Modeling Dynamic Systems for Multi-Step Prediction with Recurrent Neural Networks

by

Nima Mohajerin

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Mechanical and Mechatronics Engineering

Waterloo, Ontario, Canada, 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Dr. Stefan C. Kremer Professor
Supervisor(s)	Dr. Steven L. Waslander Associate Professor
Internal Member	Dr. Sanjeev Bedi Professor
Internal Member	Dr.William Melek Professor
Internal-external	Dr. Fakhri Karray Professor

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis investigates the applicability of Recurrent Neural Networks (RNNs) and Deep Learning methods for multi-step prediction of robotic systems. The unmodeled dynamics and simplifying assumptions in classic modeling methods result in models that yield rapidly diverging predictions when the model is used in an iterative fashion, i.e., for multi-step prediction. However, the effect of the unmodeled dynamics can be captured by collecting datasets of the system. Deep Learning provides a strong set of tools to extract patterns from data, however, large datasets are commonly required for the methods to work well. Collecting a large amount of data from a robotic system can be a cumbersome and expensive approach.

In this work, Deep Learning methods, particularly RNNs, are studied and employed for the purpose of learning models of two aerial vehicles from experimental data. The feasibility of employing RNNs is first studied to learn a model of a quadrotor based on a simulated dataset, which yields a Multi Layer Fully Connected (MLFC) architecture. Models can be learned for multi-step prediction, recovering excellent predictions over 500 timesteps in the presence of simulated disturbances to the robot and noise on the measurements.

To learn models from experimental data, the RNN state initialization problem is defined and formulated. It is shown that the RNN state initialization problem can be addressed by creating and training an initialization network jointly with the multi-step prediction network, and the combination can be used in a black-box modeling approach such that the model produces predictions which are immediately accurate. The RNN based black-box methods are trained on an experimental dataset gathered from a quadrotor and a publicly available helicopter dataset. The quadrotor dataset, which encompasses approximately 4

hours of flight data in various regimes, has been released and is now available publicly online.

Finally, a hybrid network, which combines the proposed RNN based black-box models with a physics based quadrotor model into a single RNN-based modeling system is introduced. The proposed hybrid network solves many of the limitations of the existing state of the art in long-term prediction for robotics systems. Trained on the quadrotor dataset, the hybrid model provides accurate body angular rate and velocity predictions of the vehicle over almost 2 seconds which is suitable to be used in a variety of model-based controller applications.

Acknowledgements

First and foremost, I would like to express my appreciation and gratitude to my supervisor, Professor Steven Waslander, for his help, support, and for allowing me to pursue my passion in an area which had not been explored much before. His trust and patience during my PhD made this journey possible.

I would like to thank my committee, especially Professor William Melek for his help and support during my supervisor's sabbatical and Professor Sanjeev Bedi for providing his lab space to collect the quadrotor dataset.

I would also like to thank all of my colleagues in the Waterloo Autonomous Vehicle Laboratory (WAVELab), for their support, kindness, professionalism and friendship. Particularly, I would like to thank Melissa Mozifian, who helped me for parts of the coding and software implementation, Adeel Akhtar for his help on the hardware and data collection, Michael Smart for all the interesting conversations, and Pranav Ganti for the fun times we had during our lunch and coffee breaks.

Last but not least, I would like to sincerely thank the hero of my life, my mum, Dr. Manzar Sheibani. There are no words to explain her kindness. I will always be in her debt for selflessly providing me with her love and emotional support.

Dedication

For my family,

Manzar, Naser, Niloufar, Alborz, Afra.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Predicting the behaviour of a dynamic system has always been a challenging and important problem in engineering. The prediction accuracy, i.e., the similarity between the predicted and actual behaviour, is mainly dependent on the model that generates the prediction. In its mathematical form, a model is a set of rules, formulated as mathematical equations, that represents a phenomenon or a physical process. For instance, Newton's second law of motion, $f = ma$, explains the relation between the acceleration (a) of a point mass (m) and the force (f) acting on it.

A useful model should be less complex and cheaper to run than the real system, otherwise the modeling process is not justified. Therefore, many simplifying assumptions are usually made in modeling. In the context of modeling dynamic systems, these simplifying assumptions lead to *unmodeled dynamics* which are part of the system not explained by the model, and therefore, increase the prediction error. When the prediction is required over very short periods into the future, the unmodeled dynamics may cause negligible error, however, using the model over longer prediction horizons, the unmodeled dynamics lead to

drastic growth in the prediction error over time. In the discrete-time domain[1], a prediction required for one time step into the future is referred to as a single-step prediction, and a prediction many steps into the future is referred to as a multi-step prediction.

Multi-step prediction has many applications in state estimation, simulation and control [76, 12]. For instance, in a moving vehicle when some measurements, such as GPS readings, are temporarily unavailable, a multi-step prediction can account for the missing measurements and approximate the system position and speed over the blockout period. As another example, model based control schemes, such as Model Predictive Control (MPC) [57], can extensively benefit from an accurate long-term prediction. MPC calculates a control input sequence by optimizing a cost function, which penalizes the deviation of the model output from a desired trajectory, over a finite horizon. To avoid modeling errors affecting control performance, MPC applies the first element of the calculated input to the system and discards the rest. Then it recedes the prediction horizon one step forward in time and repeats the optimization. Accurate multi-step predictions allow for a slower update rate of the MPC, reducing the overall computational burden while maintain smoothness and accuracy of the resulting control system response.

Modeling methods can be classified as white-box, black-box or grey-box [55], referring to the level of opacity in the representation of the underlying system. White-box systems rely on models developed using the laws of physics. Black-box systems are driven entirely by collected data and avoid specifying any physical constraints on the system states. Grey-box models lie in between, with a portion that is derived from first principles and a portion that relies on learned knowledge from collected data.[2] There are two major difficulties in

[1]Throughout this thesis, the discrete-time domain is considered only.

[2]It should be noted that some researchers use the term grey-box equivalent to parameter identification process in modeling. While this usage is also correct, throughout this thesis, grey-box modeling refers to *hybrid* models that partially benefit from first principles and partially from black-box models.

the white-box approach. First, the developed model will contain many parameters which describe the system physical characteristics (mass, drag coefficient, etc.) that must be properly identified prior to using the model. Second, many properties of the system might be too difficult to model explicitly, such as the vortex-ring effect on a quadrotor vehicle [37]. Identifying the parameters of a model can be expensive. For instance, measuring the blade drag coefficient of a quadrotor needs a wind tunnel. Moreover, by changing the system physical properties slightly, the model should be adapted accordingly, which may involve new measurements and cumbersome tests. An example of the need to repeat system identification frequently in real systems can be seen when adding a payload to an aerial vehicle, which modifies the vehicle aerodynamic properties and mass distribution.

Black-box models can be categorized based on their structure. Polynomial models, such as the Wiener model, are based on polynomials for their realization [69]. Fuzzy models [92] employ a set of linguistic rules which are converted to mathematical equations using Fuzzy Inference Engines [74]. Neural Networks [32] provide a network of simple interconnected computational units (neurons) to approximate an unknown function [69]. Regardless of the method, a black-box model has many degrees-of-freedom (DOF), depicted as parameters or weights, that should be found based on a set of input-output observations from the system. This search to find the appropriate values for the model parameters is usually done through an optimization process and, since many black-box models are Machine Learning (ML) methods, the parameter optimization process is frequently referred to as a *learning* process.

In recent years, ML has been going through a rapid progress. Deep Learning (DL) [35, 34] is revolutionizing the ML field and solving problems that could not have been solved before, such as speech recognition [33] and object detection and classification in images [7,

85]. Three main components conduct this revolution; the newly available hardware capable of performing efficient parallel processing on giant numerical models, methods to train these models and availability of massive datasets. Because of the last reason, the applications of deep learning have been mainly focused on vision and image classification. However, DL methods can be also used in modeling and control of robotic systems. In fact, in mobile robotics, where the robot is deployed in an unstructured environment with noisy and uncertain sensory information, learning from observation can deal with problems that are either too difficult or too expensive to handle with classic methods.

In this thesis, DL methods are employed to model mobile robotic systems for a multi-step prediction problem. Several DL methods are comprehensively studied and extended as black-box models to learn the dynamics of two real aerial vehicles, a helicopter and a quadrotor. The helicopter dataset is publicly available, however, the quadrotor dataset had to be specifically collected for this work. Both datasets are described, with more focus on the collected quadrotor data. A grey-box approach is also proposed which combines the approximation property of the black-box models and the quadrotor motion model. Comprehensive study of the error distributions over prediction horizons are presented to assess the prediction performance of the proposed and implemented models. This work embodies the capability of neural networks in learning unmodeled dynamics of a real and challenging robotic system for multi-step prediction, for the first time, and may serve as a basis for future development and application of more sophisticated ML methods in modeling and control of such systems.

1.1 Methodology and Literature Review

The data driven approach of the black and grey-box models lies in the field of machine learning. A wide range of machine learning methods have been adopted, including neural networks, support vector machines, fuzzy inference systems, etc. Although each of these methods can be adapted to address the problem at hand, neural networks are chosen for this work because of the following reasons.

Neural networks are *universal function approximators* meaning that for every continuous bounded function, there exists a neural network which can approximate that function to any desired level of accuracy [38, 24, 17]. Support vector machines [2] and fuzzy inference systems [44] share the same property. In fact neural networks, fuzzy inference systems and support vector machines are functionally equivalent [49, 5]. However, each of them is designed for specific tasks. Support vector machines are well suited for classification problems. Although a classification problem can be considered a function approximation problem, the reverse is not always true because the output of a function can be real-valued while in a classification problem the output of the function (classifier) is a set of discrete values, each representing a class. There is no such clear boundary between fuzzy inference systems and neural networks, as both are capable of representing similar systems accurately. Fuzzy inference relies on expert knowledge in order to represent the system, however, and so tends to be employed only in situations where such knowledge is available. For quadrotors dynamics, it is not clear that expert knowledge can be advantageous.

Neural networks can be represented by computational graphs. The nodes are called *neurons* and the edges are called *weights* (or *synaptic* weights). Each neuron has an activation function which maps the values it receives from the incoming edges to its outgoing

edges. For convenience, and without loss of generality, neurons are gathered in *layers*. All neurons in a layer share the same activation function. In a neural network, a *hidden* layer does not directly provide the output. However, hidden layers contribute drastically to the representational capacity of the network [46]. There are two types of neural networks, Feed-Forward Neural Networks (FFNN) and Recurrent Neural Networks (RNNs). FFNNs are acyclic graphs meaning that the connection between layers do not form a cycle. They constitute a rich class of *static* maps. On the other hand, due to the presence of feedback within RNNs, they implement a rich class of *dynamic* mappings [43]. RNNs also possess the universal approximation property and in theory can reconstruct state-space trajectories of dynamic systems arbitrary well [25, 40, 80].

FFNNs have been used extensively in modelling and control of dynamic systems [2, 70, 18, 8, 6, 3]. In a control problem, they may be employed as a modelling part of a controller in a Lyapunov design approach. In this method using a Lyapunov function, the equations for evolution of the neural network weights are extracted so that the closed loop controller stabilizes the system [58, 13, 19, 71]. Since FFNNs lack exhibiting dynamics, they are mainly used as single-step predictor or compensator.

RNNs possess dynamics and are universal approximators for reconstructing state-space trajectories of dynamic systems [25, 40, 80], which make them suitable candidate models for multi-step prediction problem. In [25], it is shown that any finite-time trajectory of a dynamic system can be approximated by some RNNs to any desired level of accuracy given a *proper initial state*. This result is extended to discrete RNNs in [40]. One main issue with using RNNs having hidden neurons is the *state initialization* problem, i.e., how to properly assign initial values to the hidden neuron outputs. The common approach is to initialize the RNN states to zero or random values and run the RNN until the effect

of the initial states is washed out [95, 39]. However, this results in an arbitrary transient response of the RNN. For control applications, the immediate response of the model plays an important role and cannot be considered arbitrary.

Nonlinear Auto-Regressive (NAR) models are classic tools to model dynamic systems [69, 68, 15]. In [68], Narendra et al. devised methods to use Multi-Layer Perceptrons (MLPs) in the Non-linear Autoregressive eXogenous (NARX) framework. In a discrete-time fashion, NARX framework implements a dynamic system whose output at any given time, $y(k)$, is a function of the input at that time, $u(k)$, and the system output and input at previous time steps,

$$y(k) = f\Big(y(k-1), \ldots, y(k-\tau_y), u(k), u(k-1), \ldots, u(k-\tau_u)\Big),$$

where the length of the memory buffers, i.e., τ_u and τ_y, are usually given or determined through a hyper-parameter optimization process. The function $f(.)$ can be realised by various methods. In [68], the function $f(.)$ is realised by an MLP. To avoid confusion, the method to implement this function is added to the NARX abbreviation as a suffix. For instance, if $f(.)$ is realised by an MLP then the architecture will be referred to as NARX-MLP. A NARX-MLP is essentially an MLP equipped with buffers and feedback connections. Hence, it can be classified as an RNN.

The NARX-MLP architectures are often trained via a *Series-Parallel* [68] mode which uses the network target values instead of the network past outputs as the delayed version(s) of the network output. This method is also known as *teacher forcing* [68]. Clearly, this mode converts the NARX architecture into a feedforward one which therefore loses the main advantage of an RNN and limits the ability of the method to represent dynamic sys-

tems accurately. On the other hand, training NARX-MLP in a closed-loop form (Parallel mode) to model dynamic systems can be difficult due to numerical stability issues in the calculation of the gradient for learning based optimization. As it will be demonstrated in this work, NARX-MLPs, when trained in a closed-loop fashion, may become unstable or converge very slowly.

One alternative to NARX model is to define an internal state, $x(k)$, and use a one step memory buffer,

$$\begin{aligned} x(k) =& f\Big(x(k-1), y(k-1), u(k)\Big), \\ y(k) =& g\Big(x(k)\Big). \end{aligned}$$

This architecture is an example of a Recurrent Multilayer Perceptron (RMLP). An RMLP is made by one (or a few) locally recurrent layers of sigmoid neurons [43]. RMLPs have been used in a number of dynamic system identification and modelling problems, such as a heat exchanger [75], engine idle operation [51] and wind power prediction [50].

It is not clear whether using RMLPs is more advantageous than NARX-MLPs. However, in [45] it is shown that NARX-MLPs, in a single-step prediction scenario, outperform RMLPs in modelling a real helicopter. NARX RNNs have been extensively studied [82, 53] and used in various modelling and identification problems [9, 4, 52]. In [91], a Radial Basis Function (RBF) network in a NARX architecture, i.e., NARX-RBF, is used to model and control a quadrotor with three horizontal and one vertical propeller. In [86], another form of NARX-RBF is employed and trained using Levenberg-Marquardt learning method (LM) to model a small-scale helicopter. Both approaches employ a Series-Parallel training method.

Recently, Abbeel et al. used Rectified Linear Units neural networks to model dynamics of a helicopter from experimental data [78]. Although they have not used RNNs, their dataset is also used in this thesis to assess the performance of RNNs. In [94], a deep neural network, trained by a Model-Predictive Control (MPC) policy search, is used as a policy learner to control a quadrotor. The network generates one step policies and has a feed-forward architecture. In [73], a few NN architectures, such as MLPs, RMLPs, Long-Short-Term-Memory (LSTM) and Gated Recurrent Unit cells are compared against each other in one step prediction of a few small robotic datasets. In [47], a hybrid of recurrent and feed-forward architectures is used to learn the latent features for MPC of a robotic end-effector to cut 20 types of fruits and vegetables. Although the authors use recurrent structure, they also state that using their *Transforming Recurrent Units* (TRUs) in a multi-step prediction scheme results in "instability in the predicted values", so they use their proposed network as a one-step predictor. However, the recurrent latent state helps to improve the predictions.

1.2 Contributions

The main contributions of this thesis are as follows.

- Traditional RNN architectures are implemented and trained to model a simulated quadrotor from input-output data for multi-step prediction. It is demonstrated that these models do not generally perform well on the task. Therefore, a novel structurally deep RNN is proposed that is capable of learning the simulated quadrotor model for producing accurate multi-step prediction for over 5 seconds flight time. The proposed architecture employs inter-layer connections (skip connections) to alleviate

the vanishing/exploding gradient problem, which arises in deep architectures. [63, 64]

- The RNN state initialization problem is defined for multi-step prediction and the importance of initial RNN state values for control purposes is highlighted. A history-based initialization method is proposed which leads to a novel deep architecture employing a cascade of neural networks; one network generates the initial state values for the second network, which is an RNN to carry out the multi-step prediction task. The cascaded architecture provides a suitable black-box model to be trained for learning models of dynamic systems from input-output data.

- The proposed state initialization method is applied to two main classes of RNNs, sigmoid layers and gated architectures, and are trained on the input-output dataset of two real aerial vehicles, a helicopter and a quadrotor. The quadrotor dataset is collected as a part of this work and is made publicly available[3]. This work provides a first comprehensive comparison on employing various RNN architectures in modeling real robotic systems for multi-step prediction [67, 66].

- A novel grey-box approach is proposed which employs first principle model of a quadrotor, formulated as a motion model, with the proposed black-box models. The proposed grey-box models, trained on the quadrotor dataset, produce velocity and body rate predictions whose errors on average are less than 3 centimetres per second and 2 degrees per second [65].

[3] https://github.com/wavelab/pelican_dataset

1.3 Thesis organization

The remainder of this thesis is organized as follows:

- **Chapter 2** provides a background in neural networks and deep learning. The model of a neuron and the basics of the architectures to be used in this thesis are introduced. Methods to obtain the network gradient and supervise-train the Recurrent Neural Networks are explained.

- **Chapter 3** formulates the multi-step prediction problem for dynamic systems. To show the feasibility of using RNNs in addressing the multi-step prediction problem, a novel structurally deep RNN is proposed and formulated. Through simulation, the proposed RNN is trained for multi-step prediction of a simulated quadrotor. The simulation results show a superior performance over traditional RNN architectures.

- **Chapter 4** formulates the state initialization problem in RNNs which is encountered when the training trajectories do not start from a zero initial condition. Two solutions are proposed and formulated which result in novel deep architectures for RNNs in the context of multi-step prediction. Results on modeling two aerial vehicles from experimental input-output data, using various architectures, are presented and compared after the datasets are introduced.

- **Chapter 5** describes a grey-box modeling approach, which results in two hybrid architectures. The hybrid architectures employ black-box models and a motion model of a quadrotor and are trained on the quadrotor dataset. The prediction results are presented and compared with the predictions from white-box and black-box models of the quadrotor.

- **Chapter 6** concludes the thesis and provides suggestions on future works and expansions.

Chapter 2

Background

In this chapter, the principles governing neural networks, and more specifically RNNs, are presented. The interested reader is referred to the following texts for further details on the field of neural networks [32, 43, 11, 27]. Also, since the main system to be modeled in this thesis is a real quadrotor, in this chapter the quadrotor vehicle is introduced and a model for it is developed using first principles. There are multiple ways to obtain a model of a quadrotor [37, 89, 23]. In this work, the approach described in [23] is adopted.

2.1 Artificial Neural Networks

An Artificial Neural Network[1] is a network of interconnected simple processing elements referred to as *neurons*. Multiple neuron models have been proposed [32, 62, 16, 20]. In general, a neuron is a (nonlinear) function which maps a multi-dimension input to a scalar. One class of commonly used neurons, and the one used extensively in this thesis, is based

[1] Artificial Neural Networks and neural networks are referred to interchangeably.

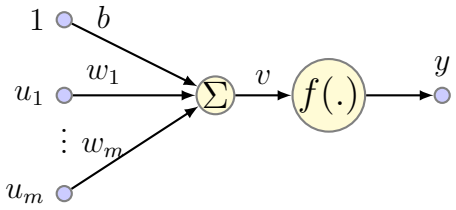


Figure 2.1: A general model of a neuron.

on the pioneering McCulloch-Pitts model:

$$y = f(\sum_{i=1}^{n} u_i w_i + b) = f(\mathbf{w}^\top \mathbf{u} + b) = f(v) \tag{2.1}$$

In this model, $\mathbf{u}^\top = [u_1, \ldots, u_m] \in \mathbb{R}^m$ is the input vector to the neuron, $\mathbf{w}^\top = [w_1, \ldots, w_m] \in \mathbb{R}^m$ is the multiplicative weight vector and b is the bias term. The scalar valued v, which is an *affine* mapping of the input, is commonly referred to as *activation potential* or *induced local field*. There are many choices for the activation function $f(.)$. If $f(.)$ represents a threshold function,

$$f(v) = \begin{cases} 0, & v < 0 \\ 1, & v \geqslant 0 \end{cases}$$

then the model represented by Equation (2.1) is called McCulloch-Pitts model. To avoid confusion, the class of all neurons with any activation function applied on an affine transformation of the input are referred to as *affine neurons*. The activation function used in this thesis is the *hyperbolic tangent* function,

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{2.2}$$

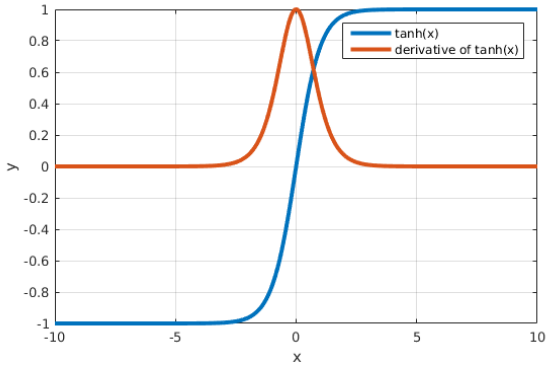


Figure 2.2: Graph of the hyperbolic tangent function.

and it is plotted in Figure 2.2. The hyperbolic tangent function is *sigmoidal*. A sigmoidal function $f(.)$ is a real-valued, monotonic and differentiable function such that

$$\lim_{x\to\infty} f(x) = +1, \quad \lim_{x\to-\infty} f(x) = -1. \tag{2.3}$$

There are other activation functions such as Radial Basis Functions (RBF) and Rectified Linear Units (ReLUs) [27]. There is no significant difference between any sigmoidal function in the problems being investigated in this thesis. RBFs, however, are suited more for problems where spatial information among the observations carry significant information, and ReLUs are suited more for classification problems [27].

A basic example of a neural network is shown in Figure 2.3. It has m inputs and n outputs. Note that in this architecture, signals flow in one direction: from input nodes toward outputs. This architecture, and similar ones which satisfy this property, are called Feed-Forward Neural Networks (FFNNs).

In Figure 2.3, each stack of neurons is called a layer (shaded regions) and a circle denoted by $\mathcal{N}_{i,j}$ represents the i^{th} neuron inside the j^{th} layer. If all neurons inside a layer

have the same activation function, it is convenient to write the equation for a layer as,[2]

$$\mathbf{y}_l = \mathbf{f}(\mathbf{v}_l) = \mathbf{f}(\mathbf{W}_l\mathbf{u}_l + \mathbf{b}_l), \tag{2.4}$$

where l represents the layer index. The activation potential is denoted by $\mathbf{v}_l$. Let the number of neurons in the l^{th} layer be n_l, i.e., $\mathbf{y}_l \in \mathbb{R}^{n_l}$. If the input to the l^{th} layer, represented by Equation (2.4), has m_l elements (i.e. $\mathbf{u}_l \in \mathbb{R}^{m_l}$), then $\mathbf{W}_l \in \mathbb{R}^{n_l} \times \mathbb{R}^{m_l}$ and $\mathbf{b}_l \in \mathbb{R}^{n_l}$.

The layer at which the outputs are generated is called the output layer. All the intermediate layers between the input and output layers are called hidden layers. The example shown in Figure 2.3 has one hidden layer. If the activation function of neurons inside the hidden layer(s) in an FFNN is sigmoidal then it is called a Multi-Layer Perceptron (MLP).

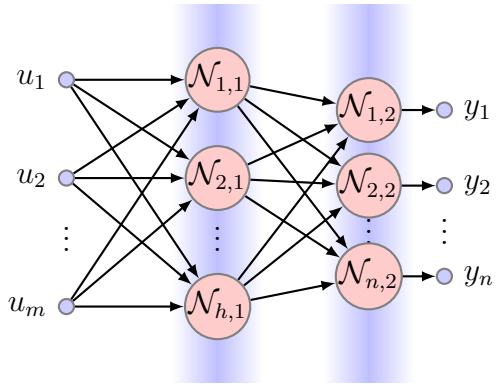


Figure 2.3: A simple neural network.

Initially, the sigmoidal activation functions were the primary choice for neural networks because of the *universal function approximation* property. Informally, the universal approximation theorem states that for every continuous bounded function, there exists an MLP network which can approximate that function to any desired level of accuracy. Formal

[2]In this text, bold lower-case letters indicate vectors, and bold upper-case letters indicate matrices.

definition of this theorem as well as proofs can be found in [32, 38, 24, 17]. Neural networks with other type of activation functions, such as RBF, are also universal approximator, for more details see [31] and [83].

2.2 Supervised Training in Neural Networks

In ML, supervised training is performed through an optimization process. In neural networks, the optimization is dominantly chosen to be a variant of gradient descent approaches. In brief, a cost function is defined which reflects the training goal. Then the derivative of the cost with respect to the network weights (the network gradient) is calculated using *backpropagation*. Backpropagation (BP) is essentially a repeated application of the chain rule on the cost function to calculate its gradient. Because the cost is a function of the network output, the chain of derivatives start from the output and is followed backwards toward the network inputs. Backpropagation through many layers result in a phenomena known as *vanishing gradient problem*.

Consider a deep MLP, i.e., an MLP with $N > 1$ layers each having h neurons. For simplicity, assume $N = 5$, which is illustrated in Figure 2.4. The equation to calculate all layer outputs is given by (2.4), where $m_l = n_{l-1} = h$ for $l = 2, ..., 5$ and $m_1 = n_5 = 1$.

As a part of the BP algorithm, let us focus on two derivatives, the derivative of the output y w.r.t. $\mathbf{w}_1$ and $\mathbf{w}_5$.[3] From Figure 2.4 it is clear that $\mathbf{u}_l = \mathbf{y}_{l-1}$ for $l = 2, ..., 5$ where $\mathbf{y}_l^\top = [y_{l1}, y_{l2}, ..., y_{lh}]$. To obtain $\frac{\partial y}{\partial \mathbf{w}_5}$, we apply the chain rule once to the network output

[3]Note that because both the input and output have dimension one, the weights at the first and last layers are denoted using lower-case bold letter.

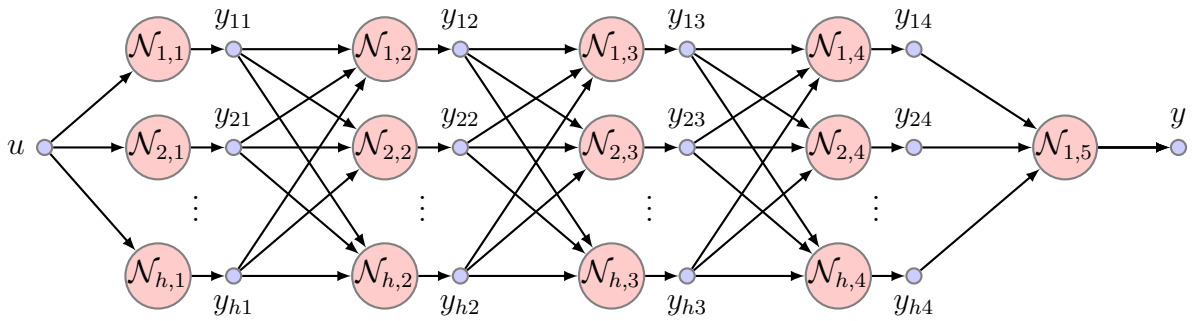


Figure 2.4: A deep MLP with 5 layers.

y,

$$\frac{\partial y}{\partial \mathbf{w}_5} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial \mathbf{w}_5} = f'(\mathbf{w}_5^\top \mathbf{y}_4 + b_5)\mathbf{y}_4 \tag{2.5}$$

where $f'(.)$ is the derivative of $f(.)$ w.r.t. its argument. Repeatedly applying the chain rule, one can obtain $\frac{\partial y}{\partial \mathbf{w}_1}$ using the following equation,

$$\frac{\partial y}{\partial \mathbf{w}_1} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial \mathbf{y}_4}\prod_{i=4}^{2}\frac{\partial \mathbf{y}_i}{\partial \mathbf{v}_i}\frac{\partial \mathbf{v}_i}{\partial \mathbf{y}_{i-1}}\frac{\partial \mathbf{y}_1}{\partial \mathbf{v}_1}\frac{\partial \mathbf{v}_1}{\partial \mathbf{w}_1}, \tag{2.6}$$

where the index i starts at 4 and decreases so that the matrix multiplications are in the correct order.[4] There are two types of terms in Equation (2.6). One is the derivative of a layer output w.r.t its activation potential, e.g., $\frac{\partial \mathbf{y}_5}{\partial \mathbf{v}_5}$, and the other is the derivative of a layer activation potential w.r.t. the input to the layer, e.g, $\frac{\partial v_5}{\partial \mathbf{y}_4}$. The former corresponds to the derivative of the activation function w.r.t. its argument and the latter is equivalent to the layer multiplicative weights $\mathbf{W}_i$,

$$\frac{\partial \mathbf{v}_i}{\partial \mathbf{y}_{i-1}} = \mathbf{W}_i. \tag{2.7}$$

[4]The derivative of a vector-valued function with p elements w.r.t. a variable vector with q elements is a matrix, known as Jacobian, which is a member of $\mathbb{R}^p \times \mathbb{R}^q$.

The network weights should be initialized to small values for training. Therefore, using a sigmoidal activation functions, all of the terms in Equation (2.6) are values (much) less than 1. Comparing Equations (2.5) and (2.6) it is clear that,

$$\frac{\partial y}{\partial \mathbf{w}_1} << \frac{\partial y}{\partial \mathbf{w}_5}. \tag{2.8}$$

This results in an optimization process which searches the weight space unevenly and punishes the shallow weights ($\mathbf{w}_5$) much larger than the deep weights ($\mathbf{w}_1$). The decaying gradient, as described above, can cause the numerical optimization process involved in training a deep NN to become unstable due to severely fluctuating cost values. Moreover, the tiny deep gradient values may also cause machine precision errors. Clearly, the deeper a network, the more severe the vanishing gradient problem becomes. In Recurrent Neural Networks, sigmoidal layers are *virtually* placed in a series architecture which will be discussed next.

2.3 Recurrent Neural Networks

Consider a network in which inputs, activations and outputs are time varying signals:

$$\mathbf{y}(k) = \mathbf{f}(\mathbf{W}\mathbf{u}(k) + \mathbf{b}), \tag{2.9}$$

where k indicates a discrete-time index. A continuous time formulation can also be employed [25]. This work focuses on the discrete time case, as it most easily represents the discrete nature of network input measurements from sensors.

As stated earlier, the flow of signals in FFNNs is unidirectional. Therefore, after an

FFNN is trained and the weights are fixed it becomes a static mapping from inputs to outputs. Although FFNNs are universal approximators, they lack the capability to exhibit any form of dynamical behaviour [32, 43]. Recurrent Neural Networks (RNNs) are not only universal approximators [25, 40, 80] but also have internal dynamics, and are therefore considered strong candidate for accurate representation of dynamical systems. An example of a Fully Connected RNN (FCRNN) is illustrated in the Figure 2.5.

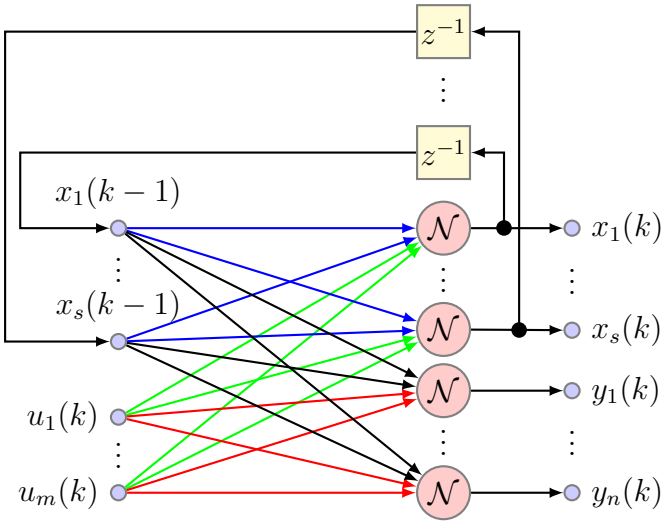


Figure 2.5: An example of FCRNN with m inputs, s hidden and n output neurons. Each circle denoted with $\mathcal{N}$ represents a neuron. Different colors represent different weights.

A convenient way to present the equations governing the dynamic of an RNN is a *state-space* representation. The name arises from the similarity with state-space representation of dynamic systems. In fact, RNNs *are* nonlinear dynamic systems,

$$\begin{aligned} \mathbf{x}(k) &= \mathbf{f}(\mathbf{A}\mathbf{x}(k-1) + \mathbf{B}\mathbf{u}(k) + \mathbf{b}_x) \\ \mathbf{y}(k) &= \mathbf{g}(\mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) + \mathbf{b}_o). \end{aligned} \tag{2.10}$$

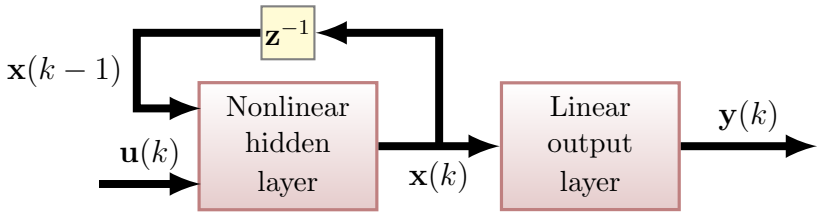


Figure 2.6: RMLP illustrated as a state-space model.

In this representation, which corresponds to Figure 2.5, the following hold,

$$\begin{aligned}
\text{RNN state vector: } & \mathbf{x}(k) \in \mathbb{R}^s, \\
\text{RNN output vector: } & \mathbf{y}(k) \in \mathbb{R}^n, \\
\text{Independent input vector: } & \mathbf{u} \in \mathbb{R}^m, \\
\text{State feedback weights: } & \mathbf{A} \in \mathbb{R}^s \times \mathbb{R}^s, \text{blue connections in Figure 2.5} \\
\text{Input-to-state weights: } & \mathbf{B} \in \mathbb{R}^s \times \mathbb{R}^m, \text{green connections in Figure 2.5} \\
\text{State-to-output weights: } & \mathbf{C} \in \mathbb{R}^n \times \mathbb{R}^s, \text{black connections in Figure 2.5} \\
\text{Input-to-output weights: } & \mathbf{D} \in \mathbb{R}^n \times \mathbb{R}^m, \text{red connections in Figure 2.5} \\
\text{State bias term: } & \mathbf{b}_s \in \mathbb{R}^s, \\
\text{Output bias term: } & \mathbf{b}_o \in \mathbb{R}^n.
\end{aligned} \tag{2.11}$$

Also, the vector-valued functions, $\mathbf{f}(.)$ and $\mathbf{g}(.)$, are the state and output activation functions. There are a variety of architectures in RNNs. The explained architecture, which uses an MLP with feedback connections (Equation (2.10)), is called a Recurrent MLP or RMLP. Figure 2.6 shows the block diagram of an RMLP suited for system identification. Note that the output layer activation function is the identity. A comprehensive study on RNN architectures can be found in [59, 43].

One interesting architecture, particularly for dynamic system identification, is called

Non-linear Auto-Regressive eXogenous, or NARX, model. In statistical signal processing auto-regressive models are used to estimate the output of a linear dynamic process in discrete time domain. In an auto-regressive scheme, the model is represented by a mapping from the previous input and output values to the present output. NARX is an extension of the same idea to the nonlinear dynamic systems. In a NARX framework, the system output at a time step k is modelled as a nonlinear mapping of the previous values of the system output, the current and past values of the input. In practice, the past time-horizon is finite. For a system with scalar input and output, the NARX model is,

$$y(k) = g\Big(u(k), u(k-1), \ldots, u(k-d_x), y(k-1), \ldots, y(k-d_y)\Big). \tag{2.12}$$

One of the earliest successfully employed neural network models for modeling a dynamic system is based on the NARX architecture [68], where the function $f(.)$ is realized by an MLP. Although any other FFNN might be used to realize it, the NARX architecture with a single hidden-layer MLP has been used commonly ever since [2, 70, 18, 8, 6, 3]. The dynamics of this model with m inputs and n outputs is governed by the following equation,

$$\mathbf{y}(k) = \mathbf{W}_2\mathbf{f}\Big(\mathbf{W}_1\boldsymbol{\phi}(k)\Big), \tag{2.13}$$

where $\boldsymbol{\phi}(k)$ is called a *regressor* and is a vector which contains the vectors $\mathbf{u}(k-i)$ for all $i = 0, 1, \ldots, d_x$ and $\mathbf{y}(k-i)$ for $j = 1, \ldots, d_y$. For each k we shall have $\mathbf{y}(k) \in \mathbb{R}^n$, $\mathbf{u}(k) \in \mathbb{R}^m$ and therefore $\boldsymbol{\phi}(k) \in \mathbb{R}^{m(d_x+1)+nd_y}$.

To provide a delayed version of a signal in discrete time, it is customary to employ a bank of buffers, referred to as Tapped Delay Line (TDL) [32]. A TDL with length of t is basically a bank of t buffers arranged in a serial fashion as depicted in Figure 2.7.

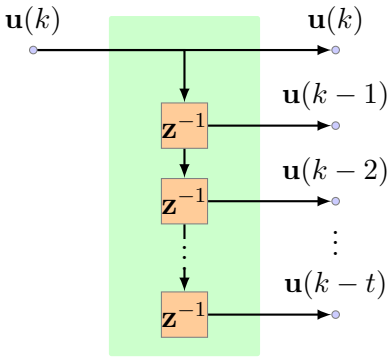


Figure 2.7: A TDL with t number of buffers which provides delayed versions of a signal.

2.4 Learning Algorithms for RNNs

As stated earlier, at the heart of training a neural networks lies an optimization process which in a supervised learning case is often derivative-based. Two common methods exist: Real Time Recurrent Learning (RTRL) and Back Propagation Through Time (BPTT). In RTRL the network gradient is continually updated as the network receives input elements and the weights are updated in the gradient descent direction using a learning rate. In BPTT, however, the gradient is calculated for a (finite) time horizon and any gradient-based method can be applied to update the network weights. Both methods are briefly explained in the next two subsections, however, more details can be found in [32, 43, 27, 90].

2.4.1 Real Time Recurrent Learning

RTRL was originally proposed by Ronald J. Williams and David Zipser in 1989 [90]. This algorithm is suitable when it is required to train the network while continually running it. In the original description, RTRL is formulated for a fully connected RNN (FCRNN) with an arbitrary number of neurons and input lines, where the states are equal to the outputs (Figure 2.8). However, the concept is applicable to a number of other architectures.

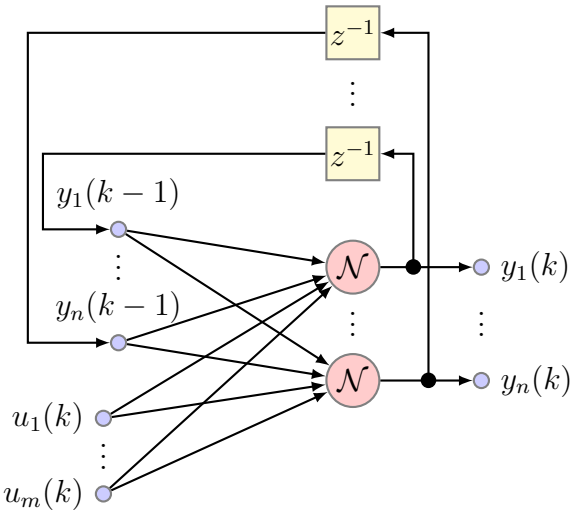


Figure 2.8: An example of FCRNN with m input, n hidden neurons where states and outputs are the same.

To show how RTRL works, let us first concatenate inputs ($\mathbf{u}(k)$) and outputs ($\mathbf{y}(k)$) to form the $(m+n)$-tuple $\mathbf{z}(k)$. Thus, if $\mathcal{I}$ is the set of input indices and $\mathcal{O}$ the set of output indices, then

$$z_i(k) = \begin{cases} u_i(k) & \text{if } i \in \mathcal{I} \\ y_i(k) & \text{if } i \in \mathcal{O} \end{cases} \tag{2.14}$$

In a similar manner we can arrange all weights, that exist between all neurons in the network, into an $n \times (m+n)$ weight matrix $\mathbf{W}$. Since the network is fully connected, the activation of each neuron at time k is

$$v_i(k) = \sum_{l \in \mathcal{O} \cup \mathcal{I}} w_{il} z_l(k) \tag{2.15}$$

and the network outputs at the next time step will be calculated by

$$y_i(k+1) = f_i(v_i(k)) \tag{2.16}$$

where i ranges over $\mathcal{U}$. Equations (2.14), (2.15) and (2.16) define the dynamics of the network for which the RTRL algorithm is presented next.

Let $\mathcal{T}(k)$ denote the set of indices of neurons for which a desired value $y_i^d(k)$ exists at time k. Then the error signal $e_i(k)$ is

$$e_i(k) = \begin{cases} y_i^d(k) - y_i(k) & \text{if } i \in \mathcal{T}(k) \\ 0 & otherwise \end{cases} \tag{2.17}$$

The instantaneous error of the network is

$$E(k) = \frac{1}{2} \sum_{i \in \mathcal{O}} e_i^2(k) \tag{2.18}$$

The minimization cost function can be either the instantaneous error or a total error over a given period such as

$$L = E(k)\Big|_{k=t_0}^{T+t_0} = \sum_{k=t_0}^{T+t_0} E(k). \tag{2.19}$$

RTRL is based on the gradient descent algorithm and adjusts the weights along the negative of the gradient of the cost function, i.e., $\nabla_{\mathbf{W}} E(k)$. Thus, for both of the above cost functions (i.e., Equations (2.18) and (2.19)) we need to compute the partial derivative of $E(k)$ with respect to the individual weights at time k:

$$\Delta w_{ij}(k) = -\eta \nabla_{\mathbf{W}} E(k) = -\eta \frac{\partial E(k)}{\partial w_{ij}} = -\eta \sum_{l \in \mathcal{O}} e_l(k) \frac{\partial y_l(k)}{\partial w_{ij}}, \tag{2.20}$$

where η is the learning rate which can be either a fixed positive value or determined using

a line search method. Using Equations (2.16) and (2.15) for $l \in \mathcal{O}$:

$$\frac{\partial y_l(k)}{\partial w_{ij}} = f_l'(v_l(k)) \left[\sum_{p \in \mathcal{O}} w_{lp} \frac{\partial y_p(k)}{\partial w_{ij}} + \delta_{il} z_j(k) \right], \tag{2.21}$$

and δ_{il} denotes the Kronecker delta, that is $\delta_{il} = 1$ if $i = l$, and 0 otherwise.

Assuming the initial state of the network has no functional dependence on the weights we have

$$\frac{\partial y_l(t_0)}{\partial w_{ij}} = 0. \tag{2.22}$$

Therefore, Equations (2.21) and (2.22) constitute a recursive formula to compute $\frac{\partial y_l(k)}{\partial w_{ij}}$ and using Equation (2.20) the RTRL weight update rule is obtained.

In the case of the cumulative cost function (Equation (2.19)), one must sum all the individual updates $\Delta w_{ij}(k)$ over the interval $[t_0, T+t_0]$ and the final weight update becomes

$$\Delta w_{ij} = \sum_{k=t_0}^{T+t_0} \Delta w_{ij}(k). \tag{2.23}$$

2.4.2 Back Propagation Through Time

Back Propagation Through Time or BPTT is the temporal extension of the BP method to RNNs. As the name implies, to use this method, one first needs to *unfold* the network back in time either an infinite or finite number of time-steps. This procedure yields a *deep* FFNN where the standard BP algorithm is applicable, bearing in mind that the weight values are shared across the layers. As described in Section 2.2, such a deep architecture will result in the vanishing gradient problem. However, since the weights are shared across the layers, there is a slight difference between the vanishing gradient problem arises in deep

FFNNs and RNNs. Although both have the same effect, for clarity the term *structural vanishing problem* is used, in this work, to refer to the former and *temporal vanishing problem* to refer to the latter. In an RNN, if the weight values are larger than 1, the repetitive multiplications that occur when applying the chain rule will result in gradient values which can be extremely large. This is referred to as the *exploding* gradient problem and similar to the vanishing gradient problem deteriorates the training of an RNN. For more details on the vanishing/exploding gradient problem in RNNs, see [36].

The temporal vanishing/exploding gradient problem becomes particularly important when there are long-term dependencies, i.e., the output at a time step is significantly dependent on information fed to the network in the distant past. However, the structural vanishing gradient problem occurs when there are multiple layers connected in series and disables the network to efficiently learn deep weights. Long Short Term Memory cells (LSTMs) are probably the most successful attempt to resolve the temporal vanishing gradient problem in RNNs. For more information and detailed discussions refer to [81, 29].

2.5 Long Short Term Memory RNNs

In an attempt to facilitate the flow of information throughout a network, LSTMs employ *cell*s equipped with *gates* to remember, forget or output information [81]. Basically, gates in LSTMs are affine layers with sigmoid activation function ($\frac{1}{1+e^{-x}}$) which are trained to let pass the information throughout the network in such a way that the gradient of information is preserved across time. There are many versions of LSTMs [29]. The LSTMs employed in this work are described in [79] and depicted in Figure 2.9 which are equipped with *peephole* connections. Peephole connections are connection from the cell, $\mathbf{c}(.)$, to the gates. The

equations of the LSTM in this work are given by the following,

$$
\begin{aligned}
\mathbf{g}_i(k) =& \boldsymbol{\sigma}\Big(\mathbf{W}_i^i\mathbf{u}(k) + \mathbf{W}_i^m\mathbf{m}(k-1) + \mathbf{W}_i^c\mathbf{c}(k-1) + \mathbf{b}_i\Big),\\
\mathbf{g}_f(k) =& \boldsymbol{\sigma}\Big(\mathbf{W}_f^i\mathbf{u}(k) + \mathbf{W}_f^m\mathbf{m}(k-1) + \mathbf{W}_f^c\mathbf{c}(k-1) + \mathbf{b}_f\Big),\\
\mathbf{g}_o(k) =& \boldsymbol{\sigma}\Big(\mathbf{W}_o^i\mathbf{u}(k) + \mathbf{W}_o^m\mathbf{m}(k-1) + \mathbf{W}_o^c\mathbf{c}(k) + \mathbf{b}_o\Big),\\
\mathbf{c}(k) =& \mathbf{g}_i(k) \odot \mathbf{f}\Big(\mathbf{W}_c^i\mathbf{u}(k) + \mathbf{W}_c^m\mathbf{m}(k-1) + \mathbf{b}_c\Big) + \mathbf{g}_f(k) \odot \mathbf{c}(k-1),\\
\mathbf{m}(k) =& \mathbf{g}\Big(\mathbf{c}(k)\Big) \odot \mathbf{g}_o(k),\\
\mathbf{y}(k) =& \mathbf{h}\Big(\mathbf{W}_y\mathbf{m}(k) + \mathbf{b}_y\Big) = \mathbf{W}_y\mathbf{m}(k) + \mathbf{b}_y.
\end{aligned}
\tag{2.24}
$$

In this set of equations, indices i, f, o and c correspond to the *input gate*, *forget gate*, *output gate* and *cell*. The operator $\odot$ indicates an element-wise multiplication. Gate activation functions, $\boldsymbol{\sigma}(.)$, are logistic sigmoid while the cell activation function $\mathbf{g}(.)$ and the output activation function $\mathbf{h}(.)$ are chosen by the designer. Since the problem at hand is regression, the activation functions $\mathbf{h}(.)$ and $\mathbf{g}(.)$ are set to identity and tangent-hyperbolic function, respectively. Detailed gradient calculations can be found in [81, 26]. Note that in LSTMs there are two types of state: cell states, $\mathbf{c}(k)$, and hidden states, $\mathbf{m}(k)$. One significant difference between LSTMs and traditional RNNs, such as an RMLP, is that in traditional RNNs the neuron outputs are not multiplied by each other, i.e., they are *first-order* networks. However, LSTMs benefit from such inter-multiplication throughout the cell in a smart way. Therefore, LSTMs can also be categorized as *second-order* RNNs. More about second order RNNs can be found in [43].

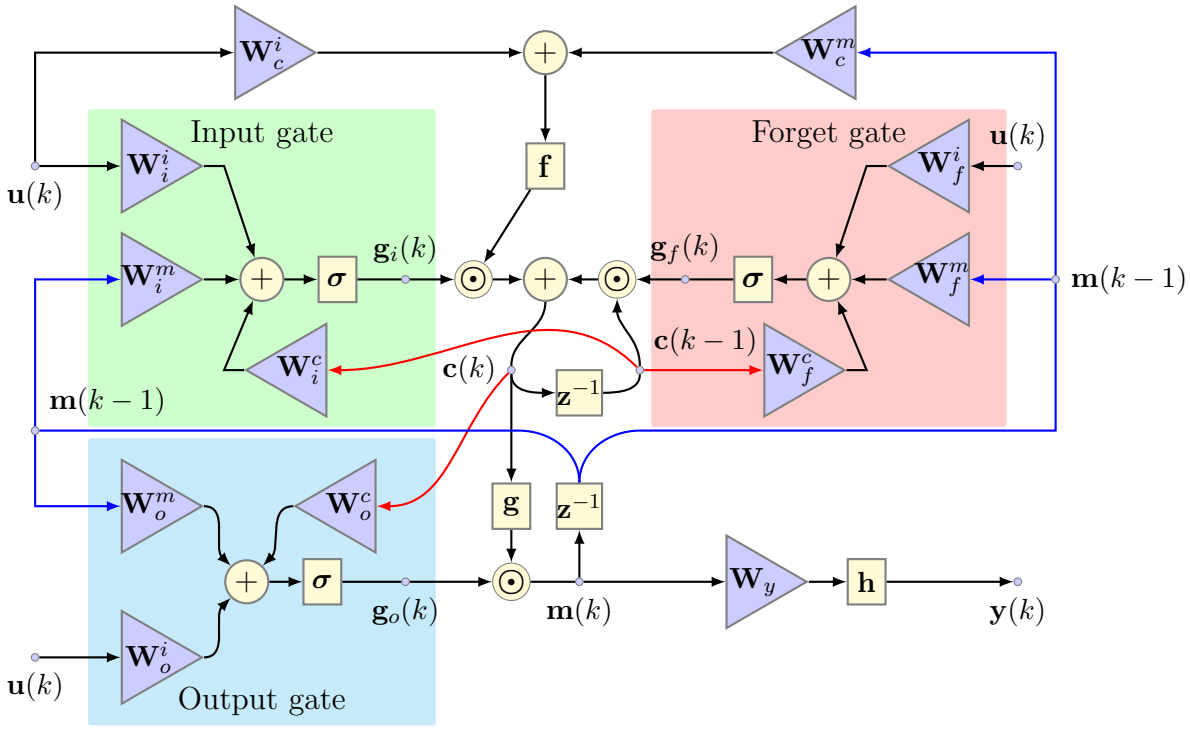


Figure 2.9: A Long-Short-Term-Memory cell with peephole connections. The blue colored connections are the feedback connections, red ones are the peepholes and the black ones are the feedforward connections.

2.6 Learning Considerations

Having calculated the gradients, the next step in training any NN is to run an optimization. Variants of first-order optimization methods have been almost established as standard methods to train NNs. Although second order methods converge faster (particularly in regression problems), the excessive memory demand and computation cost outweigh the benefit of faster convergence. All gradient-descent methods work on the same basis, however, second order methods provide a more "informative" choice for the learning step [72] based on the Hessian (or approximations to it) of the cost. First order methods, such as the Hessian Free method [61] or the Conjugate Gradients method [72], try to incorpo-

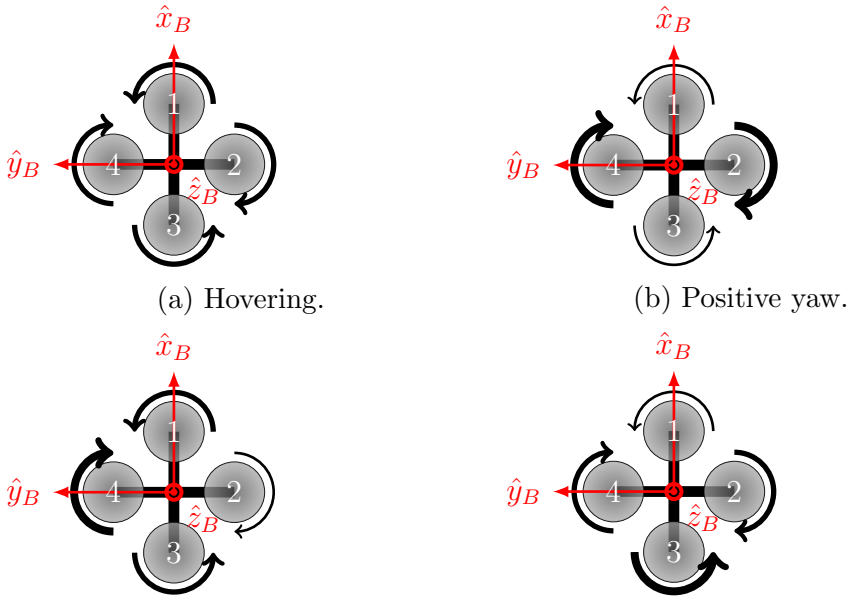
rate the second order information (the curvature information) of the cost without explicitly computing or approximating the Hessian. Other methods such as ADADELTA [93], RMSProp [87], ADAM [42], etc, which are all first-order, provide different mechanisms for updating the learning rate. In this thesis, two optimization methods are used, the Levenberg-Marquardt-Method (LMM) [30] and ADAM [42].

As with all machine learning methods, the use of large datasets can be computationally prohibitive, using small datasets can lead to overfitting and weights getting stuck in local minima. Cross-validation is commonly used to avoid such issues [11], and is employed here as well. Additionally, both regularization and randomization in the gradient descent update can improve convergence properties [77], and these methods are also employed to improve learning performance.

Although the RTRL and BPTT methods provide the two main frameworks to calculate the gradients, in large RNNs it is far from straight-forward to calculate the gradients. For instance, a slight change in the network architecture has a heavy impact on the form of Equation (2.21). Quite recently, due to progress in hardware and versatile software, very large networks with sophisticated architectures can be implemented and trained, the gradients can be calculated automatically and optimization can be run concurrently on many thousands of cores. In the next chapter a simple method is proposed and formulated to simplify the gradient calculations in a wide range of RMLPs with *skip-connections* which makes it much easier to modify RNNs architecture, and allows network architecture to be designed more easily for each new dynamical system.

2.7 Quadrotor Model

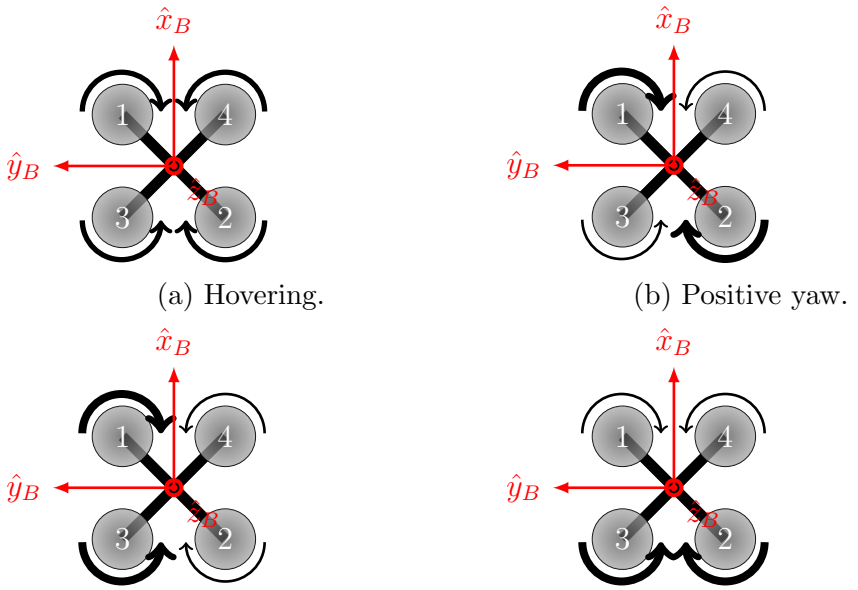
A quadrotor is a rotorcraft aerial vehicle which generates its lift by two pairs of identical fixed pitched propellers. In the commonly used quadrotors, each pair is mounted at the two ends of an arm and the two arms are attached at the center. The propellers of each pair rotate in the same direction which opposes the direction of rotation of the other pair. As the rotor disks are fixed, the movement of the quadrotor is controlled by changing the rotation speed of the propellers. Depending on how they are controlled, two configurations exist; a plus ('+') configuration and an X configuration. The effect of rotor speeds to achieve different movements are depicted in Figures 2.10 and 2.11, for the plus and X configurations, respectively. The X configuration described in this thesis is tailored to the Pelican quadrotor (Section 4.4.1).



(a) Hovering. (b) Positive yaw.

(c) Positive roll. (d) Positive pitch.

Figure 2.10: Basic quadrotor movements for plus configuration, top view. The thickness of arrows around the rotors proportionally corresponds to the rotor speed.



(a) Hovering. (b) Positive yaw.

(c) Positive roll. (d) Positive pitch.

Figure 2.11: Basic quadrotor movements for X configuration, top view. The thickness of arrows around the rotors proportionally corresponds to the rotor speed.

To study the behaviour of a quadrotor, it is convenient to describe the vehicle translation and rotation in two frames: an inertial frame $\{\mathcal{R}, (O, \hat{x}_I, \hat{y}_I, \hat{z}_I)\}$, which is attached to the earth and a body-fixed frame $\{\mathcal{R}_B, (O_B, \hat{x}_B, \hat{y}_B, \hat{z}_B)\}$ attached to the body. For each of the plus and X configuration, the body-fixed frame is illustrated in Figures 2.10 and 2.11. Note that the body-fixed frame can be converted from one configuration to the other by applying a 45 degrees rotation on the xy plane.

In Figure 2.12, both the inertial frame and the body-fixed frame are illustrated for a plus configuration. The frames are right hand coordination systems. The origin of the body-fixed frame O_B is assumed to be placed at the vehicle center of mass. Therefore, the position of the quadrotor is the position of O_B measured in the inertial frame and represented by $\boldsymbol{\xi} = [x \;\; y \;\; z]^T$. The orientation of the vehicle is represented by the Euler angle vector, which represents the angles of the body-fixed frame in the inertial frame. The

Euler angle vector, $\boldsymbol{\eta} = [\phi \;\; \theta \;\; \psi]^T$, has three components: *roll* (ϕ), *pitch* (θ) and *yaw* (ψ), illustrated in Figure 2.12.

Given a vector, $\mathbf{v}_B$, in the body-fixed frame, it can be expressed in the inertial frame,

$$\mathbf{v}_I = \mathbf{R}_{B\to I}\mathbf{v}_B,$$

using a rotation matrix:

$$\mathbf{R}_{B\to I} = \begin{bmatrix} C_\theta C_\psi & C_\psi S_\theta S_\phi - C_\phi S_\psi & C_\phi C_\psi S_\theta + S_\phi S_\psi \\ C_\theta S_\psi & S_\psi S_\theta S_\phi + C_\phi C_\psi & C_\phi S_\psi S_\theta - S_\phi C_\psi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix}, \tag{2.25}$$

where T_α, C_α and S_α are the $\tan(\alpha)$, $\cos(\alpha)$ and $\sin(\alpha)$ (for $\alpha = \phi, \theta$, or ψ).

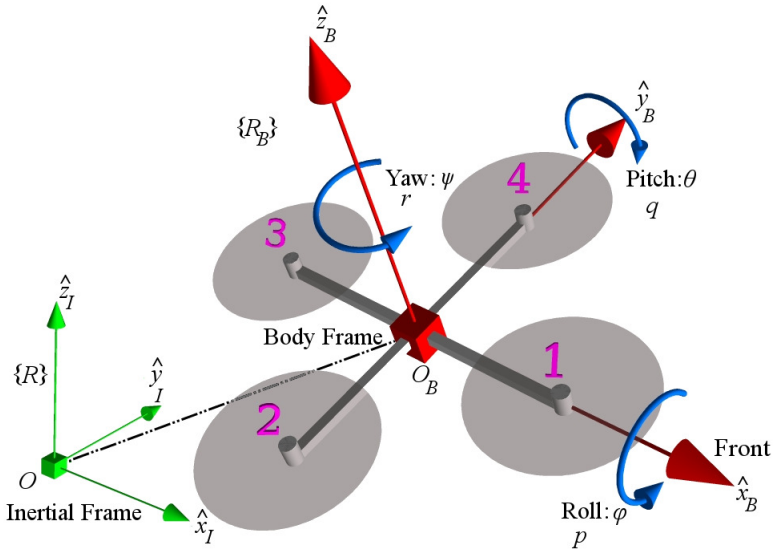


Figure 2.12: Quadrotor frames and variables.

The quadrotor velocity and Euler angle rates are represented by $\dot{\boldsymbol{\xi}}$ and $\dot{\boldsymbol{\eta}}$, respectively:

$$\dot{\boldsymbol{\xi}} = [\dot{x} \;\; \dot{y} \;\; \dot{z}]^T, \dot{\boldsymbol{\eta}} = [\dot{\phi} \;\; \dot{\theta} \;\; \dot{\psi}]^T. \tag{2.26}$$

Body angular velocities (body rates), which are the rate of change of roll, pitch and yaw, in the body frame, can be well measured by gyroscopic rate sensors. The following is the mapping between the body rates ($\boldsymbol{\omega} = [p \;\; q \;\; r]^T$) and Euler angle rates ($\dot{\boldsymbol{\eta}}$)

$$\boldsymbol{\omega} = \mathbf{M}(\phi, \theta, \psi)\dot{\boldsymbol{\eta}} \tag{2.27}$$

and the matrix-valued function $\mathbf{M}(.)$ is given by [22]:

$$\mathbf{M}(\phi, \theta, \psi) = \begin{bmatrix} 1 & 0 & -S_\theta \\ 0 & C_\phi & S_\phi C_\theta \\ 0 & -S_\phi & C_\phi C_\theta \end{bmatrix}. \tag{2.28}$$

The quadrotor state vector is formed as follows,

$$\begin{aligned} \mathbf{x}^T &= [x_1 \;\; x_2 \;\; x_3 \;\; x_4 \;\; x_5 \;\; x_6 \;\; x_7 \;\; x_8 \;\; x_9 \;\; x_{10} \;\; x_{11} \;\; x_{12}] \\ &= [\boldsymbol{\eta}^T \;\; \boldsymbol{\omega}^T \;\; \boldsymbol{\xi}^T \;\; \dot{\boldsymbol{\xi}}^T] \\ &= [\phi \;\; \theta \;\; \psi \;\; p \;\; q \;\; r \;\; x \;\; y \;\; z \;\; \dot{x} \;\; \dot{y} \;\; \dot{z}]. \end{aligned} \tag{2.29}$$

Using an Euler-Lagrange approach as in [23], the quadrotor dynamic model can be

written as,

$$\begin{aligned}\dot{\boldsymbol{\eta}} &= \mathbf{M}^{-1}(\phi,\theta,\psi)\boldsymbol{\omega},\\ \mathbb{I}\dot{\boldsymbol{\omega}} &= \boldsymbol{\tau}_B - \boldsymbol{\omega}\times(\mathbb{I}\boldsymbol{\omega}\times) - k_r\boldsymbol{\omega},\\ \ddot{\boldsymbol{\xi}} &= \mathbf{R}_{B\to I}\begin{bmatrix}0\\0\\\frac{\tau_f}{m}\end{bmatrix} - \frac{k_t}{m}\dot{\boldsymbol{\xi}},\end{aligned} \tag{2.30}$$

in which k_r and k_t are the rotational and translational drags, τ_f is the total thrust acting on the body, $\boldsymbol{\tau}_B = [\tau_p \ \tau_q \ \tau_r \ ]^\top$, is the torque around the body frame axis, '$\times$' denotes the vector cross-product and $\mathbb{I}$ represents the inertia matrix in the body frame,

$$\mathbb{I} = \begin{bmatrix} I_{xx} & 0 & 0\\ 0 & I_{yy} & 0\\ 0 & 0 & I_{zz}\end{bmatrix}. \tag{2.31}$$

The input to Equations (2.30) is the total thrust and torques, arranged in a vector, $\boldsymbol{\tau}$, as follows,

$$\boldsymbol{\tau} = \begin{bmatrix}\tau_f\\ \boldsymbol{\tau}_B\end{bmatrix} = \begin{bmatrix}\tau_f\\ \tau_p\\ \tau_q\\ \tau_r\end{bmatrix} \tag{2.32}$$

and are related to the forces generated by the rotors. For the plus configuration illustrated in Figure 2.10 this relation is given by,

$$\boldsymbol{\tau} = \begin{bmatrix} 1 & 1 & 1 & 1\\ 0 & -l & 0 & l\\ -l & 0 & l & 0\\ -d & d & -d & d\end{bmatrix}\begin{bmatrix}f_1\\ f_2\\ f_3\\ f_4\end{bmatrix}, \tag{2.33}$$

and for the X configuration illustrated in Figure 2.11 the relations is,

$$\boldsymbol{\tau} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ l & -l & l & -l \\ -l & l & -l & l \\ d & d & -d & -d \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}. \tag{2.34}$$

In Equations (2.33) and (2.34), l is the distance from the center of mass to the rotors, d is the ratio between the drag and the thrust coefficients of the blade, and f_i for $i = \{1, 2, 3, 4\}$ are the forces generated by the four rotors of the quadrotor. The force generated by the i^{th} motor rotating at a speed of ω_i is approximated by $f_i = (b_{mi} + \omega_i^2)/k_{mi}$, where b_{mi} and k_{mi} are the motor thrust coefficients.

Chapter 3

Multi-Step Prediction for Dynamic Systems

In this chapter, the main problem is formulated, i.e., the multi-step prediction of a dynamic system. As mentioned earlier, the RNNs are used to address this problem. To empirically show RNNs are a good candidate to address the problem at hand, an RNN based solution to a simplified version of the problem is proposed and formulated. The solution is applied to modeling a simulated quadrotor vehicle. The results are presented and compared with classic approaches using RNNs [63, 64].

3.1 Multi-Step Prediction Problem

Consider a dynamic system, $\mathcal{S}_n^m$, with m input and n output dimensions. The system input and output at a time instance, k, is denoted by $\mathbf{u}(k) \in \mathbb{R}^m$ and $\mathbf{y}(k) \in \mathbb{R}^n$, respectively. It is assumed that both input and output are measurable at all timesteps, k. Consider an

input sequence of length T starting at a time instance k_0+1, $\mathbf{U}(k_0+1,T) \in \mathbb{R}^m \times \mathbb{R}^T$,

$$\mathbf{U}(k_0+1,T) = \begin{bmatrix} \mathbf{u}(k_0+1) & \mathbf{u}(k_0+2) & \dots & \mathbf{u}(k_0+T) \end{bmatrix}. \tag{3.1}$$

The system response to this input is an output sequence denoted by $\mathbf{Y}(k_0+1,T) \in \mathbb{R}^n \times \mathbb{R}^T$,

$$\mathbf{Y}(k_0+1,T) = \begin{bmatrix} \mathbf{y}(k_0+1) & \mathbf{y}(k_0+2) & \dots & \mathbf{y}(k_0+T) \end{bmatrix}. \tag{3.2}$$

Definition: Given an input sequence $\mathbf{U}(k_0+1,T)$, the *multi-step prediction problem* seeks an accurate estimate of the system output, $\tilde{\mathbf{Y}}(k_0+1,T) \in \mathbb{R}^n \times \mathbb{R}^T$, over the same time-horizon, T,

$$\tilde{\mathbf{Y}}(k_0+1,T) = \begin{bmatrix} \tilde{\mathbf{y}}(k_0+1) & \tilde{\mathbf{y}}(k_0+2) & \dots & \tilde{\mathbf{y}}(k_0+T) \end{bmatrix}, \tag{3.3}$$

which minimizes the *prediction error*, that is, a measure of the error between the actual and predicted outputs. Usually, a Sum-of-Squared Errors measure (SSE) (or the mean of SSE, MSSE) is chosen,

$$L = \frac{1}{T} \sum_{k=k_0+1}^{k_0+T} \mathbf{e}(k)^\top \mathbf{e}(k), \tag{3.4}$$

$$\mathbf{e}(k) = \mathbf{y}(k) - \tilde{\mathbf{y}}(k). \tag{3.5}$$

△

3.1.1 Application Example

In Chapter 1, two application examples were briefly introduced which can benefit from a multi-step prediction model. To highlight the applicability of the multi-step prediction and

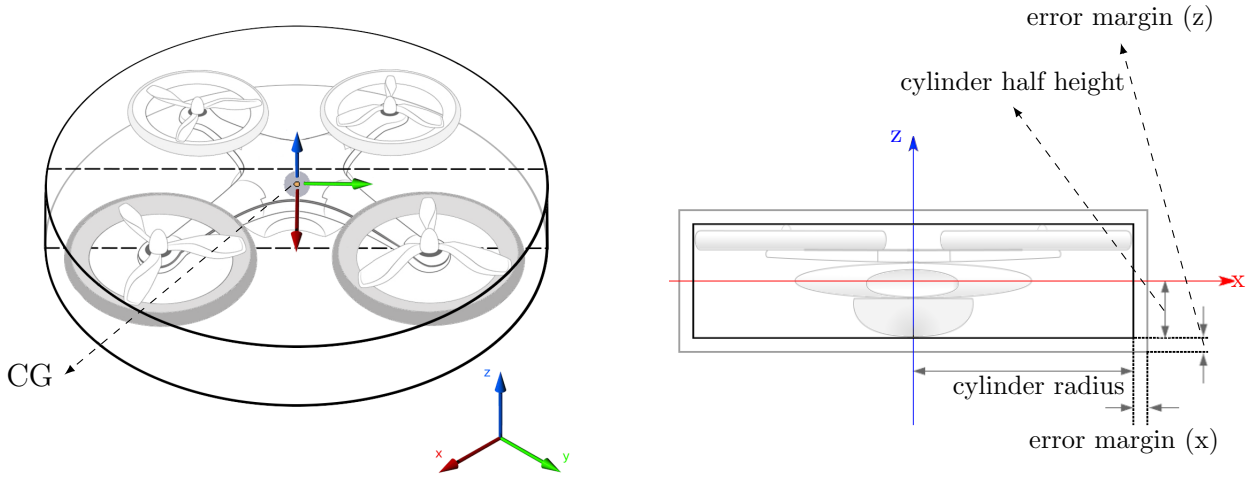
the importance of the prediction error, an application example is explained in this section which employs a multi-step prediction model.

To inspect a bridge, an autonomously flying quadrotor should follow a specific path close to the bridge structure for its sensors to inspect various joints and other parts of the structure. Assuming a detailed description of the bridge shape as well as the map of the environment surrounding it are available, a path can be devised through which the quadrotor should fly to achieve certain goals. The devised path consists of waypoints, each of which is determined by a desired vehicle state. The vehicle state is described by its position, velocity, attitude and body angular rates. The desired state values are sequentially given to an onboard controller which issues appropriate commands to the four motors to navigate the vehicle through the waypoints by minimizing the vehicle state errors. The state error is the difference between the measured state and the desired one at each waypoint. The vehicle position and velocity are measured by accurate GPS readings and the vehicle attitude and body rates are measured by an onboard Inertial Measurement Unit (IMU), both at 100Hz.

There are areas under the bridge where the GPS reading is not available. As the flying vehicle enters a GPS denied area, the position and velocity measurements are temporarily lost. However, to continue the mission safely, state predictions based on a model of the vehicle dynamics can be used to continue to pilot the vehicle in open loop until measurements are once again available. For this purpose, a multi-step prediction model can be used, which recursively updates the position and velocity from the motor speeds until the GPS signals are recovered. For simplicity it is assumed that the weather is calm and the quadrotor is not affected by the wind significantly.

Assume that the GPS readings correspond to the center of the gravity (CG) of the

quadrotor. Then the waypoints should be placed in such a way that there is enough space between the quadrotor frame and the bridge structure. For instance, if the quadrotor is hovering at a waypoint, then the Euclidean distance between this waypoint and the bridge structure should be larger than the largest distance of the points on the quadrotor frame (including all the sensors and rotor blades) from the quadrotor CG. However, a safety distance should also be considered to account for the errors in GPS readings as well as the controller transient and steady-state response. Design of such controller is out of the scope of this work and the GPS readings accuracy depends on the receiver being used. In the absence of the GPS reading and using a multi-step prediction model to update the position and velocity, the prediction error (Equation 3.4), dictates the clearance that each waypoint should meet in order to avoid hitting the bridge structure.



(a) A bounding cylinder for a quadrotor. The cylinder is centered at the vehicle CG and fully surrounds the vehicle body.

(b) The cross-section of the bounding cylinder. The bounding cylinder should be inflated to avoid collision. The amount of inflation is denoted by the error margins.

Figure 3.1: A bounding cylinder which surrounds the quadrotor body can be used for planning purposes to avoid collision with obstacles.

An illustration of the above discussion is depicted in Figure 3.1. In this figure, a cylinder

is centered at the quadrotor CG and surrounds the quadrotor. The cylinder's dimensions can be used in order to design collision-free waypoints. It is a common practice to *inflate* the dimensions of the surrounding shape (here it is a cylinder) to account for measurement errors and ensure a collision free navigation. A similar approach can also be taken to account for errors in the vehicle attitude measurements. When the position and velocity of the vehicle is being updated by the multi-step prediction model, the prediction error corresponds to the error margins depicted in Figure 3.1.

During the flight, while the GPS measurements are available, an Extended Kalman Filter (EKF) can be used to update the states of the vehicle and send them to the controller. The controller then calculates the state error and issues appropriate commands to minimize them. As soon as the GPS readings become inaccurate or unavailable, the multi-step model replaces the EKF and is used to update the states in an open-loop fashion, using the motor speeds only, until the GPS readings are recovered. The length of a safe continuation of the flight, both in terms of time and distance, depends on the multi-step prediction accuracy provided by the model over the prediction length. Throughout this thesis, several models will be developed and trained, however, the safe lengths resulting from the most accurate model will be provided as a proof of practicality of the trained model (Section 5.4).

3.2 RNNs as Sequence-to-Sequence Models

Throughout this thesis, RNNs will be playing a central role in addressing multi-step prediction problem, and therefore, are formulated for this purpose. Given an input sequence, $\mathbf{U}(k_0+1, T)$, an RNN produces an output sequence with the same number of elements. In this scenario, the RNN maps an input sequence to an output sequence with equal length.

Although it is not necessary that the input and output sequences have the same length, throughout this thesis it will be assumed they do unless otherwise mentioned. A sequence may represent a continuous signal sampled at a fixed frequency. In such a case, each occurrence of an element is referred to as a *time instance*. Similar to the description of $\mathcal{S}_n^m$ in Section 3.1, an RNN with m inputs and n outputs, $\mathcal{R}_n^m$, is a dynamic system. At each time instance, k, feeding in the input element $\mathbf{u}(k)$ causes the RNN to evolve through two major steps:

1. state update,

$$\mathbf{x}(k|\boldsymbol{\theta}) = \mathbf{f}\Big(\mathbf{x}(k-1|\boldsymbol{\theta}), \mathbf{u}(k)\Big), \tag{3.6}$$

2. output generation.

$$\tilde{\mathbf{y}}(k|\boldsymbol{\theta}) = \mathbf{g}\Big(\mathbf{x}(k|\boldsymbol{\theta}), \mathbf{u}(k)\Big), \tag{3.7}$$

where the RNN output is denoted by $\tilde{\mathbf{y}}(k)$. The vector $\boldsymbol{\theta} \in \mathbb{R}^q$ encompasses the network weights whose size, q, depends on the RNN architecture. The functions $\mathbf{f}(.)$ and $\mathbf{g}(.)$ are defined either explicitly, e.g., RMLPs - Equations (2.10), or implicitly, e.g., LSTMs - Equations (2.24), as described in Chapter 2.

Using RNNs to address the multi-step prediction problem, we seek an RNN which, given an input sequence $\mathbf{U}(k_0+1, T)$, produces an output sequence $\tilde{\mathbf{Y}}(k_0+1, T|\boldsymbol{\theta})$ which minimizes the Mean SSE (MSSE) *loss* over the prediction interval $[k_0+1, k_0+T]$,

$$L(\boldsymbol{\theta}) = \frac{1}{T}\sum_{k=k_0+1}^{k_0+T} \mathbf{e}(k|\boldsymbol{\theta})^\top \mathbf{e}(k|\boldsymbol{\theta}) \tag{3.8}$$

$$\mathbf{e}(k|\boldsymbol{\theta}) = \mathbf{y}(k) - \tilde{\mathbf{y}}(k|\boldsymbol{\theta}). \tag{3.9}$$

where $\mathbf{y}(k)$ is the system output at time $k \in [k_0{+}1, k_0{+}T]$ to the input $\mathbf{u}(k) \in \mathbf{U}(k_0{+}1, T)$. Therefore, the solution to the multi-step prediction problem is an RNN which minimizes L for all possible input-output sequences,

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \Big(L(\boldsymbol{\theta}) \Big). \tag{3.10}$$

Such an RNN will be referred to as a *predictor*.

The optimization in (3.10) is not practically possible because there are infinite input-output sequences. In practice, a dataset is collected by measuring the system input and output in a variety of cases. The dataset is comprised of time-series samples in a form of input-output tuples,

$$\mathcal{D} = \Big\{ s_i = \Big(\mathbf{U}_i(T), \mathbf{Y}_i(T) \Big) \Big\}, \;\; i = 1, ..., n_{\mathcal{D}}. \tag{3.11}$$

where T indicates the length of the trajectories and there are a total number of $n_{\mathcal{D}}$ samples in the dataset $\mathcal{D}$. Throughout this thesis, it is assumed that all of the samples have the same length.

Definition: A *complete* dataset is a dataset in which samples encompass all of the information needed to reconstruct the system input and state trajectories over T.

$\triangle$

Example: Since position is the integral of velocity, and velocity is bounded in a physical system, using velocity measurements is more convenient for normalization purposes. Also, Euler angles are the integral of Euler rates which can be obtained using vehicle body rates (refer to Sections 2.7 and 4.4.1). Therefore, if each sample in a quadrotor dataset

encompasses the four motor speeds as input and velocity and body rates as the output over T, then it is complete.

$\triangle$

Having a complete dataset, a numerical optimization is carried out to find a minimum of the total prediction loss,

$$L_{pred}(\boldsymbol{\theta}) = \frac{1}{n_{\mathcal{D}}} \sum_{i=1}^{n_{\mathcal{D}}} L_i(\boldsymbol{\theta}) = \frac{1}{T n_{\mathcal{D}}} \sum_{i=1}^{n_{\mathcal{D}}} \sum_{k=k_0+1}^{k_0+T} \mathbf{e}_i(k|\boldsymbol{\theta})^\top \mathbf{e}_i(k|\boldsymbol{\theta}). \tag{3.12}$$

where L_i is the loss due to the prediction error resulted from sample s_i. For detailed discussions refer to [39] and [95].

3.3 Multi-Layer Fully Connected RNNs

The framework described in this section is a generalization to RMLPs. It is a modular RNN with multiple locally recurrent layers and connections between all layers, both forward and backward. As a proof of concept, a Single-Input-Single-Output (SISO) nonlinear system which models the altitude dynamics of a quadrotor, is modeled using this framework. Through simulation it will be shown that in modeling this SISO system, the proposed architecture outperforms NARX-MLP and regular RMLP in terms of number of parameters, computational time and number of training samples. Then, it will be demonstrated that the Multi-Layer Fully Connected RNN (MLFC-RNN) is also capable of modelling the quadrotor as a Multi-Input-Multi-Output (MIMO) system.

3.3.1 The MLFC framework

An MLFC-RNN[1] consists of layers which are locally recurrent, denoted by $G_l, l = 1, \ldots, N$, where N is the number of layers. An example of a three layer ($N = 3$) MLFC-RNN is illustrated in Fig. 3.2 . In fact, each G_l is a dynamic MIMO system with m_l inputs and n_l outputs. The equations governing the dynamics of G_l, similar to (2.10), are as follows:

$$\begin{aligned} \mathbf{x}_l(k) &= \mathbf{A}_l \mathbf{y}_l(k-1) + \mathbf{B}_l \mathbf{u}_l(k) + \mathbf{b}_l \\ \mathbf{y}_l(k) &= \mathbf{f}_l\Big(\mathbf{x}_l(k)\Big) \end{aligned} \quad , \tag{3.13}$$

where $\mathbf{x}_l(k) \in \mathbb{R}^{n_l}$ is the state of the layer, $\mathbf{y}_l(k) \in \mathbb{R}^{n_l}$ is the output of the layer, $\mathbf{u}_l(k) \in \mathbb{R}^{m_l}$ is the input to the layer, $\mathbf{A}_l \in \mathbb{R}^{n_l} \times \mathbb{R}^{n_l}$ is the feedback weight matrix, $\mathbf{B}_l \in \mathbb{R}^{n_l} \times \mathbb{R}^{m_l}$ is the input weight matrix, $\mathbf{b}_l \in \mathbb{R}^{n_l}$ is a bias weight vector, $\mathbf{f}_l(.)$ is the layer activation function, n_l is the number of the neurons inside the layer and finally m_l is the number of input signals to the layer.

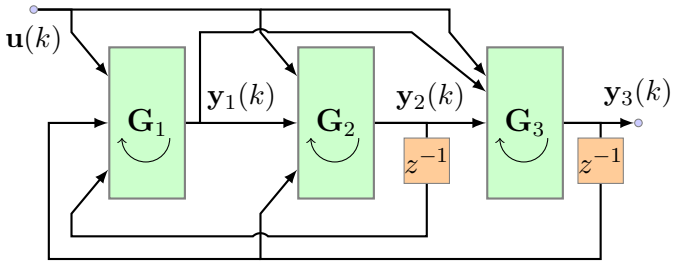


Figure 3.2: A 3 layers MLFC-RNN, with output $\tilde{\mathbf{y}}(k)$ equal to $\mathbf{y}_3(k)$.

The vectorized form of the weights inside the layer G_l is referred to by $\mathbf{p}_l^\top = [\mathbf{B}_l(:,1)^\top \ \ldots \ \mathbf{B}_l(:,m_l)^\top \ \mathbf{A}_l(:,1)^\top \ \ldots \ \mathbf{A}_l(:,n_l)^\top \ \mathbf{b}_l^\top]$, where $\mathbf{A}_l(:,j)^\top$ is the transpose of the j^{th} column of $\mathbf{A}_l$ and similarly for $\mathbf{B}_l$. The weight vector, $\mathbf{p}_l$, is in $\mathbb{R}^{q_l}$, where $q_l = n_l(m_l + n_l + 1)$ is the number of all weights inside G_l.

[1] In [63] and [64] this framework is called a Modular-Deep-Recurrent-Neural-Network (MODERNN).

The states and outputs of an MLFC-RNN can be updated in either a parallel or a serial fashion. In the parallel fashion all layers are updated at once. In this case, the input to each layer becomes,

$$\mathbf{u}_l(k) = \begin{bmatrix} \mathbf{u}^\top(k) & \mathbf{y}_1^\top(k-1) & \dots & \mathbf{y}_{l-1}^\top(k-1) & \mathbf{y}_{l+1}^\top(k-1) & \dots & \mathbf{y}_N^\top(k-1) \end{bmatrix}^\top, \tag{3.14}$$

and as all the inputs are formed, then the states (and outputs) are updated at the same time. The serial method updates each layer as the signals flow from input towards the output layer:

$$\mathbf{u}_l(k) = \begin{bmatrix} \mathbf{u}^\top(k) & \mathbf{y}_1^\top(k) & \dots & \mathbf{y}_{l-1}^\top(k) & \mathbf{y}_{l+1}^\top(k-1) & \dots & \mathbf{y}_N^\top(k-1) \end{bmatrix}^\top. \tag{3.15}$$

The parallel fashion treats all of the layers in the MLFC-RNN as a single layer, hence, it is a reformulation of RMLP. However, the sequential update taking place within MLFC-RNN in the serial fashion is a novel approach and implicitly provides an internal *time-constant* for the network which is equal to the number of layers, N. In this work, the MLFC-RNNs will be used in a serial update fashion. In either case,

$$m_l = m + \sum_{\substack{j=1 \\ j \neq l}}^{N} n_j \quad . \tag{3.16}$$

In this work, it is assumed that the N^{th} layer provides the network output as well, i.e., $\tilde{\mathbf{y}}(k) = \mathbf{y}_N(k)$. For later reference, a network with N layers is represented by a function

that maps the input sequence $\mathbf{U}(k_0, T)$ to the output sequence $\tilde{\mathbf{Y}}(k_0, T)$:

$$\tilde{\mathbf{Y}}(k_0, T) = \mathbf{\Omega}_N\Big(\mathbf{U}(k_0, T)\Big). \tag{3.17}$$

3.3.2 Network Jacobians for MLFC-RNN

In this section, a modular method to derive the network Jacobians is presented. It has been reported that for different architectures, obtaining network output derivatives is a time consuming process [61]. MLFC-RNN is an attempt to address this problem and speed up the process of designing new architectures.

To calculate the network Jacobians, let us form $\mathbf{p} \in \mathbb{R}^q$ which is a vector encompassing all the weights inside an MLFC-RNN network, i.e., $\mathbf{p}^\top = \begin{bmatrix}\mathbf{p}_1^\top & \dots & \mathbf{p}_N^\top\end{bmatrix}$. The number of all weights inside this networks is given by q:

$$q = \sum_{l=1}^{N} q_l, \quad \mathbf{p}_l \in \mathbb{R}^{q_l}.$$

Considering an arbitrary layer, G_l, the Jacobian update rule is written as follows:

$$\mathbf{J}_l^y(k) = \frac{\partial \mathbf{y}_l(k)}{\partial \mathbf{p}} = \operatorname{diag}\left[\mathbf{f}_l'\Big(\mathbf{x}_l(k)\Big)\right]\Big(\mathbf{A}_l\mathbf{J}_l^y(k-1) + \mathbf{B}_l\mathbf{J}_l^u(k) + \mathbf{\Gamma}_l(k)\Big). \tag{3.18}$$

In (3.18), $\mathbf{J}_l^y(k) \in \mathbb{R}^{n_l} \times \mathbb{R}^q$ is the Jacobian of the outputs of the layer G_l at time k and $\mathbf{J}_l^u(k) \in \mathbb{R}^{m_l} \times \mathbb{R}^q$ is the Jacobian of the inputs to that layer. The matrix $\mathbf{\Gamma}_l(k)$ corresponds to the derivatives of the layer weights with respect to the network weights,

$$\mathbf{\Gamma}_l(k) = \frac{\partial \mathbf{A}_l}{\partial \mathbf{p}}\mathbf{y}_l(k-1) + \frac{\partial \mathbf{B}_l}{\partial \mathbf{p}}\mathbf{u}_l(k) + \frac{\partial \mathbf{b}_l}{\partial \mathbf{p}}. \tag{3.19}$$

Note that Equation (3.18) is a recursive update rule for $\mathbf{J}_l^y(k)$. Therefore, an initial value is needed at time $k = 0$. In this chapter, it is assumed that the training starts from a stationary point; an assumption to be relaxed in the next chapter. Therefore the trivial zero initial condition for this matrix is assumed. Next, two matrices are formed: $\mathbf{J}_l^u(k)$ and $\mathbf{\Gamma}_l(k)$. As the independent inputs are not dependent on the network weights, the first m rows of $\mathbf{J}_l^u(k)$ are zero. Recalling the serial update fashion and Equation (3.15) we have,

$$\mathbf{J}_l^u(k) = \begin{bmatrix} \mathbf{0}_{m\times q}^\top & \mathbf{J}_1^{y\top}(k) & \dots & \mathbf{J}_{l-1}^{y\ \top}(k) & \mathbf{J}_{l+1}^{y\ \top}(k-1) & \dots & \mathbf{J}_N^{y\ \top}(k-1) \end{bmatrix}^\top. \tag{3.20}$$

The remaining elements inside the Jacobian $\mathbf{J}_l^u(k)$ are already calculated and so this term can be determined recursively.

To modularly formulate $\mathbf{\Gamma}_l(k)$, first let us define another matrix $\mathbf{\Lambda}_l(k)$ for each layer, G_l, as follows:

$$\mathbf{\Lambda}_l(k) = \frac{\partial \mathbf{A}_l}{\partial \mathbf{p}_l}\mathbf{y}_l(k-1) + \frac{\partial \mathbf{B}_l}{\partial \mathbf{p}_l}\mathbf{u}_l(k) + \frac{\partial \mathbf{b}_l}{\partial \mathbf{p}_l}. \tag{3.21}$$

In other words, $\mathbf{\Lambda}_l(k)$ plays the same role as $\mathbf{\Gamma}_l(k)$ but when the derivative is taken with respect to local weights only. Note, $\mathbf{\Lambda}_l(k)$ is in $\mathbb{R}^{n_l}\times\mathbb{R}^{q_l}$. With some algebraic manipulation, the term $\mathbf{\Lambda}_l(k)$ can be presented in the following form:

$$\mathbf{\Lambda}_l(k) = \begin{bmatrix} \mathbf{\Lambda}_{l,1}(k) & \mathbf{\Lambda}_{l,2}(k) & \mathbf{I}_{n_l} \end{bmatrix}_{n_l\times q_l}, \tag{3.22a}$$

$$\mathbf{\Lambda}_{l,1}(k) = \begin{bmatrix} y_{l,1}(k-1)\mathbf{I}_{n_l} & \dots & y_{l,n_l}(k-1)\mathbf{I}_{n_l} \end{bmatrix}, \tag{3.22b}$$

$$\mathbf{\Lambda}_{l,2}(k) = \begin{bmatrix} u_{l,1}(k)\mathbf{I}_{n_l} & \dots & u_{l,m_l}(k)\mathbf{I}_{n_l} \end{bmatrix}, \tag{3.22c}$$

and $u_{l,1}(k)$ refers to the first element inside $\mathbf{u}_l(k)$, and so on.

Since $\mathbf{\Lambda}_l(k)$ can be calculated for all layers, it is possible to form $\mathbf{\Gamma}_l(k)$:

$$\mathbf{\Gamma}_l(k) = \begin{bmatrix} \mathbf{0}_{n_l \times a} & \mathbf{\Lambda}_l(k) & \mathbf{0}_{n_l \times b} \end{bmatrix}. \tag{3.23}$$

For Equation (3.23), we define

$$a = \sum_{j=1}^{l-1} q_j \quad , \quad b = \sum_{j=l+1}^{N} q_j \tag{3.24}$$

Having $\mathbf{\Gamma}_l(k)$ defined, Equation (3.18) can be calculated and update the Jacobian recursively.

3.3.3 A Learning Algorithm for Training MLFC-RNN

As described in Chapter 2, RTRL uses a recursively updated gradient. The truncated BPTT gives us the gradients over a time horizon. MLFC-RNN, as described earlier, modularizes the calculation of the derivatives update rule. Therefore, the gradients are already calculated and can be used to construct a Jacobian by using a series of gradient values over a time horizon.

Let us consider the problem of modeling a quadrotor. In this problem the input(s) and output(s) correspond to the variables of interest of a quadrotor vehicle flying some trajectory over time. To devise the learning algorithm and for the sake of simplicity, let us assume the input and output are both scalar. That is, one training sample is represented by s,

$$s = \Big(\mathbf{u}(k_0 + 1, T), \mathbf{y}(k_0 + 1, T) \Big), \tag{3.25}$$

where

$$\mathbf{u}(k_0+1,T) = [u(k_0+1)\ u(k_0+2)\ \ldots\ u(k_0+T)\ ]^\top \in \mathbb{R}^T,$$

is the input time-series of the sample s and similarly for the output time-series, $\mathbf{y}(k_0+1,T)$.

As discussed earlier, an SSE cost function is adopted. To solve the SSE optimization problem, the Levenberg-Marquardt Method (LMM) is employed which has been frequently used and reported to be quite efficient [30]. For the sake of simplicity, let us assume $k_0 = 0$, then the cost function is:

$$L = 0.5 \sum_{k=1}^{T} \Big(\tilde{y}(k) - y(k)\Big)^2 = 0.5\mathbf{e}^\top\mathbf{e}, \tag{3.26}$$

where the error vector $\mathbf{e}$ is defined over the input-output sequence length:

$$\mathbf{e}^\top = \Big[e(1) \quad e(2) \quad \ldots \quad e(T)\Big],\ e(k) = \tilde{y}(k) - y(k),\ \text{for } k = 1,\ldots,T, \tag{3.27}$$

and the network output at time instance k is denoted by $\tilde{y}(k)$.

The LMM is essentially a second-order optimization method with variable step size which approximates the Hessian of the error vector, (3.27), with $\mathbf{J}^\top\mathbf{J}$, where $\mathbf{J}$ is the Jacobian of $\mathbf{e}$ [48, 60]. At each training iteration, the update rule in LMM is given by,

$$\Delta\mathbf{p} = -(\mathbf{J}^\top\mathbf{J} + \lambda\mathbf{I})^{-1}\mathbf{J}^\top\mathbf{e}, \tag{3.28}$$

where λ is a damping parameter. There are a number of methods to update λ [88]. However, the method proposed by Marquardt [60] is used in here.

In Equation (3.28), $\mathbf{J}$ is the Jacobian of the cost function over the time horizon , T,

given by,

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e(1)}{\partial p_1} & \cdots & \frac{\partial e(1)}{\partial p_q} \\ \vdots & \vdots & \vdots \\ \frac{\partial e(T)}{\partial p_1} & \cdots & \frac{\partial e(T)}{\partial p_q} \end{bmatrix}_{T \times q} = \begin{bmatrix} \left(\frac{\partial e(1)}{\partial \mathbf{p}} \right)^\top \\ \vdots \\ \left(\frac{\partial e(T)}{\partial \mathbf{p}} \right)^\top \end{bmatrix} = \frac{\partial \mathbf{e}}{\partial \mathbf{p}}. \tag{3.29}$$

The proposed training method divides the optimization into four nested loops. The outer most loop, o_1, handles the choice of training and validation samples. The two middle nested loops, o_2 and o_3, perform the LM optimization over the selected training set with n_v-fold cross-validation. The inner most loop, o_4, modifies the parameter vector $\mathbf{p}$ using Equation (3.28).

Assume a set of samples are available which is divided into two sets, a training set, $\mathcal{D}$, to be used for learning and a test set, $\mathcal{G}$, to test the generalization capability of the network. Also, assume that the training dataset has $n_{\mathcal{D}}$ samples in it, each is an input-output trajectory segment with T time steps (for simplicity k_0 is dropped):

$$\mathcal{D} = \left\{ s_i = \left(\mathbf{u}_i(T), \mathbf{y}_i(T) \right) \right\}, \ i = 1, ..., n_{\mathcal{D}}. \tag{3.30}$$

The test set, $\mathcal{G}$, is similar to the training set, $\mathcal{D}$, in structure. At each iteration of o_1 a subset of $\mathcal{D}$, namely $\mathcal{D}_s$, is formed by randomly selecting $n_s = n_{tr} + n_v$ samples from $\mathcal{D}$,

$$\mathcal{D}_s \subset \mathcal{D}, |\mathcal{D}_s| = n_s, \tag{3.31}$$

where $|.|$ denotes the cardinality of a set. The number of trajectory segments that the network is simultaneously trained on is n_{tr}, and n_v is the number of trajectory segments that the network is validated on. Both constants should be kept small to reduce the

computational complexity of the training process. Then, at each iteration of o_2, the set $\mathcal{D}_s$ is divided into two sets: the training set, $\mathcal{D}_{tr}$, and the validation set, $\mathcal{D}_v$, so that $|\mathcal{D}_{tr}| = n_{tr}$ and $|\mathcal{D}_v| = n_v$.

Having the training and validation sets, the optimization starts in loop o_3 and continues until the validation fails, i.e., the error over validation set starts to increase. At each iteration of o_3, the network runs over the entire $\mathcal{D}_{tr}$, the network Jacobians are updated and collected, as described in Section 3.3.2, and the errors are calculated. Then the Jacobian is formed as described by (3.29). The weight update process is performed in loop o_4, where the network weights are updated using Equation (3.28) starting with an initial (usually small) λ_0. At each iteration of o_4, the error is checked and λ is accordingly updated. That is, if the error decreases, λ also decreases, by a constant factor, and if the error increases, λ increases by the same factor. The detailed steps of this algorithm are defined in Algorithm 1. Although specific choices are made in terms of stopping criteria, step size, etc., these choices are not a requirement of the algorithm and may easily be modified to apply the MLFC-RNN to other problem instances. It is worthwhile to mention that to assess the generalization performance of the network, one *stopping criterion* can be set as a function of generalization error which can be calculated every a few iterations on the test set, $\mathcal{G}$.

It is important to note that each weight update is performed using the Jacobian that is computed over a number of flights at once. That is, each training sample inside $\mathcal{D}_{tr}$ is fed to the network individually, but the training is carried out simultaneously over the entire set, $\mathcal{D}_{tr}$.

Algorithm 1 Learning Algorithm for MLFC-RNN

Require: The initial damping parameter λ_0
Require: Maximum value for damping parameter λ_{max}
  while (Stopping criteria not met) **do** ▷ loop o_1
    $\mathcal{D}_s \leftarrow$ From $\mathcal{D}$ randomly choose n_s samples
    for o_2=1 to $\left\lfloor \frac{n_s}{n_v} \right\rfloor$ **do** ▷ loop o_2
      $\mathcal{D}_{tr} \leftarrow$ From $\mathcal{D}_s$ randomly choose n_{tr} samples
      $\mathcal{D}_v \leftarrow D_s \backslash \mathcal{D}_{tr}$
      $\gamma_1 \leftarrow$ **True** ▷ Validation fail check
      $\lambda \leftarrow \lambda_0$
      while γ_1 **do** ▷ loop o_3
        $\mathbf{e}_{v,0} \leftarrow \text{MINIBATCH}(\mathcal{D}_v, \mathbf{p}, 0)$
        $[\mathbf{e}_{tr,0}, \mathbf{J}] \leftarrow \text{MINIBATCH}(\mathcal{D}_{tr}, \mathbf{p}, 1)$
        $\gamma_2 \leftarrow$ **True** ▷ Parameter update fail check
        while γ_2 **do** ▷ loop o_4
          $\Delta\mathbf{p} \leftarrow -(\mathbf{J}^\top\mathbf{J} + \lambda\mathbf{I})^{-1}\mathbf{J}^\top\mathbf{e}$
          $\mathbf{e}_{tr,1} \leftarrow \text{MINIBATCH}(\mathcal{D}_{tr}, \mathbf{p} + \Delta\mathbf{p}, 0)$
          if $\mathbf{e}_{tr,1}^\top\mathbf{e}_{tr,1} < \mathbf{e}_{tr,0}^\top\mathbf{e}_{tr,0}$ **then** ▷ Parameter update successful
            $\lambda \leftarrow \lambda \times \frac{2}{3}$
            $\gamma_2 \leftarrow$ **False**
            $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$
          else ▷ Parameter update fail
            $\lambda \leftarrow \lambda \times \frac{3}{2}$
            if $\lambda > \lambda_{max}$ **then**
              $\gamma_2 \leftarrow$ **False**
        $\mathbf{e}_{v,1} \leftarrow \text{MINIBATCH}(D_v, \mathbf{p}, 0)$
        if $\mathbf{e}_{v,1}^\top\mathbf{e}_{v,1} > \mathbf{e}_{v,0}^\top\mathbf{e}_{v,0}$ **then** ▷ Validation fails.
          $\gamma_1 \leftarrow$ **False**

---

function $[\mathbf{e}, \mathbf{J}]$=MINIBATCH($\mathcal{D}_{tr}, \mathbf{p}, \delta$) ▷ If δ is set to 1, calculate the Jacobian
  for $s = 1$ to $|\mathcal{D}_{tr}|$ **do**
    $\mathbf{y} \leftarrow \Omega_N(\mathbf{u}_s)$ ▷ Generate network output
    $\mathbf{e}_s \leftarrow [e(1) \quad \dots \quad e(T)]^\top$ ▷ eq. (3.27)
    if $\delta == 1$ **then**
      $\mathbf{J}_s \leftarrow \frac{\partial \mathbf{e}_s}{\partial \mathbf{p}}$ ▷ eq. (3.29)
  $\mathbf{e}^\top \leftarrow \left[\mathbf{e}_1^\top \quad \dots \quad \mathbf{e}_{|\mathcal{D}_{tr}|}^\top\right]$
  if $\delta == 1$ **then**
    $\mathbf{J}^\top \leftarrow \left[\mathbf{J}_1^\top \quad \dots \quad \mathbf{J}_{|\mathcal{D}_{tr}|}^\top\right]$

3.3.4 Simulation Results

The first simulation used to demonstrate the effectiveness of MLFC-RNN for learning dynamical systems focuses on the altitude of a quadrotor. Three different RNN structures are compared: MLFC-RNN, NARX and RMLP. It will be demonstrated that MLFC-RNN outperforms the other two. Thereafter, a full MIMO modelling of a simulated quadrotor using MLFC-RNN is presented.

3.3.5 Comparison between MLFC-RNN, RMLP and NARX-MLP

The model to generate the altitude data is given by

$$\begin{aligned} \ddot{z} &= \frac{1}{m}\left(k_t u^2(1+f_{ge}^2) - c_d\dot{z} - mg\right) + \eta \\ f_{ge} &= k_{ge}\frac{h_{ge} - \min(h_{ge}, z)}{h_{ge}}. \end{aligned} \tag{3.32}$$

In Equation (3.32), z is the altitude of the vehicle and u is the sum of all four motor speeds (in RPM), which relates directly to the thrust produced. The mass of the quadrotor is m, k_t is the thrust coefficient, c_d is the vertical drag coefficient and f_{ge} is a simple model for ground effect. The ground effect acts at altitudes lower than h_{ge}, and k_{ge} is the ground effect coefficient. Finally, η is a white noise. Table 3.1 lists the values used for data generation.

To generate a dataset having flyable trajectories, each altitude trajectory is created using an input consisting of a sum of 10 sinusoids having uniformly random frequencies picked in the range of $[1, 10]$ Hz. To capture the ground effect, altitude is varied in the range of $[0, 2]$ meters over the whole dataset and normalized afterwards. The dataset is

.

m	k_{ge}	h_{ge}	k_t	z_{max}
1 kg	0.5	1 m	1.95×10^{-5} Ns^2/rad	2 m

Table 3.1: Quadrotor parameters used in the simulations

collected using a $f_s = 100$ Hz sampling frequency. In Figure 3.3 one sample of the dataset with normalized values is illustrated.

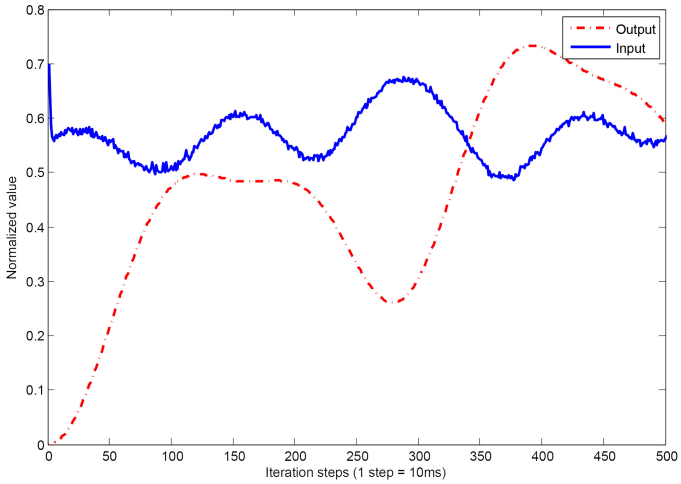


Figure 3.3: A generated data sample.

Having generated the dataset, the MLFC-RNN, RMLP and NARX-MLP architectures are trained on it. The experiments on series-parallel learning of NARX-MLP using the MATLAB toolbox failed when used in a closed loop for multi-step prediction. Therefore, a parallel-model NARX-MLP is implemented, the analytic derivatives of the network are derived and a training process based on the same method presented in Section 3.3.2 is implemented. The implementation was validated on examples given in [68].

The parameters under investigation are: the number of layers N, the number of hidden neurons in each layer h, and the size of the training set n_{tr}. For NARX-MLP, the number of

delays over input and output n_d is also investigated. Note that in multilayer cases, all layers are set to have the same number of neurons. Also, recall that the last layer has a linear output as is the case for function approximation applications. The result are summarized in Tables 3.2 and 3.3. In these tables, the training time corresponds to the codes run on an i7 Core machine. In general, it was observed that finding a working configuration becomes increasingly harder from MLFC-RNN to RMLP and to NARX-MLP. It was not possible to find a working architecture for NARX-MLP with $n_{tr} = 5, 10, 15$ and RMLP with $n_{tr} = 5, 10$, while MLFC-RNN can learn on $n_{tr} = 5$ and higher. For each of the reported cases the training was carried out 5 times, each time with a different weight (random) initialization in $[-1, 1]$ with small values. The best results are reported only. As a result of the ability to accurately learn the quadrotor altitude model with a MLFC-RNN with 48 weights, computation times were significantly improved over both RMLP and NARX-MLP, to approximately 0.5 hours from 5.5 and 7.5, respectively.

.

Network	Number of hidden neurons in each layer	Number of parameters	Number of training samples (n_{tr})	Number of delays (NARX-MLP only)	Mean error value	Training time (hours)
MLFC-RNN	5	48	5	NA	0.424	≈0.5
	5	48	10	NA	0.152	1.7
	5	48	20	NA	0.197	2.2
RMLP	5, 10, 20	42, 132, 462	5,10	NA	-	-
	10	132	15	NA	1.01	2.1
	5	42	20	NA	0.202	5.5
NARX-MLP	5, 10, 20	56, 111, 221	5,10,15	4	-	-
	5, 10, 20	66, 131, 261	5,10,15	5	-	-
	10	111	20	4	0.7175	6

Table 3.2: Comparison between 2 layers MLFC-RNN, RMLP and NARX-MLP.

Network	Number of hidden neurons in each layer	Number of parameters	Number of training samples (n_{tr})	Number of delays (NARX-MLP only)	Mean error value	Training time (hours)
MLFC-RNN	5	195	5	NA	0.322	1
	5	195	10	NA	0.612	1.2
	5	195	20	NA	0.088	3.5
RMLP	5,10,20	97,342,1282	5,10	NA	-	-
	5	97	15	NA	0.877	2.9
	5	97	20	NA	0.122	9.3
NARX-MLP	5,10,20	86,221,641	5,10,15	4	-	-
	5,10,20	96,241,681	5,10,15	5	-	-
	10	221	20	6	0.332	7.5

Table 3.3: Comparison between 3 layers MLFC-RNN, RMLP and NARX-MLP.

MIMO Modeling of a Quadrotor

In this section a full black-box model of a simulated quadrotor using a MLFC-RNN is trained. The simulator used to generate the training, validation and test data is slightly more complicated than having implemented the vehicle model (as described in Section 2.7) only. It also includes a quadratic model of the ground effect, models the rotors angular velocity and includes noise. Using this simulator, a set of 10000 flights, each having 3 seconds flight time and a sampling frequency equal to $f_s = 100$Hz were generated. For each flight, there are four desired trajectories to follow, three positions and the yaw motion. Each of these is a sum of ten sinusoids with random frequencies (less than 10Hz). To test the generalization capability of the trained networks, 100 flights were randomly picked and labeled as test dataset. The networks are not exposed with the test dataset during the training process.

As the positions can grow unboundedly, it is preferred to model the velocities of the vehicle. The inputs to the model are the individual motor speeds (in RPM) and outputs are the vehicle translational velocities and velocity in the yaw direction (refer to Figure 2.12

for a definition of yaw). Therefore, the MIMO system is 4×4. Modeling the velocity in the altitude and yaw directions is straight forward because they are directly related to the thrust generated by the four motors. However, the vehicle velocities in the x and y directions (v_x and v_y) are more challenging to model as they are results of the internal dynamics of the system. In order to reduce the computational load, the modelling of the velocities in the x and y directions were divided into two parts: from motor inputs to body angular rates, and from motor inputs and the predicted body rates to v_x and v_y. This is a reasonable division as the body angular rates can be measured using gyroscopic rate sensors, and as will be explained in the next chapter, using an Indoor Positioning System. The block diagram of the full MIMO model is illustrated in the Figure 3.4. In this figure, p, q and r are body angular velocities (refer to Section 2.7 for more details on p, q and r).

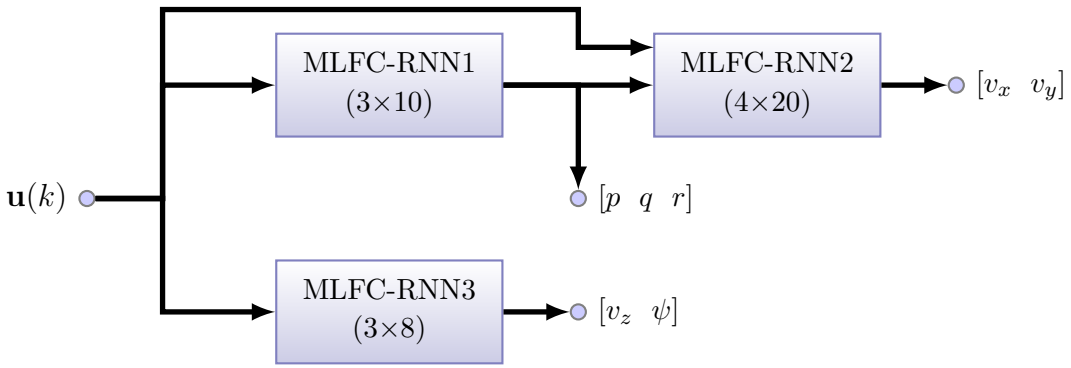


Figure 3.4: Modeling a MIMO quadrotor system.

One of the differences between a full MIMO modelling and the previously described SISO modelling is that the input now has many (in our case four) elements. Therefore, discovering the underlying function that governs the dynamics of the quadrotor from the samples is more challenging and requires more computational time. Additionally, as the underlying function becomes more complex, more nonlinearities should be placed inside the MLFC-RNN, that is, the number of neurons and layers should increase. As for other

form of neural networks, these number are determined using a trial-and-error procedure. In the Figure 3.4 the size of the employed MLFC-RNN is written on each block as $N \times h$, where N is the number of layers and h is the number of neurons inside each layer. These numbers were obtained after conducting a few experiments. Figures 3.5, 3.6 and 3.7 show the performance of the trained MLFC-RNNs on a sample test flight (from the test dataset). Each figure has two plots; the top one illustrates the predicted and the actual outputs in a multi-step prediction scenario and the bottom one plots the error. The values on the y-axis are normalized. On the prediction plots, the multi-step predicted outputs are plotted in solid lines and the actual values (simulator outputs) are plotted in dashed lines. Note that although the MLFC-RNNs are trained on 3 second flights, they can generalize beyond 3s (the figures show up to 5s). Note also that the inputs to MLFC-RNN2 is the output produced from MLFC-RNN1, as a result, the error is slightly larger.

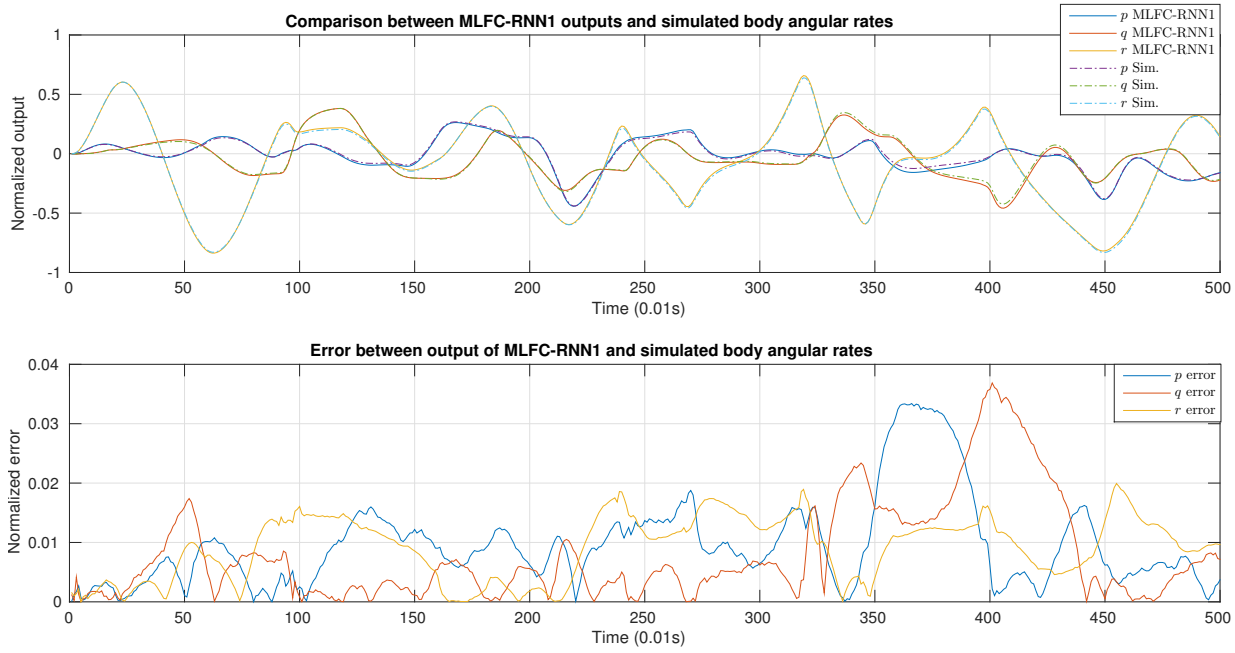


Figure 3.5: The generalization performance of the trained MLFC-RNN1.

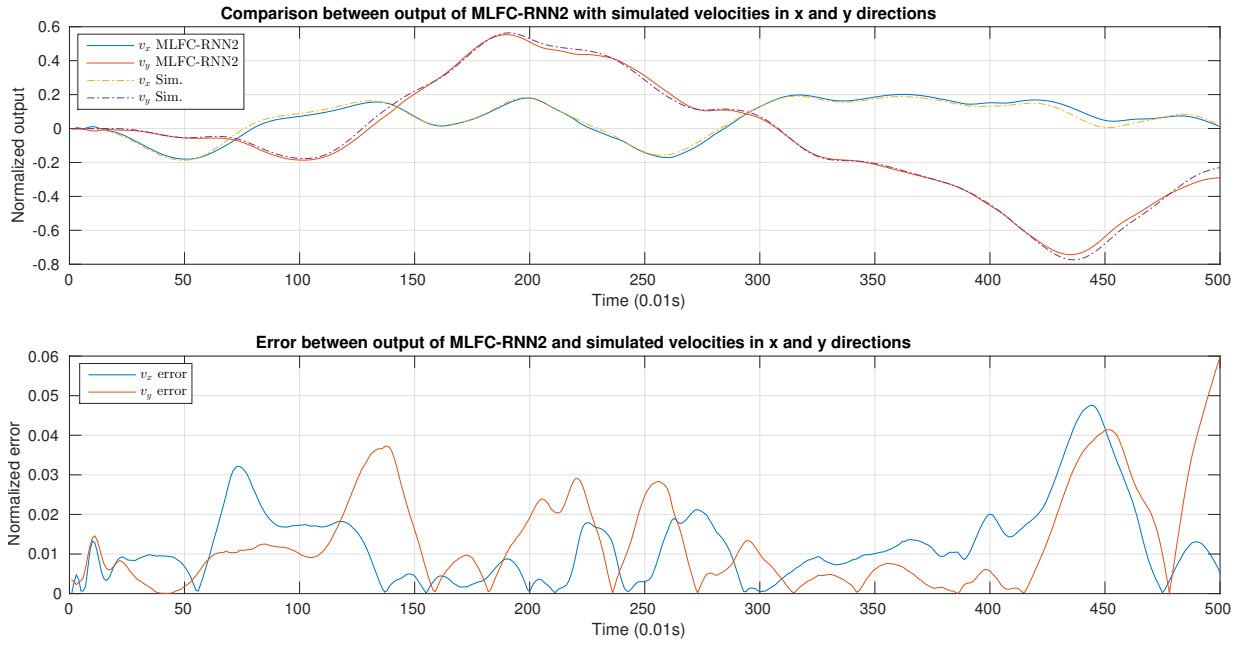


Figure 3.6: The generalization performance of the trained MLFC-RNN2.

As the networks become larger to identify the MIMO system, the computational time dramatically increases. On the same machine that the SISO results were obtained, on average training each of the MLFC-RNNs takes approximately 24 hours. The implementation in this section employed MATLAB with Parallel-Processing-Toolbox. In Chapter 4 and 5, where a real quadrotor is modeled, the proposed networks and algorithms are implemented in such a way to fully employ parallelism.

3.3.6 Effect of Forward Connections

In traditional multilayer networks, the error information starts to flow from the output layer through the middle layers back to the input layer in a sequential manner. The main reason for the structural vanishing/exploding gradients is that at each layer this information is

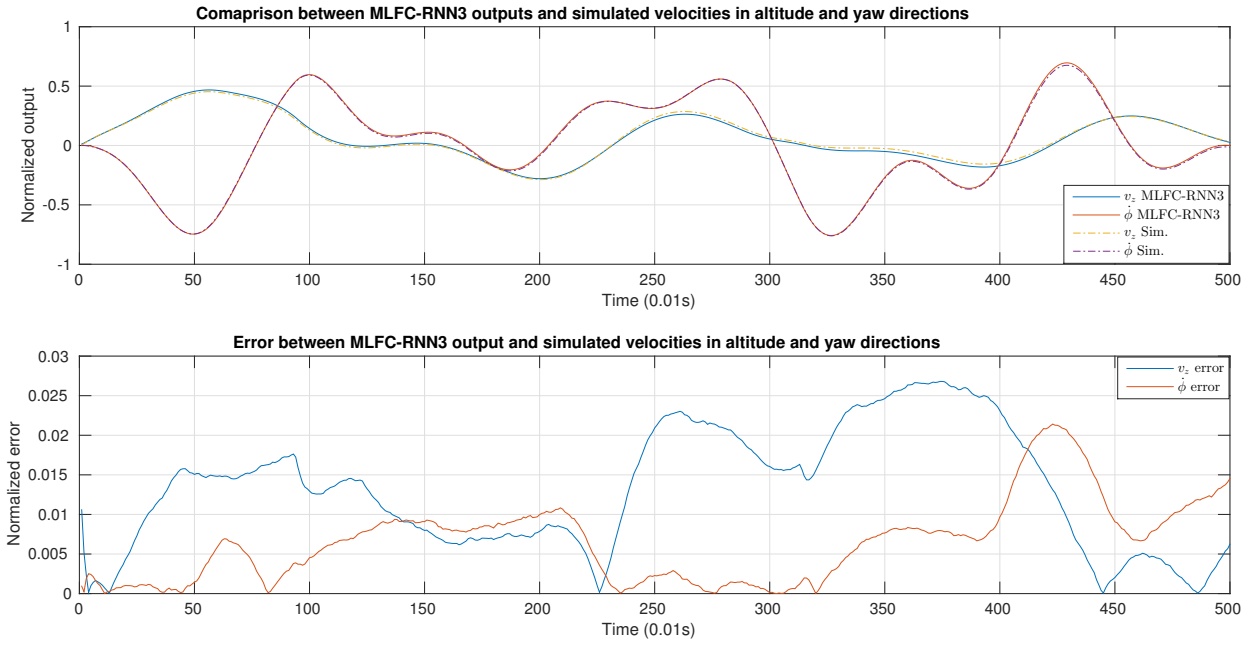


Figure 3.7: The generalization performance of the trained MLFC-RNN3.

either attenuated or amplified, and because network weights are usually initialized in the range $[-1, 1]$, attenuation occurs. However, in the version of MLFC-RNN with all inter-layer connections, the problem of vanishing/exploding gradient is less severe because direct connections from the output layer(s) to all other layers facilitate the error information transfer between layers. In fact, in an MLFC-RNN, the error information travels through a number of different paths, this multipath transfer of information contributes to faster and better learning and less attenuation of the error gradient in former layers.

In Figure 3.8, the evolution of the error gradient for each of the three networks studied in Section 3.3.5 is illustrated during 60 iterations of the training process. In this figure, the upper row corresponds to an MLFC-RNN with 3 layers and 5 neurons in each, the middle one corresponds to an RMLP with 3 layers and 5 neurons in each and the lower row

corresponds to a NARX-MLP with 2 layers, 10 hidden neurons and 4 delays over the input and the output. The left column corresponds to the initial gradient. Each column (from left to right) illustrates the gradients for the corresponding network at 15 iterations after the one on its left. The x-axis of each graph corresponds to the indexes of weights within the network sorted from input (index 1) towards output (the last index). For instance, the first 5 values on the x-axis in MLFC-RNN gradient graphs correspond to the weights connecting the input to the first layer of the MLFC-RNN ($\mathbf{A}_1$ in Equation (3.13)). As observed, the gradient in the RMLP and NARX-MLP networks is attenuated as it reaches the input weights, hence has less effect on the weights close to the input. Therefore, in the RMLP and NARX-MLP networks, the weight space is not being searched evenly in all directions. This is not the case for the MLFC-RNN. As the MLFC-RNN gradient has larger values for the weights close to the input, these weights are modified considerably and therefore the weight space is searched more evenly. The effect is likely the results of the forward connections that were explicitly added to the MLFC-RNN architecture.

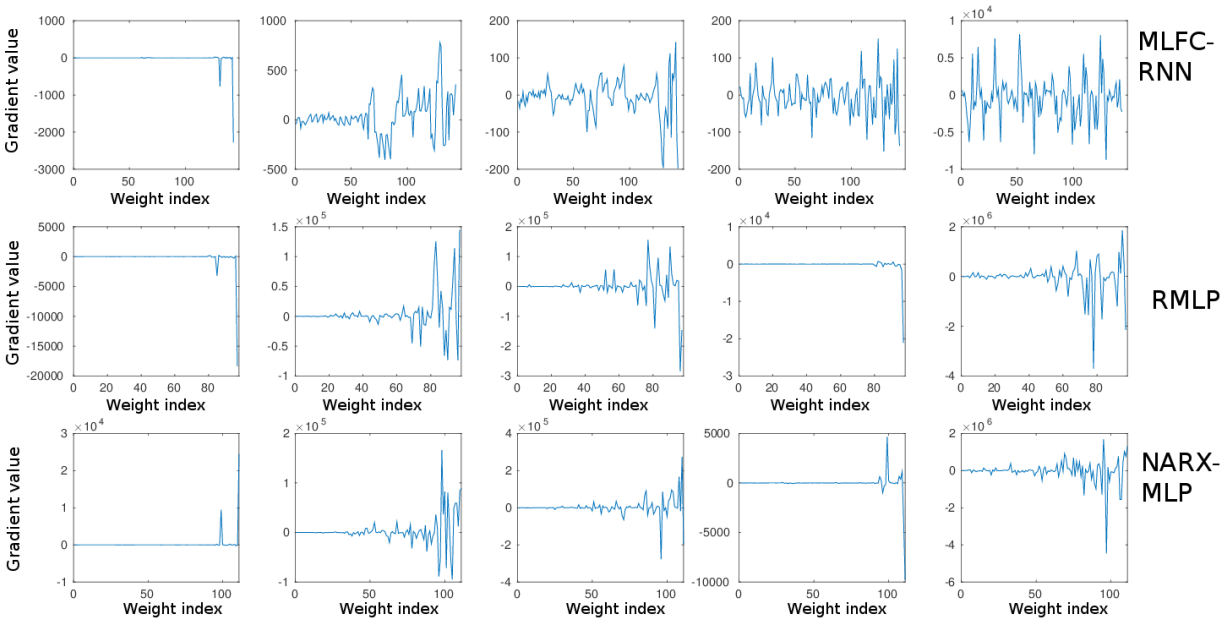


Figure 3.8: The gradient values during training. The x-axis correspond to the weight indices (integer values); the higher the x value, the shallower the weight. The y-axis corresponds to the gradient value of a weight. Ideally, we want to explore the weight space equivalently, that is, the magnitude of the gradient values should be almost uniformly distributed over the weights. However, as the learning progresses, for RMLP and NARX networks, it is observed that the gradient values for deeper weights (smaller x values) are significantly smaller in magnitude compared to the shallower weights (larger x). However, this behaviour is not present in MLFC-RNN, due to global connections.

Chapter 4

State Initialization in RNNs

In Chapter 3, RNNs were used to address the multi-step prediction problem in a special case where the system trajectories start from a zero initial condition. In the current chapter, this assumption is relaxed. A method is proposed to initialize the states of an RNN for multi-step prediction and is compared with the methods currently being used. Equipped with the proposed state initialization method, many RNN architectures are trained in a black-box modeling scheme on two experimental datasets which belong to two rotor-craft vehicles, a helicopter and a quadrotor. The datasets are described and the multi-step prediction performance of the RNNs are comprehensively studied [67, 66].

4.1 Motivation

The initial state of an RNN has a direct effect on the immediate (and transient) response of the network. If the feedback source is the RNN output, i.e., an output state, then it can be initialized using the actual measurements from the system. However, the neuron outputs

within the network do not have any meaningful physical property. Therefore, there is no physical measurement that can be used directly as the initial values for the hidden states.

The common approach to initialize the states of an RNN is to set them to zero (or random values) and then run the RNN for a number of steps until the effect of the initial state values washes out. This method is commonly referred to as the washout method [95, 39] and it suffers from two major drawbacks. First, the washout period, during which the predictions are too inaccurate to use, is hard to determine; it may vary for each input trajectory. Second, during the training, where the RNN may experience some unstable situations, the states may explode within the washout period. Additionally, in the multi-step prediction context, an RNN is sought whose output is readily applicable as an accurate approximation of the system output. Especially in control, the immediate response of the predictor is of great importance. Therefore, inaccurate early stage predictions are not acceptable.

As stated before, since modeling a dynamic system is a regression problem, the network output activation function, i.e., function $\mathbf{g}(.)$ in Equation (2.10), is an identity function which means that the network output is linearly dependent on the network states. This linear dependence is exploited to formulate the state initialization problem as an optimization problem.

4.2 State Initialization Problem Formulation

Like any other dynamic system, the solutions to the state and output trajectory of an RNN depend on the initial condition of the RNN [41]. In this section, the importance of the initial state of an RNN is outlined, and the *state initialization problem* is formally defined

and formulated for modeling dynamic systems with RNNs.

In Chapter 2, the RNN state vector was defined for the case where the feedback connections are from the output of the hidden neurons. In the discrete domain, feedback connections require some form of a memory buffer. Knowing the RNN architecture and the weight values, the output of an RNN depends solely on the buffer values. Therefore, it is reasonable and convenient to generalize the notion of states in an RNN to the *buffered* values. There are two types of states in an RNN: the output state, $\mathbf{x}_y(k) \in \mathbb{R}^n$, and the internal state, $\mathbf{x}_h(k) \in \mathbb{R}^h$. The feedback is sourced from the network output for the former, and from inside the network for the latter. They are arranged into the state vector as follows,

$$\mathbf{x}(k) = \begin{bmatrix} \mathbf{x}_y(k) \\ \mathbf{x}_h(k) \end{bmatrix} \in \mathbb{R}^s, \tag{4.1}$$

where s is the state count ($s = n + h$). Therefore, Equations (3.6) and (3.7) can be written as,

$$\mathbf{x}_y(k|\boldsymbol{\theta}) = \tilde{\mathbf{y}}(k-1|\boldsymbol{\theta}), \tag{4.2a}$$

$$\mathbf{x}_h(k|\boldsymbol{\theta}) = \mathbf{f}(\mathbf{x}(k-1|\boldsymbol{\theta}), \mathbf{u}(k)), \tag{4.2b}$$

$$\tilde{\mathbf{y}}(k|\boldsymbol{\theta}) = \mathbf{g}(\mathbf{x}(k|\boldsymbol{\theta}), \mathbf{u}(k)). \tag{4.2c}$$

Given an initial state, $\mathbf{x}(k_0)$, let us rewrite the RNN input-output equation as a sequence-to-sequence mapping (Section 3.2),

$$\tilde{\mathbf{Y}}(k_0+1, T) = \mathbf{F}\Big(\mathbf{x}(k_0), \mathbf{U}(k_0+1, T)\Big). \tag{4.3}$$

The function $\mathbf{F} : \mathbb{R}^s \times \mathbb{R}^m \times \mathbb{R}^T \rightarrow \mathbb{R}^n \times \mathbb{R}^T$ symbolizes the operations taking place sequentially inside the RNN by Equations (4.2).

From (4.2) and (4.3) it is evident that the initial state plays a key role in the immediate response of an RNN. Therefore, to have an accurate estimate one should properly *initialize* the RNN.

Definition: The RNN *state initialization problem* seeks to find initial values for the state vector of an RNN, $\mathbf{x}(k_0)$, such that the total prediction error loss (Equation (3.12)) is minimized. △

A trivial solution to this problem is zero, i.e., $\mathbf{x}(k_0) = 0$. However, since the zero state values correspond to a trivial equilibrium point of the RNNs studied in this thesis, the trivial solution requires that all of the output sequences in the dataset fulfill two conditions. Firstly, they should start from a stationary state of the system being modeled, and secondly, they should be transferable to the origin. These are restrictive assumptions. In the case of modeling an aerial vehicle, for instance, such restrictions mean that the sample trajectories should all start from a landing position. Acquiring such a dataset is not only cumbersome, but also the multi-step prediction which only predicts a take-off situation is overly restrictive.

In modeling and identification of dynamic systems using RNNs, the function that produces the network output, i.e., $\mathbf{g}(.)$ in Equation (4.2c), is the identity function. Therefore, a prediction generated at the time instance k can be written as,

$$\tilde{\mathbf{y}}(k) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k), \tag{4.4}$$

where $\mathbf{A} \in \mathbb{R}^n \times \mathbb{R}^s$ and $\mathbf{B} \in \mathbb{R}^n \times \mathbb{R}^m$ are the output layer weights (not including the

bias term) and their elements are parts of the weight vector $\boldsymbol{\theta}$, hence $\boldsymbol{\theta}$ is dropped.[1] Using (4.1) to expand (4.4) and letting $k = k_0$ we have,

$$\tilde{\mathbf{y}}(k_0) = \mathbf{A}_h \mathbf{x}_h(k_0) + \mathbf{A}_y \mathbf{x}_y(k_0) + \mathbf{B}\mathbf{u}(k_0), \tag{4.5a}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_h & \mathbf{A}_y \end{bmatrix}. \tag{4.5b}$$

According to (4.2), at each time instance $k \in [k_0 + 1, k_0 + T]$, the states $\mathbf{x}(k)$ must be updated prior to generating the output, $\tilde{\mathbf{y}}(k)$, which requires the knowledge of $\mathbf{x}(k-1)$. This sequential dependence can be followed back until the initial time, k_0, at which knowing the initial state, i.e., $\mathbf{x}(k_0)$, is necessary.

As described and proven in [40], the universal approximation property states that for an arbitrary $\epsilon > 0$ and an integer $0 < I < +\infty$, there exists an s and an RNN in the form of (4.2) with a proper initial condition $\mathbf{x}(k_0) \in \mathbb{R}^s$ such that,

$$\max_{0 \le k \le I} ||\mathbf{e}(k)|| < \epsilon, \tag{4.6}$$

where $\mathbf{e}(k)$ is the prediction error at time k and defined in (3.4). Consider the RNN* output at time k_0, $\tilde{\mathbf{y}}^*(k_0)$, whose prediction error, $\mathbf{e}^*(k_0)$, is infinitesimal, $\mathbf{e}^*(k_0) \ll 1$,

$$\tilde{\mathbf{y}}^*(k_0) = \mathbf{A}_h^* \mathbf{x}_h^*(k_0) + \mathbf{A}_y^* \mathbf{x}_y^*(k_0) + \mathbf{B}^* \mathbf{u}(k_0). \tag{4.7}$$

To expand (4.7), the prediction error $\mathbf{e}^*(k_0) = \mathbf{y}(k_0) - \tilde{\mathbf{y}}^*(k_0)$ and (4.2a) can be used,

$$\mathbf{y}(k_0) - \mathbf{e}^*(k_0) = \mathbf{A}_h^* \mathbf{x}_h^*(k_0) + \mathbf{A}_y^* \mathbf{y}(k_0 - 1) - \mathbf{A}_y^* \mathbf{e}^*(k_0 - 1) + \mathbf{B}^* \mathbf{u}(k_0).$$

[1] Note that the bias term is also dropped for the sake of notation simplicity.

Since $\mathbf{e}^*(k_0) \ll 1$ based on the universal approximation property,

$$\mathbf{A}_h^* \mathbf{x}_h^*(k_0 - 1) \approx \mathbf{c}^*, \tag{4.8}$$

where,

$$\mathbf{c}^* = \mathbf{y}(k_0) - \mathbf{A}_y^* \mathbf{y}(k_0 - 1) - \mathbf{B}^* \mathbf{u}(k_0).$$

Note that the weights are known at the time of state initialization. However, the optimal weights of RNN* are not necessarily known.

In the multi-step prediction problem, the main objective for which an RNN is trained, is to minimize the prediction error over the prediction horizon (Equation (3.12)). For a fixed network architecture, the prediction error depends on both the network weights, $\boldsymbol{\theta}$, and initial state values, $\mathbf{x}(k_0)$. Because the initial states of the *ideal* RNN, RNN*, should fulfill (4.8), for an arbitrary RNN, the state initialization problem for multi-step prediction can be written as the minimization of the following cost,

$$L_{si} = |\mathbf{A}_h \mathbf{x}_h(k_0) - \mathbf{c}|, \tag{4.9}$$

subject to $a \leq \mathbf{x}_h(k_0) \leq b$, which enforces the initial state values to remain within the range of the states' activation function. For example, if the states are generated by a tanh(.) then $a = -1, b = +1$.

Since the RNN state size, s, is usually much larger than the output size, n, directly optimizing L_{si} over the initial states $\mathbf{x}(k_0)$ leads to solving $L_{si} = 0$ which results in an infinite number of solutions over $\mathbf{x}(k_0)$. However, not all of the solutions are pertinent. As a matter of fact, the main goal in multi-step prediction is to minimize the total prediction

loss, L_{pred}, as in Equation (3.12). It is not clear which solutions, if any, of the equation $L_{si} = 0$ contributes best to minimizing the learning cost function.

Another approach to address the state initialization problem is to augment the initial states $\mathbf{x}_h(k)$ to the weight vector $\boldsymbol{\theta}$ and then train the network on the augmented weight vector [10]. Although this approach may address the state initialization problem during the training phase, it does not provide a mechanism to generate the initial state values *after* the training is finished. Clearly the initial state values cannot remain fixed and have to be set for each input sequence. Therefore, to properly address the state initialization problem, a *mechanism* should be devised in a way such that the initial state values are generated both during and after the training phase.

4.3 History-Based State Initialization

It is also possible to consider the set of RNN states as outputs of the network. Consider the ideal RNN, RNN*, where the network output defers from the desired output infinitesimally. Then we can write,

$$\mathbf{x}_y(k) = \mathbf{y}^*(k-1) \approx \mathbf{y}(k-1), \tag{4.10a}$$

$$\mathbf{x}_h(k) = \mathbf{f}(\mathbf{x}(k-1), \mathbf{u}(k)), \tag{4.10b}$$

where, $\mathbf{x}(k)$ is defined in equation (4.1). Equations (4.10) govern the dynamics of the RNN* *states*. To approximate this mapping, it is possible to employ NNs. In the following, a solution to the state initialization problem is proposed based on using an auxiliary neural network, which receives a short history of system input and output, to produce the

RNN *initial state values*. To avoid confusion, the auxiliary network will be referred to as the *initializer* and the RNN which performs the prediction as the *predictor*, as previously defined in Section 3.2.

The idea is to divide the data samples into two segments; the first segment is used as the input to the initializer, which initializes the predictor states, and the second one is used to train the whole network, i.e., the initializer-predictor pair. The number of steps in the prediction and initialization segment will be referred to as the prediction and initialization length denoted by T and τ respectively, as illustrated in Figure 4.1. The total length of the training sample is therefore $T_{tot} = \tau + T$.

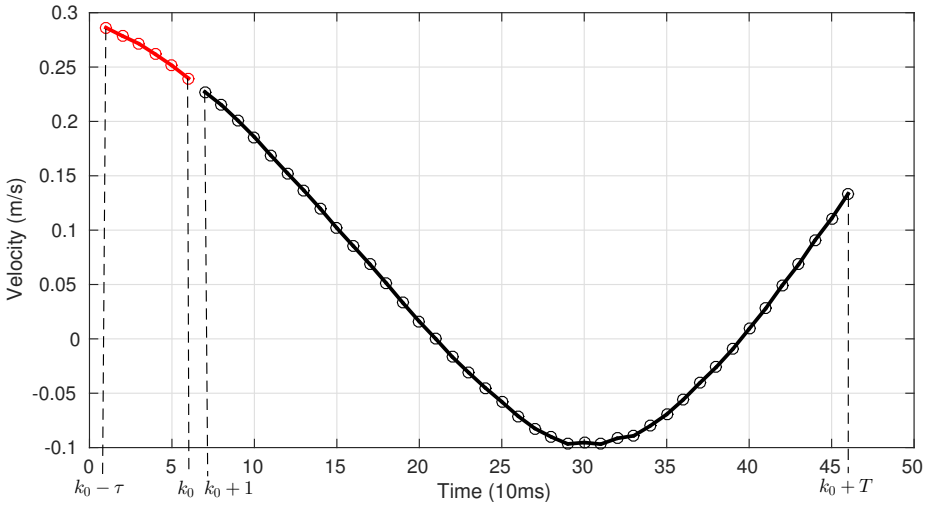


Figure 4.1: Dividing a data sample into initialization (red) and prediction (black) segments. Each small circle is one measurement from the continuous signal. In this figure, $\tau = 6$ and $T = 40$.

The desired values for the output of the initializer network, i.e., the initial RNN state values, are unknown. However, equation (4.9) proposes a penalty on the initializer network

output. Therefore, the initializer-predictor pair will be trained on the following cost,

$$L_{tot} = \alpha L_{pred} + \beta L_{si} = \alpha \frac{1}{TD} \sum_{i=1}^{D} \sum_{k=k_0+1}^{k_0+T} \mathbf{e}_i^\top(k)\mathbf{e}_i(k) + \beta |\mathbf{A}_h \mathbf{x}_h(k_0) - \mathbf{c}|, \tag{4.11}$$

where the prediction error, $\mathbf{e}_i(k)$ is defined in (3.8) and the coefficients α and β can be used to balance between the two costs.

MLP Initializer Network: An MLP, which receives a history of the measurements from the system and produces the predictor initial states, will be employed as the initializer network. Since a history of input and output measurements are used, this idea resembles the NARX-MLP in a serial-parallel (or teacher forced) manner[68],

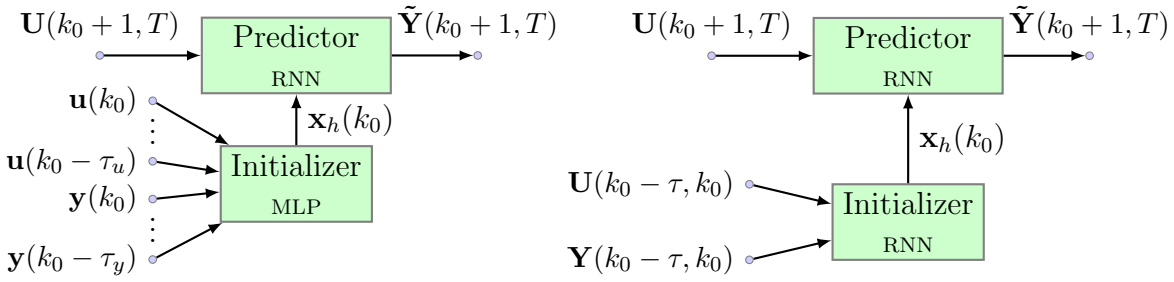
$$\mathbf{x}_h(k_0) = \boldsymbol{\zeta}\Big(\mathbf{u}(k_0-\tau_u), \mathbf{u}(k_0-\tau_u+1), ..., \mathbf{u}(k_0), \mathbf{y}(k_0-\tau_y), \mathbf{y}(k_0-\tau_y+1), ..., \mathbf{y}(k_0)\Big). \tag{4.12}$$

In Figure 4.2a the block diagram of this type of the initializer-predictor pair is illustrated. The underlying assumption in this approach is that the dynamics of the RNN states, defined in Equations (4.10), over a fixed period (i.e., the initialization length) can be approximated by a static function. The initializer network approximates that function.

Recurrent Initializer Network: Since the RNN states also possess dynamics, it is also viable to employ an RNN to model them. An RNN for the purpose of initialization can be a sequence-to-sequence model, $\boldsymbol{\xi}(.)$, which sequentially receives the system measurement history over the initialization length, τ, and produces an output sequence, $\tilde{\mathbf{Y}}_h(k_0-\tau, \tau)$,

$$\tilde{\mathbf{Y}}_h(k_0-\tau, \tau) = \boldsymbol{\xi}\Big(\mathbf{U}(k_0-\tau, \tau), \mathbf{Y}(k_0-\tau, \tau)\Big), \tag{4.13}$$

However, only the last element of the output sequence of the initializer network is used as



(a) MLP initializer

(b) RNN initializer

Figure 4.2: The two proposed initializer-predictor pairs for multi-step prediction.

the initial state value for the predictor,

$$\mathbf{x}_h(k_0) = \tilde{\mathbf{y}}_h(k_0). \tag{4.14}$$

Figure 4.2b illustrates the RNN-RNN initializer-predictor pair. The initial values of the initializer RNN states are set to zero. Clearly, the length of the initialization segment should be long enough to capture the dynamics of the predictor states.

4.4 Multi-Step Prediction of Two Real Rotorcraft Vehicles

In this section, the goal is to develop the full black-box model of two aerial vehicles, a helicopter and a quadrotor, for multi-step prediction using experimental data. For each vehicle, the black-box models map an input sequence to an output sequence which represents the vehicle's 6 Degrees-Of-Freedom (DOF) motion. For this purpose, the previously

discussed initializer-predictor pairs are trained on two datasets, an AscTec Pelican dataset[2] and the Stanford Helicopter dataset[3]. The AscTec Pelican dataset is gathered for the purpose of this work and consists of the quadrotor indoor flights in various regimes. The data collection procedure and setup are described in detail. The helicopter dataset belongs to the Stanford helicopter [1], and a brief summary of the information for the helicopter dataset is also presented.

In every black-box modeling effort, there are two main components involved: the class of functions implemented by the black-box method, and the dataset to infer the parameters of the black-box. In this thesis, the main focus is on the first component, that is, the black-box method. However, the generalization capability and prediction quality of the black-box models directly depend on the dataset representativeness. The representativeness of a dataset is difficult to quantify. In classic modeling and system identification, methods have been devised to produce inputs to excite all modes of a system so that the parameters of a first principles model can be identified. This approach, known as Persistent Excitation, employs inputs such as pseudo-random sequences, chirps, steps, ramps, etc., to generate datasets that capture all modes of the system[55]. The key point, however, is that the structure of the model is devised and fixed based on the first principles governing the dynamics of the system. The structure greatly influences the choice for the exciting input signals by providing information about the nature of the system. However, such knowledge of the model structure does not exist in the black-box modeling approach. To the best of the author's knowledge there is no universally accepted measure to evaluate the representativeness of a dataset in a black-box modeling problem.

Intuitively, the more the state space of a system is covered in a dataset, the more

[2]The quadrotor dataset is publicly available at: https://github.com/wavelab/pelican_dataset.
[3]The helicopter dataset is publicly available at: http://heli.stanford.edu/dataset/.

representative a dataset should be. Note that the entire state space of a dynamic system may not be accessible, mainly due to stability issues. Studying the representativeness of a dataset is beyond the scope of this work, however, distribution of the signals are presented to provide an idea of how representative the datasets are.

4.4.1 Quadrotor Dataset

The quadrotor dataset consists of time-series samples which are recovered from post-processing measurements of the states of a real flying quadrotor. The flights are carried out in $5 \times 5 \times 5$ meters indoor flight volume over the course of several days. The vehicle states are measured using onboard sensors as well as a precise motion capture system. The vehicle is operated by a human pilot in various flight regimes, such as hover, slight, moderate and aggressive manoeuvres.

Hardware

An Asctec[4] Pelican[5] quadrotor, illustrated in Figure 4.3, is employed to generate the flight data for the quadrotor dataset in this work. The vehicle dimension is 651 x 651 x 188 mm. It is equipped with a real-time autopilot board coupled with an onboard computer using an Intel Core i7 and 4GB of RAM.[6] The onboard computer runs Ubuntu 14.04 OS and communicates with the autopilot board via a UART connection. The Robotic Operating System (ROS) Indigo[7] software running a suitable ROS node[8] is used to collect the motor

[4] Ascending Technologies, is a part of Intel.
[5] http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-pelican/
[6] The onboard computer is AscTec Mastermind.
[7] http://wiki.ros.org/indigo
[8] http://wiki.ros.org/asctec_mav_framework

speeds and Inertial Measurement Unit (IMU) measurements. The vehicle is operated by an expert pilot using a Futaba T7C remote control.

Figure 4.3: The AscTec Pelican quadrotor used for collecting the quadrotor dataset.

The vehicle position and inertial orientation are measured at 100 Hz using a Vicon motion capture system, equipped with 16 Vantage cameras.[9] The position and orientation of the vehicle are instantaneously read by the Vantage cameras, looking at the IR reflective markers mounted on the vehicle, and sent to the Vicon server through a LAN communication (Figure 4.4). To avoid any wireless latency and/or packet drops, the measurements are logged on the Vicon server computer using the Vicon Tracker software version 3.3 which runs in the Microsoft Windows 10 OS. The Vicon system is calibrated before each data collection session to account for changes in environmental variables, such as room temperature, camera body temperature, etc. The data collection diagram is depicted in Figure 4.5.

[9] https://www.vicon.com/products/camera-systems

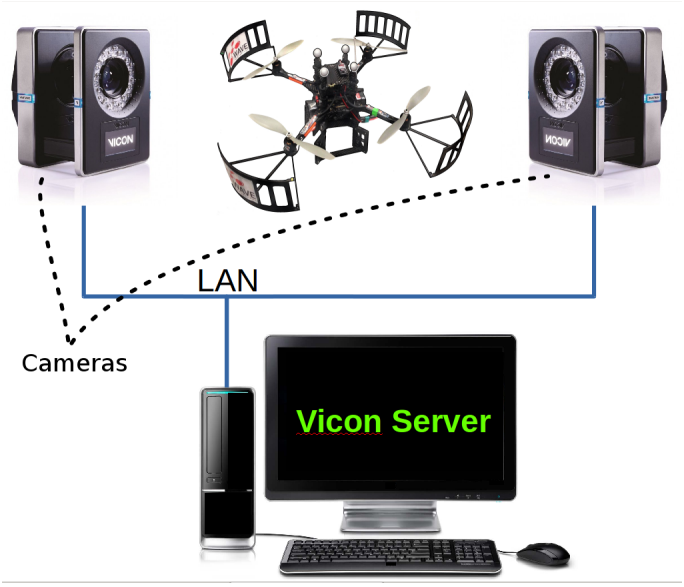


Figure 4.4: Vicon measurements of the quadrotor position and orientation.

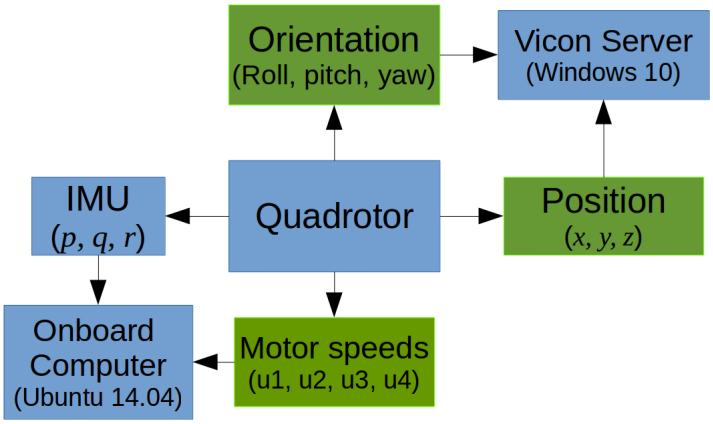


Figure 4.5: Communication block diagram for the quadrotor dataset collection.

Measurements

The logged measurements are listed in Table 4.1. The position and orientation (Euler angles) are measured in the inertial (Vicon) frame. The body rates, $\dot{\boldsymbol{\omega}}_g(k) = [p_g(k), q_g(k), r_g(k)]$, are measured in the quadrotor body frame, where the index g highlights the fact that the measurement is done by the gyroscopic sensors. All of the listed quantities are measured at 100Hz. After removing the landing sequences, the total recorded flight time is approximately 3 hours and 50 minutes, which in total corresponds to about 1.4 million samples per element.

Quantity	Unit	Source	Logged in
Motor speeds $\mathbf{u}(k) = [u_1(k), u_2(k), u_3(k), u_4(k)]$	Integer values in [0, 218]	AscTec autopilot board	Mastermind
Inertial position $\boldsymbol{\xi}(k) = [x(k), y(k), z(k)]$	mm, ±5mm accuracy	Vicon system	Vicon server
Inertial orientation, Euler angles $\boldsymbol{\eta}(k) = [\phi(k), \theta(k), \psi(k)]$	deg., ±0.1deg. accuracy	Vicon system	Vicon server
Body rates $\dot{\boldsymbol{\omega}}_g(k) = [p_g(k), q_g(k), r_g(k)]$	deg., ±300 deg/sec range	Pelican IMU	Mastermind

Table 4.1: Pelican measurements.

The ROS node provides two types of motor speed, the *commanded* and the *actual*. The actual motor speed is *estimated* by the AscTec autopilot board based on the pulse rate for the three phase excitation of the brushless motor. Both the commanded and actual speeds are available as integer values. It is possible to experimentally devise a mapping to RPM, however, the mapping is left to be learned by the NNs internally.

Because the position and heading can grow unboundedly, it is preferable to learn velocity and body rates. The velocity vector, $\dot{\boldsymbol{\xi}}(k) = [\dot{x}(k), \dot{y}(k), \dot{z}(k)]$, is obtained by taking the numerical derivative of the position vector, $\boldsymbol{\xi}(k)$. Since the IMU measurements are

extremely noisy, the Euler readings from the Vicon system, $\boldsymbol{\eta}(k)$, are converted to body rates, $\boldsymbol{\omega}(k)$. The Euler rates, $\dot{\boldsymbol{\eta}}(k) = [\dot{\phi}(k), \dot{\theta}(k), \dot{\psi}(k)]$, are obtained by taking the numerical derivative of the Euler angles and then are transferred to the body frame using the following equations (refer to Section 2.7 for further detail),

$$\boldsymbol{\omega}(k) = \mathbf{M}(\phi(k), \theta(k), \psi(k))\dot{\boldsymbol{\eta}}(k), \tag{4.15}$$

where the matrix-valued function $\mathbf{M}(.)$ is given by:

$$\mathbf{M}(\phi, \theta, \psi) = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 0 & \cos(\phi) & sin(\phi)\cos(\theta) \\ 0 & -\sin(\phi) & cos(\phi)\cos(\theta) \end{bmatrix}. \tag{4.16}$$

Although the IMU readings are not used for system modeling and identification, they are employed as a medium to adjust the time delays as described next.

Time synchronization

The Vicon server and the onboard computer run non-realtime operating systems (OS), which leads to delays and inconsistencies in the timestamps recorded with measurements on the two systems. There are three sources of delay: the onboard computer OS, the Vicon server OS and the ROS software. Perfectly synchronizing timestamps between various measurements requires sophisticated hardware and software solutions, which were not available in this work. However, it is possible to approximately align the separated measurements of the Vicon and onboard systems in time.

The IMU measurements and motor speeds are read by the autopilot board and received by the ROS node at the same time. Thus, they share the same timestamp. Aligning the IMU body rates with the Vicon converted body rates should fairly compensate for any time delays. The alignment is simply done by a cross-correlation between the two signals. Note that both the IMU and Vicon system provide measurements at the same frequency (100 Hz). The time synchronization process is as follows,

1. Calculate the numerical difference of the measured Euler angles (Euler rates).
2. Transform the Euler rates to the quadrotor body frame using equations (4.15) and (4.16).
3. Smooth the IMU body rate measurements to attenuate the noise.
4. Do a cross-correlation between the IMU body rates and the converted body rates.
5. Apply the calculated delay to align the converted body rates to the IMU body rates, hence to the motor speeds.

Post-processing

To attenuate noise, a smoothing filter is applied to all of the measurements with a window size of 5 samples. The filter is a local-regression which approximates the signal at each sample point by a 2^{nd} degree polynomial. For the actual motor speeds, a robust version of this filter is applied that assigns lower weight to outliers in the regression to reduce the effect of current spikes in the motor control units.[10] The models developed in this work map the actual motor speeds to body rates and translational velocity, hence, the quantities included in the quadrotor dataset for this work are,

[10]Refer to the MATLAB (The MathWorks Inc.) documentation for the smooth function.

- actual motor speeds, $\mathbf{u}(k) = [u_1(k), u_2(k), u_3(k), u_4(k)]$,
- velocity vector in inertial frame, $\dot{\boldsymbol{\xi}}(k) = [\dot{x}(k), \dot{y}(k), \dot{z}(k)]$,
- body rates, $\boldsymbol{\omega}(k) = [p(k), q(k), r(k)]$.

Distributions

The dataset consists of various flight regimes: hover, close to ground, light, moderate and aggressive manoeuvres in all directions, etc. Figures 4.6 and 4.7 illustrate the distribution of the measured signals and their rate of change. Figure 4.6 illustrates the actual motor speeds as well as the rates of change of the actual motor speeds. It can be observed that the rate of change distributions (plots on the right column) are all symmetric and fairly similar. The motors are of the same type, however, they are not necessarily identical. Also, the propellers have been changed many times throughout data acquisition. Therefore, it was not expected that the actual motor speeds have symmetric and similar distributions. In Figure 4.7, the distribution of the velocity, acceleration, body rates and body angular accelerations are illustrated. Noticeably, the distribution of the velocity and body rates are fairly symmetric and unbiased.

Based on the AscTec Pelican specifications [11], the maximum climb and air speed are 8 and 16 meters per second, respectively. However, because the flights were carried out in an indoor environment with limited space, the specified maximums were not achieved. The maximum values for the 6 DOFs are listed in Table 4.2.

[11] http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-pelican/

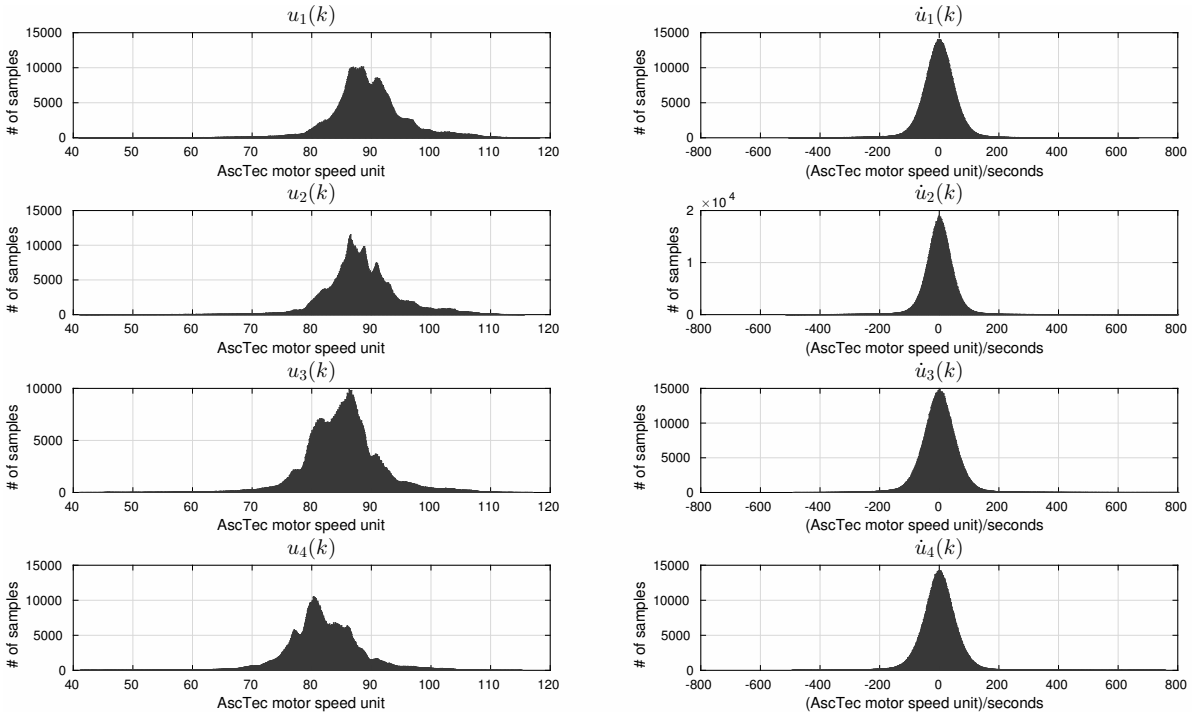


Figure 4.6: Distribution of the quadrotor motor speeds and their rate of change.

.

$u_i(k)\ i = 1,2,3,4$	$\dot{x}\ (m/s)$	$\dot{y}\ (m/s)$	$\dot{z}\ (m/s)$	$p\ (rad/s)$	$q\ (rad/s)$	$r\ (rad/s)$
120	3.9268	3.9721	5.8526	3.9116	3.8506	3.7902

Table 4.2: Maximum values for the Pelican measurements.

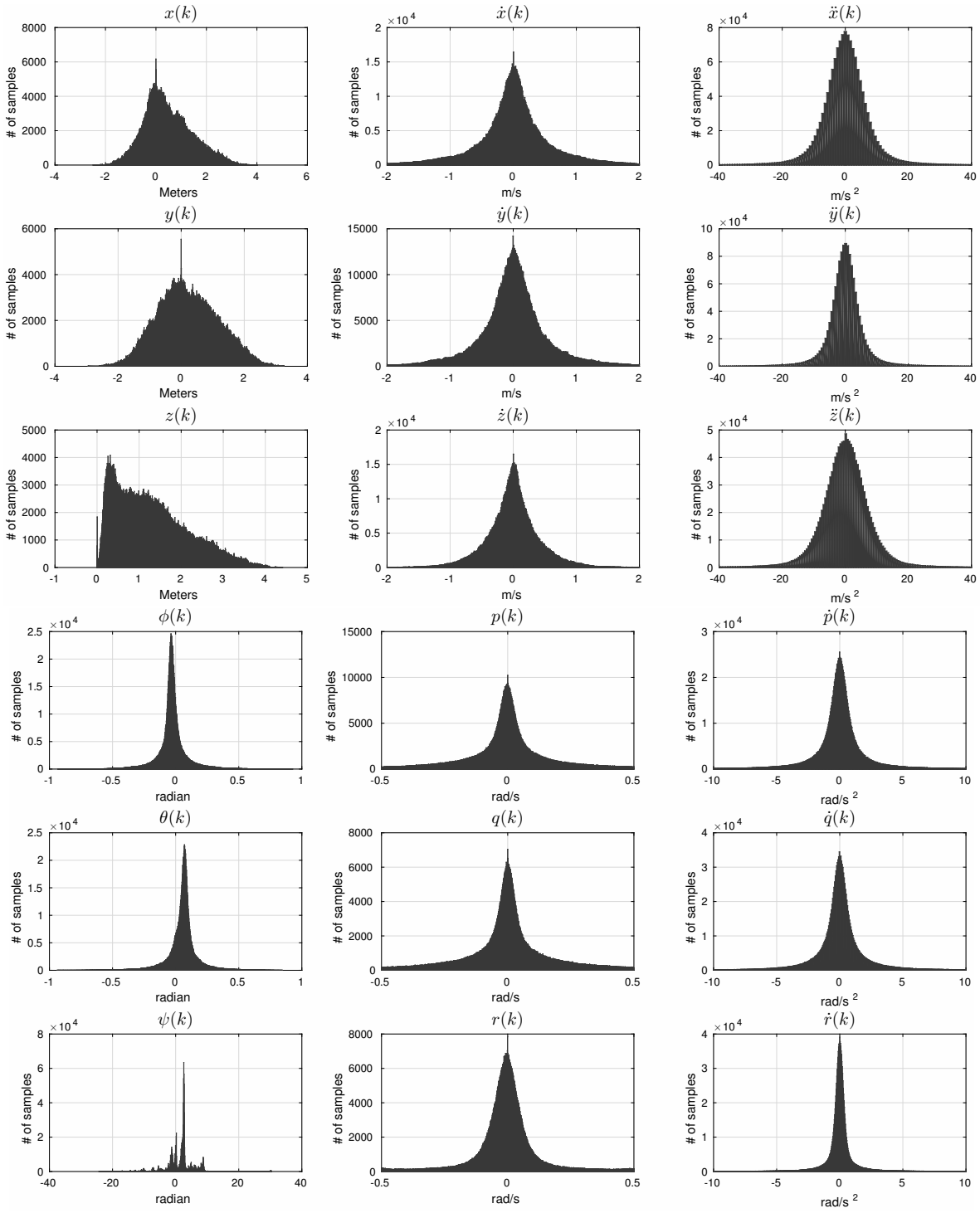


Figure 4.7: Distribution of the quadrotor data. Top: position, velocity and acceleration. Bottom: Euler angles, body rates and body rotational accelerations, in the body frame.

4.4.2 Helicopter Dataset

This dataset was collected in August 2008 as a part of research for Apprenticeship Learning [1] at Stanford University.[12] It has also been used for a single-step prediction system identification problem [78]. In this thesis, many architectures will be trained on this dataset in a multi-step prediction scenario. For this dataset, the flights are carried out in an outdoor environment, however, the dataset does not provide a wind measurement. The flight time is approximately 55 minutes and there are 335,258 samples for each quantity.

Hardware

According to the information provided with the Stanford helicopter dataset, the airframe is a Synergy N9, illustrated in Figure 4.8. The vehicle weight is 4.71 kg and its main propeller diameter is about 1.5 meter. It is equipped with a single-cylinder, two stroke engine (OS .91). The IMU sensor is Microstrain 3DMGX1[13]. The vehicle position is acquired using a ground-based vision system consisting of two cameras mounted at fixed locations on the field. The dataset provides the position and velocity as time-series signals.

Measurements

The motor speeds are not provided in this dataset. Instead, the *commands* from the remote controller in four directions are given; aileron, elevator, rudder and collective stick positions, which correspond to roll, pitch, yaw rate and total thrust, respectively. Their values are already normalized in $[-1, 1]$. In this thesis, the same notation as the quadrotor dataset, i.e., $u_i(k), i = 1, 2, 3, 4$, is used for the stick position.

[12] See http://heli.stanford.edu/index.html for details.
[13] http://www.microstrain.com/inertial/3DM-GX1

Figure 4.8: The Synergy N9 helicopter vehicle used in the Stanford helicopter dataset.

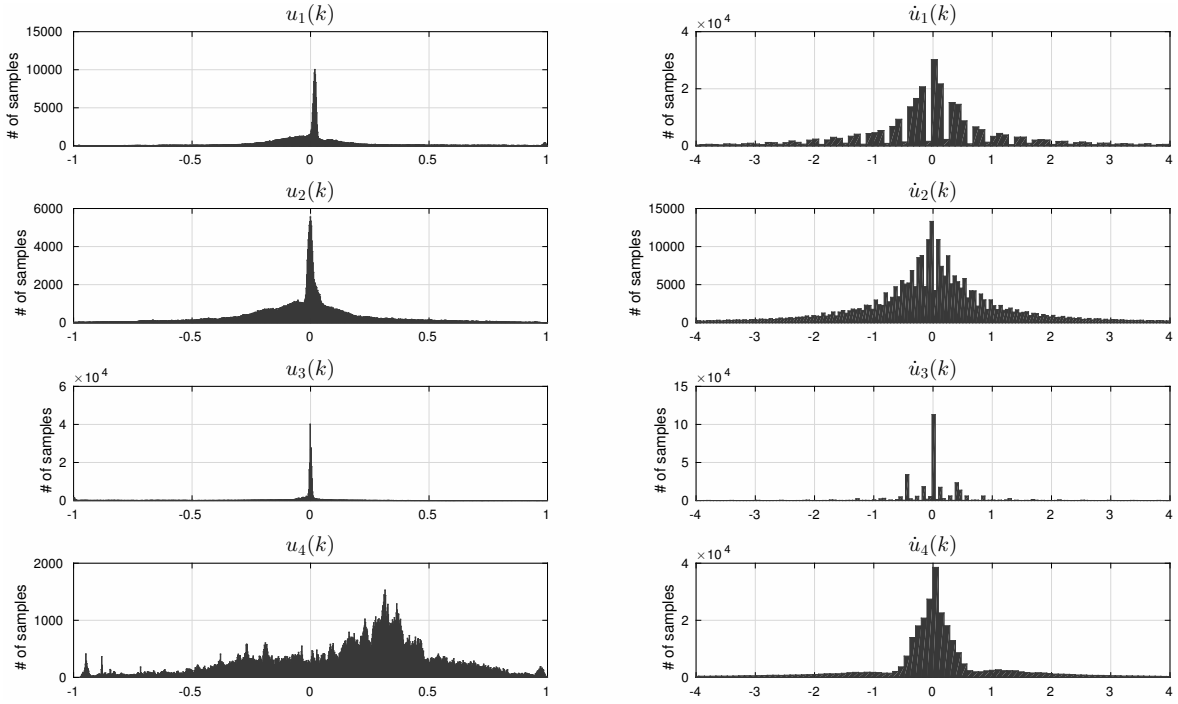


Figure 4.9: Distribution of the helicopter pilot commands (stick positions) and their rate of change.

The vehicle state measurements used for learning are the three inertial velocity and three angular rate components. The provided filtered values for the measurements are used. They are referred to by the same notation as for the quadrotor.

Distributions

Figures 4.9 and 4.10 illustrate the helicopter data distribution. Although the pilot command distributions are fairly symmetric, they are not evenly distributed, which are the result of quantization with fixed resolution integer commands. Also, the velocity and body rate distributions are narrow but heavy tailed, which is likely due to a significant portion of the dataset being in hover conditions.

As it can be seen, samples in the helicopter dataset are not well distributed. The dataset is relatively small (less than an hour flight time) and has been collected outdoors with no wind measurement. Therefore, it is not expected that the black-box model of the helicopter, based on this dataset, provides an accurate model suitable for control tasks. However, since all the learning scenarios converged during the experiments and because of its smaller size, the helicopter dataset is employed mainly for the purpose of comparing the washout with the history-based initialization method and evaluating the network sizes.

4.4.3 Learning Scenarios

For hyper-parameter optimization, assessment of various network architectures, and the performance of the proposed state initialization methods, the rotational and translational velocities are modeled separately. In this way, the problem size is divided by two and the training time is almost halved. The learning scenarios are summarized in Table 4.3.

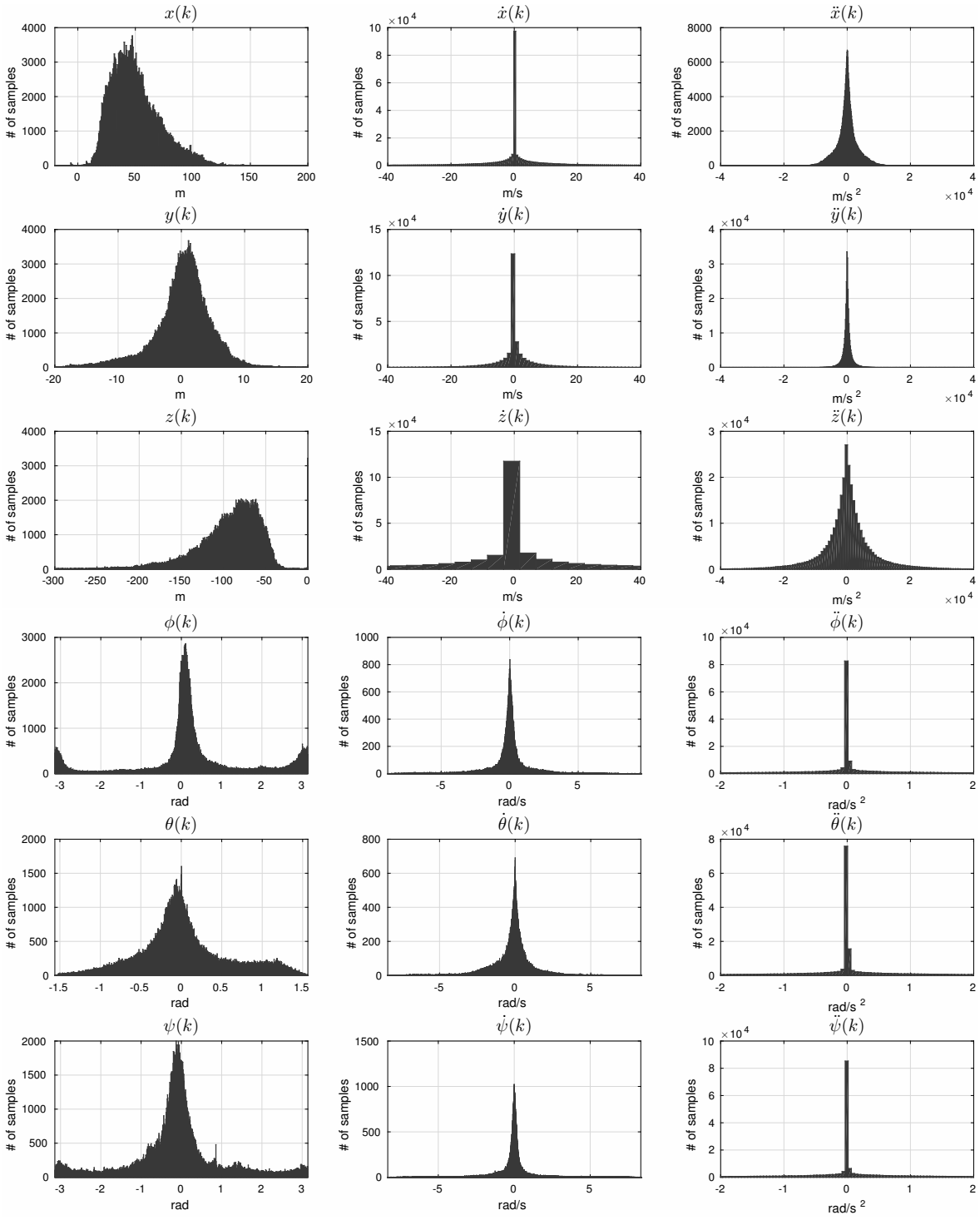


Figure 4.10: Distribution of the helicopter data. Top: position, velocity and acceleration in the inertial frame. Bottom: Euler angles, rates (angular velocity) and accelerations.

Dataset	Input	Output
Helicopter	Pilot commands $\mathbf{u}(k)$	Angular rates $\dot{\boldsymbol{\eta}}(k)$
	Pilot commands $\mathbf{u}(k)$	Velocity $\dot{\boldsymbol{\xi}}(k)$
Quadrotor	Motor speeds $\mathbf{u}(k)$	Body rates $\boldsymbol{\omega}(k)$
	Motor speeds $\mathbf{u}(k)$, Body rates $\boldsymbol{\omega}(k)$	Velocity $\dot{\boldsymbol{\xi}}(k)$

Table 4.3: Learning scenarios.

Each of the above scenarios is carried out for four sample lengths, $T_{tot} = 50, 100, 150$ and 200, which correspond to $0.5, 1.0, 1.5$ and 2.0 seconds, respectively. The initialization length (τ) is set to 10 steps throughout the scenarios, which results in a prediction length $T = T_{tot} - 10$ (see Figure 4.1). Therefore, the prediction lengths in seconds are $0.4, 0.9, 1.4$ and 1.9 seconds, respectively. Regardless of the scenarios, the input and output signals are always normalized by the maximum of the absolute value the signal can acquire.

Note that, except in the last scenario, the input and output dimensions are 4 and 3, respectively. In the conducted experiments for this work, learning the quadrotor translational velocity directly from the motor speeds always failed. One reason is that the vehicle orientation plays an important role in determining the translational velocity. However, learning the body rates alongside the velocity directly from the motor inputs frequently failed, too, as the velocity x and y components quickly start to overfit and negatively affect the other predicted outputs.

Instead of learning quadrotor translational velocity directly from the motor speeds, a two stage prediction approach is proposed. In the first stage, the body rates are predicted directly from the motor speeds. In the second stage, the velocity is predicted using the predicted body rates along with the motor speeds. Therefore, in the last scenario, the *actual* body rates, converted from the Euler angles measured by the Vicon system, are included as inputs to the network. The substitution of the predicted values with the actual

values is called *teacher forcing* [68]. After the networks are trained, the actual body rates are substituted by the predicted ones. Since this substitution is needed to employ the learned models in practice, using the predicted body rates to generate velocity prediction will be referred to as the *practical* mode. It will be seen that the error in the predicted body rates will drastically deteriorate the accuracy of the predicted velocity. In Chapter 5, this problem is circumvented with a grey-box approach.

To prepare the samples, each flight trajectory is first partitioned. Each partition is considered as one sample, as depicted by Equation (3.25). The partitions are allowed to overlap. In the experiments conducted for this thesis the overlap is 50%. The samples are then shuffled and divided into two sets, 60% training and 40% test. The dataset sizes are listed in Table 4.4. For each iteration, a batch of 100 training samples are randomly chosen from the training set and used for training. The training is carried out for a fixed number of iterations. Throughout the training, the validation error is calculated on the test dataset every few hundred iterations (100 or 1000). The model with the best validation error is picked. This method differs slightly from the standard early-stopping [27]. In the early-stopping method, as soon as the model starts to overfit, the training stops. However, in this method, the training does not stop after overfitting detection, but the model is stored. The search continues and occasionally a drop is observed after a slight increase in the test error which is lower than the last error before the start of overfitting.

4.4.4 Evaluation

For evaluation, the prediction errors and their distributions are studied. The prediction error in general is defined at each prediction step as in Equation (3.4). Three prediction

.

Prediction Length (samples):		$T = 40$ ($T_{tot} = 50$)	$T = 90$ ($T_{tot} = 100$)	$T = 140$ ($T_{tot} = 150$)	$T = 190$ ($T_{tot} = 200$)
Number of samples (quadrotor dataset)	Training set	33320	16659	11106	8329
	Test set	22212	11106	7404	5553
Number of samples (helicopter dataset)	Training set	8046	4023	2682	2011
	Test set	5364	2682	1788	1341

Table 4.4: Size of training and test datasets, over the given prediction lengths.

errors are used which correspond to the velocity, body rate and angular velocity vectors,

$$\begin{aligned}
\mathbf{e}_{\dot{\xi}}(k) =& [e_{\dot{x}}(k)\ e_{\dot{y}}(k)\ e_{\dot{z}}(k)],\ \text{measured in meters per second (m/s)} \\
\mathbf{e}_{\omega}(k) =& [e_{p}(k)\ e_{q}(k)\ e_{r}(k)],\ \text{measured in degrees per second (deg/s)} \\
\mathbf{e}_{\dot{\eta}}(k) =& [e_{\dot{\phi}}(k)\ e_{\dot{\theta}}(k)\ e_{\dot{\psi}}(k)],\ \text{measured in degrees per second (deg/s).}
\end{aligned} \tag{4.17}$$

The velocity error vector, $\mathbf{e}_{\dot{\xi}}(k)$, is calculated for both vehicles. However, the angular velocity error, $\mathbf{e}_{\dot{\eta}}(k)$, is only used for the helicopter and the body rate error, $\mathbf{e}_{\omega}(k)$, is only calculated for the quadrotor.

To study the prediction error distribution, two norms are used; $\bar{L}_1$ norm which is defined as,

$$||\mathbf{v}||_1 = \frac{1}{n}\sum_{i=1}^{n}|v_i|, \tag{4.18}$$

and the standard L_2 norm, defined as,

$$||\mathbf{v}||_2 = \sqrt{\sum_{i=1}^{n} v_i^2} \tag{4.19}$$

where in (4.18) and (4.19), the vector $\mathbf{v} \in \mathbb{R}^n$ is,

$$\mathbf{v} = [v_1 \ v_2 \ ... \ v_n]^\top.$$

The $\bar{L}_1$ norm defined here illustrates the mean of the error magnitude in any of the Euclidean directions while the L_2 norm provides information on the magnitude of the error vector in any direction. The $\bar{L}_1$ norm can be used to get more insight about the error on each component of the position vector.

To evaluate the performance of the networks based on their size (in terms of number of weights) the Root-Mean-Sum-of-Square-Error (RMSSE) value is used which is calculated over the entire prediction length and the test dataset,

$$RMSSE_{tot} = \sqrt{\frac{1}{Tn_{\mathcal{G}}} \sum_{i=1}^{n_{\mathcal{G}}} \sum_{k=\tau+1}^{T} \mathbf{e}_i^\top(k)\mathbf{e}_i(k)}, \tag{4.20}$$

where $n_{\mathcal{G}}$ is the size of the test dataset, $\mathcal{G}$.

4.4.5 Architectures and Implementation

The Google Tensorflow package in Python 2.7 is used for implementing and training the networks. The hardware used to implement and train these architectures are NVIDIA Titan X and Tesla K80 GPUs. It is important to mention that the optimization method employed throughout the experiments is ADAM [42] which is a first order method. Because the size of the implemented networks are very large, with the current state-of-the-art hardware it is not possible to benefit from a second order optimization method. Using a first order method, the number of iterations in training is chosen in the range of 300k to

600k. In this setting, training a network may take from half a day to two weeks, depending on the size of the network and the length and number of training samples.

The predictor network can be either an MLFC (Section 3.3), LSTM (Section 2.5) or LSTM with TDLs (Figure 2.7). The TDL size is 10, i.e., $t = 10$ in Figure 2.7, throughout the experiments in this thesis. Each of the predictors may be initialized in one of the three fashions: washout, with an MLP initializer or with an RNN initializer. In case of an RNN initializer, an LSTM with one layer of LSTM cells is employed. In order to refer to each configuration, the following notation is used:

[predictor]: [number of layers] × [size of each layer] - [initializer type]: [hidden layer size]×[initialization length]

For example, an LSTM predictor with 3 layers, each having 200 LSTM cells initialized by an MLP with 1000 neurons in the hidden layer and an initialization length of 10 is referred to by **LSTM: 3×200-MLP:1000×10**. As another example, an MLFC with 2 layers each having 100 neurons initialized by washout method for 5 steps is referred to by **MLFC: 2×100-Washout:5**.

4.5 Results

This section presents the results of black-box modeling of the two aforementioned aerial vehicles. First, the effect of the history-based initialization is studied and comparisons with the washout method are provided on small size networks. As the results will show, the history-based initialization methods provide more accurate immediate prediction and therefore, the rest of the experiments will employ them. Then, various architectures are

trained on the helicopter dataset and the networks with the best performance are studied further. Based on these results, a black-box model of the helicopter is presented. The architectures with the best performance on the helicopter dataset are trained on the quadrotor dataset and their prediction performance is presented and studied.

4.5.1 History-based Initialization vs. Washout

The first goal is to choose an RNN type along with a proper initialization method for further training. To this end, the performance of two RNN types, MLFCs and LSTMs, initialized by either history-based initialization or washout, are studied. To save training time, the networks are trained on three subsets of the helicopter dataset. Each dataset belongs to a Multi-Input-Single-Output (MISO) subsystem of the helicopter. It should be noted that for the experiments with history-based initialization method throughout this thesis, the balance coefficients in (4.11), α and β, are chosen to be 1.

Figures 4.11 and 4.12 compare the performance of two small size RNNs on predicting the angular rates of the helicopter directly from the pilot commands. The following architectures are trained and compared:

- **MLFC: 1×50 - MLP: 60×10**
- **MLFC: 1×50 - Washout: 10**
- **LSTM: 1×50 - MLP: 60×10**
- **LSTM: 1×50 - Washout: 10**

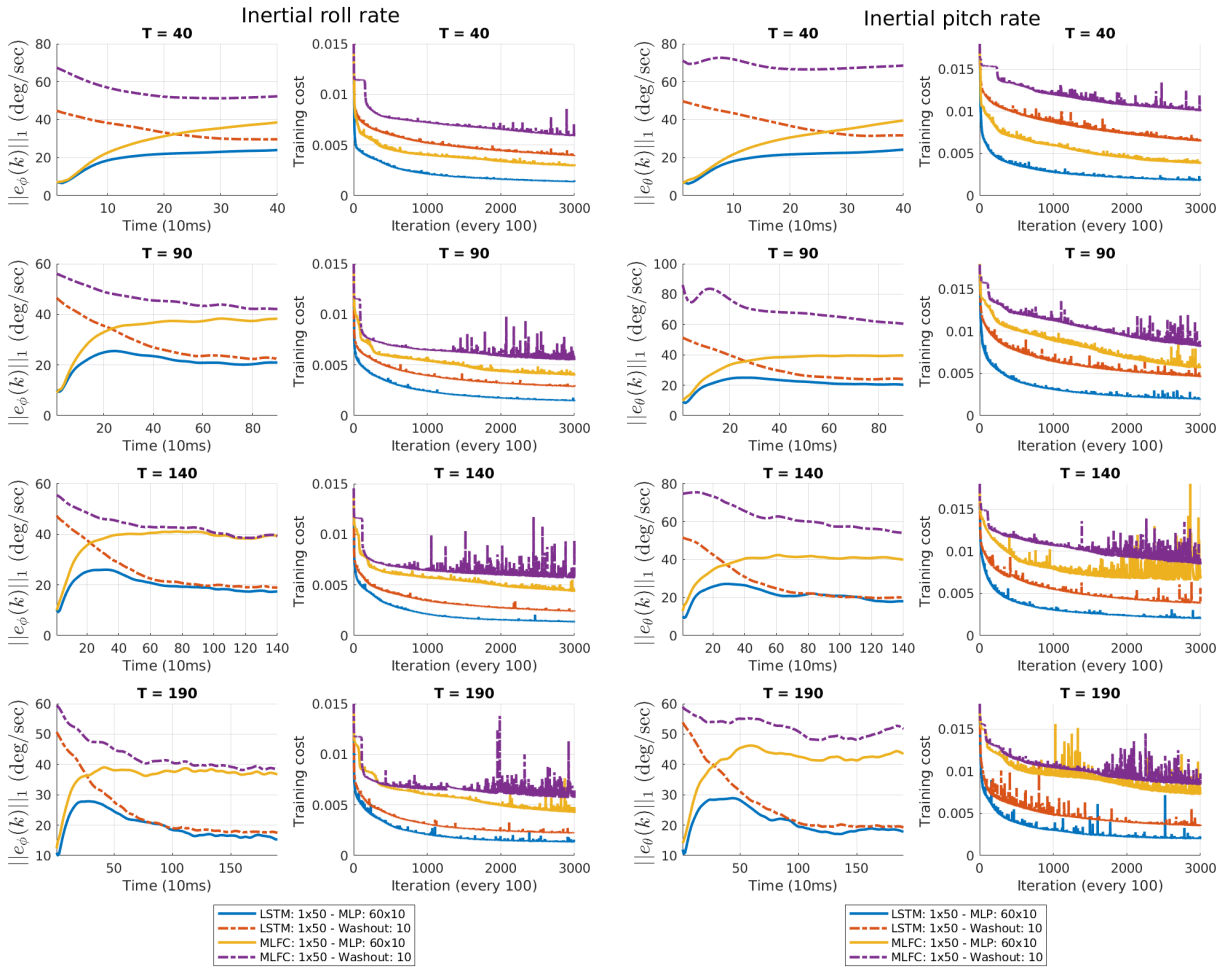
In Figure 4.11a and 4.11b, the plots on the left column illustrate the mean of the validation error over the course of prediction. On the right column, the plots show the evolution of the training cost over the training process. Each row corresponds to one prediction

length, which, from top to bottom, are $T = 40, 90, 140, 190$ samples. It can be observed that the history-based initialization method has improved both the immediate prediction error and the training cost significantly. When the prediction error is long enough, e.g., $T_{tot} = 150, 200$, the history-based and washout initialized RNN predictors converge to almost the same error eventually. However, it is also evident that an efficient washout period is difficult to determine, whereas in the proposed history-based initialization methods such a problem does not exist. The results illustrated in Figure 4.12, which belong to the yaw prediction, also confirm the same observations.

In Figures 4.13 and 4.14 the test costs ($RMSSE_{tot}$) on the test dataset versus the size of the networks (number of weights) are plotted for a variety of architectures. In each figure, the plot titles state the sample lengths. The initialization length is equal to 10 steps for all of the networks. The same datasets as the previous experiments are employed. In these graphs it is observed that the LSTMs with MLP initialization outperform other methods. In fact, LSTMs with fewer weights perform better than MLFCs. Considering the computation time needed for training each single network, as a part of hyper-parameter optimization, it is a reasonable choice to conduct the remaining experiments with LSTMs and history-based initialization.

4.5.2 MLP vs RNN Initializers

In this section, variants of LSTM networks initialized with the two history-based initializer networks are examined. The LSTM networks are comprised of layers of LSTM cells connected in series. The outputs from the last layer are fed back to the first layer. For some experiments, as will be noted, TDLs are placed at the input and output of the networks. The results belong to the helicopter scenarios mentioned in Table 4.3.



(a) Roll rate (b) Pitch rate

Figure 4.11: Comparison between MLP and washout initialization on the helicopter roll and pitch rates.

Figures 4.15 and 4.16 illustrate the total RMSSE cost on the test dataset for the helicopter velocity and angular rates over the previously mentioned sample lengths. Except for the angular rate where $T_{tot} = 50$, the LSTM TDL with RNN initialization outperforms other architectures. Note that the networks are extremely large and are almost the largest that could fit into one Titan X GPU. Since the illustrated RMSSE measures correspond to the test dataset (which is not used for training), the networks did not overfit.

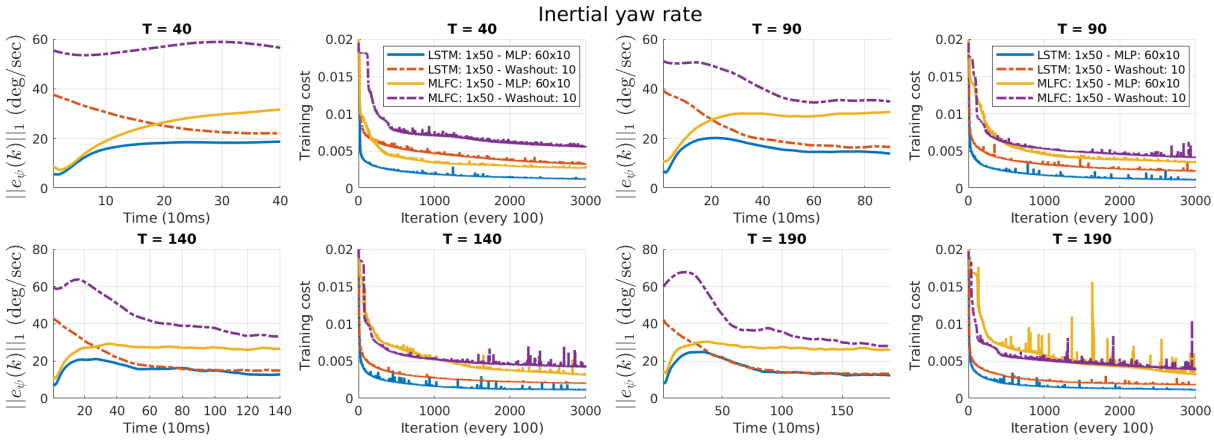


Figure 4.12: Comparison between MLP and washout initialization on the helicopter yaw rate.

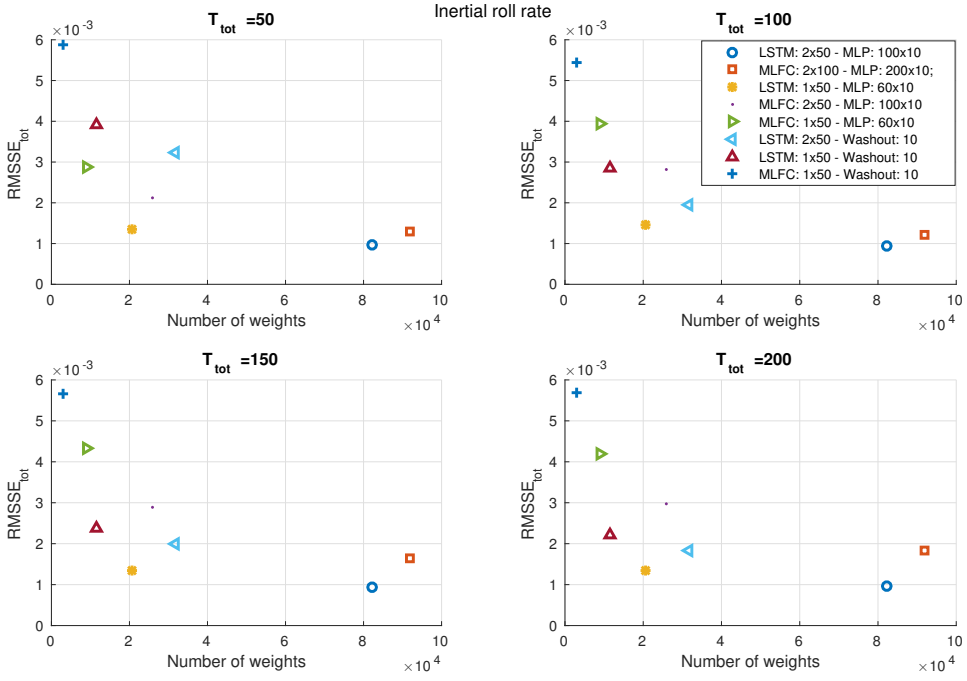


Figure 4.13: Network size vs. $RMSSE_{tot}$ for LSTMs and MLFCs using two initialization schemes. Eight architectures are trained on the helicopter roll rate.

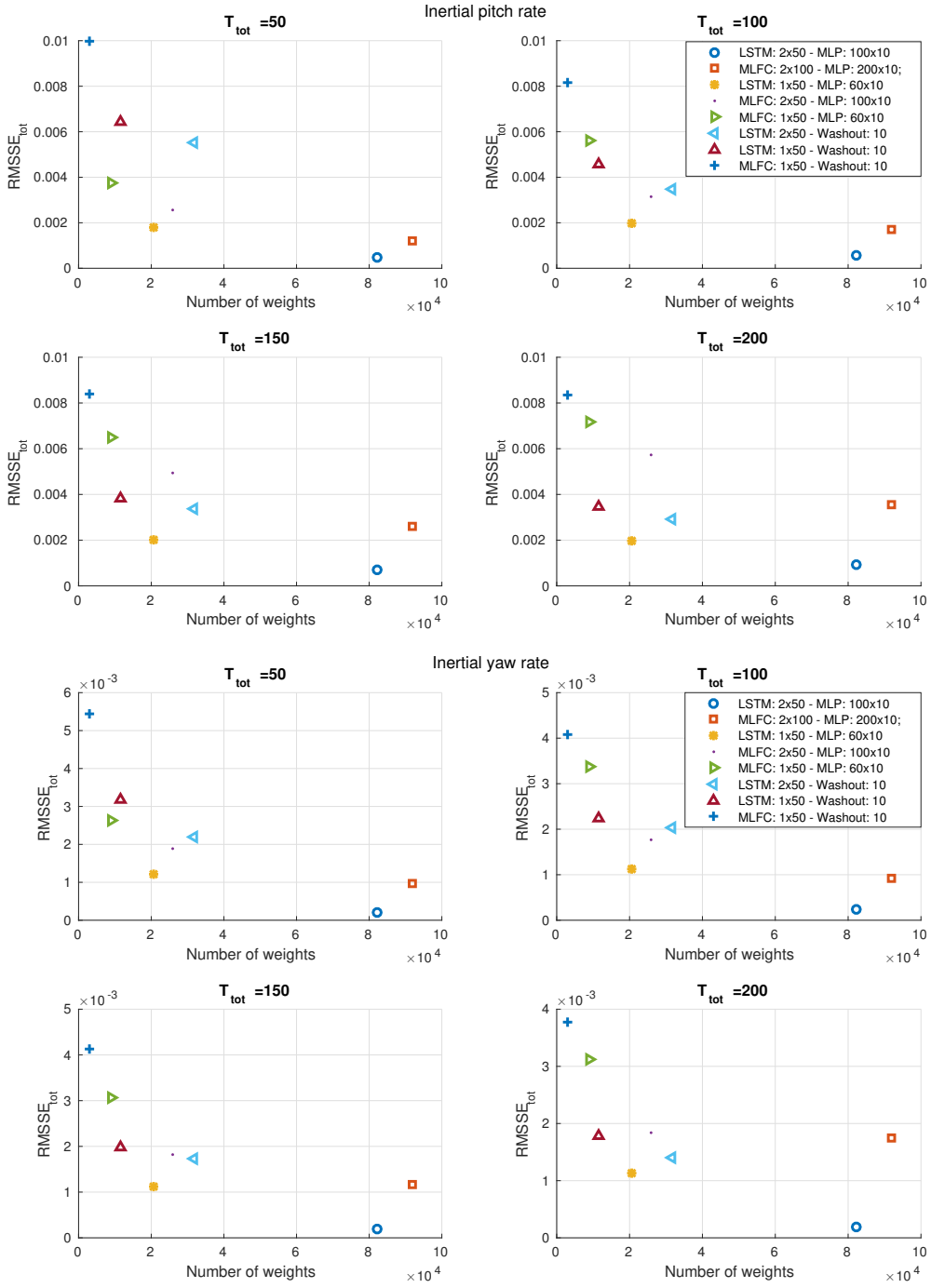


Figure 4.14: Network size vs. $RMSSE_{tot}$ for LSTMs and MLFCs using two initialization schemes. Eight architectures are trained on the helicopter inertial pitch (the four top plots) and yaw (the four bottom plots) rates.

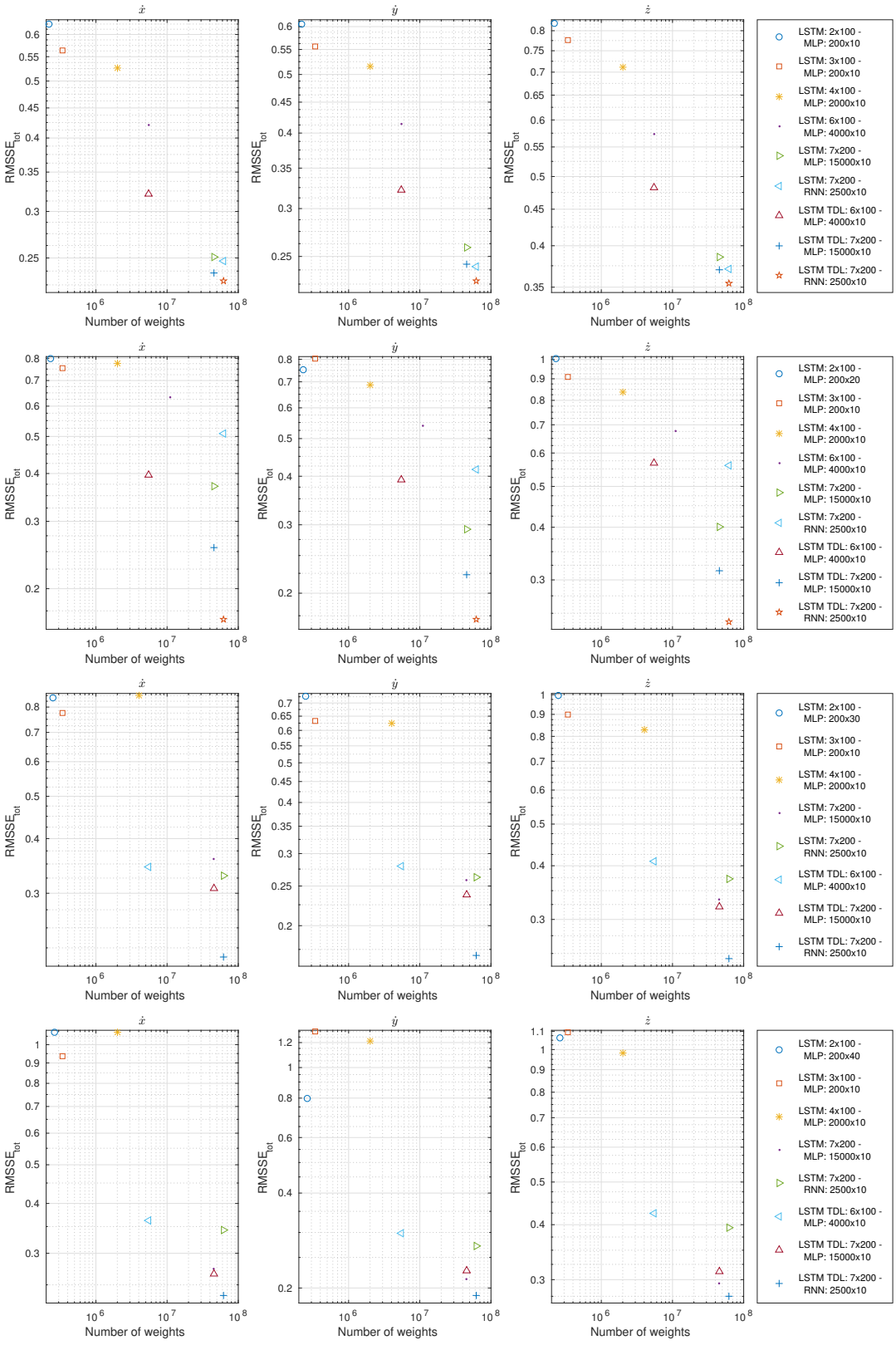


Figure 4.15: Comparisons of network sizes and initialization schemes on learning the helicopter velocity from pilot commands. (From top to bottom: $T_{tot} = 50, 100, 150, 200$)

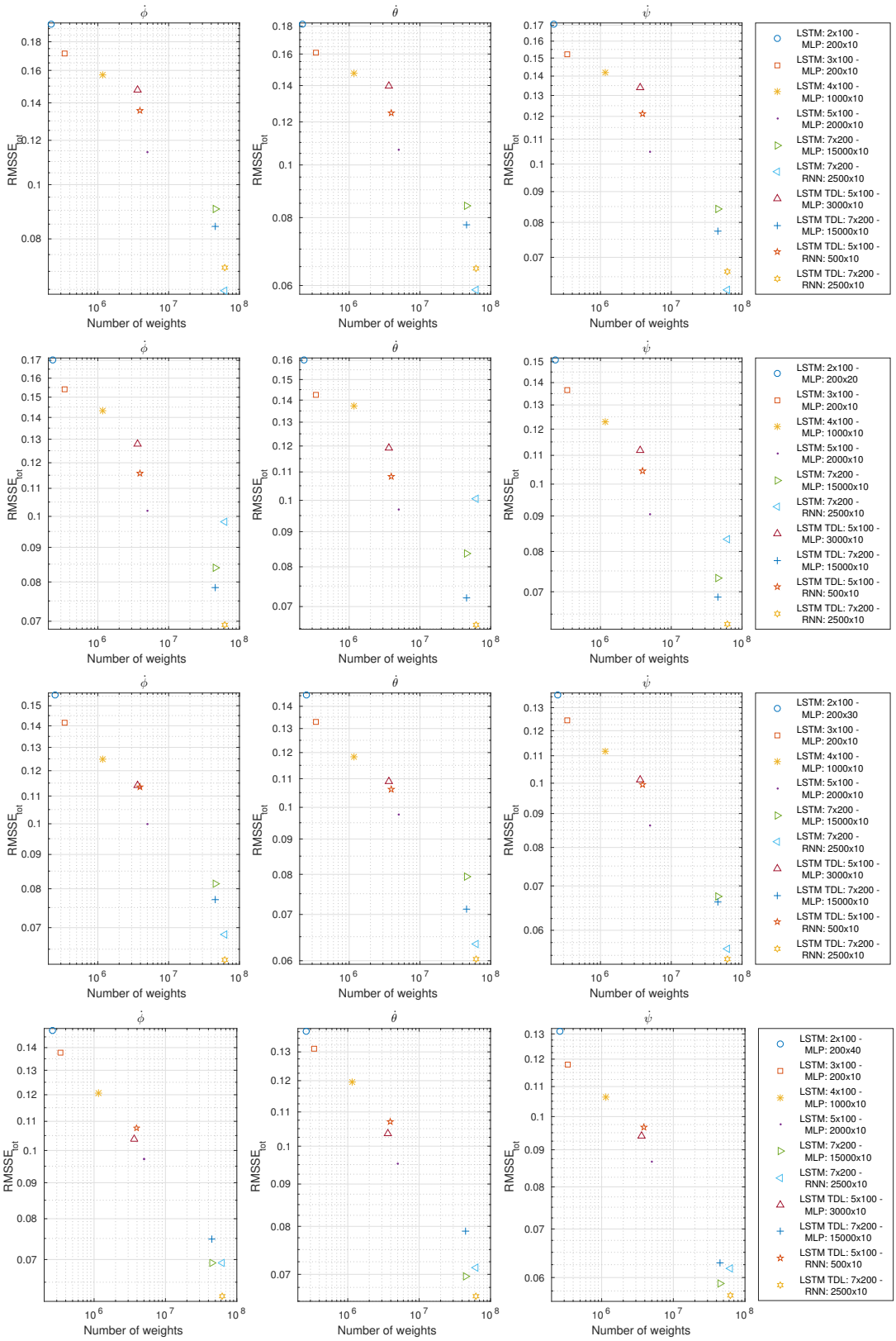


Figure 4.16: Comparisons of network sizes and initialization schemes on learning the helicopter angular rates from pilot commands. (From top to bottom: $T_{tot} = 50, 100, 150, 200$)

A couple of strategies were chosen to avoid overfitting; the network weights were initialized to tiny numbers, weight decay regularization, and the drop-out method [84] were also employed. However, training such a large network for 600k iterations on sequences with 200 samples can last for about 10 to 14 days. It was observed that after almost 400k iterations the network rarely improves on the validation set.

4.5.3 Black-box Modeling of the Helicopter

Based on Figures 4.15 and 4.16 the networks with the best performance are the following architectures,

- **LSTM: 7×200 - MLP: 15000×10**,
- **LSTM: 7×200 - RNN: 2500×10**,
- **LSTM TDL: 7×200 - MLP: 15000×10**,
- **LSTM TDL: 7×200 - RNN: 2500×10**.

In this section, the above architectures are evaluated as black-box models of the helicopter vehicle. The total RMSSE errors, illustrated in Figures 4.15 and 4.16, do not provide much insight into how good the predictions are. To study the reliability and accuracy of the predictions it is best to look at the *distribution* of the prediction error, across the datasets, throughout the prediction length.

Figure 4.17 compares the mean of the error distributions over the four prediction lengths. It would have been expected that the prediction error increases monotonically throughout the prediction length. This is more or less the case for $T_{tot} = 50$. However,

for longer prediction lengths the monotonic increasing behaviour is no longer observed. Instead, a peak appears at the early stages of the prediction and it is attenuated as we go forward in time. This is contrary to our expectation.

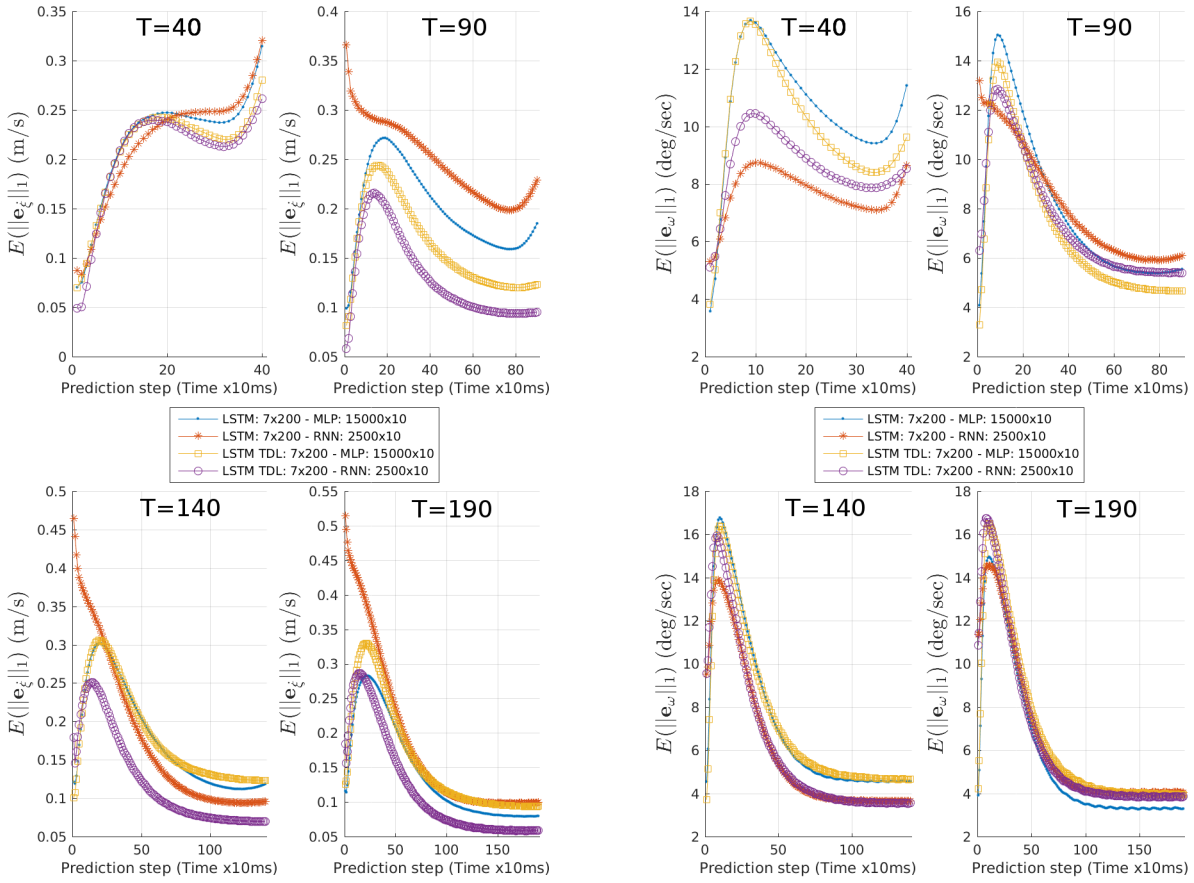


Figure 4.17: Mean of the $\bar{L}_1$ error distributions for the four black-box models of the helicopter. The plots on left correspond to the velocity and the plots on right correspond to angular rates.

Remember that the LSTMs are efficient in learning long-term dependencies. In fact, through the gated architecture, they can hold on to relevant information over longer periods. This property can be very beneficial in attenuating the noise. Therefore, the decrease

in the error over late predictions might be due to noise attenuation and accumulation of more information about the process by the LSTMs. Also, the peaks may be reduced if the initialization length increases. However, increasing the initialization length decreases the lengths to be used for training the predictor networks. It is also observed that the LSTMs, equipped by TDLs and initialized by RNNs generally perform better for longer horizons, which reinforces this hypothesis. However, for the Euler rate predictions, the behaviour of the mean error is not consistent. This inconsistency may be due to the size and quality of the dataset which will be discussed further at the end of this section.

It should also be noted that during the training process, the optimization cost assigns the same credit to each step over the prediction horizon. In a black-box scheme, the system identification is purely based on this optimization and no external information is revealed to the model that a monotonic increase in the prediction error should appear over the prediction horizon. Such information can be artificially embedded in the cost, for example by exponentially weighing the error terms in Equation (4.11). However, doing so slightly improves the early step predictions at the cost of being less accurate at the later stages.

Another interesting observation from Figure 4.17 is the role of TDLs. Considering the two types of predictors, LSTM and LSTM with TDLs, each initialized by the MLP and RNN initializers, it is observed that TDLs improve the prediction error specifically for the RNN initializer. Over all of the learning scenarios illustrated in Figure 4.17, the **LSTM TDL: 7×200 - RNN: 2500×10** network performs the best. For this particular network, let us look at the distribution of the $\bar{L}_1$ norm of the error vector. Since the network behaviour for $T = 40$ and $T = 90$ are more or less similar, as well as $T = 140$ and $T = 190$, only the two extremes, i.e., $T = 40$ and $T = 190$, are considered.

In Figure 4.18, the distribution of the $\bar{L}_1$ errors are illustrated using box-and-whiskers

plots. At each prediction step, the red dash represents the median, the lower and upper bounds of the blue rectangle correspond to the first and third quartiles, q_1 and q_3, respectively, and the whiskers' ends correspond to the extreme cases. The Interquartile Range (IRQ) is defined as $I_r = q_3 - q_1$. If the norm of a prediction error is greater than $q_3 + 1.5I_r$ or smaller than $q_1 - 1.5I_r$ then it is considered an outlier. In Figure 4.18 the outliers, which are less than %8 of the test data, are not illustrated. However, in the median and box size calculations, they are considered. In the plots it is readily seen that the uncertainty in the prediction quickly grows over time. However, in the long run, not only does the mean of the $\bar{L}_1$ error decreases, but also the error distribution improves and the predictions become more reliable. As already discussed, it is possible that the reliability of the immediate predictions improves by increasing the initialization length. It is, therefore, a trade-off between the longer prediction lengths and longer initialization lengths and the dataset size.

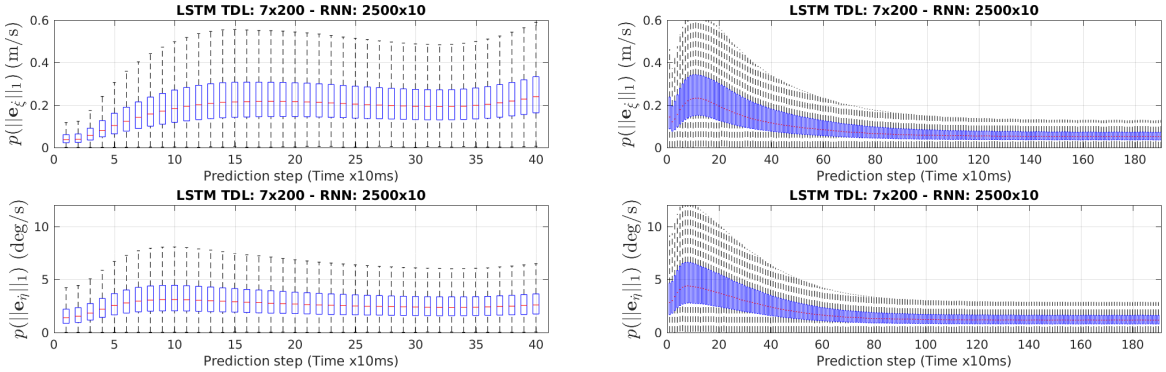


Figure 4.18: Black-box performance for the helicopter dataset evaluated by illustrating the $\bar{L}_1$ error distribution. The plots on the top row correspond to the velocity and the plots on the bottom row correspond to angular rates.

Outliers

The outliers are separately studied as a worst-case performance of the models. Approximately 8% of the test samples are outliers. The mean of the $\bar{L}_1$ error norm of the outliers is plotted in Figure 4.19. Note that the mean is calculated over the outliers *only*. Remember that the data distribution in the helicopter dataset, Figures 4.9 and 4.10, suffer from long tails and therefore the existence of outliers was already expected. Average errors slightly larger than 1 meter per second and 22 degrees per second are observed which are not satisfactory. In the following, some of the possible reasons are listed and improvements are suggested.

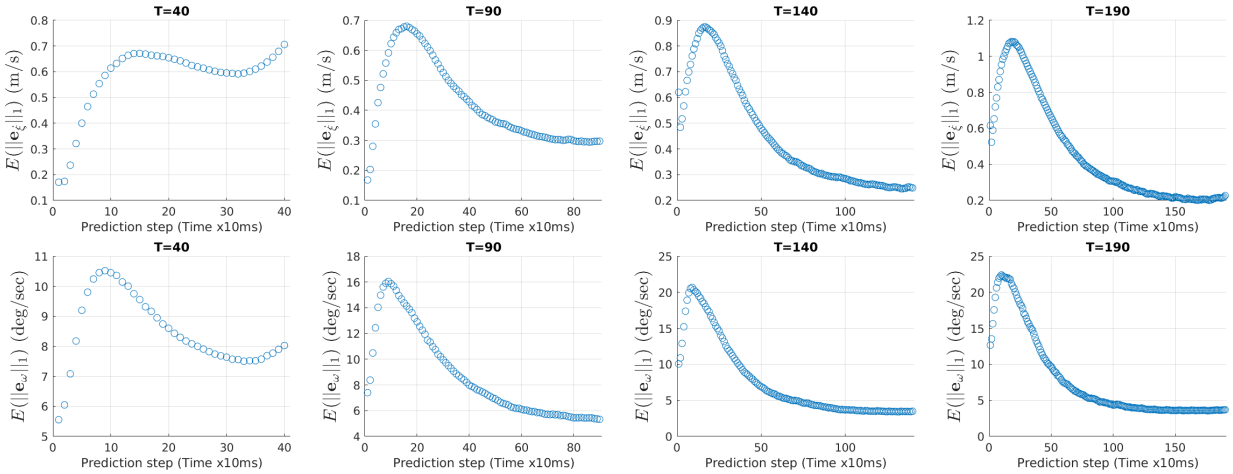


Figure 4.19: Mean of the prediction error norms over the outliers for the helicopter dataset.

1. The input to the networks is the pilot command and there are many levels of transformation which take place before the commands affect the helicopter motion. Time synchronization can also become very difficult to manage in such situations. To circumvent this, using actual motor speeds as the inputs is likely to mitigate effects of delay and command transformation.

2. Considering the complex dynamics of a helicopter, the dataset is relatively small. Better prediction performance is expected if more data is collected in a variety of flight regimes.

3. The helicopter is flown outdoors and is very likely affected by wind. However, there is no measurement of the wind available in the dataset. To obtain a predictor for the vehicle dynamic a controlled environment is more desirable.

In collecting the quadrotor dataset, all of the above drawbacks were considered.

4.5.4 Black-box Modeling of the Quadrotor

In this section, the results of black-box modeling of a quadrotor from experimental data are presented. The experiments throughout this section are the quadrotor scenarios mentioned in Table 4.3. The following four architectures are considered,

- **LSTM: 7×200 - MLP: 15000×10**,
- **LSTM: 7×200 - RNN: 2500×10**,
- **LSTM TDL: 7×200 - MLP: 15000×10**,
- **LSTM TDL: 7×200 - RNN: 2500×10**.

Figure 4.20 compares the mean of the $\bar{L}_1$ norm of the body rate prediction error measured at each prediction step for the aforementioned architectures. The accuracy of the predictions on average remains better than 3.5 degrees per second over almost 2 seconds. Similar to the helicopter case, longer prediction lengths contribute to better accuracy. In

fact, the predictor receives error information at each step, while the initializer only receives the error through the predictor and at the initial step. Therefore, more information about the system dynamics is received by the predictor over the course of prediction. As this amount of information is increased, the RNNs on average perform better, meaning that they can efficiently and effectively employ the extra information.

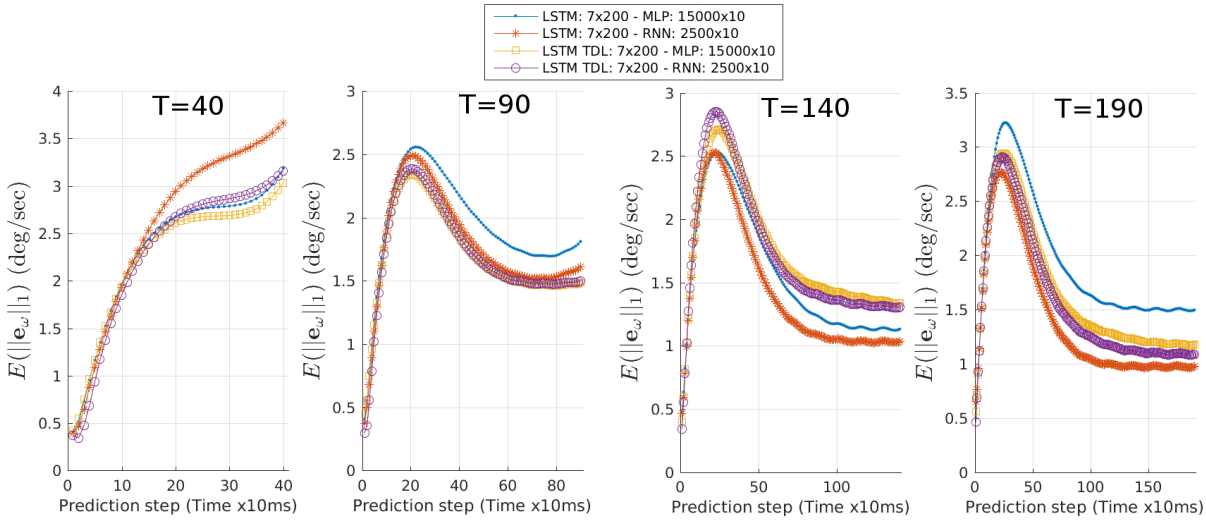


Figure 4.20: Mean of the $\bar{L}_1$ error distributions for the four black-box models of the quadrotor body rates.

In Figure 4.21 the mean of the $\bar{L}_1$ norm of the velocity prediction errors are illustrated in a teacher force mode, i.e., the samples in the test dataset include the measured body rates as inputs. A similar trend to Figure 4.20 is observed. The accuracy of the predictions on average remains better than 4 centimetres per second over almost 2 seconds. From Figures 4.20 and 4.21, it is also observed that TDLs improve the prediction accuracy, which is consistent with our observation from the helicopter dataset. It is also observed that the LSTMs initialized with RNNs (RNN-RNN pairs) have better prediction accuracy over the longer prediction lengths; a reinforcing observation on the argument that the LSTMs

efficiently employ information spread across time.

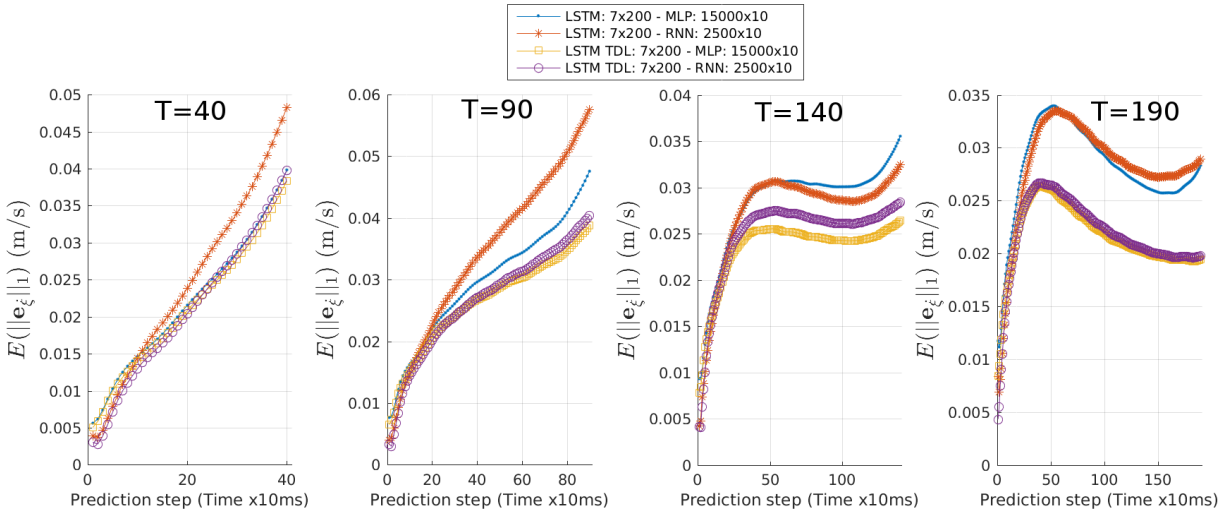


Figure 4.21: Mean of the $\bar{L}_1$ error distributions for the four black-box models of the quadrotor translational velocity in the teacher forced mode.

The actual body rates are replaced with the predicted ones to generate the velocity prediction and the results are illustrated in Figure 4.22. In this figure, the mean of the velocity prediction error is plotted for the four architectures over the four prediction lengths. In comparison with Figure 4.21, where the teacher forced results are illustrated, the velocity prediction accuracy is degraded by a factor of approximately 25. Additionally, in Figure 4.22, it can be observed that the networks with an RNN initializer suffer more from the error in body rate prediction.

The teacher force mode to evaluate the prediction performance is not realistic. In a multi-step prediction, as described in Section 3.1, the *behaviour* of the system is to be predicted for many steps ahead in time. Since the behaviour of a quadrotor is partially described by the body rates, according to the multi-step prediction problem, they are not available at the prediction stage. Therefore, the networks should be employed in the

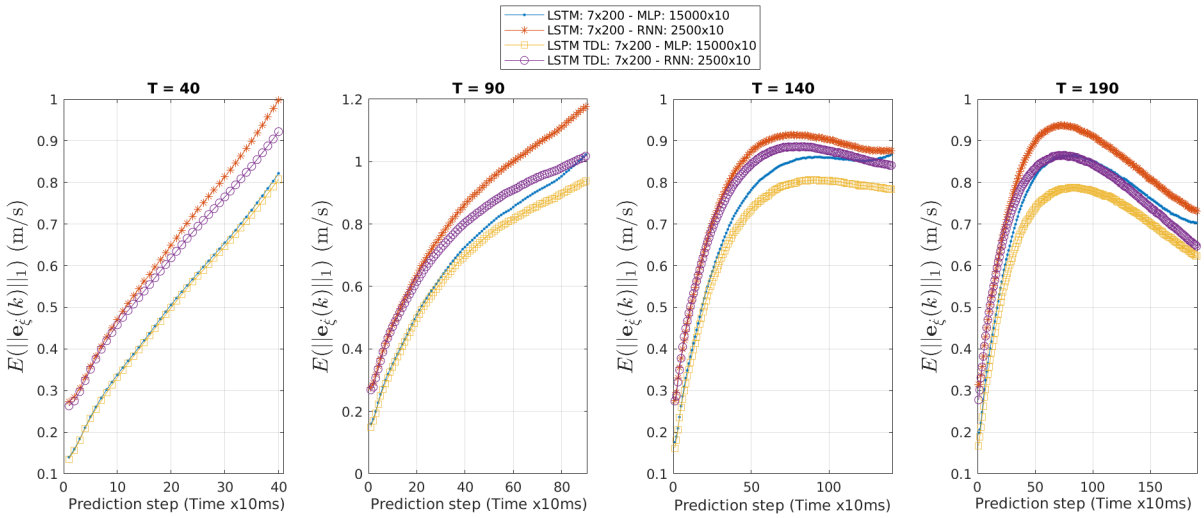


Figure 4.22: Mean of the $\bar{L}_1$ error distributions for the four black-box models of the quadrotor translational velocity. The predicted body rates are used (practical mode)

practical mode. However, the teacher forced results are provided to study the network performance only.

To study the prediction error distributions, the $\bar{L}_1$ and L_2 measures are considered over the two extreme prediction lengths, i.e., $T = 40$ and $T = 190$. The network with the best overall performance is chosen, i.e., **LSTM TDL: 7×200 - RNN: 2500×10**. In Figure 4.23, the distributions of the norms of the body rate prediction errors are illustrated. Note that for the sake of clarity, the outliers are not shown. However, their effect is taken into account to generate the distributions illustrated by the box-plots.

In Figure 4.23, it is observed that the mean grows and the reliability (how far the whiskers are stretched) degrades rapidly over the short prediction lengths. According to this figure, over the test dataset, the predicted body rate errors in each direction (depicted by the $\bar{L}_1$ norm) remain less than 9 and 7 degrees per second over $T = 40$ and $T = 190$ prediction lengths, respectively. The length of the prediction error vector (depicted by the

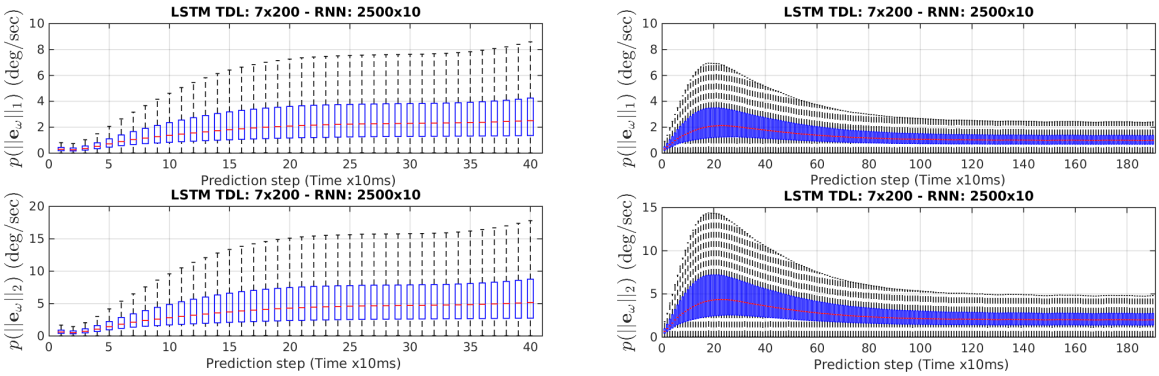


Figure 4.23: Black-box performance for the quadrotor body rate dataset evaluated by illustrating the $\bar{L}_1$ (top) and L_2 (bottom) error distribution.

$\bar{L}_2$ norm) remains less than 19 and 15 degrees per second over the same prediction lengths.

The error distribution over long prediction length exhibits a similar behaviour to the helicopter dataset, which reinforces the hypothesis discussed about the LSTMs capability in accumulating relevant information over long periods of time.

Figure 4.24 illustrates the velocity prediction error distributions for the practical case, over the extreme prediction lengths. It is observed that the mean and reliability of the predictions degrade dramatically; the errors can be as large as 1 meters per second and the mean of the $\bar{L}_1$ norm of the error can increase to about 0.9 meters per second. Such large errors are not useful in control applications. Since the teacher forced velocity predictions are much more accurate, it can be concluded that the body rate prediction errors contribute mainly to the inaccuracy of the velocity predictions.

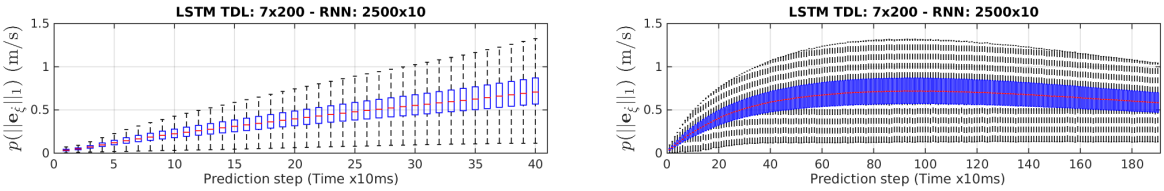


Figure 4.24: Black-box velocity prediction errors for the quadrotor dataset in the practical mode.

Outliers

The outliers consist of about 5% of the test data, for the body rates and teacher force mode velocity predictions, and about 10% for the actual mode velocity prediction. For the case of body rates prediction, over the 5% outliers, the maximum of the mean of the $\bar{L}_1$ norm is increased to about 14 degrees per second. For the velocity prediction in practical mode, the mean over the outliers exhibit an unsatisfactory performance; the prediction error norm grows up to about 3 meters per second, and the number of outliers is also increased. The mean of the $\bar{L}_1$ norm over the outliers are illustrated in Figure 4.25

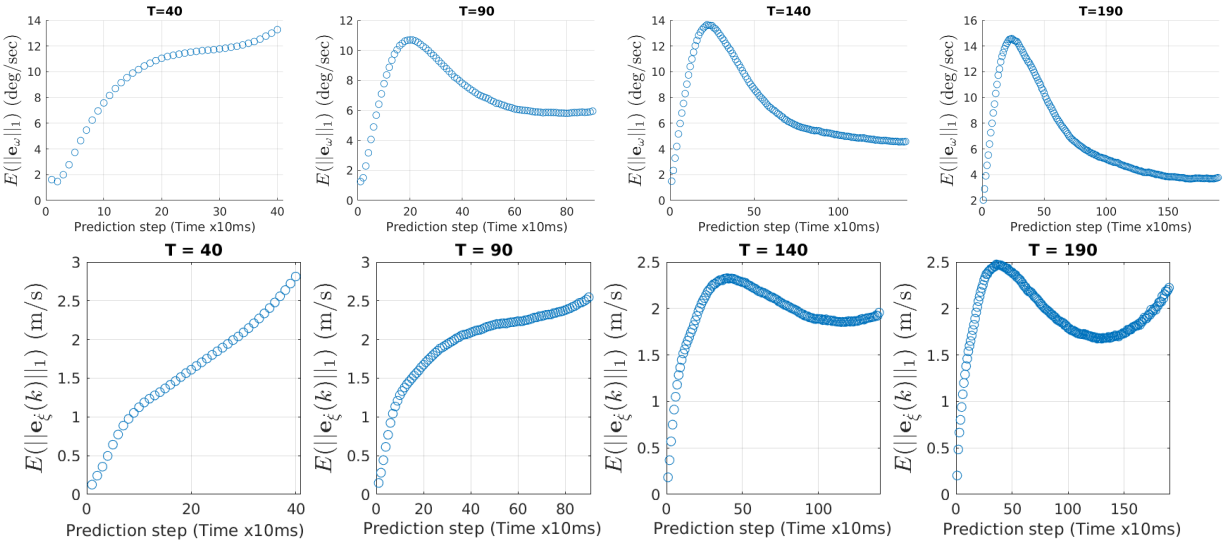


Figure 4.25: Mean of the outliers in the body rate and velocity (actual mode) prediction error distributions of the quadrotor.

4.6 Summary and Conclusion

In this chapter, RNN-based black-box architectures are proposed and comprehensively studied on learning models from experimental data. The importance of RNN state initialization for dynamic system modeling is highlighted and a novel state initialization method, based on a short history of the system measurements, is proposed. The effect of the proposed method, in comparison with the current methods to initialize the RNN states, are illustrated on modeling two aerial vehicles from experimental data. The state initialization method addresses the drawbacks of the washout method which is the inaccurate transient response of an RNN over an arbitrary length of time. Using the proposed method, various RNN architectures are trained and the results show consistent behaviour over two different datasets collected in two entirely different settings. The models studied in this chapter provide a promising method to develop black-box models purely from input-output data.

The black-box models provide an immense number of DOFs as they have millions of adjustable weights. Therefore, one possible reason for some of the unsatisfactory prediction performance in the black-box scheme provided in this chapter is the limited number of training samples. In fact, small networks, due to their limited number of nonlinearities, do not provide enough flexibility to capture complex dynamics. Large networks, on the other hand, demand large datasets that are representative of the underlying dynamics. Reducing the network size to fit for the datasets does not seem to be a viable approach, and collecting large datasets from real robotic systems can be a time consuming and expensive task. However, it is possible to exploit the nature of a robotic system based on their physical properties and incorporate this knowledge with the flexibility of a black-box model. Based on this idea, an alternate approach is proposed in the next chapter.

Chapter 5

Hybrid Models for Multi-Step Prediction

In this chapter, a hybrid of the quadrotor physics based (white-box) model, introduced in Section 2.7, and the black-box model, discussed in Chapter 4, is proposed and trained on the quadrotor dataset, for the multi-step prediction problem [65]. The hybrid model prediction performance is compared with both the white-box and black-box models. It is demonstrated that the hybrid models provide a more accurate and reliable prediction of the quadrotor body rates and velocity in comparison to the white-box and black-box models, which can be employed in model-based controllers.

5.1 Motivation

Many characteristics of a robotic system can be formulated using the laws of physics. For instance, a mobile robot, on the macroscopic scale, obeys rigid-body dynamics [21, 28].

However, some characteristics of the system might be too difficult or expensive to accurately model, such as the vortex ring effect on a quadrotor. As described in Chapter 1, a grey-box modeling approach combines two models, a white-box model which formulates the physical nature of the system using first principles, for instance, the system dynamics, and a black-box model which is solely based on numerical methods. A grey-box modeling approach can speed up the modeling process and increase the prediction accuracy of the model. Also, the model can benefit from physical measurements which are easily measurable and not variable, such as the mass of a quadrotor. The final model should reliably span a larger portion of the state-space. See [54] and [56] for more discussion on grey-box modeling.

Dynamic models for quadrotors have been studied extensively in [37, 58, 14]. First principles models of quadrotors have also been used in control [37, 89, 23]. These models tend to rely on steady state aerodynamic models that do not capture rapid dynamic motions. Additionally, to develop a white-box model, as explained in Section 2.7, some simplifying assumptions are made, such as the symmetric design, the linear dependence between the generated motor forces and the thrust and torques, the steady state thrust model, the constant drag, etc. In addition, some parts of the system remained unmodeled which can be quite complex, such as blade flapping, the vortex effect and the ground effect. The simplifying assumptions and unmodeled dynamics in the developed models will result in a rapid divergence of the predicted states when the models are used for multi-step prediction, as it is demonstrated in this work. As a result, these models are primarily used in single-step predictions for vehicle control.

A major drawback in the previously proposed black-box methods for learning the quadrotor model is the difficulty of learning translational velocity directly from motor speeds. To address this, a two stage process was employed in Chapter 4. However, it

was demonstrated that using the predicted body rates deteriorates the accuracy of the velocity prediction. In this chapter, the body rates and translational velocity are predicted concurrently by the hybrid model, demonstrating a significant improvement in the hybrid architecture to extract a long-term prediction model from the same datasets provided to both networks.

5.2 Grey-box Modeling of a Quadrotor

The hybrid model consists of two black-box modules and a white-box module. The two black-box modules are called the Input Model (IM) and Output Model (OM), and the white-box module is named the Motion Model (MM). The IM module generates the torques and thrust which are then plugged into the MM module. The MM module updates the states of the quadrotor for one step using Equations (2.30). The updated velocity and body rates are then passed through the OM module to *compensate* for the prediction error introduced by the MM module because of the unmodeled dynamics and noise. The compensated states are then fed back to the MM module. Depending on the assumption for the relation between the error and the MM output, a Serial and a Parallel configuration are considered and described next.

5.2.1 Serial Configuration

In this configuration, no restriction is placed on the relation between the MM module output and the compensation term. In fact, the output of MM does not directly contribute to the network output. The diagram of the Hybrid-Serial configuration is illustrated in Figure 5.1.

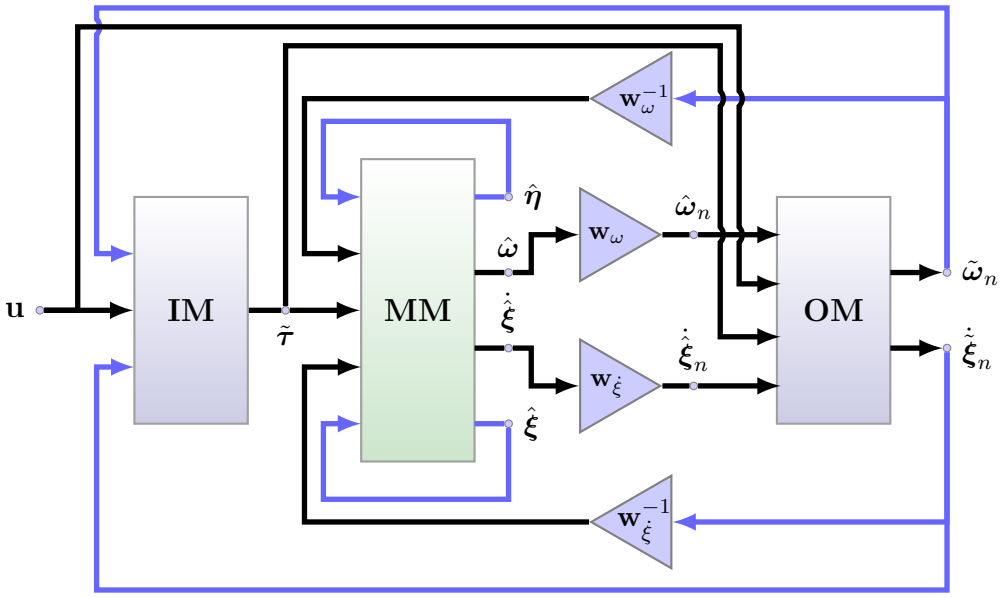


Figure 5.1: Grey-box model of a quadrotor, serial configuration. The black and light blue connections are the feedforward and feedback routes, respectively.

The following notions are used in Figure 5.1,

- $\mathbf{u}$ is the motor speeds vector, $\mathbf{u} = [u_1 \ u_2 \ u_3 \ u_4]$,
- $\tilde{\boldsymbol{\tau}}$ is the thrust and torque generated by the IM module (Equation 2.32),
- $\hat{\boldsymbol{\eta}}$, $\hat{\boldsymbol{\omega}}$, $\hat{\boldsymbol{\xi}}$ and $\dot{\hat{\boldsymbol{\xi}}}$ are the Euler angles, body rates, position and velocity respectively, which are updated by the MM module (uncompensated),
- $\mathbf{w}_\omega$ and $\mathbf{w}_{\dot{\xi}}$ are the normalization factors for the body rates and velocity respectively,
- The index n indicates a normalized quantity.

The velocity ($\dot{\hat{\boldsymbol{\xi}}}$) and body rate ($\hat{\boldsymbol{\omega}}$) updates from the MM module are normalized and then fed to the OM module. The OM module generates the compensated velocity and body rates, $\dot{\tilde{\boldsymbol{\xi}}}_n$ and $\tilde{\boldsymbol{\omega}}_n$, which are scaled back to their physical range to be used by the

MM module. Note that the normalization factors are needed to avoid saturating the input activation functions of the OM module.

Since the black-box modules have many DOFs, it is possible that in practice the MM module is not effectively employed. One way to avoid this possible outcome is to incorporate the output of the MM module directly into the hybrid model output, which leads to the second configuration.

5.2.2 Parallel Configuration

The assumption in the Hybrid-Parallel configuration is that the error from unmodeled dynamics and noise have an additive relation to the MM module output. This configuration is illustrated in Figure 5.2. The main difference between the Hybrid-Serial and Hybrid-parallel models is the inclusion of the MM module output in the total model output using addition.

5.3 Results

The hybrid models proposed in this chapter are trained on the quadrotor dataset. They are compared with the white-box model, introduced in Section 2.7, and the black-box models proposed and studied in the previous chapter. The white-box model parameters are either measured or identified using a Least-Squares (LS) method and reported in Table 5.1.

Using the identified parameter in the white-box model, Figure 5.3 shows the single-step error distribution as well as the mean error over multi-step prediction. As the error

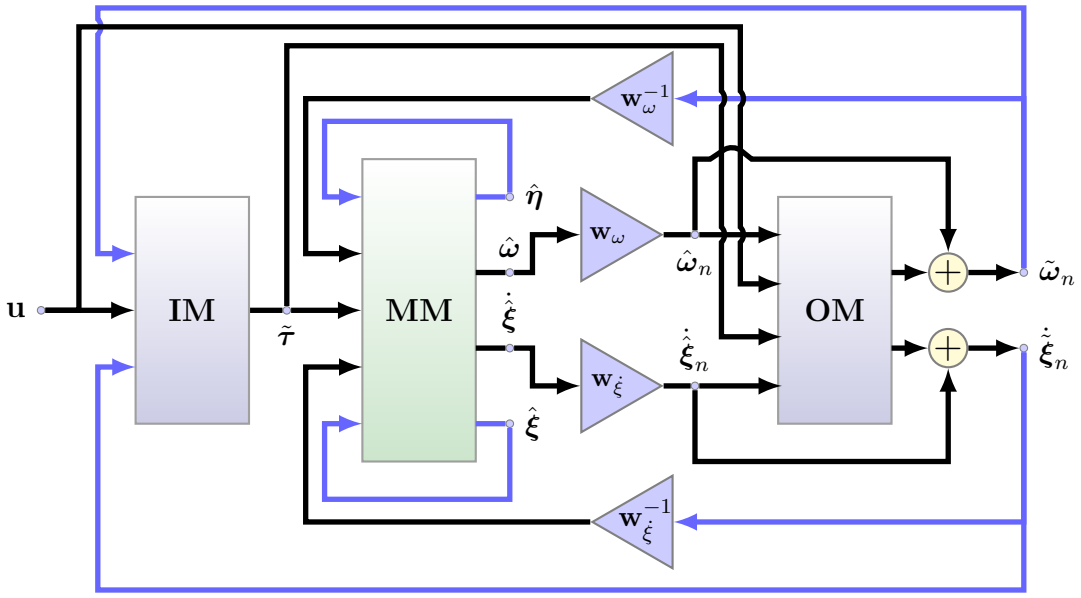


Figure 5.2: Grey-box model of a quadrotor, parallel configuration.

Param.	Value	Param.	Value	Param.	Value	Param.	Value
b_1 *	0.062	b_3 *	1.69×10^{-3}	I_{xx}	$0.002(kg.m^2)$	k_r *	0.0099
k_1 *	7.63×10^{6}	k_3 *	4.67×10^{6}	I_{yy}	$0.002(kg.m^2)$	k_t *	2.35×10^{-14}
b_2 *	0.082	b_4 *	2.28×10^{-4}	I_{zz}	$0.001(kg.m^2)$	l	$0.211(m)$
k_2 *	1.21×10^{7}	k_4 *	4.69×10^{6}	m	$1.6(kg)$	g	$9.81(m/s^2)$

Table 5.1: Quadrotor parameters obtained for the white-box model. The parameters denoted by * are identified using LS method and the rest are measured.

drastically grows over multi-step prediction, using the white-box is not considered for multi-step prediction.

In Figure 5.4, the single-step prediction performance is compared between the hybrid-parallel model and the white-box. Clearly the hybrid-parallel model performs significantly better than the white-box in the single-step prediction as well. The white-box prediction has a strong bias and is inaccurate compared to the hybrid model.

Figure 5.5 compares the two hybrid configurations with the best black-box model pre-

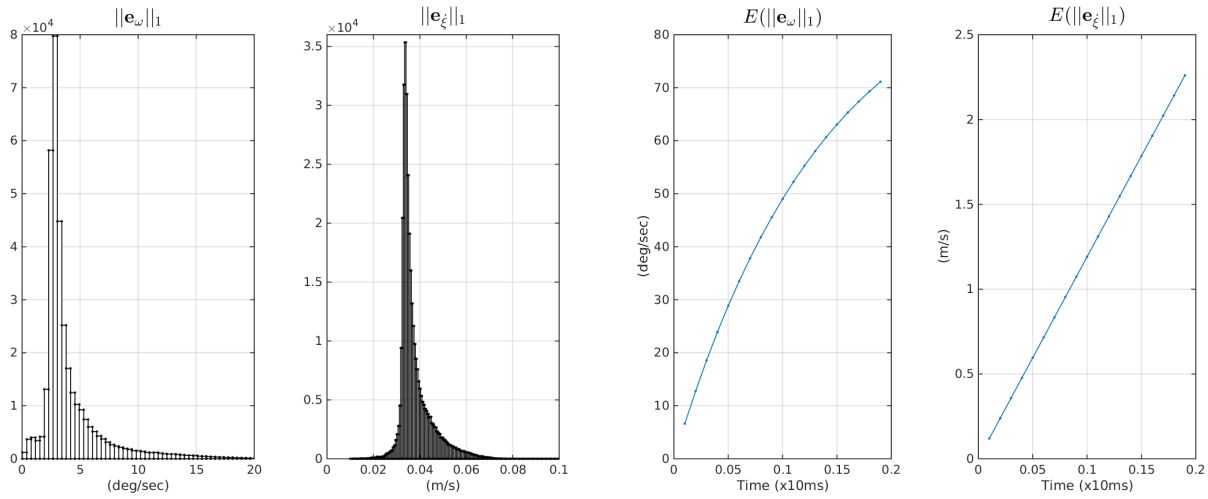


Figure 5.3: White-box model prediction performance. On the left, the prediction error distribution for the velocity and body rates are illustrated. On the right, the mean of the $\bar{L}_1$ error distributions over 20 steps are illustrated.

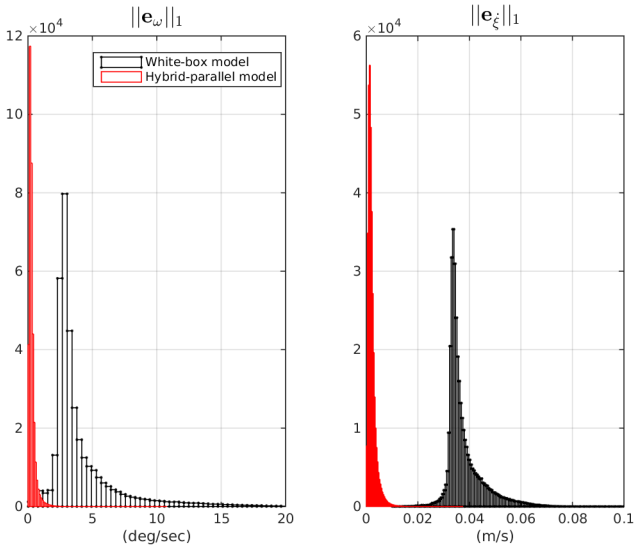


Figure 5.4: White-box model vs. hybrid-parallel model in a single-step prediction scenario.

sented in the previous chapter, i.e., **LSTM TDL: 7×200 - RNN: 2500×10**. The network configuration used in the IM and OM modules of the hybrid models are all **LSTM TDL: 4×200 - MLP: 5000×10**. For the velocity prediction, the y-axis is logarithmic because the black-box prediction error, in the practical mode, is much larger than that of the hybrid models. The hybrid-models dominantly outperform the black-box model. In fact, for the velocity prediction, the hybrid-models improve the prediction accuracy by approximately 2 orders of magnitude compared to the black-box models. The average velocity prediction error remains below 3 centimetres per second over 1.9 second prediction length. For the body rates prediction, the hybrid models improve the prediction accuracy by almost one degree per second over short prediction lengths. The average body rate prediction error remains below 2 degrees per second over a 1.9 second prediction length. However, the black-box model performs only slightly better over the late predictions to about 1.5 degrees per second. The combined performance over velocity and orientation prediction, the significant improvement over the velocity prediction and the improved immediate response of the hybrid models are the main reasons that the hybrid models are preferred over the black-box models.

The inclusion of a motion model introduces some preferences to the optimization of the neural network weights. In fact, the weight space will no longer be explored evenly and areas that are associated to a better MM output are preferred. Although this preference pays well for the short term prediction, it may reduce the capability of the networks to capture long-term dependencies and other inaccuracies. By comparing the late predictions between the hybrid-parallel and hybrid-serial, it can be seen that the latter generates more accurate late predictions. This observation aligns with the aforementioned hypothesis; because the output of the MM module is not directly penalized, the weight optimization

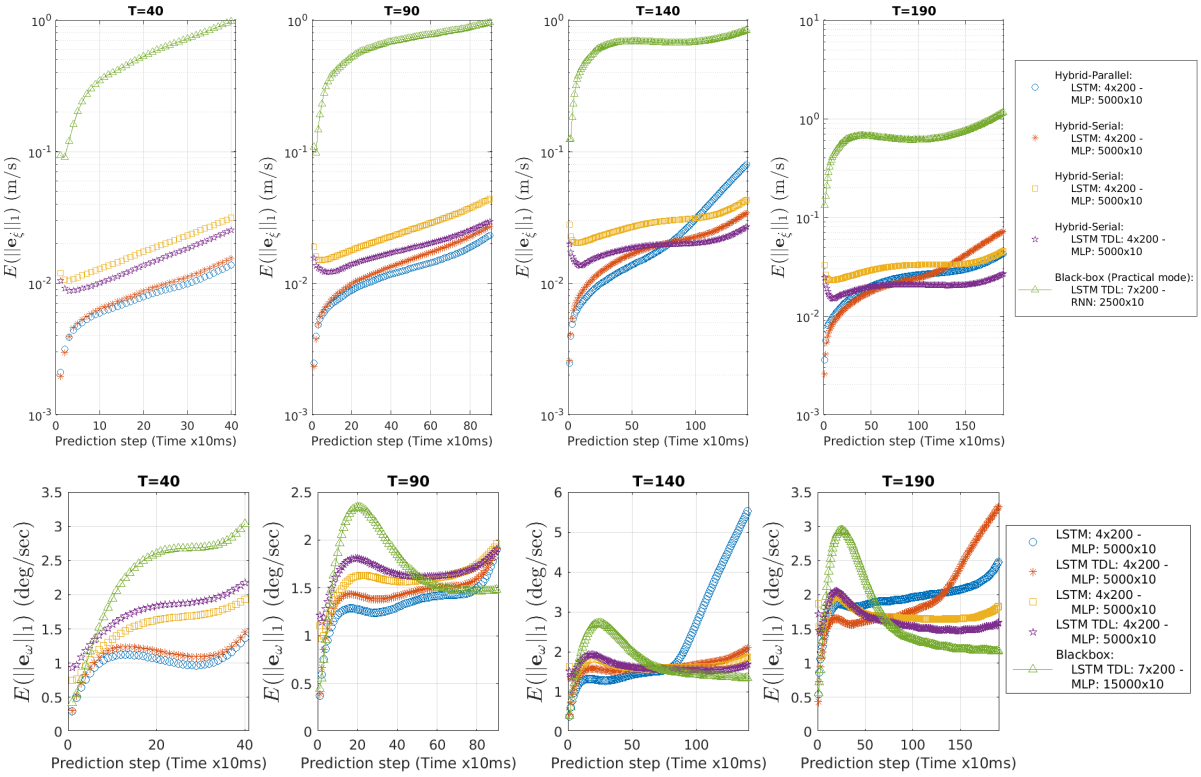


Figure 5.5: Comparison of the mean of the $\bar{L}_1$ error distributions between the hybrid models and the best black-box model obtained in Chapter 4. The plots on the top row correspond to the velocity and the bottom plots correspond to body rate predictions. The velocity prediction accuracy has improved by almost 2 orders of magnitude.

for the hybrid-serial model is affected less from the preference introduced by the MM module, and therefore, the black-boxes of the hybrid-serial model can explore the weight space more freely. This freedom, however, comes at a cost; the hybrid-serial model performs less accurately than the hybrid-parallel at the early stages of prediction.

From Figure 5.5 it is difficult to study the influence of TDLs. In the case of long prediction horizons they slightly improve the accuracy of prediction where in the case of short term predictions they slightly decrease the accuracy. Note also that the peak observed on the early stages of prediction in the black-box cases are not severely present

in the hybrid models.

In general, the hybrid-parallel model is a better candidate to be used in a control application since it provides more accurate early predictions. However, for the case $T_{tot} = 150$ a rapid increase is observed after about 100 steps of prediction, which does not exist in other cases, such as $T_{tot} = 200$. It is therefore possible that this increase is anomalous behaviour that results from the chosen training hyperparameters.

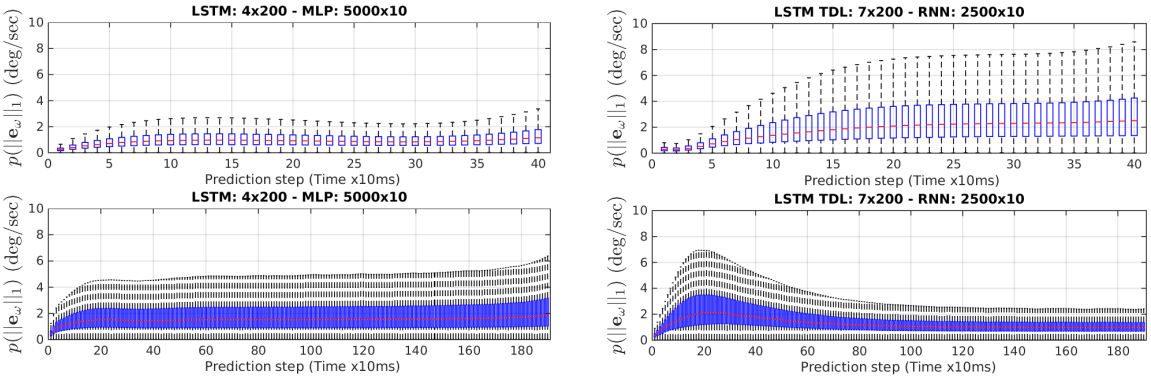


Figure 5.6: Comparison between the hybrid-parallel and black-box prediction error. The distribution of the $\bar{L}_1$ norm of the body rates prediction error are plotted for two extreme prediction lengths, $T = 40$ and $T = 190$. The plots on the left belong to the hybrid-parallel model and the ones on the right belong to the black-box model.

In Figures 5.6 and 5.7 the distribution of the body rate and velocity prediction errors for the hybrid-parallel model are presented and compared with the black-box model. The IM and OM modules of the hybrid-parallel model employ the **LSTM TDL: 4×200 - MLP: 5000×10** architecture. It can be observed that the body rate prediction is improved by almost 50% in the short prediction horizon ($T = 40$). The improvement on the longer prediction horizon, $T = 190$, is more significant on the early predictions. As we go forward in time, the black-box performs better, as discussed previously. For the velocity prediction, the approximately 2 orders of magnitude improvement is also evident on the distributions. It is worthwhile to mention that both the mean and the uncertainty of the prediction errors

are improved as observed by the length of the whiskers.

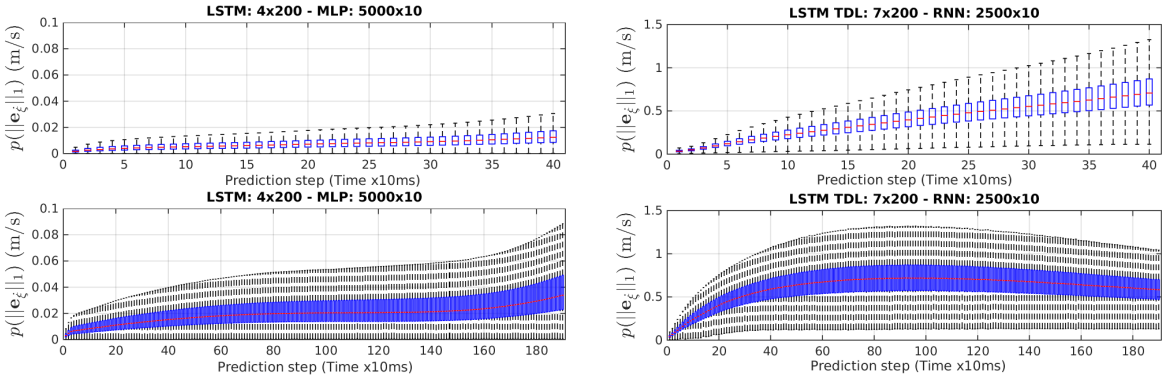


Figure 5.7: Comparison between the hybrid-parallel and black-box prediction error. The distribution of the $\bar{L}_1$ norm of the velocity prediction error are plotted for two extreme prediction lengths, $T = 40$ and $T = 190$. The plots on the left belong to the hybrid-parallel model and the ones on the right belong to the black-box model. An improvement more than an order of magnitude is observed by using the hybrid-parallel model.

5.3.1 Outliers

Overall, the percentage of outliers are slightly improved for the hybrid-parallel model, in comparison to the black-box model (Section 4.5.4), to 4.4%. The mean of the error distribution of the outliers are also improved as can be seen in Figures 5.8 and 5.9, for the body rates and velocity, respectively. In Figure 5.9, the y-axis is logarithmically scaled so that the velocity prediction error for the case of teacher forcing as well as the practical mode can be illustrated and compared with the hybrid-parallel model. The approximately 2 orders of magnitude improvement is also observed over the outliers for the velocity prediction, comparing the hybrid-parallel with the black-box model employed in practical mode. The outliers of the velocity prediction obtained from the hybrid-parallel model is also improved compared to the black-box predictions in teacher forced mode.

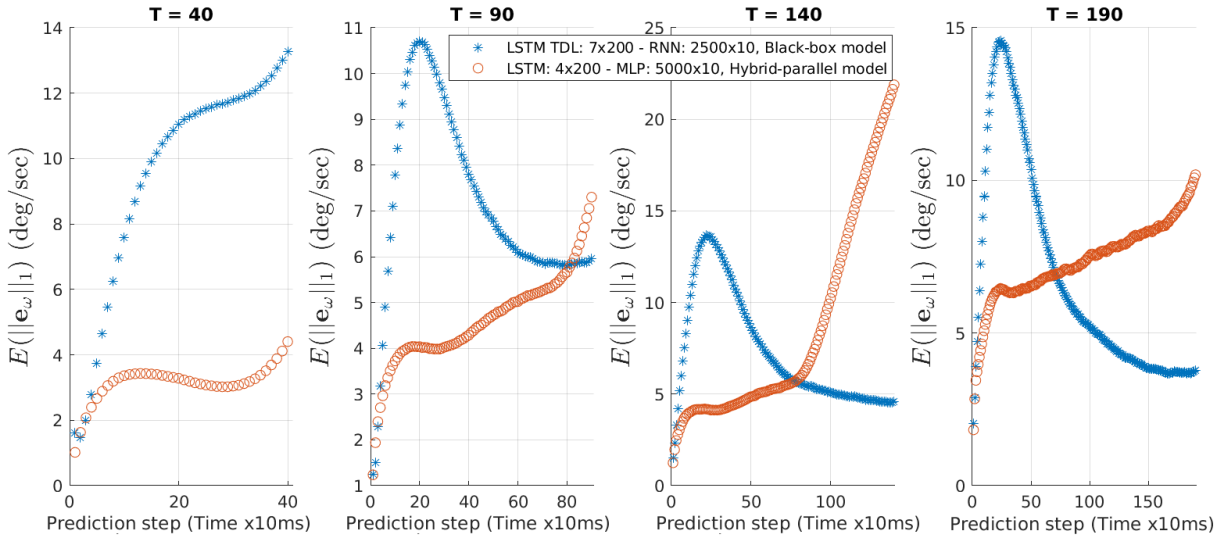


Figure 5.8: Mean of the error $\bar{L}_1$ norm over the outliers of the distribution, for the quadrotor body rates prediction compared between the black-box and the hybrid-parallel models.

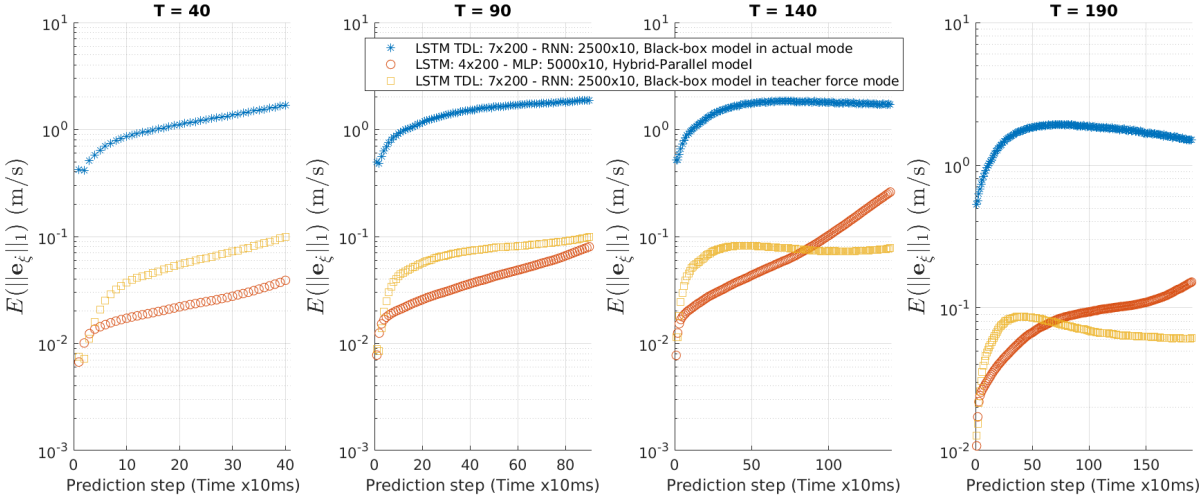


Figure 5.9: Mean of the error $\bar{L}_1$ norm over the outliers of the distribution, for the quadrotor velocity prediction compared between the black-box and the hybrid-parallel models. A logaithmic scale is used on the y-axis.

5.3.2 Compensation Effects

In Figures 5.10 and 5.11, the mean of the $\bar{L}_1$ norm of the errors are plotted over the output of the MM module and the OM module separately. These two plots illustrate the effect of the output compensation. As it can be observed, the OM module applies a correction of about 2 degrees per second on the predicted body rates, and 2 centimetres per second on the predicted velocity at each prediction update. Note that at each prediction step, the compensated outputs are fed back and used by the MM module to produce the next prediction. Clearly, if the output compensation is removed, the error will exponentially increase, as it happens for a white-box model being used in a multi-step prediction (Figure 5.3).

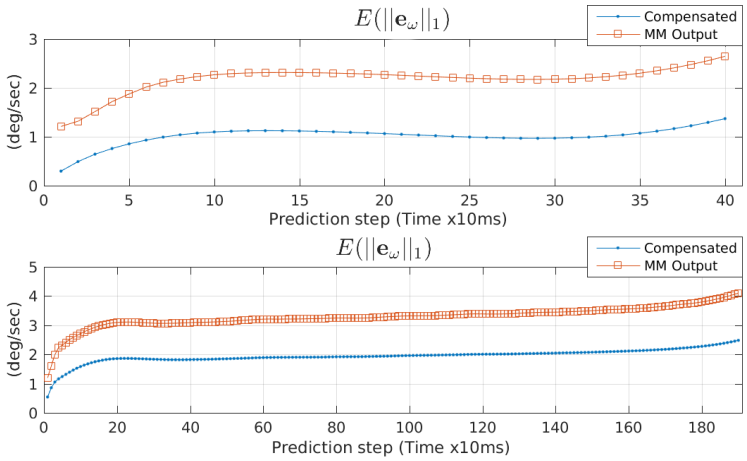


Figure 5.10: Compensation effect of the OM module on the predicted body rates from the MM module. The mean of the errors are plotted.

5.4 Summary and Discussion

A hybrid-serial and hybrid-parallel model, consisting of a first principles based white-box module and two RNN based black-box modules, are proposed and trained to learn a model

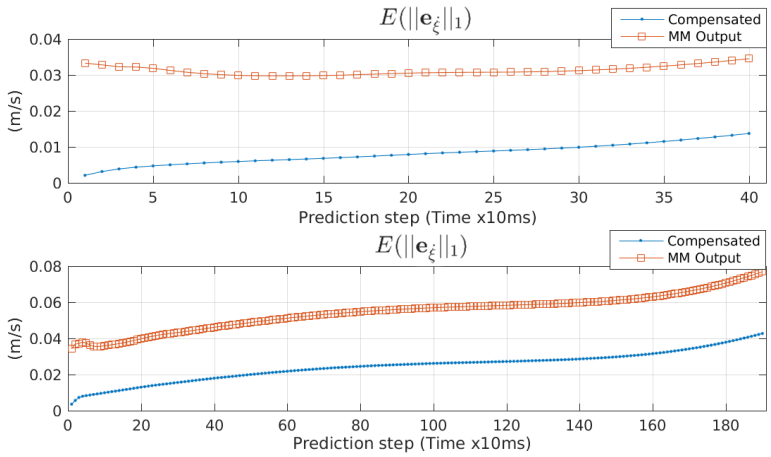


Figure 5.11: Compensation effect of the OM module on the predicted velocity from the MM module. The mean of the errors are plotted.

of a quadrotor vehicle from experimental data for multi-step prediction. Because the updated states from the motion model are included in the output of the hybrid-parallel model, the hybrid-parallel model performs better than the hybrid-serial one. Evaluating the trained hybrid-parallel model on the quadrotor dataset shows that the average of the velocity prediction error remains less than 3 centimetres per second over 1.9 second worth of prediction, and the mean error of the body rate prediction remains below 2 degrees per second over the same prediction length.

Based on the presented results, the developed model provides an accurate multi-step prediction of the quadrotor behaviour as represented by the collected dataset. The maximum velocity error (Figure 5.9) over 1.9 seconds of prediction remains less than 10 centimetres per second, and therefore, the maximum position error remains less than 19 centimetres. Using a Pelican quadrotor in the bridge inspection application described in Section 3.1.1, the error margins introduced in Figure 3.1 can be set as low as 19 centimetres. The quadrotor length and width are equal to 65.1 centimetres. Using the developed model, the error margin is about 30% of the vehicle dimensions. The maximum value for

the quadrotor velocity is about 4 meters per second (Table 4.2). Flying at its maximum speed, the vehicle can traverse a path whose length is approximately 7.6 meters in 1.9 seconds. Therefore, the safe lengths described in Section 3.1.1, are 1.9 seconds in time and 7.6 meters in distance while the position and velocity errors remain less than 19 centimetres and 10 centimetres per second, respectively.

Chapter 6

Conclusion and Future Work

Predicting the behaviour of dynamic systems over multiple time steps is a difficult problem since the error from unmodeled dynamics and noise accumulate and deteriorate the prediction accuracy. This problem is more significant when dealing with real world systems, such as a quadrotor, which are affected by many complex phenomena that are difficult to model precisely using physical models. Despite the difficulty, multi-step prediction has many applications, including model predictive control, feedforward control and simulation.

In this work, Recurrent Neural Networks are studied as a black-box modeling tool that implement a rich class of dynamic systems. The feasibility of RNNs to develop black-box models of dynamic systems from input-output data is assessed in modeling a simulated quadrotor for multi-step prediction. Traditional RNN architectures, such as RMLP and NARX-MLP, are not fully capable of modeling the simulated vehicle. When using these architectures the learning process frequently fails to converge or the trained model performs poorly in generalizing to unseen data. A novel deep RNN architecture, namely the Multi-Layer Fully Connected or MLFC architecture, is proposed that employs many sigmoid

layers in series and equips them with interlayer feedforward and feedback connections. The interlayer (or skip) connections attenuate the vanishing gradient problem that arises from the deep structure (see Section 3.3.6). The proposed MLFC architecture is formulated to facilitate and accelerate gradient computation. It is demonstrated in Chapter 3 that MLFC can be successfully trained to predict the behaviour of a simulated quadrotor with a simple ground effect model and noise for up to 5 seconds in 100Hz sampling period.

The MLFC architecture is an example of a deep RNN. Generally, deep neural networks perform better than the shallow ones on various learning tasks, including modeling dynamic systems as demonstrated by MLFC in this work. However, when introducing hidden layers, a problem that naturally arises is assigning proper initial values for the output of hidden layers (RNN states). The current method sets the initial values to zero or random numbers and runs the RNN until the effect of the initial values is washed out. This method, also known as the washout method, results in a transient response for an arbitrary length of time during which it does not represent the behaviour of the modeled system. Therefore, the washout method is not applicable when initializing RNNs for the multi-step prediction task in applications that are sensitive to the RNN transient response, such as designing a model-based controller. To fill the gap, the state initialization problem is defined and formulated (see Section 4.2). To address the state initialization problem, a history-based initialization method is proposed that employs neural networks as initializers of the RNN states. The proposed initialization method facilitates the training of RNNs in modeling dynamic systems for multi-step prediction and addresses the drawbacks of the washout method. Using the proposed initialization method, a variety of RNN architectures are trained and evaluated on experimental datasets. As the training converges successfully and the trained RNNs show appropriate transient response, the RNNs initialized by the

proposed method are therefore shown to be a good candidate to model dynamic systems for multi-step prediction. For the first time, a comprehensive study is presented which compares the behaviour of various RNN architectures, including LSTMs, on modeling dynamic systems from experimental data for multi-step prediction.

To learn the model of a dynamic system from experimental data, availability of a representative dataset is essential for developing and evaluating various modeling methods. Therefore, a quadrotor dataset consisting of more than 230 minutes indoor flight time in various regimes, including hover, moderate and aggressive manoeuvres, is collected and made publicly available. The dataset consists of various trajectories of the motor speeds (actual and commanded), body rate, and velocity. The vehicle body rate and velocity are measured using a precise motion capture system. However, the models trained for this work employ actual motor speeds to predict the velocity and body rate of the vehicle. The data distribution is presented as well as the collection process and time synchronization procedure.

In order to improve the prediction accuracy of the proposed black-box model, a grey-box modeling approach that employs a hybrid of a motion model module and two RNN modules is proposed. The approach leads to two architectures that, when trained on the quadrotor dataset, can provide accurate velocity and body rate predictions. The average prediction errors are less than 1 cm/sec and 1 degrees/sec, for a 0.4 second prediction window, and less than 3 cm/s and 2 degrees/sec, for a 1.9 second prediction window. Based on this work, it is clear that the benefits of employing RNNs in predicting motion for robotic platforms are significant, and will lead to better planning algorithms and more precise robot control systems.

In summary, the main contributions claimed in this thesis are listed as follows.

- Traditional RNN architectures in black-box modeling of a simulated quadrotor are implemented and assessed for multi-step prediction and extended to a novel structurally deep RNN [63, 64]. The proposed architecture models the simulated quadrotor solely on the input-output data for multi-step prediction providing accurate prediction over 500 time steps for a complex nonlinear MIMO system.

- Two novel deep architectures are proposed to initialize the states of any RNN with hidden neurons based on a history of observations; one is based on the FFNNs and the other on RNNs [67, 66]. The proposed methods enable RNNs to be trained on experimental flight data for which zero initial conditions are not possible for all flight segments.

- Traditional and gated RNN architectures, initialized by the proposed methods, are implemented and compared in system identification and modeling of two real aerial vehicles [66].

- A novel grey-box architecture is proposed which incorporates a motion model of a quadrotor with black-box models for multi-step prediction of a quadrotor vehicle [65]. The grey-box models lead to state of the art motion prediction capabilities for two robotic platforms.

6.1 Future Extensions

Given the versatility of the discussed methods in this work, a wide range of future extensions can be considered. In this section some of them are summarized.

- The developed model can be used in an online MPC. The main challenges for this extension fall in the hardware and software implementations.

- Wind rejection is another interesting and viable application using the models developed in this work. When operated outdoors, the observed behaviour is the result of wind acting on the vehicle as well as the quadrotor dynamics. Since the dynamic model of the quadrotor is improved with the method proposed in this work in the absence of wind, comparing the predicted and the observed behaviours should provide information about wind dynamics and helps developing methods to operate quadrotors in windy conditions.

- Inverse control is another interesting and feasible approach using the methods in this work. A controller can be trained using a black-box method if the input and output time series are switched. In fact, given a desired velocity and body rate trajectory, the network may be able to generate trajectories for the four motor speeds.

- More experiments can be run to see the effects of various hyper-parameters involved in this work. For instance, the initialization length was kept constant throughout the experiments. Extending the initialization length may contribute to a better early stage prediction, specifically for cases where an early jump in the prediction error is observed. Additionally, the effect of modifying the balance coefficient, corresponding to the initializer-predictor training cost, can be studied in order to improve the RNN transient response.

- Systems other than quadrotors can be considered to be modeled by the proposed architectures. For instance, in modeling human motion, primary movements can be modeled quite well. However, the number of DOFs and the dexterity in human hand

make the modeling of hand motions a challenging and interesting problem that can be tackled using the discussed methods. If successfully developed, the trained model can then be used in robotic hand movements.

- The models trained in this work were assessed over the collected datasets. As discussed, in a numerical modeling approach, the prediction accuracy depends on the representativeness of the dataset. Methods can be developed to automate trajectory generation in such a way that the state-space of the vehicle is covered uniformly. Richness of the collected dataset may be assessed with tools other than studying the distributions and therefore methods to improve the richness of the dataset can be devised.

- The code developed for this work is currently undergoing revisions to be publicly released. The code will provide a toolbox that implements the RNNs and methods discussed in this work.

References

[1] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[2] V. A. Akpan and G. D. Hassapis. Nonlinear model identification and adaptive model predictive control using neural networks. *ISA Transactions*, 50(2):177 – 194, 2011.

[3] R.K. Al Seyab and Yi Cao. Nonlinear system identification for predictive control using continuous time recurrent neural networks and automatic differentiation. *Journal of Process Control*, 18(6):568 – 581, 2008.

[4] S.R. Anderson, N.F. Lepora, J. Porrill, and P. Dean. Nonlinear dynamic modeling of isometric force production in primate eye muscle. *Biomedical Engineering, IEEE Transactions on*, 57(7):1554–1567, July 2010.

[5] Peter Andras. The equivalence of support vector machine and regularization neural networks. *Neural Processing Letters*, 15(2):97–104, 2002.

[6] J.C. Atuonwu, Y. Cao, G.P. Rangaiah, and M.O. Tade. Identification and predictive control of a multistage evaporator. *Control Engineering Practice*, 18(12):1418 – 1428, 2010.

[7] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, 2015.

[8] I. Baruch and C.R. Mariaca-Gaspar. A Levenberg-Marquardt learning applied for recurrent neural identification and control of a wastewater treatment bioprocess. *International Journal of Intelligent Systems*, 24:1094–1114, 2009.

[9] M. Basso, L. Giarre, S. Groppi, and G. Zappa. NARX models of an industrial power plant gas turbine. *Control Systems Technology, IEEE Transactions on*, 13(4):599–604, July 2005.

[10] VM Becerra, JMF Calado, PM Silva, and F Garces. System identification using dynamic neural networks: training and initialization aspects. *IFAC Proceedings Volumes*, 35(1):235–240, 2002.

[11] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[12] R Boné and M Crucianu. Multi-step-ahead prediction with neural networks: a review. *9emes rencontres internationales: Approches Connexionnistes en Sciences*, 2:97–106, 2002.

[13] H. Boudjedir, O. Bouhali, and N. Rizoug. Neural network control based on adaptive observer for quadrotor helicopter. *International Journal of Information Technology, Control and Automation*, 2(3):39–54, 2012.

[14] Patrick Bouffard, Anil Aswani, and Claire Tomlin. Learning-based model predictive control on a quadrotor: Onboard implementation and experimental results. In *Robotics and Automation (ICRA), IEEE International Conference on*, pages 279–284. IEEE, 2012.

[15] Sheng Chen, SA Billings, and PM Grant. Non-linear system identification using neural networks. *International journal of control*, 51(6):1191–1214, 1990.

[16] A. CC Coolen, R. Kühn, and P. Sollich. *Theory of neural information processing systems*. Oxford University Press, 2005.

[17] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, (2):303–314, 1989.

[18] A. Delgado, C. Kambhampati, and K. Warwick. Dynamic recurrent neural network for system identification and control. *Control Theory and Applications, IEE Proceedings*, 142(4):307 –314, July 1995.

[19] T. Dierks and S. Jagannathan. Output feedback control of a quadrotor uav using neural networks. *Neural Networks, IEEE Transactions on*, 21(1):50–66, 2010.

[20] Chris Eliasmith and Charles H Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT Press, 2004.

[21] Roy Featherstone. *Rigid body dynamics algorithms*. Springer, 2014.

[22] Thor I Fossen. *Marine control systems: guidance, navigation and control of ships, rigs and underwater vehicles*. Marine Cybernetics, 2002.

[23] Alessandro Freddi, Alexander Lanzon, and Sauro Longhi. A feedback linearization approach to fault tolerance in quadrotor vehicles. In *Proceedings of The 2011 IFAC World Congress, Milan, Italy*, 2011.

[24] K. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks.*, 2(3):183–192, May 1989.

[25] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801–806, 1993.

[26] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with LSTM recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

[27] Ian. Goodfellow, Yoshua. Bengio, and Aaron Courville. *Deep Learning*. 2017.

[28] Donald T Greenwood. *Principles of dynamics*. Prentice Hall, 1965.

[29] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *Neural networks and learning systems, IEEE Transactions on*, 2017.

[30] M.T. Hagan and M.B. Menhaj. Training feedforward networks with the Marquardt algorithm. *Neural Networks, IEEE Transactions on*, 5(6):989–993, 1994.

[31] Eric J Hartman, James D Keeler, and Jacek M Kowalski. Layered neural networks with gaussian hidden units as universal approximations. *Neural computation*, 2(2):210–215, 1990.

[32] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice-Hall, 1999.

[33] G. Hinton, L. Deng, D. Yu, G. E Dahl, A-R Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[34] G.E. Hinton and R.R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[35] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[36] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. A field guide to dynamical recurrent neural networks. IEEE Press, 2001.

[37] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin. Precision flight control for a multi-vehicle quadrotor helicopter testbed. *Control engineering practice*, 19(9):1023–1036, 2011.

[38] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks.*, 2(5):359–366, July 1989.

[39] Herbert Jaeger. *Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the "echo state network" approach*, volume 5. GMD-Forschungszentrum Informationstechnik, 2002.

[40] Liang Jin, Peter N Nikiforuk, and Madan M Gupta. Approximation of discrete-time state-space trajectories using dynamic recurrent neural networks. *Automatic Control, IEEE Transactions on*, 40(7):1266–1270, 1995.

[41] Dominic William Jordan and Peter Smith. *Nonlinear ordinary differential equations: an introduction to dynamical systems*, volume 2. Oxford University Press, USA, 1999.

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

[43] J. F. Kolen and S. C. Kremer. *A Field Guide to Dynamical Recurrent Networks*. IEEE Press, New York, 2001.

[44] Bart Kosko. Fuzzy systems as universal approximators. *Computers, IEEE Transactions on*, 43(11):1329–1333, 1994.

[45] M. V. Kumar, S.N. Omkar, R. Ganguli, P. Sampath, and S Suresh. Identification of helicopter dynamics using recurrent neural networks and flight data. *Journal of the American Helicopter Society*, 51(2):164–174, 2006.

[46] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[47] Ian Lenz, Ross A Knepper, and Ashutosh Saxena. DeepMPC: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*, 2015.

[48] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *The Quarterly of Applied Mathematics*, (2):164–168, 1994.

[49] Hong-Xing Li and CL Philip Chen. The equivalence between fuzzy logic systems and feedforward neural networks. *Neural Networks, IEEE Transactions on*, 11(2):356–365, 2000.

[50] Shuhui Li. Wind power prediction using recurrent multilayer perceptron neural networks. In *Power Engineering Society General Meeting*, volume 4. IEEE, July 2003.

[51] Xiaoou Li and Wen Yu. Dynamic system identification via recurrent multilayer perceptrons. *Information Sciences*, 147(1):45–63, 2002.

[52] Zhan Li, M. Hayashibe, C. Fattal, and D. Guiraud. Muscle fatigue tracking with evoked EMG via recurrent neural network: Toward personalized neuroprosthetics. *Computational Intelligence Magazine, IEEE*, 9(2):38–46, May 2014.

[53] Tsungnan Lin, B.G. Horne, P. Tino, and C.L. Giles. Learning long-term dependencies in NARX recurrent neural networks. *Neural Networks, IEEE Transactions on*, 7(6):1329–1338, Nov 1996.

[54] Lennart Ljung. Perspectives on system identification. *Annual Reviews in Control*, 34(1):1–12, 2010.

[55] Lennart Ljung and Torsten Söderström. *System identification*. MIT Press, 1983.

[56] Lennart Ljung, Qinghua Zhang, Peter Lindskog, and Anatoli Juditski. Estimation of grey box and black box models for non-linear circuit data. *IFAC Proceedings Volumes*, 37(13):399–404, 2004.

[57] J.M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002.

[58] T. Madani and A. Benallegue. Adaptive control via backstepping technique and neural networks of a quadrotor helicopter. In *Proceedings of the 17th World Congress of The International Federation of Automatic Control*, 2008.

[59] D. P. Mandic and J. A. Chambers. *Recurrent Neural Networks for Prediction: learning algorithms, architectures and stability*. John Wiley & Sons, 2001.

[60] Donald W Marquardt. An algorithm for Least-Squares estimation of nonlinear parameters. *Journal of the Society for Industrial & Applied Mathematics*, 11(2):431–441, 1963.

[61] J. Martens and I. Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade*, pages 479–535. Springer, 2012.

[62] W.S McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[63] N. Mohajerin and Steven L. Waslander. Modular deep recurrent neural network: Application to quadrotors. In *Systems, Man and Cybernetics, IEEE International Conference on*, 2014.

[64] N. Mohajerin and Steven L. Waslander. Modelling a quadrotor vehicle using a modular deep recurrent neural network. In *Systems, Man and Cybernetics, IEEE International Conference on*, 2015.

[65] Nima Mohajerin, Melissa Mozifian, and Steven L. Waslander. Deep learning a real quadrotor model for multi-step prediction. In *Submitted to Robotics and Automation (ICRA), IEEE International Conference on*. IEEE, 2018.

[66] Nima Mohajerin and Steven L. Waslander. Modeling transient response of non-linear dynamic systems using recurrent neural networks. *Submitted to Neural Networks and Learning Systems, IEEE Transactions on*.

[67] Nima Mohajerin and Steven L Waslander. State initialization for recurrent neural network modeling of time-series data. In *Neural Networks (IJCNN), International Joint Conference on*, pages 2330–2337. IEEE, 2017.

[68] K.S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *Neural Networks, IEEE Transactions on*, 1(1):4 –27, March 1990.

[69] Oliver Nelles. *Nonlinear system identification: from classical approaches to neural networks and fuzzy models*. Springer Science & Business Media, 2013.

[70] O. Nerrand, P. Roussel-Ragot, D. Urbani, L. Personnaz, and G. Dreyfus. Training recurrent neural networks: why and how? an illustration in dynamical process modeling. *Neural Networks, IEEE Transactions on*, 5(2):178–184, 1994.

[71] C. Nicol, C.J.B. Macnab, and A. Ramirez-Serrano. Robust adaptive control of a quadrotor helicopter. *Mechatronics*, 21(6):927–938, 2011.

[72] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[73] Olalekan Ogunmolu, Xuejun Gu, Steve Jiang, and Nicholas Gans. Nonlinear systems identification using deep dynamic neural networks. *arXiv preprint arXiv:1610.01439*, 2016.

[74] Jairo Jose Espinosa Oviedo, Joos PL Vandewalle, and Vincent Wertz. *Fuzzy logic, identification and predictive control*. Springer Science & Business Media, 2006.

[75] A.G. Parlos, K.T. Chong, and A.F. Atiya. Application of the recurrent multilayer perceptron in modeling complex process dynamics. *Neural Networks, IEEE Transactions on*, 5(2):255–266, March 1994.

[76] Alexander G Parlos, Omar T Rais, and Amir F Atiya. Multi-step-ahead prediction using dynamic recurrent neural networks. *Neural Networks*, 13(7):765–786, 2000.

[77] B.A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: a survey. *Neural Networks, IEEE Transactions on*, 6(5):1212–1228, Sep 1995.

[78] Ali Punjani and Pieter Abbeel. Deep learning helicopter dynamics models. In *Robotics and Automation (ICRA), IEEE International Conference on*, pages 3223–3230. IEEE, 2015.

[79] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[80] Anton Maximilian Schäfer and Hans Georg Zimmermann. Recurrent neural networks are universal approximators. In *Artificial Neural Networks–ICANN*, pages 632–640. Springer, 2006.

[81] J. Schmidhuber and S. Hochreiter. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[82] H.T. Siegelmann, B.G. Horne, and C.L. Giles. Computational capabilities of recurrent NARX neural networks. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 27(2):208–215, Apr 1997.

[83] Sho Sonoda and Noboru Murata. Neural network with unbounded activation functions is universal approximator. *Applied and Computational Harmonic Analysis*, 2015.

[84] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[85] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, Inception-ResNet and the impact of residual connections on learning. In *AAAI*, pages 4278–4284, 2017.

[86] Z. Taha, A. Deboucha, and M Bin Dahari. Small-scale helicopter system identification model using recurrent neural networks. In *TENCON IEEE Region 10 Conference*, pages 1393–1397. IEEE, 2010.

[87] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[88] Mark K Transtrum and James P Sethna. Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization. *arXiv preprint arXiv:1201.5885*, 2012.

[89] Holger Voos. Nonlinear control of a quadrotor micro-UAV using feedback-linearization. In *Mechatronics, ICM. IEEE International Conference on*, pages 1–6. IEEE, 2009.

[90] R Williams and D Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.

[91] Jun Wu, Hui Peng, Qing Chen, and Xiaoyan Peng. Modeling and control approach to a distinctive quadrotor helicopter. *ISA Transactions*, 53(1):173–185, 2014.

[92] Ronald R Yager and Lotfi A Zadeh. *An introduction to fuzzy logic applications in intelligent systems*, volume 165. Springer Science & Business Media, 2012.

[93] Matthew D Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

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

[95] Hans-Georg Zimmermann, Christoph Tietz, and Ralph Grothmann. Forecasting with recurrent neural networks: 12 tricks. In *Neural Networks: Tricks of the Trade*, pages 687–707. Springer, 2012.