Máster Universitario en Matemática Avanzada

Trabajo Fin de Máster

Recurrent neural networks for ornithopter trajectory optimization

Presented by:
Luis David Pascual Callejo

Supervised by:
Dr. José Miguel Díaz Báñez
Dpto de Matemática Aplicada II
Universidad de Sevilla

November, 2020

Abstract

Path planning is a widely studied subject due to its vast number of applications, specially for robots and unmanned vehicles. Strategies to solve it can be categorised as classical methods and heuristic methods, each one with its own advantages and disadvantages. Generally speaking, analytical methods are very complex for actual applications, whereas the heuristic methods are penalized by the size of the search space. For the case of unmanned aerial vehicles this penalization cannot be afforded, since due to weight and reaction time constrains, paths should be computed on line with fast and computationally light algorithms. In this work the use recurrent neuronal networks to contour this problem is proposed. The neuronal network is tasked with learning the underlying optimal trajectory flight dynamics, which are in turn numerically estimated by a time consuming heuristic method. More precisely, a recent heuristic method (OSPA) is used to compute a set of optimal trajectories for the ornithopter and then, the neuronal network is tasked with learning the underlying function from it. The goal is to obtain similar performances to the heuristic method with much faster computation times. The effectiveness and efficiency of the proposed algorithm are demonstrated through numerical simulations on validation data sets. In addition, far from blindly applying a recurrent neuronal network, a mathematical framework will be developed in other to justify the choices made and the resulting performance. Such framework will be supported by the universal approximation theorem, the algebraic feedforward neuronal network equations and the maximum likelihood method.

Resumen

El cálculo de trayectorias es un área con múltiples aplicaciones, especialmente para robots y vehículos no tripulados. Las estrategias para su resolución pueden ser categorizadas en métodos clásicos y heurísticos, cada uno con sus ventajas e inconvenientes. De manera general, los métodos analíticos son demasiado complicados para aplicaciones reales mientras que los heurísticos son penalizados por el espacio de búsqueda.

En el caso de vehículos aéreos no tripulados, dicha penalización no es aceptable debido a los requerimientos de tiempo de reacción y peso, lo que conlleva la necesidad de algoritmos ligeros y rápidos. En este trabajo se propone el uso de redes neuronales recurrentes para salvar dichas limitaciones. La red es encomendada con la tarea de aprender la dinámica de vuelo de trayectorias óptimas que han sido previamente estimadas numéricamente por un algoritmo heurístico. En concreto, se utiliza un novedoso algoritmo heurístico (OSPA) para calcular un set de trayectorias óptimas y la red neuronal recurrente tiene la tarea de aprender de él la función subyacente. El objetivo es obtener métricas similares en tiempos de computación mucho menores. La efectividad y eficiencia de la red será validada mediante simulaciones en un set se validación. Adicionalmente, lejos de aplicar indiscriminadamente una red neuronal, un marco matemático será desarrollado para justificar las elecciones hechas y los resultados obtenidos. Dicho marco será desarrollado a partir del teorema de aproximación universal, las ecuaciones algebraicas de la red y el método de máxima verosimilitud.

Acknowledgements

I would like to thank Dr. José Miguel Díaz-Báñez for giving me the opportunity to work on this project, for his guidance and patience throughout the project and for their instructions and support over this time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and goal

As an aeronautical engineer, I have been passionate about flight and flight dynamics. As a control engineer, how this flight could be made autonomously. On the other hand, mathematics has always been a fundamental element of robotics research, being path planning one of the most studied problems in the interplay of the two fields. This work gives me the opportunity to wrap all these things together by solving a flight path optimization problem with neural networks using a probabilistic framework. The main goal of this work consists on computing the trajectory optimization of an ornithopter by learning the underlying flight dynamics by means of a recurrent neural network.

An ornithopter is a flapping wing airplane DeLaurier (1994) which is normally designed to imitate the flight of a bird, a bat or an insect. Due to the complexity of its flight dynamics, autonomous flight is still a challenging task Baek et al. (2011). This task is even harder when there is the need not only for the ornithopter to travel from point A to B, but also for the ornithopter to do this through an optimal trajectory, where we define as trajectory optimization the process of designing a trajectory that minimizes (or maximizes) some measure of performance while satisfying a set of constraints, see for example Ross (2009). A recent approach for the problem of optimizing ornithopters trajectories is Rodríguez et al. (2020), which uses the planner OSPA (which stands for Ornithopter Segmentation-based Planning Approach) applied to real ornithopter prototype. Despite outperforming alternative probabilistic kinodynamic planners both in cost (total energy) and accuracy (distance to the target), the high computational load makes it too slow for real mid-range ornithopter online applications. On the contrary, as any other unmanned air vehicle, ornithopters require low weight and fast reaction times. Taking into account the low computational resources available, any proposed strategy should then rely on a simple and low demanding algorithm. Therefore, a much more simplistic approach is needed to reduce the computational time from minutes

Figure 1.1: Machine Learning

to fractions of a second.

This is where neuronal networks appear into scene, as they are widely known for solving complex problems with simple linear algebra operations. If a neuronal network can be trained to estimate an optimal trajectory, then it can be embarked in the ornithopter to efficiently compute the trajectory with little need of computation resources and time. Building on the ability of the OSPA algorithm to compute optimal trajectories for the ornithopter, our recurrent neuronal network will be tasked with learning the underlying flight dynamics from it. The goal is to obtain similar performances to the heuristic method with much faster computation times.

Instead of applying blindly a neural network, a mathematical framework will be given in other to justify the choices made and the corresponding performance results. Despite the wide spread use of neuronal networks, there are not many mathematical models able to explain their effectiveness. Therefore, many times neuronal networks are treated as black boxes where data is poured in one side and answers are obtained from the other side as depicted on Figure 1.1. When linked to math, neuronal networks are often associated with linear algebra as most of the common algorithms make extensive use of it. However, when interested in explaining their effectiveness, other mathematical approaches as probability or information theory are more suitable Shwartz-Ziv & Tishby (2017).

To sum-up, the goal of this work is to define, train and evaluate a recurrent neuronal network to the concrete case of the computation of the optimal trajectory for an ornithopter, so the complex underlaying flight dynamics are learnt. The training data is obtained from the results of the OSPA heuristic approach proposed in Rodríguez et al. (2020). The choices made and results will be discussed and justified using a probabilistic approach.

1.2 Related work

The ornithopter optimal trajectory problem itself is defined in Rodríguez et al. (2020), where the authors have developed an algorithm (OSPA) to compute minimal energy consumption trajectories. This thesis builds upon their experience in optimal algorithms and explores Recurrent Neural Networks as a way to contour the problems they found, especially the long computation times.

The application of neural networks to trajectory optimization is not new, see Glasius et al. (1995), Horn et al. (2012), Yang & Meng (2000) amongst others. Even the application of Recurrent Neural Networks (RNNs) to trajectory optimization is not new Wang (1999). However, most authors tackle this problem directly, embedding the trajectory constraints in the neural network equations and using the NN's loss function to optimize the trajectory. Generally speaking, a lot of work exists on using neural networks to directly solve Non-Linear Programming problems (NLP), for example approximating an optimal controller for Unmanned Aerial Vehicles Xu et al. (2007). However this approach is very complex and can only be afforded if the vehicle mechanics are simple.

As introduced in section 1.1, the flight dynamics from our ornithopter are by no means simple, but rather it is composed by a complex nonlinear differential equation system Rodríguez et al. (2020). Including such complexity in our network's loss function or network's optimization algorithm would be an impossible endeavour. Therefore the approach of this work consists on designing a NN that will mimic the behaviour of the OSPA algorithm just by learning from a data set of optimal trajectories generated by OSPA. This type of NN application is called function approximation. Previous work on NN function approximation for trajectory optimization can be found in Mordatch & Todorov (2014).

In this thesis we are going to apply a very special type of neural networks, the recurrent neural networks. The reason behind is because this type of networks are able of modeling dynamic systems, from electronics Luongvinh & Kwon (2005) to engines Tan & Saif (2000).

Instead of a blind implementation, a mathematical framework is given by the use of three pillars:

- The universal estimator theorem.
- The development of the NN algebraic equations derived from the neuron ones.
- The use of the maximum likelihood

There are many attempts to understand how neural networks work, but not that much results. A first, general overview on the mathematical aspects of neural networks can be found on Goodfellow et al. (2016). Although NN are driven by algebra, this book already introduces probably and information theory concepts

as a mean to understand NN behaviour. In Murphy (2012) the reader can get a deeper dive into a probabilistic perspective of machine leaning and neural networks in particular. However, the best insights are given in Shwartz-Ziv & Tishby (2017), where the author uses information theory to present a comprehensive theoretical understanding on how Deep Neural Networks learn. This approach is further explained on Saxe et al. (2019) by means of the information bottleneck theorem.

The RNN proposed in this work is extremely simple: only one recurrent layer with 11 neurons. Therefore, there is no point in applying the results from Shwartz-Ziv & Tishby (2017), which are intended for deep configurations. Instead, we have taken the fundamental concepts from probability and information theory and developed them into a suitable mathematical framework for our simple model.

1.3 Preliminaries

1.3.1 Introduction to neural networks

Neural networks overview

The goal of a feedforward network is to approximate some unknown function f by learning the values of the parameters θ that result in the best function approximation $f^*(x;\theta)$. It's goal is thus to output for each value of x (features) a value $\hat{y} = f^*(x)$ that is closest to y (label).

The network is associated with a directed acyclic graph (DAC) describing how the functions are composed together. For example, if we have three consecutive layers $f^1(x)$, $f^2(x)$ and $f^3(x)$ connected in a chain, their composed function is $f(x) = f^3(f^2(f^1(x)))$ Goodfellow et al. (2016).

Furthermore, in accordance with the universal approximation theorem, standard multilayer feedforward networks are capable of approximating any measurable function to any desired degree of accuracy.

Theorem 1.3.1. *Hornik et al. (1989). A feedforward network with a linear output layer and at least one hidden layer with any continuous squashing function can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.*

The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well, see Hornik et al. (1990). This can be extended to the use of rectified linear activation functions (RLUs)(Leshno et al. (1993)).

Definition 1.3.2. *A function* $\Psi : R \to [0,1]$ *is a squashing function if it is non-decreasing,* $\lim_{\lambda\to\infty} \Psi(\lambda) = 1$ *and* $\lim_{\lambda\to-\infty} \Psi(\lambda) = 0$

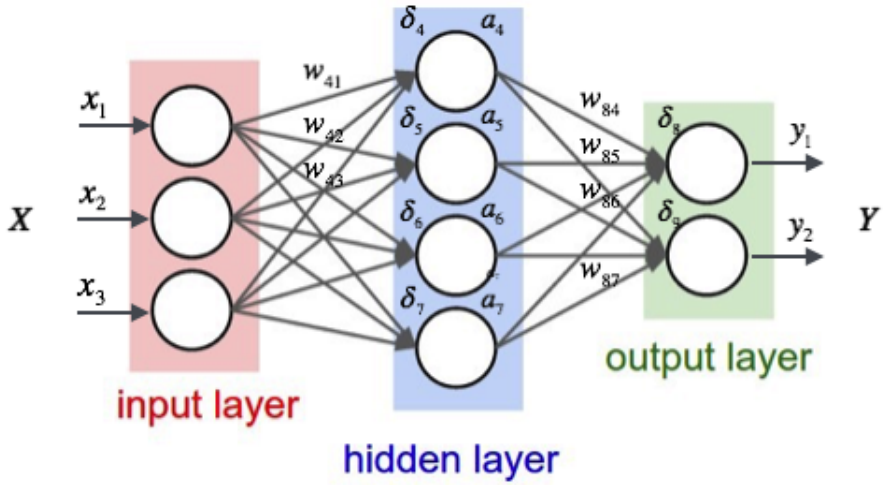

Figure 1.2: Feedforward neuron network schema.

These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations, until the output y. There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks, Goodfellow et al. (2016).

In Figure 1.2, a feedforward neuronal schema can be seen. Each color represents a layer, which is composed by several neurons. The first layer is called the input layer and the last layer is called the output layer. Any other layer in between is called a hidden layer.

As it can be seen, for every layer k, each neuron n_i^k in position i is connected to each of the neurons of the next layer via a set of weights w_{ji}^k. Therefore, w_{ji}^k denotes the connection weight from neuron i belonging to layer k to the neuron j belonging to the layer $k-1$. At every neuron, the weighted sum of the outputs from the neurons of previous layer is used to compute its output, as we will be see next in detail.

Neuron equations

Figure 1.3 shows how a single neuron works. Every neuron n_i^k has:

- A set of weights w_{ji}^k, that connect the neuron to the outputs of the precedent layer.
- A bias b_i^k, which can we rewritten as w_{0i}^k if we add for every neuron an additional input $o_0^{k-1} = 1$.

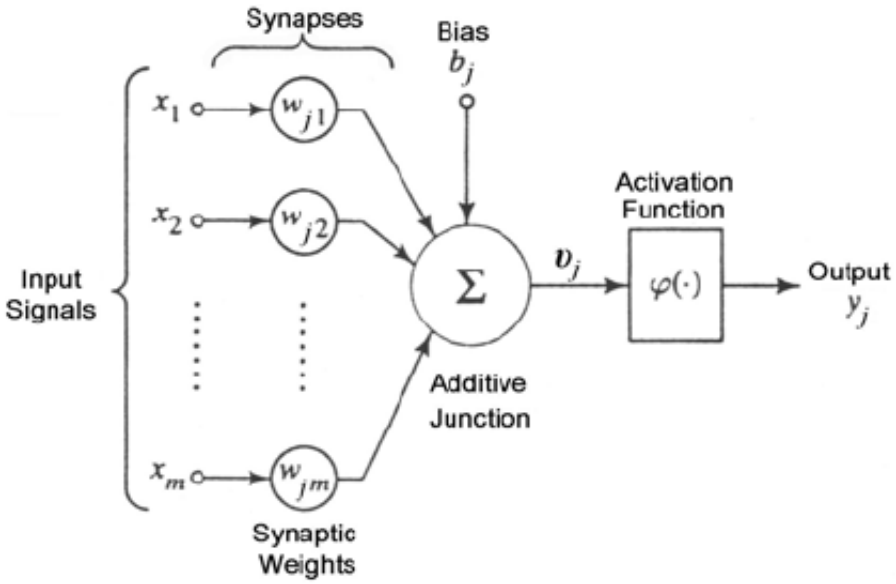

Figure 1.3: Single Neuron diagram in Guarnieri et al. (2006).

- An activation function $g(\cdot)$ which should be continuous and inversible. This functions transforms the sum of all weighted inputs including the bias (a_i^k), into the final neuron output o_i^k.

All these transformation can be summed-up in the equations below:

$$a_i^k = b_i^k + \sum_{j=1}^{r_{k-1}} w_{ji}^k o_j^{k-1} = \sum_{j=0}^{r_{k-1}} w_{ji}^k o_j^{k-1} \qquad (1.1)$$
$$o_i^k = g(a_i^k)$$

As it can be deduced, the parameters θ to be optimized in the neuronal network corresponds to the weights w_{ji}^k, including the bias w_{0i}^k.

Neural Network parameter optimization

In 1.3.1 it has been introduced how a neural network form a parametric family $\{f^*(\cdot\,;\theta) \mid \theta \in \Theta\}$, where Θ is called the parameter space. The family $f^*(\cdot\,;\theta)$ is given by the neural network architecture, which will be covered in 1.3.2. Our goal here is to find the parameter $\hat{\theta}$ (which corresponds to the optimal weights $\hat{w}_{ji}^k$) so that $f^*(x;\hat{\theta})$ best approximates the true underlying function f. As in any optimization problem, a cost function has to be defined. Although any cost function can be chosen, the usage of the maximum log likelihood framework is selected here because it is the best estimator asymptotically in terms of its rate of convergence as the number of examples $m \to \infty$, see Goodfellow et al. (2016).

Definition 1.3.3. *Goodfellow et al. (2016) Consider a set of m examples* $X = x^1, ..., x^m$ *drawn independently from the true but unknown data generating distribution* $p_{data}(x)$*. Let* $p_{model}(X;\theta)$ *be a parametric family of probability distributions over the same space indexed by* θ*. The maximum likelihood estimator for* θ *is then defined as:*

$$\theta_{ML} = \arg\max_{\theta} p_{model}(X;\theta) = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x_i;\theta) = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x_i;\theta)$$

Note that the Likelihood function it is just the joint probability density function for the independent and identically distributed random variables $p_{model}(x_i;\theta)$. Here our model tries to approximate the underlying data generation distribution $p_{data}(x)$. Finally, the Log of the likelihood has been applied for convenience reasons, which is possible since the log is monotonic and thus the θ_{ML} will not differ.

If we rescale the expression above by the number of samples m, it can be rewritten as:

$$\theta_{ML} = \arg\max_{\theta} \mathop{\mathbb{E}}_{X}[\log p_{model}(x_i;\theta)]$$

In the cases we are interested in predicting Y given X, we generalize it to the conditional probability $P(y|x;\theta)$ as:

$$\theta_{ML} = \arg\max_{\theta} L(x;\theta) = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(y_i|x_i;\theta)$$

Neural Network optimization algorithm

The simplest optimization algorithm (and the one we are going to use) is the gradient descendent back propagation. Actually, the most used optimization algorithms are built upon it. The gradient descendent simply consist on changing the value of the parameters θ in the negative direction of the gradient of the cost function $J(\theta)$ with regards to θ. In our case, the cost function is the negative of the log likelihood.

$$\begin{aligned} \theta^{t+1} &= \theta^t - \alpha \frac{\partial J(x, \theta^t)}{\partial \theta} \\ J(X,\theta) &= -L(x,\theta) \end{aligned} \tag{1.2}$$

The term back propagation comes from the fact that calculation of the gradient proceeds backwards through the network, with the gradient of the final layer of weights being calculated first and the gradient of the precedent layer's of weights being calculated one at a time until the first layer is reached. The efficiency of the algorithms comes from the fact that partial computations of the gradient in one layer are reused for computation of the gradient in precedent layers recursively.

This is a consequence of applying the chain rule to the partial derivative of the cost function with regards to every weight.

For every input-output pair (x_i, y_i) we obtain the following result:

$$\frac{\partial J}{\partial w_{ij}^k} = \frac{\partial J}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k}$$

The first term can be rewritten as:

$$\delta_j^k = \frac{\partial J}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \frac{\partial J}{\partial a_l^{k+1}}\frac{\partial a_l^{k+1}}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \delta_j^{k+1}\frac{\partial a_l^{k+1}}{\partial a_j^k}$$

where r^{k+1} denotes the number of nodes in the next layer (k+1). Taking into account that by definition $a_l^{k+1} = \sum_{j=1}^{r^k} w_{jl}^{k+1} g\big(a_j^k\big)$, this term can be expressed as:

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_j^{k+1}\frac{\partial a_l^{k+1}}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \delta_j^{k+1} w_{jl}^{k+1} g'\big(a_j^k\big)$$

The second term can be rewritten as:

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}\left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1}\right) = o_i^{k-1}$$

Combining again the first and second terms, we finally obtain the partial derivative of the error function J with respect to a weight w_{ij}^k:

$$\frac{\partial J}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'\big(a_j^k\big) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1}\delta_l^{k+1}$$

Note that this partial derivative only depends on the errors δ_l^{k+1} at the next layer, whereas the rest of the values were calculated during the forward propagation of the input values x_i through the network.

To wrap-up, feed forward neural networks are trained in two phases:

- The forward phase, where input values x_i flow forward to compute the output value y_i
- The backwards phase, where for each pair input-output, error values flow backwards to compute the gradient.

1.3.2 Neural network architecture

The neuronal network architecture and activation functions will define the parametric family $\{f^*(\cdot\,;\theta) \mid \theta \in \Theta\}$. They will be chosen in order to have a balanced network capacity. The capacity is defined by:

- Depth and width of the network: generally speaking, the deepest (more layers) and widest (more neurons per layer) the greatest number of parameters will be available and thus a greater extent of the training information can be learned.

- Activation functions and general architecture: they will define the representational capacity of the model, or in other words, the set of functions that can be learned by the NN. For example, if we only use linear functions as activation functions $g(\Delta)$, we will not be able to capture non-linear behaviours in our training data. Regarding the general architecture, adding feedback loops in our architecture will allow us to capture sequential/dynamical behaviours as it will be explained for the recurrent neuronal networks in 1.3.3.

The adjective balanced is used since NN with high capacity are prone to overfitting, whereas NN with low capacity are prone to underfitting

- Underfitting occurs when the network is not able to capture enough information from the training set and thus the learned model $f^*(x;\theta)$ is too simple to correctly approximate the underlaying function f.

- Overfitting occurs when the network is not able to generalize the information learned from the training data. That is, the learned model $f^*(x;\theta)$ is too specific for a particular set of training data and when a new unseen input x is fed into the network it is not able to make a good estimation.

This problem can be described as a bias-variance problem as follows:

First, we are going to use the Mean Squared Error (MSE) as a measure of our performance. We will see later on on 3.3 how minimizing the MSE is equivalent to maximizing the Likelihood under certain hypothesis.

The MSE is defined as:

$$MSE = \mathbb{E}_{x \in data}[(y - f^*(x))^2]$$

We also define bias and variance of our approximation function $f^*(x;\theta)$ as:

$$bias[f^*(x)] = \mathbb{E}_{D \subset data}[f^*(x)] - f(x); var[f^*(x)] = \mathbb{E}_{D \subset data}[f^*(x) - \mathbb{E}_{D \subset data}[f^*(x)])^2]$$

Here $\mathbb{E}_{D \subset data}[f^*(x;\theta)]$ is the expectancy of $f^*(x;\theta)$ over the different training sets D. Every different training set would lead to different optimal parameters θ and thus different values of $f^*(x;\theta)$ for the same sample x.

Finally we can assume that our data has some data noise with regards to the true values of the unknown underlaying function $f(x)$ and that this noise has zero mean.

$$\begin{aligned} y &= f(x) + \epsilon \\ \mathbb{E}[\epsilon] &= 0 \end{aligned}$$

We can expand the expectancy of the Squared Error of a sample over the different training sets D as:

$$\begin{aligned}
&\mathbb{E}_D(y - f^*(x))^2 = \mathbb{E}_D(f(x) + \epsilon - f^*(x))^2 \\
&= \mathbb{E}_D[(f(x) - f^*(x))^2] + \mathbb{E}_D[\epsilon^2] + 2\mathbb{E}_D[(f(x) - f^*(x))]\mathbb{E}_D[\epsilon] \\
&= \mathbb{E}_D[(f(x) - f^*(x))^2] + \sigma_\epsilon^2 \\
&= \mathbb{E}_D[(f(x) + \mathbb{E}_D[f^*(x)] - \mathbb{E}_D[f^*(x)] - f^*(x))^2] + \sigma_\epsilon^2 \\
&= \mathbb{E}_D[(f(x) - \mathbb{E}_D[f^*(x)])^2] + \mathbb{E}_D[(f^*(x) - \mathbb{E}_D[f^*(x)])^2] - 2\mathbb{E}_D[(f(x) \\
&- \mathbb{E}_D[f^*(x)])(f^*(x) - \mathbb{E}_D[f^*(x)])] + \sigma_\epsilon^2 \\
&= bias[f^*(x)]^2 + var[f^*(x)] - 2(f(x) - \mathbb{E}_D[f^*(x)])(\mathbb{E}_D[f^*(x)] - \mathbb{E}_D[f^*(x)]) + \sigma_\epsilon^2 \\
&= bias[f^*(x)]^2 + var[f^*(x)] + \sigma_\epsilon^2
\end{aligned}$$

Therefore, averaging over all test samples, we obtain the MSE as:

$$MSE = \mathbb{E}_x[\mathbb{E}_{f^*}(y - f^*(x))^2] = \mathbb{E}_x[bias[f^*(x)]^2] + \mathbb{E}_x[var[f^*(x)]] + \sigma_\epsilon^2$$

As it can be deduced, big (or at least statistically representative) training sets D reduce both the bias and the variance. Unfortunately, as we will see in subsection 2.2, our available data set is small and thus we will be very susceptible to a generalization error if the NN model is not wisely chosen Neal et al. (2018).

Finally, the error σ_ϵ cannot be reduced by any means and it will be a lower bound. On our specific ornithopter trajectory problem this error will be the inherited error from the OSPA heuristic algorithm trajectories given as input. In section 3.1 we will try to partially overcome this error by adding the true target value at the end of each OSPA trajectory.

Figure 1.4 shows the trade off between NN model bias and variance Papachristoudis (2019).

1.3.3 Recurrent neural networks

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. These connections allow previous outputs to be used as inputs while having hidden states. Therefore, this type of neuronal network exhibits temporal dynamic behavior. Mandic & Chambers (2001) This property will allow us to capture the underlying flight dynamics contained in the OSPA optimal trajectories.

Figure 1.5 shows the schema of a single recurrent neuron as in Amidi (2019).

- The neuron output at time t, y_t, is a function of the hidden state a time t, a_t through a set of coefficients W_{ya}, a bias b_y and an output activation gate g_2.

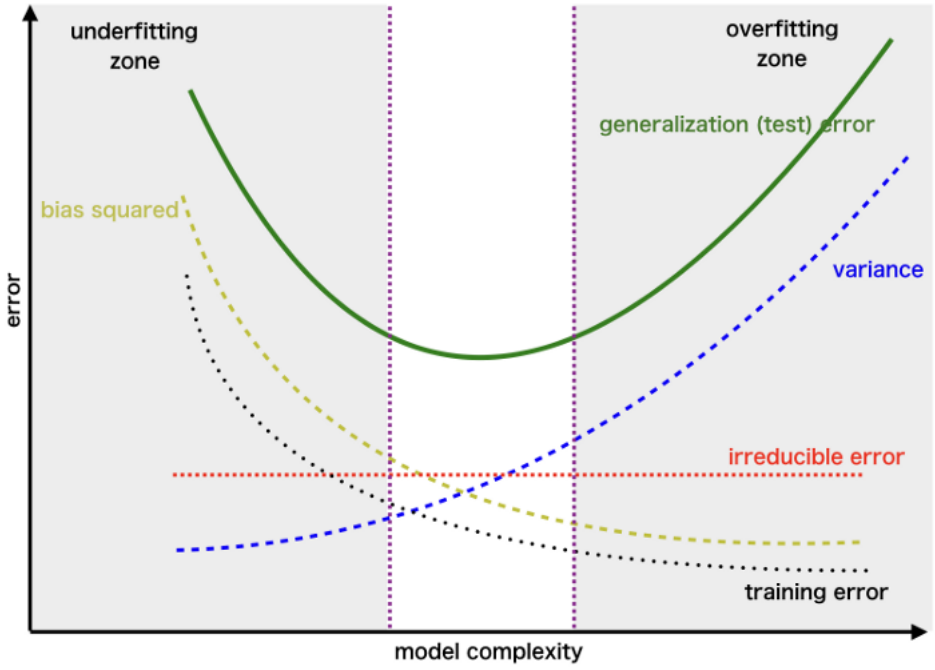

Figure 1.4: NN model bias-variance trade off Papachristoudis (2019).

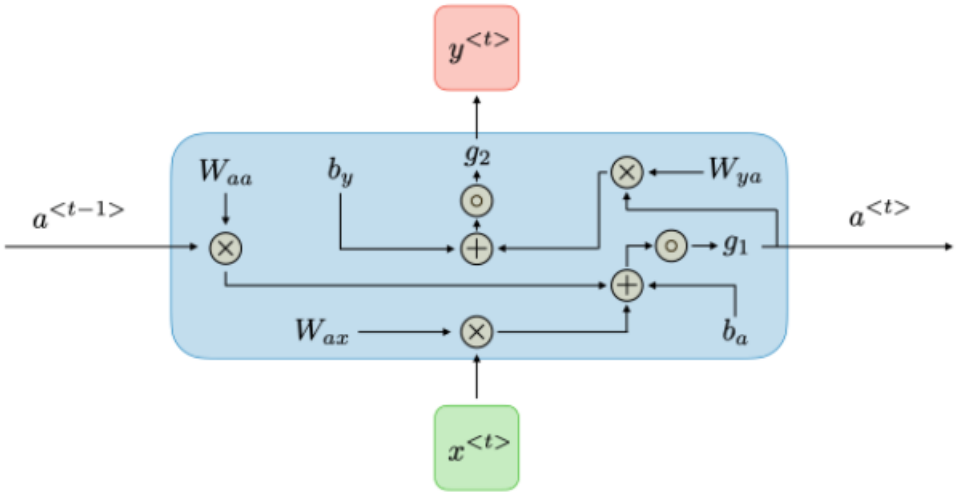

Figure 1.5: Single Recurrent Neuron schema

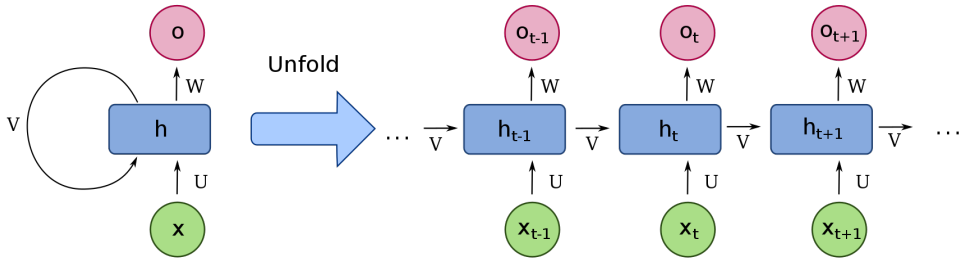

Figure 1.6: RNN architecture

- The hidden state at time t, a_t, is itself a function of the previous hidden state a_{t-1} and the input to the neuron at time t, x_t, through a set of coefficients W_{aa}, W_{ax} , a bias b_y and an output activation g_1.
- Here the interesting fact is that the hidden state a_{t-1} is used by the neuron to compute the next hidden state a_t, carrying with it information about previous time steps

The associated equations of this schema can be written as:

$$\begin{aligned} a_t &= g_1(b_a + W_{aa}a_{t-1} + W_{ax}x_t) \\ y_t &= g_2(b_y + W_{ya}a_t) \end{aligned} \tag{1.3}$$

We could retrieve back the equations of a single neuron if the input x_t was plugged directly to the last equation, or in other words, $a_t = x_t$. This would in turn remove the temporal behaviour as the output y_t would not be anymore a function of the previous states.

In order to apply all the properties seen before for feedforward networks forming Directed Acyclic Graphs in 1.3.1 and 1.3.2, we can replace each of our single recurrent neurons into a set of simple neurons as described in Figure 1.6 .

The schema depicted in Figure 1.6 shows how a RNN can be unfolded into an equivalent feed forward neural network where the same parameters θ of a simple recurrent neuron (or its equivalent set of simple neurons) are shared by each of the different unfolded neurons, one neuron layer per time step.

Theorem 1.3.4. *Goodfellow et al. (2016). The unfolding property can only be applied if the following hypothesis is met: the conditional probability distribution over the variables at t+1 given the variables at time t, is stationary.*

In Section 4.1, the training dataset will be pre-processed so we can guarantee that conditions required by Theorem 1.3.4 are met. The fact of being able of unfolding the RNN into a feed forwarded one allows us to apply all the theorems and

properties seen before. Finally, the gradient descendent back propagation algorithm can be also applied with the simplification that the parameters θ are shared between time steps (which unfold into layers in a feedforward neural network). As it can be deduced, this fact greatly reduces the number of parameters and thus calculations compared with a feedforward neural network with the equivalent M layers, one per time step. Moreover, the choice of learning a single, shared model allows generalization to sequence lengths that did not appear in the training set, and allows the model to be estimated with far fewer training examples than would be required without parameter sharing Goodfellow et al. (2016) which is specially relevant for us due to the limited amount of available data.

Chapter 2

The ornithopter trajectory optimization problem

2.1 Problem statement

The problem we want to solve is to compute an optimal trajectory of an ornithopter connecting two given positions A and B while minimizing the energy consumption. This problem is defined in detail in Rodríguez et al. (2020), where the trajectories are calculated using the OSPA heuristic method. This method has the drawback of being too computationally heavy to be used in flight. More precisely, the efficiency of such heuristic algorithm is determined by the size of the search space and the complexity of the constraints. In the case of our ornithopter we have complex constraints as the ornithopter is subject to a nonlinear differential equation system as described on Rodríguez et al. (2020) and a wide search space which is the set of possible flight maneuvers. These factors lead to mid-range optimal trajectories computational times in the order of minutes, which should be reduced to fractions of a second.

As introduced in section 1.1, the proposed approach in this work consists on using a recurrent neural network (RNN) to learn the underlaying flight dynamics that gives the optimal trajectory between two point A and B. The network is therefore trained by feeding a number of optimal trajectories computed using the planner OSPA (Ornithopter Segmentation Path Planning Approach). Once trained, the RNN is expected to estimate an new optimal trajectory given two unseen points A and B.

In this subsection we are going to explain some basic definitions regarding the ornithopter problem statement as expressed in Rodríguez et al. (2020), that will be used during the resolution via RNNs:

Definition 2.1.1. *(Flight state) A flight state* $s = (x, z, u, w, \theta, q)$ *describes an ornithopter configuration in a given instant of time, where* x *and* z *are the positional values in the plane XZ of the Earth reference frame,* u *and* w *are velocity*

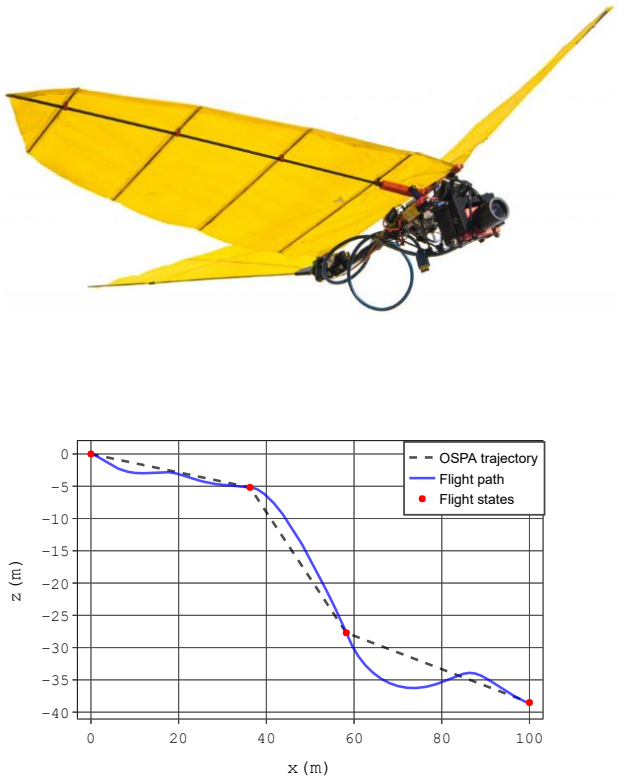

Figure 2.1: Top view, the ornithopter prototype used in Rodríguez et al. (2020). Bottom view, trajectory computed by OSPA, with three consecutive maneuvers for 100 meters before landing. The trajectory connects the flight states by integrating the dynamic model. The first state is reached with a flapping maneuver.

components in the body reference frame, θ is the pitch angle and q is the pitch angular velocity.

Definition 2.1.2. *(Flight maneuver) A flight maneuver $a = (\delta, f)$ is a control action performed by the ornithopter during its flight at a given flight state. We consider two degrees of freedom to define flight maneuvers: tail deflection, determined by the deflection angle δ (up and down); and wing flapping, determined by the flapping frequency f.*

Definition 2.1.3. *(Cost) The cost associated to trajectory is a measure of the energy consumption needed to perform the sum of the flight maneuvers needed for the trajectory.*

In Rodríguez et al. (2020), the trajectories have been discretized in equal duration time steps, where an action only takes place between time steps as it can be seen in Figure 2.1.

Given the aforementioned problem statement, a trajectory can be defined by:

- A sequence of flight states $S = (s_0, s_1, s_2, s_3...s_n)$ from point A to the vicinity of point B.
- A sequence of flight maneuvers $A = (a_0, a_1, a_2, a_3...a_{n-1})$ that drives the ornithopter following the aforementioned trajectory.

Given a trajectory, it's optimality can be measured by:

- Cost: $C = \sum_{i=0}^{n-1} c_i$, the sum of the energy consumption of each of the actions.
- Precission: The distance between the target point B and the final flight state s_n.

Note that the distance between the final flight state and the target point B has not been used as an optimization criteria in Rodríguez et al. (2020), but rather as a exclusion criteria of the solutions whose final point is deemed too far from the target. Due to the fact that the RNN only computes one trajectory and that there is no certainty that the final state will be within the desired range, we will use the precision as a measure of the correctness of the solution.

Finally, two important characteristics of the trajectory set computed by the OSPA heuristic method and used for the training of the NN are that:

- The starting point A is always the same.
- The time step between successive states is always the same.

2.2 Data set

As introduced in section 1.1, the OSPA algorithm is used as a data source to train the RNN with optimal trajectories. More precisely, the OSPA trajectories are considered to represent the true optimal trajectory distribution $p_{data}(x)$ which our neural network will try to approximate at best $p_{model}(x)$, where x corresponds to the flight states.

The data set is composed by 236 different trajectories between two random initial and target states. These low energy trajectories are sampled within the intervals in Table 2.1 and the precision required at its ending point is bounded to 3 m. Of each trajectory, the state-control waypoints returned by OSPA are stored as a set of pairs (s, m) where s is a flight state and m is the corresponding control maneuver. Finally, the time step is set constant to 31.53 seconds.

Each of the trajectories is then composed by:

- The initial $s_{initial}$. and target s_{target}. states.

- The sequence of flight states $S = (s_0, s_1, s_2, s_3...s_n)$.
- The sequence of flight maneuvers $A = (a_0, a_1, a_2, a_3...a_{n-1})$.
- The cost of the trajectory.
- The computation time for the OSPA method.

Out of the 236 trajectories, the 80% of them are randomly selected for for training whereas the remaining 20% is used as validation set to compute the generalization error and RNN performance metrics.

Variable	Initial State	Target State
$X\ (m)$	0	$[25, 100]$
$Z\ (m)$	0	$[-40, 0]$
$\Theta\ (^\circ)$	$[-30, 30]$	30
$U_b\ (m/s)$	$[1, 4]$	0

Table 2.1: Ranges of initial and target state variables used in the experiments.

Chapter 3

The recurrent neural network

3.1 Input

The data set introduced in section 2.2 is pre-processed before ingestion by the RNN. More precisely, the inputs to train our RNN are the sequences of flight states $S = (s_0, s_1, s_2, s_3...s_n)$, which will be transformed to the distance sequences $X = (x_0, x_1, x_2, x_3...x_n)$ by applying the following transformations:

- First, in order to include information about the target in each time step, the flight state sequences have been changed to the distances to the target:

$$x_i = s_{target} - s_i$$

- Second, due to the fact that the different flight components in the flight state (x, z, u, w, θ, q) have a great disparity in values, they have been normalized using their standard deviations as follows:

$$x_i = \frac{z_i}{\sigma}$$

Regarding the actions $a = (\delta, f)$, they will be used as they are since both orders of magnitude are similar.

This input data follows a probability distribution which will be represented throughout this work as $p_{data}(x)$.

3.2 Recurrent neural network layer

Feedforward neural networks learn from the training samples as if each of them were independent and identically distributed random variables. However, we know that this is not the case when dealing with our trajectory problem, or more generally, with sequences. In other words, the current state x_t is not independent

from the previous state x_{t-1}. Thus, it seems logical to use the information in the previous states to better estimate the next action. Note that for the specific case of our trajectories, the next action is only function of the current state and the target state. Thus, a simple neuronal network could do this job as it is not mandatory to consider the previous states to compute the next action. In more general sequence cases, like for example in sentence translations, it is necessary to consider all previous words and even future ones to fully put in context the current word. As discussed in the introduction, the recurrent neural networks exhibits temporal dynamic behavior. Since the ornithopter trajectory can be described by a dynamical system, a RNN seems then the perfect artificial neural network class choice.

Furthermore, in our case we will show that we comply within Theorem 1.3.4. This will in turn allow us to apply all the good properties of a feedforward network. In deed, within any given trajectory, the conditional probability distribution over the variables at $t+1$ given the variables at t is stationary. This is true since:

- For every tuple of starting and target points, it is expected to obtain the same optimal trajectory. Therefore, considering the starting point as an intermediate point at time t of a longer trajectory with same target point, the following intermediary points are expected to be same.

- When dealing with discrete trajectory state values, due to the fact that the time step is constant, the same intermediate state values are expected, making their probability distribution stationary with time.

Finally, we can also state that our problem complies the universal approximation theorem. Following Theorem 1.3.1, this is true since:

- Any continuous function on a closed and bounded subset of R^N is Borel measurable.

- Our underlaying function f must be continuous as two optimal trajectories can be arbitrarily close as far as the starting and target points are respectively as close as needed.

- Our subset is bounded in $\mathbb{R}^6$ as no infinite trajectory can be optimal (nor feasible).

Note that the universal approximation theorem ensures that a NN can approximate the optimal trajectory function to any degree of accuracy, but nothing is said about how the NN should be in terms of configuration or needed number of parameters (which can be lead in practice to unfeasible NN).

Finally, the gradient descendent algorithm can be applied as well to our RNN as introduced in section 1.3.1 for feedforward networks.)

3.3 Output

The NN approximation task can be divided into regression or classification problems

Classification model

As we will see in the implementation and results, the ornithopter possible action data set is actually a finite set of 35 different action tuples. This is due to the methodology of the heuristic method used to compute the optimal path, which requires a finite set of possible action outcomes in order to obtain a search tree. In the task of predicting the next action, the RNN can either output the tuple $a = (\delta, f)$, leading to a regression model, or it can predict which is the most probable action to apply from the aforementioned finite set of actions, leading to a classification model. The outcome thus will be the probability to apply each of the possible actions and one with the highest predicted probability will be selected. Therefore, in the classification problem, the NN aims to output the most probable action to take amongst a given set of finite options, given the corresponding preceding flight states.

In the classification model each action a_k is treated as a different category, leading to 35 different categories:

$$a_k = (\delta_k, f_k),\ k = 1, ..., 35$$

In this case, we want the RNN to output the probability that each category has to be selected. This gives the following relationship between the real probability distribution y_k and our prediction $\hat{y_k}$ to be:

$$\hat{y_k} = p(y_k|x, \theta)$$

where $\hat{y_k}$ is the vector with the predicted probabilities for each category and y_k is the "one hot" representation of each category. A "one-hot" representation of the category k consist on a vector with size the number categories where all elements are zero except the k^{th} element with value 1. For instance, a_3 is represented by $[0, 0, 1, 0, ..., 0]$. This representation indicates that the probability for category 3 is 1 whereas is zero for the rest.

Proposition 3.3.1. *It is equivalent to use the Log-likelihood or the Categorical Cross Entropy as loss function for our NN classification problem.*

Proof. As for the regression case, when computing the maximum log likelihood we end up with:

$$L = \sum_{i=1}^{m} \log p(y|x_i; \theta) = \sum_{i=1}^{m} \log \hat{y}_i = \sum_{i=1}^{m} \sum_{k=1}^{N} p(y_{ik}) \log \hat{y}_{ik}$$

The later equality is true since each probability $p(y_{ik})$ is zero except for the observed category k that is equal to 1.

Note that this last term corresponds to the negative cross-entropy of the distribution $\hat{y}_i$ relative to a distribution $p(y_i)$ which, for discrete probability distributions, can be written as:

$$H(p(y_i), \hat{y}_i) = -\sum_{k=1}^{N} p(y_{ik}) \log \hat{y}_{ik}$$

This gives us the following result

$$\theta_{ML} = \arg\max_{\theta} L = \arg\max_{\theta} -\sum_{i=1}^{m} H(p(y_i), \hat{y}_i)$$

□

The cross entropy loss function can be interpreted as the expected message-length per datum when a wrong distribution p_{model} is assumed while the data actually follows a distribution p_{data}. Or in other words, when our neural network $f^*(\cdot\,; \theta^*)$ is used instead of the real source of data, which is our OSPA planner. Therefore, the higher the cross-entropy is, the further our NN is from generating the real data probability distribution. Here the locally optimal parameter θ^* is found by the loss optimization algorithm so that the cross-entropy values is locally minimized for the parametric family $f^*(\cdot\,; \theta)$.

We have just seen a way to interpret the Likelihood loss function. However, we may be interested in an actual measure of dissimilarity between our model and the underlying true data distribution.

A measure of the dissimilarity between any two distribution is the KL divergence, and it is defined for discrete distributions as:

$$D_{\mathrm{KL}}(p_{data} \parallel p_{model}) = \sum_{x \in \mathcal{X}} p_{data}(x) \log \left(\frac{p_{data}(x)}{p_{model}(x; \theta)} \right)$$

Although the KL divergence is not a true metric, since for instance $D_{\mathrm{KL}}(p \parallel q) \neq D_{\mathrm{KL}}(q \parallel p)$, it is intuitively a pertinent loss function since it represents the amount of information lost when p_{model} is used to approximate p_{data}.

Proposition 3.3.2. *It is equivalent to use the Log-likelihood or the KL divergence as loss functions to compute the optimal parameters for our NN classification problem.*

Proof. This divergence can be rewritten as:

$$D_{\mathrm{KL}}(p_{data} \parallel p_{model}) = D_{\mathrm{KL}}(p(y) \parallel \hat{y}) = \sum_{i=1}^{m} p(y(x_i)) \log \left(\frac{p(y(x_i))}{\hat{y}(x_i;\theta)} \right)$$
$$= \sum_{i=1}^{m} p(y(x_i)) \log p(y(x_i)) - \sum_{i=1}^{m} p(y(x_i)) \log \hat{y}(x_i;\theta)$$

The first term of the right corresponds to the negative of the entropy H of $p(y)$ and does not depend on the parameters θ. The second term on the right correspond to the cross-entropy of $\hat{y}(x;\theta)$ relative to $p(y)$.

$$D_{\mathrm{KL}}(p(y) \parallel \hat{y}) = -H_y + H_{y\hat{y}}(x;\theta)$$

Here we can see that the amount of information lost when $\hat{y}$ is used to approximate y actually corresponds to the difference of Entropy.

Therefore, maximizing the log likelihood is equivalent to maximizing the similarity which in turn corresponds to minimize the cross-entropy.

$$\theta_{mDKL} = \theta_{mCH} = \theta_{ML}$$

□

When using the KL divergence as a loss function, in addition to find the locally optimal parameter θ^*, we can actually asses the quality of our model. Knowing that a zero KL divergence value corresponds to a perfect match between model and the true distribution, we can consider the values in Table 3.1 as a guideline Brownlee (2019):

KL Divergence Value (nats)	Model valuation
0.00	`perfect`
<0.02	`good`
<0.05	`on track`
<0.2	`fine`
<1.0	`poor`
>2.0	`broken`

Table 3.1: Model valuation by KL divergence loss values.

Regression model

In the regression problem, the RNN aims to output the best possible approximation to the values of the true states in $\mathbb{R}^6$ given by an OSPA trajectory. Therefore, after training, the RNN will output a prediction $\hat{y}_i$ given the input x_i. The relationship between the real value y_i and our prediction $\hat{y}_i$ can be written as:

$$y_i = \hat{y}_i + e_i$$

where e_i is the error due to either non modeled aspects. Assuming that all these aspects are independent, we can apply the central limit theorem to rewrite this error as a Gaussian distribution with the form:

$$p(e_i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{e_i}{\sigma}\right)^2} = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i-\hat{y}_i}{\sigma}\right)^2}$$

Here, it is assumed that the error has standard deviation σ and null mean. This implies that $p(y_i|\hat{y}_i)$ follows a Normal distribution $N(\hat{y}_i, \sigma^2 I)$. Due to the fact that $\hat{y}_i$ depends on x_i and θ it can be rewritten as:

$$p(y_i|\hat{y}_i;\sigma) = p(y_i|x_i;\theta;\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i-\hat{y}_i}{\sigma}\right)^2}$$

Proposition 3.3.3. *Given the above mentioned hypothesis, it is equivalent to use the Log Likelihood or the Mean Squared Error as loss functions for our NN.*

Proof. If we compute the Maximum log likelihood, we end up with:

$$L = \sum_{i=1}^{m} \log p(y_i|x_i;\theta;\sigma) = \sum_{i=1}^{m} \log \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i-\hat{y}_i}{\sigma}\right)^2} = m \log \frac{1}{\sigma\sqrt{2\pi}} - \sum_{i=1}^{m} \frac{1}{2} \left(\frac{y_i - \hat{y}_i}{\sigma}\right)^2$$

$$\theta_{ML} = \arg\max_{\theta} L = \arg\max_{\theta} m \log \frac{1}{\sigma\sqrt{2\pi}} - \sum_{i=1}^{m} \frac{1}{2} \left(\frac{y_i - \hat{y}_i}{\sigma}\right)^2 = \arg\max_{\theta} \sum_{i=1}^{m} \frac{1}{2} \left(y_i - \hat{y}_i\right)^2,$$

which turns to be equivalent to minimizing the Mean Squared Error.

□

In our Neural Network, we will use a Linear Activation to output the prediction as we expect it to be continuous in $\mathbb{R}^6$.

The MSE is already a good measure of how well our model fits the true distribution and it gives a measure of the estimation error from our model.

Nevertheless, we can also have the value of the measure of the dissimilarity between the model and true distributions via the KL divergence:

$$D_{\mathrm{KL}}(p_{data} \parallel p_{model}) = \sum_{x \in \mathcal{X}} p_{data}(x) \log \left(\frac{p_{data}(x)}{p_{model}(x;\theta)}\right)$$

Proposition 3.3.4. *It is equivalent to use the Log-likelihood or the KL divergence as loss functions to compute the optimal parameters for our NN regression problem.*

Proof. This divergence can be rewritten as:

$$D_{\mathrm{KL}}(p_{data} \parallel p_{model}) = \sum_{x \in \mathcal{X}} p_{data}(x) \log p_{data}(x) - \sum_{x \in \mathcal{X}} p_{data}(x) \log p_{model}(x; \theta)$$

The first term of the right corresponds to the negative of the entropy H of $p_{data}(x)$ and does not depend on the parameters θ. The second term on the right correspond to the negative log likelihood of $p_{data}(x)$ and $p_{model}(x; \theta)$.

$$D_{\mathrm{KL}}(p_{data} \parallel p_{model}) = -H_{p_{data}} - L(x; \theta)$$

Therefore, maximizing the similarity corresponds to maximizing the log likelihood.

$$\theta_{mDKL} = \theta_{ML} = \arg\max_{\theta} \mathop{\mathbb{E}}_{X} \log p_{model}(x; \theta)$$

□

Chapter 4

RNN Implementation

This chapter contains an overview on how our RNN problem statement has been actually implemented, layer by layer. Overall, all the scripts are programmed using Jupyter notebooks and the neural networks are computed using the TensorFlow library in Python. Jupyter Notebook allows us to explain how the code works on the same script file and TensorFlow is a widely used Machine Learning Library developed by Google. All RNN implementation code is published in https://github.com/paskymail/Neuronal_Networks/tree/master/Simple_RNN.

4.1 Input pre-processing

Prior to the ingestion of the training data, the training set is pre-processed using the script "Process_Training_Data_2.ipynb"

This pre-process consist on:

- Transforming Flight state sequences into its normalized trajectory distances sequence as explained in Section 3.1.
- Creating distance sequences of equal length by truncation when the trajectory is too long or by padding with zero distance vectors when the trajectory is too short.

The reasons for setting equal length trajectories are:

- The RNN is trained by input batches and each batch should be of equal length due to computation constraints.
- Only flight states belonging to the same trajectory be feed to the RNN in order to comply with theorem 1.3.4.

As explained in Figure 4.1, the padding consist on adding zero distance vectors when the original trajectory sequence is not long enough. For instance, if the

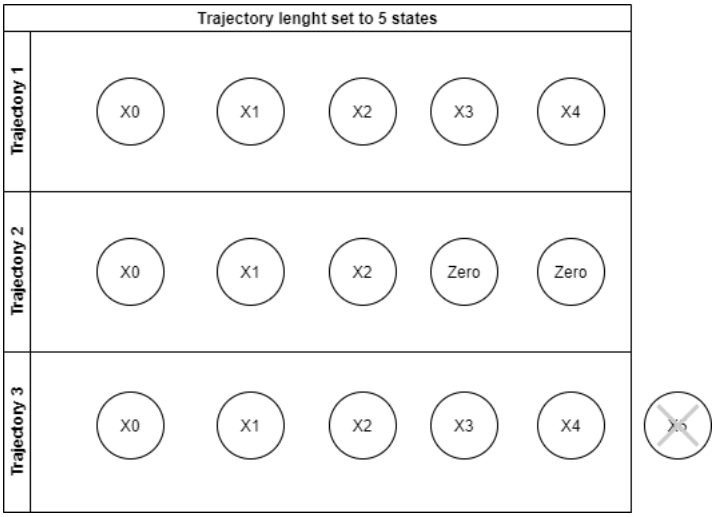

Figure 4.1: Padding and trucation operation.

trajectory length is set to 5 and our trajectory only has 3 flight states, then two zero vectors are added for the fourth and firth position of the new trajectory.

The reason for padding with zero vectors on the right is that as the trajectory unfolds, the distance to the target is expected to get closer and closer to zero. Thus, the padding will only keep the distance constant in the zero value after the trajectory is finished.

This zero distance value at the end of the trajectory will be used during the training phase to indicate the RNN that the trajectory is over and we will expect in the prediction phase from the RNN to indicate with a Zero value the last distance state of the predicted trajectory and end of the sequence.

4.2 Recurrent Neural layer

We have chosen Long Short Term Memory neurons in order to build the Recurrent Neurons layer. This is just an specific class or recurrent Neural Network choice has been made because:

- It is a common used RNN with good properties, as for instance it deals with the vanishing gradient problem encountered by traditional RNN Sherstinsky (2020).
- The TensorFlow API allow us to customize its behaviour and configuration, for instance the activation functions (g_1 and g_2) and the behaviour of its hidden state a_t

It is not the aim of this work to give much detail regarding recurrent neuron configurations and therefore we refer the reader to Amidi (2019) for a brief summary

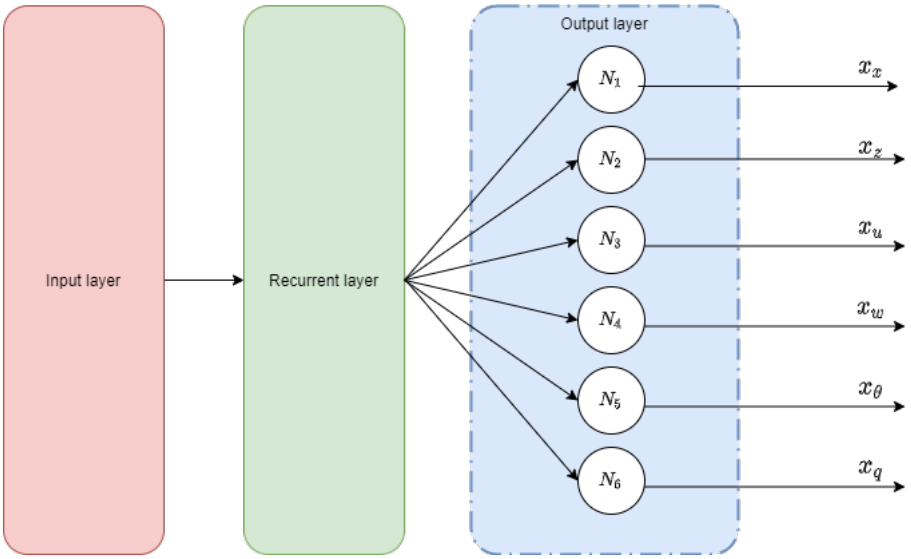

Figure 4.2: Regression model output layer.

or Sherstinsky (2020) of an in depth review.

4.3 Output layer

The objective of the output layer is to render the results of the RNN in the correct format so that it can be compared with the existing data. Therefore, it will depend whether we are dealing with a regression model or a classification model.

4.3.1 Regression model

As illustrated in Figure 4.2, the output layer transforms the output of the Recurrent Layer into the correct number of components to compare. In the case of the flight distances, we will transform an arbitrary number of outputs from the RN into the six components $x = (x_x, x_z, x_u, x_w, x_\theta, x_q)$, one per each flight state component and in the case of actions $a = (\delta, f)$ we will have the corresponding two outputs. During the training phase, we will use the nominal values of δ, f for each expected class and compare them against the predicted values of these two components. During the prediction phase, we will search for the closest nominal values δ, f in the class space and output the action class with the minimum Euclidean distance to the predicted point.

Finally, as discussed in Section 3.3, due to the fact that the expected outputs are continuous valued in $\mathbb{R}$, the activation function used in this output layer will be linear.

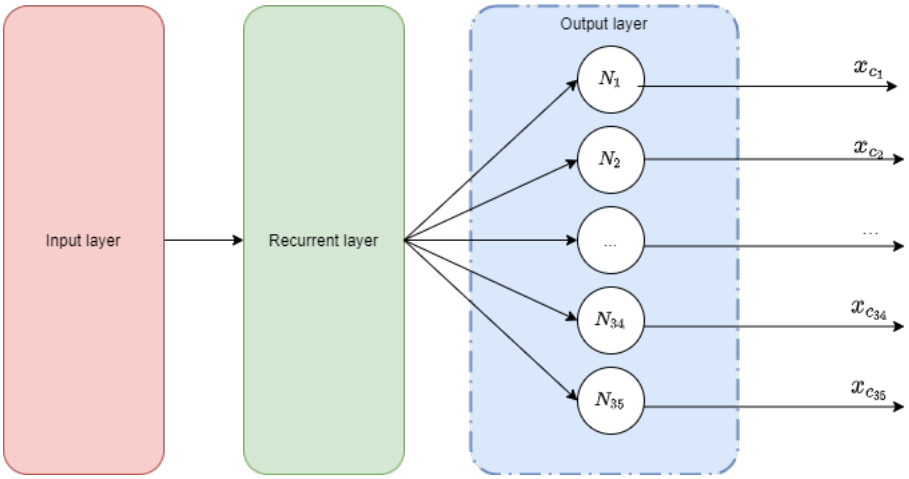

Figure 4.3: Classification model output layer.

4.3.2 Classification model

As can be seen in Figure 4.3, in this case the output layer transforms the output of the Recurrent Layer into as many outputs as classes are considered, where each output represents the probability of predicting that specific class. In our case, we will have 35 outputs, one per action class.

As discussed on Section 3.3, here the goal of the output layer is to output the probabilities of obtaining each of the categories.

Proposition 4.3.1. *An output layer with a softmax function is capable of estimating the probabilities of obtaining each of the categories.*

Proof. We use the Bayes Theorem to compute the probability of obtaining a given class y^k as follows:

$$p(y^k = 1|x) = \frac{p(x|y^k = 1)p(y^k = 1)}{p(x)} = \frac{p(x|y^k = 1)p(y^k = 1)}{\sum_{j=1}^{N} p(x|y^j = 1)p(y^j = 1)}$$

In the expression above, $p(y^j = 1)$ can be estimated from the data. If we assume that $p(x|y^k = 1)$ follows a Gaussian, which it is a fair approximation as per Figure 4.4, this probability could be rewritten as:

$$p(x|y^k = 1) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2},$$

which implies:

$$p(y^k = 1|x) = \frac{\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu_k}{\sigma_k}\right)^2} p(y^k = 1)}{\frac{1}{\sigma\sqrt{2\pi}} \sum_{j=1}^{N} e^{-\frac{1}{2}\left(\frac{x-\mu_j}{\sigma_j}\right)^2} p(y^j = 1)} = \frac{e^{-z_k(x;\theta)}}{\sum_{j=1}^{N} e^{-z_j(x;\theta)}} \tag{4.1}$$

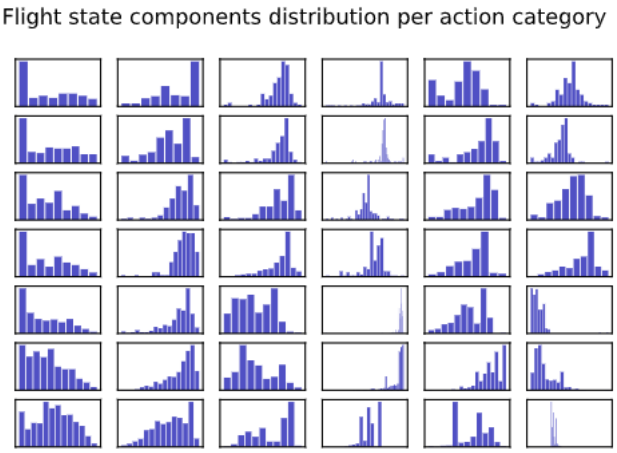

Figure 4.4: Flight state components (columns) value distribution per action category (rows).

□

As can it be seen, equation 4.1 has the form of a softmax function, which will be used as activation function for the output layer in the case of the classification problem.

4.4 Training and evaluation

As stated before, the training of the Neural Networks consists on reducing a loss function $J(x; y; \theta)$ which depends on the input values, expected output values and the network parameters. As seen before, the chosen loss function is the log-likelihood and the parameters θ are optimized to maximize it. This maximization is equivalent to minimize the MSE in the case of the regression model or the Cross-Entropy in the case of the classification model.

The fact that the loss function $J(x; y; \theta)$ depends on the input and expected output values means that the resulting optimal parameters θ may vary depending on the data. This problem is know as *the generalization problem.*

The generalization problems occurs when the parameters θ are optimal for the training set $(x_{training}; y_{training})$ but it has low performance for another unseen validation set $(x_{validation}; y_{validation})$. In other words, the networks learns to fit closely the specific training set but in unable to fit other general unseen data set. Due to this,the available data will be divided in two different and subsets:

- Training data $(x_{training}; y_{training})$ which accounts for 80 percent of the available data and is used for computing the loss function and thus for training.

- Validation data ($x_{training}; y_{training}$) which accounts for the remaining 20 percent of the available data and is used to evaluate the performance of the network when faced with previously unseen data.

The actual performance of our NN will be determined by the performance of the validation loss and the metrics values.

Finally note that as explained in the introduction section (1.3.1), the more complex our network is, the more prone to generalization error is, as it will be able to "memorize" to match the values of the training set. The simpler the network is, the higher the training error will be as it won't be able to capture all the model details. Therefore the RNN model should be carefully designed to avoid these issues.

Chapter 5

Results

In this chapter we are going to try four different architectures of recurrent neural networks and apply them to the ornithopter optimal trajectory estimation. Each of the architectures is expected to have some characteristics that will be explained. Finally, the obtained results will be analyzed and justified based on the aforementioned special characteristics.

5.1 Action prediction RNN

5.1.1 Network architecture and path computation

As explained in the problem statement chapter, our objective is to compute the optimal trajectory of an ornithopter from point A to point B with minimum consumption. Therefore, our algorithm should then be initialized with the initial and goal states and it should compute the rest of the trajectory.

As mentioned in the ornithopter problem statement, a trajectory can be also defined as sequence of actions $A = (a_0, a_1, a_2, a_3...a_{n-1})$. In practical applications, we may be interested in knowing the action to apply at each time step instead of the sequence of states. Our first approach consists thus on developing the trajectory by predicting the next action given the previous states. The ornithopter dynamic equations from Rodríguez et al. (2020) are used to compute the resulting next state derived from applying the predicted action. This is the most straightforward approach and could be accomplished by any feedforward NN since there is no dynamical behaviours learnt, only a classifying network is needed.

This computation can be visualized as adding a known computation layer that transforms the predicted action to corresponding state as in Figure 5.1. This upper layer is not learned by the RNN, it is fixed forehand. This non neural layers are called Lambda layers in Keras and are just a transformation layer that can be stacked within true neural layers.

Therefore the configuration is as follows:

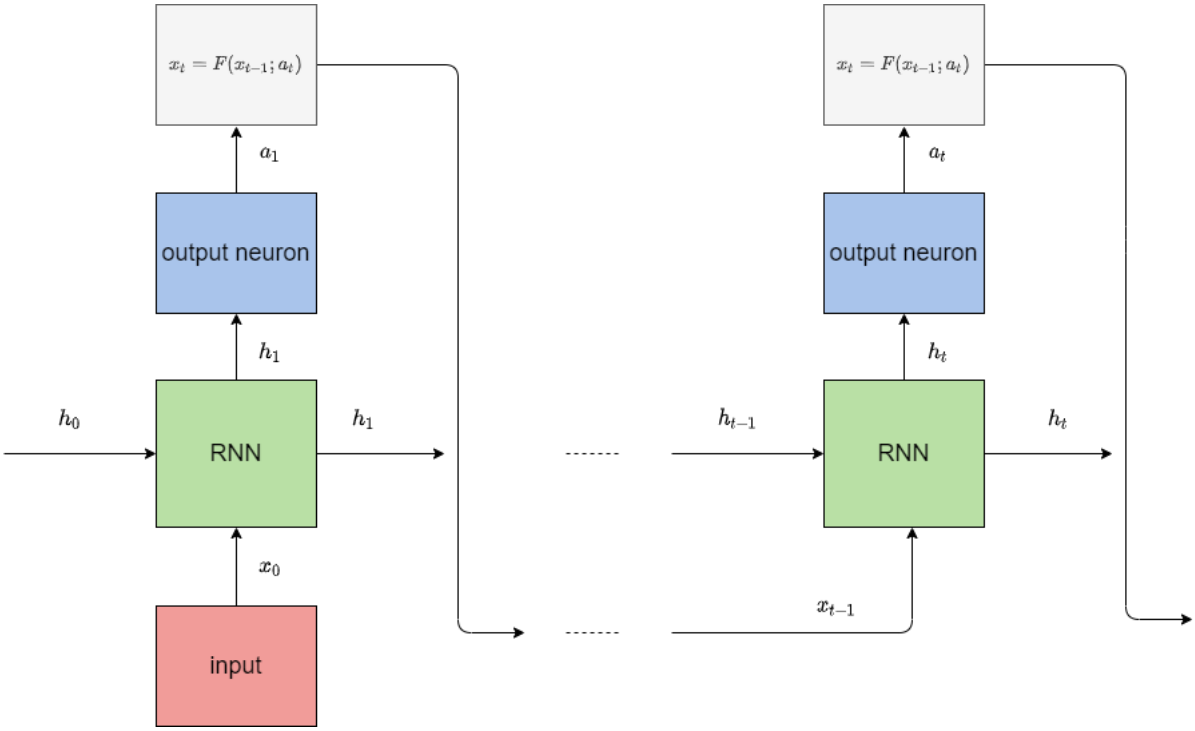

Figure 5.1: Next action prediction from previous states.

- In the input layer we will input the last predicted state.
- The output layer is expected to output the probabilities for the next action to apply. As explained during the problem statement, a softmax activation function is applied.
- The fixed function $x_t = F(x_{t-1}; a_t)$ will compute the next state based on the predicted action a_t.
- The recurrent layer is expected to learn the function $a_{t+1} = f(h_t; x_t)$, where x_t is the predicted distance at time t and $h_t = f_r(h_{t-1}; x_{t-1})$ is the hidden layer state which contains information from previous steps. These dependencies can be inferred from the recurrent neuron equations (1.3).

Taking the aforementioned points in mind, the algorithm 1 is used to construct the trajectory step by step.

In Table 5.1 a summary of the Keras Neural Network can be found. It can be seen how the output layer has 35 neurons, one per category so its probabilities can be compared with the real category "one-hot" representation. The neural network contains a total of 1212 trainable parameters.

It is important to note that this approach does not take advance of the recurrent nature of the neural network to build a complete sequence, but rather the network

Algorithm 1: trajectory loop

Result: $X = (x_0, x_1, x_2, x_3 ... x_n)$, Cost

x_0;

while $x \neq Zero$ **do**

  $a_i = RNN(\vec{x})$;

  $Cost = C(a_i)$;

  $x_i = F(a_i)$;

  $\vec{x}.append(x_i)$;

end

is used as an action classification. Any other simple network is suitable to do this job, but this architecture is used as a stating point to be able to evaluate further on the benefits of other architectures capable of exploiting the recurrent properties of the network.

5.1.2 Training

Now, we have to define the network cost function. Our goal here is to estimate the next action to apply, therefore our target it to get the action category estimate $\hat{a} = p(a|x;\theta)$ as close as possible to the true action category value $p(a|x)$.

We are going to apply what it has been already developed on subsection 3.3 and the we use the KL divergence loss function as the measure of the dissimilarity between these two distributions. which is defined for a discrete distributions as:

$$D_{\mathrm{KL}}(p_{data} \parallel p_{model}) = \sum_{x \in \mathcal{X}} p_{data}(x) \log \left(\frac{p_{data}(x)}{p_{model}(x;\theta)} \right)$$

The computation of such loss function is implemented in the TensorFlow API by the KL Divergence Loss function. Alternatively, the Categorical Cross-entropy Loss function could be used with the same results.

Despite the high number of parameters, no overfitting effects have been observed. In Figure 5.2 the loss (KL divergence) decrease during training is shown. It has to be noted that the validation loss decreases jointly with the training loss, which implies that the networks is learning properly and it is able to generalize to unseen validation data. Finally, the KL divergence loss value reached is close to 0.02, which in accordance with Table 3.1 corresponds to a good model fit.

5.1.3 Results analysis

Figure 5.3 plots the RNN predicted X, Z trajectory again the OSPA trajectory. We see I that the RNN trajectory is quite similar to the OSPA behavior.

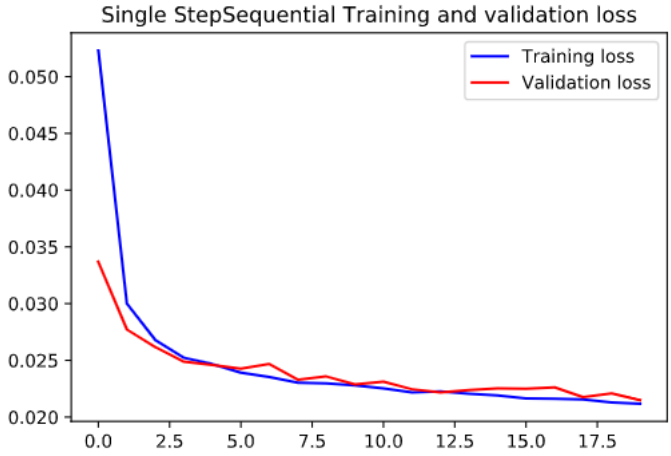

Figure 5.2: Loss (KL-divergence) decrease during training

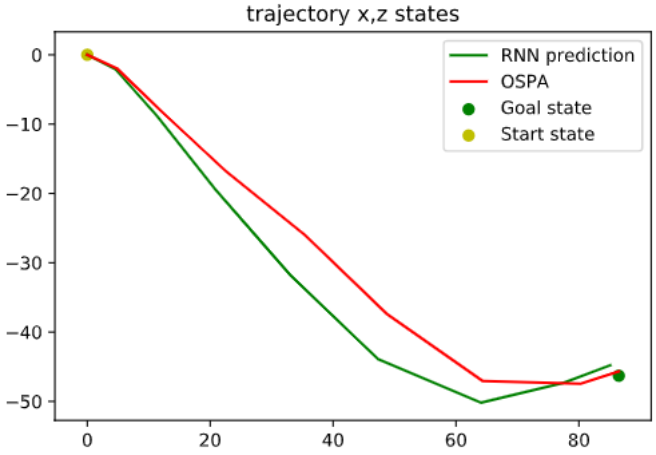

Figure 5.3: X,Z components trajectory comparison.

Layer	Output shape	parameter number
LSTM	[None, None, 11]	792
Dense	[None, None, 35]	420

Table 5.1: Action classifier network summary.

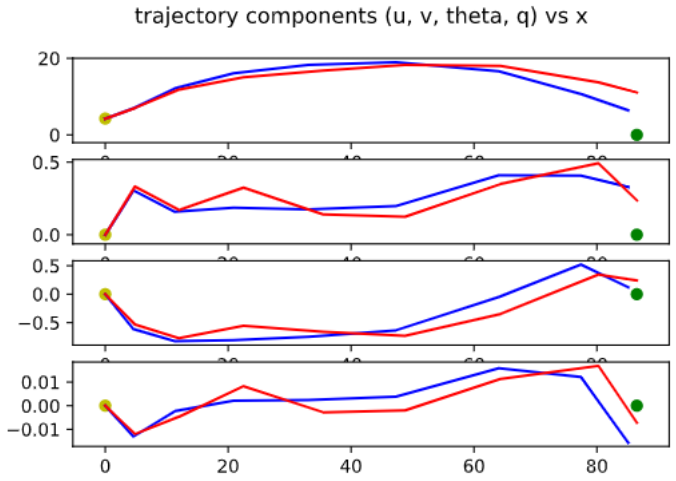

Figure 5.4: Comparison on the u,v, θ, q components.

This property is also true for the complete 6 components as shown in Figure 5.4, where the u, v, θ and q components of the flight state are plotted against the x component. This is expected since there is no special priority treatment for any component and thus the performance for each of them should be similar. In case of any component to be prioritized, a set of weights can be assigned for each one when computing the loss function in such a way errors on some components are specially penalized.

The overall results can be seen in Table 5.2, which contains the following performance metrics already introduced in section 2.1:

- Cost (W): is the total energy cost (battery) consumed by the ornithopter, and it is determined by the maneuvers (actions) performed.
- Time (s): is the total elapsed time in computing the full trajectory.
- Precision (s): is the euclidean distance in the XZ-plane from the final state to the target
- Trajectory error (m): is the mean of distance in the Z axis between the OSPA and the RNN trajectories at 10 given intermediary X-positions.

These metrics were identified as relevant for the OSPA performance itself and are not used to optimize the network parameters, they are computed only for comparison purposes.

Algorithm	Cost (W)	Time (s)	Precision (m)	Trajectory error (m)
OSPA	34.68	520	2.84	`na`
Action prediction	37.25	0.43	6.29	3.85

Table 5.2: Comparison with OSPA algorithm.

As an overall conclusion, it can be stated that the RNN is able to generate new optimal trajectories from unseen start and target points. Even if the RNN is much faster (0.43s vs 520s), precision has been worsened compared to the OSPA planer in terms of distance to the target (6.29m vs 2.84m). It has to be noted that although the overall trajectory energy consumption has also increased (37.25 vs 34.68), it has been done in less proportion, indicating that similar trajectories also have similar costs and that the energy consumption is not specially sensitive within small deviations. This fact will turn useful when the states are predicted instead of the actions and thus the cost cannot be computed.

In the next sections we will exploit the dynamic properties of the RNN to improve our results. We will maintain the same 11 neurons in the recurrent layer and reduce the neurons in the output layer, reducing the number of trainable parameters. Although a reduction in performance could be expected, we will see how the new architectures will actually improve the precision, even beyond the OSPA algorithm. In other words, even if the number of parameters will be reduced, we will change the parametric family $\{f^*(\cdot\,;\theta)\}$ for a new and better suited one, with increased capacity.

5.2 Sequence to sequence RNN

5.2.1 Network architecture and path computation

In our first attempt for a real recurrent network we are going to use a sequence to sequence configuration as the one shown in Figure 5.5. This configuration will be also the set up used for the training of the next RNN architectures.

Therefore the configuration is as follows:

- In the input layer we will input the true flight state distances trajectory sequence
- The output layer is expected to output the same trajectory sequence shifted by one time-step in the future

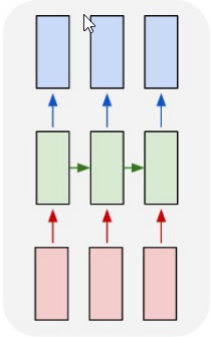

Figure 5.5: Sequence to sequence architecture.

- The hidden layer (corresponding to the recurrent layer) is the expected to learn the function $x_{t+1} = f(h_t; x_t)$, where x_t is the true distance at time t and $h_t = f_r(h_{t-1}; x_{t-1})$ is the hidden layer state which contains information from previous steps. This dependencies can be inferred from the recurrent neuron equations 1.3.

Note that if we do not perform the sequence shift, the NN would learn an identity function.

In Table 5.3 a summary of the Keras Neural Network can be found. It can be seen how the output layer has 6 neurons, one per flight state so their estimated values can be compared with the true ones at each time step. The neural network contains a total of 864 trainable parameters.

Layer	Output shape	parameter number
LSTM	[None, None, 11]	792
Dense	[None, None, 6]	72

Table 5.3: Sequence to sequence network summary.

Finally, the network schema for a given time step can be seen in the Figure 5.6:

5.2.2 Training

Now, we define as usual the network cost function. Our goal here is to estimate the same input sequence shifted by one time step, therefore our target it to get $p_{model}(x; \theta)$ as close as possible as $p_{data}(x)$.

In this case, due to the fact that we are dealing with a regression model, we are going to use the equivalence between the Maximum Likelihood and the minimum Mean Squared Error seen in 3.3 to use the Mean Square Error as our Loss function.

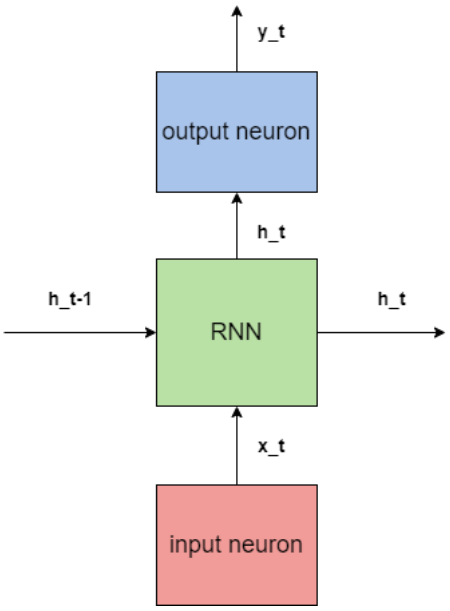

Figure 5.6: Sequence to sequence time step.

Therefore:

$$\theta_{ML} = \theta_{mMSE}$$

The computation of such loss function is implemented in the TensorFlow API by the MSE Loss function.

A smooth MSE loss decrease can be seen in 5.7 with no overfitting effects. It has to be noted that the validation loss values are very close to the training ones, which implies that the networks is able to generalize the results perfectly.

5.2.3 Results analysis

As shown in Figure 5.8, the predicted X, Z trajectory correctly matches the OSPA planner. Even when a change in direction occurs, the RNN is able to predict it. Finally, the fact that the goal state is not reached is because the trajectory is truncated to an arbitrary number of steps (10 in this case) as explained in subsection 4.1. This behavior is also true for the complete 6 components as shown in Figure 5.9. Again, the RNN is able to correctly predict a sudden change for the θ and q components.

Table 5.4 shows the overall results. There are two scores to be highlighted.

- The mean distance between the OSPA and RNN predicted trajectories is 1.59 meters, which implies that both trajectories are very close to each other and thus very similar trajectory energy costs are expected as shown in the results analysis of subsection 5.2.

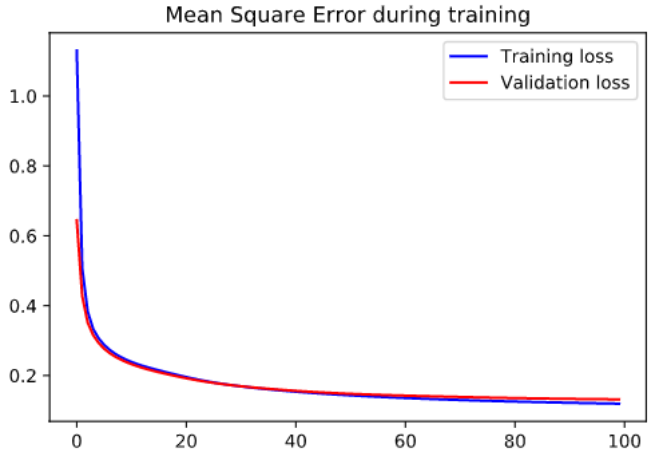

Figure 5.7: Loss (MSE) decrease during training

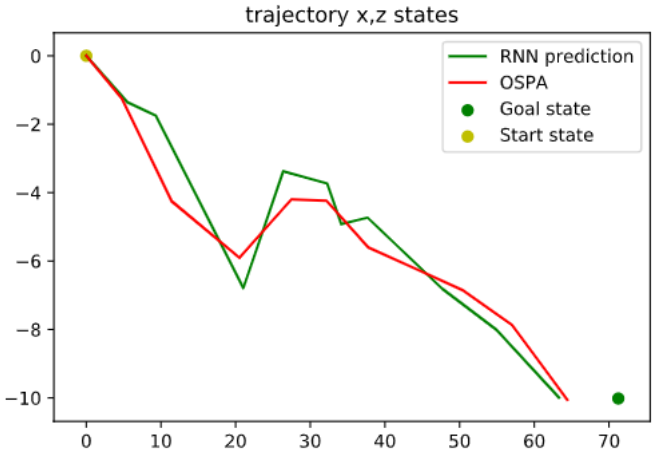

Figure 5.8: X,Z components trajectory comparison

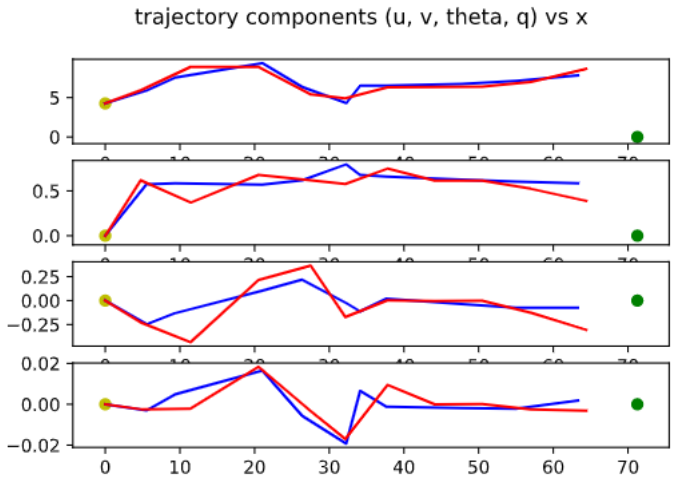

Figure 5.9: Comparison with components u,v, θ, q.

- The precision has exceeded the one reached by the OSPA planner (1.40m vs 2.84m).

The later statement may seem to be impossible at first glance, since in accordance with bias-var variance trade off explained in section 1.3.2, we are bounded by the OSPA data noise σ_ϵ^2 due to equation $MSE = \mathbb{E}_x[bias[f^*(x)]^2] + \mathbb{E}_x[var[f^*(x)]] + \sigma_\epsilon^2$. However, the fact of having introduced the real target value at the end of each sequence as explained in subsection 4.1 has allowed us to overcome such limitation and exceed the OSPA performance.

Algorithm	Cost (W)	Time (s)	Precision (m)	Trajectory error (m)
OSPA	34.68	520	2.84	`na`
RNN sequential	`na`	0.46	1.40	1.59

Table 5.4: Comparison with OSPA algorithm.

5.3 Use of RNN as decoder

5.3.1 Network architecture and path computation

In the previous configuration we have considered that the previous true distances to the target are known and that we are only interested in predicting the next distance state. However, a planner should predict the flight states of multiple steps in the future, not only the next one. This will be the Decoder case, where

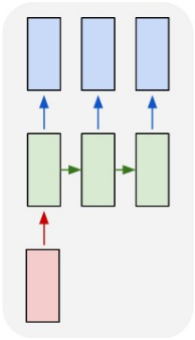

Figure 5.10: Decoder architecture.

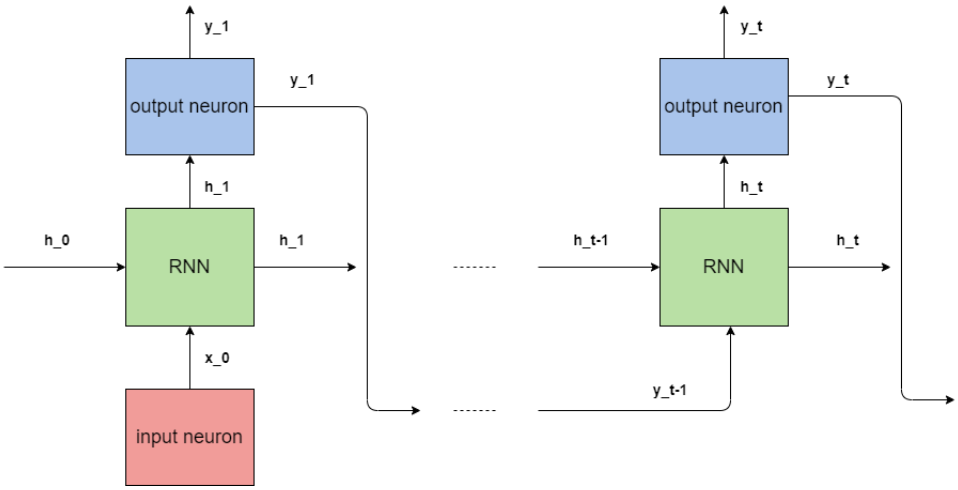

Figure 5.11: Decoder neuron.

only the initial distance is feed and the task is to develop a complete trajectory from it so it can be embarked in an ornithopter for trajectory planning purposes.

As can be seen in the Figure 5.10, a decoder has a single input and develops a sequence starting from this input. For our specific case, we will input the initial distance and we will obtain the full trajectory.

This would be possible as long as the initial input contains enough information in such a way that the optimal trajectory is defined, which is true in our example. Given an starting point A and end point B, there should be a single optimal trajectory connecting them.

Actually, the only difference with the previous networks is that at every time step, the input consists in the prediction of the previous states instead of the true previous states, see Figure 5.11.

Taking the aforementioned points in mind, the algorithm 2 is used to construct the trajectory.

Algorithm 2: trajectory loop

Result: $X = (x_0, x_1, x_2, x_3 ... x_n)$
x_0;
while $x \neq Zero$ **do**
  $x_i = RNN(\vec{x})$;
  $\vec{x}.append(x_i)$;
end

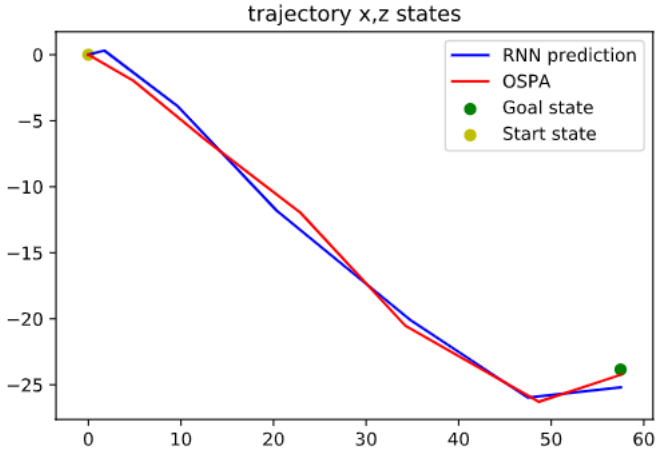

Figure 5.12: comparison with X, Z components.

5.3.2 Training

Due to the similarity with the sequence to sequence architecture, the same Keras Neural Network and training method from section 5.2 are re-used. This means that the network training is performed in exactly the same manner as for the sequence to sequence configuration. Actually, we are going to re-use the previous network training and thus use the same optimal network parameters θ^*.

5.3.3 Results analysis

Figure 5.12 shows how the OSPA trajectory is closely followed by the predicted X, Z trajectory even if only the initial distance is given as the starting point to the network. Again, this is true taking into account the 6 components as illustrated in Figure 5.13. Here, the lack of precision for some components at the target point is due to the fact that the OSPA planner has no precision constraints on u, v, θ and q components (see details in Rodríguez et al. (2020)).

The overall results can be seen in Table 5.5, where the sequence to sequence architecture is also included for comparison purposes. The following results deserve to be highlighted:

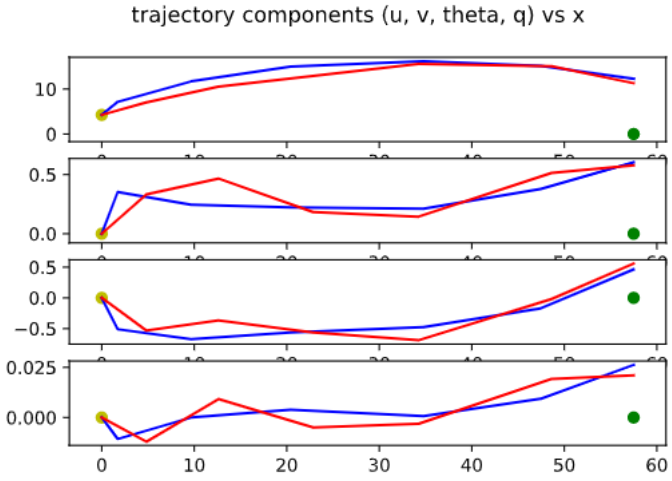

Figure 5.13: u,v, theta, q components comparison

- Far from degrading, the results have been improved with regards to the sequence to sequence configuration.
- The precision has been improved from 1.4m to 1.19m, outperforming the OSPA algorithm (2.83m).

The reason for the later improvement comes from the fact that the RNN is now free to develop the complete trajectory from the starting point with no constraints nor influence of any OSPA intermediate values. This also implies that the trajectory could end anywhere, but far from diverging it follows the right trajectory up to the target. This implies the capacity of our RNN to learn and replicate dynamic systems. More precisely, this behaviour is possible because the network has learnt

Algorithm	Cost (W)	Time (s)	Precision (m)	Trajectory error (m)
OSPA	34.68	520	2.84	`na`
RNN sequence	`na`	0.46	1.40	1.59
RNN decoder	`na`	0.54	1.19	1.56

Table 5.5: Comparison with OSPA algorithm.

a function $f^*(\theta, x)$ which approximates the underlaying trajectory flight dynamics function $f(x,t)$ for this ornithopter trajectory optimization problem. This, in turn, is sufficient to develop the complete trajectory X from the initial state distance x_0.

$$X = (x_0, x_1, x_2, x_3 ... x_n), \text{ where } x_t = f(x_0; t)$$

In the classic mechanics literature, trajectories are obtained by integration over time of the dynamic equations $f^{'}(x;t)$. Our next step will be then to modify

our RRN architecture to perform the discrete integration overtime of the dynamic equations $f^{'}(t)$, so their behaviour can be interpreted as a classic mechanics problem, that is,

$$x_t = f(x_0; t) = \int_{t_0}^{t} f^{'}(x; t)\, dt + x_0$$

5.4 Use of RNN as an Ordinary Differential Equation Integrator

5.4.1 Network architecture and path computation

While the previous architecture has proven that our network is able to learn and replicate dynamic systems, we are going to show next how we can give physical meaning and interpretation to our network behaviour.

As per equation 1.3, a recurrent neural layer, can be written as:

$$h_t = f(h_{t-1}, x_t; \theta).$$

If we consider the special case described by Figure 5.14 we obtain:

$$x_t = x_{t-1} + \Delta t * F(x_t; \theta),$$

which is Euler's formula to integrate for one time step. Therefore, the RNN prediction behaviour consists on the integration of the unknown function F over time steps Δt, which in our case is constant $\Delta t = 31.53s$.

This idea comes from a extremely simplified version from Chen et al. (2018), which proposes a RNN based ODE solver.

In Table 5.6 a summary of the Keras Neural Network can be found.

Layer	Output shape	parameter number	connected to
Input	[None, None, 6]	0	-
LSTM	[None, None, 11]	792	input
Dense	[None, None, 6]	72	LSTM
Additive	[None, None, 6]	0	input, dense

Table 5.6: ODE network summary.

In this configuration, the recurrent layer is learning the function $f^{'}(x; t)$ which is the derivative of the one learned in the previous configuration $f(x_0; t)$.

$$x_t = f(x_0; t) = \int_{t_0}^{t} f^{'}(x; t)\, dt + x_0$$

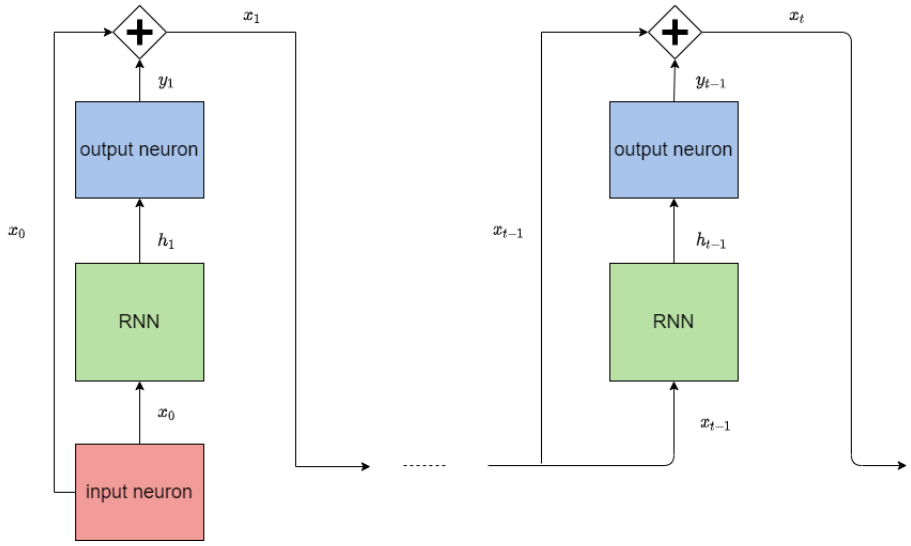

Figure 5.14: RNN ODE architecture.

In accordance with theorem 1.3.1, this function f can be arbitrary approximated by our network. Actually, the derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well, see Hornik et al. (1990).

5.4.2 Training

In Figure 5.15 the loss (MSE) decrease is shown if the same training strategy as for the previous cases is followed. It has to be noted that the validation loss decreases with the training loss up to a limit where a bias is maintained. This behaviour implies that the RNN is not able to generalize correctly.

The consequence of this training bias is reflected during the prediction by the fact that, in some cases, the trajectory diverges close to the target vicinity as can be seen in Figure 5.16

This strange behaviour is due to the padding applied to the training data set. When these additional states were added at the end of the sequence, the network learns that at the target's vicinity, the value of $\Delta x = x_t - x_{t-1}$ should be zero, which it is not true.

Making use of the bias-variance trade off explained in section 1.3.2, we can identify this error as coming from the data noise σ_ϵ^2 which induces a lower bound as per equation $MSE = \mathbb{E}_x[bias[f^*(x)]^2] + \mathbb{E}_x[var[f^*(x)]] + \sigma_\epsilon^2$. Contrary to the data noise reduction archived in section 5.2, the data pre-processing has generated this time additional data noise coming from the padding.

Therefore the padding is removed and the network is trained again. This fact increases the complexity of the RNN implementation in Keras, but still there is a

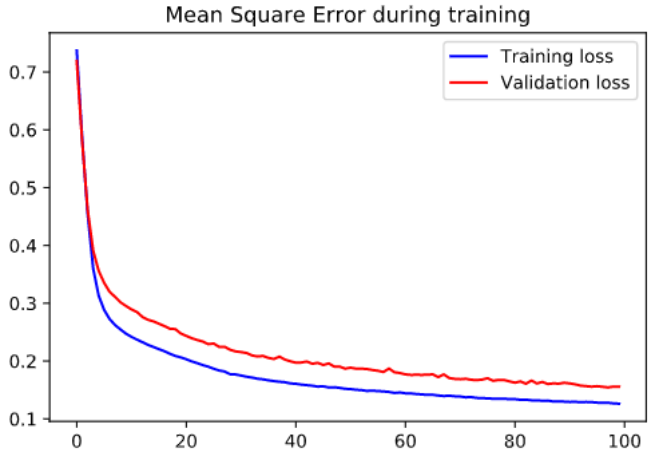

Figure 5.15: Loss (MSE) decrease during training.

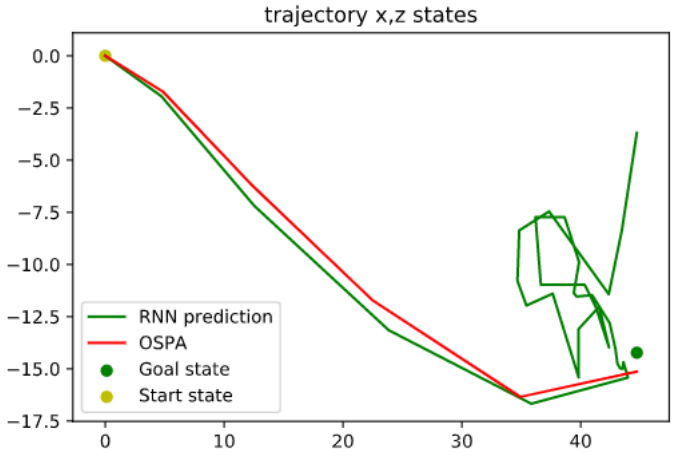

Figure 5.16: X,Z components trajectory comparison.

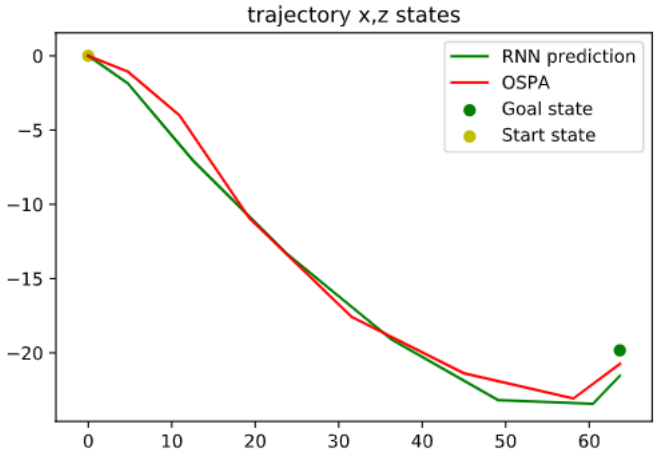

Figure 5.17: X,Z components trajectory comparison without padding.

way to do it using Ragged Tensors which allow variable size inputs during training.

5.4.3 Results analysis

Figure 5.17 shows the predicted X,Z trajectory when padding is not used, returning to a correct behaviour.

The overall results can be seen in Table 5.7, where it can be noted how the overall performance remains in the range of previous architectures, as it was expected.

Algorithm	Cost (W)	Time (s)	Precision (m)	Trajectory error (m)
OSPA	34.68	520	2.84	`na`
RNN sequence	`na`	0.46	1.40	1.59
RNN decoder	`na`	0.54	1.19	1.56
RNN ODE	`na`	0.76	1.32	1.55

Table 5.7: Metrics comparison with OSPA algorithm.

The goal of this last architecture was not to improve the performance, but rather to show how the network architecture can be chosen so we can have a physical interpretation of its behaviour.

Chapter 6

Conclusion and future work

First, we are going to sum-up what has been achieved so far:

1. It has been introduced what feedforward neural networks are, what are they for, their architecture and how their neurons work.

2. Once connected, the equations of these neurons have been developed so that the general NN algebraic equations are obtained. This was done with the aim of providing a general understanding on how actually NN are able to output a prediction given an input.

3. These equations, in turn, have shown how a given neural network form a parametric family $\{f^*(\cdot\,;\theta) \mid \theta \in \Theta\}$, where the family $f^*(\cdot\,;\theta)$ is given by the NN architecture and the parameters θ correspond to the NN weights.

4. It has been introduced the maximum likelihood framework as mean to estimate the parameter value $\hat{\theta}$ for a given family, so that under the assumed model $f^*(x;\hat{\theta})$, the observed data is the most probable. Or in other words, $f^*(x;\hat{\theta})$ is the best approximate to the true underlying function f.

5. It has been introduced that NNs are universal approximators up to any arbitrarily small error under certain conditions, and how our ornithopter trajectory optimization meets them.

6. The back propagation algorithm has been developed from the feedforward NN algebraic equations as a mean to compute the parameter $\hat{\theta}$ that gives the (local) maximum likelihood for our model.

7. The general maximum likelihood expression $\theta_{ML} = \arg\max_\theta \prod_{i=1}^{m} p_{model}(x_i;\theta)$ has been further developed into more suitable expressions depending on the problem. Actually, the ML was not possible to compute for our regression problem.

8. For our regression problem, the MSE optimization has been proven equivalent to the ML optimization under certain hypothesis.

9. For our classification problem, the minimum KL divergence or the cross-entropy have been proven equivalent to the ML. Additionally, these expressions are shown meaningful loss functions since they quantify the loss of information incurred in using our NN model instead of the true distribution.

10. The recurrent neurons have been introduced as a way to give our NN the capacity to learn temporal dynamic behaviours and thus increase the capacity of our parametric family.

11. The "virtual" unfolding of the recurrent neurons layer into feed-forwarded ones has been introduced as a way to apply the aforementioned properties and proofs.

12. The ornithopter trajectory optimization has been introduced as a physical application of all the aforementioned points.

13. The ornithopter trajectory optimization data has been presented and explained as well as the necessary data pre-processing.

14. A concrete Recurrent Neural Network candidate for the problem has been presented and explained.

15. This candidate has been adapted to both the regression and classification problems, explaining the reasons behind each output layer configuration.

16. This RNN has been applied to the ornithopter trajectory optimization problem using 4 different architectures.

17. A first architecture consisting on a classifier for the next action has shown good results improving largely the OSPA computational times while keeping a reasonable performance.

18. A second architecture consisting on a regression sequence to sequence configuration has shown even better results, outperforming the OSPA performance regarding precision to the target.

19. A third architecture consisting on a decoder configuration has exceeded the previous performance, proving the fact that the RNN is able to learn the underlaying trajectory flight dynamics.

20. A fourth and last architecture creating an Ordinary Differential Equation Integrator has shown how RNN can be designed so they behaviour can be interpreted in physical terms.

21. Finally, the bias-variance trade-off has been deduced from the MSE expression, which in turn has been used to explain undesired training behaviours.

6.1 Conclusion

The goals stated in the motivation have been achieved throughout this work:

- On the theoretical side, all neural network expressions or choices have been mathematically derived or supported by three pillars:
  1. The universal estimator theorem.
  2. The development of the NN algebraic equations derived from the neuron ones.
  3. The use of the maximum likelihood and its derivations to determine the optimal parameters for the paramedic family formed by the NN.
- On the applications side, several RNN architectures have been applied to the specific problem of the ornithopter trajectory optimization, leading to the following results:
  1. The RNN has outperformed the OSPA method both in time (0.5s vs 520s) and precision to the target (1.19m vs 2.84m).
  2. The RNN has been able to learn the underlying flight dynamics of the problem.
  3. The ODE Integrator architecture has given a physically interpreted RNN behaviour.
  4. All the results and choices have been justified using the mathematical background developed at the beginning of this thesis.

6.2 Future work

Some investigation lines for future work are the following:

- In our work, the use of information theory concepts as the cross-entropy and KL divergence has been introduced to explain the output of information of our NN with regards to the true data distribution. The use of information theory to explain how NNs work can be further developed by the use of the information bottleneck theorem Saxe et al. (2019). This theory attempts to explain the behavior of the deep learning via the transfer of information though the successive NN layers.

- One of the reasons of not having applied this theorem to this work, is due to the fact that our RNN is very simple (11 neurons) and shallow (only recurrent and output layer) to be consider deep learning. The use of a deeper and wider network can open new fields and questions as well as improve the network's performance.

- The use of deep learning is subject to having enough data. Due to the limit availability of it (236 trajectories), its usage has not been possible in this work. However, if additional data is gathered, deeper versions of the RNN can be considered.

- In line with the availability of additional trajectories, other trajectory types can be added so the capacity of the RNN to learn simultaneously different types can be tested.

- Finally, the integration of additional features to the loss function can be considered so other elements as obstacles or additional trajectory optimization metrics can be included.

Bibliography

Amidi, A. (2019), 'Cs 230 - deep learning'.
URL: *https://stanford.edu/ shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks*

Baek, S. S., Bermudez, F. L. G. & Fearing, R. S. (2011), Flight control for target seeking by 13 gram ornithopter, *in* '2011 IEEE/RSJ International Conference on Intelligent Robots and Systems', IEEE, pp. 2674–2681.

Brownlee, J. (2019), 'A gentle introduction to cross-entropy for machine learning'.
URL: *https://machinelearningmastery.com/cross-entropy-for-machine-learning/*

Chen, R. T., Rubanova, Y., Bettencourt, J. & Duvenaud, D. K. (2018), Neural ordinary differential equations, *in* 'Advances in neural information processing systems', pp. 6571–6583.

DeLaurier, J. D. (1994), 'An ornithopter wing design', *Canadian aeronautics and space journal* **40**(1), 10–18.

Glasius, R., Komoda, A. & Gielen, S. C. (1995), 'Neural network dynamics for path planning and obstacle avoidance', *Neural Networks* **8**(1), 125–133.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), 'Deep learning book', *MIT Press* **521**(7553), 800.

Guarnieri, R. A., Pereira, E. B. & Chou, S. C. (2006), 'Solar radiation forecast using artificial neural networks in south brazil', *Proceedings of the 8th ICSHMO* pp. 24–28.

Horn, J. F., Schmidt, E. M., Geiger, B. R. & DeAngelo, M. P. (2012), 'Neural network-based trajectory optimization for unmanned aerial vehicles', *Journal of Guidance, Control, and Dynamics* **35**(2), 548–562.

Hornik, K., Stinchcombe, M. & White, H. (1990), 'Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks', *Neural networks* **3**(5), 551–560.

Hornik, K., Stinchcombe, M., White, H. et al. (1989), 'Multilayer feedforward networks are universal approximators.', *Neural networks* **2**(5), 359–366.

Leshno, M., Lin, V. Y., Pinkus, A. & Schocken, S. (1993), 'Multilayer feedforward networks with a nonpolynomial activation function can approximate any function', *Neural networks* **6**(6), 861–867.

Luongvinh, D. & Kwon, Y. (2005), Behavioral modeling of power amplifiers using fully recurrent neural networks, *in* 'IEEE MTT-S International Microwave Symposium Digest, 2005.', IEEE, pp. 4–pp.

Mandic, D. & Chambers, J. (2001), *Recurrent neural networks for prediction: learning algorithms, architectures and stability*, Wiley.

Mordatch, I. & Todorov, E. (2014), Combining the benefits of function approximation and trajectory optimization., *in* 'Robotics: Science and Systems', Vol. 4.

Murphy, K. P. (2012), *Machine learning: a probabilistic perspective*, MIT press.

Neal, B., Mittal, S., Baratin, A., Tantia, V., Scicluna, M., Lacoste-Julien, S. & Mitliagkas, I. (2018), 'A modern take on the bias-variance tradeoff in neural networks', *arXiv preprint arXiv:1810.08591* .

Papachristoudis, G. (2019), 'The bias-variance tradeoff'.
URL: *https://towardsdatascience.com/the-bias-variance-tradeoff-8818f41e39e9*

Rodríguez, F., Díaz-Báñez, J. M., Sanchez-Laulhe, E., Capitán, J. & Ollero, A. (2020), 'Kinodynamic planning for an energy-efficient autonomous ornithopter', arXiv 2010.12273.

Ross, I. M. (2009), *A primer on Pontryagin's principle in optimal control*, Collegiate Publ.

Saxe, A. M., Bansal, Y., Dapello, J., Advani, M., Kolchinsky, A., Tracey, B. D. & Cox, D. D. (2019), 'On the information bottleneck theory of deep learning', *Journal of Statistical Mechanics: Theory and Experiment* **2019**(12), 124020.

Sherstinsky, A. (2020), 'Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network', *Physica D: Nonlinear Phenomena* **404**, 132306.

Shwartz-Ziv, R. & Tishby, N. (2017), 'Opening the black box of deep neural networks via information', *arXiv preprint arXiv:1703.00810* .

Tan, Y. & Saif, M. (2000), 'Neural-networks-based nonlinear dynamic modeling for automotive engines', *Neurocomputing* **30**(1-4), 129–142.

Wang, J. (1999), 'On-line path planning for autonomous mobile robots using recurrent neural networks', *IFAC Proceedings Volumes* **32**(2), 599–604.

Xu, P., Verma, A. & Mayer, R. J. (2007), Neural dynamic optimization for autonomous aerial vehicle trajectory design, *in* 'Independent Component Analyses, Wavelets, Unsupervised Nano-Biomimetic Sensors, and Neural Networks V', Vol. 6576, International Society for Optics and Photonics, p. 65760X.

Yang, S. X. & Meng, M. (2000), 'An efficient neural network approach to dynamic robot motion planning', *Neural networks* **13**(2), 143–148.