RECURRENT NEURAL NETWORKS: ERROR SURFACE ANALYSIS AND IMPROVED TRAINING

By

MANH C. PHAN

Bachelor of Science in Electrical Engineering
Hanoi University of Science and Technology
Hanoi, Vietnam
2008

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
July, 2014

RECURRENT NEURAL NETWORKS: ERROR SURFACE ANALYSIS AND IMPROVED TRAINING

Dissertation Approved:

Name: MANH C. PHAN

Date of Degree: JULY, 2014

Title of Study: RECURRENT NEURAL NETWORKS: ERROR SURFACE ANALYSIS AND IMPROVED TRAINING

Major Field: ELECTRICAL ENGINEERING

Abstract: Recurrent neural networks (RNNs) have powerful computational abilities and could be used in a variety of applications; however, training these networks is still a difficult problem. One of the reasons that makes RNN training, especially using batch, gradient-based methods, difficult is the existence of spurious valleys in the error surface. In this work, a mathematical framework was developed to analyze the spurious valleys that appear in most practical RNN architectures, no matter their size. The insights gained from this analysis suggested a new procedure for improving the training process. The procedure uses a batch training method based on a modified version of the Levenberg-Marquardt algorithm. This new procedure mitigates the effects of spurious valleys in the error surface of RNNs. The results on a variety of test problems show that the new procedure is consistently better than existing training algorithms (both batch and stochastic) for training RNNs. Also, a framework for neural network controllers based on the model reference adaptive control (MRAC) architecture was developed. This architecture has been used before, but the difficulties in training RNNs have limited its use. The new training procedures have made MRAC more attractive. The updated MRAC framework is very flexible, and incorporates disturbance rejection, regulation and tracking. The simulation and testing results on several real systems show that this type of neural control is well-suited for highly nonlinear plants.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION 1

2 DIFFICULTIES IN TRAINING RECURRENT NETWORKS: A SURVEY 3

2.1 Bifurcation in recurrent networks 3

2.1.1 One-neuron recurrent network 5

2.1.2 Two-neuron recurrent network 10

2.1.3 Problems with bifurcations of network dynamics 11

2.2 The problem of long-term dependencies 12

2.2.1 Information latching 14

2.2.2 Gradient vanishing 16

2.2.3 Some suggested solutions 20

3 ERROR SURFACE OF FIRST-ORDER RECURRENT NETWORK 22

3.1 Preliminary material 22

3.2 Linear network 26

3.2.1 Root valley 27

3.2.2 Architecture valley 29

3.3 Nonlinear network 30

3.3.1 Root valley 30

3.3.2 Architecture valley 32

4 ERROR SURFACE OF GENERAL RECURRENT NETWORKS 35

4.1 Second order linear recurrent network 35

4.1.1 Description of network 35

4.1.2 Root valleys 38

4.1.3 Intrinsic architecture valleys 40

4.1.4 Sample architecture valleys 47

4.2 Second order nonlinear recurrent network 51

4.2.1 Description of network 51

4.2.2 Root valleys 52

4.2.3 Architecture valleys 56

4.3 A general class of RNN 58

4.3.1 Layered digital dynamic networks (LDDNs) 58

4.3.2 Root valleys 59

4.3.3 Architecture valleys 60

4.3.4 Valleys for nonlinear RNNs 60

5 PROCEDURE FOR TRAINING RECURRENT NETWORKS 62

5.1 Properties of valleys of recurrent networks 62

5.2 Levenberg-Marquardt algorithm: A short description 64

5.3 Techniques for avoiding valleys in the error surface 65

5.4 Training procedure 67

5.4.1 Useful concepts 67

5.4.2 Procedure 68

6 TRAINING RECURRENT NETWORKS FOR MODELING AND CONTROL OF PHYSICAL SYSTEMS 70

6.1 System identification 70

6.2 Model reference adaptive control 70

6.3 Simulation results 72
6.3.1 Magnetic levitation 72
6.3.2 Double pendulum 77

7 A COMPARISON BETWEEN STOCHASTIC TRAINING AND BATCH TRAINING 84
7.1 Stochastic training and batch training: An overview 84
7.2 NN training with extended Kalman filter 85
7.3 Our comparison: Simulation results 86
7.3.1 Simple fitting data set 87
7.3.2 Narendra and Parthasarathy's static data set 88
7.3.3 House value data set 89
7.3.4 Engine behavior data set 90
7.3.5 Magnetic levitation data set 90
7.4 Conclusion 91

8 A NEURAL CONTROLLER WITH DISTURBANCE REJECTION 93
8.1 Classical disturbance rejection method 93
8.2 MRAC with disturbance rejection 94
8.3 Experimental results 94
8.3.1 Plant model training 95
8.3.2 Controller training 102
8.4 Summary 106

9 CONCLUSIONS AND FUTURE WORK 108

REFERENCES 111

A Proof that (4.16) **approximates** (4.17) **117**

B Proof that $\frac{d^2F_t}{dw_1^2} > 0$ if $g(Q-1) = 0$ **118**

C Proof that $\frac{d^2F_t}{dw_1^2} < 0$ if $\frac{dg(Q-1)}{dw_1} = 0$ **119**

LIST OF TABLES

Table Page

6.1 Simulation parameters for the magnetic levitation 73

6.2 Double pendulum parameters 78

6.3 Removing sequences for 65-step-ahead predictions 81

6.4 Removing sequences for different prediction horizons 82

LIST OF FIGURES

Figure Page

2.1 Eigenvalues of the Jacobian cross the unit circle and create bifurcations [1]. 4

2.2 One neuron recurrent network 5

2.3 The graph of $\tanh(wx + u)$ for various values of u (w is fixed and $w > 1$). Depending on the shift, the graph has one, two, or three intersections with the identity line. 6

2.4 A description for period-doubling bifurcation ($f(x) = \tanh(wx + u)$) . 7

2.5 Bifurcation diagram for one-neuron recurrent network 8

2.6 The outputs for different regions (two initial values: $x(0) = 0.3$ (blue) and $x(0) = -0.7$ (red)). 9

2.7 Two-neuron recurrent network 10

2.8 Bifurcation diagram corresponding to zero equilibrium point 11

2.9 An example of Neimark-Sacker bifurcation 12

2.10 Change of network output at bifurcation points 13

2.11 Latching problem [2] 13

2.12 Graphical interpretation of information latching. The curve is $x = w\tanh(a)$ and the red line is $x = a - u$. 15

2.13 Ball of uncertainty grows exponentially outside Γ [2] 18

2.14 Ball of uncertainty is bounded inside Γ [2] 19

3.1 One-layer linear network 25

3.2 Error surface and valleys for one-layer linear network 26

3.3 Error surface and valleys for one-layer nonlinear network 30

3.4 Third type valleys 33

3.5 Fourth type valleys 34

4.1 Two-layer linear network. 36

4.2 Stable region of the Fig. 4.1 network. 36

4.3 Error surface of the Fig. 4.1 network. 37

4.4 Valleys in the error surface. 38

4.5 Average surface and valleys. 41

4.6 Intrinsic architecture valleys and their equations for $Q = 15$ (top) and $Q = 14$ (bottom) 44

4.7 Impulse responses for different intrinsic architecture valleys. Last impulse response equals zero. 46

4.8 Intrinsic architecture valleys for different frequencies Θ. 47

4.9 Sample architecture valleys and their approximations. 48

4.10 Sample architecture valleys compared to intrinsic architecture valleys. 49

4.11 Additional sample valleys. 50

4.12 Error surface and valleys for a two-layer nonlinear RNN. 51

4.13 Movement of architecture valleys from the linear case to the nonlinear case (7 point sequence). 55

4.14 Outputs at basic architecture valleys. 56

4.15 Outputs at new architecture valleys. 57

5.1 Error profile along the gradient direction 63

6.1 Plant identification [3] 71

6.2 NARX recurrent network [4] 71

6.3 Model reference adaptive control structure [3] 72

6.4 Details of model reference adaptive control structure [3] 72

6.5 Magnetic levitation system. 73
6.6 Training data for magnetic levitation identification. 74
6.7 Magnetic levitation identification: 1997-steps-ahead prediction. 75
6.8 Model reference control training. 76
6.9 Model reference control training without using the new procedure. . . 76
6.10 Model reference control testing. 77
6.11 Double pendulum system. 77
6.12 Change of error profile as a sequence is removed. 78
6.13 Gradient of individual sequences. 80
6.14 Training performance for the modified LM algorithm. 81
6.15 Training results for the double pendulum (Actual outputs - thick, black and network outputs - thin, gray). 83
6.16 Testing results for the double pendulum (Actual outputs - thick, black and network outputs - thin, gray). 83

7.1 Simple fitting data 86
7.2 Comparison of convergence on simple fitting data 87
7.3 Comparison of convergence on N-P data 88
7.4 Comparison of convergence on house value data 89
7.5 Comparison of convergence on engine behavior data 90
7.6 Comparison of convergence on magnetic levitation data 91

8.1 A system with a disturbance 94
8.2 MRAC with disturbance rejection 95
8.3 Track-type tractor 96
8.4 Open loop data 97
8.5 Open loop data (magnified) 98
8.6 Closed loop data 99

8.7 Cloosed loop data (magnified) 99
8.8 NARX with an integrator 100
8.9 Plant model fitting 101
8.10 Disturbance 102
8.11 MRAC network with two saturation blocks, and a disturbance input . 103
8.12 Disturbance rejection for training data 104
8.13 Disturbance rejection for testing data 105

CHAPTER 1

INTRODUCTION

Artificial Neural Networks can be categorized as feedforward neural networks (FNNs) or recurrent neural networks (RNNs). RNNs have at least one feedback loop, while FNNs do not. In FNNs, the network output depends only on the current network inputs (and possibly a finite number of past inputs), while in RNNs, the output depends not only on the current inputs but also on past inputs, outputs and/or states of the networks.

RNNs have been used successfully in many applications. These include the identification and control of dynamic systems [5], prediction in financial markets [6], channel equalization in communication systems [7], phase detection in power systems [8], sorting [9], fault detection [10], speech recognition [11], handwriting recognition [12], learning of grammars in natural languages [13], and even the prediction of protein structure in genetics [14]. However, even though these networks have been widely used, the difficulty of recurrent network training has limited their widespread application.

Our interest is to investigate the reasons that make training RNNs difficult. Two reasons that are usually mentioned in the literature are dynamic bifurcation [15] and long-term dependencies [2, 16]. Another reason for the difficulties in RNN training is the existence of spurious valleys in the error surface, where training algorithms can be easily trapped. This issue was first introduced in [17], and later a mathematical analysis of the first-order case was provided in [18]. These valleys are not related to the true minimum of the surface, or to the problem the RNN is trying to solve.

They depend on the input sequence in the training data, the initial conditions and the network architectures. Any batch search algorithm is very likely to be trapped in these spurious valleys.

In this dissertation, first, we extend the results in [18] to general RNNs [19]. We use some basic concepts of linear system theory to develop a framework that can be used to analyze spurious valleys that appear in most practical recurrent network architectures, no matter their size. We derive approximate equations for the valleys for a very general class of RNN and have confirmed the equations experimentally for second-order networks. Second, using the knowledge of the spurious valleys, we propose a new training procedure for RNNs that could overcome the problem of spurious valleys [20]. Third, we implement the stochastic and batch training algorithms for a general class of neural networks and compare these two training schemes on a variety of problems. The advantages of the batch training demonstrate the importance of understanding the spurious valleys, so that batch algorithms can be modified to avoid the spurious valleys. Finally, we propose a new neural control scheme for disturbance rejection and test our training procedure and controller design for some practical data. These are the main contributions of this dissertation.

This dissertation is organized as follows. In Chapter 2, we review difficulties in training RNNs. Next, in Chapter 3, we review the error surface of the first-order recurrent network, and in Chapter 4, we extend the previous results to general RNNs. Chapter 5 proposes a procedure for training RNNs and Chapter 6 presents experimental results for modeling and control of two physical systems. In chapter 7, we compare stochastic training and batch training schemes. In chapter 8, we design a neural controller for disturbance rejection. Finally, Chapter 9 summarizes the dissertation and describes future work.

CHAPTER 2

DIFFICULTIES IN TRAINING RECURRENT NETWORKS: A SURVEY

In this chapter, we will review some of the problems mentioned in the literature that cause difficulties for recurrent network training. The first problem is the bifurcation of network dynamics. We will see that RNNs have very complex dynamic behaviors. The parameter space is divided into many regions with different dynamics. At bifurcation points, the network output could change discontinuously with the change of parameters, and therefore the convergence of gradient-based algorithms is not guaranteed. The second problem is long-term dependencies. There are practical applications involving long-term dependencies that RNNs have to carry out. However, training RNNs to deal with those tasks suffers from the problem of vanishing gradients. This is claimed to be the essential reason why learning long-term dependencies is difficult.

2.1 Bifurcation in recurrent networks

Bifurcation is a change in the dynamic behaviors of the network (equilibrium points, periodic orbits, stability properties,...) as a parameter is varied [21]. The values at which the changes occur are called bifurcation points. This section will discuss bifurcations in recurrent networks and will show how they affect the training process.

Consider a class of recurrent networks in the form of

$$\mathbf{x}(k+1) = \mathbf{f}(\mathbf{x}(k),\, \boldsymbol{\mu})$$

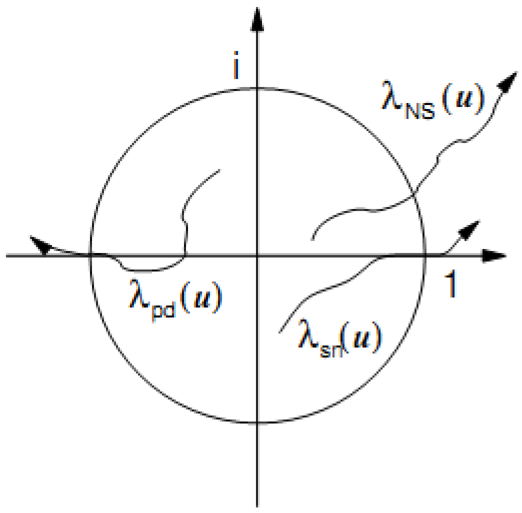


Figure 2.1: Eigenvalues of the Jacobian cross the unit circle and create bifurcations [1].

where $\mathbf{x}$ is a state vector and $\boldsymbol{\mu}$ represents all network parameters. Suppose that $\mathbf{x}^*$ is an equilibrium point, $\boldsymbol{\mu}^*$ is corresponding bifurcation point and $\mathbf{J}(\mathbf{x}^*, \boldsymbol{\mu}^*)$ is the Jacobian of $\mathbf{f}$ at $(\mathbf{x}^*, \boldsymbol{\mu}^*)$. Then a bifurcation occurs if an eigenvalue λ of $\mathbf{J}(\mathbf{x}^*, \boldsymbol{\mu}^*)$ during parameter variation leaves the unit circle in the complex plane (see Fig. 2.1). There are three types of bifurcation corresponding to the locations where λ crosses the unit circle [1]:

- Saddle-node (tangential, turning point, fold, blue-sky) [1] bifurcation: $\lambda = 1$
- Period doubling (flip) bifurcation: $\lambda = -1$
- Neimark-Sacker (Holf) bifurcation: $\lambda = e^{jw}$ where $w \neq k\pi, k \in \mathbb{Z}$

We will use this definition to investigate the bifurcation diagrams for some simple recurrent networks. The simplest recurrent network is the single layer network with only one neuron. (We will come back to this network in Chapter 3 as we analyze its error surface.)

[1]There are alternate terminologies in the bifurcation theory.

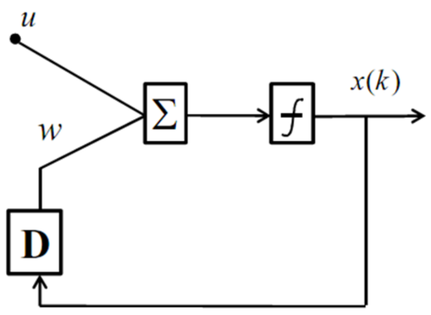


Figure 2.2: One neuron recurrent network

2.1.1 One-neuron recurrent network

The one-neuron recurrent network is shown in Fig. 2.2. The network response is as follows:

$$x(k+1) = \tanh(wx(k) + u)$$

in which w is a feedback weight and u is a time-invariant input. In this example, both w and u are considered bifurcation parameters.

First, we need to find the equilibrium points for this system. They satisfy this equation

$$x^* = \tanh(wx^* + u) \tag{2.1}$$

Thus, they are the intersections of the graph $y = \tanh(wx + u)$ and the identity line $y = x$. As shown in Fig. 2.3, we can have one, two, or three equilibrium points, depending on the values of u and w.

Now we calculate the Jacobian at the equilibrium points.

$$\begin{aligned} \mathbf{J}(x^*) &= w \tanh'(wx^* + u) \\ &= w \left[1 - \tanh^2(wx^* + u)\right] \end{aligned} \tag{2.2}$$

If we combine (2.2) and (2.1), we have

$$\mathbf{J}(x^*) = w(1 - {x^*}^2)$$

For this scalar case, the eigenvalue of $\mathbf{J}(x^*)$ is $\mathbf{J}(x^*)$ itself, and it is real. So we have

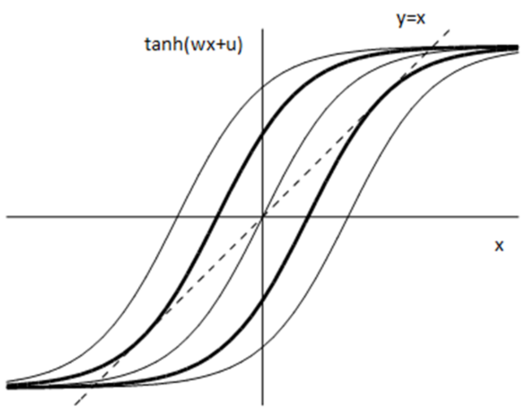


Figure 2.3: The graph of $\tanh(wx+u)$ for various values of u (w is fixed and $w>1$). Depending on the shift, the graph has one, two, or three intersections with the identity line.

two possible types of bifurcation: If

$$w(1-{x^*}^2) = 1 \tag{2.3}$$

we have saddle-node bifurcation. If

$$w(1-{x^*}^2) = -1 \tag{2.4}$$

we have period doubling bifurcation. We will look at these two in detail.

- **Saddle-node bifurcation:**

Note that in Fig. 2.3, the saddle-node bifurcation occurs as the line $y=x$ becomes tangent to $y=\tanh(wx+u)$ (the bold curves). Therefore, the slope of $y=\tanh(wx+u)$ at the tangent point equals 1, which is consistent with (2.3). Since $0 \le {x^*}^2 \le 1$, (2.3) has real root x^* if and only if $w \ge 1$. Then from (2.3) we have

$$x^* = \pm\sqrt{\frac{w-1}{w}} \tag{2.5}$$

Also, we can solve for u from (2.1):

$$u = \tanh^{-1}(x^*) - wx^* = \frac{1}{2}\ln\frac{1+x^*}{1-x^*} - wx^* \tag{2.6}$$

Substituting (2.5) into (2.6), we have

$$u = \ln(\sqrt{w} \pm \sqrt{w-1}) \mp \sqrt{w(w-1)}$$

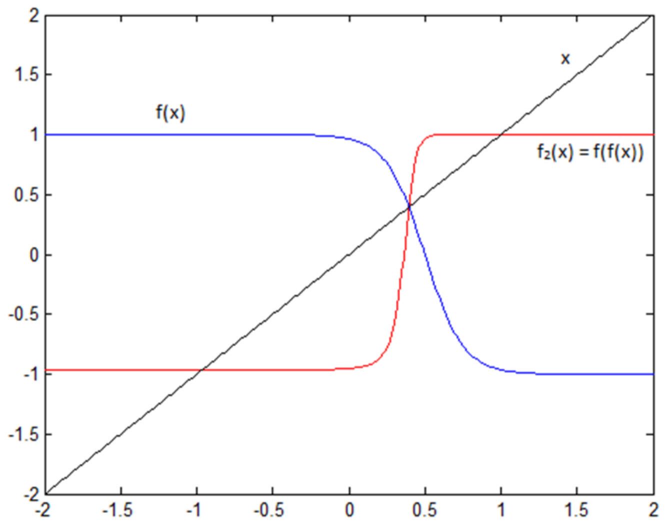


Figure 2.4: A description for period-doubling bifurcation ($f(x) = \tanh(wx + u)$)

Since

$$\ln(\sqrt{w} - \sqrt{w-1}) + \sqrt{w(w-1)} = -\left[\ln(\sqrt{w} + \sqrt{w-1}) - \sqrt{w(w-1)}\right] > 0$$

we have

$$|u| = \ln(\sqrt{w} - \sqrt{w-1}) + \sqrt{w(w-1)} \tag{2.7}$$

The set of points satisfying (2.7) is the set of saddle-node bifurcation points.

- **Period-doubling bifurcation:**

Note that (2.4) has real root x^* if and only if $w \le -1$. We can see from Fig. 2.4 that $y = \tanh(wx + u)$ has one intersection with the line $y = x$ and the period-doubling bifurcation occurs if the slope of $y = \tanh(wx+u)$ at the intersection equals -1. Also, from (2.4), we have

$$x^* = \pm\sqrt{\frac{w+1}{w}} \tag{2.8}$$

Substituting (2.8) into (2.6) (note that (2.6) is true for all equilibrium points), we have

$$u = \ln(\sqrt{-w} \pm \sqrt{-w-1}) \pm \sqrt{w(w+1)}$$

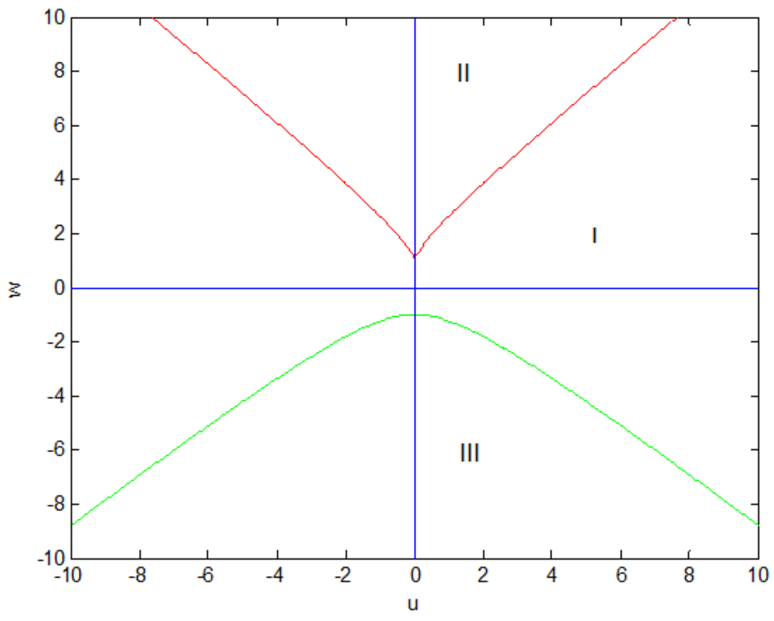


Figure 2.5: Bifurcation diagram for one-neuron recurrent network

or

$$|u| = \ln(\sqrt{-w} + \sqrt{-w-1}) + \sqrt{w(w+1)} \tag{2.9}$$

The set of points satisfying (2.7) is the set of saddle-node bifurcation points. By plotting (2.7) and (2.9) in the uw plane, we have the bifurcation diagram for the one-neuron recurrent network, as shown in Fig. 2.5. A similar diagram could be found in [22].

In region II, the system has two stable nodes and one unstable node (for this one-dimensional example, this unstable node could also be called a saddle). As the point (u, w) passes through the red lines (the bifurcation points), one stable node merges with the unstable node to become a saddle point. In region I, those two nodes disappear, and the system has just one stable node (the third one of those three). That is why this type of bifurcation is call a saddle-node bifurcation.

As the point (u, w) crosses the green line and enters region III, the stable node becomes unstable. In addition, a limit cycle with a period of two appears. The output oscillates with a period of two. That is why this type of bifurcation is called a

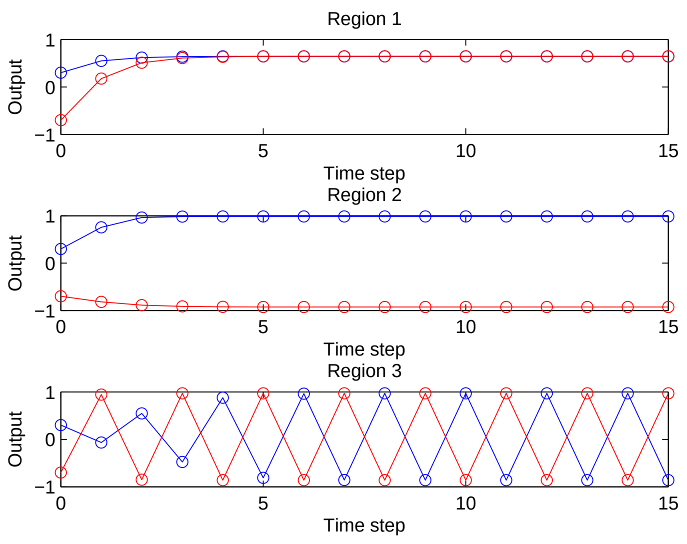


Figure 2.6: The outputs for different regions (two initial values: $x(0) = 0.3$ (blue) and $x(0) = -0.7$ (red)).

period-doubling bifurcation. This can be also seen in Fig. 2.4. The identity line $y = x$ has one intersection with $y = f(x)$, which is the unstable node, and has two other intersections with $y = f(f(x))$. The output oscillates between these two intersections. The outputs for the three regions are shown in Fig. 2.6.

We have just seen that for this simple one-neuron recurrent network, the parameter space is divided into three regions with different dynamic behaviors of the network. As the parameters cross the bifurcation boundaries (representing very small changes in the parameters), the output changes dramatically. This causes some critical problems for network training, as explained in Section 2.1.3.

It is expected that as the number of neurons in the network increases, the dynamics of the network and the bifurcation manifold become more complex. We will investigate the bifurcation for a two-neuron network in the next section.

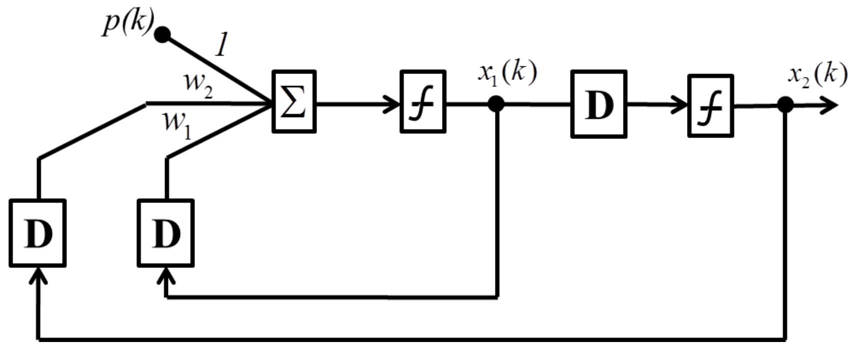


Figure 2.7: Two-neuron recurrent network

2.1.2 Two-neuron recurrent network

Consider a two-neuron recurrent network as shown in Fig. 2.7. We will come back to this network in Chapter 4 to analyze its error surface. The state equations for this network (without the input $p(t)$) are as follows:

$$\begin{aligned} x_1(k+1) &= \tanh\left[w_1 x_1(k) + w_2 x_2(k)\right] \\ x_2(k+1) &= \tanh\left[x_1(k)\right] \end{aligned}$$

For simplicity, we will not construct a full bifurcation diagram for this network as we did for the one-neuron network. We will just consider the equilibrium point $(0,0)$. The Jacobian at this equilibrium point is

$$\mathbf{J}_{|(0,0)} = \begin{bmatrix} w_1 & w_2 \\ 1 & 0 \end{bmatrix}$$

The eigenvalues of the Jacobian matrix are the roots of the characteristic equation (2.10):

$$\lambda^2 - w_1\lambda - w_2 = 0 \tag{2.10}$$

If one of the roots of (2.10) crosses the unit circle, a bifurcation occurs. The bifurcation diagram for the zero equilibrium point is shown in Fig. 2.8. In this case, as the point (w_1, w_2) crosses the triangle, the stability of the zero equilibrium point changes. In particular, if (w_1, w_2) crosses the line $w_2 = -1$, the complex root of (2.10) will cross

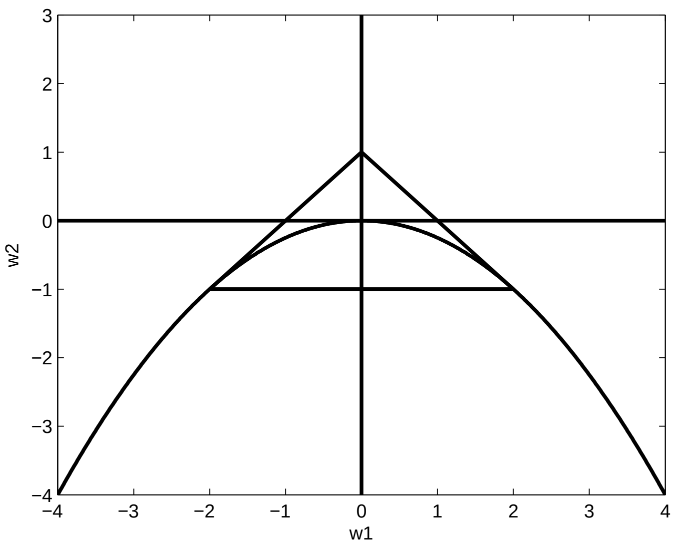


Figure 2.8: Bifurcation diagram corresponding to zero equilibrium point

the unit circle, and Neimark-Sacker bifurcation will occur. Fig. 2.9 shows the change of dynamic behaviors of the system as w_2 changes from -0.95 to -1.05 ($w_1 = 1$). We can see that the stability of the equilibrium point $(0, 0)$ changes, and a new limit cycle appears around it.

2.1.3 Problems with bifurcations of network dynamics

As the parameters cross the bifurcation boundaries, the network output could change dramatically. Let's consider a saddle-node bifurcation point, for example $w = 5$, $u = -3.0285$ (see Fig. 2.5). The network outputs for $w = 5$ and $w = 4.99$ are shown in Fig. 2.10. We can see that they have different convergence properties. Because of this, the gradient near the bifurcation points could be very large. This causes a very long jump in the parameter space. It is possible that the network undergoes almost random jumps in the parameter space until it falls in a favorite basin where the gradient learning works. This provides one reason why gradient learning in recurrent networks is difficult [15].

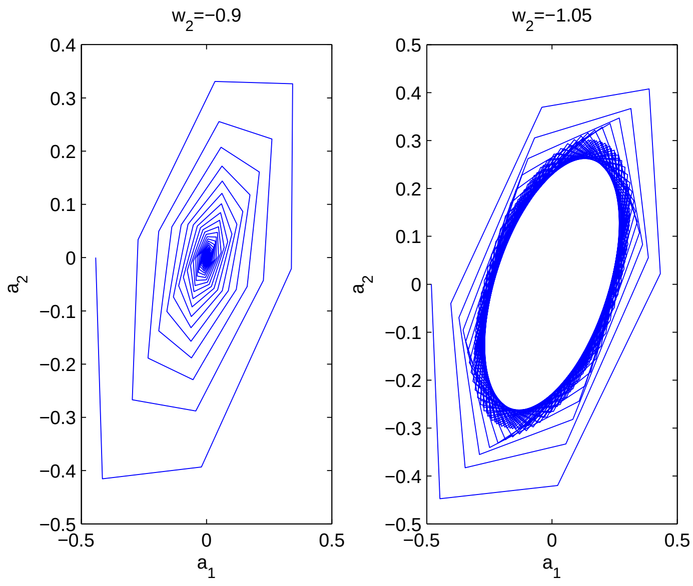


Figure 2.9: An example of Neimark-Sacker bifurcation

2.2 The problem of long-term dependencies

Recurrent networks can in principle use their feedback connections to store past inputs to produce the current desired output. This property is used in sequence recognition, time series prediction, etc. Many practical applications involve temporal dependencies spanning many time steps between relevant inputs and desired outputs. In this case, however, training networks using gradient methods becomes difficult since the gradient vanishes as it gets propagated back. This problem of so-called long-term dependencies was introduced in [2]. This section will summarize some of the main points in this paper.

A task displays long-term dependencies if the desired output at time t depends on inputs presented at times far in the past. Let's consider a very simple example of long-term dependencies described in [2]. (This is usually called the latching problem.) Suppose that we need to classify two different sets of sequences of length T. For each sequence u_1, $u_2, \ldots, u_T$ the class depends only on the first L values of the sequence. The values u_{L+1}, $u_{L+2}, \ldots, u_T$ are irrelevant for determining the class of the sequences. For example, they could be random Gaussian numbers (see Fig. 2.11). Assume that

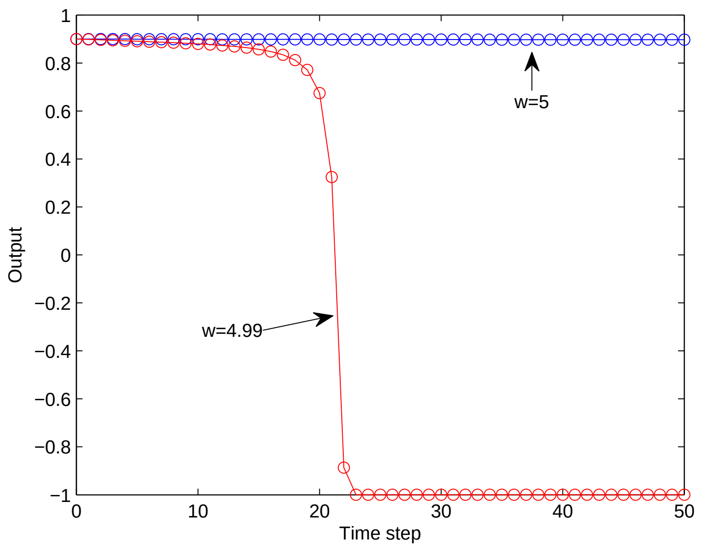


Figure 2.10: Change of network output at bifurcation points

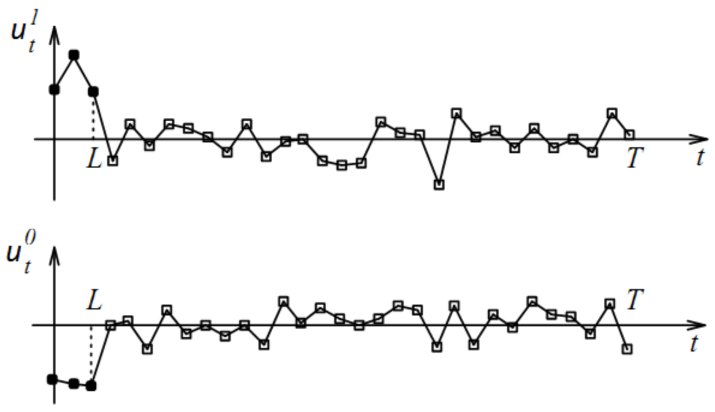


Figure 2.11: Latching problem [2]

L is fixed and the sequence length $T \gg L$. The system should provide an answer at the end of each sequence. The problem can be solved if the network is able to store information about the first L values of the input for a long period of time. Moreover, it has to latch the information robustly, i.e., in such a way that it can not be erased by events that are irrelevant to the classification criterion (such as the values u_{L+1}, $u_{L+2}, \ldots, u_T$). This concept is referred to as "information latching" and is presented below.

2.2.1 Information latching

Consider the one-neuron recurrent network in Fig. 2.2. We investigated the bifurcation of the dynamics of this network in Section 2.1. The definition of information latching is given in [23, 24]:

Definition 2.1 *A given dynamic hidden neuron latches information at t_0, represented by its activation (net input) $a(t_0)$, if $sign(a(t)) = sign(a(t_0))$ $\forall t > t_0$.*

The following theorem gives conditions for information latching to occur [24]:

Theorem 2.1 *Consider the one-neuron recurrent network in Fig. 2.2, in which the net input $a(t)$ is as follows:*

$$a(t) = w \tanh(a(t-1)) + u(t)$$

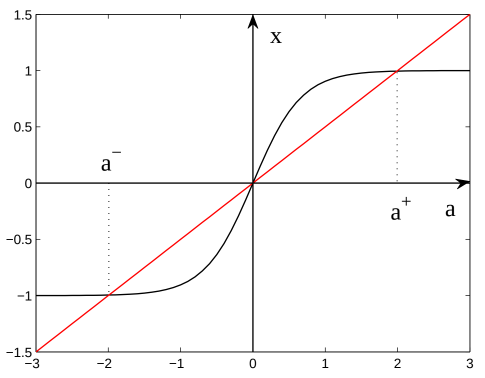
- *If $u(t) = 0$, $\forall t > t_0$, then information latching occurs provided that $w > 1$.*
- *If $w > 1$ then the latching condition also holds if $|u(t)| < b$, $\forall t > t_0$, in which*

$$b = \ln(\sqrt{w} - \sqrt{w-1}) + \sqrt{w(w-1)} \tag{2.11}$$

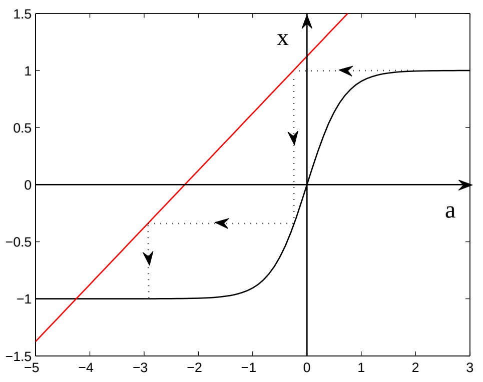
- *If $w > 1$ and $|u(t)| \geq b$, $\forall t > t_0$, the state transition occurs in a finite number of steps. More specifically,*
  - *if $u(t) > b$, $\forall t > t_0$, the state transits from low to high*
  - *if $u(t) < -b$, $\forall t > t_0$, the state transits from high to low*

Proof. See [24] for details of the proof. The following explanation is based on the bifurcation analysis in Section 2.1. It gives an intuitive way to understand information latching. ■

If the input $u = 0$ and $w > 1$, then the point (u, w) is in region II of Fig. 2.5. The system has three equilibrium points, say a^-, 0 and a^+, in which $a^- < 0$ and



(a) $u = 0$



(b) $|u(t)| \geq b$

Figure 2.12: Graphical interpretation of information latching. The curve is $x = w\tanh(a)$ and the red line is $x = a - u$.

$a^+ > 0$ are stable and 0 is unstable (see Fig. 2.12(a)). Starting from any point in the neighborhood of zero, the state trajectory goes to one of the two stable points a^- or a^+ according to the initial sign.

If $w > 1$ and $|u(t)| < b$, $\forall t > t_0$, then the point (u, w) is still in region II of Fig. 2.5. (Note that b is given in (2.11) and illustrated by the red curve in Fig. 2.5.) The system still has three equilibrium points, two of them are stable and the third one is unstable, and information latching occurs like the case $u = 0$.

For the third part of the theorem, assume that the information is latched in the high state. When input $u(t) < -b$ is applied, the red line has only one intersection with the curve (see Fig. 2.12(b)). Therefore, $a(t)$'s evolution follows the attractive trajectory towards the unique equilibrium point (see the zigzag dotted line in Fig. 2.12(b)). The $x(t)$ switches to the low state. A similar pattern occurs for the low to high transition.

This theorem indicates the conditions under which information latching occurs. Information latching is accomplished by keeping a small input for a long enough time. Small noisy inputs (smaller than b in absolute value) can not change the sign of the activation of the neuron even if applied for an arbitrarily long time. Thus the recurrent neuron of Fig. 2.2 can robustly latch one bit of information, represented by the sign of its activation. Larger w corresponds to larger b (see Fig. 2.5, the red curve), so the latching is more robust against the input noise.

2.2.2 Gradient vanishing

Now we will show that why learning long-term dependencies with gradient descent is difficult. The authors in [2] claim that if a recurrent network can robustly latch information, then the problem of vanishing gradients occurs, which makes learning difficult.

Consider a system with additive inputs:

$$\mathbf{a}_t = \mathbf{M}(\mathbf{a}_{t-1}) + \mathbf{u}_t \tag{2.12}$$

(They explain in [2] that a dynamic system with non-additive inputs can be transformed into one with additive inputs by introducing additional states and corresponding inputs. Thus they can use the model in (2.12) without loss of generality).

We will go through some definitions introduced in [2]:

Definition 2.2 *A set of points* $\mathbb{E}$ *is said to be invariant under a map* $\mathbf{M}$ *if* $\mathbb{E} = \mathbf{M}(\mathbb{E})$.

Definition 2.3 *A hyperbolic attractor* $\mathbb{X}$ *is a set of points invariant under the differentiable map* $\mathbf{M}$*, such that* $\forall \mathbf{a} \in \mathbb{X}$*, all eigenvalues of* $\mathbf{M}'(\mathbf{a})$ (*Jacobian of* $\mathbf{M}$ *at* $\mathbf{a}$) *are less than 1 in absolute value.*

Definition 2.4 *The basin of attraction of an attractor* $\mathbb{X}$ *is the set* $\boldsymbol{\beta}(\mathbb{X})$ *of points* $\mathbf{a}$ *converging to* $\mathbb{X}$ *under the map* $\mathbf{M}$*, i.e.,*

$$\boldsymbol{\beta}(\mathbb{X}) = \left\{\mathbf{a} : \forall \epsilon,\, \exists l \; s.t. \; \left\|\mathbf{M}^t(\mathbf{a}) - \mathbf{x}_t\right\| < \epsilon \, \forall t > l, \textit{for some } \mathbf{x}_t \in \mathbb{X}\right\}$$

Definition 2.5 *The reduced attracting set $\mathbf{\Gamma}(\mathbb{X})$ of a hyperbolic attractor $\mathbb{X}$ is the set of points $\mathbf{y}$ in the basin of attraction of $\mathbb{X}$ such that $\left\|\mathbf{M}^{'}(\mathbf{y})\right\| < 1$, where $\|.\|$ denotes matrix norm*[2].

By definition, for a hyperbolic attractor $\mathbb{X}, \mathbb{X} \subset \mathbf{\Gamma}(\mathbb{X}) \subset \boldsymbol{\beta}(\mathbb{X})$.

Definition 2.6 *A system is robustly latched at distance d at time t_0 to a hyperbolic attractor $\mathbb{X}$, if there exists a sequence $b_t > 0$ such that if $\|\mathbf{u}_t\| < b_t \, \forall t > t_0$, then $\mathbf{a}_t \in \mathbb{B}_d(\mathbb{X}) \, \forall t > T$ for some $T > t_0$, where $\mathbb{B}_d(\mathbb{X}) = \{\mathbf{y} | \mathbf{y} \in \mathbb{B}_d(\mathbf{x}), \mathbf{x} \in \mathbb{X}\}$ ($\mathbb{B}_d(\mathbf{x})$ is a ball of radius d around $\mathbf{x}$).*

Definition 2.7 *A map $\mathbf{M}$ is contracting on a set $\mathbb{D}$ if $\exists \alpha \in [0, 1)$ s.t. $\|\mathbf{M}(\mathbf{x}) - \mathbf{M}(\mathbf{y})\| \leq \alpha \|\mathbf{x} - \mathbf{y}\| \; \forall \mathbf{x}, \mathbf{y} \in \mathbb{D}$.*

The following theorems will explain why it is more robust to store information by keeping $\mathbf{a}_t$ in the reduced attracting set $\mathbf{\Gamma}(\mathbb{X})$ (see [2] for details of proof):

Theorem 2.2 *Assume $\mathbf{x}$ is a point of $\mathbb{R}^n$ such that there exists an open ball $\mathbb{U}(\mathbf{x})$ centered on $\mathbf{x}$ for which $\left\|\mathbf{M}^{'}(\mathbf{z})\right\| > 1$ for all $\mathbf{z} \in \mathbb{U}(\mathbf{x})$. Then there exists $\mathbf{y} \in \mathbb{U}(\mathbf{x})$ such that $\|\mathbf{M}(\mathbf{x}) - \mathbf{M}(\mathbf{y})\| > \|\mathbf{x} - \mathbf{y}\|$.*

Proof. See [2]. ■

The idea is that if $\mathbf{a}_0$ is in $\boldsymbol{\beta}(\mathbb{X})$ but not in $\mathbf{\Gamma}(\mathbb{X})$, then the ball of uncertainty around $\mathbf{a}_t$ will grow exponentially as t increases (Fig. 2.13). Therefore, small perturbations in the input could push the trajectory towards another basin of attraction (probably the wrong one). This means that the system is not resistant to input noise. In contrast, Theorem 2.4 shows that if $\mathbf{a}_0$ is in $\mathbf{\Gamma}(\mathbb{X})$, $\mathbf{a}_t$ is guaranteed to remain within a certain distance of $\mathbb{X}$ when the input is bounded.

Theorem 2.3 *Let $\mathbf{M}$ be a differentiable mapping on a convex set $\mathbb{D}$. If $\left\|\mathbf{M}^{'}(\mathbf{x})\right\| \leq \alpha < 1 \, \forall \mathbf{x} \in \mathbb{D}$, then $\mathbf{M}$ is contracting on $\mathbb{D}$.*

[2]This definition is modified based on [25].

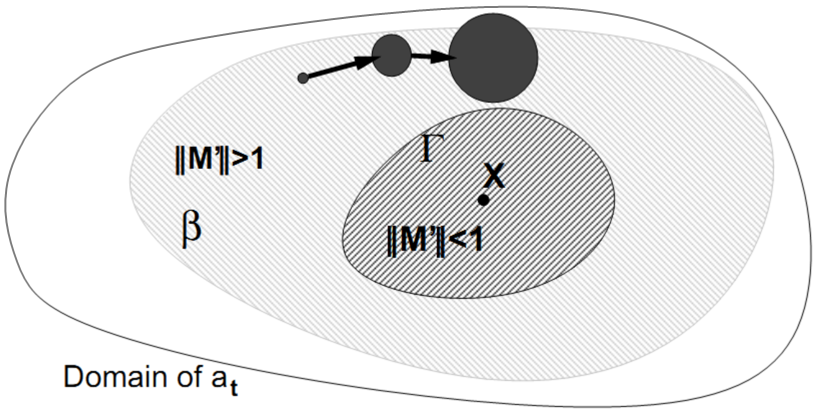


Figure 2.13: Ball of uncertainty grows exponentially outside Γ [2]

Proof. See [26], [27]. ∎

Theorem 2.4 *Let $\tilde{\mathbf{a}}_t$ be the autonomous trajectory obtained with* (2.12) *by starting at $\mathbf{a}_0$ with zero input $\mathbf{u}$. Suppose $\mathbf{a}_0 \in \mathbf{\Gamma}(\mathbb{X})$. Also, suppose $\left\|\mathbf{M}'(\mathbf{y})\right\| < \lambda_t < 1$, $\forall \mathbf{y} \in \mathbb{B}_d(\mathbf{a}_t)$ for some $d > 0$. If $\|\mathbf{u}_t\| < b_t\, \forall t > 0$, where $b_t = (1-\lambda_t)d$, then $\mathbf{a}_t \in \mathbb{B}_d(\tilde{\mathbf{a}}_t)$. Since $\mathbb{B}_d(\tilde{\mathbf{a}}_t) \subset \mathbb{B}_d(\mathbb{X})$, $\forall t > T$ for some $T > 0$, this implies that the system is robustly latched to $\mathbb{X}$ according to Definition 2.6.*

Proof. See [2]. ∎

As long as $\mathbf{a}_t$ remains in $\mathbf{\Gamma}(\mathbb{X})$, a bound on the input can be found that guarantees that $\mathbf{a}_t$ will remain within a certain distance of some points in $\mathbb{X}$ for every time t (Fig. 2.14). This means that the system is robust to the input noise. Hence, to achieve a robust latch of information to the attractor $\mathbb{X}$, the states should be kept in the reduced attracting set $\mathbf{\Gamma}(\mathbb{X})$.

Now we will show the consequences of robust latching: the vanishing gradient problem.

Theorem 2.5 *Suppose that $\left\|\mathbf{M}'(\mathbf{a}_t)\right\| \le \alpha < 1\, \forall t > 0$. Then $\frac{\partial \mathbf{a}_t}{\partial \mathbf{a}_0} \to 0$ as $t \to \infty$.*

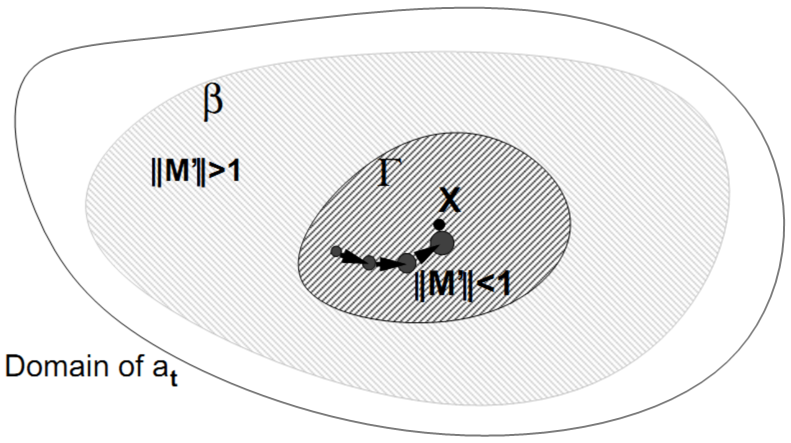


Figure 2.14: Ball of uncertainty is bounded inside Γ [2]

Proof. By the hypothesis, $\left\|\frac{\partial \mathbf{a}_t}{\partial \mathbf{a}_{t-1}}\right\| = \left\|\mathbf{M}'(\mathbf{a}_{t-1})\right\| \leq \alpha \, \forall t > 1$. Therefore,

$$\left\|\frac{\partial \mathbf{a}_t}{\partial \mathbf{a}_0}\right\| = \left\|\frac{\partial \mathbf{a}_t}{\partial \mathbf{a}_{t-1}}.\frac{\partial \mathbf{a}_{t-1}}{\partial \mathbf{a}_{t-2}}\cdots\frac{\partial \mathbf{a}_1}{\partial \mathbf{a}_0}\right\| \leq \left\|\frac{\partial \mathbf{a}_t}{\partial \mathbf{a}_{t-1}}\right\|.\left\|\frac{\partial \mathbf{a}_{t-1}}{\partial \mathbf{a}_{t-2}}\right\|\cdots\left\|\frac{\partial \mathbf{a}_1}{\partial \mathbf{a}_0}\right\| \leq \alpha^t \to 0$$

■

Now consider the derivatives of a performance index F_t at time t with respect to parameters of the system $\mathbf{W}$ (by applying the Backpropagation Through Time (BPTT) algorithm [28]):

$$\frac{\partial F_t}{\partial \mathbf{W}} = \sum_{\tau \leq t} \frac{\partial F_t}{\partial \mathbf{a}_\tau}\frac{\partial \mathbf{a}_\tau}{\partial \mathbf{W}} = \sum_{\tau \leq t} \frac{\partial F_t}{\partial \mathbf{a}_t}\frac{\partial \mathbf{a}_t}{\partial \mathbf{a}_\tau}\frac{\partial \mathbf{a}_\tau}{\partial \mathbf{W}}$$

Suppose that we are in the condition in which the network has robustly latched. For terms with $\tau \ll t$, $\left\|\frac{\partial F_t}{\partial \mathbf{a}_\tau}\frac{\partial \mathbf{a}_\tau}{\partial \mathbf{W}}\right\| \to 0$ since $\frac{\partial \mathbf{a}_t}{\partial \mathbf{a}_\tau} \to 0$. These terms are very small in comparison to terms for which τ is close to t. This means that there is no chance for the terms far from t to change the weights in such a way that allows the network's states to jump to a better region of attraction. This vanishing gradient problem is claimed to be the essential reason that learning long-term dependencies difficult.

In summary, Bengio *et al.*'s main idea is as follows: The problem of learning long-term dependencies could be solved if the network can robustly latch information, i.e. can store information for a long period of time in the presence of noise. More specifically, robust latching is accomplished if the states of the network are contained in the reduced attracting set of some hyperbolic attractors. But if that is true, then the vanishing gradient problem occurs, i.e the portion of gradient of the cost function with respect to the weights due to information far in the past is insignificant compared to the portion at recent times. Because of this problem, gradient descent methods cannot discover the relationship between the output and the input at much earlier times. In other words, learning long-term dependencies using gradient methods is difficult.

2.2.3 Some suggested solutions

To overcome the problem of long-term dependencies, several ideas have been proposed. Some of them use alternative search methods, others use special network architectures. A brief summary of these proposals is given in the following.

Some of search methods without gradient are investigated in [2], such as simulated annealing, multi-grid random search, and discrete error propagation.

In [29], the authors show that the long-term dependencies problem is lessened for NARX networks, since they can retain information longer than conventional recurrent networks. By increasing the number of delays in the output delay line, the vanishing of the gradient in NARX networks can be postponed. The output delay line acts as a jump-ahead connection, providing shortcuts for propagating the gradient information more efficiently when the network is unfolded in time.

Another network architecture is the long short-term memory network [30]. It uses a memory cell containing a self-connected linear unit. This enforces constant error flow, so that it prevents error signals from decaying quickly as they flow back in time.

Another recent attempt to resolve the problem of RNN training is the Echo State Network (ESN) of Jaeger [31]. This approach gives up on learning the hidden-to-hidden weights altogether. Instead, it uses fixed sparse connections, which are generated randomly, and just output weights are learned, so the training is easier. However, since this approach cannot learn new nonlinear dynamics (instead relying on dynamics present in the random reservoir) its power is limited. A survey of ways of generating the reservoirs and training the readouts can be found in the review paper [32].

Martens [33] recently proposed an approach to training RNNs that can solve tasks that exhibit long term dependencies. This approach uses a special variant of the Hessian-free optimization method (aka truncated-Newton or Newton-CG) augumented with a structural-damping technique.

Finally, the authors in [34] refute Bengio *et al.*'s statement by showing that recurrent networks unfolded in time and trained with a shared weight extension of backpropagation are well able to learn long-term dependencies. They say that the analysis in [2] was based on a static view, i.e. only recurrent networks with fixed weights were assumed, and the effect of learning and weight adaption was not taken into account. In [34], the networks have a regularizing effect, i.e. they are able to prolong their information flow and consequently solve the problem of vanishing gradient. Shared weights constrain the networks to change weights in every unfolded time step according to several different error flows, therefore, they allow the networks to adapt the gradient flow.

CHAPTER 3

ERROR SURFACE OF FIRST-ORDER RECURRENT NETWORK

In Chapter 2, we reviewed two mechanisms that might help explain the difficulties in training RNNs. In this chapter, we investigate another mechanism that we believe also contributes to those difficulties - the existence of spurious valleys in the error surface of RNNs, where training algorithms can be easily trapped. The insights gained from the analysis of these valleys will suggest procedures for improving the training of RNNs. This chapter will analyze the error surface of the simplest RNN - the first-order network. This work was first presented in [17], and a later work, [18], provides a mathematical analysis. We will extend the results in this chapter for more general networks in Chapter 4. In this chapter, we first introduce some preliminary material that will be helpful for the purpose of analysis. We then analyze the error surface for linear and nonlinear single neuron networks.

3.1 Preliminary material

In analyzing the spurious valleys of RNNs, we will begin with a linearized network, in which all sigmoid activation functions are replaced with linear activation functions. We will then analyze the resulting linear network using standard linear system theory. In this section, we review some of the key results from linear system theory that we will need later in this chapter and Chapter 4. (See [35] and [36] for more detailed development.) We also present some notation for describing frozen roots (introduced in [18]), which will be essential in describing one type of spurious valley.

First, consider the difference equation representation of a linear dynamic system

(linear recurrent neural network)

$$\begin{aligned} &a(t) + m_1 a(t-1) + m_2 a(t-2) + ... + m_n a(t-n) \\ &\quad = s_1 p(t-1) + s_2 p(t-2) + ... + s_n p(t-n) \end{aligned} \tag{3.1}$$

where $p(t)$ is the input to the system, and $a(t)$ is the system response.

This system can also be represented by its transfer function

$$G(z) = \frac{s_1 z^{n-1} + ... + s_n}{z^n + m_1 z^{n-1} + ... + m_n}. \tag{3.2}$$

The response of this linear system to an arbitrary input sequence, assuming zero initial conditions, can be found from the following convolution sum

$$a(t) = \sum_{i=1}^{t-1} g(i) p(t-i) = \sum_{l=1}^{t-1} g(t-l) p(l) \tag{3.3}$$

where $g(i)$ is the impulse response of (3.1), which can be found from

$$\begin{aligned} &g(i) + m_1 g(i-1) + m_2 g(i-2) + ... + m_n g(i-n) \\ &\quad = \begin{cases} s_i & i = 1, .., n \\ 0 & i > n \end{cases} \end{aligned} \tag{3.4}$$

where $g(i) = 0, i \leq 0$.

The difference equation (3.1) is characterized by the following characteristic equation

$$D_n(\lambda) = \lambda^n + m_1 \lambda^{n-1} + m_2 \lambda^{n-2} + ... + m_n = 0. \tag{3.5}$$

The roots of this equation, $\lambda_i, i = 1, ..., n$, are called the system poles. The impulse response can be written in terms of the system poles, as in

$$g(i) = \sum_{j=1}^{n} \alpha_j (\lambda_j)^i. \tag{3.6}$$

From this, it can be shown ([37], Lesson 8) that

$$\lim_{i \to \infty} \frac{g(i+1)}{g(i)} = \lambda_{max} \tag{3.7}$$

where λ_{max} is the real pole with the largest magnitude. In addition, it is known that (3.1) will be unstable if λ_{max} has magnitude greater than 1.

In addition to the difference equation in (3.1), a linear dynamic system can also be written in state space form:

$$\begin{aligned} \mathbf{x}(t+1) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}p(t) \\ a(t) &= \mathbf{C}\mathbf{x}(t) \end{aligned} \tag{3.8}$$

where $\mathbf{x} = [x_1, \cdots, x_n]^T \in R^n$ is the state vector.

The solution to the state equation for arbitrary initial condition $\mathbf{x}(1)$ can be written

$$a(t) = \mathbf{C}\mathbf{A}^{t-1}\mathbf{x}(1) + \sum_{l=1}^{t-1} \mathbf{C}\mathbf{A}^{t-l-1}\mathbf{B}p(l). \tag{3.9}$$

If we compare this with (3.3), we can see that $g(i) = \mathbf{C}\mathbf{A}^{i-1}\mathbf{B}$. The second term is the convolution sum, and the first term is the initial condition response. From the Cayley-Hamilton Theorem [35],

$$\mathbf{A}^n = -m_1\mathbf{A}^{n-1} - \cdots - m_n\mathbf{I}. \tag{3.10}$$

Therefore, (3.9) can be written

$$\begin{aligned} a(t) &= \sum_{l=1}^{n} \mathbf{C}\mathbf{A}^{t-l-1}\mathbf{x}(1)(-m_l) + \sum_{l=1}^{t-1} \mathbf{C}\mathbf{A}^{t-l-1}\mathbf{B}p(l) \\ &= \sum_{l=1}^{t-1} \mathbf{C}\mathbf{A}^{t-l-1}\mathbf{B}p'(l) = \sum_{l=1}^{t-1} g(t-l)p'(l). \end{aligned} \tag{3.11}$$

This is equivalent to the convolution sum of (3.3), where the first n elements of the input sequence have been modified to include effects from the initial conditions. These modified elements can be computed from

$$p'(l) = p(l) + p^*(l), l = 1, ..., n \tag{3.12}$$

where

$$p^*(l) = \frac{-m_l\mathbf{C}\mathbf{A}^{n-l}\mathbf{x}(1)}{\mathbf{C}\mathbf{A}^{n-l}\mathbf{B}}, l = 1, ..., n. \tag{3.13}$$

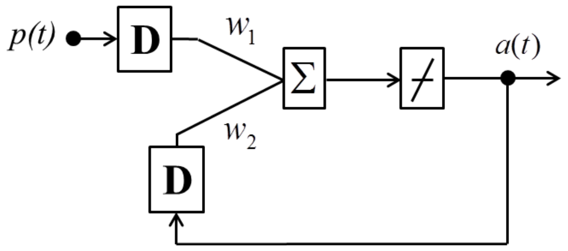


Figure 3.1: One-layer linear network

The choice of state vector is not unique, but for all choices of state, the transfer function of (3.2) can be computed as

$$\begin{aligned} G(z) &= \mathbf{C}(z\mathbf{I}-\mathbf{A})^{-1}\mathbf{B} \\ &= \frac{\mathbf{C}adj(z\mathbf{I}-\mathbf{A})\mathbf{B}}{det(z\mathbf{I}-\mathbf{A})} \\ &= \frac{s_1 z^{n-1}+s_2 z^{n-2}+\cdots+s_n}{z^n+m_1 z^{n-1}+\cdots+m_n}. \end{aligned} \tag{3.14}$$

A key to determining the location of a certain type of spurious valley are the roots of a polynomial whose coefficients are elements of the input sequence:

$$\begin{aligned} P_i^t(\lambda) &= p(i)\lambda^{t-i}+p(i+1)\lambda^{t-i-1}+...+p(t) \\ &= \sum_{l=i}^{t} p(l)\lambda^{t-l} = \sum_{l=i}^{t} p(t-l+i)\lambda^{l-i} \end{aligned} \tag{3.15}$$

At every time point, this polynomial increases in order by 1, according to

$$P_i^{t+1}(\lambda) = \lambda P_i^t(\lambda) + p(t+1). \tag{3.16}$$

We have shown in [18] that if $P_i^t(\lambda)$ has a real root greater than 1 in magnitude (and there is a high probability that this will occur), then $P_i^{t+1}(\lambda)$ also has a root at approximately the same location. We call this a “frozen” root, and it is the cause of spurious valleys in the error surface, as we will demonstrate. If the frozen root, call it r, occurs at time step $t = t^*$, then we can say that

$$P_i^t(r) = \sum_{l=i}^{t} r^{t-l} p(l) = 0, t > t^*. \tag{3.17}$$

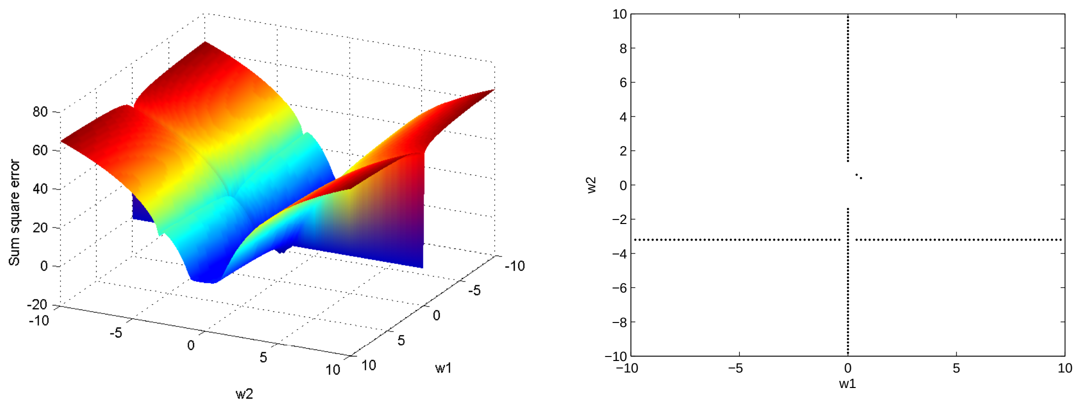


Figure 3.2: Error surface and valleys for one-layer linear network

3.2 Linear network

Consider the single-neuron RNN, with linear transfer function, shown in Fig. 3.1 (same as Fig. 2.2). The difference equation for this network is

$$a(t) - w_2 a(t-1) = w_1 p(t-1). \tag{3.18}$$

In order to generate an error surface for this network, we first generate training data. As we will demonstrate later, the spurious valleys that appear in the surface are not significantly affected by the target outputs. Therefore, we will choose a simple data set. We use a sequence of independent Gaussian random variables with mean zero and variance one for the input $p(t)$. We then use the network of Fig. 3.1 to generate the desired outputs, $d(t)$, with w_1 and w_2 set to 0.5. In this way, we know that the true minimum of the error surface should occur at $w_1 = 0.5$ and $w_2 = 0.5$.

The sum square error (SSE) is then computed with

$$F = \sum_{t=1}^{Q} (d(t) - a(t))^2 \tag{3.19}$$

where Q is the number of points in the input sequence. An error surface and valleys for a particular input sequence are shown in Fig. 3.2. We can see that even this simple network has spurious valleys on the error surface. One is located at the $w_1 = 0$ line, and the other is located at the $w_2 = -3.226$ line. (Note that the locations of these spurious valleys are unrelated to the location of the true minimum at $w_1 = 0.5$ and $w_2 = 0.5$.) In the next sections, we will explain how these spurious valleys occur.

3.2.1 Root valley

The first type of spurious valley (represented by the line at $w_2 = -3.226$ in Fig. 3.2) is determined by the input sequence, and is unrelated to the desired output, as we will show in this section. For the linear system (3.18), we can find the impulse response from

$$g(i) - w_2 g(i-1) = 0 \tag{3.20}$$

with $g(0) = 0$ and $g(1) = w_1$. The characteristic equation of (3.18) and (3.20) is

$$D_1(\lambda) = \lambda - w_2 = 0 \tag{3.21}$$

which means that the system pole is

$$\lambda_1 = \lambda_{max} = w_2 \tag{3.22}$$

From (3.7) and (3.22), we can say that

$$\lim_{i\to\infty} \frac{g(i+1)}{g(i)} = w_2. \tag{3.23}$$

For this system, the ratio will equal w_2 starting at $i = 1$. Using this fact, and (3.3), we can write

$$a(t) = g(1) \sum_{l=1}^{t-1} w_2^{t-l-1} p(l) = g(1) P_1^{t-1}(w_2). \tag{3.24}$$

(It is assumed in (3.3) that the initial conditions are equal to zero. The valleys still occur when the initial conditions are nonzero, but their shapes will vary. In this case, we would use (3.11).)

If we compare (3.17) with (3.24), we can see that the network output will be equal to zero for $w_2 = r$ (the frozen root of $P_1^t(\lambda)$). This root will produce

$$D_1(r) = 0. \tag{3.25}$$

On the other hand, since $\lambda_{max} = w_2 = r$ is greater than 1 in magnitude, the system will be unstable, which would generally lead to very large outputs.

This is a fundamental result, which we will extend in chapter 4 to a very general class of RNN. In a region in the weight space where the network should be unstable (poles of the system are outside the unit circle), the network response is close to zero. This means that if the weight w_2 is changed slightly away from r, the network response will quickly increase. This will cause a steep valley in the error surface, which we call a root valley. For the data set represented in Fig. 3.2, a root valley appears at the line $w_2 = -3.226$. This corresponds to the frozen root r of $P_1^t(\lambda)$.

We should make very clear that the location of the root valley depends only on the input sequence (and potentially on nonzero initial conditions); it does not depend on the desired output. This is a very unexpected result. During training we are trying to get the neural network to approximate some unknown dynamic system that defines the relationship between the inputs and the desired outputs. (In our example here, the unknown dynamic system is the network of Fig. 3.1 with weights $w_1 = 0.5$ and $w_2 = 0.5$.) Now we find that there are steep valleys in the error surface that depend only on the input sequence. And, in fact, they depend only on the first few points of the input sequence. This is because the frozen root generally occurs within the first five to ten time steps and then remains frozen for the rest of the sequence [18].

In chapter 4, we extend these results to a very general RNN of arbitrary size. We first investigate a network with two neurons, before moving to the general case. In each instance, we will have equations equivalent to (3.17), (3.25) and (3.24). In other words, we will have a polynomial $P_i^t(\lambda)$ whose coefficients are elements of the input sequence. (In some cases, the first few coefficients may be modified, as in (3.12).) This polynomial will have a root, r, outside the unit circle:

$$P_i^t(r) = 0, \text{ for some } |r| > 1, t > t^*. \tag{3.26}$$

We will then find a set of network weights so that a root of the characteristic equation (system pole) is equal to that root of $P_i^t(\lambda)$ outside the unit circle:

$$D_n(r) = 0, \text{ for some set of network weights.} \tag{3.27}$$

Because of this root, the network response will be very small, even though the network is unstable. This means that a small change in the weights will cause a large change in the network response, which will result in a very narrow and deep valley in the error surface. (The width of the valleys will decrease, and depth of the valleys will increase, with the length of the input sequence, because, as the length increases, the network outputs will become larger for unstable weights.) The equation that defines the valley will be $D_n(r) = 0$, where the coefficients of the $D_n(r)$ polynomial are functions of the network weights. (In some cases, the first few coefficients of the $P_i^t(\lambda)$ polynomial will also be functions of the network weights, in which case we will also have the equation $P_i^t(r) = 0$ to be satisfied.)

Because these root valleys depend only on the first few steps of the input sequence, and not on the desired outputs (or on the underlying dynamic system that we are attempting to approximate), we call these valleys "spurious." (The dictionary definition of spurious is "Not proceeding from the true source.") The minima of these valleys are unrelated to the global minimum of the error surface. They cause difficulties during training, because they trap the search for the true minimum.

3.2.2 Architecture valley

There is a second spurious valley in Fig. 3.2. This occurs on the line $w_1 = 0$. If $w_1 = 0$, and the initial condition is 0, then the output will be 0 for all values of w_2. Therefore, the sum squared error function F will equal the sum squared targets. This is much smaller than the F values as we move away from the $w_1 = 0$ line, especially in areas where the network is unstable ($|w_2| > 1$).

We call this second type of valley an "architecture" valley, because the location of the valley depends on the architecture of the network. It does not depend on the

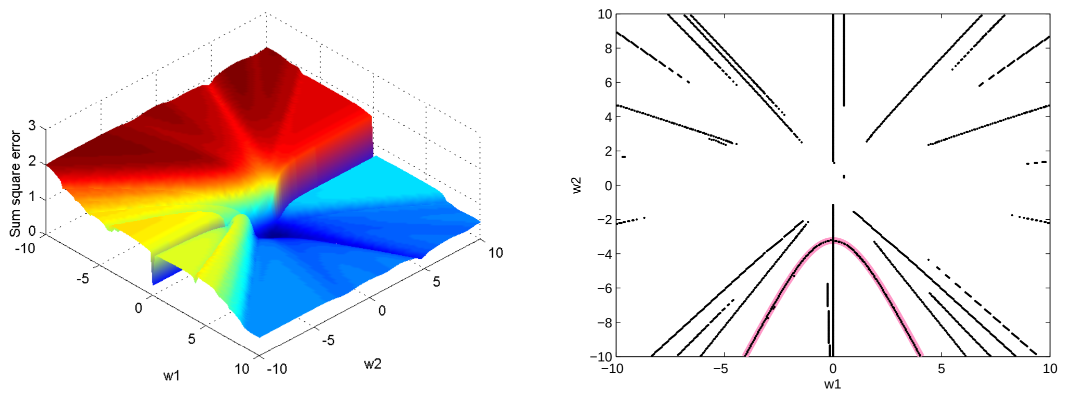


Figure 3.3: Error surface and valleys for one-layer nonlinear network

desired outputs of the training set (or on the underlying dynamic system that we are approximating). In next chapter, we will demonstrate architecture valleys for a more complex network.

3.3 Nonlinear network

Now replace the linear function in Fig. 3.1 by the tanh function. The network output becomes

$$a(t) = \tanh\left[w_1 p(t-1) + w_2 a(t-1)\right]. \tag{3.28}$$

The error surface and valleys for the same input sequence used in Section 3.2 are shown in Fig. 3.3.

3.3.1 Root valley

In this section, we will show that $w_2 = r$ $(D_1(r) = 0)$ still defines the root valley in the nonlinear case, with a modification of the meaning of the root r. We will show that on the root valley, the output is small. When the output is small, the tanh activation functions will be approximately linear, and the same analysis from the linear case can be applied. The output for the first few time steps are (assuming zero initial conditions):

$$\begin{aligned} a(1) &= 0 \\ a(2) &= \tanh\left[w_1 p(1)\right] \\ a(3) &= \tanh\left[w_1 p(2) + w_2 a(2)\right]. \end{aligned} \tag{3.29}$$

At some points (w_1, w_2) on the error surface where $w_1 p(2) + w_2 a(2)$ is small enough, the tanh function can be approximated as linear. Thus we can make the following approximation:

$$a(3) \approx w_1 p(2) + w_2 a(2).$$

If this type of approximation can be applied for the following time points, then we can use the linear difference equation of (3.18), which will be valid for $t \geq 3$, with initial conditions $a(2)$. This means that we can use the modified convolution equation of (3.11), which allows for nonzero initial conditions:

$$a(t) = \sum_{l=2}^{t-1} g(t-l)p'(l), t \geq 3 \tag{3.30}$$

where, using (3.13), we can find the modified input element, $p^{'}(2)$, to be

$$p^{'}(2) \quad = \quad p(2) + \frac{w_2 a(2)}{w_1}. \tag{3.31}$$

By using the same argument as in the linear case, we have an expression for the output similar to (3.24) as the following:

$$a(t) = g(1) \sum_{l=2}^{t-1} w_2^{t-l-1} p^{'}(l) = g(1) P'^{t-1}_{2}(w_2). \tag{3.32}$$

where $P'^{t}_{i}(\lambda)$ is defined as in (3.15), but with $p^{'}(l)$ instead of $p(l)$. If $w_2 = r$, the output will be equal to zero.

Using (3.31) and (3.29), we can rewrite $P'^{t-1}_{2}(w_2)$ as follows:

$$\begin{aligned} P'^{t-1}_{2}(w_2) &= \left[p(2) + \frac{w_2 a(2)}{w_1}\right] w_2^{t-3} + p(3) w_2^{t-4} + ... + p(t-1) \\ &= \frac{a(2)}{w_1} w_2^{t-2} + p(2) w_2^{t-3} + p(3) w_2^{t-4} + ... + p(t-1) \\ &= \frac{\tanh\left[w_1 p(1)\right]}{w_1} w_2^{t-2} + p(2) w_2^{t-3} + p(3) w_2^{t-4} + ... + p(t-1) \end{aligned} \tag{3.33}$$

The difference in the nonlinear case is that the first coefficient in ${P'}_2^{t-1}(w_2)$ depends on the network weight w_1. We have to find the frozen root r of ${P'}_2^{t-1}(w_2)$ corresponding to each value of w_1. The root valley will be formed numerically by points (w_1, w_2) for which $w_2 = r$. In the linear case, r just depends on the elements of the input sequence, so the root valley is a straight line. In the nonlinear case, r varies with the weight w_1. That is why the root valley is a curve as shown in Fig. 3.3. We can see that the approximate root valley (the red curve), determined numerically as above, accurately matches the true root valley.

The root valley in the nonlinear case is determined by finding the frozen root r of ${P'}_2^{t-1}(\lambda)$ and then plugging that root into $D_1(r) = 0$. In the general case, which will be discussed in Chapter 4, a similar pattern emerges. The root valleys are determined by similar simultaneous equations. The polynomial ${P'}_i^t(\lambda)$ has a frozen root, r (for $t \geq t^*$), which is greater than 1 in magnitude. The valley then appears along the hypersurface $D_n(r) = 0$. The difficulty in describing the valley for the general case is caused by the fact that the first few coefficients of ${P'}_i^t(\lambda)$ are functions of the network weights.

The fundamental idea that we have applied to develop the nonlinear root valley for the first-order network (and general networks in the next chapter) is that at a certain point in time (the time point i) and near the root valley, the network response will be small, and the tanh will behave as a linear function. When that is true, we can apply the analysis in the linear case to find the nonlinear root valley. The network response before the the time point i will provide the initial conditions. That is why we will use ${P'}_i^t(\lambda)$ to find the frozen root instead of $P_1^t(\lambda)$.

3.3.2 Architecture valley

Like the linear case, the architecture valley for the nonlinear neuron appears at $w_1 = 0$. If $w_1 = 0$, and the initial condition is zero, then the output equals zero for

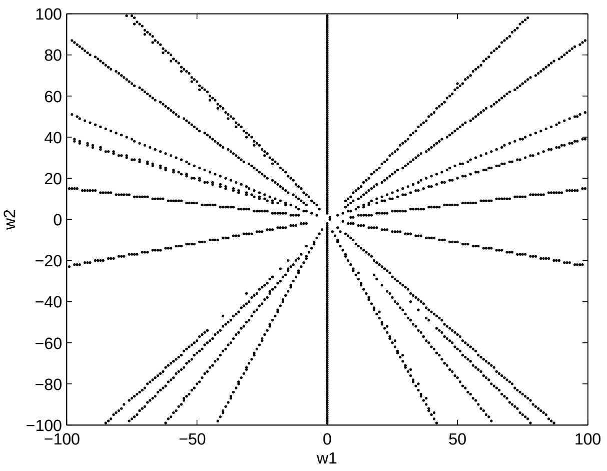


Figure 3.4: Third type valleys

any w_2.

Additional types of architecture valleys are discussed in [18]. They appear because of the saturation effect of the tanh function. One type of valley (the third type) occurs as w_1 and w_2 are large enough to cause saturation in the output at most time points. Fig. 3.4 shows the valleys of the third type. Suppose that the output at time point $k-1$ is saturated at 1 or -1. The output at the time point k is also saturated at 1 for some regions and -1 for some other regions. For some combinations of w_1 and w_2 at which the output switches from 1 to -1 (or visa versa), it will cross the target (which is a value between -1 and 1). Since the outputs at all other time points are not changing (saturated), changing this output will cause a valley in the error surface. The equations for this type of valley can be determined by substituting ± 1 for $a(k-1)$ and $t(k)$ for $a(k)$ in (3.18) (assume that we are still in the linear region of the tanh function). That equation is

$$w_1 = \frac{t(k) \pm w_2}{p(k-1)}. \tag{3.34}$$

This equation predicts the potential valleys of the third type.

The fourth type of valley occurs when w_1 is small and w_2 is large. The output for early time points are near zero. As time progresses, the order of the w_2 terms

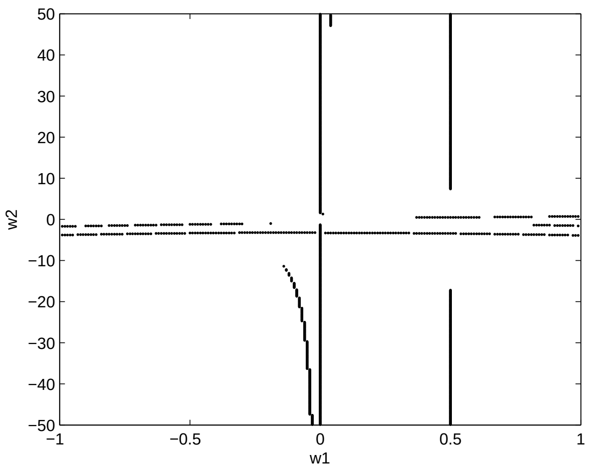


Figure 3.5: Fourth type valleys

increases and the output saturates at 1 or -1. The transition between zero and the saturated values may cause the output to cross the target for some combination of w_1 and w_2. All other outputs are equal to either zero or saturated, so the valley occurs. The equation for the valleys of the fourth type is

$$w_1 = \frac{t(k)}{{w_2}^{k-1} p(1)} \tag{3.35}$$

The curve near $w_1 = 0$ in Fig. 3.5 is a fourth-type valley.

We have investigated the error surface of the simplest RNN in this chapter. We want to extend these results to the error surfaces of more complex networks. This will be done in the next chapter.

CHAPTER 4

ERROR SURFACE OF GENERAL RECURRENT NETWORKS

This chapter extends the results in Chapter 3 for a general class of RNN. This is a further step in understanding the error surface of RNNs, which can then lead to improvements in training. We will analyze the error surface of a second order network (both linear and nonlinear) first, and then we will generalize the results for the class of Layered Digital Dynamic Networks (LDDNs). The analysis will again use the preliminary material introduced in Section 3.1 of Chapter 3. The work in this chapter can be found in [19].

4.1 Second order linear recurrent network

4.1.1 Description of network

Consider the two-layer linear network shown in Fig. 4.1 (also Fig. 2.7). The network can be represented in state space form as

$$\begin{aligned}\mathbf{x}(t+1) &= \begin{bmatrix} w_1 & w_2 \\ 1 & 0 \end{bmatrix}\mathbf{x}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} p(t) \\ a(t) &= \begin{bmatrix} 1 & 0 \end{bmatrix}\mathbf{x}(t)\end{aligned} \tag{4.1}$$

where $x_1(t) = a_2(t)$, $x_2(t) = a_2(t-1)$ and $a(t) = a_2(t)$. The corresponding transfer function is

$$G(z) = \frac{z}{z^2 - w_1 z - w_2} \tag{4.2}$$

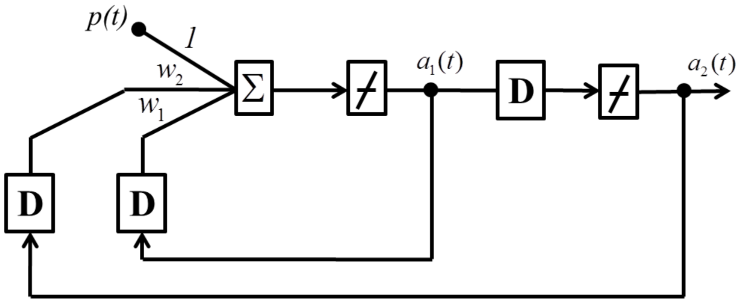


Figure 4.1: Two-layer linear network.

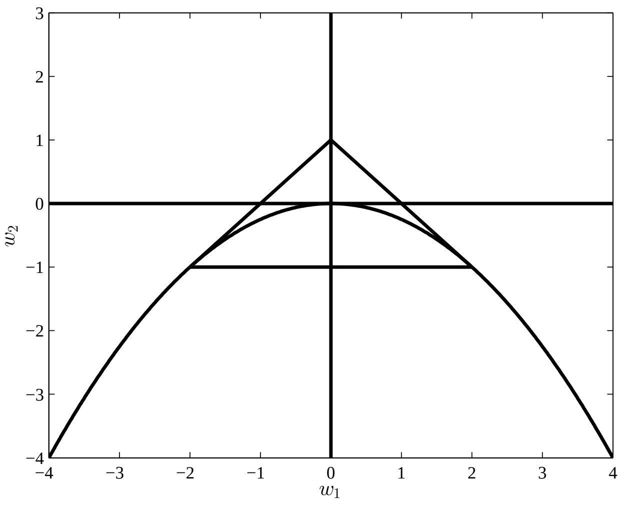


Figure 4.2: Stable region of the Fig. 4.1 network.

and the resulting difference equation is

$$a(t) - w_1 a(t-1) - w_2 a(t-2) = p(t-1) \tag{4.3}$$

which has the following characteristic equation

$$D_2(\lambda) = \lambda^2 - w_1\lambda - w_2 = 0. \tag{4.4}$$

The system will be stable if the poles (roots of the characteristic equation) are inside the unit circle. Using the Jury stability test ([38], page 185) we can find three

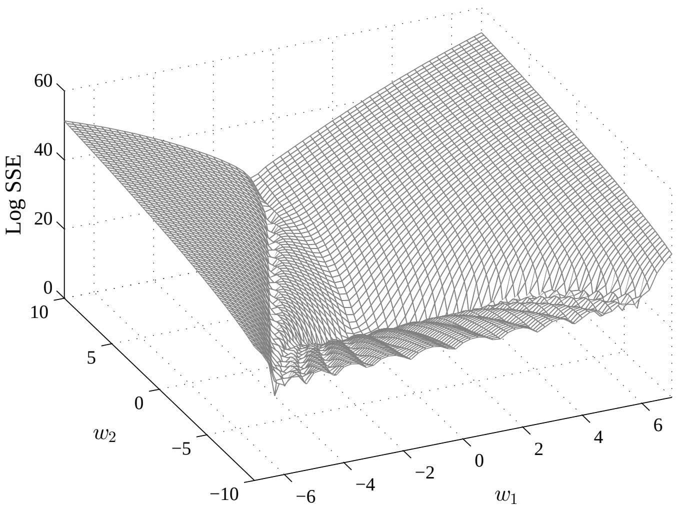


Figure 4.3: Error surface of the Fig. 4.1 network.

conditions for stability:

$$\begin{aligned} w_1 + w_2 &< 1 \\ -w_1 + w_2 &< 1 \\ |w_2| &< 1. \end{aligned}$$

These inequalities define the interior of the triangle shown in Fig. 4.2. (Compare with the bifurcation diagram in Fig. 2.8.) In the region below the parabola in Fig. 4.2 the poles are complex. This region is defined by:

$$w_1^2 + 4w_2 < 0. \tag{4.5}$$

This region will be important in determining the shape of the error surface of the network.

The training data is developed in the same way described in Section 3.2, except that $w_1 = 0.5$ and $w_2 = -0.5$. The error surface for a particular input sequence of 15 points is plotted in Fig. 4.3. Fig. 4.4 is a plot of the valleys. As in the one neuron case, we have root valleys and architecture valleys. In this case, the architecture

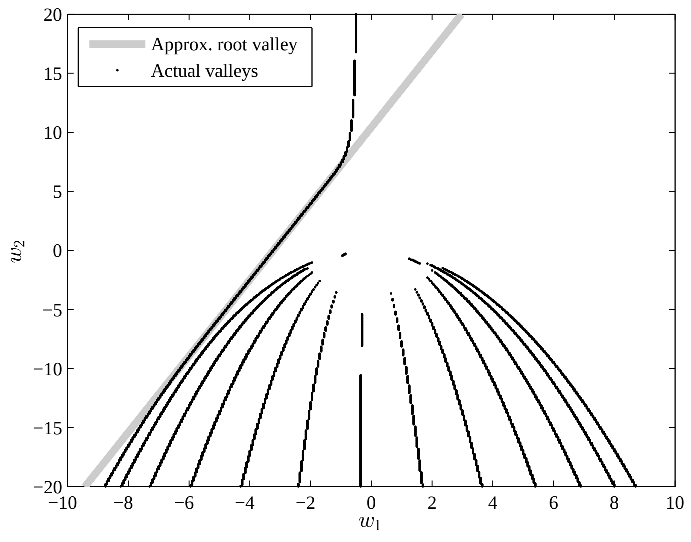


Figure 4.4: Valleys in the error surface.

valleys are a group of parabolas with different curvatures, and the root valley is a line segment to the left of the parabolas.

4.1.2 Root valleys

In order to investigate the root valleys, as in the single neuron case, we consider the polynomial $P_1^t(\lambda)$. Assume that there is a frozen root, r, starting at $t = t^*$, so that (3.17) is satisfied. As in the one neuron case, we want to find the weight values so that r is also a pole (root of the characteristic equation). If we plug in r for λ in (4.4), we have

$$D_2(r) = r^2 - w_1 r - w_2 = 0. \tag{4.6}$$

This equation will define the root valley, and we want to find the values of the weights that will cause this equation to be satisfied. We can rewrite (4.6) as

$$w_2 = -r(w_1 - r). \tag{4.7}$$

We will show that the root valley is a segment of this line.

Since $|r| > 1$, this line lies outside the stable region in Fig. 4.2. It will also be important that r be the largest root of $D_2(\lambda)$. Let $\lambda_1 = r$. Since $\lambda_1\lambda_2 = -w_2$, $\lambda_2 = -\frac{w_2}{r}$. From this condition, we will be able to tell if r is the largest root.

From (3.7), we can say that

$$\lim_{i\to\infty} \frac{g(i+1)}{g(i)} = \lambda_1 = r \tag{4.8}$$

where we are considering the segment of the line (4.7) where $|w_2| < r^2$, and, therefore, $|\lambda_1| > |\lambda_2|$.

Therefore, given $\varepsilon > 0$, there exists an i^* such that $\left|\frac{g(i+1)}{g(i)} - r\right| < \varepsilon$ for every $i \geq i^*$. Therefore, $g(i+1) \approx rg(i)$ for $i \geq i^*$. (In our tests, $i^* = 5$ is usually sufficient.) If we substitute this into the convolution (3.3), we have

$$\begin{aligned}
a(t) &= \sum_{i=1}^{i^*-1} g(i)p(t-i) + \sum_{i=i^*}^{t-1} g(i)p(t-i) \\
&\approx \sum_{i=1}^{i^*-1} g(i)p(t-i) + g(i^*)\sum_{i=i^*}^{t-1} r^{i-i^*}p(t-i) \\
&= \sum_{i=1}^{i^*-1} g(i)p(t-i) + g(i^*)\sum_{l=1}^{t-i*} r^{t-i^*-l}p(l) \\
&= \sum_{i=1}^{i^*-1} g(i)p(t-i) + g(i^*)P_1^{t-i*}(r). \qquad (4.9)
\end{aligned}$$

If $t - i^* \geq t^*$ or $t \geq t^* + i^*$, then the second term is zero (from (3.17)). So

$$a(t) = \sum_{i=1}^{i^*-1} g(i)p(t-i). \tag{4.10}$$

We need to emphasize that this is true for every $t \geq t^* + i^*$.

Therefore, every point on the line (4.7) at which $|w_2| < r^2$ will produce small outputs $a(t)$ for every $t \geq t^* + i^*$ (and so relatively small errors). However, the output at points on either side of this line segment will be significantly higher, since they are in the unstable region of the system. Thus, the SSE is small on this line segment, compared to points on either side. As a result, this line segment becomes a valley in the error surface. Note that this valley does not depend on the targets, only on the input sequence (which determines $P_1^t(\lambda)$) and the network architecture (which determines $D_2(\lambda)$). An example of this line segment is shown in Fig. 4.4, where you can see that the actual valley follows the line segment.

Again, the cause of this valley is that $P_1^t(\lambda)$ has a root, r, greater than 1 in magnitude for $t > t^*$, and that $D_2(r) = 0$ over some region of the weight space. This region of the weight space will form the valley. This same analysis can be applied to linear networks of arbitrary size, as we will show in a later section. Note that if $P_1^t(\lambda)$ has more than one root greater than 1 in magnitude, the error surface will have more valleys of this kind.

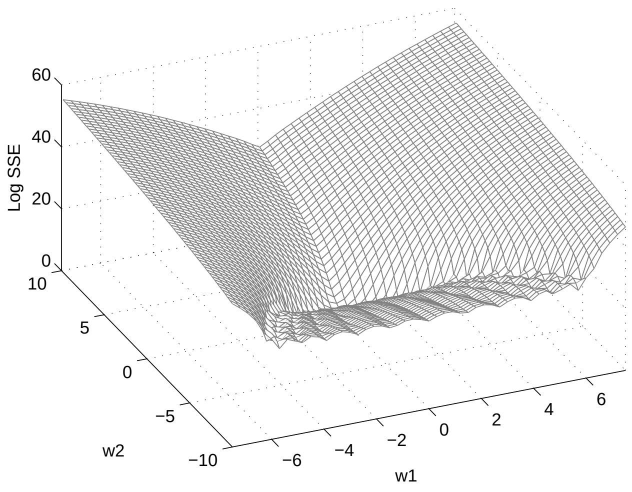
4.1.3 Intrinsic architecture valleys

In the previous sections, we found that the existence and location of root valleys in the error surface depend on the input sequence $p(t)$. How about the parabolic valleys that appear in Fig. 4.4? Does their presence depend on the input sequence? To answer that question, we will develop an error surface that is independent of the input sequence. The idea is to take the average of all possible surfaces (one for each possible input sequence). In other words, we will consider $p(t)$ as a random process and find an expression for the expectation of the SSE (3.19). This average surface is a function of only the two weights w_1 and w_2.

$$F_t(w_1, w_2) = E\left[\sum_{t=1}^{Q} (d(t) - a(t))^2\right] \tag{4.11}$$

From (3.3), we have $a(t) = \sum_{i=1}^{t-1} g(i)p(t-i)$. Therefore, $d(t) = \sum_{i=1}^{t-1} g_t(i)p(t-i)$ where $g_t(i)$ is $g(i)$ for $w_1 = 0.5$ and $w_2 = -0.5$. By substituting these in (4.11), we have

$$\begin{aligned} F_t(w_1, w_2) &= \sum_{t=1}^{Q} E\left[\left(\sum_{i=1}^{t-1} g(i)p(t-i) - \sum_{i=1}^{t-1} g_t(i)p(t-i)\right)^2\right] \\ &= \sum_{t=1}^{Q} E\left[\left(\sum_{i=1}^{t-1} (g(i) - g_t(i))\, p(t-i)\right)^2\right]. \end{aligned}$$



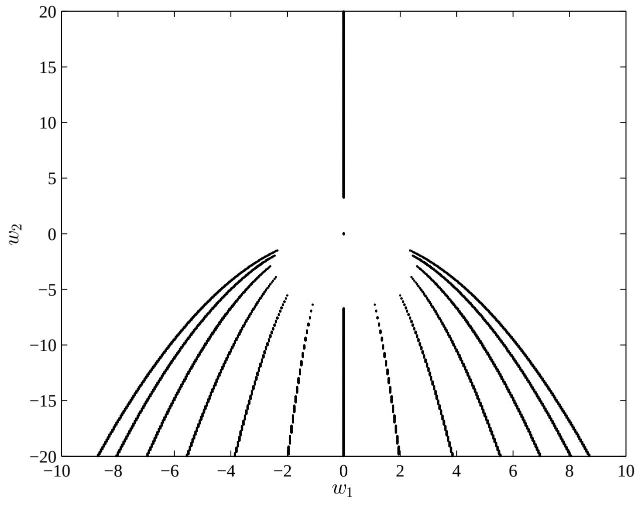


Figure 4.5: Average surface and valleys.

From (3.4) and (3.3) we can see that $g(i)$ is equal to $g_t(i)$ until $i = 1$ and $a(t)$ is equal to $d(t)$ until $t = 2$. Thus, the above equation is just applied for $t \geq 3$ and $i \geq 2$.

Suppose that the $p(i)$ are independent and identically distributed random variables. (For illustration, we will assume mean zero and variance of one.) Then the expectation of every cross term in the sum $\left(\sum_{i=1}^{t-1} (g(i) - g_t(i))\, p(t-i)\right)^2$ is zero. Therefore,

$$\begin{aligned} F_t(w_1, w_2) &= \sum_{t=3}^{Q} \sum_{i=2}^{t-1} (g(i) - g_t(i))^2 \, E\left[(p(t-i))^2\right], \\ &= \sum_{t=3}^{Q} \sum_{i=2}^{t-1} (g(i) - g_t(i))^2, \\ &= \sum_{i=2}^{Q-1} (Q-i)\,(g(i) - g_t(i))^2. \end{aligned} \tag{4.12}$$

The average surface and its valleys are plotted in Fig. 4.5. It looks like the sample surface shown in Fig. 4.3, except that there is no root valley. This confirms that the root valley depends on the input sequence, whereas the parabolic valleys do not. They are properties of the network architecture, and therefore we call them "intrinsic architecture valleys." Next, we will find equations that describe the intrinsic architecture valleys. For simplicity, we need to make some approximations first.

Consider the following approximation of $F_t(w_1, w_2)$:

$$F_{ta}(w_1, w_2) = \sum_{i=2}^{Q-1} (Q-i)\,(g(i))^2 \tag{4.13}$$

The normalized error of this approximation is

$$\frac{F_{ta}(w_1, w_2) - F_t(w_1, w_2)}{F_{ta}(w_1, w_2)} = \frac{\sum_{i=2}^{Q-1} (Q-i)\left[2g(i)g_t(i) - (g_t(i))^2\right]}{\sum_{i=2}^{Q-1} (Q-i)\,(g(i))^2} \tag{4.14}$$

Since the order of the numerator is less than the order of the denominator (notice that the $g_t(i)$ are bounded constants, and $g(i)$ increases with w_2), the above ratio goes to zero as w_2 goes to infinity. Therefore, we can approximate $F_t(w_1, w_2)$ by $F_{ta}(w_1, w_2)$ for large w_2. We will work with the approximate average surface $F_{ta}(w_1, w_2)$ to find the equations for the intrinsic architecture valleys, however, we will continue to use the notation $F_t(w_1, w_2)$ instead of $F_{ta}(w_1, w_2)$.

A natural way to find the valleys is to set the first partial derivatives of F_t equal to zero. Here, we will take the first derivative of F_t with respect to w_1 and solve for w_1. That is

$$\frac{dF_t}{dw_1} = 2 \sum_{i=2}^{Q-1} (Q-i) g(i) \frac{dg(i)}{dw_1} \tag{4.15}$$

By solving (3.4) to find the $g(i)$, and rearranging terms, this can be rewritten as

$$\begin{aligned}\frac{1}{2}\frac{dF_t}{dw_1} &= \{(Q-2)+2(Q-3)w_2+4(Q-4)w_2^2+6(Q-5)w_2^3+\cdots+a_0w_2^{Q-3}\}w_1 \\ &\quad +\{2(Q-3)+8(Q-4)w_2+22(Q-5)w_2^2+\cdots+a_1w_2^{Q-4}\}w_1^3 \\ &\quad +\cdots+a_{Q-3}w_1^{2Q-5} \qquad (4.16)\end{aligned}$$

We want to find the roots of the right-hand side of (4.16), which is a polynomial in w_1. Note that the coefficients of this polynomial are polynomials in w_2. If w_2 is large enough, we can approximate these coefficients by the highest order terms. The polynomial with these approximate coefficients is

$$\begin{aligned}\frac{1}{2}\frac{dF_t}{dw_1} &\approx a_0w_2^{Q-3}w_1+a_1w_2^{Q-4}w_1^3+\cdots+a_{Q-3}w_1^{2Q-5} \\ &= g(Q-1)\frac{dg(Q-1)}{dw_1} \qquad (4.17)\end{aligned}$$

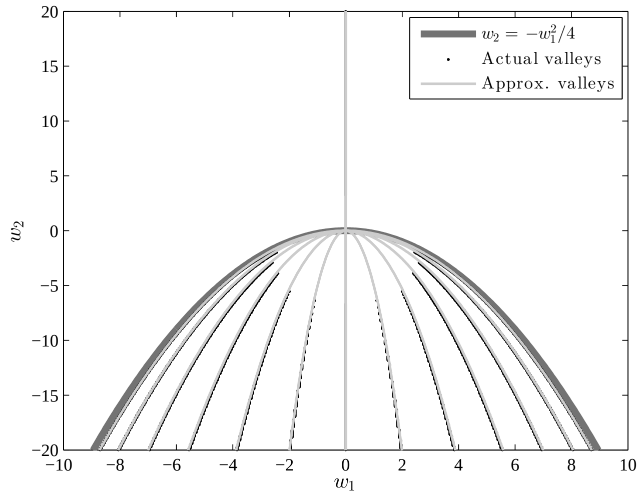
In Appendix A, we show that the roots of the polynomial in (4.16) approach the corresponding roots of the polynomial in (4.17) as w_2 increases. Therefore, to locate the valleys, we can set $g(Q-1)\frac{dg(Q-1)}{dw_1}$ equal to zero and check the second derivative of F_t with respect to w_1. If that second derivative is positive, we have valleys. If it is negative, we have ridges. The second derivative can be written

$$\frac{1}{2}\frac{d^2F_t}{dw_1^2}=\left[\frac{dg(Q-1)}{dw_1}\right]^2+g(Q-1)\frac{d^2g(Q-1)}{dw_1^2}. \qquad (4.18)$$

Now we consider two cases where $\frac{dF_t}{dw_1}=0$. The first case is $g(Q-1)=0$. The second case is $\frac{dg(Q-1)}{dw_1}=0$.

For the first case, where $g(Q-1)=0$, we show in Appendix B that $\frac{d^2F_t}{dw_1^2}>0$. Therefore, $g(Q-1)=0$ is the equation for the valleys. From [39], Corollary 10, we can write the following expressions for $g(Q-1)$:

$$g(Q-1)=\begin{cases} w_1\prod_{k=1}^{(Q-3)/2}(w_1^2+4w_2cos^2(\frac{k\pi}{Q-1})) \text{ if } Q \text{ is odd} \\ \prod_{k=1}^{(Q-2)/2}(w_1^2+4w_2cos^2(\frac{k\pi}{Q-1})) \text{ if } Q \text{ is even}\end{cases} \qquad (4.19)$$



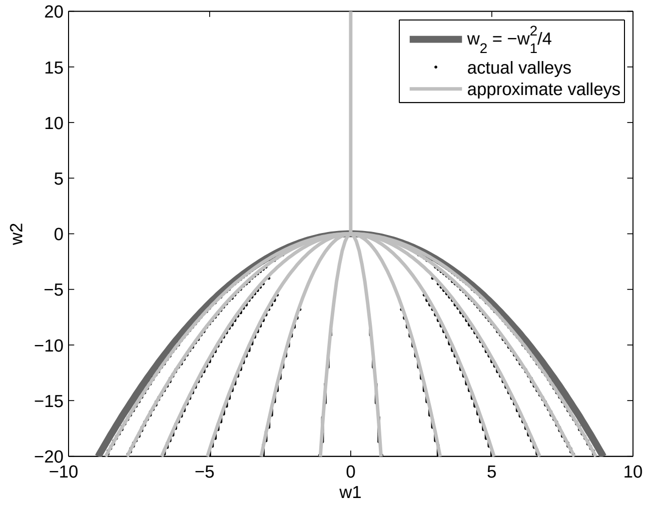


Figure 4.6: Intrinsic architecture valleys and their equations for $Q = 15$ (top) and $Q = 14$ (bottom)

For the second case, where $\frac{dg(Q-1)}{dw_1} = 0$, we show in Appendix C that $\frac{d^2F_t}{dw_1^2} < 0$ if $w_1 \neq 0$. Because the second derivative is negative, we have a ridge, instead of a valley. In this case, we have just one more valley for even Q. That is the vertical line $w_1 = 0$ with $w_2 > 0$.

To summarize, the equations for the intrinsic architecture valleys are as follows (see (4.19)):

- If Q is odd, the architecture valleys are made up of the vertical line $w_1 = 0$ and a family of parabolas $w_2 = -\frac{w_1^2}{4cos^2(\frac{k\pi}{Q-1})}$ for $k = 1, 2, \ldots, \frac{Q-3}{2}$.

- If Q is even, the architecture valleys are made up of the section of the vertical line $w_1 = 0$ with $w_2 > 0$ and the family of parabolas $w_2 = -\frac{w_1^2}{4cos^2(\frac{k\pi}{Q-1})}$ for $k = 1, 2, \dots, \frac{Q-2}{2}$.

Fig. 4.6 shows three sets of curves. The thick, dark curve is $w_2 = -\frac{w_1^2}{4}$, below which the system characteristic equation has complex roots. The intrinsic architecture valleys (except the section of the vertical line $w_1 = 0$ with $w_2 > 0$) fall below this curve. Note that there are always $(Q - 2)$ such valleys. The black curves indicate numerically computed valleys in the average performance surface. The thin, gray curves represent the equations for the approximate intrinsic architecture valleys, as described above. We can see that the approximate valleys accurately match the actual valleys.

A more intuitive way to explain the intrinsic architecture valleys is to use the impulse response $g(i)$. Note that in the region defined by inequality (4.5) (and outside the stable region), the impulse response oscillates and expands with time. Therefore, the last impulse response coefficient $g(t - 1)$, has the most significant affect on the output $a(t)$ in (3.3). At those points (w_1, w_2) where $g(t-1) = 0$, the output becomes "small" (compared to the outputs at those points where $g(t-1) \neq 0$) and the SSE is small as well. This means that $g(t-1) = 0$ is the equation of the intrinsic architecture valleys. We confirmed this experimentally.

If λ_1 and λ_2 are complex conjugate roots, the impulse response can be written as follows ([40], page 75):

$$g(i) = R^{i-1}\frac{sin(i\Theta)}{sin\Theta}$$

where $R = \sqrt{-w_2}$ and the frequency Θ is defined by

$$cos(\Theta) = \frac{w_1}{2\sqrt{-w_2}}, \quad sin(\Theta) = \frac{\sqrt{-(w_1^2 + 4w_2)}}{2\sqrt{-w_2}} \tag{4.20}$$

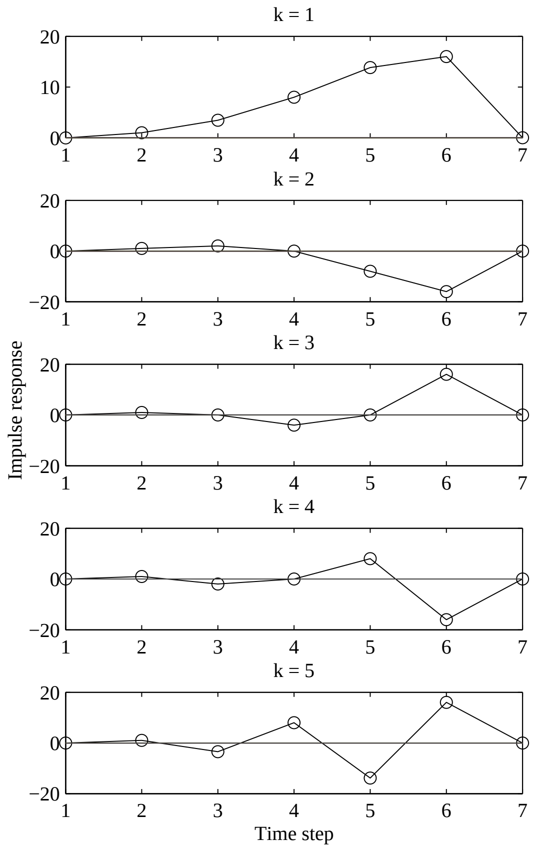


Figure 4.7: Impulse responses for different intrinsic architecture valleys. Last impulse response equals zero.

If we set the last impulse response coefficient $g(Q-1)$ equal to zero, we will have

$$\Theta = \frac{k\pi}{Q-1}, \ \ 1 \leq k \leq Q-2$$

Substituting this into (4.20), we get the same equations for the intrinsic architecture valleys: $w_2 = -\frac{w_1^2}{4cos^2(\frac{k\pi}{Q-1})}$.

Fig. 4.7 shows the impulse responses at different intrinsic architecture valleys (An input sequence of 7 points is used in this demonstration). Each valley corresponds to a different frequency of oscillation of the impulse response, however, the last impulse response coefficient $g(Q-1)$ is always small for all valleys. Fig. 4.8 plots the intrinsic architecture valleys for different frequencies of oscillation of the impulse response.

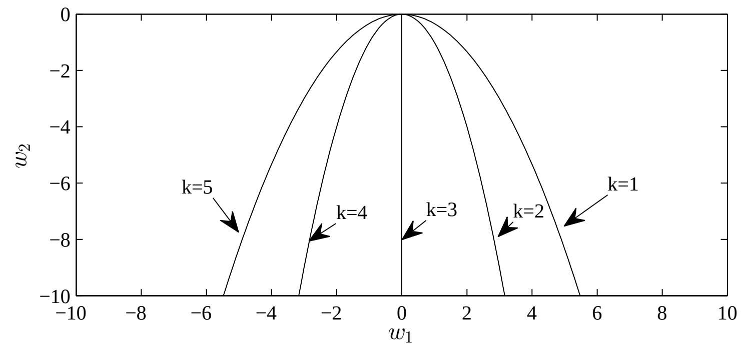


Figure 4.8: Intrinsic architecture valleys for different frequencies Θ.

4.1.4 Sample architecture valleys

In the previous section, we found the locations of architecture valleys on the average performance surface. They are independent of the input sequence. For a particular input sequence, we call these valleys "sample architecture valleys". One example of sample architecture valleys is shown in Fig. 4.4. In this section, we will investigate how the sample architecture valleys are different than the intrinsic architecture valleys.

The sample performance surface is obtained from (3.19):

$$
\begin{aligned}
F &= \sum_{t=1}^{Q} (a(t) - d(t))^2 \\
&= \sum_{t=1}^{Q} \left[\sum_{i=1}^{t-1} (g(i) - g_t(i))\, p(t-i) \right]^2
\end{aligned}
$$

We will use the argument for intrinsic architecture valleys to find the equations for sample architecture valleys. First, we ignore $g_t(i)$ in the above sum and approximate F by

$$F \simeq \sum_{t=1}^{Q} \left[\sum_{i=1}^{t-1} g(i) p(t-i) \right]^2$$

The first derivative of F with respect to w_1 is

$$\frac{1}{2}\frac{dF}{dw_1} = \sum_{t=1}^{Q} \left[\sum_{i=1}^{t-1} g(i) p(t-i) \right] \left[\sum_{i=1}^{t-1} \frac{dg(i)}{dw_1} p(t-i) \right].$$

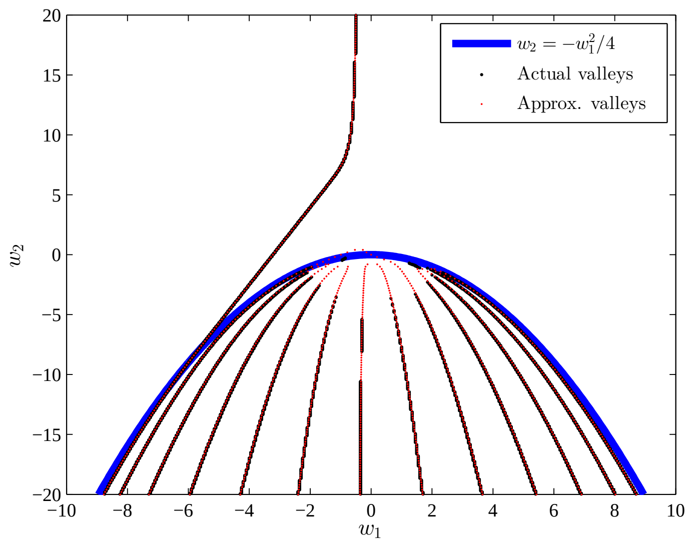


Figure 4.9: Sample architecture valleys and their approximations.

Next, approximate this derivative by the last term in the sum as follows:

$$\frac{1}{2}\frac{dF}{dw_1} \simeq \left[\sum_{i=1}^{Q-1} g(i)p(Q-i)\right]\left[\sum_{i=1}^{Q-1} \frac{dg(i)}{dw_1}p(Q-i)\right]. \tag{4.21}$$

Therefore, either

$$\sum_{i=1}^{Q-1} g(i)p(Q-i) = 0 \tag{4.22}$$

or

$$\sum_{i=1}^{Q-1} \frac{dg(i)}{dw_1}p(Q-i) = 0 \tag{4.23}$$

will define the sample architecture valleys. We will consider these two cases individually.

- $\sum_{i=1}^{Q-1} g(i)p(Q-i) = 0$: Like Appendix B, we can show that if (4.22) is satisfied, the second derivative of F with respect to w_1 is positive. So (4.22) is indeed an equation for sample architecture valleys. Comparing to (3.3), this means that the last output equals zero. This makes sense, since, like the impulse response $g(t)$, the output $a(t)$ is expanding with time. Therefore, the last output dominates the SSE. If it equals the last target (note that targets are small

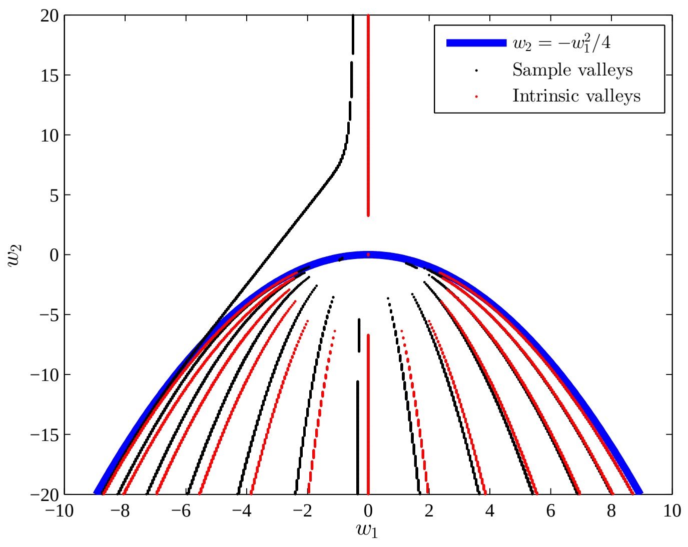
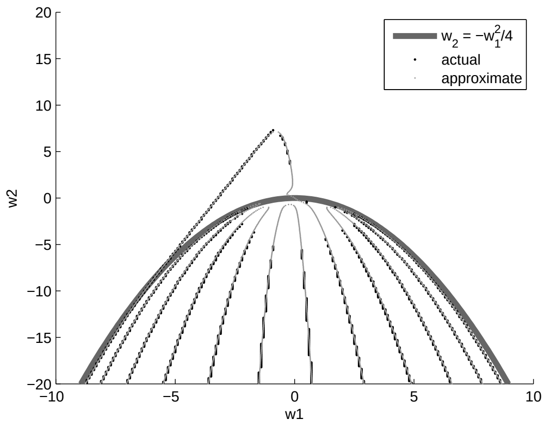


Figure 4.10: Sample architecture valleys compared to intrinsic architecture valleys.

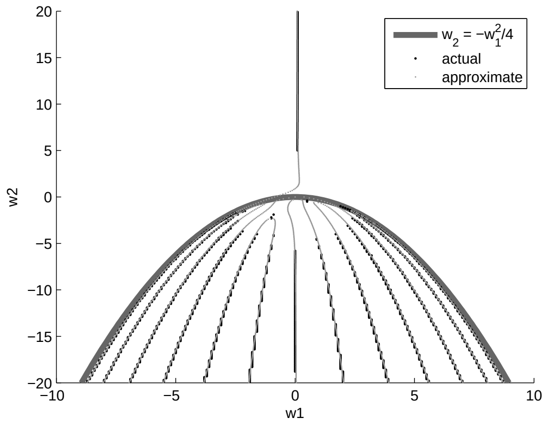
numbers), the SSE becomes small, and we have sample architecture valleys. We confirmed experimentally that at the sample architecture valleys the last output is small. The network responses for the sample architecture valleys are similar in form to the impulse responses shown in Fig. 4.7.

- $\sum_{i=1}^{Q-1} \frac{dg(i)}{dw_1} p(Q-i) = 0$: Unlike Appendix C, the sign of the second derivative of F with respect to w_1 depends on the input sequence. This means that (4.23) could be the equation for valleys or ridges depending on the input sequence. Our experiments show that, in most cases, (4.23) is the the equation for ridges. However, there are some input sequences in which some of the curves from (4.23) are valleys. One of these cases will be illustrated at the end of this section.

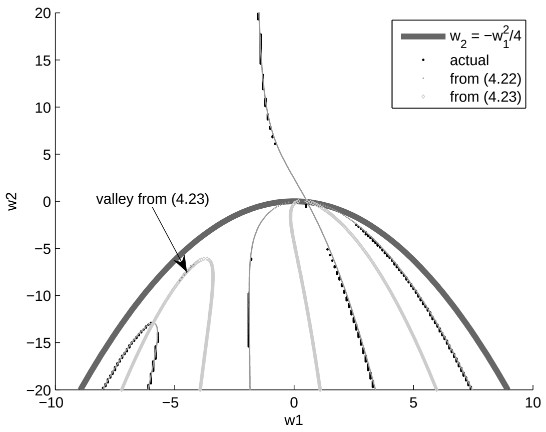
Substituting $g(i)$ from (3.4) into (4.22) and solving for w_1 in terms of w_2, we can obtain equations for the sample architecture valleys. (We can do the same thing for (4.23) to find possible additional sample architecture valleys.) Fig. 4.9 shows the actual valleys for a particular input sequence and the corresponding approximate valleys obtained from (4.22). We can see that the approximate valleys match the actual valleys.



(a) q is even ($q = 14$)



(b) No root valley



(c) There is an architecture valley that comes from (4.23) (7 point sequence)

Figure 4.11: Additional sample valleys.

We also see that, like the intrinsic architecture valleys, all sample architecture valleys (there are still $(Q - 2)$ such valleys) lie in the region below the parabola $w_2 = -\frac{w_1^2}{4}$ (the region in which (4.4) has complex roots). The root valley acts as a transition between two architecture valleys (e.g. the left-most one and the vertical one in Fig. 4.9). The sample architecture valleys are shifted versions of the intrinsic architecture valleys, as shown in Fig. 4.10. The direction of shifting depends on

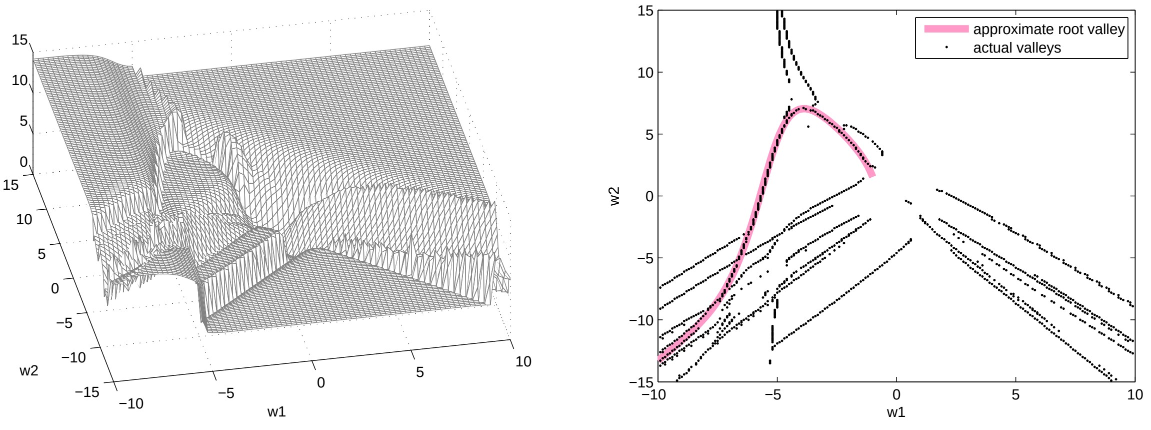


Figure 4.12: Error surface and valleys for a two-layer nonlinear RNN.

the sign of the root of the input sequence, which determines the location of the root valley.

Some more examples of valleys for different input sequences are shown in Fig. 4.11. In the first two examples, all architecture valleys are well-approximated by (4.22). In the last example, in addition to the architecture valleys that are approximated by (4.22), there is a valley that is approximated by (4.23). The remainder of the curves show the locations of the ridges on the error surface. Note that for w_2 large enough, we still have $(Q - 2)$ valleys, as in the other cases. This example illustrates our argument for (4.23) above.

4.2 Second order nonlinear recurrent network

4.2.1 Description of network

The objective of this section is to understand the effect of adding nonlinear transfer functions to the two-layer network. If the linear transfer functions in both layers of the network shown in Fig. 4.1 are replaced by the tanh function, we have

$$a_1(t) = \tanh\left[p(t) + w_1 a_1(t-1) + w_2 a_2(t-1)\right] \tag{4.24}$$

$$a_2(t) = \tanh\left[a_1(t-1)\right] \tag{4.25}$$

As in the one-neuron network case, the nonlinearity makes the shape of valleys in the error surface much more complex. The error surface, and the valleys, for a two-layer nonlinear RNN are shown in Fig. 4.12.

4.2.2 Root valleys

In this section, we will show that $D_2(r) = 0$ still defines the root valley in the nonlinear case, with a modification of the meaning of the root r. Like the one neuron case, we will show that on the root valley, the output is small. When the output is small, the tanh activation functions will be approximately linear, and the same analysis from the linear case can be applied. The layer outputs for the first few time steps are (assuming zero initial conditions, as described for the single neuron case):

$$\begin{aligned}
a_1(1) &= \tanh[p(1)] \\
a_2(1) &= 0 \\
a_1(2) &= \tanh[p(2) + w_1 a_1(1)] \\
a_2(2) &= \tanh[a_1(1)] \\
a_1(3) &= \tanh[p(3) + w_1 a_1(2) + w_2 a_2(2)] \\
a_2(3) &= \tanh[a_1(2)] \\
a_1(4) &= \tanh[p(4) + w_1 a_1(3) + w_2 a_2(3)] \\
a_2(4) &= \tanh[a_1(3)]
\end{aligned} \tag{4.26}$$

At some points (w_1, w_2) on the error surface where $p(3) + w_1 a_1(2) + w_2 a_2(2)$ is small enough, the tanh function can be approximated as linear. Because of this, $a_1(3)$ is also small. Thus we can make the following approximation:

$$a(4) = a_2(4) \approx a_1(3) \approx p(3) + w_1 a_1(2) + w_2 a_2(2)$$

If this type of approximation can be applied for the following time points, then we can use the linear difference equation of (4.3), which will be valid for $t \geq 4$, with initial conditions $a(2)$ and $a(3)$. This means that we can use the modified convolution equation of (3.11), which allows for nonzero initial conditions:

$$a(t) = \sum_{l=3}^{t-1} g(t-l)p'(l), t \geq 4 \tag{4.27}$$

where, using (3.13), we can find the modified input elements, $p^{'}(3)$ and $p^{'}(4)$, to be

$$\begin{aligned} p^{'}(3) &= p(3) + w_1 a_1(2) + w_2 a_2(2) \\ p^{'}(4) &= p(4) + w_2 a_2(3) \end{aligned} \tag{4.28}$$

Now let r be the largest root of $D_2(r) = 0$. By using the same argument as in the linear case, we have an expression for the output similar to (4.9). To be on the root valley, we need to have

$$P'^{t-i^*}_3(r) = \sum_{l=3}^{t-i^*} r^{t-i^*-l} p^{'}(l) = 0 \tag{4.29}$$

where $P'^{t}_{i}(\lambda)$ is defined as in (3.15), but with $p^{'}(l)$ instead of $p(l)$.

So now, as in the linear case, the root valley is defined by two equations:

$$\begin{aligned} P'^{t-i^*}_3(r) = 0 \\ D_2(r) = 0 \end{aligned} \tag{4.30}$$

The difference in the nonlinear case is that the first two coefficients in $P'^{t-i^*}_3(r)$ ($p^{'}(3)$ and $p^{'}(4)$) depend on the network weights w_1 and w_2, whereas in the linear case, the coefficients of $P^{t-i^*}_1(r)$ were just the elements of the input sequence. To find the root valley in the linear case, we could simply find the frozen root of $P^{t-i^*}_1(\lambda)$, and then plug that value, r, into $D_2(r) = 0$. This gave us the line $w_2 = -r(w_1 - r)$. For the nonlinear case we have to solve the simultaneous equations in (4.30).

It is difficult to find a closed form solution to these equations, but we can find a numerical solution using the following procedure. If we substitute (4.28) and $w_2 =$

$r^2 - w_1 r$ into (4.29) we have

$$\begin{aligned} {P'}_3^{t-i^*}(r) &= a_2(2)r^{t-i^*-1} + [a_2(3) - w_1 a_2(2)]\, r^{t-i^*-2} \\ &\quad + [p(3) + w_1 a_1(2) - w_1 a_2(3)]\, r^{t-i^*-3} + \sum_{l=4}^{t-i^*} p(l) r^{t-i^*-l} \end{aligned} \tag{4.31}$$

Now let $p^{''}$ be the input sequence p, except for the following points:

$$\begin{aligned} p^{''}(1) &= a_2(2) \\ p^{''}(2) &= a_2(3) - w_1 a_2(2) \\ p^{''}(3) &= p(3) + w_1 a_1(2) - w_1 a_2(3) \end{aligned} \tag{4.32}$$

(Notice that these terms are only functions of w_1, and not w_2.) Then

$$ {P'}_3^{t-i^*}(r) = {P''}_1^{t-i^*}(r) = \sum_{l=1}^{t-i^*} r^{t-i^*-l} p^{''}(l) \tag{4.33}$$

If r is the frozen root of ${P''}_1^{t-i^*}(\lambda)$, then $w_2 = -r(w_1 - r)$ is the equation for the root valley.

The procedure to find the nonlinear root valley can be summarized as follows:

For each value of w_1:

1. Compute $a_1(2)$, $a_2(2)$ and $a_2(3)$ from (4.26).

2. Form $p^{''}$ as in (4.32). Find r, which is the frozen root of ${P''}_1^t(\lambda)$.

3. Compute $w_2 = -r(w_1 - r)$. (Note that the condition $|w_2| < r^2$ needs to be satisfied, so that r is the largest root of (4.4).)

Fig. 4.12 shows an example error surface and associated valleys. You can see that the approximate root valley, determined by the procedure above, accurately matches a true valley. As in the linear case, the root valley is a part of $D_2(r) = 0$. In the linear case, r is the frozen root of $P_1^t(\lambda)$ and is fixed. In the nonlinear case, r is the frozen root of ${P''}_1^t(\lambda)$ and varies with the weight w_1 (and various initial conditions).

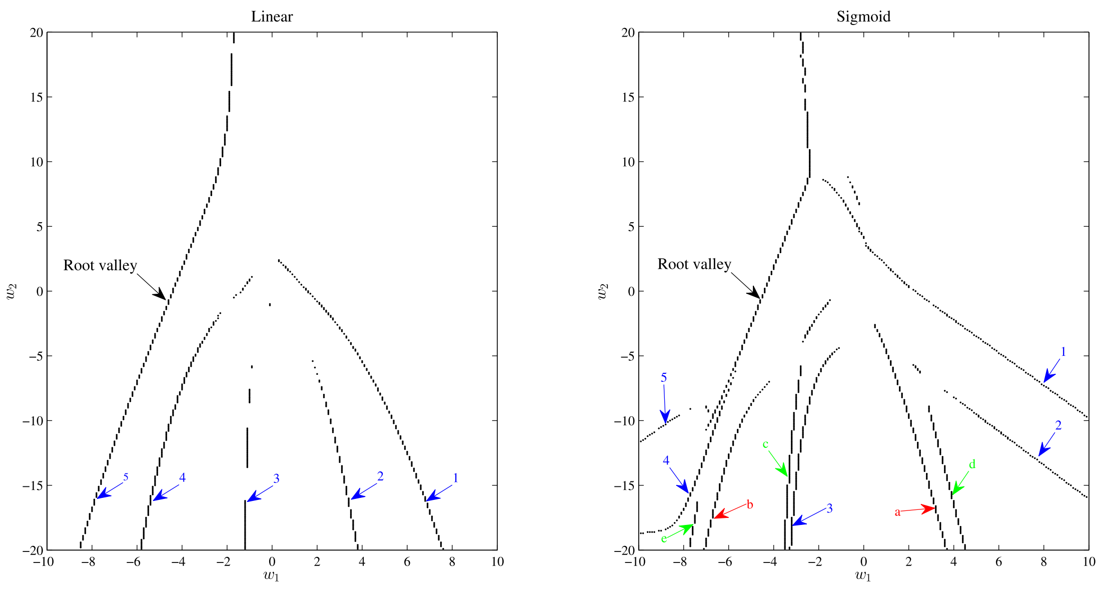


Figure 4.13: Movement of architecture valleys from the linear case to the nonlinear case (7 point sequence).

We want to emphasize that the mechanism that causes the nonlinear root valley is unchanged: the input sequence (or adjusted input sequence) has a frozen root outside the unit circle. The shape of the valley is changed because of the saturation of the sigmoid function.

In the general case, which will be discussed in a later section, a similar pattern emerges. Root valleys are determined by simultaneous equations, like those in (4.30). The polynomial ${P'}^t_i(\lambda)$ has a frozen root, r (for $t \geq t^*$), which is greater than 1 in magnitude. The valley then appears along the hypersurface $D_n(r) = 0$. The difficulty in describing the valley for the general case is caused by the fact that the first few coefficients of ${P'}^t_i(\lambda)$ are functions of the network weights. We are able to solve this numerically for the second order network, because we can solve $D_n(r) = 0$ for w_2, and then use that to eliminate w_2 from the other equation.

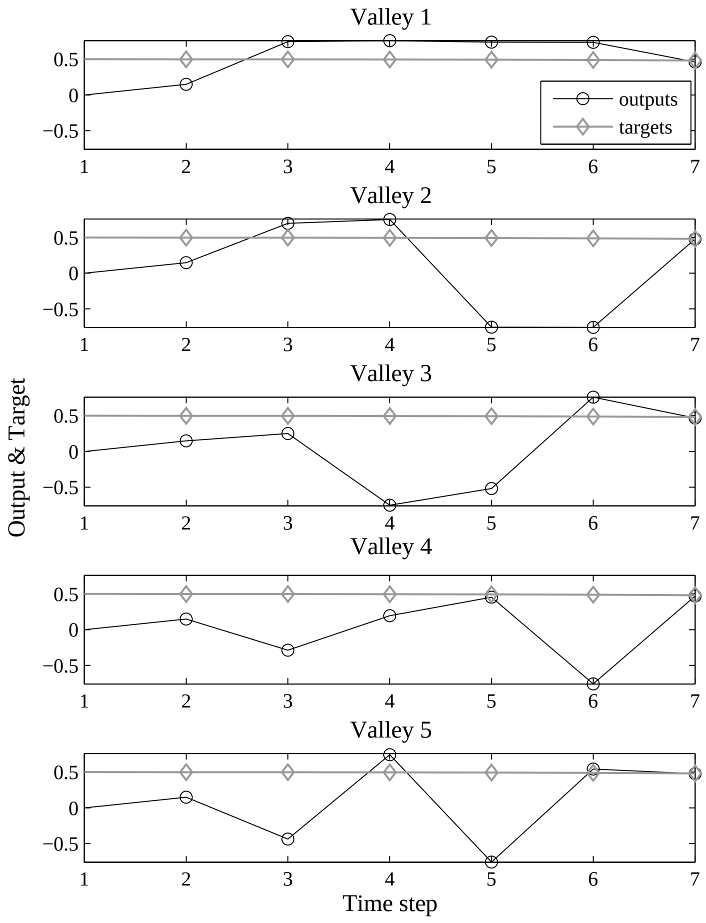


Figure 4.14: Outputs at basic architecture valleys.

4.2.3 Architecture valleys

Architecture valleys in the nonlinear case are created by the same mechanisms as the linear case, but their shapes are modified. Fig. 4.13 shows an example of how the architecture valleys change as we go from the linear case to the nonlinear case. In the nonlinear case, valleys 1 and 2 move to the right, valleys 3, 4 and 5 move to the left. (Note that valley 5 in the linear case - the part below the tangent point - is one of the architecture valleys.) In addition, some new valleys appear (see valleys a through e in Fig. 4.13). We can see that all of the architecture valleys in the linear case are still present in the nonlinear error surface. (We call them basic architecture valleys.)

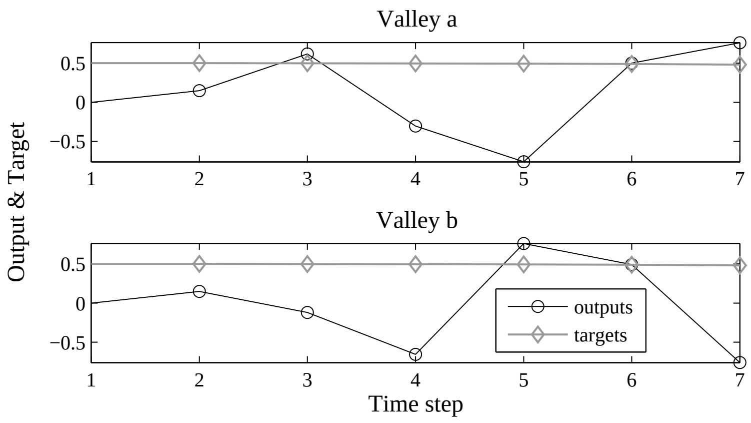


Figure 4.15: Outputs at new architecture valleys.

They just move to new locations. The new architecture valleys appear around them.

We verified experimentally that the mechanism that causes these nonlinear architecture valleys is similar to the linear case (see section 4.1.4): one of the last outputs is changing the most quickly and is crossing the corresponding target, while the other outputs are almost constant. For the basic architecture valleys, the last output is changing the most quickly between $\tanh(1)$ and $-\tanh(1)$, and it is crossing the last target (see valleys 1 through 5 in Fig. 4.13). The outputs at these valleys have patterns as shown in Fig. 4.14. Each valley corresponds to a different frequency of oscillation of the network response. This is similar to the linear case.

For the new architecture valleys, in the process of changing from one frequency to another, the last output stays saturated at $\tanh(1)$ or $-\tanh(1)$. If the second to last output changes the most quickly, and crosses the second to last target, we have new valleys (valleys a and b in Fig. 4.13). The outputs at these valleys have patterns as shown in Fig. 4.15. Furthermore, if the last two outputs are saturated, and if the third to last output changes the most quickly and crosses the third to last target, we have additional new valleys (see valleys c, d and e in Fig. 4.13). This cycle has the potential to continue until the first time point, so it can cause many more new valleys around the basic valleys. These new valleys are similar to the type 3 valleys described in [18] for the single neuron network.

We can see that, by replacing the linear transfer function with the sigmoid transfer function, we have increased the number and complexity of the architecture valleys. These valleys inherit properties of valleys that appear in the linear case, however, they are more numerous and complex because of the saturation of the sigmoid transfer function.

4.3 A general class of RNN

In this section, we will discuss how the results for the one- and two-layer RNNs can be extended to a more general class of RNN. We first analyze linear networks (where linear functions replace sigmoid functions), before proceeding to nonlinear networks.

4.3.1 Layered digital dynamic networks (LDDNs)

We now consider a very general class of RNN - the Layered Digital Dynamic Network - first introduced in [41]. The net input $n^m(k)$ for layer m of an LDDN can be computed

$$\begin{aligned}\mathbf{n}^m(k) = &\sum_{l\in L_m^f}\sum_{d\in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(k-d)\\ &+\sum_{l\in I_m}\sum_{d\in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(k-d)+\mathbf{b}^m \qquad (4.34)\end{aligned}$$

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. The output of layer m is

$$\mathbf{a}^m(k) = \mathbf{f}^m(\mathbf{n}^m(k)) \qquad (4.35)$$

for $m = 1,\ 2,\ \cdots,\ M$, where $\mathbf{f}^m$ is the transfer function at layer m. The set of M paired equations (4.34) and (4.35) 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 LDDN can be linearized by making all activation functions in (4.35) linear.

Any linear LDDN can be represented in state space form using (3.8). The corresponding transfer function can then be obtained using (3.14), which leads to difference equation (3.1) and characteristic equation (3.5). The output can be computed by convolving the input sequence with the impulse response, as in (3.3) (or (3.11) for nonzero initial conditions).

4.3.2 Root valleys

The steps we followed in section 4.1.2 for the two layer network can be followed almost exactly for the general case. If $P_1^t(\lambda)$ has a frozen root r bigger than 1 in magnitude, the error surface will have a root valley. That valley can be obtained by substituting r into the characteristic equation (3.5). In other words, the equation of the root valley is:

$$D_n(r) = r^n + m_1 r^{n-1} + \cdots + m_n = 0 \tag{4.36}$$

where $P_1^t(\lambda)$ has a frozen root, r (for $t \geq t^*$). The valley is therefore defined by the simultaneous equations

$$\begin{aligned} P_1^t(r) &= 0, t \geq t^* \\ D_n(r) &= 0. \end{aligned} \tag{4.37}$$

We will summarize the main points that lead to this conclusion.

1. For all sets of weights that satisfy (4.36), there is a system pole outside the unit circle, which means that the network is unstable. (Note that the m_i are complex functions of the LDDN layer weights.)

2. If there is a region on (4.36) where r is the root of $D_n(\lambda)$ with largest magnitude,

then we can approximate the impulse response using $g(i+1) = rg(i)$ for $i > i^*$. This leads to (4.9). Since r is the frozen root of the $P_1^t(\lambda)$, the output is small.

3. The root is frozen as time increases, so the output remains small after a certain time point.

4. As the network weights vary even slightly from the hypersuface (4.36), the network output will increase dramatically, since that hypersurface is in the unstable region of the network.

4.3.3 Architecture valleys

We can extend the analysis of the architecture valleys in the two-layer network case to explain how these valleys will form in the general case.

1. The intrinsic architecture valleys appear at those points where the last impulse response, $g(Q-1)$, equals zero. Their presence is independent of the input. They are properties of the network architecture.

2. The sample architecture valleys appear when the last output equals zero. They are shifted versions of the intrinsic architecture valleys.

4.3.4 Valleys for nonlinear RNNs

Because of the saturation effect, the nonlinear valleys are modified (in terms of shape and location) of the linear valleys. In addition, the number of valleys is multiplied.

1. The nonlinear root valley is a shifted version of the linear root valley. Starting at a certain point in time and at certain places in the error surface, the layer outputs of the network become small. Therefore, the sigmoid activation functions can be approximated as linear. Therefore, the network operation can

be described by the linear difference equation (3.1). The root, r, that is used to determine the root valley equation, using (4.36), is changing since the first n coefficients of the input sequence are modified by the first n network outputs (see (3.13)). (This involves modifying (4.37) to replace $P_1^t(r)$ with ${P'}_n^t(r)$.) Thus, the nonlinear root valley is shifted compared to the linear case. The shift depends on the s_i (see (3.1)). Note that the s_i depend on the input weights of the LDDN.

2. The architecture valleys still appear. They just move to new locations. Also, new architecture valleys appear, as described for the two layer network.

Although we can not visualize the error surface and valleys of a general RNN, the difference equation (3.1) allows us to use the method of analysis for the two-layer RNN to analyze the general RNN. The understanding of the mechanisms that cause the spurious valleys can help us avoid them during training. It is not necessary to know the exact locations of the valleys. It is enough to know that they are not related to the problem that the RNN is being trained to solve. They are functions of the network inputs in the training set, the initial states of the network, and the network architecture.

CHAPTER 5

PROCEDURE FOR TRAINING RECURRENT NETWORKS

We analyzed the error surface of RNNs in Chapter 3 and Chapter 4. The presence of spurious valleys in the error surface of RNNs makes training, especially using batch, gradient-based methods, very difficult. We know that these valleys are not related to the true minimum of the surface, or to the problem the RNN is trying to solve. They depend on the input sequence in the training data, the initial conditions and the network architecture. Using this knowledge, in this chapter, we propose a procedure for efficient training of RNNs. The procedure uses a batch training method based on a modified version of the Levenberg-Marquardt algorithm and some techniques to avoid the spurious valleys.

The structure of this chapter is as follows: We review some properties of the spurious valleys of RNNs in Section 5.1 and the LM algorithm in Section 5.2. Section 5.3 proposes techniques that can mitigate the effects of the valleys in the error surface and Section 5.4 introduces the training procedure. (Most of the material in this chapter was presented in [20].)

5.1 Properties of valleys of recurrent networks

In this section, we will review the properties of spurious valleys in the error surface of RNNs introduced in Chapter 3 and Chapter 4. One property of these valleys is that they are caused by network instability and are related to the input sequence. In the unstable region of network, the output grows without bound (for linear networks) or saturates (for nonlinear networks). However, there are some locations in the unstable

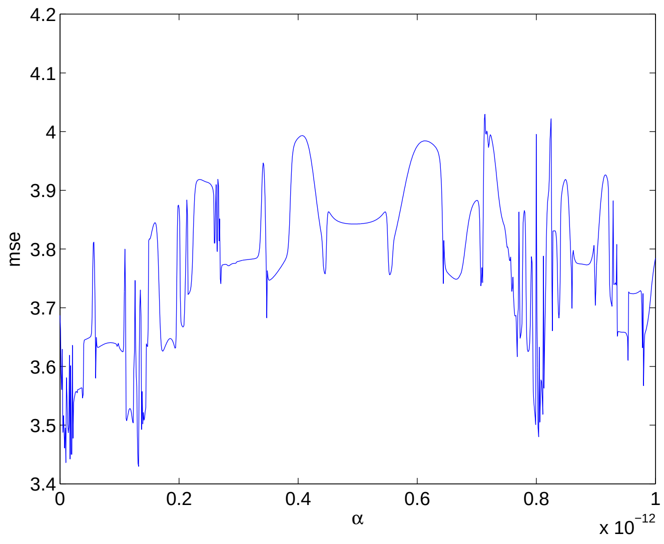


Figure 5.1: Error profile along the gradient direction

region (certain combinations of weights) where the output is still small for a particular input sequence. For example, at locations where the weights satisfy the simultaneous equations in (4.37) (the equations for the root valleys), the output equals zero. Since the outputs nearby are much bigger, a valley appears. We would like to emphasize that the valleys occur because of the instability (the valleys are in the unstable region) and the input sequence. If the input sequence (or the initial condition) is modified, it will produce a valley in a different location.

Another property of the spurious valleys, especially the root valleys, is that they are very narrow and have steep slopes. (Some of the valleys were found to have widths on the order of 10^{-15} as shown in Fig 5.1.) This is understandable, because of the sudden change in the output at the valleys. This also means that the norm of gradient of the performance index with respect to the weights would be very large. Therefore, the norm of the gradient would indicate when the search reaches a valley.

Another property is that as the training sequence gets longer, more valleys appear in the error surface. An example is shown in Section 4.1.4, where the number of architecture valleys is $Q - 2$, where Q is the training sequence length (see Fig. 4.9).

Furthermore, longer sequences also produce steeper valleys (see Fig. 4 in [18]).

As the RNNs become larger, with more layers, neurons, or feedback connections, the valleys become more numerous and more complex. An error profile for a practical network is shown in Fig. 5.1. The plot shows a cross section of the error surface (the 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 during the training process.

5.2 Levenberg-Marquardt algorithm: A short description

This section gives a brief description of the Levenberg-Marquardt (LM) algorithm, which will be modified in a later section so as to avoid the spurious valleys. Details about LM can be found in [42] and [43]. The LM algorithm is a variation of Newton's method where the performance index is sum squared error (SSE). The update rule for weights and biases $\mathbf{x}_k$ at the k^{th} iteration is

$$\Delta \mathbf{x}_k = -\left[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}\right]^{-1} \mathbf{J}^T(\mathbf{x}_k)\mathbf{e}(\mathbf{x}_k) \tag{5.1}$$

where $\mathbf{e}$ is the network error and $\mathbf{J}$ is the Jacobian matrix of the network errors with respect to the weights. $\mathbf{J}$ can be computed using backpropagation. For RNNs, we need to use dynamic backpropagation. Jacobian calculations for a general dynamic network can be found in [41].

This algorithm has the very useful feature that as μ_k is increased it approaches the steepest decent algorithm with a learning rate of $1/\mu_k$, while as μ_k is decreased to zero the algorithm becomes Gauss-Newton. The algorithm begins with a small μ_k (e.g., $\mu_k = 0.01$). If a step does not yield a smaller value for the SSE, then the step is repeated with μ_k multiplied by some factor $\vartheta > 1$ (e.g., $\vartheta = 10$). Eventually the SSE should decrease, since we would be taking a small step in the steepest descent direction. If a step does produce a smaller value for SSE, then μ_k is divided by ϑ for the next step, so the algorithm will approach Gauss-Newton, which should provide faster convergence. The algorithm provides a nice compromise between the speed of Newton's method and the guaranteed convergence of steepest descent. We can see that in the LM algorithm, the SSE is always decreasing from one iteration to the next.

One stopping criterion for the LM algorithm is μ_k reaching a maximum value (e.g., $\mu_{max} = 10^{10}$) without the SSE decreasing. This rarely happens for FNN training. (FNN training usually stops when the norm of the gradient has reduced below some minimum value.) However, this is not the case for RNN training. In RNN training, μ_k is quite likely to reach μ_{max}, especially for closed-loop training with long input sequences (see Section 5.4). As μ_k reaches μ_{max}, we would take a very small step in the steepest descent direction. If the SSE cannot be reduced without making μ_k very large, then there exists a very narrow valley at the current weights. This also means that the norm of the gradient at that weight location would be very large. Increasing μ_{max} does not help in this case, since eventually we will get trapped in that narrow valley. Therefore, the idea is to escape from the valley. The next section will propose some techniques to avoid the numerous valleys and to escape from the valleys if the search is trapped.

5.3 Techniques for avoiding valleys in the error surface

There have been several techniques that have been proposed to improve the performance of RNN training algorithms [18]. These techniques include regularization, switching training sequences, randomly setting initial conditions and maintaining network stability during training. We review those methods in this section, and then we introduce a new technique in the following section. The first modification is regularization, in which the sum square error performance function is combined with a penalty term:

$$\mathbf{J}(\mathbf{x}) = \text{SSE} + \alpha\text{SSW}$$

where SSE is the sum squared errors and SSW is the sum squared weights. This performance function helps to force the weights back into the stable region, because it overwhelms the spurious valleys for large values of the weights. The regularization factor α is decreased during training to ensure that the final trained weights are not biased.

Another technique for improved training involves using more than one training sequence. Because spurious valleys depend on the input sequence, for any two random sequences the valleys will appear in different locations. The idea is to use one sequence for a given number of iterations and then to switch to a new sequence. If we become trapped in a spurious valley, that valley will disappear when the new sequence is presented.

Another method to move the valleys is to use random initial conditions. We have seen in Chapter 4 that the spurious valleys move when the initial conditions of the network are changed. The initial conditions can be changed to small random values in combination with the switching of sequences described above.

The analysis in Chapter 3 and 4 suggests yet another approach for avoiding the spurious valleys. Because the valleys occur in the unstable regions of the parameter space, one might be able to use a constrained optimization process to avoid these instabilities during training. Some stability criteria for RNNs have been recently developed in [44] and [45]. The next step is to incorporate these constraints into recurrent RNN training algorithms.

In the next section, we will propose a new technique to improve RNN training. We explained in Section 5.2 that μ reaching μ_{max} is an indication that the LM search algorithm has fallen inside a valley. Using this knowledge, we can modify the LM algorithm to improve convergence. The overall idea is to train with multiple sequences, and then if μ reaches μ_{max} to remove the sequence (or sequences) that are associated with the valley. The removed sequence should have larger gradient than the other sequences.

5.4 Training procedure

Before introducing the training procedure, we need to explain some concepts that are commonly used in recurrent network training.

5.4.1 Useful concepts

Open-loop training and one-step-ahead prediction

We can consider the output of an RNN as an estimate of the output of the nonlinear dynamic system that we are trying to model. The output is fed back to the input of some layers in the network. Because the true output is available during the training of the network, we can use the true output instead of feeding back the estimated output. This is similar to the series-parallel configuration introduced in [46]. The advantage of this configuration is that the input to some layers of the network will be more accurate. Therefore it is easier for the algorithm to find the true minimum. Also, by removing feedback loops we reduce the chance for spurious valleys.

Because of the series-parallel configuration, we are doing one-step-ahead predictions. It turns out that doing one-step-ahead predictions is a very helpful initial step in RNN training. For this initial step, all feedback loops from the network output can be opened. After completing the one-step-ahead training, the feedback loops are closed for multi-step-ahead training.

Closed-loop training and multiple-step-ahead prediction

The original RNN is in the parallel configuration [46] (closed-loop form). We want to train this closed-loop network to perform an iterated prediction over many time steps (multiple-step-ahead prediction). (Note that for closed-loop training, we need to use dynamic backpropagation.) The time horizon of the prediction is determined by the length of the training sequences and the number of delays in the networks. For example, if the training sequences have a length of 5 time steps, and the maximum number of delays in the network is 2, then training the closed-loop network means that we are doing 3-step-ahead predictions.

Because of the relationship between the sequence length and the width of the spurious valleys mentioned above, we will start closed-loop training with short training sequences first, and then we will increase the prediction horizon. This requires that we segment the original long training sequences into shorter sequences.

5.4.2 Procedure

Using techniques mentioned in Section 5.3, we propose a procedure for training a general RNN that is based on the following steps:

1. Open-loop training (one-step-ahead predictions)

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

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

   (In addition to removing sequences, two other techniques could be used to move the valleys: make a small change in initial conditions and/or remove a couple of time points at the end of all training subsequences.)

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

In the original LM algorithm, the training stops if μ reaches μ_{max}, and in these cases the resulting trained network performance is often very poor. In this modified LM algorithm, the training will continue, using new training data with some subsequences (those with the biggest gradients) removed. This helps the algorithm avoid being trapped at the valleys in the error surface. We will demonstrate the performance of the modified algorithm on several test problems in next chapter.

CHAPTER 6

TRAINING RECURRENT NETWORKS FOR MODELING AND CONTROL OF PHYSICAL SYSTEMS

This chapter uses the procedure introduced in Chapter 5 to train practical recurrent networks. One of the very useful applications of recurrent networks is the identification and control of dynamic systems. We will show how RNNs can be trained for modeling and control of two physical systems: magnetic leviation and double pendulum.

6.1 System identification

The first step involved when using neural networks for control is system identification. In this step, we train a neural network to represent the dynamics of the plant. The error between the plant output and the neural network output is used to update the parameters of the neural network. The process is illustrated in Fig. 6.1.

One model that could be used for nonlinear identification is the nonlinear autoregressive with exogenous inputs (NARX) model (see Fig. 6.2). We will use a NARX network for modeling in Section 6.3.

6.2 Model reference adaptive control

The model reference adaptive control (MRAC) architecture was first introduced in [46]. The MRAC consists of two parts: a plant model network and a controller network, as shown in Fig. 6.3. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.

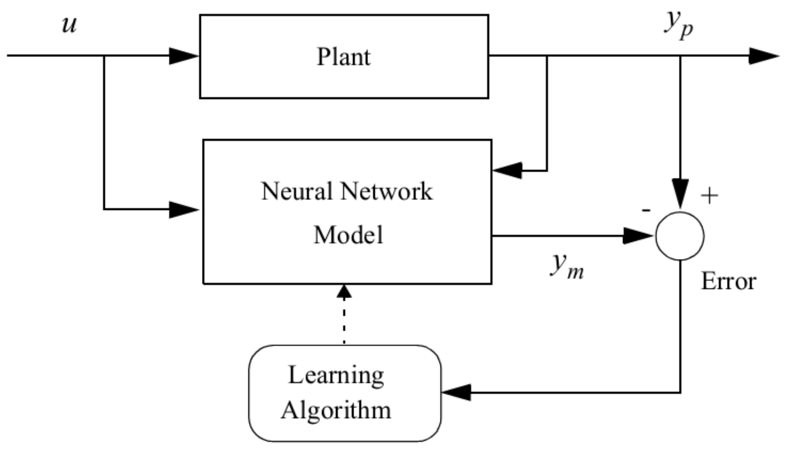


Figure 6.1: Plant identification [3]

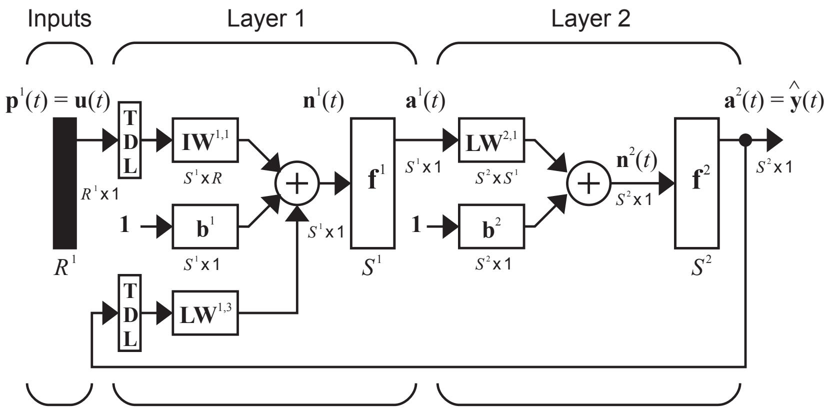


Figure 6.2: NARX recurrent network [4]

The neural network plant model is used to assist in the controller training. Fig. 6.4 shows the details of the MRAC architecture, in which the plant model is a NARX network and the controller is another NARX network. This architecture will be used for controlling the magnetic levitation system in Section 6.3.

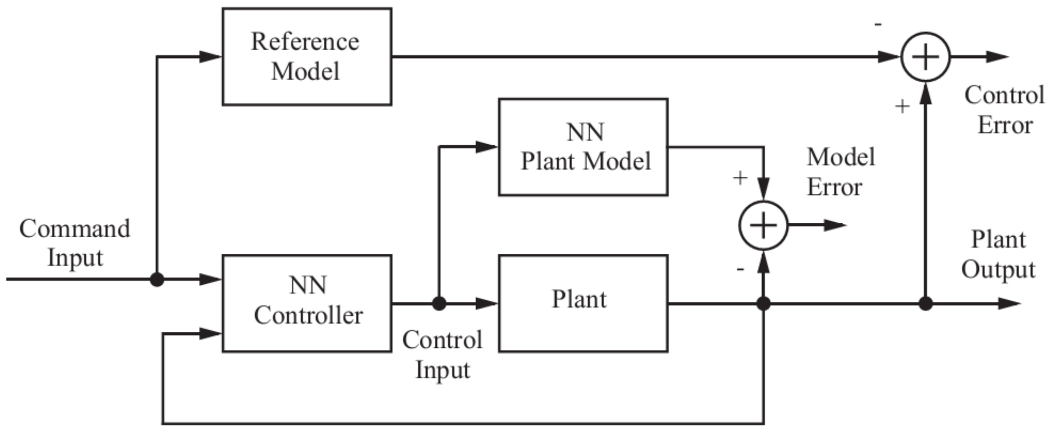


Figure 6.3: Model reference adaptive control structure [3]

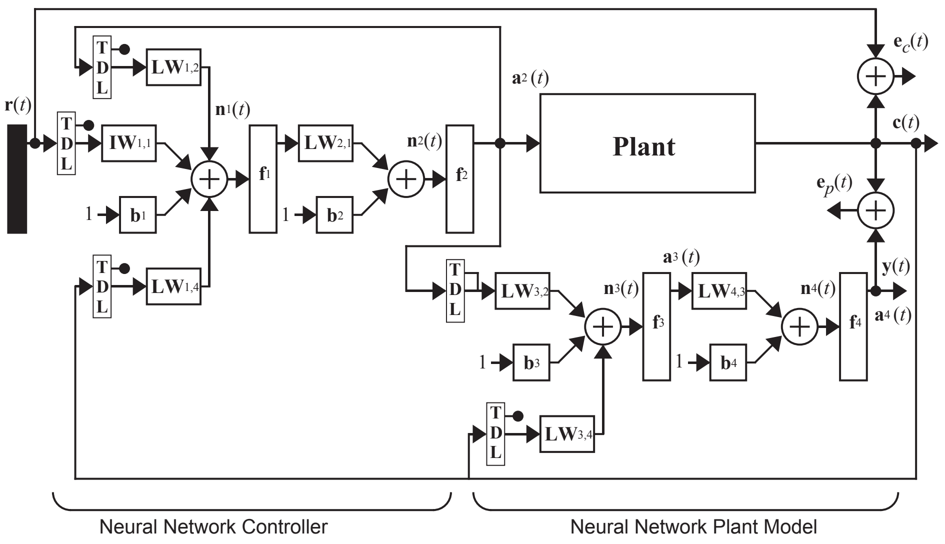


Figure 6.4: Details of model reference adaptive control structure [3]

6.3 Simulation results

6.3.1 Magnetic levitation

System description

The magnetic levitation system consists of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as shown in Fig. 6.5.

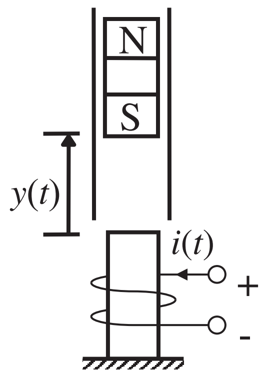


Figure 6.5: Magnetic levitation system.

Table 6.1: Simulation parameters for the magnetic levitation

β	α	g	M	$i(t)$	Sampling interval
12	15	$9.8\,\mathrm{m/s^2}$	3 kg	$2-4$ A	0.05 s

The equation of motion of the magnet is

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)sgn\,[i(t)]}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt} \tag{6.1}$$

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. Simulation parameters are given in Table 6.1 [3].

System identification

First, we need a set of training data. We apply random inputs consisting of a series of pulses of random amplitude and duration. An example of the training data is shown in Fig. 6.6. We use a NARX network (Fig. 6.2) to model this system. For this task, we use 3 input delays and 2 output delays (so the prediction begins with the fourth data point) and 10 hidden neurons.

The first step of the procedure in Section 5.4.2 is open-loop (one-step-ahead pre-

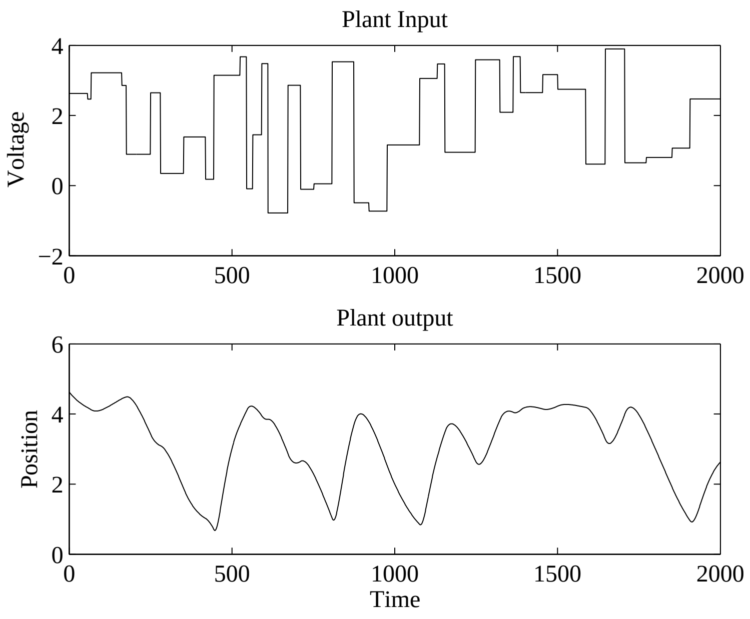


Figure 6.6: Training data for magnetic levitation identification.

diction) training. In this phase, there are two inputs to the series-parallel network: the input sequence and the target sequence.

After finishing the open-loop training, we come back to the original NARX network for multiple-step-ahead prediction. The prediction horizon k is gradually increased, based on the following list:

$$\{2 : 1 : 50\,/\,55 : 5 : 200\,/\,250 : 50 : 500\,/\,600 : 100 : 1997\}$$

where $a : m : b$ means we go from a steps to b steps with the jump distance of m steps. The selection of this list depends on how difficult the fitting problem is. In this particular example, at the beginning (the coarse tuning phase), we slowly increase k. Later, in the fine tuning phase, we could increase k faster to save the training time.

Fig. 6.7 shows the results of the 1997-step-ahead prediction. We hardly see any difference between the actual position of the magnet and the position predicted by the NARX network, since the error is so small. We will use this model to train the controller in the next section.

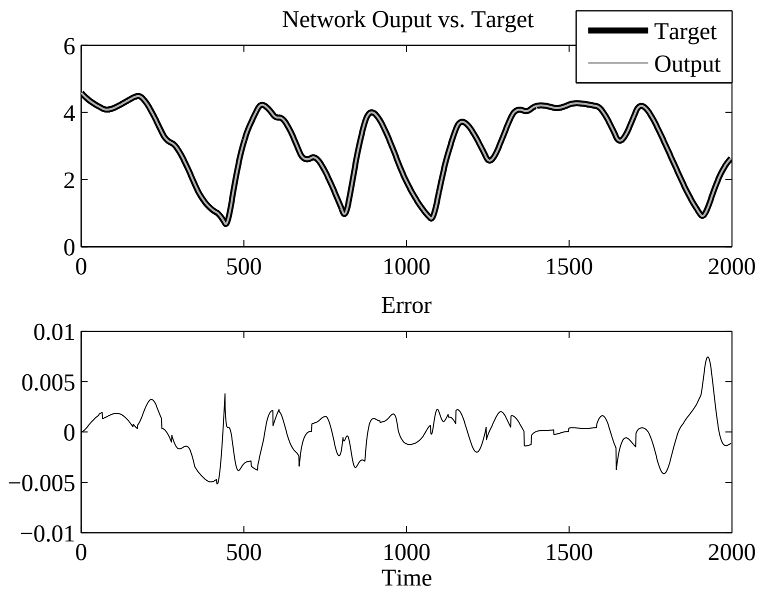


Figure 6.7: Magnetic levitation identification: 1997-steps-ahead prediction.

Controller training

We will use the MRAC architecture (see Fig. 6.4) in this section. There are three sets of controller inputs: delayed reference inputs, delayed controller outputs (plant inputs), and delayed plant outputs. The chosen number of delays for all three inputs is 2. (Typically, the number of delays increases with the order of the plant.) We use 10 hidden neurons for the controller.

For the open-loop training phase, we use the targets (reference model outputs) wherever the plant model outputs are fed back. In other words, we open two feedback connections from the last (fourth) layer to the first layer and the third layer. Notice that we still have a feedback connection in the controller subnetwork. Since the plant model network has already been trained, its weights are fixed during the controller training process. We also set the initial output weights of the controller to zero, so it gives the plant zero initial input.

After successful open-loop training, we can follow the same steps we used for plant training (given in Section 5.4.2) to train the closed-loop network.

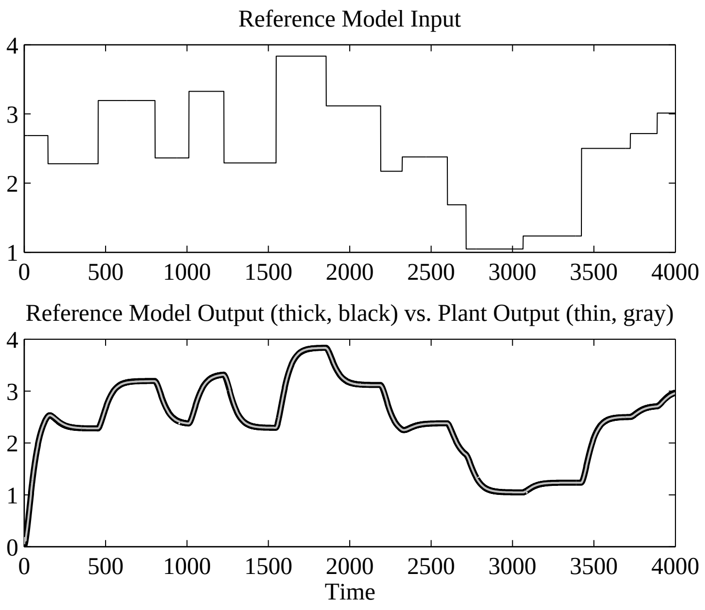


Figure 6.8: Model reference control training.

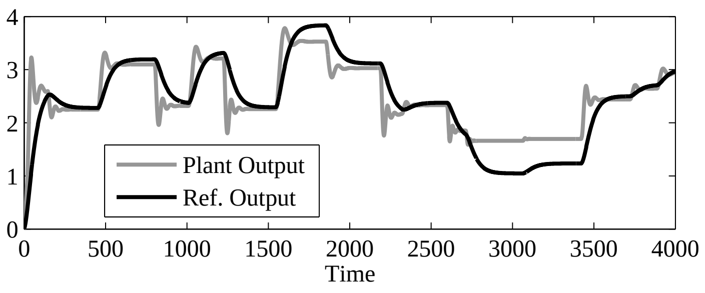


Figure 6.9: Model reference control training without using the new procedure.

The result of controller training is shown in Fig. 6.8. We can see that the plant output matches the reference model output very well. (To verify the advantage of our procedure, i.e., how incrementally increasing the prediction horizon can improve convergence, we trained the MRAC closed-loop network directly with the full-length training sequence (3997-step-ahead prediction). The training result is shown in Fig. 6.9, in which we can see that the search fails to converge to an accurate controller.)

Now we can test the operation by applying an arbitrary input to the trained MRAC network. We can see from Fig. 6.10 that the plant model output does follow the reference model input (and matches the reference model output). The result is

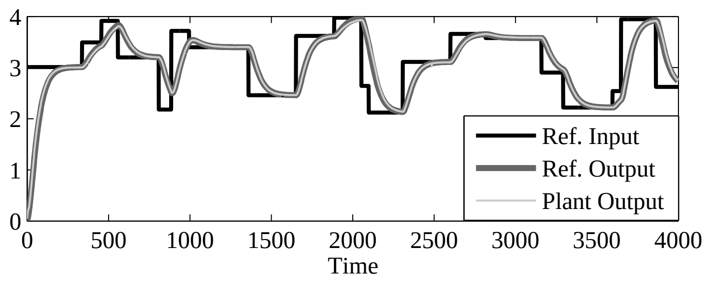
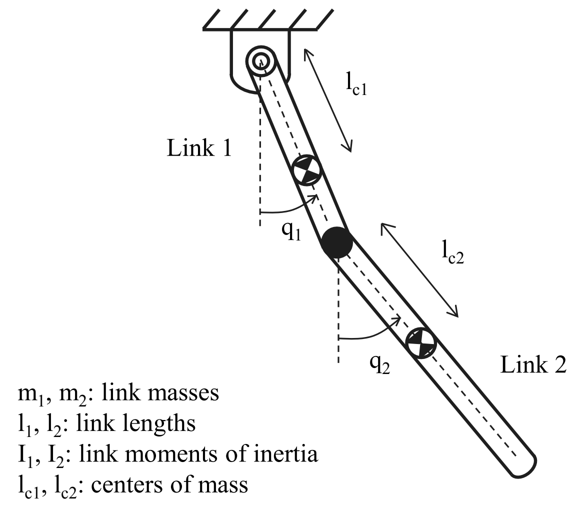


Figure 6.10: Model reference control testing.



l_{c1}
Link 1
q_1
l_{c2}
q_2
Link 2
m_1, m_2: link masses
l_1, l_2: link lengths
I_1, I_2: link moments of inertia
l_{c1}, l_{c2}: centers of mass

Figure 6.11: Double pendulum system.

good in both transient and steady-state regions, even though the input sequence is not the same as the input sequence in the training data.

6.3.2 Double pendulum

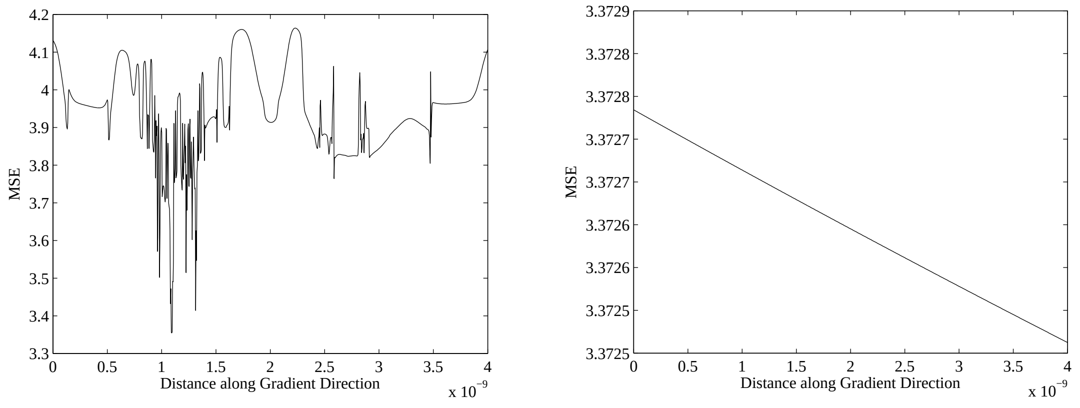
Next, we will train a recurrent network to model a double pendulum. In this section, we only consider system identification and not controller design.

System description

A double pendulum (Fig. 6.11) is a physical system that exhibits rich dynamics and is sensitive to initial conditions. It is known to be chaotic.

Table 6.2: Double pendulum parameters

l_1	l_2	m_1	m_2	I_1	I_2
1 m	2 m	1 kg	2 kg	0.1 kg m^2	0.7 kg m^2



(a) Original error profile (b) New error profile

Figure 6.12: Change of error profile as a sequence is removed.

If we define the state vector to be

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \equiv \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} \tag{6.2}$$

then equations of motion for the double pendulum can be written (from [47])

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \tag{6.3}$$

where $\mathbf{f}$ is given by

$$\mathbf{f} = \begin{bmatrix} x_3 \\ x_4 \\ f_3 \\ f_4 \end{bmatrix} \tag{6.4}$$

where

$$
\begin{aligned}
f_3 &= \frac{1}{\gamma(x_2)}\{c_1c_3(x_3+x_4)^2\sin(x_2) - c_2c_4g\sin(x_1) \\
&\quad + c_3^2x_3^2\cos(x_2)\sin(x_2) + c_3c_5g\sin(x_1+x_2)\cos(x_2)\} \\
f_4 &= \frac{1}{\gamma(x_2)}\{-[c_2+c_3\cos(x_2)](2x_3x_4+x_4^2)c_3\sin(x_2) \\
&\quad + c_2c_4g\sin(x_1) + c_3c_4g\cos(x_2)\sin(x_1) \\
&\quad - c_1c_5g\sin(x_1+x_2) - c_3c_5g\cos(x_2)\sin(x_1+x_2) \\
&\quad - [c_1+c_2+2c_3\cos(x_2)]x_3^2c_3\sin(x_2)\} \\
&\quad \gamma(x_2) = c_1c_2 - c_3^2\cos^2(x_2) \\
&\quad c_1 = m_1l_{c1}^2 + m_2l_1^2 + I_1 \\
&\quad c_2 = m_2l_{c2}^2 + I_2 \\
&\quad c_3 = m_2l_1l_{c2}^2 \\
&\quad c_4 = m_1l_{c1} + m_2l_1 \\
&\quad c_5 = m_2l_{c2}
\end{aligned}
$$

The parameters for the double pendulum are given in Table 6.2 [47]. We assume that $l_{c1} = l_1/2$ and $l_{c2} = l_2/2$.

System identification

Because of the sensitivity of the double pendulum to initial conditions, we need to collect "enough" training data to adequately represent most dynamic behaviors. In this experiment, we used 20 different initial conditions in the range $[-\pi/2, \pi/2]$ for the two angles q_1 and q_2. Also, angles q_1 and q_2 are forced to be in the range $[-\pi, \pi]$. We use a NARX network with no input, 2 outputs, 4 output delays, and 10 hidden neurons for this task.

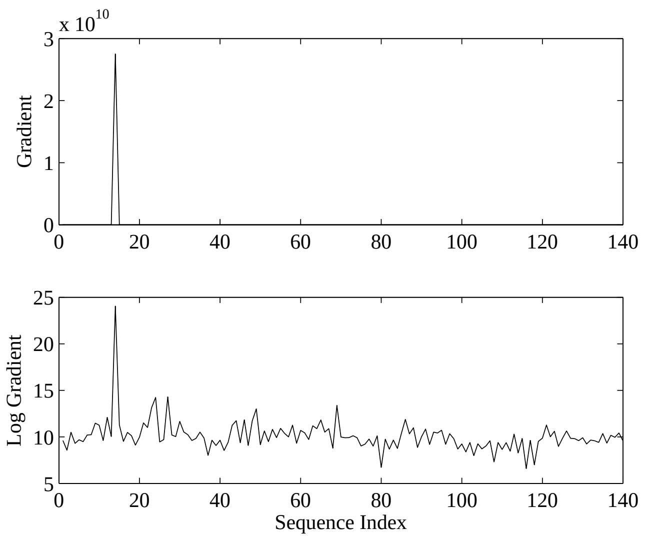


Figure 6.13: Gradient of individual sequences.

In the closed-loop training phase, μ reached μ_{max} quite often (see step 3 of the procedure in Section 5.4.2). Fig 6.12(a) shows an error profile (a cross section of the error surface along the gradient direction) when μ reached μ_{max}. Clearly, there are many valleys in the error surface. At this point, we were doing 65-step-ahead predictions. There were 140 subsequences (which were segmented from the 20 original sequences) in the training data pool. By checking the norm of gradients of individual sequences, we obtained the plot shown in Fig. 6.13. We can see that there is a sequence (sequence 14) whose gradient dominates the gradients of others. It turns out that this sequence contributes the most to the valleys in the error surface. By removing this sequence from the training data and plotting the error profile again at the same location, we obtained the new profile shown in Fig. 6.12(b). All the valleys disappear now. (Note that this new profile is very smooth, and may seem to be too good to be true. That is because we are plotting the profile over a very small region. Even in that small region, however, the original profile has many spurious valleys. This result is typical of the performance of the new proposed algorithm.) Because we have eliminated many of the spurious valleys, it will be easier for the algorithm to escape from this region.

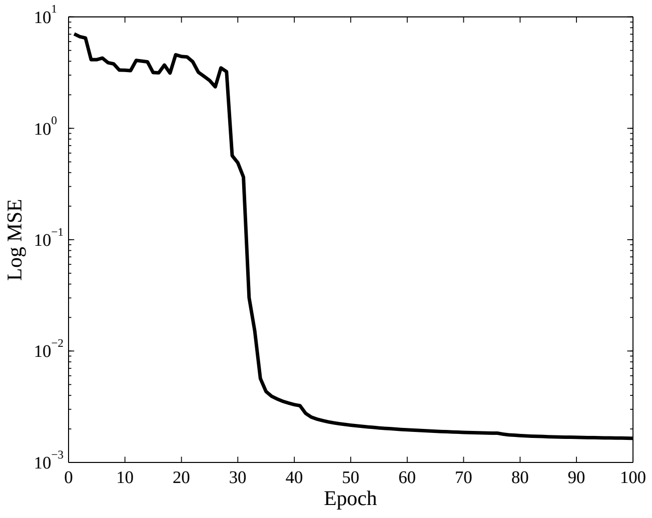


Figure 6.14: Training performance for the modified LM algorithm.

Table 6.3: Removing sequences for 65-step-ahead predictions

Epoch	4	6	7	9	10	15	17	19	25
No. seq	1	2	1	5	2	1	2	2	1

Fig. 6.14 shows the performance of the modified LM algorithm. We can see that the SSE increases at certain epochs. (In the original LM algorithm, the SSE always decreases.) This occurs when μ reaches μ_{max} and the algorithm gets trapped in a valley. Since some subsequences are removed at this point, the next step could produce weights where the SSE on the full data set does not decrease. The algorithm does escape from the valleys and eventually converges, as shown in Fig. 6.14. (Note that the standard LM algorithm would stop whenever μ reached μ_{max}. There would be no way to continue with additional iterations. The algorithm would be permanently stuck, and would almost always be stuck at a weight location where the network fit is very poor.)

Table 6.3 shows the number of sequences that were removed at certain epochs when we were doing 65-step-ahead predictions. In this training stage, μ reached μ_{max} 9 times, and the algorithm had to remove at most 5 (out of 140) subsequences to

Table 6.4: Removing sequences for different prediction horizons

k	Total no. sequences	No. times $\mu \to \mu_{max}$	Max no. sequences removed
27	360	1	1
55	180	3	2
65	140	9	5
75	120	1	1
95	100	2	1
125	60	1	1
145	60	16	1
150	60	21	1
155	60	31	2
160	60	19	3
250	20	4	3
300	20	13	1
497	20	1	1

escape from spurious valleys. Table 6.4 summarizes the operation of the modified LM algorithm at different stages (different prediction horizons) when the search encountered spurious valleys. The table shows how many times μ reached μ_{max} and how many sequences (at most) were removed.

The trained network responses for 2 out of 20 sequences in the training data are shown in Fig. 6.15. We can see that even for a chaotic system, the 497-step-ahead prediction results are quite good. (Since we were using batch training, we were able to achieve a good fit for all 20 sequences.) The performance for a testing sequence is shown in Fig. 6.16. The network outputs still track the actual outputs quite well even for a new sequence (with new initial conditions).

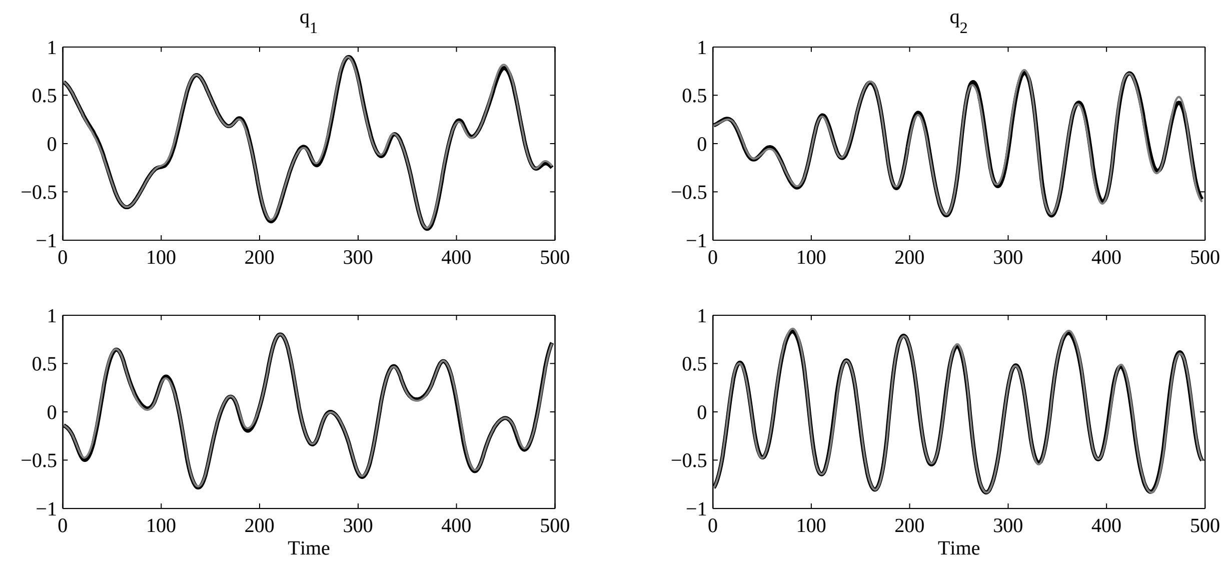


Figure 6.15: Training results for the double pendulum (Actual outputs - thick, black and network outputs - thin, gray).

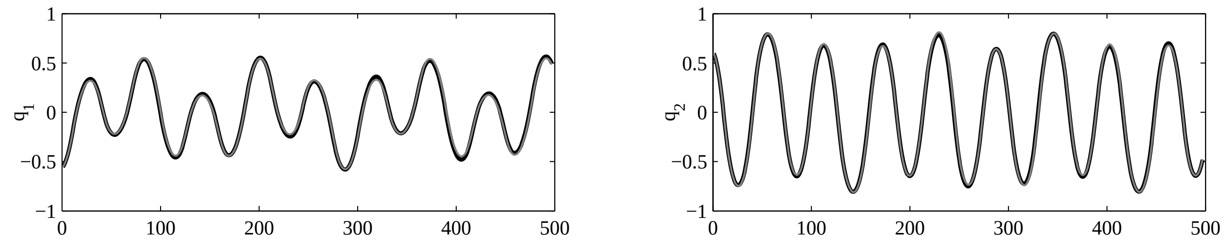


Figure 6.16: Testing results for the double pendulum (Actual outputs - thick, black and network outputs - thin, gray).

CHAPTER 7

A COMPARISON BETWEEN STOCHASTIC TRAINING AND BATCH TRAINING

7.1 Stochastic training and batch training: An overview

There are two main schemes that have been used to train neural networks: stochastic training and batch training. In stochastic training (sometimes called incremental, online or adaptive training), the weights are updated after each training example is presented, while in batch training, the weights are updated after the entire training data set is presented.

There are conflicting views in the neural network community on whether stochastic or batch training is "better". Some researchers claim that stochastic training is faster and produces better results than batch training. Others state that since batch training uses the true gradient direction for its weight updates, it is more efficient and also faster. A survey of these two views can be found in [48]. There has been some work that compares these two schemes, both empirically [48] and theoretically [49]; however, these comparisons were just for the standard steepest decent algorithm which is not a fast and efficient one.

In the stochastic training scheme, we would not expect to have problems with spurious valleys in the error surface, since different training data is used at each iteration. This would correspond to the most extreme case of the "switching sequences" strategy for avoiding the spurious valleys.

Our practical experience of training neural networks shows that certain batch algorithms such as the Levenberg-Marquardt algorithm (Chapter 5) can provide more accurate and stable training than stochastic algorithms. Therefore, the analysis of spurious valleys shown in the previous chapters is important. By understanding the spurious valleys, we can modify batch algorithms to avoid them.

In the next section, we will describe the extended Kalman filter (EKF), which is considered one of the best stochastic training algorithms for neural networks. We will then compare the performance of the EKF with the LM algorithm in Section 7.3.

7.2 NN training with extended Kalman filter

The EKF has served as the basis for a number of neural network training algorithms. It was used to train multilayer feedforward networks in [50]. In [51], the authors used the EKF to train recurrent neural networks, and later in [52], they made some modifications to speed up the training algorithm.

The NN training problem can be considered as a parameter estimation problem, where the network weights are estimated for a given set of inputs and targets. The training problem is formulated as a weighted least squares minimization problem, where the error vector $\mathbf{e}$ (with the length of N - the number of output neurons) is the difference between the network outputs and the targets. The cost function is the mean squared error. The weights are arranged into a M-dimensional vector $\mathbf{w}$, and the estimate of the weight vector at time step n is denoted by $\hat{\mathbf{w}}(n)$. The EKF algorithm requires, in addition to the updating of the weights, the updating of an approximate error covariance matrix $\mathbf{P}(n)$. The update equations for the EKF algorithm are

$$\begin{aligned}
\mathbf{A}(n) &= \left[\mathbf{R}(n) + \mathbf{H}(n)^T\mathbf{P}(n)\mathbf{H}(n)\right]^{-1}, \\
\mathbf{K}(n) &= \mathbf{P}(n)\mathbf{H}(n)\mathbf{A}(n), \\
\hat{\mathbf{w}}(n+1) &= \hat{\mathbf{w}}(n) + \mathbf{K}(n)\mathbf{e}(n), \\
\mathbf{P}(n+1) &= \mathbf{P}(n) - \mathbf{K}(n)\mathbf{H}(n)^T\mathbf{P}(n) + \mathbf{Q}(n). \qquad (7.1)
\end{aligned}$$

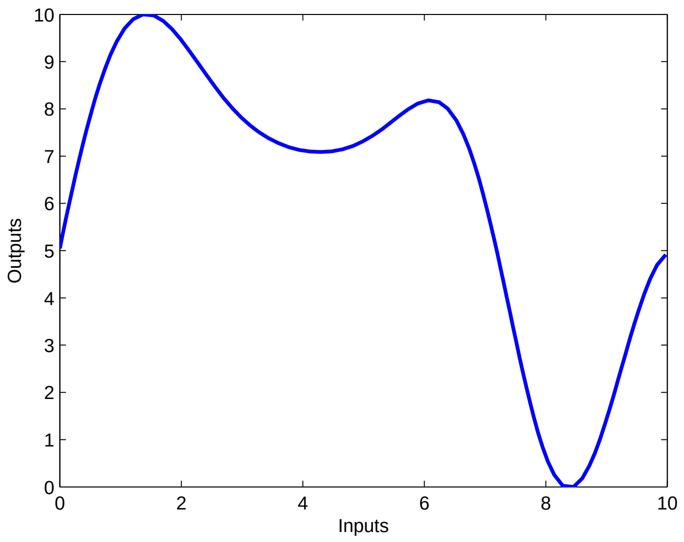


Figure 7.1: Simple fitting data

where $\mathbf{R}(n)$ is associated with the learning rate (or forgetting factor), $\mathbf{H}(n)$ is the Hessian matrix (M-by-N) - the partial derivatives of the outputs with respect to the weights - which can be calculated using the backpropagation procedure, $\mathbf{K}(n)$ is the Kalman gain matrix, and $\mathbf{Q}(n)$ is a diagonal matrix that incorporates the process noise and helps to avoid numerical divergence of the algorithm. The initial conditions $\mathbf{w}(0)$ are small random values. $\mathbf{R}(n)$ is initialized as a diagonal matrix whose diagonal components are equal or slightly less than 1. $\mathbf{P}(0)$ is a diagonal matrix with large diagonal components (e.g., $O(10^2)$), and $\mathbf{Q}(n)$ is a diagonal matrix with small diagonal components, (e.g., $O(10^{-2})$).

We can see in (7.1) that the Jacobian matrix is computed, and the weights are updated, after each input is presented. This makes the EKF a stochastic training algorithm.

7.3 Our comparison: Simulation results

In this section, we will compare the performance of the EKF algorithm and the LM algorithm on a variety of function approximation (system identification) problems.

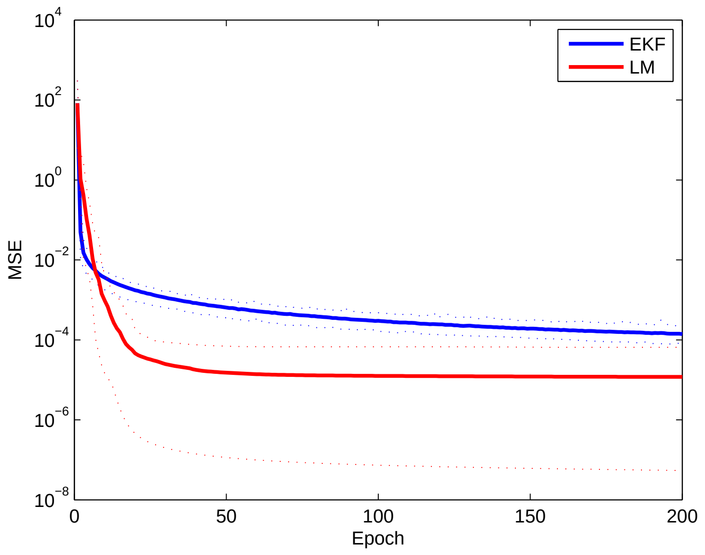


Figure 7.2: Comparison of convergence on simple fitting data

There are five data sets used for this comparison, both simulated data and real data, with different complexity and number of data points in the training sets. The first four data sets are static, and we will use feedforward networks to fit these data. The last data set is dynamic, and we will use a recurrent network (a NARX network) to build a model for this data. For each data set, we run 30 different trials, where different random initial weights were used in each trial, and then we took the average of the MSE over all trials.

7.3.1 Simple fitting data set

This is a simple function approximation data problem. A feedforward network with 1 input, 10 hidden neurons and 1 output neuron, with tansig transfer functions in the hidden layer and a linear transfer function in the output layer (a 1-10-1 network), was used to approximate the function shown in Fig. 7.1, with 94 data points in the training set. The MSE for both algorithms (the average (bold), the upper bound and the lower bound (dotted) over the 30 trails) is shown in Fig. 7.2. We can see that in this case, the LM algorithm produced smaller final errors than the EKF

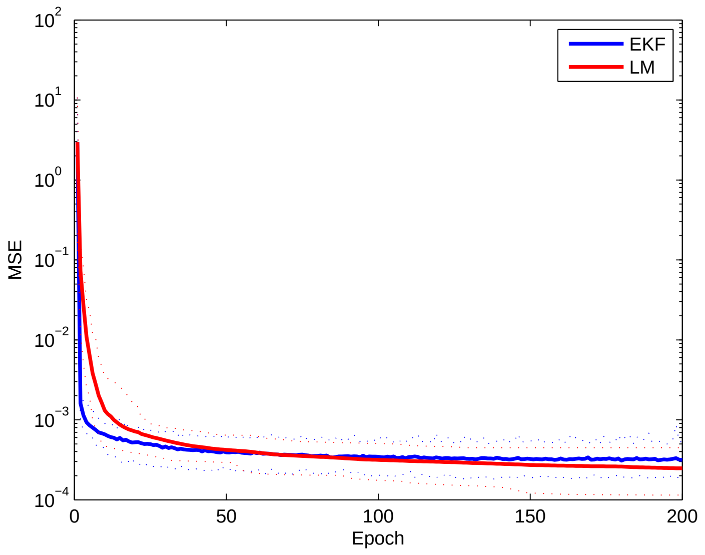


Figure 7.3: Comparison of convergence on N-P data

algorithm. (We did not consider the problem of overfitting in our experiments. We just considered the convergence accuracy.)

It should be noted that an epoch represents the presentation of all inputs in the training set. For stochastic algorithms, like the EKF, this would correspond to many iterations (weight updates). However, for batch algorithms, like the LM, an epoch corresponds to only one iteration. This makes it some what difficult to infer rates of convergence from Fig. 7.2. When the training set is large, the EKF might require many more computations in each epoch than the LM algorithm.

7.3.2 Narendra and Parthasarathy's static data set

We will compare the function approximation capabilities of the two algorithms for the problem given in [53]. This test problem was originally proposed by Narendra and Parthasarathy in [46]. The dynamics of the plant are described by the nonlinear

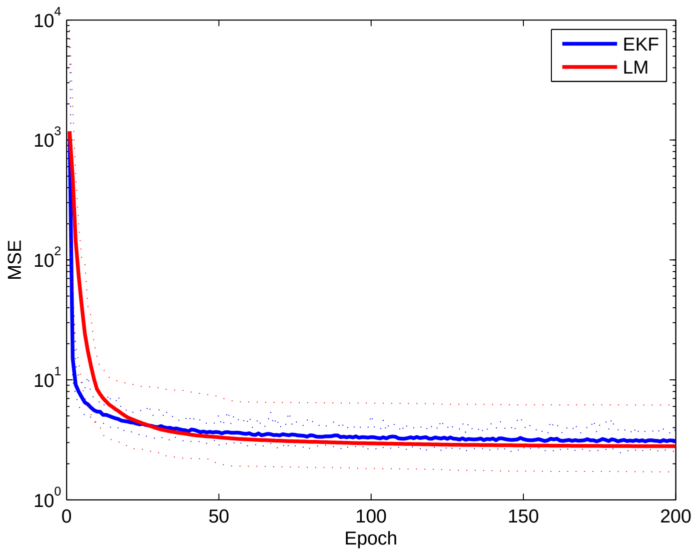


Figure 7.4: Comparison of convergence on house value data

equations

$$\begin{aligned} y_p(n+1) &= f\left[y_p(n), y_p(n-1), y_p(n-2), u(n), u(n-1)\right], \\ f\left[x_1, x_2, x_3, x_4, x_5\right] &= \frac{x_1 x_2 x_3 x_5 (x_3 - 1) + x_4}{1 + x_3^2 + x_2^2}, \end{aligned} \tag{7.2}$$

where $y_p(n)$ represents the outputs of the plant and $u(n)$ is the command input at time step n. The input $u(n)$ consisted of a 1000 random points uniformly distributed in the interval $[-1, 1]$. A 5-10-1 network was used. (The 5 inputs are 2 time-delayed command inputs and 3 time-delayed recurrent plant outputs.) The MSE for both algorithms is shown in Fig. 7.3. The LM algorithm provides smaller final errors.

7.3.3 House value data set

This data set can be used to train a neural network to estimate the median house price in a neighborhood based on neighborhood statistics such as tax rate, pupil/teacher ratio in local schools, crime rate, etc. (there are such 13 input variables and a total of 506 input/target pairs in the training set). A 13-10-1 network was used. The plot of MSE for both algorithms is shown in Fig. 7.4. This is a noisy data

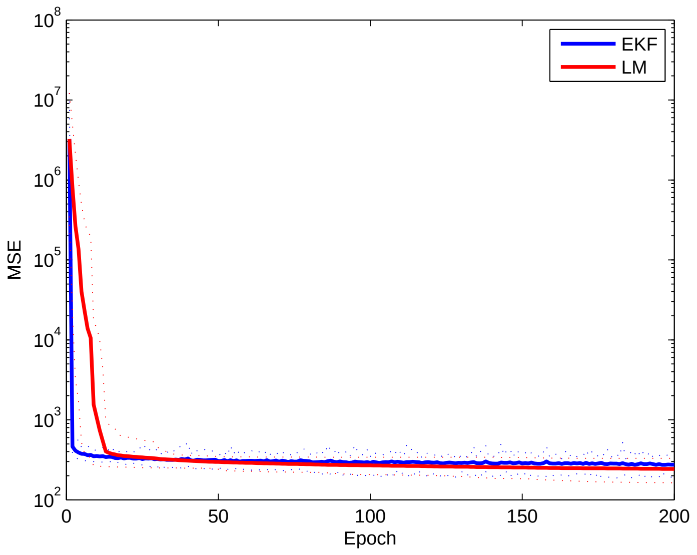


Figure 7.5: Comparison of convergence on engine behavior data

set, so we could not expect the MSE to go down as low as in the two previous data sets, but that is the best both algorithms could do. Again, the LM algorithm can provide smaller final errors compared to the EKF algorithm.

7.3.4 Engine behavior data set

This data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and nitrous oxide emission levels. A 2-10-2 network was used, and the training data set contained 1199 input/target pairs. The plot of MSE for both algorithms is shown in Fig. 7.5. A similar trend as in the house value data set can be seen here. The LM algorithm provides smaller errors at the end.

7.3.5 Magnetic levitation data set

In Chapter 6, we developed a model and trained a controller for the magnetic levitation system using the LM algorithm. Now we will try to use the EKF algorithm to fit a model for this system and compare its performance with the LM algorithm.

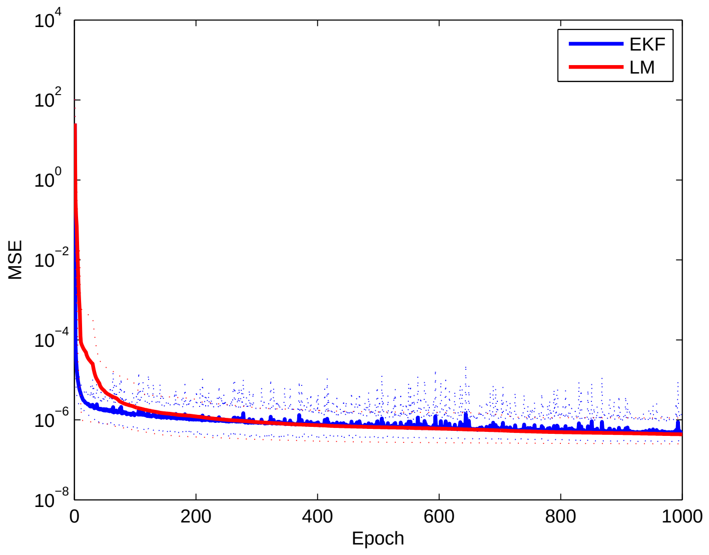


Figure 7.6: Comparison of convergence on magnetic levitation data

A NARX network with 2 input delays, 2 feedback delays and 10 hidden neurons was used, and the training data had 4001 targets. We tested the open loop training, which is a crucial step in the training procedure described in Chapter 5. The plot of MSE for both algorithms is shown in Fig. 7.6. The LM algorithm provides smaller errors.

Summary The LM algorithm generally produced smaller final errors in the tests that we ran. In the case that very accurate training is required, such as in open loop training step of the training procedure in Chapter 5, this is an advantage. In addition, in terms of parameter configuration for training, the LM algorithm is more autonomous. There is no need to adjust any parameter in the LM algorithm, while the selection of matrices R, P, and Q in the EKF algorithm can be problem dependent.

7.4 Conclusion

In a comparison of one of the best stochastic algorithms - the extended Kalman filter - with one of the best batch algorithms - the Levenberg-Marquardt algorithm - we have shown that the LM algorithm can produce more accurate and stable training than the EKF. This demonstrates the importance of understanding the spurious valleys, so that batch algorithms can be modified to avoid them. Stochastic algorithms, like the EKF, should avoid the valleys, sine the input sequence is effectively changed at each iteration. However, batch algorithms can produce smaller final errors. In Chapter 5 we introduced several procedures for mitigating the effects of spurious valleys on batch training algorithms.

CHAPTER 8

A NEURAL CONTROLLER WITH DISTURBANCE REJECTION

One of the concerns about neural network controllers (like the MRAC described in Chapter 6) is disturbance rejection. We want to train controllers that can deal with random disturbances that are input to the plant. This requires a modification of the standard MRAC structure. This chapter will review the classical method for disturbance rejection, propose our new disturbance rejection control scheme and apply that scheme for controlling a practical physical system.

8.1 Classical disturbance rejection method

The traditional method for disturbance rejection in linear systems is to make the controller gain large. Consider the control system shown in Fig. 8.1 with plant P, controller C, reference input r, output y, disturbance d and tracking error e. We then have the following relation between the disturbance d and the output y (set $r = 0$):

$$y_d = \frac{P}{1 + PC} d \tag{8.1}$$

If the gain of the controller C is large, y_d is small. Therefore, by making the controller gain large, we can reduce the effect of disturbances [54]. However, making the gain too large could move the closed loop poles to the unstable region. Therefore, the optimal gain is a compromise between the effect of disturbance rejection and the system stability.

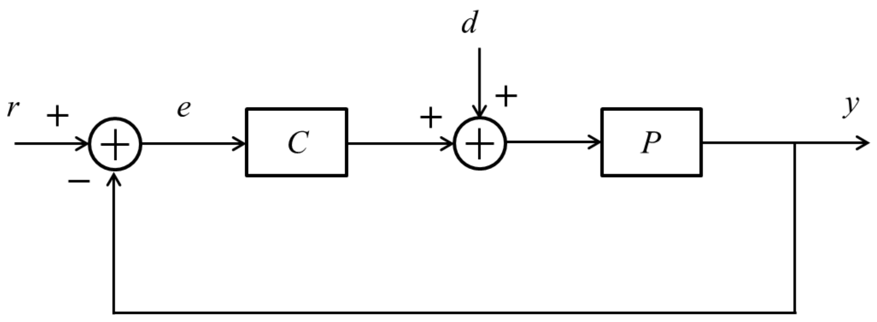


Figure 8.1: A system with a disturbance

8.2 MRAC with disturbance rejection

The control scheme for disturbance rejection is illustrated in Fig. 8.2. Compared to the standard MRAC structure (Fig. 6.3), the only difference is that the disturbance signal is added to the plant input.

In this disturbance rejection scheme, the plant is identified first as in the standard MRAC structure. Next, the controller is trained. The training data for the controller is obtained in the same manner described in Chapter 6, except that disturbances are injected into the plant input. The NN controller is trained so that the plant output follows the reference model output even in the presence of disturbances. By maintaining the desired output, the trained controller has the ability to reject disturbances.

8.3 Experimental results

In this section, we apply the control scheme proposed in Section 8.2 for controlling a practical system: the blade position of a track-type tractor. The control objective is to achieve a desired performance in the presence of some kinds of disturbances. We will begin by building a plant model (disturbance free), and then we will train a controller for disturbance rejection.

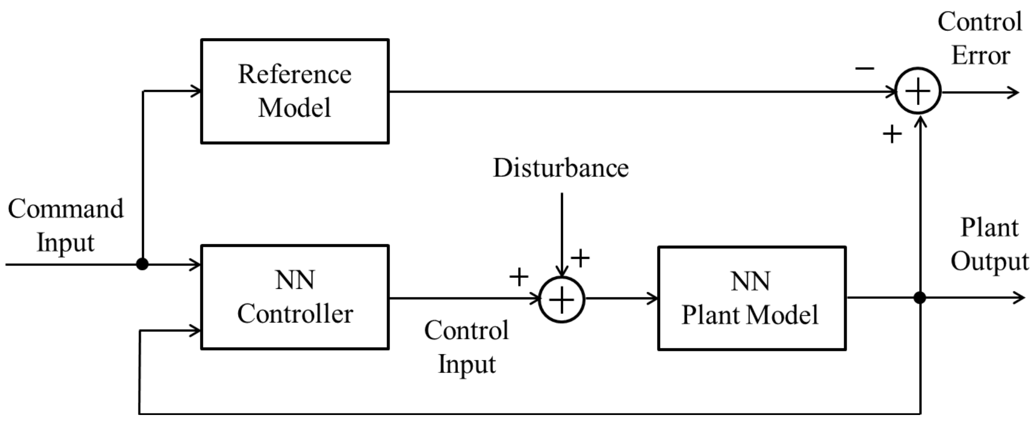


Figure 8.2: MRAC with disturbance rejection

8.3.1 Plant model training

Plant description

The primary working tool of a track-type tractor is a blade, affixed to the front and controlled by several hydraulic arms (and an optional ripper in the back) (Fig. 8.3). The blade is mainly intended for earthmoving, such as pushing up sand, dirt, or other such material during construction.

Sensors (laser or GPS based) are mounted on the machine to calculate the blade position (elevation) and/or blade velocity (slope). The objective is to design a controller that uses the information from the sensors to automatically adjust the blade via the hydraulic valves to maintain grade. The controller needs to efficiently maintain accurate blade position while maintaining operator comfort (by reducing the oscillation of the blade, for example). A PD controller is typically used, however, to improve the performance, some kinds of advanced nonlinear control, such as neural control, need to be examined.

Training data

The plant input is the commanded rate to the hydraulic actuator. (There is a nonlinear relationship between the commanded rate and the percentage that the

Figure 8.3: Track-type tractor

hydraulic valve is opened.) The output of the plant is the position of the blade. We collected two types of data. In one type, the data was collected open loop, and in the other type, the data was collected closed loop. (The data used for our tests was provided by Caterpillar Inc. from their proprietary simulation of a production track type tractor.)

a) Open loop data: For open loop data, a random commanded rate (a skyline signal) was applied to the plant, and the blade position was collected. The commanded rate (which determines the hydraulic valve opening) ranges between -306 and 264 mm/s. We collected four sets of open loop data using four different skyline signals. They are shown in Fig. 8.4. A magnification of the last portion of the fourth set is shown in Fig. 8.5. We can see that when the commanded rate is held constant, the velocity of the blade becomes constant. This shows that there is a pure integrator in the plant.

By collecting data in the open loop configuration, we have more control over the frequency content of the valve opening. However, because of the pure integrator in

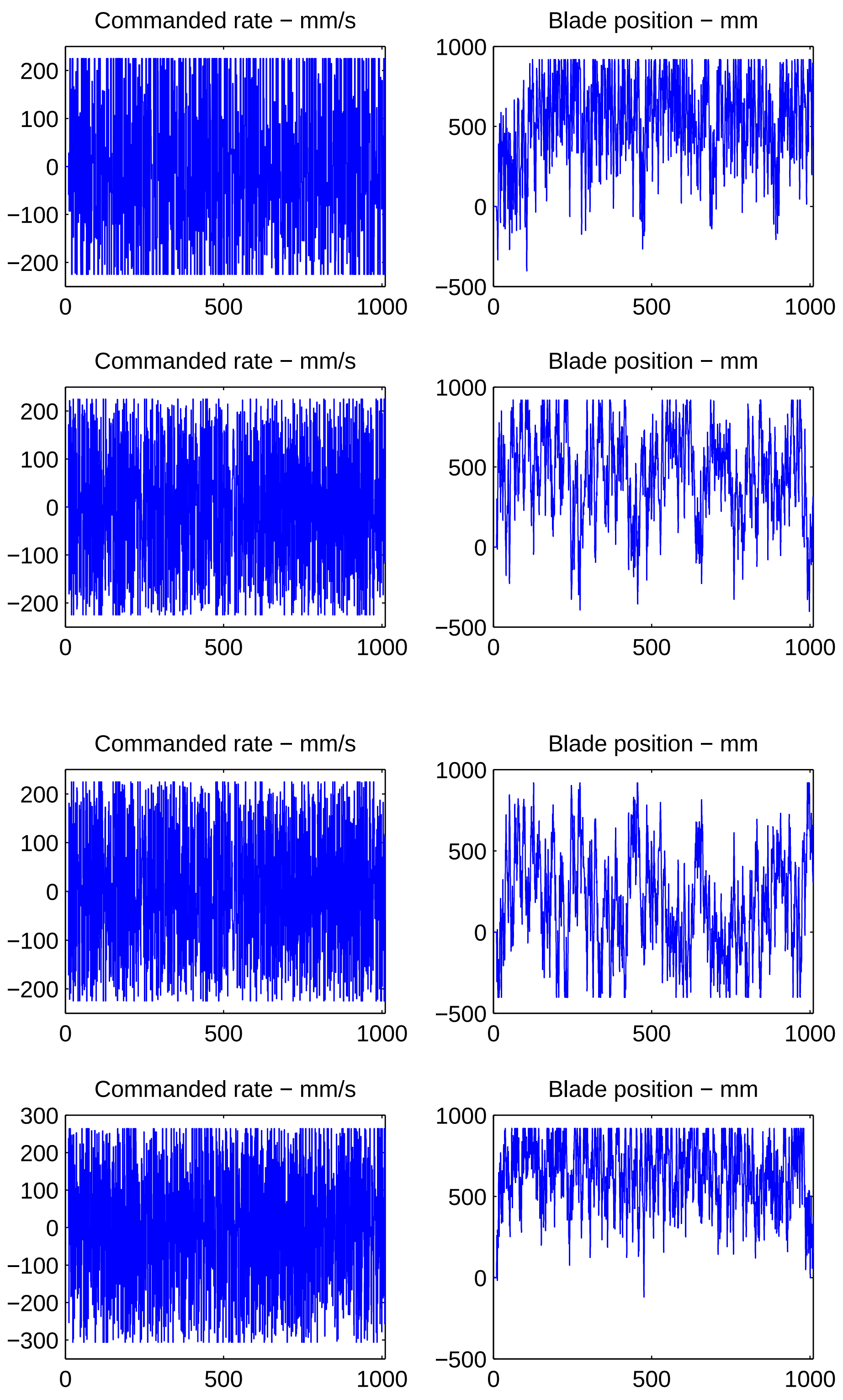


Figure 8.4: Open loop data

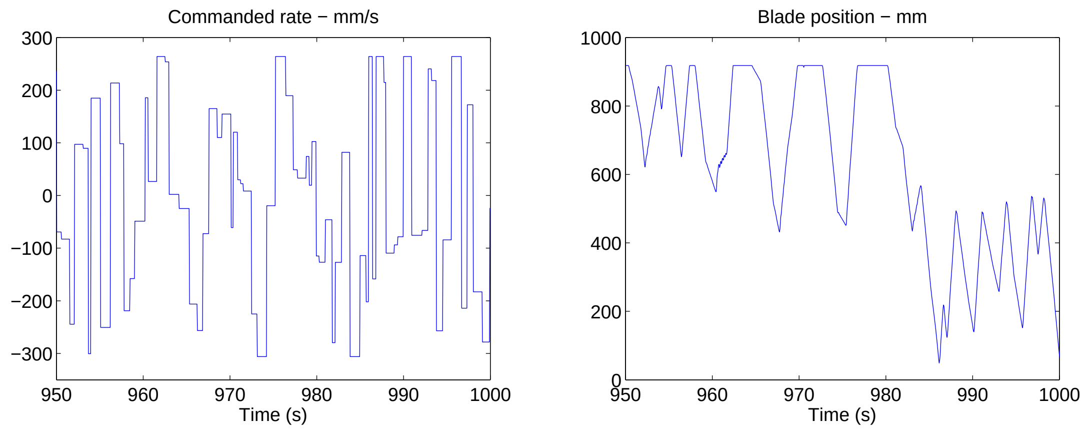


Figure 8.5: Open loop data (magnified)

the plant, there is more drift in the blade position, so the blade is more likely to hit the limits of its travel, as seen in Fig. 8.4. By contrast, if we also collect data in the closed loop configuration, the frequency content of the valve opening is limited, but we can collect more data with the blade position as shown below.

b) Closed loop data: For the closed loop data, a random reference position (another skyline signal) was applied to the control system (a Caterpillar-designed PD controller was used), and the plant input and output were collected. We collected two sets of closed loop data using two different skyline signals for the reference positions. They are shown in Fig. 8.6. A magnification of the last portion of the second set is shown in Fig. 8.7.

We can see that some values of the commanded rate at the output of the controller are beyond the limits of the actuator. These values are saturated to get the actual commanded rates. (Later, when we train a controller, we will need to add a saturation block to the controller to make sure that its output is within proper limits.) Also, notice that the plant output oscillates around the reference signal. This could be improved by using a neural controller.

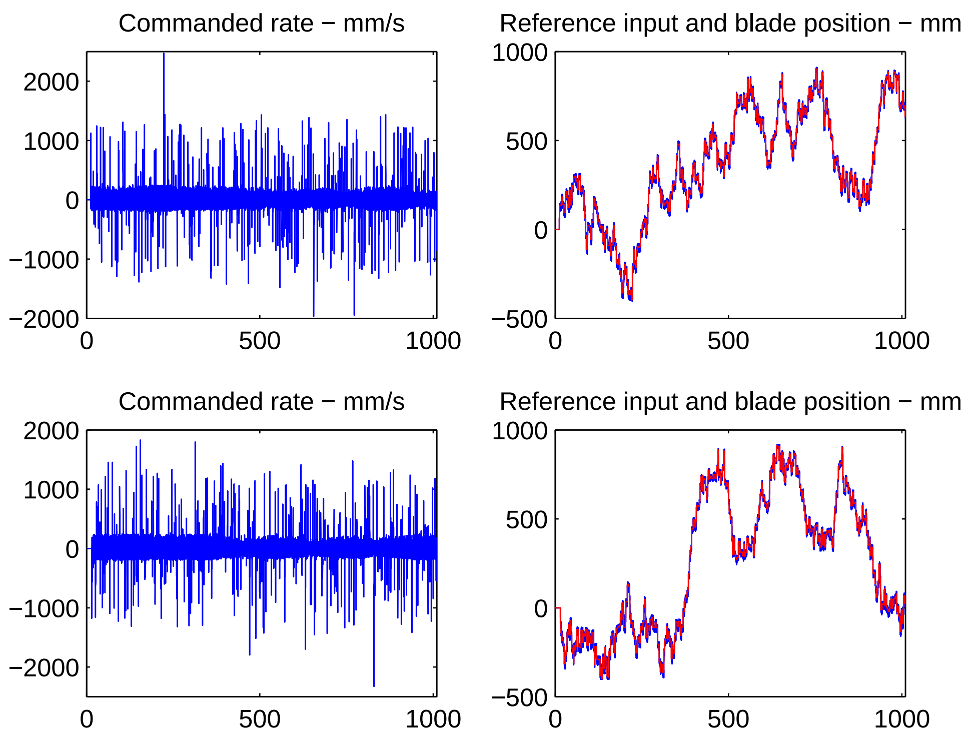


Figure 8.6: Closed loop data

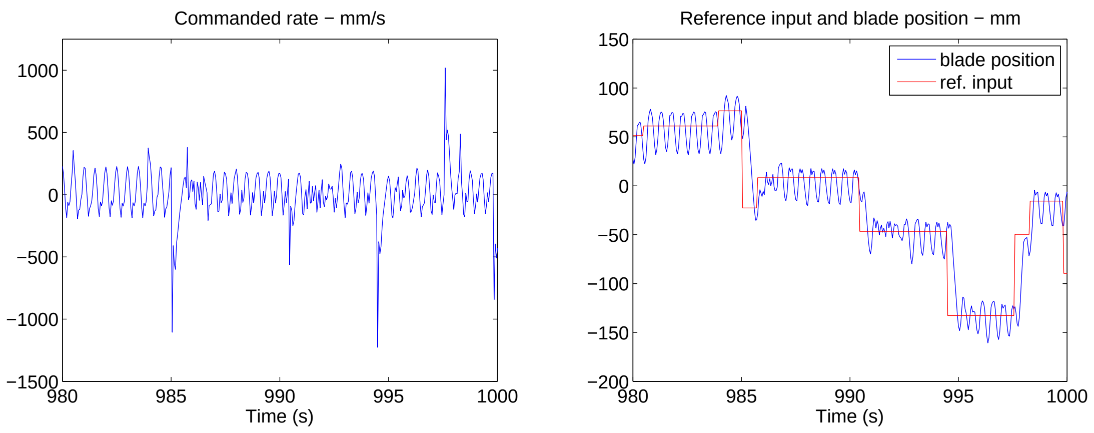


Figure 8.7: Cloosed loop data (magnified)

Network structure: NARX net with an integrator

Since there is an integrator in the plant, we will add an integrator to the NARX network in Fig. 6.2. The modified NARX network is shown in Fig. 8.8, in which the third layer is the integrator. The second layer output will be the blade velocity, and

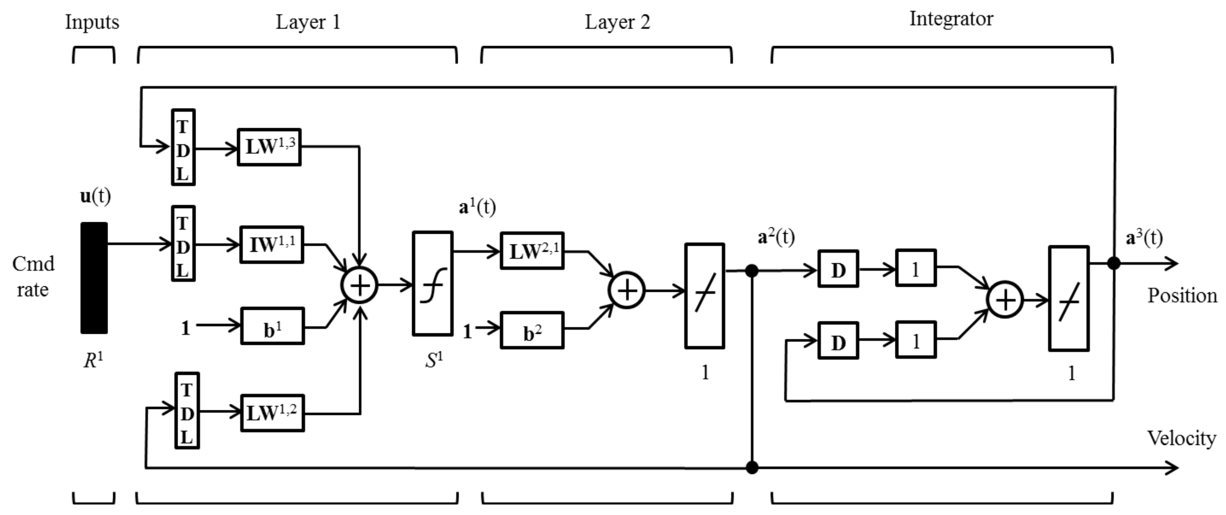


Figure 8.8: NARX with an integrator

the third layer output will be the blade position. They are both used as the network outputs and are fed back to the first layer. We trained the network so that the network blade position matched the desired position and the network blade velocity matched the derivative of the desired position.

Performance index

Since we want to fit both the position and the velocity of the blade, the performance index will be the sum of the mean squared errors of both position and velocity.

Training results

We followed the training procedure introduced in Chapter 5 to train this network. Fig. 8.9 shows the fitting results for one open loop data sequence and one closed loop data sequence (both position and velocity). This figure shows only the last 100 seconds of the sequences (the last 10 seconds of the closed loop velocity sequence) in order to make the details of the fitting accuracy clear. Note that at this point, the network is doing a 20,000-step ahead prediction. (We used a sampling time of 0.05 seconds.) It is remarkable that the network could match even the small scale

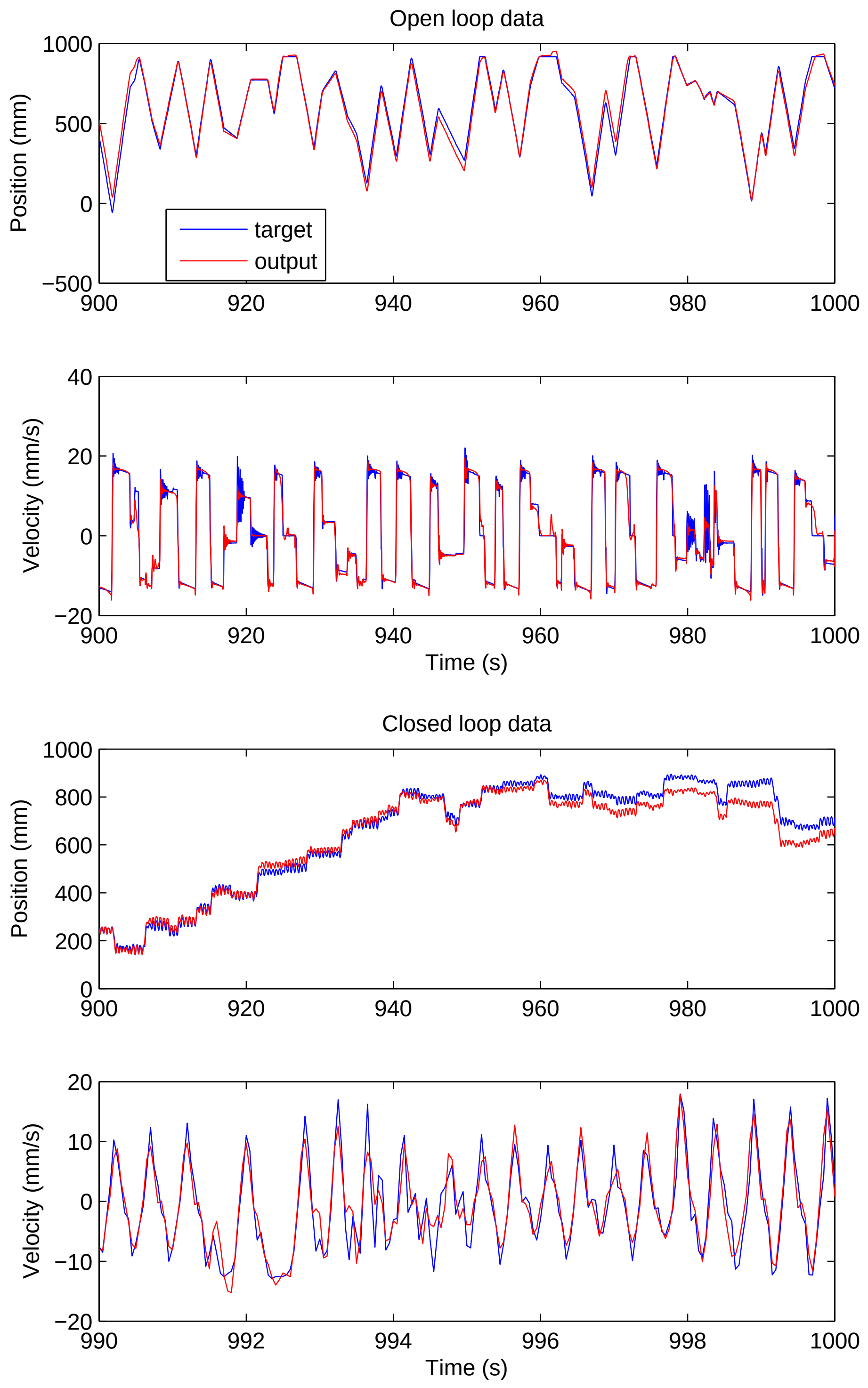


Figure 8.9: Plant model fitting

oscillations of the blade at such large prediction horizons. We obtained similar fitting results for other parts of the sequences in Fig. 8.9 as well as other sequences in the

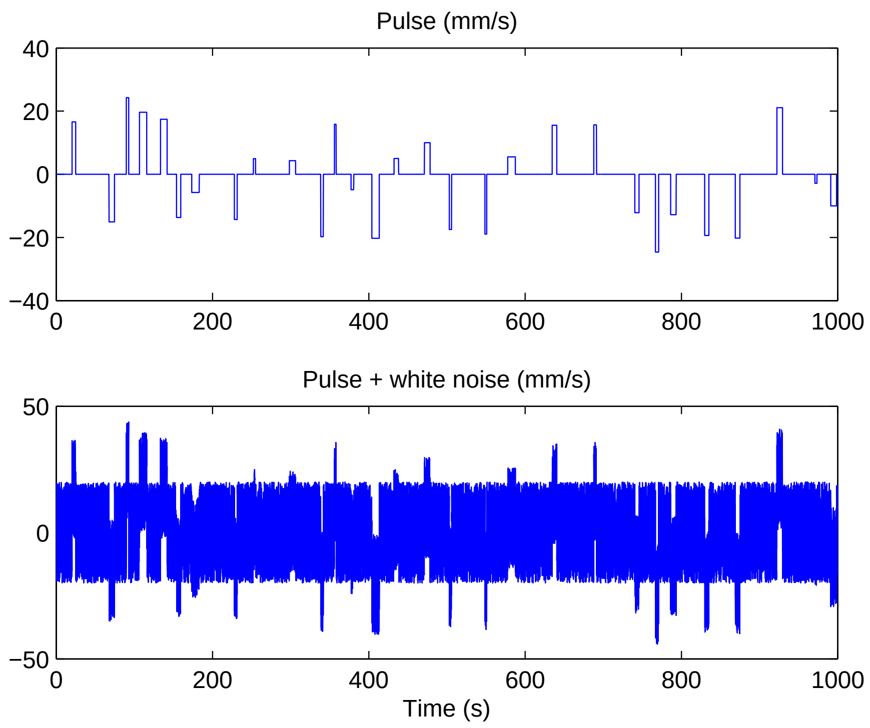


Figure 8.10: Disturbance

training data.

8.3.2 Controller training

We are now ready to train a neural controller that can reduce the effect of disturbances.

Training data

a) Reference input: For the reference model input, we used the same two skyline signals used to collect the closed loop data for plant training. (It could be any other skyline signal that covers the whole range of the blade position.). We then used a second order system to generate the reference model output (the desired blade position). As in the plant training, we took the numerical derivative of the desired position to get the desired velocity.

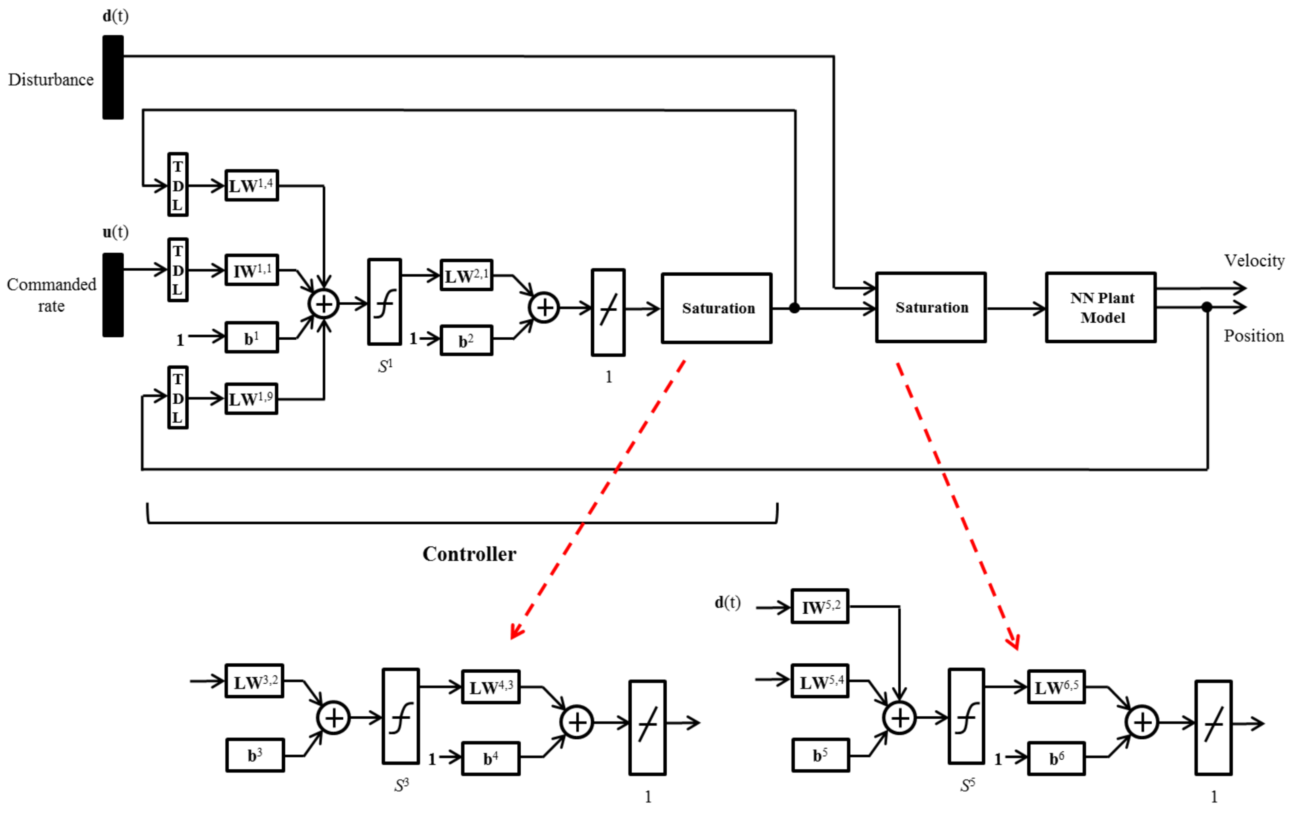
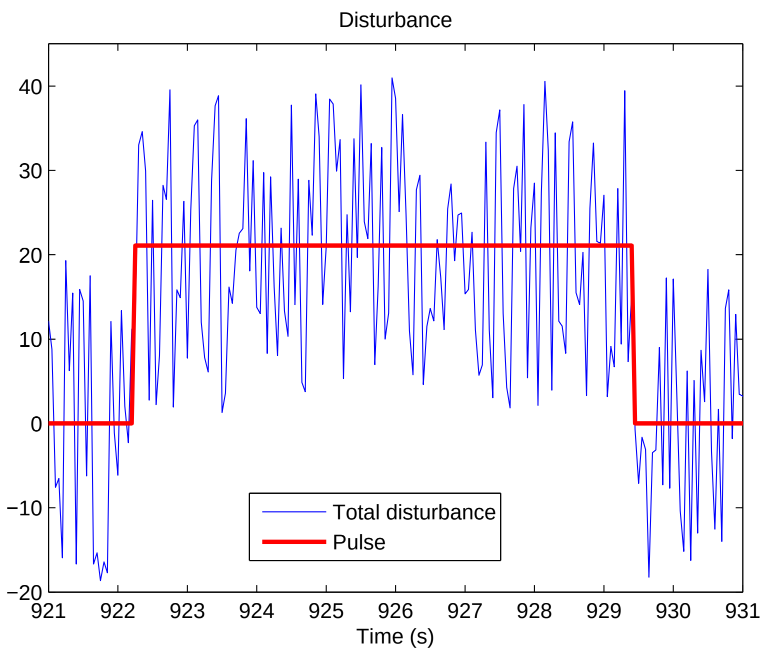


Figure 8.11: MRAC network with two saturation blocks, and a disturbance input

b) Disturbances: The second input to our control system is the disturbance. One common source of disturbances acting on the plant is external forces. In the case of the track-type tractor, in addition to the hydraulic forces, there might be other forces acting on the blade, for example when the blade hits a hard rock. To represent these disturbances, we use a combination of pulses and white noise. The pulses have different heights and widths and appear randomly at several places though the course of training. The pulses are combined with a white noise sequence. An example of the disturbance is shown in Fig. 8.10.

Network structure: MRAC

There are several modifications made to the MRAC system shown in Fig. 8.11 compared to the standard one in Fig. 6.4. A saturation block is added to the controller



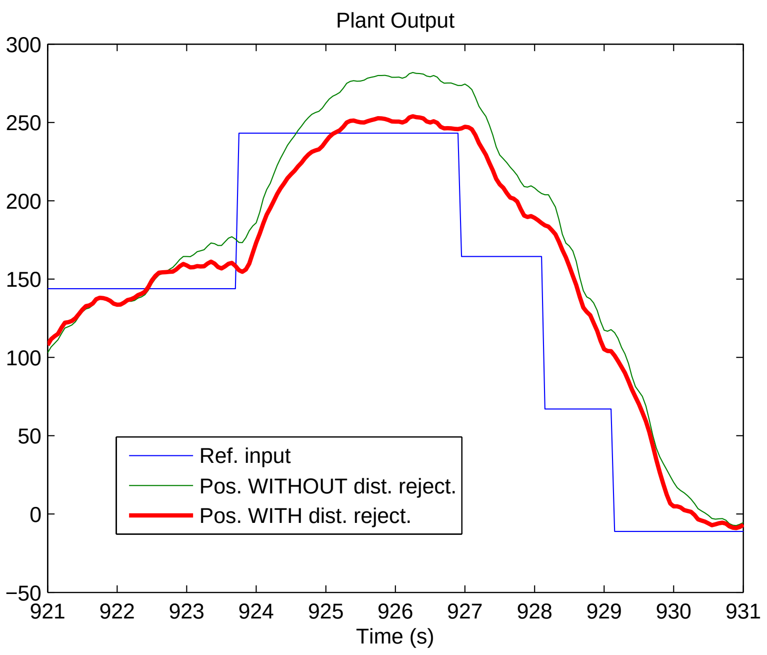


Figure 8.12: Disturbance rejection for training data

because of the limits on the commanded rate. We pre-trained a two layer feedforward network to approximate the saturation, and the weights of this network were fixed during controller training. The disturbance is a second input to the MRAC network. It is added to the output of the controller. A second saturation block is used to

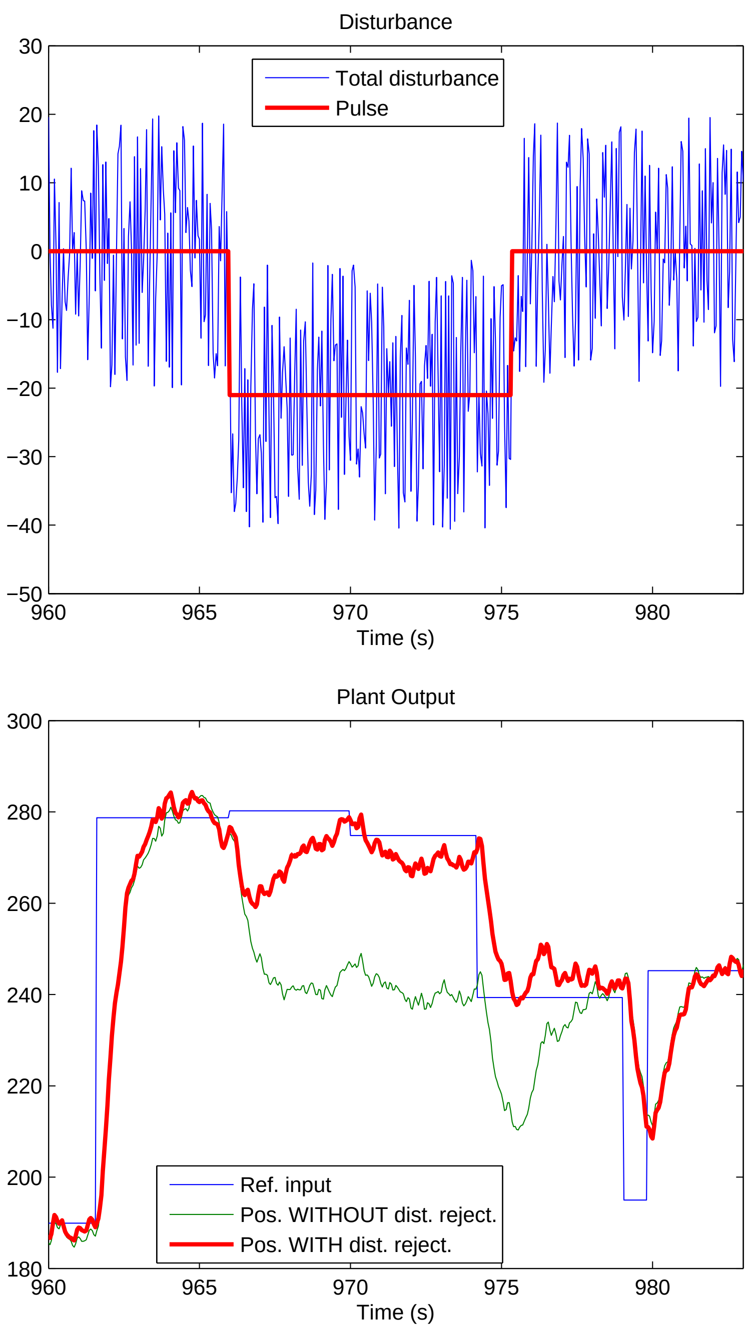


Figure 8.13: Disturbance rejection for testing data

make sure the sum of the controller output and the disturbance is in the range of the commanded rate. The weights and biases of the first two layers of the controller are trained. The rest of the network (the plant model and two saturation blocks) is fixed during training.

Performance index

Since we want to fit both position and velocity of the blade, we will use the same performance index for the controller training as the one for plant training.

Training results

Fig. 8.12 shows the training results for two controllers. One controller was trained without disturbance injection to the plant. The other controller was trained with disturbance injection. The result for 10 seconds of the second training sequence is shown. There is a pulse that appears in this interval. We can see that the controller with disturbance injection can significantly reduce the effect of the pulse. The ability to reject the pulses is more clear than that for rejecting the white noise. The overall effect of disturbance rejection can be measured by calculating the root-mean-square error (RMSE) between the plant output and the desired output. The RMSE for the controller without disturbance rejection is 15.4 mm. The RMSE for the controller with disturbance rejection is 12.7 mm which is about 17.5% less. We also see that there is no oscillation in the plant output. The nonsmoothness is caused by the white noise in the disturbance.

Testing results

The performance of the controller for a new testing sequence is shown in Fig. 8.13. The controller can still reduce the effect of the pulse quite well. The RMSEs for the controller without and with disturbance rejection are 15.6 mm and 12.9 mm, respectively, which about a 17.3% reduction.

8.4 Summary

In this chapter, we introduced a new method for disturbance rejection for nonlinear systems using neural network based MRAC. A plant model (disturbance free) was identified first, and then the disturbances were injected into the plant during controller training. We tested the control scheme for a practical system: a track-type tractor blade controller. The simulation results showed that our controller can reduce the effect of the disturbance. The results also verify the performance of the training procedure proposed in Chapter 5 for a real data set.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

There are two proposed reasons why training RNNs is difficult - dynamic bifurcations and long-term dependencies (Chapter 2). We have studied one more reason - the existence of spurious valleys in the error surface of RNNs. We have generalized the results of [18], which described mechanisms by which spurious valleys are introduced into the error surface of a first order recurrent network (Chapter 3). These valleys are spurious, because they are unrelated to the problem that we are attempting to solve with the RNN. They do not depend to any significant extent on the desired network output. They depend only on the network inputs, the initial network states and the architecture of the network.

We have used some basic concepts of linear system theory to develop a framework that can be used to analyze spurious valleys that appear in most practical recurrent network architectures, no matter their size (Chapter 4). We have shown that there are two main types of spurious valleys in the the error surface of RNNs: root valleys and architecture valleys. Root valleys depend on the input sequence and can be explained using some properties of random polynomials and concepts from linear systems. Architecture valleys appear independently of the input sequence and are fundamental characteristics of the network architecture. We have derived approximate equations for the valleys for a very general class of RNN and have confirmed the equations experimentally for second order networks.

The presence of spurious valleys in the error surface of RNNs makes training, especially using batch, gradient-based methods, very difficult. Using the knowledge of the the spurious valleys, we have proposed a new procedure for efficiently training general RNNs (Chapter 5). The new procedure is a modified version of the Levenberg-Marquardt batch training algorithm. This procedure helps to avoid the numerous spurious valleys that appear in the error surface of recurrent networks. There are two features of the proposed procedure. First, the procedure begins with short sequences (short prediction horizons) and then increases the sequence length during the training process. Second, the procedure detects when a search enters a spurious valley and removes sequences in the data set that cause the valley. We have tested the procedure for two physical systems (Chapter 6). In the first test, we developed a precise NARX model for a magnetic levitation system and trained an accurate MRAC neural controller. The second test dealt with a more difficult task, in which we trained a NARX model for a chaotic system - the double pendulum. We have shown in this task how the modified LM algorithm can handle the valleys in the error surface to achieve convergence. We believe that the problem of spurious valleys is one of the key difficulties that recurrent network training has to overcome. We suggested a possible solution to this problem.

We have implemented the dynamic backpropagtion for a very general class of neural networks using a combination of MATLAB and C/C++ programming, and then applied it to implement various training algorithms, both batch and stochastic. We compared one of the best stochastic algorithm - the extended Kalman filter - with one of the best batch algorithm - the Levenberg-Marquardt algorithm - on a variety of test data sets (Chapter 7). The LM batch algorithm has some advantages over the EKF stochastic algorithm. This shows that the understanding of the spurious valleys (Chapter 3 and Chapter 4) is important, so that the LM batch algorithm could be modified to mitigate the spurious valleys, as we have done in Chapter 5.

We have also verified our training procedure for some practical data that include disturbances. We introduced a new method for disturbance rejection using neural network based MRAC (Chapter 8). A plant model (disturbance free) was identified first, and then the disturbances were injected into the plant during controller training. We then tested our control design and training procedure for modelling and controlling a track-type tractor from Caterpillar Inc. The experimental results showed that our disturbance rejection scheme helps reduce the effect of disturbances.

Neural networks have been shown to be a good choice for modelling and controlling systems that are highly nonlinear. The neural network based MRAC architecture is well suited to these systems. With the improvement of RNN training, in terms of training time and convergence, system identification and control design using RNNs have increased potential. In the future, we want to test the neural network based MRAC scheme on additional practical applications. Also, one of the concerns about neural control systems is stability. We want to develop a method that can be used to maintain the stability of RNNs during the training process.

REFERENCES

[1] R. Haschke and J. Steil, "Input space bifurcation manifolds of recurrent neural networks," *Neurocomputing*, vol. 64, pp. 25–38, Mar. 2005.

[2] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, 1994.

[3] 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.

[4] M. H. Beale, M. T. Hagan, and H. B. Demuth. (2013) Neural network toolbox 8 - user's guide. [Online]. Available: http://www.mathworks.com/products/neural-network/

[5] M. T. Hagan and H. B. Demuth, "Neural networks for control," in *Proc. Amer. Control Conf*, San Diego, CA, 1999, pp. 1642–1656.

[6] J. Roman and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," in *Proc. 29th Hawaii Int. Conf. Syst. Sci.*, 1996, pp. 454–460.

[7] J. Feng, C. K. Tse, and F. C. M. Lau, "A neural-network-based channel-equalization strategy for chaos-based communication systems," *IEEE Trans. Circuits Syst. I: Fundam. Theory Appl.*, vol. 50, no. 7, pp. 954–957, 2003.

[8] I. Kamwa, R. Grondin, V. K. Sood, C. Gagnon, V. T. Nguyen, and J. Mereb, "Recurrent neural networks for phasor detection and adaptive identication in power system control and protection," *IEEE Trans. Instrum. Meas*, vol. 45, no. 2, pp. 657–664, 1996.

[9] Jayadeva and S. A. Rahman, "A neural network with o(n) neurons for ranking n numbers in o(1/n) time," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 51, no. 10, pp. 2044–2051, 2004.

[10] G. Chengyu and K. Danai, "Fault diagnosis of the ifac benchmark problem with a model-based recurrent neural network," in *Proc. IEEE Int. Conf. Control Appl.*, 1999, pp. 1755–1760.

[11] S. Fernndez, A. Graves, and J. Schmidhuber, "Sequence labelling in structured domains with hierarchical recurrent neural networks," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, Hyderabad, India, 2007, pp. 774–779.

[12] A. Graves, S. Fernndez, M. Liwicki, H. Bunke, and J. Schmidhuber, "Unconstrained on-line handwriting recognition with recurrent neural networks," *Advances in Neural Information Processing Systems*, vol. 20, pp. 577–584, 2008.

[13] L. R. Medsker and L. C. Jain, *Recurrent Neural Networks: Design and Applications.* Boca Raton, FL: CRC Press, 2000.

[14] P. Gianluca, D. Przybylski, B. Rost, and P. Baldi, "Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles," *Proteins: Structure, Function, Genetics*, vol. 47, pp. 228–235, 2002.

[15] K. Doya, "Bifurcations in the learning of recurrent neural networks," in *Proc. 1992 IEEE Int. Symp. Circuits and Systems*, vol. 6, 1992, pp. 2777–2780.

[16] R. Pascanu, T. Mikolov, and Y. Bengio. (2013) On the dificulty of training recurrent neural networks. [Online]. Available: http://arxiv.org/pdf/1211.5063.pdf

[17] 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.

[18] 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.

[19] M. C. Phan and M. T. Hagan, "Error surface of recurrent neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 24, no. 11, pp. 1709–1721, Nov. 2013.

[20] M. C. Phan, M. H. Beale, and M. T. Hagan, "A procedure for training recurrent networks," in *Proc. Int. Joint Conf. Neural Netw.*, Dallas, TX, Aug. 2013.

[21] H. K. Khalil, *Nonlinear systems*, 3rd ed. Prentice Hall, 2002.

[22] F. Pasemann, "Dynamics of a single model neuron," *International Journal of Bifurcation and Chaos*, vol. 3, no. 2, pp. 271–278, 1993.

[23] P. Frasconi, M. Gori, and G. Soda, "Local feedback multilayer networks," *Neural Comput.*, vol. 4, no. 1, pp. 120–130, 1991.

[24] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "Unified integration of explicit knowledge and learning by example in recurrent networks," *IEEE Trans. Knowl. Data Eng.*, vol. 7, no. 2, pp. 340–346, 1995.

[25] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," in *A Field*

Guide to Dynamical Recurrent Neural Networks, S. C. Kremer and J. F. Kolen, Eds. Piscataway, NJ: IEEE Press, 2001.

[26] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables and Systems of Equations.* Waltham, MA: Academic Press, 1960.

[27] T. V. Petersdorff. (2012) Fixed point iteration and contraction mapping theorem. [Online]. Available: http://www2.math.umd.edu/~petersd/466/fixedpoint.pdf

[28] P. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[29] T. Lin, B. Horne, P. Tino, and C. Giles, "Learning long-term dependencies in narx recurrent networks," *IEEE Trans. Neural Netw.*, vol. 7, no. 6, pp. 1329–1338, Nov. 1996.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[31] H. Jaeger, "Echo state network," *Scholarpedia*, vol. 2, no. 9, 2007.

[32] M. Lukoševičius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Comput. Sci. Rev.*, vol. 3, no. 3, pp. 127–149, Aug. 2009.

[33] J. Martens and I. Sutskever, "Learning recurrent neural networks with hessian-free optimization," in *Proc. 28th Int. Conf. Machine Learning*, Bellevue, WA, 2011, pp. 1–8.

[34] A. Schaefer, S. Udluft, and H. Zimmermann, "Learning long-term dependencies with recurrent neural networks," *Neurocomputing*, vol. 71, no. 13-15, pp. 2481–2488, 2008.

[35] C.-T. Chen, *Linear System Theory and Design.* Oxford University Press, 1998.

[36] A. V. Oppenheim, A. S. Willsky, and S. Hamid, *Signals and Systems.* Prentice Hall, 1997.

[37] B. A. Brousseau, *Linear Recursion and Fibonacci Sequences.* San Jose, CA: Fibonacci Association, 1971.

[38] K. Ogata, *Discrete-Time Control Systems*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1995.

[39] V. E. Hoggatt, "Divisibility properties of generalized fibonacci polynomials," *The Fibonacci quarterly*, vol. 12, no. 2, pp. 113–120, Apr. 1974.

[40] J. Cryer and K. Chan, *Time Series Analysis with Application in R*, 2nd ed. New York, NY: Springer, 2008.

[41] 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.

[42] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the marquardt algorithm," *IEEE Trans. Neural Netw.*, vol. 5, no. 6, pp. 989–993, Nov. 1994.

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

[44] N. H. Nguyen and M. T. Hagan, "Stability analysis of layered digital dynamic networks using dissipativity theory," in *Proc. Int. Joint Conf. Neural Netw.*, San Jose, CA, Aug. 2011, pp. 1692–1699.

[45] R. Jafari and M. T. Hagan, "Global stability analysis using the method of reduction of dissipativity domain," in *Proc. Int. Joint Conf. Neural Netw.*, San Jose, CA, Aug. 2011, pp. 2550–2556.

[46] 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.

[47] A. D. Mahindrakar and R. N. Banavar, "A swing-up of the acrobot based on a simple pendulum strategy," *International Journal of Control*, vol. 78, no. 6, pp. 424–429, 2005.

[48] D. Wilson and T. Martinez, "The general inefficiency of batch training for gradient descent learning," *Neural Networks*, vol. 16, pp. 1429–1451, 2003.

[49] T. Nakama, "Theoretical analysis of batch and online training for gradient descent learning in neural networks," *Neurocomputing*, vol. 73, pp. 151–159, 2009.

[50] S. Singhal and L. Wu, "Training feed-forward networks with the extended kalman algorithm," in *Proc. Int. Conf. Acoust, Speech, and Signal Process*, vol. 2, May 1989, pp. 1187–1190.

[51] G. Puskorius and L. Feldkamp, "Neurocontrol of nonlinear dynamical systems with kalman filter trained recurrent networks," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 279–297, Mar 1994.

[52] ——, "Extensions and enhancements of decoupled extended kalman filter training," in *Int. Conf. Neural Netw.*, vol. 3, Jun 1997, pp. 1879–1883.

[53] ——, "Decoupled extended kalman filter training of feedforward layered networks," in *Proc. Int. Joint Conf. Neural Netw.*, vol. 1, Jul 1991, pp. 771–777.

[54] R. Dorf and R. Bishop, *Modern control systems*, 12th ed. Pearson.

[55] P. Guillaume, J. Schoukens, and R. Pintelon, "Sensitivity of roots to errors in the coefficient of polynomials obtained by frequency-domain estimation methods," *IEEE Trans. Instrum. Meas.*, vol. 38, no. 6, pp. 1050–1056, Dec. 1989.

APPENDIX A

Proof that (4.16) approximates (4.17)

We will show that the roots of the polynomial in (4.16) approach the corresponding roots of the polynomial in (4.17) as w_2 increases. For simplicity, we will rewrite the polynomial in (4.17) as

$$f(w_1) = \sum_{k=0}^{Q-3} b_k w_1^{2k+1} \tag{A.1}$$

where $b_k = a_k w_2^{Q-3-k}$.

Let w_{1i} $(i = 1, 2, \ldots, Q-3)$ be the $Q-3$ roots of the $(Q-3)^{\text{rd}}$ order polynomial (A.1). We consider the right-hand side of (4.16) as a polynomial in w_1 with perturbed coefficients. The coefficient b_k is perturbed by $\triangle b_k$, which is a polynomial in w_2 with order less than the order of b_k. Because of this perturbation, the root of (4.16) corresponding to w_{1i} differs from w_{1i} by $\triangle w_{1i}$. The following formula shows the sensitivity of the roots to the perturbation of coefficients [55]

$$\left|\frac{\triangle w_{1i}}{w_{1i}}\right| = S_{b_k}^{w_{1i}} \left|\frac{\triangle b_k}{b_k}\right|$$

where $S_{b_k}^{w_{1i}}$ is the sensitivity of the relative error in the root w_{1i} to the relative perturbations of the coefficient b_k, and

$$S_{b_k}^{w_{1i}} = \left|\frac{b_k w_{1i}^{2k}}{f'(w_{1i})}\right| = \left|\frac{b_k w_{1i}^{2k}}{b_{Q-3}\prod_{j\neq i}(w_{1i} - w_{1j})}\right|$$

As we will show in the following appendix, $f'(w_{1i})$, which is $\frac{1}{2}\frac{d^2F_t}{dw_1^2}|_{w_{1i}}$, could not be zero. Therefore, $S_{b_k}^{w_{1i}}$ is finite. In other words, we can always find a bound M for all $S_{b_k}^{w_{1i}}$. Given small $\varepsilon > 0$, for large enough w_2, we have $\left|\frac{\triangle b_k}{b_k}\right| < \frac{\varepsilon}{M}$. Thus $\left|\frac{\triangle w_{1i}}{w_{1i}}\right| < M \times \frac{\varepsilon}{M} = \varepsilon$. This implies that we can make the relative error of the roots as small as desired.

APPENDIX B

Proof that $\frac{d^2F_t}{dw_1^2} > 0$ if $g(Q-1) = 0$

By (4.18), we have $\frac{1}{2}\frac{d^2F_t}{dw_1^2} = \left[\frac{dg(Q-1)}{dw_1}\right]^2 \geq 0$. We will show that $\frac{dg(Q-1)}{dw_1} \neq 0$. By induction we can prove the following identity:

$$\frac{dg(Q-1)}{dw_1} = \frac{2(Q-1)w_2g(Q-2) + (Q-2)w_1g(Q-1)}{w_1^2 + 4w_2}. \tag{B.1}$$

One property of Fibonacci polynomials is that the greatest common divisor of $g(Q-1)$ and $g(Q-2)$ is 1 ([39], Theorem 4). Thus $g(Q-1)$ and $g(Q-2)$ could not be both zero for the same w_1 and w_2. Since $g(Q-1) = 0$, $g(Q-2) \neq 0$. Also, we have $w_2 \neq 0$ (our assumption above is that $|w_2|$ is large enough). Therefore, by (B.1), we have $\frac{dg(Q-1)}{dw_1} = \frac{2(Q-1)w_2g(Q-2)}{w_1^2+4w_2} \neq 0$. This means that $\frac{d^2F_t}{dw_1^2} > 0$.

APPENDIX C

Proof that $\frac{d^2F_t}{dw_1^2} < 0$ if $\frac{dg(Q-1)}{dw_1} = 0$

By (4.18), we have $\frac{1}{2}\frac{d^2F_t}{dw_1^2} = g(Q-1)\frac{d^2g(Q-1)}{dw_1^2}$. Also, by induction, we can prove the following identity:

$$\frac{d^2g(Q-1)}{dw_1^2} = \frac{-3w_1\frac{dg(Q-1)}{dw_1} + \left[(Q-1)^2-1\right]g(Q-1)}{w_1^2+4w_2} \tag{C.1}$$

Since $\frac{dg(Q-1)}{dw_1} = 0$, $\frac{d^2g(Q-1)}{dw_1^2} = \frac{\left[(Q-1)^2-1\right]g(Q-1)}{w_1^2+4w_2}$. Thus

$$\frac{1}{2}\frac{d^2F_t}{dw_1^2} = \frac{\left[(Q-1)^2-1\right]\left[g(Q-1)\right]^2}{w_1^2+4w_2} \tag{C.2}$$

As in Appendix B, we can show that since $\frac{dg(Q-1)}{dw_1} = 0$, $g(Q-1) \neq 0$. So the numerator of $\frac{1}{2}\frac{d^2F_t}{dw_1^2}$ in (C.2) is positive. We just need to consider the denominator, which is $w_1^2+4w_2$. We consider two sub-cases:

- $w_1 = 0$: Note that this occurs when Q is even.

If $w_2 > 0$, then $w_1^2 + 4w_2 = 4w_2 > 0$, so $\frac{d^2F_t}{dw_1^2} > 0$. Thus, part of the vertical line $w_1 = 0$ with $w_2 > 0$ is a valley. Otherwise, if $w_2 < 0$, then $\frac{d^2F_t}{dw_1^2} < 0$, so part of the vertical line $w_1 = 0$ with $w_2 < 0$ is a ridge.

- $w_1 \neq 0$:

We will show that $\frac{d^2F_t}{dw_1^2} < 0$ or $w_1^2 + 4w_2 < 0$. We will prove this by contradiction. Suppose that $w_1^2 + 4w_2 > 0$, then (4.4) has two real and distinct roots λ_1 and λ_2 (assume that $|\lambda_1| > |\lambda_2|$). Note that the condition $w_1 \neq 0$ guarantees that $|\lambda_1| \neq |\lambda_2|$ (if $w_1 = 0$ then $|\lambda_1| = |\lambda_2| = \sqrt{w_2}$). It also guarantees that $g(Q-2) \neq 0$ which makes the limit below valid. Again, by [37], Lesson 8, we have:

$$\lim_{Q\to\infty}\frac{g(Q-1)}{g(Q-2)} = \lambda_1 = \frac{w_1+\sqrt{w_1^2+4w_2}}{2} \tag{C.3}$$

From (B.1) and the condition that $\frac{dg(Q-1)}{dw_1} = 0$, we have

$$w_2 = -\frac{(Q-2)}{2(Q-1)} \times w_1 \times \frac{g(Q-1)}{g(Q-2)}$$

Taking the limit of both sides as $Q \to \infty$ and using (C.3), we have

$$w_2 = -\frac{1}{2}w_1 \lim_{Q\to\infty}\frac{g(Q-1)}{g(Q-2)} = -\frac{w_1(w_1+\sqrt{w_1^2+4w_2})}{4}$$
$$\Leftrightarrow w_1^2+4w_2 = -w_1\sqrt{w_1^2+4w_2}$$

This implies that $w_1^2+4w_2 = w_1^2$ (by squaring both sides), so $w_2 = 0$. This does not satisfy our assumption that $|w_2|$ is large enough. Therefore, the assumption that $w_1^2+4w_2 > 0$ is false. Thus we have $w_1^2+4w_2 < 0$ or $\frac{d^2F_t}{dw_1^2} < 0$.

VITA

Manh C. Phan

Candidate for the Degree of

Doctor of Philosophy

Dissertation: RECURRENT NEURAL NETWORKS: ERROR SURFACE ANALYSIS AND IMPROVED TRAINING

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in July, 2014.

Completed the requirements for the Bachelor of Science in Electrical Engineering at Hanoi University of Science and Technology, Hanoi, Vietnam in 2008.