ÖREBRO UNIVERSITY

International Master's Thesis

Identification and Predictive Control Using Recurrent Neural Networks

Nima Mohajerin
Technology

Studies from the Department of Technology at Örebro University
ÖREBRO 2012

Identification and Predictive Control Using Recurrent Neural Networks

Studies from the Department of Technology at Örebro University

Nima Mohajerin

Identification and Predictive Control Using Recurrent Neural Networks

Supervisor: **Prof. Ivan Kalaykov**

Examiners: **Dr. Dimitar Dimitrov**
Dr. Boyko Iliev

Title: Identification and Predictive Control Using Recurrent Neural Networks

ISSN 1650-8580

Abstract

In this thesis, a special class of Recurrent Neural Networks (RNN) is employed for system identification and predictive control of time dependent systems. Fundamental architectures and learning algorithms of RNNs are studied upon which a generalized architecture over a class of state-space represented networks is proposed and formulated. Levenberg-Marquardt (LM) learning algorithm is derived for this architecture and a number of enhancements are introduced. Furthermore, using this recurrent neural network as a system identifier, a Model Predictive Controller (MPC) is established which solves the optimization problem using an iterative approach based on the LM algorithm. Simulation results show that the new architecture accompanied by LM learning algorithm outperforms some of existing methods. The third approach which utilizes the proposed method in on-line system identification enhances the identification/control process even more.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Ivan Kalaykov, who trusted in me and gave me the opportunity to pursue my passion in studying and conducting my research in Neural Networks. I hope I have fulfilled his expectations.
During the past two years in AASS labs at Örebro University I was blessed learning from a number of humble yet truly competent researchers among whom I would like to mention Dr. Dimitar Dimitrov, whose meticulous comments effectively aided me in improving the quiality of this final version. Our challenging discussions during a tough but fascinating humanoid walking project played inspiring roles for me to shape up the grounds on which I founded this thesis.
I want to thank my parents, for being always supporting and encouraging. I am sure nothing will replace their unconditional love and devotion. My other half, whose presence is the happiest reality in my life, her tolerance and understanding is not expressible in words. I treasure spending every minute by her side and I want to truly thank her for being there whenever I needed her. To her and my parents, I would like to dedicate this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

1.1 Motivation

Decades ago it was already clear for control engineers and computer scientists that coping with complex tasks and environments needs tools beyond traditional way of modeling them by using their physical and/or chemical properties. In the quest to find methods and approaches to control tremendously complex processes and making decisions in highly nonlinear environments, *Artificial Neural Networks (ANN)* have been receiving a considerable amount of attention, both as an identification tool and as a controller or a complex decision maker. There are essentially two main supporting reasons for this trend. First, ANNs are *Universal Apprriximators*[1]. Second is the origin of insipration behind ANNs; they are basically inspired from the complex architecture of interconnections among cells in mamals nervous systems. So, there has been continuously a hope that research in this direction will eventually lead to a system that behave, at least partially, similar to mamals nervous systems.
There exist various architectures and learning methods for ANNs, each of which is suitable for specific purpose such as classification, function approximation, data filtering, etc.[2] According to their signal-flow, ANN can be divided into two categories: *Feed Forward Neural Network*(FFNN) and *Recurrent Neural Network*(RNN). Noticeably, in FFNN signals flow from input(s) toward output(s) with no feedback, i.e., each neuron output is connected to the inputs of the next layer neurons. But in RNN, neurons can receive inputs from all or some other neurons regardless of their position within the network. An essential difference between FFNN and RNN is that FFNNs are static, i.e., each set of inputs can produce only one set of outputs. In contrast RNNs are dynamic, i.e., each set of inputs may produce different sets of outputs. One can also say that FFNNs are memoryless, but RNNs are able to temporarily

[1]Universal approximation capability is later discussed.

[2]Throughout this thesis, it is assumed that the reader is familiar with ANN. However, comprehensive introduction and valuable resources can be found in a number of books such as [36, 5, 57, 14].

memorize information given to them. Therefore, RNNs are better candidates to identify time-varying systems and cope with the situations where time plays a key role, for instance time-sequence analysis. However, nothing comes for free: feedback throughout the network adds both dynamics to the network and complexity. Dynamic systems need stability analysis and in general there is no guarantee that a network is stable. On the other hand, because of dependencies of various signals within the network (e.g., outputs of the neurons) on the history of the network, studying RNN stability is not an easy task. Moreover, learning algorithm and convergence is in general a difficult problem to address. Prior to RNN, time-delayed networks were employed to tackle the problem of identifying time dependencies in a given context. Time-delayed networks are FFNNs that have one or more time-delay units distributed over their inputs. However, in time-delayed networks, neurons in hidden layer receive inputs only from outputs of neurons in the previous layer.

One of the most important features that comes with the introduction of feedback to a neural network is the presence of *state*. States play significant role in different areas of system analysis and control engineering. By definition, *states* (or *state variables*) are the smallest set of system variables such that the values of the members of the set at time t_0 along with known forcing functions completely determine the value of all system variables for all $t \geqslant t_0$ [63]. State-space representation of the linear systems is a powerful tool for analysis and design of controllers. Although for non-linear systems it is not equally useful, there is still a significant interest in representing them in state-space forms [47].

Demonstrating such capabilities, RNNs are among the best candidates for identifying complex systems and time-dependent environments and processes. Considering control objectives, as a model is identified out of a system, it can be used in different schemes to control the original system. Among those, schemes which rely on prediction of the identified system are called *predictive* schemes. To be more precise, in predictive schemes, there should be a model estimating the system under control to a satisfying degree. Using the model, a predictive controller calculates the optimal control strategy over a horizon (which can be finite or infinite) by using predictions that comes from the model in hand. This approach which is often based on a *receding horizon* is called *Model-based Predictive Control*(MPC or MBPC) [56]. However, if the predictive controller relies only on a particular response of the system (step response) then the method is often referred to as *Generalized Predictive Control* (GPC) [20, 21]. Furthermore, if the identified model is based on ANNs then the method is normally known as *Neural Network GPC*(NNGPC)[3] [74].

MPC scheme has made a significant impact on industrial control engineering [56]. Noticeably, in industrial control problems, most often a mathematical model of the process under control is available. What distinguishes MPC

[3]Since NGPC is referred to Nonlinear GPC, while still it may also refer to Neural GPC, we use the term NNGPC for the later to prevent ambiguities, however, in the literature NGPC may also refer to Neural GPC.

over other control approaches in industrial control is that MPC can take into account different constraints, e.g. in input, output signals or states, and allows operation closer to actuator limitations. Additionally, the idea of looking ahead into the future then decide a strategy is equally interesting; this is what humans do in most of their daily tasks. Imagine a driving scenario: the driver continuously estimates the behaviour of surrounding objects: vehicles, stationary obstacles, pedestrians, etc. According to the driver's prediction about the behaviour of other objects in a very near future and desired trajectory for his vehicle, he decides about his control action over his vehicle. However, this is not very likely that one is able to mathematically model all the processes in a given environment in order to predict their exact future behaviour. This is where NNGPC becomes important. It incorporates the idea of predicting future of the system of interest with the identification power of ANN.
In this thesis, we intend to study the ability of Recurrent Neural Networks in identifying time-dependent systems and integration of these networks in a predictive control scheme. The rest of this chapter is organized as follows: after this short introduction and motivating the subject of study, next we will describe what is actually to be focused mainly in this thesis. We will try to narrow the subject as much as possible. Afterward, the structure of the thesis is outlined. The last subsection integrates the notation and terminology to be used throughout the thesis.

1.2 Contribution

As described above, the main intention of this thesis is to study a special class of ANN and incorporate this class into a predictive control scheme. Therefore, we shall start by a short introduction to this class of ANN, namely Recurrent Neural Networks. Since we intend to use a specific form of RNN, primarily as a system identifier, we will try to moderately cover various architectures of RNNs that have been used in the literature, mostly as system identifiers. In addition to that, two fundamental learning algorithms for RNNs, namely Real Time Recurrent Learning (RTRL) and Backpropagation Through Time (BPTT), are discussed. In this thesis, we will encounter two main optimization problems which are tackled using Levenberge-Marquardt algorithm (LM or LMA). Thus, LM algorithm is also reviewed.
In this thesis a class of RNNs will be proposed which encompasses a number of various architectures, some of which are already proposed and studied in the literature. We name this architecture, *Recurrent System Identifier*(RSI). As we shall shortly see, it covers a considerable number of different yet useful architectures. We shall study a specific class of RSI, namely LRSI, where L stands for *Linear*. Although, as will be seen, this specific form is not linear per se, the inputs to neurons will be a linear combination of system signals (inputs, outputs and states). A LM-based learning algorithm will be derived for LRSI.

As we know, a major drawback of derivative-based learning algorithms is the dependency of the algorithm on the architecture of the learner which in our scenario is a RNN. Additionally, recursions in RNN drastically complicate the derivation procedure. Thus, since our proposed architecture can produce a number of networks, this package, i.e., LRSI architecture with LM-based learning algorithm, is immediately applicable to a vast number of problems where time is a noticeable concern. To do so, it is only needed to modify the proposed architecture according to the requisites of a particular problem; we can still apply the same learning algorithm with no or quite minor modifications.
The second contribution of this thesis is to use LRSI in a control scenario. As a model is identified by LRSI, a MPC scheme utilizes it, as the predictor, and searches for some optimal control strategy to achieve a desired behaviour. Since the model which we are using is highly non-linear, we will encounter a non-linear non-convex optimization. We will keep on using the LM optimization technique in this later case. Thus, the controller we will propose is basically an iterative LM-MPC controller. Comprehensive formulation of the control optimization will be presented.

1.3 Thesis Structure

According to the described approach, the thesis is partitioned into 6 chapters. After this first chapter which is setting the stage for the thesis, the second chapter is organized to present a short, but enough informative introduction to RNN, both in terms of architectures and learning algorithms. In Chapter 3, the main contribution of the thesis is presented. We will describe RSI and LRSI in detail and will also derive the LM-based learning algorithm for LRSI. Chapter 4 presents the second half of the thesis, i.e., the MPC scheme based on iterative LM optimization. Comprehensive formulation of generating control actions will be presented. Chapter 5 demonstrates the configuration and setups for our experiments for which the results will also be illustrated and analyzed. The last chapter is devoted to a summary of what has been presented, conclusion on the pros and cons of the proposed schemes and the future ideas and extensions.

1.4 Notation and Terminology

We will frequently use different abbreviations which are listed in table 1.1.
In presenting mathematical formulations, we adopt the same notation that is usually used in the literature. Normal alphabets are used to indicate scalar values such as α, k. Vectors and matrices are shown using **bold** type face while lowercase means a vector, such as $\mathbf{x}$ and uppercase means a matrix, such as $\mathbf{A}$. Function vectors obey the same rule.
Most of the analysis and calculations to be presented are in discrete-time domain. So, k is specifically used to illustrate discrete time step while t may be

used to show both discrete and continuous time. Note that when a quantity is time-dependent, the time indicator appears inside parenthesis to show that particular quantity is *also* a function of time. However, in figures, they may appear as subscripts for the sake of keeping the illustrations small in size. Other mathematical formulations should be enough descriptive in their related context as they are being derived/presented.

ANN	Artificial Neural Network
BP	Back Propagation
BPTT	Back Propagation Through Time
CARIMA	Controlled Auto-Regressive Integrated Moving Average
DFA	Deterministic Finite-state Automata
DRNN	Diagonal Recurrent Neural Network
FCRNN	Fully Connected Recurrent Neural Network
FFNN	Feed-Forward Neural Network
GPC	Generalized Predictive Controller
RSI	Recurrent System Identifier
LGRF	Locally Recurrent Globally Feedforward network
LRSI	Linear Recurrent System Identifier
LM (also LMA)	Levenberg-Marquardt Algorithm
LSTM	Long-Short Term Memory
MIMO	Multi-Input Multi-Output
MISO	Multi-Input Single-Output
ML	Machine Learning
MLP	Multi-Layer Perceptron
MPC (also MPBC)	Model-based Predictive Control
NARMA	Non-linear Auto-Regressive Moving Average
NARMAX	Non-Linear Auto-Regressive Moving Average with Exogenous Input
NARX	Nonlinear AutoRegressive with Exogenous Inputs
NGPC	Nonlinear Generalized Predictive Controller
NN	Neural Networks
NNGPC	Neural Generalized Predictive Controller
PE	Processing Element
RL	Reinforcement Learning
RMLP	Recurrent Multilayer Perceptron
RNN	Recurrent Neural Network
RTRL	Real Time Recurrent Learning
SSE	Sum of Squared Errors
SSR	State-Space Representation
SRN	Simple Recurrent Network
SRWNN	Simple Recurrent Wavelet Neural Network
TDNN	Time-Delay Neural Network
TLFN	(Focused) Time Lagged Feedforward Network

Table 1.1: Abbreviations

Chapter 2
Recurrent Neural Networks

Time is an inherent property of the physical world we live in. Everything that happens, our mind perceives it in a timely fashion. Although the ability of our mind to observe all possible instances of time is so restricted, most often, the perceivable time difference plays a critical role in our daily life. This means that interacting with real environments needs to account for time-dependencies appropriately.
As previously discussed, Artificial Neural Networks (or simply *Neural Networks*)[1] has been one of the most famous methodologies in System Identification and Machine Learning (ML) community. Particularly FFNNs have been receiving an immense amount of attention in pattern recognition problems [15, 36]. However, most of these problems are naturally stationary. When it comes to time-varying problems, FFNNs capabilities become significantly inadequate due to their static nature. This huge drawback has always been a problem for which *connectionists*[2] are continuously looking for remedies.
To add dynamic to FFNN, one of the earliest attempts is to use ordinary time delay units to perform temporal processing [36]. This approach results in a network which is called *Time-Delay Neural Network* (TDNN). Accordingly, TDNN is a multilayer feedforward network whose hidden neurons and output neurons are *replicated across time*. However, it appears that TDNNs work best for a very limited class of applications (mostly, speech recognition problems).[3]
A more common approach is to place delay units on the input layers. This architecture is called *Focused Time Lagged Feedforward Networks* (TLFN). The term *focused* is to emphasize that delay units are placed on the input layer only. This way of memory placement provides the network with a history of input signals, but since there is no recurrence inside the network, it does not possess

[1]Here after, Artificial Neural Networks, Neural Networks, ANN and NN will be used interchangeably.

[2]Connectionism is a movement in cognitive science which hopes to explain human intellectual abilities using artificial neural networks [32]

[3]More on TDNNs can be found in [36, 67]

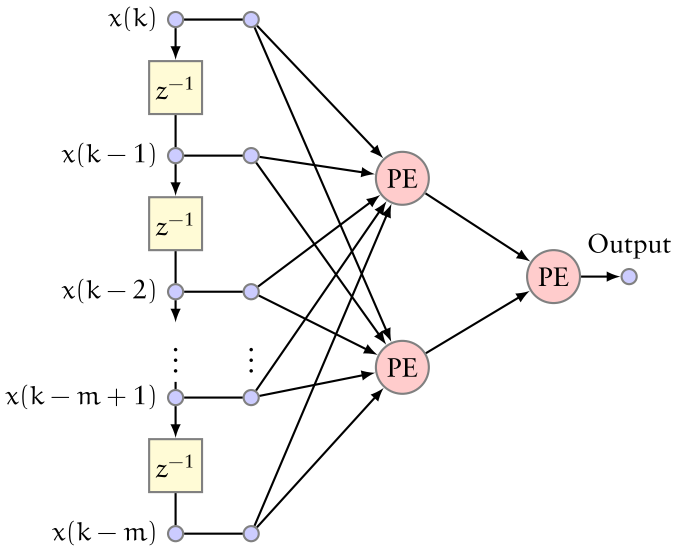


Figure 2.1: A simple TLFN.

any internal state (Figure 2.1).

Figure 2.1 illustrates a very simple form of a TLFN. In this Figure, **PE** stands for *Processing Element* which we adopted from [67]. It is the smallest processing unit of the network.[4] Detailed discussion about TLFNs can be found in [36].

A more interesting way to add dynamics to a FFNN is to introduce *feedback* connections. This leads to a new class of neural networks called *Recurrent Neural Network* (RNN). As discussed, presence of feedback signals gives dynamic to a network and leads to definition of states, but it also adds complexity and raises stability issues. In this chapter we want to present an overview on RNN, their different architectures and learning algorithms. Although RNN can be used in both continuous and discrete time domains, it should be emphasized that throughout this chapter and the rest of our discussion it is assumed that all the signals and systems are discrete-time.

Before we start our main discussion in this chapter, we would like to address a question we indirectly raised in the previous chapter: stability issues. It is clear that due to recurrence in RNNs, one needs to study stability of these networks in order to gain a full control over them. RNNs can be viewed as non-linear systems, hence nonlinear stability analysis techniques and tools can be readily applied to them, such as Lyapunov function method and LaSalle invariance principle. There are plenty of articles in the literature investigating stability in

[4]It can be a general model of a neuron. In that case **PE** will be replaced by $\mathcal{N}$.

various forms of RNNs, for example [9, 10, 13, 30, 39, 42, 43, 53, 65, 77, 82], and still it is an open issue. One main reason for that is the effect of a network architecture on stability criteria: various architecture are essentially different nonlinear systems, and for each nonlinear system one may need to investigate stability. However, there is another but loose view towards stability in on-line learning networks. If a network continuously learns and if one assures that in the learning process the error between the network output(s) and desired behaviour is always bounded, the stability of the network is then related to the stability of the system that is being learned. In this naive interpretation, one may continue using the network in the identification and/or controlling problem, but a comprehensive stability analysis is still needed. Since we are more focused on architectures and learning in RNNs we adopt the same loose viewpoint but we bear in mind that it does not replace the need for stability analysis of our scheme to be proposed in the next chapter.
This chapter is organized as follows: in the next section, common RNN architectures are studied. Afterward, famous learning algorithms are presented.

2.1 Recurrent Neural Networks Architectures

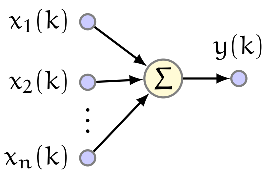
There are no universally accepted categorization of RNN architectures. In this section, we will try to outline the most general and commonly used architectures found in the literature. Before that, we introduce building blocks of RNNs. There are four basic building blocks in every (Recurrent) Neural Network. Each of these blocks has two ports: input and output, which are usually referred to by x and y, respectively. Depending on the dimension of the ports, blocks can be either MIMO (Multi-Input Multi-Output) or MISO (Multi-Input Single-Output). Note that here, single-input is assumed to be a special case of multi-input. These four blocks are shown in Figure 2.2 and are listed below:

- **Summation**(MISO): Generates summation over the inputs it receives: $y(k) = \sum_{i=1}^{n} x_i(k)$ where n is the dimension of input port. (Figure 2.2a)

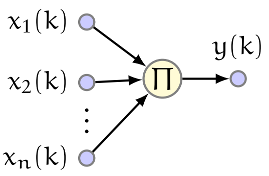
- **Multiplication**(MISO): Generates the result of multiplying signal values it receives: $y(k) = \prod_{i=1}^{n} x_i(k)$ where n is the dimension of input port. (Figure 2.2b)

- **Delay**(MIMO): Generates a delayed version of its input signal(s) on its output(s). Usually the delay is for one time step: $\mathbf{y}(k) = \mathbf{x}(k-1)$. Note that ports dimensions are equal. (Figure 2.2c)

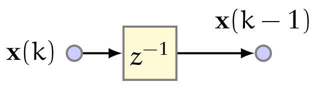
- **Non-linearity**(MIMO): Includes a non-linear function and generates the output of that function given the inputs: $\mathbf{y}(k) = \mathbf{f}(\mathbf{x}(k))$. (Figure 2.2d)



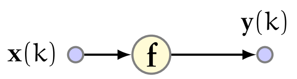
(a) Summation



(b) Multiplication



(c) Delay (memory)



(d) Non-linearity

Figure 2.2: NN Building blocks

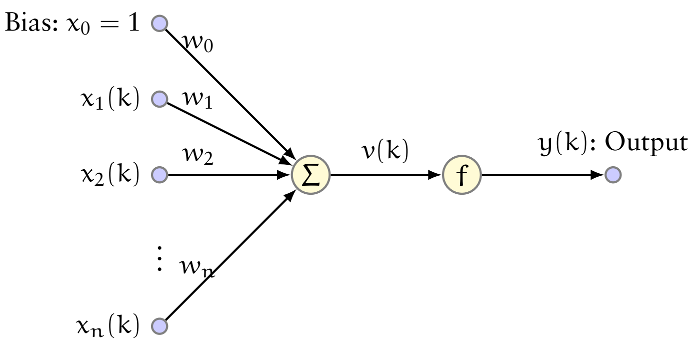


Figure 2.3: General model of a neuron.

Having introduced these building blocks, model of a neuron is illustrated in Figure 2.3. Accordingly, the output of a neuron (i.e., $y(k)$) is calculated using

$$y(k) = f(\sum_{i=1}^{n} x_i(k)w_i + w_0) = f(\mathbf{w}^T\mathbf{x}(k)) = f(v(k)). \tag{2.1}$$

In this model, $x_i(k)$ (for $i = 1, ..., n$) are inputs to the neuron at time k, where the input vector dimension is n. w_i (for $i = 0, ..., n$) are called *weights* and are learning parameters. Note that $x_0 = 1$ is fixed, therefore, w_0 is a bias term. $v(k)$ is usually referred to as *activation* or *induced local field*. $f(.)$ is usually a nonlinear function. This function is called *activation function*. There are several types of activation functions, for a comprehensive discussion on different activation functions consult [57, 36, 67]. However, for our discussion it suffices to know that the activation function should be continuous on its domain with

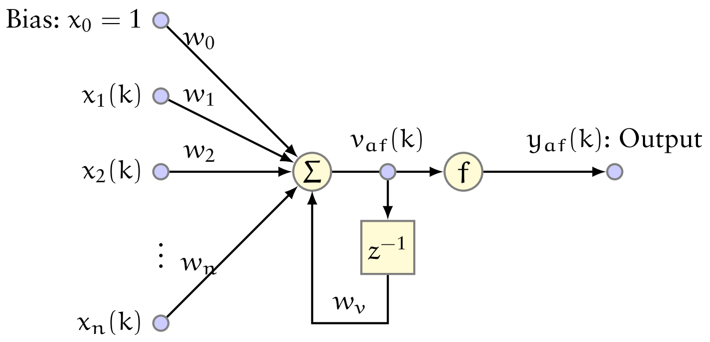


Figure 2.4: Model of a neuron with one-step-delay activation feedback.

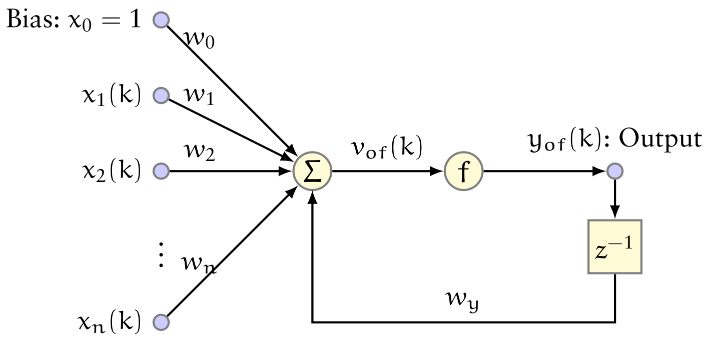


Figure 2.5: Model of a neuron with one-step-delay output feedback.

its first and second derivatives well defined. A typical choice that satisfies these conditions is *sigmoid* class of nonlinear functions. We shall discuss more about activation functions when we describe learning algorithms.

2.1.1 Locally Recurrent Globally Feedforward Networks

There are two very immediate forms of adding feedback to the general model of a neuron. One form is to feed the activation back to the summation block (*activation feedback*) and the other is to feed the neuron output back to the summation block (*output feedback*). They are illustrated in Figures 2.4 and 2.5 and mathematically described by

$$\begin{aligned} v_{af}(k) &= \mathbf{w}^T\mathbf{x} + w_v v_{af}(k-1) \\ y_{af}(k) &= f(v_{af}(k)) \end{aligned} \tag{2.2}$$

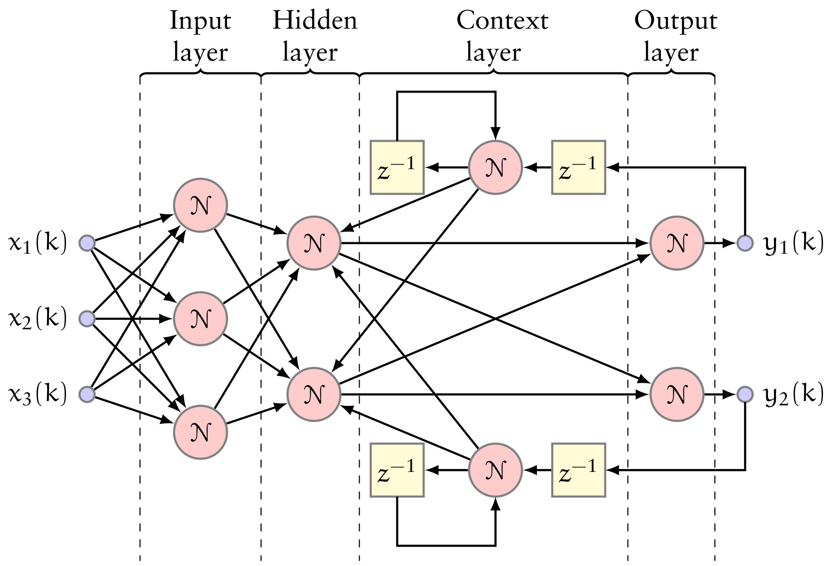


Figure 2.6: An example of Jordan recurrent neural network.

and

$$\begin{aligned} v_{of}(k) &= \mathbf{w}^T\mathbf{x} + w_y y_{of}(k-1) \\ y_{of}(k) &= f(v_{of}(k)), \end{aligned} \tag{2.3}$$

respectively, where subscripts $_{af}$ and $_{of}$ represent activation feekback and output feedback, respectively. Note that both these two forms utilize one step delay feedback.
Mandic [57] generalizes this into N step delay feedback by introducing a simple *linear dynamical system* which is simply made of banks of multipliers and delays. In his version, the input signal is scalar (however it is straightforward to have a vector input signal). Therefore, the activation feedback and output feedback for a neuron will have the following forms:

$$\begin{aligned} v_{af}(k) &= \sum_{i=0}^{M} w_{x,i}x(k-i) + \sum_{j=1}^{N} w_{v,j}v_{af}(k-j) \\ y_{af}(k) &= f(v_{af}(k)) \end{aligned} \tag{2.4}$$

$$\begin{aligned} v_{of}(k) &= \sum_{i=0}^{M} w_{x,i}x(k-i) + \sum_{j=1}^{N} w_{v,j}y_{of}(k-j) \\ y_{of}(k) &= f(v_{of}(k)). \end{aligned} \tag{2.5}$$

We notice that both forms of these feedbacks are local. A network made by neurons with local feedback is called *Locally Recurrent Globally Feedforward*

Network(LRGF). Two early forms of LRGFs were proposed by Jordan [44] and Elman [27]. Jordan network is a traditional three-layer feed-forward network with a set of context units that mirror the output layer activation and feed it back into the hidden layer. Jordan refers to the contexts as the *states* of the system. A simple example of Jordan network is illustrated in Figure 2.6. Elman network, also known as *Simple Recurrent Network*(SRN), is an LRGF but may also be categorized in *state-space models* (see next). Since the concept of *state* in SRN is better illustrated, we will describe Elman network in state-space models.

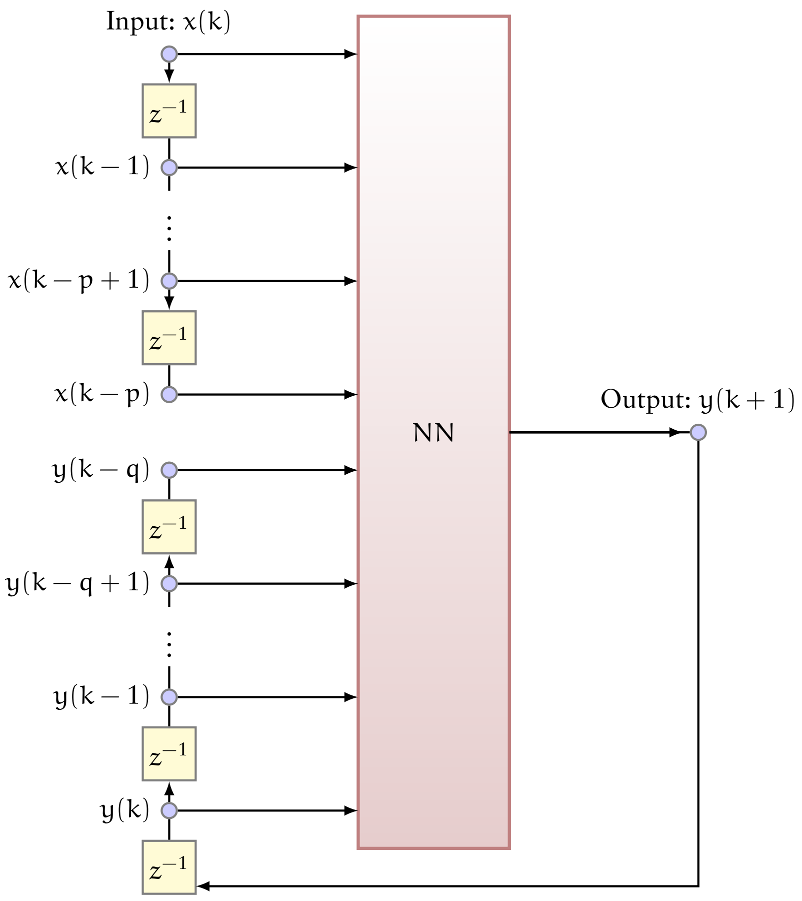


Figure 2.7: Nonlinear autoregressive with exogenous inputs (NARX) model.

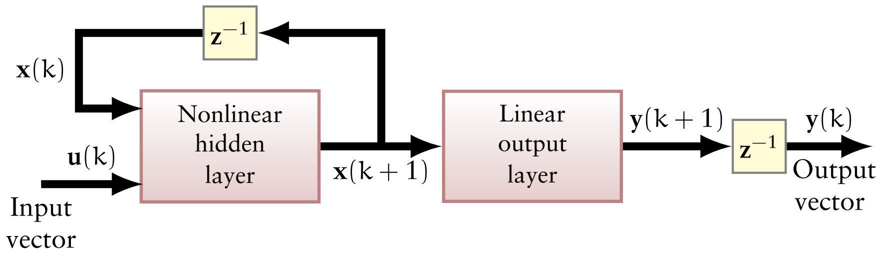


Figure 2.8: State-space model

2.1.2 Input-Output Model

Another famous network especially in system identification community is called *Nonlinear Autoregressive with Exogenous Inputs* (NARX). It usually has a single input which is passed through an array of delays. The network is an MLP which its output, passed through another array of delays, along the delayed version of input signal forms its input. This model can be described by Eq. (2.6). The architecture is shown in Figure 2.7.

$$y(k+1) = NN(y(k), y(k-1), ..., y(k-q); x(k), x(k-1), ..., x(k-p)) \quad (2.6)$$

Note that NN represents a multilayer network of perceptrons.

2.1.3 State-Space Models

Another generic architecture for RNNs is called *State-Space Model*. In this model, hidden neurons define the *state* of the network. Outputs of hidden neurons are fed back into the input layer through a bank of delays (Figure 2.8). The number of unit delays used to feed the output of the hidden layer back to the input layer determines the *order* of the model. Therefore, number of states is equal to number of those neurons whose outputs are fed beck into the input layer, not the number of all hidden neurons, however, these two may be the

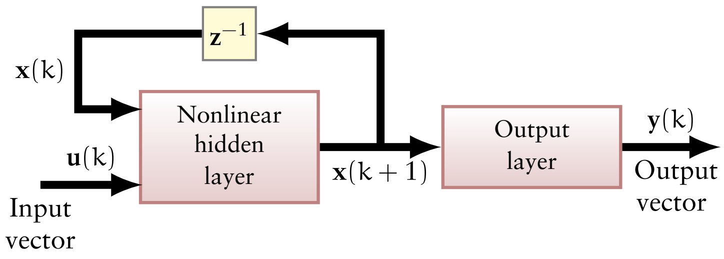


Figure 2.9: Elman network: Simple Recurrent Network (SRN)

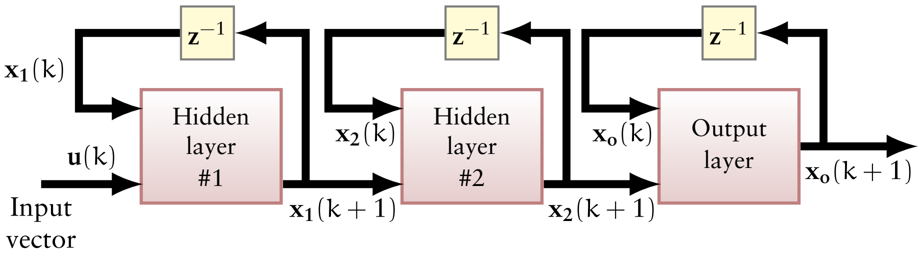


Figure 2.10: Recurrent Multilayer Perceptron with two hidden layers

same. This model is more interesting for us since the architecture to be proposed in this thesis is based on this model.
Note that here the output layer is a linear combination of states. This model is best represented by

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{f}(\mathbf{x}(k), \mathbf{u}(k)) \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k). \end{aligned} \tag{2.7}$$

As previously mentioned, Elman network (SRN) is a special case of state-space model. Its architecture is shown in Figure 2.9. The main difference between SRN and the default state-space model is that the output layer of SRN is not necessarily linear. State transition equation remains the same but output can now be a nonlinear function of states.

2.1.4 Recurrent Multilayer Perceptron

Subsuming Elman and state-space model yields *Recurrent Multilayer Perceptron*(RMLP) [68]. It is basically made of one or more hidden layers each of which has a feedback loop around it. Figure 2.10 illustrates a two hidden layer version of RMLP. In general, if an RMLP has M hidden layers with the same amount of neurons in every layer, the network equations become

$$\begin{aligned} \mathbf{x}_1(k+1) &= \mathbf{f}_1(\mathbf{x}_1(k), \mathbf{u}(k)) \\ \mathbf{x}_2(k+1) &= \mathbf{f}_2(\mathbf{x}_2(k), \mathbf{x}_1(k)) \\ &\vdots \\ \mathbf{x}_M(k+1) &= \mathbf{f}_M(\mathbf{x}_M(k), \mathbf{x}_{M-1}(k)), \end{aligned} \tag{2.8}$$

where $\mathbf{x}_M(k+1)$ is the actual network output. $\mathbf{f}_i$ $(i = 1, 2, ..., m)$ denote the activation vector functions characterizing the first hidden layer, second hidden layer,..., and output layer of the network, respectively. It can be shown that the above RMLP can be represented in a compact form as follows [36]

$$\begin{aligned} \mathbf{x}(k+1) &= \boldsymbol{\phi}(\mathbf{x}(k), \mathbf{u}(k)) \\ \mathbf{y}(k) &= \mathbf{g}(\mathbf{x}(k), \mathbf{u}(k)), \end{aligned} \tag{2.9}$$

where $\boldsymbol{\phi}$ and $\mathbf{g}$ are two nonlinear vector functions that depend on the activation functions of hidden layers. This representation is crucial for us and we will return to this in the next chapter.

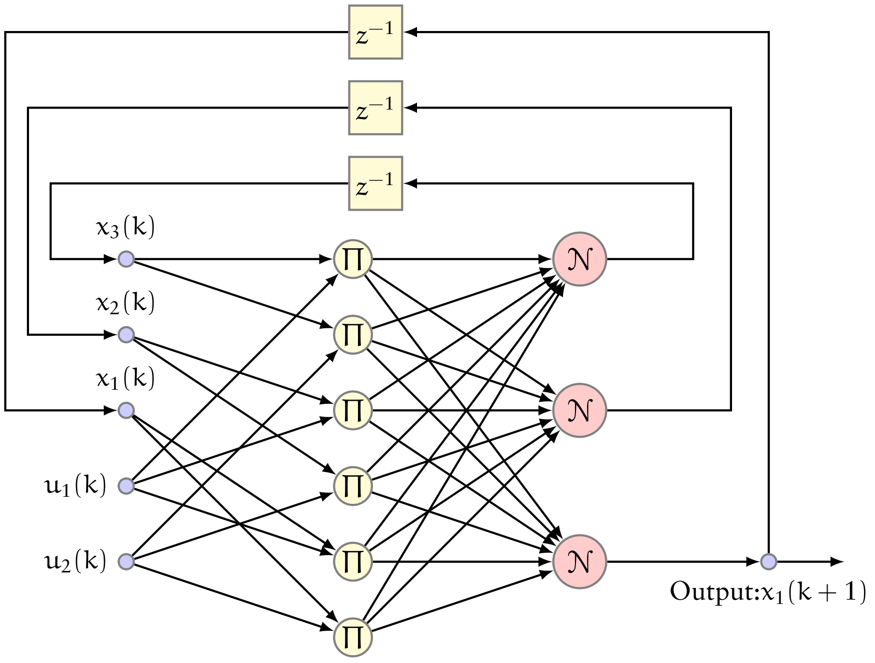


Figure 2.11: A simple second order RNN with two inputs and three states

2.1.5 Second-Order Network

So far, all the described networks receive a summation of weighted input signals in their input layers. Second-order network replaces this summation with multiplication operator. Note that in this context *order* is not the same as what we mentioned in state-space models. To be more precise, recall that in previous networks, for example in an RMLP, net input of neuron l is defined by

$$v_l = \sum_j w_{a,lj} x_j + \sum_i w_{b,li} u_i, \tag{2.10}$$

where x_j is the feedback signal from hidden neuron j and u_i is the ith element of input vector; the w's represent corresponding weights. This is a first order neuron. If the above combination is replaced by multiplication, we reach to a *second-order neuron*

$$v_l(k) = b_l + \sum_i \sum_j w_{lij} x_i(k) u_j(k), \tag{2.11}$$

where k represents time and b_l stands for bias. Figure 2.11 illustrates a very simple form of a second-order network. Note that in this Figure bias connections are omitted for the sake of simplicity. A further generalization may be to connect all the inputs and states to all the multipliers. But we will not discuss it here. Second-order networks are unique in the sense that they can represent state transition. This makes them an immediate candidate for representing and learning *deterministic finite-state automata*(DFA)[5] [36].

2.1.6 Fully Connected Recurrent Neural Network

Fully Connected Recurrent Neural Network(FCRNN) also known as Williams-Zipser network [85] is one of the most interesting networks. As the name implies, in this architecture, all neurons are connected to all other neurons and themselves. This network consists of three layers: input layer, processing layer and output layer. Figure 2.12 illustrates the architecture of this network for scalar input. At a time instant k, for the ith neuron there are $n + m + 1$ weights that can be arranged in a one dimensional vector

$$\mathbf{w}_i^T = [w_{i,1}(k), \ldots, w_{i,n+m+1}].$$

Moreover, all inputs to the same neuron can be formed into another one dimensional vector with the same magnitude

$$\mathbf{u}_i^T(k) = [s_1(k), \ldots, s_n(k), x_1(k), \ldots, x_m(k), 1],$$

where the last constant term represents bias. Thus, dynamic equations of the network can be shown by

$$\begin{aligned} x_i(k+1) &= f(\mathbf{w}_i^T \mathbf{u}_i(k)) \quad i = 1, ..., m \\ y(k) &= \mathbf{x}_1(k). \end{aligned} \tag{2.12}$$

There are some other architectures for RNN reported in literature, such as LSTM (Long-Short Term Memory). However, for our discussion the presented introduction suffices. For more discussion consult [8, 73, 72] and similar. It should be mentioned that the scheme we will propose in Chapter 3 (RSI and LRSI) are based on FCRNN and State-Space Models. Next we will introduce main approaches to learning in RNN.

[5]A DFA is an information processing device with a finite number of states.

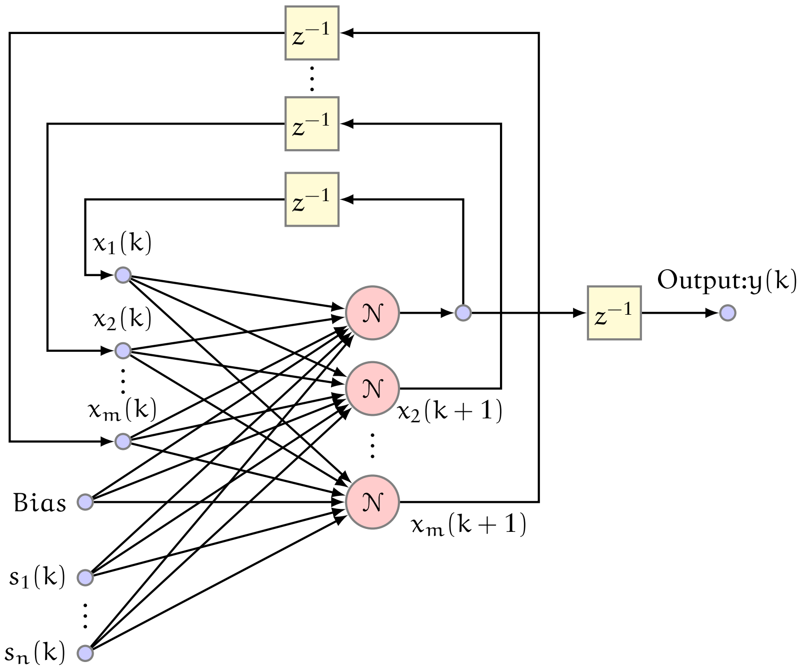


Figure 2.12: An example of FCRNN with n inputs, m hidden neurons and one neuron in output layer.

2.2 Learning Algorithms

Unlike FFNN, learning in RNN is mostly case based since that architecture of the network greatly influence the learning method. However, there are two main approaches to derive learning algorithms for RNN. These are mainly called *Back Propagation Through Time* (BPTT) and *Real Time Recurrent Learning* (RTRL). The former is an immediate result of applying the well known *Back Propagation* (BP) algorithm to an *unfolded* version of the network and the later is originally proposed by Williams and Zipser [85]. We will study both in this section.

2.2.1 Preliminaries

Before we dig into learning in RNN, we need to set the boundaries and assumptions for our discussions. There are plenty of useful articles and books on statistical and machine learning one can consult that cover all aspects of learning (for example [5, 14, 15, 36, 37, 67] and similar). However, the following short topics are chosen in a way to highlight our path through learning in neural networks. Therefore, the following discussions shall clarify our assumptions for and scope of learning in neural networks. References are cited for more details.

Learning in General

Haykin [36] defines learning in the context of neural networks as:

> *"Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place."*

A formal definition for statistical learning is given in [38]. However, we adopt Haykin's definition for learning in neural networks. From a common viewpoint, learning in artificial systems is categorized into the following types:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

The fundamental difference between these types is the presence or absence of a *teacher* signal [36]. In supervised learning, there exists a teacher who knows, at least for a set of samples, what should be the output signal for a specific given input signal.This set is called the *training set*. The training set consists of tuples of input signal(s) along corresponding desired output signal(s). System identification can be categorized as supervised learning where the system to be identified plays the teacher's role. In unsupervised learning, however, there is no teacher. The learner is simply faced with a number of data points in an n-dimensional space and tries to discover regularities within them. The result of learning in this type is a number of categories which the learner assigns data to them. Therefore, unsupervised learning is often called *clustering*. A third type is Reinforcement Learning (RL). In reinforcement learning, there is still no teacher engaged in the learning process, however, there exists an *evaluative feedback* that qualitatively informs the learner of how well it performs. Quite often RL takes place in a *trial-and-error* fashion where the learner interacts with an environment and tries to improve its performance. The evaluative feedback is also called *reinforcement signal*. For a detailed discussion on RL and methods to solve it consult [8, 76].

Error-Correction Learning

The solution to a learning problem is a set of well-defined rules called *learning algorithm* [36]. As expected, there is no unique learning algorithm for all types of neural networks. One of the most frequently used algorithms is based on *Error-Correction rule*. Based on this rule, an *error* signal is defined as

$$e(k) = y^t(k) - y(k), \tag{2.13}$$

where k is an index of time, y^t is the desired response and y is the actual response. In Error-Correction Learning, the aim is to adjust the network synaptic weights to decrease the error signal. In an on-line version, the adjustments are done in a step-by-step manner. This objective is usually achieved by minimizing a *cost function* (or *index of performance*). The cost function is a function of the error signal. One of the most famous and commonly used cost functions is SSE (*Sum of Squared Errors*) to be presented later in this chapter.

Function Approximation

Function approximation is a *learning task* in which we want to approximate an unknown input-output mapping by a neural network. Consider a non-linear input-output mapping

$$\mathbf{d} = \boldsymbol{\phi}(\mathbf{x}), \tag{2.14}$$

where $\boldsymbol{\phi}(.)$ is an unknown vector function,$\mathbf{d}$ and $\mathbf{x}$ are m-dimensional output and n-dimensional input of the unknown mapping, respectively. Assume that we are given a set of observations of the unknown mapping. These observations form the so called *training set* and we will refer to it by D. Each member of this set, i.e., each sample, is a tuple of an n-dimensional input vector $\mathbf{x}_i^t$ and an m-dimensional output vector $\mathbf{y}_i^t$. Thus, it is convenient to show D by

$$\begin{aligned} D &= \{(\mathbf{x}_i^t, \mathbf{y}_i^t)\} \quad i = 1, 2, ..., N \\ \mathbf{y}_i^t &= \boldsymbol{\phi}(\mathbf{x}_i^t), \end{aligned} \tag{2.15}$$

where N is the number of samples. Note that the inputs and outputs may be a function of time and/or any other quantity.
Now, suppose we want to approximate unknown function $\boldsymbol{\phi}(.)$ by a neural network. Usually, it is convenient to construct the neural network having the same input and output dimensions as the unknown function. Let the relation between the network input(s) and output(s) be determined by a nonlinear (continuous) vector function $\mathbf{f}$, i.e.,

$$\mathbf{y}_i = \mathbf{f}(\mathbf{x}_i, \mathbf{w}), \tag{2.16}$$

where $\mathbf{w}$ is the set of the network synaptic weights. The goal of function approximation is to determine $\mathbf{w}$ in a way that the neural network approximates the unknown mapping such that $\mathbf{f}(.)$ is close enough to $\boldsymbol{\phi}(.)$ for all x, in an Euclidean sense

$$||\mathbf{f}(\mathbf{x}) - \boldsymbol{\phi}(\mathbf{x})|| < \varepsilon \quad \forall \mathbf{x}, \tag{2.17}$$

where ε is a small positive number. An obvious example of function approximation is *system identification*. Note also that the type of learning in function approximation (and system identification) is error-correction learning.

Unconstrained Optimization

Consider a cost function $\psi(\boldsymbol{w})$ which is a continuously differentiable function of some adjustable parameters $\boldsymbol{w}$ (e.g. synaptic weights in a neural network). The goal of optimization is to find a $\boldsymbol{w}^*$ such that

$$\psi(\boldsymbol{w}^*) \leqslant \psi(\boldsymbol{w}) \quad \forall \boldsymbol{w}. \tag{2.18}$$

A necessary condition for optimality is

$$\nabla\psi(\boldsymbol{w}^*) = 0, \tag{2.19}$$

where ∇ is the *gradient operator*

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial w_1} & \cdots & \frac{\partial}{\partial w_m} \end{bmatrix}^T. \tag{2.20}$$

The cost function often has a non-linear form with respect to the adjustable parameters. Thus, an effective method to minimize it is to iteratively explore the parameter space, in an efficient way, to find the optimal ones ($\boldsymbol{w}^*$). These methods are called *iterative descent* methods. Principally, at each iteration, parameters are slightly adjusted to decrease the cost function. This adjustment is in the following form

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \eta\mathbf{l}, \tag{2.21}$$

where η is some positive *step size*. The adjustment is done in a way that the cost function is decreased (or at least not increased) after each adjustment

$$\psi(\boldsymbol{w}(k+1)) \leqslant \psi(\boldsymbol{w}(k)). \tag{2.22}$$

We see that each adjustment involves two main steps: determination of the direction of adjustment ($\mathbf{l}$) and determination of the step size (η). Those methods which utilize derivatives of the cost function in determination of $\mathbf{l}$ are called *derivative-based optimization techniques* such as *Steepest-Descent* and *Newton's Method*. Among them are *gradient-based* methods by which $\mathbf{l}$ is determined on the basis of the gradient of ψ, such as *Steepest-Descent*. We will not cover them all in this thesis; a number of useful resources exist in the literature such as [5, 15, 36, 41, 67, 86]. Only a modified version of Newton's method known as *Levenberg-Marquardt* method will be discussed in the next chapter.

Backpropagation Algorithm

In the context of multilayer neural networks (MLPs), backpropagation algorithm (or BP) is frequently applied where supervised learning is desired. Essentially, BP is the same as Steepest-Descent method. However, in MLPs, output error for hidden neurons are not directly computable because desired values for output of hidden neurons are usually not available. BP algorithm utilize the famous method of *chain rule* to *back-propagate* the error, obtained by comparing network output(s) with desired ones, to the intermediate and input layers. If an error vector is available for a neuron, whether it be an output, hidden or input neuron, it is very likely that any error-based update scheme is applicable. For a comprehensive discussion on BP consult [36] Chapter 4, and [15] Chapter 5.

Modes of Training

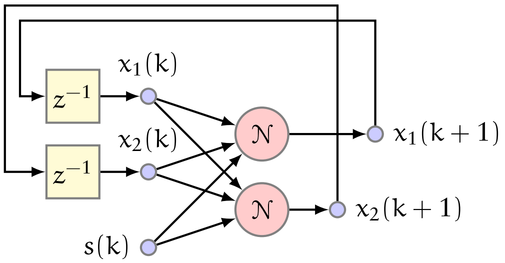
From a very general point of view, there are two modes of training RNN: *epochwise* and *continuous*. In epochwise training, for each given epoch, the recurrent network starts running from some initial state until it reaches a new state, at which point the training is stopped [36]. The important point in this mode is that each initial state should be different from the state reached by the network at the end of the previous epoch, however, it can be the same as the initial state of previous epoch. On the other hand, in continuous training the network is not reset to any initial state. This mode is suitable when an on-line learning is required and/or there are no reset states available. The distinguishing feature of this mode of training is that the network learns while being used. This feature makes this mode suitable for a number of applications such as signal processing.
Another mode of training which is not directly related to the flow of the learning process is called *teacher forcing*. Suppose that an RNN is to be trained by a continuous supervised learning method and suppose that each network output will be used as its initial condition for the next time step, entirely or partially, the case that is shown in Figure 2.14. Then, we can replace the output of the network with the actual desired output. This is called teacher forcing. In the next chapter, where we are talking about system identification with RNN we will return to this discussion.

Unfolding in Time

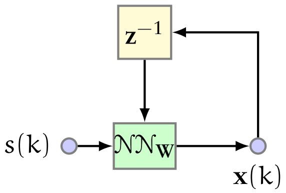
A very common approach to derive learning rules (also called *weight update rules*) in RNNs is first to *unfold* the network *in time*. This means that we try to transform the recurrent network into a feedforward network with *shared weights* by replicating the network in time. This unfolding process can be both done backward and forward in time, however, for now we only consider unfolding back in time. Let us explain this unfolding process by an example. Consider a very simple FCRNN with two input signals, one hidden neuron and one output neuron. The detailed equations of the network are

$$\begin{aligned}\mathbf{W} &= \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \\ \mathbf{u}(k) &= \begin{bmatrix} x_1(k) & x_2(k) & s(k) \end{bmatrix}^T \\ \mathbf{x}(k+1) &= \mathbf{f}(\mathbf{W}\mathbf{u}(k)).\end{aligned} \tag{2.23}$$

In this equation, **W** is the network weight matrix and consists of all the weights in the network. Noticeably, the elements of the first row of this matrix are synaptic weights connected to the first neuron (the upper one illustrated in Figure 2.13a). Similarly, the second row corresponds to the second neuron. In Figure 2.13b, a compact illustration for the sample network is shown where subscription $_{\mathbf{W}}$ is to remind us about the network weight matrix. Now let us



(**a**) A simple FCRNN



(**b**) The equivalent form

Figure 2.13: FCRNN used in example to describe unfolding back in time

unfold this network back in time for a limited number of time steps, say m^-. Bear in mind that each replication of the network in past uses the weight values at present. This is the concept of *shared weights*. The unfolded network is illustrated in Figure 2.14.

In this Figure, the whole unfolded network is a feedforward network which we know all the intermediate inputs (i.e., $s(t)$ for $t = k - m^-, ..., k$) and the initial condition which is $\mathbf{x}(k - m^-)$.

2.2.2 Back-Propagation Through Time

Back-Propagation Through Time or BPTT is one of the most commonly used method for training RNN. As the name implies, to use this method, one first needs to unfold the network back in time either an infinite or finite number of time-steps. As discussed earlier, this procedure yields a feedforward neural network where standard BP is applicable, bearing in mind that the weight val-

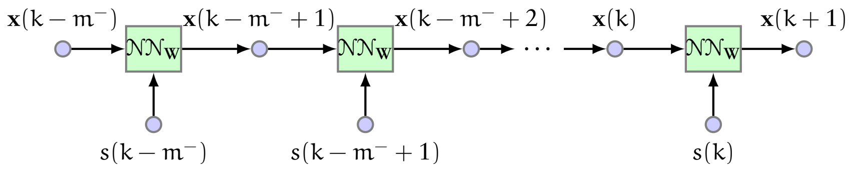


Figure 2.14: The sample FCRNN unfolded back in time for m^- time steps with shared weights

ues are replicated over time. To be more specific, we first need to define a cost function. Quite often the cost function is SSE

$$\begin{aligned} \boldsymbol{e}(k) &= \boldsymbol{y}^t(k) - \boldsymbol{y}(k) \\ \psi(t_0, t) &= \frac{1}{2}\sum_{k=t_0}^{t} (\boldsymbol{e}^T(k)\boldsymbol{e}(k)), \end{aligned} \tag{2.24}$$

where $\boldsymbol{y}(k)$ is the network output at time instance k, $\boldsymbol{y}^t(k)$ is the desired value for $\boldsymbol{y}(k)$ and $(t_0, t]$ is the duration in which the original network is unfolded. Two main versions of BPTT exists: *epochwise* BPTT and *Truncated* BPTT.

Epochwise Back-Propagation Through Time

In epochwise BPTT, at the beginning of each epoch, the state of the network, i.e., initial conditions, are set to either a random state, a reset state, or any other state other than the final state of the previous epoch. Then, a forward pass of the data through the network is performed for the whole interval $(t_0, t]$. This is done by feeding a sample input and record all the data within the network for the whole mentioned interval. After reaching to the final state, local gradients are computed using

$$\delta_j(k) = -\frac{\partial\psi(t_0, t)}{\partial v_j(k)} \tag{2.25}$$

and

$$\delta_j(k) = \begin{cases} f'(v_j(k))e_j(k) & \text{for } k = t \quad (2.26a) \\ f'(v_j(k))[e_j(k) + \boldsymbol{w}_j^T\delta(k+1)] & \text{for } t_0 < k < t. \quad (2.26b) \end{cases}$$

Note that in the above equations, we attempt to compute the local gradient of the neuron j. Therefore, Eqs. (2.26) imply that it is only at time $k = t$ that a desired value for the output of this neuron is available. For all intermediate time instances, we use the back-propagation to compute the (local)errors. This also implies that computation of local gradients is done backward, i.e., from $k = t$ back to $k = t_0 + 1$, hence the name BPTT. Bear in mind that at time

$k < t$, $\boldsymbol{\delta}(k+1)$ is a vector of local gradients of all the neurons connected to jth neuron that are already computed. Similarly, $\boldsymbol{w}_j$ is the vector of corresponding synaptic weights. f' is the derivative of the neurons activation function. It is assumed that all the neurons have the same activation function.
After computing local gradients for the whole interval, synaptic weight w_{ji} of the neuron j is adjusted according to the famous BP formula

$$\begin{aligned}\Delta w_{ji} &= -\eta \frac{\partial \psi(t_0, t)}{\partial w_{ji}} \\ &= \eta \sum_{k=t_0+1}^{t} \delta_j(k) x_i(k-1),\end{aligned} \tag{2.27}$$

where η is a learning rate and $x_i(k-1)$ is the input applied to the i^{th} synapse of neuron j at time $k-1$. The algorithm is shown in Algorithm 1.

Truncated Back-Propagation Through Time

To use BPTT in a real-time fashion, instead of accumulating the errors, we use the instantaneous error as the cost function

$$\psi(k) = \frac{1}{2}\boldsymbol{e}^T(k)\boldsymbol{e}(k). \tag{2.28}$$

Similarly, we use the negative gradient of the above cost function. However, since the weights are updated on-line, we need to *truncate* the length during which the relative data is saved. This length is called *truncation depth* and here is denoted by l_t and the algorithm is called *BPTT*(l_t). All the formulas remain the same except for the fact that t_0 is substituted by l_t. Some practical considerations should be taken care of which are discussed in [36].
BPTT works fine for small networks and small intervals. But it is readily seen that by increasing the number of neurons and/or prolonging the unfolding interval, Eqs. (2.26) becomes intractable and computationally expensive. Werbos [84] proposes a procedure in which each expression in the forward propagation of a layer gives rise to a corresponding set of back-propagation expressions. However, RTRL is another effective approach for on-line learning in RNNs, which will be discussed next.

2.2.3 Real-Time Recurrent Learning

Real-Time Recurrent Learning (or RTRL) was originally proposed by Ronald J. Williams and David Zipser in 1989 [85]. It coincides with an approach suggested in the system identification literature by McBride and Narendra [61]. Also, Robinson and Fallside had given an alternative description of the algorithm in 1987 [69]. RTRL algorithm is suitable when we want to perform

Algorithm 1 Algorithm for epochwise Back-Propagation Through Time for a typical FCRNN, Figure 2.12. One epoch is illustrated.

```
Require: A training sample $(\mathbf{s}^t, y^t)$.
Require: An interval $(t_0, t]$.
Ensure: BPTT weight update values.
     Forward propagation:
 1:  $k \leftarrow t_0$: Reset the network to a random (or reset) state.
 2:  $k \leftarrow t_0 + 1$: Apply the sample $\mathbf{s}^t$.
 3:  for all $k = t_0 + 1, ..., t$ do
 4:      Compute and record network states (i.e., $x_i(k)$, $i = 1, ..., m$) using Eqs. (2.12).
 5:  end for
 6:  Compute network output at $k = t$: $y(t) = x_1(t)$.
     Backward propagation:
 7:  Compute local gradient for the output neuron using Eq. (2.26a) with $j = 1$.
 8:  $\delta_j(t) \leftarrow 0$ for $j = 2, ..., m$.
 9:  $k \leftarrow (t - 1)$
10:  for all $k = t - 1, ..., t_0 + 1$ do
11:      $\delta_j(k+1) \leftarrow \begin{bmatrix}\delta_1(k+1) & \delta_2(k+1) & \dots & \delta_m(k+1)\end{bmatrix}^T$.
12:      Compute local gradient for the jth neuron using Eq. (2.26b).
13:  end for
     Weight update:
14:  for all $j = 1, ..., m$ do
15:      for all $i = 1, ..., m$ do
16:          Update synaptic weight $w_{ji}$ using Eq. (2.27).
17:      end for
18:  end for
```

learning on the network while continuously running it. In the original description, RTRL is formulated for an FCRNN with arbitrary number of neurons and input lines. However, the concept is applicable to a number of other architectures.

Refer to Figure 2.12 to recall a typical FCRNN. Assume an extension of this network to have m outputs, i.e., each neuron has an output, so network states are equal to network outputs. To show how RTRL works, let us first concatenate inputs ($\mathbf{s}(k)$) and outputs ($\mathbf{y}(k)$) to form the $(m+n)$-tuple $z(k)$. Thus, if I is the set of input indexes and O the set of output indexes then

$$z_i(k) = \begin{cases} s_i(k) & \text{if } i \in I \\ y_i(k) & \text{if } i \in O. \end{cases} \tag{2.29}$$

In a similar manner we can arrange all synaptic weights, that exist between all neurons in the network, into a $n \times (m+n)$ weight matrix[6] $\boldsymbol{W}$. Since the network is fully connected, the net input to each unit at time k is

$$v_i(k) = \sum_{l \in U \cup I} w_{il} z_l(k) \tag{2.30}$$

and their outputs at the next time step are

$$y_i(k+1) = f_i(v_i(k)), \tag{2.31}$$

where i ranges over U. Equations (2.29), (2.30) and (2.31) define the dynamics of the network for which the RTRL algorithm is presented next.
Let $T(k)$ denote the set of indexes of neurons for which a desired target value $y_i^t(k)$ exists at time k. Then the error signal $e_i(k)$ is

$$e_i(k) = \begin{cases} y_i^t(k) - y_i(k) & \text{if } i \in T(k) \\ 0 & otherwise. \end{cases} \tag{2.32}$$

The instantaneous error of the network is

$$E(k) = \frac{1}{2} \sum_{i \in U} e_i^2(k). \tag{2.33}$$

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

$$\psi(t_0, t) = \sum_{k=t_0}^{t} E(k). \tag{2.34}$$

RTRL is based on the gradient descent algorithm and adjusts the weights along the negative of $\nabla_{\boldsymbol{W}} \psi(t_0, t)$. Thus, for both above cost functions (i.e., equations (2.33) and (2.34)) we need to compute the partial derivative of $E(k)$ with respect to individual weights at time k, i.e., $w_{ij}(k)$:

$$\begin{aligned} \Delta w_{ij}(k) &= -\eta \frac{\partial E(k)}{\partial w_{ij}} \\ &= \sum_{l \in U} e_l(k) \frac{\partial y_l(k)}{\partial w_{ij}}, \end{aligned} \tag{2.35}$$

where η is a fixed positive learning rate.

$$\frac{\partial y_l(k)}{\partial w_{ij}} = f_l'(s(k)) \left[\sum_{l \in U} w_{lp} \frac{\partial y_p(k)}{\partial w_{ij}} + \delta_{il} z_j(k) \right], \tag{2.36}$$

[6]Bias can be already included in the input vector, so the number of real inputs are $n-1$ and we have a fixed input 1 playing the bias role.

where δ_{il} denotes the Kronecker delta. Assuming the initial state of the network has no functional dependence on the weights we have

$$\frac{\partial y_l(t_0)}{\partial w_{ij}} = 0. \tag{2.37}$$

Therefore, equations (2.36) and (2.37) constitute a recursive formula to compute $\frac{\partial y_l(k)}{\partial w_{ij}}$ and using Eq. (2.35) the RTRL weight update rule is obtained.
In case of the cumulative cost function (Eq. (2.34)), one need to sum up all the individual updates $\Delta w_{ij}(k)$ over the interval(t_0, t) and the final weight update becomes

$$\Delta w_{ij} = \sum_{k=t_0}^{t} \delta w_{ij}(k). \tag{2.38}$$

Having introduced different architectures of RNN and the two basic learning algorithms, next we will focus on the architecture of our interest, State-Space Model. We will propose a general state-space model that encompasses several different architectures. We will also propose a learning algorithm which is partially similar to RTRL.

Chapter 3
System Identification with Recurrent Neural Networks

In this chapter we are going to investigate the ability and effectiveness of Recurrent Neural Networks in system identification. In Chapter 2, section one, a brief introduction to system identification was presented. We mentioned that system identification in the context of neural networks can be classified as function approximation. It has been proven that MLPs with at least one hidden layer are *Universal Approximators* [22]. It means that they can approximate any continuous function arbitrarily closely provided sufficient amount of nonlinear neurons[1] in the hidden layer. In principle, an FCRNN, unfolded back in time, is a MLP. Thus, the universal approximation capability of a MLP can be inherited to FCRNN. This is the basis on which some authors have derived universal approximation property of their suggested recurrent scheme, for example [72]. Moreover, this is one of the fundamental reasons why neural networks are so popular in system identification. We have already claimed that RNN are very good in identifying time dependencies. Therefore, when coping with systems for which time plays a significant role, they are considerable candidates for system identification.
In this chapter, after reviewing a short history on exploiting recurrent schemes in system identification, we will present the main contribution of this thesis: *Recurrent System Identifier* (RSI). RSI is the result of studying various RNN schemes used in system identification and generalizing over a number of them. We will show that a comprehensive class of system identifiers that employe RNNs can be represented by RSI. Like any other learning system, RSI has an architecture and a learning method. Learning in RSI is done by incorporating a version of *Levenberg-Marquardt* (LM) algorithm, to be derived in this chapter. It has been frequently shown that LM algorithm is one of the most effective learning method in non-linear least squares problems. In Chapter 5, we will

[1]Neurons with nonlinear activation functions.

experimentally study the ability of RSI in system identification.

3.1 Introduction

One of the earliest attempts to exploit a recurrent scheme in system identification dates back to 1988 [83]. In his work, Werbos attempted to model a gas market using BP methods in a recurrent scheme. One year later, Pearlmutter [64] published his work on using a recurrent scheme to learn state-space trajectories. His work was based on Pineda's generalization of backpropagation to RNN [66]. In the same year Williams and Zipser published their pioneering RTRL algorithm [85] on which, later, a number of on-line identifications started to emerge. In 1994, Srinivasan et al. published their work on using Back-Propagation in RNN to identifying non-linear dynamic systems [75]. They provided a *Recurrent State Model* which is a special case of what we propose as RSI in this thesis. However, they used an adjoint model to compute the gradient. One year later, in an interesting work, Delgado et al. introduced their version of using RNN in system identification [26]. They mentioned their work as a *generalization* over *Hopfield Model*[2]. They also provided a profound proof for approximation capability of their proposed RNN architecture for a general class of non-linear systems. In their work, states of the network are a weighted sum of inputs, states and a non-linear mapping of states (Equation 7 in [26]) which still can be included in our RSI scheme as a special case. In the same year, Chao et al. proposed a Diagonal RNN (DRNN) for identification and control of non-linear systems [49]. Their DRNN is basically an LRGF network. They still used the BPTT algorithm to train their network. To the author's best knowledge, the first usage of LM algorithm in a recurrent scheme was reported in 1999 where - in a short article by Chan and Szeto - a Recurrent Network was trained with LM algorithm [17]. However, their article is not enough descriptive in terms of architecture that they might have used. Since the late 90s and early 00s, researchers have been attempting to combine RNN with several other approaches such as Fuzzy Systems and Wavelet networks [16, 19, 31, 33, 45, 50, 52, 60].
In 2000, Tan and Saif reported a practical task of nonlinear dynamic modeling using NARMA model with LM learning algorithm [78].In the same year, Griñó et. al published a work about on-line system identification with continuous RNN [34]. Their proposed architecture was very similar to Delgado et. al's except that the output in the former is a linear combination of internal states. Atia et. al attempted to classify learning algorithms in RNN and tried to generalize over them [6]. Baruch et al. proposed a RNN with linear internal

[2]Hopfield Model is one of the earliest neural networks with recurrency within the network. However, Hopfield nets serve as *content-addressable memory systems*. Since their units are binary threshold, their ability to approximate complex functions is so restricted. For more information consult [36, 70]

states but a non-linear output, i.e., states are mapped onto outputs through a non-linear vector function [11]. The learning scheme, however, is still a version of BP.

Until recently, LM algorithm is not very used in parameter learning of recurrent schemes. They have been a few reports for practically using LM in recurrent schemes [12, 24, 35, 59, 78, 79, 97][3] among which the architecture used by Baruch et al. [12] is the most similar to ours and can be regarded as a special case of our RSI. An interesting architecture is also proposed by De Jesus [25] on which later Endisch et al. proposed a LM algorithm on that [28, 29]. Another interesting work for us is a result of Schäfer's and Zimmermann's research [72, 98]. In his PhD thesis, Schäfer attempts to identify a gas turbine with a state-space model of RNN which has been proposed and studied by Zimmermann in [98]. Our architecture is inspired from their work. The next section begins with presenting a brief summary of their proposed architecture.

3.2 Recurrent System Identifier

As described in the previous chapter, one of the famous representations for RNN is the State-Space Representation (SSR). Although fundamentally there is no difference between this representation and others, because of distinguishing internal states - which are basically outputs of hidden neurons - this representation has received specific attention from systems and control researchers. Zimmermann suggest a general form for a SSR of a recurrent neural network system identifier in [98]. His suggestion is illustrated in Eq. (3.1).

$$\begin{aligned} s(k) &= NN_I(s(k-1), u(k); \mathbf{v}) && \text{state transition} \\ y(k) &= NN_O(s(k); \mathbf{w}) && \text{output transition} \end{aligned} \tag{3.1}$$

where:

$s(k)$: Network states at time k

$u(k)$: Input to the network at time k

$y(k)$: Network output at time k

$\mathbf{v}$: Parameter set of the network that corresponds to states

$\mathbf{w}$: Parameter set of the network that corresponds outputs

$NN(i;p)$: A Neural Network structure whose inputs are i and weights are p

Note that since Zimmermann uses a linear NN for output (i.e., NN_O is a linear combination of states at time k) he argues that any other combination of states and outputs is convertible to that of (3.1) [98]. Figure 3.1 illustrates a signal-flow diagram of Zimmermann's network unfolded back in time. In showing

[3]Some of the works that have used LM algorithm are referred to in the next chapter due to the fact that they used their identification in a predictive control scheme.

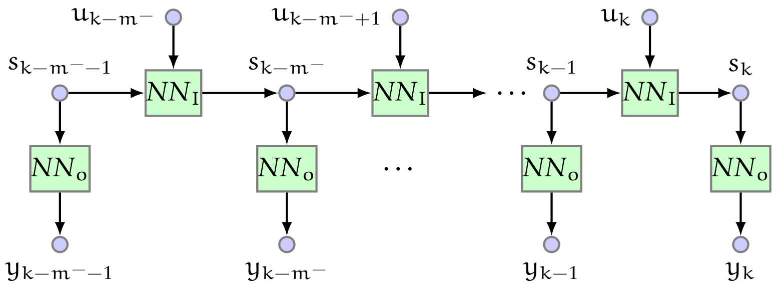


Figure 3.1: Zimmermann's network, unfolded back in time for m^- steps

the signal-flow diagrams, our scheme is very similar to that of Zimmermann's in [98]. Note that in this and subsequent figures, subscripts are time indexes.

Being inspired by Zimmermann's work, we would like to generalize this form to a *Recurrent System Identifier* (RSI). This general form is expressed by Eq. (3.2).

$$\mathbf{x}(k) = NN_I(\mathbf{x}(k-1), \mathbf{u}(k), \mathbf{y}(\mathbf{k}-\mathbf{1}), \mathbf{W}) \tag{3.2a}$$
$$\mathbf{y}(k) = NN_O(\mathbf{x}(k), \mathbf{u}(k), \mathbf{y}(\mathbf{k}-\mathbf{1}), \mathbf{P}) \tag{3.2b}$$

In general, NN_I and NN_O can be any Neural Network with corresponding inputs and weights. However, we are more interested in those networks using a linear combination of inputs, outputs, states and their corresponding weights for two main reasons. Firstly, it is very convenient to represent these networks in state-space form and later will ease stability analysis and other mathematical manipulations. Secondly, according to the *Universal Approximation* property of two layer networks with one nonlinear hidden layer, these networks will also be capable to exhibit the same feature [36, 57, 48]. Therefore, by introducing two arbitrary (continuous) vector functions $\mathbf{f}(.)$ and $\mathbf{g}(.)$ another form of RSI, namely Linear RSI (LRSI), which is more useful in practice is introduced in (3.3).

$$\mathbf{s}(k) = \begin{bmatrix}\mathbf{x}^T(k-1) & \mathbf{u}^T(k) & \mathbf{y}^T(k-1) & 1\end{bmatrix}^T \tag{3.3a}$$
$$\mathbf{r}(k) = \begin{bmatrix}\mathbf{x}^T(k) & \mathbf{u}^T(k) & \mathbf{y}^T(k-1) & 1\end{bmatrix}^T \tag{3.3b}$$
$$\mathbf{x}(k) = \mathbf{f}(\mathbf{W}\mathbf{s}(k)) \tag{3.3c}$$
$$\mathbf{y}(k) = \mathbf{g}(\mathbf{P}\mathbf{r}(k)) \tag{3.3d}$$

Although $\mathbf{f}(.)$ and $\mathbf{g}(.)$ are assumed to be arbitrary, their first and second derivatives must be well defined. Usually in practice $\mathbf{f}$ is a tangent hyperbolic and $\mathbf{g}$ is a simple linear function: $\mathbf{g}(\mathbf{x}) = \mathbf{x}$. In Eqs.(3.3) $\mathbf{x}(k)$ encompasses states of the network at time instance k. States are essentially outputs of the network hidden neurons. Similarly, $\mathbf{y}(k)$ and $\mathbf{u}(k)$ consist of network outputs and inputs, respectively, at time k. $\mathbf{W}$ and $\mathbf{P}$ are weight matrices. States, inputs and outputs are referred to as *signals*. We refer to $\mathbf{s}(k)$ and $\mathbf{r}(k)$ as pseudo-states. Bear in mind that they represent network connections and are not necessarily equal. For example, states may be independent of the network outputs which means network outputs are not fed back into the hidden neurons. In this particular case, $\mathbf{r}(k)$ remains the same but $\mathbf{s}(k)$ becomes:

$$\mathbf{s}(k) = \begin{bmatrix}\mathbf{x}^T(k) & \mathbf{u}^T(k) & 1\end{bmatrix}^T$$

Therefore, one should recall that the structures of pseudo-states are determined in the design stage and accordingly sizes of their corresponding weight matrices vary. We will shortly bring a design example to show the possible difference between $\mathbf{s}(k)$ and $\mathbf{r}(k)$.
Another equivalent form for representing LRSI which will be useful when we attempt to derive learning algorithms is shown in Eqs.(3.4). This form is also more similar to state-space representation.

$$\mathbf{x}(k) = \mathbf{f}(\mathbf{A}\mathbf{x}(k-1) + \mathbf{B}\mathbf{u}(k) + \mathbf{F}\mathbf{y}(k-1) + \boldsymbol{\theta}) \tag{3.4a}$$
$$\mathbf{y}(k) = \mathbf{g}(\mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) + \mathbf{G}\mathbf{y}(k-1) + \boldsymbol{\beta}) \tag{3.4b}$$

Comparing equations (3.3) and (3.4) it is clear that $\mathbf{W}$ and $\mathbf{P}$ are partitioned as shown in Eqs. (3.5):

$$\mathbf{W}_{n_X \times n_S} = \begin{bmatrix}\mathbf{A}_{n_X \times n_X} & \mathbf{B}_{n_X \times n_I} & \mathbf{F}_{n_X \times n_O} & \boldsymbol{\theta}_{n_X \times 1}\end{bmatrix} \tag{3.5a}$$
$$\mathbf{P}_{n_O \times n_R} = \begin{bmatrix}\mathbf{C}_{n_O \times n_X} & \mathbf{D}_{n_O \times n_I} & \mathbf{G}_{n_O \times n_O} & \boldsymbol{\beta}_{n_O \times 1}\end{bmatrix} \tag{3.5b}$$

In Eq.(3.5), $\boldsymbol{\theta}$ and $\boldsymbol{\beta}$ are vectors of biases to neurons. Also, subscriptions indicate sizes and they are listed in (3.6).

$$\begin{aligned} n_X &:\text{Number of states} \\ n_I &:\text{Number of inputs} \\ n_O &:\text{Number of outputs} \\ n_S &= a n_X + b n_I + f n_O + p_1 \\ n_R &= c n_X + d n_I + g n_O + p_2 \end{aligned} \tag{3.6}$$

Note that in (3.6), coefficients a, b, c, d, f, g, p_1 and p_2 may be either 0 or 1 depending on the network architecture. In the design example they will be more explained. But first let us draw signal-flow diagram of LRSI in Figure 3.2. Note that in this figure there is no explicit illustration of pseudo-states. Additionally, blocks that are named by $\mathbf{f}$ and $\mathbf{g}$ internally include a summation which means inputs to them are first summed and then affected by the arbitrary function $\mathbf{f}$ or $\mathbf{g}$.

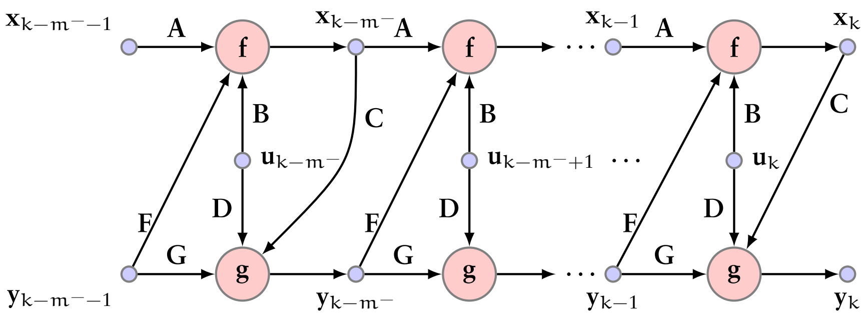


Figure 3.2: LRSI network, unfolded back in time for m^- steps, biases are not shown.

Design Example

Through this example we want to demonstrate how pseudo-states $\mathbf{s}(k)$ and $\mathbf{r}(k)$ are affected in the design stage. For this purpose, we choose an architecture proposed and used by Schäfer in modeling a gas turbine [72]. We want to investigate that if LRSI is formed to have the same architecture, what the form and size of pseudo-states will be.
Equations (4.8) in [72] represent the architecture and using our notation it is shown by the following equations

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{f}(\mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{y}(k) + \mathbf{D}\mathbf{u}(k) - \boldsymbol{\theta}) \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k), \end{aligned} \tag{3.7}$$

where $\mathbf{x}(k)$ and $\mathbf{y}(k)$ are states and outputs, respectively. Matrices **A**, **B**, **C**, **D** and vector $\boldsymbol{\theta}$ are weights with appropriate sizes. Comparing Eqs. (3.7) and (3.3), we can easily design LRSI to have the above form if we use the simple linear $\mathbf{g}(\mathbf{x}) = \mathbf{x}$. To show this version of LRSI, using Eqs. (3.4) together with Eqs. (3.7) and (3.3), we obtain the following pseudo-states for our LRSI:

$$\mathbf{s}(k) = [\mathbf{x}^T(k-1) \quad \mathbf{u}^T(k-1) \quad \mathbf{y}^T(k-1) \quad 1]^T \tag{3.8a}$$
$$\mathbf{r}(k) = [\mathbf{x}^T(k-1)]^T \tag{3.8b}$$
$$\mathbf{x}(k) = \mathbf{f}(\mathbf{W}\mathbf{s}(k)) \tag{3.8c}$$
$$\mathbf{y}(k) = \mathbf{P}\mathbf{r}(k) \tag{3.8d}$$
$$\mathbf{W}_{n_X \times n_S} = [\mathbf{A}_{n_X \times n_X} \quad \mathbf{B}_{n_X \times n_I} \quad \mathbf{F}_{n_X \times n_O} \quad \boldsymbol{\theta}_{n_X \times 1}] \tag{3.8e}$$
$$\mathbf{P}_{n_O \times n_R} = [\mathbf{C}_{n_O \times n_X}] \tag{3.8f}$$

Accordingly, to obtain the above sizes we use Eq. (3.6) with the following coefficients:

$$\begin{aligned} a = b = c = f = p_1 = 1 \\ d = g = p_2 = 0 \end{aligned} \tag{3.9}$$

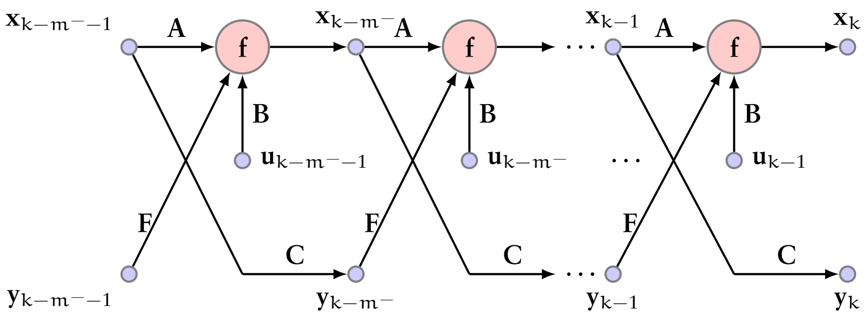


Figure 3.3: Structure of LRSI network obtained from the design example, unfolded back in time for m^- steps.

Figure 3.3 shows the signal-flow diagram of the network, unfolded back in time.

Remark 3.2.1. In this example, output is dependent on the previous states, not on the current states. Although it may imply that we are diverging from our LRSI scheme, it is not so. The only different here is the order of computing output and evolving the derivatives as shall be cleared after the next discussion.

Having defined the structure of LRSI, next we will define a learning algorithm for training the network.

3.3 Learning Algorithm

In this section we intent to derive an effective learning algorithm with update rules for the network weights, particularly **P** and **W** in equation (3.5). It is obvious that general BPTT method can be immediately applied to LRSI because of its recurrent nature. However, we found out through simulation that this method is not efficient both in terms of time, because of its very slow convergence, and in terms of computational resources. Hence we seek for a remedy and this is where Levenberg-Marquardt (LM) algorithm comes into play. Next, we will continue our discussion from the last chapter where we were talking about learning in general and will explain LM algorithm. Then we will derive the LM algorithm for our LRSI architecture.

3.3.1 Levenberg-Marquardt Method

In the non-linear least-squares problem, we wish to optimize a model by minimizing a squared error measure between the target outputs and the actual model's outputs. Assume the model output is described by Eq. (3.10) (it can be output of a neural network with synaptic weights $\mathbf{w}$).

$$y = f(\mathbf{u}, \mathbf{w}) \tag{3.10}$$

For the moment assume that the model's output is scalar, generalization to multi-output is straightforward. $\mathbf{u}$ is the vector of m inputs and $\mathbf{w}$ is the *vector* of n adjustable parameters. Assume we have a training set comprised of N samples (Eq. (2.15)). We use SSE which is the most common cost function in data fitting and nonlinear regression and is shown by

$$\begin{aligned}\psi_{SSE}(\mathbf{w}) &= \sum_{i=1}^{N} (y_i^t - y_i)^2 \\ &= \sum_{i=1}^{N} (y_i^t - f(\mathbf{x}_i^t, \mathbf{w}))^2 \\ &= \sum_{i=1}^{N} r_i^2(\mathbf{w}) \\ &= \mathbf{r}^T(\mathbf{w})\mathbf{r}(\mathbf{w}),\end{aligned} \tag{3.11}$$

where

$$\mathbf{r}(\mathbf{w}) = \begin{bmatrix} r_1(\mathbf{w}) & r_2(\mathbf{w}) & \dots & r_N(\mathbf{w}) \end{bmatrix}^T. \tag{3.12}$$

To solve this optimization problem, because of the non-linear nature of the cost function, it is commonly preferred to adopt an iterative descent algorithm such as *Steepest-Descent* method [36, 41]. However, in its simple form, Steepest-Descent suffers from serious convergence problems. Another method is the famous *Gauss-Newton method* which is computationally more expensive.

Levenberg-Marquardt algorithm (known as LM algorithm) was originally proposed by K. Levenberg in 1944 [51]. It is a combination of Steepest-Descent and Gauss-Newton methods. In the Steepest-Descent method, parameters are adjusted in the direction opposite to the gradient of the cost function by a given step size

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \lambda \nabla \psi. \tag{3.13}$$

Two major convergence problems of Steepest-Descent method are related to the learning rate λ and the direction of weight adjustments. Considering the learning rate, it is logical, for instance, to have small step size when the slope of cost function is large and to have large step size when the slope is small. In Eq. (3.13) it is the opposite. Although we may be able to tune the step by appropriately adjust λ using some methods such as line search, we would rather to find a way to systematically adjust it. On the other hand, in Gauss-Newton method, the direction of weight adjustment is obtained using both the first and the second derivatives of cost function (Jacobian and Hessian, respectively) which is a much better choice than using only the first derivative. However, as already mentioned, calculation of Hessian is computationally expensive. One remedy for this is to approximate the Hessian around the minima. But to use such an approximation, we first need to approach to some minima. This implies an approach based both on the Steepest Descent (while the algorithm is far from any minima) and the approximation to Gauss-Newton method (when a minimum is approached). Accordingly, the weight adjustment in LM algorithm is depicted by

$$\Delta \mathbf{w} = -(\mathbf{J}^T\mathbf{J} + \lambda \mathbf{I})^{-1} \nabla_{\mathbf{w}} \psi. \tag{3.14}$$

In Eq. (3.14), $\nabla_{\mathbf{w}}\psi$ is the gradient of the cost function with respect to the model's adjustable parameters $\mathbf{w}$ (Eq. (2.20)). $\mathbf{J}$ is the *Jacobian matrix* of $\mathbf{r}$ and is shown by Eq. (3.15). λ is a non-negative scalar and $\mathbf{I}$ is the identity matrix of appropriate size. Note also that the term $\mathbf{J}^T\mathbf{J}$ is the mentioned approximation to the Hessian around a minimum.

$$\mathbf{J}_{\mathbf{r}} = \begin{bmatrix} \frac{\partial r_1}{\partial w_1} & \cdots & \frac{\partial r_1}{\partial w_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial r_N}{\partial w_1} & \cdots & \frac{\partial r_N}{\partial w_n} \end{bmatrix} = \begin{bmatrix} \nabla_{\mathbf{w}}^T r_1(\mathbf{w}) \\ \vdots \\ \nabla_{\mathbf{w}}^T r_N(\mathbf{w}) \end{bmatrix} \tag{3.15}$$

Another version of LM algorithm is obtained by replacing the identity matrix in 3.14 with the diagonals of $\mathbf{J}^T\mathbf{J}$. Doing so, Eq. (3.14) becomes

$$\Delta \mathbf{w} = -(\mathbf{J}^T\mathbf{J} + \lambda\, diag[\mathbf{J}^T\mathbf{J}])^{-1} \nabla_{\mathbf{w}} \psi. \tag{3.16}$$

For the reasons behind this substitution consult [41, 58].
Having the update rules at hand, the LM algorithm for optimizing the scalar output model of Eq. (3.10) is expressed in Algorithm 2.
There are a few notes about this algorithm we shall point out through the following remarks.

Remark 3.3.1. LM algorithm is *heuristic*. It means that it is not optimal in a sense that it may not be able to find the global minimum of the cost function. But in practice it performs extremely well. However, the need for calculating the inversion is a flaw and in large-scale models with a few thousands parameters the cost of update becomes prohibitive. For moderated size models, however, it is one of the fastest algorithms so far.

Remark 3.3.2. The initial conditions for d and α are arbitrary. However, usually λ should be large at the beginning and decreasing as the algorithm approaches a minimum. This implies that at first the update rule behaves such as a Steepest Descent and as we go towards a minimum, it gradually shifts to Gauss-Newton method.

Remark 3.3.3. Stopping conditions may be one or some of these factors:

Algorithm 2 Simple LM algorithm for weight update of single output model.

Require: A training set ($D = \mathbf{u}_i^t, y_i^t$) with N samples.
Require: A model with scalar output and n adjustable parameters(eq. 3.10).
Ensure: A (nearly) optimum solution for minimizing eq. 3.11 .

```
    Initialization:
 1: $d \leftarrow 0.0001$
 2: $\alpha \leftarrow 10.0$
 3: $k \leftarrow 0$
 4: $\mathsf{STOP} \leftarrow \mathsf{FALSE}$.
 5: $\mathbf{w}(k) \leftarrow$ random values in $(-1, 1)$
    Main loop:
 6: Evaluate $\psi_{old} = \psi_{SSE}(\mathbf{w}(k))$ using eq. 3.11.
 7: Compute $\mathbf{J}$ using eq. 3.15 and the model function $f$
 8: $h_{max} \leftarrow max(diag[\mathbf{J}^T\mathbf{J}])$
 9: $\lambda \leftarrow dh_{max}$
10: while $\mathsf{STOP} \neq \mathsf{TRUE}$ do
11:   Store current network weights: $\mathbf{w}_{old} \leftarrow \mathbf{w}(k)$
12:   Update weights using eq. 3.14 or 3.16.
13:   Evaluate $\psi_{new} = \psi(\mathbf{w}(k+1))$ using eq. 3.11
14:   if $\psi_{new} < \psi_{old}$ then
15:     $d \leftarrow \frac{d}{\alpha}$
16:     $\psi_{new} = \psi_{old}$
17:   else
18:     $d \leftarrow d \times \alpha$
19:     $\mathbf{w}(k+1) \leftarrow \mathbf{w}_{old}$
20:   end if
21:   if (Stopping conditions are met) then
22:     $\mathsf{STOP} \leftarrow \mathsf{TRUE}$.
23:   end if
24:   $k \leftarrow k + 1$
25: end while
```

- A minimum error to be met,
- A maximum number of iterations,
- A minimum change on the error,

or something similar. The most important step is to compute the Jacobian. In fact, using LM algorithm in any model requires deriving the Jacobians appropriately. Thus, applying LM algorithm to LRSI needs the same ingredient. Next, we will define the Jacobian and other details to illustrate how the weights in LRSI are to be updated.

3.3.2 Levenberg-Marquardt Algorithm for LRSI

To derive the LM algorithm for LRSI the first step is to define the cost function. Recall that LRSI has n_O outputs. Thus, we first define the instantaneous error function

$$\begin{aligned} e_h(k) &= y_h^t(k) - y_h(k) \quad \text{for } h = 1, \ldots, n_O \\ E(k) &= \frac{1}{2}\sum_{h=1}^{n_O} e_h^2(k) \\ &= \frac{1}{2}\mathbf{e}^T(k)\mathbf{e}(k), \end{aligned} \tag{3.17}$$

where

$$\mathbf{e}(k) = \begin{bmatrix} e_1(k) & e_2(k) & \ldots & e_{n_O}(k) \end{bmatrix}^T. \tag{3.18}$$

The error over a finite horizon becomes

$$E(t_0, t) = \sum_{k=t_0}^{t} E(k). \tag{3.19}$$

Thus, the objective is to find **W** and **P** in Eqs. (3.3) which minimize one or both of the above cost functions (i.e., Eqs. (3.17) and (3.19)). Using LM algorithm, we need to have the derivatives of all states and outputs with respect to the network weights. As a result of recurrence in the network, weights cannot be arranged into one single vector, because states and outputs are intertwined through time. So, we have to keep the weights arranged into matrices. Taking derivative of each state (or output) with respect to a weight matrix yields a matrix, namely $\mathbf{J}_\mathbf{W}^{x_l}$ and $\mathbf{J}_\mathbf{P}^{x_l}$ (for states) or $\mathbf{J}_\mathbf{W}^{y_h}$ and $\mathbf{J}_\mathbf{P}^{y_h}$ (for outputs) , where $l = 1, ..., n_X$ and $h = 1, ..., n_O$. Therefor, in each time instance k, we have a set of matrices each of which containing derivatives of a state or an output with respect to all of the network weights. Grouping each of these matrices according to whether it belongs to a state or an output, we obtain two groups of matrices. Each of these groups can also be grouped into two smaller groups according to the weights respect to which the derivatives have been taken. Finally, we have four groups of matrices

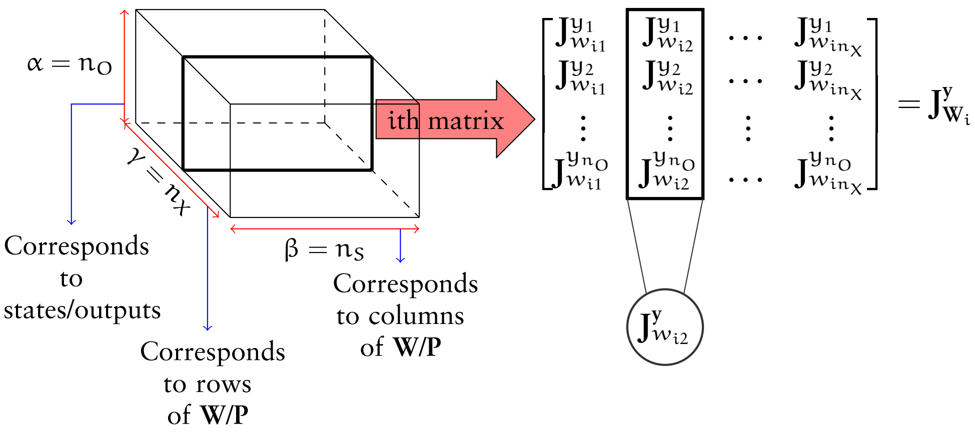
$$\mathbf{J}^{\mathbf{y}}_{\mathbf{W}}(k) = \left[\frac{\partial y_h(k)}{\partial w_{ij}}\right]_{n_O \times n_S \times n_X} \tag{3.20a}$$

$$\mathbf{J}^{\mathbf{x}}_{\mathbf{W}}(k) = \left[\frac{\partial x_l(k)}{\partial w_{ij}}\right]_{n_X \times n_S \times n_X} \tag{3.20b}$$

$$\mathbf{J}^{\mathbf{y}}_{\mathbf{P}}(k) = \left[\frac{\partial y_h(k)}{\partial p_{ij}}\right]_{n_O \times n_R \times n_O} \tag{3.20c}$$

$$\mathbf{J}^{\mathbf{x}}_{\mathbf{P}}(k) = \left[\frac{\partial x_l(k)}{\partial p_{ij}}\right]_{n_X \times n_R \times n_O}. \tag{3.20d}$$

Note that organizing the derivatives in 3D arrays eases future access to one or a number of them. We call these 3D arrays of derivatives *the Jacobians*. It is good to have a visual interpretation of the Jacobians. In Figure 3.4 $\mathbf{J}^{\mathbf{y}}_{\mathbf{W}}(k)$ is visually presented as an example.



$\alpha = n_O$
$\gamma = n_X$
$\beta = n_S$
ith matrix
Corresponds to states/outputs
Corresponds to rows of **W**/**P**
Corresponds to columns of **W**/**P**
$\begin{bmatrix} J^{y_1}_{w_{i1}} & J^{y_1}_{w_{i2}} & \cdots & J^{y_1}_{w_{in_X}} \\ J^{y_2}_{w_{i1}} & J^{y_2}_{w_{i2}} & \cdots & J^{y_2}_{w_{in_X}} \\ \vdots & \vdots & \vdots & \vdots \\ J^{y_{n_O}}_{w_{i1}} & J^{y_{n_O}}_{w_{i2}} & \cdots & J^{y_{n_O}}_{w_{in_X}} \end{bmatrix} = \mathbf{J}^{\mathbf{y}}_{\mathbf{W}_i}$
$\mathbf{J}^{\mathbf{y}}_{w_{i2}}$

Figure 3.4: Visual interpretation for $\mathbf{J}^{\mathbf{Y}}_{\mathbf{W}}(k)$.

Understanding this visual interpretation is crucial for understanding Jacobians and therefore, the update rules. Note that the formation of derivatives are chosen to be convenient for implementation. Parameters α,β and γ represent dimensions of the Jacobians. For example, for this figure that $\mathbf{J}^{\mathbf{y}}_{\mathbf{W}}(k)$ is illustrated we have $\alpha = n_O$, $\beta = n_S$ and $\gamma = n_X$. If we were to illustrate $\mathbf{J}^{\mathbf{x}}_{\mathbf{P}}(k)$ then we would have had $\alpha = n_X$, $\beta = n_R$ and $\gamma = n_O$. Thus, in Eqs. (3.20) and (3.22), i and j correspond to γ and β and l and h correspond to α. Having understood this interpretation and notation, it should be clear that, for example, the derivative of $\mathbf{x}$ with respect to a single parameter w_{ij} is equal to

$$\frac{\partial \mathbf{x}}{w_{ij}} = \left[\frac{\partial x_1}{w_{ij}} \quad \cdots \quad \frac{\partial x_{n_X}}{w_{ij}}\right] = \mathbf{J}^{\mathbf{x}}_{w_{ij}}. \tag{3.21}$$

Using dynamic equations of LRSI (Eqs. (3.3) and (3.4)) we can compute the defined Jacobians,

$$\begin{aligned} J^{y_h}_{w_{ij}}(k) &= \frac{\partial y_h(k)}{\partial w_{ij}} \\ &= g'_h(\mathbf{P}_h\mathbf{r}(k))\left(\mathbf{C}_h\tfrac{\partial \mathbf{x}(k)}{\partial w_{ij}} + \mathbf{G}_h\tfrac{\partial \mathbf{y}(k-1)}{\partial w_{ij}}\right) \\ &= g'_h(\mathbf{P}_h\mathbf{r}(k))\left(\mathbf{C}_h\mathbf{J}^{\mathbf{x}}_{w_{ij}}(k) + \mathbf{G}_h\mathbf{J}^{\mathbf{y}}_{w_{ij}}(k-1)\right), \end{aligned} \tag{3.22}$$

where index h means h^{th} row of the corresponding matrix or vector - the same holds for index l in Eqs. (3.23b) and (3.23d). Note that the inputs are not dependent on any weights, thus $\frac{\partial \mathbf{u}}{\partial w_{ij}} = \frac{\partial \mathbf{u}}{\partial p_{ij}} = \mathbf{0}$. Apparently, elements of $\mathbf{W}$ are not dependent on $\mathbf{P}$ and vice versa. Therefore, their derivatives with respect to each other are all zero.

Doing so for all the Jacobians we have

$$J^{y_h}_{w_{ij}}(k) = \frac{\partial y_h(k)}{\partial w_{ij}} = g'_h(\mathbf{P}_h\mathbf{r}(k))\left(\mathbf{C}_h\mathbf{J}^{\mathbf{x}}_{w_{ij}}(k) + \mathbf{G}_h\mathbf{J}^{\mathbf{y}}_{w_{ij}}(k-1)\right) \tag{3.23a}$$

$$J^{x_l}_{w_{ij}}(k) = \frac{\partial x_l(k)}{\partial w_{ij}} = f'_l(\mathbf{W}_l\mathbf{s}(k))\left(\mathbf{A}_l\mathbf{J}^{\mathbf{x}}_{w_{ij}}(k-1) + \mathbf{F}_l\mathbf{J}^{\mathbf{y}}_{w_{ij}}(k-1) + s_j(k)\right) \tag{3.23b}$$

$$J^{y_h}_{p_{ij}}(k) = \frac{\partial y_h(k)}{\partial p_{ij}} = g'_h(\mathbf{P}_h\mathbf{r}(k))\left(\mathbf{C}_h\mathbf{J}^{\mathbf{x}}_{p_{ij}}(k) + \mathbf{G}_h\mathbf{J}^{\mathbf{y}}_{p_{ij}}(k-1) + r_j(k)\right) \tag{3.23c}$$

$$J^{x_l}_{p_{ij}}(k) = \frac{\partial x_l(k)}{\partial p_{ij}} = f'_l(\mathbf{W}_l\mathbf{s}(k))\left(\mathbf{A}_l\mathbf{J}^{\mathbf{x}}_{p_{ij}}(k-1) + \mathbf{F}_l\mathbf{J}^{\mathbf{y}}_{p_{ij}}(k-1)\right). \tag{3.23d}$$

Arranging the Jacobian elements into their corresponding matrices, after a bit of algebraic manipulation we reach the following iterative update rules for the Jacobians

$$\mathbf{J}^{\mathbf{y}}_{\mathbf{W}_i}(k) = \textit{diag}\left(\mathbf{g}'(\mathbf{Pr}(k))\right)\left(\mathbf{CJ}^{\mathbf{x}}_{\mathbf{W}_i}(k) + \mathbf{GJ}^{\mathbf{y}}_{\mathbf{W}_i}(k-1)\right) \tag{3.24a}$$

$$\mathbf{J}^{\mathbf{x}}_{\mathbf{W}_i}(k) = \textit{diag}\left(\mathbf{f}'(\mathbf{Ws}(k))\right)\left(\mathbf{AJ}^{\mathbf{x}}_{\mathbf{W}_i}(k-1) + \mathbf{FJ}^{\mathbf{y}}_{\mathbf{W}_i}(k-1) + \boldsymbol{\delta}_i\mathbf{s}(k)\right) \tag{3.24b}$$

$$\mathbf{J}^{\mathbf{y}}_{\mathbf{P}_i}(k) = \textit{diag}\left(\mathbf{g}'(\mathbf{Pr}(k))\right)\left(\mathbf{CJ}^{\mathbf{x}}_{\mathbf{P}_i}(k) + \mathbf{GJ}^{\mathbf{y}}_{\mathbf{P}_i}(k-1) + \boldsymbol{\delta}_i\mathbf{r}(k)\right) \tag{3.24c}$$

$$\mathbf{J}^{\mathbf{x}}_{\mathbf{P}_i}(k) = \textit{diag}\left(\mathbf{f}'(\mathbf{Ws}(k))\right)\left(\mathbf{AJ}^{\mathbf{y}}_{\mathbf{P}_i}(k-1) + \mathbf{FJ}^{\mathbf{y}}_{\mathbf{P}_i}(k-1)\right), \tag{3.24d}$$

where $\boldsymbol{\delta}_i$ is a vector whose all elements are zero except for the i^{th} element which is one. Studying Eqs. (3.24) we observe that this algorithm possesses a recursive nature, hence is similar to RTRL. Consequently, the initial conditions for above Jacobians are all zero. Now that we have the update rule for Jacobians, we can proceed to derive the update rule for weights and complete the LM algorithm. Because the errors are only measurable for the outputs, the Jacobians of **x** are not directly incorporated in weight update rules, but are used to update Jacobians of **y**. We can compute the gradient of errors

$$\nabla_{\mathbf{W}_i} E(k) = \mathbf{J}^{\mathbf{y}\ T}_{\mathbf{W}_i}(k)\mathbf{e}(k) \tag{3.25a}$$

$$\nabla_{\mathbf{P}_j} E(k) = \mathbf{J}^{\mathbf{y}\ T}_{\mathbf{P}_j}(k)\mathbf{e}(k), \tag{3.25b}$$

where i and j iterate over rows of **W** and **P**, respectively. Having computed the errors, the update rules become

$$\Delta W_i(k) = -\left[\mathbf{H}_{\mathbf{W}}(i,k) + \lambda_{W_i} diag(\mathbf{H}_{\mathbf{W}}(i,k))\right]^{-1} \nabla_{\mathbf{W}_i} E(k) \tag{3.26a}$$

$$\Delta \mathbf{P}_j(k) = -\left[\mathbf{H}_{\mathbf{P}}(j,k) + \lambda_{P_j} diag(\mathbf{H}_{\mathbf{P}}(j,k))\right]^{-1} \nabla_{\mathbf{P}_j} E(k), \tag{3.26b}$$

where

$$\mathbf{H}_{\mathbf{W}}(i,k) = \mathbf{J}^{\mathbf{y}\ T}_{\mathbf{W}_i}(k)\mathbf{J}^{\mathbf{y}}_{\mathbf{W}_i}(k) \tag{3.27a}$$

$$\mathbf{H}_{\mathbf{P}}(j,k) = \mathbf{J}^{\mathbf{y}\ T}_{\mathbf{P}_j}(k)\mathbf{J}^{\mathbf{y}}_{\mathbf{P}_j}(k). \tag{3.27b}$$

In neighborhoods of extrema, Eqs. (3.27) are good approximations to corresponding Hessians. For more details see [41]. Note also that the weight updates obtained by Eqs. (3.26) are in a vector form while they are related to the *rows* of weight matrices. Thus, bear in mind that they should be transposed before being added to the rows of weight matrices.
In the light of the mentioned update rules, one iteration of the real time version of LM algorithm for LRSI at a given time instance k, consists of the following steps:

1. Measure the instantaneous error $\mathbf{e}(k)$ and evaluate the cost function $E(k)$.
2. Update the Jacobians using Eqs. (3.24).
3. Calculate the gradient of the instantaneous error using Eqs. (3.25).
4. Calculate the approximations to Hessians using Eqs. (3.27).
5. Calculate the weight updates using Eqs. (3.26) and update the weights.
6. Evaluate the new performance and decide to keep or discard the new weights.

One should note that this routine shall be performed for a number of iterations for each time instance, where for each iteration, the parameter λ needs to be updated until a stop criterion is achieved. Algorithms 3 and 4 lists the LM learning algorithm applied to LRSI in a time horizon equal to T. The following remarks are to highlight important aspects of this algorithm.

Remark 3.3.4. For the sake of simplicity, subroutines for updating matrices $\mathbf{W}$ and $\mathbf{P}$ are listed in another algorithm (algorithm 4) but with the same assumptions and shared-memory with algorithm 3.

Remark 3.3.5. Samples are given to the network one at each k. For each given sample, two loops are performed, one for updating the rows of $\mathbf{W}(k)$ and the other for the rows of $\mathbf{P}(k)$. After each of these loops, a re-evaluation is performed and it is checked if the update was effective or not and in either case the necessary actions are taken. Note that the re-evaluation and check procedure can be done inside the loops, i.e., for each row of the two matrices. However, this will lead to a greedy search and is more probable to end in local minima. Another approach would be to perform the re-evaluation and check procedure once, after the second loop only. This may lead to slower convergence but less vulnerable to being trapped to local minima.

3.3.3 Implementation Remarks

The proposed algorithm in the previous section consists of several stages. So, to study its real behaviour what matters the most is how it is implemented. Hereafter, we will give some experimental guidelines in implementing the algorithm.

Order of calculations In calculating the derivatives and generating the outputs, we need to be aware of the update order. If we intend to use the previous states in generating outputs, i.e., $\mathbf{r}(k) = [\mathbf{x}(k) \quad \ldots]$, then we must first update the states, then use them. This also applied to evolving the Jacobians.

Rank of Hessian As the parameter space grows, it is more likely for Hessians not to be full-rank. In fact, the second term to be added to the Hessian before taking the inverse (Eq. (3.26)) is to avoid inverting an ill-defined matrix. However, still it may be possible not to have a "nice" matrix. Thus, we should check the scale and formation of the matrix obtained in the parenthesis of Eqs. (3.26) before inverting them. If the calculated matrix does not sound nice, we may ignore the current iteration and wait for more information to come, or we may slightly manipulate the Hessian.

Storing the parameters Note that LM algorithm performs in a trial-and-error fashion, i.e., each time an update is calculated, it is examined whether this update results in a better performance or not. If not, we turn back to what we had before the update. So, we must have saved them as clearly illustrated in the Algorithm 3. This storing procedure is not only applied to the weights, but also to the Jacobians for two main reasons. First, the initial conditions must be saved because Jacobians are evolved through time and are dependent on their

Algorithm 3 LM algorithm for LRSI parameter update

```
Require: A training set ($D = \mathbf{u}^t(k), y^t(k)$) with T samples ($k = 1, ..., T$)
Require: An appropriate LRSI model (eq. 3.4 and 3.3)
    Initialization:
 1: $k \leftarrow 1$;
 2: $\mathbf{W}(k) \leftarrow$ random values in $(-1, 1)$; $\mathbf{P}(k) \leftarrow$ random values in $(-1, 1)$;
 3: Set all the Jacobians in eq. 3.20 to zero matrices.
    Main loop:
 4: while $k < T + 1$ do
 5:   $d_W \leftarrow 0.0001$; $d_P \leftarrow 0.0001$; $\alpha \leftarrow 10.0$; $cnt \leftarrow 1$, $STOP \leftarrow FALSE$
 6:   Store the weights: $\mathbf{W}_{old} \leftarrow \mathbf{W}(k)$ and $\mathbf{P}_{old} \leftarrow \mathbf{P}(k)$;
 7:   Store the Jacobians: $\mathbf{J}^{x,W}{}_{old} \leftarrow \mathbf{J}^x_W(k)$ and
      $\mathbf{J}^{y,W}{}_{old} \leftarrow \mathbf{J}^y_W(k)$ and $\mathbf{J}^{x,P}{}_{old} \leftarrow \mathbf{J}^x_P(k)$ and $\mathbf{J}^{y,P}{}_{old} \leftarrow \mathbf{J}^y_P(k)$;
 8:   while ($STOP \neq TRUE$) do
 9:     Calculate network states and outputs using eq. 3.3 or 3.4;
10:     Measure the instantaneous error $\mathbf{e}(k)$ and evaluate the cost function
        $E_{old} \leftarrow E(k)$ using eq. 3.17;
11:     $\mathbf{W}(k) \leftarrow$ UpdateW();
12:     Calculate network states and outputs using eq. 3.3 or 3.4;
13:     Evaluate $E_{new} \leftarrow E(k)$ using eq. 3.17;
14:     if $E_{new} < E_{old}$ then
15:       $d_W \leftarrow \frac{d_W}{\alpha}$;
16:       $E_{new} = E_{old}$;
17:     else
18:       $d_W \leftarrow d_W \times \alpha$;
19:       Restore W: $\mathbf{W}(k) \leftarrow \mathbf{W}_{old}$;
20:       Restore Jacobians: $\mathbf{J}^x_W(k) \leftarrow \mathbf{J}^{x,W}{}_{old}, \mathbf{J}^y_W(k) \leftarrow \mathbf{J}^{y,W}{}_{old}$;
21:     end if
22:     $\mathbf{P}(k) \leftarrow$ UpdateP();
23:     Calculate network states and outputs using eq. 3.3 or 3.4;
24:     Evaluate $E_{new} \leftarrow E(k)$ using eq. 3.17;
25:     if $E_{new} < E_{old}$ then
26:       $d_P \leftarrow \frac{d_P}{\alpha}$;
27:       $E_{new} = E_{old}$;
28:     else
29:       $d_P \leftarrow d_P \times \alpha$;
30:       Restore P: $\mathbf{P}(k) \leftarrow \mathbf{P}_{old}$;
31:       Restore Jacobians: $\mathbf{J}^x_P(k) \leftarrow \mathbf{J}^{x,P}{}_{old}, \mathbf{J}^y_P(k) \leftarrow \mathbf{J}^{y,P}{}_{old}$;
32:     end if
33:     $cnt \leftarrow cnt + 1$;
34:     if (Stopping conditions are met) then
35:       $STOP \leftarrow TRUE$;
36:     end if
37:   end while
38:   $k \leftarrow k + 1$;
39: end while
```

Algorithm 4 LM algorithm for LRSI parameter update - subroutines

```
 1: UpdateW();
 2: for all $i = 1$ to $n_X$ do
 3:     Update the Jacobians $\mathbf{J}^x_{\mathbf{W}_i}(k)$ and $\mathbf{J}^y_{\mathbf{W}_i}(k)$ using eq. 3.24a, 3.24b;
 4:     Calculate the gradient $\nabla_{\mathbf{W}_i} E(k)$ using eq. 3.25a;
 5:     Calculate the Hessian $\mathbf{H}_{\mathbf{W}}(i,k)$ using eq. 3.27a;
 6:     $h_{max} \leftarrow max(\mathit{diag}[\mathbf{H}_{\mathbf{W}}(i,k)f])$;
 7:     $\lambda \leftarrow d_W h_{max}$;
 8:     Calculate the weight update $\Delta \boldsymbol{W}_i(k)$ using eq. 3.26a;
 9:     Update the ith row of$\boldsymbol{W}(k)$: $\mathbf{W}_i(k) \leftarrow \mathbf{W}_i(k) + \Delta \boldsymbol{W}_i(k)$;
10: end for
11: UpdateP();
12: for all $i = 1$ to $n_O$ do
13:     Update the Jacobians $\mathbf{J}^x_{\mathbf{P}_i}(k)$ and $\mathbf{J}^y_{\mathbf{P}_i}(k)$ using eq. 3.24c and 3.24d.
14:     Calculate the gradient $\nabla_{\mathbf{P}_i} E(k)$ using eq. 3.25b.
15:     Calculate the Hessian $\mathbf{H}_{\mathbf{P}}(i,k)$ using eq. 3.27b.
16:     $h_{max} \leftarrow max(\mathit{diag}[\mathbf{H}_{\mathbf{P}}(i,k)f])$
17:     $\lambda \leftarrow d_P h_{max}$
18:     Calculate the weight update $\Delta \mathbf{P}_i(k)$ using eq. 3.26b.
19:     Update the ith row of$\mathbf{P}(k)$: $\mathbf{P}_i(k) \leftarrow \mathbf{P}_i(k) + \Delta \mathbf{P}_i(k)$.
20: end for
```

previous values. Second, the Jacobians are dependent on weights too, so if the weights are rolled back to some previous values, the Jacobians should be restored as well, and if not, they need to be appropriately updated.

3.4 Enhancements in LRSI Learning Algorithm

Up to now, the basis of our proposed learning mechanism is explained. In this section we outline the enhancements to be applied to this mechanism. In brief, these improvements are in the following forms:

- **Prolongation of the minimization horizon**: we will try to optimize the cost function over some previous observations.
- **Adding regularization-weight decay factor**: an additional term will be added to the cost function to penalize large weight values.
- **Adding momentum term**: a practical improvement in update rule is to add a momentum term to increase the learning rate yet avoiding instability.

- **Adding a forgetting term to Jacobian calculation**: to have preference over recent observations, we may add a forgetting factor to let the effect of observations be gradually decaying over time.
- **Stochastic component**: to decrease the probability of getting trapped in local minima a stochastic component will be added once we find a set of parameters that yield minimum value for the cost function.
- **Committees**: because the initial random guess for the weights has a great impact on the final solution, we may employ several LRSI networks in parallel and use them in a committee for prediction after being evaluated.

3.4.1 Prolongation of the minimization horizon

The update rules described by Eqs .(3.26) are obtained using the instantaneous cost function of Eq. (3.17). We can extend this cost function to obtain the *prolonged* version

$$E = \frac{1}{2} \sum_{k=t}^{t-m^-} \epsilon^{t-k} E(k), \tag{3.28}$$

where m^- is the length of the prolongation; to which we will refer as *prolongation depth*. The co-efficient $0 < \epsilon \leqslant 1$ can be used to favor minimization on recent errors. However, it is advised to use either this co-efficient or the forgetting term (to be later introduced). Using both of them may result in rapid loss of information. Thus, we assume $\epsilon = 1$ and will ignore it after on.
Doing so, the total update amount for weights will be the weighted summation of all the instantaneous weight updates depicted by Eqs. (3.26) while k in these equations iterates over $t - m^-$ to t. To avoid multiple inversion, we can first add all the Hessians in the prolonged window and then apply the inversion.

$$\begin{aligned} \mathbf{H_W}(\mathbf{i}) &= \frac{1}{2} \sum_{k=t}^{t-m^-} \mathbf{H_W}(i,k) \\ \mathbf{H_P}(\mathbf{j}) &= \frac{1}{2} \sum_{k=t}^{t-m^-} \mathbf{H_P}(j,k) \end{aligned} \tag{3.29}$$

where $\mathbf{H_W}(i,k)$ and $\mathbf{H_P}(j,k)$ are shown in Eq. (3.27). The addition also applies to the gradients:

$$\begin{aligned} \nabla_{\mathbf{W}_i} E &= \frac{1}{2} \sum_{k=t}^{t-m^-} \nabla_{\mathbf{W}_i} E(k) \\ \nabla_{\mathbf{P}_j} E &= \frac{1}{2} \sum_{k=t}^{t-m^-} \nabla_{\mathbf{P}_j} E(k). \end{aligned} \tag{3.30}$$

Then the update for rows of $\mathbf{W}$ and $\mathbf{P}$ in the prolonged version becomes

$$\Delta \mathbf{W}_i = -\left[\mathbf{H_W}^T(i)\mathbf{H_W}(i) + \lambda_{W_i} diag(\mathbf{H_W}(i))\right]^{-1} \nabla_{\mathbf{W}_i} E \qquad (3.31a)$$

$$\Delta \mathbf{P}_j = -\left[\mathbf{H_P}^T(j)\mathbf{H_P}(j) + \lambda_{\mathbf{P}_j} diag(\mathbf{H_P}(j))\right]^{-1} \nabla_{\mathbf{P}_j} E. \qquad (3.31b)$$

Note that the removal of time index k emphasizes the fact that updates are obtained over a finite horizon.
Although this prolongation increases the computational load, but, as to be seen in Chapter 5, the performance significantly improves especially for the cases where offline system identification is favored.

3.4.2 Weight Decay Regularization

In LRSI, number of parameters are determined by the number of internal states. For a particular problem, the optimal choice for this quantity is rarely known. Small number of states may lead to loss of some valuable data and consequently poor generalization while too many states may also lead to overfitting hence poor generalization, too. One solution is to penalize large weights.[4] To do that, we add another term to the final cost function

$$E = \frac{1}{2}\sum_{k=t}^{t-m^-} \epsilon^{t-k} E(k) + \frac{\alpha_w}{2}\mathbf{W}^T\mathbf{W} + \frac{\alpha_p}{2}\mathbf{P}^T\mathbf{P}, \qquad (3.32)$$

where α_w and α_p are the design coefficients and represent the degree of penalizing the weights. Taking into account these two additional terms, their derivatives should also appear in the Jacobians and Hessians. Let us continue with the prolonged version of the cost function, we have

$$\begin{aligned} \mathbf{H_W}(i) &= \frac{1}{2}\sum_{k=t}^{t-m^-} \mathbf{H_W}(i,k) + \alpha_w[1]_{n_S \times n_S} \\ \mathbf{H_P}(j) &= \frac{1}{2}\sum_{k=t}^{t-m^-} \mathbf{H_P}(j,k) + \alpha_p[1]_{n_R \times n_R}, \end{aligned} \qquad (3.33)$$

where $\mathbf{H_W}(i,k)$ and $\mathbf{H_P}(j,k)$ are shown in Eq. (3.27). The addition also applies to the gradients:

$$\begin{aligned} \nabla_{\mathbf{W}_i} E &= \frac{1}{2}\sum_{k=t}^{t-m^-} \nabla_{\mathbf{W}_i} E(k) + \alpha_w \mathbf{W}_i \\ \nabla_{\mathbf{P}_j} E &= \frac{1}{2}\sum_{k=t}^{t-m^-} \nabla_{\mathbf{P}_i} E(k) + \alpha_p \mathbf{P}_j. \end{aligned} \qquad (3.34)$$

[4]For a comprehensive discussion on regularization theory and its different approaches consult [15, 36].

A draw back of this form of regularization returns to the fact that not all of the weights evenly contribute to generating outputs over time. It means there might be a subset of weights that contribute more to generating the output in some period of time, while another subset of weights may be active in a later time. During the former, the first subset are more penalized hence tend to zero while in the later it is vice versa. This might be compensated by choosing a *clever* sampling time and a suitable prolongation depth.

3.4.3 Other Suggested Improvements

Adding a Momentum Term

As originally proposed by Rumelhart et al. [71] in the context of delta rule, to increase the learning rate yet avoiding the danger of instability, a practical improvement is to add a momentum term which adds a fraction of the previous update amount. Although in LM algorithm we have a more flexible learning rate, there might be situations where due to some badly scaled Hessian, an update should and may not be calculated (refer to Implementation Remarks). In these situations, momentum term speeds up the convergence. The update rule with a momentum term is shown in

$$\begin{aligned} \Delta W_i(l) &= \eta_W \Delta W_i(l-1) + \left[\mathbf{H_W}^T(i)\mathbf{H_W}(i) + \lambda_{W_i} \mathit{diag}(\mathbf{H_W}(i))\right]^{-1} \nabla_{W_i} E \\ \Delta \mathbf{P}_j(l) &= \eta_P \Delta \mathbf{P}_j(l-1) + \left[\mathbf{H_P}^T(j)\mathbf{H_P}(j) + \lambda_{P_j} \mathit{diag}(\mathbf{H_P}(j))\right]^{-1} \nabla_{P_j} E, \end{aligned} \tag{3.35}$$

where η_W and η_P control the influence of momentum term on the update amount, called *momentum constant*.

Stochastic component

In offline training, it is more preferable to find the best minimum, hopefully the global one. However, the problem at hand is highly nonlinear and it is very likely to have different minima. We suggest two approaches to increase the chance of finding the global minimum: adding a stochastic component and committees.
Once a minimum is reached in a given horizon (which can be checked by inspecting the changes to cost function or the update rates) we randomly perturb some (or all) of the weights. Although this leads to temporary increase in the cost function, the perturbation may save the solution from being trapped in the local minima.

Forgetting term

In the scenarios where the pure real-time update is used, the accumulation of information in the recursive equations of Jacobians calculation may prevent the network from rapidly adapt to new observations. This problem can be alleviated using a *forgetting factor*. The forgetting factor simply favors the most recent computed Jacobians by down-scaling the initial conditions. This term can appear as a multiplied co-efficient to the initial values of Jacobians. The inclusion of forgetting term may result in slower convergence. Overall, used along with the prolongation depth yields better result.

Committees

In on-line training, perturbing the weights may not be a good idea because time is important in this scenario. The other approach may be to employ a number of LRSIs to work in parallel. It is often more efficient to have their parameters initialized enough apart in the parameter space. The predicted output is the weighted average among them and the weights can be determined inspecting their behaviour during learning, i.e., those with a better learning behaviour contribute more in the prediction.

In this chapter, we presented our main contribution in this thesis. In the next one, we propose our second contribution which is an LM-based optimization MPC that uses LRSI as the predictor. In Chapter 5, we will study performance of identification and control using these proposed schemes.

Chapter 4
Predictive Control Based On Recurrent Neural Networks

Based on the proposed recurrent scheme for system identification, in this chapter we are aimed at proposing a predictive control strategy. Model-Based Predictive Control (*MPC* or *MPBC*) is one of the most frequently used control schemes in practical applications. Having a model that can predict the future of the environment (or process under control) to some satisfying degree, the basic idea is to find a control strategy that optimizes a cost function of the predictions, a desired trajectory and perhaps the control actions. Because it is more likely to have such a *reasonable* model of the process in industrial applications, MPC has become the dominant technique in the industrial applications. Another capability of MPC is the inclusion of possible constraints in the optimization problem. This leads to utilizing the control resources more efficiently. One may already conclude that the basic ingredient of MPC is the model. When the model is linear, or at least piece-wise linear, MPC works almost optimal, due to the well formulated optimization problem. However, when the environment becomes complex and highly nonlinear performance of MPC degrades drastically. This is where Nonlinear Generalized Predictive Control (NGPC) comes into play.[1] In NGPC, the model is nonlinear and the optimization process is usually carried out in an iterative fashion. Among different nonlinear models to be used in this scheme, applications of neural networks have been steadily increasing. As discussed earlier this is due to the approximation capability of NN.
Basic discussions about GPC and NGPC in general are beyond scope of this thesis and will not be covered.[2]. In this chapter, however, we shortly review NNGPC-based approaches with an emphasis on recurrent schemes. After that, we will define mathematics of a specific approach which will result in an LM

[1]Note that NGPC may also refer to Neural Generalized Predictive Control in the literature. To prevent any ambiguities we will refer to the later by NNGPC.

[2]Consult [56, 20, 21], for example.

update rule towards discovering the control strategy in the predictive scheme. Experimental results will be demonstrated in Chapter 5.

4.1 Introduction

Quite recently, NNGPC has been increasingly attracting researchers not only theoretically [2, 1, 4, 7, 55, 95, 96], but also in practical experiments and applications [3, 23, 54]. The later may be due to the improvement we are witnessing in processing capabilities of computers. Among these, some employed RNN schemes to solve the optimization problem as proposed and practically employed by Xia et al. in [90, 87, 88, 89]; such as [1, 95]. Some employed a version of LM algorithm in their identification part such as [2, 7, 4, 3], among which work of Atuonwu et al. [7] is more similar to ours. To the best of our knowledge, most of the reported approaches with recurrent schemes in their system identification part either employ a non-linear mapping of states and a linear combination of states as the output or use a form of ARMA methods (NARMA, CARIMA, NARX or NARMAX). Therefore, these architecture are partially recurrent. For instance, Akpan et al. [2, 1] use NARMA and NARX, as well as Lu et al. [54] and Chen et al. [18], however, Lu's architecture is quite alike to Zimmermann's architecture we have already discussed in the previous chapter. Some researchers employed wavelet networks in conjunction with SRN such as Yoo et al. [91, 92], however, their SRWNN architecture is LGRF. Similarly some researchers examined the applicability of Radial-basis functions in recurrent schemes, such as the work conducted by Huang and Lewis [40] or Venkateswarlu and Rao [80]. There are a few earlier works which are comparably interesting, such as work conducted by Zalama et al. [93] in which a recurrent competitive network is employed to adaptively control navigation of a robot or by Zamarreno et al. [94] in which they proposed a very interesting architecture but with few details. In [74], Soloway et al. applied a Newton-Raphson approach to iteratively solve optimization problem of a NNGPC where a feedforward network is employed. Although the system identifier in their context is a feedforward network, it is still interesting for us as one of the earliest attempts and as a first inspiring work.
Our work shall be based on the already proposed system identifier. We will formulate the optimization problem and accordingly we will develop a recursive LM approach to iteratively find solution to the optimization problem, i.e. the control strategy, in a finite receding horizon.

4.2 Problem Formulation

In this section we are going to formulate the control problem as follows. Assume that we have identified the process to be controlled by an LRSI (Eqs. (3.3) and (3.4)), so all the relative assumptions and notations (such as number of states, outputs, etc) hold. Let D be a sequence of desired outputs to be followed by the output of the controlled process

$$D = \big(\mathbf{y}_d(t+1), \mathbf{y}_d(t+2), \dots, \mathbf{y}_d(t+h)\big). \tag{4.1}$$

Note that in Eq. (4.1) at each time instance $(t+i)$ (where $i = 1, \dots, h$), $\mathbf{y}_d(t+i)$ is the vector with n_O elements of desired outputs for the process at that time instance. If the process is single output, then $y_d(t+i)$ is scalar.
In this context, the purpose is to find a control strategy to minimize the following cost function over a finite horizon of h steps

$$\begin{aligned} C &= \frac{1}{2h}\zeta^T\zeta + \frac{1}{2h}\mathbf{\Delta u}^T\mathbf{R}\mathbf{\Delta u} \\ &= C_1 + C_2 \\ C_1 &= \frac{1}{2h}\zeta^T\zeta \\ C_2 &= \frac{1}{2h}\mathbf{\Delta u}^T\mathbf{R}\mathbf{\Delta u}, \end{aligned} \tag{4.2}$$

where ζ and $\mathbf{\Delta u}$ are vector of all errors and changes to control signal over the whole horizon, respectively,

$$\begin{aligned} \zeta &= [\mathbf{e}^T(t+1) \quad \mathbf{e}^T(t+2) \quad \dots \quad \mathbf{e}^T(t+h)]^T \\ \mathbf{\Delta u} &= [\mathbf{\Delta u}^T(t+1) \quad \mathbf{\Delta u}^T(t+2) \quad \dots \quad \mathbf{\Delta u}^T(t+h)]^T \\ \mathbf{u} &= [\mathbf{u}^T(t+1) \quad \mathbf{u}^T(t+2) \quad \dots \quad \mathbf{u}^T(t+h)]^T. \end{aligned} \tag{4.3}$$

Recall that $\mathbf{e}(k)$ represents the error at time k (Eq. (3.18)). $\mathbf{\Delta u}(k)$ represents the change to control signal at time k

$$\mathbf{\Delta u}(k) = \mathbf{u}(k) - \mathbf{u}(k-1). \tag{4.4}$$

Obviously, the size of the vector ζ is $hn_O \times 1$ and for the vector $\mathbf{\Delta u}$ it is $hn_I \times 1$. Accordingly, matrix $\mathbf{R}$ is a diagonal matrix with hn_I number of nonzero elements r_{jj}on the diagonal and zero elements elsewhere. This matrix represents the preference of minimizing change to control actions over minimizing the error. Although this matrix is the designer choice, usually the elements are chosen so that $0 < r_{jj} < 1$.
Some authors [74] differentiate between the prediction horizon and control horizon. The difference rises from the fact that we hope to gradually decrease and finally set the control actions to zero. Hence, after the control horizon, the control signal is assumed to be zero and the process works unperturbed. However, in our context this difference does not play a significant role and we will assume one horizon for both prediction and control. Next we propose a solution to this optimization problem. Note that there might be some constraint applicable to the optimization problem. Taking into account these constraints make the optimization procedure even more complex. In this work we shall assume there are no effective constraints.

4.3 LM Iterative Solution to the Optimization Problem

In this section we attempt to propose a solution to the previously stated problem. Repeatedly reminding, an effective approach to solve a nonlinear optimization problem is the iterative approach. Here, since the cost function C depicted by Eq. (4.2) is inherently nonlinear and not convex, we adopt an iterative approach as well. To this end, we first need to compute the *Jacobians*.

$$J_1 = \frac{\partial \zeta}{\partial \mathbf{u}} \tag{4.5a}$$

$$J_2 = \frac{\partial \Delta \mathbf{u}}{\partial \mathbf{u}} \tag{4.5b}$$

Note that $\mathbf{J}_1$ can be divided into sub-matrices each of which is $n_O \times n_I$.

$$\mathbf{J}_1 = \begin{bmatrix} \frac{\partial y(1)}{\partial \mathbf{u}(1)} & \frac{\partial y(1)}{\partial \mathbf{u}(2)} & \cdots & \frac{\partial y(1)}{\partial \mathbf{u}(h)} \\ \frac{\partial y(2)}{\partial \mathbf{u}(1)} & \frac{\partial y(2)}{\partial \mathbf{u}(2)} & \cdots & \frac{\partial y(2)}{\partial \mathbf{u}(h)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y(h)}{\partial \mathbf{u}(1)} & \frac{\partial y(h)}{\partial \mathbf{u}(2)} & \cdots & \frac{\partial y(h)}{\partial \mathbf{u}(h)} \end{bmatrix}_{hn_O \times hn_I} \tag{4.6}$$

As it is seen, for all k and k′ in $1, 2, \ldots, h$, $\frac{\partial y(k)}{\partial \mathbf{u}(k')}$ is an $n_O \times n_I$ matrix. Hence $\mathbf{J}_1$ is $hn_O \times hn_I$. Note that in Eq. (4.6) we start at $t = 0$ for the sake of simplicity in notation.
Usually, current outputs do not depend on the future control actions. Therefore, $\frac{\partial y(k)}{\partial \mathbf{u}(k')} = \mathbf{0}_{n_O \times n_I}$ for all $k < k'$. From the definition of $\mathbf{y}(k)$ (Eq. (3.4b)) the diagonal sub-matrices ($k' = k$) can be readily computed

$$\frac{\partial \mathbf{y}(k')}{\partial \mathbf{u}(k')} = diag\left(\mathbf{g}'(\mathbf{Pr}(k'))\right)\mathbf{D}. \tag{4.7}$$

To compute $\frac{\partial y(k)}{\partial \mathbf{u}(k')}$ for $k > k'$ we need to obtain a recursive equation as follows. First, pay attention that by the definition of $\mathbf{y}(k)$, we obtain

$$\frac{\partial \mathbf{y}(k)}{\partial \mathbf{u}(k')} = diag\left(\mathbf{g}'(\mathbf{Pr}(k))\right)\left(\mathbf{C}\frac{\partial \mathbf{x}(k)}{\partial \mathbf{u}(k')} + \mathbf{G}\frac{\partial \mathbf{y}(k-1)}{\partial \mathbf{u}(k')}\right). \tag{4.8}$$

Second, we need to obtain the recursive equation for the derivatives of the state with respect to the control actions

$$\frac{\partial \mathbf{x}(k)}{\partial \mathbf{u}(k')} = diag\left(\mathbf{f}'(\mathbf{Ws}(k))\right)\left(\mathbf{A}\frac{\partial \mathbf{x}(k-1)}{\partial \mathbf{u}(k')} + \mathbf{F}\frac{\partial \mathbf{y}(k-1)}{\partial \mathbf{u}(k')}\right). \tag{4.9}$$

Note that the initial values for $\frac{\partial y(k)}{\partial \mathbf{u}(k')}$ are calculated on $k' = k$ which are already computed as the diagonals. To calculate the initial condition for $\frac{\partial \mathbf{x}(k)}{\partial \mathbf{u}(k')}$ we simply set $k' = k$ and take the derivative which yields

$$\frac{\partial \mathbf{x}(k')}{\partial \mathbf{u}(k')} = \mathit{diag}\left(\mathbf{f}'(\mathbf{W}\mathbf{s}(k'))\right)\mathbf{B}. \tag{4.10}$$

Having calculated the derivatives for all k and k' in $1, 2, ..., h$, the first Jacobian, namely $\mathbf{J}_1$ is completed.
To form the second Jacobian, namely $\mathbf{J}_2$, we need to pay attention to the definition of $\boldsymbol{\Delta}\mathbf{u}$. Doing so, we observe that the derivative of each element of $\boldsymbol{\Delta}\mathbf{u}(k)$ vector with respect to $\mathbf{u}(k')$ is either 1 or -1. In fact, it should be obvious that

$$\frac{\partial \Delta u(k)}{\partial u(k')} = \begin{cases} 1 & k = k' \\ -1 & k = k' + 1 \\ 0 & \text{otherwise.} \end{cases} \tag{4.11}$$

Thus, elements of the second Jacobian are all zero except for the diagonals and the ones before them

$$\mathbf{J}_2 = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ -1 & 1 & 0 & \dots & 0 & 0 \\ 0 & -1 & 1 & \vdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ddots & 1 & 0 \\ 0 & 0 & 0 & \dots & -1 & 1 \end{bmatrix}_{hn_I \times hn_I} . \tag{4.12}$$

As mentioned in the previous chapter, in LM algorithm all we need is to obtain the Jacobians. Thus, by the same estimation for the Hessian and employing the Jacobian in calculating the error gradient we obtain the LM update rule for iteratively update $\boldsymbol{\Delta}\mathbf{u}$

$$\begin{aligned} \mathbf{H} &= \frac{1}{h}\left(\mathbf{J}_1^T\mathbf{J}_1 + \mathbf{J}_2^T\mathbf{R}\mathbf{J}_2\right) \\ \nabla\mathbf{C} &= \mathbf{J}_1^T\zeta \\ \boldsymbol{\Delta}\varepsilon_i &= -(\mathbf{H} + \lambda_c \mathit{diag}[\mathbf{H}])^{-1}\nabla\mathbf{C}, \end{aligned} \tag{4.13}$$

where $\boldsymbol{\Delta}\varepsilon_i^T$ is the amount to be added to $\boldsymbol{\Delta}\mathbf{u}$ at the i^{th} iteration. Remember that LM algorithm is an iterative optimization algorithm. So, for a fixed horizon we need to run the iterative algorithm until some stopping criteria are met (Remark 3.3.3). As the iteration stops, the first control signal is applied to both the real plant and the model. Then the new observations are made and the

horizon recedes one step into the future, then the process repeats. Algorithm 5 describes the procedure. Remember that the Jacobian $\mathbf{J}_1$ is partitioned into h sub-matrices, each of which is of the size $n_O \times n_I$.
Essentially, this algorithm consists of two nested loops, the outer one iterates over the samples and the inner one is due to LM algorithm. In fact, in the inner loop we are searching for the optimal control sequence (lines 8:32). When at least one stopping condition is met, this loop finishes and the first input vector of the calculated control strategy is applied to both the plant and the model (line 33). Then the horizon recedes one step by actually shifting the desired trajectory one step back. Enough care should be taken to handle the indexes well and to reuse the network values in prediction phase to reduce the computational load.

Expectedly, the performance of LM-MPC is so much dependent on the validation performance of the identified model. The results presented in the next chapter also confirms that. However, as it will be seen in the last section of the next chapter, we can employ a parallel identification-control strategy, where we can use a premature identified model of the system to have some “rough” idea about how the real plant actually works. Then, by collecting new observations resulting from the control actions, we keep the identification process running while performing the control process to collect new data. By doing so, the parallel identification results in an increasingly better generalization performance, hence better prediction and finally more precise control. Next, we will study our proposed schemes in a number of scenarios, including this parallel identification/control scheme.

Algorithm 5 LM algorithm for MPC using LRSI

```
Require: An LRSI model
Require: A desired trajectory with $n_d$ outputs: $\mathbf{y}^d(k), \quad k = 1, 2, ..., n_d$
Require: A horizon $h \leqslant n_d$
  Initialization:
  Form $\mathbf{J}_2$ according to Eq. (4.12);
  $cnt \leftarrow 1$;
  Main loop:
  while $cnt < n_d$ do
    $\mathbf{u} \leftarrow [0]_{hn_I \times 1}$; $\mathbf{\Delta u} \leftarrow [0]_{hn_I \times 1}$;
    Predict the future of the model for $h$ steps using $\mathbf{u}$;
    Calculate the cost using Eq. (4.2) and store it in $C_{old}$;
    $d_C \leftarrow 1/0$; $\alpha \leftarrow 5.0$; $cnt \leftarrow 1$; $\mathsf{STOP} \leftarrow \mathsf{FALSE}$;
    while $\mathsf{STOP} \neq \mathsf{TRUE}$ do
      for all $k'$ = 1 to $h$ do
        Calculate the initial values for $\frac{\partial x(k')}{\partial \mathbf{u}(k')}$ and $\frac{\partial y(k')}{\partial \mathbf{u}(k')}$ using Eq. (4.10), 4.7;
        Use $\frac{\partial y(k')}{\partial \mathbf{u}(k')}$ for the diagonal sub-matrices of $\mathbf{J}_1$;
        for all $k$ = $k' + 1$ to $h$ do
          Update the derivatives $\frac{\partial x(k)}{\partial \mathbf{u}(k')}$ and $\frac{\partial y(k)}{\partial \mathbf{u}(k')}$ using Eq. (4.9), Eq. (4.8);
          Use the updated $\frac{\partial y(k)}{\partial \mathbf{u}(k')}$ as the $(k, k')$th sub-matrix, where $k$ refers to row and $k'$ refers to column;
        end for
      end for
      Calculate $\mathbf{\Delta}\varepsilon_i$ using Eq. (4.13);
      $\mathbf{\Delta u} \leftarrow \mathbf{\Delta u} + \mathbf{\Delta}\varepsilon_i$;
      Predict the future of the model for $h$ steps using $\mathbf{u}$;
      Calculate the cost using Eq. (4.2) and store it in $C_{new}$;
      if $C_{new} < C_{old}$ then
        $d_C \leftarrow \frac{d_C}{\alpha}$;
        $C_{new} = C_{old}$;
      else
        $d_C \leftarrow d_C \times \alpha$;
        $\mathbf{\Delta u} \leftarrow \mathbf{\Delta u} - \mathbf{\Delta}\varepsilon_i$;
      end if
      $cnt \leftarrow cnt + 1$;
      if (Stopping conditions are met) then
        $\mathsf{STOP} \leftarrow \mathsf{TRUE}$;
      end if
    end while
    Apply $\mathbf{u}(1 : n_I)$ to both the plant and the model;
    $cnt \leftarrow cnt + 1$;
    Shift left the samples of desired trajectory for one step;
  end while
```

Chapter 5
Experiments

In this chapter, performance of our proposed models and methods is examined through a number of experiments. Each of the experiments studies a particular aspect of our scheme, such as identification, control or both in parallel. The models to be used as plants under control are selected in a way to both examine the behaviour of the schemes and reflect their capabilities and weaknesses in the light of other reported performances. Hence, the chosen models are frequently reported and studied (e.g. [46, 81, 62, 45, 28]). Having selected both models and experiment setup precisely similar to previously reported results, we will also perform a few comparisons, where possible. This chapter is divided into 5 sections. After this short introduction, the first section describes configuration and setup of each experiment. Four next sections are devoted to illustrate and discuss the results of four separate experiments.

5.1 Configuration and Setup

We will use the following two models (Eq. (5.1)) as plants to be identified and/or controlled in the experiments to be presented later in this chapter.

$$1\text{: } y(k+1) = 0.35\left[\frac{y(k)y(k-1)(y(k-2)+2.5)}{1+y^2(k)y^2(k-1)} + u(k)\right] \tag{5.1a}$$

$$2\text{: } y(k+1) = 0.72y(k) + 0.025y(k-1)u(k-1) + 0.01u^2(k-2) + 0.2u(k-3) \tag{5.1b}$$

The first two experiments deal with identification of the above models. The block diagram of identification part is shown in Figure 5.1. During identification process, the training input signals are random signals, however, they can be a mixture of random and other signals.[1] After identification process what

[1]In general, an identification process should meet the *persistent of excitation* condition, however, in practice it is almost impossible to check if it is met. Choosing a random excitation signal is an attempt to increase the probability of meeting this condition.

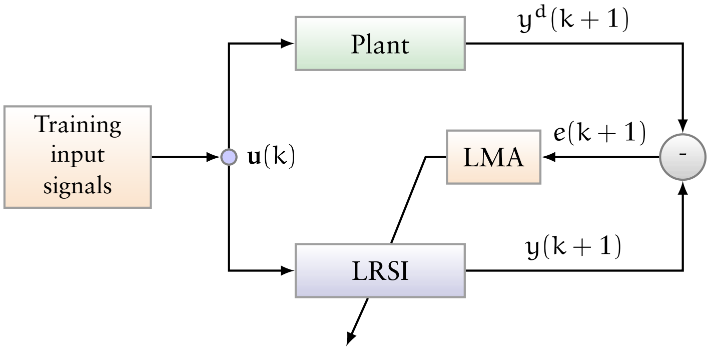


Figure 5.1: System identification configuration

considerably matters is the ability to generalize, i.e., *validation*. In other words, validation is to examine the trained network[2]. In validation, we normally apply a predefined set of input signals to both the plant and the *trained* network and observe the error signal with no learning applied to the network. Although the validation input signals can be arbitrary, without loss of generality we choose the validation input signal as shown in Eq. (5.2) (same as [45, 62, 81]).

$$u(k) = \begin{cases} \sin(\pi\frac{k}{25}) & , \quad k < 250 \\ 1.0 & , \quad 250 \leqslant k < 500 \\ -1.0 & , \quad 500 \leqslant k < 750 \\ 0.3\sin(\pi\frac{k}{25}) + 0.1\sin(\pi\frac{k}{32}) + 0.6\sin(\pi\frac{k}{10}) & , \quad 750 \leqslant k < 1000 \end{cases} \tag{5.2}$$

Although the primary goal of identification experiments is to illustrate the ability of LRSI in identification and predictiction of time-dependent and nonlinear systems, we would also like to investigate dependency of LRSI performance on key parameters, such as prolongation-depth, number of states, decay factor, etc. To this end, the first experiment is run for a number of times, each time with a different parameter setting as will be later pointed out. Although other experiments have been also run with various parameter settings, for them, only the best performances are illustrated.

The third experiment is to assess LM Iterative MPC controller, proposed in Chapter 4, in a control process. The first model is used as the plant to be controlled. Figure 5.2 shows the control configuration. In this figure, those signals that do not have time indexes are the ones that are iteratively engaged in producing the optimal control signal, i.e., $\mathbf{u}(k)$. $\mathbf{r}$ is an input sequence that shapes the reference trajectory after being applied to the Reference Model block. We choose the reference trajectory according to [81], which is re-stated in Eq. (5.3).

[2]In this chapter, network refers to an LRSI network.

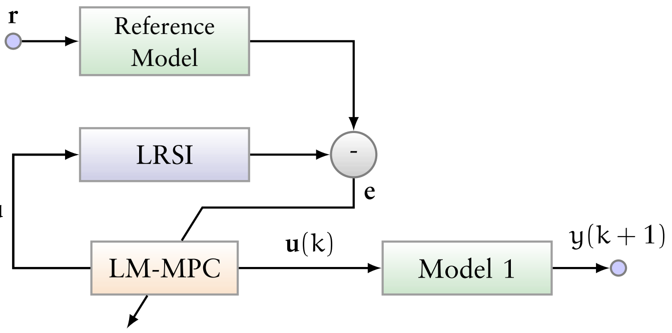


Figure 5.2: MPC using the LRSI and LM-MPC controller

The number of iterations in control process is 2000.

$$y_m(k+1) = 0.6y_m(k) + 0.2y_m(k-1) + 0.1r(k)$$

$$r(k) = \begin{cases} \sin(\pi\frac{k}{25}) & , \quad k < 500 \\ 1.0 & , \quad 500 \leqslant k < 1000 \\ -1.0 & , \quad 1000 \leqslant k < 1500 \\ 0.3\sin(\pi\frac{k}{25}) + 0.1\sin(\pi\frac{k}{32}) + 0.6\sin(\pi\frac{k}{10}) & , \quad k \geqslant 1500 \end{cases} \tag{5.3}$$

A drawback of this standard scheme is that the real output is not taken into account, yet it carries valuable information. The last part of our experiments (Experiment 4) is to show the result of incorporating this information in further identifying the plant while being controlled. This is known as on-line identification, which has a special place in control theory and practice. To do so, a copy of the trained LRSI is added which keeps the identification process going on. The reason for adding this copy, instead of adapting the original LRSI, is to prevent any possible deterioration of the prediction ability of the original one due to unexpected circumstances, as well as using the parallelism in control and identification. However, eventually these two LRSI can merge into one. Figure 5.3 demonstrates this configuration.

The last point to mention is the selected measure to assess the results, which is chosen to be RMSE (Root Mean Squared Error) and is formulated in 5.4.

$$\mathrm{RMSE} = \sqrt{\sum_{k=1}^{n_D} \frac{1}{n_D}\big(y(k) - y_m(k)\big)^2}, \tag{5.4}$$

where $y(k)$ and $y_m(k)$ are the actual and reference outputs at time k , respectively, and n_D is the number of samples. We may also use MSE, which is similar to RMSE but without taking the root of the summation.

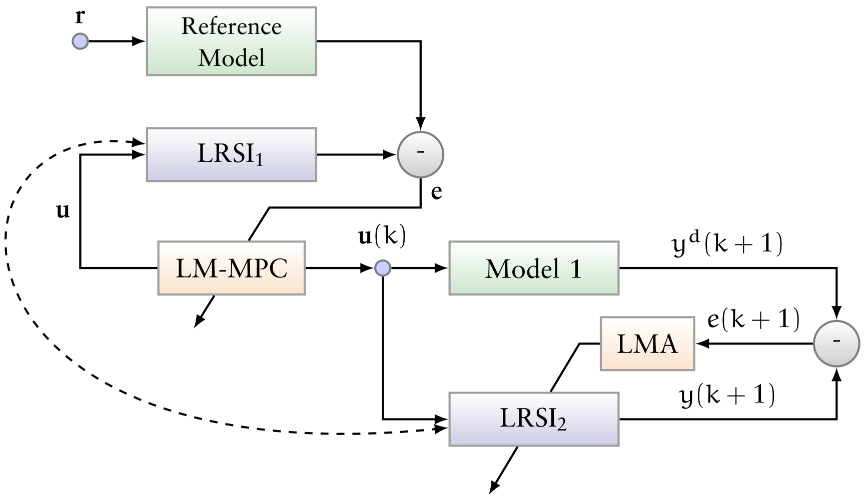


Figure 5.3: Parallel system identification and MPC

5.2 Experiment 1

In Figures 5.4- 5.8 we demonstrate the performance of LRSI in identifying the first model with five different parameter settings. In each of these five plots, we sketch the RMS of identification error and validation error signals with respect to the number of training samples. This experiment is set and run as follows

1. Initialize the network to a given parameter set,
2. Train the network by applying the first 200 samples,
3. Save the weight values,
4. Repeat this sequence (2 and 3) until all training samples are fed to the network.

Following the above algorithm, after the experiment is carried out, essentially we have $\frac{n_D}{200}$ trained network. For each of these networks, we obtain the RMS value for identification and validation errors.
In principle, there are six influential parameters:

1. T: The prolongation depth measured in time steps.
2. n_X: Number of states, i.e. hidden neurons.
3. n_D: Number of training samples.
4. λ_f: Forgetting factor.

5. $\alpha_{\mathbf{W}}$: Regularization (weigth-decay) factor for **W**.

6. $\alpha_{\mathbf{P}}$: Regularization (weigth-decay) factor for **P**.

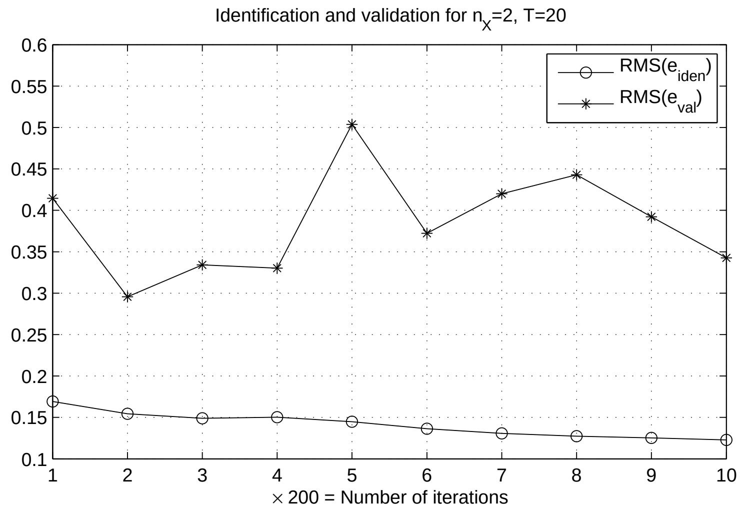


Figure 5.4: (Experiment 1) Identification of Model1 using LRSI with $T = 20$ and $n_X = 2$ and validation results.

However, only the top four ones are critical. A simple strategy can be employed to decrease the sensitivity of LRSI to the two last parameters, i.e. regularization factors. This strategy is to choose a near one initial value for each and decrease them by a factor of, say 10, each approximately 50-100 iterations. Otherwise, a few runs may be needed to find a good choice for these numbers. A rule of thumb is that the higher the number of states, the higher the generalization factors.

To investigate the influence of critical parameters, we need to analyze the aforementioned five plots. As expected, for all of the plots, $\text{RMS}(e_{iden})$ decreases as the network is exposed to more training samples, which is obvious: more information is provided to the network. Thus, the overall form of $\text{RMS}(e_{iden})$ is decreasing. The second observation to be made is the fact that increasing the number of states does not necessarily yield a better generalization performance, nor the number of samples. Although moderately increasing them will eventually lead to a best performance, which in our experiment happens to be at $n_X = 5$, $T = 50$, $n_D = 600$. The corresponding actual output (validation) and reference output are shown in Figure 5.9. As illustrated in Figure 5.6, the RMS value for validation error of this identified model is $\text{RMS}(e_{val}) = 0.0098$.

Another important point to highlight is about the "memory" of LRSI. Note that as more samples are given to the networks, we may not observe a better

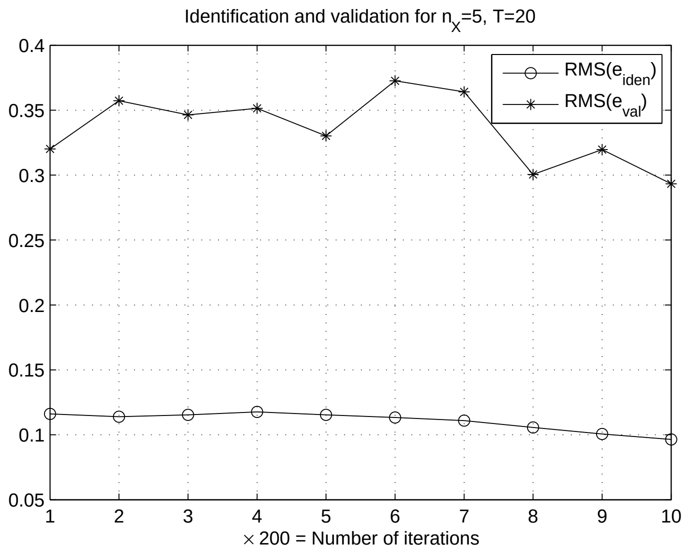


Figure 5.5: (Experiment 1) Identification of Model1 using LRSI with $T = 20$ and $n_X = 5$ and validation results.

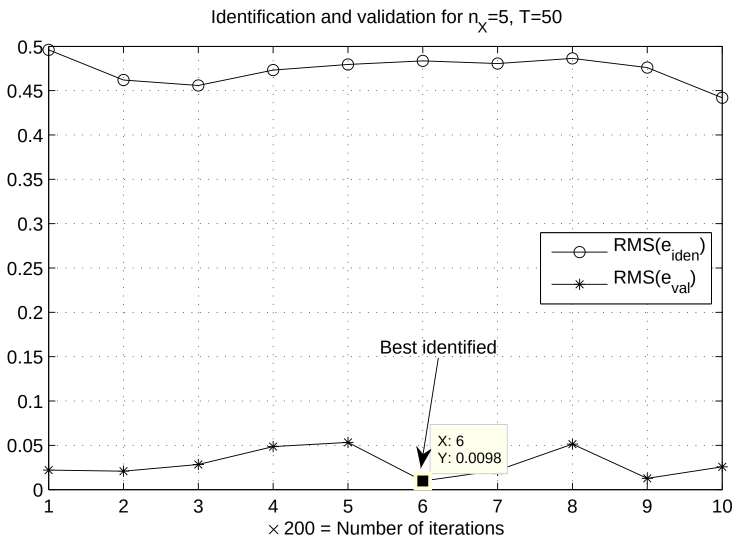


Figure 5.6: (Experiment 1) Identification of Model1 using LRSI with $T = 50$ and $n_X = 5$ and validation results.

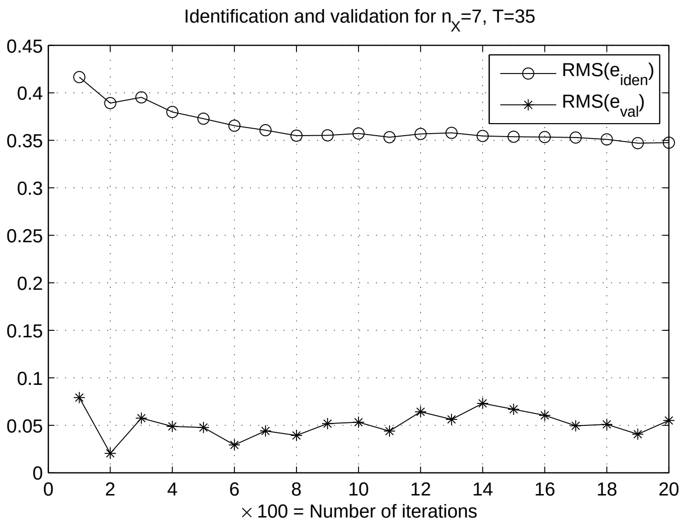


Figure 5.7: (Experiment 1) Identification of Model1 using LRSI with $T = 35$ and $n_X = 7$ and validation results.

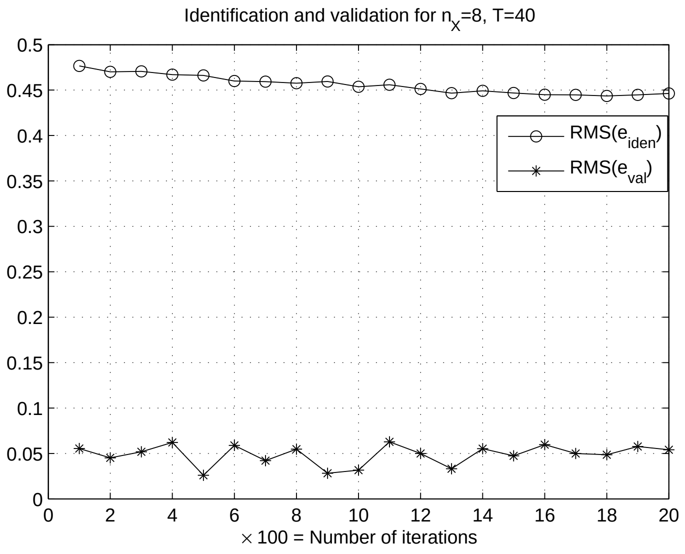


Figure 5.8: (Experiment 1) Identification of Model1 using LRSI with $T = 40$ and $n_X = 8$ and validation results.

generalization performance. That is due to two main facts: 1) memory of LRSI is limited, 2) training input signals are random. It is not yet possible to estimate the memory for our schemes. Of course, by memory we do not mean an explicit database of information, but saving important characteristics of the process to be modeled. If we set the forgetting factor to one, which gives no degradation of saved information in Jacobians, eventually the amount of accumulated information degrades the network performance, because new information are as worth as the oldest ones. Therefore, the network cannot prefer new information over the old ones. On the other hand, if we set the forgetting factor to a number less than one, which dictates a degradation of information in Jacobians, while the training input signals are random, then it is possible to not having stimulated the plant enough to discover all its underlying nonlinearities in a given time-depth. To conclude, we should emphasize that a reasonable trade-off should be made in choosing forgetting factor versus number of samples. An estimated value for the forgetting factor is between 0.9 and 0.7.

The last point to mention is about the prolongation depth. The prolonga-

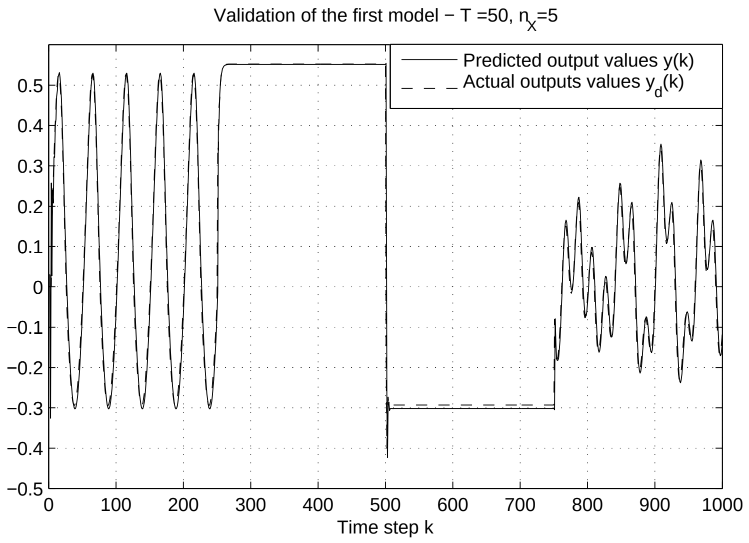


Figure 5.9: (Experiment 1) Validation performance for the best identified model, which is pointed out in Figure 5.6.

tion depth actually determines how much information of the past behaviour of the system is going to contribute in one single weight update. One immediate conclusion is the longer the better, but longer prolongation depth dictates more computation time, hence slower weight update rate. However, in networks with many states, longer prolongation-depth only imposes longer learning, yet yields no better generalization.

In general, choosing LRSI parameters can play fundamental role in their identification performance. We would like to conclude that number of states and prolongation depth are the most important parameters. Other parameters such as forgetting factor and, sometimes, regularization factors also play important roles in specific scenarios but not as significant as the number of states and prolongation depth.

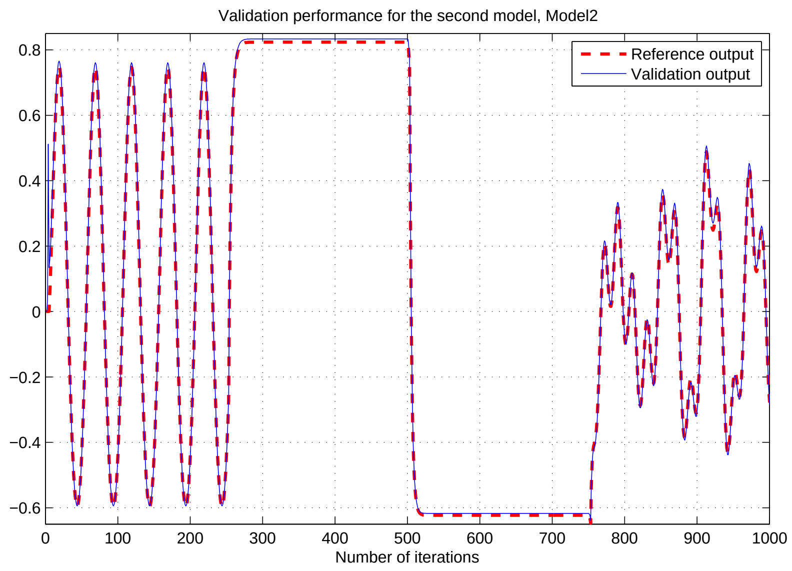


Figure 5.10: (Experiment 2) Validation output of identified version of Model2, with $n_X = 4, T = 30, n_D = 800$.

5.3 Experiment 2

In this experiment we intend to demonstrate how LRSI will perform in identifying a more complex time-dependent system, namely the second model (Model2-eq5.1b). Note that the delays are longer and additional nonlinearity is introduced over the delayed version of the input signal. Figures 5.10 and 5.11 illustrate the validation output and validation error, respectively. Figure 5.12 also shows the identification and validation RMSEs over a number of different training samples. The best RMSE for validation and identification are 0.02057 and 0.0938, respectively. Comparing this result with the reported results in [45], we observe that our result is slightly better, while we maintain approximately the same number of parameters (note TableIII in [45]). Since $n_X = 4$ and we use a fully connected LRSI, then the number of parameters which correspond to the best performance illustrating in Figures 5.10, 5.11 and 5.12 is

$n_X \times n_S + n_O \times n_R = 28 + 7 = 35$. However, the number of parameters in [45] (LRFNN-SVR), according to their report is 29.

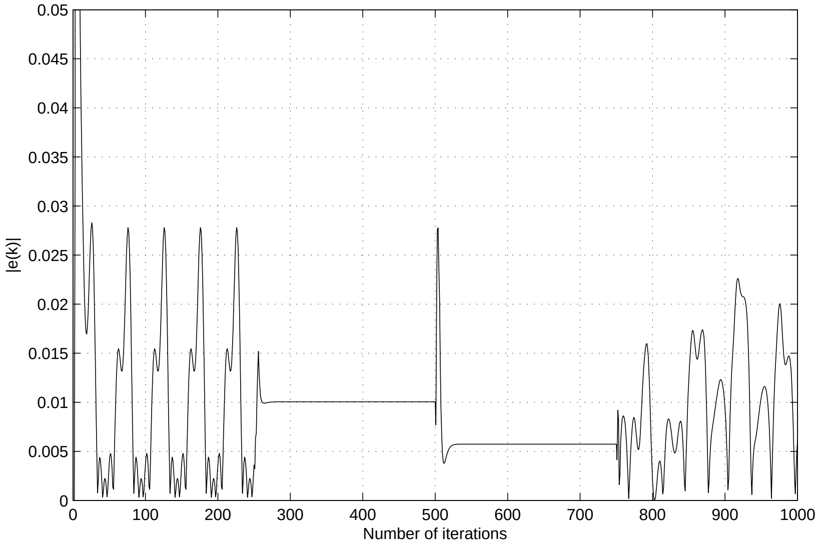


Figure 5.11: (Experiment 2) $\mathrm{RMS}(e_{val})$ of identified version of Model2, with the configuration depicted in Figure 5.10.

5.4 Experiment 3

In this experiment we would like to assess the ability of proposed iterative LM-MPC controller (chapter 4) in controlling a time-dependent plant. We choose the first model and demonstrate the control performance of LM-MPC on the identified version of model one. The result will be then compared with the results presented in [81]. Wang et al. has already shown that their approach yields a better result comparing to Narendra's and Jaung's [46, 62, 81]. The comparison is based on MSE criterion. Our result is shown in Figure 5.13. The absolute value of error signal is also plotted in Figure 5.14.

We observe that our approach yields $\mathrm{MSE} = 2.5158 \times 10^{-4}$ which seems better than FARNN_II controller which is reported to be 8.041×10^{-4} [81]. However, we still intend to improve the performance of our scheme by applying a parallel system identification. This approach is presented next.

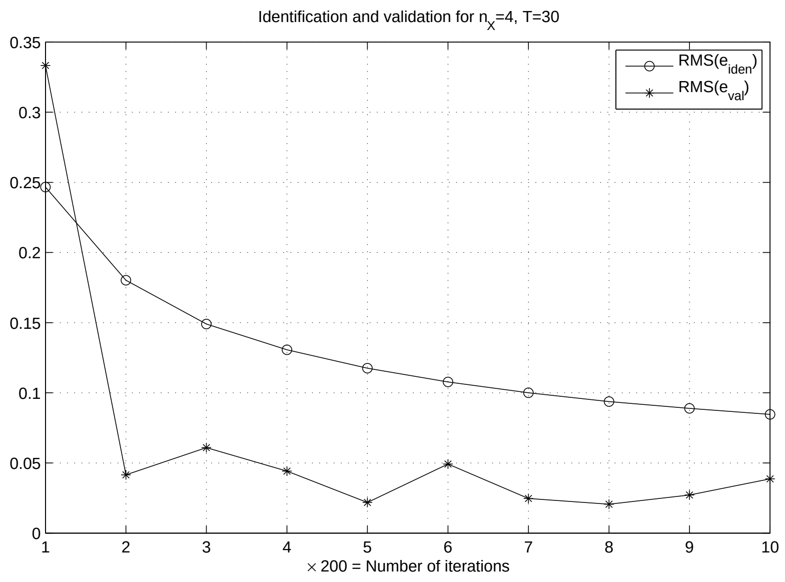


Figure 5.12: (Experiment 2) RMS of identification and validation error of Model2 using LRSI with $T = 30$, $n_X = 4$.

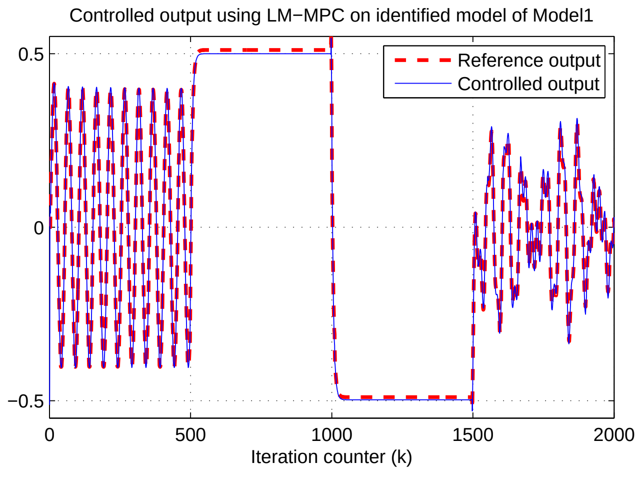


Figure 5.13: (Experiment 3) Applying LM-MPC to the identified model of Model1, with $T = 30$, $n_X = 5$, $n_D = 300$.

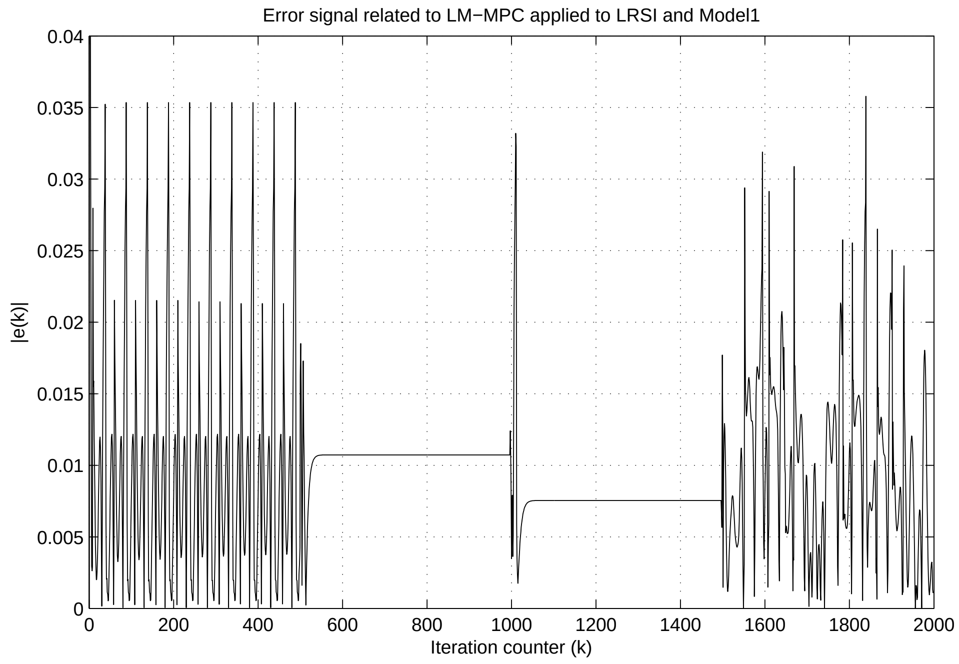


Figure 5.14: (Experiment 3) Error signal between the controlled output and the reference signal for Figure 5.13.

5.5 Experiment 4

In this experiment, we intend to demonstrate that using LRSI in parallel with LM-MPC will lead to a much better control result. We also show that in order to use our scheme, we do not need to have the best possible identified model. We choose a moderately satisfactory identified LRSI from the first experiment ($n_X = 5, T = 30, n_D = 200$) and copy it to be a parallel identifier ($LRSI_2$) as described in section 1 (Figure 5.3). Note that in this approach we will not immediately use $LRSI_2$ as the predictor. The obvious reason is that LRSI is essentially time-dependent. Hence, we need to collect some data *sequentially*, and use them as the training samples. For instance, if the prolongation depth of $LRSI_2$ is 20 it means that from an initial point we need to have 20 training samples to be able to train $LRSI_2$. Even after collecting and applying the first T samples, it is not guaranteed that the second identifier ($LRSI_2$) yields a better prediction. We need to test it for an immediate future. Therefore, in this experiment, LM-MPC starts to control the plant using the predictions come from $LRSI_1$.

After enough data ($n_D = T$) is collected for LRSI$_2$, it starts to learn, but not yet

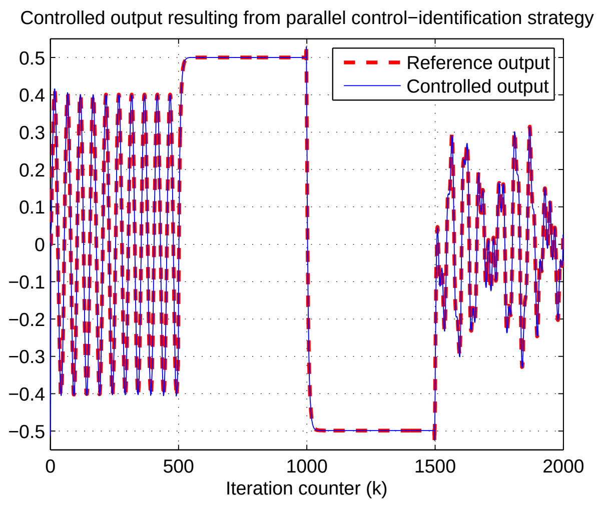


Figure 5.15: (Experiment 4) Error signal from parallel control-identification scheme.

being used as the LM-MPC predictor. As more data are collected, LRSI$_2$ learns more. After a while, LRSI$_2$ starts to perform better, and therefore, LM-MPC will start to shift over LRSI$_2$ to use it as the predictor and eventually LRSI$_2$ and LRSI$_1$ become one. For our scenario, this shift is taken place completely at 100th iteration, which is illustrated in Figure 5.15. The output is shown in Figure 5.16, however it is not much informative since the controlled output and reference output are very much close. The MSE resulted from this scenario is 1.4574×10^{-4}. But if we exclude the first 100 samples in computing MSE then MSE for iteration 100 to iteration 2000 will be 4.0546×10^{-6} which is a significant improvement.

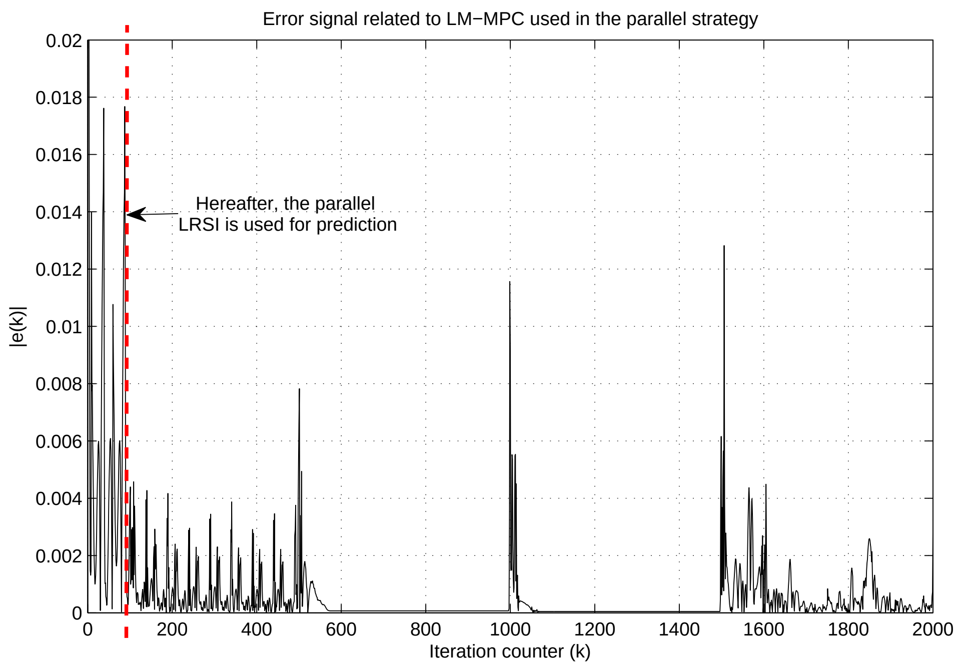


Figure 5.16: (Experiment 4) Controlled and referenced outputs from the parallel LM-MPC experiment.

Chapter 6
Conclusion and future work

In this thesis, we attempted to investigate RNN capabilities in identifying and control of time-dependent systems. We proposed a class of recurrent network system identifiers, RSI, along with a learning algorithm based on LM. We also developed a model predictive controller, based on a linear version of RSI (LRSI), in which the optimization problem was carried out using an iterative LM scheme. Through simulations, we demonstrated the capabilities and efficiency of our proposed schemes. In this short, ending chapter, we intend to review our schemes from a practical and implementation point of view. We also present a few ideas towards possible future improvements.

6.1 Summary

RSI is basically an RNN that is purposefully shaped to fit in the state-space representation of the identified system. Although RSI in its presented form is FCRNN, by using simple and straightforward manipulation it can be formed to a non-fully connected RNN, hence reduce the parameters size and computational efforts for learning. For a rapid learning convergence, the LM based optimization algorithm seems one of the best choices. We derived the LM algorithm for a linear case of RSI. In the development of this learning algorithm, we tried to keep the derivations as simple as possible. The basic algorithm was comprehensively described to be readily applicable in a practical implementation. Additionally, a few improvements are suggested to improve the performance of the fundamental scheme. However, similar to other global optimization methods, in the learning algorithm we proposed, we need to take into account almost all of the network parameters for a single parameter update. That is an intrinsic problem of global optimization methods that is alleviated in back-propagation methods. However, BP in contrast converges more slowly and is more vulnerable to trap into local minima. Although comparably, our LM scheme is computationally expensive, it is still tuned to be applicable to on line learning and system identification.

Another approach we employed to mitigate the computational expensiveness of our approach is to exploit a premature identified version of an LRSI in parallel to controlling the system at hand. This approach yields to a parallel identification-and-control mechanism that distribute the computational load by reducing the identification time and prolongation depth of learning. The results of this approach are shown in the last experiment of chapter 5 - Experiment 4- and demonstrated a superior performance comparing some existing approaches.
One of the main drawbacks, which is also common to other iterative optimization methods, is the sensitiveness of convergence to initial guess of parameters. As the number of hidden states increases, so the parameters space enlarges, this sensitiveness exponentially increases. To the best of our knowledge, there is no universal remedy for this drawback, except using committees and/or running the identification process for a number of times and choose the one with the best validation performance. We also use weight decay regularization to allow for slightly larger parameter space than needed, which may result in less trials and better generalization.

6.2 Future improvements and further works

It is desired to have more experiments running on the schemes. Particularly, we have not applied noisy data to observe the behaviour of LRSI and LM-MPC exposing to noisy situations. Additionally, the models that have been used in studying the performance of our proposed schemes are artificial and adopted based on their popularity. It is more desired to investigate the performance of our schemes in more practical problems. Hence, these two immediate future works seem most important to us.
There are several ideas yet to be incorporated into the present scheme. We will list afew of them below.

Tensors Using tensors, we might reach to a more robust learning algorithm capable of faster convergence, however, with a higher computational complexity. Specifically in representing and mathematically manipulating the Jacobians $\mathrm{J}_{\mathrm{W}}^{\mathrm{x}}$, tensors may be a better and more powerful substitutions. However, a comprehensive study over them is needed.

Delay blocks We might incorporate more than one delay block for some or all of the signals, but this increases the parameter space dimension. However, adding only one more step of delay makes it possible to use Gaussian function as the activation functions. Using Gaussian functions, we can apply the proposed schemes in kernel-based classification problems, where past observations can play the role of kernels.

Stability analysis A comprehensive stability analysis is needed both to investigate the behaviour of LRSI and to extend the parameter choices for learning algorithm.

BPTT BPPT version of learning algorithm can also be developed and implemented to RSI and LRSI then the performance of LM and BPTT can be practically compared.

References

[1] V. A. Akpan and George D. Hassapis. Nonlinear model identification and adaptive model predictive control using neural networks. *ISA Transactions*, 50(2):177 – 194, 2011. (Cited on page 52.)

[2] V.A. Akpan and G. Hassapis. Adaptive predictive control using recurrent neural network identification. In *Control and Automation, 2009. MED '09. 17th Mediterranean Conference on*, pages 61 –66, june 2009. (Cited on page 52.)

[3] R.K. Al Seyab and Yi Cao. Differential recurrent neural network based predictive control. *Computers & Chemical Engineering*, 32(7):1533 – 1545, 2008. (Cited on page 52.)

[4] R.K. Al Seyab and Yi Cao. Nonlinear system identification for predictive control using continuous time recurrent neural networks and automatic differentiation. *Journal of Process Control*, 18(6):568 – 581, 2008. (Cited on page 52.)

[5] Michael A Arbib. *The Handbook of Brain Theory and Neural Networks*. The MIT Press, 2006. (Cited on pages 1, 18, and 21.)

[6] A.F. Atiya and A.G. Parlos. New results on recurrent network training: unifying the algorithms and accelerating convergence. *Neural Networks, IEEE Transactions on*, 11(3):697 –709, may 2000. (Cited on page 30.)

[7] J.C. Atuonwu, Y. Cao, G.P. Rangaiah, and M.O. Tade. Identification and predictive control of a multistage evaporator. *Control Engineering Practice*, 18(12):1418 – 1428, 2010. (Cited on page 52.)

[8] P. Bram Backer. *The State of Mind Reinforcement Learning with Recurrent Neural Networks*. PhD thesis, Universiteit Leiden. (Cited on pages 17 and 19.)

[9] N.E. Barabanov and D.V. Prokhorov. Stability analysis of discrete-time recurrent neural networks. *IEEE Transactions on Neural Networks*, 13(2):292–303, 2002. (Cited on page 9.)

[10] N.E. Barabanov and D.V. Prokhorov. A new method for stability analysis of nonlinear discrete-time systems. *IEEE Transactions on Automatic Control*, 48(12):2250–2255, 2003. (Cited on page 9.)

[11] I. Baruch, J.M. Flores Albino, and B. Nenkova. A recurrent neural network approach for identification and control of nonlinear objects. *Problems of Engineering Cybernetics and Robotics*, (51):57–64, 2001. (Cited on page 31.)

[12] I. Baruch and C.R. Mariaca-Gaspar. A levenberg-marquardt learning applied for recurrent neural identification and control of a wastewater treatment bioprocess. *International Journal of Intelligent Systems*, 24:1094–1114, 2009. (Cited on page 31.)

[13] A. Bhaya, E. Kaszkurewicz, and V.S. Kozyakin. Existence and stability of a unique equilibrium in continuous-valued discrete-time asynchronous hopfield neural networks. *IEEE Transactions on Neural Network*, 7(2):620–628, 2003. (Cited on page 9.)

[14] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, OXFORD, 1995. (Cited on pages 1 and 18.)

[15] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. (Cited on pages 7, 18, 21, 22, and 47.)

[16] M. Boroushaki, M.B. Ghofrani, C. Lucas, and M.J. Yazdanpanah. Identification and control of a nuclear reactor core (vver) using recurrent neural networks and fuzzy systems. *Nuclear Science, IEEE Transactions on*, 50(1):159 – 174, feb 2003. (Cited on page 30.)

[17] L.W. Chan and C.C. Szeto. Training recurrent network with block-diagonal approximated levenberg-marquardt algorithm. In *IJCNN'99*, Washington D.C., jul. 1999. (Cited on page 30.)

[18] Xin Chen and Yangmin LI. Neural network predictive control for mobile robot using pso with controllable random exploration velocity. *International Journal of Intelligent Control and Systems*, 12(3), 2007. (Cited on page 52.)

[19] Lee Chi-Yung, C.H. Lin, C.H Chen, and C.L. Chang. Dynamic system identification using a recurrent compensatory fuzzy neural network. *Fuzzy Systems, IEEE Transactions on*, 8(4):349 –366, aug 2000. (Cited on page 30.)

[20] D.W. Clarke, C. Mohtadi, and P.S. Tuffs. Generalized predictive control-part i. the basic algorithm. *Automatica*, 23(2):137–148, 1987. (Cited on pages 2 and 51.)

[21] D.W. Clarke, C. Mohtadi, and P.S. Tuffs. Generalized predictive control-part ii. extensions and interpretations. *Automatica*, 23(2):149–160, 1987. (Cited on pages 2 and 51.)

[22] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, (2):303–314, 1989. (Cited on page 29.)

[23] K. Dalamagkidis, K.P. Valavanis, and L.A. Piegl. Nonlinear model predictive control with neural network optimization for autonomous autorotation of small unmanned helicopters. *Control Systems Technology, IEEE Transactions on*, 19(4):818–831, july 2011. (Cited on page 52.)

[24] I.N. Daliakopoulosa, P. Coulibalya, and I.K Tsanisb. Groundwater level forecasting using artificial neural networks. *Elsevier Journal of Hydrology*, (309):229 –240, 2005. (Cited on page 31.)

[25] O. De Jesús. *Training General Dynamic Neural Networks*. PhD thesis, Oklahoma State University, 2002. (Cited on page 31.)

[26] A. Delgado, C. Kambhampati, and K. Warwick. Dynamic recurrent neural network for system identification and control. *Control Theory and Applications, IEE Proceedings -*, 142(4):307 –314, July 1995. (Cited on page 30.)

[27] J. L. Elman. Finding structure in time. *Cognitive Science*, (14):179–211, 1990. (Cited on page 13.)

[28] C. Endisch, C. Hackl, and D. Schröder. System identification with general dynamic neural networks and network pruning. *International Journal of Information and Mathematical Sciences*, 3(4):187–195, 2008. (Cited on pages 31 and 59.)

[29] C. Endisch, P. Stolze, P. Endisch, C. Hackl, and R. Kennel. Levenberg-marquardt-based obs algorithm using adaptive pruning interval for system identification with dynamic neural networks. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 3402 –3408, oct. 2009. (Cited on page 31.)

[30] Z. Feng and Michel A.N. Robustness analysis of a class of discrete-time recurrent neural networks under perturbations. *IEEE Transactions on Circuits and Systems I, Dundamentals and Theory Application*, 46(12):1482–1486, 1999. (Cited on page 9.)

[31] Qiang Gan and C.J. Harris. Linearization and state estimation of unknown discrete-time nonlinear dynamic systems using recurrent neuro-fuzzy networks. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 29(6):802 –817, dec 1999. (Cited on page 30.)

[32] James Garson. Connectionism. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2010 edition, 2010. (Cited on page 7.)

[33] M.A. Gonzalez-Olvera and Yu Tang. Black-box identification of a class of nonlinear systems by a recurrent neurofuzzy network. *Neural Networks, IEEE Transactions on*, 21(4):672 –679, april 2010. (Cited on page 30.)

[34] R. Griñó, G. Cembrano, and C. Torras. Nonlinear system identification using additive dynamic neural networks-two on-line approaches. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 47(2):150 –165, feb 2000. (Cited on page 30.)

[35] Nihal Fatma Güler, Elif Derya íbeyli, and İnan Güler. Recurrent neural networks employing lyapunov exponents for eeg signals classification. *Expert Syst. Appl.*, 29:506–514, October 2005. (Cited on page 31.)

[36] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice-Hall, 1999. (Cited on pages 1, 7, 8, 10, 15, 17, 18, 19, 20, 21, 22, 25, 30, 32, 36, and 47.)

[37] D.O. Hebb. *The organization of Behavior: A Neuropsychological Theory*. Newyork: Wiley, 1949. (Cited on page 18.)

[38] R. Herbrich and R.C. Williamson. *The Handbook of Brain Theory and Neural Networks*, chapter III, Learning and Generalization: Theoretical Bounds, pages 619–623. The MIT Press, Cambridge, Massachusetts, 2 edition, 2003. (Cited on page 19.)

[39] S. Hu and J. Wang. Global stability of a class of discrete-time recurrent neural networks. *IEEE Transactions on Circuits and Systems I, Dundamentals and Theory Application*, 49(8):1104–1117, 2002. (Cited on page 9.)

[40] Jin-Quan Huang and F.L. Lewis. Neural-network predictive control for nonlinear dynamic systems with time-delay. *Neural Networks, IEEE Transactions on*, 14(2):377 – 389, mar 2003. (Cited on page 52.)

[41] J.-S.R. Jang, C.-T Sun, and E. Mizutani. *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Printice-Hall Inc., 1997. (Cited on pages 21, 36, 37, and 42.)

[42] L. Jin, P.N. Nikiforuk, and M.M. Gupta. Absolute stability conditions for discrete-time recurrent neural networks. *IEEE Transactions on Neural Networks*, 46(12):954–964, 1994. (Cited on page 9.)

[43] L. Jin, P.N. Nikiforuk, and M.M. Gupta. Global asymptotic stability of discrete-time analog neural networks. *IEEE Transactions on Neural Networks*, 7(6):1024–1031, 1996. (Cited on page 9.)

[44] M. Jordan. Serial order: A parallel distributed processing approach. Technical Report 8604, Institute for cognitive science, San Diego, La Jolla, CA, 1986. (Cited on page 13.)

[45] Chia-Feng Juang and Cheng-Da Hsieh. A locally recurrent fuzzy neural network with support vector regression for dynamic-system modeling. *Fuzzy Systems, IEEE Transactions on*, 18(2):261 –273, april 2010. (Cited on pages 30, 59, 60, 67, and 68.)

[46] Chia-Feng Juang and Chin-Teng Lin. A recurrent self-organizing neural fuzzy inference network. *Neural Networks, IEEE Transactions on*, 10(4):828 –845, jul 1999. (Cited on pages 59 and 68.)

[47] Hassan K. Khalil. *Nonlinear Systems*. Prentice-Hall, 3 edition, 2001. (Cited on page 2.)

[48] John F. Kolen and Stefan C. Kremer. *A Field Guide to Dynamical Recurrent Networks*. IEEE Press, New York, 2001. (Cited on page 32.)

[49] Chao-Chee Ku and K.Y. Lee. Diagonal recurrent neural networks for dynamic systems control. *Neural Networks, IEEE Transactions on*, 6(1):144 –156, jan 1995. (Cited on page 30.)

[50] Ching-Hung Lee and Ching-Cheng Teng. Identification and control of dynamic systems using recurrent fuzzy neural networks. *International Journal of Control, Automation, and Systems*, 6(5):755 –766, oct 2008. (Cited on page 30.)

[51] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *The Quarterly of Applied Mathematics*, (2):164–168, 1994. (Cited on page 36.)

[52] C.J. Lin and C.C. Chin. Prediction and identification using wavelet-based recurrent fuzzy neural networks. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(5):2144 –2154, oct. 2004. (Cited on page 30.)

[53] D. Liu and Michel A.N. Asymptotic stability of discrete-time systems with saturation nonlinearities with applications to digital filters. *IEEE Transactions on Circuits and Systems I, Fundamentals and Theory Application*, 39(10):798–807, 1992. (Cited on page 9.)

[54] Chi-Huang Lu and Ching-Chih Tsai. Adaptive predictive control with recurrent neural network for industrial processes: An application to temperature control of a variable-frequency oil-cooling machine. *Industrial Electronics, IEEE Transactions on*, 55(3):1366 –1375, march 2008. (Cited on page 52.)

[55] Maciej and ławryńczuk. Computationally efficient nonlinear predictive control based on neural wiener models. *Neurocomputing*, 74(1-3):401 – 417, 2010. (Cited on page 52.)

[56] J.M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002. (Cited on pages 2 and 51.)

[57] Daniel P. Mandic and Jonathon A. Chambers. *Recurrent Neural Networks for Prediction: learning algorithms, architectures and stability*. John Wiley & Sons, 2001. (Cited on pages 1, 10, 12, and 32.)

[58] W.D. Marquardt. An algorithm for least-squares estimation of nonliner parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963. (Cited on page 37.)

[59] F.D. Marques, L.F.R. Souza, Rebolho D.C., A.S. Caporali, E.M.D. BeloLiu, and A.N. Michel. Application of time-delay neural and recurrent neural networks for the identification of a hingless helicopter blade flapping and torsion motions. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, XXVI(2):97–103, 2005. (Cited on page 31.)

[60] P.A. Mastorocostas and J.B. Theocharis. A recurrent fuzzy-neural model for dynamic system identification. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 32(2):176 –190, apr 2002. (Cited on page 30.)

[61] L.E. Jr. McBride and K.S. Narendra. Optimiation of time-varying systems. *IEEE Transactions on Automatic Control*, 10:289–294, 1965. (Cited on page 25.)

[62] K.S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *Neural Networks, IEEE Transactions on*, 1(1):4 –27, mar 1990. (Cited on pages 59, 60, and 68.)

[63] Norman S. Nise. *Control Systems Engineering*. John Wiley & Sons, 3 edition, 2000. (Cited on page 2.)

[64] B.A. Pearlmutter. Learning state-space trajectories in recurrent neural networks. *Neural Computation*, 1:263–269, 1989. (Cited on page 30.)

[65] M.J. Perez-Ilzarbe. A new method for stability analysis of nonlinear discrete-time systems. *IEEE Transactions on Neural Networks*, 9(6):1344–1351, 1998. (Cited on page 9.)

[66] F J Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, 1987. (Cited on page 30.)

[67] Euliano N.R. Principe, J.C. and W.C. Lefebvre. *Neural and Adaptive Systems: Fundamentals Through Simulations*. John Wiley & Sons, 2000. (Cited on pages 7, 8, 10, 18, and 21.)

[68] G.V. Puskorius, L.A. Feldkamp, and L.I. Jr Davis. Dynamic neural network methods applied to on vehicle idle speed control. In *Proceedings of the IEEE*, volume 84, pages 1407–1420, 1996. (Cited on page 15.)

[69] A.J Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Engineering Department, Cambridge, England, 1987. (Cited on page 25.)

[70] R. Rojas. *Neural Networks- A Systematic Introduction*. Springer-Verlag, Berlin, New York, 1996. (Cited on page 30.)

[71] D.R. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *D.E. Rumelhart and J.L. McCleland eds.*, volume 1. Cambridge, MA: MIT-Press, 1986. (Cited on page 48.)

[72] H. Anton M. Schäfer. *Reinforcement Learning with Recurrent Neural Network*. PhD thesis, University of Osnabrück, October, 2008. (Cited on pages 17, 29, 31, and 34.)

[73] J. Schmidhuber and S. Hochreiter. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. (Cited on page 17.)

[74] Donald Soloway and Pamela J. Haley. Neural generalized predictive control:a newton-raphson implementations. In *Proceedings of the 1996 IEEE international symposium on intelligent control*, pages 277–282, September 15-18 1996. (Cited on pages 2, 52, and 53.)

[75] B. Srinivasan, U.R. Prasad, and N.J. Rao. Back propagation through adjoints for the identification of nonlinear dynamic systems using recurrent neural models. *Neural Networks, IEEE Transactions on*, 5(2):213 –228, March 1994. (Cited on page 30.)

[76] R.S Sutton and A.G Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. (Cited on page 19.)

[77] K.C. Tan, H.J. Tang, and Y. Zhang. Global exponential stability of discrete-time neural networks for constrained quadratic optimization. *IEEE Transactions of Automatic Control*, 56:399–406, 2004. (Cited on page 9.)

[78] Y. Tan and M. Saif. Neural-networks-based nonlinear dynamic modeling for automative engines. *Elsevier Journal of Neurocomputing*, 30:129–142, 2000. (Cited on pages 30 and 31.)

[79] Y. Tian, J. Zhang, and J Morris. Optimal control of a batch emulsion copolymerisation reactor based on recurrent neural network models. *Elsevier Journal on Chemical Engineering and Processing*, 41:531–538, 2002. (Cited on page 31.)

[80] Ch. Venkateswarlu and K. Venkat Rao. Dynamic recurrent radial basis function network model predictive control of unstable nonlinear processes. *Chemical Engineering Science*, 60(23):6718 – 6732, 2005. (Cited on page 52.)

[81] Jeen-Shing Wang and Yen-Ping Chen. A fully automated recurrent neural network for unknown dynamic system identification and control. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 53(6):1363–1372, june 2006. (Cited on pages 59, 60, and 68.)

[82] L. Wang and Z. Zu. Sufficient and necessary conditions for global exponential stability of discrete-time recurrent neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 53(6):1373–1380, 1994. (Cited on page 9.)

[83] P.J. Werbos. Generalization of backpropagation with application to gas market model. *Neural Networks*, 1(4):339 – 356, 1988. (Cited on page 30.)

[84] P.J. Werbos. Backpropagation through time: What it does and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990. (Cited on page 25.)

[85] R Williams and D Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989. (Cited on pages 17, 18, 25, and 30.)

[86] R Williams and D Zipser. Gradient-based learning algorithms for recurrent connectionist networks. Technical Report NU-CCS-90-9, 1990. (Cited on page 21.)

[87] Youshen Xia, H. Leung, and Jun Wang. A projection neural network and its application to constrained optimization problems. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 49(4):447 –458, apr 2002. (Cited on page 52.)

[88] Youshen Xia and Jun Wang. A general projection neural network for solving monotone variational inequalities and related optimization problems. *Neural Networks, IEEE Transactions on*, 15(2):318 –328, march 2004. (Cited on page 52.)

[89] Youshen Xia and Jun Wang. A recurrent neural network for solving nonlinear convex programs subject to linear constraints. *Neural Networks, IEEE Transactions on*, 16(2):379 –386, march 2005. (Cited on page 52.)

[90] Youshen Xia, Jun Wang, and D.L. Hung. Recurrent neural networks for solving linear inequalities and equations. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 46(4):452 –462, apr 1999. (Cited on page 52.)

[91] Sung Jin Yoo, Yoon Ho Choi, and Yoon Ho Choi. Stable predictive control of chaotic systems using self-recurrent wavelet neural network. *International Journal of Control, Automation, and Systems,*, 3(1):43 –55, march 2005. (Cited on page 52.)

[92] Sung Jin Yoo, Yoon Ho Choi, and Jin Bae Park. Generalized predictive control based on self-recurrent wavelet neural network for stable path tracking of mobile robots: adaptive learning rates approach. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 53(6):1381 –1394, june 2006. (Cited on page 52.)

[93] E. Zalama, J. Gomez, M. Paul, and J.R. Peran. Adaptive behavior navigation of a mobile robot. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 32(1):160 –169, jan 2002. (Cited on page 52.)

[94] J.M. Zamarreno and P. Vegab. Neural predictive control. application to a highly non-linear system. *Engineering Applications of Artificial Intelligence*, 12:149 –158, oct 1999. (Cited on page 52.)

[95] Yu-Jia Zhai, Ding-Wen Yu, Hong-Yu Guo, and D.L. Yu. Robust air/fuel ratio control with adaptive drnn model and ad tuning. *Engineering Applications of Artificial Intelligence*, 23(2):283 – 289, 2010. (Cited on page 52.)

[96] Liyan Zhang, Mu Pan, and Shuhai Quan. Model predictive control of water management in pemfc. *Journal of Power Sources*, 180(1):322 – 329, 2008. (Cited on page 52.)

[97] Y. Zhang and K.K. Tan. Multistability of discrete-time recurrent neural networks with unsaturating piecewise linear activation functions. *IEEE Transactions on Neural Networks*, 15(2):329–337, 2004. (Cited on page 31.)

[98] Hans-George Zimmermann and Ralph Neuneier. *A Field Guide to Dynamical Recurrent Networks*, chapter 18. IEEE Press, New York, 2001. (Cited on pages 31 and 32.)