ENHANCED RECURRENT NETWORK TRAINING

By

AMIR HOSSEIN JAFARI

Bachelor of Science in Electrical Engineering
Azad University
IRAN
2006

Master of Science in Mechatronics
American University of Sharjah
Sharjah, UAE
2011

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
May, 2016

ENHANCED RECURRENT NETWORK TRAINING

Dissertation Approved:

Dr. Martin T. Hagan

Dissertation Advisor

Dr. Carl Latino

Dr. George Scheets

Dr. Anthony Kable

ACKNOWLEDGMENTS

First and foremost, thanks to the God and the Almighty whose His blessings made me who I am. Only due to His blessings throughout my research I could finish my dissertation.

I can't express myself how thankful I am to my parents for their support and love provided by them throughout my life. This work will remain as a great honor of my life and my heart is filled with nothing but gratefulness to all of you (Leila Moeini and Jafar Jafari).

I would like to express my gratitude to my advisor, Dr.Martin Hagan for his constant encouragement and guidance throughout the research. His technical advice and expertise in the field helped me to cross all the hurdles towards the successful completion of this dissertation. I would also like to thank him for being patient with me during my dissertation and write up.

Besides my advisor, I would like to thank the rest of my dissertation committee : Dr. George Scheets, Dr. Carl Latino, and Dr. Anthony Kable, for their insightful comments and encouragement.

My debt to my elder brother Dr. Reza Jafari is definitely beyond measure and I will be forever grateful for his help.

I would like to thank my colleagues at OSU specially, my best friends for their support and honest friendship.

Acknowledgements reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: Amir Hossein Jafari

Date of Degree: MAY, 2016

Title of Study: ENHANCED RECURRENT NETWORK TRAINING

Major Field: Electrical Engineering

In this dissertation, we introduce new, more efficient, methods for training recurrent neural networks (RNNs). These methods are based on a new understanding of the error surfaces of RNNs that has been developed in recent years. These error surfaces contain spurious valleys that disrupt the search for global minima. The spurious valleys are caused by instabilities in the networks, which become more pronounced with increased prediction horizons. The new methods described in this dissertation increase the prediction horizons in a principled way that enables the search algorithms to avoid the spurious valleys.

The work also presents a novelty sampling method for collecting new data wisely. The clustering method determining when an RNN is extrapolating. The extrapolation occurs when RNN operates outside the region spanned by the training set, adequate performance cannot be guaranteed. The new method presented in this dissertation used the clustering method for extrapolation detection and collecting the novel datas. The training results are improved with the new data set by retraining the RNN.

The Model Reference control is introduced in this dissertation. The MRC is implemented on the simulated and experimental magnetic levitation system.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Artificial Neural Networks can be categorized into two general classes - static and dynamic. Static networks do not have feedback connections and do not have delays. Dynamic networks have memory, so the output depends on inputs, outputs and states. Dynamic networks that have feedback connections are called recurrent neural networks (RNNs). This research focuses on RNNs.

RNNs are used in many practical applications, such as system identification and control [8], long term predictions of chemical processes [9], financial analysis of multiple stock markets [10], filtering and control [11] and phasor detection and adaptive identification [12].

The difficulties in training recurrent neural networks (RNNs) have been well documented (see [13, 14]). One of the reasons for these difficulties is the existence of spurious valleys in the error surfaces of RNNs [2, 15, 16]. These valleys are not related to the true minimum of the surface, or to the problem the RNN is trying to solve. They are strongly dependent on the input sequence in the training data. (If the input sequence changes, even though the system being modeled stays the same, the valleys will move significantly.) Any batch search algorithm is very likely to be trapped in these spurious valleys.

Alternate training methods have been developed to mitigate the effects of these spurious valleys [7, 15]. Because the spurious valleys depend so strongly on the input sequence, one alternate method is to divide the data into multiple subsequences. The subsequences can be alternated during training, which will move the valleys and prevent the algorithm from becoming trapped [15]. Recently, [7] demonstrated a modified procedure, in which

the error gradient associated with each subsequence is monitored during training. Large gradient magnitudes indicate that the training algorithm is located within a spurious valley for those subsequences, and so those subsequences can be removed temporarily from the training process.

Another technique that was introduced in [7] was to increase the prediction horizon gradually during the training process. The initial training segment used a one-step-ahead prediction. This was increased at each training segment, until the prediction horizon during the final training segment covered the full length of the original sequences. This process can require long training times, if the prediction horizon is increased too slowly, but will fail to converge if the prediction horizon is increased too quickly. We are introducing a method that searches for an optimal horizon step at each training segment [17]. We demonstrate the process on a practical system identification problem.

Even after a recurrent network has been successfully trained, satisfactory performance can only be ensured if the network inputs are similar to those in the training set. This is also true for feedforward networks, but extrapolation is a more urgent problem for recurrent networks, where, because of feedback connections, responses can become unstable when network inputs (including feedback signals) fall outside the training set. The process of detecting network inputs that are outside the training set is called novelty detection [18]. We are proposing a type of novelty detection that makes use of self organizing maps (SOMs). We demonstrate that the proposed technique is able to detect incipient network failures and instabilities well before they occur.

We are also going to use the Self Organizing Map (SOM) to collect additional data in order to improve the training procedure. In other words, we are collecting data wisely in the regions where we are extrapolating. It is unlikely that the original data set will effectively cover the full range of conditions where the network will be used. The RNN is extrapolating when network inputs fall outside the space spanned by the training data set. We are going to collect additional training data. Then, we will retrain the RNN network

with the new data combined with the initial training data set. This procedure is known as *novelty sampling* [19]. This will be done in phases until no novel conditions are detected after many additional tests.

Also, we will test all of our new procedures on a physical system - a magnetic levitation system. The experimental data will verify and validate our procedure.

Chapter 2 presents some basic background material on the general types of recurrent networks that we address in this dissertation and will describe the spurious valleys that cause training difficulties. Chapter 3 briefly reviews the recurrent training procedures first introduced in [7], which will form the foundation for the changes presented in this work. The new procedures for determining the horizon step size are introduced in Chapter 4, and the new novelty detection method is presented in Chapter 6. The new procedures are tested through simulation on a practical system identification problem, and the results are shown in Chapter 5. The modeling and control of a simulated magnetic levitation system is described in Chapter 7. Experimental results on a real magnetic levitation system are presented in Chapter 8. Chapter 9 summarizes the work and presents some conclusions and suggestions for future work.

CHAPTER 2

RECURRENT NETWORKS AND SPURIOUS VALLEYS

In this chapter, we briefly introduce Layered Digital Dynamic Networks in section 2.1, and we review some properties of the spurious valleys of recurrent neural networks (RNN) in section 2.2.

2.1 LAYERED DIGITAL DYNAMIC NETWORKS (LDDNs)

Neural networks can be categorized into two classes, static and dynamic networks. Static networks have have no delays and no feedback connections. The output of these networks are computed directly from feedforward connections. Dynamic networks can have feedback connections and contain tapped delay lines. The output of dynamic networks depend on current inputs, outputs, and states of the network.

A general class of dynamic network, the Layered Digital Dynamic Network (LDDN), was first introduced in [20]. The net input $n^m(k)$ for layer m of an LDDN can be computed

$$\mathbf{n}^m(k) = \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(k-d)$$

$$+ \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(k-d) + \mathbf{b}^m \tag{2.1}$$

where $\mathbf{p}^l(k)$ is the lth input to the network at time k, $\mathbf{IW}^{m,l}$ is the input weight between input l and layer m, $\mathbf{LW}^{m,l}$ is the layer weight between layer l and layer m, $\mathbf{b}^m$ is the bias vector for layer m, $DL_{m,l}$ is the set of all delays in the tapped delay line between layer l and layer m, I_m is the set of indices of input vectors that connect to layer m, and L_m^f is the set of indices of layers that connect directly forward to layer m [1].

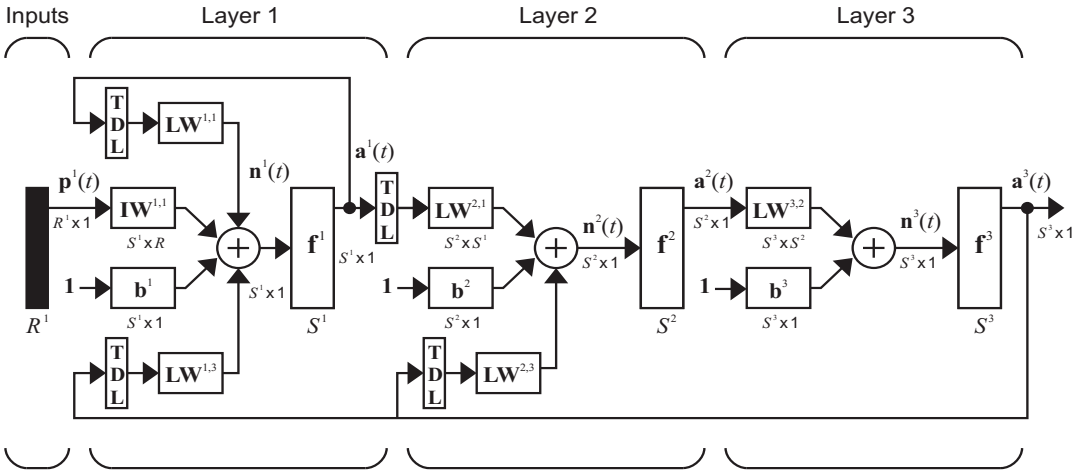


Figure 2.1: Example Dynamic Network [1]

The output of layer m is

$$\mathbf{a}^m(k) = \mathbf{f}^m(\mathbf{n}^m(k)) \tag{2.2}$$

for $m = 1, 2, \cdots, M$, where $\mathbf{f}^m$ is the transfer function at layer m. The set of M paired equations, 2.1 and 2.2, describes the LDDN. LDDNs can have any number of layers, any number of neurons in any layer, and arbitrary connections between layers (as long as there are no zero-delay loops). An example of an RNN represented in LDDN notation is shown in Figure 2.1.

2.2 SPURIOUS VALLEYS

RNNs are typically trained using gradient or Jacobian based optimization algorithms, such as variants of conjugate gradient, quasi-Newton and Levenberg-Marquardt [21]. The gradients and Jacobians are computed using dynamic backpropagation algorithms, such as backpropagation through time, and real-time recurrent learning [20]. Difficulties in gradient based training occur when spurious valleys appear in the error surface [2, 16]. These valleys have a number of critical properties. First, they are unrelated to the true minimum of the error surface and are, instead, principally determined by the input sequence. If a different input sequence is used, the spurious valleys will move to different locations.

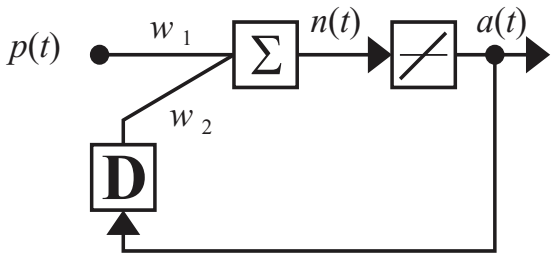


Figure 2.2: One-layer linear network [2]

Secondly, the valleys are located in regions where the network response is unstable. The network response is close to zero at the bottom of the valleys, but slight changes in the network weights will cause the response to increase dramatically. Finally, the widths of the valleys become very narrow as the prediction horizon increases.

To illustrate these spurious valleys, consider a single-neuron recurrent network with a linear transfer function, as shown in Figure 2.2. The equation for the output $a(k)$ is as follows:

$$a(k) = w_1 p(k) + w_2 a(k-1). \tag{2.3}$$

By accumulating the output using (2.3), from a zero initial condition, we have

$$a(k) = w_1 \left[p(k) + w_2 p(k-1) + \ldots + w_2^{k-1} p(1) \right]. \tag{2.4}$$

The term inside the bracket is a polynomial in w_2, with the input sequence as coefficients. If this polynomial has a root outside the unit circle, then at that root the output is zero, even though the output would increase rapidly for a small change in w_2. The interesting thing is that, when this polynomial has a root outside the unit circle at time step k, that root remains the same in the next polynomial at time step $k+1$. This phenomenon is called the "frozen root" [2]. As a result, the output is zero for all future time steps for the same w_2. Since the output for values of w_2 on either side of the root is significantly higher (the system is unstable), a valley appears at the root.

The frozen root phenomenon is illustrated in Figure 2.3, which shows the movement of

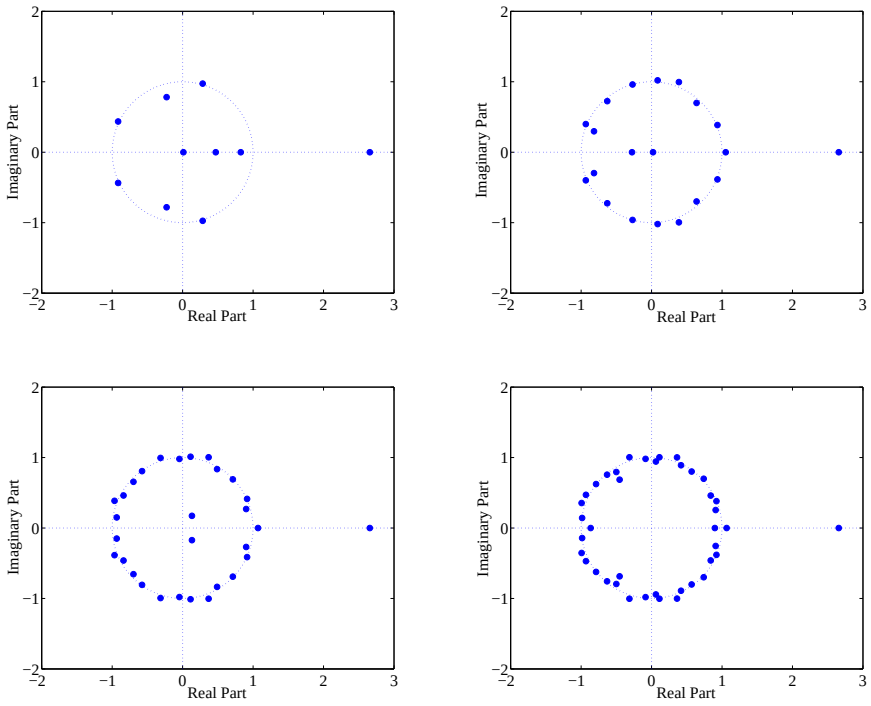


Figure 2.3: Movement of roots as order is increased ($k = 10, 20, 30, 40$) [2]

the roots of a polynomial as the order of the polynomial is increased. Note that when the order is 10, there is one root at approximately 2.66. This root stays in its place as the order of the polynomial is increased, while the other roots move toward the unit circle [2].

An example cross-section of an error surface for this network is shown in Figure 2.4. This figure shows how the surface changes as the prediction horizon increases. You can see that there is a valley that appears at approximately $w_2 = -3.8239$. (This is because the polynomial in Equation 2.4 has a root at $w_2 = -3.8239$.) As the prediction horizon increases, the valley becomes narrower.

As the RNNs become larger, with more layers, neurons, or feedback connections, the valleys become more numerous. An error profile for a practical network is shown in Figure 2.5. The plot shows a cross section of the error surface (the mean square error (MSE) along the gradient direction, where α represents the fractional change in the weights). We can see that there are many valleys in such a small range.We want to escape from this region

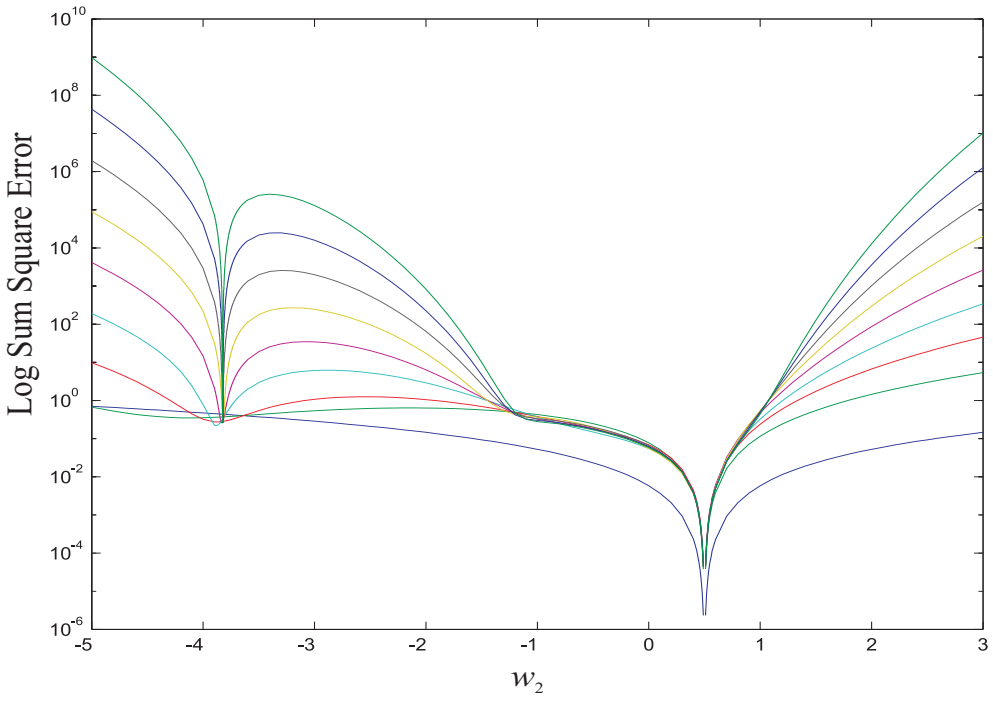


Figure 2.4: Error profile for single neuron network [2]

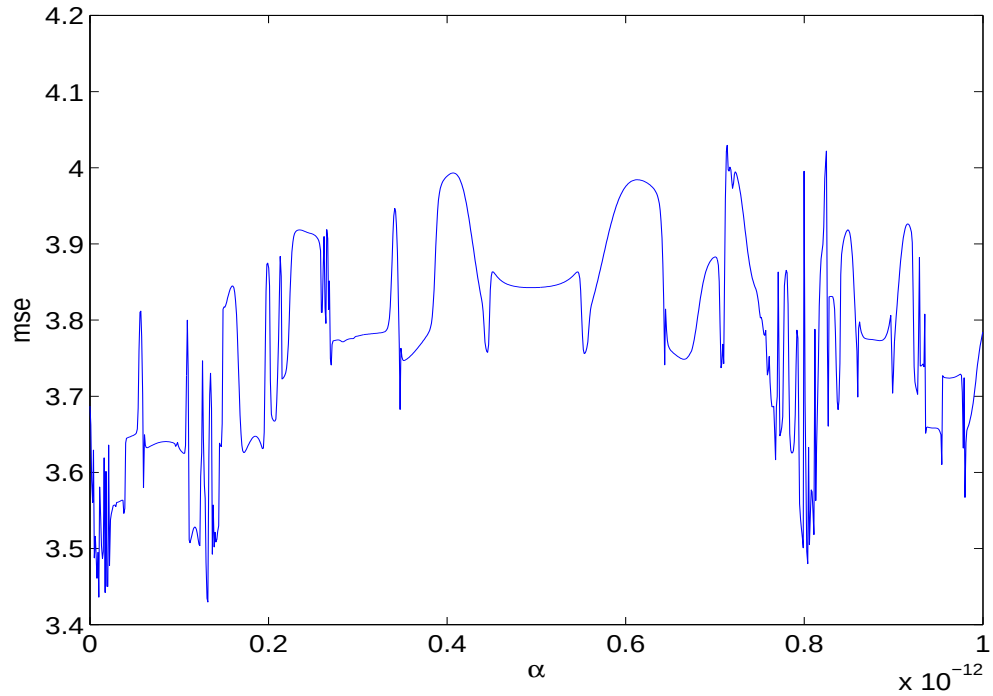


Figure 2.5: Error profile for a practical network [2]

during the training.

CHAPTER 3

MODIFIED RECURRENT NETWORK TRAINING

In this chapter our discussion begins with a brief review of performance optimization, which is the basis for neural network training. First, we describe the steepest descent algorithm in section 3.1.1. The key component of the steepest descent algorithm is the gradient calculation. In 3.1.2 we will present other optimization algorithms such as Newton's method and the Levenberg-Marquardt algorithm, which combines Newton's method and steepest descent to produce very efficient optimization. After that we will discuss various implementations of derivative calculations in section 3.1.3.

After discussing neural network training methods, we will discuss data preparation in section 3.2.1. We will introduce the idea of multiple training sequences, prediction horizon and horizon steps. We will also describe how a certain adjustable parameter in the LM algorithm can be used to detect spurious valleys. Finally, a brief summary of modifications to the LM training algorithm that can avoid spurious valleys is given in section 3.3.

3.1 STANDARD TRAINING

Before going into the details of the training algorithms, we need to get into some fundamental concepts of performance optimization, which is the basis of network training. One of the most classic performance optimization methods is the steepest descent algorithm.

3.1.1 STEEPEST DECENT

The objective of optimization is to find the value of $\mathbf{x}$ which minimizes a performance index $F(\mathbf{x})$. We start from an arbitrary initial guess $\mathbf{x}_0$ and then update our guess over series

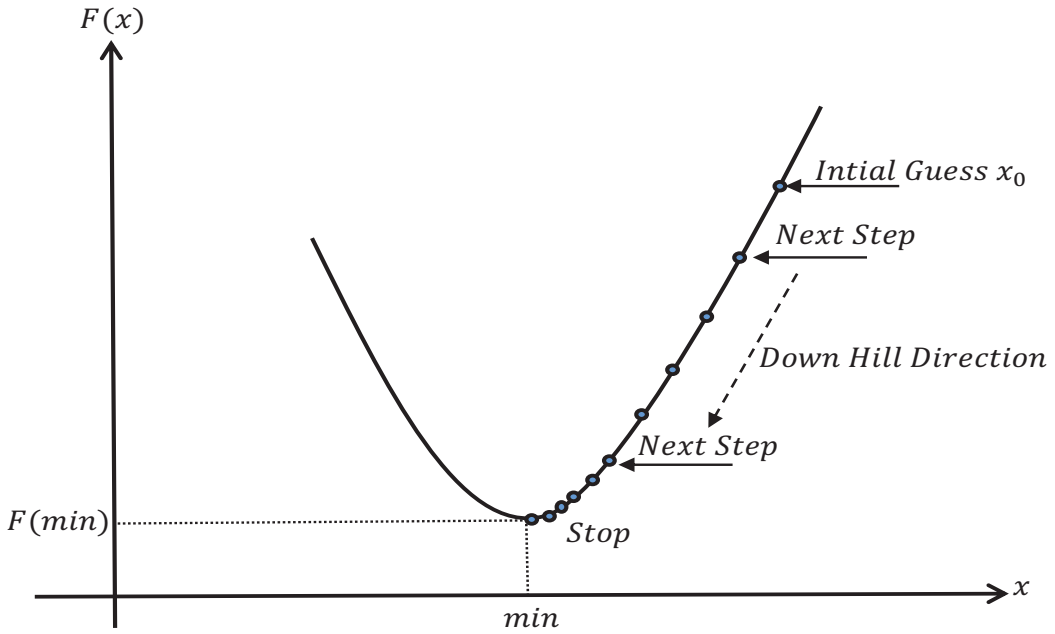


Figure 3.1: Minimizing the Performance Index

of iterations:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{3.1}$$

or

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k \tag{3.2}$$

where $\mathbf{p}_k$ is the search direction and α_k is the learning rate (scalar and positive). The length of each step is determined by the learning rate. In order to minimize the performance index, we would like the function to decrease at each iteration. We will choose the direction $\mathbf{p}_k$ so that we will move in a downhill direction, as shown in Figure 3.1.

Consider the first order Taylor series expansion:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta \mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta \mathbf{x}_k \tag{3.3}$$

where $\mathbf{g}_k$ is the gradient evaluated at $\mathbf{x}_k$.

$$\mathbf{g} = \left[\frac{\partial F}{\partial \mathbf{x}_i} \right] \tag{3.4}$$

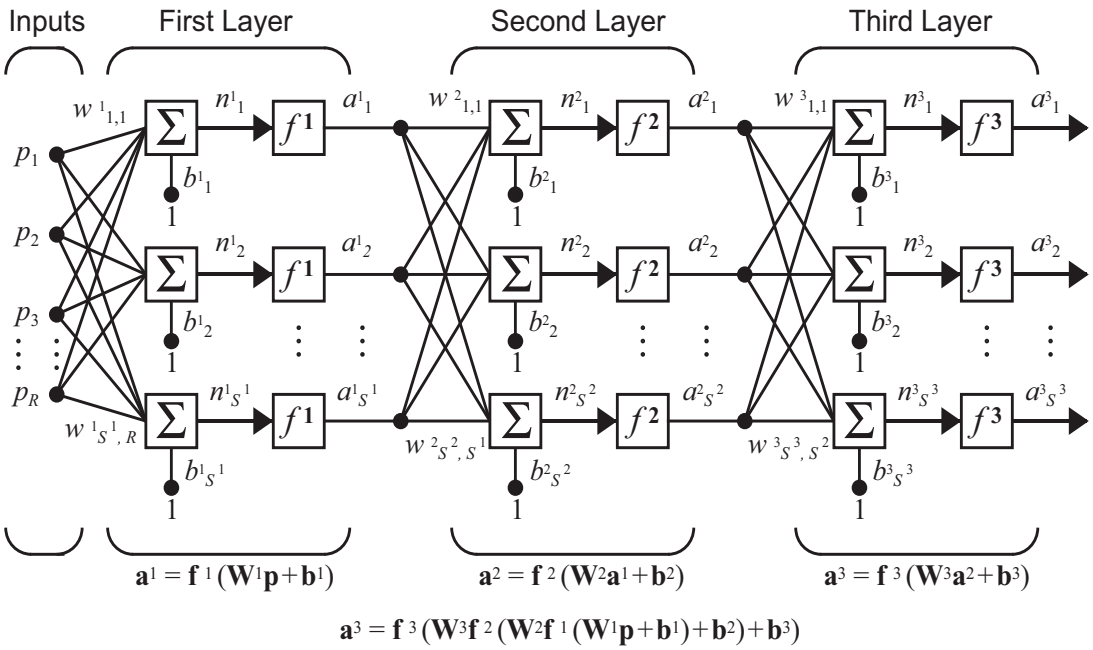


Figure 3.2: Three-Layer-Network [1]

In order for the function to decrease at each iteration the second term on the right hand side of Equation 3.3 must be negative. Any search direction that satisfies this condition is called a "descent direction". If we use $\mathbf{p}_k = -\mathbf{g}_k$ in Equation 3.1, we have the steepest descent algorithm:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k \tag{3.5}$$

There are several methods that can be used to choose the learning rates. However, if we choose a small enough value for the learning rate α_k, the steepest descent algorithm always converges to a local minimum of the performance function.

To use the steepest descent algorithm for training neural networks, we first need to compute the gradient. We will begin by defining the network structure for a static multilayer network, and then we will demonstrate how the gradient is computed. The operation of a

single neuron is defined by:

$$n = wp + b \tag{3.6}$$

$$a = f(n) \tag{3.7}$$

where w is the weight, b is the bias, n is the net input, f is the transfer function and a is the neuron output.

We can stack a finite number of neurons in a first layer and connect that layer to a second layer and so on, to create a multilayer perceptron (MLP), as shown in Figure 3.2. We can use MLPs for function approximation and pattern classification. MLPs are static networks, because they do not have feedback connections or delays.

For training, the network is provided with a set of network inputs and target network outputs $\{(p_1,t_1),(p_2,t_2),...,(p_Q,t_Q)\}$. The network is trained to minimize the sum square errors between the actual network outputs and the targets:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} \|e_q\|^2 = \sum_{q=1}^{Q} \|t_q - a_q\|^2 = \sum_{q=1}^{Q} (t_q - a_q)^T (t_q - a_q) = \sum_{q=1}^{Q} F_q(\mathbf{x}) \tag{3.8}$$

where $\mathbf{x}$ is the vector of network weights and biases. The elements of the gradient that we need to compute are

$$\frac{\partial F_q}{\partial w_{i,j}^m} \tag{3.9}$$

$$\frac{\partial F_q}{\partial b_i^m} \tag{3.10}$$

Because the error has an indirect relationship with the weights and biases in the hidden layers, we use the chain rule of calculus to calculate these derivatives:

$$\frac{\partial F_q}{\partial w_{i,j}^m} = \frac{\partial F_q}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \tag{3.11}$$

$$\frac{\partial F_q}{\partial b_i^m} = \frac{\partial F_q}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} \tag{3.12}$$

Since we used the chain rule, it is easier to calculate each derivative, since the second term in each equation includes the net input to layer m, which is an explicit function of the weights and biases. The first term is defined as a sensitivity:

$$s_i^m \equiv \frac{\partial F_q}{\partial n_i^m} \tag{3.13}$$

The derivatives in (3.9) and (3.10) can be computed as:

$$\frac{\partial F_q}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \tag{3.14}$$

$$\frac{\partial F_q}{\partial b_i^m} = s_i^m \tag{3.15}$$

The s^m terms need to be calculated using another application of the chain rule. This is the process that gives us the term backpropagation, because the sensitivity at layer m is computed from the sensitivity at layer $m+1$. To derive the relationship between the sensitivities we will use the following Jacobian matrix

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \equiv \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix} \tag{3.16}$$

$$\begin{aligned} \frac{\partial n_i^{m+1}}{\partial n_j^m} &= \frac{\partial \left(\sum_{l=1}^{s^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1}\right)}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial a_j^m}{\partial n_j^m} \\ &= w_{i,j}^{m+1} \frac{\partial f^n(n_j^m)}{\partial n_j^m} = w_{i,j}^{m+1} \dot{f}^m(n_j^m) \end{aligned} \tag{3.17}$$

The Jacobian matrix can be written in a matrix form:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1} \dot{\mathbf{F}}^m(\mathbf{n}^m) \tag{3.18}$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}^m(n_2^m) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \dot{f}^m(n_{S^m}^m) \end{bmatrix} \tag{3.19}$$

Now, with the help of chain rule, the sensitivity calculations can be written in matrix form:

$$\begin{aligned} \mathrm{s}^m = \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = (\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m})^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}} = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}} \\ = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathrm{s}^{m+1} \end{aligned} \tag{3.20}$$

It is clear how the backpropagation algorithm derives its name. The sensitivities are propagated backward through the network from the last layer to the first layer:

$$\mathrm{s}^M \rightarrow \mathrm{s}^{M-1} \rightarrow \ldots \rightarrow \mathrm{s}^2 \rightarrow \mathrm{s}^1 \tag{3.21}$$

After the sensitivities are computed using the backpropagation process, the gradient can be computed using (3.14) and (3.15). Then the weights can be updated using the steepest descent algorithm (3.5). The backpropagation method of computing derivatives, however, can be used for other optimization methods like Newton's method or Levenberg-Marquardt, which we will discuss in section 3.1.2.

3.1.2 LEVENBERG-MARQUARDT ALGORITHM

One of the faster alternatives to steepest descent for training neural networks is the Levenberg-Marquardt (LM) algorithm. The LM algorithm is a variation of Newton's method. Lets start with Newton's method and show how the LM algorithm can be derived by modifying Newton's method. The update equation for the Newton's method is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k \tag{3.22}$$

where the Hessian matrix $\mathbf{A}_k \equiv \nabla^2\mathbf{F}(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_k}$ and $\mathbf{g}_k$ is gradient. If we assume that $\mathbf{F}(\mathbf{x})$ is a sum of squared errors:

$$\mathbf{F}(\mathbf{x}) = \sum_{i=1}^{N} e_i^2(\mathbf{x}) = \mathbf{e}^T(\mathbf{x})\mathbf{e}(\mathbf{x}) \tag{3.23}$$

the gradient can be written in matrix form:

$$\nabla\mathbf{F}(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{e}(\mathbf{x}) \tag{3.24}$$

where

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_1(\mathbf{x})}{\partial x_1} & \frac{\partial e_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial e_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial e_2(\mathbf{x})}{\partial x_1} & \frac{\partial e_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial e_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_N(\mathbf{x})}{\partial x_1} & \frac{\partial e_N(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial e_N(\mathbf{x})}{\partial x_n} \end{bmatrix} \tag{3.25}$$

The Hessian matrix can be written in matrix form:

$$\nabla^2\mathbf{F}(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x}) \tag{3.26}$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} e_i(\mathbf{x})\nabla^2 e_i(\mathbf{x}) \tag{3.27}$$

If we can assume that $\mathbf{S}(\mathbf{x})$ is small, then we can approximate the Hessian matrix as

$$\nabla^2\mathbf{F}(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) \tag{3.28}$$

By substituting Equation 3.24 and Equation 3.28 into Equation 3.22 we obtain the Gauss-Newton method:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{e}(\mathbf{x}_k) \tag{3.29}$$

The advantage of Gauss-Newton method is that it does not require calculation of second derivatives (in $\nabla^2 e_i(\mathbf{x})$). However, in the Gauss-Newton method sometimes the approximate Hessian matrix $\mathbf{H} = \mathbf{J}^T\mathbf{J}$ is not invertible. This can be overcome with the following modification.

$$\mathbf{G} = \mathbf{H} + \mu\mathbf{I} \tag{3.30}$$

This leads to the Levenberg-Marquardt algorithm [1]:

$$\Delta\mathbf{x}_k = -\left[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathrm{x}_k) + \mu_k\mathbf{I}\right]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{e}(\mathbf{x}_k) \tag{3.31}$$

The important feature of the LM algorithm is that as μ_k becomes large it reverts to steepest descent with a small learning rate, which guarantees that $\mathbf{F}(\mathbf{x})$ must decrease if μ is made large enough. The algorithm starts with a small μ_k, and if $\mathbf{F}(\mathbf{x})$ does not decrease at any iteration, the algorithm increases μ_k by factor of 10. If $\mathbf{F}(\mathbf{x})$ decreases, μ_k is reduced by a factor of 10, because the algorithm converges faster in the Gauss-Newton mode (μ_k small).

If μ_k reaches a large value (e.g., $\mu_k = 10^{10}$) without reducing the SSE, the training is generally stopped. This is one criterion for stopping the LM algorithm. This often happens prematurely in RNN training. As μ_k reaches a large value, the LM algorithm takes a very small step in the steepest descent direction. If the SSE is not reduced without making μ_k very large, there may exist a very narrow valley. This would be due to the spurious valleys in the error surface described in the pervious chapter. We will use this characteristic of the LM algorithm in the next section to detect and avoid the spurious valleys.

In dynamic networks the derivative computation will be more complex and sophisticated. Instead of standard backpropagation, dynamic backpropagation algorithms need to be used. In the next section we will briefly go over some variations of dynamic derivative calculations.

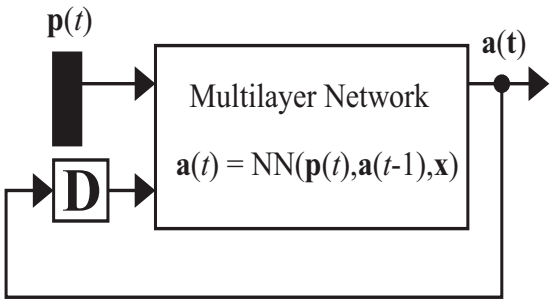


Figure 3.3: Simple Dynamic Network [1]

3.1.3 DERIVATIVE CALCULATION

Dynamic networks can be trained using the same optimization algorithms as static networks like steepest descent (which uses the gradient), or Gauss-Newton and Levenberg-Marquardt (which use the Jacobian). The difference between training the static and dynamic networks is the way of computing the gradients and Jacobian. Dynamic networks contain delays, and they operate on a sequence of inputs. These types of networks can also have feedback connections. Consider the dynamic network in Figure 3.3. It contains a static multilayer network, with a feedback connection with a single delay from the output to the inputs of the network. The vector $\mathbf{a}(t)$ is the output of network at time step t.

With dynamic networks, we need to modify the standard backpropagation algorithm. There are two different ways to approach this problem. One is backpropagation through time (BPTT), and the other is real time recurrent learning (RTRL). They both use the chain rule, but they are implemented in different ways.

In the BPTT algorithm, which is shown in (3.32) and (3.33) for the network in Figure 3.3, the network response is computed for all time points, and then the gradient is computed by starting at the last time point and working backwards in time [22]. This algorithm is computationally efficient for the gradient calculation, but it is difficult to implement online, because the algorithm works backward in time from the last time step [1].

$$\frac{\partial \mathbf{F}}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial \mathbf{F}}{\partial \mathbf{a}(t)} \tag{3.32}$$

$$\frac{\partial \mathbf{F}}{\partial \mathbf{a}(t)} = \frac{\partial^e \mathbf{F}}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}^T(t)} \times \frac{\partial \mathbf{F}}{\partial \mathbf{a}(t+1)} \tag{3.33}$$

The gradient in the RTRL algorithm can be calculated at the same time as the network response. The RTRL algorithm works forward through time as shown in (3.34) and (3.35). It requires more calculations than BPTT for calculating the gradient, however RTRL is suitable for on-line implementation. Jacobian calculations for the RTRL algorithm are generally more efficient than the BPTT algorithm.

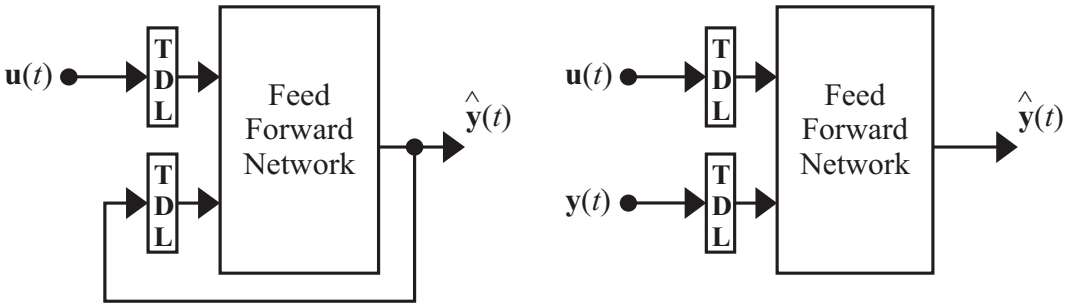
$$\frac{\partial \mathbf{F}}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e \mathbf{F}}{\partial \mathbf{a}(t)} \tag{3.34}$$

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} + \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{a}^T(t-1)} \times \frac{\partial \mathbf{a}(t-1)}{\partial \mathbf{x}^T} \tag{3.35}$$

Both algorithms calculate the same gradient, and therefore they produce the same final results. The BPTT and RTRL representations can also be used to compute Jacobian matrices which are needed in the Levenberg-Marquardt algorithm [1].

3.2 MODIFICATIONS (MULTIPLE SEQUENCES)

Before getting into the training procedure, there are some concepts and terms that need to be explained, because they are frequently used in recurrent network training. We are going to explain the difference between a sequence and a subsequence. The concept of open loop and closed loop training will be introduced. Also, the effect of increasing the length of subsequences on the prediction horizon will be discussed. Finally, the concept behind the horizon step will be clarified.



(a) Parallel Architecture (b) Series-Parallel Architecture

Figure 3.4: Parallel and Series-Parallel Architecture [1]

3.2.1 TRAINING SEQUENCE

RNNs are very good candidates to represent dynamic systems, because they have memory. In order to train a RNN to approximate a dynamic system, we need appropriate data. Unlike static networks, where each input/target pair stands on its own, RNN data must consist of ordered sequences of inputs and target outputs.

The original training data set may consist of multiple sequences. For RNNs, the length of the sequence determines the prediction horizon - the number of steps ahead that neural network is predicting. For example, consider the RNN in Figure 3.4(a), which consists of a static feedforward network with a single feedback connection. Consider the case when both tapped delay lines (TDLs) each have two delays. If we had a sequence of 10 elements, the first two elements of the input sequence would be used to fill the input TDL, and the first two elements of the target outputs would be used to fill the feedback TDL. The third target output would then be the first target used for the training, and would represent a one step prediction, because the first two targets are used in the calculation. The fourth target output would be the second training target and would represent a two step prediction, since the feedback TDL would contain the one step prediction from pervious time step. This process would continue until time step eight, when the tenth target output would be the eighth target used for the training, which would reperesent on eight step ahead prediction.

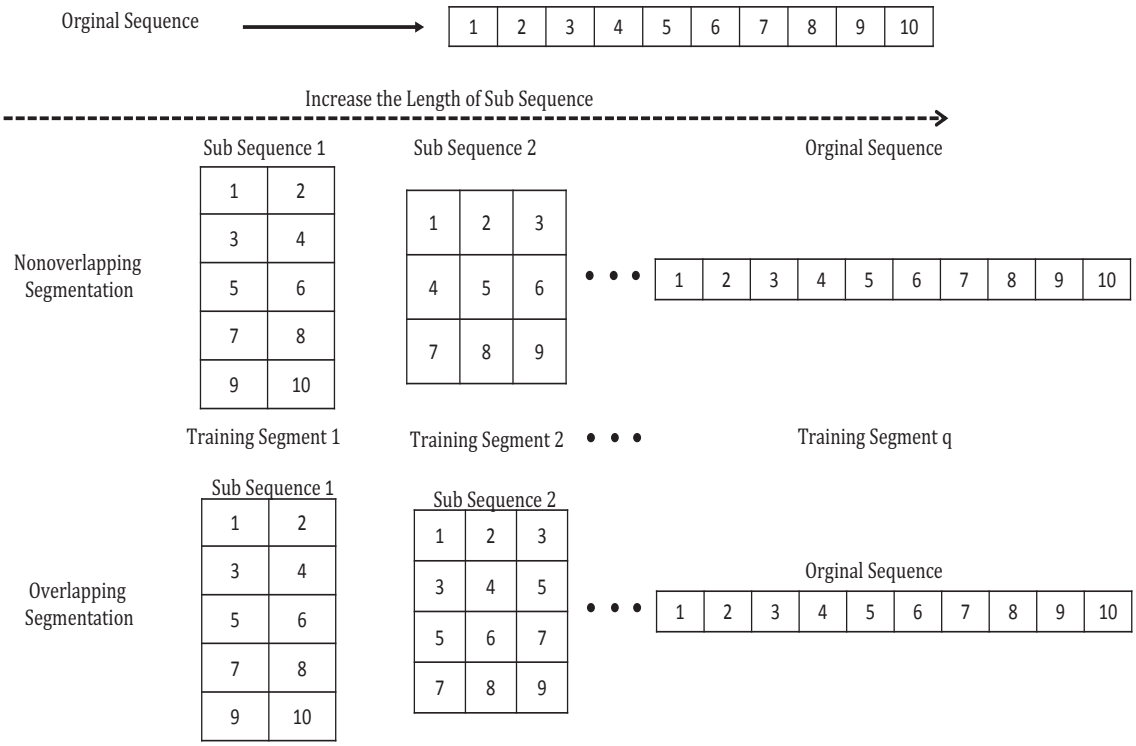


Figure 3.5: Forming Subsequences

To summarize, if we have a sequence of length Q and an RNN with maximum delay D, then network will be making predictions with prediction horizon from 1 to $Q-D$ steps ahead.

We know from [2] and [16] that the spurious valleys in the error surface become narrower and more numerous as the prediction horizon increases. For these reasons, we don't want to begin training with large prediction horizons. To avoid this, we divided the data into smaller subsequences.

3.2.2 PREDICTION HORIZON

We begin training with one step ahead prediction. This is best implemented using the series-parallel architecture shown in Figure 3.4(b). Since we are doing only one step predictions, the feedback TDL is always filled with the target outputs. We call this open loop training, because the feedback loop is effectively cut. With open loop training, the

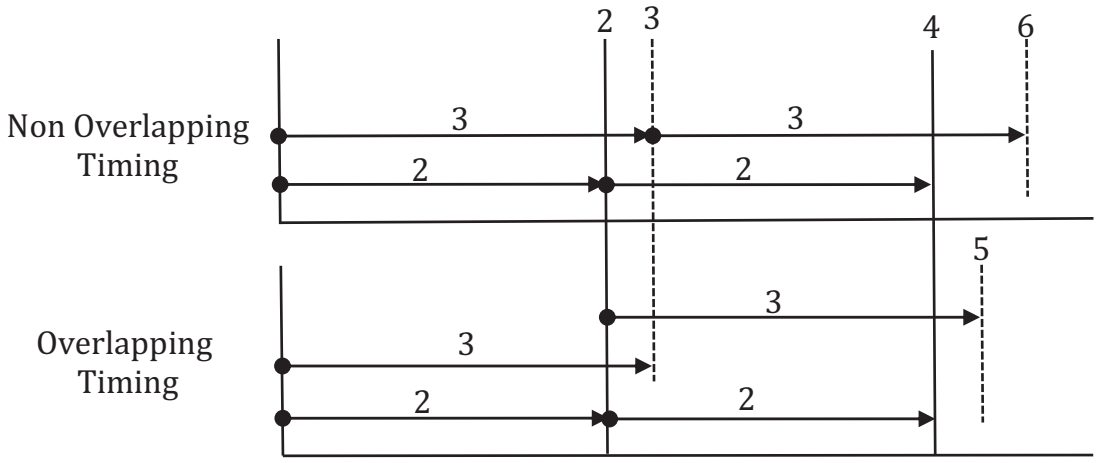


Figure 3.6: Timing Diagram

spurious valleys discussed earlier do not occur.

The open loop training proceeds for a specified number of iterations, which completes the first training segment. At this point the prediction horizon is increased by the first horizon step, and another training segment is performed. The overall training process is defined by the schedule of horizon steps. We would like to take large horizon steps in order to complete training with the full sequence (maximum prediction horizon) as quickly as possible. However, when the horizon step is too large, we are more likely become trapped in a spurious valley. Later, we will discuss improved procedures for selecting the horizon steps.

3.2.3 HORIZON STEP

The prediction horizon is increased by the horizon step at each training segment. This means that the training sequences must be re-segmented, since the length of the sequence determines the prediction horizon, as we discussed earlier. There are two approaches to the segmentation of the original sequences. In one approach the subsequences are nonoverlapping, and in the second approach there is some overlap. This is illustrated in Figure 3.5. At the top of the figure we see the progression of the subsequences that are obtained if the

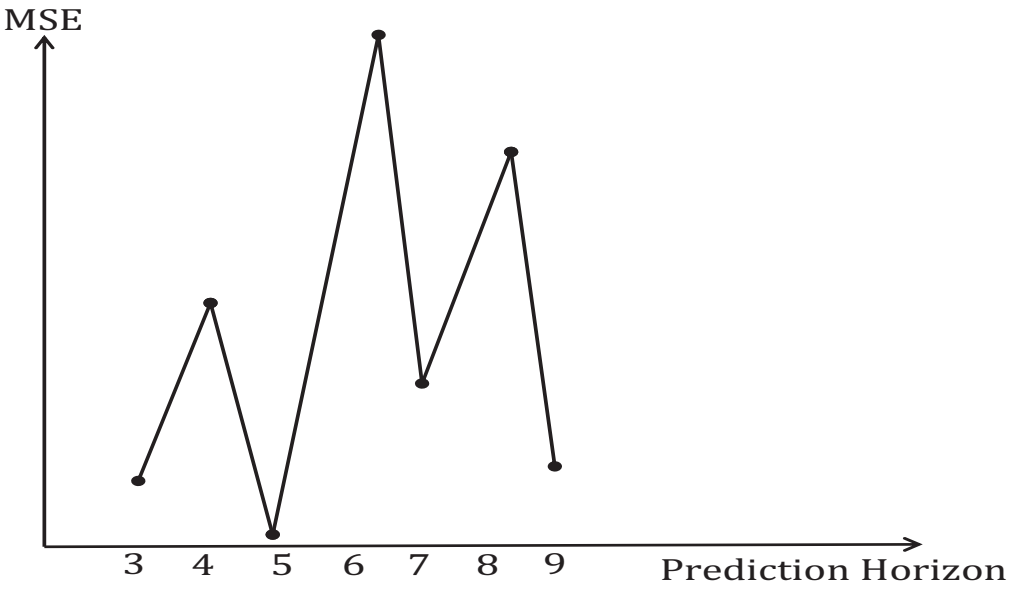


Figure 3.7: MSE vs. Prediction Horizon for Nonoverlapping Subsequences

horizon step is fixed at one, and the subsequences are selected so that they do not overlap. At bottom of the figure we see a case of maximum overlap, in which the initial time points in every sequence remain the same at each training segment. Notice that, if nonoverlapping segmentation is used, after a few training segments the subsequences are unrelated to the original subsequences. However, in the maximum overlap case the initial part of each subsequence remains the same. This is also illustrated in Figure 3.6.

The type of segmentation can affect the choice of horizon step at each training segment. For example, if fully overlapping sequences are used, then as the horizon step is increased, with fixed weights, the mean square error will increase. However, when non-overlapping or minimally overlapping sequences are used, the mean square error will not generally increase monotonically with horizon step. Figure 3.7 illustrates this case. In a later chapter we will use this concept to select a different horizon step at each training segment.

3.2.4 MODIFICATIONS TO THE LM ALGORITHM

The LM algorithm, which we described earlier, is used for network training - with one modification. As described earlier, when μ_k reaches a large value the standard LM algorithm would terminate. However, when training RNNs, large μ_k can indicate that the algorithm is trapped in spurious valleys [7]. In addition, the valley is generally only associated with one or two of the subsequences. This lead to the following modification of the LM algorithm . Whenever μ_k reaches a maximum value, the gradients of MSE for each subsequence are computed. The subsequence with largest gradient is then removed, and an iteration of LM is performed. The removed sequence is subsequently returned and training continues.

3.3 SUMMARY OF MODIFIED TRAINING

The training procedure first introduced in [7] can be summarized as follows:

1. The first training segment uses open-loop training (one-step-ahead predictions). All training segments involve a fixed number of iterations of the training algorithm.

2. Closed-loop training with increasing prediction horizon: Do k-step-ahead prediction ($k > 2$). This includes segmentation of the original long sequences into smaller subsequences.

3. At each iteration of the LM algorithm, if μ reaches μ_{max}, remove the sequence with largest gradient. If the MSE does not decrease, keep removing the sequence with next largest gradient until the MSE decreases (the algorithm escapes from the valleys). Add the removed sequences back to the training data before proceeding to next iteration.

4. Increase the prediction horizon k (sequence length). If all sequences are removed, shorten the prediction horizon and go back to step 2.

The prediction horizon was incremented according to a preplanned schedule. The schedule was conservative, with small horizon steps, since it is difficult to know the optimal horizon. Because the schedule was conservative, training times could become quite long. In the next section, we will describe a modification to [7], in which the horizon step is adaptively selected for each training segment [23].

CHAPTER 4

NEW PROCEDURE FOR HORIZON SELECTION

In this chapter a new procedure for selecting the horizon step will be introduced. Before going into the details of this new procedure, the effect of prediction horizon on the training process will be explained in Section 4.1. In the last chapter, we explained that the prediction horizon is increased by the horizon step at each training segment. This increment of the prediction horizon has major effects on the training process. Some of these effects, such as large oscillations in the training subsequences, will be discussed.

In addition, a new procedure for selecting the best horizon step will be discussed in Section 4.2. The new procedure explains how the best horizon step will be chosen. This method will become part of the modified training procedures that were discussed in Section 3.3.

4.1 EFFECT OF PREDICTION HORIZON ON TRAINING

Before we describe the procedure for determining the optimum horizon step, we want to discuss the effect of the prediction horizon on the training process. As shown in [2] and [16], the spurious valleys appear in regions of the weight space where the network is unstable (see section 2.2). Even though the network output is very small at the bottom of the valley, small changes in the weights will result in unstable responses. Fig. 4.1 illustrates a typical network response when the weights are inside a valley, but not at the bottom. The network response is small, until it reaches approximately time step 220, when oscillatory behavior begins. With a prediction horizon of less than 220, the oscillation would not have occurred. If the oscillatory behavior only takes place over a small region at the end of the

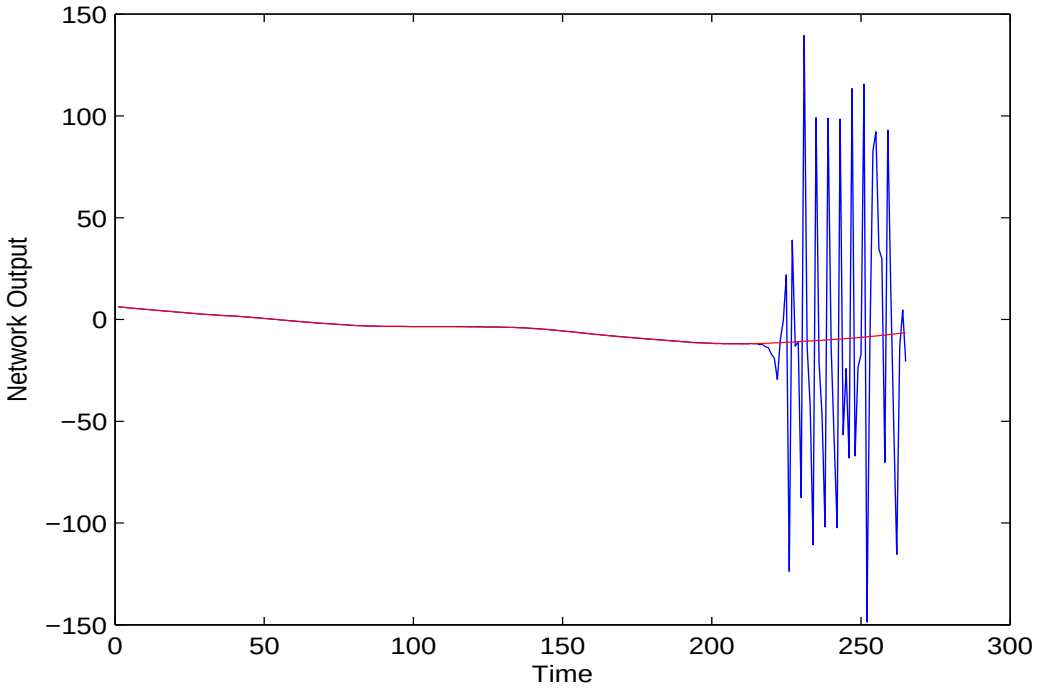


Figure 4.1: Network response inside a spurious valley.

sequence, then training can continue successfully.

Recall that in the pervious chapter we discussed two approaches for preparing data for each training segment. If the fully overlapping (see in Figure 3.6) approach is used, the MSE will increase monotonically as the horizon step increases. For example, in Figure 4.1, as the horizon step increases beyond 220, the amount of oscillation within the prediction horizon increases. However, if the non-overlapping approach is used, the training subsequences will have less in common as the horizon step increases, so the error is not monotonic with respect to horizon step, as in Figure 3.7. The data segmentation approach is very important and has an effect on the training procedures.

When oscillations occur over a significant percentage of the sequence, it is difficult for the training to recover a stable response. Usually, these oscillations produce a large MSE value. This means it would be very difficult for the LM optimization algorithm to minimize a very large initial MSE value. Figure 4.2 shows one example of a network response with

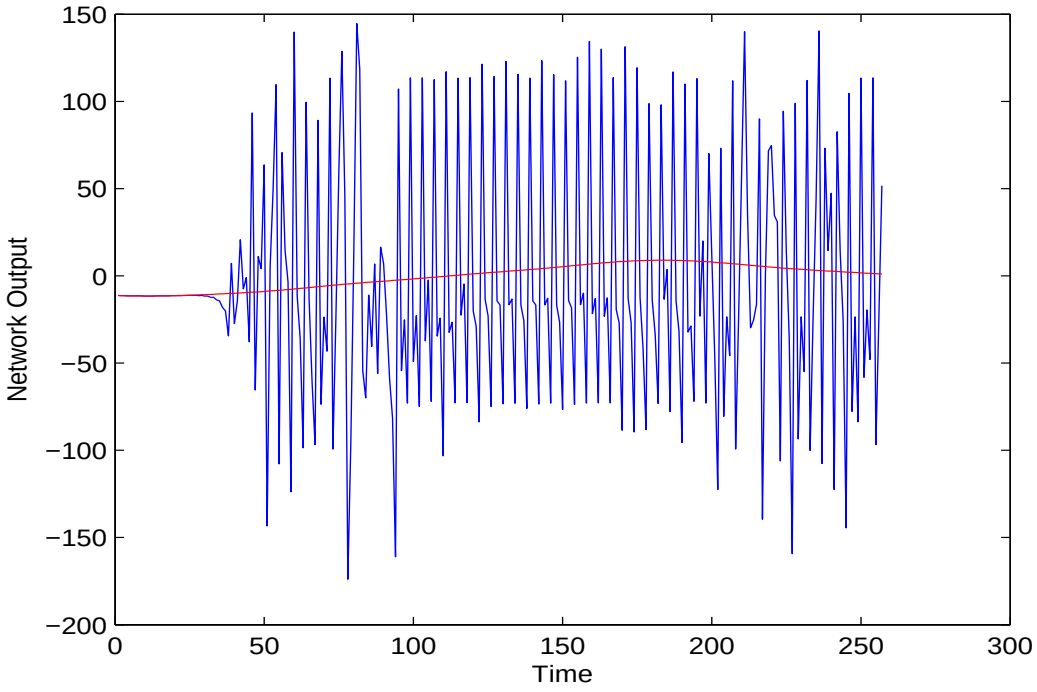


Figure 4.2: Network response with large oscillation.

large oscillations. The large oscillations cover almost 80% of the training sequence. These large oscillations in Figure 4.2 usually occur in one or two of the subsequences and will cause a large initial MSE, which makes it very hard for the LM algorithm to minimize the performance index.

We want to increase the prediction horizon as much as possible at each training segment, but we do not want to increase it so far that unstable behavior occurs over too large a percentage of any of the subsequences. In order to judge the percentage of oscillation, we will compute the MSE. We should mention that the percentage oscillation is calculated by dividing the length of oscillation by the length of prediction horizon at each horizon step. We have found that there is a linear relationship between MSE and percentage oscillation. This is illustrated in Fig. 4.3, which is a typical scatter plot of percentage oscillation versus MSE for individual subsequences. Each circle in the scatter plot represents the worst sequence percentage oscillation vs. the MSE for a particular horizon step. In this figure the

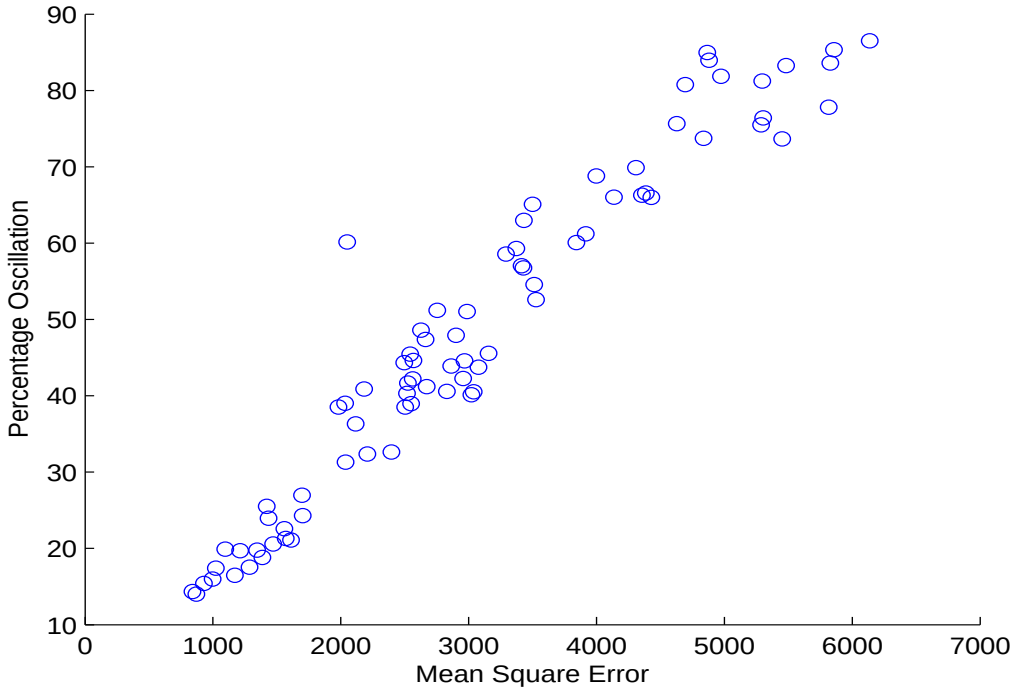


Figure 4.3: Relationship between MSE and percent oscillation.

range of prediction horizon is from 240 to 320. Also, the slope of percentage oscillation versus the MSE can vary with the problem, but we can set a limit on the MSE that will also set a limit on the percent oscillation.

At the successful completion of each training segment, there should be no unstable behavior in any of the subsequences. If the horizon step for the next training segment is chosen too large, however, oscillations could occur over too long a time interval in the new subsequences to allow successful training. The objective of our proposed horizon step selection method is to find the largest horizon step for which the oscillations occur over a sufficiently small percentage of the sequence.

4.2 TRAINING PROCEDURE

For the algorithm we will discuss in this section, we will need to decide parameters such as the number of iterations for each training segment and the horizon step at each training

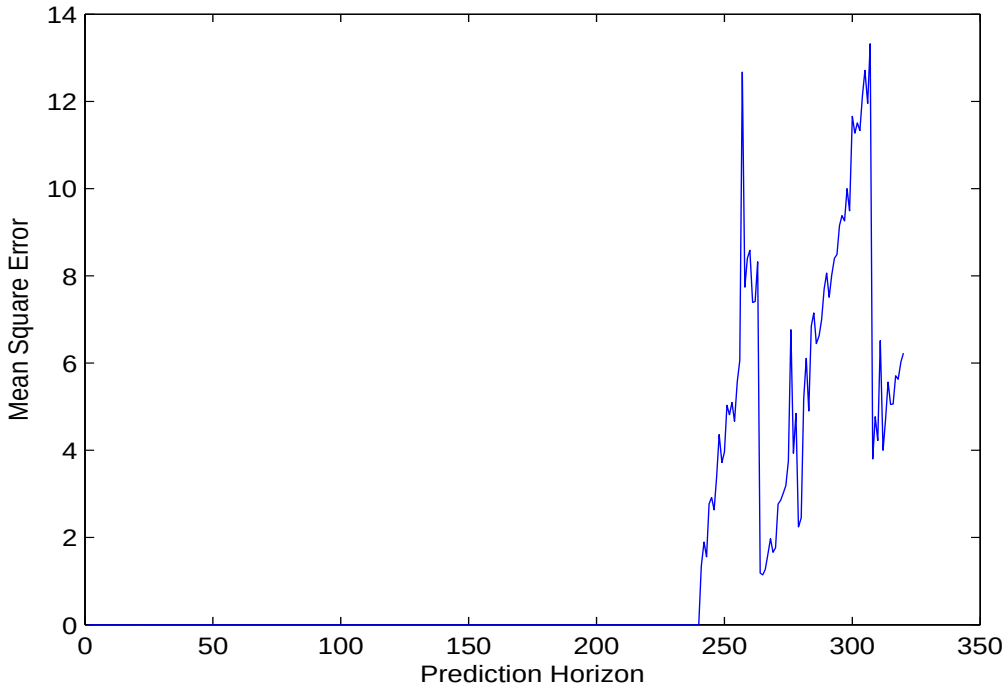


Figure 4.4: MSE versus Prediction Horizon.

segment. Also we should know which subsequences to use at each iteration and the total number of training segments. We should note that the training algorithm discussed in this section is a batch algorithm. This means that at each iteration (defined as an update of the network weights) of the algorithm, the gradient is computed for the entire training data set. We use the BPTT algorithm, which was mentioned in the previous chapter.

The method that was introduced in [7] increased the prediction horizon with a small horizon step at each training segment. We begin the initial training segment with a one-step-ahead prediction (open loop training). The prediction horizon is increased at each training segment, until the prediction horizon during the final training segment covers the full length of the original sequences. This process can require long training times, if the prediction horizon is increased too slowly, but will fail to converge if the prediction horizon is increased too quickly. We are introducing a new method that searches for a good horizon step at each training segment.

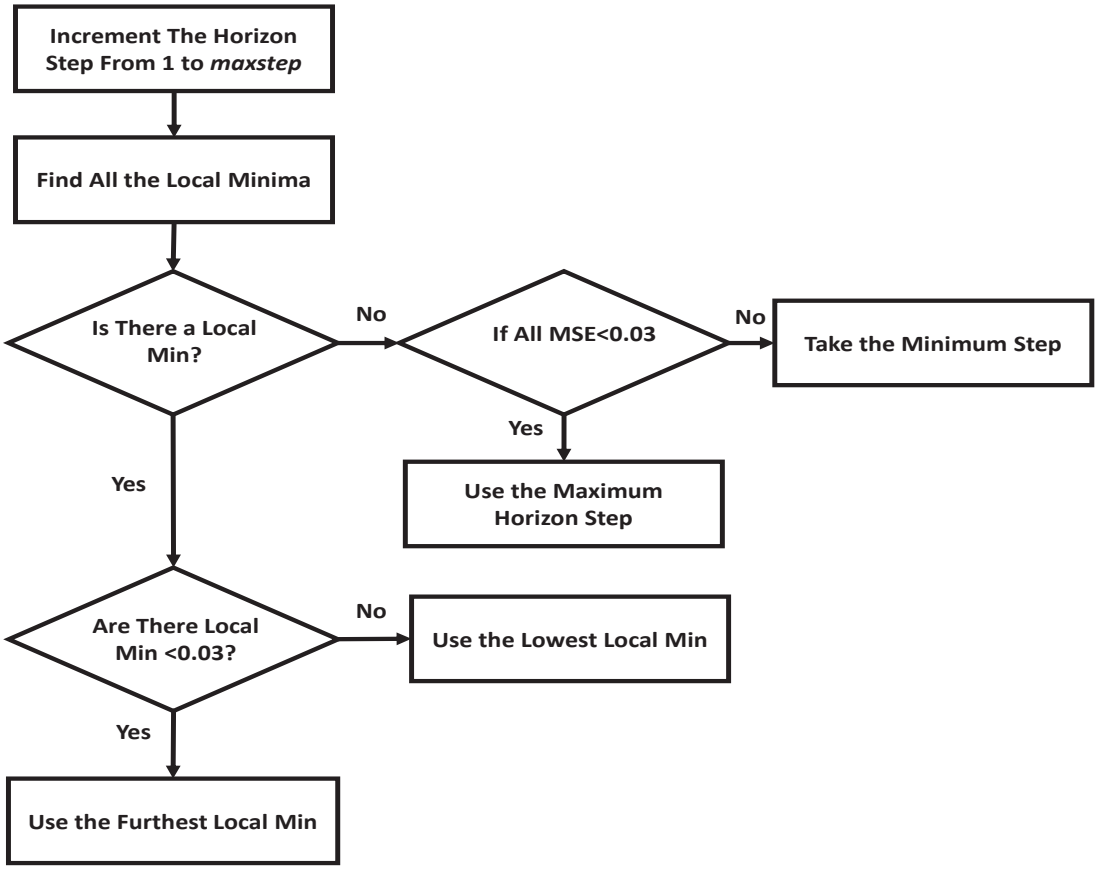


Figure 4.5: Flow Chart of Choosing the Optimum Horizon Steps.

The new procedure will follow the same steps outlined in the previous section, including the use of the LM algorithm, but in the final step, where the prediction horizon is increased, the following procedure will be implemented to determine the horizon step.

Using the weights determined at the completion of the previous training segment, the MSE will be computed for prediction horizons from 1 to *maxstep* steps ahead of the prediction horizon used in the previous training segment. (For each increment in the prediction horizon, the original training sequences will need to be resampled into new subsequences of appropriate length.)

At this point, the algorithm will find all local minima of the MSE with respect to the prediction horizon. It will then select the local minimum with the smallest MSE. For example, consider Fig. 4.4, which shows MSE versus prediction horizon for a typical problem,

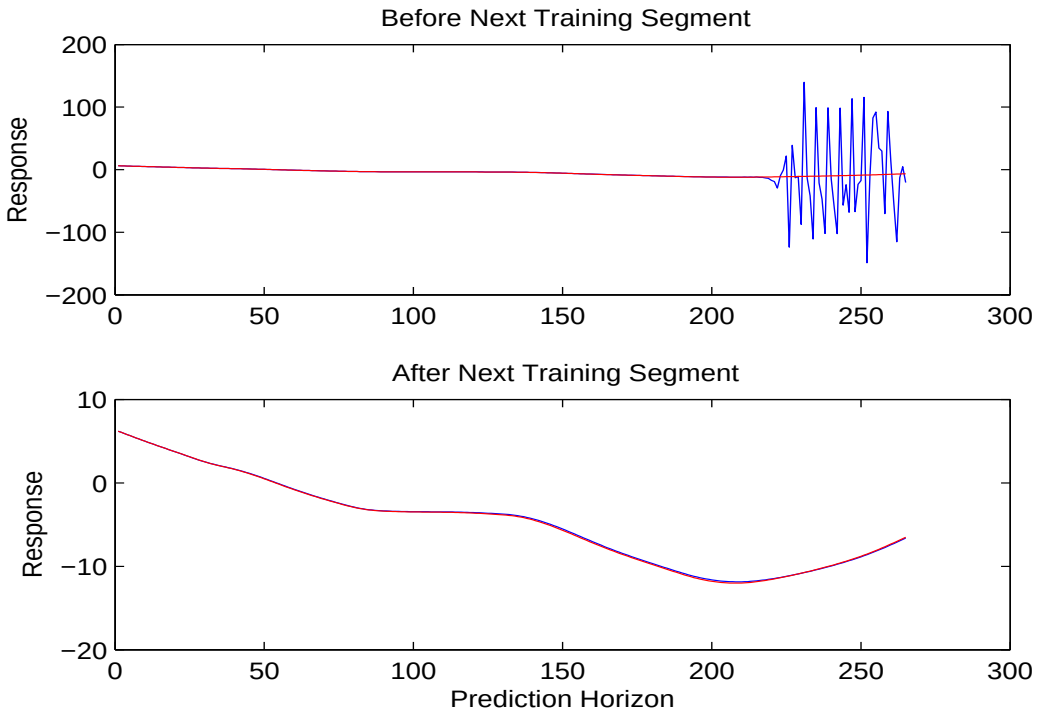


Figure 4.6: Network response on worst sequence, before and after training.

in which the prediction horizon for the previous training segment was 240. The local minimum that has the smallest MSE (1.144) occurs at prediction horizon 265, and therefore 265 was chosen for the prediction horizon for the next training segment. This represents an horizon step of 25.

In order to take larger horizon steps, the algorithm can be slightly modified. If the MSE is less than a small threshold (we used 0.03) for some local minima beyond the global minimum, then the furthest of these minima is selected. If no local minima exists, then the furthest prediction horizon with MSE less than the threshold is selected. The flowchart for choosing the optimum horizon steps selection is shown in Figure 4.5.

The effect of the training is illustrated in Fig. 4.6. The top axis shows the response of the network for the prediction horizon of 265 on the worst subsequence, before the next training segment is complete. This subsequence is largely responsible for the MSE of 1.144 mentioned in the previous paragraph; none of the other subsequences had unstable

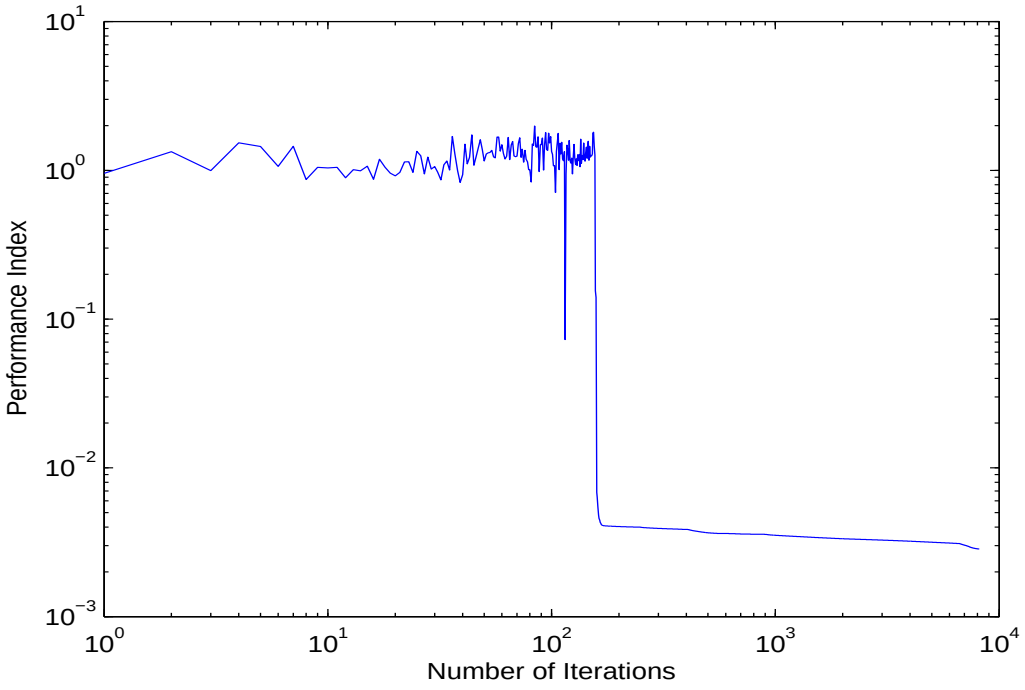


Figure 4.7: Performance Index.

oscillations for the prediction horizon of 265. We can see that the oscillations only occur over about 10% of the subsequence, which is acceptable. The lower axis shows the response for the same subsequence, after the next training segment is complete. As we can see, the training was able to successfully overcome the oscillatory behavior, because of the limited initial oscillations. When the initial oscillations occur over much more than 10% of the subsequence at the beginning of the training segment, the training will have difficulty producing a good response. (This corresponded to an MSE of approximately 2, which was the threshold used.) For example, we attempted training the network for the prediction horizon corresponding to the next local minimum, with an MSE larger than 2, but the training was not successful; the oscillation occurred over too large a percentage of the prediction horizon.

This new procedure is very helpful in several ways. The time of training can be reduced, and the optimum jump in the prediction horizon enables more successful completion of training segments. In order to have a successful training segment, we need to reduce MSE

to a relatively small value.

A typical performance index plot is shown in Figure 4.7 for a practical problem. There are some jumps in the performance index at the beginning of training, This is due to the modified training procedure. In the modified training procedure, whenever the LM algorithm is trapped in a spurious valley (μ_k reaches a maximum value), the gradients of MSE for each subsequence are computed. The subsequence with largest gradient is then removed, and an iteration of LM is performed. After the iteration is completed, the removed sequence is subsequently returned and training continues. Although the LM algorithm is guaranteed to reduce the MSE of the data in the training set, the training set is modified during training, so the MSE oscillates some during the modified training.

CHAPTER 5

DEMONSTRATION OF HORIZON SELECTION

In this chapter, we will begin in Section 5.1 with the description of a practical application, the single-link robot arm. In Section 5.2, we will demonstrate the training procedure introduced in Section 4.2 to identify the robot arm system.

5.1 SYSTEM DESCRIPTION (SINGLE ROBOT ARM)

One of the very useful applications of RNNs is system identification (modeling of dynamic systems). The demonstration problem for the new training method presented in Chapter 4 is the modeling, or system identification, of a single-link robot arm, driven by a DC motor (see Figure 5.1). The single-link robot arm is an electromechanical system which includes the motor dynamics (electrical part) and the arm dynamics (mechanical part). The equations of motion of the arm are given by:

$$V_a = RI_a + L\dot{I}_a + K_b\dot{\theta} \tag{5.1}$$

$$J\ddot{\theta} = K_t I_a - B\dot{\theta} - mglsin(\theta) \tag{5.2}$$

where $V_a(t)$ is the voltage applied to the motor, $I_a(t)$ is the armature current, J is the moment of inertia of the arm, g is the gravitational constant, K_b is the back emf constant, K_t is the torque constant, L is the inductance, l is the length of the arm and B is the viscous friction coefficient. Simulation parameters, including the sampling time t_s are given in Table 5.1.

The single-link robot arm is a classical dynamical system for a system identification problem. In order to make this system a challenging problem, the parameters were chosen

K_b	K_t	g	B	L	R	J	m	l	t_s
0.055	0.055	9.8	0.0025	08e−3	0.4	18e−4	0.1	0.5	0.001

Table 5.1: Simulation Parameters for the Robot Arm

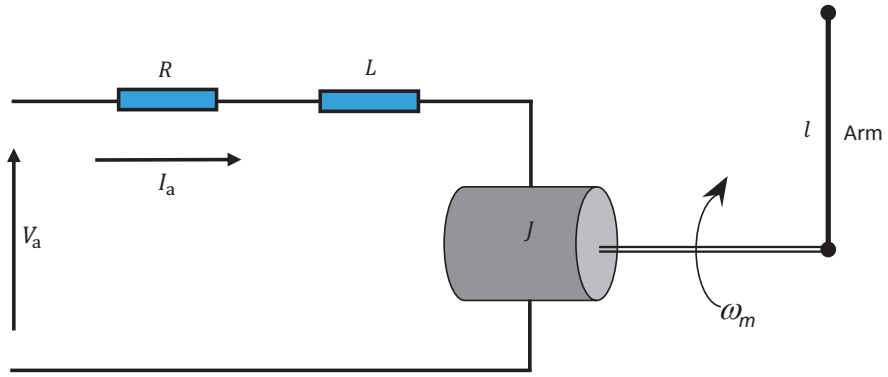


Figure 5.1: Single robot arm driven by DC motor.

in such fashion that the system responds quickly and has a strong nonlinearity.

The single-link robot arm is a third order system. The three states of this system are chosen to be angular position, angular velocity and current. The downward position of the arm represents the zero position, and clockwise is the positive direction.

Figure 5.1 shows the arm location in an upward position. This is a critical location for the pendulum (neighborhood of π), since the response here is very sensitive to the initial condition. A very small change in the π region will result in a different angular position. In order to model this system, this sensitivity makes it a very challenging task for an RNN model. The model needs to be very accurate over all the regions, especially in the upward position. Therefore, we need a very good training procedure to model this system over the full dynamic range.

In order to model this system using RNNs, we need to collect data for training purposes. We use Simulink as a tool to gather data from this dynamic system. In the next section we will go into the details of training data collection.

Training data							
Name	Min height	Max height	Min width	Max width	Max total height	Min total height	First order TF
Data_Train1	-6	6	0.004	0.01	12	-12	YES
Data_Train2	-6	6	0.004	0.01	12	-12	NO
Data_Train3	-6	6	0.004	0.02	12	-12	YES
Data_Train4	-6	6	0.004	0.02	12	-12	NO
Data_Train5	-6	6	0.004	0.03	12	-12	YES
Data_Train6	-6	6	0.004	0.03	12	-12	NO
Data_Train7	-12	12	0.004	0.01	12	-12	YES
Data_Train8	-12	12	0.004	0.01	12	-12	NO
Data_Train9	-12	12	0.004	0.02	12	-12	YES
Data_Train10	-12	12	0.004	0.02	12	-12	NO
Data_Train11	-12	12	0.004	0.03	12	-12	YES
Data_Train12	-12	12	0.004	0.03	12	-12	NO
Data_Train13	-24	24	0.004	0.01	12	-12	YES
Data_Train14	-24	24	0.004	0.01	12	-12	NO
Data_Train15	-24	24	0.004	0.02	12	-12	YES
Data_Train16	-24	24	0.004	0.02	12	-12	NO
Data_Train17	-24	24	0.004	0.03	12	-12	YES
Data_Train18	-24	24	0.004	0.03	12	-12	NO
Data_Train19	-24	24	0.004	0.03	12	-12	YES
Data_Train20	-6	6	0.004	0.01	12	-12	NO

Table 5.2: Training data skyline range

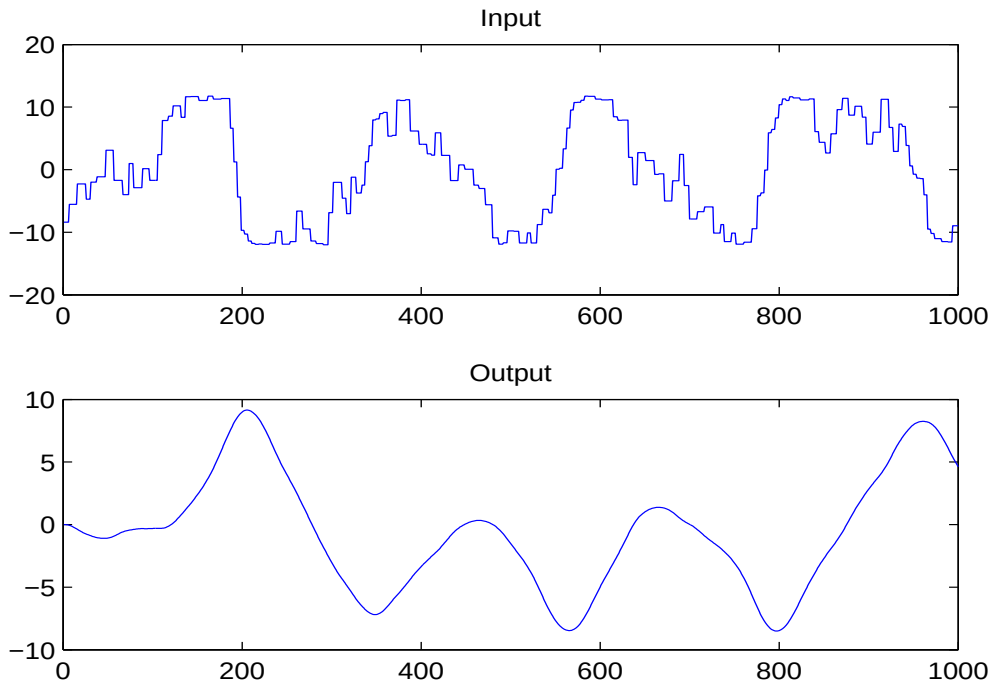


Figure 5.2: Sample training sequence.

5.2 TRAINING DATA

To obtain the training data, we need to apply random input voltages (in the Simulink model) to represent all possible input sequences to the motor drive. This input sequence consists of a series of pulses of random heights and widths, known as a skyline function [24]. An example of one sequence of the training data is shown in Figure 5.2.

We need to modify the standard skyline function so that the angular position does not move outside the relevant range. For example, in Figure 5.2 the output (angular position) stays in the range 3π to -3π. The heights and widths of pulses are randomly selected in the standard skyline function. However, in the modified skyline function, we consider the angular position as an external parameter, and when it is out of our specified location, we will choose random heights and widths in such way that the angular position will remain within our specified active region. Table 5.2 shows the height and width of random pulses for 20 training sequences. We tried 10, 20 and 30 millisecond pulse widths with different

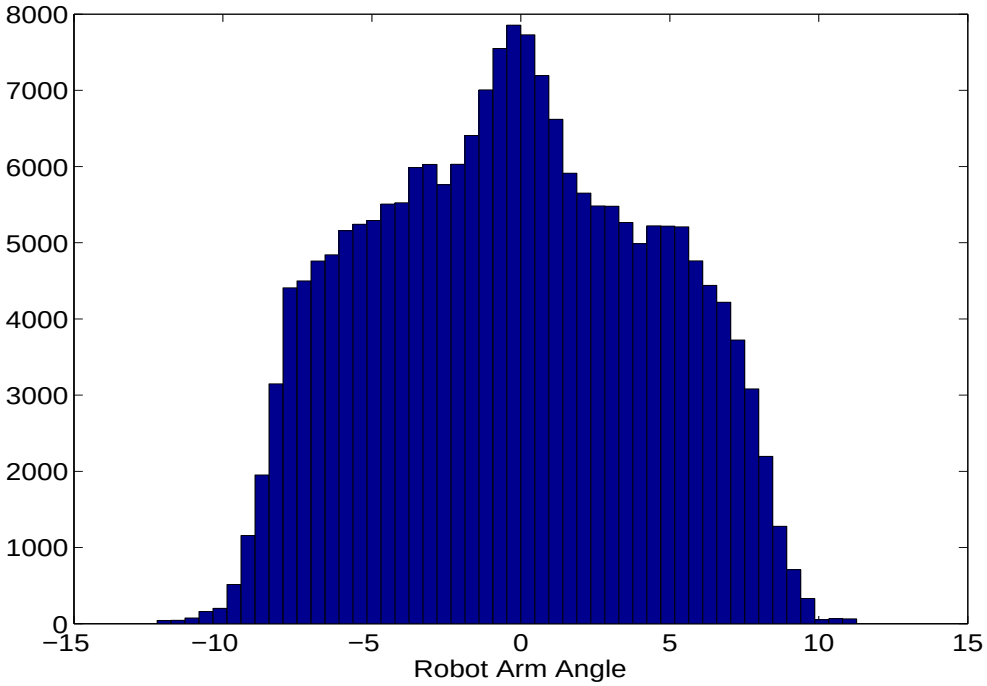


Figure 5.3: Histogram of robot arm angles contained in the training set.

heights.

A total of 20 such sequences were generated to produce the entire training data set, in order to cover the full range of required network operation. Fig. 5.3 shows a histogram of robot arm angles in the training set (in radians), which demonstrates the coverage of the modeling. The robot arm dynamic system is to be modeled as the angle varies in the range from -3π to 3π and as the motor voltage varies from -12 to $+12$.

For dynamic modeling and system identification, the NARX network (Nonlinear Auto Regressive model with exogenous input) shown in Figure 5.4 is popular. The NARX network is a, RNN that has feedback connections around a multilayer network. This has a parallel architecture, as shown in Figure 3.4(a). The NARX model is based on the linear ARX mode, which is commonly used in time-series modeling. The defining equation for the NARX model is:

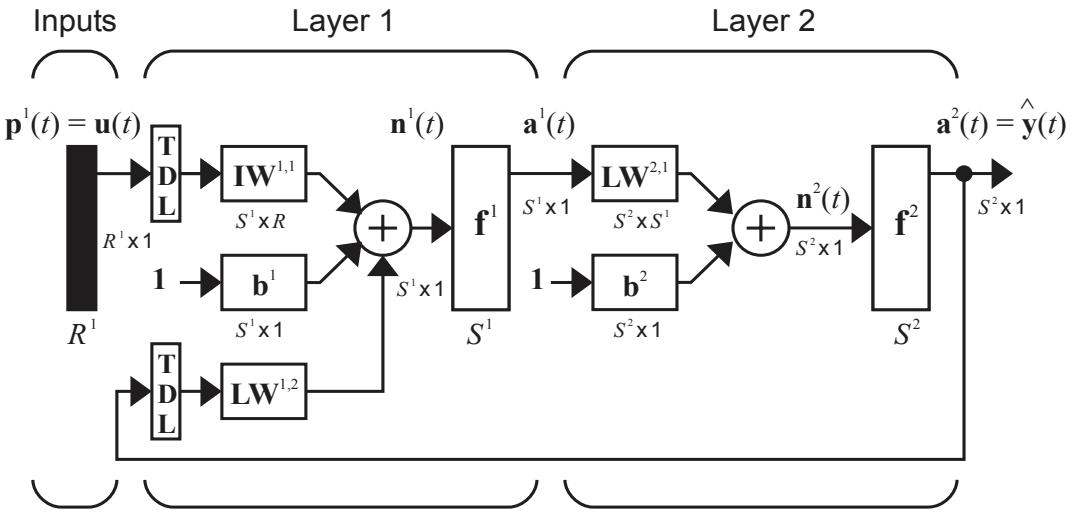


Figure 5.4: NARX recurrent network. [1]

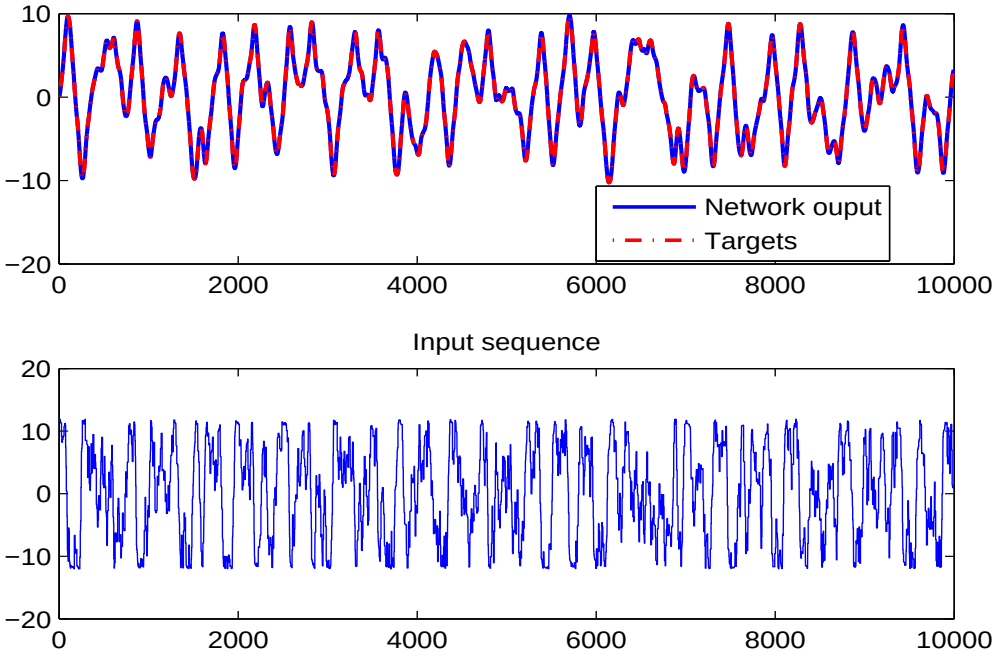


Figure 5.5: Open loop training data.

Parallel computing (Using 4 core CPU)		
Method	Hours	Days
Standard Training Over All Time	684	28
Modified Training Over All Time	170	7

Standard computing (Using 1 core CPU)		
Method	Hours	Days
Standard Training Over All Time	1368	56
Modified Training Over All Time	340	14

Table 5.3: Training time for the full data set

$$y(t) = f(y(t-1), y(t-2), ..., y(t-n_y), u(t-2), ..., u(t-n_u)) \tag{5.3}$$

The current value of the output signal $y(t)$ is predicted based on previous values of the output signal and previous values of the input (exogenous) signal. The NARX network models the function f by using a feedforward neural network. The NARX network can be used for modeling of nonlinear dynamic systems. In addition, the implementation of this network allows us to use multidimensional inputs and outputs.

We used the NARX network shown in Fig. 5.4 to model the robot arm system. We used 4 input delays and 4 feedback delays (so the training begins with the fifth data point) and 25 hidden neurons.

Fig. 5.6 compares the new training method, described in Section 4.2, in which an horizon step is selected at each training segment, with a standard method, in which a horizon step of one is used at each training segment. For prediction horizons up to 200, both methods produce similar MSE. However, the new method requires many fewer computations to achieve the same result, because many intermediate horizon steps are skipped (see Table 5.5). (Each circle in the figure represents an horizon step.) For prediction horizons greater than 200, the single step method is very erratic, with the MSE eventually becoming much larger for the single step method. Notice that for the new method, the MSE does not increase significantly after the prediction horizon is larger than 500. We increased the

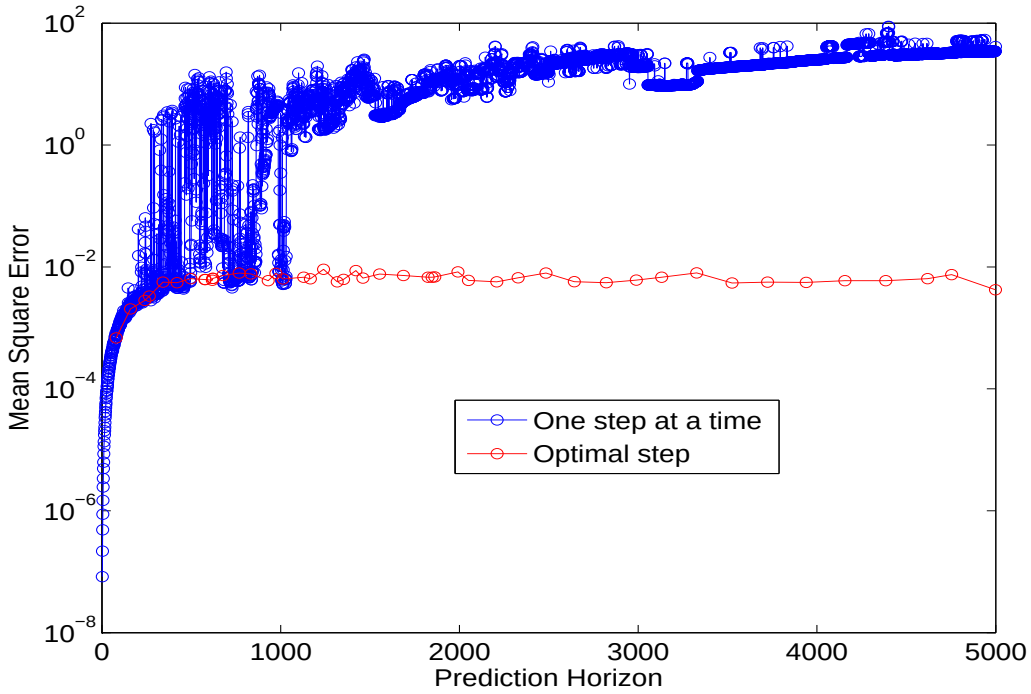


Figure 5.6: Change of MSE with increasing prediction horizon.

prediction horizon up to the full sequence length of 10,000, and the trend continued.

Table 5.3 shows the total training time for standard and modified training procedures. The comparison shows that standard training is impractical. It requires much more computation, and it does a poor job in minimizing the MSE. The modified training algorithm, which adaptively selects the horizon step, is more time efficient and produces a much smaller MSE.

Table 5.5 shows the detailed horizon step selection for the single-link robot arm. In some training segments, very small horizon steps are taken. This indicates that the algorithm chose the lowest local minimum. However, in some other training segments the furthest local minimum is selected, and the horizon step is bigger. This algorithm was described in Figure 4.5.

One advantage of the horizon selection method is that the total number of times the algorithm stops due to spurious valleys (indicated by μ becoming larger than μ_{max}) is

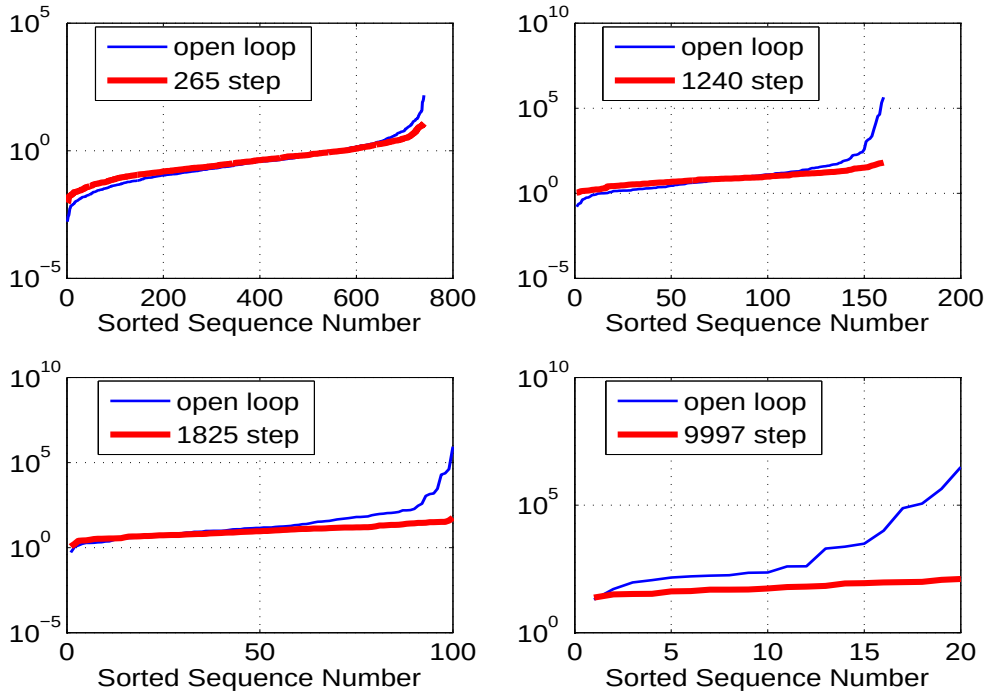


Figure 5.7: Sorted MSE for all subsequences, using different prediction horizons.

kurtosis			
265 step	1240 step	1825	9997
21.9488	8.6697	6.7662	2.1798

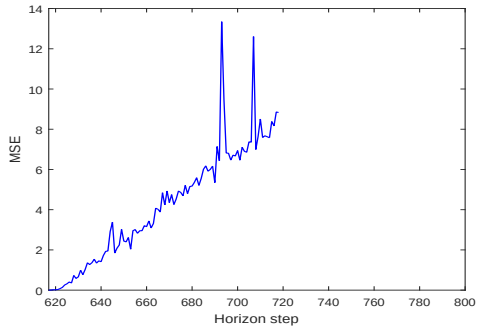
Table 5.4: kurtosis

greatly reduced when compared to the standard method. For example, in the single-link robot arm system, we reached μ_{max} in just two training segments (training segment number 4 and 16), out of the 77 training segments needed for the prediction horizon to reach the full length of the original sequence. This compares with 9505 times that μ_{max} was reached using the standard method.

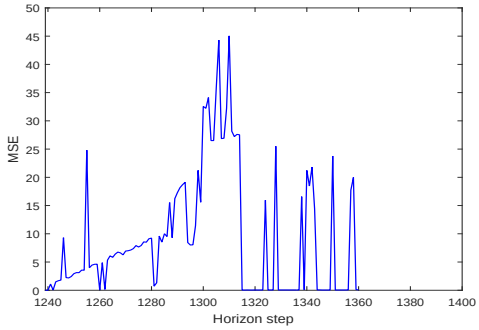
Figure 5.7 provides some insight into the effect of training the network for multi-step-ahead predictions. The plots show the MSE sorted from lowest to highest for each training subsequence. Each plot corresponds to a multi-step prediction, trained for a different prediction horizon, compared to the open loop training (trained for single-step prediction).

Training Segment	Pervious Prediction Horizon	Horizon Step	# of Times Mu max reached	Training Segment	Pervious Prediction Horizon	Horizon Step	# of Times Mu max reached
1	openloop	80	0	41	3327	198	0
2	80	80	0	42	3525	199	0
3	160	80	0	43	3724	216	0
4	240	25	156	44	3940	216	0
5	265	77	0	45	4156	229	0
6	342	78	0	46	4385	234	0
7	420	78	0	47	4619	134	0
8	498	78	0	48	4753	246	0
9	576	2	0	49	4999	137	0
10	578	40	0	50	5136	8	0
11	618	2	0	51	5144	280	0
12	620	2	0	52	5424	85	0
13	622	55	0	53	5509	7	0
14	677	91	0	54	5516	300	0
15	768	65	1578	55	5816	315	0
16	833	97	0	56	6131	38	0
17	930	43	0	57	6169	2	0
18	973	40	0	58	6171	340	0
19	1013	116	0	59	6511	34	0
20	1129	37	0	60	6545	9	0
21	1166	74	0	61	6554	2	0
22	1240	76	0	62	6556	380	0
23	1316	36	0	63	6936	48	0
24	1352	69	0	64	6984	380	0
25	1421	39	0	65	7364	57	0
26	1460	93	0	66	7421	7	0
27	1553	134	0	67	7428	10	0
28	1687	138	0	68	7438	3	0
29	1825	22	0	69	7441	400	0
30	1847	15	0	70	7841	411	0
31	1862	130	0	71	8252	401	0
32	1992	58	0	72	8653	42	0
33	2050	159	0	73	8695	2	0
34	2209	119	0	74	8697	460	0
35	2328	156	0	75	9157	325	0
36	2484	159	0	76	9482	377	0
37	2643	179	0	77	9859	138	0
38	2822	168	0				
39	2990	141	0				
40	3131	196	0				

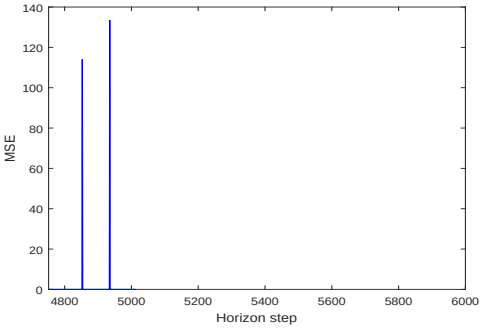
Table 5.5: Horizon step selection table



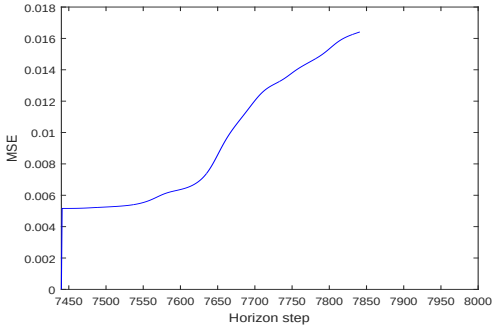
(a) Training segment 11



(b) Training segment 23



(c) Training segment 49



(d) Training segment 70

Figure 5.8: Four different horizon step cases

(Note that the number of subsequences decreases as the prediction horizon increases.) Even for modest prediction horizons, the largest errors are significantly reduced, when compared with the single-step training. For the largest prediction horizons, errors are reduced for all cases.

Table 5.4 shows the kurtosis for 4 intermediate horizon steps. A higher kurtosis indicates that the data distribution has bigger tails (more large and small values), and a lower kurtosis indicates a smaller tails. In this case the lower kurtosis means that training for multi step ahead predictions reduces the largest MSE errors.

Figure 5.8 shows four different examples of selecting the best horizon step. We need to take a shorter steps in some training segments (see Figure 5.8(a)). However, in some training segments, we can take a larger step (see Figure 5.8(b)). Figure 5.8(c) shows that if we did not have a new method of choosing the horizon step we would be trapped in

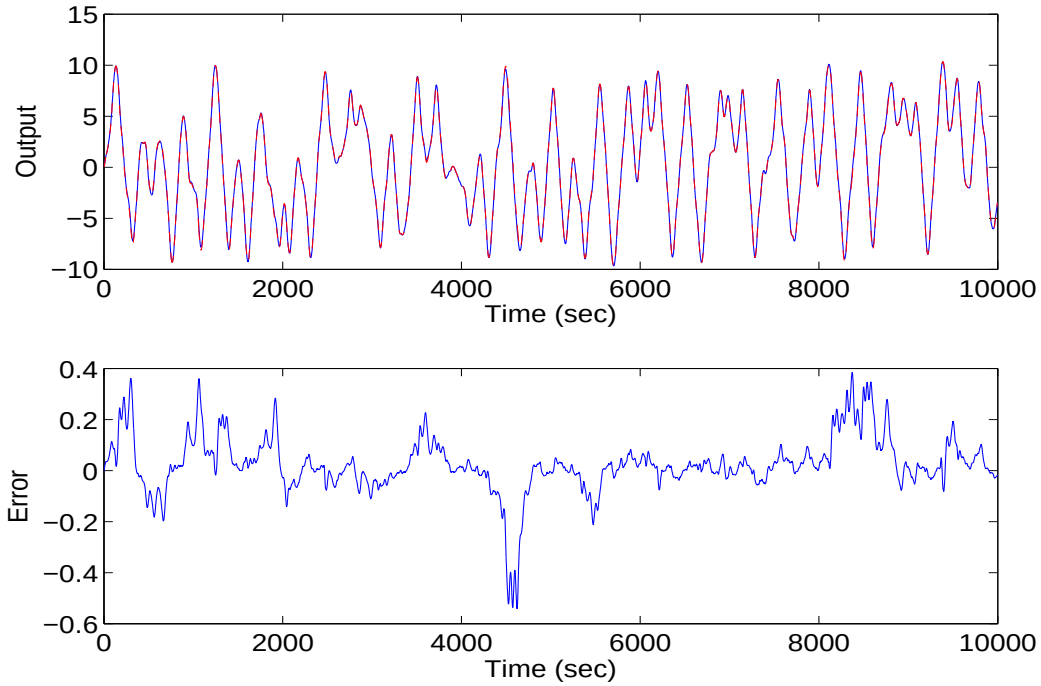


Figure 5.9: Target and accurate network response on test sequence.

some locations which would be impossible for the LM algorithm to escape. Figure 5.8(d) shows a case where all of the horizon steps have very small MSE, therefore we can take the maximum step.

After training was completed for the maximum prediction horizon, the network response closely followed the target response for all 20 of the original training sequences. In order to validate the network, it was tested on 20 additional sequences of 10,000 time points, which were not used for training. For 13 of these test sequences, the network response was very accurate. A typical response is shown in Fig. 5.9. However, in the other 7 test sequences, there was some oscillation in the response. A typical oscillatory response is shown in Fig. 5.10.

This type of oscillatory response is characteristic of RNNs. The feedback connections allow for the possibility of instabilities. However, when the network has been accurately trained over the relevant input space, it should be possible to avoid these issues. Our hypoth-

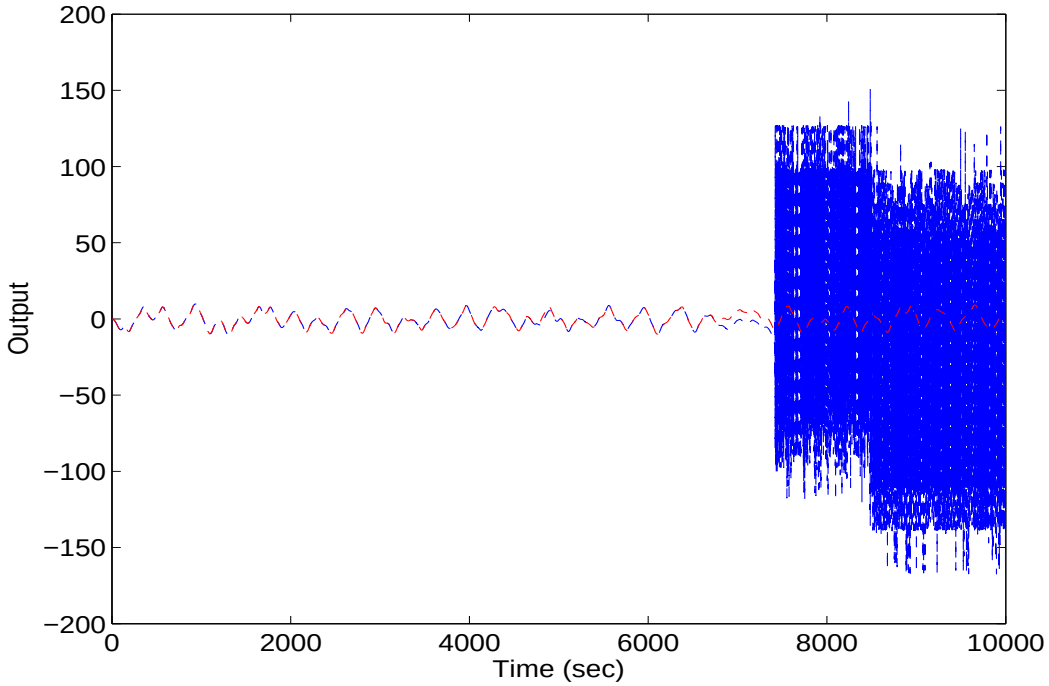


Figure 5.10: Target and oscillatory network response on test sequence.

esis was that the instabilities occur when the network inputs fall outside the space spanned by the training data set. In this case, the network would be extrapolating, and reasonable performance could not be guaranteed. In the next section, we will introduce a method to determine when the network is extrapolating. The objective will be to detect pending oscillatory behavior in a trained network, well before the oscillation occurs.

CHAPTER 6

NOVELTY DETECTION

In this chapter, we will begin in Section 6.1 with the description of a clustering method, the Self Organizing Map (SOM). We will introduce the structure of the SOM and its training procedure. In Section 6.2, the SOM will be used in RNN applications in order to predict oscillatory behaviour in the network response.

6.1 SELF ORGANIZING MAP

When training recurrent networks, it will not be possible to guarantee reasonable network performance if the network inputs move outside the range of the data on which the network is trained. It is important to be able to detect when this extrapolation is occurring. For example, if the RNN is part of a feedback control system, [24], we would want to disable the RNN when extrapolation occurs, and replace with a conventional controller.

Detecting extrapolation is a form of *novelty detection*. We want to know when the inputs to the network fall outside the range of the training data. In this section we propose a type of novelty detection for RNNs. A number of approaches to novelty detection are reviewed in [18]. The approach proposed here is a type of clustering method, in which the inputs from the training set are characterized by a small set of prototype vectors. The minimum distance of a new input to the nearest prototype is used to quantify novelty. For example, if the distance from the new input to the nearest prototype is larger than the maximum distance of that prototype to the cluster of training inputs that are assigned to it, then the new input could be considered novel. (One might also use other similar types of threshold distances to indicate novelty.)

The clustering method we are using here for novelty detection is the Self-Organizing Map (SOM) [25], which is a network for identifying clusters in a data set. This is a topology preserving network, in that neurons within the network have neighbor relationships that are preserved by the training process. The idea will be to train the SOM on combination vectors that represent the inputs to the network, augmented with the target network output. For example, consider the NARX network in Figure 5.4. The tapped delay lines connected to weights $\mathbf{IW}^{1,1}$ and $\mathbf{LW}^{1,2}$ represent the inputs to the static portion of the network. For our example, each of these tapped delay lines has four elements. If we then combine these eight elements with the target network output, we have a nine-element vector. The SOM is then trained on all nine-element vectors in the training set.

6.1.1 BASIC COMPETITIVE NETWORK

The feed forward neural network in which the neurons of the output layer compete with each other to determine a winner is called a competitive network. The winner indicates which prototype pattern is most representative of the input pattern. The competition is implemented by set of connections between the neurons in the output layer. In this section, we are going to illustrate how this competition can be performed and what are the basic training procedures.

Figure 6.1 shows the structure of a competitive network. The prototype vectors are stored in the rows of $\mathbf{W}$. The net input $\mathbf{n}$ calculates the negative distance between the input vector $\mathbf{p}$ and each prototype ${}_i\mathbf{w}$. To simplify the competition layer, we will define a transfer function that does competition with the other neurons to calculate the winning neuron [1]. The output of the competitive network is computed as:

$$\mathbf{a = compet\ (n)} \tag{6.1}$$

The transfer function works by finding the index i^* of the neuron with the largest net input, and setting its output to 1 (with ties going to the neuron with the lowest index). All other outputs are set to 0.

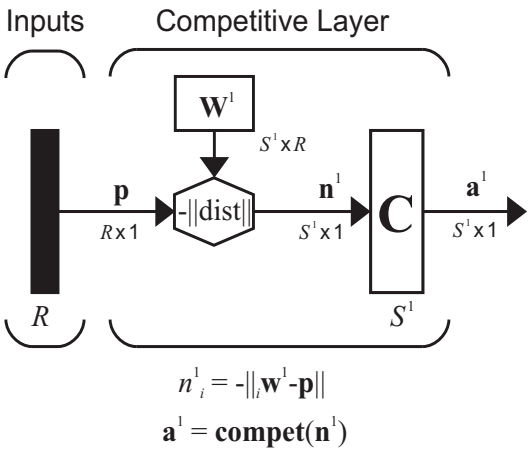


Figure 6.1: Competitive Layer [1]

$$a_i = \begin{cases} 1 & , i = i^* \\ 0 & , i \neq i^* \end{cases} \tag{6.2}$$

where $n_{i^*} \geq n_i, \ \forall i,$ and $i^* \leq i, \ \forall n_i = n_{i^*}$

We can design a competitive network by setting the rows of $\mathbf{W}$ to the desired prototype vectors. However, we would like to have a learning rule that could be used to train the weights in a competitive network, without knowing the prototype vectors [1]. One such learning rule is:

$$_i\mathbf{w} = {}_i\mathbf{w} + \alpha a_i(q)(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \tag{6.3}$$

For the competitive network, $\mathbf{a}$ is only nonzero for the winning neuron$(i = i^*)$. Therefore, we can simplify our learning rule to the Kohonen rule [1]:

$$\begin{aligned} {}_{i^*}\mathbf{w} &= {}_{i^*}\mathbf{w} + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}(q-1)) \\ &= (1-\alpha){}_{i^*}\mathbf{w}(q-1) + \alpha\mathbf{p}(q) \end{aligned} \tag{6.4}$$

and

$$_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) \qquad i \neq i^* \tag{6.5}$$

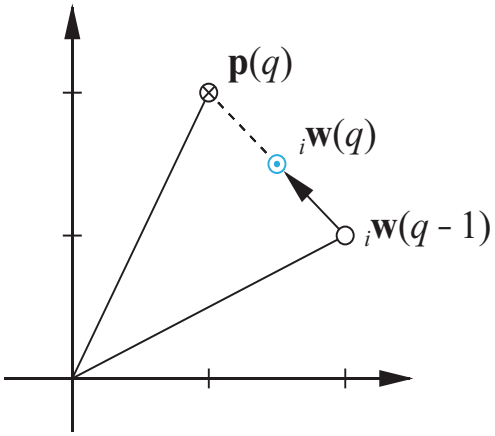


Figure 6.2: Graphical Representation of the Kohonen Rule [1]

Thus, the row of the weight matrix that is closest to the input vector moves toward the input vector. It moves along a line between the old row of the weight matrix and the input vector, as shown in Figure 6.2.

Let's make a simple example, Figure 6.3, to demonstrate how a competitive layer learns to classify vectors. Our competitive network will have three neurons, and therefore it can classify vectors into three classes. Initial weights are randomly chosen and normalized [1].

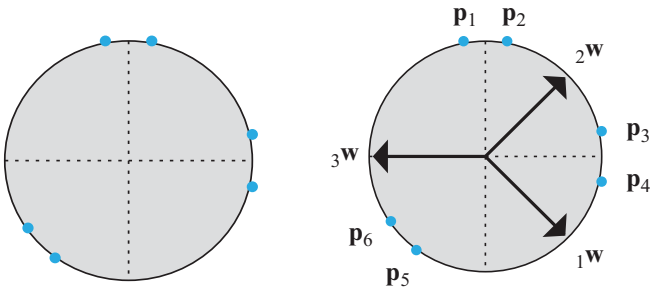
$$ {}_1\mathbf{w} = \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix}_2 \quad \mathbf{w} = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}_3 \quad \mathbf{w} = \begin{bmatrix} -1.0000 \\ 0.0000 \end{bmatrix} \tag{6.6} $$

where $\mathbf{W}$ is

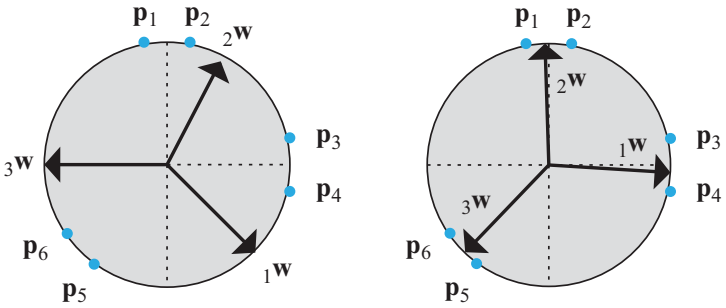
$$ \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ {}_3\mathbf{w}^T \end{bmatrix} \tag{6.7} $$

The data vectors are shown in Figure 6.3(a), with the weight vectors displayed as arrows shown in Figure 6.3(b). Lets present the vector $\mathbf{p}_2$ to the network [1].

$$ \mathbf{a} = \mathbf{compet}(-\begin{bmatrix} \|{}_1\mathbf{w} - \mathbf{p}_2\| \\ \|{}_2\mathbf{w} - \mathbf{p}_2\| \\ \|{}_3\mathbf{w} - \mathbf{p}_2\|1 \end{bmatrix}) = \mathbf{compet}\left(\begin{bmatrix} -1.7634 \\ -0.5796 \\ -1.5467 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{6.8} $$



(a) Sample Input Vectors (b) Initial Weights



(c) Moves toward the closet input (d) Final Weights

Figure 6.3: Graphical Representation of Kohonen rule [1]

The second neuron's weight vector was closest to $\mathbf{p}_2$, so it won the competition ($i^* = 2$) and output a 1. We now apply the Kohonen learning rule to the winning neuron with a learning rate of $\alpha = 0.5$.

$$ {}_2\mathbf{w}^{new} = {}_2\mathbf{w}^{old} + \alpha(\mathbf{p}_2 - {}_2\mathbf{w}^{old}) \tag{6.9} $$

$$ = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} + 0.5 \left(\begin{bmatrix} 01961 \\ 0.9806 \end{bmatrix} - \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} \right) = \begin{bmatrix} 0.4516 \\ 0.8438 \end{bmatrix} \tag{6.10} $$

The Kohonen rule moves ${}_2\mathbf{w}$ closer to $\mathbf{p}_2$, as can be seen in Figure 6.3(c). If we continue choosing input vectors at random and presenting them to the network, then at each iteration the weight vector closest to the input vector will move toward that vector. Eventually, each weight vector will point at a different cluster of input vectors. Each weight vector becomes a prototype for a different cluster [1], as shown in Figure 6.3(d).

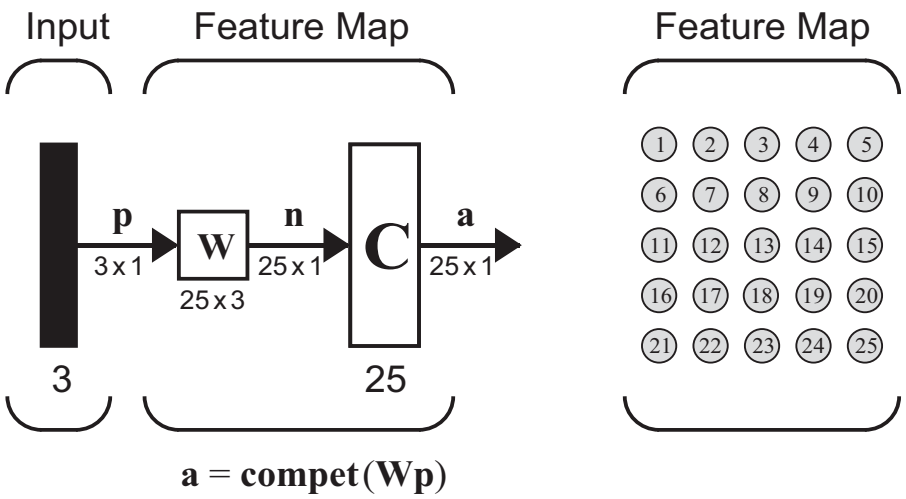


Figure 6.4: Self-Organizing Feature Map [1]

6.1.2 CONCEPT OF NEIGHBORHOOD

In the previous section we did not mention how neurons are physically organized within a layer. This ordering of the neurons forms the topology of the network. In biological neural networks, neurons are typically arranged in two-dimensional layers, in which they are densely interconnected. Figure 6.4 shows a layer of twenty-five neurons arranged in a two dimensional grid. It turns out that this approximation of biological competitive layers not only reinforces the winning neuron itself but also those neurons close to it. The main advantage of a two-dimensional topology is that it allows visualization of high-dimensional spaces [1].

In order to use the biological competitive systems, Kohonen designed his SOM network. In his SOM design, the winning neuron i^* is found using the same procedure as the competitive layer. However, the weight vectors for all neurons within a certain neighborhood of the winning neuron are updated using the Kohonen rule [1].

$$\begin{aligned} {}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \\ &= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) \qquad i \in N_{i^*}(d) \end{aligned} \tag{6.11}$$

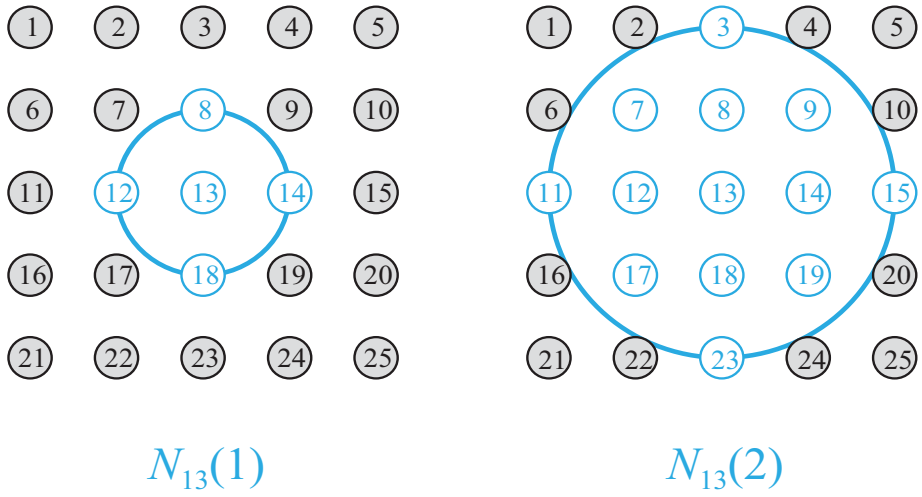


Figure 6.5: Neighborhoods [1]

where the neighborhood $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*:

$$N_i(d) = \{j, d_{ij} \leq d\} \tag{6.12}$$

When a vector **p** is presented, the weights of the winning neuron and its neighbors will move toward **p**. The result is that, after many presentations, neighboring neurons will have learned vectors similar to each other.

To demonstrate the concept of a neighborhood, consider the two diagrams shown in Figure 6.5. The left diagram in the figure illustrates a two-dimensional neighborhood of radius $d = 1$ around neuron 13. The right diagram shows a neighborhood of radius $d = 2$. The definition of these neighborhoods would be [1]:

$$N_{13}(1) = \{8, 12, 13, 14, 18\} \tag{6.13}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\} \tag{6.14}$$

We should mention that the neurons in an SOM do not have to be arranged in a two-dimensional pattern. It is possible to use a one-dimensional arrangement, or even three

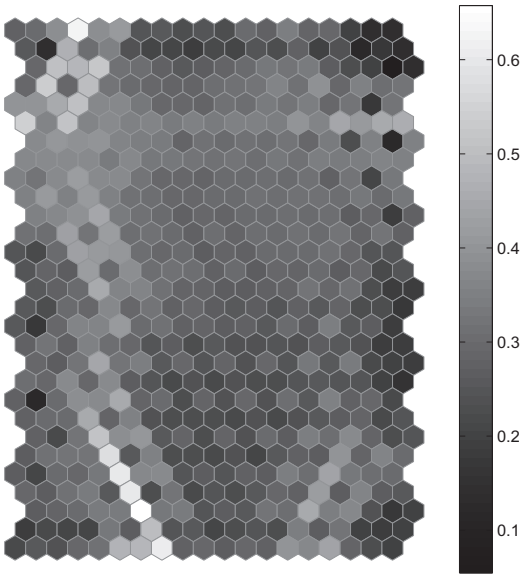


Figure 6.6: U-Matrix for Trained SOM [1]

or more dimensions. For a one-dimensional SOM, a neuron will only have two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line). It is also possible to define distance in different ways. For instance, Kohonen has suggested rectangular and hexagonal neighborhoods for efficient implementation. The performance of the network is not sensitive to the exact shape of the neighborhoods [1].

6.1.3 GRAPHICAL REPRESENTATIONS

After the network has been trained, we will investigate the results to see if the network response is valid. One tool for analyzing the SOM is the unified distance matrix, or *u-matrix*. This visualization tool shows the distances between neighboring neurons in the feature map. The u-matrix has a cell for each neuron in the feature map and an additional cell between each pair of neurons. The cells between neurons are color-coded with the distance between the corresponding weight vectors. The cells that represent the neurons are coded with the mean of the surrounding values. Figure 6.6 shows the u-matrix for our

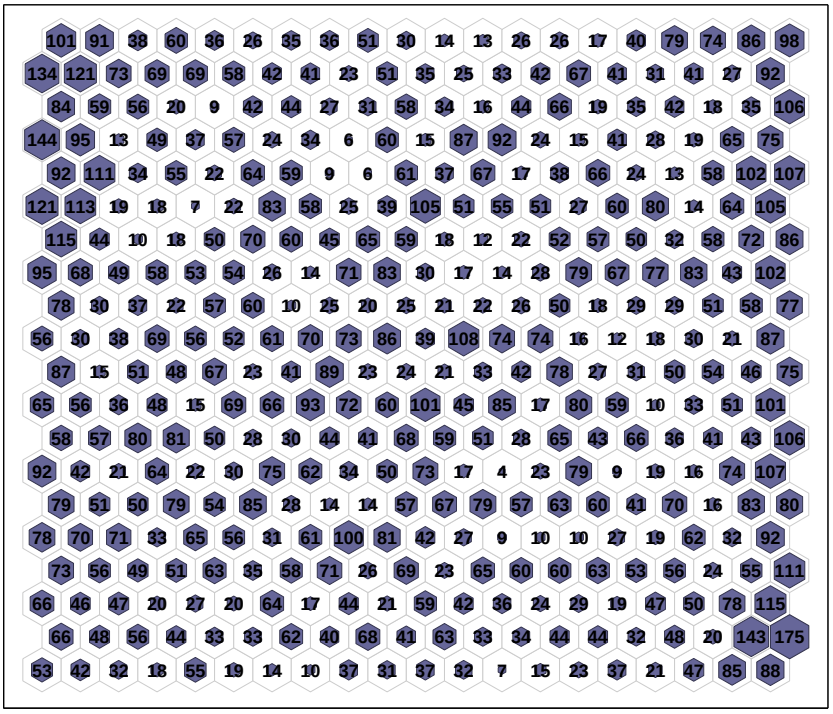


Figure 6.7: Hit histogram for 20x20 trained SOM

trained SOM [1]. The light-colored cells in Figure 6.6, represent large distances between neurons. We can see that there is a string of light colored cells on the left side of the feature map. This indicates that the clusters associated with the neurons on the left side of the map are significantly different than those in the middle and right sides of the map.

To get more insight into how the SOM has clustered the data, we can produce a hit histogram. For this graph, we count how many times each neuron was the winning neuron for the entire data set. Such a graph is displayed in Figure 6.7. In each cell you can see a hexagram. The sizes of the hexagrams indicate how many times the corresponding neuron was the winning neuron.

	Quntization Error	Topographic Error	Average Quntization Error	Average Topographic Error
Network 1	1.2767	0.1940	1.1827	0.1915
Network 2	1.0482	0.1567		
Network 3	0.9535	0.1464		
Network 4	1.4489	0.2657		
Network 5	1.1850	0.1978		
Network 6	1.0457	0.1638		
Network 7	1.4282	0.2525		
Network 8	1.1817	0.1821		
Network 9	1.0581	0.1535		
Network 10	1.2014	0.2031		

Table 6.1: Statistics

6.2 APPLICATION OF SOM TO EXTRAPOLATION DETECTION

We trained an SOM network on data from the single-link robot system that was described in Chapter 5. Consider the NARX in Figure 5.4. The tapped delay lines connected to weights $\mathbf{IW}^{1,1}$ and $\mathbf{LW}^{1,2}$ represent the inputs to the static portion of the network. For our example, each of these tapped delay lines has four elements. If we combine these eight elements with targets network output, we have a nine element vector. The SOM is then trained on all nine elements vectors in the training set. The data set was very large, therefore, we needed to segment the data into 10 sections and train a different SOM on each section.

In order to cluster this large data set, we experimented with several different SOM sizes, such as 10x10, 15x15 and 20x20, and we looked at each individual cluster center after the training. It is important that all members of each cluster be close to the cluster centers.

There are several ways to measure SOM performance, such as the hit histogram and statistics such as topographic error and quantization error.

The hit histogram of Fig. 6.7 is a representation of the trained 20x20 SOM. Each

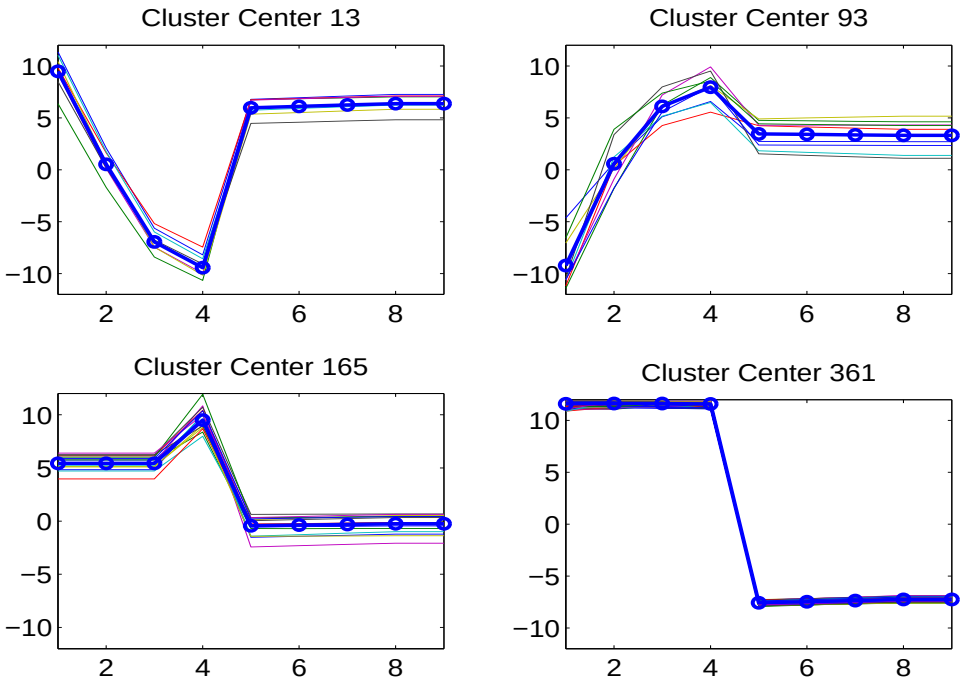


Figure 6.8: Four clusters in the trained SOM network.

hexagon represents a neuron (cluster center), and the sizes of the hexagons indicate the number of training set input vectors associated with each cluster center of the 20x20 SOM. The larger the hexagon, the more inputs are associated with that cluster. We can see that the inputs are well-distributed throughout the network.

Fig. 6.8 illustrates four of the 400 clusters associated with one of the trained SOMs. (The numbering of the clusters starts with 1 in the lower left of Fig. 6.7, and continuing row-by-row to the top right of that figure.) Each subfigure shows a cluster center as a bold line with circles and the inputs that are associated with that cluster as the thinner lines.

Quantization error is the average distance between each input vector and the closest prototype vector. It measures the map resolution. Another measure of SOM performance is topographic error. This is the proportion of all input vectors for which the closest prototype vector and the next closest prototype vector are not neighbors in the feature map topology. Topographic error measures the preservation of the topology. In a well trained

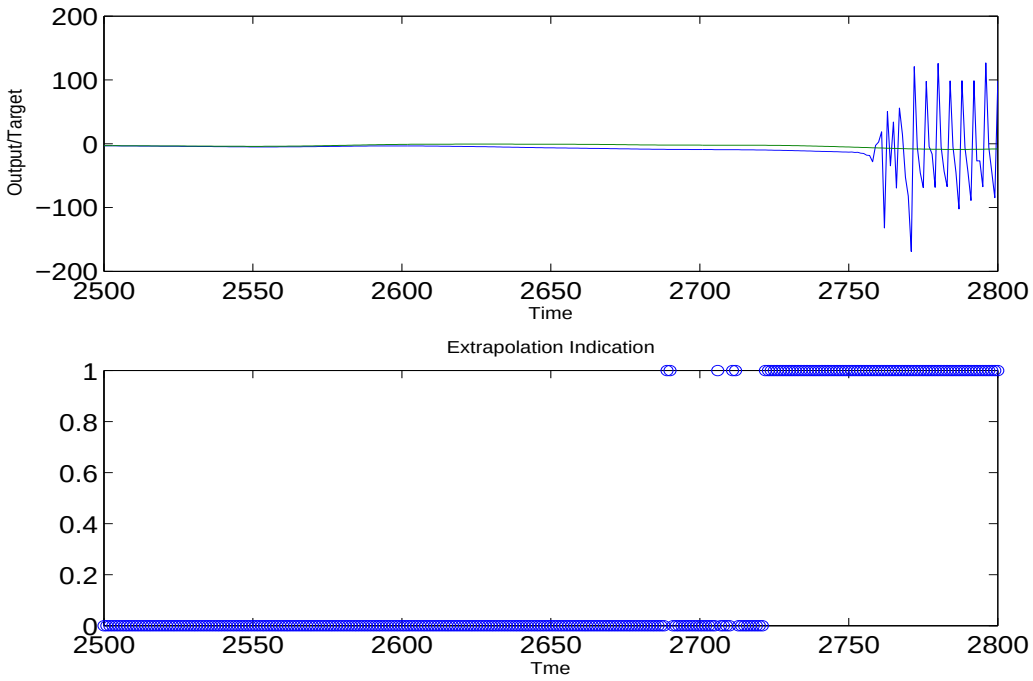
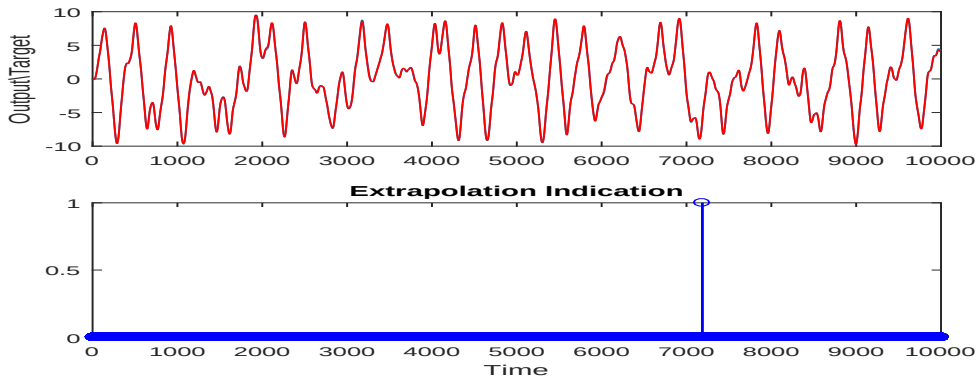


Figure 6.9: Network response and extrapolation detection.

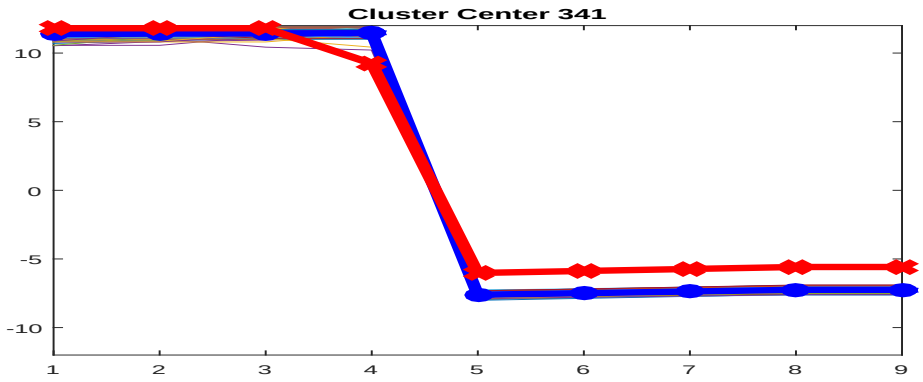
SOM, prototypes that are neighbors in the topology should also be neighbors in the input space [1].

For our trained SOM, the final quantization error was 1.1827, and the final topographic error was 0.1916. This means that for less than 20% of all input vectors, the winning neuron and the next closest neuron were not adjacent to each other (see Table 6.1 for the errors of each SOM).

For each cluster, we have calculated the maximum distance between the cluster center and the most distant member of that cluster in the training set. We then use those maximum distances to determine when the RNN is extrapolating. While the RNN is operating, at each time step we create an input vector to the SOM. (When the target output is not available, the last element of the SOM input vector is replaced with the RNN output.) The distance of this SOM input to the nearest cluster center is compared with the maximum distance associated with that cluster. If the current input distance is larger than the maximum distance, there



(a) False positive extrapolation detection



(b) Cluster Members

Figure 6.10: Illustration of false positive extrapolation detection

is potential extrapolation under way. Fig. 6.9 demonstrates the process. The top subfigure shows the network response for a new test sequence, along with the target output. The bottom subfigure shows the extrapolation indication. The extrapolation flag goes high many time steps before oscillation begins in the network output.

In some cases in the test data, we have an extrapolation indication when no oscillation occurs. Figure 6.10 shows one of these cases. We need to investigate these false positive cases, therefore we need to find the closest cluster center to which this input vector belongs. Figure 6.10(b) shows the prototype vector as a thick blue line and the false positive input vector as a thick red line.

In order to reduce the number of false positives, we can indicate extrapolation only

Indication Width	Sensitivity	Specificity	Average Steps After Indication Before Oscillation	False Positive Rate
1	1	0.2359	63.54	0.7640
2	1	0.5384	66.13	0.4615
3	0.9259	0.6986	58.60	0.3013
4	0.9393	0.9402	49.12	0.0596
5	0.9393	0.9552	47.22	0.0447
6	0.9393	0.9552	45.70	0.0447
7	0.9393	0.9552	43.25	0.0447
8	0.9411	0.9696	44.37	0.0303
9	0.9411	0.9696	41.09	0.0303
10	0.9428	0.9845	39.09	0.0154

Table 6.2: Sensitivity and Specificity

when the distance of the SOM input to the nearest cluster center is larger than the maximum cluster distance for a specified number of consecutive time steps, which we will call the indication width. To select a reasonable indication width, we generated 100 test sequences.

For each indication width we counted the number of time steps before the oscillation occurred for all 100 sequences. Also, we counted the number of false positives, where there is no oscillation within 100 time steps of the oscillation (defined as the latent time). Table 6.2 shows the average number of time steps before oscillation for indication widths ranging from 1 to 10 for all of the 100 test sequences.

Sensitivity and specificity are also statistical measure of performance of classification methods. Sensitivity (true positive rate) measures the proportion of actual positives which are correctly identified (e.g., the percentage of oscillatory sequences which are correctly identified). Specificity (true negative rate) measures the proportion of negatives which are correctly identified (e.g., the percentage of good sequences which are correctly identified).

		True	
		bad	good
Indicated	bad	11	68
	good	0	21
Indication width 1			

		True	
		bad	good
Indicated	bad	22	36
	good	0	42
Indication width 2			

		True	
		bad	good
Indicated	bad	25	22
	good	2	51
Indication width 3			

		True	
		bad	good
Indicated	bad	31	4
	good	2	63
Indication width 4			

		True	
		bad	good
Indicated	bad	31	3
	good	2	64
Indication width 5			

		True	
		bad	good
Indicated	bad	31	3
	good	2	64
Indication width 6			

		True	
		bad	good
Indicated	bad	31	3
	good	2	64
Indication width 7			

		True	
		bad	good
Indicated	bad	32	2
	good	2	64
Indication width 8			

		True	
		bad	good
Indicated	bad	32	2
	good	2	64
Indication width 9			

		True	
		bad	good
Indicated	bad	33	1
	good	2	64
Indication width 10			

Table 6.3: Confusion table

Table 6.2 shows the sensitivity and specificity for different indication widths. The ideal predictor is 100 percent sensitive and 100 percent specific.

The confusion matrix is a table that provides insight into classification performance. Each column of the matrix represents the actual class, and each row represents the predicted class. Table 6.3 show confusion matrices as the indication width varied from 1 to 10.

Based on the results shown in Table 6.2 and 6.3, it appears that an indication width of about 5 provides the best combination of sensitivity and specificity, although slightly larger indication widths do not change the performance significantly. Since increasing the indication width reduces the lead time in predicting the oscillation, we would like to use the minimum indication width that produces reasonable accuracy.

CHAPTER 7

MODELING AND CONTROL OF A SIMULATED MAGNETIC LEVITATION SYSTEM

In this chapter, we will begin in Section 7.1 with a description of training a recurrent neural network, using the modified training algorithm from the pervious chapter, to model a simulated magnetic levitation system. The Self Organizing Map (SOM) is used to collect additional data to improve the training procedure (collecting data wisely in the regions where we are extrapolating) in Section 7.2 . We will test and verify the final trained RNN model after phases of retraining in Section 7.2.2. Next we will introduce the Model Reference Control (MRC) algorithm in conjunction with the plant model in Section 7.3. Finally, we will test and verify the MRC model on the simulated magnetic levitation system.

7.1 TRAINING NARX NETWORK FOR IDENTIFICATION OF PLANT

In this section, we will consider the basic magnetic levitation system. Magnetic levitation has been used in several industrial applications, such as transportation systems.

7.1.1 SYSTEM DESCRIPTION

In our simulated magnetic levitation system system, we will suspend a magnet above an electromagnet. A magnetic levitation train works in a similar manner. The purpose and goal of this magnetic levitation system is to control the position of a magnet above an electromagnet.

Figure 7.1 shows the magnetic levitation system, which consists of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the

β	α	g	M	$i(t)$	$SamplingInterval$
12	15	$9.8(\frac{m}{s^2})$	$3(Kg)$	$-1\ to\ 4(A)$	0.010

Table 7.1: Simulation Parameters for the Magnetic Levitation

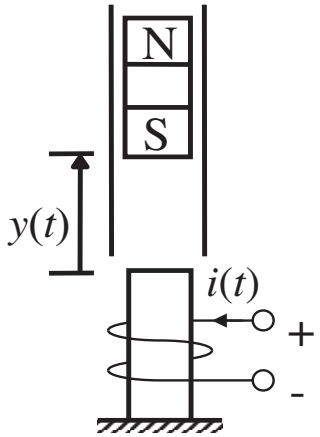


Figure 7.1: Magnetic levitation system [1]

vertical direction. The equation of motion of the magnet is

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M} \times \frac{i^2(t)sgn[i(t)]}{y(t)} - \frac{\beta}{M} \times \frac{dy(t)}{dt} \tag{7.1}$$

where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current in the electromagnet, M is the mass of the magnet, g is the gravitational constant, β is a viscous friction coefficient and α is a field strength constant. Table 7.1 shows the simulation parameters.

7.1.2 SYSTEM IDENTIFICATION

In order to model this system using RNNs, first we need to collect a set of training data. We used Simulink as a tool to gather data from this dynamic system, and we applied random inputs consisting of a series of pulses of random widths and heights, known as a skyline function, as we discussed in Section 5.2. An example is shown in Figure 7.2.

We use a NARX network, shown in Figure 5.4, to model this system. The network we

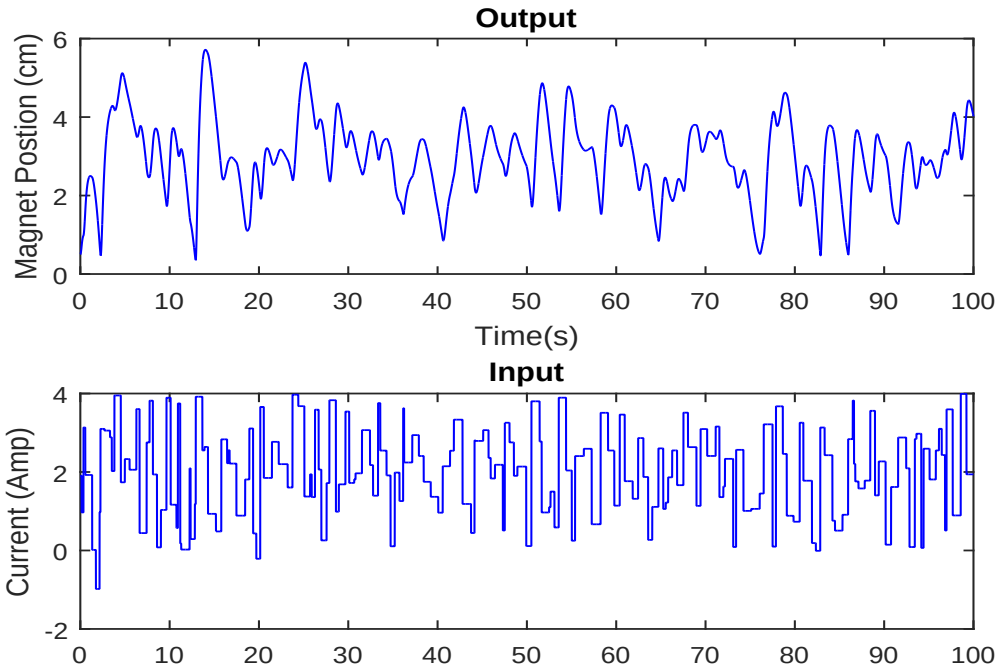


Figure 7.2: Training data for magnetic levitation

Training data				
Name	Min height	Max height	Min width	Max width
Data_Train1-20	-1	4	0.1	1

Table 7.2: Training data skyline range for magnetic levitation

used had 3 input and output delays (the prediction begins with the fourth data point) and also initially had 15 hidden neurons.

A total of 20 sequences were generated to produce the entire training data set, in order to cover the full range of required network operation. Figure 7.3 shows a histogram of magnet position in the training set (in centimeters), which demonstrates the coverage of the modeling. The magnetic levitation system is to be modeled as the magnet position varies in the range from 0 to 6 centimetres and as the current varies from -1 to 4 amp. Table 7.2 shows the height and width of random pulses for 20 training sequences.

The first step of the modified training algorithm in Section 3.3 is open-loop (one step

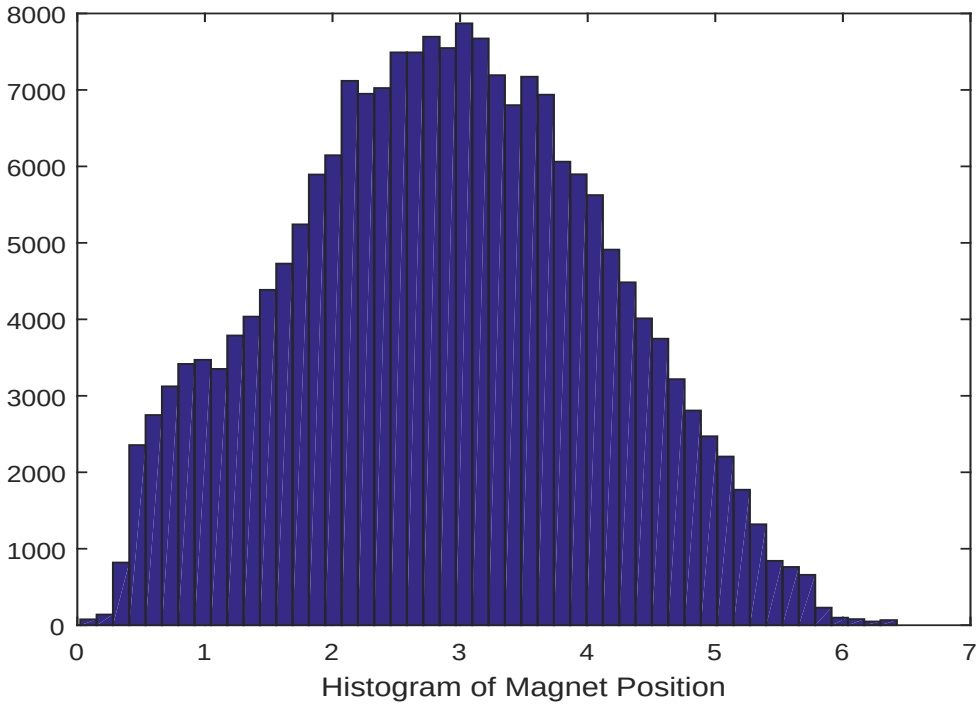


Figure 7.3: Histogram of Magnet position contained in the training set.

ahead prediction) training. In this phase, as we discussed in Section 3.2.1, there are two inputs to the series-parallel architecture shown in Figure 3.4(b) which are the input sequence and the target sequence.

After performing the open loop training, we do multiple step ahead prediction for the NARX network (close the feedback connections). The horizon step is chosen automatically by the new procedure discussed in Section 4.2.

Table 7.3 shows the detailed horizon step selection for the magnetic levitation system. The algorithm adaptively chose the next horizon step as we discussed in Section 4.2. We reached μ_{max} in just 14 training segments, as shown in Table 7.3, out of the 67 training segments needed for the prediction horizon to reach the full length of the original sequence.

After training was completed for the maximum prediction horizon, the network response closely followed the target response for all 20 of the original training sequences. Figure 7.4 shows the results of the 9998 step ahead prediction. We hardly see any differ-

Training Segment	Pervious Prediction Horizon	Horizon Step	# of Times Mu max reached	Training Segment	Pervious Prediction Horizon	Horizon Step	# of Times Mu max reached
1	openloop	1	0	41	4394	100	2
2	2	1	0	42	4494	203	0
3	3	1	0	43	4697	241	4
4	4	1	0	44	4938	151	1
5	5	1	0	45	5089	250	1
6	6	98	0	46	5339	221	0
7	104	98	0	47	5560	39	0
8	202	97	0	48	5599	14	0
9	299	95	0	49	5613	12	0
10	394	94	0	50	5625	250	0
11	488	137	1	51	5875	101	0
12	625	147	27	52	5976	250	0
13	772	144	3	53	6226	240	0
14	916	146	0	54	6466	300	0
15	1062	148	0	55	6766	300	0
16	1210	144	0	56	7066	258	0
17	1354	145	0	57	7324	300	0
18	1499	148	0	58	7624	114	0
19	1647	17	0	59	7738	300	0
20	1664	146	0	60	8038	198	0
21	1810	146	5	61	8236	300	0
22	1956	149	0	62	8536	247	0
23	2105	91	1	63	8783	50	0
24	2196	1	0	64	8833	334	4
25	2197	1	0	65	9167	193	0
26	2198	195	0	66	9360	294	0
27	2393	196	2	67	9654	344	0
28	2589	197	0				
29	2786	122	0				
30	2908	2	0				
31	2910	29	0				
32	2939	197	7				
33	3136	193	1				
34	3329	198	3				
35	3527	196	0				
36	3723	194	0				
37	3917	197	0				
38	4114	194	0				
39	4308	84	0				
40	4392	2	0				

Table 7.3: Horizon step selection table for magnetic levitation

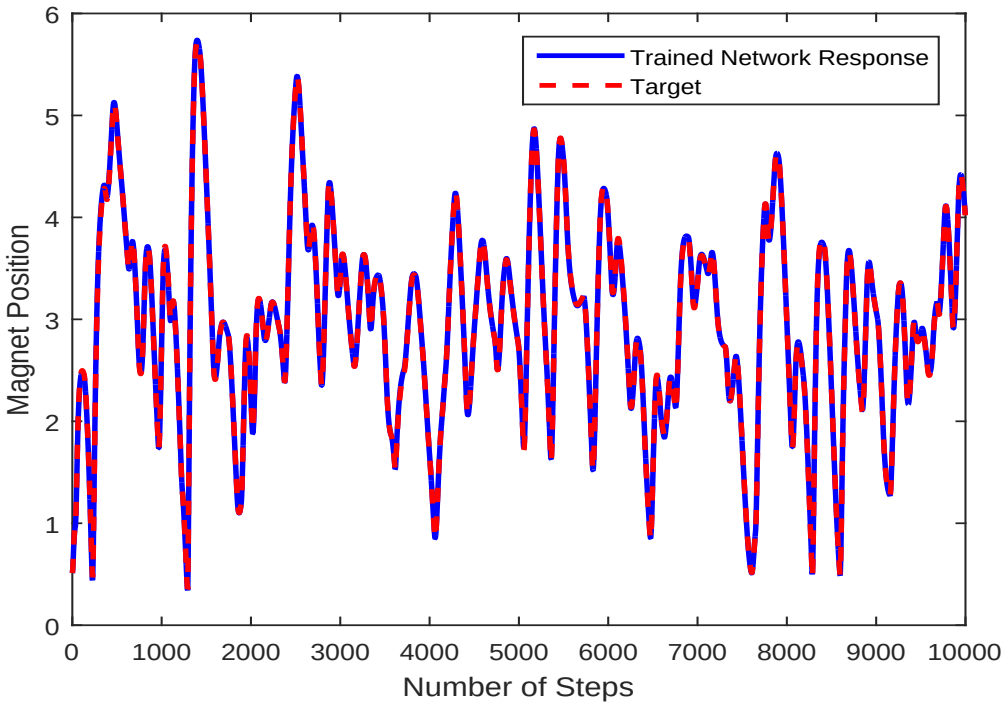


Figure 7.4: Target and trained network response on training data set

ence between the actual position of the magnet and the position predicted by the NARX network, and the mean square error is quite small (The MSE value is 2.5724×10^{-4}).

In order to validate the NARX network, we generated 100 additional test sequences of 10,000 time points, which were not used for training. In some of the cases the network is extrapolating, and the network response is inaccurate. As we discussed in Chapter 6, we use the SOM to detect the extrapolation. Then, we add the extrapolating test sequences to the training data, in order to expand the region where the network provides an accurate fit. In Section 7.2, we will go into the details of data collection and show more statistics in this regard.

7.2 SOM FOR NOVELTY SAMPLING

It is unlikely that the original data set will effectively cover the full range of conditions where the network will be used. As we discussed in Chapter 6, the RNN is extrapolating

when network inputs fall outside the space spanned by the training data set. We are going to collect additional training data wisely using the SOM. Then, we will retrain the NARX network with the new data combined with the initial training data set. This procedure is known as *novelty sampling*. This will be done in phases until no novel conditions are detected after many additional tests.

7.2.1 RETRAINING PROCESS

For the magnetic levitation system, we performed two phases of novelty sampling and retraining. The novelty sampling provides more information in the regions where network performance is not reasonable, and after the retraining the network we expect an enhanced network response.

7.2.1.1 FIRST RETRAINING PROCESS

In the first retraining process, we trained the SOM over the 20 training sequences. We used the same procedure discussed in Section 6.2. We trained a 20x20 SOM with a 7 element input vector.

Table 7.4 shows the statistics of the SOM for 100 generated test sequences. The false positive rate drops from 69 percent to 24 percent when the indication width is 3, so we chose an indication width of 3 for collecting the new sequences. We add sequences which have indication widths of 3 and more to the original training data set. The new training data set will grow in phases until there are no significant errors on new tests sets. Even after a network has been implemented, the SOM can continue to monitor performance, and the network can be retrained when a significant number of sequences have been added to the training set.

In this part of retraining we collected 39 additional sequences, therefore, the first retraining will use 59 sequences. The new data set is increased by almost 3 times.

We were able to reach the full horizon and complete the training process. The first

Indication Width	Sensitivity	Specificity	Average Steps After Indication Before Oscillation	False Positive Rate
1	1	0.0106	18.0	0.9894
2	0.75	0.3095	0.1818	06905
3	0.8077	0.7568	-0.80	0.2432
4	0.8077	0.7703	-1.80	0.2297
5	0.8077	0.7703	-2.80	0.2297
6	0.8077	0.7973	-3.80	0.2027
7	0.8077	0.8108	-4.80	0.1892
8	0.8077	0.8243	-5.80	0.1757
9	0.8077	0.8514	-6.80	0.1486
10	0.8077	0.8649	-7.80	0.1351

Table 7.4: Sensitivity and specificity first retraining process

phase of retraining is completed and it is time to check the new network with some test sequences. In this part of process, we should be able to see an improvement on the test results. (We expect the number of oscillatory responses to decrease.)

7.2.1.2 SECOND RETRAINING PROCESS

This phase of retraining has 59 training sequences. We generated 100 more test sequences. The network response at this stage for the 100 test sequences was outstanding for all the sequences except 1. Figure 7.5 shows this test sequence, which has an oscillatory response.

An indication width of 2 was used for the new data set, and 8 new sequences were collected and added to our training data set for a total of 67 sequences. We performed modified training on this new data set. We begin with one step ahead prediction and then multi step ahead prediction to reach the full horizon. The training algorithm reached the full horizon step with very small MSE. The second stage of retraining is completed.

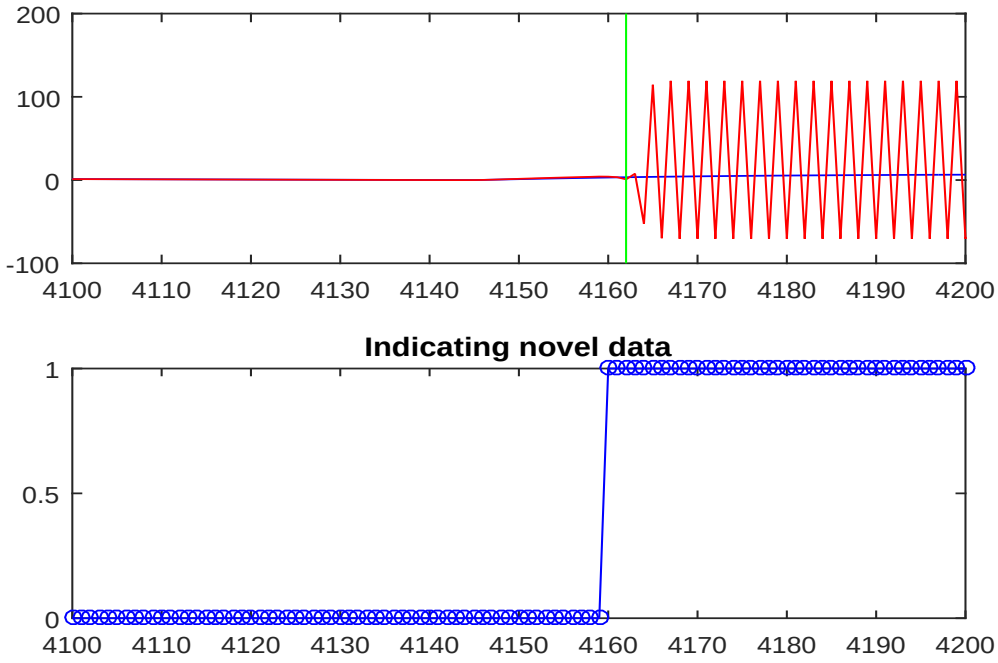


Figure 7.5: Only test sequence with oscillatory response

7.2.1.3 FINAL RESULTS OF THE RETRAINING PROCESS

In this part of the retraining process, we trained an SOM on the 67 training data sequences; in addition, another 100 test sequences were generated. We calculated all the statistics to collect more data if the SOM detected any extrapolation. The network response at this stage was perfect, and no extrapolation or oscillation was detected. The results shows that the retraining process was successful and enhanced the network performance. Finally, we found a stable network for all test sequences. This stable network makes it easier to train the controller.

7.2.2 TEST AND VERIFY THE MODEL

In summary, the novelty sampling method combined with the modified training algorithm results in a very stable and trustable model for the magnetic levitation system. Figure 7.6 shows the final results for the test sequence. As shown in the figure, it is very hard to

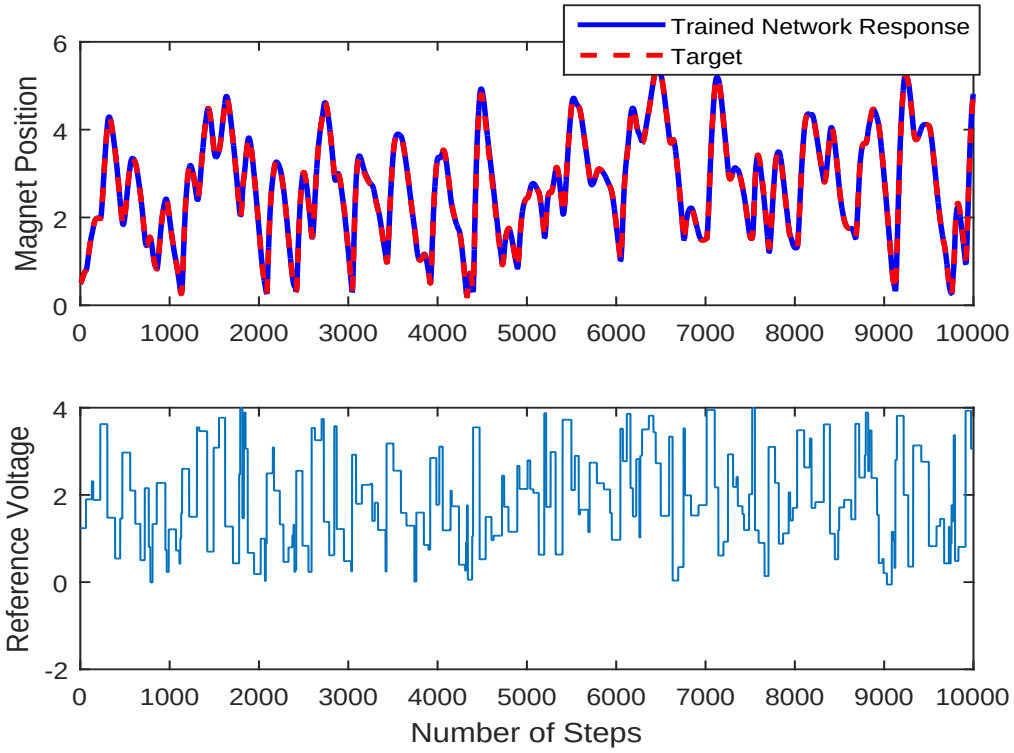


Figure 7.6: Target and trained network response after retraining for a test sequence

distinguish between the network response and target.

We created Simulink blocks of an actual plant and a NARX model. The Simulink model gives us a flexibility in simulating the plant model in real time in conjunction with the trained NARX network. This approach is visual and gives us a better understanding of the overall system. Figure 7.7 shows the Simulink block diagram of the system. We will see, in Section 7.3 and Chapter 8, the advantages of having Simulink blocks check the controller and program the microcontroller.

7.3 MODEL REFERENCE CONTROL TRAINING

The first step in neural network control is modeling of system dynamics (system identification) as shown in Figure 7.8. A popular network for nonlinear system identification is the NARX network. After we train a neural network to represent the plant, we are going to use the model reference adaptive control architecture to control the plant output. The MRC

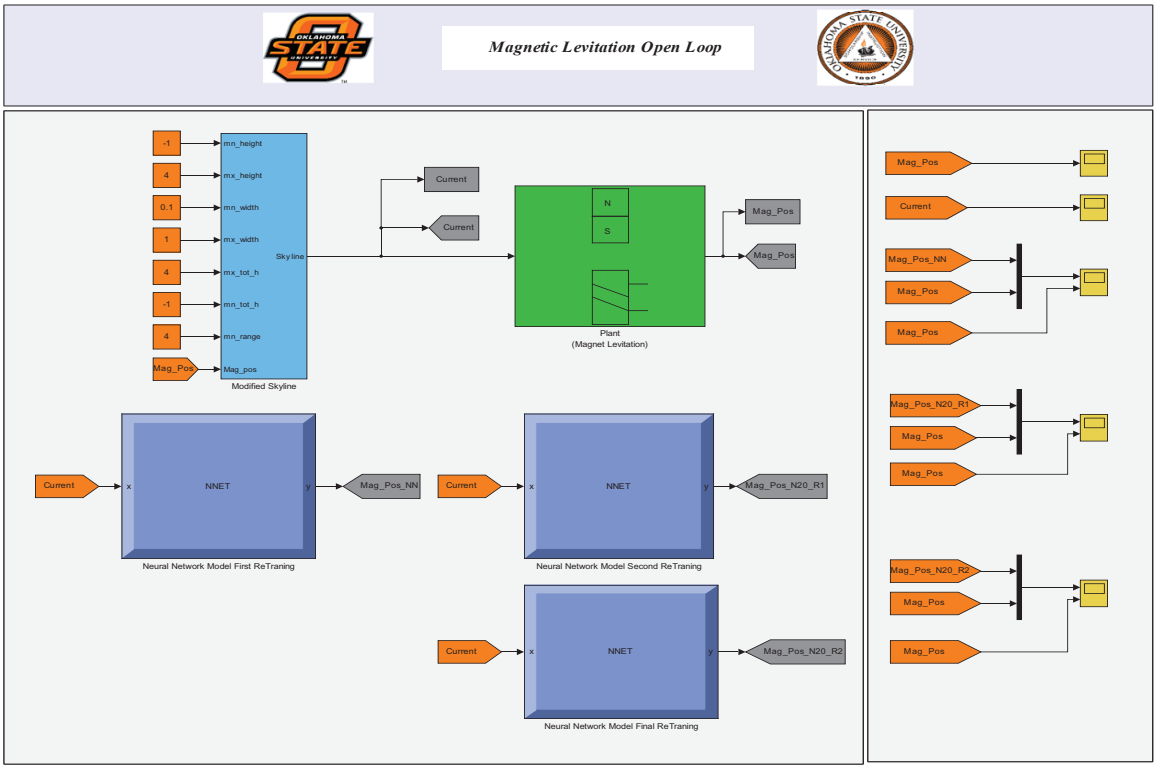


Figure 7.7: Simulink block diagram of magnetic levitation

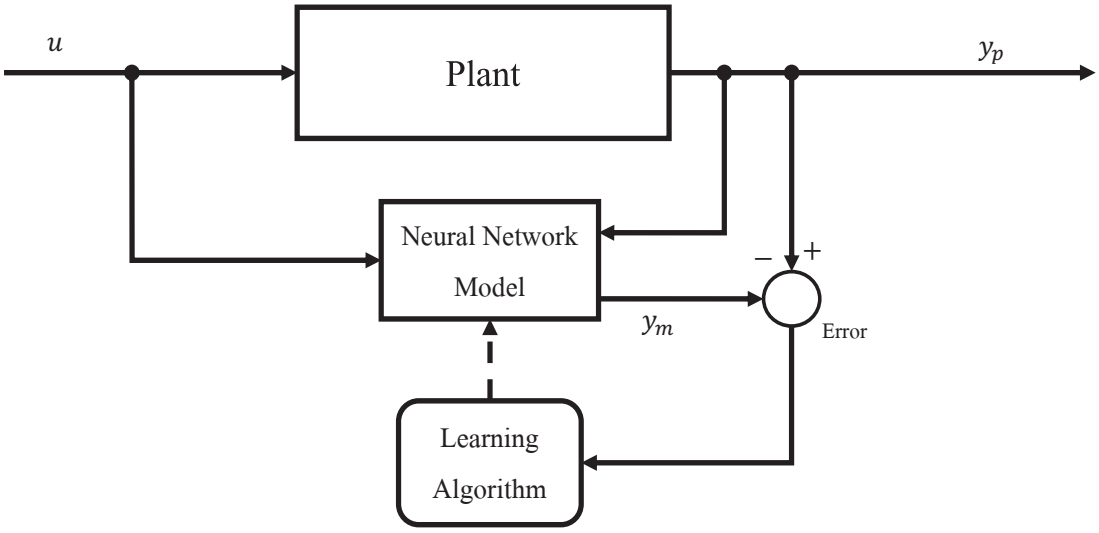


Figure 7.8: Plant Identification

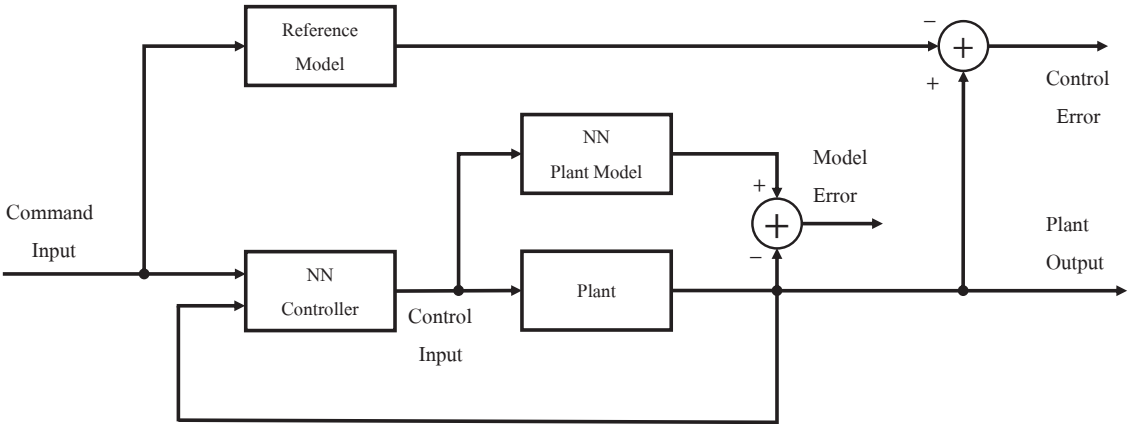


Figure 7.9: Model reference adaptive control structure

architecture was first introduced in [26], and it consists of two parts:

The first part is a plant model, as shown in Figure 7.9. The plant model is identified using the NARX network. The second part is an NN controller, and the controller network is trained so that the plant output follows the reference model output.

Based on the work of the pervious section, we have a very stable and accurate model of the magnetic levitation system. This model has been tested and verified using a wide variety of test sequences, and has demonstrated very accurate predictions. Therefore, the plant identification part of the MRC is completed and we are ready to train the controller network.

The complete architecture of the MRC network is shown in Figure 7.10. The model reference control is a 4 layer dynamic network consist of two major parts: the first two layers make up the controller, and the third and fourth layers make up the plant model (trained NARX network). There exist three sets of controller inputs: delayed reference inputs, delayed controller outputs (plant inputs), and delayed plant outputs. The reference input is delayed by 2, and the controller outputs and plant outputs are delayed by 3. The number of delays increases the order of the plant. We use 10 hidden neurons for the controller, and the trained NARX plant model has 20 hidden neurons.

The next step is to train the controller network. In the open-loop training process,

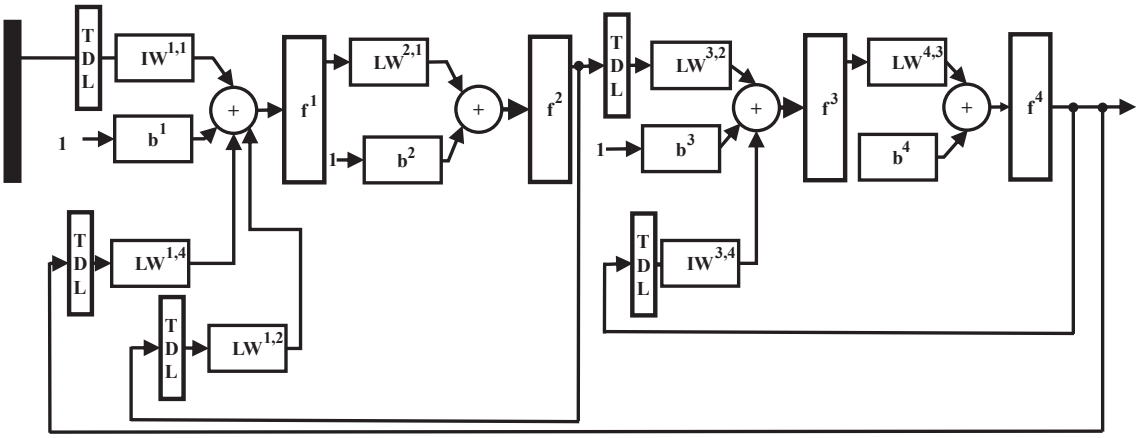


Figure 7.10: Model reference control network

we replace the plant feedback terms with the targets (output of the reference model.) This means that we cut two feedback connections from the fourth layer (plant outputs) to the first layer and also to the third layer. The controller network (first and second layer) feedback connection remains. Since the third and fourth layer are the trained NARX network (plant model), their weights and biases are fixed during the controller training process. We also set the weights of the controller output layer to zero, in order to give the plant zero initial input (preventing the instability of the closed loop system in the first iteration).

After the open loop training is completed, we perform multi step ahead prediction (train the closed loop network). Figure 7.11 shows the result of controller training. We can see that the plant output is very close to the reference model output.

Now we can test the MRC by applying a test input (skyline function) to the trained MRC network. Figure 7.12 shows that the plant model output follows the reference model input. The reference model is a critically damped second order system.

The trained MRC and the actual plant with the NARX model are all converted to the Simulink block diagrams shown in Figure 7.13. Simulink is s great tool for simulating control applications. It is easy to visualize the main control structure and to access all the system's states. Note that in each run of magnetic levitation system a new random test

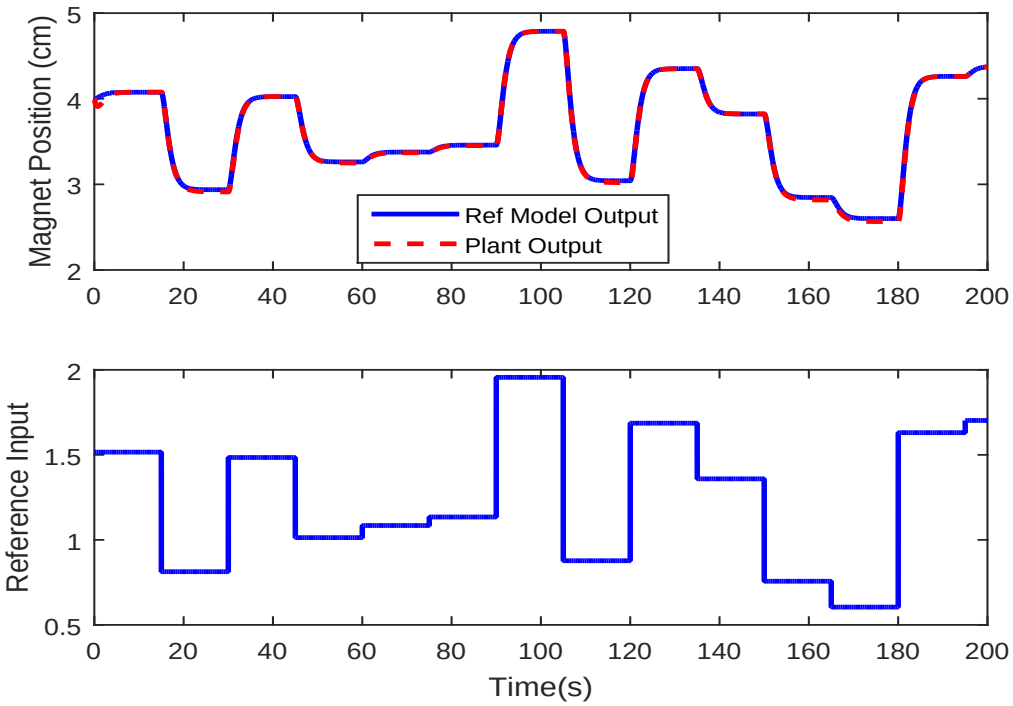


Figure 7.11: Model reference control training.

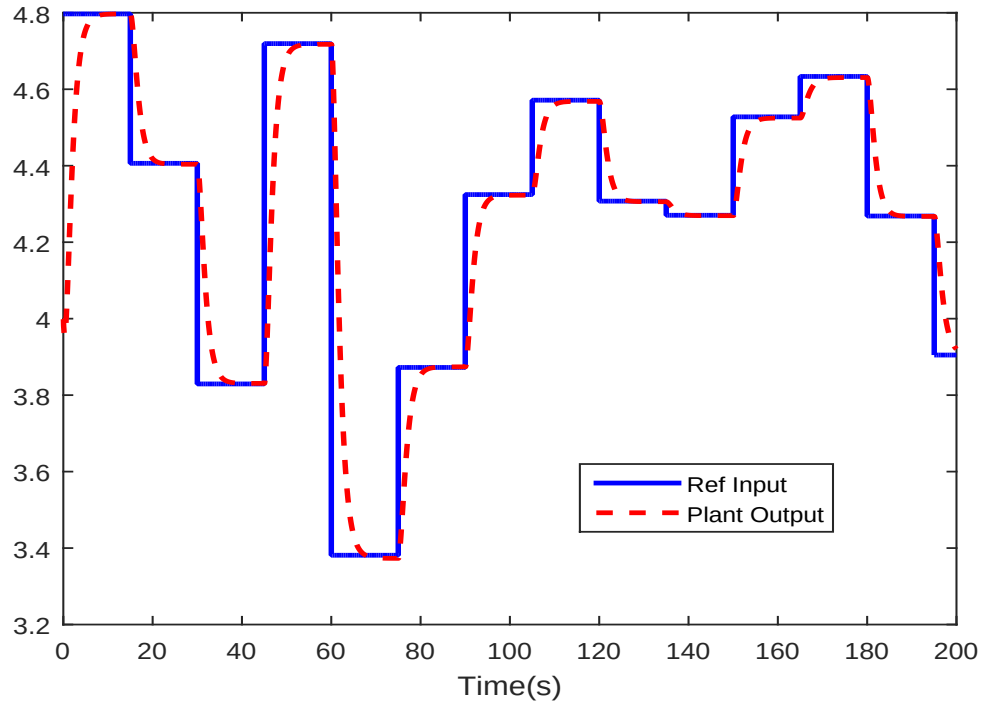


Figure 7.12: Model reference control

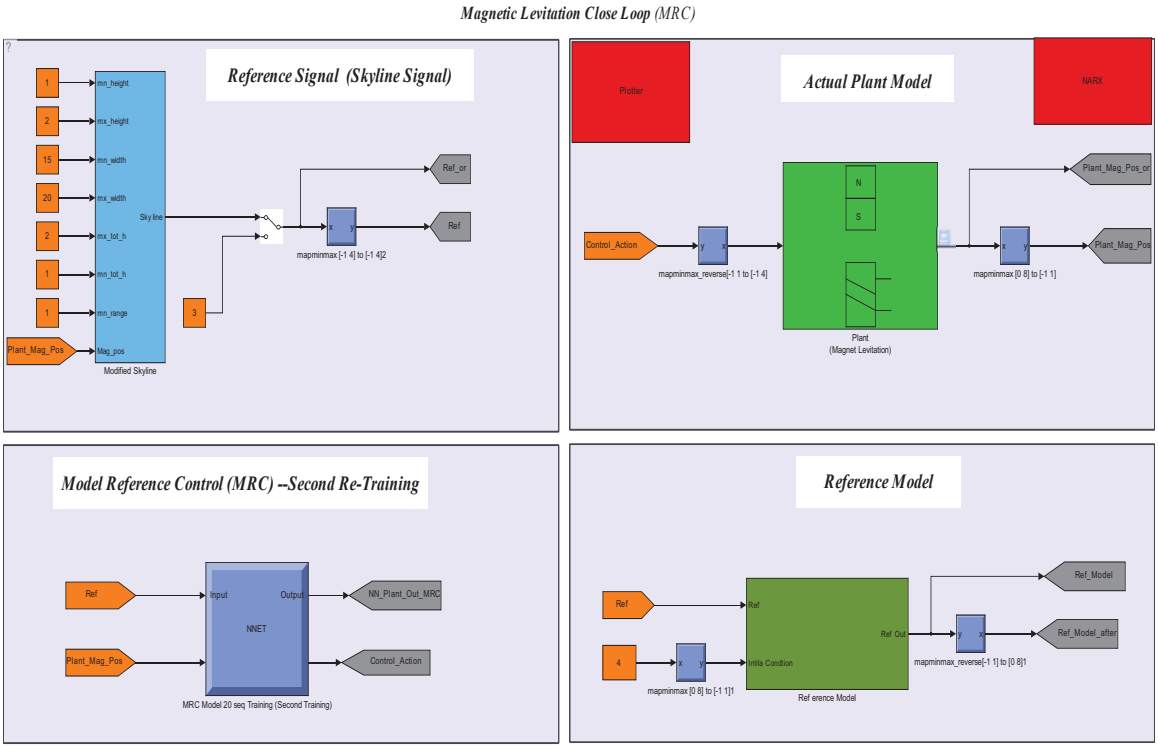


Figure 7.13: MRC Simulink diagram

sequence will be generated.

As shown in Figure 7.13, the control action from the MRC is controlling the simulated plant. In this way we are simulating the real control problem. In other words, in real life the simulated plant is replaced by the physical system, and we need to drive the physical system with the controller.

The modified training algorithm, in conjunction with the novelty sampling method, pays off in controller network training. The very accurate and stable NARX plant model helped improve the controller training procedure. We did not see any oscillation in the MRC response.

7.4 USE SOM TO OBTAIN MORE DATA FOR CONTROLLER

In this section we collect more data for the MRC training using novelty sampling. Training an SOM on MRC is more complicated than on the NARX network. We have 3 tapped

delay lines shown in Figure 7.10. The tapped delay lines connected to weights $\mathbf{IW}^{1,1}$ and $\mathbf{LW}^{1,4}$ represent the inputs to the static portion of the network. The tapped delay lines connected to weights $\mathbf{LW}^{1,2}$ represent the output of the controller. This tapped delay line data needs to be filled with the output of the controller. We need to simulate the controller to obtain these data.

The SOM for MRC has a 10 element input vector, three tapped delay lines for each feedback connection plus the target. We trained the SOM on the 20 sequences of training data set, and we generated 100 test sequences. After looking at the statistics, we decided to choose the indication width of 3 or more. We retrained the MRC with 120 sequences, and the MRC response improved.

SOM clustering is very sensitive in MRC training. Sometimes extrapolation is detected when the cluster centers are very well clustered and the input distance is slightly greater than the max distance.

In summary, although we did not see any oscillatory behaviour, the SOM did detect some extrapolation. We added more sequences, and it improved the MRC responses.

CHAPTER 8

EXPERIMENTAL RESULTS

In this chapter, we design the MRC architecture for the magnetic levitation system in Section 8.1. We describe the magnetic levitation design process in Section 8.1.1. The main component in this design process is the core processing unit with its software packages, which is described in Section 8.1.2.1. We explain the building process (using 3D printing technology) in Section 8.1.3. Next, we describe the magnetic levitation setup, collect experimental data from the setup and apply the modified training algorithm in Section 8.2. Then, we train the MRC network and implement it on the physical system in Section 8.3. The classical PID controller structure is also explained in Section 8.4. Finally, we compare the classical PID controller and the neural network controller in Section 8.5.

8.1 DESIGN AND BUILD THE MAGNETIC LEVITATION SYSTEM

In this section, we illustrate the design process for building the magnetic levitation system which is shown in Figure 7.1. In most practical control system applications, there exists 3 basic components: an embedded system that plays the role of the main processing unit (e.g., microprocessors, microcontrollers, or microcomputers), the physical system to be controlled and finally actuators and sensors which provide the input and output to the system. In the next sections we provide details of our design process and each of these basic components.

8.1.1 DESIGN PROCESS

Our goal in this design process is to build a magnetic levitation system. First, we need to consider couple of factors in this process which may be helpful later on. We did all of our simulation and training of the RNNs in the Matlab environment. The testing and verifying of the MRC controller is done in the Simulink. Our strategy is to use Matlab and Simulink, instead of C coding, during our design process.

In most control system projects, we need to have a data acquisition card or other tools that can collect data from the system (e.g Labview, dSpace or microcontrollers). The data acquisition cards are usually very expensive and not portable, we need to find a better solution in order to have a such a tool to collect data. We used a microcontroller to do this task: it is a very economical choice compared to other options. Also, they are portable and very compact. We used an Arduino microcontroller to collect data for the magnetic levitation system. We will explain the specifications of the Arduino microcontroller in Section 8.1.2.

Any microcontroller needs to be programmed to perform a specific task. For example, if we need to output a 5 volt DC signal from the microcontroller, we need to write a program to output the 5 volt signal to one of the board ports. Usually, any microcontroller uses specific software for downloading the programs. The Arduino also has its own software called Arduino IDE. However, the Arduino can also be programmed through the Simulink software. This makes it easier to implement algorithms (as opposed to C coding). In other words, programming the Arduino using Simulink gives us the opportunity to do visual programming and not much standard coding is involved. The standard way of programming the Arduino requires more experience than the Simulink approach, despite the fact that both approaches will produce the same results.

The next phase in the design process is to design and build a magnetic levitation system. There are several ways to build such a system, like machining (using a CNC machine), welding, and 3D printing. Recently, 3D printing technology has become very popular.

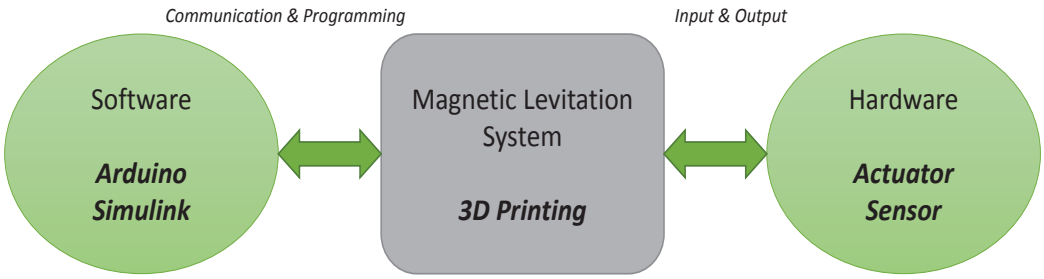


Figure 8.1: Design Process.

Some of the reasons for this are

- It is economical.
- It is easy to design and build in small space.
- A variety of 3D printers are available.

We used 3D printing technology for our magnetic levitation structure. Our design was done with SolidWorks software. We will explain the mechanical design in Section 8.1.3.

Finally, we need to add sensors and actuators to complete our system. We used an electromagnet as an actuator (to drive the magnet in the vertical direction) and an IR distance sensor to measure the magnet position. The design process is shown in Figure 8.1.

8.1.2 SOFTWARE AND HARDWARE

In the following sections, we will describe the software and hardware integration of the magnetic levitation system.

8.1.2.1 ARDUINO AND SIMULINK

The Arduino Mega 2560 is a microcontroller board that uses the ATmega chip. It has 54 digital input and output pins,of which 14 are used as PWM output. There are also 16 analog inputs. This board has 4 UARTs (RS232 communication protocol), and uses a 16 MHz crystal oscillator. The Arduino Mega 2560 has a USB connection and a power jack with a reset button. This board contains all the necessary components that are needed for our purpose. It can connect to a computer with a USB cable and it can be powered by a power jack or a battery [4].

The Arduino Mega 2560 uses the ATmega1280 chip which has 128 KB of flash memory for storing code [4]. This amount of memory is enough for our purposes.

The Arduino Mega 2560 uses several methods of communication, such as serial, wireless and bluetooth. One of its 4 UARTs is the main serial communication port (pins 0 and 1), as shown in Figure 8.2. After connecting the Arduino Mega 2560 to a computer, the software program creates a virtual com port on the computer side. When the board is downloading the program the RX and TX LEDs turn on (indicating that the board is downloading the program). The pin outs are shown in Figure 8.2.

In our design we used Simulink for programming the Arduino Mega 2560. Therefore, we need to download the driver for the Arduino Mega and its software(see [4]), and then we need to configure Simulink in the configuration parameters toolbar (see [27]). Simulink uses the same main serial communications to program the Arduino, and to deploy the model into the board. After Simulink configuration is done, we can compile, build and deploy any Simulink model into the Arduino board. For example, if we need to read data from an analog sensor, we bring the analog input data block from the Arduino repository library. Then, we will press the *build button* on the Simulink toolbar in order to download and deploy it into the board (Simulink automatically generates the code for that block and will put it into Arduino board). In this way the board is programmed to read analog input from a chosen pin. In the next section, we will show the components needed to build the

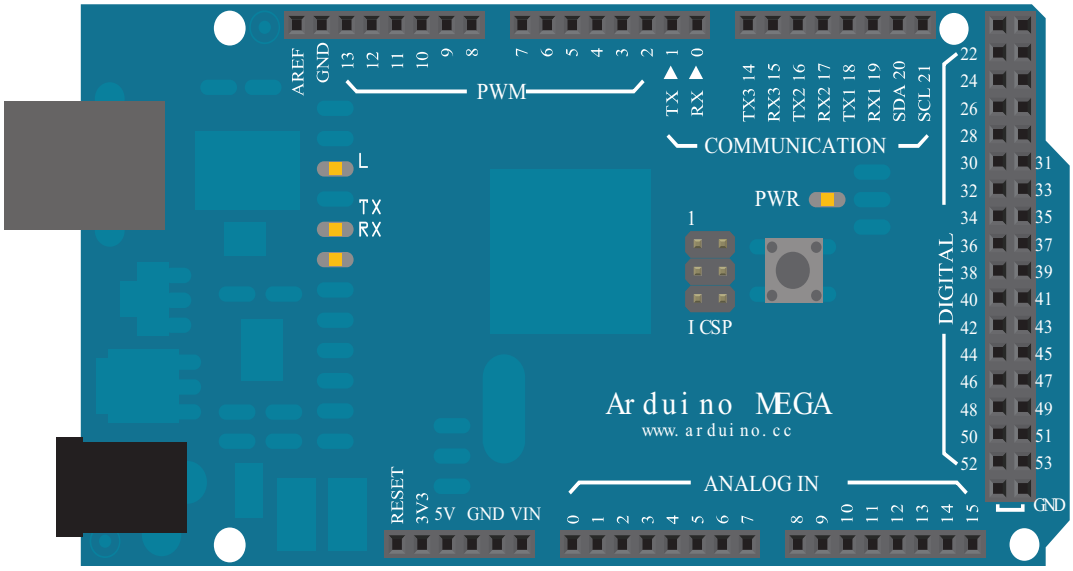


Figure 8.2: Arduino Mega 2560 [3].

Figure 8.3: Arduino Mega Programming [4].

Figure 8.4: Sharp Sensor [5].

MaximumRange	*MinimumRange*	*Sampling*	*Outputtype*	*Supplycurrent*
$15cm$	$2cm$	$60Hz$	*AnalogVoltage*	$12mA$

Table 8.1: Sharp specification

magnetic levitation system.

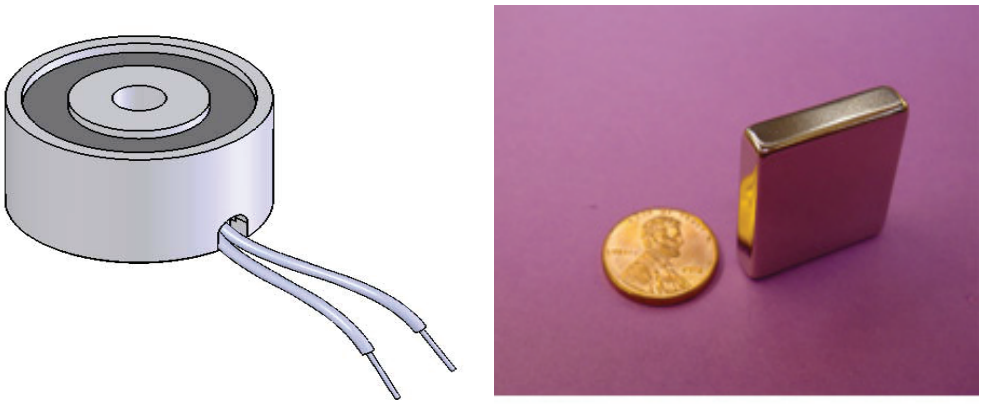
8.1.2.2 SENSOR AND ACTUATOR

In this section, we introduce four main components of the magnetic levitation system.

Distance Sensor: We used the Sharp IR analog distance sensor (GP2Y0A51SK0F) that operates in the shortest-range shown in Figure 8.4. This sensor has higher resolution at shorter ranges. This sensor features a detection range from 2 cm to 15 cm. It is very easy to use because the distance is indicated by analog voltage [5], and the Arduino has analog inputs. The Sharp distance sensor uses IR technology, and it is more economical that the sonar or laser range finders.

The Sharp sensor interface with the microcontroller is straight forward. The analog output will be connected to the analog input of the Arduino. This IR sensor has the characteristics shown in Table 8.1

Electromagnet: An electromagnet uses a magnetic field produced by an electric cur-



(a) Electromagnet [28] (b) Magnet [29]

Figure 8.5: Electromagnet and Magnet.

rent. There is no magnetic filed when the current is turned off. An electromagnet consists of many wires wrapped closely together around a core to create a magnetic field. The magnetic core is usually made of iron, to increase the strength of the field.

We can control the magnetic filed by changing the amount of electric current in the windings. An electromagnet needs a consistent supply of current to maintain its magnetic field.

Circular Electromagnets are used in manually operated or automated applications such as motors, MRI devices, speakers and etc. It is energized with a D.C. power source, and the outer side (or pole) produces a north or south magnetic field that surrounds the center, as shown in Figure 8.5(a).

Motor Shield: The Arduino Motor shield shown in Figure 8.6 is a dual full-bridge driver (which uses the L298P chip) designed to drive inductive loads, such as solenoids, permanent magnet DC motors and stepping motors. This motor shield is plug compatible with the Arduino Mega and can drive an electromagnet.

This motor shield uses an external power supply, and it can provide up to 2 Amps of current. The L298P chip can drive a 7-12 Volt electromagnet. Also, we use the Pulse Width Modulation (PWM) signal to control the voltage. The Arduino supports a PWM

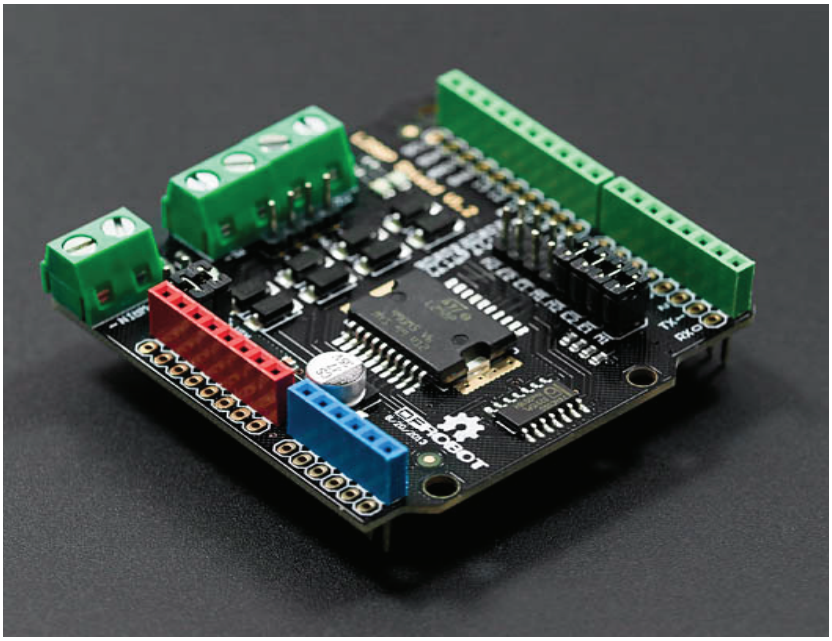


Figure 8.6: Motor Shiled [6].

signal block with duty cycle resolution of 0-255. Zero produces zero voltage, and 255 corresponds to 100% duty cycle, or maximum average voltage.

Permanent Magnet: The permanent magnet that will be levitated by the electromagnet is a neodymium block magnet. It is $1'' \times \frac{1}{4}'' \times 1''$ (see in Figure 8.5(b)). The poles are located on the top and bottom side of the magnet, not the large flat side.

8.1.3 SOLIDWORK DESIGN AND 3D PRINTING

SolidWorks is a solid modeling computer-aided design program for mechanical engineers to draw and visualize objects in 3D space. We used this software to design each part of the magnetic levitation system. Then, we assembled the parts to have the final design. Figure 8.7 shows the final design of the magnetic levitation system.

The next step is to build the design. We used 3D printing technology due to its ease and availability. In 3D printing, layers of plastic material are formed to create any object. We used the *ROBO 3D R1* printer to build the magnetic levitation system. This 3D printer has 100 Micron Maximum Resolution and uses 1.75mm ABS, PLA plastic and Flexible

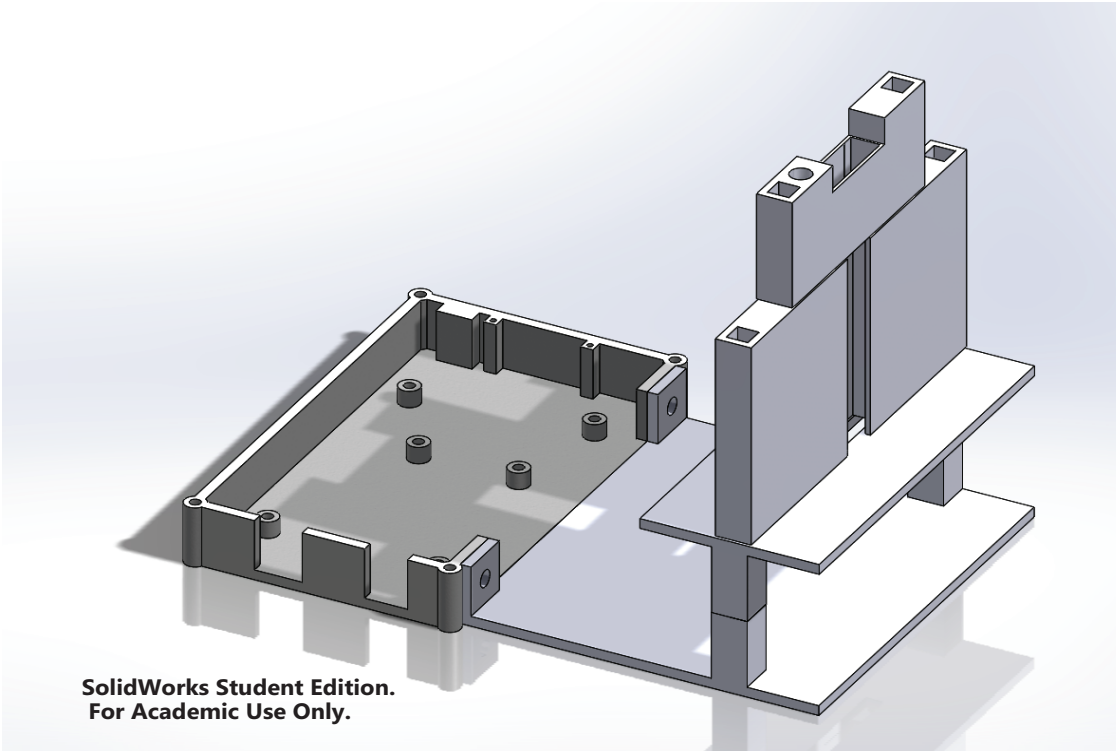


Figure 8.7: SolidWork Final Design.

Filament.

8.1.4 ASSEMBLY AND TEST

We need to put the sensor, electromagnet, Arduino and motor shield in place and connect all the wires. We need to connect the Motor Shield to the electromagnet and connect the IR sensor to the analog input pin. The external power source is used in this case (maximum 12 Volt). Figure 8.8 shows the setup with all the connections and its components.

After assembling the experimental setup, we need to connect the board to a computer with a USB connector. Simulink is configured to program the Arduino to read the sensor data and drive the electromagnet. This part of the system integration is very important in order to obtain the right sensor data in the right sampling interval.

Figure 8.9 shows the major sections of the open loop Simulink model. The *Skyline Function* block produces the random input voltages with random widths, and we convert

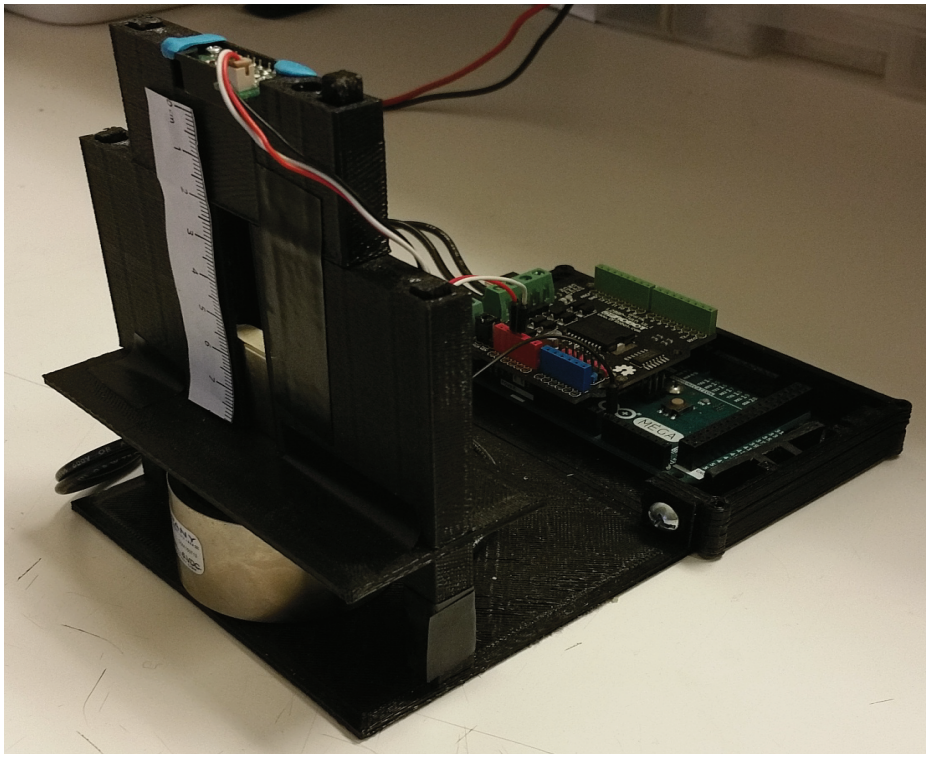


Figure 8.8: Magnetic levitation setup.

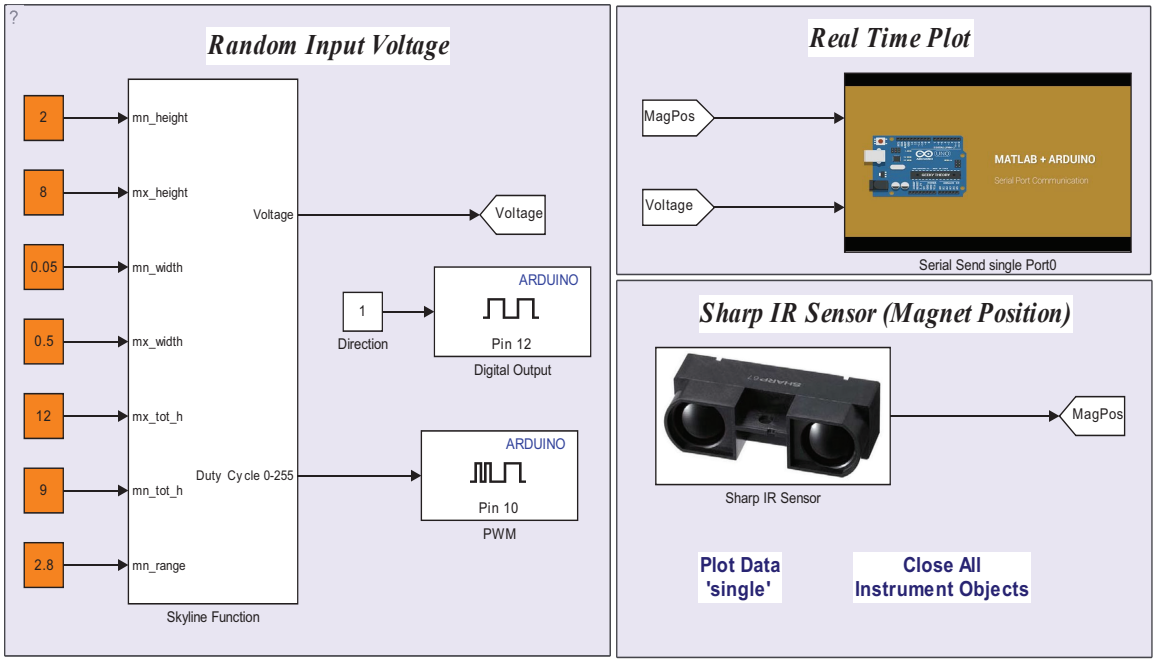


Figure 8.9: Experimental Open loop System .

these random voltages into digital values between 0-255. These values are sent to the PWM block (pin 10), and it will produce the equivalent voltage value. The *Sharp IR sensor* block is connected to the analog input (analog pin 8) and reads the magnet distances (we used a look up table block to convert the voltage value in to a distance). The *Serial Sent signal Port 0* block is the communication block which transfers data to and from the board. Finally, after we put these blocks together, we press the *build button* in the Simulink toolbar, and it generates the code and downloads it to the board. Now we can read the real time input/output data through the main serial communication port zero.

In the next section, we obtain the experimental data from the magnetic levitation system and use them to train the NARX network. The training process remains the same as Chapter 7, and we will show the capability and power of the NN apprach in this regard.

8.2 TRAIN THE NRAX MODEL WITH THE REAL DATA

In order to model the magnetic levitation experimental system using RNNs, first we need to collect a set of training data. In this case, the input data is the voltage (produced by the skyline function) which goes into the motor shield and the target data is the magnet position which comes from the Sharp IR distance sensor.

Note that the IR senor is on the top of the experimental setup and the electromagnet is placed under the magnet. Therefore, for ease of looking at figures, we converted the magnet positions in such way that the bottom is 0 *cm* and the top is the close to 5*cm*. In other words, when we apply a positive voltage the magnet goes up and when there is no voltage the magnet drops due to gravity. Also, since this IR has some noise in its readings, we applied a 1-D median filter to remove the noise spikes. The 1-D median filter is very useful in our case. An example of one filtered sequence is shown in Figure 8.10.

Figure 8.11 shows a sample of an experimental training data set. We used voltage from 9 to 12 volt in order to keep the magnet levitated. In other words, the magnet will not move and sit on the base when the voltage is less than 9 volt.

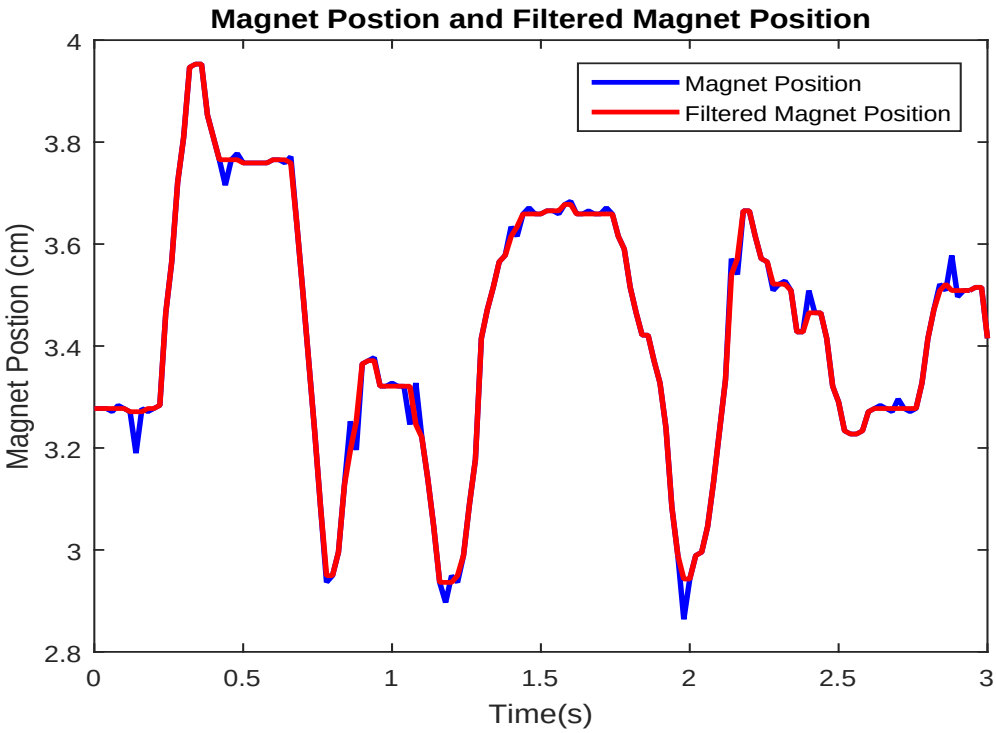


Figure 8.10: Filtered magnet position

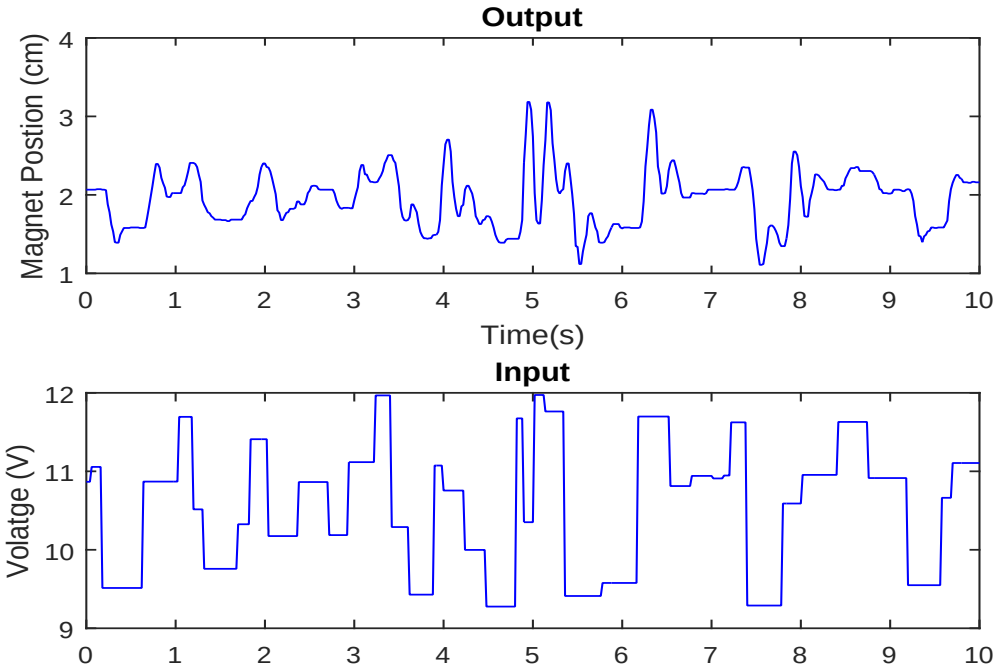


Figure 8.11: Experimental training data for magnetic levitation

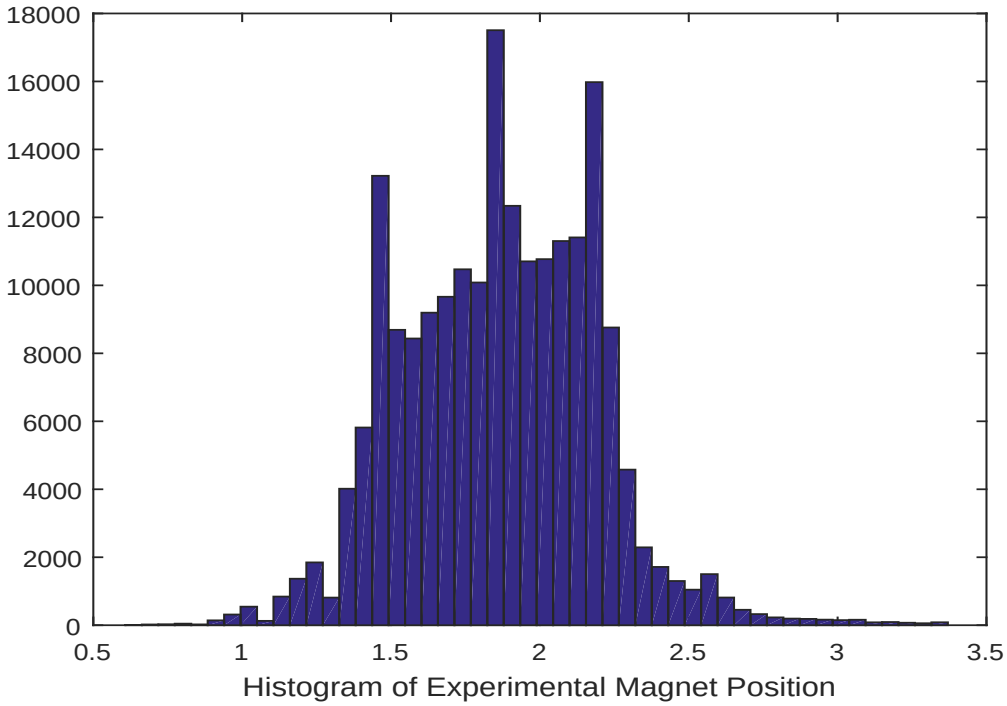


Figure 8.12: Histogram of the experimental magnet positing

A total of 20 such sequences were obtained from the experimental setup to make the entire training data set. Figure 8.12 shows a histogram of magnet position in the training set (in centimeters), which demonstrates the coverage of the modeling (we will use this histogram information in the MRC). The magnetic levitation system is to be modeled as the magnet position varies in the range from 0 to 5 centimetres and as the voltage varies from 9 to 12 volt.

We use a NARX network, shown in Figure 5.4, to model this system. The network we used had 4 input and output delays (the prediction begins with the fifth data point) and also initially had 10 hidden neurons.

We performed the first step of the modified training algorithm which is open-loop (one step ahead prediction) training. In this phase, as we discussed in Section 3.2.1, there are two inputs to the series-parallel architecture shown in Figure 3.4(b) which are the input sequence and the target sequence.

After performing the open loop training, we do multiple step ahead prediction for the NARX network (closing the feedback connections). The horizon step is chosen automatically by the new procedure discussed in Section 4.2.

After training was completed for the maximum prediction horizon, the network response followed the target response for all 20 of the original training sequences. However, the response in the beginning of the sequences is slightly off compared to the end of the sequences, and that is due to heat. We ran the experimental setup for 200 seconds with a sampling time of 0.020 seconds in order to get the training data. Usually, the electromagnet is hot when the data is collected. After some time, we have to let the system cool down for a while before collecting more data. For data that was collected when the system was hot, the network response is very close to the actual magnet position. Figure 8.13 and 8.14 show the results of the 9997 step ahead prediction for one training sequence when the electromagnet is cool at the beginning of the sequence and when it is hot at the end of sequence.

Now it is time to test and verify the model. We used 5 more test sequences and checked the results. The network response is very close to targets. Figure 8.15 shows an example of a test sequence. No oscillatory responses were detected and we did not find any large errors on the test results. Therefore, we continued to go a further step into controller training.

8.3 MODEL REFERENCE CONTROLLER (NEURO CONTROLLER)

The first step in MRC training is system identification of the plant. We used the trained NARX network in the pervious section to represent the plant. Then, we begin to train the controller.

We used the same MRC architecture shown in Figure 7.10. The model reference control consists of a 4 layer dynamic network with two major parts: plant model and controller. The reference input is delayed by 3, and the controller outputs and plant outputs are delayed by 4. The number of delays increases the order of the plant. We use 10 hidden neurons for the controller, and the trained NARX plant model has 10 hidden neurons.

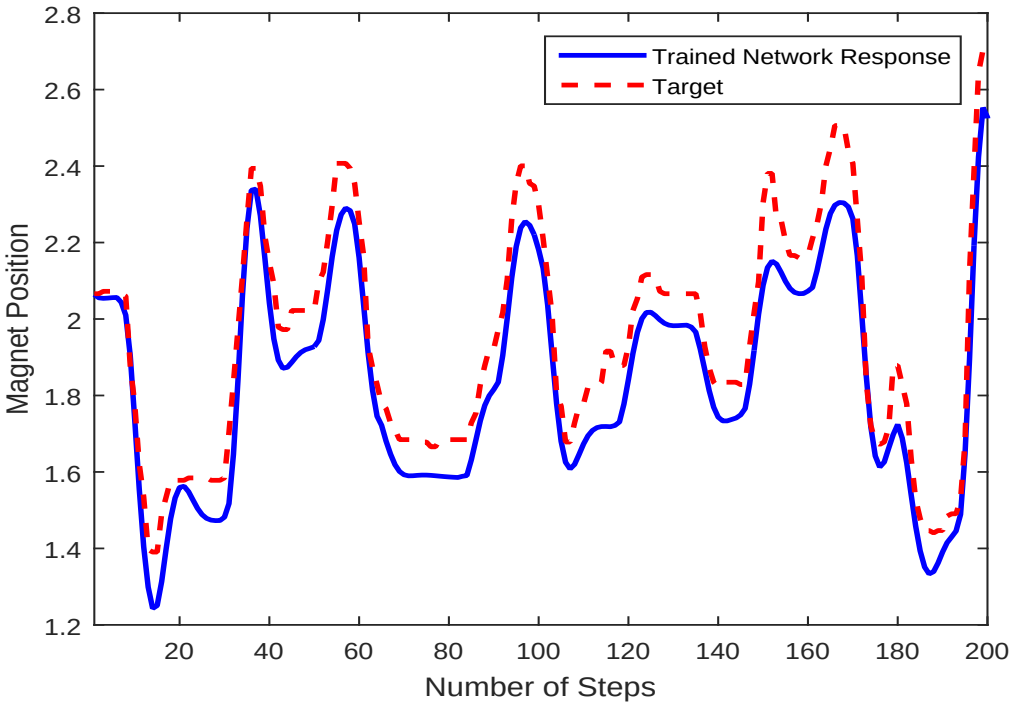


Figure 8.13: Target and trained network response (Cool)

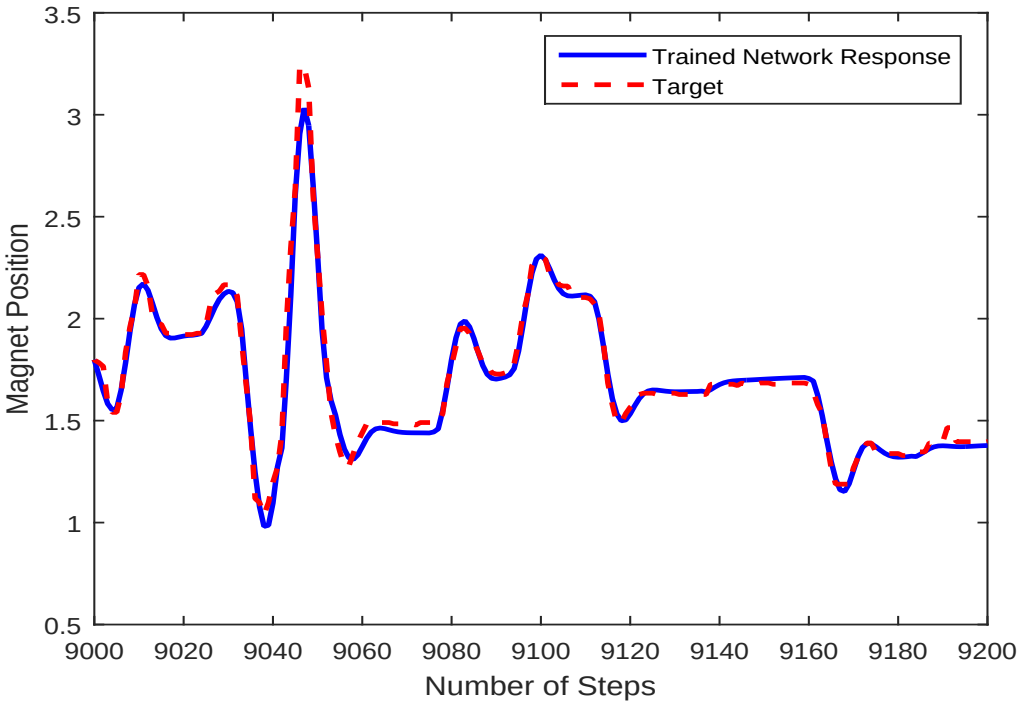


Figure 8.14: Target and trained network response (Hot)

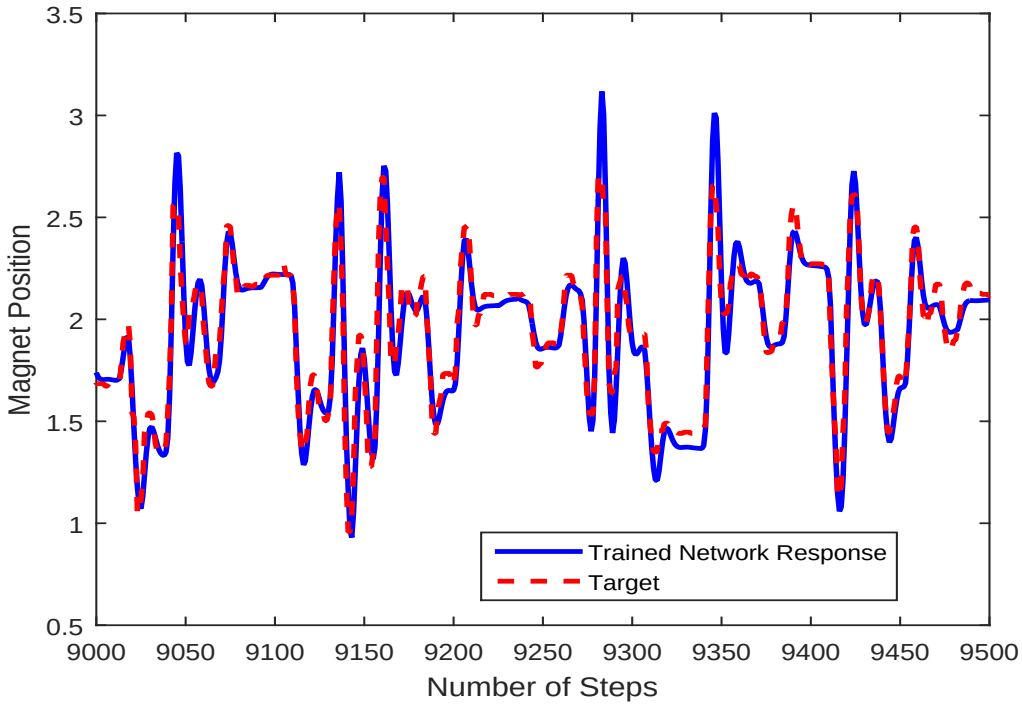


Figure 8.15: Target and trained network response for test sequence

We did open-loop training first. We replaced the plant feedback terms with the targets and cut two feedback connections from the fourth layer to the first layer and also to the third layer. The other feedback connections remain the same as before. Since the third and fourth layers are the plant model, their weights and biases are fixed during training. We also set the weights of the controller output layer to zero, in order to prevent the instability of the closed loop system in the first iteration.

After the open loop training is completed, we perform multi step ahead prediction and train the closed loop network. We created a Simulink block of the MRC, and we put it into the Arduino to test the MRC controller on the real system. The closed loop Simulink diagram is shown in Figure 8.16 and 8.17.

Figure 8.16 shows the closed loop system with a linear filter for the magnet position reading. Since there is noise in the sensor reading, we used the linear filter in order to reduce the noise in the magnet position measurements. There are four sections in this

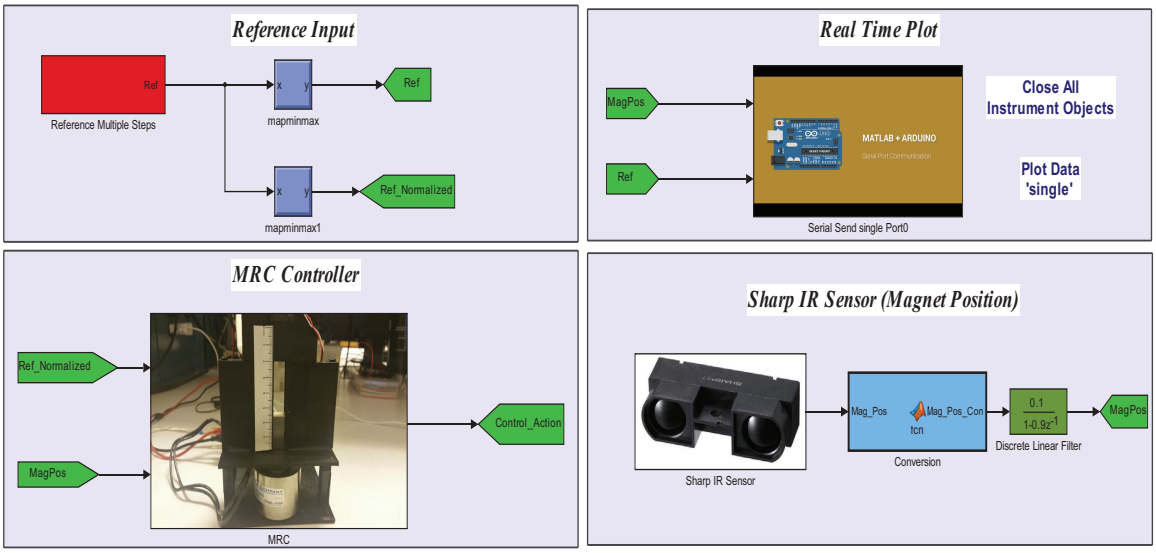


Figure 8.16: Closed loop Simulink block diagram (MRC with linear filter)

figure: reference input voltage (skyline function); sharp IR sensor, with the conversion and the linear filter; MRC controller, which consists of the MRC controller and the magnetic levitation setup inputs (PWM block and direction); and finally the real time plot, which shows the real time sensor reading and input voltage injected to the magnetic levitation system.

8.3.1 NEURAL NETWORK FILTER

Figure 8.17 shows the closed loop system with an NN filter. Since we have an accurate model of the plant (which we can use to predict magnet position) and noisy sensor readings, we combined these two in order to enhance the position estimate. The NN filter works in the following manner. If the absolute value of the difference between the raw magnet position measurement and the NN magnet position estimate is less than a threshold, we use Equation 8.1. Otherwise, we use the NN magnet position estimate as the filter value. In other words, if the difference is small we smooth the signal and get the average (with $\alpha = 0.5$), else there would be a spike in reading and we used the NN estimate.

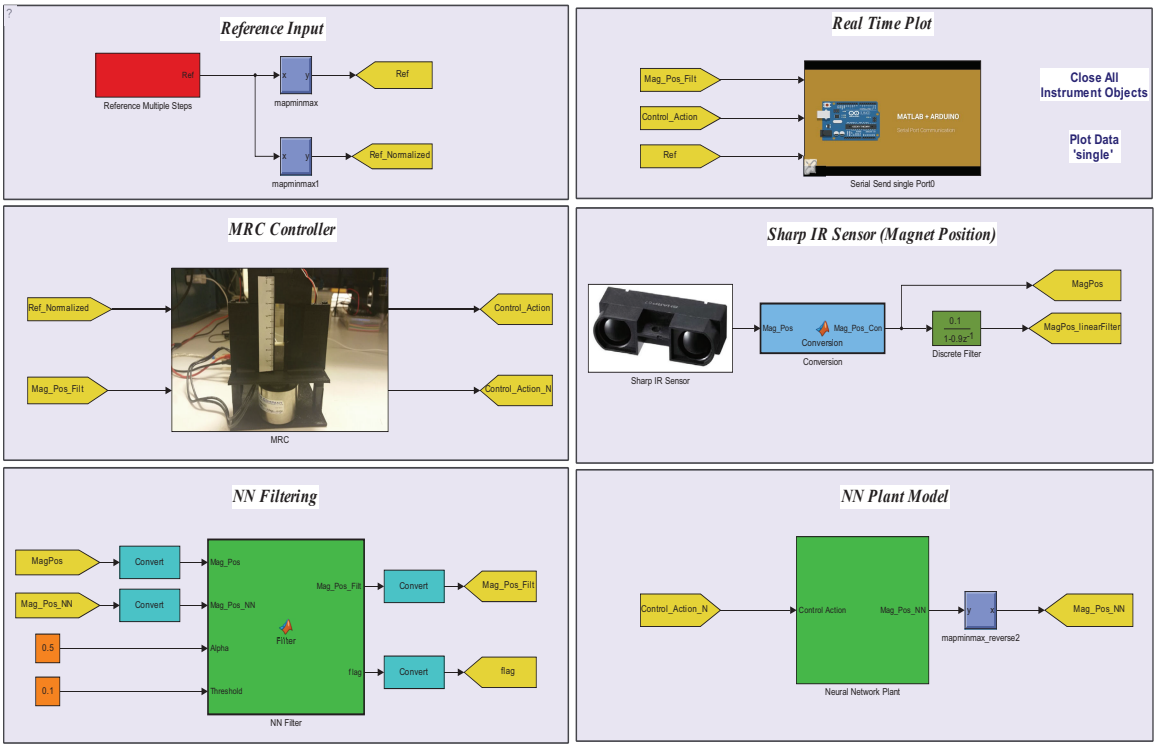


Figure 8.17: Closed loop Simulink block diagram (MRC with NN filter)

$$MagPosFilt = (\alpha * MagPos) + (1 - \alpha) * MagPosNN \tag{8.1}$$

Where *MagPos* is the raw magnet position, *MagPosNN* is the NN magnet position estimate and α is the weighting constant.

We observed that this method of filtering improved the sensor reading and it is very unique since we used the NN plant model. This is one of the advantages we can get from the NN approach.

Figure 8.18 shows the test results of the MRC controller from the experimental setup for 50 seconds. The magnet accurately follows the reference input (the NN filter is used to enhance the results).

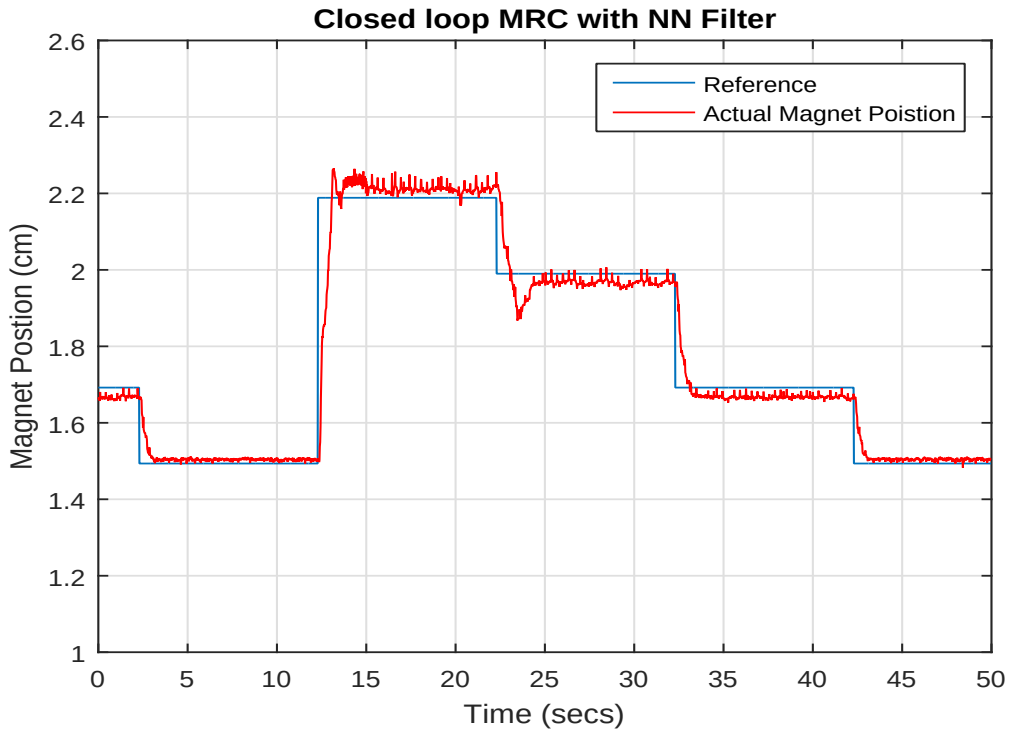


Figure 8.18: Experimental data MRC with NN filter for test reference input

8.4 PID CONTROLLER (CLASSICAL CONTROLLER)

In order to verify the performance of the neuro controller, we need to compare it with other types of controllers. The most popular classical controller is the proportional, integral and derivative (PID) control structure.

The system represented in the Figure 8.19 is the magnetic levitation system controlled by a PID controller. The K_p, K_i and K_d values are PID gains, the set point is the reference signal, the output y is magnet position and u is the control action. The Control action is a weighted summation of the proportional, integral, and derivative actions. A mathematical description of the PID controller is:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \tag{8.2}$$

PID controllers are used in variety of practical industrial processes. They are widely

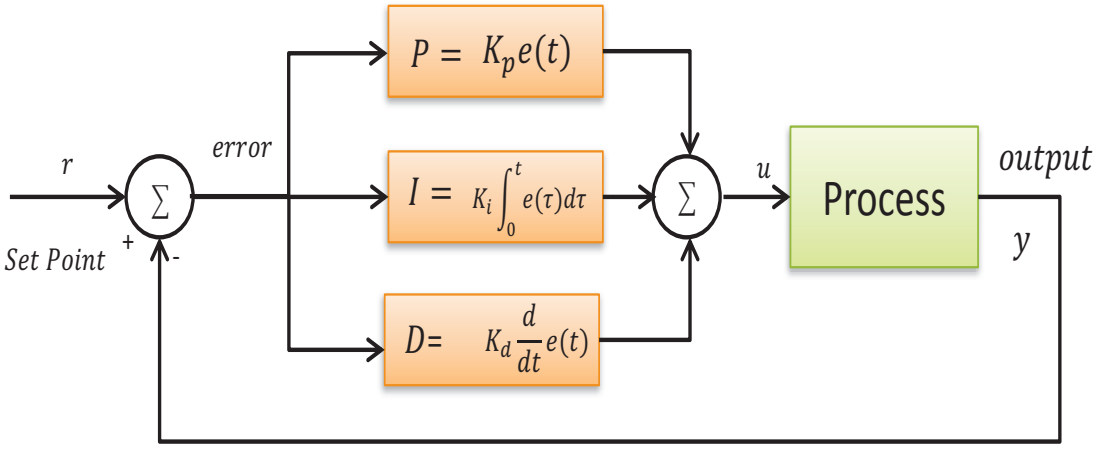


Figure 8.19: PID control structure

accepted in industrial applications due to their ease of implementation. The key to implement the PID controller is to find gain values. There are couple of techniques to decide the gain values that meet the specifications.

All these techniques require a linear mathematical model of the process. In our case we need to model the magnetic levitation system mathematically. We used the NN plant model and linearized it around the nominal point to obtain the linear model. This is another advantage of NN plant modeling. Since we have the linear model, we used the Matlab *PID tuning* toolbox in order to find the gain value for the PID controller. We used this toolbox in order to get a critically damped second order system response, as in the reference model for the MRC system.

Figure 8.20 shows the Simulink block diagram of the PID controller for the magnetic levitation system. It has four sections, three sections remain same as before, but we added the PID controller and used the tuned gains which we got from the toolbox.

This is the one of the best ways of tuning the PID controller, since we are using the NN nonlinear model of the plant and Matlab toolbox tuning. In other words, we are trying to get the best results we can from the PID controller in order to compare it with MRC controller. Figure 8.21 shows the magnet response with the PID controller. We can see that

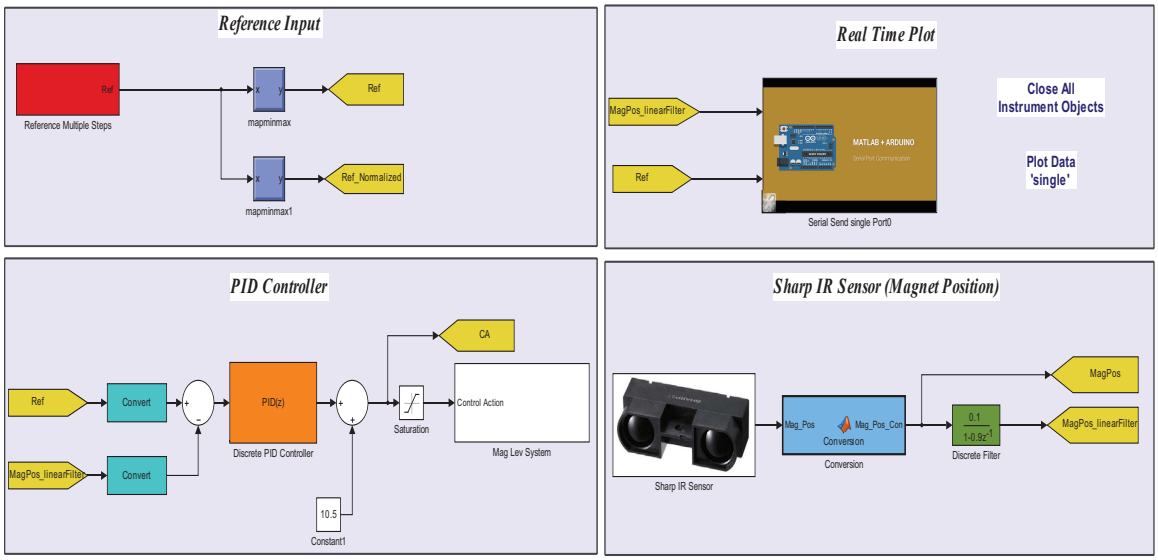


Figure 8.20: PID controller Simulink block diagram

the tracking of the PID controller is not as good as the NN MRC controller, as shown in Figure 8.18

8.5 SUMMARY AND COMPARISON OF CONTROLLERS

In order to compare the PID controller with MRC, we calculated the mean square error between the reference signal and the magnet position. The MSE of the NN MRC is 0.0061, and the MSE of the PID controller is 0.0170. These experimental results show the power and flexibility of the NN approach.

Note that the NN plant model was used in three ways. First, it was used to train the MRC controller. Second, we used it to filter the noisy sensor (most real life applications have noise) in a very effective way. Third, the NN plant model is used to find the linear model. This can be an interesting feature for industrial applications.

As we mentioned before, there are several advantages to the NN approach:

- The NN plant model can be used as a filter to enhance the measurements.
- The NN plant model can be used to find a linearized model (which can be used to

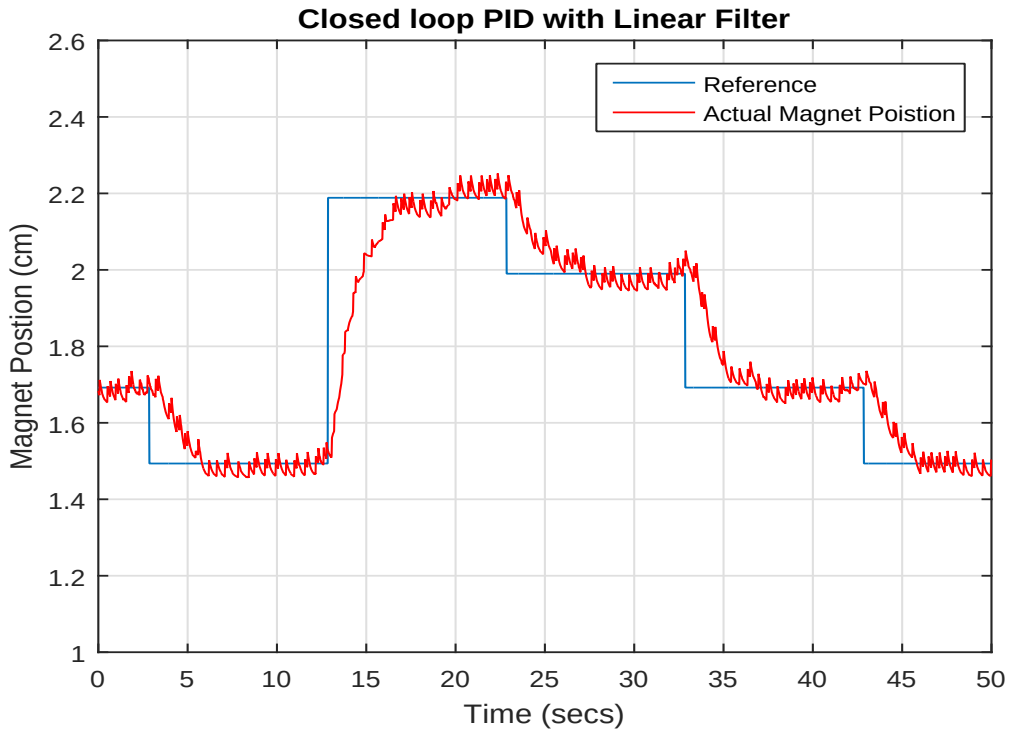


Figure 8.21: Experimental data PID with linear filter for test reference input

tune a PID or other linear controller).

- The NN MRC controller is more flexible than linear controllers.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 SUMMARY

We introduced some basic neural network concepts in Chapter 2, and we categorized them into two general classes: static and dynamic networks. Dynamic networks have tapped delay lines, which give the networks memory. The output of dynamic networks depend on current inputs, outputs, and states of the network. Dynamic networks with feedback are recurrent Neural Networks (RNNs).

Training RNNs is difficult, because of the existence of spurious valleys in the error surface. In Chapter 2, we briefly introduced the spurious valleys and their characteristics. These valleys are spurious, because they are unrelated to the true minimum of the error surface. They only depend on the network inputs. We introduced a first order recurrent network in Chapter 2, in order to show the properties of these valleys. The spurious valleys in the error surface of RNNs make training very difficult, especially using the batch, gradient-based methods.

With the knowledge of the spurious valleys, in Chapter 3 we described a recently introduced procedure for training a general class of RNNs. The standard training procedure uses the Levenberg-Marquardt batch algorithm. However, standard training is not usually successful on RNNs, therefore modifications were required. Briefly, the modified training procedure starts with short prediction horizons and increments the prediction horizon by some intermediate horizon steps. When the procedure detects spurious valleys, it removes the sequences which cause the valley.

One difficulty with the modified training procedure is the selection of the horizon steps.

± 6 Voltage range		± 12 Voltage range		± 24 Voltage range	
Fail	35	Fail	41	Fail	51
Fits	65	Fits	59	Fits	49

Table 9.1: Trained network accuracy on 100 test sequences

If they are chosen too small, training times will become impractical, but large horizon steps can result in steep spurious valleys, which make training very difficult.

In Chapter 4, we introduced a new procedure for horizon step selection. The new procedure follows the same steps as the modified training described in Chapter 3, including the use of the LM algorithm, but in the final step, where the prediction horizon is increased, the following procedure is used to determine the horizon step. Using the weights determined at the completion of the previous training segment, the MSE is computed for prediction horizons from 1 to *maxstep* steps ahead of the prediction horizon. At this point, the algorithm will find all local minima of the MSE with respect to the prediction horizon. It will then select the local minimum with the smallest MSE. This gives us the best horizon step.

We have tested the new procedure using the modified training for one physical system in Chapter 5. In this test, we developed an accurate NARX model for a single-link robot arm system. We selected the system parameters in such way that the system responds quickly and with a strong nonlinearity. To verify the model, we generated new test data which network had not seen. We tested the model for 100 new test sequences.

Table 9.1 shows the number of times that the trained network had oscillatory behaviour, failing to accurately reproduce the correct system response. For example, when the input voltage was limited to the range ± 6 volts, the network had oscillatory responses 35% of time, and 65% of time it precisely fit the true system response.

We believe that the oscillatory behaviour of the network response occurs when the network input is outside the range of the training data set. It would not be possible to guarantee reasonable network performance if the network inputs move outside the range of the data on which the network is trained. It is important to be able to detect when

this extrapolation is occurring. For example, if the RNN is part of a feedback control system, [24], we would want to disable the RNN when extrapolation occurs, and replace it with a conventional controller.

In Chapter 6, we developed a clustering network to detect when inputs move outside the range of the data on which the network is trained. This extrapolation detection is a form of *novelty detection*. This new technique is a type of clustering, in which the inputs from the training set are characterized by a small set of prototype vectors. The minimum distance of a new input to the nearest prototype is used to quantify novelty. For example, if the distance from the new input to the nearest prototype is larger than the maximum distance of that prototype to the cluster of training inputs that are assigned to it, then the new input could be considered novel.

We performed the novelty detection on the single-robot arm system, and in Chapter 6 we extensively tested this new method. We need to obtain smaller quantization and topographic errors. Therefore, either we need to add more cluster centers or improve the SOM training. The SOM clustering method is used to collect more data in Chapter 7.

In this Chapter 7, we trained a recurrent neural network, using the modified training algorithm, to model a simulated magnetic levitation system. The Self Organizing Map (SOM) was used to collect additional data to improve the training (collecting data wisely in the regions where we were extrapolating). The RNN is extrapolating when network inputs fall outside the space spanned by the training data set. The SOM detected extrapolation and guided the collection of additional training data. Then, the NARX network was retrained with the new data combined with the initial training data set. This procedure is known as *novelty sampling*. This is done in phases until no novel conditions are detected after many additional tests. We tested and verified the final trained RNN model after phases of retraining.

Next, we trained the Model Reference Control (MRC) algorithm in conjunction with the plant model. Finally, we tested and verified the MRC model on the simulated magnetic

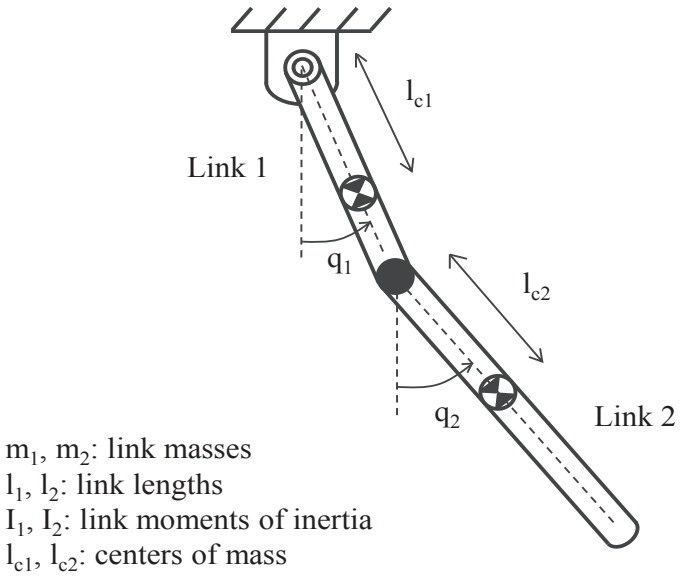


Figure 9.1: Double pendulum [7]

levitation system.

Chapter 8 describes the design and construction of an experimental magnetic levitation system. We verified that the new RNN training procedures and the SOM novelty sampling technique could be successfully used to train an NN MRC system in practice. We also designed and demonstrated the use of the neural network plant model as a filter to reduce measurement error. Finally, we compared the performance of the NN MRC with the industry standard PID controller. The NN MRC demonstrated significantly improved mean square error.

9.2 FUTURE WORK

The experimental magnetic levitation system demonstrated how parameters can change through time. In the future we would like to make our control algorithm adaptive. We may use the SOM to remove or add some sequences, since the system parameters are changing through time.

In the future, we want to test the MRC architecture on other complex dynamic sys-

tems like the double pendulum shown in Figure 9.1. We can build and design the double pendulum using the 3D printer technology.

Also, in the experimental section of this work, we can use micro computers that are more powerful than Arduino, such as the raspberry pi. This will make it possible to implement larger neural networks, which would be required for larger industrial applications.

REFERENCES

[1] M. T. Hagan, H. B. Demuth, and M. H. Beale, *Neural Network Design*. Boston, MA: PWS, 1996.

[2] J. Horn, O. D. Jesus, and M. T. Hagan, "Spurious valleys in the error surface of recurrent networks - analysis and avoidance," *IEEE Trans. Neural Netw.*, vol. 20, no. 4, pp. 686–700, Apr. 2009.

[3] Fritzing, figures of arduino mega. [Online]. Available: http://www.nemsim.com/ece395blimp/fritzing/parts/svg/core/breadboard/

[4] Arduino official website. [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardMega

[5] Sharp gp2y0a51sk0f analog distance sensor 2-15cm. [Online]. Available: https://www.pololu.com/product/2450

[6] 2a motor shield for arduino. [Online]. Available: http://www.dfrobot.com/index.php?route=product/product&product_id=69#.VrfbetsrKUl

[7] M. Phan and M. T. Hagan, "A procedure for training recurrent networks," in *Proc. Int. Joint Conf. Neural Netw.*, Aug. 2013, pp. 1–8.

[8] M. T. Hagan and H. B. Demuth, "Neural networks for control," in *American Control Conference, 1999. Proceedings of the 1999*, vol. 3. IEEE, 1999, pp. 1642–1656.

[9] H. T. Su, T. J. McAvoy, and P. Werbos, "Long-term predictions of chemical processes using recurrent neural networks: A parallel training approach," *Industrial & engineering chemistry research*, vol. 31, no. 5, pp. 1338–1352, 1992.

[10] J. Roman and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,*, vol. 2. IEEE, 1996, pp. 454–460.

[11] M. T. Hagan, O. D. Jess, and R. Schultz, "Chapter 12 training recurrent networks for filtering and control."

[12] I. Kamwa, R. Grondin, V. Sood, C. Gagnon, J. Mereb *et al.*, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," *Instrumentation and Measurement, IEEE Transactions on*, vol. 45, no. 2, pp. 657–664, 1996.

[13] A. F. Atiya and A. G. Parlos, "New results on recurrent network training: Unifying the algorithms and accelerating convergence," *IEEE Trans. Neural Netw.*, vol. 11, no. 3, pp. 697–709, May 2000.

[14] M. Gori, B. Hammer, P. Hitzler, and G. Palm, "Perspectives and challenges for recurrent neural network training," *Logic Journal of the IGPL*, vol. 18, no. 5, pp. 617–619, 2010.

[15] O. D. Jesus, J. Horn, and M. T. Hagan, "Analysis of recurrent network training and suggestions for improvements," in *Proc. Int. Joint Conf. Neural Netw.*, Jul. 2001, pp. 2632–2637.

[16] M. Phan and M. T. Hagan, "Error surface of recurrent networks," *IEEE Trans. Neural Netw. and Learn. Sys.*, vol. 24, no. 11, pp. 1709 – 1721, Oct. 2013.

[17] A. H. Jafari and M. T. Hagan, "Enhanced recurrent network training," in *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015, pp. 1–8.

[18] M. A. Pimentel, D. A. Clifton, L. Clifton, and L. Tarassenko, "A review of novelty detection," *Signal Processing*, vol. 99, pp. 215–249, June 2014.

[19] L. Raff, M. Malshe, M. Hagan, D. Doughan, M. Rockley, and R. Komanduri, "Ab initio potential-energy surfaces for complex, multichannel systems using modified novelty sampling and feedforward neural networks," *The Journal of chemical physics*, vol. 122, no. 8, p. 084104, 2005.

[20] O. D. Jesus and M. T. Hagan, "Backpropagation algorithms for a broad class of dynamic networks," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 14–27, 2007.

[21] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the marquardt algorithm," *Neural Networks, IEEE Transactions on*, vol. 5, no. 6, pp. 989–993, 1994.

[22] P. J. Werbos, "Backpropagation through time: What it is and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, p. 15501560, Oct. 1990.

[23] M. Phan, "Recurrent neural networks: Error surface analysis and improved training," Ph.D. dissertation, Oklahoma State University, July 2014.

[24] M. T. Hagan, H. B. Demuth, and O. De Jesus, "An introduction to the use of neural networks in control systems," *International Journal of Robust and Nonlinear Control*, vol. 5, no. 6, pp. 989–993, Nov. 2002.

[25] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, p. 14641480, Sep. 1990.

[26] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Trans. Neural Netw.*, vol. 1, no. 1, pp. 4–27, Mar. 1990.

[27] Mathwork, setup and configuration. [Online]. Available: http://www.mathworks.com/help/supportpkg/arduino/setup-and-configuration.html

[28] Round electromagnet. [Online]. Available: http://catalog.apwcompany.com/category/electromagnets

[29] Magnet, k & j magnetics. [Online]. Available: http://www.kjmagnetics.com/proddetail.asp?prod=BX04X0

[30] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, p. 270280, 1989.

VITA

Amir Hossein Jafari

Candidate for the Degree of

Doctor of Philosophy

Dissertation: ENHANCED RECURRENT NETWORK TRAINING

Major Field: Electrical and Computer Engineering

Biographical:

Personal Data: Tehran, Iran, September, 1983

Education:
Received the B.S. degree from Iran, 2006, Electrical Engineering
Received the M.S. degree from Sharjah, UAE, 2011, Mechatronics
Completed the requirements for the degree of Doctor of Philosophy with a major in Electrical and Computer Engineering Oklahoma State University in May, 2016.