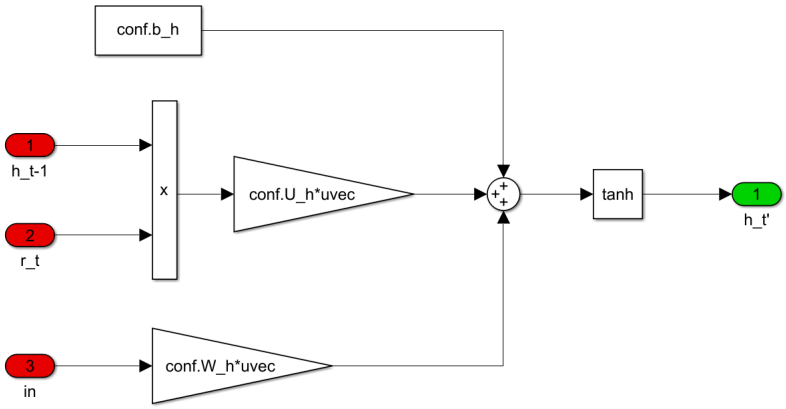


Figure 3.21: Current memory content

$$\tilde{h}_t = tanh(W_h x_t + r_t * U_h h_{t-1} + b_z)$$

```
PGRU =

  struct with fields:

         units: 5
          name: 'GRU'
     timesteps: 3
      features: 5
    batch_size: 1
      b_z_file: 'b_z.txt'
      b_r_file: 'b_r.txt'
      b_h_file: 'b_h.txt'
      U_z_file: 'U_z.txt'
      U_r_file: 'U_r.txt'
      U_h_file: 'U_h.txt'
      W_z_file: 'W_z.txt'
      W_r_file: 'W_r.txt'
      W_h_file: 'W_h.txt'
   reset_after: 0

      W_z: [5×5 double]
      W_r: [5×5 double]
      W_h: [5×5 double]
      U_z: [5×5 double]
      U_r: [5×5 double]
      U_h: [5×5 double]
      b_z: [0 0 0 0 0]
      b_r: [0 0 0 0 0]
      b_h: [0 0 0 0 0]
```

Figure 3.22: Output of **cGRU.m** file

3.1.2.4 Convolution 2D

The library's **Convolution** folder contains the masked Simulink model representing the layer and the associated configuration file in OCEANS format. The model performs the two-dimensional convolution operation through a Matlab Function which calculates the result from the input matrix and the parameters specified in the configuration file. It is good to point out that there are also other types of convolution, one-dimensional and three-dimensional, which are not addressed in this work.

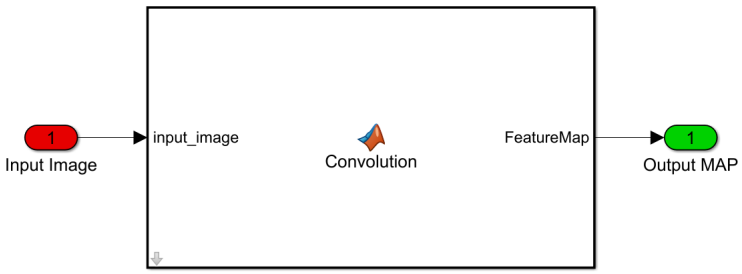


Figure 3.23: Simulink model of the Convolution 2D layer

The operation

$$y_{i^{l+1},j^{l+1},d} = \sum_{i=0}^{H} \sum_{j=0}^{W} \sum_{d^l=0}^{D^l} f_{i,j,d^l,d} \times x^l_{i^{l+1}+i,j^{l+1}+j,d^l}$$

is realised in Matlab through the following algorithm:

```
FeatureMap = zeros(convRow,convCol);
% Convolution
for filterNum = 1 : numFilters
    for channelNum = 1 : filterchannels
        % Obtention de la caractéristique (filterDim x filterDim) nécessaire à la convolution
        ff = filters(:,:,channelNum,filterNum);
        for h = 1 : convRow
            for w = 1 : convCol
                % Extraire le champ réceptif local de l'image d'entrée ; l'option de 'padding' doit être ajoutée
                localReceptiveField = image((h - 1) * stride(1) + 1 : (h - 1) * stride(1) + filterDimRow, ...
                                            (w - 1) * stride(2) + 1 : (w - 1) * stride(2) + filterDimCol, ...
                                            channelNum);
                % Multiplication et somme avec les éléments  du filtre
                FeatureMap(h, w) = sum(localReceptiveField(:) .* ff(:));
            end
        end
    end
    FeatureMap = FeatureMap + b(filterNum);
end
```

Figure 3.24: Convolution 2D operation in Matlab

The configuration file in the first step allows us to initialize all the parameters necessary to perform the convolution operation:

- The dimensions of the kernel that is applied to the input matrix in terms of number of rows, columns, channels.
- The step of movement of the same window during the operation (the 'stride') that in this case has two dimensions: the vertical one and horizontal one.
- The padding option

In the second step of the configuration file, all dependent parameters entering the Simulink model are calculated:

- The dimensions of the pooling output matrix are then calculated according to the following formula that we have already seen in the state of the art: $O = \frac{I-F+P_{start}+P_{end}}{S} + 1$
- We initialize the entry conf.poolSize which contains the dimensions of the pooling window.
- We initialize the entry conf.filters and conf.dimFilters which contains the dimensions of the filters.
- The final values of the displacement step conf.Stride in the vertical and horizontal direction.

3.1.2.5 Pooling

The library's **Pooling** folder contains the hidden Simulink model representing the layer and the associated configuration file in OCEANS format. The model performs the **Max or Average pooling** operation on the outgoing element of the convolution layer: this operation is performed in Simulink using a Matlab Function which calculates the result of the pooling operation at from the input matrix and the parameters specified in the configuration file.

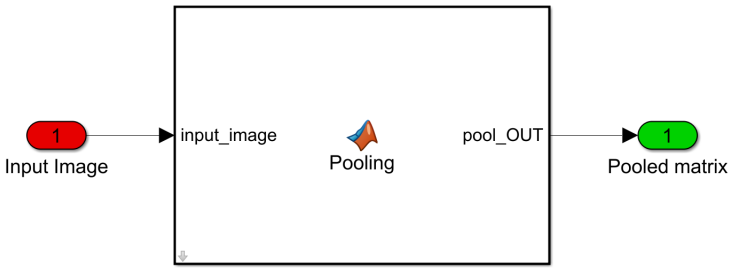


Figure 3.25: Simulink model of the Pooling layer

The operations

$$Max : y_{i^{l+1},j^{l+1},d} = \max_{0 \leq i < H, 0 \leq j < W} x^{l}_{i^{l+1} \times H + i, j^{l+1} + \times W + j, d}$$

$$Average : y_{i^{l+1},j^{l+1},d} = \frac{1}{HW} \sum_{0 \leq i < H, 0 \leq j < W} x^{l}_{i^{l+1} \times H + i, j^{l+1} + \times W + j, d}$$

are realised in Matlab through the following algorithm:

```
for h = 1:PoolDim(1)
    for w = 1:PoolDim(2)

        % Extract the local field to be pooled
        field = image((h - 1) * Stride(1) + 1 : (h - 1) * Stride(1) + poolSize(1), ...
                      (w - 1) * Stride(2) + 1 : (w - 1) * Stride(2) + poolSize(2));

        if type == 1
            pool_OUT(h,w) = max(field(:));
        else
            pool_OUT(h,w) = sum(field(:))/sum(poolSize);
        end

    end
end
```

Figure 3.26: Pooling operation in Matlab

The configuration file in the first step allows us to initialize all the parameters necessary to perform the pooling operation on the convolution output matrix:

- The dimensions of the pooling window that will be applied to the input matrix
- The step of movement of the same window during the operation (the 'stride')
- The padding option

- The choice of the type of pooling (Max or Average)

In the second step of the configuration file, all dependent parameters entering the Simulink model are calculated:

- The dimensions of the pooling output matrix are then calculated according to the following formula that we have already seen in the state of the art: $O = \frac{I-F+P_{start}+P_{end}}{S} + 1$
- We initialize the entry conf.poolSize which contains the dimensions of the pooling window.
- The final values of the displacement step conf.Stride

Chapter 4

AI models validation

The validation of the models has been done individually on each layer created, comparing Python and Simulink results each time. For each of the configurable models created in the AI folder, then, a network was created in Python from which all the weights were appropriately extracted separately, at first saved as a **.txt** file and then loaded into Matlab via the **load** function, later extracted from the network h5 models via the **extractH5_weights** function in the **Utilities** folder. Moreover, to make this process officially acceptable within the CNES AOCS service, we used neural networks previously created within the department in which all the layers we individually defined would appear. These models were loaded into Python through their .h5 files by then extracting their weights, verifying that Simulink would give the same results.
Regarding the convolution layers, these were validated individually 'by hand' as we had no previous models to implement and verify the equality of the results.

4.1 Single layers validation

This section is dedicated to the validation of the individual network layers implemented in Simulink: in a very basic way, we will check that Python and Simulink predictions match against the same input. The network will therefore consist of the individual layer/cell that we wish to validate.

4.1.1 Dense layer validation

Once it had been verified that the simple dense layer worked in the same way as on Python, the first elementary network models were implemented relating to two basic problems of Machine Learning, namely binary classification and multiclassification. Binary Classification is where each data sample is assigned one and only one label from two mutually exclusive classes. Multiclass Classification is where each data sample is assigned one and only one label from more than two classes. As is well known, depending on the problem to be solved, the structure of the network must be designed and adapted. Consequently, for a binary classification problem, it is necessary to have a network with a dense layer composed of only one neuron at the end, as the prediction is made on only one value, to which the Sigmoid activation function must be applied in order to extract a probability. On the other hand, if there are more classes to which the input data can be traced back, it is necessary to have in the last layer of the network a number of neurons equal to that of the classes in total, this because it is necessary to be able to ex-

tract from each output a probability via the Softmax function and then assign the highest probability output to the corresponding class.

Binary classification

The objective of the classification in this case is to predict whether the patient, based on the specifications in the database, is more likely to have diabetes or not. Each row of the database represents a patient with his specifications and the classification, thus, takes place between two distinct elements to which we associate two different probabilities, 0 in the case of no problem and 1 in the case of a positive one. Therefore, a simple Python model was first constructed and the accuracy of the prediction checked, after which the weights were extracted and the structure reproduced in Simulink, verifying that the same confusion matrix was obtained.

```python
from numpy import loadtxt
from numpy import savetxt
from numpy import concatenate
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# load the dataset
dataset = loadtxt('./pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]

# define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
# The line of code that adds the first Dense layer is doing two things:
# defining the input or visible layer and the first hidden layer.
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10, verbose=0)

# evaluate the keras model
_, accuracy = model.evaluate(X, y, verbose=0)
print('Accuracy: %.2f' % (accuracy*100))

# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
# summarize the first 40 cases
for i in range(20):
 print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

Figure 4.1: Simple binary classification in Python

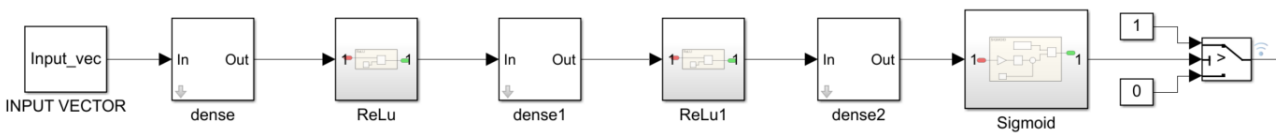


Figure 4.2: Binary classificaion

The Python model and Simulink have the same level of accuracy and therefore the same confusion matrix. We can actually see that the model is not very accurate as it is very basic but can easily be improved by increasing the training time or the size of the network.

		Predicted	
		Positive	Negative
Expected	Positive	9	13
	Negative	2	16

Table 4.1: Confusion matrix for binary classification

Multiclassification

In the case of multiclassification, we followed the same procedure as in the previous case by making minor changes to the Python code and using a different database that had 3 different classes to distinguish, specifically three species of flowers, indicated in the following confusion matrix as 0, 1 and 2. As also mentioned earlier, it is necessary in this case to adapt the final activation function, as the probability for each output element must be calculated.

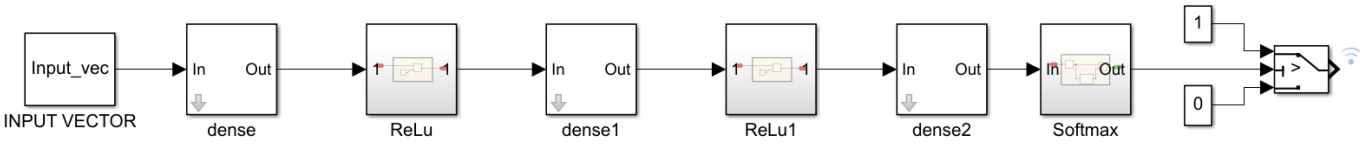


Figure 4.3: Multiclassification net

4.1.2 LSTM layer validation

The procedure for LSTM layer validation involves several steps: first, single cell validation, then a single layer with multiple time steps, and finally a double layer of LSTM cells with multiple time steps. Complementing this process is the validation of the CNES internal network, found in the 'Validation' file under the name **'lstm_model.h5'**, used for a previous case study through the models created in the library.

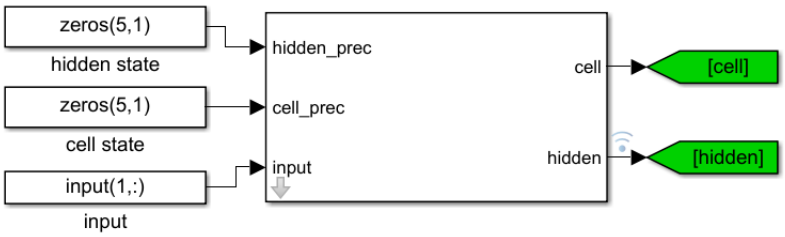


Figure 4.4: Single LSTM cell validation

Feature	Python output	Simulink output	**Abs error (e-07)**	**Rel error (e-06)**
1	-0.01554293	-0.01554293	0.07075011	-0.45519162
2	-0.01791598	-0.017915980	0.0018479	-0.01031436
3	-0.53773195	-0.53773195	0.0935482	-0.01739680
4	0.34675896	0.346758960	0.3203675	0.09238910
5	-0.11371579	-0.113715790	0.0368490	-0.03240448

Table 4.2: Single LSTM cell accuracy

By evaluating the cell over several time steps, it is necessary to replicate its structure, which will therefore have the same weights, by a number equal to that of the steps, and to transmit the calculated hidden state and the cell state between one cell and the next. The hidden state in fact represents the specific output to be obtained, while the cell state corresponds to a long-term memory in which the main information to be filtered is stored.

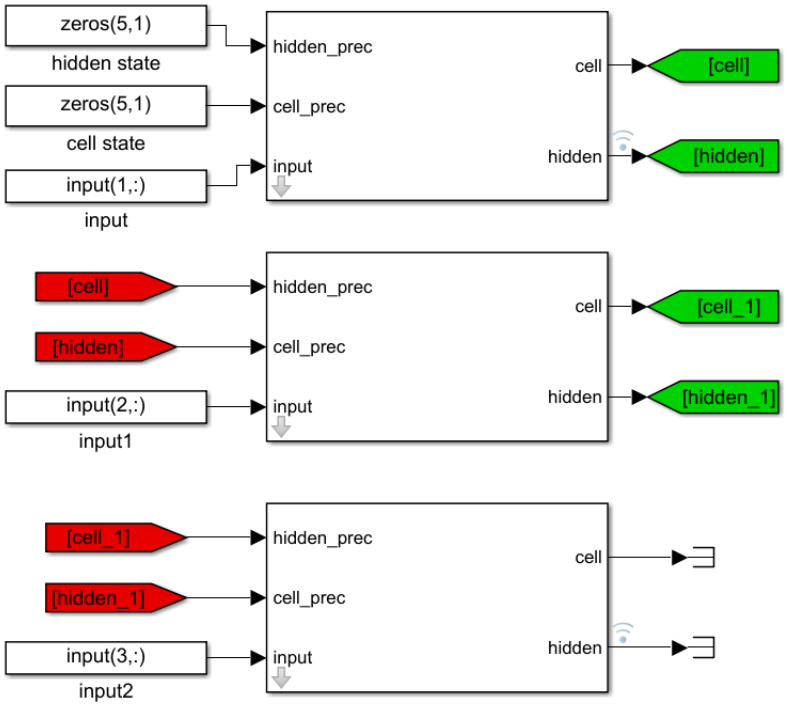


Figure 4.5: LSTM cell validation with 3 timesteps

Feature	Python output	Simulink output	**Abs error (e-07)**	**Rel error (e-06)**
1	-8.60686228e-02	-0.086068642	0.197504548	-0.229473345
2	1.18189655e-05	0.00001181	0.00005581	0.47224728
3	1.27405690e-06	0.00000127	0.000000998	0.078347898
4	-8.78330842e-02	-0.087833120	0.36717212	-0.418033962
5	8.22952688e-02	0.082295281	0.130186392	0.158194261

Table 4.3: LSTM cell with 3 timesteps accuracy

4.1.3 GRU layer validation

The validation of the GRU layer also took place in this case in two phases. We first verified that the single cell respected the Python output and then we applied other time steps to ensure that it works on different inputs.

```
units = 5
timesteps = 3
features = 5
batch_size = 1

data = np.arange(0,batch_size * timesteps * features)
data = data.reshape(batch_size, timesteps, features).astype(np.float32)
#print('Input data is :', data)

model = Sequential()
model.add(GRU(units, input_shape=(timesteps,features), return_sequences=True))

GRU_layer = model.layers[0]
model.summary()
output = model.predict(data)
Y = np.array(output).reshape(-1)
poids = model.get_weights()
#print(poids)
print('The predicted output is :', output)
```

Figure 4.6: Simple Python code for GRU cell implementation

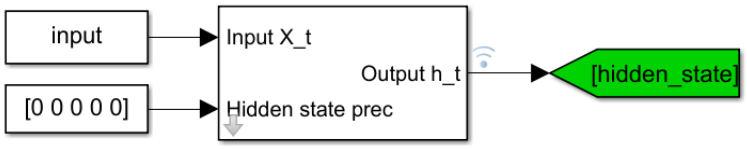


Figure 4.7: GRU single cell validation

Feature	Python output	Simulink output	**Abs error (e-07)**	**Rel error (e-06)**
1	-0.118742082	-0.11874209	0.07775790	-0.0654847
2	0.59500497	0.595005	0.25177177	0.042314227
3	-0.0338271	-0.03382717	0.05651234	-0.16706200
4	0.24623746	0.24623741	0.51057482	0.20735063
5	-0.14533847	-0.14533848	0.040251415	-0.027694947

Table 4.4: Single GRU cell accuracy

In the case of multiple time steps, the previous hidden state of a cell corresponds to the output of the previous cell. In contrast to the LSTM, in this case there is no cell state to be propagated to all cells for all time steps.

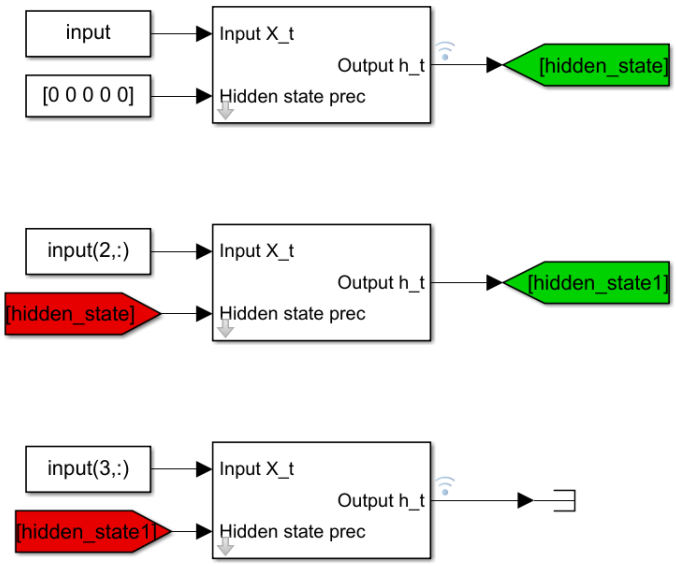


Figure 4.8: GRU cell validation with 3 timesteps

Feature	Python output	Simulink output	**Abs error (e-03)**	**Rel error (e-04)**
1	0.9954368	0.99543320	0.00359261	0.03609079
2	-0.01692256	-0.01692256	0.00000525	-0.00310337
3	0.40168986	0.40168984	0.00001166	0.00029045
4	0.12935196	0.12955516	0.20320233	15.70925825
5	0.3348373	0.33483716	0.00013194	0.00394070

Table 4.5: GRU network accuracy (last timestep)

Inevitably, as time steps increase, we see error propagation between Simulink and Python output as the initial error is not very small. In fact, the iteration of the calculations will certainly amplify the distance between the outputs of the two predictions.

4.1.4 Convolution & Pooling validation

The convolution and pooling operations were both validated 'by hand' with a matrix of input numbers, similar to an image in which each number represents a pixel. The following simple kernel was then considered for convolution:

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}$$

applied to the input matrix in the picture below. Regarding the pooling operation, this was applied on the same input image while preserving a kernel of 3x3 size. The outputs of the algorithms shown in the previous chapter are therefore depicted below.

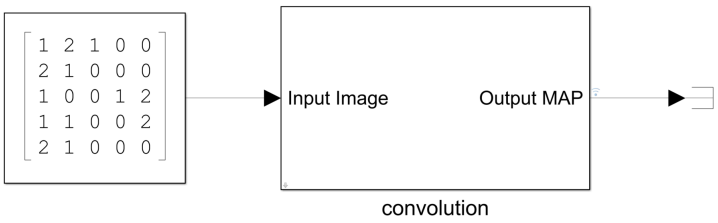


Figure 4.9: Convolution operation

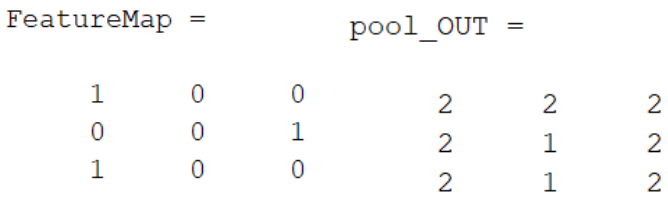


Figure 4.10: Result of the convolution operation in Matlab (sx); pooling operation with a kernel of 3x3 (dx)

4.2 Nets validation

Differently from the previous section in this part, the aim is to use networks that have already been trained and validated internally at CNES and verify that, by extracting the optimised weights and using the structures in Simulink, the predictions match. The nets were used in a previous study to predict torques and forces acting on the satellite due to propellant sloshing in the tank following specific AOCS manoeuvres. PAPIERRRR

4.2.1 Multilayer perceptron

This is a simple network called **'mlp_lag1_best.h5'** consisting of two dense layers between which there is an activating ReLu. According to its use within the study, at the input we have the 3 velocities and angular accelerations, at the output the forces and torques acting on the satellite. In order to test the simple correct operation in prediction, a vector of zeros was given as input. Obviously a non-zero output is to be attributed to the presence of the biases, clearly not optimised to have a null input.

```
model = Sequential()
model.add(Input(shape=(6,)))
model.add(Dense(10,activation='relu'))
model.add(Dense(6,activation='linear'))
model.load_weights('mlp_lag1_best.h5')
x = np.array([[0,0,0,0,0,0]])
y = model.predict(x)
```

Figure 4.11: Python code of the the 'mlp_lag1.h5'

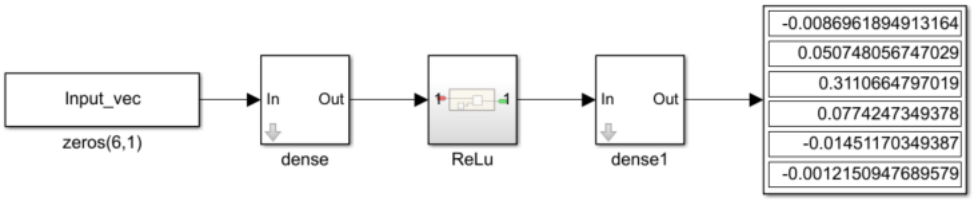


Figure 4.12: Simulink net of the 'mlp_lag1.h5'

As can be easily seen from the table below, the comparison of the prediction results in the two environments in terms of absolute and relative error gives us a very clear picture of the actual coincidence of the two output values: the orders of magnitude of these errors, in fact, are certainly negligible and probably due to machine accuracies.

Feature	Python output	Simulink output	**Abs error (e-08)**	**Rel error (e-05)**
1	-8.6961e-03	-0.0086961	0.1522	-0.0175
2	5.07480e-02	0.05074805	0.0916	0.0018
3	3.11066e-01	0.31106647	0.1211	0.0004
4	7.74247e-02	0.07742473	0.0107	0.0001
5	-1.4511e-02	-0.0145117	0.0951	-0.0066
6	-1.21510e-03	-0.0012150	0.5519	-0.4542

Table 4.6: Dense network accuracy

In order to use more data to improve prediction performance, the **'mlp_lag30_best.h5'** network was also used. The difference lies in the size of the input batch: in fact, in the latter case, not a single vector of 6 elements is input to the network, but rather a 30x6 matrix; i.e. data from 30 time steps are to be used in order to make the prediction. However, a dense layer is not able to process a matrix as input because it only accepts one vector, so a flatten operation must be performed, i.e. transforming the 30x6 matrix into a vector of 180 elements. Obviously, the weight matrices of each individual layer will also have different dimensions than in the previous case. This specific layer whose objective is to transform a matrix into a vector is called **'flatten layer'**:

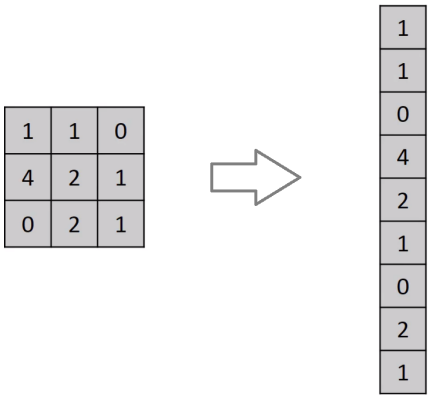


Figure 4.13: Flattening

This operation is already directly present in Simulink and can be done through the specific block **'Reshape'**. The Reshape block changes the dimensionality of

the input signal to a dimensionality that has been specified, using the Output dimensionality parameter.

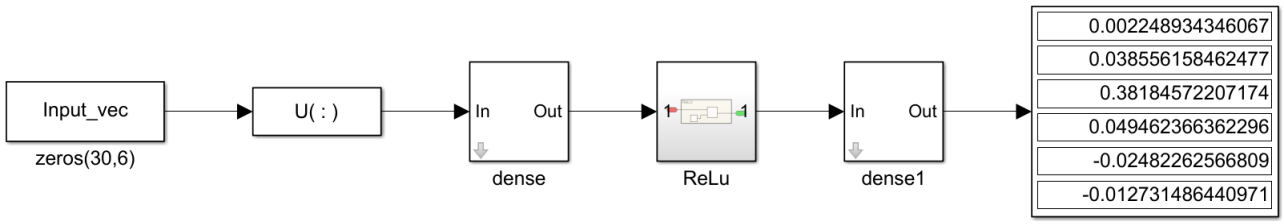


Figure 4.14: Dense network with an input batch of [30,6]

4.2.2 LSTM networks

As in previous cases, we will here validate models in which one or more layers of LSTM appear. The two networks in question are **'lstmmodel.h5** and **'lstm-model_2022.h5'** in which respectively we see one layer and two layers of LSTM appear, followed by a dense layer.

```
model = Sequential()
input_data = tf.random.normal([1,6])
np.savetxt('input_data.txt',input_data)
input_data = tf.reshape(input_data, (1, 1, 6))
model.add(Input(shape=(1,6)))
model.add(LSTM(units=10))
units = 10
model.add(Dense(6,activation='linear'))
model.summary()
model.load_weights('lstmmodel.h5')
y = model.predict(input_data)
```

Figure 4.15: Python code for 'lstmmodel.h5'

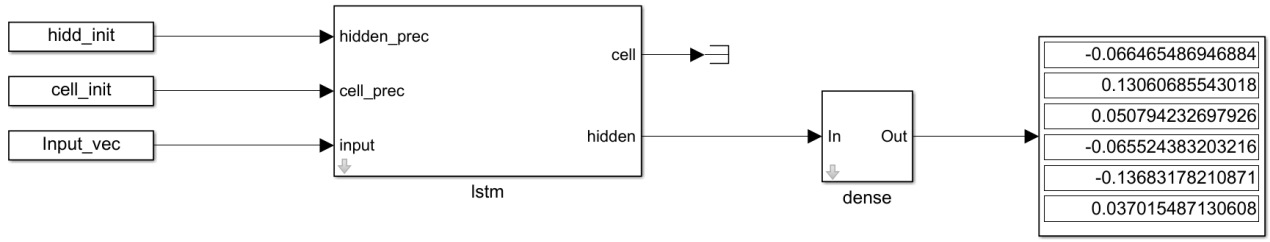


Figure 4.16: Simulink model of 'lstmmodel.h5'

Feature	Python output	Simulink output	**Abs error (e-07)**	**Rel error (e-06)**
1	-0.06646547	-0.06646548	0.169468840	-0.25497275
2	0.13060682	0.130606855	0.354301764	0.27127355
3	0.05079425	0.050794232	0.1730207428	0.340630569
4	-0.06552438	-0.065524383	0.032032161	-0.048885867
5	-0.13683178	-0.136831782	0.02108713	-0.015410992
6	0.03701548	0.037015487	0.071306080	0.19263854195

Table 4.7: LSTM network accuracy

In the case of stacked LSTM cells, it is necessary that the hidden state of the current layer is the input of the next one and that the latter and the cell state are correctly provided for each time step. As already pointed out, the actual prediction output turns out to be the hidden state and not the state of the cell, the latter being a kind of long-term memory to be filtered for current information.

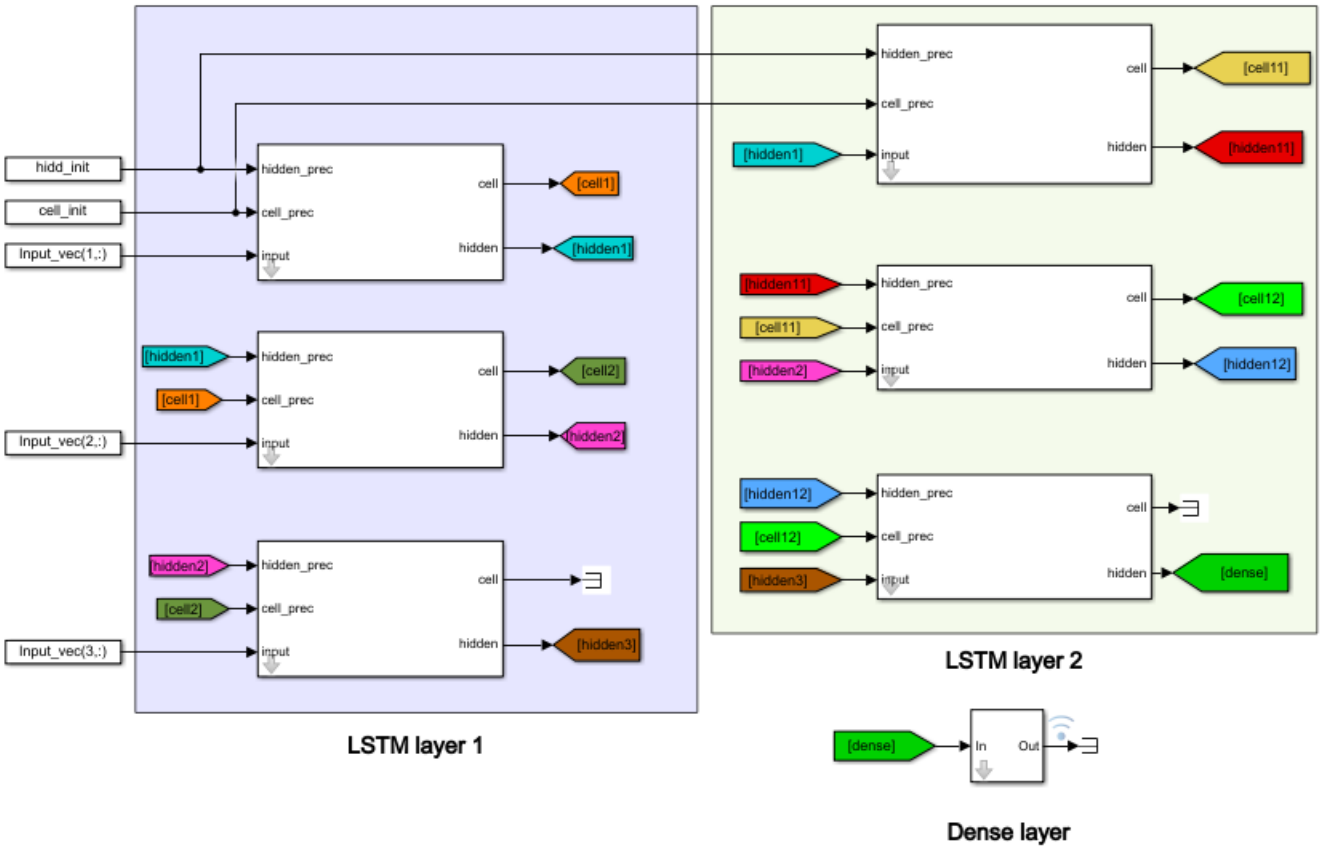


Figure 4.17: LSTM cell validation with 3 timesteps

Feature	Python output	Simulink output	**Abs error (e-07)**	**Rel error (e-05)**
1	0.11504051	0.115040518	0.0801536826	0.0069674310
2	0.04372127	0.04372127	0.018557389	0.004244476
3	0.10359959	0.10359961	0.253486206	0.024467877
4	-0.01406942	-0.01406943	0.146432380	-0.104078476

Table 4.8: LSTM cell with 3 timesteps accuracy

Chapter 5

Models integration

This chapter aims to investigate the use of the network models created to reproduce the machine learning models used in the prediction of forces and torques acting on the satellite as a result of propellant sloshing in the tank, i.e. the models trained internally in a previous study to learn CFD and illustrated in the paper presented by Airbus Defence and Space at the ESA GNC Conference 2023: **'Propellant sloshing effect modelling of spacecraft with Machine Learning'** (Oscar Ortiz Casanova et al.) [6].

The goal of the work exposed in the paper was to model the sloshing of propellant in a tank during manoeuvres in the context of agile missions (which are the most affected by this type of disturbance). During these missions, in fact, especially when the tank fill rate is around 50 %, propellant movement is the main cause of instability in the pointing of specific tagets leading to sub-optimal image quality. Thanks to recent major developments in the development of increasingly accurate CFD codes [34], especially with data from experiments in microgravity (FLUIDICS on the ISS [35]) that have enabled their correct implementation, modelling the phenomenon of sloshing during manoeuvres is now relatively efficient and quite close to reality. However, such computational models, although effective in representing the phenomenon, are still computationally too heavy and therefore their real-time implementation is not yet possible.

Figure 5.1: Astronaut Thomas Pesquet with FLUIDICS tanks on the ISS [36]

It is therefore in this case that Machine Learning methods are likely to be more easily applicable, capable of maintaining the same prediction accuracy with lower

computational costs. In particular, there are three main reasons why these methods are more effective:

- The problem solution does not lie in an effective and accurate analytical method that can reproduce reliable results in different application contexts
- Presence of common features and regularities that can be easily learnt and reproduced
- Possibility of generating training data using CFD calculation

Having accurate models of the forces and torques caused by these fluctuations of propellant on the entire satellite is therefore crucial for creating controllers capable of reducing relaxation times and performing effective closed-loop simulations. The developed networks whose weights were extracted for the Simulink configuration were trained with 600 CFD simulations of which 60% were dedicated to the actual training of the network, 30% for validation and finally 10% for final performance testing. We will therefore see that the models reflect the same predictions given by Python especially for the simplest network model, i.e. the MLP.

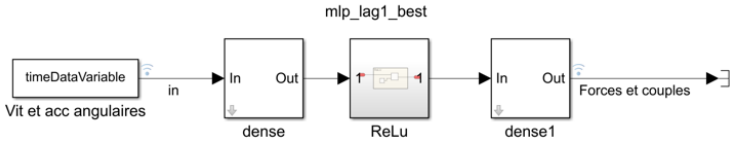


Figure 5.2: Simple MLP net in Simulink

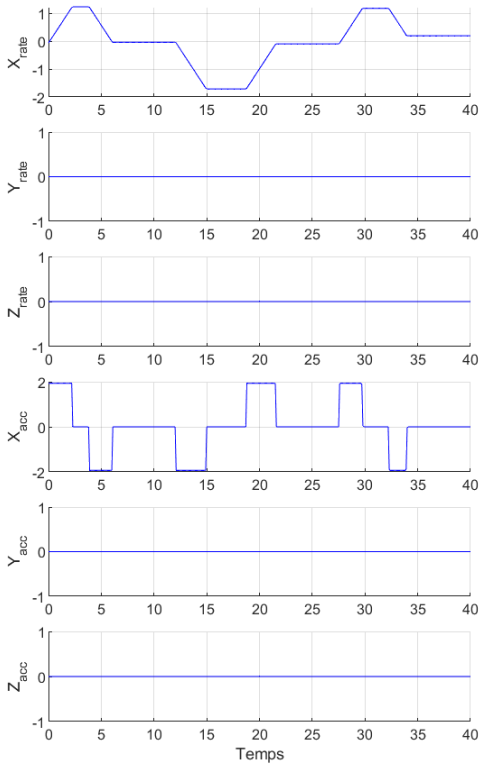


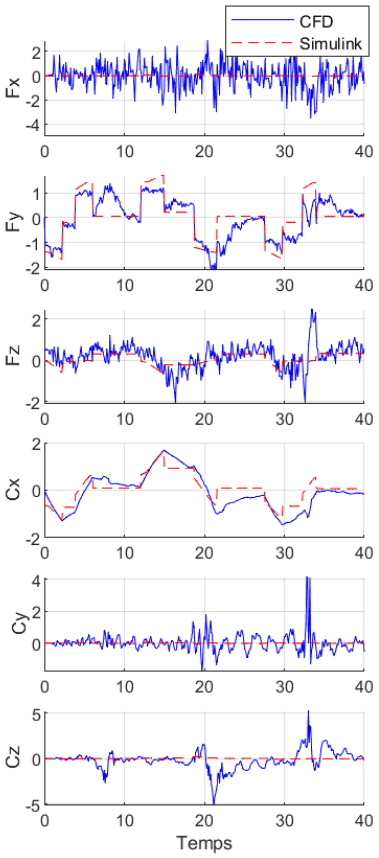
Figure 5.3: Example of input values for angular speed and acceleration

Taking into account the large amplitude differences in the force and torque profiles on the Y and X axes compared to the others, a normalisation of the input and output data of the ML predictions was applied to prevent these differences from generating error propagation and falsifying the results. The scaling operation was performed by calculating the mean m and standard deviation σ on the training data:

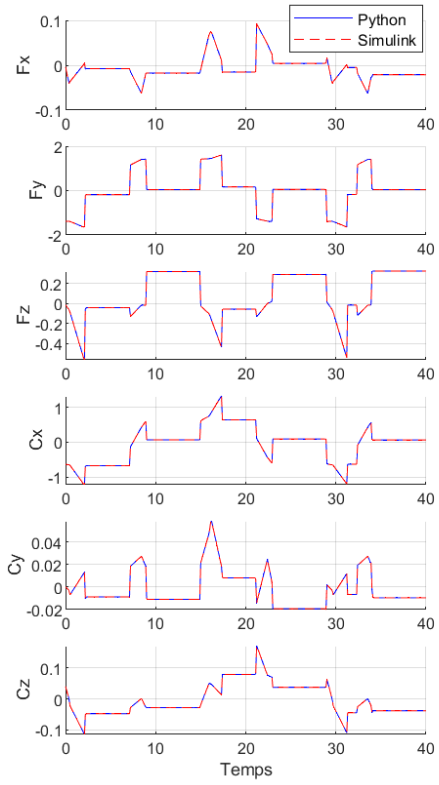
$$Z_{norm} = \frac{Z - m}{\sigma} \tag{5.1}$$

5.1 Open-loop predictions

The following results are obtained in open loop, i.e. considering only the predictions of the simple MLP neural network model. On the bottom left we can see the comparison between the Deep Learning prediction and the CFD data: although we manage to learn the dynamics of the output system well, there are still cases of inaccuracy due to the fact that we are not yet applying the best performing model, i.e. the one with lag.



(a) Comparaison between CFD prediction and MLP prediction in Simulink



(b) Comparaison between ML predictions in Python and Simulink of the MLP model

For each simulation, the 40-second input profiles of velocities and angular accelerations are then loaded from which the force and momentum profiles acting on the satellite can be predicted. A comparison of the open-loop prediction in the

ML case in Pyhton and in Simulink was therefore made in order to verify that the same output predictions were made between the two programming languages. In the following, the model with lag = 30 is shown and also applied to open-loop predictions. As we have seen above, the simple MLP model fails to reproduce certain phenomena that are instead well represented in the graphs below by the model with lag. The dynamics between the two cases are therefore very similar, in some areas of the temporal simulation completely coinciding but nevertheless quite different in the representation of the forces and torques acting on the satellite.

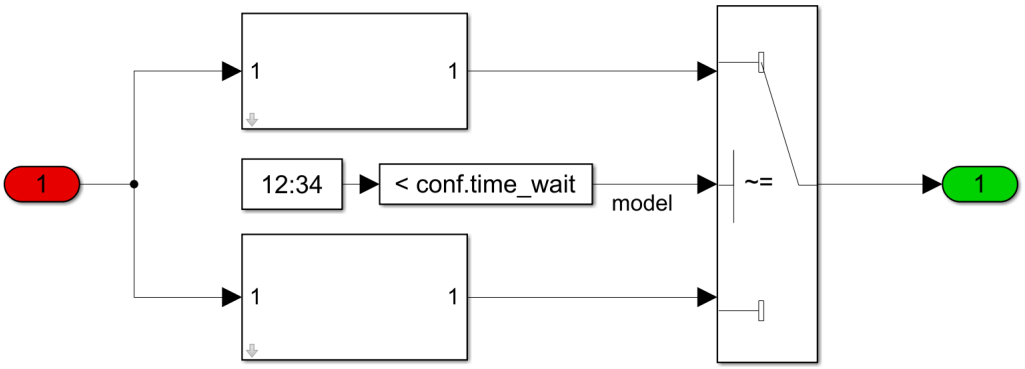


Figure 5.5: MLP with lag

In the following graphs the results obtained with our Simulink models are showed:

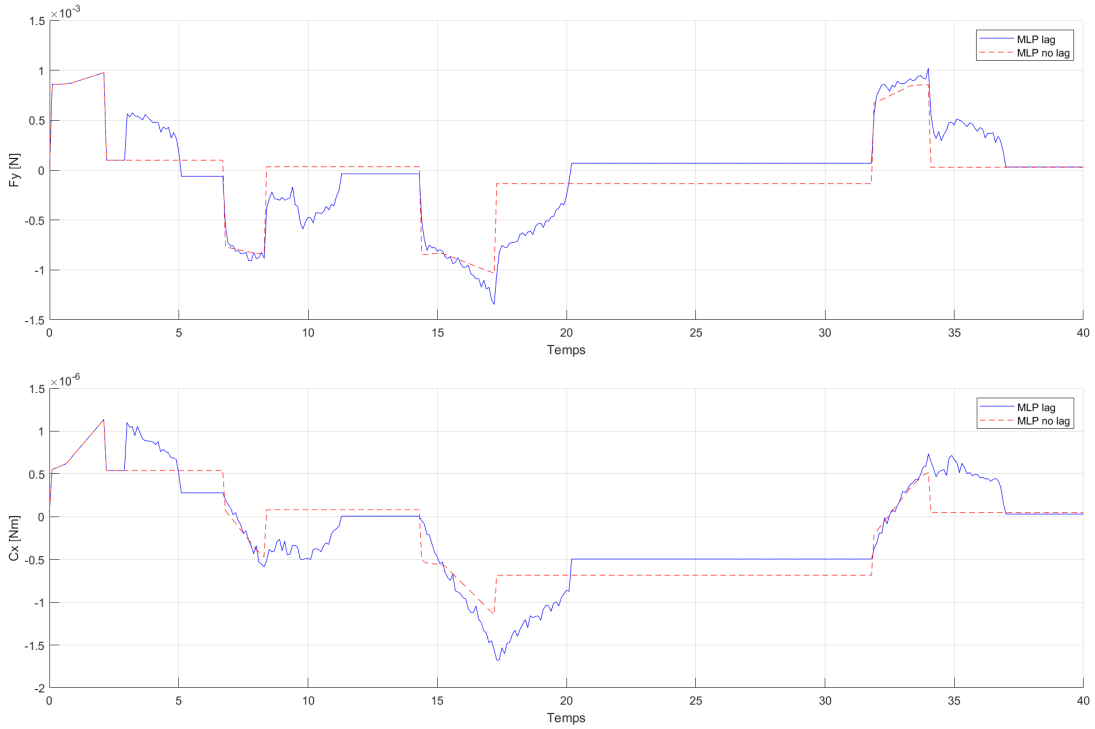


Figure 5.6: Comparison of the predictions of the simple MLP model and the model with lag for the same manoeuvre

5.2 Closed-loop predictions

The following closed-loop simulations were obtained through the model with lag. The latter differs from the former in that it needs the previous 30 time steps to make a prediction, instead of only the previous one. The accuracy therefore is much better and makes it the favored model for learning CFD dynamics and obtaining the output forces and torques, especially having in mind that it is a very simple model to reproduce and train.

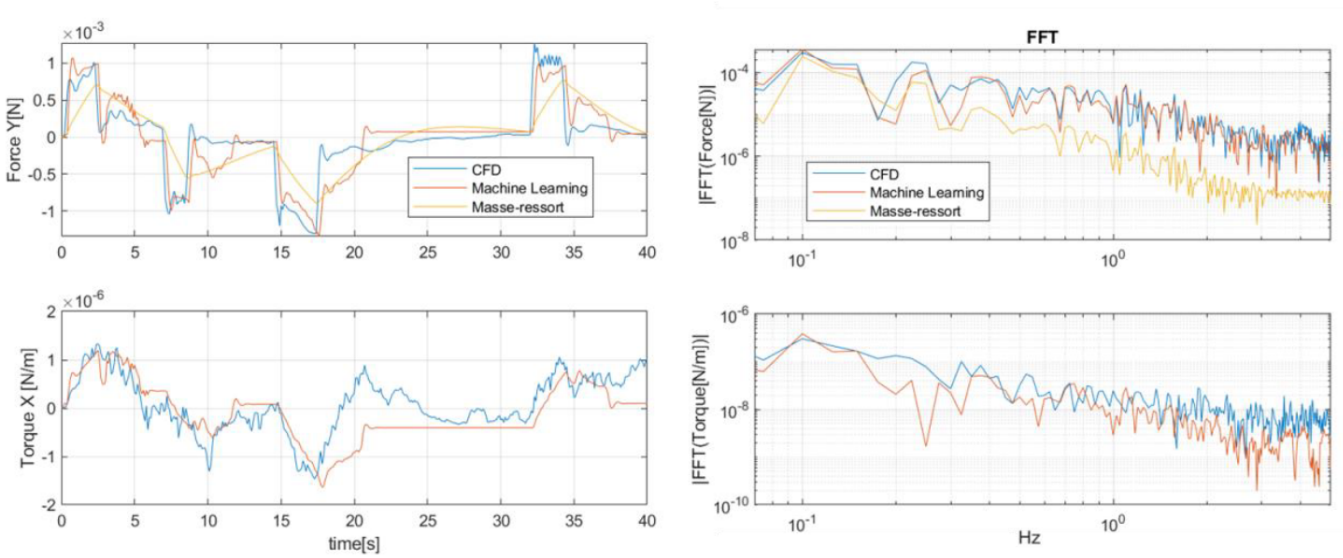


Figure 5.7: Original model accuracy in closed-loop [6]

In the following graphs the results obtained with our Simulink models are showed:

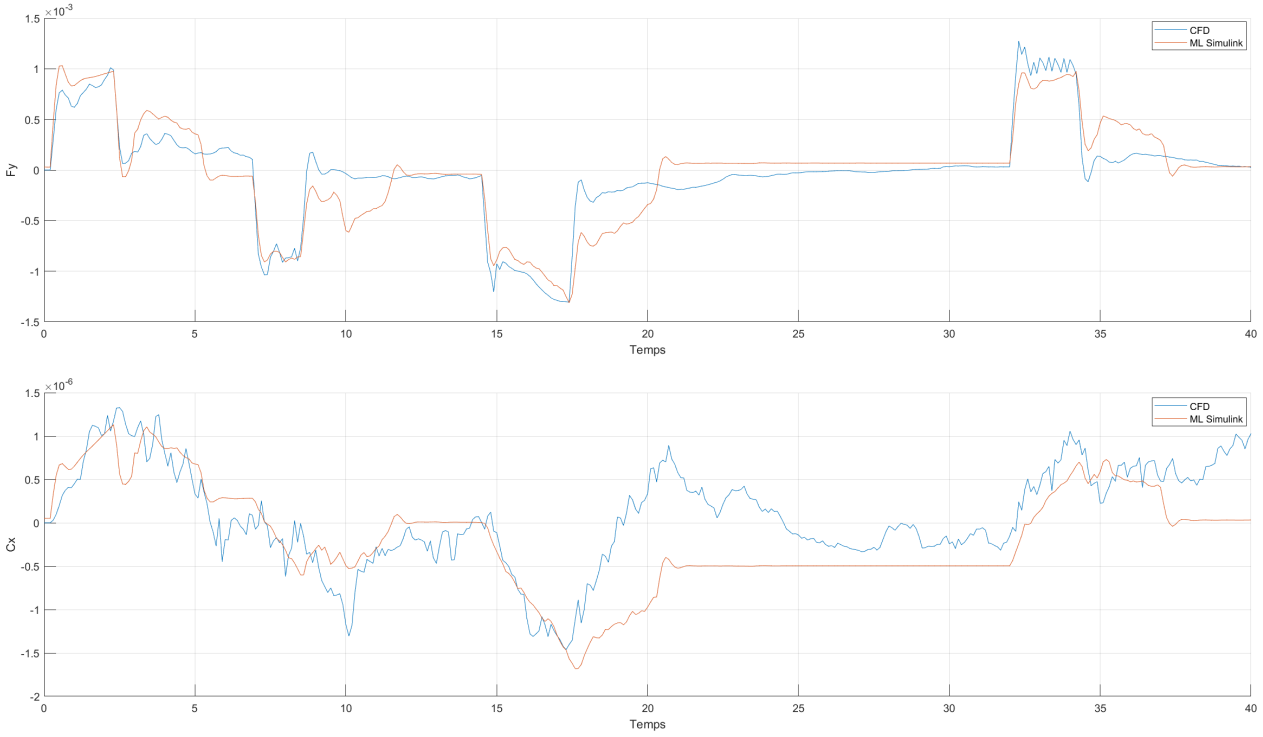


Figure 5.8: Closed-loop time simulations of Fy and Cx

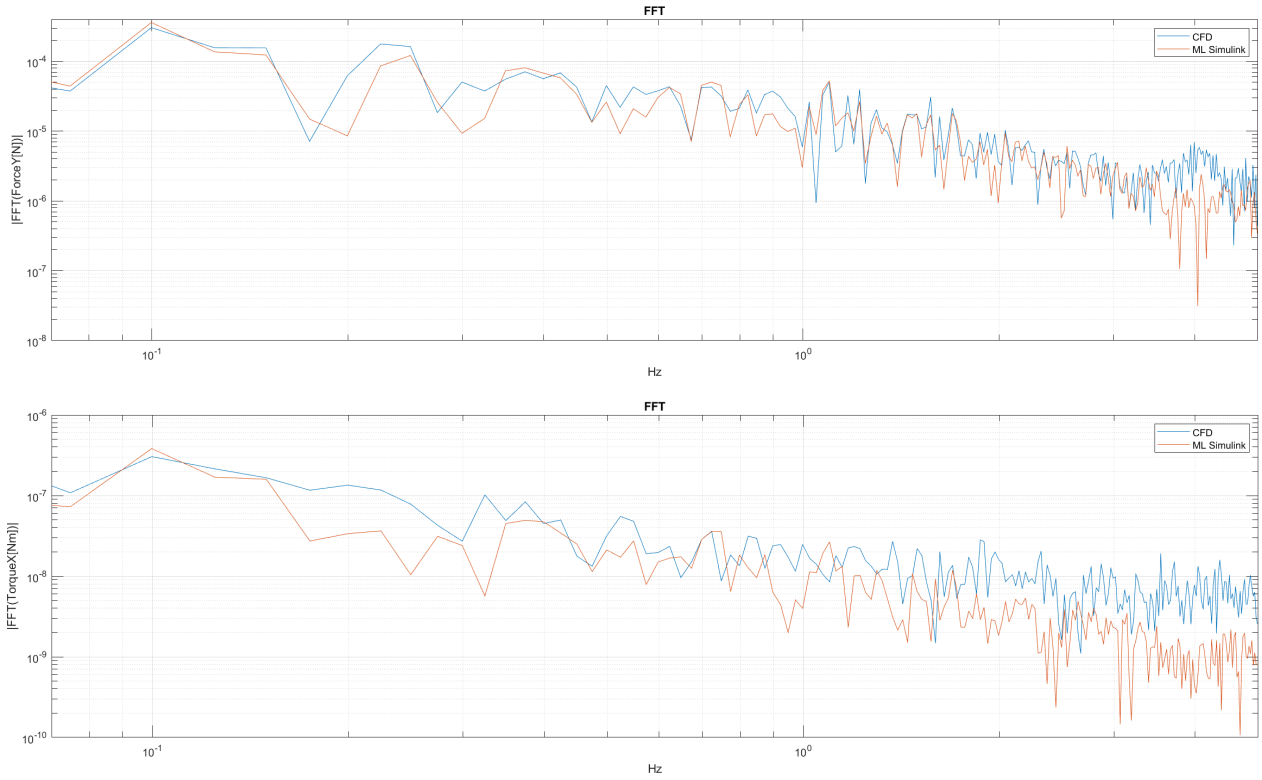


Figure 5.9: Fourier transforms of the quantities Fy and Cx

The first two graphs show by means of time simulations the comparison between CFD and model predictions with lag. At first one can easily see the coincidence of the Simulink models with the original ones reproduced in the article. We can then see that although the model with lag is the most accurate, there are still areas where this accuracy decreases. Helping to visualise the comparison between these two predictions are the last two graphs showing the Fourier transforms of the model dynamics, highlighting the frequency content of the two cases studied. There is here a slight divergence between the CFD and the ML Simulink prediction, especially at high frequencies, a symptom of a model that should still be improved. However, it is worth reiterating that the training times of such a simple MLP network are of the order of a minute for an accuracy that is not far removed from sophisticated CFD models.

Chapter 6

Perspectives

This long research work led us to discover the most important types of neural networks and their important applications. Each of these was then studied in depth and analysed in the theory of the mathematics that composes them in order to be able to reproduce them correctly in Simulink. In the second-to-last chapter, we then realised how the predictions given by the models are almost identical, and how it is indeed possible to reproduce Python's functioning in Simulink, while guaranteeing a versatile structure and a graphical interface that makes it possible to understand how all layers are connected. Equally simple is the management of the weights and configuration elements of the networks, which are easily handled between the two programming languages via the configuration files. Certainly, however, there are still many improvements to be made to the library in order to ensure its simple and effective use while maintaining the same prediction capabilities and accuracy. First of all, the convolution and pooling processes must be made more automatic by facilitating the integration of the Simulink model with the matrix inputs consisting of the images on which the operations are performed. Furthermore, the operations that the shown codes are capable of performing will certainly have to be supplemented with more advanced options that guarantee better analysis performance. Improvements must certainly be made to the utilities function in order to allow a more automatic interpretation of the neural networks imported via .json and .h5 files. In a future perspective, it would indeed be desirable to be able to extract from these not only the structure and main dimensions of each layer and the weight matrices, but also all the weight values. It would therefore be preferable to have a large and unique function in Matlab that is able to extract from the saved files all the elements relating to the architecture of the network and the weight values of the matrices, storing them in a suitable structure in Matlab with different fields. Finally, the part where major improvements should definitely be made is the interface with AOCS simulators. In fact, despite the satisfactory results obtained in terms of prediction between Simulink and Python, it is still very difficult to integrate these networks into simulators due to the differences in code origin. In the context of this work, the networks used in the validation, once the Simulink models have been inserted into the simulators, are found to retain the same qualitative trends with regard to the forces and torques predicted. It is therefore still necessary to investigate the reasons for these differences between the two trends and above all to think about better formatting of the input and output data for better interpretation. The objective is therefore to have ready and efficient network models that can be used in prediction once the optimised weights have been extracted. The use in this case can in fact be

multiple: such networks can in fact constitute a controller that is able to learn the dynamics of a system and synthesise the optimal control law or be used, as in the case of the validated networks, instead of complex models for environments and dynamics. It will certainly be necessary to implement and provide for a better handling of the prediction input and output data for their efficient utilisation and interpretation. In fact, these data will necessarily have to be adapted to the closed cloop AOCS simulations, and the feedforward prediction flows will have to respect the due sampling times and simulation frequencies.
In conclusion, it will be necessary in the future to provide better Python - Matlab data handling and processing interfaces by improving the existing ones created with this work in order to effectively implement neural network models within simulation environments. The challenge of the study in fact lays in the extreme difficulty in retrieving the data fundamental to the use of the networks in the simulator and, above all, in understanding how the source code in Python was connected to it. In this, in fact, physical/dynamic phenomena are more evident that are not perceptible in the pure code and therefore do not make it easy to interpret.

Conclusions

In the global scope of this work, we have had the opportunity to understand how much AI is impacting our society and what aerospace applications its implementation may have the most space in. The study of each type of network has allowed us to understand the mathematics and theory behind these complex mechanisms that appear from the outside as black boxes within which it is very complex to distinguish their elements and architecture. We have therefore had the opportunity to see how in-depth study can lead to the correct reproduction of these complex mathematical models even in a programming language that is not the one for which they were conceived, namely Matlab/Simulink. The development of the models and of the entire library, which was not without difficulty, made it possible to develop a versatile and easy-to-use AI environment, also and above all thanks to Simulink's clear graphical interface, which clearly allows one to see which all the constituent elements of these models are. Definitely helping the Matlab - Python interface were the purposely created functions for reading the network saved files, which allowed a quick reading of the configurations to be tested as well as a simple extraction of the corresponding weights. We have thus been able to demonstrate how the reproduction of the networks is very effective in Simulink without losing any precision compared to Python and thus guaranteeing very good feedforward performance. It is certainly still a long way from having the same versatility and calculation capacity as Python, but in view of exclusive use in prediction of already optimised networks, it is certainly a very good result. One of the main difficulties in processing this research were certainly the correct reproduction, in mathematical and architectural terms, of each network layer so that there was no difference between Python and Matlab prediction. Being black boxes enough in themselves, it was not easy to extrapolate from the source codes of the networks the fundamental elements for their reproduction in a programming language other than the one for which they were conceived. Another major difficulty was in the final part of this internship the integration of these models with the AOCS simulators, which was by no means straightforward and simple and certainly requires more attention. The points that certainly need to be worked on in the future are the optimisation of the networks and the process of uploading weights, but above all the interface with the AOCS simulators and their coding modes. We can therefore say that the objective of creating an AI environment useful to the CNES AOCS service in order to integrate its neural network models and to be able to exploit them in feedforward by using the predictions made has been achieved: each saved file, as we have seen typically .json or .h5, of these networks can in fact be used easily and from these the elements necessary for configuring the models can then be extracted. Uploading the weights makes it possible to have a network ready to make the appropriate predictions, and above all, the networks can fulfil multiple functions, as the models are versatile and dependent on Python output. In mathematical and theoretical terms, in fact, a network retains the same structure

whether it is the prediction of forces and torques or the parameters of a controller after learning the dynamics of a complex system. As this structure is ready and configurable via the library created for this purpose, one only needs the appropriately trained network to be able to reproduce it correctly in Matlab/Simulink and place it in the context of an AOCS simulator.

However, it is worth pointing out the profound difficulties encountered in the integration and management of data from Python. In fact, since Python is a calculation model in its purest form and cannot be physically interpreted, the extraction of data and its use does not integrate well with Simulink, which represents an environment in which physical quantities are of great importance, especially in a simulation environment such as OCEANS. It is therefore important to understand how data is structured and how often it is handled at the execution stage. This is in fact the main reason for the difficulties that did not allow us to effectively reproduce the outputs of the propellant sloshing study. Hence, it is important to bear in mind the great difference between these two totally different environments in the management of data in which the frequency content of the predictions, management and utilisation of the data are also very different. Despite the fact that it was therefore not totally possible to reproduce Python predictions in functional terms in our simulation environment, we still managed to reproduce a mature and valid AI environment. The next steps will certainly concern better management of the data used and a more optimal understanding of the starting Python model, improving the interface with Simulink at the frequency and time level, and adapting the simulation environment to receive input of a format not usual to classic AOCS simulations. This work therefore constitutes the starting point to better investigate how neural network environments obtained with Python can be made more physically interpretable in the Simulink environment, up to their possible and complete integration in simulation tools. In fact, having a clear and visual idea of the mathematics involved in such models, as Matlab/Simulink guarantees, certainly makes it possible to open that black box that Python represents and also to better understand how a data set is treated.

References

[1] Michele Lavagna Stefano Silvestrini. "Deep Learning and Artificial Neural Networks for Spacecraft Dynamics, Navigation and Control". In: *drones* (2022).

[2] Łukasz Łacny. "MODELLING OF THE DYNAMICS OF A GYROSCOPE USING ARTIFICIAL NEURAL NETWORKS". In: *JOURNAL OF THEORETICAL AND APPLIED MECHANICS* (2012).

[3] Roberto Furfaro. "DEEP LEARNING FOR AUTONOMOUS LUNAR LANDING". In: *AAS 18-363* ().

[4] *Vision-Based Navigation (VBN) processing solutions on a space-grade processor*. 2022. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/Vision-Based_Navigation_VBN_processing_solutions_on_a_space-grade_processor.

[5] Jules Simo Zibin Sun and Shengping Gong. "Satellite Attitude Identification and Prediction Based on Neural Network Compensation". In: *Space:Science and Technology* (2023).

[6] Oscar Ortiz Casanova et al. "Propellant sloshing effect modelling of spacecraft with Machine Learning". In: *ESA GNC 2023* (2023).

[7] *The Concept of Artificial Neurons (Perceptrons) in Neural Networks*. 2021. URL: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc.

[8] *Logistic Regression — Detailed Overview*. 2018. URL: https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc.

[9] *Activation Functions in Neural Networks*. 2017. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[10] *Softmax Activation Function Explained*. 2020. URL: https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60.

[11] *Activation function*. 2023. URL: https://en.wikipedia.org/wiki/Activation_function.

[12] *Derivative of the Softmax Function and the Categorical Cross-Entropy Loss*. 2021. URL: https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1.

[13] S.Jadhav. "PERFORMANCE EVALUATION OF MULTILAYER PERCEPTRON NEURAL NETWORK BASED CARDIAC ARRHYTHMIA CLASSIFIER". In: (2012).

[14] *Classification of Neural Network Hyperparameters*. Sept. 2022. URL: https://towardsdatascience.com/classification-of-neural-network-hyperparameters-c7991b6937c3.

[15] *How Does Back-Propagation Work in Neural Networks?* 2022. URL: https://towardsdatascience.com/how-does-back-propagation-work-in-neural-networks-with-worked-example-bc59dfb97f48.

[16] *Difference Between a Batch and an Epoch in a Neural Network.* 2022. URL: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.

[17] *Loss Functions and Their Use In Neural Networks.* 2022. URL: https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9.

[18] *How to Convert an RGB Image to Grayscale.* 2019. URL: https://e2eml.school/convert_rgb_to_grayscale.

[19] *Convolutional Neural Networks cheatsheet.* 2019. URL: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks.

[20] *Pengenalan Convolutional Neural Network (CNN).* 2019. URL: https://skillplus.web.id/pengenalan-convolutional-neural-network-cnn/.

[21] Jianxin Wu. "Introduction to Convolutional Neural Networks". In: *National Key Lab for Novel Software Technology* (2017).

[22] *CNN | Introduction to Pooling Layer.* URL: https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/.

[23] *Convolutional Neural Network (CNN) and its Application- All you need to know.* 2021. URL: https://medium.com/analytics-vidhya/convolutional-neural-network-cnn-and-its-application-all-u-need-to-know-f29c1d51b3e5.

[24] *Understanding RNN and LSTM.* 2019. URL: https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e.

[25] URL: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/keras/layers/recurrent.py.

[26] *LSTM: Understanding the Number of Parameters.* 2020. URL: https://www.kaggle.com/code/kmkarakaya/lstm-understanding-the-number-of-parameters.

[27] *Stacked Long Short-Term Memory Networks.* 2019. URL: https://machinelearningmastery.com/stacked-long-short-term-memory-networks/.

[28] Nicholas Lee. "Study of long short-term memory in flow-based network intrusion detection system". In: *Journal of Intelligent and Fuzzy Systems, vol. 35* (2018).

[29] *Understanding LSTM Networks.* 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[30] *Understanding Input and Output shapes in LSTM | Keras.* 2019. URL: https://shiva-verma.medium.com/understanding-input-and-output-shape-in-lstm-keras-c501ee95c65e.

[31] Rahul Dey Fathi M. Salem. "Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks". In: *Circuits, Systems, and Neural Networks (CSANN) LAB* ().

[32] *Understanding GRU Networks*. 2017. URL: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.

[33] *Part I: Saving and Loading of Keras Sequential and Functional Models*. 2020. URL: https://medium.com/swlh/saving-and-loading-of-keras-sequential-and-functional-models-73ce704561f4.

[34] Alexis Dalmon. "Simulation numérique du ballottement d'ergol et modélisation de l'interaction fluides-membranes". In: *Phd Thesis University of Toulouse 3* (2018).

[35] R.Pierre. "Fluid dynamics in space experiment". In: *ESA GNC 2017* (2017).

[36] *ISS: FLUIDICS*. 2023. URL: https://www.eoportal.org/other-space-activities/iss-fluidics#references.

Acknowledgements

A Gianluca e Riccardo, per essere stati dal primo all'ultimo momento porto sicuro e riferimento incrollabile, le prime persone a cui ho voluto assomigliare e con cui ho condiviso principi e valori, Torino è cominciata con voi e non finirà mai a ricordarci sempre dove abbiamo cominciato.

A Marco e Ferdinando, per aver sempre rappresentato il punto di vista in più, l'opinione non ordinaria e l'affetto incondizionato in ogni momento, la caparbietà, l'ostinazione, la sfrontatezza, il cambiamento, il caos e la quiete, la possibilità nell'incertezza e l'opportunità nella sfortuna.

Ad Antonio e Cesare, con cui ho iniziato a vedere il grigio tra bianco ed nero, ad apprezzare le sottili sfumature, i dettagli e le imperfezioni, a misurarmi con il mondo e dimensionare la mia voglia di cambiarlo.

Ad Antonio Finelli, per il tuo cuore e la tua spontaneità, per essere stato in questa città mio fratello maggiore, per dare sempre tutto senza aspettare mai nulla in cambio, per avermi sempre accolto a braccia aperte.

A Ciro e Francesco, per essere sempre stati fuori dagli schemi incarnando astuzia, coraggio e determinazione. Per avermi insegnato ad essere sempre positivi e a vedere il lato positivo delle cose, ad allontanare ogni inquietudine e a vivere ogni momento.

A Giulia, per la vita che sei, per il tuo cuore e la tua purezza, da te ho imparato a vivere il presente.

Ad Anna, Marta e Santo per avermi fatto capire che si può sempre migliorare.

A mio cugino Gianluca, mio fratello maggiore ed esempio costante di vita.

A mio fratello Marco, che da sempre mi sostiene e mi è vicino.

A Torino, la città che mi ha accolto e mostrato la vita.