UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

RÓGER MATEUS SEHNEM

NONLINEAR CONTROLLER DESIGN USING UNIVERSAL APPROXIMATORS THROUGH THE VIRTUAL REFERENCE METHOD

Porto Alegre
2023

RÓGER MATEUS SEHNEM

NONLINEAR CONTROLLER DESIGN USING UNIVERSAL APPROXIMATORS THROUGH THE VIRTUAL REFERENCE METHOD

Thesis presented to Programa de Pós-Graduação em Engenharia Elétrica of Universidade Federal do Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Electrical Engineering.
Area: Control and Automation

ADVISOR: Prof. Dr. Alexandre Sanfelici Bazanella

Porto Alegre
2023

RÓGER MATEUS SEHNEM

NONLINEAR CONTROLLER DESIGN USING UNIVERSAL APPROXIMATORS THROUGH THE VIRTUAL REFERENCE METHOD

This thesis was considered adequate for obtaining the degree of Master in Electrical Engineering and approved in its final form by the Advisor and the Examination Committee.

Advisor: _____
Prof. Dr. Alexandre Sanfelici Bazanella, UFRGS
Doctor by the Federal University of Santa Catarina, UFSC - Florianópolis, Brazil

Examination Committee:

Prof. Dr. Fabrício Gonzalez Nogueira, UFCE
Doctor by the Federal University of Pará – Belém, Brazil

Prof. Dr. João Manoel Gomes da Silva Jr., UFRGS
Doctor by the Université Paul Sabatier – Tolouse, France

Prof. Dr. Diego Eckhard, UFRGS
Doctor by the Federal University of Rio Grande do Sul – Porto Alegre, Brazil

Coordinator of PPGEE: _____
Prof. Dr. Jeferson Vieira Flores

Porto Alegre, November 2023.

ACKNOWLEDGMENTS

I am grateful for the quality public education that allowed me to get here.

To my parents, Claudio and Roselita, my brother João, my cousins Josué, Josiane and Jonathan, and all of my friends, especially Linda, Ana, Paula, Kenedy and Jamie, for inspiring and supporting me to follow my dreams.

To my advisor, Professor Alexandre Bazanella, for the guidance and freedom that allowed me to pursue my interests.

To my partner, Livia Alves, who supported and inspired me throughout this work.

This work was realized with the support of Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

ABSTRACT

This work deals with the usage of universal approximators, mainly Deep Neural Networks (DNNs), as parametrizations for the controller of nonlinear dynamical systems in the context of the Virtual Reference Feedback Tuning. One of the challenges in the utilization of the VRFT method is the definition of the controller parametrization. Since nonlinear systems present a very rich behavior, defining a parametrization that allows to approximate the ideal controller can be a very complex task. With the intention to ease this definition, the usage of DNNs was proposed, which are known for being universal approximators. However the literature uses a filter that is quite limiting for the practical application of the method with DNNs which, in addition to being computationally costly, requires a plant model to be identified to apply the method, going against the data-driven principles. For this reason, in this work, the usage of DNNs with the linear VRFT filter is evaluated. Since regularization techniques are a key component in the usage of DNNs, they are evaluated and utilized not only with DNNs, but also with simpler parametrizations, linear in the parameters, allowing an easier understanding of its effects. The study results are illustrated with the application of the proposed method, using simulations, in two different nonlinear systems, and suggest that the proposed approach is viable for obtaining data-based controllers. A new analysis is also proposed for the database, using the concept of virtual reference is possible to create what would be the ideal database for the controller identification. This ideal database is compared, using heatmaps, with the database obtained experimentally, using the VRFT method. This analysis is used, in all case studies, to explain the controller's performance.

Keywords: Nonlinear control, virtual reference feedback tunning, deep neural network, universal approximators.

RESUMO

Este trabalho trata da utilização de aproximadores universais, principalmente Redes Neurais Profundas (RNP), como parametrização para o controlador de sistemas dinâmicos não lineares no contexto do método Virtual Reference Feedback Tuning (VFRT). Um dos desafios na utilização do método VRFT é a definição da parametrização do controlador, como sistemas não lineares apresentam comportamento bastante rico, definir uma parametrização que permite aproximar o controlador ideal pode ser uma tarefa bastante complexa. Com a intenção de facilitar esta definição, foi proposta utilização de RNPs conhecidas por serem aproximadores universais. Contudo a literatura utiliza um filtro bastante limitante para a aplicação prática com RNPs que, além de computacionalmente custoso, necessita que um modelo da planta seja identificado para a aplicação do método, indo contra os princípios do controle baseado em dados. Por esta razão, neste trabalho, é avaliada a utilização de RNPs com o filtro do VRFT linear. Como técnicas de regularização são uma componente chave na utilização de RNPs, as mesmas são avaliadas e utilizadas não somente com RNPs, mas também com parametrizações lineares nos parâmetros, permitindo um entendimento facilitado dos seus efeitos. Os resultados do estudo são ilustrados com a aplicação do método proposto, por meio de simulações, em dois sistemas não lineares diferentes, e sugerem que a abordagem proposta é viável para a obtenção de controladores baseado em dados. É proposta, também, uma nova análise da base de dados, utilizando o conceito da referência virtual é possível montar o que seria a base de dados ideal para a identificação do controlador. Essa base de dados ideal é comparada, por mapas de calor, com a base obtida experimentalmente, através do método VRFT. Esta análise é utilizada para explicar, em todos os estudos de caso, o desempenho dos controladores.

Palavras-chave: Controle não linear, virtual reference feedback tunning, rede neural profunda, aproximadores universais.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADAM	Adaptative Moments Estimation
ANN	Artificial Neural Network
BPTT	Backpropagation Through Time
BP	Backpropagation
CBT	Correlation Based Tuning
DBN	Deep Belief Network
DC	Direct Current
DD	Data-Driven
DNN	Deep Neural Network
FDT	Frequency Domain Tuning
GRU	Gated Recurrent Unit
IFT	Iterative Feedback Tuning
IO	Input-Output
LASSO	Least Absolute Shrinkage and Selection Operator
LQR	Linear Quadratic Regulator
LSTM	Long Short Term Memory
MBC	Model Based Control
MIMO	Multi-Input Multi-Output
MPL	Multi Layer Perceptron
MRAC	Model Reference Adaptive Control
MRC	Model Reference Control
NADAM	Nesterov Adaptative Moments Estimation
OCI	Optimal Controller Identification
PID	Proportional Integral Derivative
PRBS	Pseudo Random Binary Sequence
RHS	Right Hand Side

RL	Reinforcement Learning
RNN	Recurrent Neural Networks
RNP	Rede Neural Profunda
ReLU	Rectified Linear Unit
S2S	Sequence-to-Sequence
S2V	Sequence-to-Vector
SGD	Stochastic Gradient Descent
SISO	Single-Input Single-Output
SLP	Single Layer Perceptron
TTSSDM	Taylor Series Sampled Data Model
VRFT	Virtual Reference Feedback Tuning
XOR	Exclusive Or
ZOH	Zero Order Hold

CONTENTS

1 INTRODUCTION

1.1 Motivation

One of the most distinct characteristic of life is the ability to adapt itself to the most diverse environments. The examples are numerous, to cite two, birds can adapt their wings to keep them on their desired track and warm-blooded animals often use many mechanisms, such as sweat and altering blood flow patterns, to keep their internal temperature at an optimal level. This ability is even more impressive when we consider the complexity of those organisms. This self-regulating property gives them the ability to, at least to some extent, be independent of any external input, for instance, we do not spend much thinking into when or how our bodies should be sweating, this process happens automatically.

This amazing self-regulating characteristic is very desirable in virtually any human-made system and is what automatic control aims to do. In a more pragmatic view, what automatic control does is to create a controller that acts on a system, this controller is simply a function that, given some measured outputs of the system, calculates adequate inputs to it, such that the system's outputs behave in a desirable manner. Automatic control is used in many areas of different complexities, going from the temperature control of a refrigerator to the autopilot of a spacecraft with very successful applications.

Most control design methods rely on the precise knowledge of the system: a model, which comes in the form of a differential or difference equation, that describes the system's variables evolution in time. With a model of the system, it is possible to use many control design methods to define the controller that gives the system the intended self-regulatory behavior. The design methods that rely on the system's model to formulate the controller are deemed Model Based Control (MBC) techniques.

To use any MBC technique a system model is required, this model can be obtained by knowing the physics and fundamental principles that are applicable, by system identification or by a combination of those two. System identification builds models from observed input-output (IO) data of the system, and it is an active research area that goes further than discovering models solely for the purpose of control. More about system identification can be found on (LJUNG, 1999; SÖDERSTRÖM; STOICA, 1989).

When identification is used for control some considerations needs to be made with respect to the model structure and order. Also, it is important to see if the controller defined using this model performs well on the real system. Those aspects make a compelling argument for using data-driven (DD) control design methods, where the controller is directly found from the IO data. With it there is no need to identify a model to design a controller, which can potentially save a lot of engineering time in designing controllers for systems with an unknown model.

One potential downside of using DD control design methods is that in realistic applications, since the system dynamics is unknown and corrupted by noise, there are not, usually, stability guarantees. This means that the found controllers need, at least in the beginning, to be implemented in closed loop with caution.

Another potential downside is on applying DD control design methods on unstable systems. A priori, all DD methods can be applied on unstable systems. The problem, however, is on how to excite the system, since the system is unstable, unless it is already stabilized by some controller, defining an input trajectory that keeps the system inside the operational bounds can be quite difficult.

There are many data-driven control methods in the literature, such as the Ziegler-Nichols tuning method, Iterative Feedback Tuning (IFT), Correlation Based Tuning (CBT), Reinforcement Learning (RL), Frequency Domain Tuning (FDT), Optimal Controller Identification (OCI) and the Virtual Reference Feedback Tuning (VRFT). It is relevant to have a sense on how those DD methods work, what are their particularities, pros and cons. For this reason, in the next section, some previous works on DD are presented.

1.2 Previous work on DD control design methods

The Ziegler-Nichols method (ZIEGLER; NICHOLS, 1942) is a classic method for tuning controllers. It was originally developed to tune controllers of the Proportional Integral Derivative (PID) class, requiring a simple experiment on the system to reveal its ultimate gain and oscillation period. From those, the controller's parameters are calculated from a table. The method has limited effectiveness since the table itself is a heuristic that works for most systems analyzed by Ziegler and Nichols, and it might not work well for systems with significantly different characteristics than those analyzed.

In the IFT method (HJALMARSSON; GUNNARSSON; GEVERS, 1994) the control objective is to make the output of the unknown system close to a desired output, which can be given by a reference model. The unknown system is controlled by a controller of fixed structure and variable parameters. Those are, in turn, selected by minimizing a cost function that penalizes the error between the output of the closed loop and the desired system.

The IFT method is closely related to the Model Reference Adaptive Control (MRAC)

(ÅSTRÖM; WITTENMARK, 2013) when the method is used with the reference model, instead of being used with only the desired output, and when the cost function is quadratic (HJALMARSSON *et al.*, 1998). The need in the IFT method to use a sequence of experiments to calculate the controller's parameters is why it is named an iterative method.

The CBT method adjusts the parameters of an initial stabilizing controller, often initially designed for a reduced-order model of the system. It does that by means of minimizing a cost function that measures the correlation of the closed-loop output error, i.e., the difference of the desired and the achieved outputs of the system, with the reference signal (KARIMI; MIŠKOVIĆ; BONVIN, 2004).

In the Reinforcement Learning (RL) approach the objective is to solve a discrete time optimal control problem where the system can be unknown. As in the usual optimal control problem setup, a cost function is defined and the control objective is to extremize this cost function (BUŞONIU *et al.*, 2018). In the RL approach the cost function is often defined as the summation of the reward values. The RL methodology can be applied when the system dynamics is unknown, one of the most famous such methodology is known as Q-learning, where the Q-value, a representation of the rewards, is given by a recurrence equation that depends only on the observed states and rewards (WATKINS; DAYAN, 1992). To deal with continuous systems the Q-functions and the control policy can be approximated by a parametrized version. When those parametrizations are given by Deep Neural Networks, the method is named Deep Reiforcement Learning. This approach is, in part, the technique used by the DeepMind team in the challenging problem of creating a controller capable of playing more than 40 Atari 2600 games using only the pixels and game scores as inputs (MNIH *et al.*, 2015).

The Frequency Domain Tuning (FDT) method can be seen as a frequency domain variant of the IFT algorithm (KAMMER; BITMEAD; BARTLETT, 2000). The cost function to be minimized penalizes the output size and the control size and the controller has a fixed structure with some unknown vector of parameters. As in the IFT method, an estimate of the derivative of the cost function is made using IO data and an extra experiment with an specific reference. In the FDT, however, the cost function derivatives with respect to the parameters are calculated via a spectral analysis of the closed-loop experimental data.

The Optimal Controller Identification (OCI) method was presented in (CAMPESTRINI *et al.*, 2017) for linear SISO systems. OCI is a model reference control design method, which requires the definition of a parametrization of the controller. To find the parameters that make the closed loop system behave as the reference model, the unknown dynamics is parametrized as a function of the controller parameters and the model is then identified using standard prediction error identification, since the model is parametrized only by the controller parameters, the identification returns directly the optimal parameters.

The VRFT method was first introduced in (CAMPI; LECCHINI; SAVARESI, 2002)

for linear systems and further extended to nonlinear systems in (CAMPI; SAVARESI, 2006). Similar to the IFT method, the objective is to make the output of the unknown system close to the reference output by tuning the parameters of a predefined control structure, however, the VRFT method requires that this reference output is given by a known reference model.

The basic idea of the method is to interpret the IO data as if it was taken from the reference model. This allows, via the inverse dynamics of the reference model, to calculate the virtual reference, i.e., the reference fed to the reference model such that the output is the same as the one from the collected output. This approach transforms the problem of identifying the system dynamics to tune a controller into a problem of directly identifying the controller that achieves a desired closed loop performance.

1.3 Contribution and organization

In order to use the VRFT method the user has first, as will be seen, to define a controller parametrization. This is not a trivial thing to do since the ideal controller, i.e., the one that when in closed-loop with the system makes it behave as the reference model, needs to be at least almost achievable using the defined parametrization. The problem is that the ideal controller class is not known, since to know it precisely one needs to also know the system's class, an information that is unavailable by hypothesis. This problem is even bigger when dealing with nonlinear systems, as will be the case in this work.

Since linearly parametrized controllers have a closed form solution in the VRFT method, that is usually the chosen parametrization form and was the one in which the nonlinear extension of the method was presented (CAMPI; SAVARESI, 2006). The problem is that for some nonlinear system, this ideal linear parametrization might need too many parameters[1] and accompanying terms in the dictionary, which can be cumbersome to define in a realistic application.

Deep Neural Networks (DNNs) are known for being universal approximators (CHARU, 2018; GÉRON, 2022; GOODFELLOW; BENGIO; COURVILLE, 2016; MONTÚFAR, 2014), and thus offer an interesting option of being used as the controller's parametrization, which is also the approach used in Deep Reinforcement Learning. In this work, the usage of DNNs and polynomial basis will be explored and compared in the context of the nonlinear VRFT method.

The work is organized as follows, first, some concepts and theorems that pertain the nonlinear VRFT method are present in Chapter 2, where the system is formally defined, the control architecture is fixed and the VRFT algorithm is presented in an ideal scenario, then the method properties are present in more challenging and realistic scenarios.

Chapter 3 defines the controller parametrizations of relevance to the work, together

[1] Possibly infinitely many.

with the analysis of their properties. There, polynomial basis are defined along with an analysis of their use as controller parametrizations, then general DNNs are presented, followed by the Recurrent Neural Networks (RNNs), which are a specialized type of DNNs well suited for modeling dynamical systems such as controllers, training algorithms and most of the relevant concepts that allow the usage of DNNs are also presented and discussed. One important concept that is present is Regularization, which is introduced for general DNNs and discussed specifically in the context of the VRFT method.

The nonlinear VRFT method is then applied to two different dynamical systems in chapter 4, where some different controller parametrizations are tested and compared in order to get a more concrete sense of the method's effectiveness and properties when used with nonlinear parametrizations. The work ends with general conclusions in Chapter 5.

2 VIRTUAL REFERENCE FEEDBACK TUNING

The Virtual Reference Feedback Tuning (VRFT) is a method of the class of data-driven control methods. The main feature of this class consists in tuning the parameters of a class of controllers for a given system using only input-output (IO) data collected in a experiment realized on it.

In this chapter the ideas and concepts that pertain the nonlinear VRFT method will be developed to give a solid base that will latter be used to create a data-based nonlinear controller. In the next section, the system that will be controlled is defined along with some notation. The next section discusses the control architecture and what exactly will be the controller that will be tuned by the VRFT method.

With these definitions in hand, the VRFT algorithm is presented and discussed on a general nonlinear setup, its main concepts, namely the virtual reference and error, are presented along with the theorems that validate the approach and the use of the method in a more realistic setup, that includes noise and lack of representational power of the controller parametrization.

2.1 System Definition

In the Data-Based Control approach the objective is to control a system without knowing its dynamics. It is important to define it, however, to make clear the general properties and notation that will be used to develop the results that make the nonlinear VRFT. This is also important to make explicit the assumptions made in the process.

The system that will be controlled and from which data will be collected is continuous, nonlinear and single-input single-output (SISO), with states $\boldsymbol{x}(t)$, control $\mathrm{u(t)}$ and output $\mathrm{y(t)}$. Since the control action is calculated at constant intervals and one usually have no access to all states, the input-output behavior of the continuous system is approximated by a discrete time SISO nonlinear system. This unknown nonlinear system of interest have, similarly, an input $u(t)$ and output $y(t)$, and is defined as

$$y(t) = \mathcal{S}[y(t)\boldsymbol{q}_{n_s}, u(t)\boldsymbol{q}_{n_s}] + \nu(t), \tag{1}$$

where the time $t \in \mathbb{Z}$, the map $\mathcal{S} : \mathbb{R}^{n_s} \times \mathbb{R}^{n_s} \to \mathbb{R}$ and the measurement noise $\nu(t) \in \mathbb{R}$

is a stationary process. The backward shift vector is defined as

$$\boldsymbol{q}_n^a = \begin{bmatrix} q^{-a} & q^{-(a+1)} & q^{-(a+2)} & \cdots & q^{-n} \end{bmatrix}^T, \tag{2}$$

where $a \leq n$ with $a, n \in \mathbb{W}$ and q is the forward shift operator, e.g. $qy(t) = y(t+1)$ and $q^{-1}y(t) = y(t-1)$. To ease the notation, whenever $a = 1$, the backward shift vector is simply $\boldsymbol{q}_n$.

It is a well known result that when the continuous system is linear, it can be represented exactly by a discrete time counterpart. However, when that is not the case, in general, no exact representation exists and an approximate one hast to be used instead.

This approximate representation is sufficient for control purposes as long as the error is small enough, a measure that depends on the specific system and control requirements. An interesting and useful result that validates using such approximation is the Truncated Taylor Series Sampled Data Model (TTSSDM) presented in (YUZ; GOODWIN, 2005). As shown in (CARRASCO; GOODWIN; YUZ, 2012), using this model, the Global Vector Fixed Time Truncation Error, i.e., the error between each state in the continuous and discrete time models after evolved for a fixed time, can be made arbitrarily small given that the sampling frequency can be made arbitrarily big.

Another representation is given in (SCHOUKENS; RELAN; SCHOUKENS, 2017), however this one has less strict requirements on the properties of the control signal. This approximate representation also allows for arbitrarily small errors but with a smaller sampling frequency than required using the TTSSDM.

The above discussion motivates the following assumption

Assumption 2.1.1. *The continuous output is close to the discrete output in the sampled times, i.e.* $y(t_i) - \mathrm{y}(t_i) \approx 0, \ \forall t_i$.

It should be noted, however, that no specific discrete model such as TTSSDM is enforced. The argument for the Assumption 2.1.1 is made in the sense that some discrete time representation of continuous the system exists with bounded and sufficiently small errors.

2.2 Control Architecture

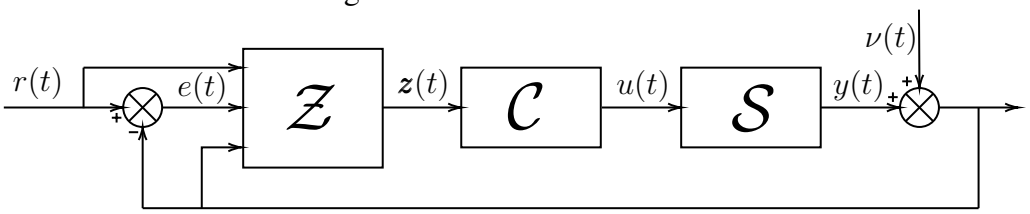
The control action, when applied to $\mathcal{S}$, is calculated in constant time intervals and applied in the sample and hold fashion, i.e., using ZOH to transform the discrete signal to a continuous one. Thus, unlike the system model, which is continuous by nature, the control action can be thought directly as a discrete time system.

The control architecture is shown in the Figure 1, where $\mathcal{C}$ is the control map and $\mathcal{Z}$ is the measure map, both are discrete time nonlinear systems.

The control $u(t)$ is generated by

$$u(t;\boldsymbol{\theta}) = \mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \boldsymbol{z}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}], \tag{3}$$

Figure 1 – Control architecture



Source: author

where $\boldsymbol{z}(t) \in \mathbb{R}^{m_z}$ is a vector of measurements of the system with m_z measurements, going all n_c samples in the past, $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$ is the parameter vector, and, for a given $\boldsymbol{\theta}$, the map $\mathcal{C} : \mathbb{R}^{n_c \times m_z} \times \mathbb{R}^{n_c} \to \mathbb{R}$. The error is defined as

$$e(t) = r(t) - y(t), \tag{4}$$

where $r(t)$ is the reference signal.

Similarly to the control, the measurement vector $\boldsymbol{z}(t)$ is given by

$$\boldsymbol{z}(t) = \mathcal{Z}[r(t)\boldsymbol{q}^0_{n_z-1}, e(t)\boldsymbol{q}^0_{n_z-1}, y(t)\boldsymbol{q}^0_{n_z-1}], \tag{5}$$

where the map $\mathcal{Z} : \mathbb{R}^{n_z} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_z} \to \mathbb{R}^{m_z}$.

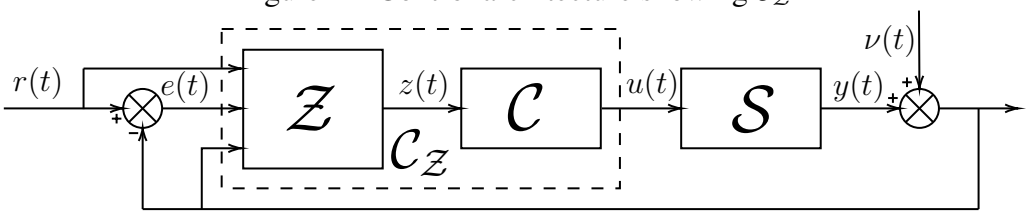
Unlike $\mathcal{C}$, the measurement map is not parameterized and, instead, is completely defined by the user. This definition should be made in a way that make the solution of the VRFT problem simpler in the control parametrization, e.g, if the control objective is to have a zero steady state error, one interesting measurement to add to $\mathcal{Z}$ is the integrated error signal $z_I(t)$ as in

$$z_I(t) = \sum_{\tau=0}^{t} e(\tau), \tag{6}$$

where it is assumed that n_z depends on t in (5).

To see the role of the measurement map it is interesting to see a control architecture similar to the one in Figure 1 but with $\mathcal{Z}$ and $\mathcal{C}$ as a single system, which is shown in Figure 2.

Figure 2 – Control architecture showing $\mathcal{C}_\mathcal{Z}$



Source: author

In Figure 2, the parameterized control system is now $\mathcal{C}_\mathcal{Z}$, in this fashion the control is given by

$$u(t;\boldsymbol{\theta}) = \mathcal{C}_\mathcal{Z}[r(t)\boldsymbol{q}^0_{n_z-1}, e(t)\boldsymbol{q}^0_{n_z-1}, y(t)\boldsymbol{q}^0_{n_z-1}; \boldsymbol{\theta}], \tag{7}$$

where, for a given $\boldsymbol{\theta}$, the map $\mathcal{C}_{\mathcal{Z}} : \mathbb{R}^{n_z} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_z} \to \mathbb{R}$.

If the goal was, for instance, to find a set of parameters $\boldsymbol{\theta}$ that makes $\mathcal{C}_{\mathcal{Z}}$ a zero steady state error controller, $\boldsymbol{\theta}$ and $\mathcal{C}_{\mathcal{Z}}$ would have to, internally, calculate the integrated error signal. With the approach of Figure 1, it is possible to directly create the signals that are known to be necessary to give the controller the desired closed loop performance. Since $\mathcal{C}$ is found by minimizing a cost function, defining the measurement map in this fashion has the result of decreasing the complexity of $\mathcal{C}$[1], thus decreasing the optimization problem complexity.

2.3 VRFT Algorithm

If the unknown system of interest in Eq. (1) is operated in closed loop with the controller in Eq. (7), its closed loop dynamics would be expressed as

$$y(t;\boldsymbol{\theta}) = \mathcal{S}[y(t;\boldsymbol{\theta})\boldsymbol{q}_{n_s}, \mathcal{C}_{\mathcal{Z}}[r(t)\boldsymbol{q}^0_{n_z-1}, e(t)\boldsymbol{q}^0_{n_z-1}, y(t)\boldsymbol{q}^0_{n_z-1}; \boldsymbol{\theta}]\boldsymbol{q}_{n_s}] + \nu(t), \tag{8}$$

where the dependence on the controller parametrization is made explicit.

The objective of the VRFT method is to perform a model reference control without having to resort on the knowledge of the system model $\mathcal{S}$ in Eq. (1). That is, it is desired to control the system $\mathcal{S}$ such that it behaves as the reference system $\mathcal{S}_r$, defined as

$$y_r(t) = \mathcal{S}_r[y_r(t)\boldsymbol{q}_{n_{s_r}}, r(t)\boldsymbol{q}_{n_{s_r}}], \tag{9}$$

where y_r is the reference output.

Thus, in the Model Reference Control (MRC) paradigm, the control objective can be expressed, using Eq. (8) and Eq. (9), as minimizing the following cost function

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{t=1}^{N} E\left[y_r(t) - y(t;\boldsymbol{\theta})\right]^2, \tag{10}$$

where N is the number of samples, both systems are operating with the same reference $r(t)$ and E denotes the expected value. Since the number of samples N is often big, (10) can be seen as

$$J(\boldsymbol{\theta}) = \lim_{N\to\infty} \frac{1}{N}\sum_{t=1}^{N} E\left[y_r(t) - y(t;\boldsymbol{\theta})\right]^2 = \bar{E}\left[y_r(t) - y(t;\boldsymbol{\theta})\right]^2. \tag{11}$$

The cost function in Eq. (10) has its global minimum when the system $\mathcal{S}$ is operated in closed loop with the ideal controller $\mathcal{C}_r$, i.e., the controller that makes $y_r(t) = y(t)$.

There are two problems with the cost function in Eq. (10), the first is that it depends, via $y(t;\boldsymbol{\theta})$, on the system model. However, in the data-based control paradigm, we only have access to the system via an experiment realized on it. More specifically, we have only

[1] In the integrator example, since $\mathcal{C}_{\mathcal{Z}}$ needs a integrator, if $\mathcal{Z}$ contains one, $\mathcal{C}$ does not need one.

its input-output data $y(t), u(t),\ t = 1, \cdots, N$. The second is that it is non-convex even for linear systems with linearly parametrized controllers (CAMPI; SAVARESI, 2006; BAZANELLA; CAMPESTRINI; ECKHARD, 2011).

To deal with these problems, in the VRFT approach, we define the virtual reference $\bar{r}(t)$, using Eq. (9), as

$$y(t) = \mathcal{S}_r[y(t)\boldsymbol{q}_{n_{s_r}}, \bar{r}(t)\boldsymbol{q}_{n_{s_r}}]. \tag{12}$$

This mean that the virtual reference is the signal that would have to be feed to the reference system $\mathcal{S}_r$ to make its output the same as the observed output in the actual experiment, i.e., $y(t)\ t = 1, \cdots, N$. The virtual reference is baptized as such because this signal was never in place in the actual experiment.

Similarly, it is possible to define the virtual error, using Eq. (4), as

$$\bar{e} = \bar{r}(t) - y(t). \tag{13}$$

The central idea of the VRFT method is to perceive the input-output data as if was taken from system $\mathcal{S}$ operating in closed loop with the ideal controller $\mathcal{C}_r$. With this, it is possible to transform the problem of minimizing $J(\boldsymbol{\theta})$ into the problem of minimizing the controller error $J^V(\boldsymbol{\theta})$. To this end, as in the linear case (CAMPI; LECCHINI; SAVARESI, 2002; BAZANELLA; CAMPESTRINI; ECKHARD, 2011), the nonlinear VRFT method uses, instead of Eq. (10), a cost function that minimizes the controller error

$$J^V(\boldsymbol{\theta}) = \bar{E}\left[u(t) - \mathcal{C}[\boldsymbol{q}^0_{n_c-1}\bar{\boldsymbol{z}}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}]\right]^2 \tag{14}$$

where $\bar{\boldsymbol{z}}(t)$ is virtual version of the measurement vector, i.e., where every instance of the reference signal and the error is replaced by the virtual reference $\bar{r}(t)$ and virtual error $\bar{e}(t)$.

The validity of minimizing the VRFT cost function of Eq. (14) as a means to minimize the MRC cost function of Eq. (10) will be more deeply analyzed in sub-section 2.3.1. The intuitive reason behind it is the following: assuming that, via the minimization of J^V, the ideal controller is found, i.e., $\mathcal{C}[\boldsymbol{q}^0_{n_c-1}\bar{\boldsymbol{z}}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}_0] = \mathcal{C}_r$, where $J^V(\boldsymbol{\theta}_0) = 0$. If this controller, together with $\mathcal{Z}$, is put in closed loop with the system $\mathcal{S}$ and fed by $\bar{r}(t)$, the controller's output is $u(t)$ exactly, which would make the system output the desired $y(t)$.

The beauty of minimizing the Eq. (14) is twofold: first, it makes possible to design a nonlinear controller for a nonlinear system using only input-output data collected from a single experiment; second, if the controller is linearly parametrized, $J^V(\boldsymbol{\theta})$ is convex, whereas $J(\boldsymbol{\theta})$, for the same setup, it is not.

2.3.1 $J^V(\boldsymbol{\theta})$ as a means to $J(\boldsymbol{\theta})$

Although minimizing $J^V(\boldsymbol{\theta})$ as a means to minimize $J(\boldsymbol{\theta})$ seems intuitively sound, in this sub-section the assumptions and theorem that make this approach valid will be given

in order to better understand the method's limitations and foundation.

Let's first begin with the following assumptions:

Assumption 2.3.1 (Controller in class)**.** *The ideal controller is in the class of controllers given by the parametrization* $\mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \bar{\boldsymbol{z}}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}]$*, i.e.* $\exists \boldsymbol{\theta}_0 \mid \mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \bar{\boldsymbol{z}}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}_0] = \mathcal{C}_r$.

Assumption 2.3.2. *For any finite control signal* $u(t)$*, the output of* $\mathcal{S}$ *exists and is unique.*

Assumption 2.3.3 (Noiseless System)**.** *The system is noise free, i.e.,* $\nu(t) = 0$.

Assumption 2.3.4. $J^V(\boldsymbol{\theta})$ *has a unique minimizer.*

Using assumptions 2.3.1 to 2.3.4, the following theorem, similar to the one given in (CAMPI; SAVARESI, 2006), states

Theorem 2.3.1. *If* $\boldsymbol{\theta}_0$ *leads to perfect tracking, i.e.,* $J(\boldsymbol{\theta}_0) = 0$ *in* (10)*, then* $\boldsymbol{\theta}_0$ *is also a minimizer of* $J^V(\boldsymbol{\theta})$ *in* (14)*. Then, in view of assumption 2.3.4, arg min* $J^V(\boldsymbol{\theta}) =$ *arg min* $J(\boldsymbol{\theta}) = \boldsymbol{\theta}_0$.

Proof. Since $\boldsymbol{\theta}_0$ makes $J(\boldsymbol{\theta}_0) = 0$, it follows that

$$y_r(t) = y(t, \boldsymbol{\theta}_0) \ \ \forall t, \tag{15}$$

where the parametrized output is the system output when fed by the virtual reference, i.e.

$$y(t, \boldsymbol{\theta}_0) = \mathcal{S}[y(t, \boldsymbol{\theta}_0)\boldsymbol{q}_{n_y}, \mathcal{C}_{\mathcal{Z}}[\bar{r}(t)\boldsymbol{q}_{n_z-1}^0, \bar{e}(t)\boldsymbol{q}_{n_z-1}^0, y(t, \boldsymbol{\theta}_0)\boldsymbol{q}_{n_z-1}^0; \boldsymbol{\theta}_0]\boldsymbol{q}_{n_y}], \tag{16}$$

and since the reference output is, by construction, the collected output $y(t)$ generated by the collected input $u(t)$, is possible to write

$$y_r(t) = \mathcal{S}[y_r(t)\boldsymbol{q}_{n_s}, u(t)\boldsymbol{q}_{n_s}], \tag{17}$$

with this, it follows that the control generated with the virtual reference is the same as the collected control, i.e.

$$\mathcal{C}_{\mathcal{Z}}[\bar{r}(t)\boldsymbol{q}_{n_z-1}^0, \bar{e}(t)\boldsymbol{q}_{n_z-1}^0, y(t, \boldsymbol{\theta}_0)\boldsymbol{q}_{n_z-1}^0; \boldsymbol{\theta}_0] = u(t) \tag{18}$$

which, from (15), gives

$$\mathcal{C}_{\mathcal{Z}}[\bar{r}(t)\boldsymbol{q}_{n_z-1}^0, \bar{e}(t)\boldsymbol{q}_{n_z-1}^0, y(t)\boldsymbol{q}_{n_z-1}^0; \boldsymbol{\theta}_0] = u(t), \tag{19}$$

seeing $\mathcal{C}_{\mathcal{Z}}$ as $\mathcal{C}$ in (14), and since $\boldsymbol{\theta}_0$ is the unique minimizer of $J^V(\boldsymbol{\theta})$, (19) is the minimum of (14) with $\boldsymbol{\theta}_0$ being the arg min of both (14) and (10). □

Theorem 2.3.1 gives the main result of the of VRFT approach, in the next sections some of the assumptions made in its development will be dropped in order to show the consequences and to make the problem closer to what will be latter used.

2.4 Mismatched Controller

When Assumption 2.3.1 is not satisfied, the argument used in 2.3.1 is not valid anymore since no parameter vector $\boldsymbol{\theta}$ would make $J(\boldsymbol{\theta}) = 0$. This case is known as the mismatched case.

To deal with this problem the idea is to make, in a way, $J(\boldsymbol{\theta}) \approx J^V(\boldsymbol{\theta})$ by means of a filter applied on the data. For the linear case the filter is formulated in a way that the approximation is made globally whereas in the non-linear case, the approximation is made in a way that only the second derivative of the cost functions is approximated.

The filter itself is a SISO linear system defined as

$$s(t) = \mathcal{L}\left[s(t)\boldsymbol{q}_{n_l}, v(t)\boldsymbol{q}_{n_l}\right] \tag{20}$$

where $s(t)$ is the output and $v(t)$ is the input. Since the filter $\mathcal{L}$ is linear, it can be written as a transfer function, i.e. as

$$s(t) = L(q)v(t), \tag{21}$$

where $L(q)$ is the transfer function of the system $\mathcal{L}$.

Whether in the linear or nonlinear case, a new cost function for the filtered VRFT is defined as

$$J^{VF} = \bar{E}\left[L\left(u(t) - \mathcal{C}[\boldsymbol{q}^0_{n_c-1}\bar{\boldsymbol{z}}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}]\right)\right]^2. \tag{22}$$

In the next sub-sections, the linear and nonlinear filter versions will be developed to show the approach used in the VRFT method.

2.4.1 Linear VRFT

To better understand the role of the matching filter in the nonlinear VRFT, it is interesting and productive to, first, understand its role in the linear case, as the results can be shown in a cleaner and compact way.

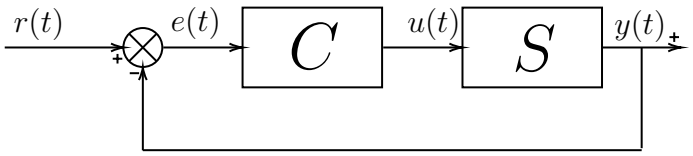
To do so, Assumptions 2.3.2 to 2.3.4 and the following assumption are in order for this subsection

Assumption 2.4.1. *All systems are linear, i.e., the maps $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{Z}$ are linear.*

In the linear case the authors of (CAMPI; LECCHINI; SAVARESI, 2002) make the two cost functions $J(\boldsymbol{\theta})$ and $J^{VF}(\boldsymbol{\theta})$ equal by means of filtering both the virtual error and the measured control with L. To simplify the analysis, it is considered that the measurement system is such that $z(t) = e(t)$, and also, analogous to the filter transfer function definition of (21), the transfer functions of the system, controller, ideal controller and reference system are defined as C, S, C_d and S_r. The control architecture for this setup is shown in Figure 3.

In view of linearity, it is also possible to write the model reference cost function (10) as

Figure 3 – Control architecture for the simplified linear case.



Source: author

$$J(\boldsymbol{\theta}) = \bar{E}\left[\left(S(q,\boldsymbol{\theta}) - S_r(q)\right) r(t)\right]^2, \tag{23}$$

where the closed loop relations, $S(q,\boldsymbol{\theta})$ and $S_r(z)$ are, using the transfer functions, given by

$$S(q,\boldsymbol{\theta}) = \frac{C(q,\boldsymbol{\theta})S(q)}{1 + C(q,\boldsymbol{\theta})S(q)}; \qquad S_r(q) = \frac{C_r(q)S(q)}{1 + C_r(q)S(q)}. \tag{24}$$

The VRFT cost function can be written as

$$J^{VF}(\boldsymbol{\theta}) = \bar{E}\left[L(q)\left(u(t) - C(q,\boldsymbol{\theta})\bar{e}\right)\right]^2. \tag{25}$$

As in the analysis made in (BAZANELLA; CAMPESTRINI; ECKHARD, 2011), applying the Parseval's Theorem to (23) gives, after some manipulation,

$$J(\boldsymbol{\theta}) = \frac{1}{2\pi}\int_{-\pi}^{\pi} |S(e^{j\omega})|^2 |T(e^{j\omega},\boldsymbol{\theta})|^2 |T_r(e^{j\omega})|^2 |C(e^{j\omega},\boldsymbol{\theta}) - C_r(e^{j\omega})|^2 \Phi_r(e^{j\omega}) d\omega \tag{26}$$

where the sensitivities $T(e^{j\omega},\boldsymbol{\theta})$ and $T_r(e^{j\omega})$ are

$$T(e^{j\omega},\boldsymbol{\theta}) = \frac{1}{1 + S(q)C(q,\boldsymbol{\theta})}; \qquad T_r(e^{j\omega}) = \frac{1}{1 + S(q)C_r(q)}. \tag{27}$$

Similarly, applying Parseval's Theorem to (25) gives

$$J^{VF}(\boldsymbol{\theta}) = \frac{1}{2\pi}\int_{-\pi}^{\pi} |L(e^{j\omega})|^2 \frac{|S(e^{j\omega})|^2 |T_r(e^{j\omega})|^2}{|S_r(e^{j\omega})e^{j\omega}|^2} |C_r(e^{j\omega}) - C(e^{j\omega},\boldsymbol{\theta})|^2 \Phi_u(e^{j\omega}) d\omega. \tag{28}$$

In this case, it is also possible to prove Theorem 2.3.1 by noting that both (26) and (28) have its minimum at $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ since $C_r(q) = C(q,\boldsymbol{\theta}_0)$, which makes the term $|C_r(e^{j\omega}) - C(e^{j\omega},\boldsymbol{\theta})|^2$ be zeroed in both cost functions.

Here, however, there is no parameter vector $\boldsymbol{\theta}$ that makes $C_r(q) = C(q,\boldsymbol{\theta})$, and, since all the other multiplicative terms are different between (26) and (28), the argument that minimizes the two cost functions does not need to be the same.

Theorem 2.4.1 gives conditions that validate the VRFT approach in the mismatched controller case.

Theorem 2.4.1. *If the filter $L(q)$ is such that*

$$|L(e^{j\omega})|^2 = |S_r(e^{j\omega})|^2 |T(e^{j\omega},\boldsymbol{\theta})|^2 \frac{\Phi_r(e^{j\omega})}{\Phi_u(e^{j\omega})}, \tag{29}$$

the cost functions $J(\boldsymbol{\theta})$ and J^{VF} have their minimum at the same $\boldsymbol{\theta}$.

Proof. Using (29) into (28) reduces the latter to (26). □

Since $T(e^{j\omega}, \boldsymbol{\theta})$ depends on the system model, the filter itself does as well. There are many ways to deal with this problem. For instance, one could, with the IO data, identify the system model and use it in the filter. The explicit identification of the system is, however, part of what Data Based control methods try to avoid.

Another way to deal with the problem of the dependence of L onto the system model is to assume that:

Assumption 2.4.2. *The sensitivities $T(e^{j\omega}, \boldsymbol{\theta})$ and $T_r(e^{j\omega})$ are close. i.e., $T(e^{j\omega}, \boldsymbol{\theta}) \approx T_r(e^{j\omega})$.*

With this assumption, the filter can be written as

$$|L(e^{j\omega})|^2 = |S_r(e^{j\omega})|^2 |1 - S_r(e^{j\omega})|^2 \frac{\Phi_r(e^{j\omega})}{\Phi_u(e^{j\omega})}, \tag{30}$$

which is as valid as the Assumption 2.4.2 is.

2.4.2 Nonlinear VRFT

In the nonlinear case, the authors of (CAMPI; SAVARESI, 2006) make $J(\boldsymbol{\theta}) \approx J^{VF}(\boldsymbol{\theta})$ only in the second derivative. i.e., the filter $L(q)$ in (22) is such that

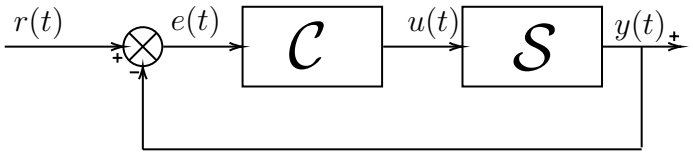
$$\left.\frac{\partial^2 J^{VF}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\right|_{\boldsymbol{\theta}_0} = \left.\frac{\partial^2 J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}\right|_{\boldsymbol{\theta}_0}. \tag{31}$$

To obtain the filter the following assumption is made in (CAMPI; SAVARESI, 2006):

Assumption 2.4.3. *The reference model map $\mathcal{S}_r$ is linear with transfer function $S_r(q)$.*

The control architecture is also simplified to consider that $z(t) = e(t)$, which is shown in Figure 4.

Figure 4 – Control Architecture



Source: author

Theorem 2.4.2 gives a filter that satisfies the condition on (31).

Theorem 2.4.2. *Let L be the cascade connection between L_1 and L_2 such that the filter input signal $u(t)$ is filtered to $u_L(t)$ as*

$$u_L(t) = \mathcal{L}[u_L(t)\boldsymbol{q}_{n_{s_r}}, u(t)\boldsymbol{q}_{n_s}] = \mathcal{L}_2[u_L(t)\boldsymbol{q}_{n_{s_r}}, \mathcal{L}_1[v(t)\boldsymbol{q}_{n_s}, u(t)\boldsymbol{q}_{n_s}]\boldsymbol{q}_{n_{s_r}}], \tag{32}$$

where $v(t)$ is the output of $\mathcal{L}_1$, n_s is the number of delays used in the map $\mathcal{L}_1$ and n_{s_r} is the number of delays used in the map $\mathcal{L}_2$.

With $\mathcal{L}_1$ as

$$\mathcal{L}_1[v(t)\boldsymbol{q}_{n_s}, u(t)\boldsymbol{q}_{n_s}] = v(t)\boldsymbol{q}_{n_s}^T \frac{\partial \mathcal{S}}{\partial y(t)\boldsymbol{q}_{n_s}} + u(t)\boldsymbol{q}_{n_s}^T \frac{\partial \mathcal{S}}{\partial u(t)\boldsymbol{q}_{n_s}}, \tag{33}$$

where the terms of $\mathcal{S}$ were omitted, i.e. $\mathcal{S} = \mathcal{S}[y(t)\boldsymbol{q}_{n_s}, u(t)\boldsymbol{q}_{n_s}]$ and, e.g., the last partial derivative is the partial derivative of $\mathcal{S}$ with respect to the vector of past controls $u(t)\boldsymbol{q}_{n_s}$.

And $\mathcal{L}_2$ have its transfer function $L_2(q)$ given by

$$L_2(q) = 1 - S_r(q). \tag{34}$$

The condition in (31) *is met with the filter* (32)*.*

See the appendix of (CAMPI; SAVARESI, 2006) for the proof of the Theorem 2.4.2.

Approximating the second derivative of the two cost functions is not the only sensible choice, in fact the authors of (ESPARZA; SALA; ALBERTOS, 2011) use the filter to accomplish a different approximation. There the objective of the filter is to approximate only the fist derivative of the two cost functions, i.e.

$$\left.\frac{\partial J^{VF}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}_0} = \left.\frac{J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}_0} \tag{35}$$

The approach to satisfy this requirement is to filter, with a filter $\mathcal{L}$, the derivative of the controller with respect to its parameters. Theorem 2.4.3 states how to obtain a filter that satisfies the condition on (35). Where it is also considered that the measurement is such that $z(t) = e(t)$.

Theorem 2.4.3. *Let $\mathcal{L}$ be the cascade connection between $\mathcal{L}_1$ and $\mathcal{L}_2$ such that the input signal $\partial\mathcal{C}/\partial\boldsymbol{\theta}$ is filtered to $dy(t;\boldsymbol{\theta})/d\boldsymbol{\theta}$ as*

$$\begin{aligned}\frac{dy(t;\boldsymbol{\theta})}{d\boldsymbol{\theta}} =& \mathcal{L}\left[\frac{dy(t;\boldsymbol{\theta})}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_s}, \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}\right] \\ =& \mathcal{L}_2\left[\frac{dy(t;\boldsymbol{\theta})}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_s}, \mathcal{L}_1\left[\frac{du(t)}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_c}, \frac{dy(t;\boldsymbol{\theta})}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_c}, \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}\right]\boldsymbol{q}_{n_s}\right]\end{aligned} \tag{36}$$

where $\mathcal{L}_1$ is

$$\begin{aligned}\frac{du(t)}{d\boldsymbol{\theta}} = \mathcal{L}_1\left[\frac{du(t)}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_c}, \frac{dy(t;\boldsymbol{\theta})}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_c}, \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}\right] = \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}} - \frac{\partial \mathcal{C}}{\partial e(t)\boldsymbol{q}_{n_c}}^T \frac{dy(t;\boldsymbol{\theta})\boldsymbol{q}_{n_c}}{d\boldsymbol{\theta}} + \\ \frac{\partial \mathcal{C}}{\partial u(t)\boldsymbol{q}_{n_c}}^T \frac{du(t)\boldsymbol{q}_{n_c}}{d\boldsymbol{\theta}},\end{aligned} \tag{37}$$

and

$$\begin{aligned}\frac{dy(t;\boldsymbol{\theta})}{d\boldsymbol{\theta}} = \mathcal{L}_2\left[\frac{dy(t;\boldsymbol{\theta})}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_s}, \frac{du(t)}{d\boldsymbol{\theta}}\boldsymbol{q}_{n_s}\right] = \frac{\partial \mathcal{S}}{\partial u(t)\boldsymbol{q}_{n_s}}^T \frac{du(t)\boldsymbol{q}_{n_s}}{d\boldsymbol{\theta}} + \\ \frac{\partial \mathcal{S}}{\partial y(t)\boldsymbol{q}_{n_s}}^T \frac{dy(t)\boldsymbol{q}_{n_s}}{d\boldsymbol{\theta}}.\end{aligned} \tag{38}$$

The condition on (35) *is met with the filter* (36)*.*

See (ESPARZA; SALA; ALBERTOS, 2011) for a proof of Theorem 2.4.3.

2.4.3 Filter application

As shown, in the linear case the filter is applied on the virtual error, obtained using the collected output data, and on the input data, also obtained in the experiment. Although the ideal filter does depend on $\mathcal{S}$, a filter approximation is constructed using only the reference model $\mathcal{S}_r$ and the spectrum of the input and output data.

For the nonlinear case, however, the filter is dependent on the system model $\mathcal{S}$ and no approximate filter is proposed in the presented literature. In fact, it is recommended to, from the IO data or basic physical principles, formulate the system model to use in the filter. Using the method in such a fashion does make it an indirect method, as discussed in (CAMPI; SAVARESI, 2006), which is, perhaps, an undesired result.

Another thing to mind is the complexity of implementing a filter as proposed in (36). With this filter, the application of the optimization algorithm on $\mathcal{C}$ would have to rely on two connected difference equations with input and output of the size of the parameter vector $\boldsymbol{\theta}$, which, for parametrizations that rely on many parameters, as is usually the case with Deep Neural Networks, could have a prohibitive computational cost. This might be the reason to why no DNN with more than 21 parameters was used in (ESPARZA; SALA; ALBERTOS, 2011).

It is interesting to note the role of the filter presented in Theorem 2.4.2. As pointed out in (CAMPI; SAVARESI, 2006), the first part of the filter, i.e., $\mathcal{L}_1$, is responsible to account for the effect of the input, used in $J^V(\boldsymbol{\theta})$ cost, on the output, used in the $J(\boldsymbol{\theta})$ cost. The second part of the filter, i.e., $\mathcal{L}_2$, has its transfer function L_2 with a small magnitude frequency response in frequencies where $S_r(q) \approx 1$. This effect incorporates on $J^{VF}(\boldsymbol{\theta})$ the low sensitivity of $J(\boldsymbol{\theta})$ to errors on those frequencies where $S_r(q) \approx 1$. Since the magnitude of $S_r(q)$ also usually drops over high frequencies, the main function of $\mathcal{L}_2$ is to penalize less the errors on the frequency range where $S_r(q) \approx 1$.

As shown in (CAMPI; SAVARESI, 2006), the filter in (32) is a generalization of the linear filter version of (29), which, in turn, can be approximated by (30), which does not require the system model $\mathcal{S}$. That is the reason why the filter $\mathcal{L}_D$, with transfer function L_D given by

$$L_D(q) = (1 - S_r)\frac{S_r a}{1 - q}, \tag{39}$$

is used in the rest of this work. In (39), the term $a/(1-q)$ is as such because it is assumed that this is the spectrum of the reference signal over the spectrum of the control signal. The therm a is calculated to make $L_D(q)$ have unitary steady state gain. This is important because the control signal will be filtered by $L_D(q)$ and if it does not have a unitary gain, the filtered control might have a range far off the unfiltered control.

2.5 Non Unique Minimizer of $J^V(\boldsymbol{\theta})$

Assumption 2.3.4 is made only to simplify the proof of the Theorem 2.3.1 because it allows to think that each parameter vector forms a single map $\mathcal{C}$. But the actual requirement is that, whatever the parameter vector is, its parametrized control map, say $\mathcal{C}(\cdot;\boldsymbol{\theta}_n)$, generates the collected input when feed by the virtual signals. This means that the same argument of the proof of Theorem 2.3.1 can be made even if this assumption is not met, as states the Theorem 2.5.1.

Theorem 2.5.1. *Given the assumptions 2.3.1 to 2.3.3, let both $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_1$, with $\boldsymbol{\theta}_0 \neq \boldsymbol{\theta}_1$, be the global minimum of $J(\boldsymbol{\theta})$, i.e., they're such that $J(\boldsymbol{\theta}_0) = J(\boldsymbol{\theta}_1) = 0$. Then it follows that arg min $J^V(\boldsymbol{\theta})$ = arg min $J(\boldsymbol{\theta})$ = $\boldsymbol{\theta}_0$ and arg min $J^V(\boldsymbol{\theta})$ = arg min $J(\boldsymbol{\theta})$ = $\boldsymbol{\theta}_1$. Moreover, it follows that*

$$\begin{aligned} u(t) &= \mathcal{C}_{\mathcal{Z}}[\bar{r}(t)\boldsymbol{q}^0_{n_z-1}, \bar{e}(t)\boldsymbol{q}^0_{n_z-1}, y(t)\boldsymbol{q}^0_{n_z-1}; \boldsymbol{\theta}_0] \\ &= \mathcal{C}_{\mathcal{Z}}[\bar{r}(t)\boldsymbol{q}^0_{n_z-1}, \bar{e}(t)\boldsymbol{q}^0_{n_z-1}, y(t)\boldsymbol{q}^0_{n_z-1}; \boldsymbol{\theta}_1], \end{aligned} \tag{40}$$

which means that, although $\boldsymbol{\theta}_0 \neq \boldsymbol{\theta}_1$, the maps generated by $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_1$ are the same.

Proof. Since $J(\boldsymbol{\theta}_0) = 0$ and $J(\boldsymbol{\theta}_1) = 0$ it follows that

$$y_r(t) = y(t, \boldsymbol{\theta}_0) \tag{41}$$

and

$$y_r(t) = y(t, \boldsymbol{\theta}_1) \tag{42}$$

which makes

$$y_r(t) = y(t, \boldsymbol{\theta}_0) = y(t, \boldsymbol{\theta}_1) \tag{43}$$

with $y(t, \boldsymbol{\theta}_0)$ given by (16) and $y(t, \boldsymbol{\theta}_1)$ is given by replacing $\boldsymbol{\theta}_0$ with $\boldsymbol{\theta}_1$ in (16). Using (17) it follows (18) and that

$$\mathcal{C}_{\mathcal{Z}}[\bar{r}(t)\boldsymbol{q}^0_{n_z-1}, \bar{e}(t)\boldsymbol{q}^0_{n_z-1}, y(t, \boldsymbol{\theta}_1)\boldsymbol{q}^0_{n_z-1}; \boldsymbol{\theta}_1] = u(t) \tag{44}$$

from which, using (43), (40) follows. □

Thus, as shown, the fact that the parametrization of $\mathcal{C}$ can make $J^V(\boldsymbol{\theta})$ have multiple global minima does not pose a conceptual problem, since all sets of parameters that minimize $J^V(\boldsymbol{\theta})$ also minimize $J(\boldsymbol{\theta})$ and generate the same control map $\mathcal{C}$.

2.6 Effects of Measurement Noise

When the noiseless system assumption, i.e., Assumption 2.3.3 is not satisfied, the parameter vector that globally minimizes the cost function $\hat{\boldsymbol{\theta}}_0$ is no longer deterministic, but instead it is a stochastic estimate of the real parameter vector $\boldsymbol{\theta}_0$.

The VRFT method transforms the control design problem of an unknown system into an identification problem. Dealing with measurement noise in a system identification problem is the norm and many approaches for those problems are presented in standard textbooks such as (LJUNG, 1999; SÖDERSTRÖM; STOICA, 1989). However, although the system $\mathcal{S}$ has measurement noise, this is not the case with the identification done by minimizing the $J^V(\boldsymbol{\theta})$ cost function, since the identification is done on the virtual measurement $\bar{\boldsymbol{z}}(t)$ and the collected control $u(t)$.

The difference arises from the fact that the virtual error $\bar{e}(t)$, a component of $\bar{\boldsymbol{z}}(t)$, is constructed using the system output and, this way, the control system $\mathcal{C}$ has its input realization contaminated by noise. This problem is known as Errors-in-Variables which is recognized as a more difficult problem (SÖDERSTRÖM, 2007, 2018).

The virtual error noise contamination is better described and more easily understood in the linear case. Because of that, in the following development, Assumptions 2.3.2 to 2.3.4 and 2.4.1 are considered true.

The virtual error is, using the transfer functions and the definition of (13),

$$\bar{e}(t) = (S_r^{-1}(q) - 1)(S(q)u(t) + \nu(t)); \tag{45}$$

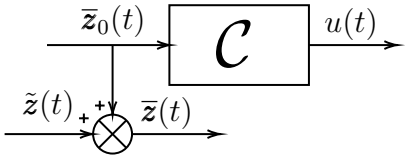
$$\bar{e}(t) = (S_r^{-1}(q) - 1)S(q)u(t) + (S_r^{-1}(q) - 1)\nu(t); \tag{46}$$

where its noted that the virtual error is formed by the noiseless virtual error $\bar{e}_0(t)(S_r^{-1}(q) - 1)S(q)u(t)$ and a noise term which contains all the virtual error's noise $\tilde{e}(t) = (S_r^{-1}(q) - 1)\nu(t)$. With this, the virtual error is

$$\bar{e}(t) = \bar{e}_0(t) + \tilde{e}(t). \tag{47}$$

Since the cost function $J^V(\boldsymbol{\theta})$ uses a controller parametrization that is fed with only noise free signals, the ideal controller itself is only fed with such signals. This means that the true ideal control is propagated without noise while what is actually available for identification is a corrupted measure of $\boldsymbol{z}(t)$, i.e. $\bar{\boldsymbol{z}}(t) = \bar{\boldsymbol{z}}_0(t) + \tilde{\boldsymbol{z}}(t)$, where similarly, $\bar{\boldsymbol{z}}_0(t)$ is the noiseless virtual measure and $\tilde{\boldsymbol{z}}(t)$ is the virtual measure's noise. The situation is better depicted in the Figure 5.

Figure 5 – Errors in variables in the VRFT.



Source: author

As shown in (BAZANELLA; CAMPESTRINI; ECKHARD, 2011; CAMPI; LECCHINI; SAVARESI, 2002) even for the linear case, the parameter vector is shown to be biased in the standard VRFT approach, i.e., when $\boldsymbol{\theta}$ is obtained, for a linearly parametrized

controller, using the Least Squares formula. This is a serious drawback of the method since in this case, increasing the number of data samples N does not improve the estimate $\hat{\boldsymbol{\theta}}_0$.

The usual approach to deal with this is to use Instrumental Variables, which is used in (BAZANELLA; CAMPESTRINI; ECKHARD, 2011; CAMPI; SAVARESI, 2006; CAMPI; LECCHINI; SAVARESI, 2002) when the controller is linearly parametrized by $\boldsymbol{\theta}$. However, this approach is not so easily expanded for the case of non-linearly parametrized controllers such as DNNs, and it is still an area of active research, see (LIU; SHANG; CHENG, 2020) and references therein.

The effect of having a biased $\hat{\boldsymbol{\theta}}$ is that the predicted control, generated by $\mathcal{C}_{\mathcal{Z}}[\cdot, \cdot; \hat{\boldsymbol{\theta}}]$, is itself biased. One way to measure the prediction bias is to divide the collected data into two sets, namely the training and test sets.

As the name suggests, the controller model is trained, i.e., has its parameter vector $\hat{\boldsymbol{\theta}}$ calculated via the minimization of $J^V(\boldsymbol{\theta})$, using only the data in the train set. The cost is then calculated for the train set, as $J^V_{train}(\boldsymbol{\theta})$, and the test set, as $J^V_{test}(\boldsymbol{\theta})$. An unbiased predictor[2] is then one that makes $J^V_{train}(\hat{\boldsymbol{\theta}}) \approx J^V_{test}(\hat{\boldsymbol{\theta}})$. This approach is common in machine learning where the bias problem is seen as the incapability of some model to generalize to the test data, or simply as over-fitting.

There are many approaches to deal with over-fitting of machine learning models. One of the most common approaches is the use of L1 and L2 regularization, which basically introduce a penalization on the model's weights. Another technique commonly used in iterative learning algorithms such as gradient descent, is known as early stopping. When the model is a DNN, approaches such as batch-normalization are also a viable choice to reduce over-fitting (GOODFELLOW; BENGIO; COURVILLE, 2016).

Since this is a central problem in this work, all those approaches will be better developed and explained in the chapter 3.

[2] In the prediction bias sense.

3 CONTROLLER PARAMETRIZATIONS

So far, the only assumption that was made about the controller $\mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \bar{\boldsymbol{z}}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}]$ is that it is parametrized by $\boldsymbol{\theta}$, no assumption was made on how this parametrization is and what class of functions they belong to. Some parametrizations offer nice properties, e.g., when the function $\mathcal{C}$ is linear in $\boldsymbol{\theta}$, the minimum of J^V has a closed form and can be readily calculated without any iterative process. But this is not the case of a more general $\mathcal{C}[\cdot; \boldsymbol{\theta}]$, where the parametrization is not linear in $\boldsymbol{\theta}$. Even for parametrizations that are non-linear in the parameters, some offer desirable properties when gradient based optimization is to be performed on it. that is the case of Deep Neural Networks (DNN) where, because of its compositional structure, the gradient with respect to the parameters can be calculated efficiently and at a relative low computational cost, even for a parametrization with thousands of parameters (GOODFELLOW; BENGIO; COURVILLE, 2016).

It is worth noting that a parametrization that is linear in the parameters is not necessarily linear in the outputs, thus, is possible to parametrize any C^∞ function with a polynomial basis using its Taylor series expansion. This, however, does not always translate well to practical applications, since the complete Taylor series expansion can require an infinitely long $\boldsymbol{\theta}$ to perfectly match most functions. And although it is possible to truncate the Taylor series to get an approximation of a function, the truncated version might not be sufficiently rich for some functions[1].

One could hope that the problem of defining a controller can be solved by simply choosing a parametrization with many parameters, since then $\mathcal{C}_r$ is more likely to be in the defined class of functions, but if this is done carelessly, the gradient calculation might not be computationally viable, the optimization procedure might be inefficient or even unstable. Moreover, a richer parametrization is not only more prone to over-fitting, i.e., identifying spurious patterns on the data as part of the true generating statistics, but it also requires more data to find the set of parameters that minimizes the cost function.

The problem of selecting a good controller parametrization is thus a complex one. To avoid over-fitting, it is desired to select the smallest possible basis[2] that contains $\mathcal{C}_r$. At the

[1] Or, alternatively, the truncated version might not be sufficiently small.

[2] In the sense of the number of parameters.

same time, since the system is unknown, so is the controller class, and defining a smaller parametrization that does not fully capture $\mathcal{C}_r$ might leave room for better performance or even result in a controller that is unstable.

Sometimes, usually from physical insight, it is known that $\mathcal{C}_r$ belongs to some compact parametrization of given structure, but that is not usually the case. Thus, often, the designer must choose a parametrization that is rich and contains many more parameters than what is needed to represent $\mathcal{C}_r$ and deal with the over-fitting in some other way.

To make matters worse, as n_θ grows, so does the computational complexity of the parametrization itself. Additionally, it is important to consider that in most real life problems one does not have a practical way of getting as much data as wanted or needed. Working with limited datasets also introduces errors in the parameter estimation and those usually grow with a bigger n_θ and smaller dataset.

However, the use of nonlinear parametrizations does not come for free. The trade-off of such approach is that there is no closed form to calculate the parameters and one has to rely on an iterative algorithm to minimize J^V, which for many machine learning algorithms and for most DNNs training approaches is based on gradient descent (CHARU, 2018; GOODFELLOW; BENGIO; COURVILLE, 2016; GÉRON, 2022). Performing minimization with gradient descent is much more computationally complex than calculating optimal parameters directly via a closed form solution consisting of some matrix multiplication and inversion, as is the case with linearly parametrized basis. But, once in possession of an adequate $\boldsymbol{\theta}$, performing inferences, i.e., calculating the output of $\mathcal{C}$ from it is inputs, can be made faster than what would be needed using a polynomial basis to represent some complex $\mathcal{C}_r$. And for parametrizations with the same number of parameters, one can expect similar inference complexity regardless of being linear in the parameters or not.

The chapter begins with polynomial basis, its definition, discussion, motivations and limitations are presented. Since polynomial basis are linear in the parameters, they're simple to define and deal with, offering a good starting point to discuss some properties of the nonlinear VRFT and a good setup to understand the effect of regularization and the effects of over-fitting.

After the definition of polynomial basis, DNNs are studied and analyzed, its general idea is presented in the context of deep feed forward networks, where the motivation and properties are apparent and easily understood. Then, a review of recurrent neural networks is presented along with two famous cells, namely, long short term memory (LSTM) and gated recurrent unit (GRU).

Then, a review of regularization in the machine learning context is presented. Regularization is important because it is possible to mitigate the effects of big n_θ with its use, as will be shown in section 3.3.3.

3.1 Polynomial

Polynomial basis can be considered the simplest nonlinear extension of linear basis. This is because the polynomial basis are a super-set[3] of linear basis where the extra elements are given by raising powers and multiplying elements of the linear basis.

The total degree of a multivariate polynomial is the highest total degree of its monomials with non-zero coefficients. The total degree of a monomial, in turn, is simply the sum of each variable exponent.

Let the variable vector $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T$ with $\boldsymbol{x} \in \mathbb{R}^n$, the index vector $\boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_n \end{bmatrix}^T$ with $\boldsymbol{\alpha} \in \mathbb{N}^n$ and the parameter vector $\boldsymbol{\theta} = \begin{bmatrix} \theta_1 & \theta_2 & \cdots & \theta_{n_\theta} \end{bmatrix}^T$ with $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$. Then the n variable polynomial $P(\boldsymbol{x}; \boldsymbol{\theta})$ of total degree m has the index set

$$I = \{\boldsymbol{\alpha} \mid \|\boldsymbol{\alpha}\|_1 \leq m\}, \tag{48}$$

which is the set of exponents that will be used to construct the polynomials and that has cardinality $|I| = n_\theta$ and elements $I = \{\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, , \cdots, \boldsymbol{\alpha}_{n_\theta}\}$. Where the j-th element of the i-th index vector of the set is represented as $\alpha_{i,j}$. With this, using the index set, $P(\boldsymbol{x}; \boldsymbol{\theta})$ is defined as

$$P(\boldsymbol{x}; \boldsymbol{\theta}) \coloneqq \sum_{i=1}^{n_\theta} \theta_i \prod_{j=1}^{n} x_j^{\alpha_{i,j}}. \tag{49}$$

It should be noted that, in (49), the parameters θ_i appear inside the summation, and one could express $P(\boldsymbol{x}; \boldsymbol{\theta})$ by

$$P(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{X}, \tag{50}$$

with

$$\boldsymbol{X} = \begin{bmatrix} \prod_{j=1}^{n} x_j^{\alpha_{1,j}} \\ \prod_{j=1}^{n} x_j^{\alpha_{2,j}} \\ \vdots \\ \prod_{j=1}^{n} x_j^{\alpha_{i,j}} \\ \vdots \\ \prod_{j=1}^{n} x_j^{\alpha_{n_\theta,j}} \end{bmatrix}. \tag{51}$$

Which makes explicit the fact that although $P(\boldsymbol{x}; \boldsymbol{\theta})$ is not linear in x_i, it is linear in the parameter vector $\boldsymbol{\theta}$. This makes its use as the controller parametrization specially appealing since then $J^V(\boldsymbol{\theta})$ may have a single minimum with a closed expression for its value.

In this fashion, to define the controller $\mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \boldsymbol{z}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}]$, is useful to define the matrix $Z = \boldsymbol{q}_{n_c-1}^0 \boldsymbol{z}^T(t)$ and to note that the i-th column $\boldsymbol{z}_i$ correspond to the i-th measurement signal of the system with n_{c-1} past samples.

[3] A set that contains at least this set.

With this, one can define a controller basis that is a separate polynomial in each measure and output as

$$\mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \boldsymbol{z}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}] = \sum_{i=1}^{n_z} P(\boldsymbol{z}_i; \boldsymbol{\theta}_{z_i}) + P(u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}_u), \tag{52}$$

where $\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_{z_1}^T & \boldsymbol{\theta}_{z_2}^T & \cdots & \boldsymbol{\theta}_{z_{n_z}}^T & \boldsymbol{\theta}_u^T \end{bmatrix}^T$.

This parametrization would imply that the system has a form of separation between its variables since no cross term would appear in the polynomial of (52). This can be a valid assumption for some systems but, more importantly, reduces the final n_θ, offering a more parsimonious controller.

Although valid for some systems, this simplifying assumption might be too strong, when the cross terms are required, one can define the variable vector as

$$\boldsymbol{x}_c = \begin{bmatrix} \boldsymbol{z}_1^T & \boldsymbol{z}_2^T & \cdots & \boldsymbol{z}_{n_z}^T & u(t)\boldsymbol{q}_{n_c}^T \end{bmatrix}^T, \tag{53}$$

in this case the controller parametrization is

$$\mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \boldsymbol{z}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}] = P(\boldsymbol{x}_c; \boldsymbol{\theta}). \tag{54}$$

It should be noted that, apart from the different indices for the parameters, the parametrization in (52) is a subset of (54).

On the other hand, both (52) and (54) can be made simpler by making $P(\boldsymbol{x}; \boldsymbol{\theta})$ itself simpler. In this notation, this can be done by simply changing the index set. One possible such change is to define a simpler index set I_s as

$$I_s = \{\boldsymbol{\alpha} \mid \|\boldsymbol{\alpha}\|_1 \leq m \text{ and } \alpha_i \leq 1\} \tag{55}$$

with its polynomial $P_s(\boldsymbol{x}; \boldsymbol{\theta})$

$$P_s(\boldsymbol{x}; \boldsymbol{\theta}) \coloneqq \sum_{\boldsymbol{\alpha} \in I_s} \theta_i \prod_{j=1}^{n} x_j^{\alpha_j}. \tag{56}$$

Using (56) in the controller parametrization (52) to generate the following parametrization

$$\mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \boldsymbol{z}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}] = \sum_{i=1}^{n_z} P_s(\boldsymbol{z}_i; \boldsymbol{\theta}_{z_i}) + P_s(u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}_u), \tag{57}$$

or in (54) with $\boldsymbol{x}_c$ from (53)

$$\mathcal{C}[\boldsymbol{q}_{n_c-1}^0 \boldsymbol{z}^T(t), u(t)\boldsymbol{q}_{n_c}; \boldsymbol{\theta}] = P_s(\boldsymbol{x}_c; \boldsymbol{\theta}). \tag{58}$$

On comparing the parametrization in (52) with its analogue (57), it is clear that the latter is a subset of the former, with usually many fewer parameters. The same is true when comparing (54) with (58).

The reasoning in the definition of $P_s(\boldsymbol{x};\boldsymbol{\theta})$ is that the likelihood of squared and higher order exponents terms rapidly increase as the total degree of the polynomial grows. Thus, unless the true generating statistics have higher order exponents, the parametrization will rapidly have many terms that do not belong to $\mathcal{C}_r$.

Those simpler parametrization, when used as $\mathcal{C}$, has the underlying simplifying assumption that the reference controller $\mathcal{C}_r$ is still sufficiently close to the defined class. Which, when true, allows a more parsimonious controller[4] to be found using the VRFT method.

Still, without knowing the underlying system dynamics and, by consequence, the reference controller basis, is hard to know which one of these basis is the best. Which means that, at least a priori, no particular basis presented in this section is inherently more efficient or better than the others.

3.2 Deep Neural Networks

Deep Neural Networks can be seen as a result of the study of Artificial Neural Networks (ANNs), a machine learning model inspired in networks of biological neurons in human brains. They were first introduced in 1943 in (MCCULLOCH; PITTS, 1943), where a simplified computational model of the brain was developed to perform complex computations. In this first model, the weights could be set by a human operator such that the output is as desired from the inputs.

The perceptron, proposed in (ROSENBLATT, 1958), was the first ANN model that had a way of automatically learning the correct weights. The algorithm used was a heuristic update in the weights and it was not posed in terms of the optimization of a loss function as is common today, although the objective was implicitly to minimize the prediction error.

The perceptron, however, has many limitations, most of them due to the fact that the perceptron has a single layer, which can be fixed by simply adding more layers to the perceptron, creating what is called the multi layer perceptron (MPL). The only problem is that then the heuristic to the weights update proposed for the single layer perceptron (SLP) no longer works.

Soon after that, the idea of posing the problem as the minimization of a cost index and using gradient descend to update the weights was already discussed (GÉRON, 2022). This requires the calculation of the gradient of the cost index with respect to the parameters, which is easily achieved using the chain rule of differentiation and allows training MLPs. The problem, however, was on how to perform such a calculation using the limited computing power available at the time for complex models with many parameters.

This problem was solved in (LINNAINMAA, 1976, 1970) where the reverse mode

[4] One with fewer parameters.

of automatic differentiation algorithm was presented. This algorithm was introduced in the context of calculating gradients of acyclic graphs, without any specific reference to ANNs. In fact, it took about four decades to this algorithm be used to efficiently calculate gradients in the ANN context (SCHMIDHUBER, 2015).

In this section, the SLP will be first presented to define the basic notation and to set the main intuition about DNNs. The DNN will then be introduced in the context of the simpler dense feed-forward neural network.

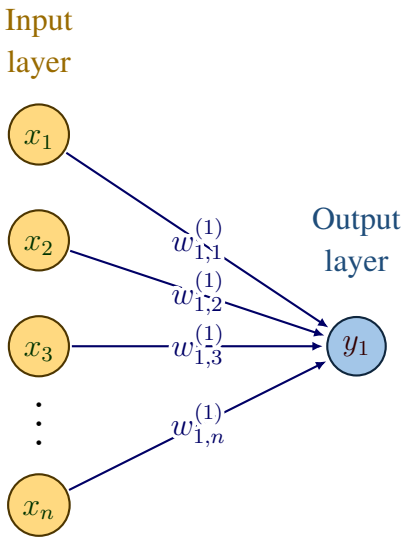
Following that, a discussion on how to train DNNs using stochastic gradient descent (SGD) will then be present. Following with a discussion on the use of backpropagation to calculate the gradients needed in the SGD.

In the sequence, Recurrent Neural Networks will be presented and discussed. Followed by their specific training algorithm and other particularities, as the different types of cells used for some problems such as Long Short Term Memory and Gated Recurrent Unit cells.

3.2.1 The Perceptron

The single layer percepetron is nothing more than a very simple neural network, it contains an input layer and an output layer. The simplest possible perceptron has a single output node. Its architecture is shown in Figure 6.

Figure 6 – The perceptron.



Source: author

In Figure 6, all inputs x_i are in the input layer, since this layer does not perform any calculation, its not counted in the total number of layers of the neural network, hence, the figure shows a single layer NN. The output y_1, in the perceptron, is the result of the activation function σ contained in its node, the input of this function is given by the linear combination of all inputs x_i, where the weights or coefficients of this linear combination

are $w_{1,i}^{(1)}$ and the bias[5] is $b^{(1)}$, where the upper 1 means first layer, a notation that will become useful later on. More precisely, the output is given by

$$y_1 = \sigma\left(b^{(1)} + \sum_{i=1}^{n} w_{1,i}^{(1)} x_i\right), \tag{59}$$

where the activation function σ is any nonlinear function and it is problem dependent.

Equation (59) can also be written in matrix form, which simplifies the notation. Let $W^{(i)} \in \mathbb{R}^{1\times p_i}$, where p_i is the number of parameters of the i-th layer, be the weight matrix of layer i with entry $w_{j,k}^{i}$, $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T$. The output of the perceptron is then given by

$$y = \sigma\left(W^{(1)}\boldsymbol{x} + b^{(1)}\right). \tag{60}$$

The original perceptron, as defined in (ROSENBLATT, 1958), used a step for activation function. This was used an approximation of what biological neurons were thought do or compute at the time, this is why in some contexts the computational nodes in a general neural network are named neurons.

The perceptron was generally used in the context of classification (GÉRON, 2022), where it was a somewhat capable classifier, although it was famously known to not be able to approximate the exclusive or (XOR) function, a problem that is solved in the Multilayer Perceptron.

By organizing the weights and biases in a single parameter vector $\boldsymbol{\theta}$, one can see the perceptron as a simple parametrized function $h(\boldsymbol{x};\boldsymbol{\theta})$, which, for a given $\boldsymbol{\theta}$, $h : \mathbb{R}^n \to \mathbb{R}$. The optimal or desired $\boldsymbol{\theta}$, in this context, is the one that makes h close to the function that generated the data, and the process of learning is nothing more than finding this set of parameters.

It is also possible to generalize the perceptron from a multi-input, single-output to a multi-input, multi-output, i.e., to make h such that $h : \mathbb{R}^n \to \mathbb{R}^m$. The multi-output perceptron is shown in Figure 7.

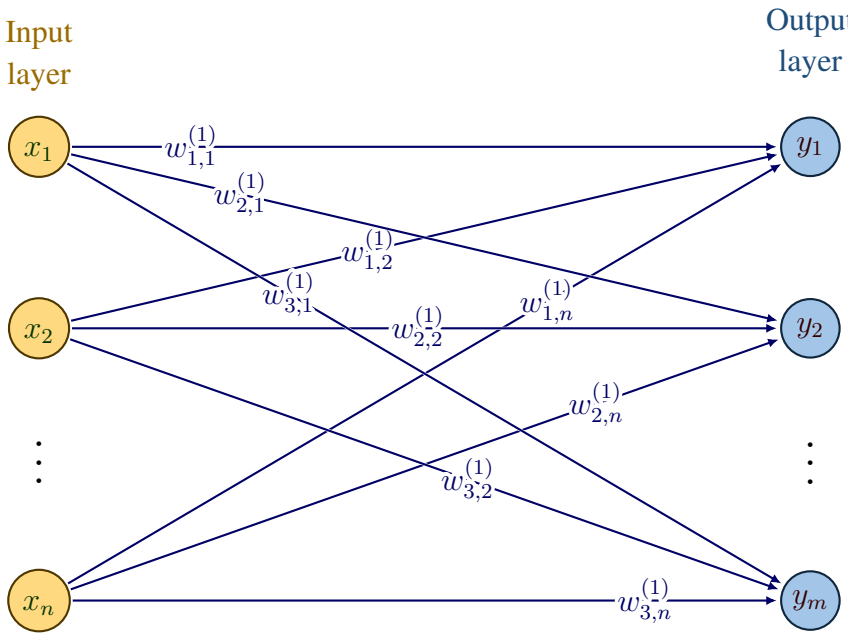
Making the output as $\boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & \cdots & y_m \end{bmatrix}^T$, the perceptron in Figure 7 calculates its output vector from the inputs as

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} \sigma_1\left(b^{(1)} + \sum_{i=1}^{n} w_{1,i}^{(1)} x_i\right) \\ \sigma_2\left(b^{(1)} + \sum_{i=1}^{n} w_{2,i}^{(1)} x_i\right) \\ \vdots \\ \sigma_m\left(b^{(1)} + \sum_{i=1}^{n} w_{m,i}^{(1)} x_i\right) \end{bmatrix}. \tag{61}$$

The representation in (61) is general in the sense that it assumes that each output has its own activation function. However, the usual approach is to consider that each layer has a single activation function that operates in an element-wise fashion, i.e., in each element of the vector. With this, (61) is simplified to

[5] Not represented in the Figure.

Figure 7 – The multi-output perceptron.



Source: author

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \sigma \left(\begin{bmatrix} b^{(1)} + \sum_{i=1}^{n} w^{(1)}_{1,i} x_i \\ b^{(1)} + \sum_{i=1}^{n} w^{(1)}_{2,i} x_i \\ \vdots \\ b^{(1)} + \sum_{i=1}^{n} w^{(1)}_{m,i} x_i \end{bmatrix} \right). \tag{62}$$

Making the bias vector as $\boldsymbol{b}^{(1)} = \begin{bmatrix} b^{(1)}_1 & b^{(1)}_2 & \cdots & b^{(1)}_m \end{bmatrix}^T$ and using the matrix notation, the multi-output perceptron calculation, represented in Figure 7, is given by

$$\boldsymbol{y} = \sigma^{(1)}(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}). \tag{63}$$

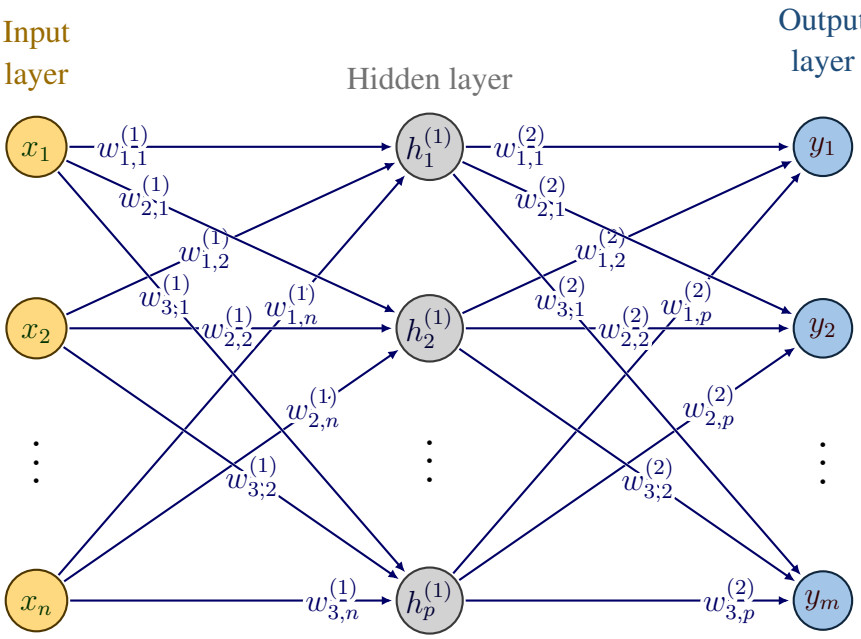
Since the perceptron was conceived with the intent of being used as a classifier, i.e., given some features $\boldsymbol{x}$ the objective is to correctly predict in which class y_i the features belong to, its training algorithm was created to increase the chance of the classification performed by the perceptron being correct (CHARU, 2018).

3.2.2 The Multilayer Perceptron

The perceptron is mainly effective to classify data that is linearly separable, i.e., if $\boldsymbol{x} \in \mathbb{R}^n$, each class y_i is correctly divided by a hyper-plane in $\mathbb{R}^n$. Real world data, however, does not always fall in this category and the perceptron struggles to perform well in those cases.

The solution to this problem is to increase the representative power of the perceptron, which is done by simply adding more layers. A multilayer perceptron with two layers is depicted in Figure 8.

Figure 8 – The two layer perceptron.



Source: author

The output $\boldsymbol{y}$ of the MLP in Figure 8 is, in vector notation, given by

$$\begin{aligned} \boldsymbol{y} &= \sigma^{(2)}\left(W^{(2)}\sigma^{(1)}\left(W^{(1)}\boldsymbol{x}+\boldsymbol{b}^{(1)}\right)+\boldsymbol{b}^{(2)}\right) = \boldsymbol{h}^{(2)}\left(\boldsymbol{h}^{(1)}\left(\boldsymbol{x}\right)\right) \\ \boldsymbol{y} &= \boldsymbol{h}^{(2)}\circ\boldsymbol{h}^{(1)}\circ\boldsymbol{x} \end{aligned} \tag{64}$$

where $\boldsymbol{h}^{(2)} = \boldsymbol{y}$ and the weight matrix of layer i, $W^{(i)} \in \mathbb{R}^{p_{i-1}\times p_i}$, where p_i is the number of parameters of the i-th layer, has entry $w_{j,k}^{i}$.

But a MLP can have any number of hidden layers. A general MLP with l layers, as shown in Figure 9, can be represented by

$$\begin{aligned} &\boldsymbol{y} = \sigma^{(l)}\left(W^{(l)}\sigma^{(l-1)}\left(\cdots\sigma^{(2)}\left(W^{(2)}\sigma^{(1)}\left(W^{(1)}\boldsymbol{x}+\boldsymbol{b}^{(1)}\right)+\boldsymbol{b}^{(2)}\right)\cdots+\boldsymbol{b}^{(l-1)}\right)+\boldsymbol{b}^{(l)}\right); \\ &\boldsymbol{y} = \boldsymbol{h}^{(l)}\circ\boldsymbol{h}^{(l-1)}\circ\cdots\circ\boldsymbol{h}^{(2)}\circ\boldsymbol{h}^{(1)}\circ\boldsymbol{x}; \end{aligned} \tag{65}$$

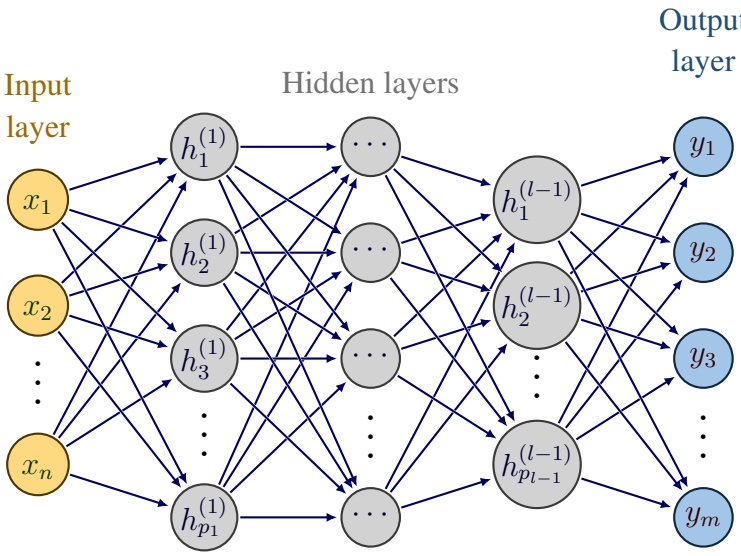
where $W^{(i)} \in \mathbb{R}^{p_{i-1}\times p_i}$, which, letting $\boldsymbol{h}^{(1)} = \sigma^{(1)}\left(W^{(1)}\boldsymbol{x}+\boldsymbol{b}^{(1)}\right)$, can be represented more clearly as a recurrence

$$\boldsymbol{h}^{(i)} = \sigma^{(i)}\left(W^{(i)}\boldsymbol{h}^{(i-1)}+\boldsymbol{b}^{(i)}\right)\ ,\ \ i\in\{2,\cdots,l\}, \tag{66}$$

where $\boldsymbol{y} = \boldsymbol{h}^{(l)} = \sigma^{(l)}\left(W^{(l)}\boldsymbol{h}^{(l-1)}+\boldsymbol{b}^{(l)}\right)$ and $\boldsymbol{h}^{(i)} \in \mathbb{R}^{p_i}$.

One characteristic of the weight update algorithm of the perceptron, i.e., the training algorithm, is that the weights are updated in a heuristic fashion that does not expand well to the case of the multilayer perceptron, the reason is that the algorithm updates the weights based on the error made by the output, but there is no clear way to calculate the

Figure 9 – The multilayer perceptron.



Source: author

error in the hidden layers since the collected data does not contain any information about what should be the values in those internals representations.

The remedy for this problem is to pose the training as an optimization and to use gradient descent to update the weights, this is the subject of subsection 3.2.5. The multilayer perceptron is also often referred to as the deep neural network (DNN) because the information from the input layer travels many hidden layers before getting into the output layer. There is not a consensus on how many layers are need to a model to be considered deep. Some authors, like (GÉRON, 2022), consider deep any MLP, this is the convention that will also be adopted here.

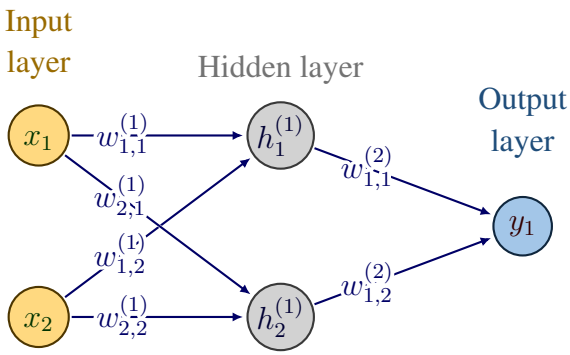
As discussed in subsection 3.2.1, the perceptron is not capable of learning the XOR function. The MPL, however, is capable of that using only a single hidden layer with two nodes, as shown in the next example, adapted from (GOODFELLOW; BENGIO; COURVILLE, 2016).

Example 3.2.1 (XOR function as an MLP)**.** *The exclusive or function has two inputs,* x_1 *and* x_2*, and one output* y_1*. It returns true only if one input is true and the other is false. All possible inputs and outputs of this function are captured in the input matrix* X *and output matrix* Y*, where each line is a data point, and has values*

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}; \quad Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{67}$$

The MLP capable of capturing this behavior is shown in Figure 10.

Figure 10 – The multilayer perceptron for the XOR problem.



Source: author

In Figure 10, the activation function of the hidden layer is a ReLU, the output layer's activation function is simply a linear function. The weights are organized in matrices $W^{(k)}$ *with elements* $w_{i,j}^{(k)}$ *shown in Figure 10 and with values*

$$W^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}; \quad W^{(2)} = \begin{bmatrix} 1 & -2 \end{bmatrix}; \tag{68}$$

and biases

$$b^{(1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}; \quad b^{(2)} = 0; \tag{69}$$

The input matrix is organized with one example per line, the same as the output vector (or matrix, in some other case where there is more than one output). The beauty of the matrix representation is of course more than just notational sugar, with it one can efficiently perform predictions on a whole batch in one single pass. Although this makes little difference in this case, where all the data fits in a batch which contains four examples, it makes a big difference for the usual use case of DNNs, where one usually have thousands of examples.

The first layer calculation of the neural network prediction is then

$$\begin{aligned} H^{(1)} &= \sigma^{(1)}\left(W^{(1)}X^T + B^{(1)}\right) = \sigma^{(1)}\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \end{bmatrix}\right); \\ &= \sigma^{(1)}\left(\begin{bmatrix} 0 & 1 & 1 & 2 \\ -1 & 0 & 0 & 1 \end{bmatrix}\right) = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \end{aligned} \tag{70}$$

where the bias matrix $B^{(1)}$ *is constructed with the repeated concatenation of the bias vector* $\boldsymbol{b}^{(1)}$ *in a way that matches the dimension of* X^T *and, similarly,* $H^{(1)}$ *is given by the concatenation of the hidden vectors* $\boldsymbol{h}^{(1)}$*, where each column is generated by a corresponding line in the data matrix.*

Notice that the last step is simply the aplication of the ReLU activation function in a element-wise fashion, i.e., applied in each element of the matrix, general activation functions will be shortly discussed in subsection 3.2.3, but the ReLU is given by $\sigma_{ReLU}(x) = max(0, x)$.

Since $\sigma^{(2)}$ *is a simple linear activation function and the bias vector is zero, the output layer calculation is given by a single matrix multiplication*

$$\hat{Y} = \sigma^{(2)}\left(W^{(2)}H^{(2)} + B^{(2)}\right) = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{71}$$

The predicted output, $\hat{Y}$ *is exactly equal the target output* Y*, and the mean squared prediction error, measured on* $\hat{Y}$ *of* (71)*, is zero.*

As shown, a DNN is far more powerful than a simple perceptron, this is due to composing nonlinear functions. It should be noted that a deep neural network composed only of linear activation functions cannot be more representative than a single layer perceptron. Because then, for any layer, its output are just the linear combination of the previous layer, which leads to the output of the DNN itself being a linear combination of the inputs.

The usefulness of depth in DNNs is that it allows a more efficient representation, as shown in (MONTÚFAR, 2014), a deep belief network (DBN), a kind of DNN, with finite deepness and width (the number of nodes in a layer), can approximate any probability distribution with arbitrarily small error.

3.2.3 Activation Functions

As shown, each node in a DNN can be decomposed into two major parts, first, the multiplication of the node's weights with the node's inputs, resulting in a scalar second, passing the result trough a activation function, which is often a non-linearity.

This non-linearity can, theoretically, be any function $\sigma : \mathbb{R} \to \mathbb{R}$. But, in most practical applications, only some activation functions that offer some benefit in training or are specially useful on giving the DNN a desired behavior are used. One such property is to have an activation function that produces non-saturating gradients, which helps with the problem of vanishing gradients.

In DNNs, a single activation function is typically defined for a whole layer, which helps to make the DNN's architecture more succinct. In those cases, the activation function is often represented as acting on the whole layer and it is implied that it acts separately in each element, is said to act element-wise.

Since many activation functions are squashing functions, i.e., a function that has a bounded image for an unbounded domain, the activation function of the output layer is

often an affine function, unless, of course, it is desired that the output is in fact bounded or possess some other useful property.

The Machine Learning community have developed many activation functions for many specific use cases, in this work, however, only the activation functions used will be presented.

3.2.3.1 Linear

The linear activation function is simply

$$\sigma_a(x) = x. \tag{72}$$

The σ_a has the property of not being bounded, and is, for this reason, used mainly in the output layer of a DNN.

3.2.3.2 Rectified Linear Unit (ReLU)

The ReLU activation function is given by

$$\sigma_{\text{ReLU}}(x) = \begin{cases} x, & \forall x \geq 0; \\ 0, & \forall\, x < 0; \end{cases} \tag{73}$$

This function can be seen as a negative saturated affine function, it is used in many famous architectures and shown to perform as well as many other activation functions. All while allowing faster calculations and easier code optimization than functions like sigmoid or hyperbolic tangent (GÉRON, 2022).

3.2.3.3 Signal

The signal function is given by

$$\sigma_{\text{sign}}(x) = \begin{cases} -1, & \forall x < 0; \\ 0, & \text{if } x = 0; \\ 1, & \forall x > 0. \end{cases} \tag{74}$$

The signal function is used in binary classifiers, i.e., when the model is to choose if a given input belongs to class 1 or -1. The problem with this activation function is that it generates a model that has a cost function that is not differentiable, which causes problem in training, for this reason the σ_{sign} activation function is rarely used in modern DNNs.

3.2.3.4 Sigmoid

The sigmoid function is given by

$$\sigma_{sgm}(x) = \frac{1}{1 - e^{-x}}. \tag{75}$$

The sigmoid function's image is in the range $[0, 1]$, it is a smooth squashing function that gives a given node (or layer) the property of boundedness, this can be beneficial in problems where the nodes would explode if used with a non-squashing function.

The sigmoid function is also a good choice for classifiers when the output can (or should) be interpreted as a probability. This function is very famous and many famously known architectures uses it, that is the case of, for instance, the LSTM cell used in recurrent neural networks (CHARU, 2018).

3.2.3.5 Hyperbolic tangent

The hyperbolic tangent function is give by

$$\sigma_{tanh} = \frac{e^{-2x} - 1}{e^{2x} + 1}. \tag{76}$$

The σ_{tanh} is a scaled and translated version of the sigmoid function, it has the same properties as the sigmoid except that it allows its output to have negative values, its image in the range $[-1, 1]$.

3.2.4 Cost Functions

Training a DNN is nothing more than minimizing a cost function, usually some form of measure of the error made by the prediction. This cost function can be any metric, but it is desirable to have a cost function that measures this error in a differentiable way, to allow the use of gradient descent optimization techniques.

The ideal cost function is problem dependent, classification problems often use a cross-entropy loss while regression problems often use the mean squared error. Some problems, however, need more specific losses to allow a good representation of what the model should do.

Whatever the loss is, all different losses are, for a predefined dataset, a function of the parameters of the chosen parametrization, but since the loss measures the distance between the true output matrix Y and the predicted output matrix $\hat{Y}$, where in a general setup, each output example $\boldsymbol{y}_i \in \mathbb{R}^{n_y}$ forms the dataset's output matrix as $Y = \begin{bmatrix} \boldsymbol{y}_1 & \cdots & \boldsymbol{y}_N \end{bmatrix}^T$, with elements $y_{i,j}$ and the prediction output matrix is constructed analogously, sometimes is preferable to represent the cost function as being dependent of the true and predicted output matrices, i.e., the cost function used in training $J(\boldsymbol{\theta})$, a function of the parameters $\boldsymbol{\theta}$, is represented as

$$J(\boldsymbol{\theta}) = J(Y, \hat{Y}(\boldsymbol{\theta})), \tag{77}$$

or $J(Y, \hat{Y})$ for short.

Both representations in (77) are equivalent since, for a given dataset, Y is fixed. Using $J(\boldsymbol{\theta})$ is more explicit when in the context of taking derivatives of the cost function with respect to the parameters, which is the case when in the context of training, while

$J(Y, \hat{Y})$ is more explicit on the prediction error. For this reason, in this subsection, the cost function will be referred to as $J(Y, \hat{Y})$.

In this work, only regression models will be trained and only with some specific losses, those will be presented in the next subsections.

3.2.4.1 Mean Squared Error

The mean squared error $J_{mse}(Y, \hat{Y})$ was already discussed in the context of the VRFT method, it is simply the averaged squared difference of the true and predicted values, i.e., the mean squared error. In this more general case, where the output can be a vector instead of just a scalar as in (10), the mean squared error is

$$J_{mse}(Y, \hat{Y}) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{n_y} \sum_{j=1}^{n_y} (y_{i,j} - \hat{y}_{i,j})^2. \tag{78}$$

3.2.4.2 Mean Squared Logarithmic Error

Some problems require a model that is precise for outputs that are big and small. In those cases the $J_{mse}(Y, \hat{Y})$ can be problematic, the reason is that it looses sensitivity around zero. A cost function that penalizes something closer to a percentage error is the mean square logarithmic error, $J_{msle}(Y, \hat{Y})$, given by

$$J_{msle}(Y, \hat{Y}) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{n_y} \sum_{j=1}^{n_y} (log_e(1 + y_{i,j}) - log_e(1 + \hat{y}_{i,j}))^2. \tag{79}$$

When $Y, \hat{Y} \approx 0$ the cost functions $J_{mse} \approx J_{msle}$, thus, the way that J_{msle} increases the sensitivity around zero, when compared to J_{mse}, is by decreasing the importance of errors when the true and predicted values are greater in magnitude (CHARU, 2018).

It should be noted that to use this cost function, the output dataset needs to be scaled to be in the range $(-1, \infty]$, in application the prediction of the DNN can be just re-scaled back to the original range using the inverse scaling transformation.

3.2.4.3 Mean Combined Squared Error

In some control problems, such as the VRFT method when on systems that require operation with very small control inputs, it is important to include the J_{mse}, since the J_{msle} is biased to underestimation[6] and, also, away from the zero error region, it is more interesting to minimize the J_{mse}, since it is closer to the formal definition in the VRFT. In those cases, a convex combination of the two cost functions, J_{mse} and J_{msle}, is defined as

$$J_{mcse}(Y, \hat{Y}) = \alpha_c J_{msle}(Y, \hat{Y}) + (1 - \alpha_c) J_{mse}(Y, \hat{Y}), \tag{80}$$

with the combination factor $\alpha_c \in [0, 1]$.

[6]The same difference between the real and predicted values produces a greater error if the predicted value is smaller than the true value than if the predicted value is greater than the true value.

The idea behind the J_{mcse} cost function is that J_{msle} dominates when the true and predicted values are small and J_{mse} looses sensibility. When the two values are big, the mean squared error dominates and $J_{mcse}(Y, \hat{Y}) \approx J_{mse}(Y, \hat{Y})$.

3.2.5 Stochastic Gradient Descent

The example of subsection 3.2.2 showed that a DNN can behave as a XOR function with a given set of weights and biases. Since the dataset in the example consists of only four samples and the function has a fairly simple behavior, one could select the weights by trial and error. However, in more realistic applications, one usually has thousands of samples in the dataset and the underlying generating function is much more complex.

The approach to select the DNN parameters , in those cases, is to define the problem as a minimization problem and to perform gradient descent on it. Gradient descent just updates the model's parameters in the direction that locally minimizes some cost function[7] $J(\boldsymbol{\theta})$, i.e.,

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha \frac{\partial J(\boldsymbol{\theta}_n)}{\partial \boldsymbol{\theta}_n}, \tag{81}$$

where $\alpha \in \mathbb{R}$ is known as the learning rate, a scalar that determines the size of the step in the direction that has, locally, the smallest $J(\boldsymbol{\theta})$.

A key point to consider is that the cost function is just the average of the per sample loss, i.e.,

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=0}^{N} J_i = E\left[J_i\right] \ \forall i \in \mathbb{N}^N. \tag{82}$$

Thus, to perform a weight update, one has to compute all the per sample loss and derivatives with respect to the DNN parameters, which is often in the range of thousands or millions. This can make the computational time of a single weight update prohibitively long.

An alternative to this approach reveals itself on remembering that the data is assumed to be collected from a quasi-stationary process. Hence, one can expect that, for a large enough batch size N_b, the following approximation is valid

$$J(\boldsymbol{\theta}) \approx \hat{J}(\boldsymbol{\theta}) = E\left[J_i\right] \ \forall i \in \mathbb{N}^{N_b}. \tag{83}$$

Performing the gradient descent with the approximation $\hat{J}(\boldsymbol{\theta})$ instead of with the full $J(\boldsymbol{\theta})$ is what is known as Stochastic Gradient Descent (SGD). For it to work properly, $\hat{J}(\boldsymbol{\theta})$ needs to be sufficiently accurate and unbiased, which translates to both selecting the batch samples uniformly and with a sufficiently large N_b.

SGD has many advantages over regular gradient descent, but the main one is that it is memory efficient and fast, this is due to the fact that one can now perform the weights

[7] In the VRFT problem, the function that will be minimized is actually $J^V(\boldsymbol{\theta})$, $J(\boldsymbol{\theta})$ is referred as the function minimized by the DNN for a cleaner presentation with a less cluttered notation.

update without having to compute all the per sample loss and derivatives of the whole dataset.

Selecting the correct batch size is somewhat problem dependent, generally one wants to select the largest possible batch size that does not overflow the machine's memory in which the model is being trained, since this will make the better use of parallel capable computing hardware[8] and deliver a better estimate of the true gradient. But selecting small batch sizes also has its advantages, as note by (WILSON; MARTINEZ, 2003), small batches offer a regularizing effect. Intuitively what happens is that, since the batch is chosen at random, small local minima that would appear using the whole dataset might disappear using a small part of it, which can promote the DNN to not get stuck in local minima.

Although selecting smaller batches can help with local minima, this alone is often not enough to deal with this problem. Many changes to the SGD method were proposed to deal with those problems and to accelerate training, in the following subsections some of them will be presented.

3.2.5.1 Momentum

An effective approach to deal with local minima and flat regions, proposed in (POLYAK, 1964), is to use a form of momentum in the parameter update, the intuition of momentum based optimization is clearer when one considers that the parameter update of (81) represents the dynamics of a moving physical object that lives in the $\mathbb{R}^{n_\theta}$ space.

In this case the objective is to make the object go to the lowest point in the $J(\boldsymbol{\theta})$ metric, which can be though as an altitude in that space. In the analogy, since the object dynamics is dependent of the local curvature, a mass-less object would be very prone to be stuck in local holes. If, however, one considers that the object has a mass, then a small hole in the object's path would not stop it from moving, which in turn would make it go out of the hole.

Defining the weight update change as a velocity, i.e.,

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \boldsymbol{v}_n \tag{84}$$

it is possible to see the effect of adding a mass to the object dynamics as making the velocity depend on its past values, i.e.,

$$\boldsymbol{v}_{n+1} = \beta \boldsymbol{v}_n - \alpha \frac{\partial J(\boldsymbol{\theta}_n)}{\partial \boldsymbol{\theta}_n}, \tag{85}$$

with $\beta \in \mathbb{R}$ known as the friction parameter and $\boldsymbol{v}_0 = 0$. Going back to the analogy with a moving object, when $\beta = 0$ the object has no mass. When $\beta > 0$ and $\alpha \neq 0$, it has mass and its velocity also depends on the past velocity values. This approach makes the

[8] Such as GPUs.

parameter update automatically speed up in flat regions with small gradients as well as make the parameter update less sensitive to local minima.

3.2.5.2 Nesterov Momentum

Nesterov momentum is named after its creator, which presented in (NESTEROV, 1983) a modification to the momentum algorithm. The Nesterov momentum algorithm consists in calculating the gradient not at the current position $\boldsymbol{\theta}_n$, but in what would be the future position if only the momentum part of (85) was considered.

Nesterov's algorithm uses the same weight update of (84), but the velocity update is modified to

$$\boldsymbol{v}_{n+1} = \beta \boldsymbol{v}_n - \alpha \frac{\partial J(\boldsymbol{\theta}_n + \beta \boldsymbol{v}_n)}{\partial \boldsymbol{\theta}_n}. \tag{86}$$

The rationale of why (86) works is that the next parameter $\boldsymbol{\theta}_n$ will be at least $\boldsymbol{\theta}_n + \beta \boldsymbol{v}_n$, thus, using this information to calculate the gradient gives some sort of correction to the gradient calculation of the standard momentum.

3.2.5.3 RMSProp

The traditional momentum based optimization techniques such as ones from subsections 3.2.5.1 and 3.2.5.2 uses the idea of increasing the consistency of the direction in which the gradient moves by adding a momentum term in its update rule. This can also be achieved by forcing some kind of consistency on each element of the weight update, which translates to having a per parameter learning rate.

The RMSProp algorithm is a slight modification to the adaptative gradient algorithm, also known as AdaGrad, presented in (DUCHI; HAZAN; SINGER, 2011). RMSProp uses the exponential averaging of the square magnitude of the partial derivative of the cost function with respect to the parameter while AdaGrad uses the aggregated square magnitude. This modification was first presented informally in an online class. The main effect is that the parameter learning rate does not grow unbounded for a constant local derivative.

The i-th parameter learning rate of the $n+1$ update of the RMSProp is, according to (CHARU, 2018), given by

$$A_{i,n+1} = \beta_2 A_{i,n} + (1 - \beta_2) \left(\frac{\partial J(\boldsymbol{\theta}_n)}{\partial \theta_{i,n}} \right)^2, \forall i \in \mathbb{N}^{n_\theta}, n \in \mathbb{N}. \tag{87}$$

with $A_{i,0} = 0$.

The parameter update is then calculated with the gradient normalized by the square root of the per parameter learning rate, i.e.,

$$\theta_{i,n+1} = \theta_{i,n} - \frac{\alpha}{\sqrt{A_{i,n}}} \frac{\partial J(\boldsymbol{\theta}_n)}{\partial \theta_{i,n}}, \forall i \in \mathbb{N}^{n_\theta}, n \in \mathbb{N}. \tag{88}$$

3.2.5.4 ADAM

The adaptative moments estimation (ADAM) algorithm, presented in (KINGMA; BA, 2014), can be seen as a combination of both RMSProp and Momentum, with the addition of solving the bias given by initializing the velocity and per parameter learning rate at zero.

The algorithm first calculates the velocity at iteration $n+1$ as

$$\boldsymbol{v}_{n+1} = \beta_1 \boldsymbol{v}_n + (1-\beta_1)\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}, \tag{89}$$

where $\boldsymbol{v}_{n+1} = \begin{bmatrix} v_{0,n+1} & v_{1,n+1} & \cdots & v_{n_\theta,n+1} \end{bmatrix}^T$ then the per parameter learning rate is calculated with (87).

The bias due to initializing both $\boldsymbol{v}_0$ and $A_{i,0}$ to zero is remediated using the following recurrence equation for the velocity

$$\hat{\boldsymbol{v}}_{n+1} = \boldsymbol{v}_n \frac{1}{1-\beta_1^n}, \forall n > 0; \tag{90}$$

and for the per parameter learning rate

$$\hat{A}_{i,n+1} = \frac{\hat{A}_{i,n}}{1-\beta_2^n}, \forall i \in \mathbb{N}^{n_\theta}, n \in \mathbb{N}. \tag{91}$$

The parameter update is then given by

$$\theta_{i,n+1} = \theta_{i,n} + \frac{\alpha}{\sqrt{\hat{A}_{i,n}}}\hat{v}_{i,n}, \forall i \in \mathbb{N}^{n_\theta}, n \in \mathbb{N}. \tag{92}$$

3.2.5.5 NADAM

Nesterov Adaptative Moments Estimation (NADAM) was presented by (DOZAT, 2015), and it basically introduced the Nesterov Momentum to ADAM. This translates to changing the velocity update of (89), to

$$\boldsymbol{v}_{n+1} = \beta_1 \boldsymbol{v}_n + (1-\beta_1)\frac{\partial J(\boldsymbol{\theta}_n + \beta_1 \boldsymbol{v}_n)}{\partial \boldsymbol{\theta}_n}, \tag{93}$$

and the per parameter learning rate of (87), to

$$A_{i,n+1} = \beta_2 A_{i,n} + (1-\beta_2)\left(\frac{\partial J(\boldsymbol{\theta}_n + \beta_1 \boldsymbol{v}_n)}{\partial \theta_{i,n}}\right)^2, \forall i \in \mathbb{N}^{n_\theta}, n \in \mathbb{N}. \tag{94}$$

with $A_{i,0} = 0$.

All the other updates remain the same as in ADAM. NADAM delivers better models than ADAM, RMSProp and pure SGD, for the problem in (DOZAT, 2015) and is considered one of the best algorithms for training DNNs (GÉRON, 2022).

3.2.6 Backpropagation

As seen, to train a DNN, usually some extension of the SGD algorithm is used. Many such extensions were discussed in section 3.2.5 and all of them require the calculation of the cost function's gradient with respect to the parameters of the DNN.

Since the DNN is compactly represented as a repeated composition of functions, a recurrence over the layers, to calculate the gradients w.r.t the parameters one needs to recursively apply the chain rule of differentiation, which if not done carefully, can make the gradient calculation become computationally prohibitive. In fact, if the algorithm to calculate the gradient of a DNN is naive, i.e., it blindly selects a weight and calculates the gradient of the cost function w.r.t that weight, its computational complexity is $\mathcal{O}(2^n)$ for a DNN with n nodes (GOODFELLOW; BENGIO; COURVILLE, 2016).

However, the effective evaluation of the gradients can be done using dynamical programming, its use to calculate derivatives over computational graphs was rediscovered many times by independent researches, but it was first published by Seppo Linnainmaa in his Phd thesis (GRIEWANK, 2012; LINNAINMAA, 1976, 1970) and deemed automatic differentiation, where it was applied to general acyclic graphs. Its application in DNNs to calculate gradients was popularized by (RUMELHART; HINTON; WILLIAMS, 1986) and deemed backpropragation (BP) in this context.

The backpropragation algorithm main idea is to note that to calculate the gradients w.r.t two close parameters in the DNN graph many of the calculations are repeated due to its compositional structure. Thus, to save computational time, one could save the intermediate calculations and just reuse them when needed, an approach known as memoisation and that makes gradient calculations have a computational cost of at most $\mathcal{O}(n^2)$ (GOODFELLOW; BENGIO; COURVILLE, 2016), greatly reducing the computational cost of the gradient calculation when compared with the naive approach.

The backpropagation algorithm works by applying the chain rule in a specific order, starting in the output node and going progressively deeper, up to the first layer, aggregating the derivatives, this backwards calculation is the reason behind the algorithm name. A more in-depth discussion of the backpropagation algorithm is presented in (CHARU, 2018; RUMELHART; HINTON; WILLIAMS, 1986; GOODFELLOW; BENGIO; COURVILLE, 2016).

3.2.7 Recurrent Neural Networks

Although a deep neural network is powerful enough to represent any possible function, for some problems this comes at the cost of a prohibitively big number of parameters. There are techniques to alleviate and reduce the number of parameters that are simple to implement and are effective, one of such techniques is trading layer width for number of layers, i.e., making the DNN deeper.

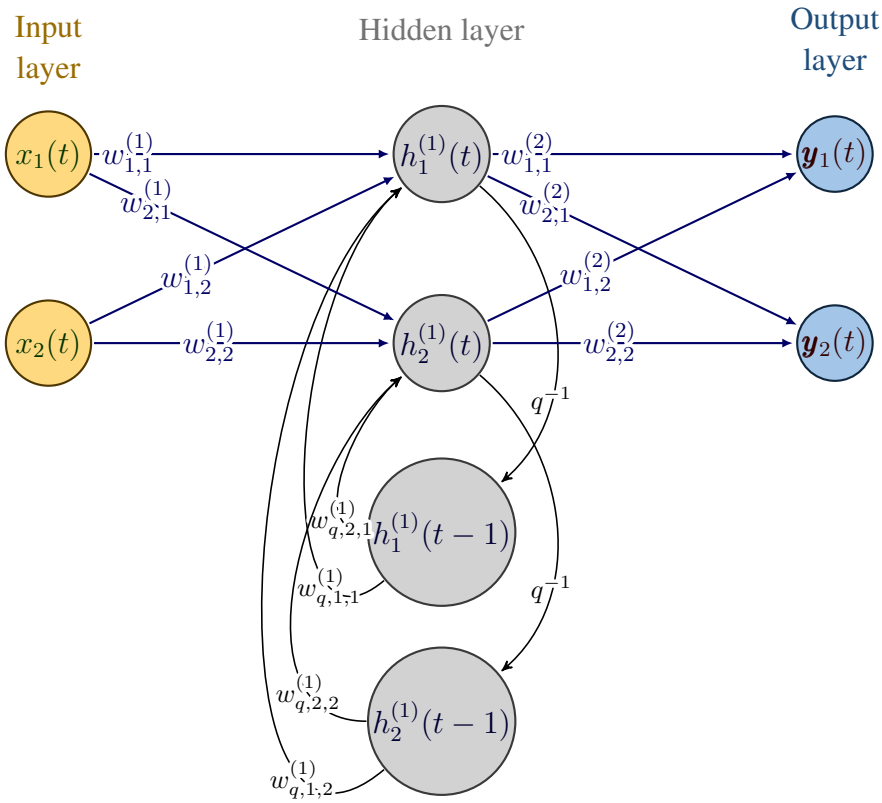
This option solves the problem of a big number of parameters but at the cost of training

time, which can also often be prohibitively. Another option is to share parameters between two sections of the neural network that should have similar properties, this is the case with encoder-decoder architectures (GOODFELLOW; BENGIO; COURVILLE, 2016).

For sequential data such as data generated with dynamical systems, text or any system that generates data that can be interpreted as a sequence, a very successful special architecture is known as Recurrent Neural Networks, which basically introduces cycles in the DNN graph. This means that each neuron computation is not static anymore but rather a recurrence, the origin of the Recurrent in the name of such architectures.

A useful tool in interpreting such architectures is to consider that every sequence is a sequence that varies in time, which allows us to treat RNNs as any other discrete time system and simplifies the notation, a simple Recurrent Neural Networks (RNN) is shown in Figure 11.

Figure 11 – Recurrent Neural Network.



Source: author

As shown in Figure 11, each input and output of the neural network, including the outputs of the recurrent hidden layers, are represented as a function of time, this is because RNNs operate in a sequence. Also, the recurrent hidden neurons have as inputs the outputs

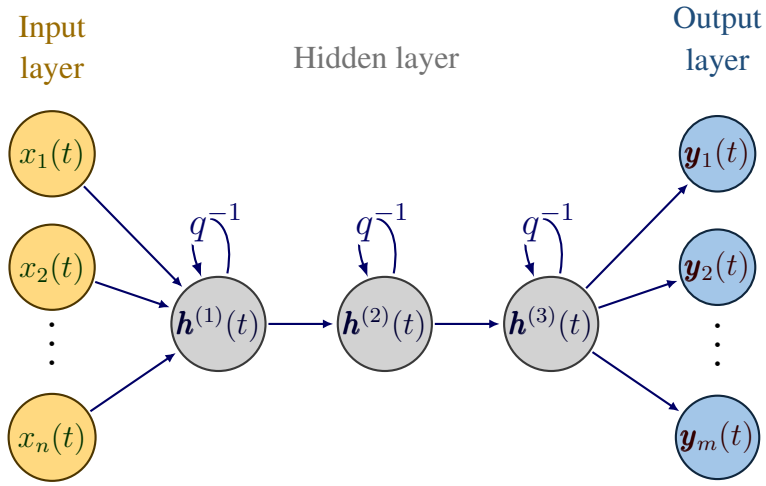
of the previous hidden layer[9] and its own output in the last time step, both parametrized by its sets of weights, $w_{i,j}^{(1)}$ for the feed-forward inputs and $w_{q,i,j}^{(1)}$ for the feed-back. In this fashion, the output of the i-th hidden layer at time t, is, in the matrix notation of (60), given by,

$$\boldsymbol{h}^{(i)}(t) = \sigma^{(i)}(W^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_q^{(i)}\boldsymbol{h}^{(i)}(t-1) + \boldsymbol{b}^{(i)}) = \boldsymbol{h}^{(i)}(\boldsymbol{h}^{(i)}(t-1), \boldsymbol{h}^{(i-1)}(t)). \quad (95)$$

where $\boldsymbol{h}^{(0)}(t) = \boldsymbol{x}(t)$, $W^{(i)} \in \mathbb{R}^{p_{i-1} \times p_i}$ and $W_q^{(i)} \in \mathbb{R}^{p_i \times p_i}$.

Another way to graphically represent a RNN is to use the matrix representation of each layer output, as given in (95), and done in Figure 12.

Figure 12 – Deep Recurrent Neural Network in Vector Format



Source: author

Figure 12 shows a RNN of the same type as the one shown in Figure 11, although with two more hidden layers and in a much compacter form than before.

In simple RNNs, as is the case of the one shown in Figure 11, the output of each hidden layer is only a function of its output in a previous time and the output of the previous hidden layer in the actual time, but this need not be the case. In fact, the most successful kinds of RNNs have outputs that differ from its internal states, as is the case with the Long Short Therm Memory (LSTM) cell and with the Gated Recurrent Unit (GRU).

Recurrent networks need to be trained using an extension of the Backpropagation algorithm, known as Backpropagation Through Time (BPTT), this is because the standard backpropagation algorithm cannot handle cycles in its graph. What the BPPT algorithm does is to "unroll" the graph in time by calculating the recurrence, once that is done the resulting equation is not a recurrence anymore and normal BP can be applied (CHARU,

[9] In this case, where there is only a single hidden layer, the previous hidden layer is the input layer.

2018; GOODFELLOW; BENGIO; COURVILLE, 2016; GÉRON, 2022). Unrolling in time can be better understood with the following simple example.

Example 3.2.2 (Unrolled RNN)**.** *Considering a RNN that has one input signal with five input samples, i.e., the input signal $x(t)$ enters the RNN as*

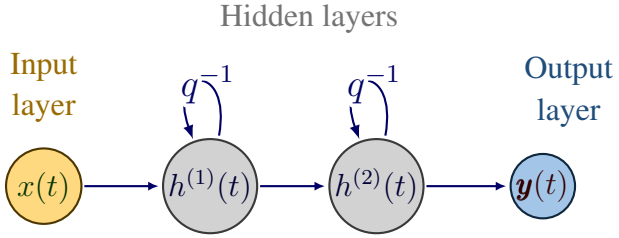
$$\boldsymbol{x}(t) = \begin{bmatrix} x(t-4) & x(t-3) & \cdots & x(t) \end{bmatrix}^T, \tag{96}$$

and the RNN uses this to predict the next four samples of the output signal $y(t)$, i.e., the output predicted by the RNN is

$$\boldsymbol{y}(t) = \begin{bmatrix} y(t+1) & y(t+2) & \cdots & y(t+4) \end{bmatrix}^T. \tag{97}$$

The RNN architecture for this setup consists of two hidden layers with one input node and one output node, the RNN representation is shown in Figure 13.

Figure 13 – Deep Recurrent Neural Network in Vector Format



Source: author

The RNN in Figure 13 has, in every node, an implicit associated recurrence equation in time, as in (95)*. E.g., the output of the second hidden node in the third sample is given by*

$$h^{(2)}(t-2) = h^{(2)}(h^{(1)}(t-2), h^{(2)}(t-3)), \tag{98}$$

which depends on the output of the previous layer on the present and the output of itself in the previous time step. Thus, the RNN graph has, as shown in Figure 11, cycles, which makes the application of the backpropagation algorithm impossible.

Since the cycles in RNN's graphs are generated by recurrences in time, it is actually possible and simple to get rid of them by applying an operation that is usually known as unrolling. Unrolling an RNN just means to evaluate the recurrence up to the first input, for the case of (98)*, ignoring the input of the previous layer*[10]*, this amounts to*

$$h^{(2)}(t-2) = h^{(2)} \circ\ h^{(2)}(t-3) \circ\ h^{(2)}(t-4) \circ\ h^{(2)}(t-5), \tag{99}$$

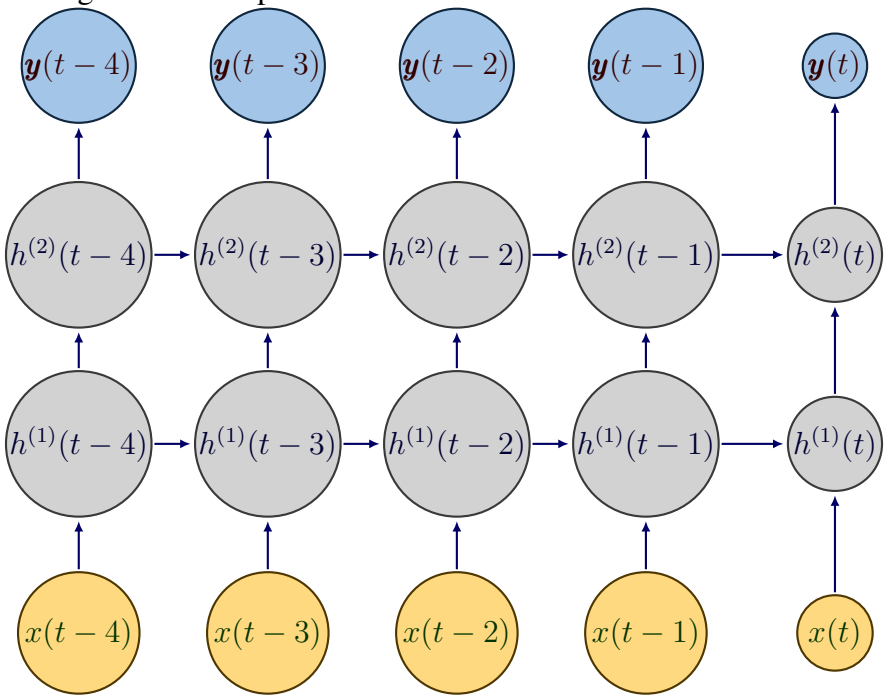
where the initial state is $h_1^{(2)}(t-5) = 0$.

Noticeable, equation (99) *is not a recurrence equation anymore, thus, applying the recurrence up to the point where it reaches its initial states results in function that, although*

[10] For notational clarity.

very complex, does not depend on some unknown past state. Unrolling all recurrent layers results in a graph representation similar to the normal DNN graph, Figure 14 shows the unrolled RNN of Figure 13.

Figure 14 – Deep Recurrent Neural Network in Vector Format



Source: author

The major effect of unrolling the RNN graph is that it makes it a feed-forward graph, as shown in Figure 14, where the normal backpropagation algorithm can be applied on. Hence, BPPT amounts to just unrolling the RNN graph and applying normal BPP.

It should be noted that the process of unrolling an RNN is done automatically by many packages, specially by package used in this work to create and train DNNs: the TensorFlow python package (GÉRON, 2022). This make implementing RNNs just as simple as implementing RNNs.

Also, very importantly, as shown in the 3.2.2, the initial state is assumed to be zero, although this common practice for many problems, it does not need to be always the case (GÉRON, 2022; GOODFELLOW; BENGIO; COURVILLE, 2016; CHARU, 2018). In this work, all recurrent networks have zeroed initial states, even when in application.

Another interesting result the Example 3.2.2 shows is that although the RNN shown in Figure 14 has output predictions at each input time step, it is possible to see the RNN as a function that maps from $\boldsymbol{x}(t)$ to $\boldsymbol{y}(t)$ by just ignoring all the other output vectors. Thus, it would be possible to create a dense feed-forward network to do the same mapping, which would, in comparison with the RNN shown in Figure 14, have much more connections and, hence, parameters. Also, the hidden nodes $h^{(1)}$ and $h^{(2)}$ appear multiple times in the graph, this means that the unrolled graph not only have fewer connections than its dense

counterpart would, but also many of the connections use the same parameters, that is one of the main advantages of using RNNs.

It is interesting to notice that the true system that drove $x(t)$ to $\boldsymbol{y}(t)$ is assumed to be the same system that drove $x(t+1)$ to $\boldsymbol{y}(t+1)$, [11] and since the RNN explicitly enforces this, the parametrization starts closer to the true system than an alternative dense feed-forward representation would.

3.2.7.1 Sequence-to-vector

The RNN shown in Figures 14 is an input sequence type RNN, which means that at each time step, one sample is provided to the RNN. It is also possible to have an input vector type RNN, in that case, the same input sample, say $x(t)$, would be provided to the RNN in each time step, which is an approach often used in language models (GÉRON, 2022).

A similar division is made for the output type of the RNN, which can be either of sequence or vector type. All input sequence type RNNs work as the unrolled graph of Figure 14 shows, i.e., after each input sample, an output with as many samples as defined in the RNN architecture is produced. The difference between sequence-to-vector (S2V) from sequence-to-sequence (S2S) model types is that in the first model type, although the model produces one output at each time step, only the output of the last time step is used to calculate the loss function and, hence, to update the weights in training (GÉRON, 2022).

To train a S2V model, the data contained in the dataset needs to be divided, since a RNN operates on sequences, the model definition does not directly determine the input size, and the same model can be applied on a sequence of, say, 10 samples and 20 samples. That would not be the case if the model was a DNN, since it operates in a predefined number of input samples.

To train the model, however, the data in the dataset is divided in a fixed number of input samples n_i and output samples n_o, and a dataset with N samples can be transformed, by windowing (GÉRON, 2022), into a dataset with $N-(n_i+n_o)+1$ samples in it. Thus, for an input vector $\boldsymbol{x}(t) \in \mathbb{R}^{n_x}$, output vector $\boldsymbol{y}(t) \in \mathbb{R}^{n_y}$ and batch size N_b, the input dataset is a tensor $X \in \mathbb{R}^{N_b \times n_i \times n_x}$, with elements $X_{i,j,k}$, the i-th element of the dataset's input is, thus, given by

$$X_i(t) = \begin{bmatrix} \boldsymbol{x}^T(t-(n_i-1)) \\ \boldsymbol{x}^T(t-(n_i-2)) \\ \vdots \\ \boldsymbol{x}^T(t-1) \\ \boldsymbol{x}^T(t) \end{bmatrix}; \tag{100}$$

where each t from the sampled data generates an input example.

[11] Which is as valid as the assumption 2.1.1.

In training the model reads each j-th dimension in sequence, thus resulting in a similar training procedure as with the feed-forward model type, where a whole batch of data can be processed at once and in parallel. Since usually $N_b << N - (n_i + n_o) + 1$, there are $(N - (n_i + n_o) + 1)/N_b$ batches of inputs X, with the last one, if it is the case, with the rest of the data[12], i.e., assuming that the last batch has N_{bl} samples, with $N_{bl} < N_b$, the last input tensor $X \in \mathbb{R}^{N_{bl} \times n_i \times n_z}$.

Although the model makes a prediction after every input sample, only the last one is used to calculate the training cost function, hence, the dataset's output is a tensor $Y \in \mathbb{R}^{N_b \times n_o \times n_y}$, with elements $Y_{i,j,k}$, the i-th element of the dataset's input is, thus, given by

$$Y_i(t) = \begin{bmatrix} \boldsymbol{y}^T(t+1) \\ \boldsymbol{y}^T(t+2)) \\ \vdots \\ \boldsymbol{y}^T(t+(n_o-1) \\ \boldsymbol{y}^T(t+n_o) \end{bmatrix}; \tag{101}$$

where each t from the sampled data generates an output example.

3.2.7.2 Sequence-to-sequence

In the S2S model type, the output of all time step, not only the last one, is used to train the model, even to in usage only the last prediction is used. This means that the loss function is calculated after each time step along with the gradients and weight update. This might seem quite counterproductive, but this approach results in many more weight updates that generalize better and converges faster, thus, using a sequence-to-sequence model results in a model that trains faster and has better generalization to the test set (GÉRON, 2022).

Since both S2V an S2S model types have a sequence input, the input dataset to train the model does not change. The output, however, does change, for the S2S model type the dataset's output is a tensor $Y \in \mathbb{R}^{N_b \times n_i \times n_o \times n_y}$ with elements $Y_{i,j,k,l}$, one $Y_i(t)$ of (101) for each input sample[13]. The (i,j)-th element is the output of the i-th batch sample for the j-th input, and is given by

$$Y_{i,j}(t) = \begin{bmatrix} \boldsymbol{y}^T(t-(n_i-j)+1) \\ \boldsymbol{y}^T(t-(n_i-j)+2) \\ \vdots \\ \boldsymbol{y}^T(t-(n_i-j)+n_o-1) \\ \boldsymbol{y}^T(t-(n_i-j)+n_o) \end{bmatrix}; \tag{102}$$

[12] Since the number of batches given by the formula is usually not an integer, the last input tensor has the rest of the samples.

[13] Instead of just one for the last input sample in the S2V case.

where, once again, each t from the sampled data generates an output example.

Equation (102) shows how each output of the dataset is transformed in training, the next example shows this transformation for a toy dataset, which, hopefully, better depicts the discussed transformations.

Example 3.2.3 (Sequence-to-sequence dataset)**.** *To clarify the dataset's transformations alone, consider the simplest possible setup: the input and output vector are one dimensional, i.e.,* $n_x = n_y = 1$*; there is only a single example per batch, i.e.,* $N_b = 1$*; the input and output samples are* $n_i = n_o = 2$*.*

The input and output data are the same, implicitly this means that the model is given the past two values of some measured signal and is to predict the next two values of the same signal. The input-output signal is

$$X = Y = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix}^T. \tag{103}$$

In the following, to better show the dataset's input and output tensors, all dimensions with size 1 will be collapsed.

The input dataset is formed as defined in (100)*, which can be seen as sliding a window of size 2 in the data, the dataset's input, a rank 2 tensor in this case, is*

$$X = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 5 & 6 \end{bmatrix}; \tag{104}$$

which would be the same for both S2V and S2S model types.

The output vector, a rank 3 tensor in this case, is shown sliced in the batch dimension, i.e., each Y_i *element at time.*

$$Y_1 = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \quad Y_2 = \begin{bmatrix} 3 & 4 \\ 4 & 5 \end{bmatrix} \quad Y_3 = \begin{bmatrix} 4 & 5 \\ 5 & 6 \end{bmatrix} \quad Y_4 = \begin{bmatrix} 5 & 6 \\ 6 & 7 \end{bmatrix} \quad Y_5 = \begin{bmatrix} 6 & 7 \\ 7 & 8 \end{bmatrix} \tag{105}$$

In this example, although in a very simple setup, it is possible to see that the for each input sample, i.e., each line in X*, of every example in the dataset, there is an associated output. For the more intuitive S2V model type, the output dataset would be just the second row of each* Y_i *2-tensor. Thus showing that, in this simple case where* $n_i = n_o = 2$*, the S2S model type has already double the amount of gradient calculations, and, more importantly, weight updates.*

3.2.7.3 LSTM

Although the RNN was presented with a recurrent layer that has states equaling outputs and that calculate each new output/state via (95), this need not be the case. In fact, the

recurrent node can have any number of internal states, in those more complex cases the recurrent node is more often referred to as cell, an abbreviation to memory cell, technically, however, all recurrent neurons are cells.

A cell can be seen as a form of memory, since it contains information of past input/output values encoded in its states. When dealing with long sequences, the simple RNN of (95) have a tendency to forget the first inputs since the next state only explicitly depends on the past state.

One cell that tries to fight this problem is the Long Short Term Memory (LSTM), which is constructed to have, in essence, an extra state vector that holds the most relevant features of the input sequence for longer. In practice this is done using four fully connected layers inside each cell, thus, the i-th LSTM layer, formed with p_i LSTM cells, has, each:

- one layer that controls what to store in the long-term state $\boldsymbol{c}^{(i)}(t) \in \mathbb{R}^{p_i}$;
- one that controls what to erase from $\boldsymbol{c}^{(i)}(t-1)$;
- one that analyses the short-term memory state $\boldsymbol{h}^{(i)}(t) \in \mathbb{R}^{p_i}$ and input $\boldsymbol{h}^{(i-1)}(t)$;
- one that controls the output, which, in this case, is the short-term hidden state $\boldsymbol{h}^{(i)}(t)$;

A more complete evaluation of the parts of this cell is provided in the original paper (HOCHREITER; SCHMIDHUBER, 1997) and in (GÉRON, 2022; CHARU, 2018), the calculations that a layer of LSTM cells performs at each time step are given by

$$\begin{aligned}
\boldsymbol{i}^{(i)}(t) &= \sigma_{\text{sgm}}(W_i^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_{qi}^{(i)}\boldsymbol{h}^{(i)}(t-1) + \boldsymbol{b}_i^{(i)}) \\
\boldsymbol{f}^{(i)}(t) &= \sigma_{\text{sgm}}(W_f^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_{qf}^{(i)}\boldsymbol{h}^{(i)}(t-1) + \boldsymbol{b}_f^{(i)}) \\
\boldsymbol{o}^{(i)}(t) &= \sigma_{\text{sgm}}(W_o^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_{qo}^{(i)}\boldsymbol{h}^{(i)}(t-1) + \boldsymbol{b}_o^{(i)}) \\
\boldsymbol{g}^{(i)}(t) &= \sigma_{\text{tanh}}(W_g^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_{qg}^{(i)}\boldsymbol{h}^{(i)}(t-1) + \boldsymbol{b}_g^{(i)}) \\
\boldsymbol{c}^{(i)}(t) &= \boldsymbol{f}^{(i)}(t) \otimes \boldsymbol{c}^{(i)}(t-1) + \boldsymbol{i}^{(i)}(t) \otimes \boldsymbol{g}^{(i)}(t) \\
\boldsymbol{h}^{(i)}(t) &= \boldsymbol{o}^{(i)}(t) \otimes \sigma_{\text{tanh}}(\boldsymbol{c}^{(i)}(t))
\end{aligned} \tag{106}$$

where it is assumed that $\boldsymbol{h}^{(0)}(t) = \boldsymbol{x}(t)$, the bias vectors, i.e., $\boldsymbol{b}_i^{(i)}$, $\boldsymbol{b}_f^{(i)}$, $\boldsymbol{b}_o^{(i)}$ and $\boldsymbol{b}_g^{(i)}$, are all $\in \mathbb{R}^{p_i}$. The short-term state, which is the output of the cell, is $\in \mathbb{R}^{p_i}$ and the weight matrices have dimensions that match the biases dimensions and make the equations valid, e.g., $W_i^{(i)} \in \mathbb{R}^{p_{i-1} \times p_i}$. Also, in (106), $\otimes$ is the hadamard product of two vector, i.e., for vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^p$ the hadamard product is $\boldsymbol{c} = \boldsymbol{a} \otimes \boldsymbol{b}$ and has the i-th element given by $c_i = a_i b_i$.

In (106) the calculations are shown for a whole layer, thus the signals on the right hand side (RHS) belongs to $\mathbb{R}^{p_i}$, for a single cell, the signal $i^{(i)}(t)$ is the input gate signal,

$f^{(i)}(t)$ is the forget gate signal and $o^{(i)}(t)$ is the output gate signal. Each one of those signals play a role in controlling the long and short-term states.

The LSTM equations also show that each LSTM layer has significantly more parameters that a simple RNN layer would, from (95) the weight matrices $W \in \mathbb{R}^{p_i \times p_i}$ $W_q \in \mathbb{R}^{p_{i-1} \times p_i}$ which gives a total of $p_i(p_i + p_{i-1})$ parameters per layer, while the LSTM has $4p_i(p_i + p_{i-1})$, which can more easily be seen by noting that each of the signals $i^{(i)}(t)$, $f^{(i)}(t)$, $o^{(i)}(t)$ and $g^{(i)}(t)$ can be considered a separate simple RNN.

LSTMs also give better characteristics on training since they are less prone to the vanishing and exploding gradients, a common problem in simple RNNs (GÉRON, 2022; CHARU, 2018). This, together with the capability of learning longer patterns in data more easily, makes the LSTMs a powerful tool to model the behavior of long sequences in general and is specially useful for the purpose of representing a controller map $\mathcal{C}$.

3.2.7.4 GRU

The Gated Recurrent Unit (GRU) was introduced by (CHO *et al.*, 2014) and can be seen as a simplification of the LSTM cell, where both state vectors are merged in a single state vector $\boldsymbol{h}(t)$ and where only one gate controller signal $\boldsymbol{z}(t)$ controls the input and forget properties. The calculations that a GRU layer perform are

$$\begin{aligned}
\boldsymbol{z}^{(i)}(t) &= \sigma_{\text{sgm}}(W_z^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_{qz}^{(i)}\boldsymbol{h}^{(i)}(t-1) + \boldsymbol{b}_z^{(i)});\\
\boldsymbol{r}^{(i)}(t) &= \sigma_{\text{sgm}}(W_r^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_{qr}^{(i)}\boldsymbol{h}^{(i)}(t-1) + \boldsymbol{b}_r^{(i)});\\
\boldsymbol{g}^{(i)}(t) &= \sigma_{\text{tanh}}(W_g^{(i)}\boldsymbol{h}^{(i-1)}(t) + W_{qg}^{(i)}(\boldsymbol{r}^{(i)}(t) \otimes \boldsymbol{h}^{(i)}(t-1)) + \boldsymbol{b}_g^{(i)});\\
\boldsymbol{h}^{(i)}(t) &= \boldsymbol{z}^{(i)}(t) \otimes \boldsymbol{h}^{(i)}(t-1) + (1 - \boldsymbol{z}^{(i)}(t)) \otimes \boldsymbol{g}^{(i)}(t);
\end{aligned} \tag{107}$$

where it is assumed that $\boldsymbol{h}^{(0)}(t) = \boldsymbol{x}(t)$, the matrices and vectors sizes works as in the LSTM cell.

The GRU have similar performance to the LSTM cell (GREFF *et al.*, 2016), all while using 3/4 of the total number of parameter[14], this gives a model with fewer parameters and that needs less data to generalize well.

3.3 Regularization

Complex models have the capability to fit a great number of functions of the most diverse classes, this is very powerful and allows to easily represent problems where the true generating function is unknown. This power, however, comes with a downside, since the data is often contaminated by noise and datasets are of finite size, a very powerful model can be induced to learn the noise realization as if it was part of the true generating function, this is often known as over-fit. Since the noise realization is dataset dependent,

[14] Considering a RNN with the same number of cells and layers.

thus, a model that over-fits some dataset $\mathcal{A}$ is a model that yields a smaller cost function on $\mathcal{A}$ than on some other dataset $\mathcal{B}$.

It should be noted that this noise contamination in the dataset is not necessarily originated just from $\nu(t)$ on (1), it can also be originated by a dataset that does not explore all the input space in a sufficient manner. To see this, suppose that the dataset input x and output y is given by the true generating function as $y = f(x)$ with no noise, assume also that 99% of the input values of the dataset are in range $x \in [0, 1]$, and that 1% of the input values are in the range $x \in [2, 3]$, with a $f(x)$ that has a fairly distinct behavior in those two regions. Without knowing $f(x)$ and looking only to pairs (x, y), one could think that the different behavior of this 1% of data is actually originated from noise, while in reality is just a region of the input space that is not sufficiently explored.

There are two main approaches to deal with over-fitting, increasing the dataset and restricting the model complexity. On the first approach, over-fitting is prevented because the noise is more easily distinguished from the uncontaminated data. On the second approach, the model itself is made just powerful enough to learn the true generating statistics while not being so powerful that is allowed to memorize the noise realization.

Since acquiring more data tends to be expensive or even impossible, ideally, one of the best approaches is to have a model that is just powerful enough to learn the true generating statistics. This ideal size model would then not only not over-fit the data but would also be faster to train and need fewer data to do so. The problem with this approach is that one has usually little knowledge of the true generating statistics and, hence, of the needed complexity of the model. Finding this ideal model size is, thus, a problem and one solution is to manually increase the model size while monitoring the train and test set cost functions, the ideal model is then the smaller model that makes the cost function on the train and test set evaluate to close values.

Regularization is any process or strategy that automatically induces a model to be simpler, have less complexity or simply that produces a model with a smaller test error (GOODFELLOW; BENGIO; COURVILLE, 2016). There are many ways to regularize a model, the regularization strategies used in this work will be presented in the following subsections.

3.3.1 Parameter Regularization

One option to restrict the model capability is to penalize the model's parameters, this is done by modifying the cost function that is minimized. Assuming that the cost function of interest is $J(\boldsymbol{\theta}, X, Y)$, where $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$ is the parameter vector, X is the input values matrix and Y is the output, the new modified cost function is given by

$$J_p(\boldsymbol{\theta}, X, Y) = J(\boldsymbol{\theta}, X, Y) + \alpha_r P(\boldsymbol{\theta}); \tag{108}$$

where the penalizing function $P(\boldsymbol{\theta}) : \mathbb{R}^{n_\theta} \to \mathbb{R}$ is some measure of the parameters and α_r is the hyperparameter that weights the penalization.

This penalizing function is usually either the L^2 norm of the parameters or the L^1 norm, but it can be any metric. In this work only L^2 and L^1 penalizations will be used and presented since they're the ones that offer the properties that are of most interest.

3.3.1.1 L^2 Regularization

In the L^2 regularization the penalizing function is given by the squared L^2 norm of the vector, i.e.,

$$P(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{\theta}||_2^2 = \frac{1}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}. \tag{109}$$

Applying (109) into (108) and taking the gradient w.r.t $\boldsymbol{\theta}$, gives a weight update in the Stochastic Gradient Descent of

$$\boldsymbol{\theta}_{n+1} = (1 - \alpha_r\alpha)\boldsymbol{\theta}_n - \alpha\frac{\partial J(\boldsymbol{\theta}_n)}{\partial \boldsymbol{\theta}_n}. \tag{110}$$

Equation (110) shows that the weight update is now modified to also decrease the weights in an amount that is proportional to the weights.

By analyzing the cost function (108) and (110) it is possible to see that the minimum of J_p is given by some set of parameters that minimize J while being as small as possible, this property is shown more clearly by the weight update rule, which shows that the weights walk on a path that is a compromise between making J and themselves smaller, this prevents the weights from growing too much to fit spurious patterns on the data as part of the true generating statistics, thus working as a form of regularization.

3.3.1.2 L^1 Regularization

L^1 regularization works the same as L^2 but uses, instead, the L^1 norm of the vector, i.e.,

$$P(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_1 = \sum_{i=1}^{n_\theta} |\theta_i| \,. \tag{111}$$

Applying (111) into (108) and taking the gradient w.r.t $\boldsymbol{\theta}$ gives a weight update for the SGD of

$$\boldsymbol{\theta}_{n+1} = \alpha_r \text{sign}(\boldsymbol{\theta}_n) - \alpha\frac{\partial J(\boldsymbol{\theta}_n)}{\partial \boldsymbol{\theta}_n}. \tag{112}$$

where the signal[15] function sign is applied element-wise.

With the L^1 penalization, the weight update rule of (112) is, as with L^2, in the direction that minimizes J and, unlike L^2, happens in the weights' direction (positive or negative) always with the same value, regardless of the weight's values, this promotes non-important weights not only to be small but to be exactly zero. It is interesting to notice that L^2 regularization does not zero the unimportant weights because the smaller a

[15] sign(a) is 1 if $a > 0$, -1 if $a < 0$ and 0 otherwise.

weight is, the less it will be pushed to zero, which is a consequence of the weight update penalization being proportional to the weight. Because of this property, L^1 regularization can be seen as an automatic feature selection mechanism.

3.3.1.3 Parameter Regularization in RNNs

It is possible to divide the parameters of a RNN layer in three sets, input, recurrent and bias weights. The input weights are those that multiply the output from the previous layer[16], while the recurrent are those who multiply the output of the actual layer in the previous time step. In this work, all recurrent matrices are indicated with a subscript q, e.g., in a LSTM layer, the recurrent weight matrix of the input gate signal is W_{qi}.

Since in RNNs those different parameters have very different roles, they also often have very different sizes. Thus, a parameter regularization that penalizes input and recurrent weights with the same intensity might penalize one much more than the other. For this reason, in actual implementation, the parameter regularization of (108) is modified to

$$J_p(\boldsymbol{\theta}, X, Y) = J(\boldsymbol{\theta}, X, Y) + \alpha_{ri} P_i(\boldsymbol{\theta}_i) + \alpha_{rr} P_r(\boldsymbol{\theta}_r) + \alpha_{rb} P_b(\boldsymbol{\theta}_b); \tag{113}$$

where $\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_i^T & \boldsymbol{\theta}_r^T & \boldsymbol{\theta}_b^T \end{bmatrix}^T$, and $\boldsymbol{\theta}_i$ is the input weights vector, $\boldsymbol{\theta}_r$ is the recurrent weight vector and $\boldsymbol{\theta}_b$ is the bias weight vector, with obvious names for the penalization parameters $\alpha_{r[\cdot]}$ and penalizing functions.

This approach allows controlling more precisely the penalization strength in each set of weights and often gives better results.

3.3.2 Gaussian Noise Contamination

The work of (SIETSMA; DOW, 1991) indicate that injecting Gaussian noise with zero mean on the input of the DNN during training greatly helps them to generalize better. This was latter proved to be equivalent to adding an extra term to the error function by (BISHOP, 1995).

To inject noise in the inputs, before the n-th weight update, the input matrix $X \in \mathbb{R}^{N \times n_x}$ is summed with the matrix Ψ. With the element of the i-th row and j-th column $\psi_{i,j}$ given by

$$\psi_{i,j} \sim N(0, \sigma_\psi^2), \tag{114}$$

where the standard deviation σ_ψ is a hyper-parameter to be selected. It should be noted that the noise here is applied only in the input of the model, and not at every layer.

This form of regularization can be seen as an artificial dataset augmentation (GOODFELLOW; BENGIO; COURVILLE, 2016). The reasoning is that, assuming a small enough σ_ψ, an input $\boldsymbol{x}_\sigma$, the contaminated version of $\boldsymbol{x}$ whose output is $\boldsymbol{y}$, is close to its uncontaminated counterpart and should have an output that is also close to $\boldsymbol{y}$. This

[16] Which is the input of the actual layer.

approach make so that virtually every data presented to the model during training is different, which augments the dataset proportionally to the number of weight updates.

This form of regularization also helps the model to be less sensitive to noise as (SIETSMA; DOW, 1991) suggests, making for a form of regularization that is very interesting for models to be used in noisy environments, as is the case with $\mathcal{C}$. And, as the results of this work indicates, is very effective, helping both training and in reducing the model's noise sensitivity.

The idea of Gaussian noise contamination can be extended to the inputs of every layer in the model, and also to the parameters itself, as shown in (GOODFELLOW; BENGIO; COURVILLE, 2016), which shows also another famous form of regularization known as Dropout, which can be seen as a form of constructing a new input via multiplying them by noise.

3.3.3 Regularization in VRFT context

Regularization is also studied in the context of the VRFT for linearly parametrized controllers, as (FORMENTIN; KARIMI, 2014) shows, L^2 regularization improves the statistical performance significantly when the data is corrupted by noise, namely, it reduces the variance when the VRFT problem is solved with instrumental variables, the same result is extended to multi-input multi-output (MIMO) systems by (BOEIRA; ECKHARD, 2019), which also had better closed loop performance when using L^2 regularization.

In the work of (FORMENTIN; KARIMI, 2014), the selection of penalizing factors is done using an automatic procedure that does not allow including prior knowledge of the system onto the penalizing factors. The work of (RALLO *et al.*, 2016) expands on L^2 regularization when there is available knowledge of the plant model, the proposed approach gives better performance than the L^2 regularization of (FORMENTIN; KARIMI, 2014) when there is some knowledge about the system to be controlled.

In (PILLONETTO *et al.*, 2014), a broader evaluation of regularization in the system identification area is made, the work shows that regularization eases the problem of selecting a model complexity and is beneficial as long as the penalizing factors are chosen carefully.

Overall, the effect of introducing regularization techniques into the linear VRFT is of trading more bias for less variance, but since the true objective is not to produce an unbiased estimate of the parameters but to minimize J, regularization produces a closed loop controller that is statistically better with respect to the model reference cost function. This shows that L^2 regularization technics are a valuable tool in VRFT, and indicates that other forms of regularization, as the ones presented in this section, can be helpful to the problem's solution even in the nonlinear case, this hypothesis is corroborated with the case studies results, which all show better performance with regularization.

4 CASE STUDIES

In this chapter the VRFT method will be applied to two dynamical systems, a simple pendulum[1] and a direct current (DC) motor, to show the resulting controller, the main problems and the results.

4.1 Simple Pendulum

The simple pendulum is a common and intuitive dynamical system which is often used as test to control design methods. For this reason the VRFT method will be applied to this system to show some of the properties of the method in this more intuitive system.

Although the system dynamics will not be used to derive $\mathcal{C}$, it is important to precisely describe it, since this is what will generate the dataset and is where the control map will be applied. The angular position x_1 and velocity x_2 are given by

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -b_1 \sin(x_1) - b_2 x_2 + u \\ y &= x_1 + \nu(t) \end{aligned} \tag{115}$$

where b_1 is a constant that depends on the pendulum's moment of inertia, its mass and the gravitational constant, b_2 is the rotational friction, u is a torque applied to control the pendulum and the output y is the angular position contaminated by $\nu(t) \sim N(0, \sigma^2)$ with $\sigma = 2.31 \cdot 10^{-4}$. In this experiment, $b_1 = 1$ and $b_2 = 0.1$.

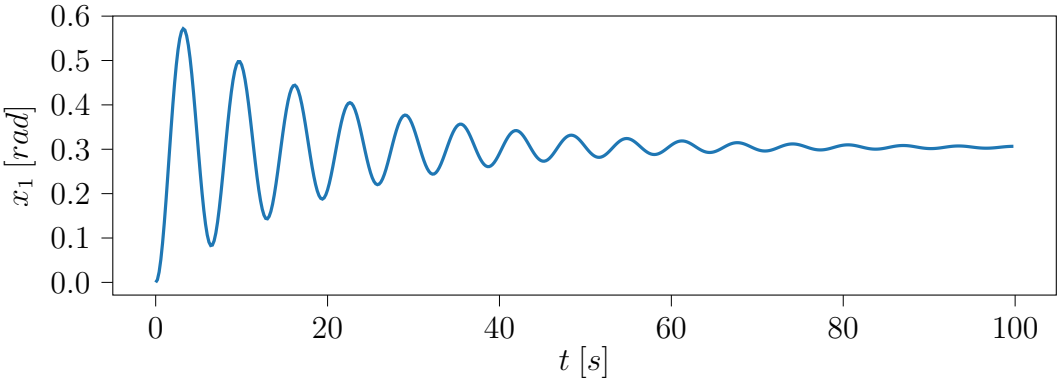
The reference model choice need to be made reasonably, because even without restricting bounds in the control, the physical system has limitations in the trajectories it can go through. Thus, if the designer chooses a reference model that is too far away from what can be achieved by the system, no controller class will make $J^V(\boldsymbol{\theta}) = 0$ and the minimum of $J^V(\boldsymbol{\theta})$ might give a controller that has little to do with $\mathcal{C}_r$.

When the system to be controlled is continuous, as is often the case, it makes more sense to define the reference as a continuous system as well, this decouples the choice of the reference model and the sampling period.

[1] A rod with mass fixed on a pin that allows rotation.

Choosing an adequate reference model requires some knowledge on the system's behavior. One could, of course, use the dynamics of (115) to determine that, but, in a realistic application, one would not have a precise model of the system, if any at all. Thus, an alternative that gives some information on the general characteristics of the system is to apply a step input and watch its response. The response of the system (115) to a step of 0.3 is presented in Figure 15.

Figure 15 – Step response of (115).



Source: author

From the Figure 15 one can see that the system has a strong oscillatory response, with an overshoot almost as big as the input itself. Also, the settling time is observed to be in the neighborhood of $70\ s$. With this information, seems reasonable to expect that the linearized system is at least second order.

4.1.1 Case 1

4.1.1.1 Reference Model

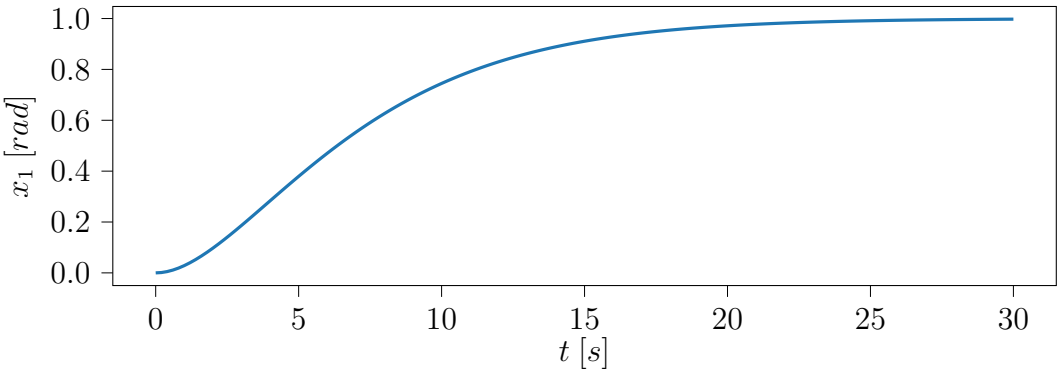
If one wants to have a closed loop system without overshoot and with about half the settling time of the open loop system, is sensible to define the continuous time Reference Model as the one with transfer function

$$S_r = \frac{0.3^3}{0.4}\frac{s+0.4}{(s+0.3)^3}, \tag{116}$$

with step response in the Figure 16.

The intuition on the choice of the reference model is that it accomplishes the goal of no overshoot with a unitary steady state gain while preserving some of the step response shape and with a settling time that, although smaller, is not too far away from the system's natural settling time.

Figure 16 – Step response of (116).



Source: author

There is another feature of the Reference Model, it has zeroes close to zero, this is important to mind because the virtual reference is calculated from the inverse of $\mathcal{S}_r$, thus, small zeroes will produce small poles in the inverted system, which helps to attenuate the noise effects in the output, which, in turn, ease the complexity of optimizing $J^V(\boldsymbol{\theta})$.

4.1.1.2 Experiment

To generate the IO-data required to apply the VRFT method, one needs to consider which input will be used to excite the system (115). For linear systems one of the most common signals used for this purpose is the Pseudo Random Binary Sequence (PRBS) signal, which is basically a sequence of steps where each one has amplitude α and interval T_i. The PRBS signal is so often used for two main reasons, first, it meets the excitation persistency requirement, second, it is easy to generate.

For a linear system, the amplitude α is the same for all steps because the local behavior of the system is also its global behavior. This is not the case for nonlinear systems, applying a PRBS signal in this case would likely result in gathering data of only a small portion of the system state space, which would be insufficient to characterize the whole operational region of the system.

A simple change that can be made to the PRBS signal and that allows the system to explore more of its state space is to give each step in the sequence its own amplitude α_i. To more precisely represent this signal, let

$$\mathcal{I}_i(t) = \begin{cases} 1, & \forall\ t \in A_i \\ 0, & \forall\ t \notin A_i \end{cases} \tag{117}$$

be the indicator function of the time interval $A_i = [t_{A_i,a}\ \ t_{A_i,b})$ with $t_{A_i,a}, t_{A_i,b} \in \mathbb{R}$ and with interval $T_i = t_{A_i,b} - t_{A_i,a}$. With this, the variable amplitude pseudo random binary sequence signal[2] $s_s(t)$ is

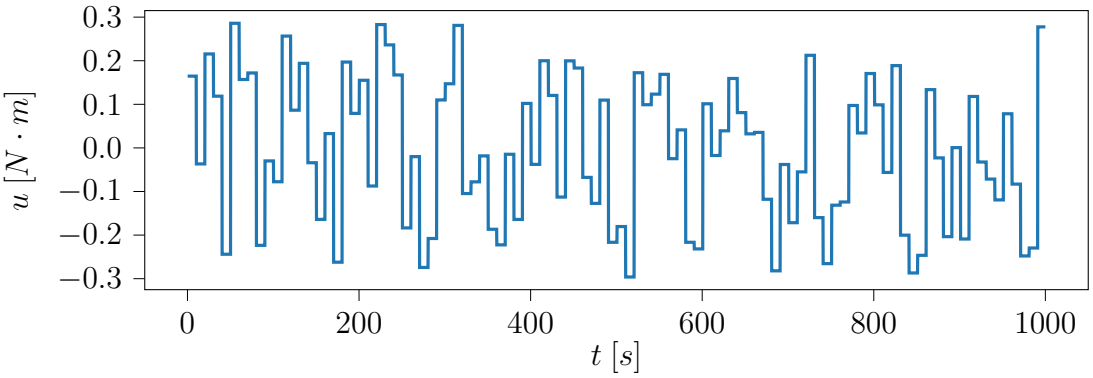
$$s_s(t) = \sum_{i=0}^{n_A} \alpha_i \mathcal{I}_i(t), \tag{118}$$

[2]Also known as multi-level PRBS.

where n_A is the number of steps and each amplitude α_i and time interval T_i are sampled from a uniform distribution.

For this problem, the system will run for $T = 10^4\ s$ and the data will be collected with a sampling time of $\Delta t = 0.2\ s$. To properly stimulate the system, a $s_s(t)$ with $n_A = 10^3$, $T_i = 10\ s$ and $\alpha_i \sim U(-0.3, 0.3)$ was chosen, a portion of this signal is presented in Figure 17.

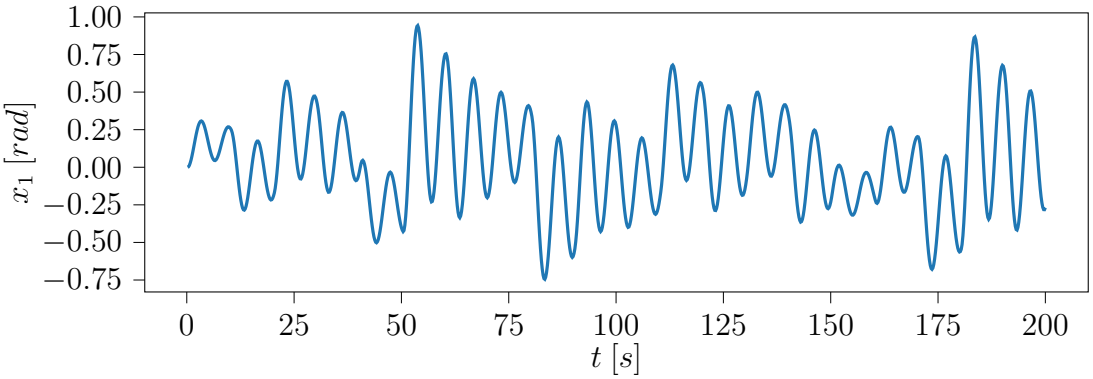
Figure 17 – Part of the sequence of steps used to excite the system.



Source: author

To obtain the data, the system of (115) is simulated with the control of Figure 17, since the output of the system is corrupted with a white noise of zero mean and standard deviation $\sigma = 2.31 \times 10^{-4}$, it has a signal-to-noise ratio of about 10^3. The noisy output is shown, partially, in Figure[3] 18.

Figure 18 – Part of the output of the system.



Source: author

4.1.1.3 Filtered Signals and Final Dataset

To solve this problem, the filter of (39), that depends on S_r, will be used. Its application is as simple as applying the filter in the linear case, with no need to apply the filter in the predicted control or to propagate the derivatives with respect to the parameters, as

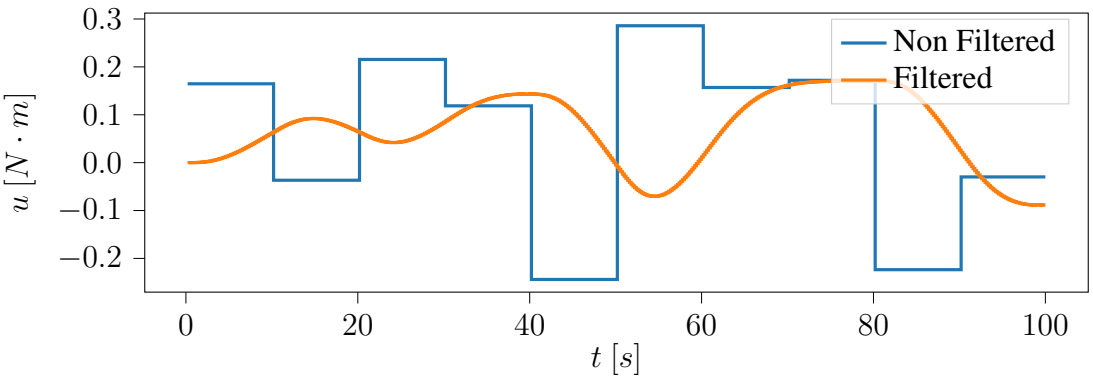
[3] It is hard to see the noise in the Figure, that is because the signal to noise ration is high in this case.

is the case with the other two filters. In this fashion, the filtered signals are created by applying (116) in (39) and calculating an a that gives L_D unitary steady state gain. Thus, u_L and e_L are

$$u_L = L_D(q)u(t); \qquad e_L = L_D(q)e(t); \tag{119}$$

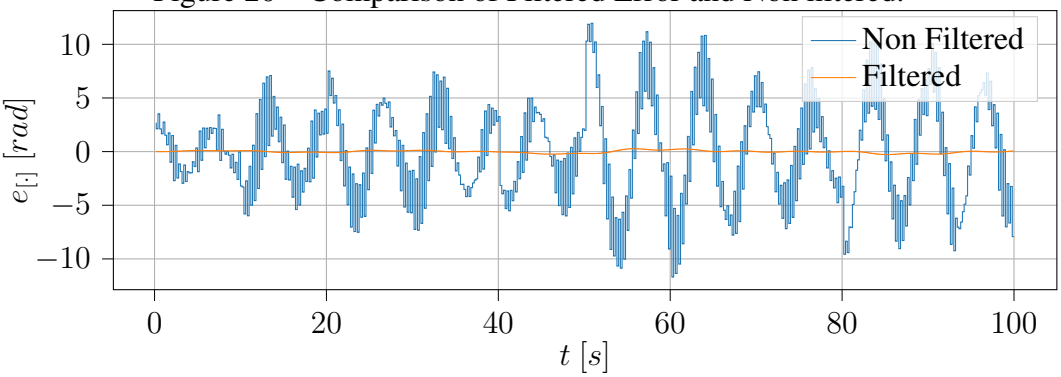
A comparison between the filtered and unfiltered signals is presented in Figures 19 and 20 .

Figure 19 – Comparison of filtered and non filtered input.



Source: author

Figure 20 – Comparison of Filtered Error and Non filtered.



Source: author

As shown in Figures 19 and 20, the application of the filter greatly affects the signals $u(t)$ and $e(t)$, where higher frequencies are attenuated, an effect that is more evident in $e(t)$. Although the filter wasn't designed with this specific effect as objective, the filtered signals have characteristics that seem to better match with the ones of the reference model.

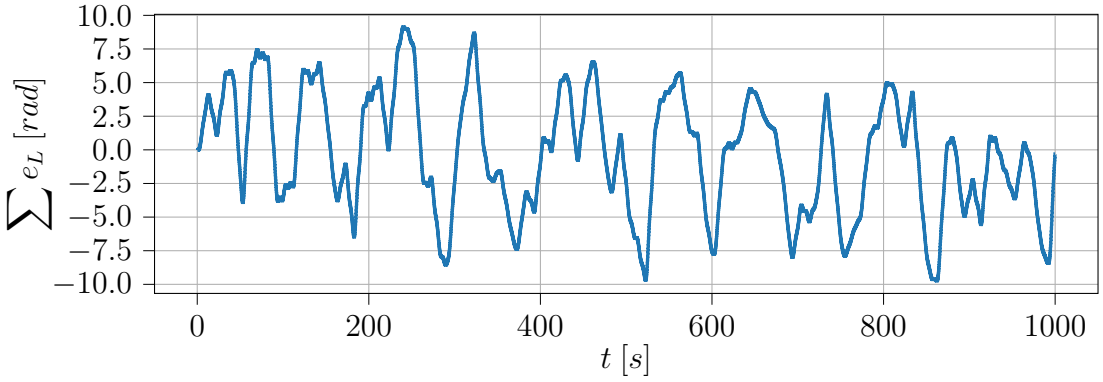
Whether using a polynomial or DNN parametrization, it is necessary, before training the model, to define the measure map $\mathcal{Z}$. The definition should be made in a way that eases the problem of finding $\mathcal{C}$. Since the controller needs to give the closed loop system a unitary steady state gain, the controller as whole, i.e., $\mathcal{C}_\mathcal{Z}$, must have an integrator on the error, unless, of course, if the system happens to have one already, but since one can not

usually know that in advance[4], it is the best practice to include one in the measurement map. It is also interesting to input the controller with a direct measure to the system output and the error, a measurement map that accomplishes those requirements is

$$\mathcal{Z}[r(t)\boldsymbol{q}_t^0, e_L(t)\boldsymbol{q}_t^0, y(t)\boldsymbol{q}_t^0] = \begin{bmatrix} y(t) \\ e_L(t) \\ \sum_{\tau=0}^{\tau=t} e_L(\tau) \end{bmatrix}, \tag{120}$$

With this definition it is possible to filter with $\mathcal{Z}$ all the data to construct all the signals that, along with the control signal, will be used to train $\mathcal{C}$. It is interesting to observe that the integrated error, presented in Figure 21, is much smoother than the virtual error, which potentially helps to find a simpler map $\mathcal{C}$.

Figure 21 – Integrated Error (filtered).



Source: author

To train the controller all that is needed is to construct the database, the input database[5] is thus defined as

$$X = \begin{bmatrix} u_L(t_0) & y(t_0) & e_L(t_0) & e_L(t_0) \\ u_L(t_1) & y(t_1) & e_L(t_1) & e_L(t_1) + e_L(t_0) \\ \vdots & \vdots & \vdots & \vdots \\ u_L(t_N - 1) & y(t_N - 1) & e_L(t_N - 1) & e_L(t_N - 1) + e_L(t_{N-1}) \end{bmatrix}; \tag{121}$$

and the output as

$$Y = \begin{bmatrix} u_L(t_1) & u_L(t_2) & \cdots & u_L(t_N) \end{bmatrix}^T; \tag{122}$$

It is important to scale each signal in the database to force them to be in the same range, this is a common approach in machine learning and general optimization (GÉRON, 2022) and it is important because without it, the model will potentially be biased to place more significance into inputs of greater scale. Also, with DNNs specifically, scaling the data is important because if a saturating activation function is used, using data that has

[4] Unless a specific test is made on the system.

[5] Before the S2S transformation is applied.

a big range might result in many neurons being saturated, which effectively reduce their potential influence in the output.

The database is scaled such that every column is in the range $[a\ b]$, which is accomplished by what is usually known as min-max scaler, the scaled database input is thus created with

$$X_{i,S} = (X_i - \min(X_i))\frac{a-b}{\max(X_i) - \min(X_i)} + b, \tag{123}$$

where X_i is the i-th column of the database and, in this problem, $a = -1$ and $b = 1$. The same procedure is applied to create the scaled database output.

4.1.1.4 DNN-Controller

With the database created, all that is left to do is to define the DNN architecture. In this case a more complex architecture than a polynomial basis will be used, in the attempt to capture more of the controller dynamics, giving a closed loop response that is closer to the reference model.

The chosen architecture is composed of an input layer of 4 neurons, followed by a Gaussian Noise layer with $\sigma = 0.1$, two GRU layers, each with 64 cells, and a dense output layer with linear activation function and one neuron. This architecture gives a model with 38465 trainable parameters. The GRU layers have a hyperbolic tangent for the output activation function and a sigmoid for the states. Also, only L_1 recurrent regularization is applied, with $\alpha_{rr} = 1$. It is interesting to notice that this architecture was chosen after some experimentation and that using a number of cells smaller than 64 generates controllers with only a slightly worse performance than the one herein presented, all while using much less parameters. The same applies to the activation functions used and all other hyper-parameters , the ones here presented are the ones that was found to produce the best predictors, i.e., they were defined via experimentation.

The input layer has four neurons because that is the number of signals in the database's input. Since this is a recurrent model, the input size does not tell the whole story, it is also necessary to define how many input samples (in time) are provided for each prediction, for this model, the database is constructed to give the last ten samples of the input database in order to predict the next control output, i.e., the control at time t, in this case, is given by

$$u(t) = \mathcal{C}[\boldsymbol{q}_9^0 \boldsymbol{z}(t), u(t)\boldsymbol{q}_{10}]. \tag{124}$$

The model is also chosen to be of type sequence-to-sequence since this improves convergence and diminishes the training time. For this, the database output is modified accordingly, as presented in subsection 3.2.7.2. The batch has size $N_b = 64$, which is chosen as compromise between training time and the regularization properties of using small batches. The training set consists of 64% of the whole dataset, the validation set is

16% and the test set consists of 20%.

The training is made using a modified loss function that also slightly penalizes the predicted control, this can be seen as a form of regularization that prevents the predicted control to be bigger than the actual control, thus forcing the errors to be in the smallest magnitude side. The modified loss function is

$$J^{Vm}(\boldsymbol{\theta}) = J^{V}(\boldsymbol{\theta}) + \alpha_u \sum_{t=0}^{N} \hat{u}(t)^2, \tag{125}$$

where, for this problem, the penalization factor $\alpha_u = 1 \times 10^{-3}$.

The selected optimizer for this problem is the NADAM, with initial learning rate of 1×10^{-4}. The plateau learning rate reducer is utilized in training, it is configured to shrink the learning rate by 10% after 5 epochs[6] with no improvement in the validation set, up to a minimum learning rate of 1×10^{-6}. The model is trained up to the point where 20 epochs have passed with no improvement on the validation set, which helps to prevent over-fitting.

The final training error, measured on the unscaled signals for a better understanding, is of $J^{V}_{train}(\boldsymbol{\theta}) = 2.42 \times 10^{-6}$ with a test set error of $J^{V}_{test}(\boldsymbol{\theta}) = 2.19 \times 10^{-6}$, which shows good generalization of the model in the test set.

4.1.1.5 Analysis

Since the optimization is performed on $J^{Vm}(\boldsymbol{\theta})$, a modified version of $J^{V}(\boldsymbol{\theta})$, having a small $J^{V}_{train}(\boldsymbol{\theta})$ and $J^{V}_{test}(\boldsymbol{\theta})$ does not necessarily guarantee a good controller, this is because in this case, where noise is present and the parametrization is not linear in the parameters, the value of J^{V} might be a local minimum or, in case of a $\mathcal{C}(\boldsymbol{\theta})$ with low representational power, a minimum that although global, still does not make $\mathcal{C}(\boldsymbol{\theta}) \approx \mathcal{C}_r$.

To evaluate the trained controller performance, the best approach is to close the loop with the found $\mathcal{C}(\boldsymbol{\theta})$ and to directly calculate the Model Reference cost function J. To this end, a simulation is performed using the dynamics of (115), where the control action is calculated at every $\Delta t = 0.2\ s$ and applied in the continuous system in a ZOH fashion. The simulation is performed in python, using the *scipy* implementation of an explicit Runge-Kutta method of order 5, presented in (WANNER; HAIRER, 1996).

Using this simulation, Figure 22 shows the response to a step of $0.3\ rad$.

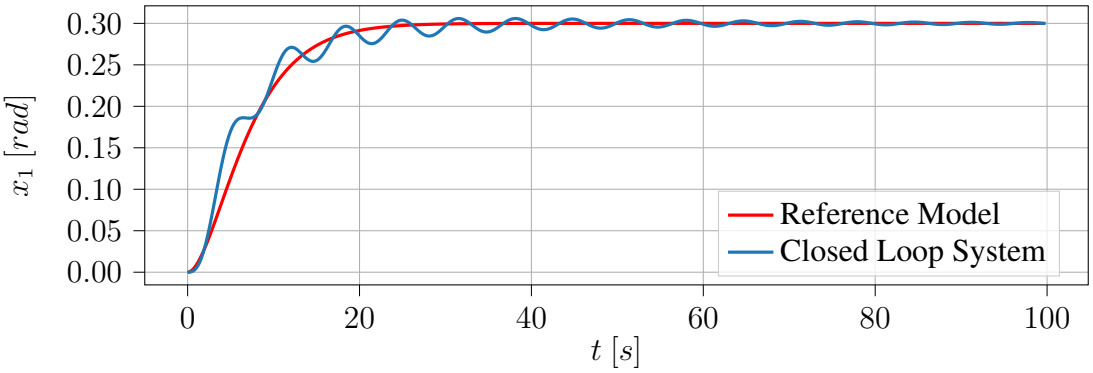
The step response in Figure 22 has a model reference error of $J(\boldsymbol{\theta}) = 1.13 \times 10^{-4}$, measured on all the $N = 500$ samples of the figure, and $J(\boldsymbol{\theta}) = 2.74 \times 10^{-4}$ if measured on the first $N = 200$ samples, i.e., up to $t = 40\ s$.

The control action for this case, up to $t = 40\ s$, is show in Figure 23.

In Figure 23 is possible to confirm that the control action does not try to correct the faster dynamics of the system, that ultimately makes the closed loop system oscillate

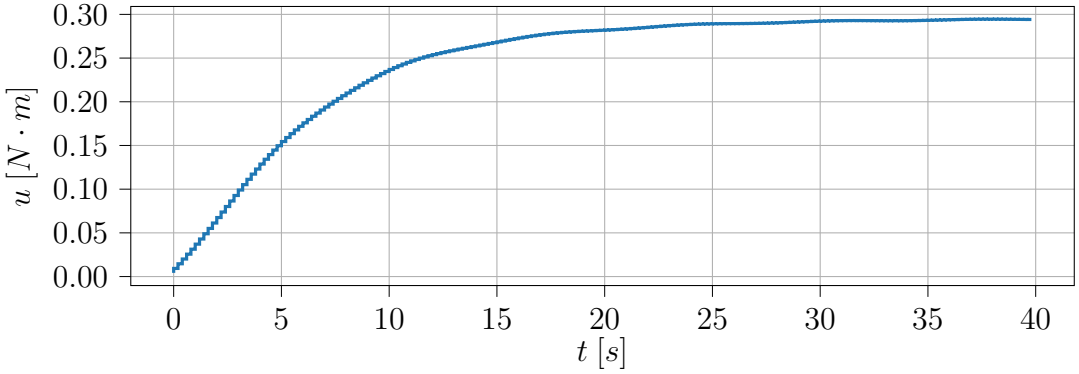
[6] An epoch represents a full pass over the entire dataset.

Figure 22 – Comparison of closed loop system with the Reference Model to a step response.



Source: author

Figure 23 – Closed loop control action of Figure 22.



Source: author

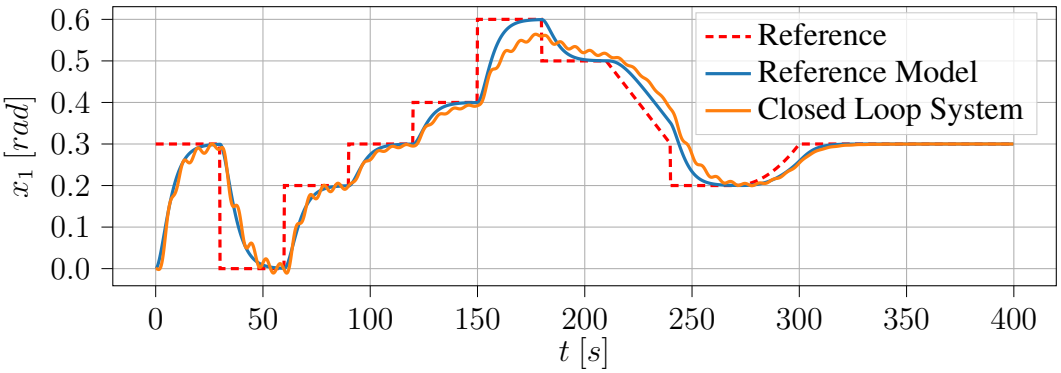
around the reference model output. This seems to indicate that the controller $\mathcal{C}(\boldsymbol{\theta})$ is close to $\mathcal{C}_r$ only in low frequencies. Moreover, this also seems to indicate that $\mathcal{C}(\boldsymbol{\theta})$ has bigger prediction errors particularly in this kind of trajectory.

To test this, one can consider that the closed loop response was actually made by the Reference Model, which allows calculating the virtual reference and hence all the other necessary inputs to $\mathcal{C}(\boldsymbol{\theta})$. In this fashion, it is possible to calculate the prediction error in this specific trajectory, which amounts to $J^V_{closed} = 2.06 \times 10^{-5}$, an error of 763.84% when compared with J^V_{test}. It is interesting to notice that this is not, at least in the usual sense, a case of over-fitting, since, as shown, the test set has a prediction error similar to the train set. What this indicates is that the entire dataset is not representative of the trajectories of interest.

This hypothesis will be further investigated later, first, it is also fruitful to analyze the closed loop system behavior to a more challenging reference. Figure 24 shows the system and reference model response to a sequence of varying amplitude steps, ramps and second order reference.

As shown, up to a reference of $0.3\ rad$, the closed loop system behaves similarly

Figure 24 – Comparison of closed loop system with the Reference Model to a more challenging reference.



Source: author

regardless of to the step amplitude and operating point, showing that, at least in this range, $\mathcal{C}$ cancels out the nonlinearity of the system, making the closed loop behavior close to linear[7], closely following the Reference Model output for the most part.

When the reference is bigger than $0.3\ rad$, however, the closed loop response starts to degrade, this is mainly due to fact that the database only contains data with control in the range of $[-0.3, 0.3]$, thus making $\mathcal{C}$ not generalize so well when far away from this range.

Going back to the hypothesis that the database is not representative of the dynamics in the region of interest: ideally the objective is to identify a controller that is capable of rendering closed loop responses exactly equal to the response of the Reference Model to any type of reference. This, however, might be hard to do in practice and, since the closed loop system will often only follow references of a step type, a good database would be representative of the Reference Model following such references.

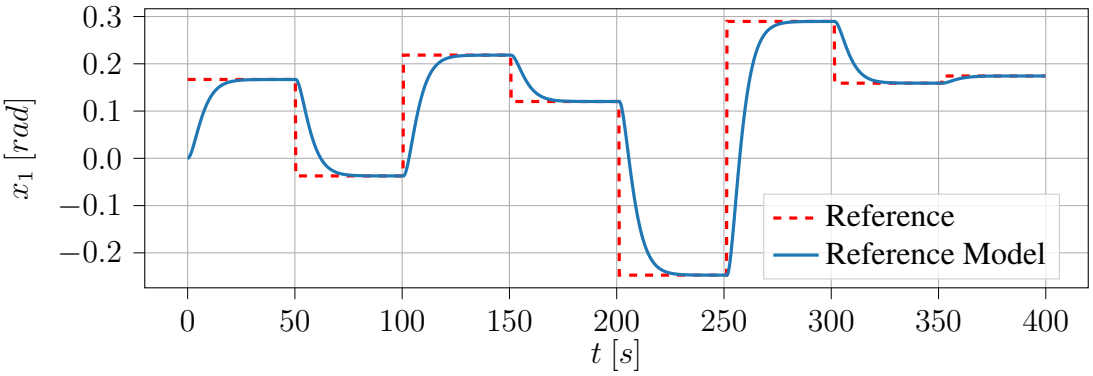
If the system was to be controlled by the ideal controller, it would behave exactly the same as the reference model, and, in this setup, the ideal controller would map errors to a control action that the system would, in turn, map to an output. Thus, an ideal dataset $\mathcal{D}_I$ would be one that has the same distribution of errors and outputs that the reference model has when subject to a sequence of varying amplitude steps.

Creating this error-output distribution amounts to simply applying a sequence of steps of varying amplitudes to the Reference Model and to track the distribution of errors and outputs. To this end, a sequence of steps $s_s(t)$ is created with $n_A = 200$, $\alpha_i \sim U(-0.3, 0.3)$ and $T_i = 50\ s$. This reference and the Reference Model output are shown, up to $t = 400\ s$, in Figure 25.

The reference in Figure 25 is chosen with $T_i = 50\ s$ to give the Reference Model time to achieve its steady state, that is because the system, when in application, would be feed by references that also allow the same behavior. Changing the reference would

[7] As desired, since the Reference model itself is linear.

Figure 25 – Reference and Reference Model output for the creation of the ideal dataset.

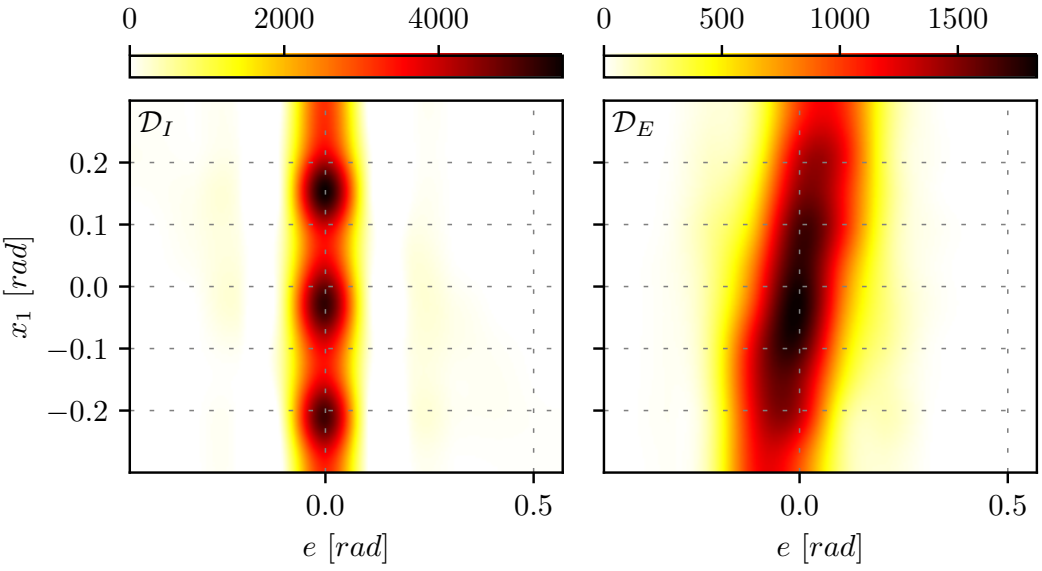


Source: author

of course change the error-output distribution and thus ideal dataset. This means that the ideal dataset is only in fact ideal if the final objective is to make the closed loop system follow references exactly like the one presented.

A good way to see the data distribution is to put the data in a plane $e - x_1$, divide it in small regions and count how many times a point in the dataset falls into this region, assigning then a color to each amount creates what is often known as a heatmap, which is shown in Figure 26 for the ideal dataset $\mathcal{D}_I$ and experimental dataset $\mathcal{D}_E$.

Figure 26 – Ideal and experimental dataset distribution.



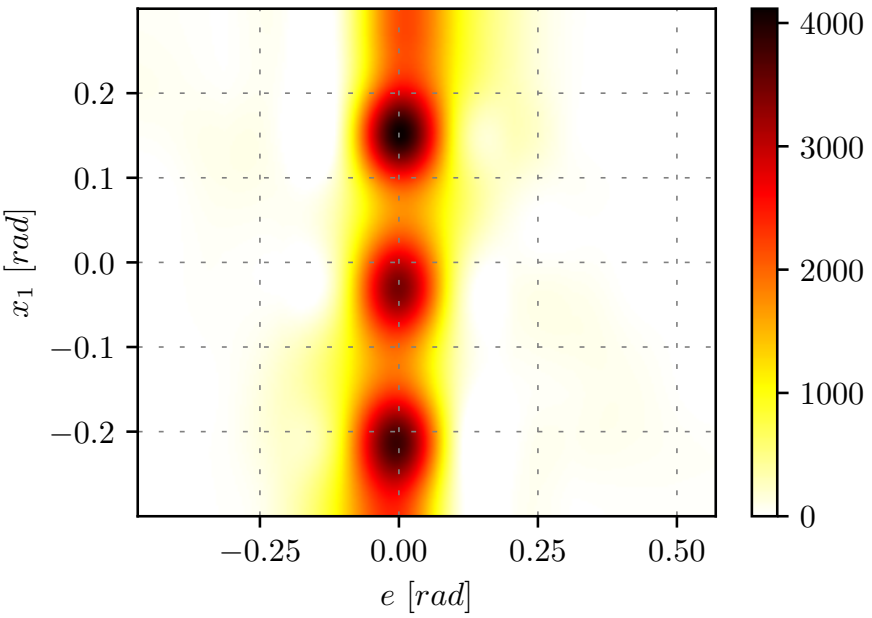
Source: author

The $\mathcal{D}_I$ heatmap of Figure 26 reveals many properties of the ideal dataset, first, the data is concentrated in regions with small errors, which is to be expected since the reference changes slow enough for the Reference Model to attain zero error most of the time. Also, the dataset has regions of bigger errors fairly distributed, those account for times where bigger changes happen in the reference, something that has a smaller chance due

to the uniform distribution of amplitudes in the sequence of steps. Other, perhaps unexpected, property of the ideal dataset is that the position is concentrated in regions instead of being uniformly distributed in the range $[-0.3,\ 0.3]$, this, however, follows the same distribution of the reference signal and it is a consequence of the small sample number $n_A = 200$, used to create the reference signal.

The $\mathcal{D}_E$ heatmap, partially used to train $\mathcal{C}(\boldsymbol{\theta})$, is shown in Figure 26, in the same range used in the $\mathcal{D}_I$ to a better comparison. It is interesting to calculate the difference of the heatmaps, to do this all that is needed is to take absolute difference of the number of samples in each sub-region, and to assign a color that goes from zero to maximum difference, this creates another form of heatmap, this time, however, the strongest color represents the higher difference in the datasets, this difference heatmap is shown in Figure 27, where it should be noted that although the ideal dataset is created with the reference amplitudes α_i sampled from a uniform distribution, the set of sampled amplitudes in $s_s(t)$ is not so uniform due to the low number of samples.

Figure 27 – Difference heatmap between ideal and experimental datasets.



Source: author

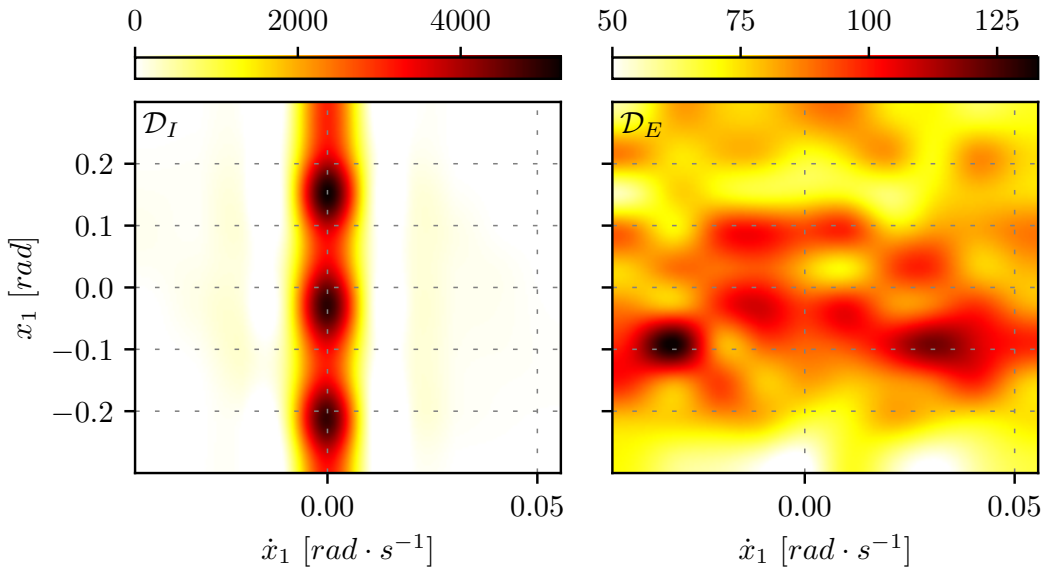
The difference heatmap, shown in Figure 27, shows that the experimental dataset does cover most of the areas that the ideal dataset covers, as indicated by the maximum difference shown in the colored label. Apart from the three black regions created by the low sample number in the sequence of steps, three more major differences exists between the datasets. First, the error e is much larger in the experimental datasets; Second, negative outputs tend to have negative errors while positive outputs tend to have positive errors, this is a feature of the experimental dataset that is not observed in the ideal dataset; The last major difference is that the outputs are more normally distributed around zero than in

the ideal dataset. Also, it is interesting to notice that the experimental dataset's heatmap is created in the same range as the ideal heatmap, but 28.24% of the data is outside the shown range.

Although useful, the heatmaps just presented do not tell the whole story about the dataset, this is because in its creation there is no consideration about the sequence of samples, which is something of great importance given that $\mathcal{C}$ maps not only from samples at the current time but also from past times. The rate at which the output and error changes is also of importance and, ideally, the experimental and ideal datasets would have similar velocity distributions.

To make this similarity analysis, the output's rate of variation is calculated using the second order finite difference method for both the ideal and experimental datasets. The data distribution's heatmap in the $x_1 - \dot{x}_1$ plane is shown in Figure 28 for the datasets $\mathcal{D}_I$ and $\mathcal{D}_E$.

Figure 28 – Ideal and experimental dataset velocity distribution.



Source: author

The $\mathcal{D}_I$ heatmap of Figure 28 shows that the output velocity is very small and centered around zero in a shape that, although in a different scale, resemble the error distribution of the $\mathcal{D}_I$ dataset of Figure 26. The same, however, does not happen with the output's velocity heatmap of the experimental dataset, which shows no similarity with the $\mathcal{D}_I$ heatmap. The difference is so big that the data outside $\mathcal{D}_E$ heatmap range's amounts to 83.73% of the total dataset.

All those differences between the ideal and experimental dataset make for a very hard problem to solve when training $\mathcal{C}(\boldsymbol{\theta})$, since a great deal of generalization would be required of it. This means that even if $\boldsymbol{\theta}$ minimizes $J^V(\boldsymbol{\theta})$ in the training set, because the experimental dataset is not so representative of the dynamics in the region one wish to

apply $\mathcal{C}(\boldsymbol{\theta})$ to, $\boldsymbol{\theta}$ does not necessarily minimizes $J^V(\boldsymbol{\theta})$ in this region of interest. This is exactly why $\boldsymbol{\theta}$ minimizes $J^V_{train}(\boldsymbol{\theta})$ and $J^V_{test}(\boldsymbol{\theta})$ but does not minimize $J^V_{closed}(\boldsymbol{\theta})$.

Another way to understand these effects is that the representative power of the DNN is spent learning features of the dataset that are of little to no use in the regions explored by the system when in closed loop, thus, less of DNN representative power is spent where it actually matters, resulting in a suboptimal predictor and controller.

4.2 DC Motor

This plant is a third order model of a field-controlled DC motor, its dynamics is extracted from (KH, 1996) and given by

$$\begin{aligned}
\dot{x}_1 &= -\frac{R_f}{L_f}x_1 + \frac{1}{L_f}u \\
\dot{x}_2 &= -\frac{R_a}{L_a}x_2 + \frac{1}{L_a}v_a - \frac{k_b}{L_a}x_1x_3 \\
\dot{x}_3 &= \frac{k_m}{J}x_1x_2 - \frac{K_f}{J}x_3 \\
y &= x_3 + \nu(t)
\end{aligned} \tag{126}$$

where u is the field voltage, x_3 is the angular velocity, v_a is the armature voltage and $\nu(t) \sim N(0, \sigma^2)$ with $\sigma = 0.78$. The other parameters are given in Table 1.

Table 1 – DC-Motor parameters

Parameters		
$R_f = 5$	$L_f = 3 \times 10^{-4}$	$R_a = 2$
$L_a = 6 \times 10^{-4}$	$v_a = 15$	$k_b = 5 \times 10^{-2}$
$k_m = 5 \times 10^{-2}$	$J = 7 \times 10^{-6}$	$K_f = 48 \times 10^{-9}$.

To obtain some information about the system's characteristics, a unitary step is applied onto the system, the results are shown in Figure 29.

Figure 29 – Step response of (126).

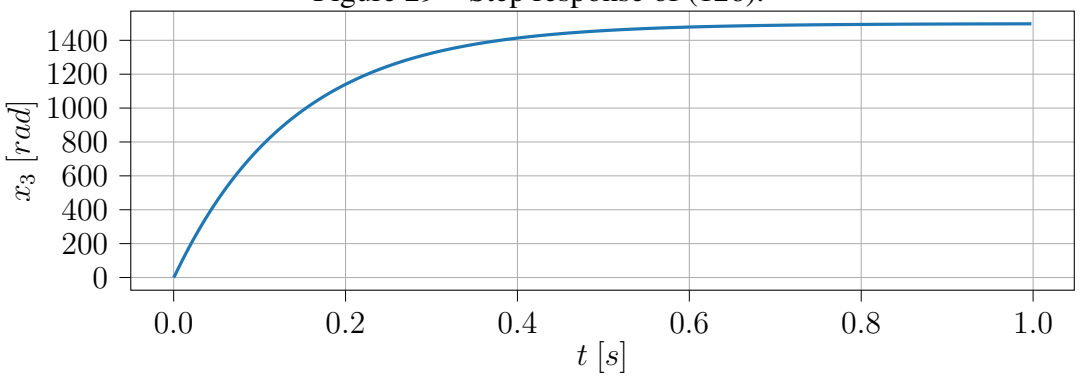


Figure 29 shows that the system is not oscillatory, has a settling time of about $t_s = 0.4\ s$ and a gain of 1500 in this unitary step. Applying steps of smaller amplitudes shows that the system has both settling time and gain that grows inversely proportional to the step amplitude. From the system response's shape, seems reasonable to expect that the linearized system is at least first order.

4.2.1 Case 1: polynomial controller

The results of this subsection were published in (BAZANELLA; ECKHARD; SEHNEM, 2023).

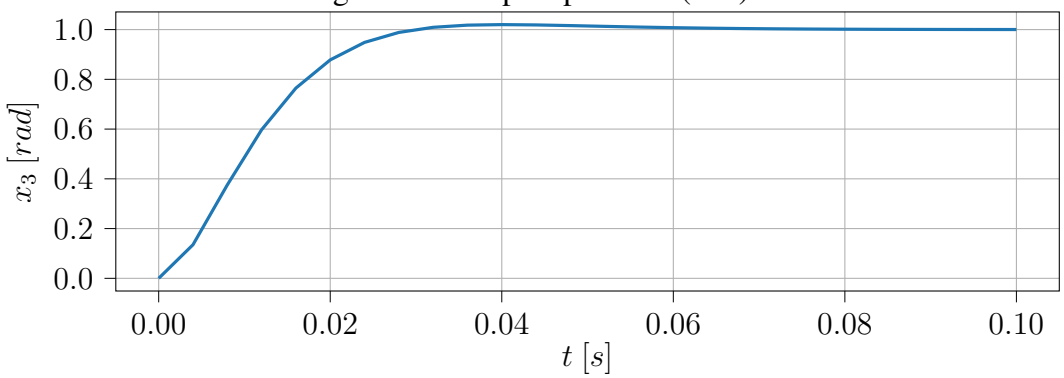
4.2.1.1 Reference Model

The reference model is more easily derived in the continuous time domain, since this is also the natural domain of the plant. The desired continuous transfer function is

$$T_d(s) = \frac{120^3}{70}\frac{(s+70)}{(s+120)^3} \tag{127}$$

with step response shown in Figure 39.

Figure 30 – Step response of (127).



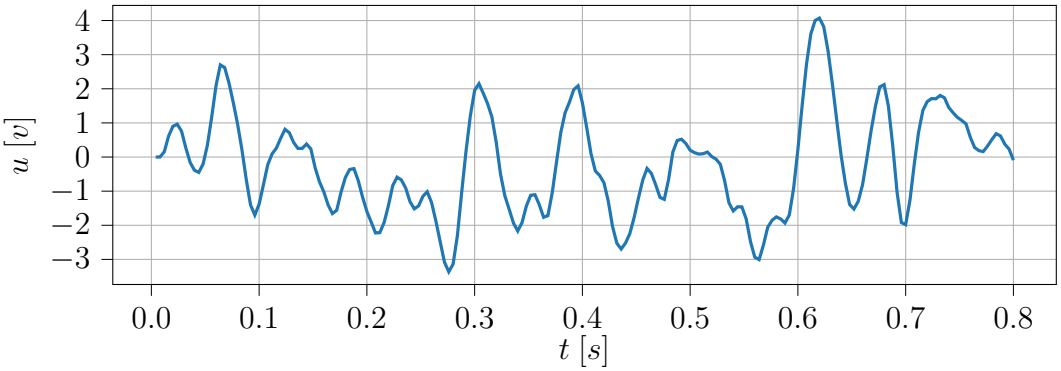
Source: author

The poles here are selected to give the system a settling time that although smaller, is not too far off of what is observed in open loop, whereas the zeros are selected to give $T_d(s)$ the same relative degree as the plant, without causing too much overshoot. The discrete time reference model $T_d(z)$ is obtained by applying a ZOH to (127) with sampling time of $\Delta t = 4\ ms$.

4.2.1.2 Experiment

To generate the IO-data for this case, a white noise where each sample $w(t) \sim U(-1, 1)$ is first created, this white noise is then filtered using the filter of (39) and linearly scaled to be in range $[-5, 5]$, this input data is shown in Figure 31.

Figure 31 – Part of the input used to excite the system.



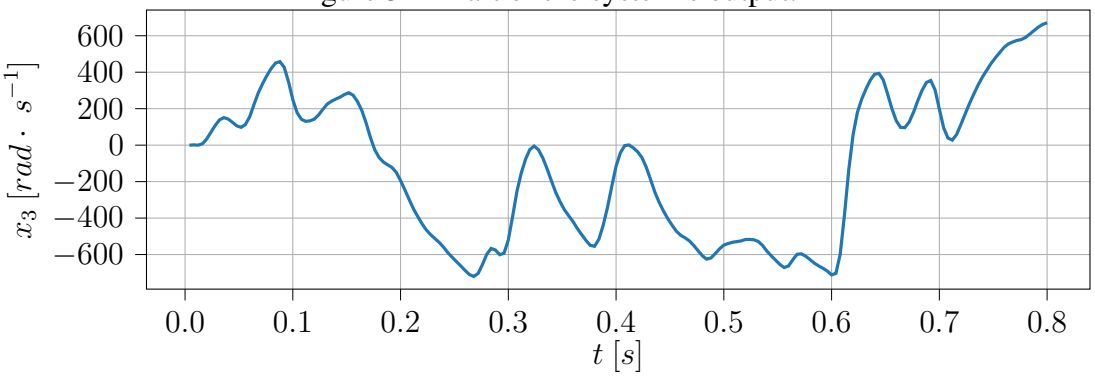
Source: author

Using a filtered white noise is also a viable and practical way of defining an input signal to excite the system, it is used here to show that in this simpler case, where the parametrization is linear in the parameters, it suffices to use such signal.

Figure 31 shows that the input used to excite the system never settles to any particular point and constantly oscillates around zero in a way that seems to explore a lot of the control, i.e., oscillations of all amplitudes in all the $[-5, 5]$ range.

To obtain the IO-data, the system of equation (126) is simulated with the signal of Figure 31 for $T = 20\ s$, since this gives enough time for the system to meaningfully explore its dynamics. The data is collected with a sampling time of $\Delta t = 4 \times 10^{-3}$. The noisy collected output is shown, partially, in Figure 32.

Figure 32 – Part of the system's output.



Source: author

The output of the system shown in Figure 32 shows that the system outputs seems to follow about the same pattern as the input, oscillations of various amplitudes in all the range of interest, which is ideal to capture, in the dataset, the complete dynamical behavior of the system.

4.2.1.3 Filtered Signals and Final Dataset

The virtual error is constructed using the output signal presented in Figure 32 with (13) while the input signal is the one of Figure 31, this, along with the measurement map $\mathcal{Z}$, is used to construct the database to train $\mathcal{C}(\boldsymbol{\theta})$. The measurement map is selected to be the error integral since zero tracking error is required, a measurement map that accomplishes this requirement is

$$\mathcal{Z}[r(t)\boldsymbol{q}_t^0, e_L(t)\boldsymbol{q}_t^0, y(t)\boldsymbol{q}_t^0] = \sum_{\tau=0}^{t} e(\tau) \tag{128}$$

4.2.1.4 Polynomial Controller

If the general form of the system's dynamics is known, the usage of a simpler parametrization for the controller, i.e., one that is linear in the parameters, is possible. This form of parametrization requires the definition of a dictionary of nonlinear functions that will map the measurements, using a linear combination of the dictionary's functions, to the control action. This definition needs to be made carefully since the class of controllers need to contain (or almost contain) the ideal controller.

Since it is expected that $\mathcal{C}_r$ is composed of polynomials in $z(t)$, a polynomial dictionary is created for this case. To give a more parsimonious controller, the polynomial dictionary of (57) is used, since $n_z = 1$, the controller is defined, after some experimentation, as

$$\mathcal{C}[\boldsymbol{q}_4^0 z(t), u(t)\boldsymbol{q}_3; \boldsymbol{\theta}] = P_s(\boldsymbol{q}_4^0 z(t); \boldsymbol{\theta}_z) + P_s(u(t)\boldsymbol{q}_3; \boldsymbol{\theta}_u), \tag{129}$$

with parameter vector $\boldsymbol{\theta} = [\boldsymbol{\theta}_z^T \ \boldsymbol{\theta}_u^T]^T$ and total degree $m = 5$. The functions of this dictionary $\boldsymbol{\psi}(t) = \begin{bmatrix}\psi_1 & \psi_2 & \cdots & \psi_{38}\end{bmatrix}^T$ are shown in Table 2.

Table 2 – Nonlinear Functions of the dictionary

Functions	
$\psi_1 = z(t)$	$\psi_{20} = \psi_1\psi_3\psi_5$
$\psi_2 = z(t-1)$	$\psi_{21} = \psi_1\psi_4\psi_5$
$\psi_3 = z(t-2)$	$\psi_{22} = \psi_2\psi_3\psi_4$
$\psi_4 = z(t-3)$	$\psi_{23} = \psi_2\psi_3\psi_5$
$\psi_5 = z(t-4)$	$\psi_{24} = \psi_2\psi_4\psi_5$
$\psi_6 = \psi_1\psi_2$	$\psi_{25} = \psi_3\psi_4\psi_5$
$\psi_7 = \psi_1\psi_3$	$\psi_{26} = \psi_1\psi_2\psi_3\psi_4$
$\psi_8 = \psi_1\psi_4$	$\psi_{27} = \psi_1\psi_2\psi_3\psi_5$
$\psi_9 = \psi_1\psi_5$	$\psi_{28} = \psi_1\psi_2\psi_4\psi_5$
$\psi_{10} = \psi_2\psi_3$	$\psi_{29} = \psi_1\psi_3\psi_4\psi_5$
$\psi_{11} = \psi_2\psi_4$	$\psi_{30} = \psi_2\psi_3\psi_4\psi_5$
$\psi_{12} = \psi_2\psi_5$	$\psi_{31} = \psi_1\psi_2\psi_3\psi_4\psi_5$
$\psi_{13} = \psi_3\psi_4$	$\psi_{32} = u(t-1)$
$\psi_{14} = \psi_3\psi_5$	$\psi_{33} = u(t-2)$
$\psi_{15} = \psi_4\psi_5$	$\psi_{34} = u(t-3)$
$\psi_{16} = \psi_1\psi_2\psi_3$	$\psi_{35} = u(t-1)u(t-2)$
$\psi_{17} = \psi_1\psi_2\psi_4$	$\psi_{36} = u(t-1)u(t-3)$
$\psi_{18} = \psi_1\psi_2\psi_5$	$\psi_{37} = u(t-2)u(t-3)$
$\psi_{19} = \psi_1\psi_3\psi_4$	$\psi_{38} = u(t-1)u(t-2)u(t-3)$

Since this parametrization is linear in the parameters, the minimum of $J^V(\boldsymbol{\theta})$ has a closed form. Making the regression matrix as

$$\Psi = \begin{bmatrix} \boldsymbol{\psi}(0) & \boldsymbol{\psi}(1) & \cdots & \boldsymbol{\psi}(N-4) \end{bmatrix}^T, \tag{130}$$

the minimum of $J^V(\boldsymbol{\theta})$ is at $\boldsymbol{\theta}_0$

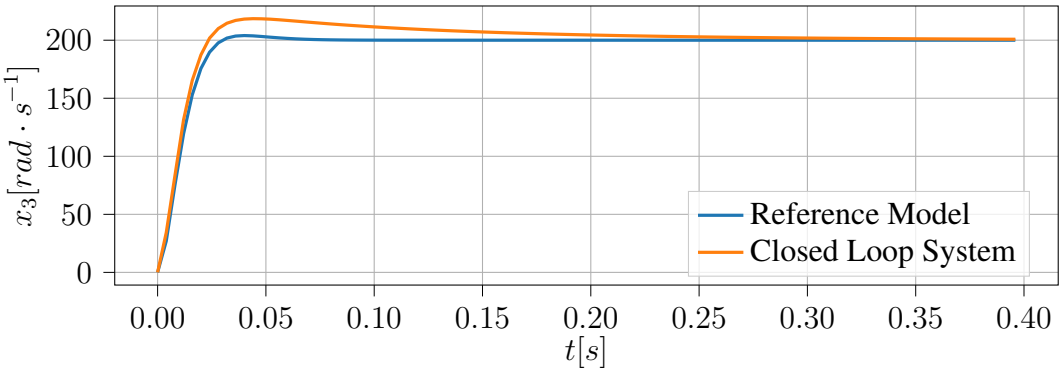
$$\boldsymbol{\theta}_0 = \left(\Psi^T\Psi\right)^{-1}\Psi^T u(t) \tag{131}$$

which gives $J^V(\boldsymbol{\theta}_0) = 0.37$.

4.2.1.5 Analysis

To evaluate the closed loop performance, the controller of (129) is used to calculate the control action at every $\Delta t = 4 \times 10^{-3}\ s$ and applied in the continuous system in a ZOH fashion. This is done in a simulated environment similar to the pendulum case, but here, the dynamics of (126) are used instead. The result of this simulation when a step reference of $r = 200\ rad$ is shown in Figure 33.

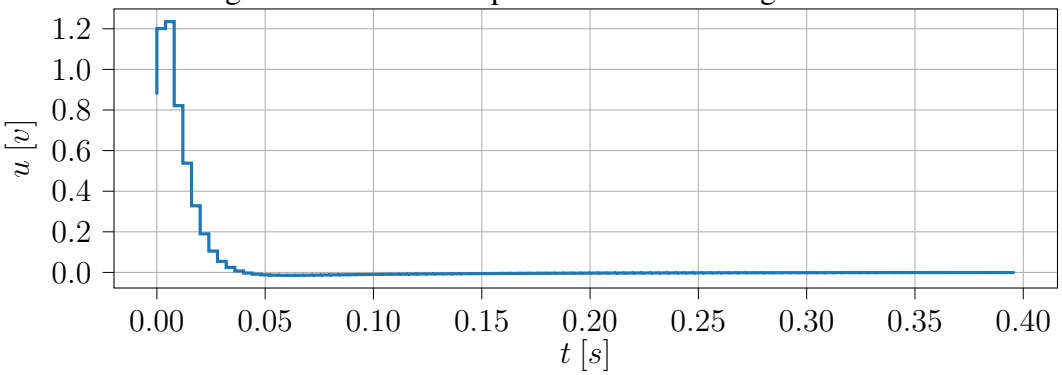
Figure 33 – Comparison of closed loop system with the Reference Model to a step response.



Source: author

Figure 33 shows that the closed loop system follows the reference model's output reasonably, with a bigger overshoot and settling time than given by it. The model reference cost $J(\boldsymbol{\theta})$ in Figure 33 evaluates to $J(\boldsymbol{\theta}) = 60.68$, measured on all $N = 100$ samples of the Figure. The control action for this step response is shown in Figure 34.

Figure 34 – Closed loop control action of Figure 33.

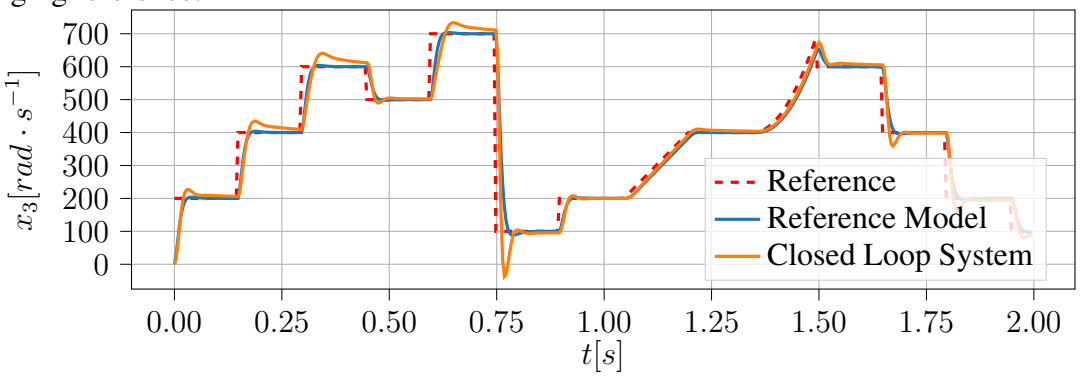


Source: author

Figure 34 shows that the control applied to the plant in closed loop changes considerably in each sample, indicating that, perhaps, a smaller sampling time would be of interest. The control signal rapidly approaches a value close to zero, as required for zero tracking error of a constant reference, although the control does not cross to the negative side and seems not to be fast enough to prevent the overshoot observed in Figure 33.

In order to show the capabilities of the found controller, Figure 35 shows the closed loop system and reference model output to a more elaborate reference signal.

Figure 35 – Comparison of closed loop system with the reference model for a more challenging reference.



Source: author

Figure 35 shows that the response is somewhat similar for almost all steps, indicating that the found controller is able to cancel out most of the nonlinearity of the plant while making the closed loop system follow the reference model closely in all operation points.

The main divergence of the closed loop system and reference model in Figure 35 happens at $t = 0.75\ s$, where in the presence of a big negative step the system overshoots considerably. This overshooting in decreasing steps pattern also appears in the other such steps of the Figure, but in a smaller degree.

Since the parametrization is linear, it is possible and also relevant to look at the energy of each one of the thirty-eight terms of the controller. The energy is defined as

$$E_{u_m} = \sum_{m=1}^{N} [\theta_m \psi_m(t)]^2. \tag{132}$$

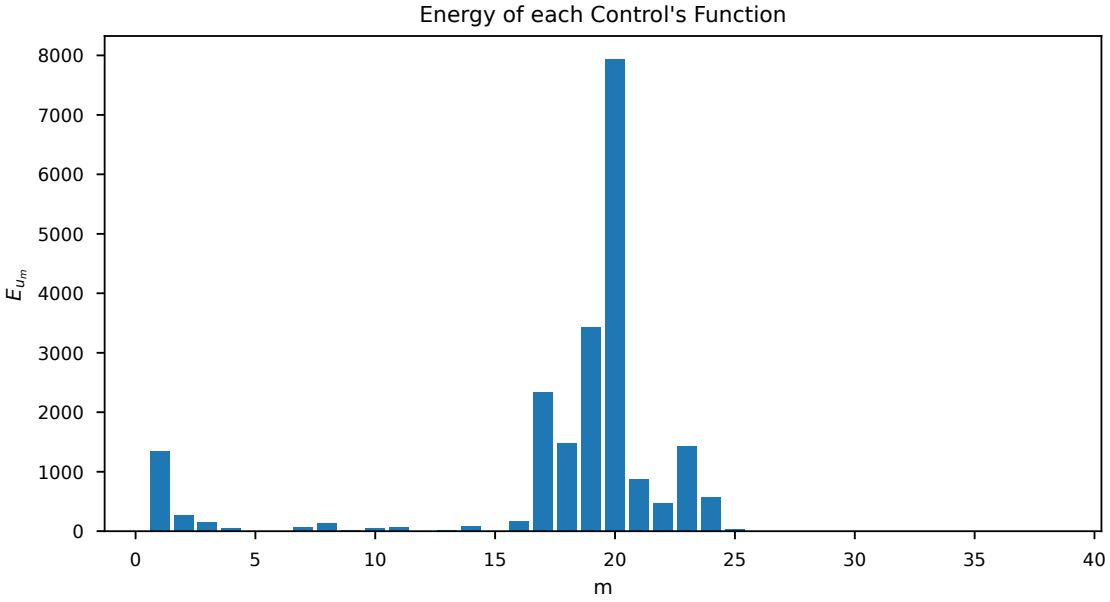
The higher the energy of a given term, the more it is contributing to the control. Figure 36 shows the energy of each term for a step response.

Figure 36 shows that the most relevant terms are the ones for $m \approx 20$, which are polynomials of order three in the measurement. On the other hand, the linear terms of the controller (the first ones) contribute much less to the control, which indicates that the controller is nonlinear.

4.2.1.6 Noise and L^1 regularization

It is also possible to solve the same problem using the least absolute shrinkage and selection operator (LASSO) cost function of (111), thus creating a regularized version of $J^V(\boldsymbol{\theta})$, $J^{VL}(\boldsymbol{\theta})$. Choosing an adequate α_r is highly problem dependent, generally speaking, systems that need higher control signals for its operation will also need a higher α_r to meaningfully penalize the parameters, while systems that need control signals closer to zero will have to use a smaller penalization. One option to automatically choose the penalization is to divide the IO-data into testing and validation sets, minimize $J^{VL}(\boldsymbol{\theta})$

Figure 36 – Energy of each control signal.



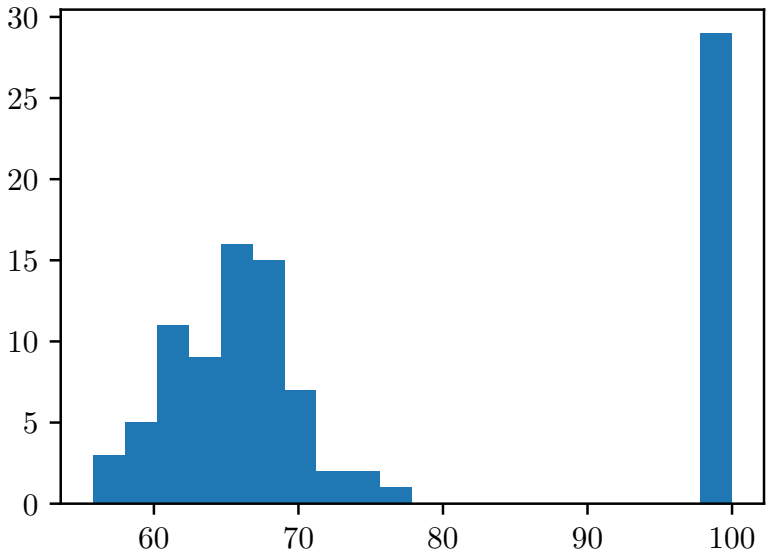
Source: author

using the data in the validation set alone and then measure $J^V(\boldsymbol{\theta})$ on the test set, one can than gradually increase α_r up to a point where the cost on the validation and test sets are close. This approach gives something close to a minimum penalization that gives the set of $\boldsymbol{\theta}$ that makes the predictor $\mathcal{C}(\boldsymbol{\theta})$ generalize well into the validation set. Here, however, the parameter was chosen empirically, although using an algorithm close to the one described above, to $\alpha_r = 10$.

The controllers found using LASSO are similar to the ones found without it, the main difference is on the robustness of the method, i.e., the probability of finding a set of parameters that generate a stable closed loop is higher using LASSO than not. To show this point, 100 Monte Carlo simulations are run for the system (126) with $\sigma = 73.5$, the generated data is then used to find 100 $\boldsymbol{\theta}$ using LASSO and 100 $\boldsymbol{\theta}$ not using it.

The generated controllers are then put in closed loop with the system and a step of $400rad \cdot s^{-1}$ is applied to it, the model reference cost function $J(\boldsymbol{\theta})$ for the controllers are shown in the histogram of Figure 37 for the controllers found using LASSO.

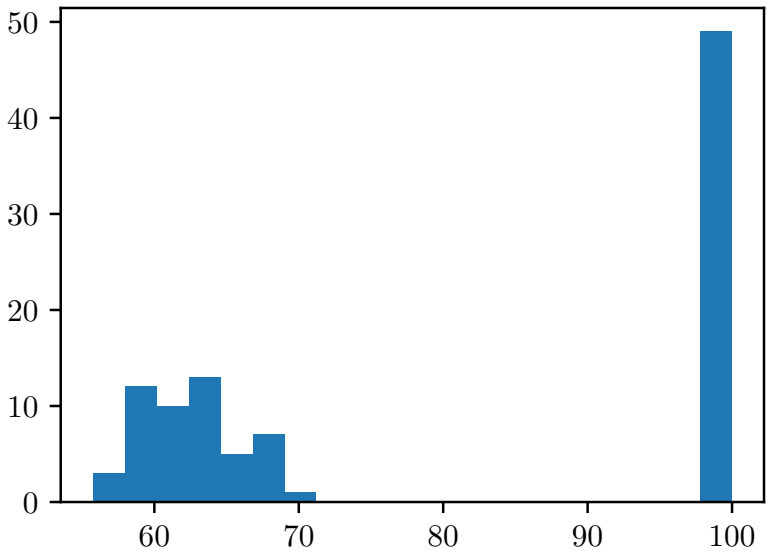
Figure 37 – Model Reference Cost histogram with LASSO.



Source: author

The controllers have a $J(\boldsymbol{\theta})$ distribution close to normal and are centered around a cost of 65. The cost functions are all clipped to have a maximum of 100, all those who have a model reference cost function clipped to 100 are unstable, with LASSO they amount to 27 out of 100. This seems bad but it is important to consider that the signal-to-noise ratio in this case equals to five. Figure 38 shows the results for the problem solved without using LASSO.

Figure 38 – Model Reference Cost histogram without LASSO.



Source: author

Without LASSO the model reference cost functions of Figure 38 are, for the stable systems, lower on average. But on the negative side, almost half of the closed loop systems are unstable without LASSO.

These results show that on using regularization, one trades performance for robustness of the method, which is somewhat expected. Lossing performance is reasonable since now the cost function is not just minimizing J^V while gaining robustness is expected because the model is less prone to over-fit.

4.2.2 Case 2: DNN controller

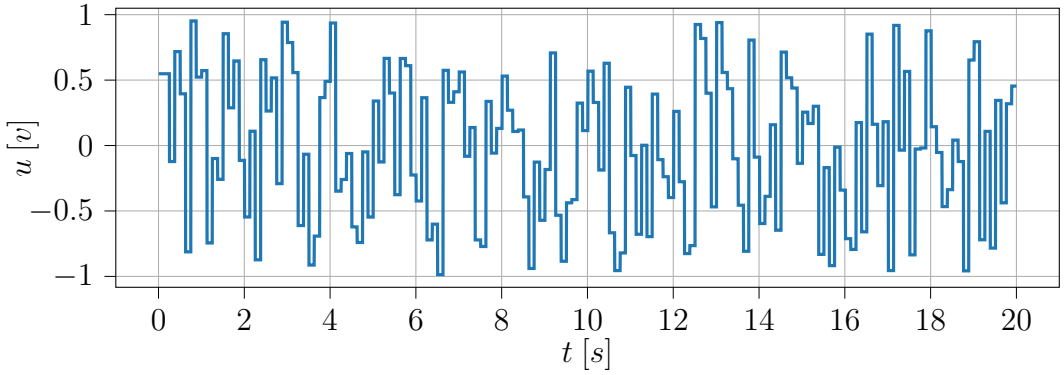
4.2.2.1 Reference Model

The same Reference Model as in the case of subsection 4.2.1 is used in this case to allow a better and easy comparison of the controller's performance.

4.2.2.2 Experiment

The input signal used to generate the IO-data is a variable amplitude PRBS, i.e., the $s_s(t)$ of (118) with $n_A = 1600$, $T_i = 0.1\ s$ and $\alpha_i \sim U(-1, 1)$, signal shown, partially, in Figure 39.

Figure 39 – Part of the sequence of steps used to excite the system.

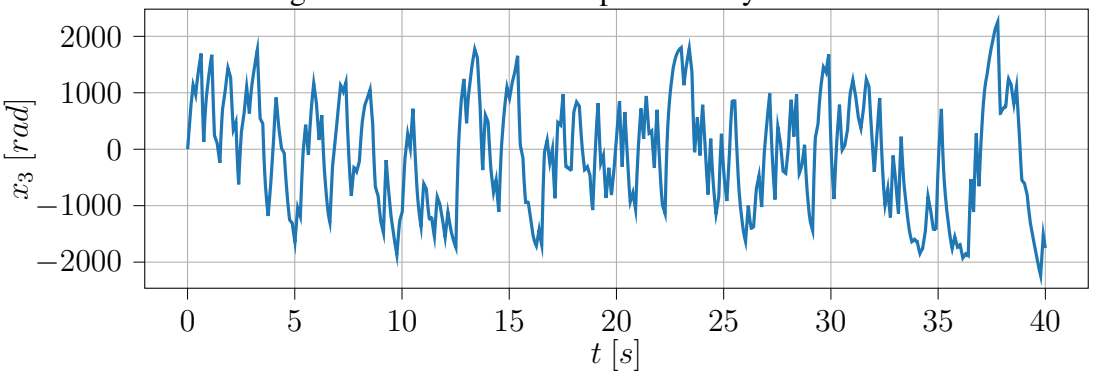


Source: author

The control signal of Figure 39 shows a control signal that explore all control levels in the desired range, with step amplitudes of various levels, which is desirable to explore the system's dynamics.

To obtain the IO-data the system of equation (126) is simulated with the signal of Figure 39 for $T = 200\ s$. The data is collected with a sampling time of $\Delta t = 4 \times 10^{-3}$. With the system output noise contamination, the signal-to-noise ratio is approximately 10^3. The noisy collected output is shown, partially, in Figure 40.

Figure 40 – Part of the output of the system.



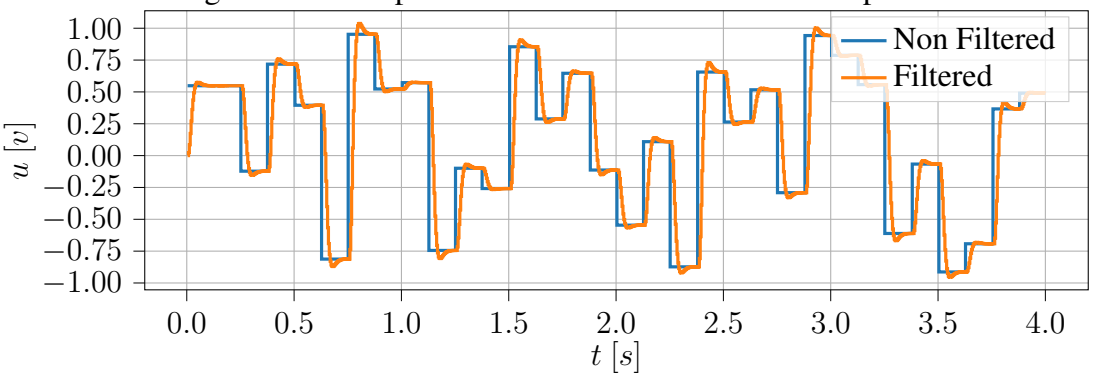
Source: author

The system output, shown in Figure 40 shows an output that explores all the output range of interest with a rate of change that seems to be at least close to what would be required of the system to operate as the reference model.

4.2.2.3 Filtered Signals and Final Dataset

As in the pendulum case, the filter that will be utilized is the one of (39), a comparison between the input $u(t)$ and the filtered input $u_L(t)$ is shown in Figure 41 and between the error $e(t)$ and the filtered error $e_L(t)$ is shown in Figure 42.

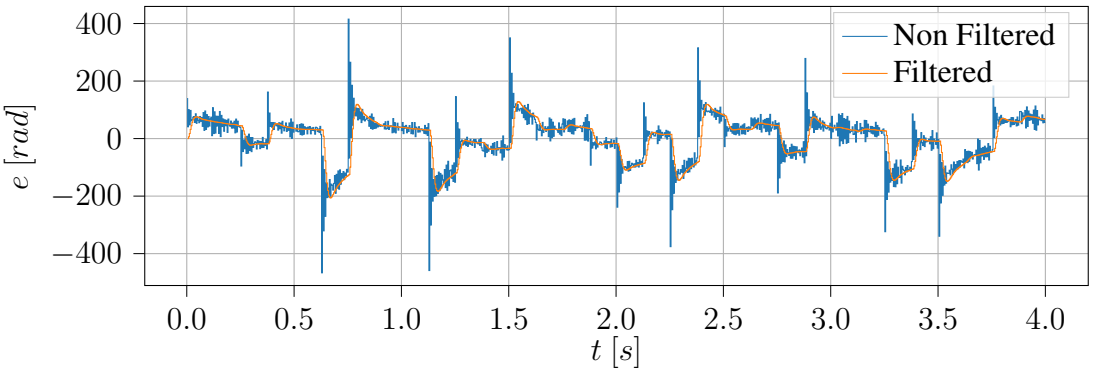
Figure 41 – Comparison of filtered and non filtered input.



Source: author

The filtered control, shown in Figure 41 is close to the unfiltered control, the main difference is a smoothing of the steps corners, which seems to be closer to what would be required of the system in closed loop to follow the reference model, a similar pattern is shown in Figure 42 for the virtual error.

Figure 42 – Comparison of Filtered and Non filtered Error.



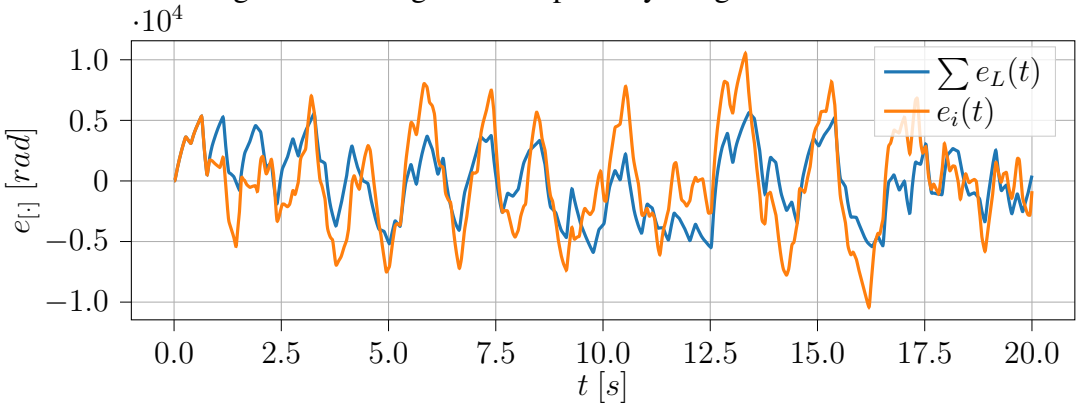
Source: author

Differently from the pendulum case, the filtered and non filtered signals are closer and the filter has almost no effect in the lower frequency range. The filtered signals are used, with the measurement map $\mathcal{Z}$, to construct the database to train $\mathcal{C}(\boldsymbol{\theta})$. The measurement map, once again, is selected in a way that eases the problem of finding $\mathcal{C}(\boldsymbol{\theta})$, for that, in this case, a partial integrator is used for the error. The integrator is deemed partial because it only accumulates the error over the past 200 samples, i.e., the partially integrated signal $e_i(t)$ is the rolling sum

$$e_i(t) = \begin{cases} \sum_{\tau=0}^{\tau=t} e_L(\tau) \ \ \forall t \leq 200; \\ \sum_{\tau=t-200}^{\tau=t} e_L(\tau) \ \ \forall t > 200. \end{cases} \tag{133}$$

The integrated error and the partially integrated error are shown in Figure 43.

Figure 43 – Integrated and partially integrated error.

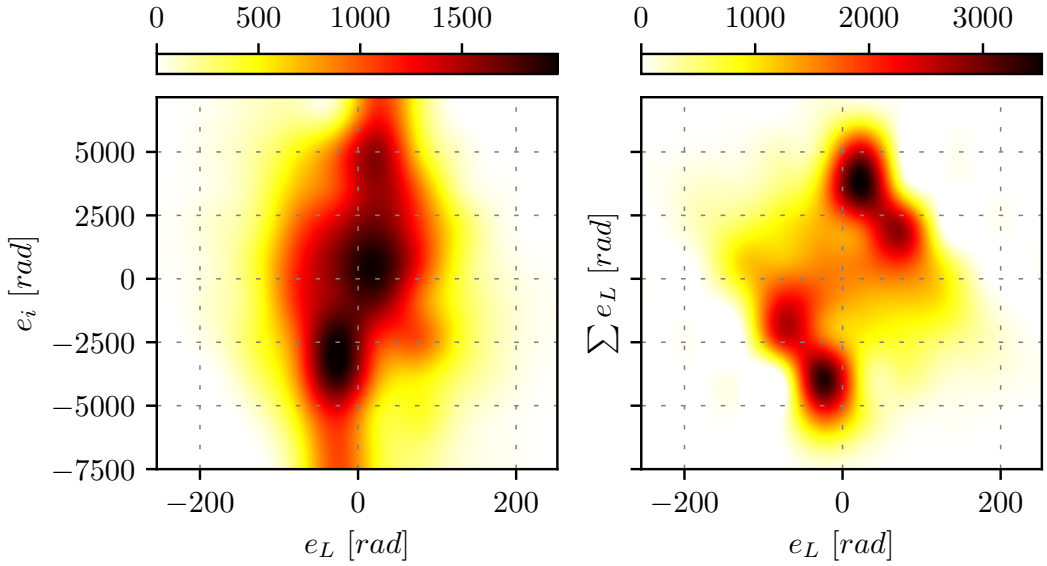


Source: author

Figure 43 shows that the $e_i(t)$ has a smaller mean and a higher variance than $\sum e_L(t)$, but that is not the important factor, the real benefit of using the partially integrated error is

more clearly seen when looking at the data distribution on the $e_L(t) - e_i(t)$ plane, shown in Figure 44.

Figure 44 – Integrated and partially integrated error distribution over e_L .



Source: author

On comparing the integrated error distribution with the partially integrated error distribution, shown both in Figure 44, it is clear that e_i is more centered in $e_L = 0$ and better distributed in its range than the integrated error. The first feature induces the controller to use the partially integrated error only in regions of low error, which is desirable since this signal function is to, mostly, provide zero steady state error. The second feature, on the other hand, gives a better chance of obtaining a predictor that is more easily generalizable.

Thus, using (133), the measurement map is defined as

$$\mathcal{Z}[r(t)\boldsymbol{q}_t^{t-200}, e_L(t)\boldsymbol{q}_t^{t-200}, y(t)\boldsymbol{q}_t^{t-200}] = \begin{bmatrix} y(t) \\ e_L(t) \\ e_i(t) \end{bmatrix}. \tag{134}$$

Using the measurement map (134) is possible to create the database to fit $\mathcal{C}(\boldsymbol{\theta})$. The database input is defined as

$$X = \begin{bmatrix} u_L(t_1) & y(t_0) & e_L(t_0) & e_i(t_0) \\ u_L(t_2) & y(t_1) & e_L(t_1) & e_i(t_1) \\ \vdots & \vdots & \vdots & \vdots \\ u_L(t_N - 1) & y(t_N) & e_L(t_N) & e_i(t_N) \end{bmatrix}; \tag{135}$$

and the output as

$$Y = \begin{bmatrix} u_L(t_0) & u_L(t_1) & \cdots & u_L(t_N) \end{bmatrix}^T. \tag{136}$$

The scaled versions of the input and output databases are created using the same min-max scaler of (123) utilized in the pendulum using the same range, i.e., $a = -1$ and $b = 1$.

4.2.2.4 DNN-Controller

The chosen architecture is composed of an input layer of 4 neurons, a Gaussian Noise layer with $\sigma = 0.1$, 4 GRU layers, each with 64 cells, and a dense output layer with linear activation function and 10 neurons. This architecture gives a model with 88970 trainable parameters. It should be noted that this architecture was chosen after some experimentation, e.g., the same architecture as described above, but with 16 cells instead of 64, was trained. The smaller simpler architecture has only about 2000 parameters and only a slightly worse result than the one presented here, the only reason why this is the chosen architecture to show is to guarantee that any possible poor performance is not due to the lack of the model's expressive power.

The GRU layers have a hyperbolic tangent for the output activation function and a sigmoid for the states. Also, only L_1 recurrent regularization is applied, with $\alpha_{rr} = 0.1$. The model is of type sequence-to-sequence and all the other parameters are equal to the pendulum case. With this, the control at time t is given by

$$u(t) = \mathcal{C}[\boldsymbol{q}_{19}^0 \boldsymbol{z}(t), u(t)\boldsymbol{q}_{20}]. \tag{137}$$

Having 10 output neurons means that at time t_i, the DNN predicts the next 10 control samples, $\hat{u}(t_{i+1}),\ \hat{u}(t_{i+2}),\ \cdots\ , \hat{u}(t_{i+10})$, although in application only the first one, $\hat{u}(t_{i+1})$, is utilized. This works as a form of regularization that promotes control predictions which do not change so abruptly and prediction errors that are more averaged than would be if only the next control sample was predicted. This is a common practice with time-series predictions using DNNs (GOODFELLOW; BENGIO; COURVILLE, 2016; GÉRON, 2022). To be precise, consider that the DNN map $\mathcal{D}_{\mathcal{NN}}$ is defined as

$$\hat{\boldsymbol{u}}(t) = \mathcal{D}_{\mathcal{NN}}[\boldsymbol{q}_{19}^0 \boldsymbol{z}(t), u(t)\boldsymbol{q}_{20}], \tag{138}$$

with $\hat{\boldsymbol{u}}(t) = [\hat{u}(t),\ \hat{u}(t+1),\ \cdots\ , \hat{u}(t+10)]^T$. The control at time t of (137) is then

$$u(t) = \hat{u}(t). \tag{139}$$

To attenuate this averaging effect, the true and predicted controls are linearly decreased, from 100% in t_{i+1} to 90% in t_{i+10}. Since it is a sequence-to-sequence model, at each of the twenty input time steps, the next ten samples are predicted. Thus, the linear decrease is made in each of these predictions.

A particularity of the DC-Motor is that in steady-state, since the dynamics assumes that the motor runs without load, the control input only needs to compensate the friction, which, for this system, is very small. Thus, to achieve good tracking error the predictions

need to be accurate in this particular region of almost zero control input. As discussed, this is somewhat problematic for the Mean Square Error loss function as it loses sensibility around zero. For this reason, the J_{mcse} cost function is used, with $\alpha_c = 0.8$.

As in the pendulum case, training is made using a modified loss function that also slightly penalizes the predicted control, but in this case, since the next ten samples are predicted, the penalization is applied in the decreased predictions

$$\hat{\boldsymbol{u}}_d(t) = \begin{bmatrix} \hat{u}_d(t) & \hat{u}_d(t+1) & \cdots & \hat{u}_d(t+N_u) \end{bmatrix}^T,$$

with this, the modified training cost function is

$$J^{Vm}(\boldsymbol{\theta}) = J^V_{mcse}(\boldsymbol{\theta}) + \alpha_u \sum_{t=0}^{N} \hat{\boldsymbol{u}}_d(t)^T \hat{\boldsymbol{u}}_d(t), \tag{140}$$

where, for this problem, the penalization factor $\alpha_u = 1 \times 10^{-3}$.

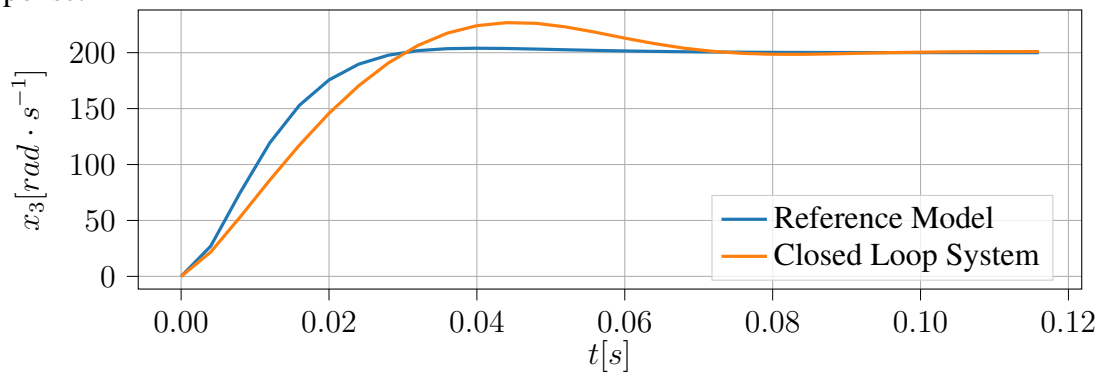
As in the pendulum case, the selected optimizer is the Nadam, with initial learning rate of 1×10^{-4}. The plateau learning rate reducer is utilized in training, it is configured to shrink the learning rate by 10% after 5 epochs with no improvement in the validation set, up to a minimum learning rate of 1×10^{-6}. The model is trained up to the point where 20 epochs have passed with no improvement on the validation set.

The final training error, measured on the unscaled signals for a better understanding, is of $J^V_{train}(\boldsymbol{\theta}) = 1.38 \times 10^{-4}$ with a test set error of $J^V_{test}(\boldsymbol{\theta}) = 9.92 \times 10^{-5}$, showing good generalization of the model in the test set.

4.2.2.5 Analysis

To evaluate the trained controller performance, the control is calculated using the found $\mathcal{C}(\boldsymbol{\theta})$ in simulation, which is performed using the dynamics of (126), where the control action is calculated at every $\Delta t = 4 \times 10^{-3}\ s$ and applied in the continuous system in a ZOH fashion, using the same setup as in the pendulum case. The result of this simulation when a step reference of $r = 200\ rad$ is applied into the system is shown in Figure 45.

Figure 45 – Comparison of closed loop system with the Reference Model to a step response.

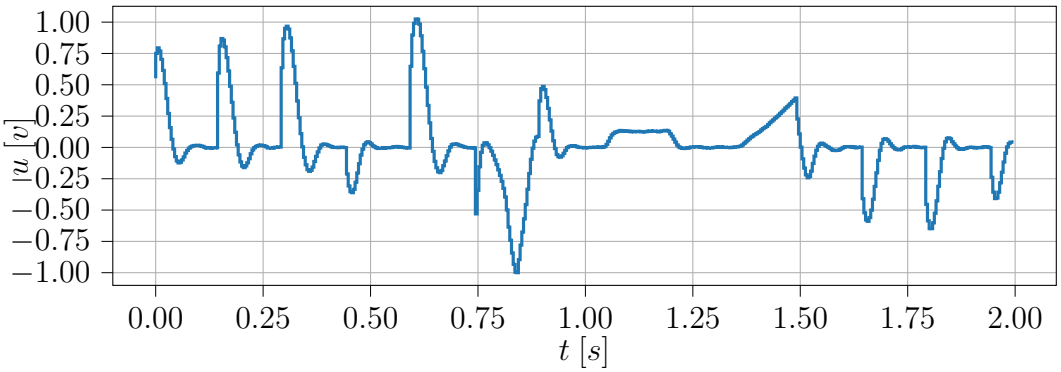


Source: author

Figure 45 shows that the closed loop system follows the Reference Model output, although with a significantly bigger overshoot and settling time. Its step response has a model reference error of $J(\boldsymbol{\theta}) = 228.85$, measured on all the $N = 30$ samples of the figure.

The control action, generated with $\mathcal{C}$ for this case is shown in Figure 46.

Figure 46 – Closed loop control action of Figure 45.



Source: author

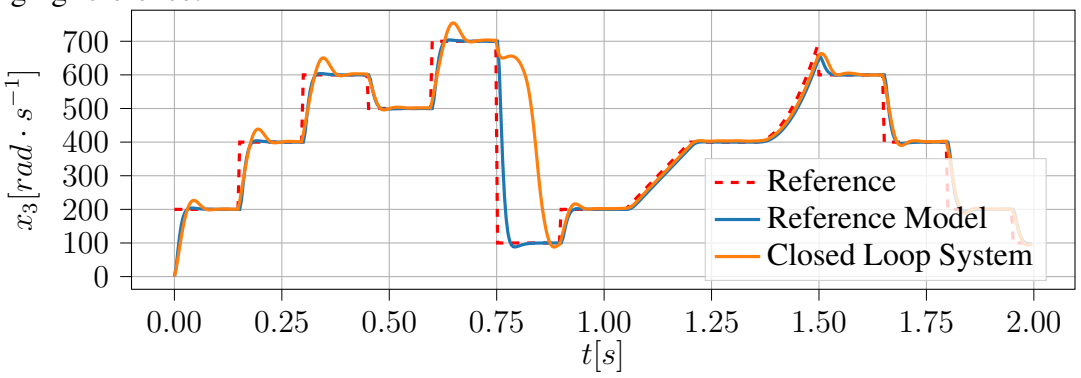
The control signal shown in Figure 46 shows a control signal that seems to be slow and that changes significantly in each sampling time, there is also a significant cross to the negative control which, together with its slowness, seem to be the cause of the observed overshoot.

Unlike in the pendulum case, the controller herein found does not seem to leave any important feature of the system's dynamics out, its only defect is its slowness, which seems to be the cause of the observed overshoot, as seen in Figures 45 and 46.

Evaluating the closed loop prediction error, i.e., J^V_{closed}, for the step of Figure 45 tells a misleading story, the reason is that much of the predicted control is very close to zero (when not exactly on it), which causes the low sensitivity of J^V for this trajectory. To

present this metric in a more meaningful setup and to show the controller's capability, the system is simulated in a more challenging reference. Figure 47 shows the system and reference model response to a sequence of varying amplitude steps, ramps and second order reference.

Figure 47 – Comparison of closed loop system with the Reference Model to a more challenging reference.

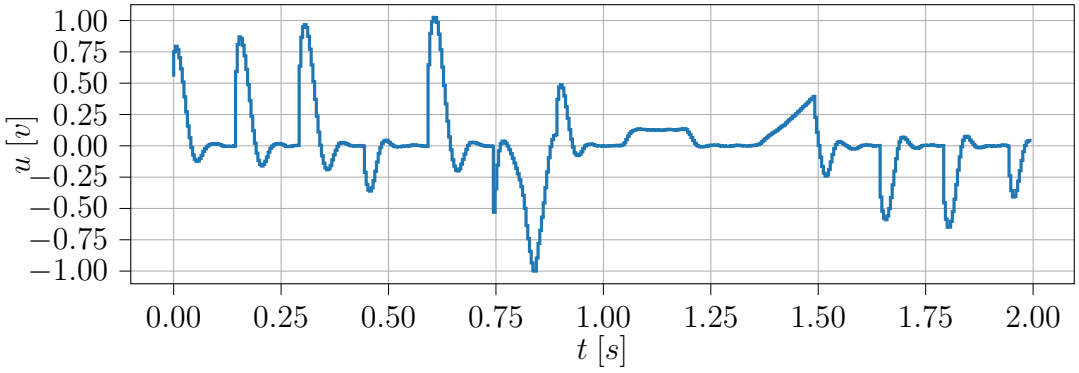


Source: author

As shown in Figure 47, the closed loop system response is similar for small steps of same amplitude and different operating points, the main difference being the size of the overshoot, indicating that the found controller $\mathcal{C}$ does not fully cancel out the nonlinearity of the system. This effect is even clearer at time $t = 0.75\ s$, where in the presence of a bigger step size, the closed loop system output is nowhere close the reference model's output, although it still is able to recover and follow the step at its end. Another noticeable effect is that steps that go from a higher to lower operating points generates outputs from the closed loop system that are much closer to the reference model, this effect is likely due to the bias on the J_{msle} cost function.

Figure 48 shows the control signal of the closed loop system of Figure 47. It is possible to see that, at $t = 0.75\ s$, the controller predicted a control that went to about $-0.5\ v$ then, instead of continuing to decrease, it went back up to the zero region. The control action then decreased, in a much less abrupt way, to about $-1\ v$ at the end of the step, indicating that the reference was only followed due to the integrator property of the found controller.

Figure 48 – Control of the closed loop system of Figure 47.

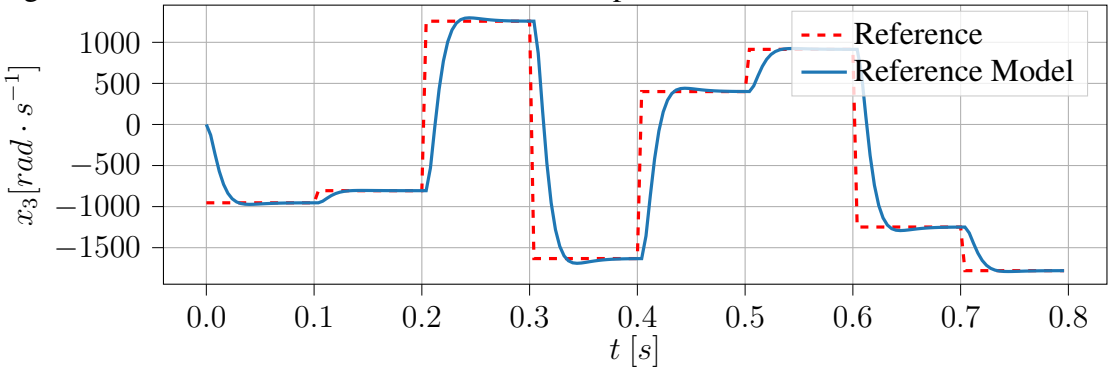


Source: author

To calculate the closed loop estimation error, the virtual error is calculated for the trajectory of Figure 47, and the predictor $\mathcal{C}$ is inputted with this signal instead of the actual closed loop error, as in the pendulum case, this allows to calculate the prediction error in this specific trajectory. Also, the error is better represented in this trajectory, where the control signal average is not so close to zero as in the single step of Figure 45. The prediction error in this trajectory is of $J^V_{close} = 6.49 \times 10^{-3}$, a percent error of 6440% when compared with J^V_{test}.

This result seems to indicate that the experimental dataset is not fully representative of the system's dynamics and that a better experiment could be performed to explore it. To show the difference between the ideal and experimental data distributions, the ideal dataset is created using a sequence of steps $s_s(t)$ with $n_A = 2000$, $T_i = 0.1\ s$ and $\alpha_i \sim U(-2000, 2000)$, the reference and the reference model output are shown, up to $t \approx\ 0.8s$, in Figure 49.

Figure 49 – Reference and Reference Model output for the creation of the ideal dataset.

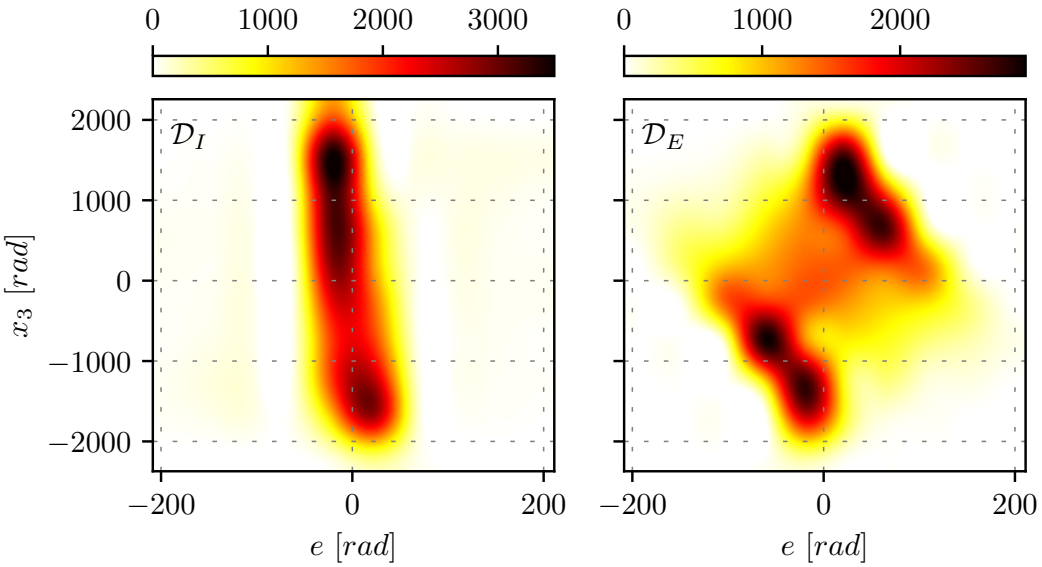


Source: author

The reference and reference model signals from Figure 49 are created to have the exact same number of samples as the experimental dataset, which allows a more straightforward comparison and to have the same properties as in the ideal dataset creation of the pendulum case, i.e., reasonably slow to allow the reference model to settle.

Figure 50 shows the heatmaps for the ideal $\mathcal{D}_I$ and experimental $\mathcal{D}_E$ datasets. The Figure is created in the same way as in the pendulum case, here, however, the maximum range is from $\mathcal{D}_E$, this way, all the experimental data is in the heatmap while not all the $\mathcal{D}_I$ is in its heatmap. In other words, to better show the patterns in the data, the heatmaps are created using the minimum data range among the two datasets.

Figure 50 – Ideal and Experimental datasets distribution.

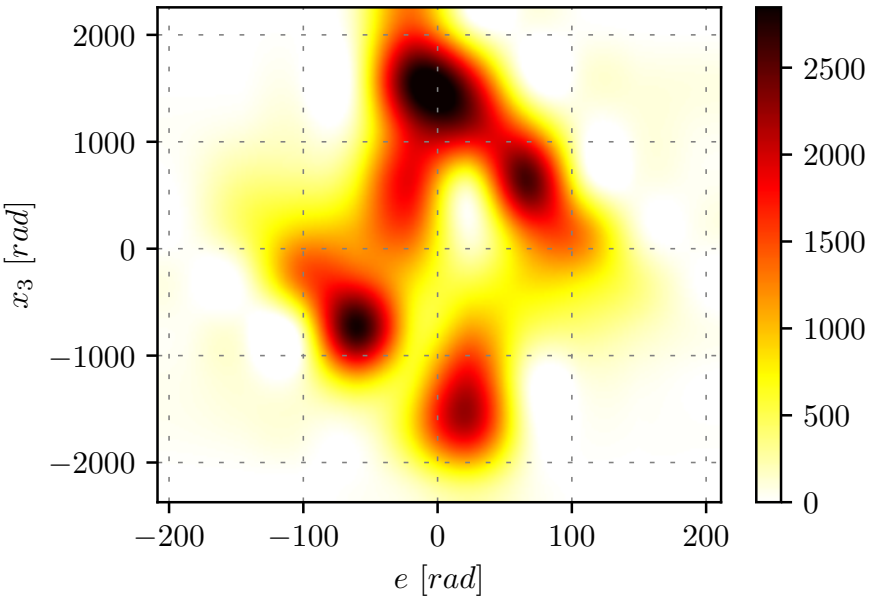


Source: author

As shown in Figure 50, the ideal dataset has its data much more centered around zero error, much like in the pendulum case. Here, however, the ideal dataset has a much larger range than the experimental dataset. The $\mathcal{D}_I$ heatmap shows a slight tendency on the data, where positive errors tend to have negative outputs and negative errors tend to have positive outputs, the exact opposite pattern happens in the $\mathcal{D}_E$ heatmap, indicating that the signal used to excite the system is not ideal.

The experimental dataset also emphasizes regions of greater error magnitude more than the ideal dataset, although $\mathcal{D}_I$ have regions of much greater error, this is so much so that about 17% of the ideal dataset's data is not in the Figure 50. This shows that the experimental dataset does not even have data in the range of the expected closed loop operation. To makes matters worse, where it does have data, the distribution differs from the ideal dataset, this is better shown in the difference heatmap, created just like in the pendulum case and shown, for this case, in Figure 51.

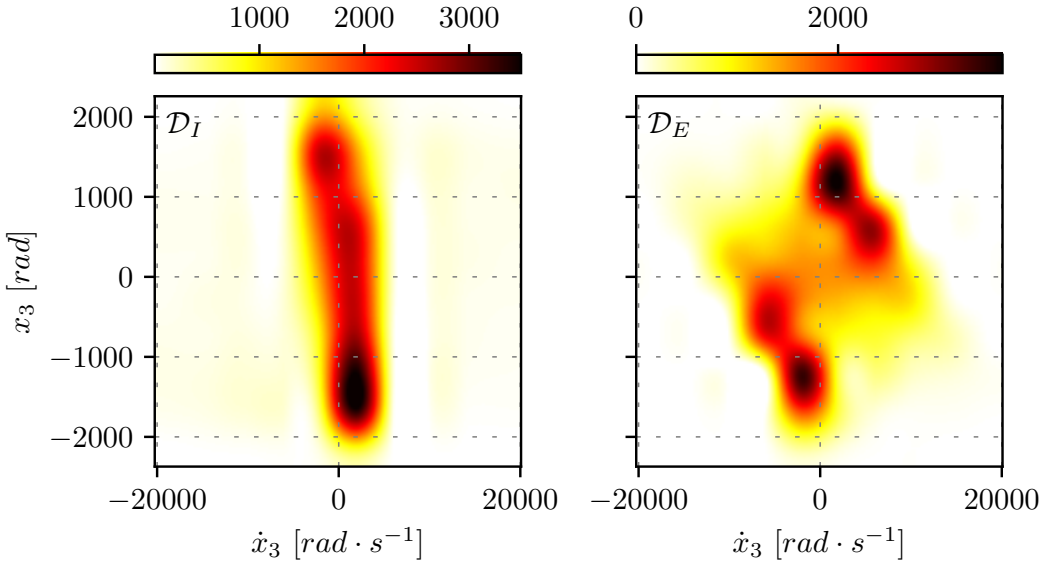
Figure 51 – Difference heatmap between ideal and experimental datasets.



Source: author

The difference heatmap of Figure 51 shows that the differences between $\mathcal{D}_I$ and $\mathcal{D}_E$ are substantial, which is expected since the ideal dataset have data concentrated in regions that the experimental dataset does not even cover, which is the reason why the difference go as high as the maximum samples per region in both datasets. The difference, however, is lower around zero error and output, showing that particularly with low positive errors and outputs the experimental dataset is representative of the region of interest. The same pattern appears in the output velocity datasets comparison, shown in Figure 52.

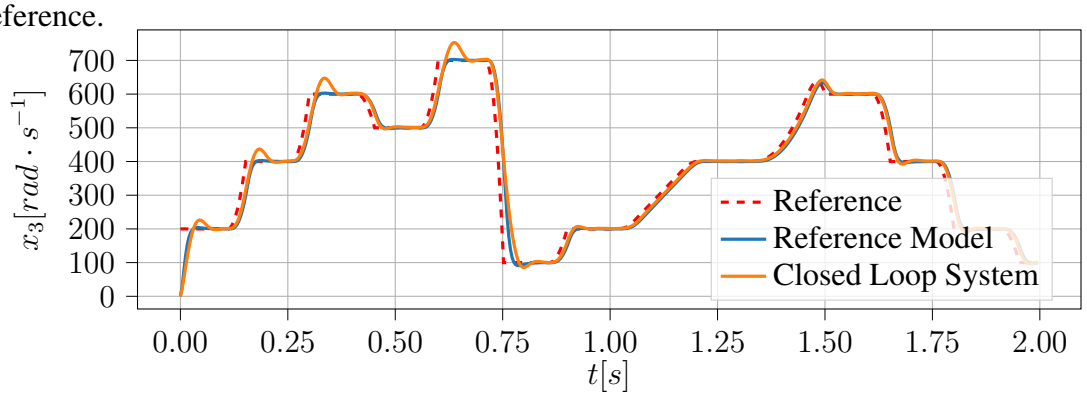
Figure 52 – Ideal and Experimental datasets velocity distribution.



Source: author

This difference along with the $\mathcal{D}_E$ distribution explains why, in Figure 47, the majority of the steps and other kinds of reference are closer to the reference model's output than the big step that go from $700\ rad$ to $100\ rad$: the error in the beginning of the step is of $600\ rad$, which is not even close to what is in the experimental dataset, in fact, the system is only able to recover in this case because of the partially integrated error signal, which happens to be in range[8] and it is the reason why the system takes some time to start properly following the step. To show that this is indeed the case, Figure 53 shows the closed loop response using a smoothed version of the reference in Figure 47.

Figure 53 – Comparison of closed loop system with the Reference Model to the smoothed reference.



Source: author

Since the reference in Figure 53 does not abruptly changes, the error never grows to be bigger than what was present in the experimental dataset and thus the predictor $\mathcal{C}$ does not make errors as big as it does in the unfiltered reference, in fact, $J^V_{close} = 4.30 \times 10^{-4}$, a percent error of 333.42% when compared with J^V_{test}, much smaller than in the unfiltered case. The model reference error measured on all $N = 500$ samples of the filtered input is of $J(\boldsymbol{\theta}) = 162.76$.

This shows that, once again, the main problem with the DNN approach is with its generalization. The generated training data does not seem to explore well the output-error space of interest in application, which ends up requiring more generalization power of the DNN than its available, delivering a model that has poor generalization in the closed loop set.

4.2.3 Case 3: DNN controller with modified Reference Model

4.2.3.1 Reference Model

In this case a different reference model, although with the same control objectives, i.e., a closed loop system with unitary gain and settling time of about $t_s = 4 \times 10^{-2}\ s$, will be used to analyze if the problems encountered in subsection 4.2.2 can be solved by

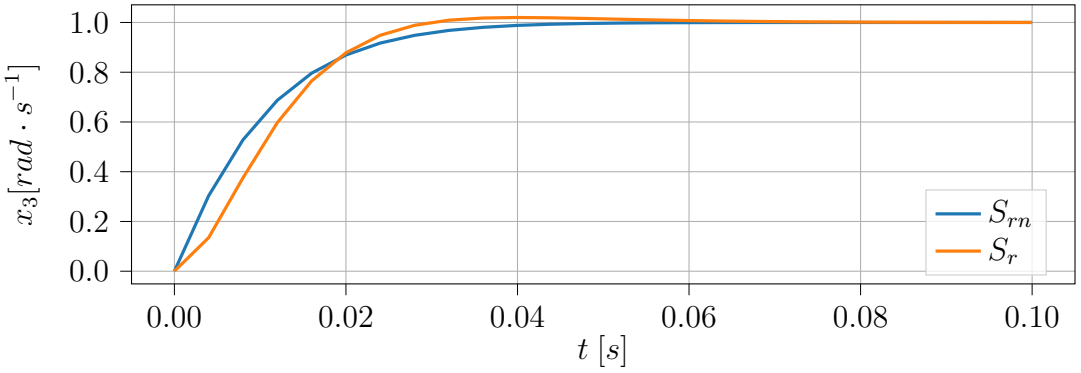
[8] Not shown here.

simply changing the reference model. The continuous time reference model is defined as

$$S_{rn} = \frac{120^3}{100 \cdot 200} \frac{(s+100)(s+200)}{(s+120)^3}, \tag{141}$$

with step response in Figure 54, shown in comparison with the step response of S_r, of (127).

Figure 54 – Step response of (141).



Source: author

As shown in Figure 54, the new reference model S_{rn} has an output that is very close to the S_r's, which accomplishes the goal of satisfying the same control objectives.

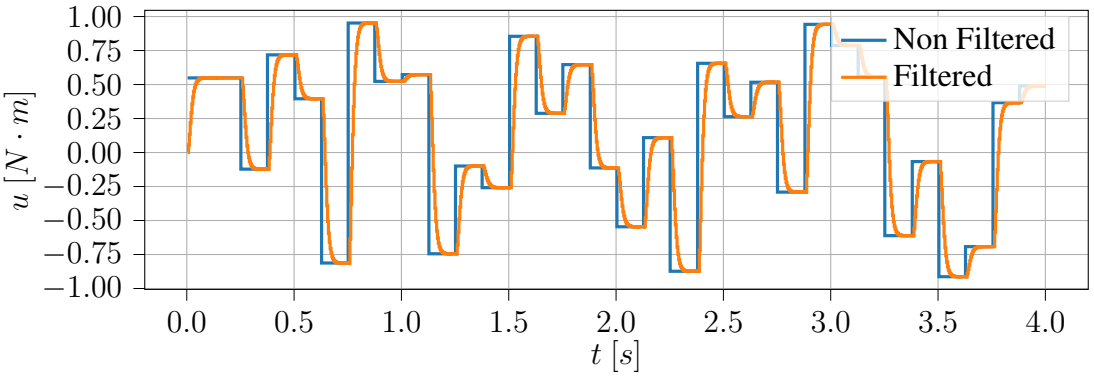
4.2.3.2 Experiment

To show that the differences in the reference model alone, an experiment is performed on the system using the same input as in the last case, i.e., the input shown of Figure 39. The output is, appart from the different realization of noise, the same as in Figure 40 and the data is collected with sampling time of $\Delta t = 4 \times 10^{-3}$.

4.2.3.3 Filtered Signals and Final Dataset

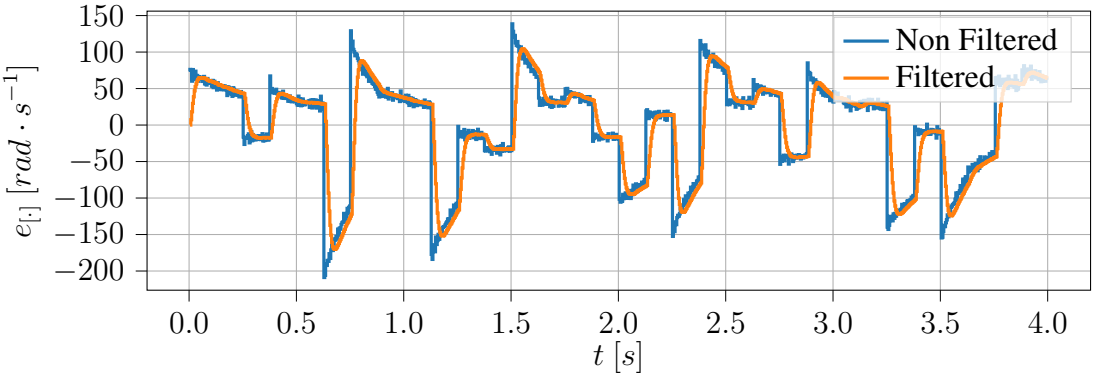
Since the filter of (39) depends on the Reference Model, even with the data input output data being the same, the filtered data will be different. Moreover, the virtual error is different from the start since it depends on the Reference Model as well. A comparison between the input $u(t)$ and the filtered input $u_L(t)$ is shown in Figure 55 and between the error $e(t)$ and the filtered error $e_L(t)$ is shown in Figure 56.

Figure 55 – Comparison of filtered and non filtered input.



Source: author

Figure 56 – Comparison of Filtered Error and Non filtered.



Source: author

Although the control in Figure 55 has no overshoot when compared with the one of 41, the main difference here is on the noise level of the non-filtered error signal, which is much lower in Figure 56 than in Figure 42. In fact, the reference model itself was chosen in part for this specific characteristic.

For consistency, all the other characteristics of the final dataset definition are the same as presented in sub-section 4.2.2.3.

4.2.3.4 DNN-Controller

Again, for consistency, the DNN architecture is the same as presented in sub-section 4.2.2.4.

The final training error, measured on the unscaled signals for a better understanding, is of $J^V_{train}(\boldsymbol{\theta}) = 2.41 \times 10^{-4}$ with a test set error of $J^V_{test}(\boldsymbol{\theta}) = 1.73 \times 10^{-4}$, showing, once again, good generalization of the model in the test set.

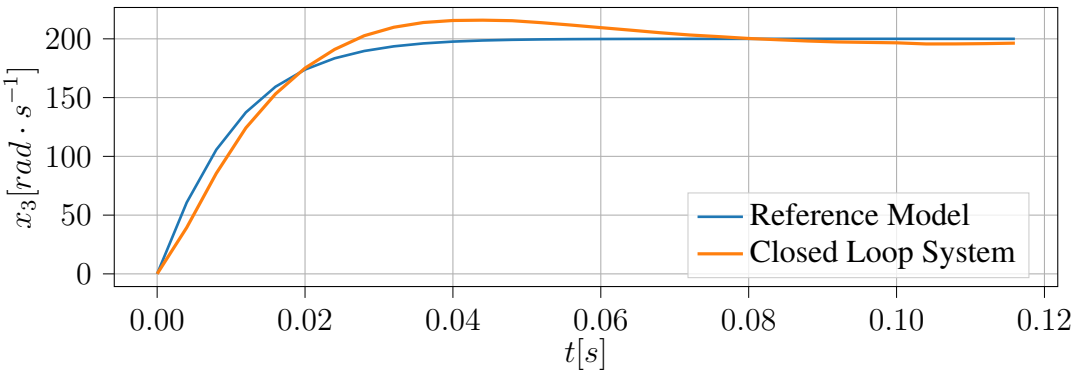
It should be noted that the training error in this case is almost 1.7 times bigger than in the past case. Since the DNN has its weights initialized randomly and the system have noise in its output, that alone could explain the difference. Another possible explanation

is that the virtual error is different from the past case, therefore, the bigger error could indicate a more complex dataset, one that is harder to predict on due to its richer generating dynamics and explored spaces.

4.2.3.5 Analysis

As before, to evaluate the trained controller performance, the control is calculated using the found $\mathcal{C}(\boldsymbol{\theta})$ and a simulation is performed using the dynamics of (126), where the control action is calculated at every $\Delta t = 4 \times 10^{-3}\ s$ and applied in the continuous system in a ZOH fashion. The result of this simulation, when a step reference of $r = 200\ rad$ is applied onto the system, is shown in Figure 57.

Figure 57 – Comparison of closed loop system with the Reference Model to a step response.

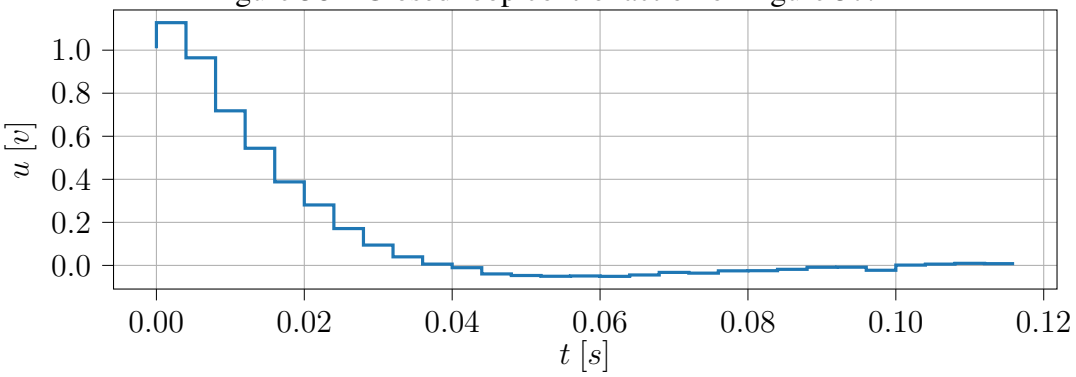


Source: author

The step response in Figure 57 has a model reference error of $J(\boldsymbol{\theta}) = 113.35$, measured on all the $N = 30$ samples of the figure. The closed loop system follows the reference model with some overshoot but this time its smaller than in the last case when compared to the reference model, which is confirmed by the reference model error.

The control action, generated with $\mathcal{C}$ for this case is shown in Figure 58.

Figure 58 – Closed loop control action of Figure 57.

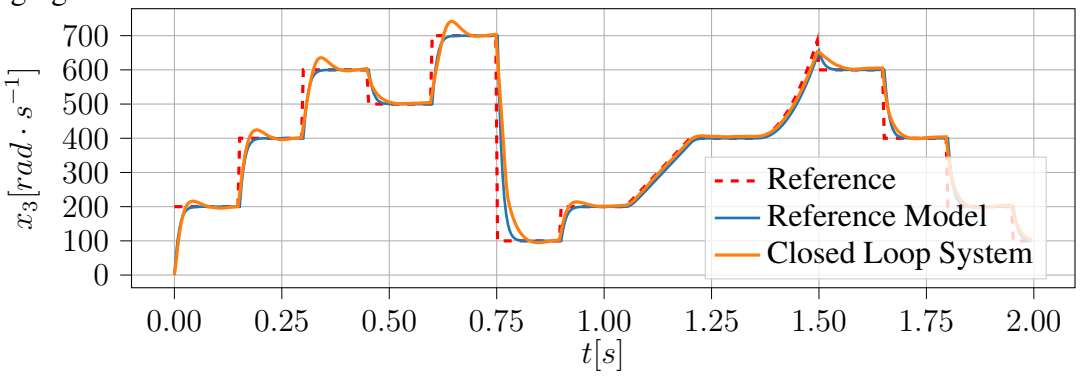


Source: author

Both output and input look almost the same as in the Figures 45 and 46, but even small, the differences are enough to give a much better response, as indicated by the model reference errors of the two cases.

Figure 59 shows the system and reference model response to the same reference as the one of Figure 47.

Figure 59 – Comparison of closed loop system with the Reference Model to a more challenging reference.



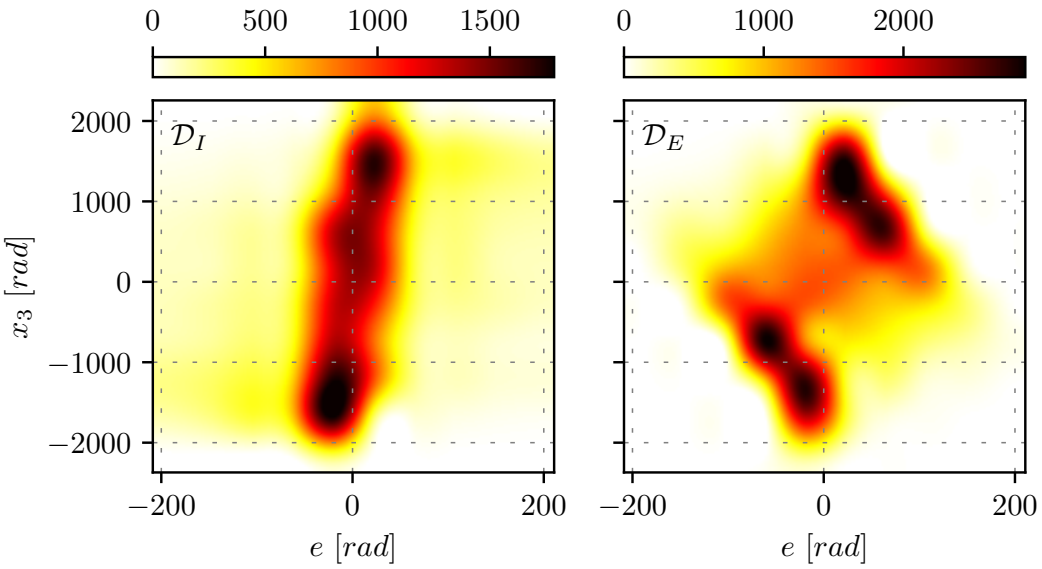
Source: author

As shown in Figure 59, the closed loop system response is similar for small steps of same amplitude and different operating points, the main difference being, once again, the size of the overshoot, indicating that the found controller $\mathcal{C}$ does not fully cancel out the nonlinearity of the system. There is, however, a much better tracking and matching with the Reference Model's output. Now, nothing special happens at time $t = 0.75\ s$ and the closed loop system has basically the same overall behavior in all of its output space.

One last analysis to make is to look at the datasets distribution, the ideal dataset is created using the same reference as shown in Figure 49 but using the Reference Model output of S_{rn}.

Figure 60 shows the heatmaps for the ideal $\mathcal{D}_I$ and experimental $\mathcal{D}_E$ datasets.

Figure 60 – Ideal and Experimental datasets distribution.

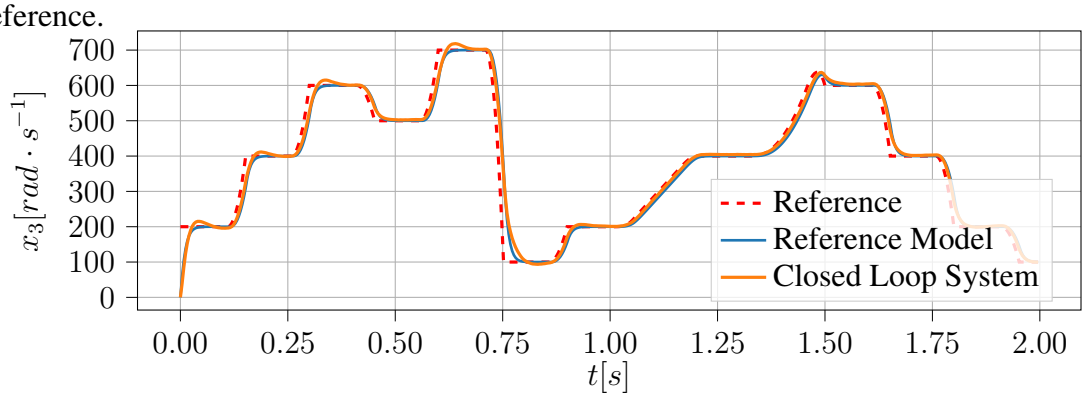


Source: author

While in Figure 50, the ideal dataset has the $\mathcal{D}_I$ heatmap showing a slight tendency on the data, where positive errors tend to have negative outputs and negative errors tend to have positive outputs, the exact opposite pattern happens in the ideal dataset $\mathcal{D}_I$, shown in Figure 60. Now the $\mathcal{D}_E$ heatmap matches the ideal dataset better. The same pattern matching is observed in the velocity distributions of $\mathcal{D}_I$ and $\mathcal{D}_E$. This seems to be the main reason why in this case the closed loop system performs better, the experimental dataset better explores the dynamics of the system in the regions that are of interest when in closed loop.

Although $\mathcal{D}_I$ and $\mathcal{D}_E$ are a better match in this case, they're still different in many regions, thus, it is interesting to see the performance metrics of this closed loop system with the smoothed reference of Figure 53, the system's response is shown in Figure 61.

Figure 61 – Comparison of closed loop system with the Reference Model to the smoothed reference.



Source: author

Here, the model reference error measured on all $N = 500$ samples of the filtered input is of $J(\boldsymbol{\theta}) = 77.53$, about 2.1 times less than in the previous case. As in the last case the main problem with the DNN approach seems to be with generating training data that explores the output-error space of interest in application.

4.2.4 Case 4: super-sampling

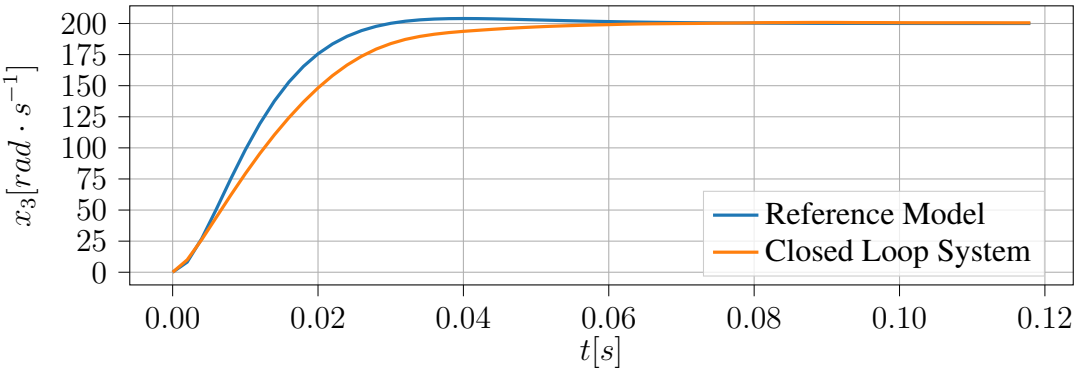
Both controllers found using the Reference Models S_r and S_{rn} seem to have a somewhat slower response than their reference model's. One way to improve this aspect is to simply lower the sampling time.

Consider that the discrete representation $\mathcal{C}_{dis}$ has, for a sampling time of Δt, a continuous representation $\mathcal{C}_{con,1}$, with poles $\boldsymbol{p}_1$ and has, for a sampling time $\Delta t/2$, a continuous representation $\mathcal{C}_{con,2}$, with poles $\boldsymbol{p}_2$.

In a linear controller, the effect of lowering the sampling time from Δt to $\Delta t/2$ while maintaining the same discrete representation $\mathcal{C}_{dis}$ is of increasing the poles $\boldsymbol{p}_1$ to $\boldsymbol{p}_2$ with $\boldsymbol{p}_2 = 2\boldsymbol{p}_1$, thus making the controller response of $\mathcal{C}_{con,2}$ faster than $\mathcal{C}_{con,1}$. Therefore, one possible way of improving the performance of the controllers found in subsections 4.2.2 and 4.2.3 is to just super-sample them.

To show that this does indeed improve the closed loop performance, the system (126) is simulated as before, the same controller found in subsection 4.2.2 is used. But, this time, the control action is calculated at every $\Delta t = 2 \times 10^{-3}$ and applied in the continuous system in a ZOH fashion. The system's closed loop response to a step of $200\ rad \cdot s^{-1}$ is shown in Figure 62.

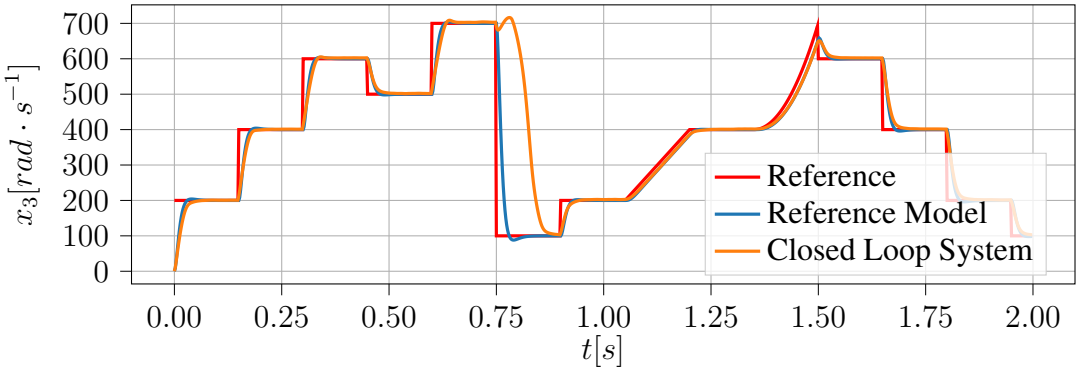
Figure 62 – Comparison of closed loop system with the Reference Model to a more challenging reference.



Source: author

The step response in Figure 62 has a model reference error of $J(\boldsymbol{\theta}) = 127.49$ measured on all the 60 samples, i.e., up to the same time as in the case of subsection 4.2.2, an error 1.7 times smaller than before. The same behavior holds in all the output space of the closed loop system, as shown in Figure 63.

Figure 63 – Comparison of closed loop system with the Reference Model to a more challenging reference.

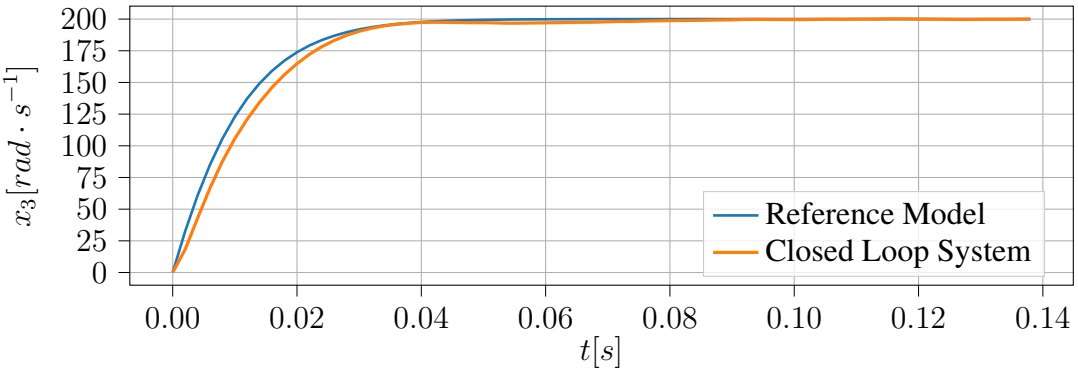


Source: author

Figure 63 shows also that the same behavior with the negative step happens in this super sampled system, which is reasonable since the same big error appears to a controller that wasn't trained to such error magnitudes.

Figure 64 shows the step response of the closed loop system using the same controller as the one found in subsection 4.2.3, but sampled at $\Delta t = 2 \times 10^{-3}$.

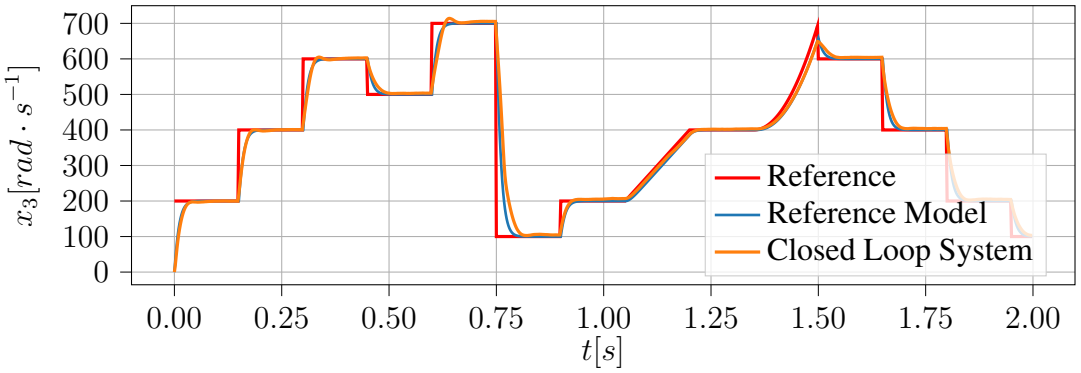
Figure 64 – Comparison of closed loop system with the Reference Model.



Source: author

The step response in Figure 64 has a model reference error of $J(\boldsymbol{\theta}) = 42.51$ measured on all the 60 samples, i.e., up to the same time as in the case of subsection 4.2.3, almost 2.7 times less than before. Once again, the same behavior holds in all the output space of the closed loop system, as shown in Figure 65.

Figure 65 – Comparison of closed loop system with the Reference Model to a more challenging reference.



Source: author

The closed loop response shown in Figure 65 shows that the super sampled closed loop system follows the Reference Model much closer than before.

The above examples show that super-sampling the closed loop system is an option to improve the closed loop response of the system without the need to train another controller. It is also interesting to notice that sampling the system at $\Delta t = 2 \times 10^{-4}$ to create the IO-data from the start does not work, the problem becomes much harder to solve since the noise in the virtual error grows substantially.

5 CONCLUSION

In this work, the nonlinear VRFT method was used to design nonlinear controllers using two different classes of parametrizations, polynomial and Deep Neural Networks, where in both of them a form of regularization was used to limit the model complexity and to better minimize the VRFT cost function on the test set.

Polynomial parametrizations are an interesting alternative, because they're linear in the parameters, the VRFT problem can be solved without an iterative procedure. Nonlinear systems, however, can have fairly complex behavior, and a controller parametrization that is linear in the parameters can become cumbersome to define and work with, since they might require a number of parameters that is too large.

One of the reason why DNNs are used in this work is to alleviate the controller parametrization definition, one of the most challenging tasks involved in the method's application. Since, as shown, the same DNN can be used as a controller parametrization even when different systems are considered. Using DNNs in the context of VRFT is not a new idea, for instance, (ESPARZA; SALA; ALBERTOS, 2011) used DNNs in this context, there, however, the filter used to match the J and J^V cost functions is a linear system that needs to be evolved with the optimization procedure for all parameters, an approach that adds on the computational complexity of training the model.

One of the contributions of this master thesis is using DNNs as the controller, but instead of using the filter of (ESPARZA; SALA; ALBERTOS, 2011) or (CAMPI; SAVARESI, 2006), using the same filter as in the linear VRFT. This filter is applied before training in the virtual error and control signals, and does not add extra complexity in training, allowing the usage of much bigger and complex DNNs. The usage of the GRU cells is, to the best of this author's knowledge, new in this context and the results indicate that its usage as a controller parametrization is promising.

Another contribution of this work is on the provided analysis to the obtained results, mainly the dataset mismatch of what would be an ideal dataset and the obtained dataset in experiment. These results seem to indicate that a better exciting signal should be used in place of the one used in this work, the imperfections in the controller's performance seems at least partially, as the results shows, to be caused by a lack of data of the system in the

region where the reference model operates. It is not clear how much of the imperfections are caused by this reason and by the VRFT method alone and this analysis is left for future works, where, perhaps, one can utilize other one-shot DD methods, such as OCI, to compare the results.

Regularization has been shown to be very important when using DNNs as controllers, allowing them to perform well on the test set and ultimately in indirectly minimizing the model reference cost function. This was also observed in the simpler linearly parametrized controller, where a Monte Carlo simulation indicated that the use of L^1 regularization gives more robustness to the method's application[1] when on environments with lower signal-to-noise ratios.

From the results of this work it is possible to conclude that the usage of the VRFT with DNN controllers is a viable option with the simpler linear filter herein used. Also, more attention should be taken in designing the input signal to excite the system for the data acquisition. Due to the complexity of nonlinear systems, an input signal that better explores the desired system's dynamics in closed loop is essential in the success of the method's application. Thus, a promising direction for future works in this area seems to be in designing input signals that attempt to excite the system in a way that not only the input-error heatmap is matched but also the input-velocity heatmap.

Another interesting future line of work that seems to be promising is in designing a RNN model that is more generalizable for this problem. Ideally both the model generalization and the dataset quality should be improved, but improving the generalization power of the model might show itself easier to deal with. The machine learning community has many regularization techniques specifically designed for RNNs and DNNs in general, see (GÉRON, 2022; GOODFELLOW; BENGIO; COURVILLE, 2016; CHARU, 2018) for examples, some of them were used in this work, but not all. It seems likely that a better generalizing method can be applied to the RNNs herein used, giving better controllers.

[1] In the sense that increases the chance of a controller rendering the closed loop dynamics stable.

REFERENCES

ÅSTRÖM, K. J.; WITTENMARK, B. **Adaptive control**. [S.l.]: Courier Corporation, 2013.

BAZANELLA, A. S.; CAMPESTRINI, L.; ECKHARD, D. **Data-driven controller design**: the h2 approach. [S.l.]: Springer Science & Business Media, 2011.

BAZANELLA, A. S.; ECKHARD, D.; SEHNEM, R. M. Nonlinear VRFT with LASSO. *In*: BRAZILIAN SYMPOSIUM ON INTELLIGENT AUTOMATION (SBAI, MANAUS, AMAZONAS, OCTOBER 2023), 30., 2023, Manaus, Amazonas. **Proceedings [...]** [S.l.: s.n.], 2023. v. 1, n. 1.

BISHOP, C. M. Training with noise is equivalent to Tikhonov regularization. **Neural computation**, [S.l.], v. 7, n. 1, p. 108–116, 1995.

BOEIRA, E. C.; ECKHARD, D. C. Multivariable virtual reference feedback tuning with Bayesian regularization. *In*: CONGRESSO BRASILEIRO DE AUTOMÁTICA-CBA, 2019. **Proceedings [...]** [S.l.: s.n.], 2019. v. 1, n. 1.

BUŞONIU, L. *et al.* Reinforcement learning for control: performance, stability, and deep approximators. **Annual Reviews in Control**, [S.l.], v. 46, p. 8–28, 2018.

CAMPESTRINI, L. *et al.* Data-driven model reference control design by prediction error identification. **Journal of the Franklin Institute**, [S.l.], v. 354, n. 6, p. 2628–2647, 2017.

CAMPI, M. C.; LECCHINI, A.; SAVARESI, S. M. Virtual reference feedback tuning: a direct method for the design of feedback controllers. **Automatica**, [S.l.], v. 38, n. 8, p. 1337–1346, 2002.

CAMPI, M. C.; SAVARESI, S. M. Direct nonlinear control design: the virtual reference feedback tuning (vrft) approach. **IEEE Transactions on Automatic Control**, [S.l.], v. 51, n. 1, p. 14–27, 2006.

CARRASCO, D. S.; GOODWIN, G. C.; YUZ, J. I. Vector measures of accuracy for sampled data models of nonlinear systems. **IEEE Transactions on Automatic Control**, [S.l.], v. 58, n. 1, p. 224–230, 2012.

CHARU, C. A. **Neural networks and deep learning**: a textbook. [S.l.]: Spinger, 2018.

CHO, K. *et al.* Learning phrase representations using RNN encoder-decoder for statistical machine translation. **arXiv preprint arXiv:1406.1078**, [S.l.], 2014.

DOZAT, T. Incorporating Nesterov Momentum into ADAM. **OpenReview**, [S.l.], 2015. Available at: <https://api.semanticscholar.org/CorpusID:620137>. Accessed in: 30 nov 2023.

DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. **Journal of machine learning research**, [S.l.], v. 12, n. 7, 2011.

ESPARZA, A.; SALA, A.; ALBERTOS, P. Neural networks in virtual reference tuning. **Engineering Applications of Artificial Intelligence**, [S.l.], v. 24, n. 6, p. 983–995, 2011.

FORMENTIN, S.; KARIMI, A. Enhancing statistical performance of data-driven controller tuning via L2-regularization. **Automatica**, [S.l.], v. 50, n. 5, p. 1514–1520, 2014.

GÉRON, A. **Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow**. New York: " O'Reilly Media, Inc.", 2022. 856 p.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. [S.l.]: MIT press, 2016.

GREFF, K. *et al.* LSTM: a search space odyssey. **IEEE transactions on neural networks and learning systems**, [S.l.], v. 28, n. 10, p. 2222–2232, 2016.

GRIEWANK, A. Who invented the reverse mode of differentiation. **Documenta Mathematica, Extra Volume ISMP**, [S.l.], v. 389400, 2012.

HJALMARSSON, H.; GUNNARSSON, S.; GEVERS, M. A convergent iterative restricted complexity control design scheme. *In*: IEEE CONFERENCE ON DECISION AND CONTROL, 1994., 1994. **Proceedings [...]** [S.l.: s.n.], 1994. v. 2, p. 1735–1740.

HJALMARSSON, H. *et al.* Iterative feedback tuning: theory and applications. **IEEE control systems magazine**, [S.l.], v. 18, n. 4, p. 26–41, 1998.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, [S.l.], v. 9, n. 8, p. 1735–1780, 1997.

KAMMER, L. C.; BITMEAD, R. R.; BARTLETT, P. L. Direct iterative tuning via spectral analysis. **Automatica**, [S.l.], v. 36, n. 9, p. 1301–1307, 2000.

KARIMI, A.; MIŠKOVIĆ, L.; BONVIN, D. Iterative correlation-based controller tuning. **International journal of adaptive control and signal processing**, [S.l.], v. 18, n. 8, p. 645–664, 2004.

KH, K. **Nonlinear Systems**. [S.l.]: Englewood Cliffs, NJ, Prentice-Hall, 1996.

KINGMA, D. P.; BA, J. Adam: a method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, [S.l.], 2014.

LINNAINMAA, S. **The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors**. 1970. 66 p. Master Thesis (in Finnish) — University of Helsinki, 1970.

LINNAINMAA, S. Taylor expansion of the accumulated rounding error. **BIT Numerical Mathematics**, [S.l.], v. 16, n. 2, p. 146–160, 1976.

LIU, R.; SHANG, Z.; CHENG, G. On deep instrumental variables estimate. **arXiv preprint arXiv:2004.14954**, [S.l.], 2020.

LJUNG, L. **System Identification**: theory for the user. [S.l.]: Prentice Hall PTR, 1999. (Prentice Hall information and system sciences series).

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, [S.l.], v. 5, p. 115–133, 1943.

MNIH, V. *et al.* Human-level control through deep reinforcement learning. **nature**, [S.l.], v. 518, n. 7540, p. 529–533, 2015.

MONTÚFAR, G. F. Universal approximation depth and errors of narrow belief networks with discrete units. **Neural computation**, [S.l.], v. 26, n. 7, p. 1386–1407, 2014.

NESTEROV, Y. E. A method of solving a convex programming problem with convergence rate O\bigl(k^2\bigr). *In*: DOKLADY AKADEMII NAUK, 1983. **Proceedings [...]** [S.l.: s.n.], 1983. v. 269, n. 3, p. 543–547.

PILLONETTO, G. *et al.* Kernel methods in system identification, machine learning and function estimation: a survey. **Automatica**, [S.l.], v. 50, n. 3, p. 657–682, 2014.

POLYAK, B. T. Some methods of speeding up the convergence of iteration methods. **Ussr computational mathematics and mathematical physics**, [S.l.], v. 4, n. 5, p. 1–17, 1964.

RALLO, G. *et al.* Virtual reference feedback tuning with bayesian regularization. *In*: EUROPEAN CONTROL CONFERENCE (ECC), 2016., 2016. **Proceedings [...]** [S.l.: s.n.], 2016. p. 507–512.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, [S.l.], v. 65, n. 6, p. 386, 1958.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, [S.l.], v. 323, n. 6088, p. 533–536, 1986.

SCHMIDHUBER, J. Deep learning in neural networks: an overview. **Neural networks**, [S.l.], v. 61, p. 85–117, 2015.

SCHOUKENS, J.; RELAN, R.; SCHOUKENS, M. Discrete time approximation of continuous time nonlinear state space models. **IFAC-PapersOnLine**, [S.l.], v. 50, n. 1, p. 8339–8346, 2017.

SIETSMA, J.; DOW, R. J. Creating artificial neural networks that generalize. **Neural networks**, [S.l.], v. 4, n. 1, p. 67–79, 1991.

SÖDERSTRÖM, T. Errors-in-variables methods in system identification. **Automatica**, [S.l.], v. 43, n. 6, p. 939–958, 2007.

SÖDERSTRÖM, T. **Errors-in-variables methods in system identification**. [S.l.]: Springer, 2018.

SÖDERSTRÖM, T.; STOICA, P. **System Identification**. [S.l.]: Prentice Hall, 1989. (Prentice-Hall Software Series).

WANNER, G.; HAIRER, E. **Solving ordinary differential equations II**. [S.l.]: Springer Berlin Heidelberg New York, 1996. v. 375.

WATKINS, C. J.; DAYAN, P. Q-learning. **Machine learning**, [S.l.], v. 8, p. 279–292, 1992.

WILSON, D. R.; MARTINEZ, T. R. The general inefficiency of batch training for gradient descent learning. **Neural networks**, [S.l.], v. 16, n. 10, p. 1429–1451, 2003.

YUZ, J. I.; GOODWIN, G. C. On sampled-data models for nonlinear systems. **IEEE transactions on automatic control**, [S.l.], v. 50, n. 10, p. 1477–1489, 2005.

ZIEGLER, J. G.; NICHOLS, N. B. Optimum settings for automatic controllers. **Transactions of the American society of mechanical engineers**, [S.l.], v. 64, n. 8, p. 759–765, 1942.