EXTENDED KALMAN FILTER BASED PRUNING ALGORITHMS AND SEVERAL ASPECTS OF NEURAL NETWORK LEARNING

By
John Pui-Fai SUM

Co-supervised By
Professor Wing-Kay KAN
Professor Gilbert H. YOUNG

A Dissertation
submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
Division of Computer Science & Engineering
The Chinese University of Hong Hong
July 1998

Acknowledgement

I would like to express my gratitude to Professor Wing-kay Kan and Professor Gilbert H. Young for their supervision and advice. Furthermore, I also want to thank Perofessor Lai-wan Chan, Professor Chun-hung Cheng, Professor Kin-hong Wong and Professor Peter Tam for their valuable comments on the earlier version of this dissertation. In particular, I am indebted to Professor Peter Tam and Dr. Chi-sing Leung for their valuable suggestion. I would also like to thank Mr. Philip Leetch for his help in the final statges of this dissertation. Finally, I would like to thank my family, my girlfriend Venus Poon and my friend Ricky Wong for their encouragement toward the completion of this research.

Abstract

In recent years, more and more researchers have been aware of the effectiveness of using the extended Kalman filter (EKF) in neural network learning since some information such as the Kalman gain and error covariance matrix can be obtained during the progress of training. It would be interesting to inquire if there is any possibility of using an EKF method together with pruning in order to speed up the learning process, as well as to determine the size of a trained network. In this dissertation, certain extended Kalman filter based pruning algorithms for feedforward neural network (FNN) and recurrent neural network (RNN) are proposed and several aspects of neural network learning are presented.

For FNN, a weight importance measure linking up prediction error sensitivity and the by-products obtained from EKF training is derived. Comparison results demonstrate that the proposed measure can better approximate the prediction error sensitivity than using the forgetting recursive least square (FRLS) based pruning measure. Another weight importance measure that links up the *a posteriori* probability sensitivity and by-products obtained from EKF training is also derived. An adaptive pruning procedure designed for FNN in a non-stationary environment is also presented. Simulation results illustrate that the proposed measure together with the pruning procedure is able to identify redundant weights and remove them. As a result, the computation cost for EKF-based training can also be reduced.

Using a similar idea, a weight importance measure linking up the *a posteriori* probability sensitivity and by-products obtained from EKF training is derived for RNN. Application of such a pruning algorithm together with the EKF-based training in system identification and time series prediction are presented. The computational cost required for EKF-based pruning is also analyzed. Several alternative pruning procedures are proposed to compare with EKF-based pruning procedure. Comparative analysis in accordance with computational complexity, network size and generalization ability are presented. No simple conclusion can be drawn from the comparative results. However, these results provide a guideline for practitioners once they want to apply RNN in system modeling.

Several new results with regard to neural network learning are also presented in this dissertation. To provide a support for the use of recurrent neural network in system modeling, the approximate realizability of an Elman recurrent neural network is proved. It is also proved that FRLS training can have an effect identical to weight decay. This provides more evidence showing the advantages of using FRLS in training a neural network. Another theoretical result is the proof of the equivalence between a NARX model and recurrent neural network. Finally, a parallel implementation methodology for FRLS training and pruning on a SIMD machine is presented.

Contents

Chapter 1

Introduction

1.1 Introduction

Good generalization ability and fast training speed are two basic criteria used for evaluating the performance of the learning methods in neural networks. In recent years, more and more researchers have become aware of the effectiveness of using the recursive least squares method[1] (RLS) [4, 65] and extended Kalman filter (EKF) [2, 99] in neural network learning. Kollias & Anastassiou [48] first applied the recursive least squares method in training multi-layered perceptron. Billings, Chen, Cowan, Grant and their co-workers applied the recursive least squares method in training the NARX model[2] and radial basis function network [6, 7, 10, 11, 14]. Puskorius & Feldkamp extended the idea to recurrent neural network training [90]. Gorinevsky [28] provided the persistence of excitation condition for using the recursive least squares method in neural network training.

Along the same line of research, Matthews & Moschytz [73] Iiguni *et al.* [37], Singhal & Wu [97], and Shan *et al.* [95] independently applied an extended Kalman filter in training a multi-layered perceptron and showed that its performance was superior to using a conventional backpropagation training method. Ruck *et al.* [93] gave a comparative analysis of the use of the extended Kalman filter and backpropagation in training. To further speed up the training rates, Puskorius & Feldkamp proposed a decoupled method for the extended Kalman filter [89]. More recently, Wan & Nelson proposed a dual Kalman filtering method for feedforward neural network training [121]. Using the same idea, Williams [123] and Suykens [113] independently formulated the training of a recurrent neural network as a state estimation problem and applied the extended Kalman filter for training. The EKF approach is an online mode training in which the weights are updated immediately after the presentation of a training pattern. This training method is useful because they do not require the storage of the entire input output history.

One concern in neural networks is how to find the optimal size neural network for a given problem. If the network is too small, it may not be feasible to train it to solve the given problem. On the other hand, if it is too large, overfitting will usually occur [74] and the

[1]RLS is a special case of the forgetting recursive least squares (FRLS) method. In neural network research, there are generally no clear cut between these two. In this dissertation, these two terms are used interchangeable.

[2]NARX is a short form for the so-called nonlinear auto-regressive with exogenous input model.

resources will be wasted. An approach to determine the size is the so-called pruning [91]. Various methods have previously been proposed for neural network pruning [91], such as optimal brain damage [55] and optimal brain surgeon [31]. In such methods, a large network is trained first and some unimportant weights are removed later. The importance of a weight is defined by how much the training error increases when it is removed. The estimation of all the error sensitivities is based on the Hessian matrix of the training error [55]. The time complexity to get such a Hessian matrix is of the order $\mathcal{O}(Nn_\theta^2)$, where n_θ and N are the total number of weights and the number of training patterns respectively. Although we can get the importance of a weight with a $\mathcal{O}(Nn_\theta)$ method [86] without requiring the exact calculation of the Hessian matrix, we still need $\mathcal{O}(Nn_\theta^2)$ times to get the importance of each weight. It should be noticed that to avoid overfitting, the number of training patterns is usually much greater than the total number of weights, i.e. $N \gg n_\theta$. Also in the on-line situation, the Hessian matrix is usually unavailable since the training patterns are not held after training. Adapting from the idea of system identification theory, Leung *et al.* [56, 57] has introduced RLS-based on-line pruning algorithms for feedforward neural networks in order to reduce the computational overhead due to the computation of the Hessian matrix.

As the EKF approach is a well-known fast training method [33] and some information such as the Kalman gain and error covariance matrix can be obtained during the progress of training, it would be interesting to inquire if there is any possibility of using an EKF method together with pruning in order to speed up the learning process, as well as to determine the size of a trained network.

1.2 Overview of the Dissertation

In this dissertation, certain extended Kalman filter based pruning algorithms for feedforward neural networks (FNN) and recurrent neural networs (RNN) are proposeds and several aspects of neural network learning are presented. Extensive simulation results are presented to verify the validity of the algorithms. The major contributions made by this dissertation are briefly introduced in the following paragraphs.

EKF-Based Pruning Algorithms

EKF-based on-line pruning for FNN

To implementan on-line EKF-based pruning algorithm that resembles the RLS-based pruning, one major problem is that the objective function of using the EKF approach is different from using the RLS. The objective of the RLS method is minimizing the sum squares error while the EKF is maximizing the *a posteriori* probability [2]. From that point, using an error sensitivity approach for pruning will no longer be effective since we still have to wait until the training finishes, and evaluate the Hessian matrix by using a $\mathcal{O}(Nn_\theta^2)$ method to get the importance of each weight.

In order to obviate such a limitation, it is necessary to define an alternative weight importance measure which can (1) make use of the by-product obtained from EKF-based training and (2) hopefully be related to some meaningful objective functions. Considering these two points, *two novel weight importance measures are defined in this dissertation for* *the feedforward neural network.* The first measure is defined in accordance with the mean prediction error. The second one is defined in accordance with the *a posteriori* probability. Both of them can be evaluated by using the covariance matrix and the estimated weight vector obtained via the EKF method. The advantages of using such importance measures are (1) pruning can be made adaptive and (2) pruning a time varying model can be accomplished. *Extensive simulation results are reported in order to justify the viability of two proposed measures compared with other measures.*

Details of this contribution are presented in Chapters 5 and 6. Part of the results have also appeared in the following papers.

John Sum, Chi-sing Leung, Lai-wan Chan, Wing-kay Kan and Gilbert H. Young, On the Kalman filtering method in neural network training and pruning, accepted for publication in *IEEE Transactions on Neural Networks.*

John Sum, Chi-sing Leung, Lai-wan Chan, Wing-kay Kan and Gilbert H. Young, An adaptive Bayesian pruning for neural network in non-stationary environment, to appear in *Neural Computation.*

More background of this contribution can be found in the following papers.

Chi-sing Leung, *Pui-fai Sum*, Ah-chung Tsoi and Lai-wan Chan, Several aspects of pruning methods in recursive least square algorithms for neural networks, *Theoretical Aspects of Neural Computation : A Multidisciplinary Perspective*, K. Wong *et al.* (eds.) Springer-Verlag, p.71-80, 1997.

C.S.Leung, K.W.Wong, *John Sum*, and L.W.Chan, On-line training and pruning for RLS algorithms, *Electronics Letters*, Vol.32, No.23, 2152-2153, 1996.

EKF-based pruning algorithm for RNN

Pruning a recurrent neural network based on the idea of error sensitivity is generally a difficult task. One problem is that conventional training methods such as real-time-recurrent-learning (RTRL) [124] are time consuming. When the allowable time for training is not very long, the quality of the trained neural network will be poor. Hence, using RTRL in conjunction with an error sensitivity measure will not be effective. Another problem is that the initial conditions of all the hidden units are usually not known and so are usually assumed to be all zeros. Therefore, the Hessian matrix evaluated suffers from the same assumption.

In order to avoid these two problems, one approach is to use an EKF method in training the RNN since an EKF can on-line estimate the hidden units' activities and at the same time the EKF is an effective training method for RNN [123]. As the training is rather effective, *a new weight importance measure is proposed in this dissertation for RNN pruning which can make use of the information obtained after training, the weight vector and its error covariance matrix.* Since the objective of extended Kalman filtering is to maximize the *a posteriori* probability, not to minimize training error, the use of error sensitivity will not be appropriate to prune the network. Therefore the sensitivity of the *a posteriori* probability as a measure of the importance of the weight is proposed and several pruning procedures are devised for recurrent neural networks. Furthermore, *extensive simulation results are presented in this dissertation to justify the viability of the proposed measure.* Details of this contribution are presented in Chapters 7 and 8.

Several Aspects of Neural Network Learning

Approximate realization property of RNN

Recurrent neural networks have been proposed for about a decade [94] and then many different recurrent neural network models have been proposed. Regarding the universal approximation property of some multi-layered feedforward neural networks [21, 24, 35], *this dissertation proves that the Jordan model [33] and the recurrent radial basis function network [33] are able to realize any discrete-time nonlinear state-space system with arbitrary accuracy.* Besides, using the result derived by Funahashi [24], *this dissertation also proves that a fully connected recurrent neural network with sigmoidal or RBF hidden unit is able to realize any discrete-time nonlinear state-space systems.*

Details of this contribution are presented in Chapter 9 and part of the results also appeared in the following paper.

John Sum, Lai-wan Chan., On the approximation property of recurrent neural network, to appear in *Proceedings of World Multiconference on Systemics, Cybernetics and Informatics*, 1997.

Regularization property of FRLS

Due to its fast convergence rate and its adaptive behavior, the forgetting recursive least square (FRLS) method has recently been applied widely to the training of feedforward neural networks. As in many applications such as system identification and time series prediction, a batch of training data usually cannot be obtained in advance. Therefore, conventional batch mode training techniques such as backpropagation, Newton method and other nonlinear programming techniques, cannot be easily applied. Thus, the FRLS method or other adaptive training methods become inevitable. With the increasing popularity of using FRLS in neural network learning [12] [15] [48] [56] [57] [95] and pruning[56] [57], *this dissertation investigates the regularization ability of using FRLS in training.* Regularization is a method that aims at reducing the model complexity [42] and preventing the training problem from being ill-conditioned [114], [53], [67], [74], [75], [76] and [79]. Many articles that focus on the design of a regularizer [128], the use of regularization [42] [67] and the effect of regularization in model complexity [75] [76] [79] can be found in the literature. In the conventional batch mode training approach, regularization is usually realized by adding an extra term or a penalty term such as weight decay term [74] and Tikhonov regularizer [8, 42] to the training error function. Using the FRLS method, the training error function is a kind of weighted sum square error function. *This dissertation shows that this objective function for FRLS is similar to that of adding a weight decay penalty term. Hence, based on this finding, an elegant on-line training method which accomplishes the same effect as weight decay can be realized by using FRLS.*

Details of this contribution are presented in Chapter 10 and part of the results also appeared in the following paper.

John Sum, Wing-kay Kan and Gilbert H. Young, On the regularization of forgetting recursive least square, submitted to *IEEE Transactions on Neural Networks.*

Equivalence of NARX and RNN

Owing to their structural differences, the NARX model and RNN are basically studied independently. Only a few papers have presented results on their similarities [19, 84]. Olurotimi [84] has recently shown that every RNN can be transformed into a NARX model and thus he has derived an algorithm for RNN training with feedforward complexity. Inspired by Olurotimi's work, *this dissertation will show that every RNN can be transformed into a first order NARX model and vice versa if the condition that the neuron transfer function is a hyperbolic tangent. Besides, every NARX model of order larger than one can be transformed to a RNN if the neuron transfer function is piece-wise linear.* According to these results, there are three advantages from which we can benefit. (i) If the output dimension of a NARX model is larger than the number of its hidden unit, the training of a NARX model can be speeded up by an indirect method, i.e. the NARX model is transformed into a RNN and is trained. Once the training finishes, the RNN is transformed back to an NARX model. On the other hand, (ii) if the output dimension of a RNN model is smaller than the number of its hidden units, the training of a RNN can be speeded up by using this similar method. (iii) There is a simpler way to accomplish RNN pruning, i.e. the corresponding NARX model is pruned instead of the RNN. After pruning, the NARX model is transformed back to the equivalent RNN.

Details of this contribution are presented in Chapter 11 and a part of the results has appeared in the following paper.

John P.F. Sum, Wing-kay Kan and Gilbert H. Young, A note on the equivalence of NARX and RNN, accepted for publication in *Neural Computing and Applications*

Parallel Implementation of Training and Pruning Algorithms for FNN

In recent years, most of the implementations of ANN are accomplished by using general purpose, serial computers. This approach though flexible is often too slow. Besides, an optimal network size is always hard to determine in advance. One usually starts with a large network which consists of a large number of hidden units. After the training is finished, redundant weights are removed [55]. In order to speed up the training process, intensive research on the mapping of ANN onto parallel computing architectures [36, 17], such as mesh structure [63], array structure [52, 18, 38, 44, 129], ring structure [1, 77] and hypercube structure [51, 72] have been carried out. These methods solely apply a backpropagation approach to training the neural network. FRLS is an alternative method which has been applied to train feedforward neural networks in recent years [28, 48]. Experimental results have demonstrated that the time complexity of using the FRLS approach is usually much smaller than using backpropagation even though the one-step computation complexity in a FRLS method is higher than backprogation, (see [33] for detail.). Although there are several advantages in using FRLS to train a neural network, implementation of such an algorithm in a SIMD machine is scarce. *This dissertation proposes a methodology for the implementation of both FRLS-based training and pruning on a SIMD machine. The mapping of the training* *algorithm is indeed a direct extension of the algorithm presented in [89] and [95]. For pruning, an algorithm is suggested to facilitate the SIMD architecture. An analysis of the time complexity of using such a parallel architecture is given.*

Details of this contribution are presented in Chapter 12 and part of the results also appeared in the following papers.

John Sum, Gilbert H. Young and Wing-kay Kan, Toward a design of recursive least square based training and pruning using SIMD machine, to be presented in *10th International Conference on Parallel and Distributed Computing and Systems, Las Vegas, Nevada.*

Wing-kay Kan, *John Sum* and Gilbert H. Young, Parallel extended Kalman filter approach for neural network learning and data cleaning, to be presented *International Symposium on Operations Research nd its Applications* (ISORA'98) August 20-22, 1998, Kumming, China.

1.3 Organization of the Dissertation

All the above contributions will be elucidated in this dissertation. For clarity, the subsequent chapters will be organized into five parts. Part I consists of three chapters providing the prerequisites for understanding the results presented in the dissertation. Chapter 2 reviews the general principles of neural network learning. Two major factors governing neural learning — architecture and learning objective — will be described. The approximation property of neural networks will be presented. Existing techniques in neural network training and pruning will be summarized in Chapter 3. A discussion of their limitations and the motivation for using an extended Kalman filter for pruning will be presented in Chapter 4.

The main results are presented in Parts II to IV. Part II comprises of two chapters which present the use of an extended Kalman filter in pruning feedforward neural networks. In Chapter 5, a weight importance measure linking up prediction error sensitivity and the by-products obtained from EKF training is derived. Comparison results with FRLS-based pruning are also reported in this chapter. Another weight importance measure that links up *a posteriori* probability sensitivity and the by-products obtained from EKF training is derived in Chapter 6. Applications of such a probability sensitivity measure in pruning a feedforward neural network in a non-stationary environment are also presented.

Part III consists of two chapters which present the use of the extended Kalman filter in pruning recurrent neural networks. Chapter 7 starts by presenting an EKF-based pruning algorithm for a recurrent neural network. Then the application of such a pruning algorithm together with EKF-based training in system identification, and time series prediction are reported. A comparative analysis of an EKF based pruning procedure and random ranking pruning procedure is also given in this chapter. The computational costs required for EKF-based pruning are analyzed in Chapter 8. In this chapter, several alternative pruning procedures are proposed and compared with the EKF-based pruning procedures. Comparative analysis in accordance with computational complexity, network size and generalization ability is reported.

Part IV consists of three chapters that present new analytical and implementation results related to neural learning. Chapter 9 presents a proof showing that the Elman recurrent neural network can approximately realize any discrete time non-linear state-space system.

This proof provides another support for the use of recurrent neural networks in system modeling. Chapter 10 shows that FRLS training can have an effect identical to weight decay. This result provides further evidence showing the advantages of using FRLS in training a neural network. Another theoretical result is the proof of the equivalence between a NARX model and a recurrent neural network which is presented in Chapter 11. The last chapter of Part IV, Chapter 12 presents a methodology for the implementation of FRLS training and pruning onto a SIMD machine.

Part V consists of only one chapter, Chapter 13, which will summarize the research results of this dissertation and some possible future work.

In order to make the dissertation self-contained, the essential concepts of extended Kalman filtering such as its objective of estimation and its recursive Bayesian behavior, will briefly be introduced in the Appendix.

Part I

Neural Network Learning

Chapter 2

Neural Network Learning

This chapter and the next two chapters will review the very basic principle of neural network learning. Essentially, this principle resembles that of system modeling. That is to say, given a set of measured data, one tries to find out a good model which can describe the data as neatly as possible. Four factors determine the process of learning. They are, (1) neural network architecture, (2) basic objective, (3) smoothing objective and (4) neural network complexity. These four factors will be elucidated in this chapter. For a more detailed survey of neural network learning, one can refer to [32, 33, 34, 60, 92, 94], and the references therein.

2.1 System Modeling

In system identification, one might be given a set of input-output data $\{x_i, y_i\}_{i=1}^N$ arising by an unknown process, see Figure 2.1 for example, and asked to find a mathematical model for such an unknown process. As the true model is not known in advance, one has to assume a model for it, a feedforward neural network (FNN) for instance. A mathematical representation of FNN is given by

$$y_k = \sum_{i=1}^{n} c_i \tanh\left(a_i x_k + b_i\right), \tag{2.1}$$

where a_i, b_i and c_i are the input weight, bias and output weight respectively for the ith hidden unit. The non-linear function tanh is defined as follows :

$$\tanh(s) = \frac{\exp(s) - \exp(-s)}{\exp(s) + \exp(-s)}.$$

Let

$$\theta = (a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n, c_1, c_2, \ldots, c_n)^T,$$

a *representation* of the set of data can be obtained by solving the following optimization problem.

$$\hat{\theta} = \arg\min_{\theta} \left\{ \frac{1}{N} \sum_{k=1}^{N} (y_k - \hat{y}(x_k;\theta))^2 \right\}.$$

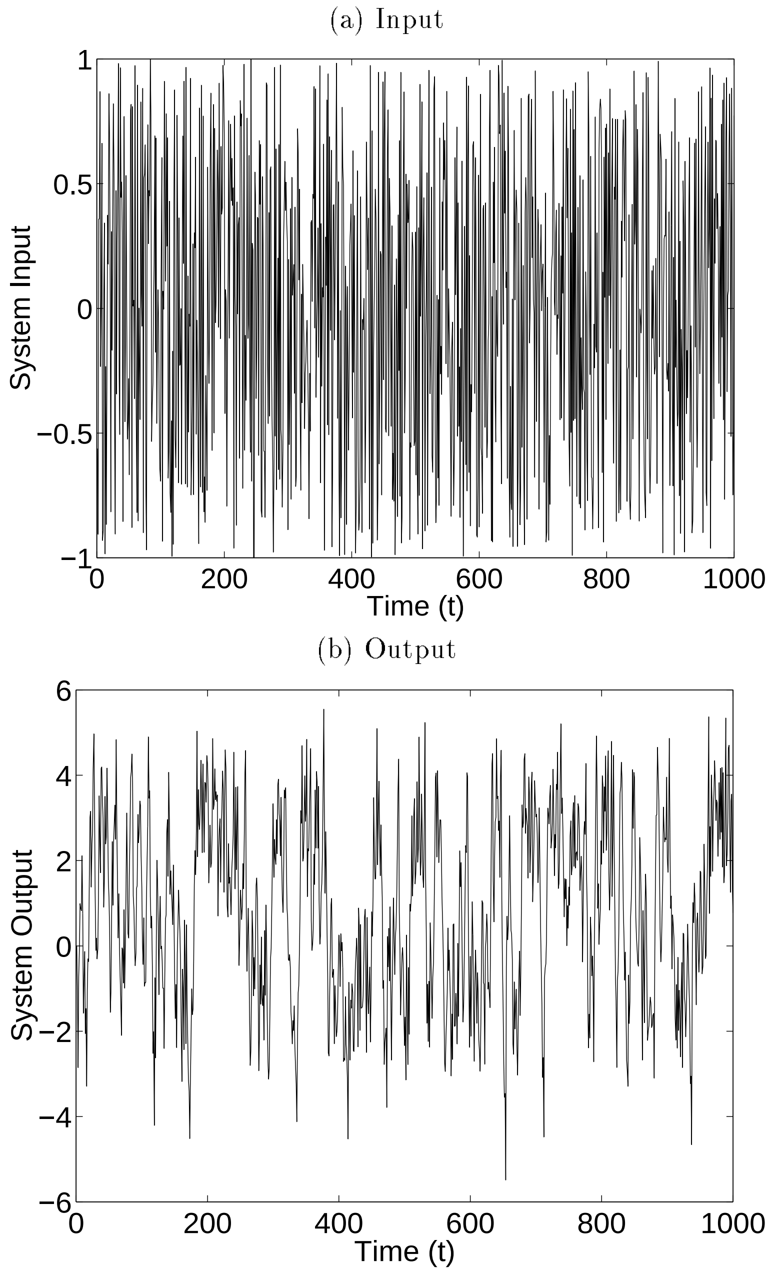


Figure 2.1: A set of input-output data arising by an unknown process.

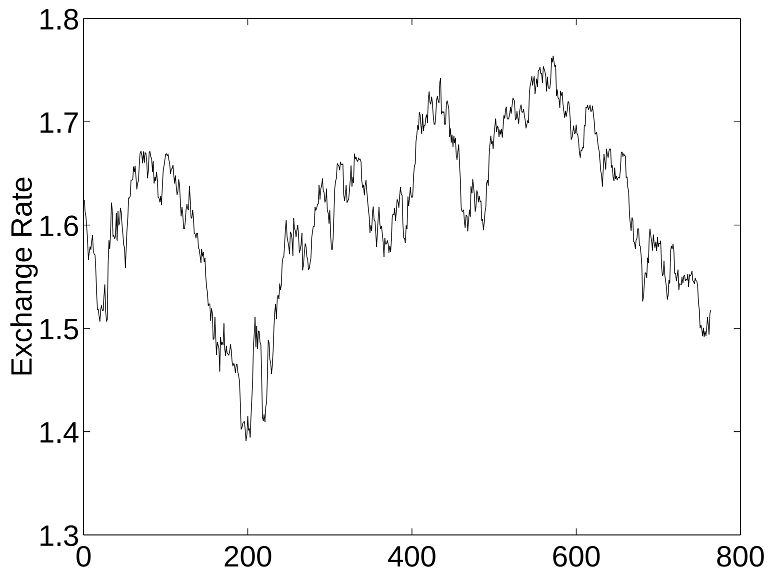


Figure 2.2: Foreign exchange rate of USD/DEM vs working day.

Since the true model is unknown, one can also assume that the underlying model is defined as follows :

$$y_k = \sum_{i=1}^{n} c_i \tanh \left(a_{i0} x_k + \sum_{t=1}^{m} a_{it} y_{k-t} + b_i \right). \tag{2.2}$$

It is the so-called NARX model. Let $\vec{a}_i = (a_{i0}, a_{i1}, a_{i2}, \ldots, a_{im})^T$ and

$$\theta = (\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n, b_1, b_2, \ldots, b_n, c_1, c_2, \ldots, c_n)^T,$$

another *representation* of the set of data can be given by

$$\hat{\theta} = \arg \min_{\theta} \left\{ \frac{1}{N} \sum_{k=1}^{N} (y_k - \hat{y}(x_k; \theta))^2 \right\}.$$

In time series modeling, one may be given a set of time series data, $\{y_i\}_{i=1}^{N}$, which has arisen by an unknown environment. A foreign exchange rate is an example of such time series data. In a stock market, foreign exchange rate is an index which indicates the relation between two currencies. Figure 2.2 displays the change of the exchange rate between USD and DEM. One might be asked to find a model for predicting this change. As a matter of fact, there is so far no good model for these exchange rate data. We can assume a feedforward neural network,

$$y_k = \sum_{i=1}^{n} c_i \tanh \left(\sum_{t=1}^{m} a_{it} y_{k-t} + b_i \right), \tag{2.3}$$

or a recurrent neural network

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} \sigma(a_{11} x_1(t-1) + a_{12} x_2(t-1) + e_1) \\ \sigma(a_{21} x_1(t-1) + a_{22} x_2(t-1) + e_2) \end{bmatrix},$$

$$y(t) \quad = \quad \begin{bmatrix} c_1 x(t) \\ c_2 x(t) \end{bmatrix},$$

for the exchange rate data. Here the non-linear function σ is defined as follows :

$$\sigma(s) = \frac{\exp(s) - \exp(-s)}{\exp(s) + \exp(-s)}.$$

This model is a two-hidden-unit recurrent neural network. Let θ be the parametric vector containing all the model parameters, the representation can be obtained by solving the same problem as the case in system identification, i.e.

$$\hat{\theta} = \arg\min_{\theta} \left\{ \frac{1}{N} \sum_{k=1}^{N} (y_k - \hat{y}_k(\theta))^2 \right\}.$$

Obviously, there are many options in the choice of architecture in the use of neural networks for system modeling. For simplicity, only FNN and RNN will be presented as the results presented in this dissertation are developed for these two models.

2.2 Architecture

In neural network learning, the very first step is to select an appropriate architecture. Generally, there are two basic architectures for use, the feedforward neural network and the recurrent neural network.

A feedforward neural network model is a non-linear model defined as follows :

$$y(x) = \begin{bmatrix} \sum_{i=1}^{n} c_{1i}\sigma(a_i^T x + b_i) \\ \sum_{i=1}^{n} c_{2i}\sigma(a_i^T x + b_i) \\ \cdots \\ \sum_{i=1}^{n} c_{li}\sigma(a_i^T x + b_i) \end{bmatrix}, \tag{2.4}$$

where $y \in R^l$, $x \in R^m$, $a_i \in R^m$, $c_{ji}, b_i \in R$ for all $i = 1, \cdots, n$ and $j = 1, \cdots, l$. Common choices of the non-linear scalar function $\sigma(s)$ are sigmoidal, $\frac{1}{1+\exp(-s)}$ and hyperbolic tangent, $\frac{\exp(s)-\exp(-s)}{\exp(s)+\exp(-s)}$. Figure 2.3 shows a simple feedforward neural network with $l = 1$, $m = 2$ and $n = 2$. In compact form,

$$y(x) = C\tanh(Ax + b) \tag{2.5}$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix},$$

$$b = [b_1 \ \ b_2 \ \ \cdots \ \ b_n]^T,$$

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{l1} & c_{l2} & \cdots & c_{ln} \end{bmatrix}.$$

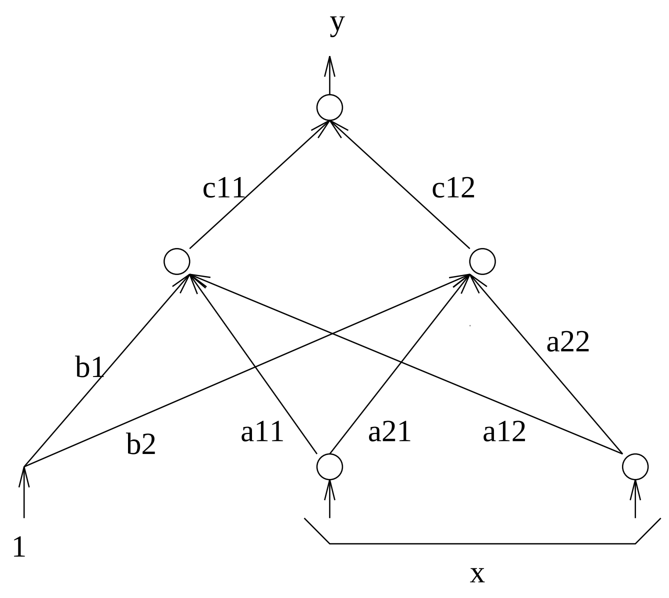


Figure 2.3: A feedforward neural network model.

A recurrent neural network model is a non-linear model which is defined as follows :

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \cdots \\ x_n(t) \end{bmatrix} = \begin{bmatrix} \sigma(a_1^T x(t-1) + b_1^T u(t) + e_1) \\ \sigma(a_2^T x(t-1) + b_2^T u(t) + e_2) \\ \cdots \\ \sigma(a_n^T x(t-1) + b_n^T u(t) + e_n) \end{bmatrix}, \tag{2.6}$$

$$y(t) = \begin{bmatrix} c_1^T x(t) \\ c_2^T x(t) \\ \cdots \\ c_l^T x(t) \end{bmatrix}, \tag{2.7}$$

where $y(t) \in R^l$, $x(t) \in R^n$, $u(t) \in R^m$, $a_i \in R^n$, $b_i \in R^m$, $e_i \in R$ for all $i = 1, 2, \cdots, n$ and $c_j \in R^n$ for all $j = 1, 2, \cdots, l$. The non-linear scalar function $\sigma(s)$ can be defined as either $\frac{1}{1+\exp(-s)}$ or $\sigma(s) = \frac{\exp(s)-\exp(-s)}{\exp(s)+\exp(-s)}$. Figure 2.4 shows a simple recurrent neural network with $l = 1$, $m = 2$ and $n = 2$.

Similarly, Equation (2.6) and (2.7) can also be rewritten in compact form as follows :

$$\begin{aligned} x(t) &= \sigma(Ax(t-1) + Bu(t) + e) && (2.8) \\ y(t) &= Cx(t), && (2.9) \end{aligned}$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix},$$

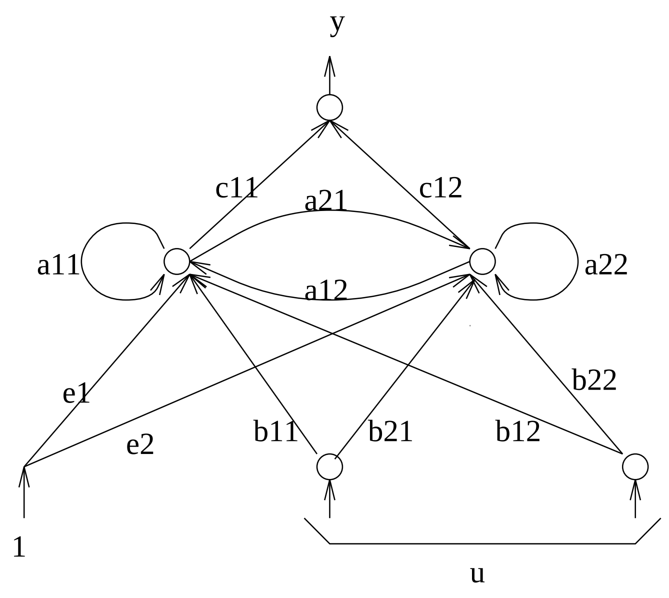


Figure 2.4: A recurrent neural network model.

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix},$$

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{l1} & c_{l2} & \cdots & c_{ln} \end{bmatrix},$$

$$e = [e_1 \ \ e_2 \ \ \cdots \ \ e_n]^T.$$

For clarity, notations used for FNN and RNN are depicted in Table 2.1.

	FNN	RNN
x	Network input	Hidden units' activity
y	Network output	Network output
u	—	Network input
f	$y = f(x, \theta)$	$x(t+1) = f(x(t), \theta, u(t+1))$
h	—	$y(t) = h(x(t), \theta)$

Table 2.1: Summary of the notations used for FNN and RNN.

2.3 Objectives function for neural networks learning

To apply the learned weight models in time series prediction and system identification, a training objective has to be defined to measure the performance of how good the prediction is or how accurate the identification is.

In general, we cannot know every detail about the system. The only information is a batch of input-output data obtained from observations or measurements collected from the system. Hence, the purpose of training can be stated conceptually in the following way :

Statement of problem : *Given a set of real-valued input/output data pair,* $\{(u(i), y(i)); i = 1, 2, \ldots .t\}$ *collected from an unknown system, devise an estimator to mimic the unknown environment as "precisely" as possible.*

Let θ and $\hat{y}(i|\theta)$ be the estimator parameter vector and the output of the estimator given input $x(i)$, one possible definition of "precision" of the estimator can be considered as the sum squares error, $\sum_{i=1}^{t} \|\hat{y}(i|\theta) - y(i)\|^2$. One type of optimal estimator, denoted by $\hat{\theta}$ is thus defined as the one minimizing least sum square error:

$$\hat{\theta} = arg \min_{\theta} \{\sum_{i=1}^{t} \|\hat{y}(i|\theta) - y(i)\|^2\}. \tag{2.10}$$

It is well-known that the least squares method is sensitive to outliers and bias [43], so other objective functions have been proposed for obtaining a "good" estimator. According to the theory of neural network learning, these objective functions can be summarized into three types of objective: (i) basic, (ii) validation and (iii) complexity.

2.3.1 Basic objective

Least square error

The least square error is one of the basic objectives for estimation. [49] [66]. Once a set of i/o pairs, $\{(x(i), y(i)) : i = 1, \ldots, t\}$, is given, the basic criterion is to minimize the mean squares error. We use $E_1(\theta)$ to denote this objective.

$$E_1(\theta) = t^{-1} \sum_{i=1}^{t} \|\hat{y}(i|\theta) - y(i)\|^2. \tag{2.11}$$

Maximum likelihood

An alternative basic objective function is derived from the principle of likelihood [65]. Suppose that the system is noise corrupted:

$$y(i) = \hat{y}(i|\theta) + \epsilon, \tag{2.12}$$

where ϵ is a Gaussian noise. Then, given the system parameter θ, and input $x(i)$, the posterior probability of $y = y(i)$ is given as normal distribution as well:

$$p(y(i)|\theta) = ((2\pi)^n|\Sigma|)^{-1/2} \exp\left\{-\frac{1}{2}(y(i) - \hat{y}(i|\theta))^T \Sigma^{-1} (y(i) - \hat{y}(i|\theta))\right\}, \tag{2.13}$$

where Σ is the covariance matrix of the estimator and $|\Sigma|$ is its determinant, n is the dimension of vector $y(i)$. The likelihood with which the output $\{y(i)\}_{i=1}^{t}$ will be generated by the system θ will be given by the multiplication of the factor $p(y(i)|\theta)$ from $i = 1$ to $i = t$. Hence, the log-likelihood function is written as follows, ignoring the scalar constant for simplicity:

$$E_2(\theta) = -\frac{1}{2}\{(2\pi)^n|\Sigma|\} - \sum_{i=1}^{t}\left\{\frac{1}{2}(y(i) - \hat{y}(i|\theta))^T\Sigma^{-1}(y(i) - \hat{y}(i|\theta))\right\}. \tag{2.14}$$

Note that $E_2(\theta) = E_1(\theta)$ if Σ is an identity matrix.

2.3.2 Smoothing objective

It was noticed that the senses of 'precision' in the least square method or maximum likelihood are dependent on the training input-output set, $\{x(i), y(i)\}_{i=1}^{t}$. Suppose we pick up $\{x(i), y(i)\}_{i=1}^{t-1}$ to train a neural network and use sum squares error as objective, an estimator $\hat{\theta}$ can be obtained. Once $\hat{\theta}$ is validated by $\{x(t), y(t)\}$, it is obvious that $(y(t) - \hat{y}(t))^2$ might not be equal to $(t-1)^{-1}\sum_{i=1}^{t-1}(\hat{y}(i|\theta) - y(i))^2$.

It is then relevant to ask how 'good' the estimator will be if it is applied to other sets of data. This raises three equivalent problems: (1) smoothing problem [118, 46, 47, 29], (2) regularization[1] problem [114, 27] and (3) generalization problem [74, 76], which have been discussed actively in recent years.

One simple solution to this problem in time series prediction is proposed by Whittaker in 1923 [47] who suggests that the solution $\{y(i)\}_{i=1}^{t}$ should balance a tradeoff between *the goodness of fit to the data* and *the goodness of fit to a smoothness criterion*:

$$\min_{y(i|\theta)}\left\{\sum_{i=1}^{t}[y(i) - y(i|\theta)]^2 + \lambda^2\sum_{i=1}^{t}[\partial_\theta^k y(i|\theta)]^2\right\}. \tag{2.15}$$

The first term is the infidelity of the data measure while the second term is the infidelity to the smoothness constraint. Compare this equation with $E_1(\theta)$ and $E_2(\theta)$, it is readily shown that Whittaker's objective can be extended as follows:

$$E_3(\theta) = E(\theta) + \lambda^2\sum_{i=1}^{t}\|\partial_\theta^k y(i|\theta)\|^2, \tag{2.16}$$

where $E(\theta)$ is either E_1 or E_2. Common choices of k are either one [114] or two [29]. As we generalize the above equation as

$$E(\theta) + \alpha \times S_1(\theta), \tag{2.17}$$

[1]It should be noted that there is another interpretation of regularization in the area of system identification (see chapter 6 of [65]). In system identification, one may require to compute the solution x in a matrix equation, say $Ax = b$. If matrix A is non-singular, it is well known that a unique solution exists. However, as the matrix A is near singular, the solution of x is difficult to compute even if we know that A is non-singular. In such a case, some particular technique, such as the Levenbeg-Marquardt method, is applied. Such a technique applied to solve these ill-posed problems is called regularization.

convention weight decay methods can also be treated as a regularizer[2]. There are two common weight decay methods: (1) keeping the weight values as small as possible and (2) keeping the weight values close to a predefined scalar value θ^0. The first weight decay method is indeed equivalent to the Tikhonov stabilizer [114]. Table 2.2 summarizes four common regularizers, where θ^j is the jth element of the vector θ. Gustafsson and Hjalmarsson recently proposed one special type of regularizer for linear estimation, a stable regularizer [30]. The purpose of this regularizer is to ensure that the stability of the estimator can be guaranteed.

Regularizer	Definition
Whittaker	$\sum_{i=1}^{t} \|\partial_\theta^k y(i\|\theta)\|^2$
Tikhonov	$\int \|\partial_\theta^k y(u\|\theta)\|^2 du$
Rissanen	$\|\theta\|^2$
Rumelhart	$\sum_{j=1}^{p} \frac{(\theta^j)^2}{(\theta^0)^2+(\theta^j)^2}$

Table 2.2: Summary of common regularizers.

2.3.3 Complexity objective

In accordance with *Parsimonious principle* [65], there should be a trade-off between model fit and model complexity. That is to say, it the unknown system can be modeled by more than one model, the simplest one should be preferable. To measure the complexity of a model, many criteria have been proposed in the last three decades.

Table 2.3 summarizes some model selection criteria which are commonly applied in system identification [30, 66] and neural network time series prediction [74]. Here, the Akaike FPE and GCV are not stated in their initial form. Originally, FPE and GCV are defined as $\frac{1+p/N}{1-p/N}E_1(\theta)$ and $E_1(\theta)(1-p/N)^{-2}$. where $p = \dim(\theta)$ and N is the total number of training data. Taking logarithm to both, the Akaike FPE and GCV can be rewritten in the form depicted in Table 2.3.

It should be noted that criteria of Table 2.3 are basically derived for the linear systems. Therefore, they suggest only a guideline for non-linear systems. Besides, no matter which criteria are used for measuring model complexity, model selection is a very difficult problem in particular when the size of a neural network is large[3]. Selecting a good neural network model will rely on heuristic techniques such as *network growing* [33] and *network pruning* [91]. In this dissertation, we concentrate on network pruning.

[2]It should be noted that there are several terminologies for $\alpha \times S_1(\theta)$. Two commonly used terminologies are *smoothing regularizer* and *regularizer*.

[3]It should be noted that the model selection problem is essentially NP-Complete even for a linear model. Thus, the selection of the best neural network structure will at least be NP-Complete.

2.4 Approximation Property

In the application of neural networks in solving problems, one basic assumption is that the applied model is able to realize any non-linear system. This section will briefly state some essential theorems which have been proven in recent years. These theorems lay the necessary conditions for neural network. These theorems will be revisited in a later chapter.

2.4.1 Universal Approximation of FNN

The following theorems will state that, for any given state space non-linear system, feed-forward neural networks are able to model such behavior as accurately as possible.

Theorem 1 (Funahashi Theorem[24]) *Let K be a subset of R^n and $f : K \to R^m$ be a continuous mapping. Then for an arbitrary $\epsilon > 0$, there exists a multi-layered perceptron with a finite number of hidden nodes such that*

$$\max_{x \in K} \|f(x) - W_1 \sigma(W_2 x + \theta)\| < \epsilon \tag{2.18}$$

holds, where $\sigma : R^n \to R^n$ is a sigmoid mapping. The matrix W_1, W_2 corresponds to the hidden to output connection matrix and input to hidden connection matrix.

$\square\square\square$

This theorem is the so-called universal approximation property for multilayer perceptron. It should be remarked that some other researchers have also obtained the same result, using a different approach [21] [35].

2.4.2 Approximate Realization Property of RNN

For recurrent neural network, we need a similar theorem showing its approximate realizability. A review of the earlier theorems regarding the approximate realization property of RNN will be presented in detail in Chapter 9.

Theorem 2 *Let D be a open subset of S, and $f : S \times R^n \to R^n$ be a continuous vector-valued function which defines the following non-autonomous non-linear system*

$$x(k+1) = f(x(k), u(k)), \tag{2.19}$$

$$y(k) = Cx(k). \tag{2.20}$$

Then, for an arbitrary number $\epsilon > 0$ and an integer $0 < I < +\infty$, there exist an integer N and a recurrent neural network of the form

$$x(k+1) = \sigma(Ax(k) + Bu(k)), \tag{2.21}$$

$$y(k) = Dx(k), \tag{2.22}$$

where $x \in R^N$ and $y \in R^n$, with an appropriate initial state $x(0)$ such that

$$\max_{0 \le k \le I} \|z(k) - y(k)\| < \epsilon. \tag{2.23}$$

(*Proof*) See Chapter 9.

2.5 Summary

In summary, this chapter has briefly reviewed some fundamental concepts on neural network learning. Two neural network architectures are presented and three objectives for neural learning are described. The approximate realization property of a recurrent neural network is also presented. In the next chapter, several techniques for neural learning will be reviewed.

Criteria	Definition
Akaike FPE	$\log E(\theta) + \log(1 + \frac{p}{N}) - \log(1 - \frac{p}{N})$
Akaike AIC	$\log E(\theta) + \frac{2p}{N}$
Akaike BIC, MDL	$\log E(\theta) + \frac{p \log N}{N}$
GCV	$\log E(\theta) - 2\log(1 - \frac{p}{N})$
Corrected AIC	$\log E(\theta) + \frac{2p}{N} + 2\frac{(p+1)(p+2)}{N-p-2}$
ϕ criterion	$\log E(\theta) + \frac{p \log\log N}{N}$

Table 2.3: Common criteria for model selection. FPE stands for final prediction error. GCV stands for generalized cross validation. AIC stands Akaike information criteria. BIC stands for Bayesian information criteria. MDL stands for minimum description length. $E(\theta)$ is the training error.

Chapter 3

Techniques for Neural Learning

In the last chapter, the general principle of neural learning has been introduced. Once an neural network architecture has been selected and the number of hidden units has been defined, the problem of neural learning will be equivalent to an optimization problem. The cost function is defined as a combination of the basic objective and the smoothing objective, and the goal of learning is simply to search for the minima of this cost function. In neural network literature, it is called *training*. Many existing numerical methods [69], such as gradient descent and Newton's method, can be applied. Where the complexity objective is taken into account, the problem will become complicated. One approach for solving this difficult problem is to incorporate a pruning procedure in conjunction with training. In this chapter, several techniques for solving these problems in neural learning will be presented.

3.1 Training Neural Network

Recall the objective of neural learning : *Given a set of real-valued input/output data pairs,* $\{(x(i), y(i)); i = 1, 2, \ldots .t\}$ *collected from an unknown system, devise an estimator to mimic the unknown environment as "precisely" as possible.* and one common objective to measure the "precision" is the mean squares error,

$$E_1(\theta) = \frac{1}{N}\sum_{i=1}^{N} \|\hat{y}(i|\theta) - y(i)\|^2. \tag{3.1}$$

Another common objective is $J(\theta) = E_1(\theta) + S(\theta)$, where $S(\theta)$ is a regularizer. Some exemplar regularizers, denoted by $S(\theta)$, are listed as follows.

Regularizer	Definition
Whittaker	$\sum_{i=1}^{N} \|\partial_\theta^k y(i|\theta)\|^2$
Tikhonov	$\int \|\partial_\theta^k y(u|\theta)\|^2 du$
Weight decay	$\|\theta\|^2$

Here, parameter N in Whittaker's regularizer denotes the total number of training data. Note that there are many other definitions for $S(\theta)$. Please refer to the last chapter or [33] for detail.

Considering $J(\theta)$, which is given by

$$J(\theta) = E_1(\theta) + S(\theta),$$

is the cost function for learning, the training problem will be equivalent to a non-linear programming problem. Hence, training a neural network can be accomplished by applying techniques in numerical method.

3.2 Gradient Descent Approach

Batch mode FNN training

The gradient descent method is the simplest training method, which is defined as follows :

$$\theta(k+1) = \theta(k) + \mu \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta(k)}, \tag{3.2}$$

where μ is the step size. If $S(\theta) = \frac{\lambda}{2\mu}||\theta||^2$, the updating equation can be rewritten as follows :

$$\theta(k+1) = (1-\lambda)\theta(k) + \frac{\mu}{N}\sum_{t=1}^{N} \frac{\partial \hat{y}(t,\theta)}{\partial \theta} e(t), \tag{3.3}$$

where $e(t) = y(t) - \hat{y}(t,\theta)$ and $\hat{y}$ is the output of the neural network. This is the batch mode training equation as all N training data are required.

On-line mode FNN training

In case the set of training data is not known in advance or the number of training data is too large, on-line gradient descent will be needed. The update equation for θ can be written as follows.

$$\theta(t+1) = \theta(t) + \mu_t \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta(t)} \tag{3.4}$$

for $J(\theta) = E_1(\theta)$ and μ_t satisfies the following conditions :

$$\sum_{t=1}^{\infty} \mu_t = \infty, \quad \sum_{t=1}^{\infty} \mu_t^2 < \infty.$$

If $S(\theta) = \frac{\lambda}{2\mu}||\theta||^2$, the updating equation can be rewritten as follows :

$$\theta(t+1) = (1-\lambda)\theta(t) + \mu_t \left. \frac{\partial \hat{y}(t,\theta)}{\partial \theta} \right|_{\theta=\theta(t)} e(t), \tag{3.5}$$

where $e(t) = y(t) - \hat{y}(t,\theta)$ and $\hat{y}$ is the output of the neural network.

This type of on-line method is also called the stochastic gradient descent method [65] and it has been discussed extensively in the area of signal processing and system modeling. The most well-known online training method is the back-propagation algorithm (BPA) [94].

Batch mode RNN training

The idea of training a RNN is essentially the same as training a FNN. Without loss of generality, a RNN can be written as a non-linear state-space model.

$$\begin{aligned} x(t,\theta) &= g(u(t), x(t-1,\theta), \theta), \\ \hat{y}(t,\theta) &= h(x(t,\theta)), \end{aligned}$$

where f and g are any non-linear vector function. Suppose $J(\theta) = E_1(\theta)$, the batch mode updating equation can readily be obtained as follows :

$$\frac{\partial \hat{y}}{\partial \theta} = \frac{\partial h(t)}{\partial \theta} + \frac{\partial h(t)}{\partial x(t)} \frac{\partial x(t)}{\partial \theta} \tag{3.6}$$

and $\frac{\partial x(t)}{\partial \theta}$ is obtained recursively via the following equation :

$$\frac{\partial x(t)}{\partial \theta} = \frac{\partial g(t)}{\partial \theta} + \frac{\partial g(t)}{\partial x(t-1)} \frac{\partial x(t-1)}{\partial \theta} \tag{3.7}$$

The updating equation for θ is given by

$$\theta(k+1) = \theta(k) + \frac{\mu}{N} \sum_{t=1}^{N} \frac{\partial \hat{y}(t,\theta)}{\partial \theta} e(t), \tag{3.8}$$

where $e(t) = y(t) - \hat{y}(t,\theta)$ and $\hat{y}$ is the output of the neural network. The factor $\frac{\partial \hat{y}}{\partial \theta}$ is obtained by Equations (3.6) and (3.7). In case,

$$J(\theta) = E_1(\theta) + \frac{\lambda}{2\mu} \|\theta\|^2,$$

$$\theta(k+1) = (1-\lambda)\theta(k) + \frac{\mu}{N} \sum_{t=1}^{N} \frac{\partial \hat{y}(t,\theta)}{\partial \theta} e(t),$$

On-line mode RNN training

The idea of on-line training RNN is also similar to on-line training a feedforward neural network,

$$\theta(t+1) = \theta(t) + \mu_t \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta(t)}, \tag{3.9}$$

where μ_t satisfies the following conditions :

$$\sum_{t=1}^{\infty} \mu_t = \infty, \quad \sum_{t=1}^{\infty} \mu_t^2 < \infty.$$

If $S(\theta) = \frac{\lambda}{2\mu}\|\theta\|^2$, the updating equation can be rewritten as follows :

$$\theta(t+1) = (1-\lambda)\theta(t) + \mu_t \left. \frac{\partial \hat{y}(t,\theta)}{\partial \theta} \right|_{\theta=\theta(t)} e(t), \tag{3.10}$$

where $e(t)$ is the output error. Back-Propagation Through Time (BPTT) [33] and Real-Time Recurrent Learning (RTRL) [33] are two well-known gradient descent training methods for recurrent neural networks.

3.3 Forgetting least squares

On-line FNN training

Let $y = f(x, \theta)$ be the transfer function of a single layer feedforward neural network where $y \in R$ is the output, $x \in R^m$ is the input and $\theta \in R^n$ is its parameter vector. Given a set of training data $\{x(i), y(i)\}_{i=1}^{N}$, let $\hat{\theta}(0)$ be the initial parametric vector, $P(0) = \delta^{-1} I_{n(m+2)\times n(m+2)}$, and the training of a feedforward neural network can be accomplished by the following recursive equations [48], [56], [57], [95]:

$$\begin{aligned} P(t) &= (I - L(t)H(t))\frac{P(t-1)}{1-\alpha} &(3.11)\\ \hat{\theta}(t) &= \hat{\theta}(t-1) + L(t)[y(x_t) - \hat{y}(x_t)], &(3.12)\end{aligned}$$

where

$$\begin{aligned} L(t) &= \frac{P(t-1)H(t)}{H^T(t)P(t-1)H(t) + (1-\alpha)}\\ H(t) &= \left.\frac{\partial f}{\partial \theta}\right|_{\theta=\hat{\theta}(t-1)}\end{aligned}$$

and α is the forgetting factor in between zero and one.

In the theory of system identification [43], the objective of the above recursive algorithm is to minimize the cost function $J(\theta(t))$, where

$$J(\theta(t)) = \sum_{k=1}^{t} w_k (y(x_k) - f(x_k, \theta(t)))^2 + \delta \|\theta(t)\|^2, \tag{3.13}$$

where $w_k = (1-\alpha)^{t-k}$.

- Note that $1 \geq w_i > w_j \geq 0$, for all $1 \leq i < j \leq t$. These weighting factors better capture the effect of the most recent training data. For $k = t$, the weighting on $(y(x_k) - f(x_k, \theta(t)))$ is one. When $k = t-1$, the weighting on $(y(x_k) - f(x_k, \theta(t)))$ is $(1-\alpha)$. This factor is smaller than one. As a result, the factor w_k serves as a weighting factor which counts the effect of the most recent training data more than the earlier ones.

- It should also be noted that an index t is associated with the parametric vector θ. This indicates that the best estimate of θ at time t might not be the same as the best estimate at time $t+1$.

- Once $\alpha = 0$, this algorithm will reduce to the standard recursive least squares method.

Since the model f is non-linear, FRLS can only be treated as a heuristic algorithm searching for the minimum $J(\theta)$. Fortunately, experimental studies always demonstrate that FRLS can give a good solution in neural network training and it converges much faster than the backpropagation approach. Theoretically, it is proved that FRLS has a regularization effect identical to that of weight decay. The proof will be presented in Chapter 10.

On-line RNN training

Puskorius & Feldkamp have recently extended the idea of FRLS to RNN training [90]. The formulation is actually a combination of gradient descent and FRLS. Suppose that RNN is defined as follows

$$\begin{aligned} x(t,\theta) &= g(u(t), x(t-1,\theta), \theta), \\ \hat{y}(t,\theta) &= h(x(t,\theta)), \end{aligned}$$

where f and g are any non-linear vector function, and

$$J(\theta(t)) = \sum_{k=1}^{t} w_k (y(k) - h(x(k), \theta(t)))^2 + \delta \|\theta(t)\|^2,$$

where $w_k = (1-\alpha)^{t-k}$.

Minimizing $J(\theta)$ via FRLS setting,

$$\begin{aligned} P(t) &= (I - L(t)H(t)) \frac{P(t-1)}{1-\alpha} \\ \hat{\theta}(t) &= \hat{\theta}(t-1) + L(t)[y(x_t) - \hat{y}(x_t)], \end{aligned}$$

where

$$\begin{aligned} L(t) &= \frac{P(t-1)H(t)}{H^T(t)P(t-1)H(t) + (1-\alpha)} \\ H(t) &= \left. \frac{\partial h}{\partial \theta} \right|_{\theta = \hat{\theta}(t-1)} \end{aligned}$$

and α is the forgetting factor in between zero and one. As $h(x(t,\theta))$ is a function of θ and $x(t)$, and $x(t)$ is a function of θ and $x(t-1)$, the computation of $H(t)$ will be given by

$$H(t) = \frac{\partial \hat{y}}{\partial \theta} = \frac{\partial h(t)}{\partial \theta} + \frac{\partial h(t)}{\partial x(t)} \frac{\partial x(t)}{\partial \theta}.$$

Here $\frac{\partial x(t)}{\partial \theta}$ can be obtained recursively via the following equation :

$$\frac{\partial x(t)}{\partial \theta} = \frac{\partial g(t)}{\partial \theta} + \frac{\partial g(t)}{\partial x(t-1)} \frac{\partial x(t-1)}{\partial \theta} \tag{3.14}$$

Since this setting does not concern the initial condition of the hidden layer, application of FRLS in RNN training will suffer from the initial hidden value problem. A detailed explanation of this effect will be presented in Chapter 4.

3.4 Extended Kalman filter based training

On-line FNN training

Let $y = f(x,\theta)$ be the transfer function of a single layer feedforward neural network where $y \in R$ is the output, $x \in R^m$ is the input and $\theta \in R^n$ is its parameter vector. Given a set of training data $\{x(i), y(i)\}_{i=1}^{N}$, the training of a neural network can be formulated as a filtering problem [2, 97] assuming that the data are generated by the following noisy signal model :

$$\begin{aligned} \theta(t) &= \theta(t-1) + v(t) && (3.15) \\ y(t) &= f(x(t), \theta(t)) + \epsilon(t) && (3.16) \end{aligned}$$

where $v(t)$ and $\epsilon(t)$ are zero mean Gaussian noise with variance $Q(t)$ and $R(t)$. A *good* estimation of the system parameter θ can thus be obtained via the extended Kalman filter method [37, 95, 90, 121] :

$$\begin{aligned} S(t) &= H^T(t)[P(t-1) + Q(t)]H(t) + R(t) && (3.17) \\ L(t) &= [P(t-1) + Q(t)]H(t)S^{-1}(t) && (3.18) \\ P(t) &= (I_{n \times n} - L(t)H(t))P(t-1) && (3.19) \\ \hat{\theta}(t) &= \hat{\theta}(t-1) + L(t)(y(t) - f(x(t), \hat{\theta}(t-1))) && (3.20) \end{aligned}$$

where $H(t) = \frac{\partial f}{\partial \theta}|_{\theta=\hat{\theta}(t-1)}$. For simplicity, we rewrite Equation (3.19) in the following form.

$$P^{-1}(t) = [P(t-1) + Q(t)]^{-1} + H(t)R^{-1}H^T(t). \qquad (3.21)$$

The EKF approach is an online mode training in that the weights are updated immediately after the presentation of a training pattern. The training methods are useful in that they do not require the storage of the entire input output history. With EKF algorithms, the learning speed is improved and the number of tuning parameters is reduced. Furthermore, EKF is able to track the time-varying parameter. This makes it particular useful in modeling a time-varying system.

On-line RNN Training

To train a recurrent neural network, we assume that the training data set is generated by a stochastic signal model as follows [123]:

$$\begin{aligned} x(t+1) &= g(Ax(t) + Bu(t) + D) + v(t), && (3.22) \\ y(t+1) &= Cx(t+1) + w(t), && (3.23) \end{aligned}$$

where $v(t)$ and $w(t)$ are zero mean Gaussian noise. If the parameters (A, B, C, D) are known, we can use the extended Kalman filter to predict the $y(t+1)$.

If the parameters are not known, we need to estimate them. In EKF [123], training a recurrent network is treated as a non-linear estimation problem, where the parameters $\{A, B, C, D\}$ and $x(t)$ are the unknown states being estimated. Hence, the state equations are :

$$\begin{aligned} x(t+1) &= g(A(t)x(t) + B(t)u(t) + D(t)) + v(t), && (3.24) \\ \theta(t+1) &= \theta(t) + e(t), && (3.25) \\ y(t) &= C(t)x(t) + w(t). && (3.26) \end{aligned}$$

Let θ be the collection of the state $\{A, B, C, D\}$. Put $x(t)$ and $\theta(t)$ as a single state vector, the state equations become :

$$\begin{aligned} \begin{bmatrix} x(t+1) \\ \theta(t+1) \end{bmatrix} &= g_1(x(t), u(t), \theta(t)) + \begin{bmatrix} v(t) \\ e(t) \end{bmatrix} && (3.27) \\ y(t) &= f_1(x(t), \theta(t)) + w(t), && (3.28) \end{aligned}$$

where

$$\begin{aligned} g_1(x(t), u(t), \theta(t)) &= \begin{bmatrix} g(A(t)x(t) + B(t)u(t) + D(t)) \\ \theta(t) \end{bmatrix} && (3.29) \\ f_1(x(t), \theta(t)) &= C(t)x(t). && (3.30) \end{aligned}$$

The simultaneous estimation of $x(t)$ and parametric vector $\theta(t)$ can be obtained recursively via the following recursive equations :

$$\begin{aligned} x(t|t-1) &= g(\hat{x}(t-1|t-1), u(t), \hat{\theta}(t-1)) && (3.31) \\ P(t|t-1) &= F(t-1)P(t-1|t-1)F^T(t-1) + Q(t-1) && (3.32) \end{aligned}$$

$$\begin{aligned} \begin{bmatrix} \hat{x}(t|t) \\ \hat{\theta}(t) \end{bmatrix} &= \begin{bmatrix} \hat{x}(t|t-1) \\ \hat{\theta}(t-1) \end{bmatrix} + L(t)\left(y^*(t) - H^T(t)\begin{bmatrix} \hat{x}(t|t-1) \\ \hat{\theta}(t-1) \end{bmatrix}\right) && (3.33) \\ P(t|t) &= P(t|t-1) - L(t)H^T(t)P(t|t-1), && (3.34) \end{aligned}$$

where

$$F(t+1) = \begin{bmatrix} \partial_x g(\hat{x}(t|t), u(t+1), \hat{\theta}(t)) & \partial_\theta g(\hat{x}(t|t), u(t+1), \hat{\theta}(t)) \\ 0_{n_\theta \times n} & I_{n_\theta \times n_\theta} \end{bmatrix}, \quad (3.35)$$

$$H^T(t) = [\partial_x^T y(t) \;\; \partial_\theta^T y(t)] \quad (3.36)$$

$$L(t) = P(t|t-1)H(t)[H^T(t)P(t|t-1)H(t) + R(t)]^{-1} \quad (3.37)$$

The initial $P^{-1}(0|0)$ is set to be zero matrix and $\hat{\theta}(0)$ is a small random vector. Given the data set $\{u(t), y^*(t)\}_{t=1}^N$ and iterating the above equations N times, the parametric vector $\hat{\theta}(N)$ will then be assigned as the network parameters.

Since the actual values of $Q(t)$ and $R(t)$ are not known in advance, they can be estimated recursively, as in Iiguni *et al.* (1992) :

$$\begin{aligned} R(t) &= (1-\alpha_R)R(t-1) + \alpha_R(y^*(t) - y(t|t-1))^2 && (3.38) \\ Q(t) &= (1-\alpha_Q)Q(t-1) + \alpha_Q L(t)L(t)^T(y^*(t) - y(t|t-1))^2, && (3.39) \end{aligned}$$

where α_R and α_Q are two small positive values.

3.5 Pruning Neural Network

As mentioned in the last chapter, pruning is a technique facilitating model selection. Classically, searching for the best model was achieved by trial-and-error. That is to say, if the performance of a ten hidden units neural network is not good enough, try fifteen hidden units. If the performance is still not satisfactory, try twenty units and so on. Obviously, this approach will require a large amount of computational resources and time. Pruning algorithms work by training a large size neural network and then redundant weights are removed. A survey of pruning algorithms can be found in [91]. The essential idea of pruning is to remove those weights which are not important. In convention, one important measure is defined by LeCun *et al.* [55], namely optimal brain damage.

In this dissertation, we only focus on pruning algorithms.

The essential idea of conventional pruning algorithms is to remove those weights which are not sensitive to the network performance once they are pruned away. Two well known algorithms are optimal brain damage (OBD) [55] and optimal brain surgeon (OBS) [31].

3.6 Optimal Brain Damage

In OBD, the weight importance measure is defined in terms of error sensitivity. Let $E_1(\hat{\theta})$ be the training error, where $\hat{\theta}$ is the parametric vector containing all the weight values. Mathematically, '*pruning the i^{th} weight*' means setting the value of this weight, $\hat{\theta}_i$ to zero. Let $\hat{\theta}$ be $[\hat{\theta}_1, \ldots, \hat{\theta}_{n\theta}]^T$ and $\hat{\theta}_i$ be $[\hat{\theta}_1, \ldots, \hat{\theta}_{i-1}, 0, \hat{\theta}_{i+1}, \ldots, \hat{\theta}_{n\theta}]^T$, '*pruning the i^{th} weight*' means setting

$$\hat{\theta} \Longrightarrow \hat{\theta}_{\backslash i}.$$

Therefore, the error sensitivity of the i^{th} parameter will be given by

$$S(\hat{\theta}_i) = E_1(\hat{\theta}_{\backslash i}) - E_1(\hat{\theta}),$$

where E_1 is the mean squares training error. Suppose $\hat{\theta}_i \approx 0$, expanding $E_1(\hat{\theta}_{\backslash i}))$ locally in Taylor series about $\hat{\theta}$ and ignoring higher order terms, $E_1(\hat{\theta}_{\backslash i})$ can be approximated as follows :

$$E_1(\hat{\theta}_{\backslash i} \approx E_1(\hat{\theta}) + \frac{\partial E_1(\hat{\theta})}{\partial \theta_i}\theta_i + \frac{1}{2}\hat{\theta}_i{}^2 \left(\frac{\partial^2 E_1(\hat{\theta})}{\partial \theta^2} \right)_{ii}. \tag{3.40}$$

Since $\frac{\partial E_1(\hat{\theta})}{\partial \theta_i} = 0$ once training is finished, the important measure for the i^{th} weight, denoted by $S(\hat{\theta}_i)$, can be approximated by

$$S(\hat{\theta}_i) \approx \frac{1}{2}\hat{\theta}_i{}^2 \left(\frac{\partial^2 E_1(\hat{\theta})}{\partial \theta^2} \right)_{ii}, \tag{3.41}$$

where $(A)_{ii}$ is the i^{th} diagonal element of the matrix A and $\hat{\theta} = [\hat{\theta}_1, \ldots, \hat{\theta}_{n_\theta}]^T$. n_θ is the total number of weights in the network. Then, the importance of weights can be ranked in accordance with the magnitude of their $S(\hat{\theta}_i)$. The one at the bottom of the ranking list is of least importance and is removed first.

Several remarks have to be made about in using such an OBD pruning algorithm in selecting the best neural network model.

1. As the Taylor expansion of $E_1(\hat{\theta}_{\setminus i})$ assumes that $\hat{\theta}_i \approx 0$, the approximation of the error sensitivity will not be accurate for large values of $\hat{\theta}_i$.
2. Once a weight has to be pruned away, it is necessary to re-train the network since

$$\hat{\theta} = \arg\min_{\theta} \{E_1(\theta)\} \not\Longrightarrow \hat{\theta}_{\setminus i} = \arg\min_{\theta_{\setminus i}} \left\{E_1(\theta_{\setminus i})\right\}.$$

3. If we let $\hat{\theta}_{\setminus i}^{retrain}$ be the minima of $E_1(\theta)$ after re-training,

$$E_1(\hat{\theta}_{\setminus i}) < E_1(\hat{\theta}_{\setminus j}) \not\Longrightarrow E_1(\hat{\theta}_{\setminus i}^{retrain}) < E_1(\hat{\theta}_{\setminus j}^{retrain}).$$

4. As the solution $\hat{\theta}$ depends on the training method being employed and the objective function being minimized, the error sensitivity measure is training method and objective function dependent.
5. Since a pruning procedure can only be carried out if the training is finished, a poor training method will delay the pruning process.

3.7 Optimal Brain Surgeon

In order to alleviate the problem aroused from the second and third remarks, Hassibi & Stork [31] proposed an alternative importance measure, namely optimal brain surgeon (OBS). The basic idea of OBS is the same as OBD except that it includes the concept of re-training. Suppose the surface of the error function $E_1(\theta)$ is locally quadratic around θ, the error sensitivity of the i^{th} parameter is defined as follows :

$$S(\hat{\theta}_i^{retrain}) = E_1(\hat{\theta}_{\setminus i}^{retrain}) - E_1(\hat{\theta}). \tag{3.42}$$

The parametric vector $\hat{\theta}_i^{retrain}$ is obtained by solving the following constraint optimization problem :

$$\begin{array}{ll} \text{Minimize} & E_1(\theta) \\ \text{Subject to} & \theta_i = 0. \end{array}$$

Applying the technique of Lagrange multiplier, the above problem can be solved by minimizing the following function.

$$J(\theta) = E_(\hat{\theta}) + \frac{1}{2}\left(\theta - \hat{\theta}_i\right)^T \frac{\partial^2 E_1(\hat{\theta})}{\partial \theta^2}\left(\theta - \hat{\theta}_i\right) + \vec{\lambda}^T \theta, \tag{3.43}$$

where $\vec{\lambda}$ is the Lagrange multiplier vector. With certain mathematical manipulation,

$$S(\hat{\theta}_i^{retrain}) \approx \frac{1}{2}\frac{\hat{\theta}_i^2}{\left(\frac{\partial^2 E_1(\hat{\theta})}{\partial \theta^2}^{-1}\right)_{ii}}. \tag{3.44}$$

where $(A^{-1})_{ii}$ is the i^{th} diagonal element of the inverse of matrix A.

3.8 FRLS based Pruning

As indicated in the list of remarks about OBD, the actual effectiveness of a pruning algorithm relies pretty much on the training method being used. When backpropagation is applied to train a neural network, the total time taken for pruning will be very long as the training process takes a long time to finish. The forgetting recursive least squares method is a well-known *fast* on-line training method [48, 6, 90], the estimation of such an importance measure can thus be obtained effectively.

Recall that the matrix $P(t)$ in FRLS is calculated recursively by the following equations.

$$\begin{aligned} P(t) &= (I - L(t)H(t))\frac{P(t-1)}{1-\alpha} \\ L(t) &= \frac{P(t-1)H(t)}{H^T(t)P(t-1)H(t) + (1-\alpha)}, \end{aligned}$$

where

$$H(t) = \frac{\partial f(\hat{\theta}(t-1))}{\partial \theta}.$$

Applying matrix inversion lemma, it is readily seen that

$$\begin{aligned} P^{-1}(t) &= (1-\alpha)P^{-1}(t-1) + H^T(t)H(t) \\ &= (1-\alpha)P^{-1}(t-1) + \frac{\partial f(\hat{\theta}(t-1))}{\partial \theta}\frac{\partial f(\hat{\theta}(t-1))}{\partial \theta}^T. \end{aligned}$$

Therefore, the matrix $P^{-1}(N)$ can be given by

$$\begin{aligned} P^{-1}(N) &\approx P^{-1}(0) + \sum_{t=1}^{N}(1-\alpha)^{N-t}H(t)H^T(t) \\ &\approx P^{-1}(0) + \frac{1}{\alpha}\left[\frac{1}{N}\sum_{t=1}^{N} H(t)H^T(t)\right] \end{aligned}$$

Therefore the OBD type importance measure can be obtained by the following formula.

$$S(\hat{\theta}_i) = \alpha\frac{\hat{\theta}_i^2}{2}\left(P^{-1}(N) - P^{-1}(0)\right)_{ii}.$$

Applying matrix inversion lemma, the OBS type importance measure can be obtained by the following formula.

$$S(\hat{\theta}_i^{retrain}) = \alpha\frac{\hat{\theta}_i^2}{2}\frac{1}{(P(N) - P(N)(P(N) - P(0))^{-1}P(N))_{ii}}.$$

If $P^{-1}(0)$ is very small,

$$S(\hat{\theta}_i^{retrain}) = \alpha\frac{\hat{\theta}_i^2}{2}\left(P(N)\right)_{ii}.$$

One major advantage of using such formulae is that weight importance can be estimated without waiting for the whole training process to be completed.

3.9 Procedure

The pruning procedure for both optimal brain damage and optimal brain surgeon are the same. It is summarized as follows :

Algorithm 3.1 (Pruning Procedure)

```
1  Start with a neural network of large size.
2  Train the neural network.
3  Evaluate the weights' importance according to their sensitivity measures.
4  Define the ranking as $\{\pi_1, \pi_2, \cdots, \pi_{n\theta}\}$.
5  $k = 1$
6  while the validation error is less than a threshold.
7      Set $\theta_{\pi_k} = 0$.
8      Evaluate the validation error
9      $k \leftarrow k + 1$
10 end while
```

The sensitivity measure can be defined as that in Equation (3.41) or Equation (3.44). It should be remarked that error sensitivity is just one type of measure. In a general setting, the sensitivity can be defined with respect to any cost function. For example, a cost function can consist of the error term and a smoothing regularizer, [126, 127].

3.10 Summary

This chapter has reviewed several essential training and pruning techniques for neural network learning. Besides, an on-line evaluation of a weight importance measure using FRLS has also been presented. Although these techniques have been widely applied for feedforward neural network learning, there are limitations when they are applied to recurrent neural networks. The next chapter will describe some of these issues.

Chapter 4

Limitation of Existing Techniques

In the last two chapters, the basis of neural network learning, and several training and pruning techniques have been reviewed. In this chapter, we will discuss certain limitations of these techniques and give a summary of those techniques.

4.1 Gradient descent and FRLS in RNN training

Training a RNN using gradient descent or FRLS approach usually works fine when the training set is defined as $\{u_i, y_i\}_{i=1}^{N}$ and the validation set is defined as $\{u_i, y_i\}_{i=N+1}^{N+T}$. However, in case the validation set is defined as $\{u_i, y_i\}_{i=1}^{T}$ and the training set is defined as $\{u_i, y_i\}_{i=T+1}^{N+T}$ (or the form of the training input is different from the validation input), it might happen that the validation error would be much larger than the training error. This phenomenon is due to the assumption that the initial condition $x(0)$ is zero vector. As a matter of fact, this information and even $x(1)$, $x(2)$ and so on, are not known in advance.

The reason why gradient descent or FRLS is able to facilitate RNN training, can be explained as follows. Without loss of generality, we only give the reason for gradient descent. Recall that a RNN is defined by

$$\begin{aligned} x(t) &= g(x(t-1), \theta) \\ y(t) &= h(x(t), \theta), \end{aligned}$$

where θ is the weight vector governing the behavior of the RNN. Using gradient descent, the weight updating can be accomplished by the following equation :

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \mu \sum_{t=1}^{N} \frac{\partial h(\hat{x}(t), \hat{\theta}(k-1))}{\partial \theta} e(t), \tag{4.1}$$

where μ is the update step size and

$$\hat{x}(t, \hat{\theta}(k-1)) = g(\hat{x}(t-1), \hat{\theta}(k-1)) \tag{4.2}$$

$$e(t, \hat{\theta}(k-1)) = y(t) - \hat{y}(t) = y(t) - h(\hat{x}(t), \hat{\theta}(k-1). \tag{4.3}$$

Considering Equation (4.2), $\hat{x}(t, \hat{\theta}(k-1))$ and $\hat{y}(t, \hat{\theta}(k-1))$ can then be rewritten as follows :

$$\hat{x}(t, \hat{\theta}(k-1)) = \underbrace{go \cdots og}_{t \text{ times}}(x(0), \hat{\theta}(k-1)) \tag{4.4}$$

$$\begin{aligned} &= \mathcal{G}_t(x(0), \hat{\theta}(k-1)) && (4.5)\\ \hat{y}(t, \hat{\theta}(k-1)) &= h(\mathcal{G}_t(x(0), \hat{\theta}(k-1))) = \mathcal{H}_t(x(0), \hat{\theta}(k-1)). && (4.6)\end{aligned}$$

Similar, the gradient of $\hat{x}(t, \hat{\theta}(k-1))$ and $\hat{y}(t, \hat{\theta}(k-1))$ can be written as follows :

$$\begin{aligned} \nabla_\theta \hat{x}(t, \hat{\theta}(k-1)) &= \nabla_\theta \mathcal{G}_t(x(0), \hat{\theta}(k-1)) && (4.7)\\ \nabla_\theta \hat{y}(t, \hat{\theta}(k-1)) &= \nabla_\theta \mathcal{H}_t(x(0), \hat{\theta}(k-1)). && (4.8)\end{aligned}$$

Suppose that μ is small,

$$\begin{aligned} \mathcal{G}_t(x(0), \hat{\theta}(k-1)) &\approx \mathcal{G}_t(x(0), \hat{\theta}(k-2)) + \nabla_\theta \mathcal{G}_t(x(0), \hat{\theta}(k-2))(\hat{\theta}(k-1) - \hat{\theta}(k-2))\\ &= \hat{x}(t, \hat{\theta}(k-2)) + \mu \nabla_\theta \mathcal{G}_t(x(0), \hat{\theta}(k-2)) \times\\ &\quad \sum_{t=1}^{N} \nabla_\theta \mathcal{H}_t(x(0), \hat{\theta}(k-2)) e(t, \hat{\theta}(k-2)).\end{aligned}$$

For μ is small, both $\hat{\theta}(k)$ and $\{\hat{x}(t)\}_{t=1}^{N}$ are updated simultaneously,

$$\begin{aligned} \hat{\theta}(k) &= \hat{\theta}(k-1) + \mu \sum_{t=1}^{N} \nabla_\theta \mathcal{H}_t(x(0), \hat{\theta}(k-1)) e(t, \hat{\theta}(k-1)), && (4.9)\\ \hat{x}(t, \hat{\theta}(k)) &= \hat{x}(t, \hat{\theta}(k-1)) + \mu \nabla_\theta \mathcal{G}_t(x(0), \hat{\theta}(k-1)) \times && (4.10)\\ &\quad \sum_{t=1}^{N} \nabla_\theta \mathcal{H}_t(x(0), \hat{\theta}(k-1)) e(t, \hat{\theta}(k-1)), && (4.11)\end{aligned}$$

in order to minimize

$$\frac{1}{N} \sum_{t=1}^{N} e^2(t).$$

Once training is finished,the parametric vector $\hat{\theta}$ and $\hat{x}(N)$ will be frozen. $\hat{x}(N+1)$, $\hat{x}(N+2)$ and so on will be estimated by the following recursive equation :

$$\hat{x}(t) = g(\hat{x}(t-1), \hat{\theta}(N))$$

for $t > N$. Since the error feedback correction for $x(t)$ will also stop, the output error $(y(t) - \hat{y}(t))$ for all $t > N$ will gradually increase if the trained RNN is used. In sequel, without a proper feedback mechanism for updating $x9t)$ after training, a RNN is unable to be applied in a long run.

Besides, if the trained RNN is not applied immediately after training, the performance will be even worse. This can be explained by the following example.

Example 1 *Suppose the true system is deterministic :*

$$\begin{aligned} x(t+1) &= \tanh(Ax(t)) && (4.12)\\ y(t) &= Cx(t), && (4.13)\end{aligned}$$

where $x(t) \in R^3$, $y(t) \in R$,

$$A = \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 0.9 & 3 \\ 0 & 3 & 0.6 \end{bmatrix}.$$

$$C = [0 \;\; 0 \;\; 1].$$

A recurrent neural network is applied to learn system behavior. Assuming that a RNN with three hidden units is trained perfectly and $\hat{x}(N)$ is equal to the true value, the recurrent weight matrix and the output weight vector are exactly equal to A and C respectively. Suppose that

$$\hat{x}(N) = x(N) = (-0.5, -0.3, 0.4).$$

The RNN is not used immediately but t_0 steps later. The system state $x(N + t_0)$ is $(-0.5, 0.3, -0.4)$. Then we plug in the RNN to do the prediction. Figure 4.1 to Figure 4.3 compare the evolution of the state values between the true system and the trained RNN.

It can be easily observed that, except for $x_1(t)$, the values of the rest of the other units (and output) deviate a lot from the true system.

This explains why the use of gradient descent approach in RNN training is not effective. This also brings out the reason why EKF based training is more effective compared with these types of training methods.

4.2 Error sensitivity based pruning for RNN

When a gradient descent trained RNN is pruned, the validity of using the error sensitivity approach for measuring the importance of a weight will not be appropriate. After a weight is pruned, the output of a pruned RNN is given by

$$\begin{aligned}
y_{rnn}(1) &= h_{rnn} \circ g_{rnn}(x_{rnn}(0), \theta_p) \\
y_{rnn}(2) &= h_{rnn} \circ g_{rnn} \circ g_{rnn}(x_{rnn}(0), \theta_p) \\
&\cdots \\
y_{rnn}(N) &= h_{rnn} \circ g_{rnn}(\circ g_{rnn})^{N-1}(x_{rnn}(0), \theta_p)
\end{aligned}$$

where θ is the augmented vector which contains all the weight parameters of the RNN. The augmented vector of the pruned RNN is denoted θ_p where the i^{th} element of θ_p will be equal to the i^{th} element of θ if the i^{th} weight has not been pruned away and otherwise the i^{th} element of θ_p will be equal to zero. The error of such a pruned RNN is defined as

$$\frac{1}{T}\sum_{t=1}^{T}(y(t) - y_{rnn}(t, \theta_p))^2 = \frac{1}{T}\sum_{t=1}^{T}(y(t, x(0)) - y_{rnn}(t, \theta_p, x_{rnn}(0)))^2$$

Due to the composite weight removal and initialization error effect, it is even harder to identify how much the validation error is due to the weight removal and how much is due to initialization error.

4.3 Error sensitivity based pruning in non-stationary environments

A shortcoming of using the sensitivity based pruning methods, such as OBD and OBS, is that the error sensitivity term can only be obtained after training is finished. Once the

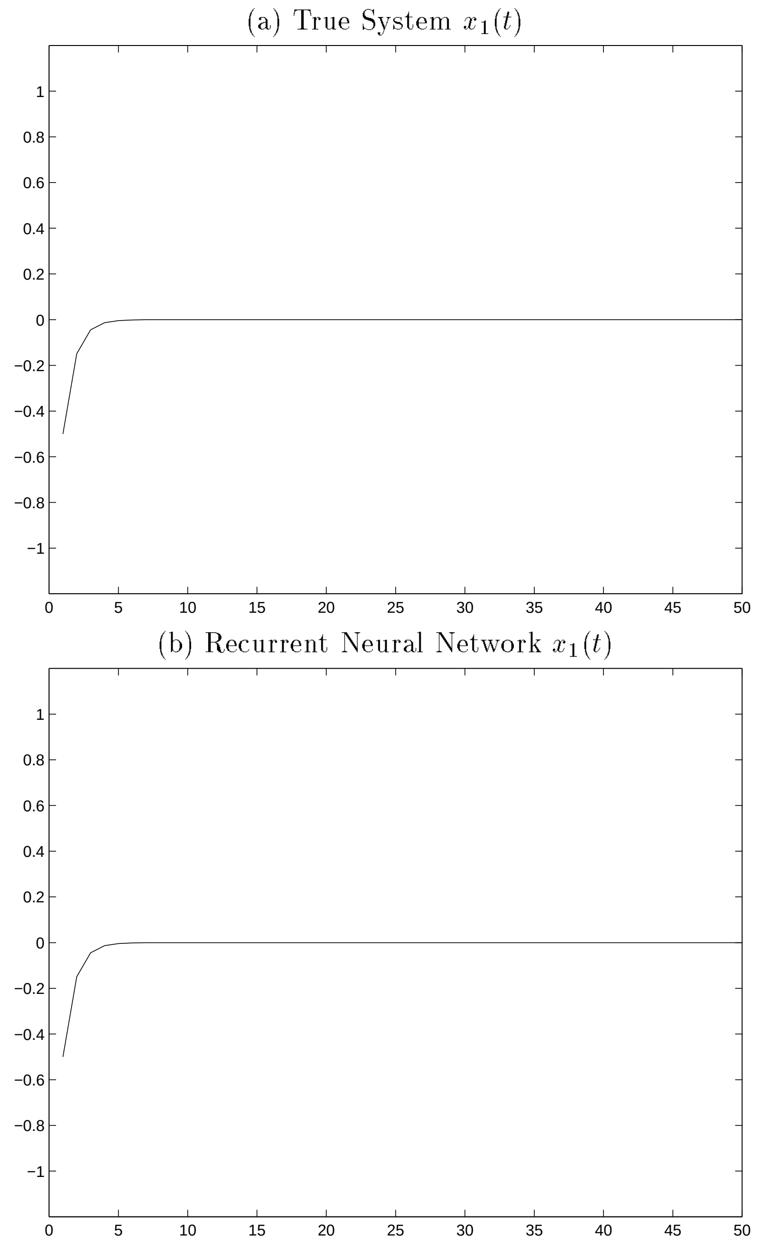


Figure 4.1: Comparison of the value of $x_1(t)$ for the true system and the recurrent neural network. The horizontal axis corresponds to time $t \geq 0$ while the vertical axis corresponds to the value of $x_1(t)$.

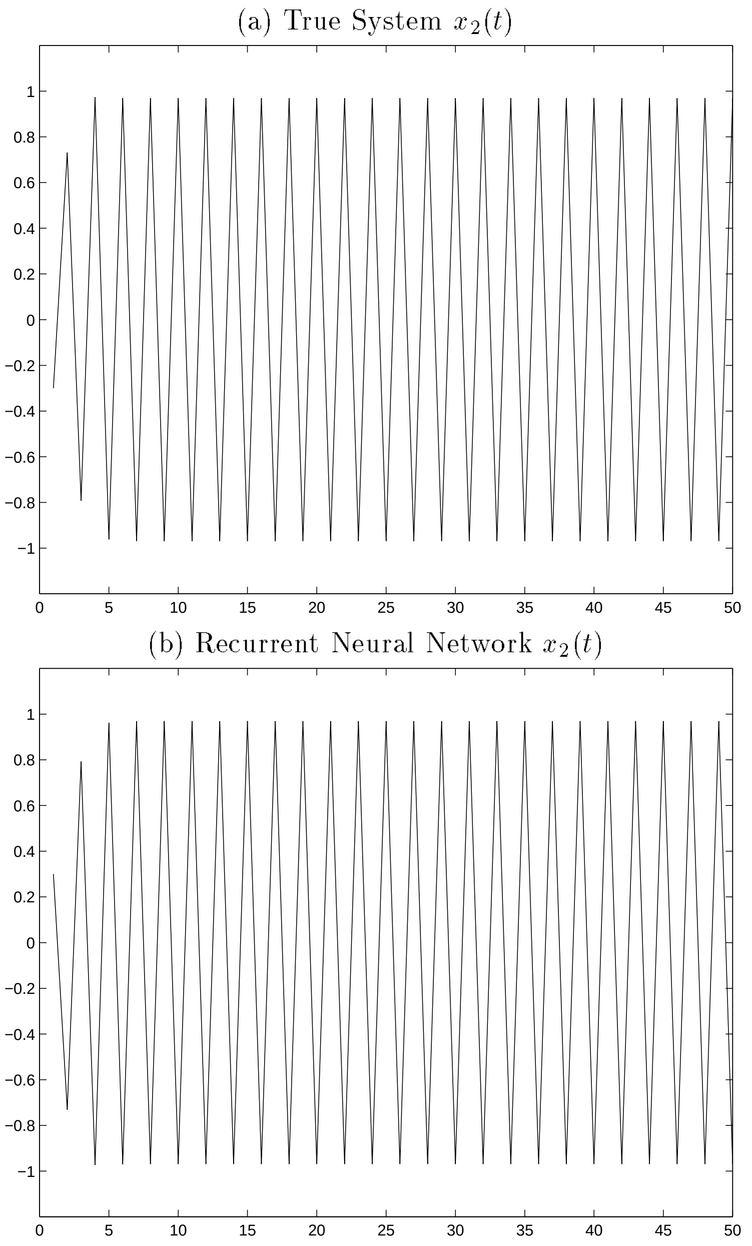


Figure 4.2: Comparison of the value of $x_2(t)$ for the true system and the recurrent neural network. The horizontal axis corresponds to time $t \geq 0$ while the vertical axis corresponds to the value of $x_2(t)$.

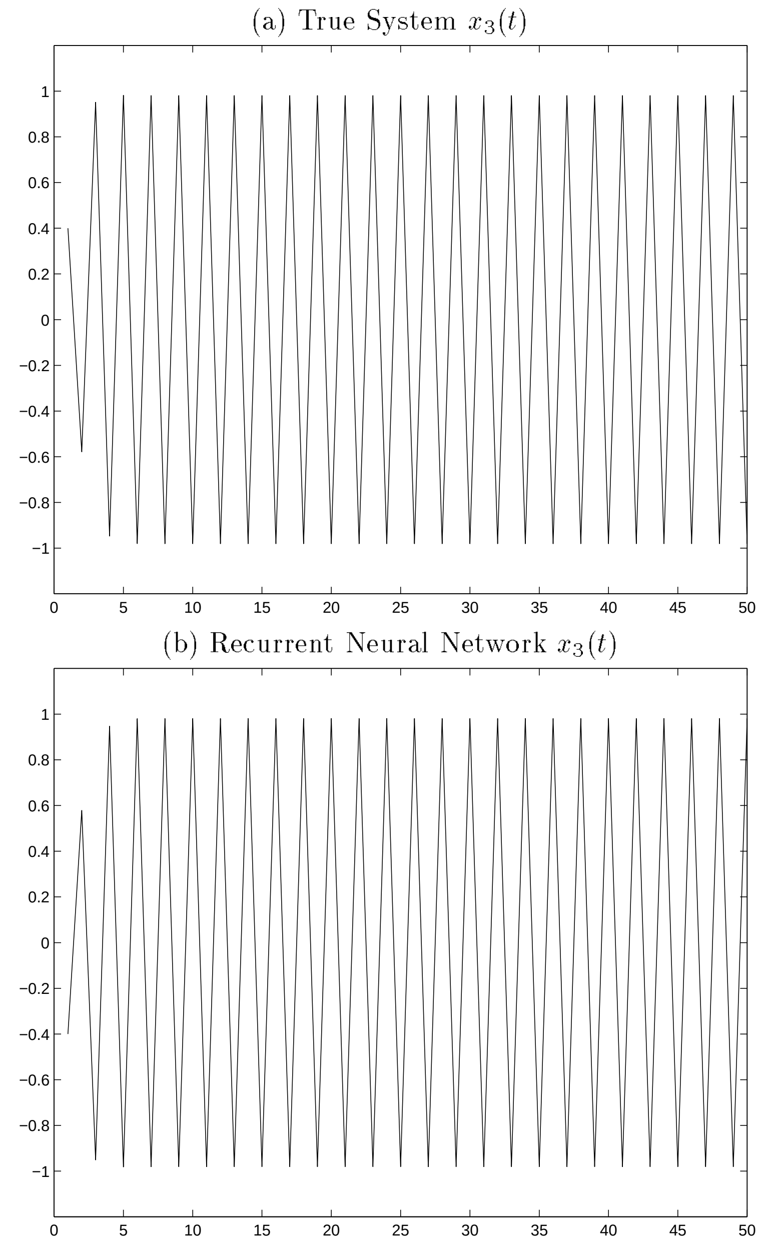


Figure 4.3: Comparison of the value of $x_3(t)$ for the true system and the recurrent neural network. The horizontal axis corresponds to time $t \geq 0$ while the vertical axis corresponds to the value of $x_3(t)$.

training is finished, an extra data set has to be passed to the neural network for obtaining the Hessian matrix. Suppose that the cost function $J(\theta)$ is defined as the combination of sum-squares-error and a regularization term $S(\theta)$, i.e.

$$J(\theta) = S(\theta) + \frac{1}{N_v}\sum_{x=1}^{N_v}(y(x) - \hat{y}(x,\theta))^2,$$

the Hessian matrix $\frac{\partial^2 J(\theta)}{\partial \theta^2}$ is given by the following equation.

$$\frac{\partial^2 J(\theta)}{\partial \theta^2} = \frac{\partial^2 S(\theta)}{\partial \theta^2} - \frac{2}{N_v}\sum_{x=1}^{N_v}(y(x) - \hat{y}(x,\theta))\frac{\partial^2 \hat{y}(x,\theta)}{\partial \theta^2} + \frac{2}{N_v}\sum_{x=1}^{N_v}\frac{\partial \hat{y}(x,\theta)}{\partial \theta}\frac{\partial \hat{y}(x,\theta)}{\partial \theta}^T.$$

In case the training method converges slowly, backpropagation for instance, re-training processes are quite time-consuming, especially when the training method converges slowly. The actual time for obtaining a good network structure may thus be quite long. If the nature of the problem is *non-stationary*, it will be much more difficult to implement such a pruning method since training is never finished and then the ranking of the weight importance obtained at time t might not be equal to the weight importance obtained at other times. That is

$$S(\hat{\theta}_i(t)) < S(\hat{\theta}_j(t))$$

might not imply that

$$S(\hat{\theta}_i(t+1)) < S(\hat{\theta}_j(t+1)).$$

This will make the pruning of a neural network using the sensitivity approach in a non-stationary environment more difficult.

4.4 Summary of the techniques for neural network learning

Taking these limitations into consideration, it is obvious that no single training method can outperform all the others. Table 4.1 to Table 4.4 summarize the essential properties of five training methods which are presented in Chapter 2.

According to Table 4.1, the most flexible training method for FNN is gradient descent since it can be applied to both stationary and non-stationary environments with any objective function but its training speed is slow.

According to Table 4.2, for recurrent neural network training, the extended Kalman filter approach is the most appropriate method since it is able to train a RNN no matter whether $x(0)$ is given or not given. Besides, its training speed is fast.

According to Table 4.3, pruning RNN with $x(0)$ not given is a challenging problem since not all training methods can be applied to train RNN if $x(0)$ is not given.

Table 4.4 summarizes the relation between different training methods and their generalization abilities. As there is no restriction on the objective function, training a neural network using the gradient descent approach is able to improve the generalization ability if an appropriate objective function is defined. Since the objective of the RLS training method is

$$\sum_{t=1}^{N} e^2(t) + \theta^T P^{-1}(0)\theta,$$

Training methods	Problem nature	Objective function	Training mode	One-step computation complexity	Training speed
GD	S	Any	Batch	Low	Slow
SGD	S/NS	Any	On-line	Low	Slow
RLS	S	Error	On-line	High	Fast
FRLS	S/NS	Error	On-line	High	Fast
EKF	S/NS	$\mathcal{P}(\theta \vert Y^N)$	On-line	High	Fast

Table 4.1: Summary of various training methods for FNN. 'S' means stationary while 'NS' means non-stationary.

Methods	$x(0)$ given	$x(0)$ not given
GD	Slow	-
SGD	Slow	-
RLS	Fast	-
FRLS	Fast	-
EKF	Fast	Fast

Table 4.2: Comparison of the speed of different training methods for RNN.

RLS training might help improving the network generalization ability. The factor $\theta^T P^{-1}(0)\theta$ is equivalent to the so-called weight decay regularizer. If the size of training data is small, we can control $P^{-1}(0)$ in order to improve the generalization ability. If the number of training data is very large, the weight decay effect will be lost. The generalization ability of FRLS and EKF training methods are more difficult to determine since it is not easy to decompose their objective functions in the form of

$$\text{mean squares error} + \text{smooth regularizer}.$$

In Chapter 10, we will start from the first principle to show that FRLS has a similar effect to weight decay.

4.5 Summary

This chapter has presented several limitations on the existing techniques for neural network learning and a summary of those existing training methods is given. From the summary, it is found that in order to prune a RNN effectively, an extended Kalman filter based pruning method is useful. In Part II, we will derive two weight importance measures for use in FNN pruning. Both measures are in terms of $\hat{\theta}$ and $P(N)$. In Part III, we extend this idea to the case of RNN. A similar measure will also be derived for measuring the weight importance. Extensive simulation results will be presented in order to confirm the effectiveness of these measures.

Methods	FNN	RNN $x(0)$ given	RNN $x(0)$ not given
GD	$\surd$	$\surd$	$\times$
SGD	$\surd$	$\surd$	$\times$
RLS	$\surd$	$\surd$	$\times$
FRLS	$\surd$	$\surd$	$\times$
EKF	?	?	?

Table 4.3: Relation between different training methods and their pruning abilities. '$\surd$' means the method is able to facilitate pruning, '$\times$' means the method is unable to facilitate pruning while '?' means unknown.

Methods	FNN	RNN $x(0)$ given	RNN $x(0)$ not given
GD	$\surd$	$\surd$	$\times$
SGD	$\surd$	$\surd$	$\times$
RLS	$\surd$	$\surd$	$\times$
FRLS	?	?	$\times$
EKF	?	?	?

Table 4.4: Relation between different training methods and their generalization abilities. '$\surd$' means the method is able to improve generalization, '$\times$' means the method is unable to improve generalization while '?' means unknown.

Part II

Pruning Algorithms for FNN

Chapter 5

EKF based On-Line Pruning for FNN

In Part I, the basis of neural network learning has been presented. We have also presented several limitations of existing training and pruning techniques and brought out the advantages of using the extended Kalman filter approach in neural network training. In this chapter, we will present some results connecting the extended Kalman filter and neural network pruning. Specifically, we would like to present how those results obtained by using the extended Kalman filter training method can be applied to measure the importance of a weight in a network and give simulated examples to illustrate this idea.

5.1 Use of matrix P in the evaluation of weight saliency

It should be noted that after training with EKF, the information that we have is that *i) the parametric vector $\hat{\theta}$ and ii) the covariance matrix P.* Suppose that the weight parameter and the covariance matrix P both converge, the matrix $Q(t)$ is a constant diagonal matrix denoted by Q_0, where

$$Q_0 = qI,$$

and

$$R = 1,$$

we can readily establish the asymptotic behavior for matrix P :

$$P_\infty^{-1} = [P_\infty + Q_0]^{-1} + H(t)H^T(t), \tag{5.1}$$

where

$$\begin{aligned} H(t) &= \left.\frac{\partial f}{\partial \theta}\right|_{\theta=\hat{\theta}(t-1)}, \\ P_\infty &= \lim_{t\to\infty} P(t). \end{aligned}$$

Further assuming that there exists a time t_0 such that for all time $t > t_0$, all $P(t)$ will be the limiting matrix P_∞, it is possible to deduce that

$$\frac{1}{N-t_0}\sum_{t=t_0+1}^{N} P_\infty^{-1} - [P_\infty + Q_0]^{-1} = \frac{1}{N-t_0}\sum_{t=t_0+1}^{N} H(t)H(t)^T \tag{5.2}$$

When N is large, the second term on the right hand side will approach to the expectation of $H(t)H^T(t)$ which is dependent on the input x. Hence,

$$P_\infty^{-1} = [P_\infty + Q_0]^{-1} + E[H(t)H^T(t)] \tag{5.3}$$

Since Q_0 is positive definite, it is readily shown that P_∞ is also positive definite. Thus, we can decompose[1] the matrix P_∞ by

$$P_\infty = UDU^T,$$

where D is the diagonal matrix with the eigenvalues of P_∞ as the elements, namely $\lambda_1, \lambda_2, \ldots, \lambda_{n_\theta}$ and the matrix U contains the corresponding eigenvectors. Under such decomposition, (5.3) can be rewritten as follows :

$$U\left[D^{-1} - (D+qI)^{-1}\right]U^T = E[H(t)H^T(t)]. \tag{5.4}$$

Here, $E[x]$ means the expectation of x. Now let β_k be the k^{th} diagonal element of the matrix $[D^{-1} - (D+qI)^{-1}]$,

$$\beta_k = \frac{1}{\lambda_k} - \frac{1}{\lambda_k + q} = \frac{q}{\lambda_k(\lambda_k+q)}. \tag{5.5}$$

Two special cases can thus be obtained :

$$\beta_k \approx \begin{cases} \lambda_k^{-1} & \text{if } q \gg max\{\lambda_k\} \\ q\lambda_k^{-2} & \text{if } q \ll min\{\lambda_k\} \end{cases} \tag{5.6}$$

Empirically, we have found that λ_ks are usually much larger than q. So in the rest of the paper, we assume that $\beta_k \approx q/\lambda_k^2$.

It should be noted that $P_\infty^{-1} = UD^{-1}U^T$; we can then put the values of β_k into Equation (5.3) and get an approximated equation :

$$P_\infty^{-2} \approx q^{-1}E[H(t)H^T(t)]. \tag{5.7}$$

Practically, we cannot know $E[H(t)H^T(t)]$, so we approximate it by the mean average $\sum_t H(t)H^T(t)$. Putting back the definition of $H(t)$, we get that

$$E\left[\frac{\partial f}{\partial \theta_k}^2\right] \approx q(P_\infty^{-2})_{kk}. \tag{5.8}$$

Here $(A)_{kk}$ denotes the k_{th} diagonal element of a matrix A. With this, the expected change of error, denoted by E_k, due to the setting of θ_k being zero,can be discussed.

[1]Here, we assume that all the eigenvalues of the matrix P_∞ are distinct.

5.1.1 Single pruned weight

Recall that the *true* function is defined as a non-linear regressor, $y(x) = f(x, \theta_0) + noise$, with parameter θ_0. After training, an approximation, $\hat{y}(x) = f(x, \hat{\theta})$ is obtained. We drop the subscript N from $\hat{\theta}$ for simplicity. Let $\hat{\theta}_1, \ldots, \hat{\theta}_{n_\theta}$ be the elements of $\hat{\theta}$, and σ^2 be the variance of the output noise, the expected predicted square error (EPSE) of this approximated function would be given by

$$\begin{aligned} E[(y(x) - \hat{y}(x))]^2 &= E[(f(x, \theta_0) + noise - f(x, \hat{\theta}))^2] &(5.9)\\ &= E[(f(x, \theta_0) - f(x, \hat{\theta}))^2] + \sigma^2 &(5.10)\end{aligned}$$

for $\hat{\theta} \to \theta_0$. It should be remarked that the expectation is taken over to future data.

Now consider that the k^{th} element of $\hat{\theta}$ is being set to zero; let $\hat{y}_p(x)$ be the approximated function and $\hat{\theta}_p$ be the corresponding parametric vector, we can have the EPSE given by

$$E[(y(x) - \hat{y}_p(x))]^2 = E[(f(x, \theta_0) + noise - f(x, \hat{\theta}_p))^2], \tag{5.11}$$

As the difference $(y(x) - \hat{y}_p(x))$ can also be decomposed as follows :

$$\begin{aligned} y(x) - \hat{y}_p(x) &= [f(x, \theta_0) + noise - f(x, \hat{\theta})] \\ &+ [f(x, \hat{\theta}) - f(x, \hat{\theta}_p)]. &(5.12)\end{aligned}$$

In case that $\hat{\theta}$ is already very close to θ_0, the first term on the right side will be a Gaussian noise term which is independent of the second term. This implies that

$$\begin{aligned} E[(y(x) - \hat{y}_p(x))^2] &\approx E[(f(x, \theta_0) - f(x, \hat{\theta}))^2] + \sigma^2 \\ &+ E[(f(x, \hat{\theta}) - f(x, \hat{\theta}_p))^2]. &(5.13)\end{aligned}$$

When the magnitude of θ_k is small, the third term on the right hand side of (5.13) can be expanded in Taylor series and thus the expected predicted square error can be approximated by

$$E[(y(x) - \hat{y}_p(x))^2] \approx E[(f(x, \theta_0) - f(x, \hat{\theta}))^2] + \sigma^2 + E\left[\frac{\partial f(x, \hat{\theta})}{\partial \theta_k}^2\right] \hat{\theta}_k^2. \tag{5.14}$$

Comparing (5.14) with (5.10), it is observed that the last term is the expected error increment due to pruning the k^{th} element of $\hat{\theta}$. Using equation (5.8), we can now relate the matrix P_∞ and the parametric vector $\hat{\theta}$ in the following manner :

$$\Delta E_k = E\left[\frac{\partial f(x, \hat{\theta}_0)}{\partial \theta_k}^2\right] \hat{\theta}_k^2 \approx q(P_\infty^{-2})_{kk} \theta_k^2. \tag{5.15}$$

Here, we use the notation ΔE_k to denote the incremental change of the expected prediction error due to pruning the k^{th} element, that is the expected prediction error sensitivity. Using (5.15), the weights' importance can be defined.

5.1.2 Multiple pruned weights

With the use of (5.15), the weight importance can thus be ranked in accordance with their ΔE_k and the ranking list is denoted by $\{\pi_1, \pi_2, \ldots, \pi_{n_\theta}\}$, where $\Delta E_{\pi_i} \leq \Delta E_{\pi_j}$ if $i < j$. If we let $\hat{\theta}_a$ be a vector which $\pi_1, \ldots, \pi_k$ elements being zeros and other elements being identical to the corresponding elements in $\hat{\theta}$, we can estimate the incremental change of mean prediction error by the following formula :

$$\Delta E_{[\pi_1, \pi_k]} \approx q \hat{\theta}_a^T (P_\infty^{-2}) \hat{\theta}_a. \tag{5.16}$$

With this equation, we can thus estimate the number of weights (and which one) should be removed given that $\Delta E_{[\pi_1,\pi_k]} < threshold$. It is extremely useful in pruning a neural network. As mentioned in [87], one problem in effective pruning is to determine which weights and how many weights should be removed simultaneously in one pruning step. Equation (5.16) sheds light on solving that problem as it is an estimation that amount of training error will be increased if the π_1^{st} to π_k^{th} weights are removed.

5.2 Testing on the approximation

To verify that ΔE_k (5.15) and $\Delta E_{[\pi_1,\pi_k]}$ (5.16) are good estimations of the change in training error, the generalized XOR problem is being solved. The function to be approximated is defined as follows :

$$y(x_1, x_2) = sign(x_1) sign(x_2).$$

A feedforward neural network with 20 hidden units is trained using the extended Kalman filter method with

$$P(0) = I.$$

The value of q is set to be 10^{-4}. 8000 training data are generated. The actual training error is defined as follows :

$$E_{train} = \frac{1}{8000} \sum_{t=1}^{8000} (y_t - \hat{y}_t)^2.$$

After training, another 100 pairs of data are passed to the network and the mean prediction error is defined as the mean squares testing error.

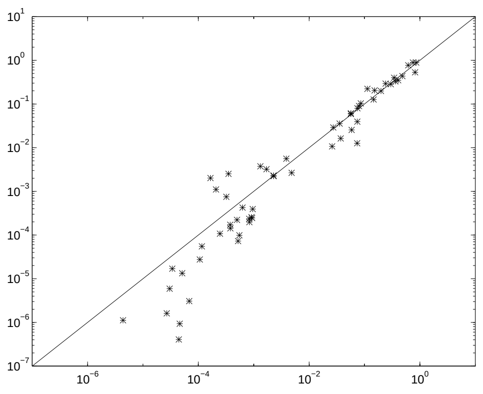
$$E_{test} = \frac{1}{100} \sum_{t=1}^{100} (y_t - \hat{y}_t)^2.$$

The importance of the weights is ranked according to the ΔE_k, (5.15). The actual change of error is obtained by removing k^{th} weight from the trained neural network and then passing 100 pairs of testing data to the pruned network. We calculate this testing error by the formula :

$$E_p(k) = \frac{1}{100} \sum_{t=1}^{100} (y_t - \hat{y}_t)^2.$$

The actual change of error is thus evaluated by

$$\Delta E_k(actual) = E_p(k) - E_{test}.$$


(a) ΔE_k for $n = 20$

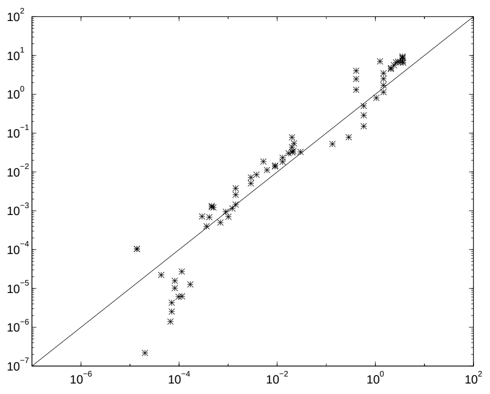

(b) $\Delta E_{[\pi_1,\pi_k]}$ for $n = 20$

Figure 5.1: The log-log plot of *the estimated* ΔE *against the actual* ΔE. The vertical axis corresponds to the estimated ΔE and the horizontal axis corresponds to the actual ΔE. The neural network has 20 hidden units. Simulation results confirm that ΔE_k and $\Delta E_{[\pi_1,\pi_k]}$ are good estimations of the actual change of error.

This error term is then compared with the estimate, ΔE_k, in Figure 5.1a. The x-axis corresponds to $E_p(k) - E_{train}$, i.e. the actual change of error while the y-axis corresponds to ΔE_k, the estimate.

In regard to the ranking list, the accumulative error is estimated via $\Delta E_{[\pi_1,\pi_k]}$ (5.16). Similarly, to evaluate the actual change of error, another 100 data pairs are generated. According to the ranking list, say $\{\pi_1, \ldots, \pi_{n_\theta}\}$, the π_1 up to π_k weights are removed. Then passing the 100 data pairs to the pruned network, we calculate the actual mean square error by the formulae :

$$E_p[\pi_1, \pi_k] = \frac{1}{100}\sum_{t=1}^{100}(y_t - \hat{y}_t)^2.$$

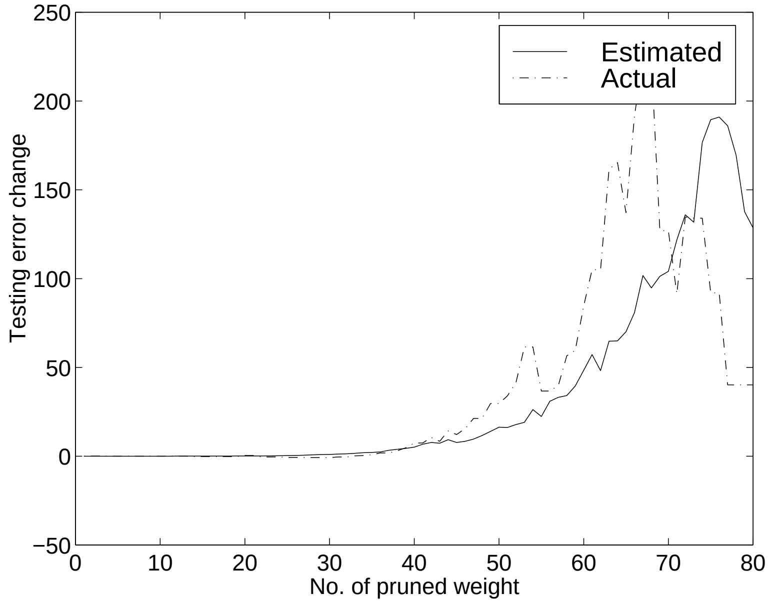
And thus the actual change of error is $E_p[\pi_1, \pi_k] - E_{test}$. This error term is then compared with the estimate, $\Delta E_{[\pi_1,\pi_k]}$ and shown in Figure 5.1b.

Figure 5.2 and 5.3 show the comparison between the estimated testing error $\Delta E_{[\pi_1,\pi_k]}$ and the actual $\Delta E_{[\pi_1,\pi_k]}$ against number of weights pruned for different values of q. The neural network has 20 hidden units. It is also found that the estimated $\Delta E_{[\pi_1,\pi_k]}$ can closely estimate the actual $\Delta E_{[\pi_1,\pi_k]}$ for k up to 40.

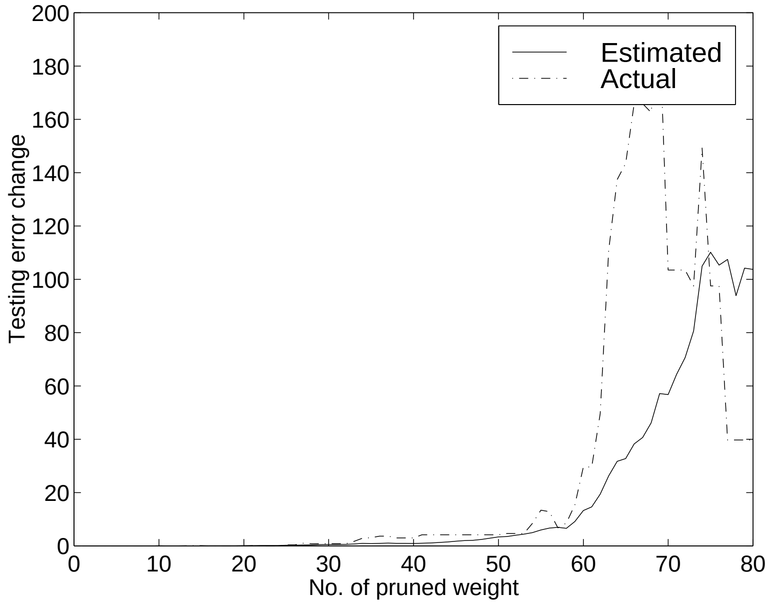
The same experiment has also been carried out for the FRLS training method. The weights are ranked in accordance with

$$S(\hat{\theta}_i) = \alpha\frac{\hat{\theta}_i^{\,2}}{2}\left(P^{-1}(N) - P^{-1}(0)\right)_{ii}.$$

Figure 5.2 and 5.3 show the comparison between the estimated testing error $\Delta E_{[\pi_1,\pi_k]}$ and the actual $\Delta E_{[\pi_1,\pi_k]}$ against number of weights pruned for different values of α. It is found that for small α the estimated $\Delta E_{[\pi_1,\pi_k]}$ can closely estimate the actual $\Delta E_{[\pi_1,\pi_k]}$ for k

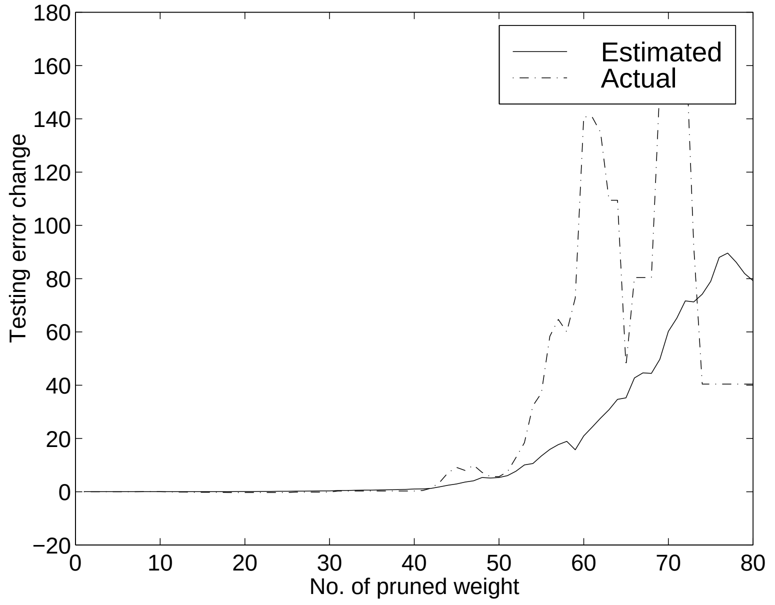


(a) $q = 0.001$

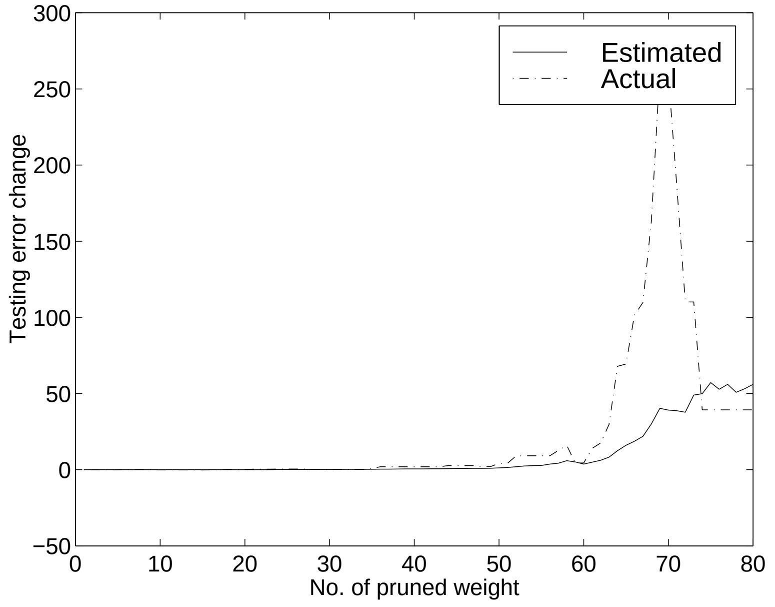


(b) $q = 0.002$

Figure 5.2: Testing error change $\Delta E_{[\pi_1,\pi_k]}$ against number of weights pruned for q equals to 0.001 and 0.002.



(c) $q = 0.005$



(d) $q = 0.010$

Figure 5.3: Testing error change $\Delta E_{[\pi_1, \pi_k]}$ against number of weights pruned for q equals to 0.005 and 0.010.

around 50. For large α, the estimated $\Delta E_{[\pi_1,\pi_k]}$ can closely estimate the actual $\Delta E_{[\pi_1,\pi_k]}$ for k around 30.

Comparing these two results, we can see that the EKF-based weight importance measure can closely approximate the actual incremental change of prediction error for k up to around $0.5n_\theta$, where n_θ is the total number of weights.

5.3 Relation to optimal brain damage

In case the system being tackled is static, the noise term $v(t) = 0$ for all $t \geq 0$ $(Q(t) = 0)$,

$$\begin{aligned} \theta(t) &= \theta(t-1) && (5.17) \\ y(t) &= f(x(t), \theta(t)) + \epsilon(t). && (5.18) \end{aligned}$$

The probability density function for $\theta(t)$ given $\theta(t-1)$ would be a delta function :

$$P(\theta(t)|\theta(t-1)) = \begin{cases} 1 & \text{if } \theta(t) = \theta(t-1) \\ 0 & \text{otherwise.} \end{cases} \quad (5.19)$$

Putting this equation into the right hand side of Equation (6.11), we obtain

$$\frac{P(y(t), x(t)|\theta(t))\mathcal{P}(\theta(t)|Y^{t-1})}{\int P(y(t), x(t)|\theta(t))\mathcal{P}(\theta(t)|Y^{t-1})d\theta(t)}. \quad (5.20)$$

Assuming that $\mathcal{P}(\theta(t-1)|Y^{t-1})$ is Gaussian and using Equation (5.17), it can easily see that $\mathcal{P}(\theta(t)|Y^{t-1})$ is a Gaussian distribution with mean $\hat{\theta}(t-1)$ and variance $P(t-1)$. Linearizing Equation (5.18) locally at $\hat{\theta}(t)$, $P(y(t), x(t)|\theta(t))$ can be approximated by a Gaussian distribution with mean $f(x(t), \hat{\theta}(t-1))$ and variance $H^T(t)P(t-1)H(t) + R$. Let $R = 1$, the *a posteriori* probability of $\theta(t)$ given Y^t would also be a Gaussian distribution with mean and variance given by

$$\begin{aligned} \hat{\theta}(t) &= \hat{\theta}(t-1) + L(t)(y(t) - f(x(t), \hat{\theta}(t-1))) && (5.21) \\ P^{-1}(t) &= P^{-1}(t-1) + H(t)H^T(t), && (5.22) \end{aligned}$$

where

$$\begin{aligned} L(t) &= P(t-1)H(t)[H^T(t)P(t-1)H(t) + 1]^{-1} && (5.23) \\ P^{-1}(0) &= \lambda I_{n_\theta \times n_\theta}, \quad 0 < \lambda \ll 1 && (5.24) \\ \hat{\theta}(0) &= 0. && (5.25) \end{aligned}$$

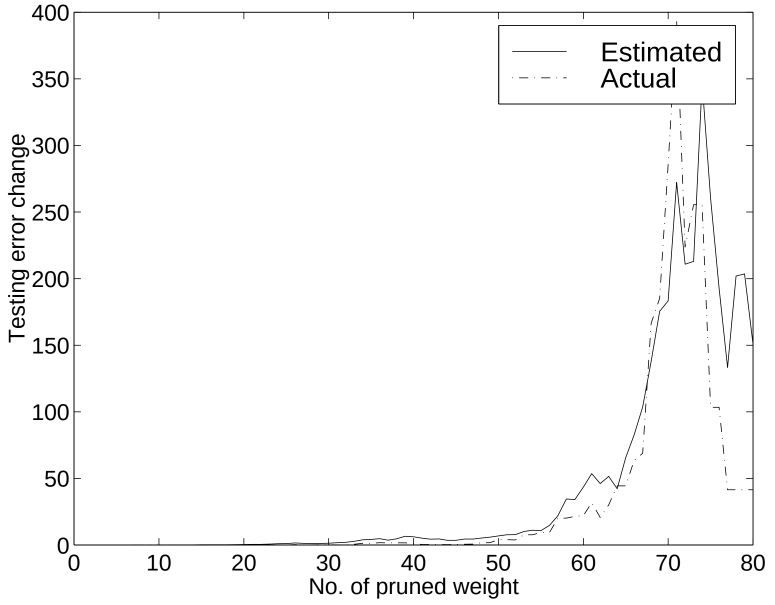
This algorithm is the standard recursive least square method [2].

After N iteration,

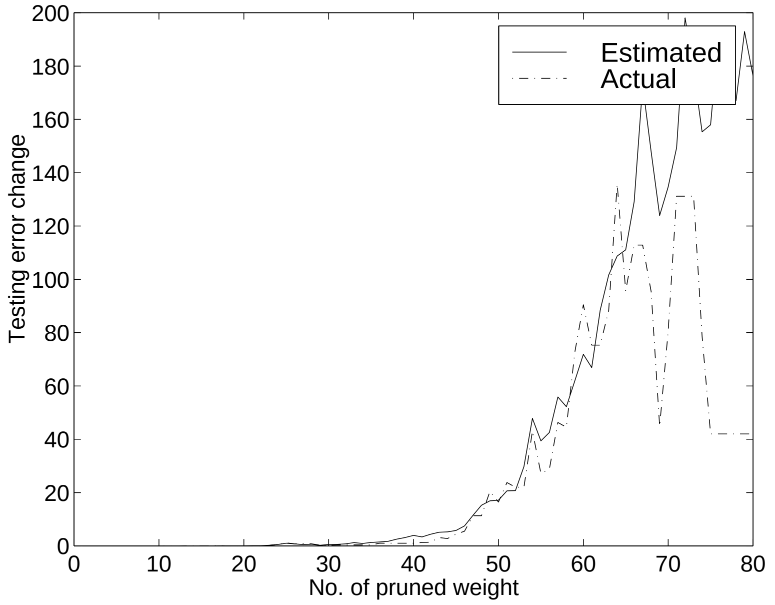
$$P^{-1}(N) = P^{-1}(0) + \sum_{k=1}^{N} H(k)H^T(k)$$

Suppose N is large and the error function $E(\theta)$ is given by

$$E(\theta) = \frac{1}{N}\sum_{k=1}^{N}(y(x_k) - f(x_k, \theta))^2 \quad (5.26)$$

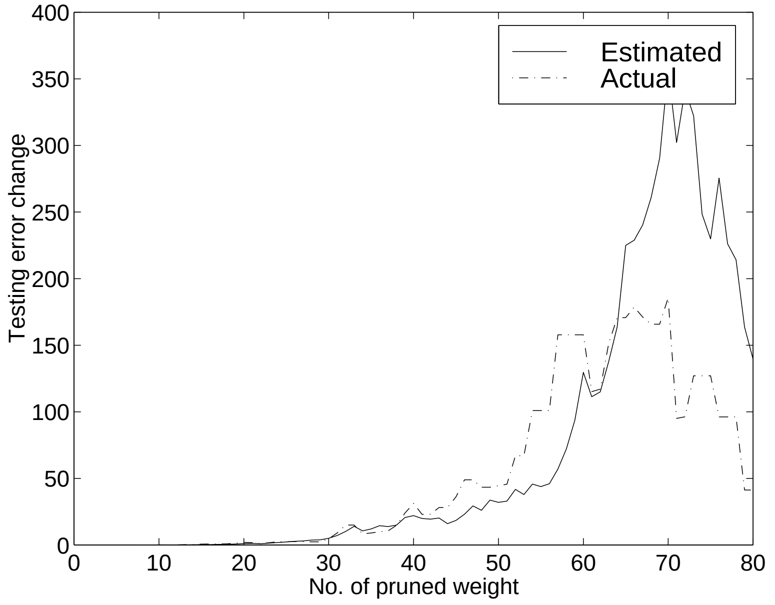


(a) $\alpha = 0.001$

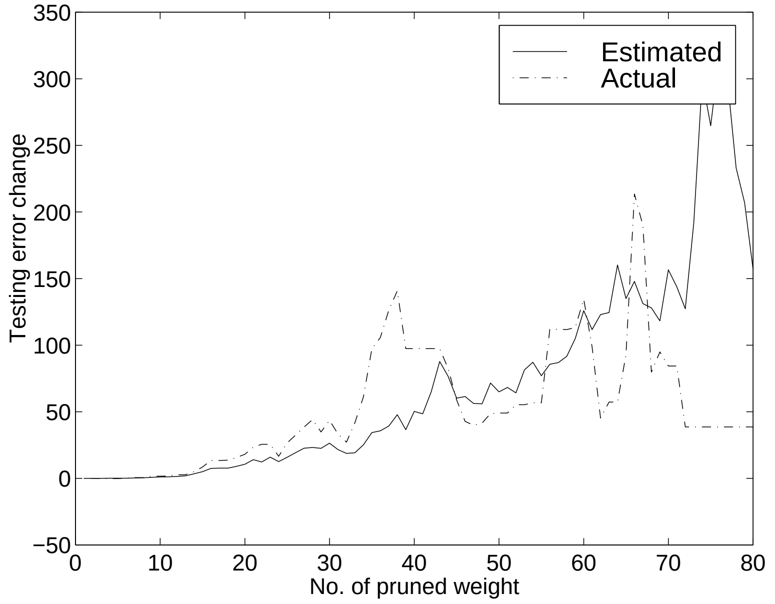


(b) $\alpha = 0.002$

Figure 5.4: Testing error change $\Delta E_{[\pi_1,\pi_k]}$ against number of weights pruned for α equals to 0.001 and 0.002.



(c) $\alpha = 0.005$



(d) $\alpha = 0.010$

Figure 5.5: Testing error change $\Delta E_{[\pi_1,\pi_k]}$ against number of weights pruned for α equals to 0.005 and 0.010.

Multiplying the k^{th} diagonal element of $N^{-1}P^{-1}(N)$ with the square of the magnitude of the k^{th} parameter, we can approximate the second order derivative of $E(\theta)$ by

$$\nabla\nabla E(\theta) = \frac{1}{N}\left[P^{-1}(N) - P^{-1}(0)\right]. \tag{5.27}$$

The weight importance measure of the k^{th} weight will be approximated by the following equation :

$$\begin{aligned} E(\hat{\theta}^k) - E(\hat{\theta}) &\approx \hat{\theta}_k^2 \left(\nabla\nabla E(\theta)\right)_{kk} && (5.28)\\ &\approx \frac{1}{N}\hat{\theta}_k^2 \left(P^{-1}(N) - P^{-1}(0)\right)_{kk} && (5.29)\\ &\approx \frac{1}{N}\hat{\theta}_k^2 \left(P^{-1}(N)\right)_{kk} && (5.30)\end{aligned}$$

With this equation, we can thus interpret the idea of optimal brain damage [55] and optimal brain surgeon [31] in probability sense[2] :

$$E(\hat{\theta}^k) - E(\hat{\theta}) \approx -\frac{2}{N}\log\left\{c_0^{-1}\mathcal{P}\left(\hat{\theta}^k(N)|Y^N\right)\right\}, \tag{5.31}$$

The weight being pruned away is the one whose posteriori distribution is very flat compared to its mean value. This also makes a link to McKay's Bayesian method [70, 71].

5.4 Summary

In summary, we have presented a method for pruning a neural network solely based on the results obtained by Kalman filter training such as the weight vector $\hat{\theta}$ and the $P(N)$ matrix. With the assumptions that (i) the training converges, (ii) $\hat{\theta}$ is close to θ_0 and the size of the training data is large enough, we have derived an elegant equation approximating the incremental change of mean prediction error due to pruning. Besides, the incremental change of mean prediction error due to the removal of multiple weights can also be estimated. Making use of these equations, it is possible to estimate how many weights and which weights should be removed. The effectiveness of the proposed weight importance measure is confirmed by extensive simulation results. Finally, the relation of EKF based pruning and other existing pruning methods have also been discussed.

[2]Other alternative derivations of the above relation can also be found in Larsen (1996) and Leung *et al.* (1996).

Chapter 6

EKF based Pruning for FNN in a Non-stationary Environment

In this chapter, we will elucidate how the extended Kalman filter can be applied to implement an adaptive pruning method. In section one, three types of non-stationary environment will be described. The formulation of neural network training under time varying environments as the extended Kalman filtering problem will be reviewed in section two. A simple example will be given to illustrate the advantage of EKF in neural network training. In section three, a formula for evaluating the importance measure will be devised and an adaptive pruning algorithm called adaptive Bayesian pruning based on sensitivity measure in terms of *posteriori* probability will be presented. Two sets of simulation results are presented in section four. Finally, we conclude the paper in section five.

6.1 Non-stationary environments

Of course, '*non-stationary environment*' is a very general term. It can be used to describe many situations. (a) It can be a non-linear regressor with fixed structure but the parameters, denoted by vector $\theta(t)$, are varying :

$$y(t) = f(y(t-1), y(t-2), x(t), \theta(t)), \tag{6.1}$$

where $x(t)$, $y(t)$ and $\theta(t)$ are respectively the input, the output and the model parameter of the system at time t. (b) It can be a switching type regressor with two different structures swapping from one to another. For example,

$$y(t) = \begin{cases} a_{11}y(t-1) + a_{12}y(t-2) + b_1x(t) + c_1 & \text{if } (2n-1)T > t \geq 2nT \\ a_{21}y(t-1) + a_{22}y(t-2) + b_2x(t) + c_2 & \text{if } 2nT > t \geq (2n+1)T \end{cases} \tag{6.2}$$

where a_{ij} are constant for all $i, j = 1, 2$ and b_1, b_2, c_1 and c_2 are all constant. T is the length of the time interval between switching and n is a positive integer. (c) Certainly, it can also be a system with changing structure and varying parameter throughout time. For example,

$$y(t) = \begin{cases} a_{11}(t)y(t-1) + a_{13}(t)y(t-1)y(t-2) + b_1x(t) + c_1 & \text{if } (2n-1)T > t \geq 2nT \\ a_{21}y(t-1) + a_{22}(t)y(t-2) + b_2(t)x(t) & \text{if } 2nT > t \geq (2n+1)T \end{cases} \tag{6.3}$$

This system switches from a non-linear regressor to a linear regressor.

Obviously, pruning a neural network under these conditions can be very dangerous, in particular for system (6.2) and system (6.3). This makes the problem of pruning a neural network in a non-stationary environment a real challenge.

In this chapter, we will focus on the first case only. That means, we assume the the structure of the system is fixed, Equation (6.1). The only non-stationary part is the system parameter. We further assume that this non-stationary system can be represented by a feedforward neural network with fixed but unknown structure. Our goal is to design a method which is able to find the structure of this feedforward neural network.

Obviously, if we have the information about the structure of the feedforward neural network, the training problem is simply a parameter tracking problem [2]. However, this information is usually not available. In such a case, one approach is to train a large size neural network. Once the tracking is good enough, the redundant weights are identified and pruned away. Eventually, better parameter tracking can be achieved and a good network structure can be obtained.

For such an effective adaptive pruning method, the training method must be fast enough to track the time varying behavior. If possible, the training method should provide information for measuring weight importance and hence pruning can be accomplished without much additional computational cost. To do so, we suggest applying the extended Kalman filter approach as the training method. One reason is because it is a fast adaptive training method which can track the time varying parameter. The other reason is because the weight vector and the error covariance matrix can provide information for pruning (as described later).

In the rest of this chapter, we will elucidate how the extended Kalman filter can be applied to implement such an adaptive pruning method. In the next section, the formulation of training a neural network under time varying environment via extended Kalman filtering problem will be reviewed. A simple example will be given to illustrate the advantage of EKF in neural network training. In section three, a formula for evaluating the importance measure will be devised and an adaptive pruning algorithm called adaptive Bayesian pruning based on a sensitivity measure in terms of *a posteriori* probability will be presented. Two simulation results are presented in section four. Section five presents the similarity between adaptive Bayesian pruning and optimal brain damage. Finally, we conclude the paper in section six.

6.2 Training neural network under time-varying environment

Throughout the paper, we let $y(x,t) = f(x, \theta(t))$ be the transfer function of a single layer feedforward neural network where $y \in R$ is the output, $x \in R^m$ is the input and $\theta \in R^n$ is its parameter vector. This mapping is assumed to be a time varying model determined by a time varying parametric vector $\theta(t)$, in contrast to the conventional feedforward network model in which it is assumed that it is a constant vector. This set-up is to ensure that the network is able to learn in a non-stationary environment. Given a set of training data $\{x(i), y(i)\}_{i=1}^{N}$, the training of a neural network can be formulated as a filtering problem. Let us assume that the data are generated by the following noisy signal model :

$$\begin{aligned} \theta(k) &= \theta(k-1) + v(k) && (6.4) \\ y(k) &= f(x(k), \theta(k)) + \epsilon(k) && (6.5) \end{aligned}$$

where $v(t)$ and $\epsilon(t)$ are zero mean Gaussian noise with variance $Q(t)$ and $R(t)$. A *good* estimation of the system parameter θ can thus be obtained via the extended Kalman filter method [37, 95, 90, 121] :

$$\begin{aligned} S(k) &= F^T(k)[P(k-1) + Q(k)]F(k) + R(k) && (6.6) \\ L(k) &= [P(k-1) + Q(k)]F(k)S^{-1}(k) && (6.7) \\ P(k) &= (I_{n \times n} - L(k)F(k))P(k-1) && (6.8) \\ \hat{\theta}(k) &= \hat{\theta}(k-1) + L(k)(y(k) - f(x(k), \hat{\theta}(k-1))) && (6.9) \end{aligned}$$

where $F(k) = \frac{\partial f}{\partial \theta}$. For simplicity, Equation (6.8) can be rewritten as that

$$P^{-1}(k) = [P(k-1) + Q(k)]^{-1} + F(k)R^{-1}F^T(k). \qquad (6.10)$$

The above equation can be rewritten as follows :

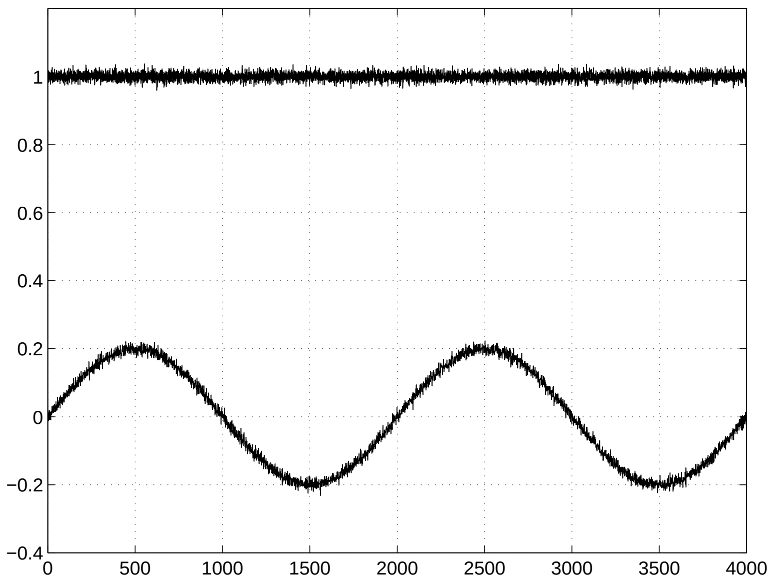
$$P^{-1}(k) = [I + P^{-1}(k-1)Q(k)]^{-1}P^{-1}(k-1) + F(k)R^{-1}(k)F^T(k).$$

As $P(k-1)$ and $Q(k)$ are symmetric, it can be proven that the eigenvalues of $[I + P^{-1}(k-1)Q(k)]^{-1}$ are in between zero and one for non-zero matrix $Q(k)$. Comparing this equation to the standard recursive least square method [48, 97, 56], $P^{-1}(k) = P^{-1}(k-1) + F(k)F^T(k)$, EKF training can be viewed as forgetting learning equipped with an adaptive forgetting matrix $[I + P^{-1}(k-1)Q(k)]^{-1}$. This factor controls the amount of information (stored in $P(k)$) being removed and the importance of the new training data.

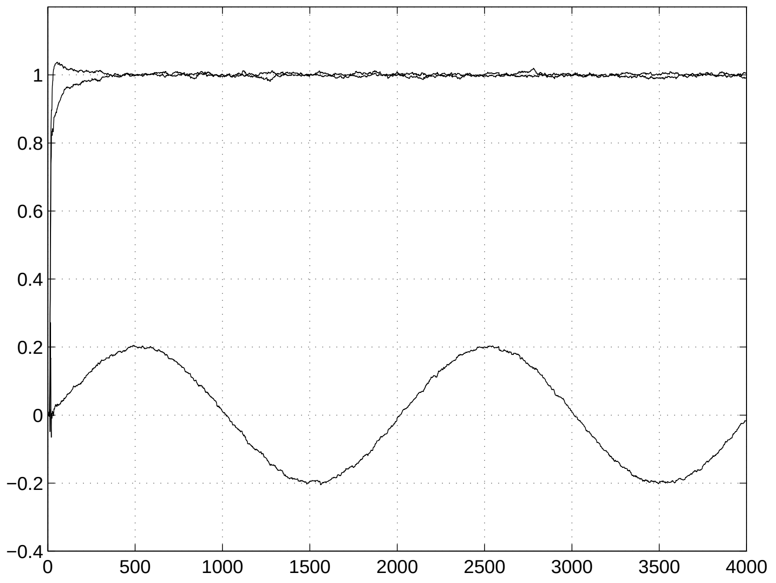
The advantage of using EKF can be perceived from a simple example. Consider a simple time varying function defined as follows :

$$\begin{aligned} y(x) &= c(t)\tanh(b(t)x + e(t)), \\ c(t) &= 1 + noise_c(t), \\ b(t) &= 1 + noise_b(t), \\ e(t) &= 0.2\sin(\frac{2\pi t}{20}) + noise_e(t). \end{aligned}$$

All the noise factors are independent zero mean Gaussian noise with 0.2 standard deviation. It can be seen that the function can be implemented by a single neuron neural network with three parameters as shown in Figure 6.1a : the input-to-hidden weight and hidden-to-output weight are constant while the threshold is a time varying parameter $0.2\sin(\frac{2\pi t}{20})$. At every 0.01 time interval, an x is generated randomly (uniformly) from the interval $[-2, 2]$. The corresponding $y(x)$ is evaluated and the data pair $\{x, y(x)\}$ are fed to a single neuron neural network as training data. It can be seen from Figure 6.1b EKF is able to track all three parameters and even filter away random noise.



(a) Time varying parameter



(b) Parameter tracked by EKF

Figure 6.1: An example using EKF in tracking the parameters of a non-stationary mapping. (a) The time varying parameters : $c(t) = 1 + noise_c(t)$, $b(t) = 1 + noise_b(t)$, $e(t) = 0.2\sin(\frac{2\pi t}{20}) + noise_e(t)$. (b) The estimated time varying parameters : $\hat{c}(t)$, $\hat{b}(t)$ and $\hat{e}(t)$. The horizontal axis is the time axis.

6.3 Adaptive Bayesian pruning

When the EKF approach is adopted as a training method, we need to decide a reasonable measure on the weight importance which can make use of the by-products such as $P(N)$ and $\hat{\theta}(N)$. To do so, we must first look at the Bayesian nature of Kalman filtering.

6.3.1 EKF and recursive Bayesian learning

Considering the objective of EKF training, we learn from the theory of extended Kalman filtering [2] that the EKF effectively evaluate the *a posteriori* probability $\mathcal{P}(\theta(t)|Y^t)$ follows a recursive Bayesian approach :

$$\mathcal{P}(\theta(t)|Y^t) = \frac{\int P(y(t), x(t)|\theta(t)) P(\theta(t)|\theta(t-1)) \mathcal{P}(\theta(t-1)|Y^{t-1}) d\theta(t-1)}{\int\int P(y(t), x(t)|\theta(t)) P(\theta(t)|\theta(t-1)) \mathcal{P}(\theta(t-1)|Y^{t-1}) d\theta(t-1) d\theta(t)}. \quad (6.11)$$

with the assumption that $\mathcal{P}(\theta(t)|Y^t)$ and $P(y(t), x(t)|\theta(t))$ are Gaussian distributions. The last assumption is accomplished by linearizing the non-linear function $f(x(t), \theta(t-1))$ locally at $\hat{\theta}(t-1)$. With the Bayesian interpretation and Gaussian assumption, the estimated $\hat{\theta}(t)$ and $P(t|t)$ loosely approximate the conditional mean and its convariance.

6.3.2 Importance measure for pruning a single weight

Since $\mathcal{P}(\theta(t)|Y^t)$ is a Gaussian distribution approximating the actual *a posteriori* probability given the measurement data Y^t, we can write down the equation explicitly :

$$\mathcal{P}(\theta(t)|Y^t) = c_0 \exp\left\{-\frac{1}{2}\left(\theta(t) - \hat{\theta}(t)\right)^T P^{-1}(t)\left(\theta(t) - \hat{\theta}(t)\right)\right\}, \quad (6.12)$$

where c_0 is a normalizing constant, the parameters $\hat{\theta}(t)$ and $P(t)$ are the results obtained via Equation (6.6) to (6.9) at the t^{th} time step.

Let $\hat{\theta}^k(t)$ be the parametric vector with all elements equal to $\hat{\theta}(t)$ except that the k^{th} element is zero (i.e. $\hat{\theta}^k(t) = [\hat{\theta}_1(t) \ldots \hat{\theta}_{k-1}(t)\ 0\ \hat{\theta}_{k+1}(t) \ldots \hat{\theta}_{n_\theta}(t)]^T$),

$$\mathcal{P}(\hat{\theta}^k(t)|Y^t) = c_0 \exp\left\{-\frac{1}{2}\hat{\theta}_k^2 \left(P^{-1}(t)\right)_{kk}\right\}, \quad (6.13)$$

where $\left(P^{-1}(t)\right)_{kk}$ is the k^{th} diagonal element of the inverse of $P(t)$. Note that c_0 is equal to $\mathcal{P}(\hat{\theta}(t)|Y^t)$. Obviously, the smaller the value of the factor $\hat{\theta}_k^2 \left(P^{-1}(t)\right)_{kk}$, the higher *a posteriori* probability $\mathcal{P}(\hat{\theta}^k(t)|Y^t)$. Therefore, if we just want to prune one weight at a time, Equation (6.13) can be treated as a measure on the importance of the weight.

6.3.3 Importance measure for pruning multiple weights

Generally speaking, we would like to prune more than one weight at a time in order to reduce the storage and computational complexity during training. To remove a set of weights, we need a measure for pruning multiple weights. As we already have a measure (6.13), we now rank the weights accordingly. Let $\{\pi_1, \ldots, \pi_{n_\theta}\}$ be the ranking list and $\hat{\theta}_{[1,k]}$ be the vector

Step 1. Using the recursive equations to obtain $\hat{\theta}(N)$ and $P(t)$.

Step 2. Estimate the training error $E_{tr}(t)$ by $t^{-1}\sum_{i=1}^{t}(y(i)-\hat{y}(i))^2$.

Step 3. If $E_{tr}(t) < E_{tr0}$,

1. Evaluating $P^{-1}(t)$ and hence $\theta_k^2\left(P^{-1}(t)\right)_{kk}$ for all k from 1 to n_θ.
2. Rearranging the index $\{\pi_k\}$ according to the ascending order of $\theta_k^2\left(P^{-1}(t)\right)_{kk}$.
3. For π_k from 1 to n_θ, evaluate $\mathcal{P}(\hat{\theta}_{[1,k]}(t)|Y^t)$ as if θ_{π_1} up to θ_{π_k} are removed.
4. Remove θ_{π_1} up to θ_{π_k} if $\log\mathcal{P}(\hat{\theta}_{[1,k]}(t)|Y^t) - \log c_0 < E_0$

Figure 6.2: The adaptive pruning procedure for use in a time varying environment.

with elements from π_1 up to π_k being zeros and the rest of the elements being identical to $\hat{\theta}_{\pi_{k+1}}$ to $\hat{\theta}_{\pi_{n_\theta}}$, the importance of the weights indexed from π_1 up to π_k as follows :

$$\mathcal{P}(\hat{\theta}_{[1,k]}(t)|Y^t) = c_0 \exp\left\{-\frac{1}{2}\hat{\theta}_{[1,k]}^T P^{-1}(t)\hat{\theta}_{[1,k]}\right\}. \tag{6.14}$$

Therefore, Equation (6.13) together with Equation (6.14) define the essential part of the adaptive pruning procedure which is summarized in Figure 6.2.

Note that Prechelt (1996, 1997) has recently proposed an adaptive pruning procedure for feedforward neural networks based on the importance measure suggested in Finnoff *et al.*(1993). Based on the observation that the statistics of the weight importance measure follows (roughly) a normal distribution, a heuristic technique is proposed to decide how many weights should be pruned away and when pruning should be started. One essential difference between Prechelt's approach and ours is that Prechelt's approach requires a validation set in conjunction with the importance measure to determine the set of weights to be removed while our method does not. Besides, his algorithm is applied to stationary classification problems.

6.4 Illustrative examples

In this section two simulated results are reported to demonstrate the effectiveness of the proposed pruning procedure. In the first experiment, we approximate the time varying function defined in Section 2 using a feedforward neural network with two hidden units and use the EKF together with the proposed pruning algorithm to show the importance of the reduction of network redundancy. The second experiment will be on the tracking of a moving Gaussian function.

6.4.1 Simple function

Using the same example as demonstrated in Section 2, we now define the initial network as a two-hidden units feedforward network. (Obviously, one neuron is redundant). Applying the proposed adaptive pruning together with the extended Kalman filter training, we can observe the advantage of using pruning from Figure 6.3a,b. If pruning is not imposed, the redundant neuron (solid lines) can greatly affect the tracking ability of the neuron (dotted lines) which has the tendency to mimic the actual underlying model. On the other hand, if the pruning procedure is invoked (see Figure 6.3b) the redundant neuron (whose weights are shown by solid lines) can be identified at a early stage and hence removed at the very beginning. So in the long run, the only neuron (whose weights are shown by dotted lines) can track perfectly the parameters. The same results are observed even though $e(t)$ is a noisy square wave with amplitude 0.2 and period 5000, Figure 6.3c,d.

6.4.2 Moving Gaussian function

In this simulated example, we apply a feedforward neural network to approximate a non-stationary function with two inputs and one output. The function being approximated is defined as follows :

$$y(x_1, x_2, t) = \exp\left(-4[(x_1 - 0.2\sin(0.02\pi t))^2 + (x_2 - 0.2\cos(0.02\pi t))^2]\right). \qquad (6.15)$$

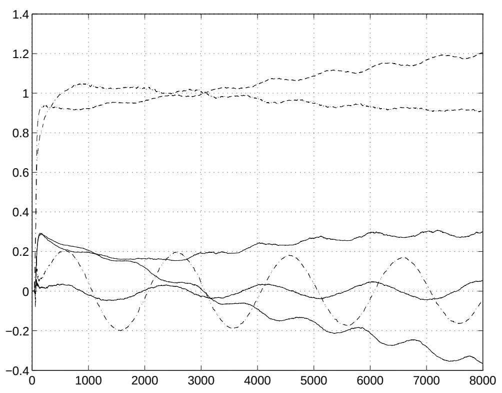
This corresponds to a Gaussian function whose center is rotating around the origin with period $T = 100$. 16×10^4 data are generated at time instance $t = 0.01, 0.02$ and so on up to $t = 1600$. At each time instance, an input point $(x_1(t), x_2(t))$ is randomly (uniformly) generated from $[-1,1] \times [-1,1]$ and the corresponding output is evaluated by adding noise to the output evaluated by using Equation (6.15).

The threshold E_{tr0} is set to be 0.01 and pruning can only be carried out at every 200 steps. And the small value E_0 is set to be δE_{tr0}. The initial network consists of sixteen hidden units, two input units and one output unit, in total 64 weights. The output of the training data is corrupted by zero mean Gaussian noise with variance 0.2. The *threshold* value and the value of δ are set to be 0.2 and 0.5 respectively. For comparison, we repeat the experiment 5 times. The average results are depicted in Table 6.4.2.

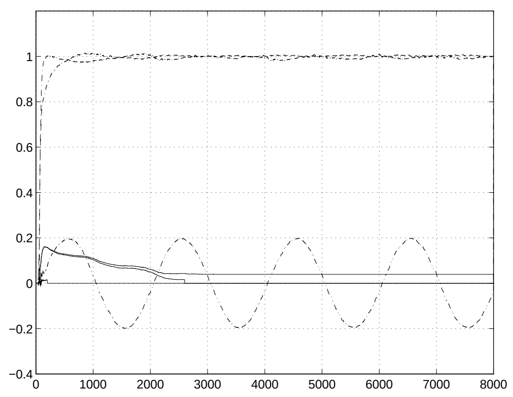
Figure 6.4 shows the training curves for the case when the pruning procedure is invoked. Although the asymptotic training error of using the pruning procedure is larger than no-pruning and pruning does not help much in improving the generalization, it can help to reduce a large amount of network redundancy, see Figure 6.5, and hence reduce a considerable amount of storage and computation complexity. For comparison, Figure 6.6 shows the resultant shape of the Gaussian function at $t = 800$ when pruning is not invoked while Figure 6.7 shows the case when pruning is invoked.

6.5 Summary and Discussion

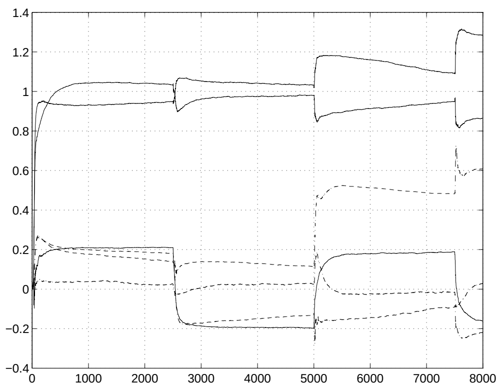
In this chapter, an adaptive pruning procedure for use in a non-stationary environment is developed. To maintain a good tracking ability, we adopted the method of EKF in training the neural network. In order not to introduce much cost in doing cross validation and the evaluation of error sensitivity, we proposed a new measure for the weight importance and hence an adaptive Bayesian pruning procedure was devised. In a noisy time varying environment, we demonstrated that the proposed pruning method is able to reduce network



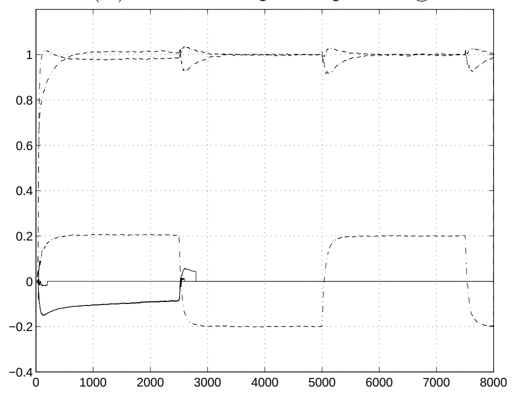
(a) Without pruning



(b) With adaptive pruning



(c) Without pruning



(d) With adaptive pruning

Figure 6.3: The change of the six weight values against time. Solid lines and dotted lines correspond to the weight values of the two neurons. (a) pruning is not invoked when $e(t)$ is a noisy sin, (b) pruning is invoked when the input is noisy sin, (c) pruning is not invoked when $e(t)$ is square wave and (d) pruning is invoked when $e(t)$ is square wave. According to the subplots (b) and (d), it is observed that the adaptive pruning is able to remove the redundant neuron, whose weights are shown by solid lines.

δ	Avg. MSE	Average number of weight pruned	Storage Complexity	Computational Complexity
0	0.0450	0	4096	262144
0.2	0.0451	27	1369	50653
0.5	0.0449	28	1296	46656

Table 6.1: Comparison of the average MSE, average number of weights pruned and the complexity between when pruning is invoked and is not invoked. The storage complexity is determined by the size of $P(t)$, i.e.$\mathcal{O}(n_\theta^2)$, and the computational complexity is determined by the matrix multiplication, i.e.$\mathcal{O}(n_\theta^3)$.

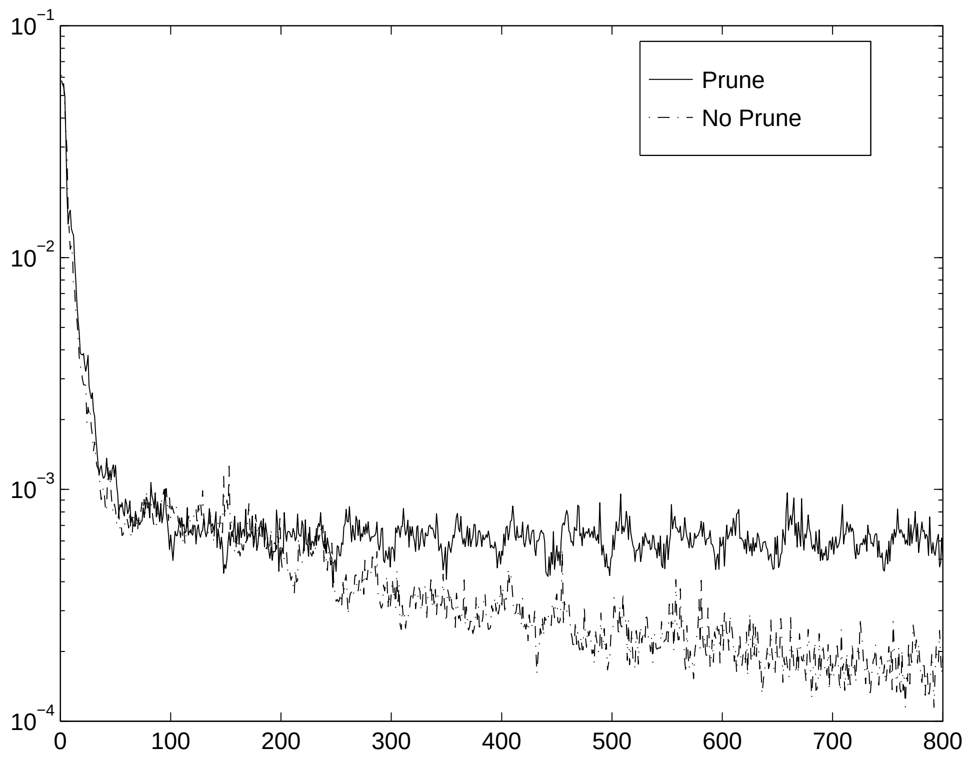


Figure 6.4: Training error against time curves for pruning procedure being invoked and not invoked. The vertical axis corresponds to training error while the horizontal axis corresponds to time.

redundancy adaptively but still preserve the same generalization ability as the fully connected one. Consequently, the storage complexity and computational complexity of using EKF in training are largely reduced. As we assume that the non-stationary environment is a system with fixed structure, the only time varying part is the system parameter. Once tracking of these time varying parameters is good enough, the redundant parameter can be identified and removed. The system does not reinstate the pruned weight. So, in case the actual system structure is not fixed (system (6.3) for example), certain methods would be required to reinstate those pruned weights if they turn out to be needed later. In Part III, this pruning method will be extended to the RNN case.

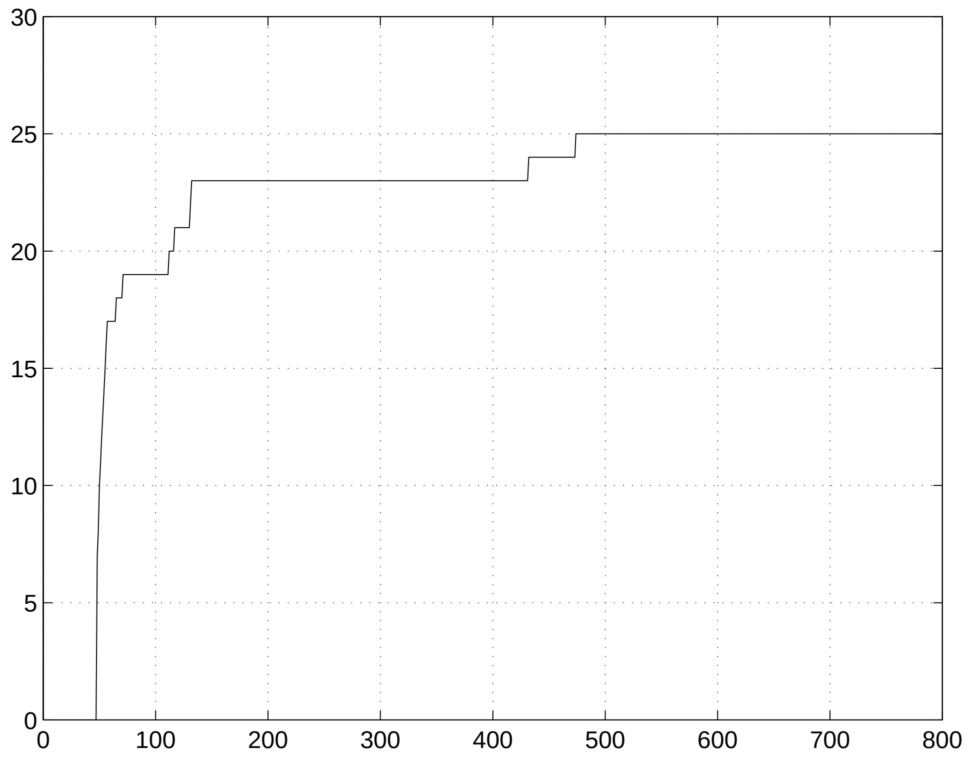


Figure 6.5: The number of weights pruned against time.

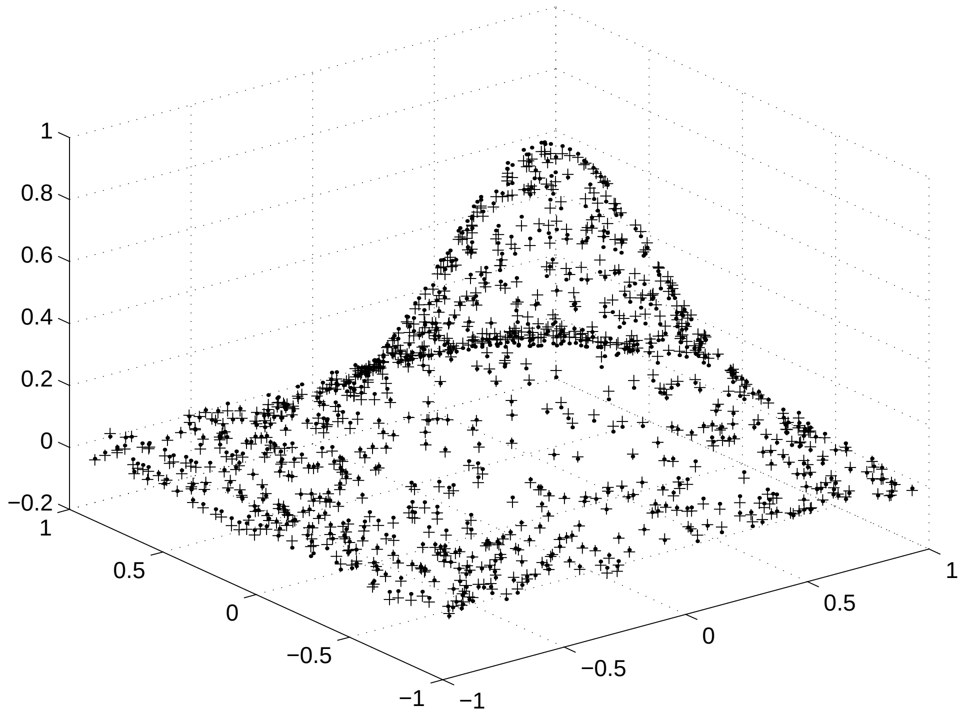


Figure 6.6: The resultant shape of the Gaussian function at $t = 800$ if the proposed pruning procedure is not invoked.

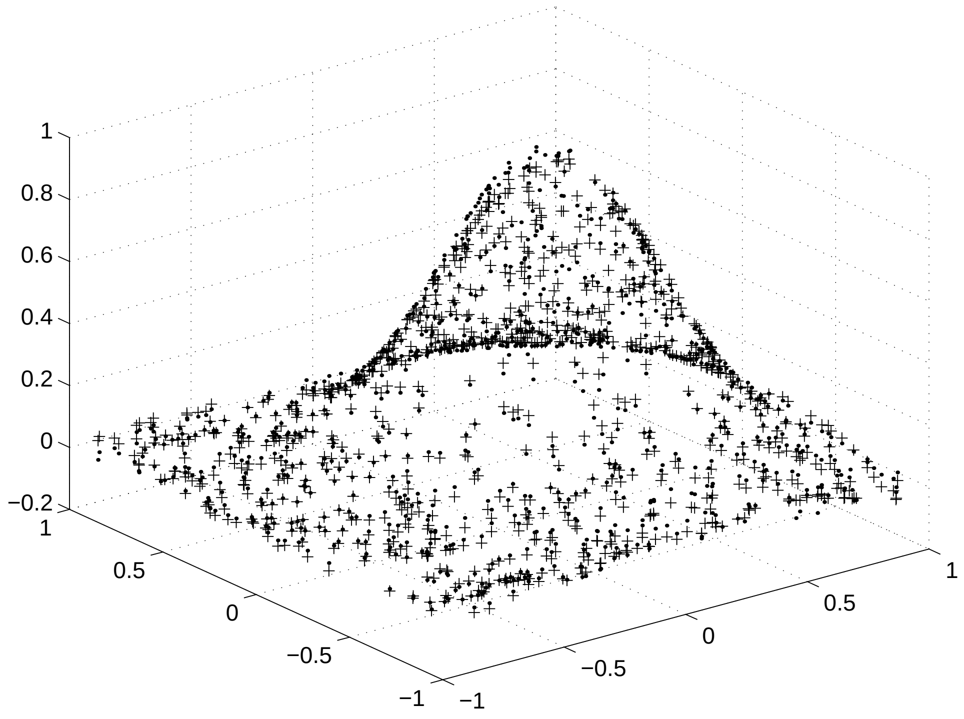


Figure 6.7: The resultant shape of the Gaussian function at $t = 800$ if the proposed pruning procedure is invoked.

Part III

Pruning Algorithms for RNN

Chapter 7

EKF based Pruning for RNN

Traditionally, real-time-recurrent-learning [124] or a back-propagation through time method [94] is employed in training the recurrent networks. Recently, Williams (1992) has formulated the RNN training as a non-linear filtering problem and applied the extended Kalman filter technique to find the weight values. Simulation results demonstrated that the speed up can increase by ten times. As the training is rather effective, it would be valuable to see if we can make use of the information obtained after training, the weight values and error covariance matrix, to prune the network. In this chapter we suggest considering the sensitivity of the *a posteriori* probability as a measure of the importance of the weight and devising a pruning procedure for recurrent neural network.

In the rest of the chapter, we will present a pruning procedure based on this idea. In section one, a background to the EKF will be presented. The EKF based training method for a recurrent network will be elucidated in section two. Section three will describe the pruning scheme based on the probability sensitivity. A method of evaluating the validation error will be presented in section four. Three simulated examples will then be given in section five for illustrating the effectiveness of this pruning scheme. In section six, we will discuss the necessity of the ranking heuristic and give two examples to show the usefulness of such a heuristic. Finally, we present the conclusion in section seven.

7.1 Background on extended Kalman filter

Without loss of generality, we consider a non-linear signal model as follows :

$$x(t+1) = f(x(t)) + v(t), \tag{7.1}$$
$$y(t) = h(x(t)) + w(t), \tag{7.2}$$

where $f(.)$ and $h(.)$ are non-linear functions of $x(t)$. The noise process $\{v(t)\}$ and $\{w(t)\}$ are mutually independent zero mean Gaussian noise and

$$E\{v(t)v(t)^T\} = Q(t), \quad E\{w(t)w(t)^T\} = R(t).$$

$x(0)$ is a Gaussian random variable with mean $\hat{x}(0)$ and variance P_0.

Let Y^t be the observations $\{y(0), y(1), \ldots, y(t)\}$, the estimation of the state $x(t)$ based on the observations Y^t or Y^{t-1} can be accomplished by using the extended Kalman filter.

Let $\hat{x}(t|t)$ and $\hat{x}(t|t-1)$ be the expectation of $x(t)$ given Y^t and Y^{t-1} respectively, the estimation of the $x(t)$ can be achieved by plugging in the following Kalman filter equations :

$$\begin{aligned}
\hat{x}(t|t) &= \hat{x}(t|t-1) + L(t)[y(t) - h(\hat{x}(t|t-1))] && (7.3)\\
\hat{x}(t+1|t) &= f(\hat{x}(t|t)) && (7.4)\\
L(t) &= P(t|t-1)H(t)[H^T(t)P(t|t-1)H(t) + R(t)]^{-1} && (7.5)\\
P(t|t) &= P(t|t-1) - L(t)H^T(t)P(t|t-1) && (7.6)\\
P(t+1|t) &= F(t)P(t|t)F^T(t) + Q(t), && (7.7)
\end{aligned}$$

where

$$\begin{aligned}
F(t) &= \frac{\partial f(\hat{x}(t|t))}{\partial x} && (7.8)\\
H(t) &= \frac{\partial h(\hat{x}(t|t))}{\partial x}. && (7.9)
\end{aligned}$$

The initial conditions $\hat{x}(0|-1)$ and $P(0|-1)$ are defined as P_0 and x_0 if we know that the initial value $x(0)$ is a Gaussian random variable with mean x_0 and variance P_0. In general, we may not have the information. In such a case, we can simply set x_0 as a small random number and P_0 to be a diagonal matrix with large diagonal values.

Theoretically, if $f(x)$ and $h(x)$ are linear functions of x, say $f(x) = Ax$ and $h(x) = Cx$, it is known that the probability density function for $x(t)$, denoted by $\mathcal{P}(x(t)|Y^t)$ is Gaussian distribution with mean $\hat{x}(t|t)$ and variance $P(t|t)$. Similarly, $\mathcal{P}(x(t)|Y^{t-1})$ is Gaussian distribution with mean $\hat{x}(t|t-1)$ and variance $P(t|t-1)$. Hence, $\hat{x}(t|t)$ **is the maximum** ***a posteriori*** **(MAP) estimation of** $x(t)$. In case $f(x)$ or $h(x)$ is a non-linear function, $\hat{x}(t|t)$ **and** $P(t|t)$ **will only be an approximation of the mean and variance of the** ***a posteriori*** **probability density function** $\mathcal{P}(x(t)|Y^t)$.

7.2 EKF based training

In this section, we will review how the training of a recurrent neural network can be accomplished as a non-linear state estimation problem. For the formulation of EKF in RNN training, one can also refer to Williams (1992)[1].

To train a recurrent neural network, we assume that the training data set is generated by a stochastic signal model as follows [123]:

$$\begin{aligned}
x(t+1) &= g(Ax(t) + Bu(t) + D) + v(t), && (7.10)\\
y(t+1) &= Cx(t+1) + w(t), && (7.11)
\end{aligned}$$

[1]We remark that at least two more approaches apart from Williams approach using extended Kalman filter to train recurrent neural networks have been so far proposed. Puskorius and Feldkamp (1994) and Suykens *et al.* (1995) considered that the learning is acting on the weight vector alone. In each time step, the derivative of each of the output units with respect to the weight vector is calculated recursively through a sensitivity network. The weight vector is then updated via the recursive least square method. Wan and Nelson (1996) extended the idea of dual linear Kalman filter to train a recurrent type neural network. The idea is to apply two Kalman filter equations to update the weight vector and hidden unit activity simultaneously in each time step. The main theme of this chapter is to elucidate how pruning can be used in conjunction with Williams approach.

where $v(t)$ and $w(t)$ are zero mean Gaussian noise. If the parameters (A, B, C, D) are known, we can use the extended Kalman filter to predict the $y(t+1)$, see Appendix.

If the parameters are not known, we need to estimate them. In EKF [123], training a recurrent network is treated as a non-linear estimation problem, where the parameters $\{A, B, C, D\}$ and $x(t)$ are the unknown states being estimated. Hence, the state equations are :

$$\begin{aligned} x(t+1) &= g(A(t)x(t) + B(t)u(t) + D(t)) + v(t), && (7.12)\\ \theta(t+1) &= \theta(t) + e(t), && (7.13)\\ y(t) &= C(t)x(t) + w(t). && (7.14) \end{aligned}$$

Let θ be the collection of the state $\{A, B, C, D\}$. Put $x(t)$ and $\theta(t)$ as a single state vector, the state equations become :

$$\begin{aligned} \begin{bmatrix} x(t+1) \\ \theta(t+1) \end{bmatrix} &= g_1(x(t), u(t), \theta(t)) + \begin{bmatrix} v(t) \\ e(t) \end{bmatrix} && (7.15)\\ y(t) &= f_1(x(t), \theta(t)) + w(t), && (7.16) \end{aligned}$$

where

$$\begin{aligned} g_1(x(t), u(t), \theta(t)) &= \begin{bmatrix} g(A(t)x(t) + B(t)u(t) + D(t)) \\ \theta(t) \end{bmatrix} && (7.17)\\ f_1(x(t), \theta(t)) &= C(t)x(t). && (7.18) \end{aligned}$$

Comparing Equations (7.15) and (7.16) with Equations (7.1) and (7.2), we can see that the simultaneous estimation of $x(t)$ and parametric vector $\theta(t)$ can be as obtained recursively via the following recursive equations :

$$\begin{aligned} x(t|t-1) &= g(\hat{x}(t-1|t-1), u(t), \hat{\theta}(t-1)) && (7.19)\\ P(t|t-1) &= F(t-1)P(t-1|t-1)F^T(t-1) + Q(t-1) && (7.20) \end{aligned}$$

$$\begin{aligned} \begin{bmatrix} \hat{x}(t|t) \\ \hat{\theta}(t) \end{bmatrix} &= \begin{bmatrix} \hat{x}(t|t-1) \\ \hat{\theta}(t-1) \end{bmatrix} + L(t)\left(y^*(t) - H^T(t)\begin{bmatrix} \hat{x}(t|t-1) \\ \hat{\theta}(t-1) \end{bmatrix}\right) && (7.21)\\ P(t|t) &= P(t|t-1) - L(t)H^T(t)P(t|t-1), && (7.22) \end{aligned}$$

where

$$F(t+1) = \begin{bmatrix} \partial_x g(\hat{x}(t|t), u(t+1), \hat{\theta}(t)) & \partial_\theta g(\hat{x}(t|t), u(t+1), \hat{\theta}(t)) \\ 0_{n_\theta \times n} & I_{n_\theta \times n_\theta} \end{bmatrix}, \qquad (7.23)$$

$$H^T(t) = [\partial_x^T y(t) \ \ \partial_\theta^T y(t)] \qquad (7.24)$$

$$L(t) = P(t|t-1)H(t)[H^T(t)P(t|t-1)H(t) + R(t)]^{-1} \qquad (7.25)$$

The initial $P^{-1}(0|0)$ is set to be zero matrix and $\hat{\theta}(0)$ is a small random vector. Given the data set $\{u(t), y^*(t)\}_{t=1}^N$ and iterating the above equations N times, the parametric vector $\hat{\theta}(N)$ will then be assigned as the network parameters.

Since the actual values of $Q(t)$ and $R(t)$ are not known in advance, they can be estimated recursively, as in Iiguni *et al.* (1992) :

$$\begin{aligned} R(t) &= (1-\alpha_R)R(t-1) + \alpha_R(y^*(t) - y(t|t-1))^2 & (7.26) \\ Q(t) &= (1-\alpha_Q)Q(t-1) + \alpha_Q L(t)L(t)^T(y^*(t) - y(t|t-1))^2, & (7.27) \end{aligned}$$

where α_R and α_Q are two small positive values.

7.3 Pruning scheme

As mentioned in the previous sections, the aim of the extended Kalman filter is to construct an approximated posteriori probability function for the hidden state $(x^T(N), \theta^T(N))^T$ given the training set $\mathcal{D} = \{u(t), y^*(t)\}_{t=1}^N$. The posteriori probability function is approximated by a Gaussian density function with :

$$\text{Mean} = \begin{bmatrix} \hat{x}(N|N) \\ \hat{\theta}(N|N) \end{bmatrix}; \ \text{Variance} = P(N|N),$$

which are obtained recursively via Equations (7.19) to (7.27) until $t = N$. Equivalently, this posteriori probability can be expressed as follows :

$$\mathcal{P}(\phi(N)|\mathcal{D}) = \hat{\mathcal{P}}(\phi(N)|\mathcal{D}) \exp\left\{-\frac{1}{2}\left(\phi(N) - \hat{\phi}(N)\right)^T P^{-1}(N|N)\left(\phi(N) - \hat{\phi}(N)\right)\right\}, \quad (7.28)$$

where $\phi(N) = (x^T(N)\ \theta^T(N))^T$ and $\hat{\phi}(N) = (x^T(N|N)\ \theta^T(N|N))^T$.

$$\hat{\mathcal{P}}(\hat{\phi}(N)|\mathcal{D}) = \left[(2\pi)^{n_\theta + n}\det(P(N|N))\right]^{-1/2}$$

To remove excessive weights from the network, we start by considering the information given by this posteriori probability which is readily obtained from the extended Kalman filter training. In accordance with Equation (7.28), $P(N|N)$ provides the cue about the shape of the probability function describing $\phi(N)$. And assuming the probability distribution is Gaussian and decomposing $P(N|N)$ into four block matrix :

$$\begin{bmatrix} P_{xx}(N|N) & P_{x\theta}(N|N) \\ P_{x\theta}^T(N|N) & P_{\theta\theta}(N|N) \end{bmatrix}, \quad (7.29)$$

where $P_{xx}(N|N) \in R^{n\times n}$, $P_{x\theta}(N|N) \in R^{n\times n_\theta}$ and $P_{\theta\theta}(N|N) \in R^{n_\theta \times n_\theta}$, the posteriori probability function for $\theta(N)$ can be approximated by integrating Equation (7.28) with respect to $x(N)$:

$$\begin{aligned} \mathcal{P}(\theta(N)|\mathcal{D}) &= \int \mathcal{P}(x(N), \theta(N)|\mathcal{D})dx(N) \\ &= \hat{\mathcal{P}}_{\theta|\mathcal{D}} \exp\left\{-\frac{1}{2}\left(\theta(N) - \hat{\theta}(N)\right)^T P_{\theta\theta}^{-1}(N|N)\left(\theta(N) - \hat{\theta}(N)\right)\right\}, \quad (7.30) \end{aligned}$$

where

$$\hat{\mathcal{P}}_{\theta|\mathcal{D}} = \left[(2\pi)^{n_\theta}\det(P_{\theta\theta}(N|N))\right]^{-1/2}.$$

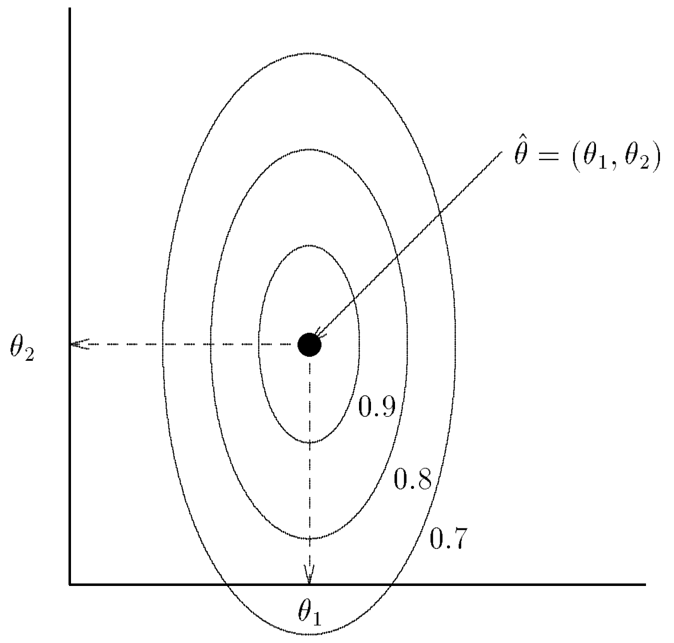


Figure 7.1: The idea of probabilistic pruning. We use the extended Kalman filter based training method to obtain an approximated posteriori probability distribution for the network parametric vector θ. It is a Gaussian distribution with mean $\hat{\theta}$, where $\hat{\theta} = (\theta_1, \theta_2)$, and variance P. The *a posteriori* probability is $\mathcal{P}(\hat{\theta}|\mathcal{D})$. Suppose, θ_2 is set to be zero, the posteriori probability reduces to around $0.75\mathcal{P}(\hat{\theta}|\mathcal{D})$. If θ_1 is set to be zero, the posteriori probability will reduce to smaller than $0.7\mathcal{P}(\hat{\theta}|\mathcal{D})$. This suggests that θ_2 should be ranked higher and this should be eliminated before θ_1, since the posteriori probability is not that sensitive to the change of θ_2 compared with θ_1.

Let $\hat{\theta}^k(N)$ be the parametric vector with all elements equal to $\hat{\theta}(N|N)$ except that the k^{th} element is set to be zero, i.e.

$$\hat{\theta}^k(N) = [\hat{\theta}_1(N|N) \ldots \hat{\theta}_{k-1}(N|N)\ 0\ \hat{\theta}_{k+1}(N|N) \ldots \hat{\theta}_{n_\theta}(N|N)]^T.$$

Thus the posteriori probability of $\hat{\theta}^k(N)$ given $\mathcal{D}$ can readily be written as follows :

$$\mathcal{P}(\hat{\theta}^k(N)|\mathcal{D}) = \hat{\mathcal{P}}_{\theta|\mathcal{D}} \exp\left\{-\frac{1}{2}\hat{\theta}_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}\right\}, \tag{7.31}$$

where $\left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$ is the k^{th} diagonal element of the inverse of $P_{\theta\theta}(N|N)$. Obviously, the smaller the value of the factor $\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$, the higher the posteriori probability $\mathcal{P}(\theta^k(N)|\mathcal{D})$. Figure 7.1 depicts the graphical interpretation of the above idea.

Suppose that the posteriori probability is a Gaussian distribution with mean $\hat{\theta}$, where $\hat{\theta} = (\theta_1, \theta_2)$, and variance P. The *a posteriori* probability is $\mathcal{P}(\hat{\theta}|\mathcal{D})$. Suppose, θ_2 is set to zero, the posteriori probability reduces to around $0.75\mathcal{P}(\hat{\theta}|\mathcal{D})$. If θ_1 is set to zero, the

posteriori probability will reduce to a value smaller than $0.7\mathcal{P}(\hat{\theta}|\mathcal{D})$. This suggests that θ_2 should be eliminated, as the posteriori probability is not that sensitive to the change of θ_2 compared with θ_1.
Hence, the pruning procedure can be summarized as follows :

1. **Initialization.**
   (a) Setting $\hat{\theta}(0)$ as small random vector.
   (b) Setting $P(0|0)$ to be a diagonal matrix with very large value.
   (c) Setting $x(0|-1)$ to be a small random vector.
2. **Training**
   (a) Using the recursive equations to obtain $\hat{\theta}(N)$.
   (b) Checking the before-pruning one-step prediction error, denoted by E_{bp}, based on the method described in Appendix.
3. **Pruning**
   (a) Decomposing the matrix $P(N|N)$ into block matrix to get $P_{\theta\theta}$.
   (b) Evaluating $P_{\theta\theta}^{-1}$ and hence $\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$ for all k from 1 to n_θ.
   (c) Rearranging the saliency index $\{\pi_k\}$ according to the ascending order of $\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$.
   (d) Set $k = 1$.
   (e) While $(k \leq n_\theta)$,
      i. Setting θ_{π_k} to zero.
      ii. Checking the validation error.
      iii. Setting θ_{π_k} back to its original value if the validation error is greater than a predefined threshold.
      iv. $k = k + 1$.

The validation error is used for checking whether or not the weight should be removed. The pruning does not stop until all the weights have been checked. In contrast to the conventional pruning procedures such as OBD or OBS, the pruning procedure stops once the validation error is larger than a threshold value. In some problems, if the available data set is small, we can simply treat the training data set as the validation set in Step 3e.

Remark :

It should be noted that there are other proposals that share similar ideas to those presented here. Finoff *et al* (1993) used the ratio of the weight magnitude over its fluctuation as a measure of the importance of a weight. The fluctuation is approximated by the standard deviation of the change of the weight value during training. Cottrell *et al.* (1995) independently suggested an idea where the fluctuation is approximated by $\theta_k^2 \left(\Sigma^{-1}\right)_{kk}$ where Σ is the Hessian matrix describing the shape of the error surface and simultaneously the shape of probability distribution around the estimated weight vector. Larsen (1996) generalized

Methods	Measure of weight Importance	Retrain	Reestimate	Model
OBD	Error Sensitivity	$\surd$	$\times$	FNN
OBS	Error Sensitivity	$\times$	$\surd$	FNN
Finoff *et al.* (1993)	Sensitivity in Prob.	$\surd$	$\times$	FNN
Cottrell *et al.* (1995)	Sensitivity in Prob.	$\surd$	$\times$	FNN
Larsen (1996)	Sensitivity in Prob.	$\times$	$\surd$	FNN
Ours	Sensitivity in Prob.	$\times$	$\times$	RNN

Table 7.1: Summary of difference pruning method. As Larsen (1996) has unified the idea of statistical pruning to both OBD and OBS, we include them in this table for comparison.

these ideas together with OBD and OBS under a statistical pruning framework. All these methods and ours share one similar point : *If the mean value of a weight is small but its value's fluctuation is large, this weight should not be an important weight. It should be removed with higher priority.* However, there are three major differences between our approach and theirs : (a) The *problem* we are dealing with is recurrent neural network pruning while all the aforementioned papers contribute to feedforward neural network pruning. (b) The *motivation* of the probabilistic pruning idea is due to the using of the extended Kalman filter approach to train the neural network. (c) *Practically*, we suggest no retraining and reestimation of the network weights. Besides, we have no problem in determining the number of weights to be removed. For clarity, we summarize the similarities and differences between their pruning methods and ours in Table 7.1.

7.4 Evaluation of the validation error.

To do a one-step prediction based on the trained recurrent network, we need to again apply the extended Kalman filter equation. Once the training has been finished and thus the network parameter θ fixed, we still have the only hidden variable $x(t)$ as an unknown. To predict the output $y(t)$ based on the past information $y(t-1), \ldots, y(1)$ and $u(t), u(t-1), \ldots, u(1)$, we first estimate the value of state $x(t)$ and then use this value to predict the output $y(t)$. The estimation of state $x(t)$ can readily be achieved by using Kalman filter equations.

Recall that the model of a recurrent network can be defined as a state-space model :

$$
\begin{aligned}
x(t+1) &= g(Ax(t) + Bu(t) + D) + v(t), && (7.32)\\
y(t+1) &= Cx(t+1) + w(t). && (7.33)
\end{aligned}
$$

Here, we assume that both the hidden state and output are contaminated by zero mean Gaussian noise. Let P_x, L_x be the estimated covariance matrix and the Kalman gain respectively for the estimate x. Moreover, we let Q_x and R_x be the system noise covariance and the measurement noise covariance. The prediction $y(t|t-1)$ will then be obtained via the following equations :

$$
\begin{aligned}
x(t|t-1) &= g(Ax(t-1|t-1) + Bu(t) + D) && (7.34)\\
P_x(t|t-1) &= F_x(t-1)P_x(t-1|t-1)F_x^T(t-1) + Q_x(t-1) && (7.35)
\end{aligned}
$$

$$
\begin{aligned}
y(t|t-1) &= Cx(t|t-1) && (7.36)\\
x(t|t) &= x(t|t-1) + L_x(t)\left(y^*(t) - y(t|t-1)\right) && (7.37)\\
P_x(t|t) &= P_x(t|t-1) - L_x(t)CP_x(t|t-1), && (7.38)
\end{aligned}
$$

where

$$F_x(t+1) = \partial_x g(x(t|t), u^*(t+1), \theta(t)), \qquad (7.39)$$

$$L_x(t) = P_x(t|t-1)C[CP_x(t|t-1)C^T + R_x(t-1)]^{-1} \qquad (7.40)$$

Here, we encounter the same problem as in the case of training. We do not have any information about the covariance matrix R_x and Q_x. So, we need to apply the same technique to estimate the values for both of them :

$$
\begin{aligned}
R_x(t) &= (1-\alpha_R)R_x(t-1) + \alpha_R(y^*(t) - y(t|t-1))^2 && (7.41)\\
Q_x(t) &= (1-\alpha_Q)Q_x(t-1) + \alpha_Q L_x(t)L_x(t)^T(y^*(t) - y(t|t-1))^2. && (7.42)
\end{aligned}
$$

The above equations can be viewed as follows. Suppose, $x(t-t|t-1)$ is the near optimal estimation of $x(t-1)$ based on the information given up to the $(t-1)^{st}$ time step. In accordance with these values, we can estimate $x(t)$ from Equation (7.34) and predict the output $y(t|t-1)$ as well. Then, once we have the true value $y^*(t)$, the one-step prediction error can be obtained. Based on this prediction error, we can proceed to estimate the value of $x(t)$ using Equation (7.37) and to update the system noise covariance and measurement noise covariance using Equations (7.41) and (7.42). Let the number of testing data be N_t, the one-step prediction error can be defined as $\frac{1}{N_t}\sum_{i=1}^{N_t}(y^*(i) - y(t|t-1))^2$. However, as we know that the value $x(t)$ estimated during the transient stage are usually not near optimal, the prediction would inevitably be large. It is better to define the prediction error based on the output value when the estimation of $x(t)$ has reached near optimal. Of course, we do not know when this will happen. So we simply assume that the estimation of $x(t)$ will reach near optimal when $t > 0.1N_t$. Thus, the prediction error is defined as follows :

$$E_{test} = \frac{1}{N_t - \tau}\sum_{i=\tau+1}^{N_t}(y^*(i) - y(t|t-1))^2, \qquad (7.43)$$

where $\tau = 0.1N_t$.

7.5 Simulation results

In this section, we will demonstrate the efficacy of the proposed pruning scheme through three examples. The first one is a simple time series prediction problem. The second one is a non-linear single input single output system identification problem which is adapted from Narendra and Parthasarathy (1992). The third one is a real life example, the prediction of exchange rate.

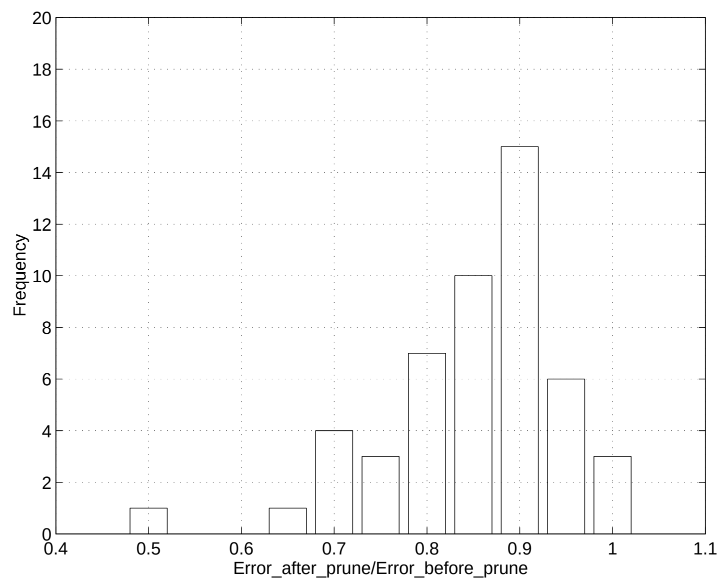


Figure 7.2: The histogram of the ratio $\frac{err_{ap}}{err_{bp}}$. The vertical axis corresponds to the frequency while the horizontal axis corresponds to the ratio. In almost all cases, the ratios are smaller than one. This means that pruning can improve generalization ability.

7.5.1 Simple linear time series

The linear time series we used is defined as follows:

$$y(t) = 0.6sin(t/30) + 0.2sin(t/5) + 0.01w(t),$$

where $w(t)$ is a zero mean unit variance Gaussian noise. Eight hundred data are generated. Six hundred are used for training while two hundred are used for testing. The recurrent network is constituted by one input unit, six hidden units and one output unit. Thus the total number of weights is 54. The weight values are initialized randomly around zeros with small variance. The same experiment is repeated for fifty trials.

Let err_{bp} and err_{ap} be the testing error before and after pruning respectively. The threshold is set to be the training error. Figure 7.2 and Figure 7.3 show the statistics of the ratio $\frac{err_{ap}}{err_{bp}}$ and the number of weights removed for these fifty trials. It is observed that : (i) the ratios of err_{ap} and err_{bp} are smaller than one in 47 out of 50 cases, and (ii) in most of the cases, the number of weights being removed is equal to or more than 20, which is about 0.37 of the total number of weights. This indicates that the above algorithm is able to improve the generalization of a recurrent neural network in simple linear time series.

7.5.2 Non-linear system identification

We follow an example mentioned in [82]. The plant is described by the second-order non-linear difference equations as follows:

$$\tilde{x}_1(k+1) = \frac{\tilde{x}_1(k) + 2\tilde{x}_2(k)}{1 + \tilde{x}_2^2(k)} + u(k) \tag{7.44}$$

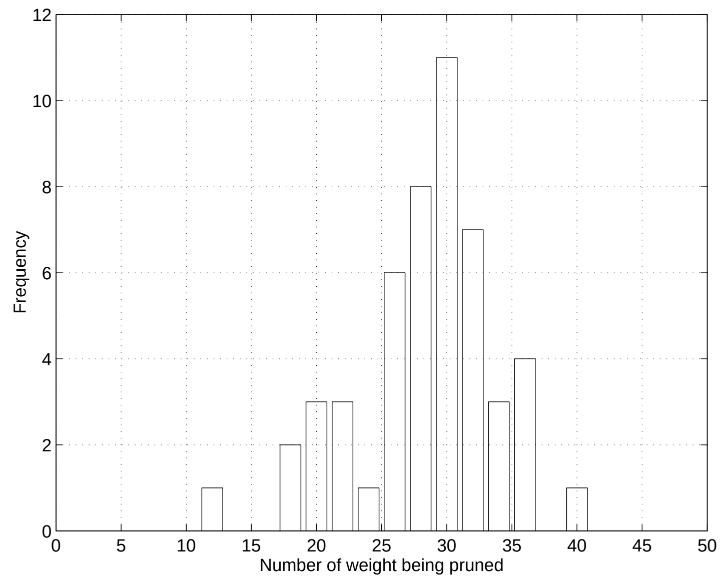


Figure 7.3: The histogram of the number of weights pruned away. The horizontal axis corresponds to the number of weights pruned away while the vertical axis corresponds to the frequency. In most of the trials, the number of weights pruned away is more than thirty which is about 3/5 of the total number of weight connections.

$$\tilde{x}_2(k+1) = \frac{\tilde{x}_1(k)\tilde{x}_2(k)}{1+\tilde{x}_2^2(k)} + u(k) \tag{7.45}$$

$$y_p(k) = \tilde{x}_1(k) + \tilde{x}_2(k) \tag{7.46}$$

The recurrent network is composed of one output unit $(y(k))$, ten hidden units $(x(k))$ and one input unit $(u(k))$. During training, an iid random signal with a uniform distribution over $[-1, 1]$ was used as the input to the non-linear plant. One thousand input-output pairs are generated. The first eight hundred data are used for training while the last two hundred are used for validation.

In this example, as the input data are random signal and the system is complex, eight hundred data are not sufficient for network training. Therefore, we have to retrain the network using the same training set several times. Thus we suggest the following alternative steps for the network training:

1. Passing through the training sequence $\{u(k), y^*(k)\}_{k=1}^{N}$ and use the Kalman filter formulation to learn the network parameters
2. Fixing the network parameters and passing the validation data set.
3. Evaluating the validation error by comparing the network output with the actual plant output.
4. Going to Step 1 if the validation error does not converge.
5. Stopping the training process if the validation error converges.

Training	
No. of hidden units	10
Initial hidden activity	$0.01 \times \ randno$
Initial weight	$0.0004 \times \ randno$
α_R	0.005
α_Q	0.005
$P(0)$	$30I_{140\times140}$
$Q(0)$	$I_{140\times140}$
$R(0)$	1
Testing	
α_R	0.005
α_Q	0.005
$P_x(0)$	$30I_{10\times10}$
$Q_x(0)$	$I_{10\times10}$
$R_x(0)$	1

Table 7.2: The values of the parameters used for non-linear system identification.

Once the training process is finished, we follow the steps stated in Section 4 to prune the network. The training parameters are depicted in Table 7.2. Figure 8.1a and 8.1b show the result after the network training has been finished. Figure 8.1a shows the output of the network and the output of the plant. It is observed that the recurrent network can closely predict the plant output.

To demonstrate that generalization can be improved if some weights in the neural network have been pruned, we feed in another set of input signals, a sine signal in this example, to test the pruned recurrent neural network and the non-linear plant. Figure 8.1b shows the case when the input is fed with this sine signal:

$$u(k) = sin\left(\frac{2\pi k}{25}\right).$$

δ	Validation error	Testing error	No of weights removed
Initial	0.3325	0.6976	–
0.1	0.3581	0.6068	15
0.2	0.3986	0.5962	15
0.3	0.4295	0.6058	21
0.4	0.4520	0.6892	22
0.5	0.4977	0.6747	25

Table 7.3: The pruning results for the non-linear system identification. The results are the average of 50 trials.

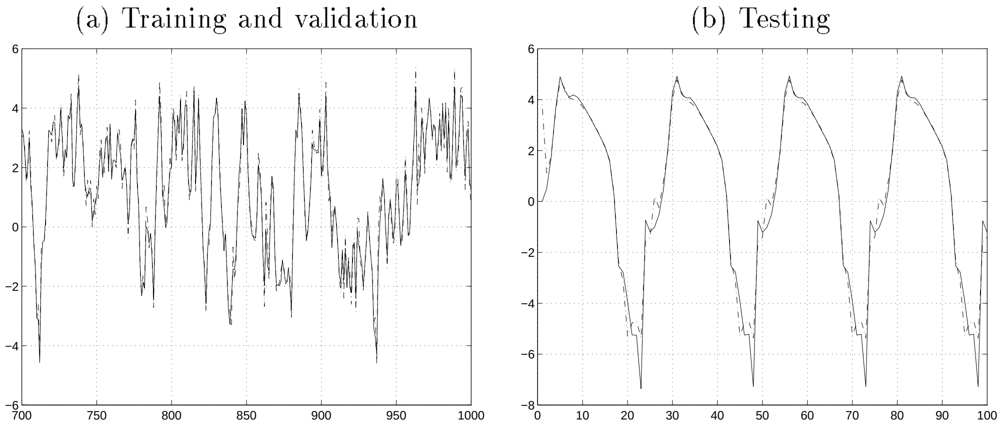


Figure 7.4: The neural network output against time when training is finished. The solid lines correspond to the output of the actual plant and the dot-dash lines correspond to the output of the recurrent network. (a) shows part of the training data and validation data when the input is random signal. (b) shows the testing data when the input is sine signal. The validation set are the data from 801 to 1000 in (a).

It shows that the trained network is able to model the non-linear system. Again, we let E_{bp} be the validation error before pruning. We define the threshold as $(1+\delta)E_{bp}$. Five values of δ $(0.1, 0.2, 0.3, 0.4, 0.5)$ are examined. The results are depicted in Table 7.3. Figure 7.5 shows the output of the network and the plant for the cases when δ is equal to 1/10.

By setting $\delta = 0.1$, the number of weights removed is increased to fifteen. Although the validation error with $\delta = 0.1$ is larger than the error before pruning and the testing error is smaller than the error before pruning, we can observe from Figure 7.5 that the network can still predict the output of the true system very close.

In conclusion, this example has illustrated that the proposed pruning scheme is able to improve the generalization ability of a recurrent network. Besides, this example also demonstrates that the setting of the threshold to a value larger than the validation error before pruning may lead to an even better neural network model with fewer weight connections but better generalization.

7.5.3 Exchange rate prediction

We apply the proposed method to predict the daily USD/DEM exchange rate $z(t)$. The range of the data are selected from Nov 29 1991 to Nov 3 1994, altogether 764 working days. The first 564 data are used as training data and the last 200 as testing data. The recurrent neural network model is constituted by one input unit, ten hidden units and one output unit. The input is fed with small random noise during training and the output of the network is to predict the $\{\ln\ z(t)\}$ sequence. Altogether, there are 130 connection weights. The training method is the same as the one we used in the last example except for the initial condition of the matrix Q. The data are passed to the network ten times. After each pass, the testing error is evaluated based on the method discussed in the Appendix.

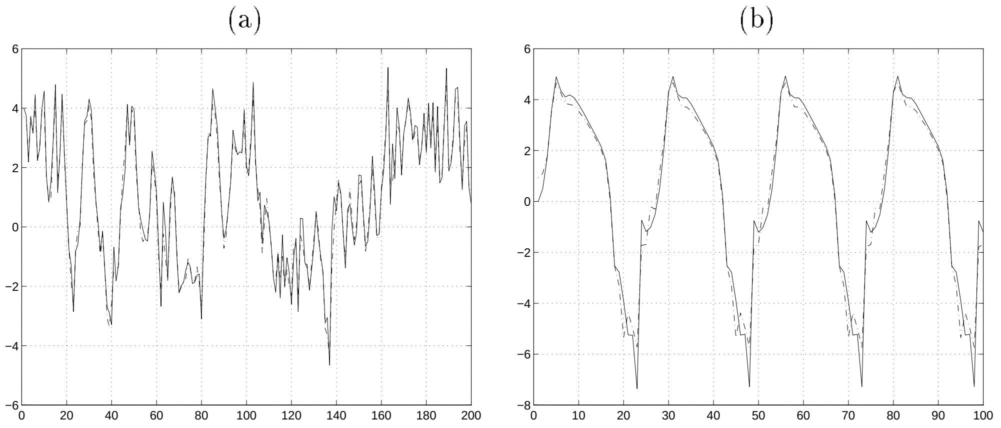


Figure 7.5: The output of the recurrent network after pruning with δ setting to be 1/10. The solid lines correspond to the actual plant output while the dot-dash lines correspond to the network output. (a) shows part of the training data and validation data when the input is random signal. (b) shows the testing data when the input is sine signal.

The training stops once the testing error converges. The training parameters are depicted in Table 7.4. As in the previous example, after training, we prune the network and check the resultant testing error. Seven values of δ are examined : $0\ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6$.

Table 7.5 and Figure 7.6 show the testing error and the number of weights being removed after pruning. It shows that as δ increases, more and more weights can be pruned away. The testing error reduces from 0.00080 to a minimum of 0.00032 when $\delta = 0.2$. This gives the best network architecture (with 49 connection weights) for this problem. As δ increases further, the testing error increases again. When $\delta = 0.5$, the testing error rises back to the level before pruning has taken place. At this moment, only 42 weights are left, compared to the original 130 weights.

7.6 The Saliency Ranking

We have shown that our pruning method can improve the generalization ability. Next, we closely examine the ranking generated from our saliency term.

7.6.1 Non-linear system identification

We repeat the experiment using the data from example 2 (Section 5.2). Everything remains the same except the pruning procedure Step 3(c) in Section 4. Now instead of using the ranking obtained from our saliency term, we replace it by a random list $\{\pi_1, \pi_2, \ldots, \pi_{n_\theta}\}$.

We set the threshold to be $(1+\delta)E_{bp}$, where δ is $0.1, 0.2, 0.3, 0.4$ and 0.5 respectively. The above experiment is repeated for 50 trials. The average testing errors and the average number of weights removed are shown in Figure 7.8. The dot-dash lines correspond to random ranking pruning while the solid lines correspond to the saliency ranking pruning.

Training	
No. of hidden units	10
Initial hidden activity	$0.001 \times$ *randno*
Initial weight	$0.0001 \times$ *randno*
α_R	0.005
α_Q	0.005
$P(0)$	$30I_{140\times140}$
$Q(0)$	$0.01I_{140\times140}$
$R(0)$	1
Testing	
α_R	0.005
α_Q	0.005
$P_x(0)$	$I_{10\times10}$
$Q_x(0)$	Q_e after training
$R_x(0)$	R after training

Table 7.4: The values of parameters used for the exchange rate prediction problem.

δ	Testing error	No. of weights removed
Before prune	0.00080	–
0.0	0.00037	48
0.1	0.00038	79
0.2	0.00032	81
0.3	0.00048	84
0.4	0.00054	85
0.5	0.00081	88
0.6	0.00113	93

Table 7.5: The pruning result for the exchange rate prediction problem.

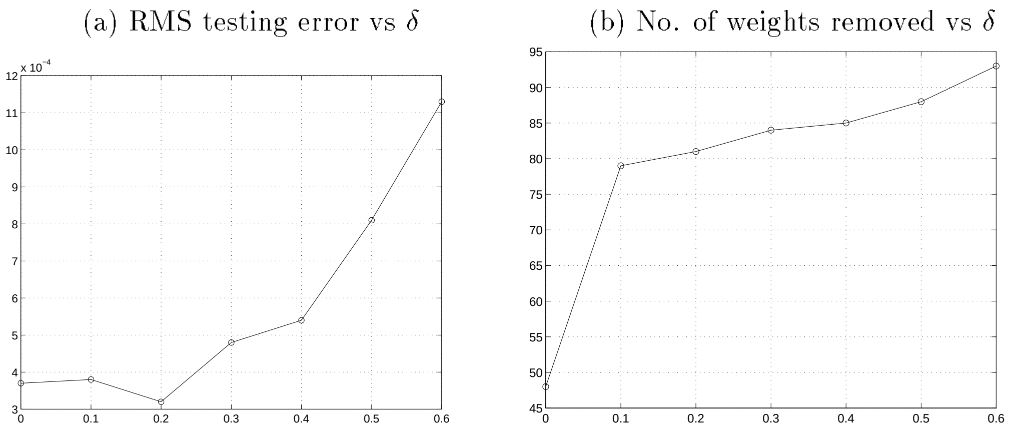


Figure 7.6: The mean square testing error and the number of weights removed against the value of δ in the prediction of exchange rate.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3281	0.6703	14
0.2	0.3976	0.7041	16
0.3	0.4280	0.7470	19
0.4	0.4613	0.8430	23
0.5	0.5011	0.9014	26

Table 7.6: The pruning results for the non-linear system identification when ranking heuristic is not imposed.

From these figures, we can observe that the number of weights being removed by both pruning methods are similar. Saliency ranking pruning however generates a smaller testing error compared with the random ranking. Beside, by inspecting the normalized frequency curves[2] — which show the normalized frequence of a weight being removed — we can even find that the heuristic rank list is a good *cue* for weight pruning.

Figure 7.7 shows the normalized frequency curve for the case when $\delta = 0.2$ and the weight ranking is initialized randomly. As random ranking indicates no information on which weight is not important, we have to search the whole list to make sure that the pruning is finished. It is rather time-consuming and with high computational cost.

If we check carefully the set of weights being removed and plot the normalized frequency curves against the saliency ranking list, Figure 7.9, it is found that those weights being pruned away based on random ranking are indeed located in the first half of the heuristic

[2]As we have repeated the experiment for 50 trials, the normalized frequency for the k^{th}-weight is defined as *total number of times it is being pruned/50.*

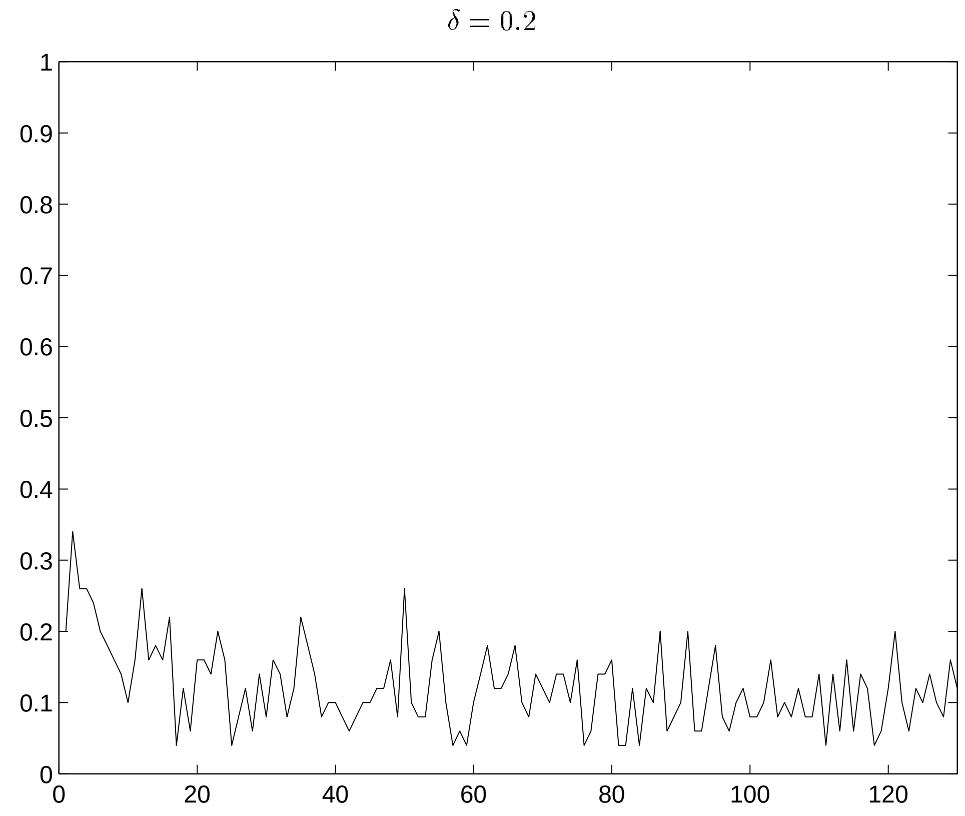


Figure 7.7: The normalized frequency curve for the random ranking case. The parameter δ is set to be 0.2 and the ranking of the weight importance is initialized randomly.

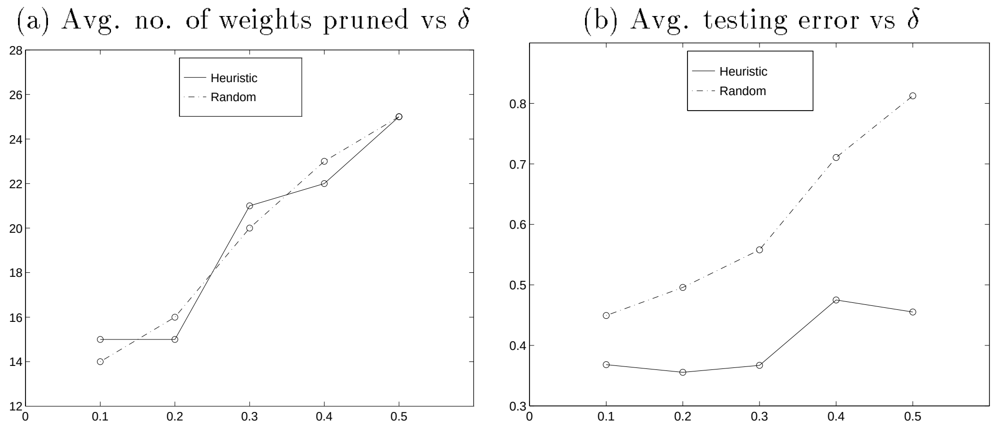


Figure 7.8: Comparison result between the cases when the proposed EKF-based heuristic pruning method is imposed (solid lines) and when the heuristic pruning procedure is not imposed (dot-dash lines).

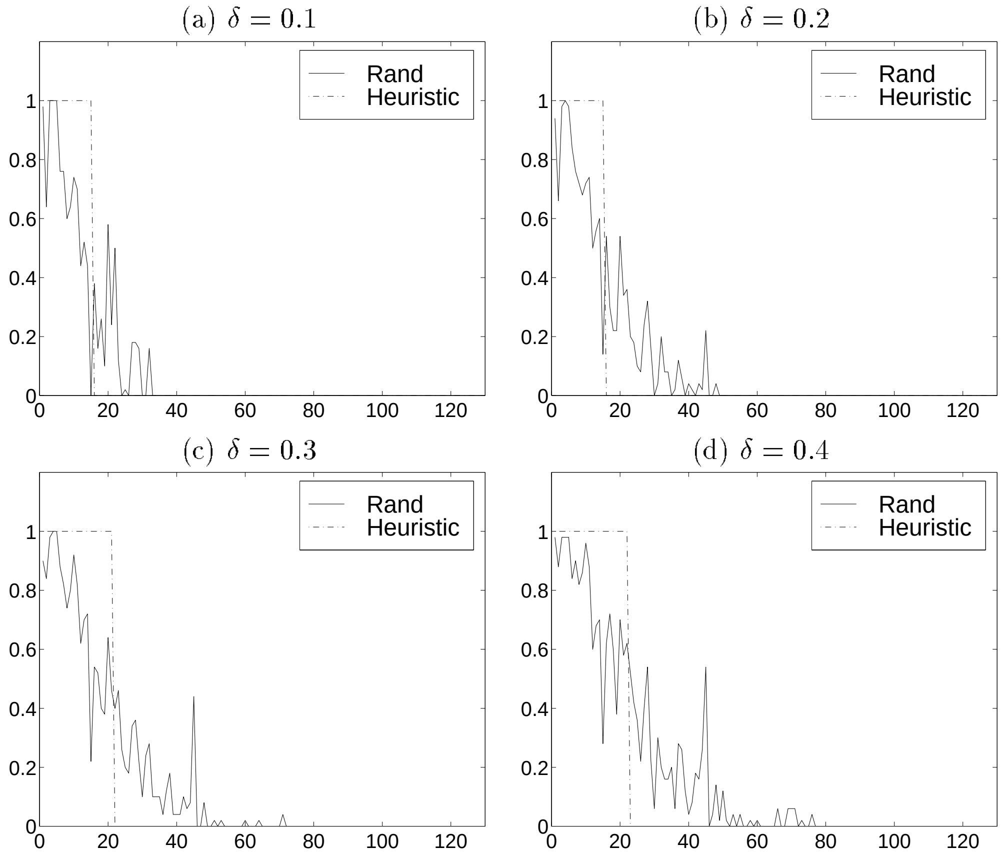


Figure 7.9: Comparison results between saliency rank pruning and random rank pruning. The dot-dash lines are the frequency curves for the heuristic case while the solid lines are the frequency curves for the random ranking case. The horizontal axis corresponds to the index of weight (according to the saliency ranking).

ranking list. This reveals that *no matter which weights are being pruned away by using random ranking or saliency ranking, the actual set of weights being removed is in fact located in the beginning portion of the saliency ranking list* and their chance of being removed decreases as their location is far from the beginning. That means, based on the EKF-based heuristic ranking list, searching the whole list is not necessary[3].

[3]It should be noted that it does not mean that the algorithm could be terminated once the first weight is found which cannot be pruned. Observed from Figure 7.9, it appears that the weights pruned using the saliency measure are always whose saliency falls below a certain threshold. It is not guaranteed in general. We have found experimentally that if δ is set to be 0.05 or smaller, the second weight on the list will not be pruned away but the third one will. Therefore, if the algorithm terminates once the first weight is found which cannot be pruned, we can only prune away one weight.

δ	Validation error	Testing error	No of weight removed
Initial	0.0769	0.0318	–
0	0.0426	0.0774	6
0.05	0.0458	0.0656	11
0.10	0.0412	0.0734	13
0.15	0.0512	0.0509	17
0.20	0.0410	0.0991	21

Table 7.7: The pruning result for a recurrent neural network being trained to identify a simple linear state-space model when the ranking heuristic is imposed.

7.6.2 Linear system identification

The model being discussed in this subsection is a simple linear state space model :

$$\begin{aligned} x_1(t+1) &= 0.7x_1(t) + 0.08x_2(t) + u(t) && (7.47) \\ x_2(t+1) &= x_1(t) && (7.48) \\ y(t) &= 0.22x_1(t). && (7.49) \end{aligned}$$

The recurrent neural network consisting of one input unit, five hidden units and one output unit is being trained. During training, 500 input-output data pairs are generated. The inputs are randomly (uniformly) drawn from $[-1, 1]$. After training, the network is pruned and tested by a new input signal (the testing set) :

$$u(k) = sin\left(\frac{2\pi k}{25}\right).$$

The experiment is repeated for 20 trials and the results obtained by using ranking are shown in Table 7.7. When the ranking heuristic is imposed, it is observed that the testing error increases progressively as δ increases. On the other hand, when the ranking heuristic is not imposed, the testing error suddenly increases when δ is increased from 0 to 0.05, Table 7.8. Similarly, this indicates that the ranking heuristic has encoded the importance of the weight with respect to the testing data.

7.7 Summary and Discussion

In summary, we have presented a pruning procedure for the recurrent neural network. The essence of this pruning method is the utilization of the result obtained from the extended Kalman filter training : the parametric vector $\hat{\theta}(N)$ and the covariance matrix $P(N|N)$. Instead of considering the error sensitivity as a measure of the importance of the weight, we take the *a posteriori* probability sensitivity. In accordance with the theory of optimal filtering, $\hat{\theta}(N)$ and $P(N|N)$ can be treated as an approximation of the mean and covariance of this *a posteriori* probability, the sensitivity can thus be calculated.

Applying this pruning method together with the recurrent neural network to three problems such as the prediction of a linear time series, the modeling of a non-linear system

δ	Validation error	Testing error	No of weight removed
Initial	0.0769	0.0318	–
0	0.0593	0.0603	5
0.05	0.0427	0.1580	9
0.10	0.0439	0.1400	16

Table 7.8: The pruning result for a recurrent neural network being trained to identify a simple linear state-space model and the ranking heuristic is not imposed.

and the prediction of the exchange rate, we observed that the proposed procedure not only can reduce the number of weights of a recurrent network, but can also improve the generalization ability of the networks.

Furthermore, we have also demonstrated that the heuristic ranking list is indeed a good *cue* for the removal of weight. Comparing the results with random ranking listing on two problems : a non-linear system identification and linear system identification, we observed that the ranking list generated by the factor $\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$ not only tells which weight is of more importance, but also helps to improve the generalization ability.

One should also note that there are many ways of applying the recursive least square method or extended Kalman filter in feedforward neural network training, [48, 97, 12, 89, 93, 95]. The pruning scheme proposed in this paper can be readily applied in conjunction with these methods. Besides, as the extended Kalman filter is an adaptive method for state estimation and parameter identification, we suspect that this pruning scheme can readily be extended as an adaptive pruning method and hence the search for a better model can also be feasible for time varying systems.

In recent years, a considerable amount of effort has been put into the generalization and the pruning of a feedforward neural network [91, 98, 122]. The generalization and the pruning problem of a recurrent neural network have rarely been discussed. With Pedersen and Hansen (1995) derived a recursive algorithm for the evaluating of the second order derivative of the error function so that pruning based on error sensitivity is possible. Wu and Moody (1996) derived a smoothing regularizer for recurrent neural networks so as to improve the RNN generalization ability. The results presented in this paper may shed some light on the development of a more effective pruning method.

Chapter 8

Alternative Pruning Methods for RNN

In this chapter, several alternative pruning methods will be described. Their pros and cons in terms of computation complexity and generalization behavior will be discussed. In section one, the EKF-based pruning method for RNN will be reviewed. The computational complexity of using such a pruning approach will be derived in section two. In section three, several alternative pruning methods which require no weight ranking initialization will be introduced. Their computational complexity will be derived in section four. Simulation results on a nonlinear system modeling will be presented in section five and a comparative analysis of the difference between these pruning methods will be presented in section six. Then, we conclude this chapter in section seven.

8.1 EKF-based Pruning for RNN

Using the EKF approach, the weight importance measure is defined by probability sensitivity. Let $\theta(t)$ and $x(t)$ be the weight vector and the hidden unit vector respectively at the t^{th} time step; $(\hat{\theta}(t|t), \hat{x}(t|t))$ be the estimate of $(\theta(t), x(t))$ at the t^{th} time step via EKF equations.

$$(\hat{\theta}(t|t), \hat{x}(t|t)) \approx \arg \max_{(\theta(t), x(t))} \mathcal{P}(\theta(t), x(t)|\{u_i, y_i\}_{i=1}^{t}). \tag{8.1}$$

for $t \geq 1$. Assuming that

$$\mathcal{P}(\theta(t), x(t)|\{u_i, y_i\}_{i=1}^{t}) = \mathcal{P}(\theta(t)|\{u_i, y_i\}_{i=1}^{t})\mathcal{P}(x(t)|\{u_i, y_i\}_{i=1}^{t})$$

for N is larger, the weights importance are ranked in according with their sensitivities to the *a posterior* probability $\mathcal{P}(\theta(t)|\{u_i, y_i\}_{i=1}^{t})$, i.e.

$$\mathcal{P}(\hat{\theta}(N|N)|\{u_i, y_i\}_{i=1}^{N}) - \mathcal{P}(\hat{\theta}^k(N|N)|\{u_i, y_i\}_{i=1}^{N}),$$

where $\hat{\theta}^k(N|N)$ is identical to $\hat{\theta}(N|N)$ except that the k^{th} element is null. The least importance weight is the one with the least sensitivity.

Further assuming that both $\mathcal{P}(\hat{\theta}(N|N)|\{u_i, y_i\}_{i=1}^N)$ and $\mathcal{P}(x(t)|\{u_i, y_i\}_{i=1}^t)$ converge for $t \geq N$, the predicted output probability,

$$\begin{aligned}
& \mathcal{P}(y(t+1)|\{u_i, y_i\}_{i=1}^t, \hat{\theta}(N|N)) \\
\approx & \int\int \mathcal{P}(y(t+1)|\theta(t), x(t))\mathcal{P}(\theta(t), x(t)|\{u_i, y_i\}_{i=1}^t)\delta(\theta(t) - \hat{\theta}(N|N))d\theta(t)dx(t) \\
\approx & \int\int \mathcal{P}(y(t+1)|\theta(t), x(t))\mathcal{P}(\theta(t)|\{u_i, y_i\}_{i=1}^t)\mathcal{P}(x(t)|\{u_i, y_i\}_{i=1}^t)\delta(\theta(t) - \hat{\theta}(N|N))d\theta(t)dx(t) \\
\approx & \mathcal{P}(\hat{\theta}(N|N)|\{u_i, y_i\}_{i=1}^t)\int \mathcal{P}(y(t+1)|\hat{\theta}(N|N), x(t))\mathcal{P}(x(t)|\{u_i, y_i\}_{i=1}^t)dx(t), \qquad (8.2)
\end{aligned}$$

where $\delta(\theta(t)-\hat{\theta}(N|N))$ is the delta function centered at $\hat{\theta}(N|N)$. Suppose that $\mathcal{P}(x(t)|\{u_i, y_i\}_{i=1}^t)$ converges to a Gaussian distribution with constant variance and the mean is equal to the true $x(t)$, the predicted output probability distribution,

$$\int \mathcal{P}(y(t+1)|\hat{\theta}(N|N), x(t))\mathcal{P}(x(t)|\{u_i, y_i\}_{i=1}^t)dx(t),$$

will also converge to a Gaussian distribution with constant variance and the mean is equal to the true $y(t+1)$. The probabilistic sensitivity measure evaluated using the estimated $\hat{\theta}(N|N)$ can approximate the probabilistic sensitivity measure for the validation data set. That is to say, the measure is not just meaningful for the training data set, but also meaningful for the validation data set or testing data set in the sense of probability measure.

As presented in the last chapter, the sensitivity measure of the i^{th} weight can be represented by

$$\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk},$$

where $P_{\theta\theta}(N|N)$ is the approximated covariance matrix for $\hat{\theta}(N|N)$ and $\left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$ is the k^{th} diagonal element of its inverse. Two pruning procedures can be defined. If stop pruning criterion is included, a *no-skipping procedure* can be defined as follows :

1. **Initialization.**
   (a) Setting $\hat{\theta}(0)$ as small random vector.
   (b) Setting $P(0|0)$ to be a diagonal matrix with very large value.
   (c) Setting $x(0|-1)$ to be a small random vector.
2. **Training**
   (a) Using the recursive equations to obtain $\hat{\theta}(N)$.
   (b) Checking the before-pruning one-step prediction error, denoted by E_{bp}.
3. **Pruning (no-skipping)**
   (a) Decomposing the matrix $P(N|N)$ into block matrix to get $P_{\theta\theta}$.
   (b) Evaluating $P_{\theta\theta}^{-1}$ and hence $\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$ for all k from 1 to n_θ.

(c) Rearranging the saliency index $\{\pi_k\}$ according to the ascending order of

$$\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}.$$

(d) Setting $k = 1$.

(e) While validation error is less than a threshold,

   i. Setting θ_{π_k} to zero and checking the validation error.

   ii. $k = k + 1$.

(f) Setting θ_{π_k} back to its original value

If stop pruning criterion is not included, a *skipping pruning procedure* can be defined as follows :

1. **Initialization.** Same as above procedure.
2. **Training** Same as above procedure.
3. **Pruning (with skipping)**
   (a) Decomposing the matrix $P(N|N)$ into block matrix to get $P_{\theta\theta}$.
   (b) Evaluating $P_{\theta\theta}^{-1}$ and hence $\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}$ for all k from 1 to n_θ.
   (c) Rearranging the saliency index $\{\pi_k\}$ according to the ascending order of

$$\theta_k^2 \left(P_{\theta\theta}^{-1}(N|N)\right)_{kk}.$$

   (d) Setting $k = 1$.
   (e) While $(k \leq n_\theta)$,
      i. Setting θ_{π_k} to zero and checking the validation error.
      ii. Setting θ_{π_k} back to its original value if the error is greater than a predefined threshold.
      iii. $k = k + 1$.

The validation error is used for checking whether or not the weight should be removed. The pruning does not stop until all the weights have been checked. In some problems, if the available data set is small, we can simply treat the training data set as the validation set in Step 3e. In contrast to the conventional pruning procedures such as OBD or OBS, the pruning procedure stops once the validation error is larger than a threshold value.

8.2 High computational cost in building ranking list for the EKF-based pruning method

Suppose the number of input dimensions is m (a small constant), the number of hidden units is n and the number of output unit is one, the total number of weights in RNN will be given by

$$n_\theta = mn + n^2 + 2n = (m + n + 2)n. \qquad (8.3)$$

During training, vector x and vector θ are put together as a single state vector, hence the dimension of this augmented vector will be equal to $(m+n+3)n$. Ignoring the computational cost of training and the computational cost of matrix decomposition, the major costs of pruning will be determined by the following steps :

1. Evaluating $P_{\theta\theta}^{-1}$
2. Rearranging the saliency index in the ascending order
3. Checking the validation error

Cost of evaluating $P_{\theta\theta}^{-1}$ The cost of this step is easily evaluated. Generally, it requires n_θ^3 multiplication. From Equation (8.3), the number of multiplications in this step is $(m+n+2)^2n^2$.

Cost of rearranging saliency index This step is to sort n_θ scalar values in ascending order. It requires $n_\theta \log n_\theta$ comparison[1], i.e. $(m+n+2)n\log(m+n+2)n$ multiplication steps.

Cost of checking validation error We can see that in the validation phase, an extended Kalman filter is running in order to update the vector x. Ignoring the addition operation, the number of multiplications for updating the variables, such as the vector x and the matrix P, are depicted in Table 8.1. Summing up all their cost, it can readily be shown that the total number of multiplications in one iteration is $2n^3+9n^2+nm+4n+5$. If stop-pruning criterion is applied, the cost will be

$$(\text{no. of weight pruned}) \times (2n^3+9n^2+nm+4n+5).$$

If stopping criterion is not applied, i.e. skipping is allowed, the computational cost will be

$$(m+n+2) \times (2n^3+9n^2+nm+4n+5).$$

The burden is in the step of updating the $P_x(t|t-1)$ matrix. Let C_{mul} and C_{comp} be the costs of a single multiplication and a single comparison respectively, the number of multiplication steps in using EKF based pruning will be given by

$$\begin{aligned} Cost_{ekfp} &= (m+n+2)n\log(m+n+2)nC_{comp} + (m+n+2)^3n^3C_{mul} \\ &\quad +(m+n+2)n(2n^3+9n^2+nm+4n+5)C_{mul}. \end{aligned}$$

Obviously, this computational cost will be dominated by the cost in calculating the inverse of the matrix $P(N|N)$, i.e. the factor $(m+n+2)^3n^3C_{mul}$. For example, if $n=10$ and $m=1$,

$$\begin{aligned} (m+n+2)n\log(m+n+2)n &\approx 633. \\ (m+n+2)^3n^3 &= 2197000. \\ (m+n+2)n(2n^3+9n^2+nm+4n+5) &= 150150. \end{aligned}$$

[1]Suppose heap sort is used.

Table 8.1: The number of multiplications in each equation of the EKF in validation. Note that n is the dimension of vector x.

$x(t\|t-1)$	$n^2 + nm$
$P_x(t\|t-1)$	$2n^3$
$y(t\|t-1)$	n
$x(t\|t)$	n
$P_x(t\|t)$	$2n^2$
$F_x(t+1)$	n^2
$L_x(t)$	$n^2 + 1$
$R_x(t)$	$2n^2 + n + 2$
$Q_x(t)$	$2n^2 + n + 2$
Total	$2n^3 + 9n^2 + nm + 4n + 5$

The overhead in calculating the inverse of matrix $P(N|N)$ is ten times more than the actual computational cost in evaluating the validation error. If stopping criterion is included in the algorithm, the complexity ratio, i.e.

$$\frac{\text{Cost of ranking list generation}}{\text{Cost of validation}}.$$

will be much larger.

As the inverse of $P(N|N)$ is needed only for building the ranking list, it will be worthwhile to see whether we can simply discard the ranking list or seek an alternative ranking list. Furthermore, it is also interesting to look into whether there is any alternative procedure which can better utilize the computational resource.

8.3 Alternative methods for RNN pruning

As the actual relationship between the heuristic derived saliency measure and validation error is vague, using generalization error (testing error) to determine the performance of a pruning algorithm can only be a reference. The computational cost of the pruning algorithm and the size of the resultant pruned network should be better criteria to determine the performance of a pruning algorithm.

To reduce the pruning complexity, there are other tricks that can be applied. An intuitive one is weight magnitude. Using error sensitivity (ES) or probabilistic sensitivity (PS) approaches, one can see that the saliency of a weight is proportional to the square of its magnitude.

$$\begin{aligned} \text{ES}: &\quad saliency = (\nabla\nabla_{\theta_k\theta_k} E)\theta_k^2 \\ \text{PS}: &\quad saliency = (\nabla\nabla_{\theta_k\theta_k} \mathcal{P})\theta_k^2 \end{aligned}$$

Apart from using weight magnitude as a ranking heuristic, we can also introduce other tricks to further improve the performance of the algorithm.

Skipping/No-skipping As a weight being removed will cause a sudden increase in the validation error, do we have to stop the prune and recover the weight or do we simply recover the weight and move on to the next weight on the list ? Obviously, in terms of network size, allowing skipping should result in a better performance.

Single level/Multiple level Suppose skipping is allowed, could there be further weight removal if we repeat the pruning loop one or several times more ?

Heuristic initial rank/Random initial rank As the removal of weight is in regard to the ranking list, this list is generated at the initial stage, by using probabilistic sensitivity, error sensitivity, weight magnitude or simply random.

Fixed ranking/Dynamic ranking Usually, the weight importance ranking list will be used throughout the whole pruning process once it is generated in the initial stage. Dynamic ranking allows us to regenerate the ranking list each loop of pruning. That is to say, once all the weights have been checked and some of them have been pruned, it is intuitive to regenerate the ranking list. In order not to introduce additional cost, this new list can be generated by using the validation error.

The algorithms defined in this section are defined in accordance with one or more of the above factors.

8.3.1 Single level pruning procedure : Algorithm 1 and 2

Before proceeding, let us define two general pruning algorithms. The first one is called the single level no skipping procedure. The second one is called the single level skipping procedure.

Algorithm 1 (Single level no skipping procedure)

```
1. Initialize ranking list $\{\pi_1, \pi_2, \ldots, \pi_n\}$.
2. Set $E_v = 0$,
3. Set $k = 1$,
4. While ($E_v \leq (1+\delta)E_{tr}$) Begin
   (a) Set $\theta_{\pi_k} = 0$
   (b) Evaluate validation error, $E_v(\theta_{\pi_k} = 0)$
   (c) $k = k + 1$
   End
```

δ is a nonzero value determining the level of tolerance. Three algorithms can be defined with respect to their initial ranking list generation. The ranking list can be generated by random, by means of weight magnitude or by using probability sensitivity.

Algorithm 2 (Single level skipping procedure)

```
1. Initialize ranking list $\{\pi_1, \pi_2, \ldots, \pi_n\}$.
2. For $k = 1$ to $n$,
   Begin
```

```
    (a) Set $tmp = \theta_{\pi_k}$
    (b) Set $\theta_{\pi_k} = 0$
    (c) Evaluate validation error, $E_v(\theta_{\pi_k} = 0)$
    (d) If $E_v(\theta_{\pi_k} = 0) \geq (1+\delta)E_{tr}$, set $\theta_{\pi_k} = tmp$.
  End
```

Similarly, δ is a nonzero value determining the level of tolerance and three algorithms can be defined with respect to their initial ranking list generation. The ranking list can be generated by random, by means of weight magnitude or by using probability sensitivity.

8.3.2 Multilevel procedure : Algorithm 3 and 4

One can imagine, after one pass, the behavior of the network will be changed. Some un-pruned weights which are important in the very first step may not be important after the first pass. In that case, checking the importance of all the un-pruned weights after each pass could probably remove some more weights. Hence a smaller size neural network could be obtained.

Algorithm 3

```
1. Initialize ranking list $\{\pi_1, \pi_2, \ldots, \pi_n\}$.
2. While (Total no. of pruned weights increases)
   Begin: For $k = 1$ to $n$, Begin
    (a) Set $tmp = \theta_{\pi_k}$
    (b) Set $\theta_{\pi_k} = 0$
    (c) Evaluate validation error, $E_v(\theta_{\pi_k} = 0)$
    (d) If $E_v(\theta_{\pi_k} = 0) \geq (1+\delta)E_{tr}$, set $\theta_{\pi_k} = tmp$.
   End End
```

Once again δ is a nonzero value determining the level of tolerance. The ranking list can be generated by random, by means of weight magnitude or by using probability sensitivity. Three algorithms can be defined. The stopping criterion for these algorithms is as follows : the pruning procedure will be stopped if no more weight can be removed after several passes.

Since the computation cost in validation is far below that of generating the ranking list, we can even make use of the validation error to generate the ranking list.

Algorithm 4 (One weight at a time pruning)

```
1. Initialize ranking list $\{p_1, p_2, \ldots, p_n\}$.
2. While (Total no. of pruned weight increases) Begin
    (a) Set $\pi = p$
    (b) For $j = k$ to $n$, Begin
        i. Set $tmp = \theta_{\pi_k}$
        ii. Set $\theta_{\pi_k} = 0$
```

	Prob. heurist	Weight mag. heurist	Random
NS	1a	1b	–
SLSP	2a	2b	2c
MLSP	3a	3b	3c
OWAT	–	–	4

Table 8.2: A summary of the pruning algorithms being proposed. NS stands for *no-skipping* pruning procedure. SLSP stands for *single level with skipping pruning.* MLSP stands for *multilevel skipping pruning procedure.* OWAT stands for *one weight at a time* pruning procedure.

Algorithm	Initial ranking	Level	Skipping	Ranking regeneration
1a	prob. sensitivity	Single	No	No
1b	weight magnitude	Single	No	No
2a	prob. sensitivity	Single	Yes	No
2b	weight magnitude	Single	Yes	No
2c	Random	Single	Yes	No
3a	prob. sensitivity	Multiple	Yes	No
3b	weight magnitude	Multiple	Yes	No
3c	Random	Multiple	Yes	No
4	Random	Multiple	Yes	Yes

Table 8.3: Characteristics of different pruning algorithms being discussed in the paper.

```
            iii. Evaluate the validation error, $E_v(\theta_{\pi_k} = 0)$
            iv. Set $\theta_{\pi_k} = tmp$
            End
        (c) Regenerate ranking list $\{p_k, \ldots, p_n\}$ in according to the ascending order of the
            validation error.
        (d) Set $\theta_{p_1} = 0$ if $E_v(\theta_{p_1} = 0) \leq (1 + \delta)E_{tr}$
    End
```

For clarity, Table 8.3 summarizes the essential features of these pruning algorithms for comparison. Note that the Algorithm 1c is not included. The computational requirement of each algorithm is summarized in Table 8.4.

8.4 Complexity of the alternative pruning procedures

Using the same technique in Section 8.2, the computation complexity of the presented pruning procedures can readily be derived. The notations used in derivation are depicted

Algorithm	Initial ranking	Ranking regeneration
1a	Matrix inversion & Sorting	No
1b	Sorting	No
2a	Matrix inversion & Sorting	No
2b	Sorting	No
2c	No	No
3a	Matrix inversion & Sorting	No
3b	Sorting	No
3c	No	No
4	No	Sorting

Table 8.4: Computational requirement for different pruning algorithms

Notation	Meaning
n	Number of hidden units
m	Number of input units
T	Length of the validation data
C_{comp}	Cost of comparing two floating point numbers
C_{mul}	Cost of multiplying two floating point numbers
n_{1a}	Total number of weights being pruned using Algorithm 1a
n_{1b}	Total number of weights being pruned using Algorithm 1b
L_{3a}	Total number of levels required using Algorithm 3a
L_{3b}	Total number of levels required using Algorithm 3b
L_{3c}	Total number of levels required using Algorithm 3c
L_4	Total number of levels required using Algorithm 4

Table 8.5: Summary of notation

in Table 8.5. The computational complexity of the above algorithms include three essential components :

- Cost of the generation of initial pruning order list $\{\pi_1, \pi_2, \cdots, \pi_{n_\theta}\}$.
- Cost of the evaluation of validation error

$$\frac{1}{T}\sum_{k=1}^{T}(y(k) - y^*(k))^2,$$

  where $y^*(k)$ is the target output of the system while $y(k)$ is the output of the neural network.

In Section 8.2, we have derived the cost of generating the initial list for EKF based pruning. For those alternative pruning algorithms, this cost can be neglected. If the initial list is

Algo.	Computational complexity
1a	$(m+n+2)^3n^3C_{mul} + ((m+n+2)n\log(m+n+2)n)C_{comp}$
	$+n_{1a}T(2n^3+9n^2+nm+4n+5)C_{mul}$
1b	$((m+n+2)n\log(m+n+2)n)C_{comp}$
	$+n_{1b}T(2n^3+9n^2+nm+4n+5)C_{mul}$
2a	$(m+n+2)^3n^3C_{mul} + ((m+n+2)n\log(m+n+2)n)C_{comp}$
	$+(m+n+2)T(2n^3+9n^2+nm+4n+5)C_{mul}$
2b	$((m+n+2)n\log(m+n+2)n)C_{comp}$
	$+(m+n+2)T(2n^3+9n^2+nm+4n+5)C_{mul}$
2c	$(m+n+2)T(2n^3+9n^2+nm+4n+5)C_{mul}$
3a	$(m+n+2)^3n^3C_{mul} + L_{3a}((m+n+2)n\log(m+n+2)n)C_{comp}$
	$+nT(2n^3+9n^2+nm+4n+5)C_{mul}$
3b	$((m+n+2)n\log(m+n+2)n)C_{comp}$
	$+L_{3b}nT(2n^3+9n^2+nm+4n+5)C_{mul}$
3c	$L_{3c}nT(2n^3+9n^2+nm+4n+5)C_{mul}$
4	$L_4[nT(2n^3+9n^2+nm+4n+5)C_{mul}$
	$+((m+n+2)n\log(m+n+2)n)C_{comp}]$

Table 8.6: Summary of the computational complexity for different pruning algorithms. Note that the total number of weight in a recurrent neural network is in the order of $\mathcal{O}(n^2)$.

generated by random, this cost is null. If the initial list is generated in regard to the weight magnitude, the cost would simply be the cost for sorting $(n+m+2)$ real numbers, i.e. $(n+m+2)\log(n+m+2)C_{comp}$. This cost is still small. Table 8.6 summarizes the complexities of each pruning algorithm. In case n is large, the asymptotic computational complexity bounds of the presented pruning algorithms are shown in Table 8.7. From the fact that $n_{1a}, n_{1b} < n$ and assuming that L_{3a}, L_{3b} and L_{3c} are much smaller than n,[2] it can easily be seen that the complexity of EKF based pruning (including 1a, 2a and 3a) is indeed much larger than the other presented algorithms, even if the weights are removed one at a time (algorithm 4). Their complexities can be ranked as follows :

$$1b < 2b, 2c < 3b, 3c < 4, 5 < 1a < 2a < 3a.$$

The EKF based pruning requires much larger computational cost than the other one. Weight magnitude based ranking requires the least computational cost.

8.5 Experimental comparison on generalization and network size

In the previous section, we have analyzed the performance of different pruning algorithms in terms of their computation complexities. In this section, we will look into their performance in terms of their generalization ability.

[2]We will confirm this assumption in the next section.

Algo.	Computational complexity
1a	$\mathcal{O}(n^6 C_{mul} + n_{1a} T n^3 C_{mul})$
1b	$\mathcal{O}(n_{1b} T n^3 C_{mul})$
2a	$\mathcal{O}(n^6 C_{mul} + T n^4 C_{mul})$
2b	$\mathcal{O}(T n^4 C_{mul})$
2c	$\mathcal{O}(T n^4 C_{mul})$
3a	$\mathcal{O}(n^6 C_{mul} + L_{3a}((m+n+2)n\log(m+n+2)n)C_{comp} + T n^4 C_{mul})$
3b	$\mathcal{O}(L_{3a}((m+n+2)n\log(m+n+2)n)C_{comp} + T n^4 C_{mul})$
3c	$\mathcal{O}(L_{3c} T n^4 C_{mul})$
4	$\mathcal{O}(L_4 T n^4 C_{mul})$

Table 8.7: Summary of the asymptotic bound computational complexity for different pruning algorithms assuming that $m \ll n$.

A recurrent neural network is first trained, using the extended Kalman filter approach, to identify a nonlinear system which is defined as follows :

$$\tilde{x}_1(k+1) = \frac{\tilde{x}_1(k) + 2\tilde{x}_2(k)}{1 + \tilde{x}_2^2(k)} + u(k) \tag{8.4}$$

$$\tilde{x}_2(k+1) = \frac{\tilde{x}_1(k)\tilde{x}_2(k)}{1 + \tilde{x}_2^2(k)} + u(k) \tag{8.5}$$

$$y_p(k) = \tilde{x}_1(k) + \tilde{x}_2(k) \tag{8.6}$$

The recurrent network is composed of one output unit ($y(k)$), ten hidden units ($x(k)$) and one input unit ($u(k)$). During training, an iid random signal with a uniform distribution over $[-1, 1]$ is used as the input to the nonlinear plant. One thousand input-output pairs are generated. The first eight hundred data are used for training while the last two hundred are used for validation.

As the input data is random signal and the system is complex, eight hundred data are not sufficient for network training. Therefore, we have to retrain the network using the same training set several times. The training parameters are depicted in Table 8.8. Figure 8.1a shows the output of the network and the output of the plant. To demonstrate that generalization can be improved if some weights in the neural network have been pruned, we feed in another input signal, a sine signal in this example, to test the pruned recurrent neural network and the nonlinear plant. Figure 8.1b shows the case when the input is fed with this sine signal:

$$u(k) = sin\left(\frac{2\pi k}{25}\right).$$

It shows that the trained network is able to model the non-linear system. Again, we let E_{bp} be the validation error before pruning. We define the threshold as $(1+\delta)E_{bp}$. Five values of δ $(0.1, 0.2, 0.3, 0.4, 0.5)$ are examined.

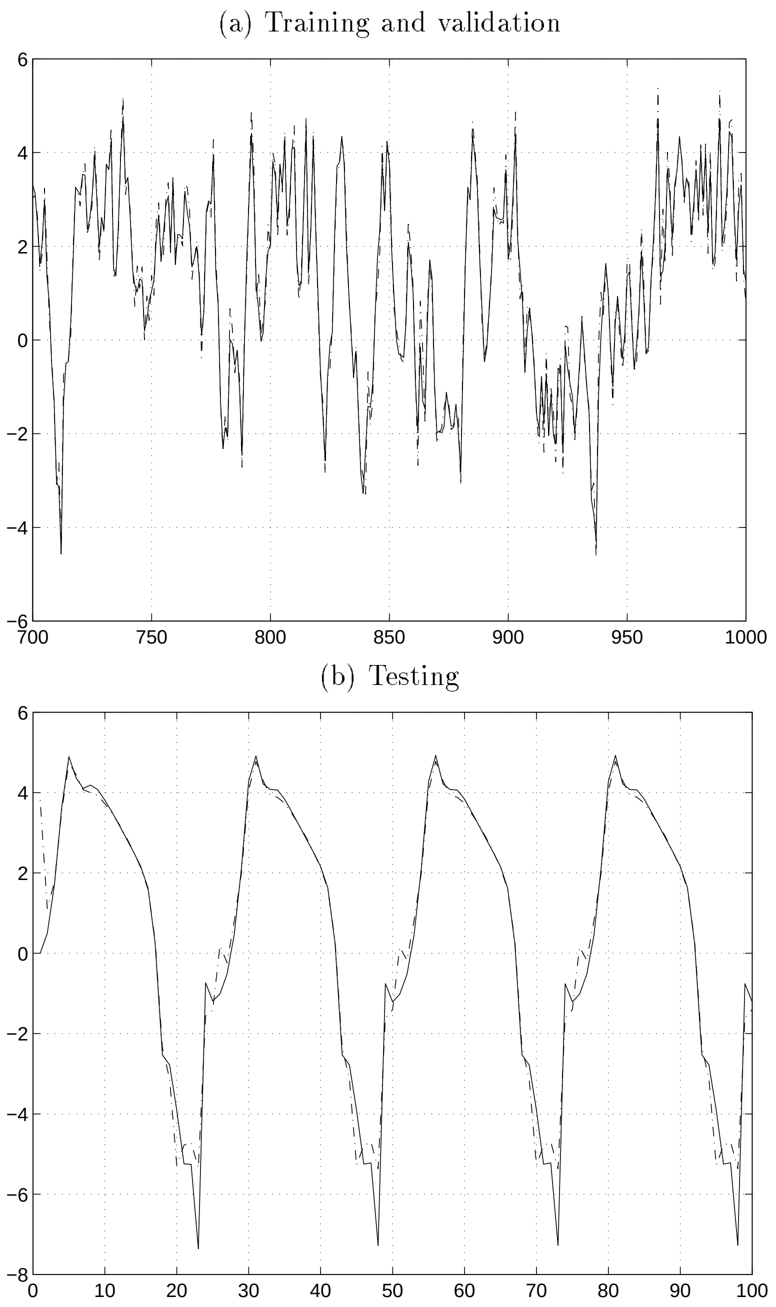


Figure 8.1: The result after the network training is finished. The solid lines correspond to the output of the actual plant and the dot-dash lines correspond to the output of the recurrent network. (a) shows part of the training data and validation data when the input is random signal. (b) shows the testing data when the input is sine signal. The validation set are those data from 801 to 1000 in (a).

Training	
No. of hidden units	10
Initial hidden activity	$0.01 \times$ *randno*
Initial weight	$0.0004 \times$ *randno*
α_R	0.005
α_Q	0.005
$P(0)$	$30 I_{140\times 140}$
$Q(0)$	$I_{140\times 140}$
$R(0)$	1
Testing	
α_R	0.005
α_Q	0.005
$P_x(0)$	$30 I_{10\times 10}$
$Q_x(0)$	$I_{10\times 10}$
$R_x(0)$	1

Table 8.8: The values of those parameters used for nonlinear system identification.

8.5.1 Algorithm 1

Obviously, if the initial ranking list is generated by random, pruning is not possible. Therefore, for Algorithm 1, we concentrate on the probabilistic heuristic and weight heuristic only. Table 8.9 and Table 8.10 summarize the results for networks being pruned by using the probabilistic heuristic and weight heuristic respectively. The skipping procedure is not imposed in these algorithms. It is found that the number of weights being removed by using either heuristic can be up to twenty. Besides, there exists a range of δ in which the testing error generated by the pruned network is smaller than the testing error generated by the original un-pruned network.

8.5.2 Algorithm 2

If the skipping procedure is imposed, it is found that the probabilistic measure does not always outperform the others. According to Table 8.11, 8.12, 8.13 and Figure 8.3, the probabilistic heuristic outperforms in the sense of generalization. In the sense of the number of weights being removed, the weight magnitude ranking and random ranking both perform similar to the probabilistic ranking. If the δ value is 0.1, all three algorithms can make the pruned network generalize better.

8.5.3 Algorithm 3

For the multi-level skipping procedure type, the results are different. For small values of δ, all three ranking systems perform similarly. All of them can prune the network to have better generalization, see Figure 8.4. However, for large values of δ, it is found that the random ranking method can prune more weights than the other two methods.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3687	0.6408	13
0.2	0.4095	0.6220	16
0.3	0.4391	0.6027	17
0.4	0.4780	0.6021	19
0.5	0.5102	0.6511	21

Table 8.9: The pruning results for the nonlinear system identification when the probabilistic ranking heuristic is imposed but the skipping procedure is not imposed.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3756	0.6277	9
0.2	0.4398	0.6415	11
0.3	0.4398	0.6415	11
0.4	0.5140	0.7362	21
0.5	0.4140	0.7362	21

Table 8.10: The pruning results for the nonlinear system identification when the weight magnitude ranking heuristic is imposed but the skipping procedure is not imposed.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3581	0.6068	15
0.2	0.3986	0.5962	15
0.3	0.4295	0.6058	21
0.4	0.4520	0.6892	22
0.5	0.4977	0.6747	25

Table 8.11: The pruning results for the nonlinear system identification when the probabilistic ranking heuristic together with the skipping procedure are imposed.

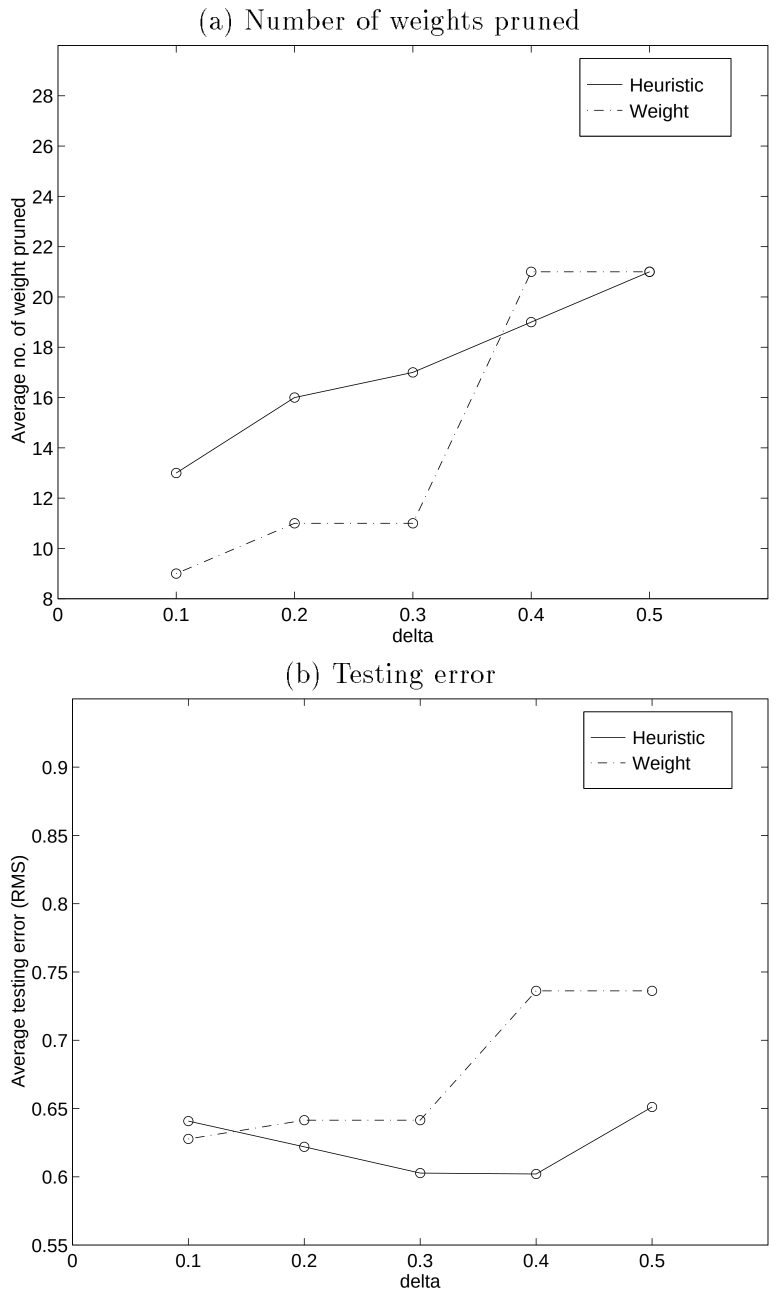


Figure 8.2: The pruning results for the nonlinear system identification when the skipping procedure is not imposed.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3571	0.6550	14
0.2	0.3959	0.6340	15
0.3	0.4234	0.6926	22
0.4	0.4655	0.7462	28
0.5	0.4903	0.7938	27

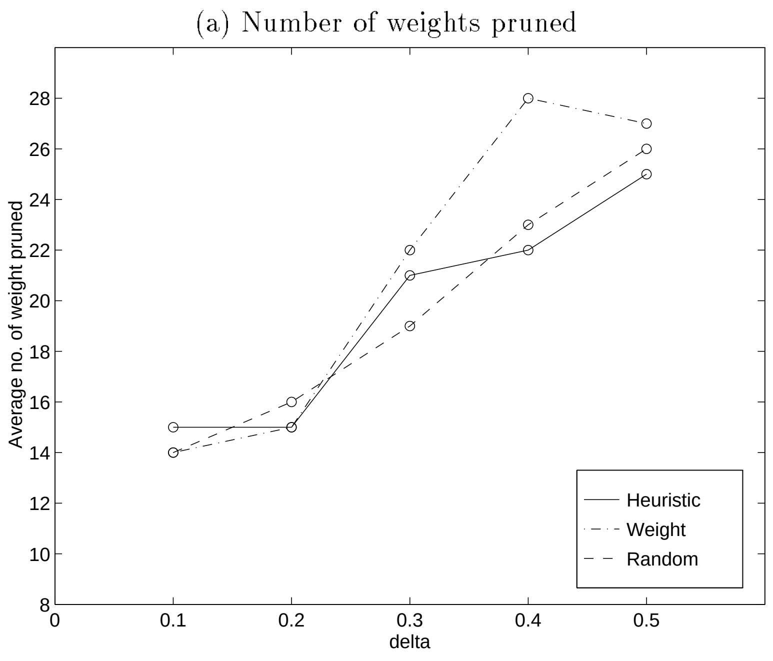
Table 8.12: The pruning results for the nonlinear system identification when the weight magnitude ranking heuristic together the skipping procedure are imposed.

δ	Validation error	Testing error	No. of weights removed
. Initial	0.3325	0.6976	–
0.1	0.3281	0.6703	14
0.2	0.3976	0.7041	16
0.3	0.4280	0.7470	19
0.4	0.4613	0.8430	23
0.5	0.5011	0.9014	26

Table 8.13: The pruning results for the nonlinear system identification when the random ranking together with the skipping procedure are imposed. The results are the average of 50 trials.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3618	0.6563	21
0.2	0.3986	0.6335	15
0.3	0.4322	0.6544	23
0.4	0.4603	0.7605	25
0.5	0.4929	1.0315	31

Table 8.14: The pruning results for the nonlinear system identification when the probabilistic ranking heuristic together with the multi-level skipping procedure are imposed.



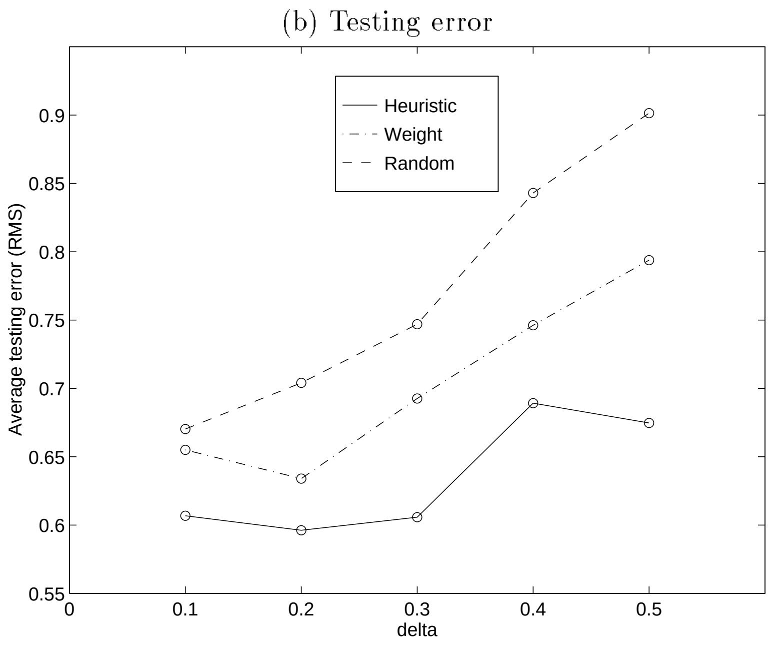


Figure 8.3: The pruning results for the nonlinear system identification when the single level skipping procedure is imposed.

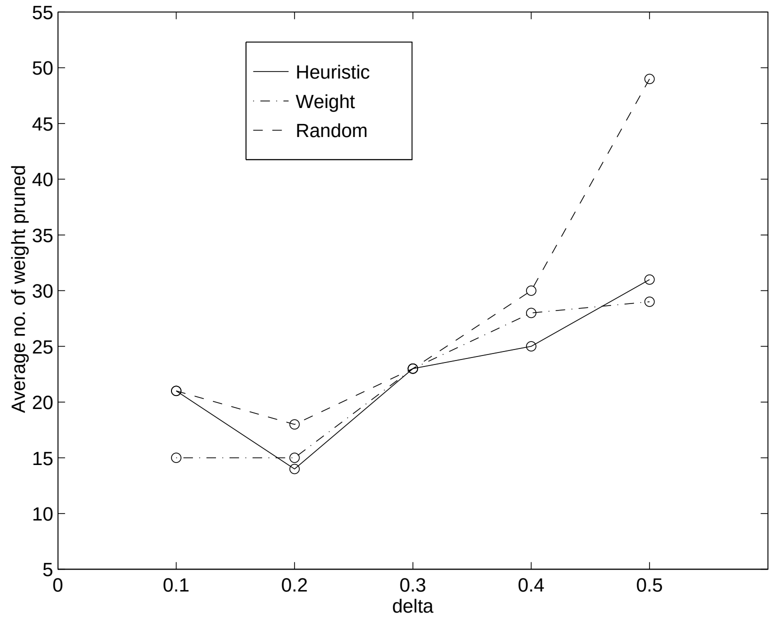
δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3643	0.6605	15
0.2	0.3959	0.6339	15
0.3	0.4315	0.8210	23
0.4	0.4655	0.7461	28
0.5	0.4976	0.8855	29

Table 8.15: The pruning results for the nonlinear system identification when the weight magnitude ranking heuristic together with multi-level skipping procedure are imposed.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3618	0.6547	21
0.2	0.3976	0.6257	18
0.3	0.4263	0.8015	23
0.4	0.4610	0.8575	30
0.5	0.4955	1.1965	49

Table 8.16: The pruning results for the nonlinear system identification when the random ranking together with multi-level skipping procedure are imposed. The results are the average of 50 trials.

(a) Number of weights pruned



(b) Testing error

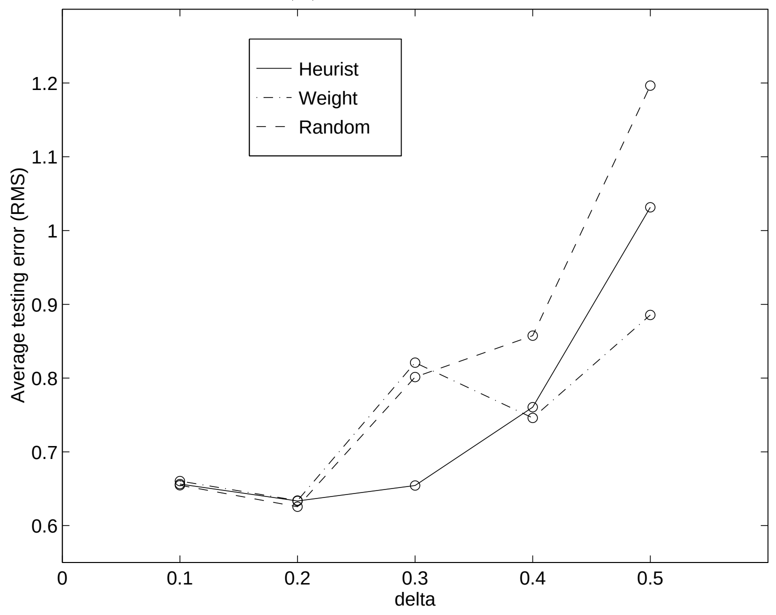


Figure 8.4: The pruning results for the nonlinear system identification when the multi-level skipping procedure is imposed.

δ	Validation error	Testing error	No. of weights removed
Initial	0.3325	0.6976	–
0.1	0.3810	0.6830	22
0.2	0.3969	0.6871	30
0.3	0.4048	0.7223	33
0.4	0.4350	0.7040	34
0.5	0.4762	0.8760	35

Table 8.17: The pruning results for the nonlinear system identification when the sequence re-generation strategy and the multi-level skipping procedure are imposed.

	Prob. heurist	Weight mag. heurist	Random
NS	$(0.4, 0.6021, 19)$	$(0.1, 0.6277, 9)$	–
SLSP	$(0.2, 0.5962, 15)$	$(0.2, 0.6340, 15)$	$(0.1, 0.6703, 14)$
MLSP	$(0.2, 0.6335, 16)$	$(0.2, 0.6339, 15)$	$(0.2, 0.6257, 18)$
OWAT	–	–	$(0.1, 0.6830, 22)$

Table 8.18: Best case comparison. NS stands for the *no-skipping* pruning procedure, i.e. Algorithm 1. SLSP stands for the *single-level-with-skipping* pruning procedure, i.e. Algorithm 2. MLSP stands for the *multilevel-skipping* pruning procedure, i.e. Algorithm 3. OWAT stands for the *one-weight-at-a-time* pruning procedure, i.e. Algorithm 4.

8.5.4 Algorithm 4

Table 8.17 shows the results when the ranking list is allowed to be re-generated after each pass. It is found that this procedure can greatly increase the number of weights being pruned. The number of weights being removed increases as the value of δ increases. However, it seems that the increasing rate drops as the value of δ increases and the number of weights being removed will approach a limit, around 37, see Figure 8.5a. In a case when the value of δ is smaller than or equal to 0.4, it is also observed that the pruned networks perform similarly. All of them generate similar testing error. Besides, when the value of δ is set to be less than or equal to 0.2, the pruned network manifests better generalization ability, see Figure 8.5b.

8.6 Analysis

The best case results are depicted in Table 8.18. Figure 8.6 plots the average testing error against the number of weights removed.

According to the above simulation results, we have the following observations :

- Using *probability sensitivity* to build the initial ranking list can lead to *better generalization.*

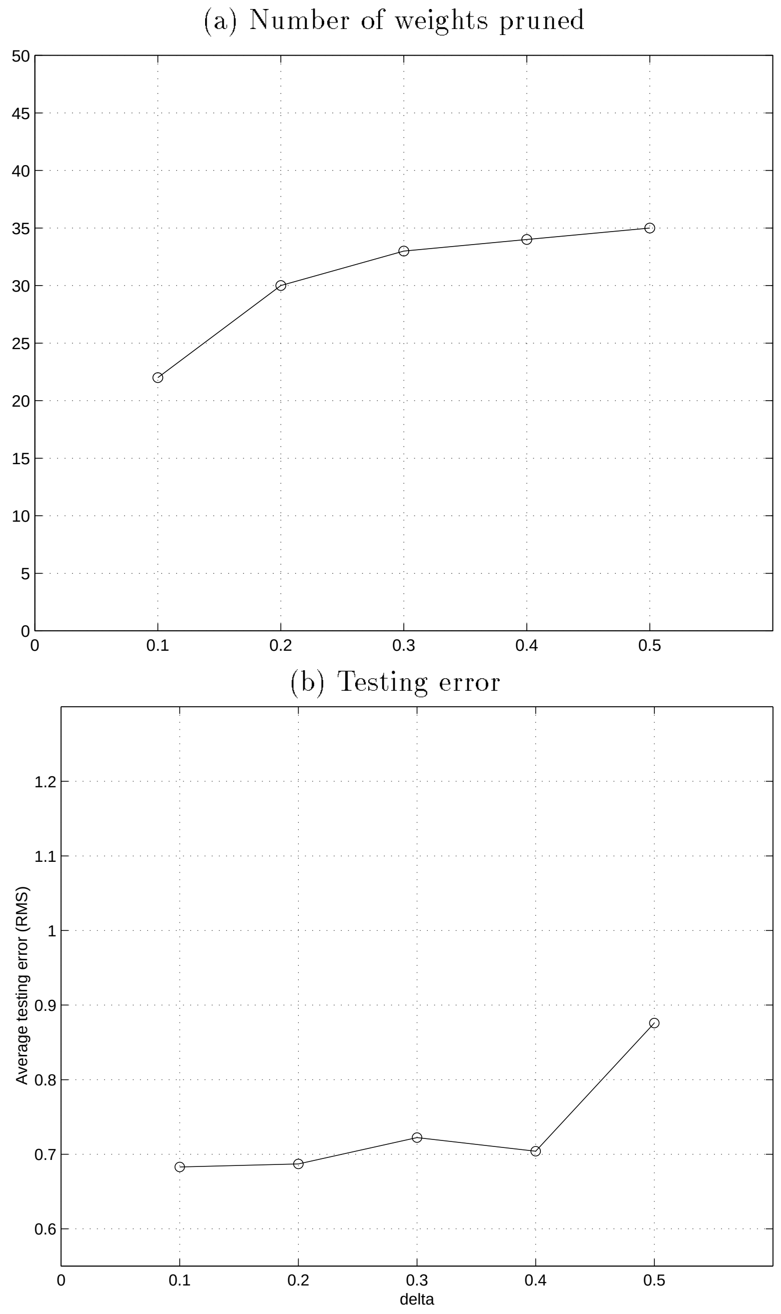


Figure 8.5: The pruning results for the nonlinear system identification when the sequence re-generation strategy and the multi-level skipping procedure are imposed.

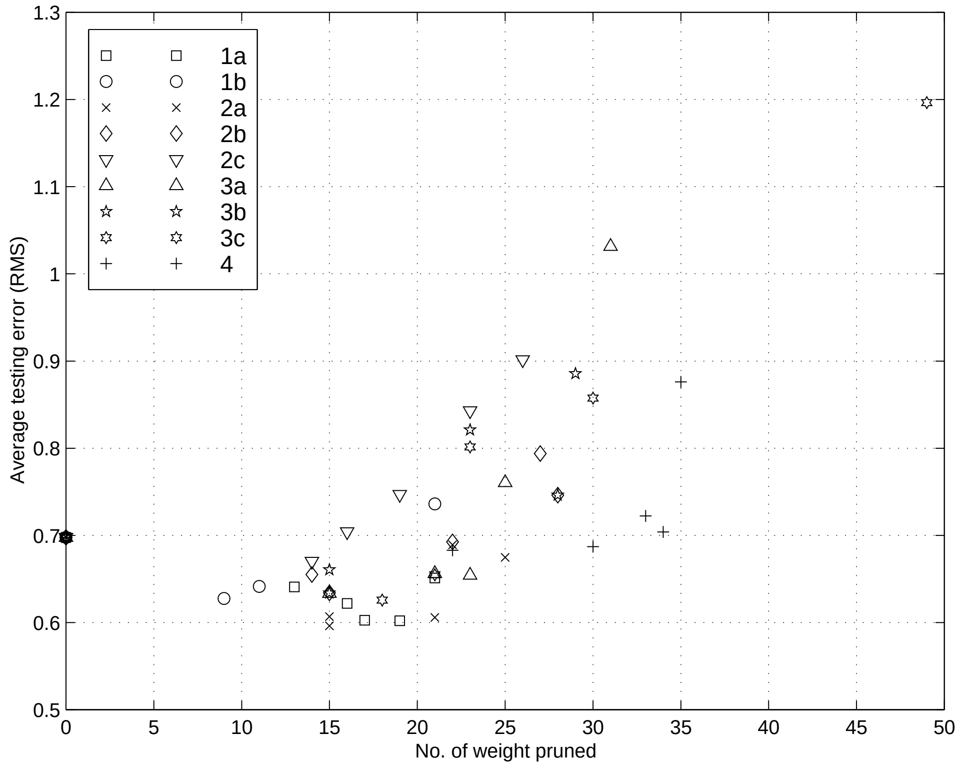


Figure 8.6: Summary of the pruning results for all the proposed pruning procedures.

Algorithm	Complexity[1]	Size[2]	Generalization[3]
1a	4	5	2
1b	1	9	4
2a	4	2	1
2b	2	4	7
2c	2	8	8
3a	4	3	5
3b	2	7	6
3c	2	6	3
4	3	1	9

Table 8.19: Comparison of the performance of different pruning procedures. [1]The ranking is in accordance with order of complexity. [2]The ranking is in according with the size of the pruned networks. At the same time, the networks must have a testing error smaller than 0.6976. The one with the least number of weights will be rank number one. [3]The ranking is simply following the data depicted in the last section.

- No matter what type of pruning procedure is used, *skipping procedure* can always lead to a *smaller size* neural network.
- For large δ values,the alternative pruning procedures can usually remove more weights than the probability based procedure. This is one interesting observation that we cannot explain at this point.

There are some other observations that we have made. They are not shown from the above simulation results.

- For large network size, both RTRL and second order training methods are not able to converge.
- The number of levels in Algorithm 3, such as L_{3a}, L_{3b} and L_{3c}, are smaller than 5. They are much smaller than the total number of weights in the recurrent neural network.
- The value of L_4 is equal to the number of weights being removed. This value is usually proportional to (but smaller than) the total number of weights in the network. That is to say, the order of L_4 should be in between $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$.

By comparing the size of the pruned network, the generalization ability and the pruning complexity (see Table 8.19), it is found that using the probability heuristic (EKF based pruning) together with the skipping procedure should be the best approach to pruning recurrent neural networks if pruning complexity is not considered. The weight magnitude heuristic with no skipping procedure is the one which requires the least computational cost. Whereas, *one weight at a time* (Algorithm 4) pruning can reduce a recurrent neural network to a smaller size.

8.7 Summary

In summary, we have reviewed the EKF-based pruning method for the recurrent neural network pruning. Because of its high computation costs, we thus proposed four alternative pruning algorithms trying to get rid of these problems. The essential idea behind these alternative algorithms is the inclusion of *skipping* and *re-pruning*. In contrast to the conventional method — once a removal of weight will cause a large degradation of the network performance, we simply recover the weight and let the pruning process continue for the next weight in the list until all the weights on the list have been checked. In this case, experimental results demonstrate that the size of the network can be further reduced without much loss to the generalization ability. Since the cost of checking the list one time or two time is just different by a constant, we proposed re-pruning. Once the whole list of weights have been checked, the pruning process re-runs again and again until no more weight can be removed. These skipping and re-pruning ideas are normally inefficient when we are dealing with feedforward neural network pruning. As pruning a recurrent neural network is already a difficult problem, skipping and re-pruning does not introduce much overhead.

The pruning performance of the presented pruning methods, in terms of *computational complexity, network size and generalization ability* are analyzed. No simple conclusion has yet been drawn from the analysis. No single algorithm is found to be effective in all three performance instances. The results presented in this chapter hopefully can offer certain guidelines for the practitioner if they really want to apply recurrent neural network in solving real life problems such as system identification and time series prediction.

Part IV

Several Aspects of Neural Network Learning

Chapter 9

Approximate Realization Property of RNN

Recurrent neural networks have been proposed for about a decade [94] and during that time many different recurrent neural network models have been proposed. The Jordan model [33] and Elman model [33] are two typical models in the literature of fully connected recurrent neural networks. Recently some other models have been proposed as well. Tsoi and Back [117] proposed a class of local recurrent global feedback networks for solving problems mainly in non-linear system identification and time series modeling. Giles and coworkers [25] proposed a second order recurrent neural network applied to grammar learning. Chen *et al.* [13] developed a recurrent radial basis function network for use in a non-linear system modeling problem. In applying such recurrent neural networks in system identification, one basic assumption is that the applied model is able to realize any non-linear system. That is to say, for any given state space non-linear system, such recurrent neural networks are able to model such behavior as accurately as possible.

In this chapter, we will attempt to discuss this approximate realization property. Using some of the latest theoretical results concerning the universal approximation property of some multilayered feedforward neural networks, we can readily show that the Jordan model and recurrent radial basis function networks are able to realize any non-linear state-space system with arbitrary accuracy. Besides, using the result derived by Funahashi [24], we can show that any fully connected recurrent neural network with hidden units being defined by sigmoid function or radial basis function is able to realize any discrete-time non-linear state-space system.

In section one, some recent results on the approximation property of multilayer perceptron and recurrent networks will be presented. The new result on the realization property of the Elman network will be given in Section two. Section three will present a conclusion for this chapter.

9.1 Recent Results

First of all, let us state three theorems without proof. The prove of the approximation capabilities of some recurrent network models will be based on these theorems. For simplicity of later discussion, some of these theorems are briefly described, reader can refer to the reference cited for the original statement of such theorems.

9.1.1 Multilayer perceptron

Theorem 3 (Funahashi Theorem[24]) *Let K be a subset of R^n and $f : K \to R^m$ be a continuous mapping. Then for an arbitrary $\epsilon > 0$, there exists a multilayer perceptron with finite number of hidden nodes such that*

$$\max_{x \in K} \|f(x) - W_1\sigma(W_2 x + \theta)\| < \epsilon \tag{9.1}$$

holds, where $\sigma : R^n \to R^n$ is a sigmoid mapping. The matrix W_1, W_2 corresponds to the hidden to output connection matrix and input to hidden connection matrix.

□□□

Theorem 4 (Girosi-Poggio Theorem[26]) *Let K be a subset of R^n and $f : K \to R^m$ be a continuous mapping. Then for an arbitrary $\epsilon > 0$, there exists a multilayer perceptron with finite number of hidden nodes such that*

$$\max_{x \in K} \|f(x) - W_1\eta(x, W_2)\| < \epsilon \tag{9.2}$$

holds, where $\eta : R^n \to R^n$ is a radial basis function.

□□□

Both of the above theorems show that a multilayer feedforward perceptron is a universal approximator no matter whether the hidden units are defined as a sigmoid or radial basis function. It should be remarked that some other researchers have also obtained the same result, using different approaches [21] [35].

9.1.2 Recurrent neural networks

Following Theorem 1, Funahashi and Nakamura further proved the approximate realization property of recurrent neural networks as stated in the following theorem.

Theorem 5 (Funahashi-Nakamura Theorem[24]) *Let D be an open subset of R^n, $f : D \to R^n$ be a C^1-mapping. Suppose that $\frac{dx}{dt} = f(x)$ be defined a dynamical system on D. Let K be a subset of D and we consider trajectories of the system on the interval $I = [0, t_0]$. Then, for any $\epsilon > 0$, there exists a recurrent neural network with n output and a finite number of hidden units such that for any trajectory of the system with initial value $x(0) \in K$ and an appropriate initial state of the network,*

$$\max_{t \in I} \|x(t) - y(t)\| < \epsilon \tag{9.3}$$

holds, where y is the internal state of the output units of the network.

□□□

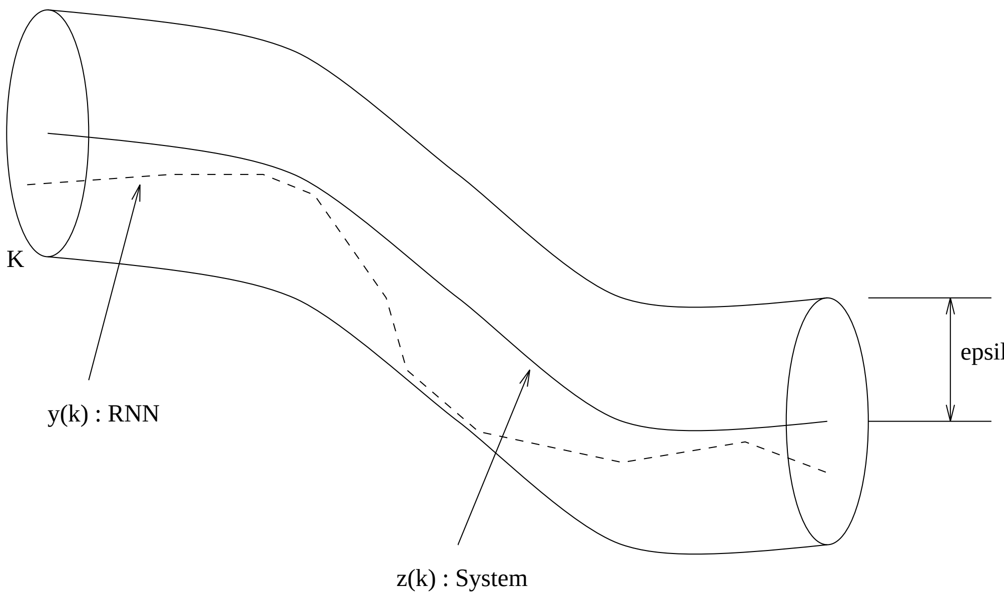


Figure 9.1: A graphical interpretation of the meaning of Theorem 3 and Theorem 4.

The discrete-time trajectory approximation is recently proven by Jin *et al.* [40].

Theorem 6 (Jin-Nikiforuk-Gupta Theorem[40]) *Let D be an open subset of S, and $f : S \times R^n \to R^n$ be a continuous vector-valued function which defines the following non-autonomous non-linear system*

$$z(k+1) = f(z(k), u(k)), \;\; z \in R^n, \;\; u \in S, \tag{9.4}$$

with an initial state $z(0) \in K$. Then, for any arbitrary number $\epsilon > 0$ and an integer $0 < I < +\infty$, there exist an integer N and a recurrent neural network of the form

$$\begin{aligned} x(k+1) &= -\alpha x(k) + A\sigma(x(k) + Bu(k)), & (9.5)\\ y(k) &= Cx(k), & (9.6) \end{aligned}$$

where $x \in R^N$ and $y \in R^n$, with an appropriate initial state $x(0)$ such that

$$\max_{0 \le k \le I} \|z(k) - y(k)\| < \epsilon. \tag{9.7}$$

□□□

The basic achievement of the above theorems can be shown graphically in Figure 9.1.

Theorem 1 concludes that the Jordan model is an universal approximator. Theorem 2 implies that the recurrent RBF net is also a universal approximator.

9.2 Realization property of Elman Model

The approximation property of the recurrent RBF net can be implied from Theorem 2. Suppose the matrix A in equation (9.5) is operated before the non-linear function σ, it can be shown that the universal approximation property still holds. The idea of proof is based on Theorem 1.

Theorem 7 *Let D be an open subset of S, and $f : S \times R^n \to R^n$ be a continuous vector-valued function which defines the following non-autonomous non-linear system*

$$\begin{aligned} x(k+1) &= f(x(k), u(k)), & (9.8)\\ y(k) &= Cx(k). & (9.9) \end{aligned}$$

Then, for an arbitrary number $\epsilon > 0$ and an integer $0 < I < +\infty$, there exist an integer N and a recurrent neural network of the form

$$\begin{aligned} x(k+1) &= \sigma(Ax(k) + Bu(k)), & (9.10)\\ y(k) &= Dx(k), & (9.11) \end{aligned}$$

where $x \in R^N$ and $y \in R^n$, with an appropriate initial state $x(0)$ such that

$$\max_{0 \le k \le I} \|z(k) - y(k)\| < \epsilon. \tag{9.12}$$

(*Proof*) In accordance with Theorem 1, there exists a multilayer perceptron with finite number of hidden nodes such that

$$|f(x(t), u(t)) - U\sigma(V_1 x(t) + V_2 u(t))| < \epsilon \tag{9.13}$$

holds, where $\sigma : R^n \to R^n$ is a sigmoid mapping. The matrix V_1, V_2 corresponds to the hidden to output connection matrix and input to hidden connection matrix. The corresponding structure is shown in Figure 9.2a. As $x(t+1)$ in the output will feedback to the input for next iteration, the structure can be redrawn as Figure 9.2b. Suppose we let $\hat{x}(t)$ be the output of the hidden layer, we can establish a relation for $\hat{x}(t+1)$ and $\hat{x}(t)$ as :

$$\hat{x}(t+1) = \sigma(UV_1\hat{x}(t) + V_2 u(t)), \tag{9.14}$$

and the output of the network being

$$y(t) = CU\hat{x}(t). \tag{9.15}$$

As the existence of U, V_1 and V_2 is guaranteed, $A = UV_1$, $B = V_2$, and $D = CU$ exist. This implies that every non-linear state-space model in the form of (9.8) and (9.9) can be modeled by an Elman network. And the proof is completed.

□□□

It should be noted that there are certain differences between our result and those of the others. *(i)* In Jin *et al.*'s work, the model considered is a local recurrent model similar to that proposed in [117] where the feedback is before the non-linearity, while the model proven in the last theorem is with the feedback after the non-linearity. *(ii)* In the Funahashi-Nakamura Theorem and Jin*et al.* Theorem, the output of the recurrent neural network is used for the modeling of the *state* of the state-space model. In our work, the output of

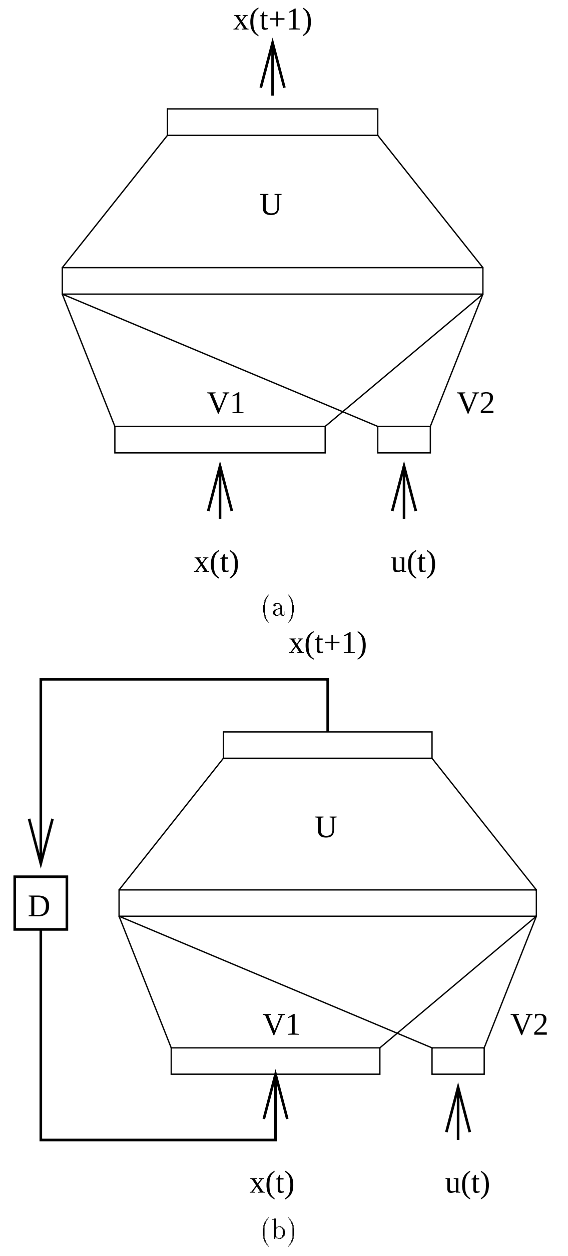


Figure 9.2: The universal approximation property.

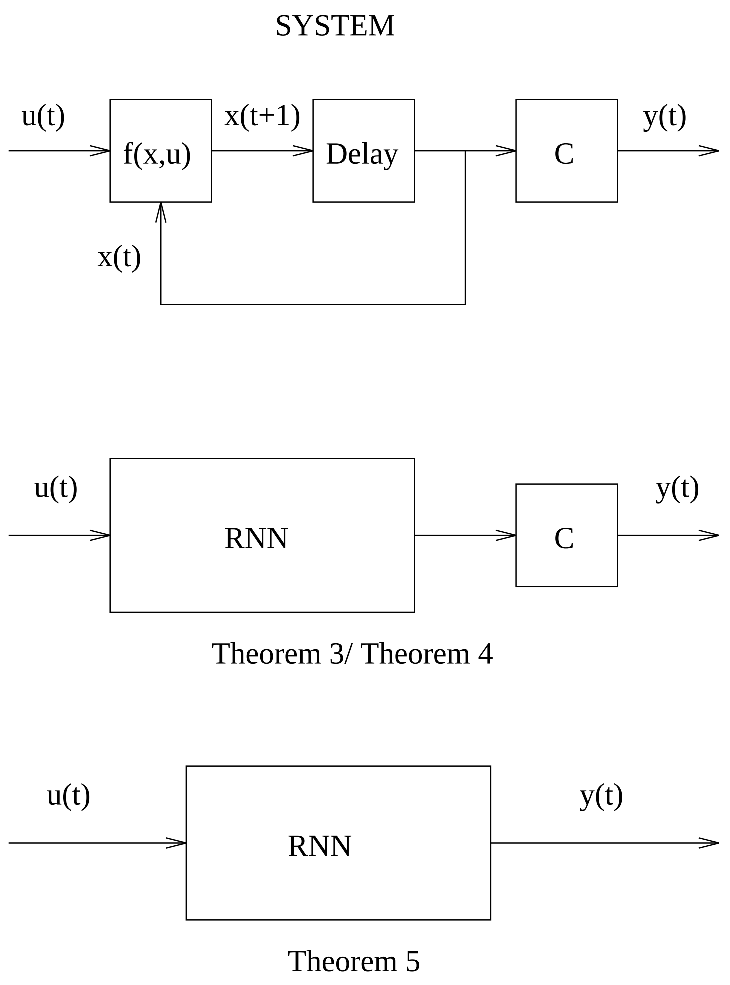


Figure 9.3: Comparison of the difference between Theorem 3, 4 and 5.

the recurrent neural network is for the identification of the output of the state model, (see Figure 9.3). This property is extremely important. Since we in general have no idea about the dimension of the state of the non-linear system being identified, even if we can have information on the dimension, it may not be possible to measure such state information. In practice, we can generally have the series of input-output data. Therefore, Theorem 3 and 4 are not sufficient to support such a situation. *(iii)* The Funahashi-Nakamura Theorem[24] is for continuous non-linear autonomous systems while Theorem 5 is for discrete-time non-linear non-autonomous systems. For clarity, Table 9.1 summarizes the above theorems and their differences.

It should be remarked that the same result can be readily shown if the non-linear function σ in Equation (9.10) is replaced by radial basis function, $\eta : R^n \to R^n$. The proof is based on the result in Theorem 2.

Thm	Model
1	MLP with g defined as sigmoidal Jordan model
2	MLP with g defined as radial function Recurrent RBF network
3	Continuous-time RNN
4	$x(k+1) = -\alpha x(k) + A\sigma(x(k) + Bu(k))$
5	$x(k+1) = \sigma(Ax(k) + Bu(k))$ Elman Model

Table 9.1: Summary of the significance of the theorems.

9.3 Summary

In this chapter, we have reviewed some of the recent results concerning the approximation property of multilayer perceptrons and recurrent networks. Following the theorem stated in [24] and [26], we are able to argue that *the Jordan model and recurrent RBF network are universal approximators.* Furthermore, we are able to show that *the discrete-time recurrent network model proposed by Elman can approximate any discrete-time state space system.* These results provide a theoretical foundation for those who apply the Elman network in system identification and time series prediction. Using similar techniques, it is readily shown that if the hidden unit nonlinearity of the Elman network is replaced by a radial basis function, the recurrent network is still able to realize any non-linear state-space system.

Chapter 10

Regularizability of FRLS

Due to its fast convergence rate and its adaptive behavior, the forgetting recursive least square (FRLS) method has recently been applied widely in the training of feedforward neural network. As in many applications, such as system identification and time series prediction, a batch of training data usually cannot be obtained in advance. Therefore, conventional batch mode training techniques such as backpropagation, the Newton method and other nonlinear programming techniques, cannot be easily applied. Thus, the use of the FRLS method or other adaptive training methods become inevitable. With the increasing popularity of using FRLS in neural network learning [12] [15] [48] [56] [57] [95] and pruning[56] [57], it is interesting to further investigate other properties besides its adaptive behavior.

Regularization is one approach to facilitate a better generalization for the neural network training. There have been many articles focusing on the design of a regularizer [128], the use of regularization [42] [67] and the effect of regularization in model complexity [75] [76] [79]. In general, regularization is a method which aims at reducing the model complexity [42], [53], [67], [74], [75], [76] and [79]. In the conventional batch mode training approach, regularization is usually realized by adding an extra term or a penalty term to the training error function. Three commonly used definitions are weight decay term [74], Tikhonov regularizer [114, 8, 42] and smooth regularizer [128].

In this chapter, we will discuss the regularization behavior of the FRLS training method. This chapter is organized into eight sections. In section one, the FRLS training method will be introduced. The main result will briefly be presented in section two, and the relationship between FRLS and weight decay in section three. We derive, from the very first principle, two equations describing the expected mean training error and the expected mean prediction error. The former one will be derived in section four and the latter one will be derived in section five. The derivation of the main result will thus be presented in section six. In section seven, by comparing these with the error equations obtained for recursive least square, we show that, under certain conditions, the model complexity and the expected prediction error of a model being trained by FRLS could both be smaller than that gained when the model is trained using RLS method. Finally, we conclude the chapter in section eight.

10.1 Preliminary

The model being discussed in this chapter is the generalized linear model defined as follows :

$$y(x) = \varphi^T(x)\theta_0 + \epsilon, \tag{10.1}$$

where $y, \epsilon \in R$, $\theta_0, \varphi(x) \in R^n$ and ϵ is a mean zero Gaussian noise. $\varphi(x)$ is a nonlinear vector function depending on the input $x \in R^m$. θ_0 is assumed to be the true model parameter.

In neural network literature, model (10.1) represents many types of neural network model. One example is the radial basis function network [28] if the i^{th} element of $\varphi(x)$, $\varphi_i(x)$, is defined as

$$\exp\left(-\frac{1}{2}(x - m_i)^T\Sigma_i(x - m_i)\right),$$

where $\Sigma_i \in R^{m\times m}$ is a fixed positive definite matrix and $m_i \in R^m$ is a fixed m-vector. θ_0 would then be the output weight vector. In nonlinear system modeling, model (10.1) can also represent a Volterra series [43]. Suppose that $x = (x_1, x_2, \ldots, x_m)^T$ and $\varphi(x)$ is a 2^m-vector which consists 1, x_i (for all $i = \{1, \ldots, m\}$), x_ix_j (for all $i, j = \{i, \ldots, m\}$), $x_ix_jx_k$ (for all $i, j, k = \{i, \ldots, m\}$,..., $x_1x_2\cdots x_m$, as the elements.

Considering model (10.1), we usually define the estimator as follows :

$$\hat{y}(x) = \varphi^T(x)\hat{\theta}, \tag{10.2}$$

where $\hat{\theta}$ is the estimate of the true parameter θ_0.

By feeding the training data one by one, the estimate $\hat{\theta}$ can be updated iteratively based on the forgetting recursive least square method [33]. Let $\hat{\theta}(t)$ be the optimal estimation of θ_0 when t data have been fed, the training can be accomplished via the following recursive equations :

$$\begin{aligned}
S(t) &= \varphi^T(x_t)P(t-1)\varphi(x_t) + (1-\alpha) && (10.3)\\
L(t) &= P(t-1)\varphi(x_t)S^{-1}(t) && (10.4)\\
P(t) &= (I_{n\times n} - L(t)\varphi(x_t))\frac{P(t-1)}{1-\alpha} && (10.5)\\
\hat{\theta}(t) &= \hat{\theta}(t-1) + L(t)(y(x_t) - \varphi^T(x_t)\hat{\theta}(t-1))), && (10.6)
\end{aligned}$$

with the initial conditions :

$$\begin{aligned}
\hat{\theta}(0) &= 0 && (10.7)\\
P(0) &= \delta^{-1}I_{n\times n}, && (10.8)
\end{aligned}$$

and α is the forgetting factor in between zero and one.

In the theory of system identification [43], the objective of the above recursive algorithm is to minimize the cost function $J(\theta(t))$, where

$$J(\theta(t)) = \sum_{k=1}^{t} w_k(y(x_k) - \varphi^T(x_k)\hat{\theta}(t))^2 + \delta\|\theta(t)\|^2, \tag{10.9}$$

where $\{\varphi(x_k), y(x_k)\}_{k=1}^t$ is the training data set and $w_k = (1-\alpha)^{t-k}$. Note that $1 \geq w_i > w_j \geq 0$, for all $1 \leq i < j \leq t$. These weighting factors will lead to capturing the information obtained from the most recent training data more than the earlier training data. For $k = t$, the weighting on $(y(x_t) - \varphi^T(x_t)\hat{\theta}(t))$ is one. When $k = t-1$, the weighting on $(y(x_{t-1}) - \varphi^T(x_{t-1})\hat{\theta}(t))$ is $(1-\alpha)$. This factor is smaller than one. As a result, the factor w_k serves as a weighting factor which counts the information obtained from the most recent training data more than the earlier training data.

10.2 The Main Result

A criterion for measuring the performance of (10.2) is the mean prediction error [5], that is the accuracy of the model in predicting the output of an unseen data x^F :

$$MPE(t) = \int_{\Omega_\epsilon}\int_{\Omega_x} (y(x^F) - \varphi^T(x^F)\hat{\theta}(t))^2 p(x^F)p(\epsilon)dx^F d\epsilon, \tag{10.10}$$

where $p(x^F)$ and $p(\epsilon)$ are the probability density functions of x^F and ϵ respectively. This $MPE(t)$ depends on the estimator $\hat{\theta}(t)$ and hence it is a random variable dependent on the training set, $\{\varphi(x_k), y(x_k)\}_{k=1}^t$. Therefore, another criterion would be the expected mean prediction error [75, 79, 98] which is defined as follows :

$$\langle MPE(t)\rangle_{\xi_t} = \left\langle \int_{\Omega_\epsilon}\int_{\Omega_x} (y(x^F) - \varphi^T(x^F)\hat{\theta}(t))^2 p(x^F)p(\epsilon)dx^F d\epsilon \right\rangle_{\xi_t}. \tag{10.11}$$

$\langle . \rangle_{\xi_T}$ denotes the expectation over the training set, $\xi_t = \{\varphi(x_k), \epsilon_t\}_{k=1}^t$.

Assuming that t is large enough and δ is very small, by using a similar technique to that depicted in papers [5], [53], [75], [79] and [98], we can derive that

$$\langle MPE(t)\rangle_{\xi_t} \approx \lambda_0 \left[1 + \frac{2}{t}\sum_{k=1}^n \left(\frac{\hat{\delta}_k}{\hat{\delta}_k + \alpha\delta}\right)^2\right], \tag{10.12}$$

where λ_0 is the variance of the output noise ϵ_t and $\hat{\delta}_k$ is the k^{th} eigenvalue of the matrix

$$H = \frac{1}{t}\sum_{k=1}^t \varphi(x_k)\varphi^T(x_k)$$

and

$$\lim_{t\to\infty} H = \langle \varphi(x)\varphi^T(x)\rangle_{\Omega_x}.$$

Besides, if we define the mean training error as follows :

$$\langle MTE(t)\rangle_{\xi_t} = \left\langle \frac{1}{t}\sum_{k=1}^t (y(x_k) - \varphi^T(x_k)\hat{\theta}(t))^2 \right\rangle_{\xi_t}, \tag{10.13}$$

we could further relate the prediction error and the training error by the following equation :

$$\langle MPE(t)\rangle_{\xi_t} \approx \langle MTE(t)\rangle_{\xi_t} + 2\frac{\lambda_0}{t}\sum_{k=1}^n \frac{\hat{\delta}_k}{\hat{\delta}_k + \alpha\delta}. \tag{10.14}$$

The derivation of Equation (10.14) will be shown in the following sections.

10.3 FRLS and weight decay

Comparing (10.14) to that obtained from weight decay [75], it will be realized that the FRLS training method has an effect similar to weight decay training. This result is extremely useful. The reason can be explained as below.

In weight decay, the cost function is defined as follows :

$$J_{WD}(\theta) = \frac{1}{N}\sum_{k=1}^{N}(y(x_k) - \varphi^T(x_k)\theta)^2 + c_0\|\theta\|^2, \tag{10.15}$$

where c_0 is the regularization factor controlling the penalty due to large weight. The estimate $\hat{\theta}$ is the one which maximizes $J_{WD}(\theta)$, that is

$$\hat{\theta} = \arg\max_{\theta}\left\{\frac{1}{N}\sum_{k=1}^{N}(y(x_k) - \varphi^T(x_k)\theta)^2 + c_0\|\theta\|^2\right\}.$$

By comparing Equation (10.15) with the objective function of recursive least square,

$$\frac{1}{N}\sum_{k=1}^{N}(y(x_k) - \varphi^T(x_k)\hat{\theta}(N))^2 + c_0\|\hat{\theta}(N)\|^2,$$

one can readily apply RLS method by setting $\theta(0)$ to be zero vector and $P(0) = (c_0N)^{-1}I_{n\times n}$. However, in on-line mode training, we usually do not know what N exactly is. We just know that training data will come one after the other. In such a case, a simple recursive algorithm cannot be easily derived.

Therefore, based on the finding that the FRLS training method is asymptotically identical to weight decay training, we can now have *an elegant on-line training method which can accomplish the same effect as weight decay* if $\alpha\delta = c_0$.

10.4 Derivation of the Expected Mean Training Error

In accordance with the theory of identification, the objective of FRLS is to minimize the cost function defined as (10.9) :

$$J(\hat{\theta}(t)) = \sum_{k=1}^{t}(1-\alpha)^{t-k}(y(x_k) - \varphi^T(x_k)\hat{\theta}(t))^2 + \delta\|\hat{\theta(t)}\|^2,$$

where $\{\varphi(x_k), y(x_k)\}_{k=1}^{t}$ is the training data set. Differentiating (10.9) once with respect to $\hat{\theta}(t)$ and equating it to zero, we can derive the solution of $\hat{\theta}(t)$:

$$\hat{\theta}(t) = \left[\sum_{k=1}^{t}(1-\alpha)^{t-k}\varphi(x_k)\varphi^T(x_k) + \delta I\right]^{-1}\left[\sum_{k=1}^{t}(1-\alpha)^{t-k}\varphi(x_k)y(x_k)\right]. \tag{10.16}$$

Replacing $y(x_k)$ by its definition, Equation (10.1), and using Equation (10.16), it can be shown that

$$\begin{aligned} y(x_k) - \varphi^T(x_k)\hat{\theta}(t) &= \epsilon_k + \delta\varphi^T(x_k)G_1^{-1}\theta_0 \\ &\quad -\varphi^T(x_k)G_1^{-1}\left[\sum_{k=1}^{t}(1-\alpha)^{t-k}\varphi(x_k)\epsilon_k\right], \end{aligned} \tag{10.17}$$

where

$$G_1 = \sum_{l=1}^{t}(1-\alpha)^{t-l}\varphi(x_l)\varphi^T(x_l) + \delta I. \tag{10.18}$$

Note that for $k = 1, \ldots, t$, ϵ_k is a zero mean Gaussian noise with variance λ_0 for all $k = 1, 2, \ldots, t$. By squaring Equation (10.17), summing up for k from 1 to t and taking the expectation over the set ξ_t, we can thus obtain an equation for the expected training error. Assuming that t is large enough,

$$G_1 \approx \frac{1}{\alpha}\langle\varphi(x)\varphi^T(x)\rangle_{\Omega_x} + \delta I. \tag{10.19}$$

$$\begin{aligned}\left\langle\sum_{k=1}^{t}(y(x_k) - \varphi^T(x_k)\hat{\theta}(t))^2\right\rangle_{\xi_T} &= t\lambda_0 + \delta_0^2\sum_{k=1}^{t}\varphi^T(x_k)G_1^{-1}\theta_0\theta_0^TG_1^{-1}\varphi(x_k) \\ &+\lambda_0\sum_{k=1}^{t}\varphi^T(x_k)G_1^{-1}H_2G_1^{-1}\varphi(x_k) \\ &-2\lambda_0\sum_{k=1}^{t}(1-\alpha)^{t-k}\varphi^t(x_k)G_1^{-1}\varphi(x_k) \qquad (10.20) \\ &\approx t\lambda_0 + \delta^2 tr\{HG_1^{-1}\theta_0\theta_0^TG_1^{-1}\} \\ &-2\lambda_0 tr\{H_1G_1^{-1}\} + \lambda_0 tr\{HG_1^{-1}H_2G_1^{-1}\}, (10.21)\end{aligned}$$

where tr is the trace operator,

$$\begin{aligned}H_1 &= \sum_{k=1}^{t}(1-\alpha)^{t-k}\varphi(x_k)\varphi^T(x_k) \\ &\approx \frac{1}{\alpha}\langle\varphi(x)\varphi^T(x)\rangle_{\Omega_x}. \qquad (10.22)\end{aligned}$$

$$\begin{aligned}H_2 &= \sum_{k=1}^{t}(1-\alpha)^{2(t-k)}\varphi(x_k)\varphi^T(x_k) \qquad (10.23) \\ &\approx \frac{1}{1-(1-\alpha)^2}\langle\varphi(x)\varphi^T(x)\rangle_{\Omega_x} \qquad (10.24)\end{aligned}$$

Therefore, the expected mean training error can be rewritten as follows :

$$\begin{aligned}\langle MTE(t)\rangle_{\xi_T} &= \frac{1}{N}\left\langle\sum_{k=1}^{t}(y(x_k) - \varphi^T(x_k)\hat{\theta}(t))^2\right\rangle_{\xi_T} \qquad (10.25) \\ &= \lambda_0 + \frac{\lambda_0}{N}\left(tr\{HG_1^{-1}H_2G_1^{-1}\} - 2tr\{H_1G_1^{-1}\}\right) \\ &+\frac{\delta^2}{N}tr\{HG_1^{-1}\theta_0\theta_0^TG_1^{-1}\}. \qquad (10.26)\end{aligned}$$

10.5 Derivation of the Expected Mean Prediction Error

Next, we are going to derive the equation for the expected mean prediction error defined in Equation (10.11). First, let us derive an equation for $\theta_0 - \hat{\theta}(t)$. Using the result in Equation (10.16) once again, we can readily show that

$$\begin{aligned}\theta_0 - \hat{\theta}(t) &= \theta_0 - G_1^{-1}\left[\sum_{k=1}^{t}(1-\alpha)^{t-k} y(x_k)\varphi(x_k)\right] \\ &= \delta G_1^{-1}\theta_0 - G_1^{-1}\sum_{k=1}^{t}(1-\alpha)^{t-k}\epsilon_k \varphi(x_k) \qquad (10.27)\end{aligned}$$

and hence

$$\langle(\theta_0 - \hat{\theta}(t))(\theta_0 - \hat{\theta}(t))^T\rangle_{\xi_t} = \delta^2 G_1^{-1}\theta_0\theta_0^T G_1^{-1} + \lambda_0 G_1^{-1} H_2 G_1^{-1}. \qquad (10.28)$$

Recall that the definition of the expected mean prediction error is as follows :

$$\langle MPE(t)\rangle_{\xi_t} = \left\langle \int_{\Omega_\epsilon}\int_{\Omega_x} (y(x^F) - \varphi^T(x^F)\hat{\theta}(t))^2 p(x^F)p(\epsilon)dx^F d\epsilon \right\rangle_{\xi_t}.$$

Since $\hat{\theta}(t)$ is a random variable independent of x and ϵ, Equation (10.11) can be rewritten as follows :

$$\langle MPE(t)\rangle_{\xi_t} = \lambda_0 + tr\left\{\int_{\Omega_x}\varphi(x^F)\varphi^T(x^F)p(x^F)dx^F \langle(\theta_0 - \hat{\theta}(t))(\theta_0 - \hat{\theta}(t))^T\rangle_{\xi_t}\right\}. \qquad (10.29)$$

Suppose that t is large enough, we approximate $\int_{\Omega_x}\varphi(x)\varphi^T(x)p(x)dx$ by $t^{-1}H$. By using (10.28), we can show that

$$\langle MPE(t)\rangle_{\xi_t} \approx \lambda_0 + \frac{\delta^2}{t} tr\left\{\delta^2 G_1^{-1}\theta_0\theta_0^T G_1^{-1}\right\} + \frac{\lambda_0}{t} tr\left\{G_1^{-1} H_2 G_1^{-1}\right\}. \qquad (10.30)$$

10.6 Derivation of Equation for MPE and MTE

Comparing Equation (10.30) and (10.26), it can be shown that

$$\langle MPE(t)\rangle_{\xi_T} \approx \langle MTE(t)\rangle_{\xi_T} + \frac{2\lambda_0}{t} tr\{H_1 G_1^{-1}\}. \qquad (10.31)$$

As when t is large,

$$\begin{aligned} H_1 &= \sum_{k=1}^{t}(1-\alpha)^{t-k}\varphi(x_k)\varphi^T(x_k) \qquad (10.32)\\ &\approx \frac{1}{\alpha t} H \qquad (10.33)\\ &\approx \frac{1}{\alpha t}\int_{\Omega_x}\varphi(x)\varphi^T(x)p(x)dx, \qquad (10.34)\end{aligned}$$

and

$$\begin{aligned} G_1 &\approx \frac{1}{\alpha t} H + \delta I & (10.35) \\ &\approx \frac{1}{\alpha t} \int_{\Omega_x} \varphi(x)\varphi^T(x)p(x)dx + \delta I. & (10.36) \end{aligned}$$

Using the asymptotic approximations, Equation (10.33) and Equation (10.35), for H_1 and G_1, we could get that

$$tr\{H_1 G_1^{-1}\} \approx tr\left\{\frac{1}{t} H \left[\frac{1}{t} H + \alpha\delta I\right]^{-1}\right\}. \quad (10.37)$$

Let $\hat{\delta}_k$ be an estimate of the k^{th} eigenvalue of the matrix $\int_{\Omega_x} \varphi(x)\varphi^T(x)p(x)dx$,

$$\langle MPE(t)\rangle_{\xi_T} \approx \langle MTE(t)\rangle_{\xi_T} + 2\frac{\lambda_0}{t}\sum_{k=1}^{n}\frac{\hat{\delta}_k}{\hat{\delta}_k + \alpha\delta}. \quad (10.38)$$

10.7 Comparison with recursive least square

Once the factor α is zero, it should be noted that the algorithms (10.3)-(10.6) can be reduced to the standard recursive least square (RLS) method. Using a similar technique, Equations (10.18), (10.22) and (10.24), the following equalities will be obtained.

$$\begin{aligned} G_1(\alpha = 1) &= G. & (10.39) \\ H_1(\alpha = 1) &= H. & (10.40) \\ H_2(\alpha = 1) &= H. & (10.41) \end{aligned}$$

Then the mean prediction error and the mean training error for RLS method can readily be derived.

$$\langle MPE(t)\rangle_{\xi_T} \approx \lambda_0 + \frac{\delta^2}{t} tr\left\{\delta^2 G^{-1}\theta_0\theta_0^T G^{-1}\right\} + \frac{\lambda_0}{t} tr\left\{G^{-1}HG^{-1}\right\}. \quad (10.42)$$

$$\langle MTE(t)\rangle_{\xi_T} = \lambda_0 + \frac{\lambda_0}{t}\left(tr\{HG^{-1}HG^{-1}\} - 2tr\{HG^{-1}\}\right) + \frac{\delta^2}{t} tr\{HG^{-1}\theta_0\theta_0^T G^{-1}\}. \quad (10.43)$$

In such a case the difference between the expected mean prediction error and the expected mean training error will be equal to $2\frac{\lambda_0}{t} tr\{HG^{-1}\}$, i.e.

$$\begin{aligned} \langle MPE(t)\rangle_{\xi_T} &\approx \langle MTE(t)\rangle_{\xi_T} + 2\frac{\lambda_0}{t} tr\{HG^{-1}\} \\ &= \langle MTE(t)\rangle_{\xi_T} + 2\frac{\lambda_0}{t}\sum_{k=1}^{n}\frac{\hat{\delta}_k}{\hat{\delta}_k + \delta/t}. & (10.44) \end{aligned}$$

Suppose t is very large, the second term in Equation (10.44) would be equal to $2\lambda_0 n/t$.

If we define the network complexity as the effective number of parameters, Equation (10.38) and Equation (10.42) reveal that the complexity of the models being trained by using FRLS is usually smaller than that by using RLS.

Apart from the difference in the model complexity, we can also show that under certain conditions, the expected mean prediction error generated by the network being trained by FRLS is smaller than that by using RLS. Again, we consider the asymptotic situation. We let $\langle\varphi(x)\varphi^T(x)\rangle$ be $\int_{\Omega_x}\varphi(x)\varphi^T(x)p(x)dx$. The following approximations can readily be obtained.

$$H \approx \frac{1}{t}\langle\varphi(x)\varphi^T(x)\rangle \tag{10.45}$$

$$H_1 \approx \frac{1}{\alpha}\langle\varphi(x)\varphi^T(x)\rangle \tag{10.46}$$

$$H_2 \approx \frac{1}{2\alpha-\alpha^2}\langle\varphi(x)\varphi^T(x)\rangle \tag{10.47}$$

$$G_1 \approx \frac{1}{\alpha}\langle\varphi(x)\varphi^T(x)\rangle + \delta I. \tag{10.48}$$

Using these approximated equations and considering the factors $G_1^{-1}H_2G_1^{-1}$ and $G^{-1}HG^{-1}$ in the Equations (10.30) and (10.42), one can show that:

$$G_1^{-1}H_2G_1^{-1} \approx \left[\frac{1}{\alpha}\langle\varphi\varphi^T\rangle+\delta I\right]^{-1}\left[\frac{1}{2\alpha-\alpha^2}\langle\varphi\varphi^T\rangle\right]\left[\frac{1}{\alpha}\langle\varphi\varphi^T\rangle+\delta I\right]^{-1} \tag{10.49}$$

$$G^{-1}HG^{-1} \approx \left[t\langle\varphi\varphi^T\rangle+\delta I\right]^{-1}\left[t\langle\varphi\varphi^T\rangle\right]\left[t\langle\varphi\varphi^T\rangle+\delta I\right]^{-1}, \tag{10.50}$$

and *if*

$$\frac{1}{\alpha} > t > \frac{1}{2\alpha} \tag{10.51}$$

or equivalently

$$\frac{1}{t} > \alpha > \frac{1}{2t}, \tag{10.52}$$

the expected mean prediction error of using FRLS will be smaller than that of using RLS.

10.8 Summary

In this chapter, we have presented certain analytical results regarding the use of the forgetting recursive least square method in the training of a linear neural network. The expected mean prediction error and the expected mean training error are derived from the first principle with the assumptions that the number of training data is large and the output noise ϵ is a zero mean Gaussian noise. Using these error equations, we are able to analyze and compare the behavior of FRLS with RLS. First, we have shown that *FRLS has an inherent weight decay (regularization) effect.* In RLS training, this effect is not persistent. It will fade out as the number of training data is increasing [56][57]. Second, we have shown that *the expected mean prediction error of using FRLS can be smaller than that of using RLS if the forgetting factor α is set appropriately.*

Chapter 11

Equivalence of NARX and RNN

Nonlinear Autoregressive models with exogenous input (NARX model) and recurrent neural networks (RNN) are two models commonly used in system identification, time series prediction and system control. Formally, a NARX model [10, 11, 62, 82, 96], is defined as follows :

$$y(t) = g(y(t-1), \ldots, y(t-n_y), u(t), \ldots, u(t-n_u)), \tag{11.1}$$

where $u(t)$ and $y(t)$ correspond to the input and output of the network at time t; n_y and n_u are the input order and output order respectively. A simple example is illustrated in Figure 11.1 with $n_y = 1$ and $n_u = 0$. This model is expressed as follows :

$$y(t) = \sum_{i=1}^{3} \alpha_i \tanh(\beta_i y(t-1) + \gamma_i u(t) + \delta_i),$$

where

$$\tanh(x) = \frac{e^x - e^x}{e^x + e^x}.$$

Parameters β_i and γ_i are usually called the input weight and the parameter δ_i is called bias. The parameter α_i is called output weight. This model is basically a multilayer perceptron [94] except that the output is feedback to the input.

On the other hand, a RNN [39, 40, 41, 125, 90, 101, 123] is defined in a state-space form :

$$\begin{aligned} \vec{s}(t) &= g(\vec{s}(t-1), u(t), u(t-1), \ldots, u(t-n_u)) & (11.2) \\ y(t) &= c^T \vec{s}(t), & (11.3) \end{aligned}$$

where $\vec{s}(t)$ is the output of the hidden units at time t and c is the output weight vector. A simple example which consists of three hidden units is illustrated in Figure 11.2. This model is expressed as follows :

$$s_1(t) = \tanh\left(\sum_{k=1}^{3} \tilde{\beta}_{1k} s_k(t) + \tilde{\gamma}_1 u(t) + \tilde{\delta}_1\right), \tag{11.4}$$

$$s_2(t) = \tanh\left(\sum_{k=1}^{3} \tilde{\beta}_{2k} s_k(t) + \tilde{\gamma}_2 u(t) + \tilde{\delta}_2\right), \tag{11.5}$$

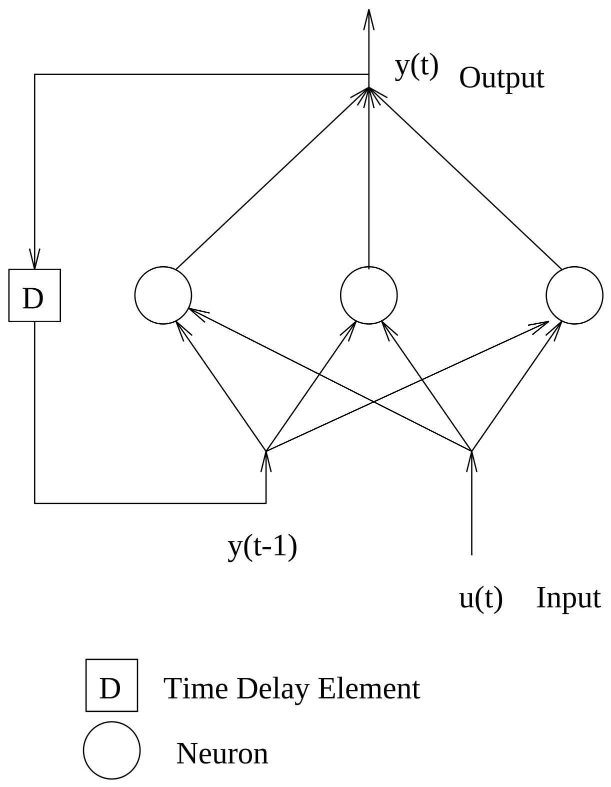


Figure 11.1: A NARX model

$$
\begin{aligned}
s_3(t) &= \tanh\left(\sum_{k=1}^{3} \tilde{\beta}_{3k} s_k(t) + \tilde{\gamma}_3 u(t) + \tilde{\delta}_3\right), && (11.6)\\
y(t) &= \sum_{k=1}^{3} \tilde{\alpha}_k s_k(t). && (11.7)
\end{aligned}
$$

$$\vec{s}(t) = (s_1(t), s_2(t), s_3(t))^T$$

and

$$c = (\tilde{\alpha}_1, \tilde{\alpha}_2, \tilde{\alpha}_3)^T.$$

Parameters $\tilde{\gamma}_i$, $\tilde{\beta}_{ij}$, $\tilde{\delta}_i$ and $\tilde{\alpha}_i$ are called the input weight, the recurrent weight, the bias and the output weight respectively.

In contrast to the NARX model, RNN does not have feedback connections from the output to the input. The feedback connection exists only amongst the neurons in the hidden layer.

According to their structural difference, the NARX model and RNN are studied independently. Only a few papers have presented results concerning their similarity [19, 84]. Olurotimi [84] has recently showed that every RNN can be transformed into a NARX model. Thus he derived an algorithm for RNN training with feedforward complexity.

Inspired by Olurotimi's work, in the rest of the chapter, we would like to present some other aspects regarding the equivalence between NARX and RNN. Section two presents

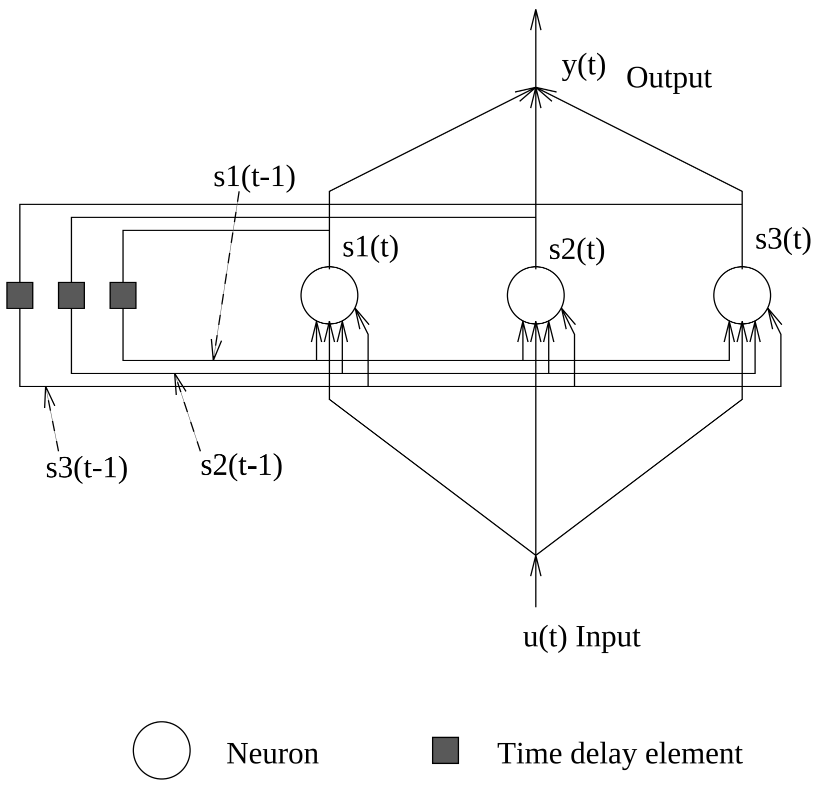


Figure 11.2: Recurrent neural network model.

the major result, the model equivalence between NARX and RNN. Three issues concerning the use of these equivalence results are studied in section three. Finally, we conclude the chapter in section four.

11.1 Model Equivalence

Assuming that the system being identified is deterministic and given by

$$x(t+1) = g(x(t), u(t+1)) \tag{11.8}$$

$$y(t+1) = cx(t+1), \tag{11.9}$$

where $x(t), y(t)$ is the system input and output, due to the universal approximation property of a feedforward neural network [21, 100, 101, 107], the nonlinear function g thus can be approximated by a feedforward neural network. Hence, the above system can be rewritten as follows :

$$x(t+1) = \sum_{i=1}^{n} d_i \tanh(a_i x(t) + b_i u(t+1) + e_i) \tag{11.10}$$

$$y(t+1) = cx(t+1), \tag{11.11}$$

where $\{a_i, b_i, d_i, e_i\}_{i=1}^{n}$ and c are the system parameters.

11.1.1 tanh neuron

If x, y, u are scalars

Obviously, system (11.10) and (11.11) is equivalent to a NARX model if we substitute $x(t)$ in Equation (11.10) by $c^{-1}y(t)$. That is,

$$y(t+1) = \sum_{i=1}^{n} \alpha_i \tanh(\beta_i y(t) + \gamma_i u(t+1) + \delta_i) \tag{11.12}$$

with $\alpha_i = cd_i$, $\beta_i = a_i c^{-1}$, $\gamma_i = b_i$ and $\delta_i = e_i$.

If, we let $s_i(t+1) = \tanh(a_i x(t) + b_i u(t+1) + e_i)$ for all $i = 1, \ldots, n$, Equation (11.10) and (11.11) can be rewritten as

$$s_i(t+1) = \tanh\left(\sum_{k=1}^{n} \tilde{\beta}_{ik} s_k(t) + \tilde{\gamma}_i u(t+1) + \tilde{\delta}_i\right), \tag{11.13}$$

$$y(t) = \sum_{k=1}^{n} \tilde{\alpha}_k s_k(t), \tag{11.14}$$

where $\tilde{\beta}_{ik} = a_i d_k$, $\tilde{\gamma}_i = b_i$, $\tilde{\delta}_i = e_i$ and $\tilde{\alpha}_i = cd_i$.

By comparing the coefficients amongst (11.10), (11.11), (11.12), (11.13) and (11.14), we can define the following transformations :

$$\begin{aligned} \tilde{\beta}_{ik} &= \beta_i \alpha_k, & (11.15)\\ \tilde{\gamma}_i &= \gamma_i, & (11.16)\\ \tilde{\delta}_i &= \delta_i, & (11.17)\\ \tilde{\alpha}_i &= \alpha_i. & (11.18) \end{aligned}$$

Let $[\tilde{\beta}]$ be the matrix $(\tilde{\beta}_{ik})_{n\times n}$, the vector form is given by

$$\left[\tilde{\beta}\right] = \beta\alpha^T, \quad \tilde{\gamma} = \gamma, \quad \tilde{\delta} = \delta, \quad \tilde{\alpha} = \alpha. \tag{11.19}$$

This establishes a way to transform a NARX to a recurrent neural network. The inverse transformation of a RNN to a NARX model can be accomplished via the following equations :

$$\begin{aligned} \beta &= (\tilde{\alpha}^T\tilde{\alpha})^{-1}\left[\tilde{\beta}\right]\tilde{\alpha}, & (11.20)\\ \gamma &= \tilde{\gamma}, & (11.21)\\ \delta &= \tilde{\delta}, & (11.22)\\ \alpha &= \tilde{\alpha} & (11.23) \end{aligned}$$

as long as α (or $\tilde{\alpha}$) is non-zero vector.

If x, u, y are vectors

If $u \in R^M$, $x, y \in R^N$, the vector NARX model is given by

$$\vec{y}(t+1) = \sum_{i=1}^{n} W_1 \tanh(W_2\, \vec{y}(t) + W_3\, \vec{u}(t+1) + W_4), \tag{11.24}$$

where $W_1 \in R^{N\times n}$, $W_2 \in R^{n\times N}$, $W_3 \in R^{n\times M}$ and $W_4 \in R^n$. Similarly, an equivalent transformation can be established for RNN

$$\begin{aligned} \vec{s}(t+1) &= \tanh(\tilde{W}_2\, \vec{s}(t) + \tilde{W}_3\, \vec{u}(t+1) + \tilde{W}_4) \qquad (11.25)\\ \vec{y}(t+1) &= \tilde{W}_1\, \vec{s}(t+1) \qquad (11.26) \end{aligned}$$

via the following equations :

$$\begin{aligned} \tilde{W}_1 &= W_1, \qquad (11.27)\\ \tilde{W}_2 &= W_2 W_1, \qquad (11.28)\\ \tilde{W}_3 &= W_3, \qquad (11.29)\\ \tilde{W}_4 &= W_4. \qquad (11.30) \end{aligned}$$

For getting back the NARX model, we can perform the inverse transformation defined as follows :

$$\begin{aligned} W_1 &= \tilde{W}_1 \qquad (11.31)\\ W_2 &= \tilde{W}_2 \tilde{W}_1^T (\tilde{W}_1 \tilde{W}_1^T)^{-1} \qquad (11.32)\\ W_3 &= \tilde{W}_3 \qquad (11.33)\\ W_4 &= \tilde{W}_4 \qquad (11.34) \end{aligned}$$

for $\tilde{W}_1$ is nonsingular.

11.1.2 Piece-wise linear neuron

Once the neuron's transfer function is piece-wise linear, i.e.

$$\vec{y}(t+1) = W_1 f(W_2 \vec{y}(t) + W_3 \vec{u}(t+1) + W_4), \tag{11.35}$$

where

$$f(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } -1 \le x \le 1 \\ -1 & \text{if } x < -1 \end{cases} \tag{11.36}$$

this result can be extended to higher order NARX models. Without loss of generality, we consider a second order NARX :

$$\vec{y}(t+1) = W_1 f(W_{20}\vec{y}(t) + W_{21}\vec{y}(t-1) + W_3\vec{u}(t+1) + W_4). \tag{11.37}$$

Now we define a state-vector $\vec{z}(t) = (\vec{s}(t)\ \vec{s}(t-1))^T$, where $\vec{s}(t+1) = f(W_{20}\vec{y}(t) + W_{21}\vec{y}(t-1) + W_3\vec{u}(t+1) + W_4)$. Since $s = f(s)$ if s is bounded by ± 1, we can rewrite the model as follows :

$$\begin{aligned} \vec{z}(t+1) &= f(\tilde{W}_2\vec{z}(t) + \tilde{W}_3\vec{u}(t+1) + \tilde{W}_4) && (11.38) \\ \vec{y}(t+1) &= \tilde{W}_1\vec{z}(t+1) && (11.39) \end{aligned}$$

where

$$\tilde{W}_1 = \begin{bmatrix} W_1 & O_{N\times n} \end{bmatrix}. \tag{11.40}$$

$$\tilde{W}_2 = \begin{bmatrix} W_{20}W_1 & W_{21}W_1 \\ I_{n\times n} & O_{n\times n} \end{bmatrix} \tag{11.41}$$

$$\tilde{W}_3 = \begin{bmatrix} W_3 \\ O_{n\times M} \end{bmatrix} \tag{11.42}$$

$$\tilde{W}_4 = \begin{bmatrix} W_4 \\ O_{n\times 1} \end{bmatrix}. \tag{11.43}$$

Note that $\tilde{W}_1 \in R^{N\times 2n}$, $\tilde{W}_2 \in R^{2n\times 2n}$, $\tilde{W}_3 \in R^{2n\times M}$ and $\tilde{W}_4 \in R^{2n\times 1}$.

11.2 Implications of the equivalence property

11.2.1 On training

One should note from the equation, $\tilde{W}_2 = W_2W_1$, that the number of parameters in RNN is fewer than NARX if $n < N$ (since $\tilde{W}_2 \in R^{n\times n}$ while $W_2 \in R^{n\times N}$), where n is the number of hidden units and N is the number of output units.

In most of the applications of NARX in dynamic system modeling, the output dimension is small. Suppose we ignore the computational complexity in each training step, and we purely look at the number of parameters being updated, training a NARX should be faster than training a RNN. It will be a doubtful if the dimension of $\vec{y}$ is larger than the number of hidden units. That is $n < N$.

Suppose a NARX is being trained by using the forgetting recursive least square[1] (FRLS) method. Let θ be the augmented vector including all the parameters $\{W_1, W_2, W_3, W_4\}$, $x_t = (\vec{y}^T(t-1), \vec{u}^T(t))^T$ and $\varphi(t) = \frac{\partial y(t)}{\partial \theta}$, the training can be accomplished via the following recursive equations.

$$\begin{aligned} S(t) &= \varphi^T(t)P(t-1)\varphi(t) + (1-\lambda)I_{N\times N} && (11.44) \\ L(t) &= P(t-1)\varphi(t)S^{-1}(t) && (11.45) \\ P(t) &= (I_{\dim\theta\times\dim\theta} - L(t)\varphi(x_t))\frac{P(t-1)}{1-\lambda} && (11.46) \\ \theta(t) &= \theta(t-1) + L(t)(y(t) - y(x_t, \theta(t-1))), && (11.47) \end{aligned}$$

[1]We pick up FRLS for discussion simply because it is a fast training method for a feedforward neural network [11, 56, 95].

with the initial conditions $\theta(0) = 0$ and $P(0) = \mu^{-1} I_{\dim\theta \times \dim\theta}$ and $0 < \lambda < 1$, and μ is a small positive number. $y(x_t, \theta(t-1))$ is the output of the NARX model at the t^{th} step. The computational burden is on Equation (11.45) and Equation (11.46) which requires $\mathcal{O}(\dim\theta^3)$ multiplication. Even though some decomposition on $P(t)$ can speed up the training [89, 90], the complexity is still of the same order if $N \gg n$.

Next, if the same dynamic system is identified by a RNN, the extended Kalman filter approach [104, 113, 123] is one fast method which can simultaneously estimate the state vector $\vec{s}(t)$ and identify the parametric vector $\hat{\theta}(t)$:

$$\begin{aligned} \hat{s}(t|t-1) &= g(\hat{s}(t-1|t-1), u(t), \hat{\theta}(t-1)) && (11.48)\\ P(t|t-1) &= F(t-1)P(t-1|t-1)F^T(t-1), && (11.49) \end{aligned}$$

$$\begin{aligned} \begin{bmatrix} \hat{s}(t|t) \\ \hat{\theta}(t) \end{bmatrix} &= \begin{bmatrix} \hat{s}(t|t-1) \\ \hat{\theta}(t-1) \end{bmatrix} + L(t)e(t) && (11.50)\\ e(t) &= \left(\vec{y}(t) - \vec{y}(\hat{s}(t|t-1), \hat{\theta}(t-1))\right) \end{aligned}$$

$$P(t|t) = P(t|t-1) - L(t)H^T(t)P(t|t-1), \qquad (11.51)$$

where

$$F(t+1) = \begin{bmatrix} F_{11}(t+1) & F_{12}(t+1) \\ 0_{\dim\theta \times n} & I_{\dim\theta\times\dim\theta} \end{bmatrix}, \qquad (11.52)$$

$$\begin{aligned} F_{11}(t+1) &= \partial_s g(\hat{s}(t|t), u(t+1), \hat{\theta}(t))\\ F_{12}(t+1) &= \partial_\theta g(\hat{s}(t|t), u(t+1), \hat{\theta}(t)) \end{aligned}$$

$$H^T(t) = [\partial_s^T y(t) \;\; \partial_\theta^T y(t)] \qquad (11.53)$$

$$\begin{aligned} L(t) &= P(t|t-1)H(t)S^{-1}(t) && (11.54)\\ S(t) &= H^T(t)P(t|t-1)H(t) + rI_{N\times N} \end{aligned}$$

The initial $P^{-1}(0|0)$ is set to be a zero matrix and $\hat{\theta}(0)$ is a small random vector. We have

$$\begin{aligned} & g(\hat{s}(t|t), u(t+1), \hat{\theta}(t))\\ = \;& \tanh(\hat{W}_2(t)\; \hat{s}(t|t) + \hat{W}_3(t)\; \vec{u}(t+1) + \hat{W}_4(t)). \end{aligned} \qquad (11.55)$$

The computational burden is again on $P(t|t)$ which requires $\mathcal{O}(\dim\hat{\theta}^3)$ multiplication.

Since θ is the augmented vector including all the parameters $\{W_1, W_2, W_3, W_4\}$ and $\hat{\theta}$ is the augmented vector including all the parameters $\{\tilde{W}_1, \tilde{W}_2, \tilde{W}_3, \tilde{W}_4\}$, the dimension of θ will be the total number of elements in W_1, W_2, W_3 and W_4. That is

$$\dim\theta = n(2N + M + 1).$$

Similarly, the dimension of $\hat{\theta}$ is given by

$$\dim\hat{\theta} = n(n + M + N + 1).$$

By comparing their computational complexities on updating the matrix $P(t)$ and $P(t|t)$ respectively, it is observed that training RNN may not be more time consuming than training a NARX model. So, we suggest the following indirect method for training NARX and RNN :

a. If $N > n$ and NARX has to be trained, we can first initialize a random NARX model and transform it to a RNN model. We then train the RNN using the extended Kalman filter method. Once the training is finished, we inversely transform RNN to a NARX model.

b. If $N < n$ and RNN has to be trained, we can first initialize a random RNN and transform it to a NARX. We then train the NARX using the forgetting recursive least square method. Once the training is finished, we inversely transform a NARX to RNN model.

11.2.2 On pruning

One should also realize that this equivalence result sheds light on the design of a more effective RNN pruning algorithm. As indicated in some papers [62, 125, 104], pruning a RNN is basically more difficult than pruning a feedforward neural network. One reason is the evaluation of the second order derivative of the error function. Therefore, it will be interesting to see whether we can reformulate the RNN pruning in such a way that is similar to a feedforward network pruning.

The idea is simple. After the RNN has been trained, it is transformed to an equivalent NARX model. Then we can apply optimal brain damage [55, 91] or other techniques [31, 76, 126] to prune the NARX model. Empirically, pruning a feedforward neural network is usually easier than pruning a recurrent neural network [56, 104]. Once the pruning procedure is finished, we transform it back to a RNN model. Of course, this kind of indirect pruning procedure for RNN does not ensure that the number of weights will be reduced.

Note that not all pruning techniques for feedforward networks can be applied. Two examples are the statistical stepwise method (SSM) [20] and RLS based pruning [56] as their pruning methods require information which can only be obtained during training. To do so, we will have to transform the RNN to an equivalent NARX model at the very beginning. Once the RNN is initialized, it is transformed to an equivalent NARX model. Once training of this equivalent NARX model is finished, we can apply methods such as statistical stepwise method, RLS based pruning and non-convergent method [22] to prune the NARX model. After pruning is finished, the pruned NARX model is transformed back to a RNN. For clarity, we summarize all these training and pruning ideas graphically in Figure 11.3.

11.2.3 On stability analysis of NARX model

Stability is one concern that researchers would like to know once a dynamic system has been identified. In RNN, some results on this issue have recently been derived [41, 100, 101]. In accordance with the equivalence of NARX and RNN, we can readily use these theorems to analyze the system stability.

Theorem 8 *A NARX model defined as in (11.24) is stable if the magnitude of all the eigenvalues of* W_2W_1 *are smaller than one.*

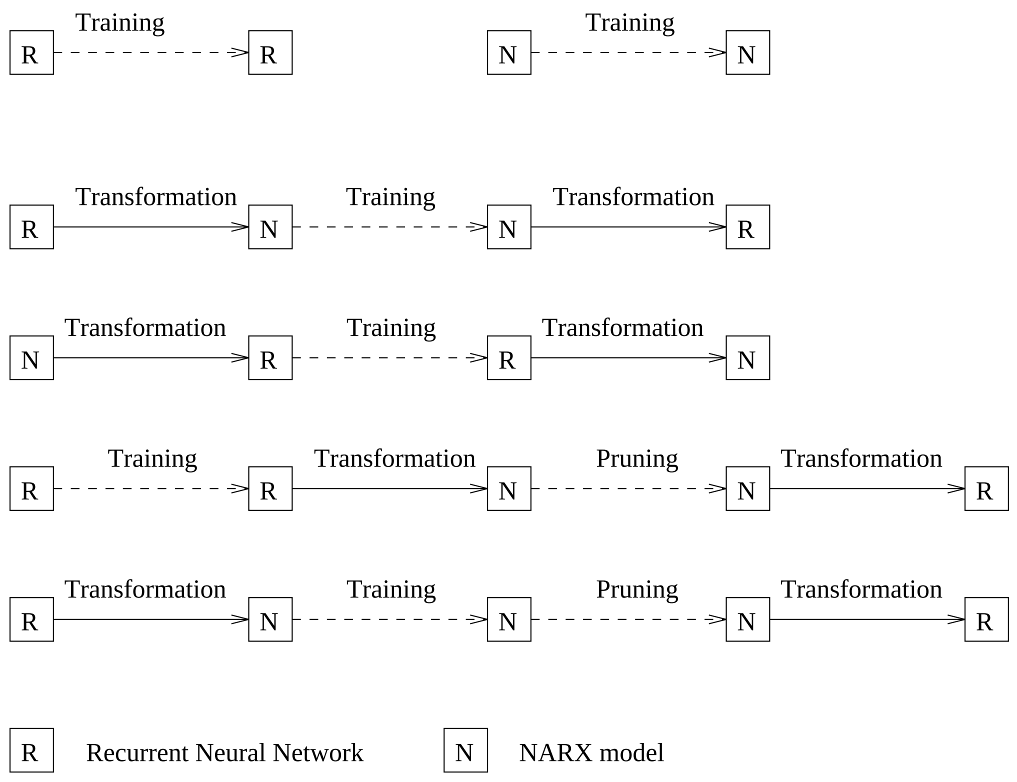


Figure 11.3: Summary of the training and pruning ideas implied by the model equivalence.

(Proof) Using the equivalence relation, a NARX model

$$\vec{y}(t+1) = \sum_{i=1}^{n} W_1 \tanh(W_2\ \vec{y}(t) + W_3\ \vec{u}(t+1) + W_4),$$

can be transformed to

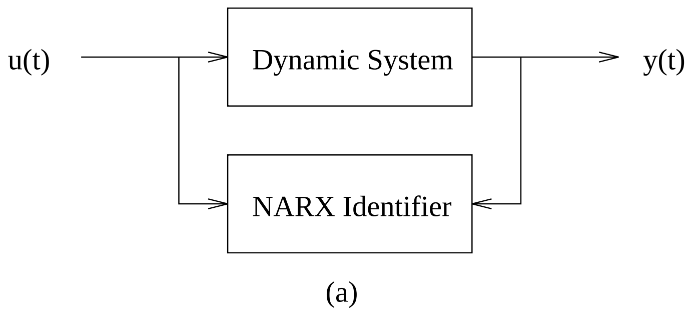
$$\begin{aligned} \vec{s}(t+1) &= \tanh(\tilde{W}_2\ \vec{s}(t) + \tilde{W}_3\ \vec{u}(t+1) + \tilde{W}_4) \\ \vec{y}(t+1) &= \tilde{W}_1\ \vec{s}(t+1) \end{aligned}$$

When no input is fed to the system, the difference between $\vec{s}(t+1)$ and $\vec{s}(t)$ is given by

$$\begin{aligned} \|\vec{s}(t+1) - \vec{s}(t)\| &= \|\tanh(W_2W_1\vec{s}(t) + W_4) \\ &\quad - \tanh(W_2W_1\vec{s}(t-1) + W_4)\| \\ &\leq \|W_2W_1\|\|\vec{s}(t) - \vec{s}(t-1)\|. \end{aligned}$$

Therefore, if all the eigenvalues of W_2W_1 are smaller than one, $\lim_{t\to\infty} \|\vec{s}(t+1) - \vec{s}(t)\| = 0$, which implies that $\vec{s}(t)$ will converge to a constant vector $\vec{s}_0$. Hence $\lim_{t\to\infty} \vec{y}(t) = W_1\vec{s}_0$. And the proof is completed.

□



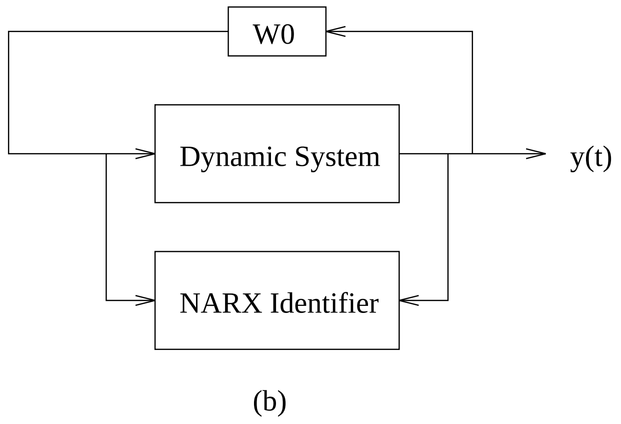


Figure 11.4: (a) Training of a NARX model to identify an unknown dynamic system. (b) Once the system is identified, an output feedback controller, $\vec{u}(t+1) = W_0\vec{y}(t)$, can be designed.

One consequence of Theorem 8 is on the design of a feedback controller for a dynamic system. Assuming that an unknown system is already identified by a NARX model with $\vec{y}_0$ being an equilibrium and the system is unstable, due to disturbance, the output of $\vec{y}_0$ shifts to $\vec{y}_0 + \Delta\vec{y}$. In order to make the output of the system go back to $\vec{y}_0$, one can design an output feedback controller as shown in Figure 11.4b,

$$\vec{u}(t+1) = W_0\vec{y}(t)$$

with W_0 satisfies the condition that all the eigenvalues of $(W_2 + W_3W_0)W_1$ are smaller than one. Usually, researchers proposed to use two neural networks, one for identification and the other for control [50, 80, 89, 90]. In order to make the controller work, two-phase training is needed. In the first phase, a neural network identifier has to be trained to identify the dynamical behavior of the system. Then, in the second phase, the weight values of the identifier are fixed and the controller network is trained. This will be a time consuming and difficult to implement as an on-line control method.

11.3 Summary

In this chapter, we have presented several results on the equivalence of the NARX model and a RNN. First, we have shown that *if the neuron transfer function is* tanh*, every first order NARX model can be transformed to a RNN and vice versa.* Second, we have also shown that *if the neuron transfer function is a piecewise linear function, every NARX model (irrespective of its order) can also be transformed to a RNN and vice versa.* In accordance with this equivalent relationship, we are able to

- speed up the training of a NARX or a RNN by indirect method,
- simplify the pruning procedure of a RNN,
- analyze the stability behavior of a NARX, and
- design an output feedback controller for the unknown dynamic system.

Chapter 12

Parallel Implementation of Training and Pruning Algorithms for FNN

The use of massively parallel processing architecture is one of the major reasons for the growth in the area of artificial neural networks (ANN). In recent years, most of the implementation of ANN have been accomplished by using general purpose, serial computers. This approach is though flexible but is often too slow. In training such a network, a suitable network size is hard to determine in advance, so one usually starts with a large network which consists of a large number of hidden units. After training is finished, the redundant weights are removed [55]. In order to speed up the training process, intensive research on the mapping of ANN onto parallel computing architectures [36, 17], such as mesh structure [63], array structure [52, 18, 38, 44, 129], ring structure [1, 77] and hypercube structure [51, 72] has been carried out. These methods solely apply the backpropagation approach to train the neural network.

FRLS is an alternative method which has been applied to train feedforward neural networks in recent years [28, 48]. Experimental results have demonstrated that the time complexity of using the FRLS training method is usually much smaller than that of using the backpropagation training method even though the one-step computation complexity in the FRLS method is higher than in the backprogation. Many advantages can also be gained by using FRLS.

- the FRLS method can easily be decoupled by assuming that the weights associated with one hidden neuron are uncorrelated with those weights associated with another hidden neuron [89, 95].
- the FRLS method can easily be modified to include the effect of regularization in neural network training [57].
- On-line pruning can easily be accomplished by using the FRLS method [56].

The rest of the chapter will describe how the FRLS training and pruning algorithm can be implemented on parallel architecture. The mapping of the training algorithm is indeed a direct extension of the algorithm presented in [89] and [95]. We add an analysis of the time complexity of using such parallel architecture. Besides from training, we describe how pruning can be realized by using parallel architecture.

In section one, the FRLS method in feedforward network training and pruning will be described. In section two, we describe the method by which FRLS training can be mapped onto a parallel architecture which consists of n processor element, where n is the number of hidden units. The time complexity is also analyzed in that section. The mapping of this pruning algorithm onto such a parallel architecture is presented in section three. The analysis of the speed up is presented in section four, and section five concludes the chapter.

12.1 Review of FRLS training and pruning

Without loss of generality, we will consider a neural network consists of n hidden units, one output unit and m input units. The function of a neural network is written as follows :

$$\hat{y}(x) = f(x, \theta), \tag{12.1}$$

where $y \in R$ is the output of the network, $x \in R^m$ is the input, $\theta \in R^{n(m+2)}$ is the parametric vector.

12.1.1 Training

Let $\hat{\theta}(0)$ be the initial parametric vector and $P(0) = \delta^{-1} I_{n(m+2)\times n(m+2)}$, the training of a feedforward neural network can be accomplished by the following recursive equations [48], [56], [57], [95]:

$$\begin{aligned} P(t) &= (I - L(t)H(t))P(t-1) & (12.2)\\ \hat{\theta}(t) &= \hat{\theta}(t-1) + L(t)[y(x_t) - \hat{y}(x_t)], & (12.3)\end{aligned}$$

where

$$\begin{aligned} L(t) &= \frac{P(t-1)H(t)}{H^T(t)P(t-1)H(t)+1} \\ H(t) &= \left.\frac{\partial f}{\partial \theta}\right|_{\theta=\hat{\theta}(t-1)} \end{aligned}$$

Suppose $f(x, \theta)$ is a linear function, the objective of the above algorithms is minimizing the mean square prediction $E(\theta)$ is given by

$$E(\theta) = \frac{1}{N}\sum_{k=1}^{N}(y(x_k) - f(x_k, \theta))^2. \tag{12.4}$$

Once f is a nonlinear function, FRLS can only be treated as a heuristic algorithm searching for the minimum $E(\theta)$. Fortunately, experimental studies always demonstrate that FRLS can give good solutions in neural network training. Besides, a FRLS based method converges much faster than the backpropagation approach.

12.1.2 Pruning

Basically, pruning is a model reduction technique which attempts to remove excessive model parameters. One heuristic pruning method is based on the idea of error sensitivity (see optimal brain damage [55] and optimal brain surgeon [31]). A weight is ranked in regard to its corresponding error sensitivity. If its error sensitivity is small, it is assumed to be less important. That is to say, $\hat{\theta}_i$ *can be set to zero if* $\nabla\nabla_{\theta_i\theta_i}E(\theta)$ *is small.*

Since after N iteration, the matrix

$$P^{-1}(N) \approx P^{-1}(0) + \sum_{k=1}^{N} H(k)H^T(k). \tag{12.5}$$

Multiplying the k^{th} diagonal element of $\frac{1}{N}P^{-1}(N)$ with the square of the magnitude of the k^{th} parameter, we can approximate the second order derivative of $E(\theta)$ by

$$\nabla\nabla E(\theta) = \frac{1}{N}\left[P^{-1}(N) - P^{-1}(0)\right]. \tag{12.6}$$

The saliency measure of the k^{th} weight can thus be computed by the following equation :

$$E(\hat{\theta}^k) - E(\hat{\theta}) \approx \frac{1}{N}\hat{\theta}_k^2\left(\left(P^{-1}(N)\right)_{kk} - \lambda\right) \tag{12.7}$$

if we let $P^{-1}(0)$ be equal to λI. Thus, the pruning algorithm can be summarized as following.

1. Evaluating $P^{-1}(N)$ and $\theta_k^2\left((P^{-1}(N))_{kk} - \lambda\right)$ for all k from 1 to n_θ.
2. Rearranging the index $\{\pi_k\}$ according to the ascending order of $\theta_k^2\left((P^{-1}(N))_{kk} - \lambda\right)$.
3. Setting $E(\hat{\theta}_{[1,0]}) = 0$ and $k = 1$
4. While $E(\hat{\theta}_{[1,k-1]}) < E_0$,
   (a) Computing validation error $E(\hat{\theta}_{[1,k]})$
   (b) $k = k + 1$.

Here $\hat{\theta}_{[1,k]}$ is the parameter vector where π_1 to π_k elements are zero and the π_{k+1} to $\pi_{n\theta}$ elements are identical to $\hat{\theta}_{\pi_{k+1}}$ up to $\hat{\theta}_{\pi_{n\theta}}$.

12.2 Mapping FRLS training onto n parallel processors

To map the training algorithm onto a parallel processor structure, some notations have to be defined. Let $w_1, w_2, \ldots, w_n$ be the weight vectors associated with the 1^{st} to the n^{th} hidden unit.

Example 2 *Suppose a neural network consists of two hidden units, one output unit and one input unit,*

$$\begin{aligned} f(x,\theta) &= \theta_{10}\,\sigma(\theta_{11}x+\theta_{12}) \\ &+ \theta_{20}\,\sigma(\theta_{21}x+\theta_{22}), \end{aligned}$$

where σ is the nonlinear sigmoidal function. In this model, $w_1 = (\theta_{10},\theta_{11},\theta_{12})^T$ and $w_2 = (\theta_{20},\theta_{21},\theta_{22})^T$.

Next, we assume that w_i and w_j are statistically uncorrelated. The FRLS algorithm can thus be decoupled into n filter equations [89, 95]. For $i = 1,\ldots,n$,

$$P_i(t) = (I - L_i(t)H_i(t))P_i(t-1) \tag{12.8}$$

$$\hat{w}_i(t) = \hat{w}_i(t-1) + L_i(t)[y(x_t) - \hat{y}(x_t)], \tag{12.9}$$

where

$$L_i(t) = \frac{P_i(t-1)H_i(t)}{H_i^T(t)P_i(t-1)H_i(t)+1} \tag{12.10}$$

$$H_i(t) = \left.\frac{\partial f}{\partial w_i}\right|_{w_i=\hat{w}_i(t-1)}. \tag{12.11}$$

For implementation purposes, we introduce a notation s_i for i from 1 to n, where $s_1 = \theta_{10}\,\sigma(\theta_{11}x+\theta_{12})$ and $s_2 = \theta_{20}\,\sigma(\theta_{21}x+\theta_{22})$.

Parallel algorithm The parallel algorithm can be described as follows :

```
1. PARBEGIN
   Initialize $w_i(0)$ and $P_i(0)$
   PAREND
2. FOR $t = 1,\ldots,T$ DO
   BEGIN
     (a) Input $x_t$ to PE.
     (b) Input $y_t$ to P0.
     (c) PARBEGIN
         Evaluate $s_i(x_t)$
         Output $s_i(t)$ to P0
         Evaluate $H_i(t)$
         Evaluate $L_i(t)$
         Evaluate $P_i(t)$
         Input $e(t) = [y(x_t) - \hat{y}(x_t)]$ from P0
         $\hat{w}_i(t) = \hat{w}_i(t-1) + L_i(t)e(t)$
         PAREND
   END
```

The parallel architecture for such implementation is shown in Figure 12.1.

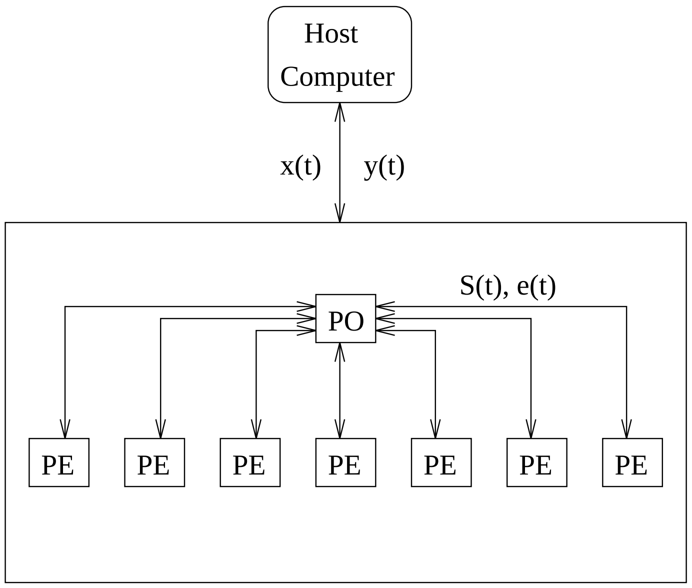


Figure 12.1: Parallel architecture for the implementation of FRLS based training.

Time complexity analysis Consider the case that the dimension of x_t is m and the element is denoted by x_{t1} to x_{tm}. Without loss of generality, we assume the output dimension is one.

For the evaluation of $s_i(x_t)$, it requires $(m+2)$ flops[1]. From Eqn (12.11), one can readily derive that

$$H_i(t) = \begin{bmatrix} \sigma_i \\ w_{i0}\ \sigma_i\ (1-\sigma_i)x_{t1} \\ w_{i0}\ \sigma_i\ (1-\sigma_i)x_{t2} \\ \cdots \\ w_{i0}\ \sigma_i\ (1-\sigma_i)x_{tm} \\ w_{i0}\ \sigma_i\ (1-\sigma_i) \end{bmatrix}, \tag{12.12}$$

where $\sigma_i = \sigma(\sum_{k=1}^{m} w_{ik}(t-1)x_{tk}+w_{i,m+1})$. This step requires $(m+2)$ flops. As $P_i(t-1)H_i(t)$ is also involved in the evaluation of $P_i(t)$, $P_i(t-1)H(t)$ is first computed and then stored in a register. The complexity of computing $P_i(t-1)H(t)$ is $\frac{1}{2}(m+2)(m+3)$. Hence, the evaluation of $L_i(t)$ requires $\frac{1}{2}(m+2)(m+3)+(m+2)$ flops. To compute $P_i(t)$, it requires only $(m+2)^2$ flops. Finally, it requires $(m+2)$ flops for the update of $w_i(t)$.

For the time being, we ignore the communication delay. Summing them altogether, the computation cost for *one-step update* is

$$\frac{1}{2}(m+2)(3m+15)$$

which is $\mathcal{O}(m^2)$. As we have ignored the communication and computation cost due to *P0*[2], the time complexity of using n parallel processor structure to realize FRLS is $\mathcal{O}(m^2)$.

[1]We assume the computation cost on addition and evaluation of $\sigma(.)$ are relatively very small.

[2]It is reasonable to have this assumption. Since the purpose of *P0* is receiving signal y_t from the host

Method	Processor Complexity	Time Complexity
Original	1	$\mathcal{O}((n+m)^2 d_f)$
Parallel	$\mathcal{O}(n)$	$\mathcal{O}(m^2 d_f)$

Table 12.1: Complexity difference between the original and parallel FRLS in one-iteration step.

Compared with the original single machine scheme, Table 12.1, the speed up can be very large. In case communication delay is counted, the time complexity will be

$$\max\left\{0,\ (n-1)d_c - \frac{1}{2}(m+2)(3m+7)d_f\right\} + \frac{1}{2}(m+2)(3m+15)d_f$$

where d_f and d_c are floating point multiplication delay and communication delay. The time complexity will be the maximum of $(n-1)d_c + 2(m+2)d_f$ and $\frac{1}{2}(m+2)(3m+15)d_f$. The total time delay for complete training will be the maximum of

$$N_t\left((n-1)d_c + 2(m+2)d_f\right),\ \ \frac{N_t}{2}(m+2)(3m+15)d_f$$

if the total number of training data is N_t.

12.3 Implementation of FRLS based pruning in parallel processors

Because of the use of the decoupling FRLS training method, the pruning process becomes much simpler. Instead of using the one-phase approach as described in the last section, we can define a two-stage pruning scheme.

In stage-1, we evaluate the error sensitivity for each output weight. If the error sensitivity of an output weight is smaller than a threshold, the corresponding output weight is removed. Hence those weights associated with this neuron have to be examined[3].

In stage-2, the remaining un-pruned weights are then ranked in accordance with their error sensitivity. This can be accomplished in two sub-steps. First, each PE will evaluate their local ranking list $\pi^{(i)} = (\pi_1^{(i)}, \ldots, \pi_{m+1}^{(i)})$ for w_{i1} to $w_{i,m+1}$ according to the ascending order of their error sensitivities. Then all these $\pi^{(i)}$ will merge and sort to give a global ranking list ϖ with at most $n(m+1)$ elements. Weights are thus removed one by one until the accumulative error E_{acc} is larger than a threshold.

computer and computer the predicted network output $\hat{y}_t$ by

$$\hat{y}_t = \sum_{i=1}^{n} s_i(x_t).$$

This involves only addition. Based on the assumption that computation cost for addition is much smaller than multiplication, we can ignore this cost as well.

[3]This is valid only for a single output unit neural network. Suppose the network output is larger than one, a similar technique can be applied.

Parallel algorithm This pruning scheme can be summarized in the following algorithm.

```
1. PARBEGIN
   Evaluating $SEN_{i0} = w_{i0}^2 \left( \left( P_i^{-1}(N) \right)_{00} - \lambda \right)$
   PAREND
   Rearrange index $\{\pi_i\}$
   $E_{acc} = 0$ and $k = 1$
   WHILE $E_{acc} < E_0$ DO
   BEGIN
      (a) Disable $PE_1$ to $PE_k$
      (b) Evaluate validation error $E_{acc} = E(w_{\pi_{10}} = \ldots = w_{\pi_{k0}} = 0)$
      (c) $k = k + 1$
   END
2. For the activated $PE$
   PARBEGIN[4]
      (a) Evaluating $SEN_{ij} = w_{ij}^2 \left( \left( P_i^{-1}(N) \right)_{jj} - \lambda \right)$ for all $j$
      (b) Rearrange index $\{\pi_j^{(i)}\}$
      (c) Output $\pi^{(i)} = (\pi_1^{(i)}, \ldots, \pi_{m+1}^{(i)})$ to host computer[5]
   PAREND
   Set $E_{acc} = 0$ and $k = 1$
   WHILE $E_{acc} < E_0$ DO
   BEGIN
   PARBEGIN
   Host computer output $\varpi_k$
   Evaluate validation error $E_{acc} = E(w_{\varpi_1} = \ldots = w_{\varpi_k} = 0)$
   $k = k + 1$
   PAREND
   END
```

The way of generating ϖ_k can be described by the following example. Suppose there are three sorted lists : $(1, 3, 5)$, $(7, 8, 9)$ and $(2, 4, 6)$ respectively. Comparing $\pi_1^{(i)}$ for $i = 1$ to 3, '1' is the minimum of the nine numbers. Therefore, $\varpi_1 = (1, 1)$. Once the ϖ_1^{st} weight is removed, the numbers being compared are $\pi_2^{(1)}, \pi_1^{(2)}, \pi_1^{(3)}$, i.e. $3, 7, 2$. The operation for the first step up to the 4th step is shown in Figure 12.2.

[4]Note that SEN_{i0} is not involved in this state.

[5]Certainly, these data can also be output to processor PO for sorting if the local memory size in it is larger enough to handle the prune-index generation.

	5	9	6	
Initial	3	8	4	$\rightarrow \varpi_1 = (1,1)$
	1	7	2	
	–	9	6	
step 1	5	8	4	$\rightarrow \varpi_2 = (3,1)$
	3	7	2	
	–	9	–	
step 2	5	8	6	$\rightarrow \varpi_3 = (1,2)$
	3	7	4	
	–	9	–	
step 3	–	8	6	$\rightarrow \varpi_4 = (3,2)$
	5	7	4	
	–	9	–	
step 4	–	8	–	$\rightarrow \varpi_5 = (1,3)$
	5	7	6	

Figure 12.2: The pruning index generation algorithm.

Time complexity analysis Similar to the analysis in training, we discard the communication cost. The evaluation of SEN_{i0} requires only $2d_f$ time delay. The rearrangement of index π_i requires a sorting algorithm. This can be done by $n \log n d_s$ time delay in the host computer. Here d_s is the time delay for one comparison. The time delay bound for stage-1 pruning will be

$$2d_f + (n \log n)d_s + nN_v\left((m+2)d_f + (n-1)d_c\right)$$

where N_v is the total number of validation data.

In stage-2, the analysis is similar. For the evaluation of SEN_{ij}, the time delay is $2md_f$. For the generation of the list $\{\pi_j^{(i)}\}$, the time delay is $(m+1)d_s$. This is the time required by each processor. To generate the first index using the idea described above, $(n \log n)d_s$ delay is required. Therefore, the time delay bound for stage-2 pruning will be

$$n(m+1)\Delta + 2(m+1)d_f + (n \log n)d_s$$

where

$$\Delta = \max\{(\log n)d_s, N_v\left((m+2)d_f + (n-1)d_c\right)\}.$$

Summing two time delay bound, we can get that the time delay bound for pruning is

$$4d_f + 2(n \log n)d_s + nN_v(m+2)d_f + n(m+1)\Delta.$$

If $N_v(m+2)d_f \ll \log n d_s$, the time delay bound is in the order of $\mathcal{O}(n)$. On the other hand, if $N_v(m+2)d_f > (\log n)d_s$, the time complexity may be larger than $\mathcal{O}(n \log n)$.

From these simple analyse, we can see that the sorting preprocessing step can be a burden in pruning if the size of the validation set is small. In that case, other non-heuristic algorithms may outperform the conventional pruning method in terms of time complexity.

12.4 Speed up

Suppose communication delay is not taken into account, i.e. $d_c = 0$, the time delay for complete training using SIMD architecture is given by

$$T_{train}(SIMD) = \frac{N_t}{2}(m+2)(3m+15)d_f. \tag{12.13}$$

Using a single processor, the time delay will be given by

$$T_{train}(SIN) = \frac{N_t}{2}n(m+2)(3n(m+2)+9)d_f. \tag{12.14}$$

In the pruning phase, the time delay bound for pruning using SIMD architecture is given by

$$T_{prune}(SIMD) = 4d_f + 2(n\log n)d_s + nN_v(m+2)d_f + n(m+1)\Delta \tag{12.15}$$

where

$$\Delta = \max\{(\log n)d_s, N_v\left((m+2)d_f + (n-1)d_c\right)\}.$$

Using a single processor, the time delay bound for pruning will be given by

$$T_{prune}(SIN) = 2n(m+2)d_f + (n\log n)d_s + nN_v(m+2)d_f + [n(m+1)\log n(m+1)]d_s + n(m+1)(\log n)d_s \tag{12.16}$$

The speed up factor can thus be defined as follows :

$$F_{su} = \frac{T_{train}(SIN) + T_{prune}(SIN)}{T_{train}(SIMD) + T_{prune}(SIMD)} \tag{12.17}$$

Figure 12.3 shows the case when $m = 5$, $N_t = 200$ and $d_f = 5d_s$. From the speed up curve, we can see that the speed up is not linear. It is asymptotically equal to a constant which depends on the ratio of N_t to N_v.

12.5 Summary

In summary, we have presented a parallel algorithm for the realization of recursive least square based training and pruning for a neural network implemented on a SIMD machine. Discarding the delay due to data communication, time delay bounds for both training and pruning are deduced. From such derived results, preliminary analysis such as total time delay and speed-up can be accomplished. Finally, it should be noted that the method presented in this chapter can equally be applied to the extended Kalman filter based training method.

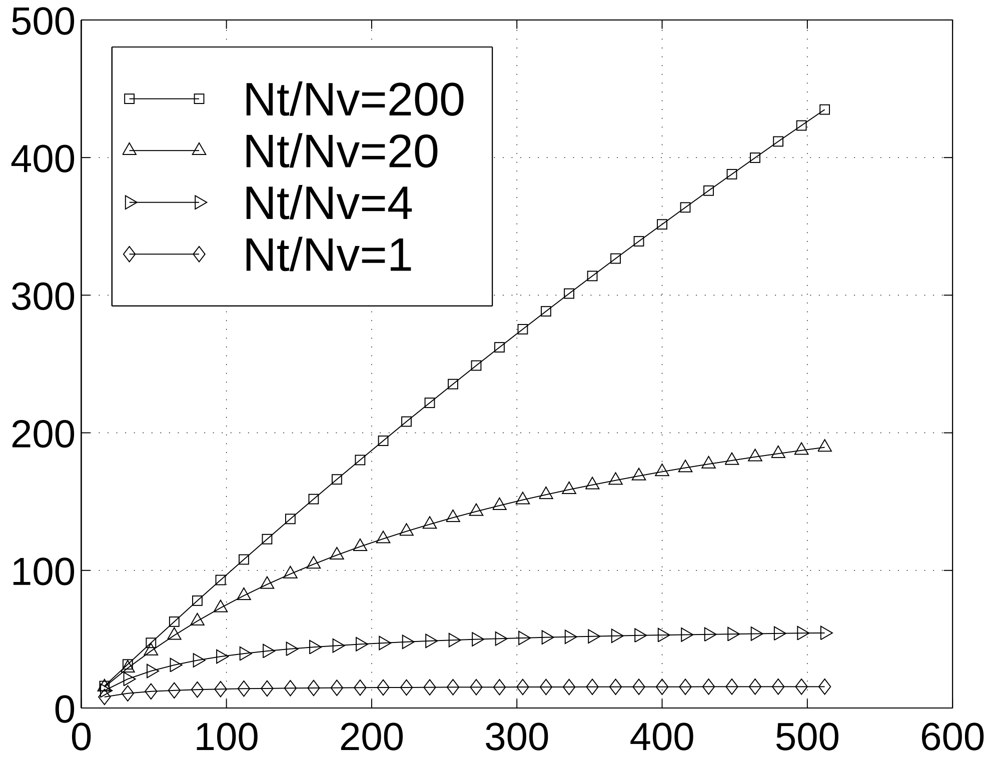


Figure 12.3: The speed up factor against the number of processors used in the parallel architecture for the implementation of FRLS based training.

Part V

Conclusion

Chapter 13

Conclusion

Being the last chapter of this dissertation, its first section summarizes the contribution of the research work done in previous chapters. The second section discusses some future work that can be extended from the work done here.

13.1 Revisit of the contributions of the dissertation

We have reviewed several contemporary techniques in neural network learning and revealed some of their limitations in training and pruning. Then, two weight importance measures based on the ideas of

1. *expected prediction error* sensitivity and
2. *a posterior probability* sensitivity,

have been proposed for both feedforward neural network (FNN) and recurrent neural network (RNN). Extensive simulation and comparison results have been included to justify the effectiveness of those algorithms. Moreover, several theoretical and implementation issues have also been presented to supplement some background analysis on neural network learning. The contributions made in this dissertation are highlighted in the following paragraphs.

13.1.1 EKF-based pruning

For FNN, a weight importance measure linking up prediction error sensitivity and the by-products obtained from EKF training has been derived. Comparison results have demonstrated that the proposed measure can better approximate the prediction error sensitivity than using the forgetting recursive least square (FRLS) based pruning measure. Another weight importance measure that links up the *a posteriori* probability sensitivity and by-products obtained from EKF training has also been derived. An adaptive pruning procedure designed for FNN in a non-stationary environment has presented. Simulation results illustrate that the proposed measure together with the pruning procedure is able to identify redundant weights and remove them. As a result, the computation cost for EKF-based training can also be reduced.

Using a similar idea, a weight importance measure linking up the *a posteriori* probability sensitivity and by-products obtained from EKF training has been derived for RNN. Applications of such a pruning algorithm together with the EKF-based training in system identification and time series prediction have been presented. The computational cost required for EKF-based pruning has analyzed. Several alternative pruning procedures have been proposed to compare with EKF-based pruning procedure. Comparative analysis in accordance with computational complexity, network size and generalization ability has been presented. No simple conclusion can be drawn from the comparative results. However, these results provide a guideline for practitioners once they want to apply RNN in system modeling.

13.1.2 Several aspects of neural nework learning

Several new results with regard to neural network learning have been presented in this dissertation. To provide a support for the use of recurrent neural network in system modeling, the approximate realizability of the Elman recurrent neural network has been proved. It has also been proved that FRLS training can have an effect identical to weight decay. This provides more evidence showing the advantages of using FRLS in training a neural network. Another result is the proof of the equivalence between a NARX model and recurrent neural network. Finally, a parallel implementation methodology for FRLS training and pruning on a SIMD machine has been presented.

13.1.3 Review on the limitation in existing learning techniques

Apart from proposing new measures for pruning and presenting new theoretical results for neural network learning, this dissertation has also revealed several limitations in the existing learning techniques.

First, it has revealed that training a RNN by gradient descent or the FRLS approach usually works well when the training set is defined as $\{u_i, y_i\}_{i=1}^{N}$ and the validation set is defined as $\{u_i, y_i\}_{i=N+1}^{N+T}$. However, in case the validation set is defined as $\{u_i, y_i\}_{i=1}^{T}$ and the training set is defined as $\{u_i, y_i\}_{i=T+1}^{N+T}$ (or the form of the training input is different from that of the validation input), it might happen that the validation error will be much greater than the training error.

Second, it has revealed that the error sensitivity measure is not appropriate for pruning RNN due to the composite weight removal and initialization error effect. It is hard to identify how much the validation error is caused by the weight removal and how much by initialization error.

Third, it has also discussed a shortcoming of using error-sensitivity-based pruning methods, such as OBD and OBS. Since backpropagation is a slow training method, the actual time for obtaining a good network structure using the backpropagation training method together with the error-sensitivity-based pruning method may thus be time-consuming. If the nature of the problem is *non-stationary*, it will be much more difficult to implement such a pruning method since training will never finish and then the ranking of the weight importance obtained at time t may not be equal to the weight importance obtained at other times.

13.2 Future work

The work presented in this dissertation is only a small part of the whole neural network learning theory. Lots of work ought to be done to make the learning theory complete. Some possible work leading from that presented in this dissertation is summarized in the following paragraphs.

Alternative training and pruning methods for RNN

One future direction is to develop an alternative training method for RNN. In this dissertation, the training method employed is the Williams method. The idea can be described as follows :

$$\cdots \overset{\text{EKF}}{\rightarrow} \left[\begin{array}{c} \hat{x}(t-1|t-1) \\ \hat{\theta}(t-1) \end{array} \right] \overset{\text{EKF}}{\rightarrow} \left[\begin{array}{c} \hat{x}(t|t) \\ \hat{\theta}(t) \end{array} \right] \overset{\text{EKF}}{\rightarrow} \cdots$$

That is, both $\hat{x}(t|t)$ and $\hat{\theta}(t)$ are assumed to be coupled with each other. This is one reason why an error sensitivity measure is not easy to define. One approach to facilitate the definition of an error sensitivity measure for RNN is to decouple the updating of $\hat{\theta}(t)$ and estimation of $\hat{x}(t|t)$. That is, an on-line training can be accomplished by gradient descent whereas an on-line estimation of $x(t)$ is accomplished by EKF.

$$\begin{array}{ccccccc} \cdots & \overset{\text{EKF}}{\rightarrow} & \hat{x}(t-1|t-1) & \overset{\text{EKF}}{\rightarrow} & \hat{x}(t|t) & \overset{\text{EKF}}{\rightarrow} & \cdots \\ \cdots & \overset{\text{GD}}{\rightarrow} & \hat{\theta}(t-1) & \overset{\text{GD}}{\rightarrow} & \hat{\theta}(t) & \overset{\text{GD}}{\rightarrow} & \cdots \end{array}$$

Using this idea, a better estimation of $\hat{\theta}(t)$ and $\hat{x}(t-1|t-1)$ might be obtained. Hence, the significance of $\hat{\theta}(t)$ and $P_{\theta\theta}(t)$ might be improved. Useful tools for the design and performance analysis of such an approach should be investigated.

Imprecise Neural Computation

To control a dynamic system in real-time, a real-time system identifier is usually required. Figure 13.1 shows the block diagram of a self-tuning regulator. Assuming a parametric model for the system, the *identifier* estimates the model parameters from the sampled input-output data pair. Then a model parameter is passed to the *design consideration* block to generate a regulator parameter, which is used for suggesting a suitable regulation scheme. Once the regulator has received its parameter, it generates a sequence of control signals according to the control scheme.

This design methodology has been used by a neural network researchers who employ neural network as identifier and regulator. As training of a neural network is time consuming, application of such a real-time controller can only be feasible for soft real-time control problems such as process control [102]. To deal with such problems, we have recently proposed a conceptual framework called imprecise neural computation [110] to extend the idea of imprecise computation, by introducing a concept of mandatory structure.

Imprecise neural computation is a generic principle which tries to facilitate the software and hardware design of a neural network based system in dealing with time-critical problems. Traditionally, a neural network is simply treated as a black box. Once a batch of data has been collected from a dynamic system, a neural network identifier will be trained

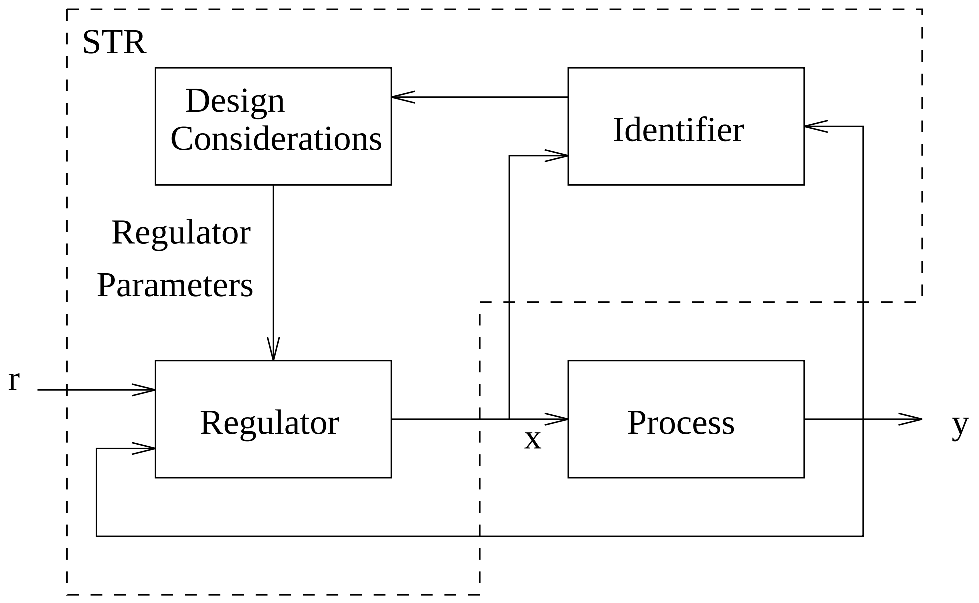


Figure 13.1: Block diagram of a real-time self-tuning regulator. Adapted from Astrom K. and B. Wittenmark, *Computer-Controlled Systems*, Prentice Hall, 1990.

to learn the system behavior and then a neural network controller will be trained to control the system. The cost of building such a controller/identifier system is solely determined by the quality of the system output with respect to the desired output. Heavy computational cost and the error incurred due to incomplete computation are usually ignored.

In real-time problems, one always needs to face the following important problem :

Problem 1 *If it is not possible to finish all computation tasks within the given time, what should we do?*

Similarly, in real-time neural computing, we face the same problem :

Problem 2 *If it is not possible to train a fully connected neural network with the whole data set within a given allowable time span, what should we do?*

These problems can be answered in the light of imprecise computation [83, 115, 103].

Principle 1 (Imprecise Computation) *If the computation of a task can be partitioned into a mandatory part and a optional part (where the mandatory part is the necessary portion which must be executed in order to achieve an acceptable result according to a performance measure while the optional part is the portion for refining the result), the optional part may be left unfinished when it is not feasible to complete all tasks.*

In a time-critical situation, the system can decide how much of the optional part can be executed.

Therefore in the design of an algorithm for solving such a problem, the execution time for the mandatory part should be small enough in order to meet the critical time constraint by trading off the quality of the results [58, 59, 63, 61].

Borrowing the idea from imprecise computation in real-time computing, the imprecise neural computation [110] which provides a conceptual framework for the design and analysis of real-time neural system can be proposed. Imprecise neural computation not just concerns the issue of computation complexity, but also the complexity of the neural network being used. The basic principle of imprecise neural computation can be stated as follows :

Principle 2 (Imprecise Neural Computation) *If the learning for neural computation can be partitioned into a mandatory part and optional part where the mandatory part is the necessary portion which must be executed in order to achieve an acceptable result according to a performance measure while the optional part is the portion for refining the result), the optional part may be left unfinished when it is not feasible to complete all tasks.*

In a time-critical situation, the system can decide how much of the optional part can be executed.

From the implementation point of view, imprecise neural computation provides a new conceptual framework for studying methodologies of employing a neural network approach in real-time applications. With the advancement of the hardware and parallel computing technology, true real-time neural systems for solving real-time problems will be possibly achievable in the future.

Appendix A

Extended Kalman filter

A.1 Kalman filter

In the theory of linear system identification and state estimation, Kalman filter is one useful approach [2, 4, 9, 99]. Basically, the essential problem that the Kalman filter solves is a state estimation problem in linear systems.

Optimal state estimation: *Consider a non-autonomous system:*

$$\begin{aligned} x(t+1) &= Fx(t) + Gu(t) + v(t), & \text{(A.1)} \\ y(t) &= Hx(t) + e(t), & \text{(A.2)} \end{aligned}$$

where $v(t)$ and $e(t)$ are white noise sequences with zero means and covariance matrice

$$E\begin{pmatrix} v(t) \\ e(t) \end{pmatrix} (v^T(t)\; e^T(t)) = \begin{pmatrix} R_1 & R_{12} \\ R_{21} & R_2 \end{pmatrix}. \tag{A.3}$$

At time t, the available information is the measurements, $Y^t = \{y(t), u(t), y(t-1), u(t-1), \ldots, y(t_0), u(t_0)\}$. Now find the optimal state estimate of $x(t)$ given the measurements.

Denoting $\hat{x}(t|s) = E[x(t)|Y^s]$ and $P(t|s) = E[x(t) - \hat{x}(t|s)][x(t) - \hat{x}(t|s)]^T$ to be the conditional mean and the associated error covariance matrix, the one step ahead state estimate, $\hat{x}(t+1|t)$, can be calculated as follows [99]:

$$\begin{aligned} \hat{x}(t+1|t) &= F\hat{x}(t|t-1) + Gu(t) + K(t)[y(t) - H\hat{x}(t|t-1)], & \text{(A.4)} \\ x(t_0|t_0-1) &= Ex(t_0), & \text{(A.5)} \end{aligned}$$

where

$$\begin{aligned} K(t) &= [FP(t|t-1)H^T + R_{12}][HP(t|t-1)H^T + R_2]^{-1}, & \text{(A.6)} \\ P(t+1|t) &= FP(t|t-1)F^T + R_1 - K(t)[R_{21} + HP(t|t-1)F^T], & \text{(A.7)} \\ P(t_0|t_0-1) &= cov(x(t_0)). & \text{(A.8)} \end{aligned}$$

A.2 Kalman filter for linear system identification

Now suppose that the measurements of the input $\{u(t)\}$ and output $\{y(t)\}$ are available in real time for the following ARMA system:

$$y(t) + a_1 y(t-1) + \ldots + a_n y(t-n) = a_{n+1} u(t-1) + \ldots + a_{n+m} u(t-m). \tag{A.9}$$

It is readily shown that the identification of system parameters $\{a_i\}$ can be formulated as an state estimation problem and solved by the Kalman filter.

Define $x(t) = [a_1\ a_2 \ldots a_{n+m}]^T$ as a state vector,

$$H^T(t) = [-y(t-1)\ -y(t-2) \ldots -y(t-n)\ u(t-1)\ u(t-2) \ldots u(t-m)] \tag{A.10}$$

as a row vector and define the process $\{z(t)\}$ by $z(t) = y(t)$. Then the system identification problem can be written as the following state estimation problem:

$$\begin{aligned} x(t+1) &= x(t) + v(t), && \text{(A.11)} \\ z(t) &= H^T(t) x(t) + e(t). && \text{(A.12)} \end{aligned}$$

Suppose that the noise processes, $\{v(t)\}$ and $\{e(t)\}$, are independent, the identification problem (A.12) can be solved by making use of the above Kalman filtering algorithm [2]:

$$\begin{aligned} \hat{x}(t+1|t) &= [I - K(t)H^T(t)]\hat{x}(t|t-1) + K(t)z(t), && \text{(A.13)} \\ K(t) &= P(t|t-1)H(t)[H^T(t)P(t|t-1)H(t) + R_2]^{-1}, && \text{(A.14)} \\ P(t+1|t) &= (I - K(t)H^T(t))P(t|t-1) + R_1. && \text{(A.15)} \end{aligned}$$

Remarks

- It should be noted that the values of R_1 and R_2 are in general not known. Therefore, we need to estimate it as *a prior* information or estimate it adaptively. For instance,

  $$R_1(t) = (1-\delta)R(t-1) + \delta(z(t) - H^T(t)x(t))^2,$$

  where δ is a small positive number.

- The above algorithm is also a version of the recursive least square method in for the linear regression problem once the value R_1 is set to be zero and R_2 is unity (see section 6.3 of [78] for a detailed discussion on the relation between the recursive least square method and Kalman filtering.).

- If R_1 is a zero matrix and R_2 is unity, it can be easily checked that the covariance matrix P is identical to the inverse of the input covariance matrix, i.e.

  $$P^{-1}(N|N) = \sum_{t=1}^{N} H(t)H^T(t) + P(0|-1).$$

A.3 Extended Kalman filter

For the case that the system is non-linear, the extended Kalman filter is applied as a sub-optimal filter. The key idea of the extended Kalman filter stamps on the approximation of non-linear terms by first order linear approximation. Consider the non-linear autonomous model:

$$\begin{aligned} x(t+1) &= f_t(x(t)) + g_t(x(t))v(t), \quad &(A.16)\\ z(t) &= h_t(x(t)) + e(t). \quad &(A.17)\end{aligned}$$

The non-linear functions f_t, g_t, h_t are suffixed by t indicating that these non-linear function may even be time varying. If the these functions are sufficiently smooth, they can be expanded in a Taylor series:

$$\begin{aligned} f_t(x(t)) &= f_t(\hat{x}(t|t)) + F_t(x(t) - \hat{x}(t|t)) + \ldots \quad &(A.18)\\ g_t(x(t)) &= g_t(\hat{x}(t|t)) + \ldots = G_t + \ldots \quad &(A.19)\\ h_t(x(t)) &= h_t(\hat{x}(t|t-1)) + H_t^T(x(t) - \hat{x}(t|t-1)) + \ldots, \quad &(A.20)\end{aligned}$$

where

$$F_t = \frac{\partial f_t(\hat{x}(t|t))}{\partial x}, \quad H_t^T = \frac{\partial h_t(\hat{x}(t|t-1))}{\partial x}, \quad G_t = g_t(\hat{x}(t|t)).$$

Ignoring those higher order terms, the non-linear model (A.17) can be approximated as follows:

$$\begin{aligned} x(t+1) &= F_t x(t) + G_t v(t) + u(t), \quad &(A.21)\\ z(t) &= H_t^T x(t) + e(t) + y(t) \quad &(A.22)\end{aligned}$$

where

$$u(t) = f_t(\hat{x}(t|t)) - F_t\hat{x}(t|t),$$

and

$$y(t) = h_t(\hat{x}(t|t-1)) - H_t^T\hat{x}(t|t-1).$$

The Kalman filter for this approximated non-linear model is then given as the following **extended Kalman filter** [2]:

$$\begin{aligned} \hat{x}(t|t) &= \hat{x}(t|t-1) + L_t[z(t) - h_t(\hat{x}(t|t-1))] \quad &(A.23)\\ \hat{x}(t+1|t) &= f_t(\hat{x}(t|t)) \quad &(A.24)\\ L_t &= P(t|t-1)H_t[H_t^T P(t|t-1)H_t + R_2]^{-1} \quad &(A.25)\\ P(t|t) &= P(t|t-1) - L_t H_t^T P(t|t-1) \quad &(A.26)\\ P(t+1|t) &= F_t P(t|t) F_t^T + G_t R_1 G_t^T \quad &(A.27)\end{aligned}$$

where

$$R_1 = E\{vv^T\},$$
$$R_2 = E\{ee^T\}.$$

It should be remarked that the solution obtained by using the extended Kalman filter approach is not optimal, in contrast to the Kalman filter [2], and the convergence is also not ensured [64].

Bibliography

[1] d'Acierno A. and R. Del Balio, Nested-rings architecture for feedforward networks, in E.R. Caianiello (ed) *Parallel Architectures and Neural Networks*, World Scientific, 1991.

[2] Anderson B.D.O. and J. Moore (1979). *Optimal Filtering*, Prentice Hall Inc.

[3] Astrom K. and B. Wittenmark, *Computer-Controlled Systems*, Prentice Hall, 1990.

[4] K.J. Astrom and B. Wittenmark, *Adaptive Control*, 2nd ed., Addison Wesley, 1995.

[5] Barron A., Prediction squared error: A criterion for automatic model selection. In *Self-Organizing Methods on Modeling*, S. Farlow, ed., Marcel Dekker, New York. 1984.

[6] S.A. Billings *et al.*, Properties of neural networks with applications to modelling non-linear dynamical systems, *International Journal of Control*, Vol.55(1), 193-224, 1992.

[7] Billings S.A. and C.F. Fung, Recurrent radial basis function networks for adaptive noise cancellation. *Neural Networks*, Vol.8(2), 273-190, 1995.

[8] Bishop C.M., Training with noise is equivalent to Tikhonov regularization, *Neural Computation*, Vol.7(1), 108-116, 1995.

[9] R.S. Bucy, Linear and nonlinear filtering, *Proceedings of the IEEE*, Vol.58(6), 854-864, 1970.

[10] Chen S., S.A. Billings and P.M. Grant. Non-linear system identification using neural networks. *International Journal of Control*, Vol. 51(6), 1191-1214, 1990.

[11] S. Chen *et al.*, Practical identification of NARMAX models using radial basis functions, *International Journal of Control*, Vol.52(6), 1327-1350, 1990.

[12] Chen S., C. Cowan, S.A. Billings and P.M. Grant (1990). Parallel recursive prediction error algorithm for training layered neural networks. *International Journal of Control*, Vol.51(6), 1215-1228.

[13] S. Chen *et al.*, Recursive hybrid algorithm for non-linear system identification using radial basis function networks, *International Journal of Control*, Vol.55(5), 1051-1070, 1992.

[14] S. Chen *et al.*, A clustering technique for digital communication channel equalization using radial basis function networks, *IEEE Transactions on Neural Networks*, Vol.4(4), 570-579, 1993.

[15] Chen S. and J. Wigger, Fast orthogonal least squares algorithm for efficient subset model selection, *IEEE Transactions on Signal Processing*, Vol.43(7), 1713-1715, 1995.

[16] Cho J., Y. Kim and D. Park (1997). Identification of nonlinear dynamic systems using higher order diagonal recurrent neural network, *Electronics Letters*, Vol.33(25), 2133-2135.

[17] Cong B, N. Yu and W. Zhou, Image data classification by neural networks and SIMD machines, *Proceedings of PDCS'96*, Chicago, 213-217, 1996.

[18] Cong B., Mapping of ANNs on linear array with a reconfigurable pipelined bus system, *Proceedings of PDPTA'97*, Vol.I, 522-529, 1997.

[19] Connor J.T., D. Martin and L.E. Atlas, Recurrent neural networks and robust time series prediction, *IEEE Transactions on Neural Networks* Vol.5(2), 240-254, 1994.

[20] Cottrell M., *et al.* (1995). Neural modeling for time series: A statistical stepwise method for weight elimination. *IEEE Transactions on Neural Networks* Vol.6(6), 1355-1362.

[21] G. Cybenko, Approximation by superpositions of a sigmoidal function, *Mathematics of Control, Signals, and Systems*, Vol.2, 303-314, 1989.

[22] Finnoff W., F. Hergert and H.G. Zimmermann (1993). Improving model selection by nonconvergent methods. *Neural Networks*, Vol 6.:771-783.

[23] K. Funahashi, On the approximate realization of continuous mappings by neural networks, *Neural Networks*, Vol.2, 183-192, 1989.

[24] K. Funahashi and Y. Nakamura, Approximation of dynamical systems by continuous time recurrent neural networks, *Neural Networks*, Vol.6(6), 801-806, 1993.

[25] C.L. Giles *et al.*, Learning and extraction finite state automata with second-order recurrent neural networks, *Neural Computation*, Vol.4, 393-405, 1992.

[26] F. Girosi and T. Poggio, Networks and the best approximation property, *Biological Cybernetics*, Vol.63, 169-176, 1990.

[27] Girosi F. *et al.* (1995), Regularization theory and neural networks architectures, *Neural Computation*, Vol.7, 219-269.

[28] Gorinevsky D., On the persistency of excitation in radial basis function network identification of nonlinear systems, *IEEE Transactions on Neural Networks*, Vol.6(5), 1237-1244, 1995.

[29] P.J. Green and B.W. Silverman, *Nonparametric Regression and Generalized Linear Models*, Chapman and Hall, 1994.

[30] F. Gustafsson and H. Hjalmarsson, Twenty-one ML estimation for model selection, *Automatica*, Vol.31(10), 1377-1392, 1995.

[31] Hassibi B and D.G. Stork (1993). Second order derivatives for network pruning: Optimal brain surgeon. In Hanson *et al.* (eds) *Advances in Neural Information Processing Systems*, 164-171.

[32] Hassoun M.H., *Fundamentals of Artificial Neural Networks*, Bradford Book, MIT Press. 1995.

[33] S. Haykin, *Neural networks: A comprehensive foundation*, Macmillan College Publishing Company, Inc. 1994.

[34] Hertz J., A. Krogh and R.G. Palmer (1991), *Introduction to the theory of neural computation*, Addison Wesley.

[35] K. Hornik *et al.*, Multilayer feedforward network networks are universal approximators, *Neural Networks*, Vol.2(5), 359-366, 1989.

[36] Hwang J. and S. Kung, Parallel algorithms/ architectures for neural networks, *Journal of VLSI Signal Processing*, 1989.

[37] Iiguni Y., H. Sakai and H. Tokumaru (1992). A real-time learning algorithm for a multilayered neural network based on the extended Kalman filter, *IEEE Transactions on Signal Processing*, Vol.40(4), 959-966.

[38] Jang M. and K.Y. Yoo, Snake-like systolic array design for back-propagation algorithm, *Proceedings of PDCS'96*, Chicago, 348-351, 1996.

[39] L. Jin *et al.*, Absolute stability conditions for discrete-time recurrent neural networks, *IEEE Transactions on Neural Networks*, Vol.5(6), 954-964, 1994.

[40] L. Jin *et al.*, Approximation of discrete-time state-space trajectories using dynamic recurrent neural networks. *IEEE Transactions on Automatic Control,* Vol.40(7), 1266-1270, 1995.

[41] Jin L. and M.M. Gupta, Globally asymptotical stability of discrete-time analog neural network, *IEEE Transactions on Neural Networks*, Vol.7(4), 1024-1031, 1996.

[42] Johansen T.A. On Tikhonov regularization, bias and variance in nonlinear system identification, *Automatica*, Vol.33(3), 441-446, 1997.

[43] Johansson R. (1993) *System Modeling and Identification.* Prentice-Hall.

[44] Jun Y., C. Park and H. Lee, A new parallel array architecture design for neural network, *Proceedings of PDCS'96*, Chicago, 398-402, 1996.

[45] Kimura A., I. Arizono and H. Ohta. An improvement of a back propagation algorithm by extended Kalman filter and demand forecasting by layered neural networks. *Internationa Journal of System Science*, Vol.27(5), 473-482, 1996.

[46] G. Kitagawa and W. Gersch, A smoothness priors – state space modeling of time series with trend and seasonality. *Journal of the American Statistical Association*, Vol.79(June), 378-389, 1984.

[47] G. Kitagawa and W. Gersch, A smoothness priors time-varying AR coefficient modeling of nonstationary covariance time series, *IEEE Transactions on Automatic Control*, Vol.30(1), 48-56, 1985.

[48] Kollias S. and D. Anastassiou (1989). An adaptive least squares algorithm for the efficient training of artificial neural networks. *IEEE Transactions on Circuits and Systems*, Vol.36(8), 1092-1101.

[49] B. Kosko, *Neural Network and Fuzzy System*, Prentice Hall Inc., 1992.

[50] Ku C. and K. Lee (1995), Diagonal recurrent neural networks for dynamic systems control, *IEEE Transactions on Neural Networks*, Vol.6(1), 144-156.

[51] Kumar V., S. Shekhar and M. Amin, A scalable parallel formulation of the back-propagation algorithm for hypercubes and related architecture, *IEEE Transactions on Parallel and Distributed Systems*, Vol.5(10), 1073-1090, 1994.

[52] Kung S.Y. and W. Chou, Mapping neural networks onto VLSI array processors, in K.W. Przytula and V.K. Prasanna (eds) *Parallel Digital Implementations of Neural Networks*, Prentice Hall, 1993.

[53] Larsen J. and L.K. Hansen. Generalization performance of regularized neural network models. *Proc. IEEE Workshop on Neural Networks for Signal Processing* IV, 42-51, 1994.

[54] Larsen J. (1996). *Design of Neural Network Filters.* PhD Thesis, CONNECT, Department of Mathematical Modeling, Technical University of Denmark.

[55] LeCun Y. *et al.* (1990). Optimal brain damage, *Advances in Neural Information Processing Systems 2* (D.S. Touretsky, ed.) 396-404.

[56] Leung C.S., K.W Wong, J. Sum and L.W.Chan. (1996) On-line training and pruning for RLS algorithms. To appear in *Electronics Letter.*

[57] Leung C.S., P-F. Sum, A-C. Tsoi and L Chan, Several aspects of pruning methods in recursive least square algorithms for neural networks, *Theoretical Aspects of Neural Computation : A Multidisciplinary Perspective*, K. Wong *et al.* (eds.) Springer-Verlag, p.71-80, 1997.

[58] Leung J.Y.T., A survey of scheduling results for imprecise computation tasks, in S. Natarajan (eds) *Imprecise and Approximate Computation*, Kluwer Academic Publication, 1995.

[59] Lin, K-J., S. Nataragin and J. W-S. Liu, Imprecise results : Utilizing partial computations in real-time systems, *Proc. of the 8th Real-Time Systems Symposium*, San Franciso, CA, 1987.

[60] R. Lippman. An introduction to computing with neural nets, *IEEE ASSP Magazine*, Vol. 4, pp.4-22, 1987.

[61] Liu J., K. Lin, A.C. Yu, J. Chung and W. Zhao, Algorithms for scheduling imprecise computations, *IEEE Computer*, May, 1991.

[62] Lin T., C. Lee Giles, B.G. Horne and S.Y. Kung (1997), A delay damage model selection algorithm for NARX neural networks, Accepted for publication in *IEEE Transactions on Signal Processing*, Special Issue on "Neural Networks for Signal Processing".

[63] Lin W., V. Prasanna and K. Przytula, Algorithmic mapping of neural network models onto parallel SIMD machines, *IEEE Transactions on Computers*, Vol.40(12), 1390-1401, 1991.

[64] L. Ljung, Asymptotic behavior of the extended Kalman filter as a parameter estimator for linear systems, *IEEE Transactions on Automatic Control*, Vol.24(1), 36-50, 1979.

[65] L. Ljung and T. Soderstrom, *Theory and Practice of Recursive Identification*, MIT Press, 1983.

[66] L. Ljung, *System Identification: Theory for the user*, Prentice Hall Inc., 1987.

[67] Ljung L. J. Sjöberg and T. McKelvey. On the use of regularization in system identification. Technical Report of Dept. of Elec. Engg., Linkoping University, Sweden. 1992.

[68] Lo J., Synthetic approach to optimal filtering, *IEEE Transactions on Neural Networks*, Vol.5(5), 803-811, 1994.

[69] Luenberger D.G., *Introduction to linear and nonlinear programming*, Addison-Wesley, Reading, Mass., 1973.

[70] Mackay D.J.C. (1992), A Practical Bayesian Framework for Backprop Networks, *Neural Computation*, Vol.4(3) 448-472.

[71] Mackay D.J.C. (1995), Bayesian Methods for Neural Networks: Theory and Applications, Course notes for Neural Networks Summer School.

[72] Malluhi Q.M., M. Bayoumi and T. Rao, Efficient mapping of ANNs on hypercube massively parallel machines, *IEEE Transactions on Computers*, Vol.44(6), 769-779, 1995.

[73] Matthews M.B. and G.S. Moschytz (1990), Neural network non-linear adaptive filtering using the extended Kalman filter algorithm, *Proceedings of the INNC'90 Paris*, Vol.I, 115-119.

[74] Moody J.E. (1991), Note on generalization, regularization, and architecture selection in nonlinear learning systems, *First IEEE-SP Workshop on Neural Networks for Signal Processing.*

[75] Moody J. The effective number of parameters: An analysis of generalization regularization in nonlinear learning systems, *Advances in Neural Information Processing Systems 4*, 847-854, 1992.

[76] Moody J. (1994), Prediction risk and architecture selection for neural networks, in *From Statistics to Neural Networks: Theory and Pattern Recognition Application*, V.Cherkassky *et al.* (eds.), Springer-Verlag.

[77] Morgan N., The RAP: A ring array processor for layered network calculations, *Proc. Conference on Application Specific Array Processors*, 296-308, 1990.

[78] Mosca E. (1995), *Optimal Predictive and Adaptive Control.* Prentice Hall.

[79] Murata N., S. Yoshizawa and S. Amari. Network information criterion–Determining the number of hidden units for an artificial neural network model, *IEEE Transactions on Neural Networks*, Vol.5(6), pp.865-872, 1994.

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

[81] K.S. Narendra and K. Parthasarathy, Gradient methods for the optimization of dynamical systems containing neural networks, *IEEE Transactions on Neural Networks*, Vol.2(2), 252-262, 1991.

[82] Narendra K.S. and K. Parthasarathy (1992). Neural networks and dynamical systems. *International Journal of Approximate Reasoning*, Vol.6, 109-131, 1992.

[83] Natagajan S., *Imprecise and Approximate Computation*, Kluwer Academic Publisher, 1995.

[84] Olurotimi O., Recurrent neural network training with feedforward complexity, *IEEE Transactions on Neural Networks* Vol.5(2), 185-197, 1994.

[85] van Overbeek A.J.M. and L. Ljung, (1982). On-line structure selection for multivariable state-space models. *Automatica*, Vol.18(5), 529-543.

[86] B.A. Pearlmutter, Fast exact multiplication by the Hessian, *Neural Computation*, Vol.6, 147-160, 1994.

[87] Prechelt L. Comparing adaptive and non-adaptive connection pruning with pure early stopping. *Progress in Neural Information Processing*, 46-52, 1996.

[88] Prechelt L. (1997) Connection pruning with static and adaptive pruning schedules, in press in *Neurocomputing.*

[89] Puskorius G.V. and L.A. Feldkamp (1991), Decoupled extended Kalman filter training of feedforward layered networks, in *Proceedings of IJCNN'91*, Vol.I, 771-111.

[90] Puskorius G.V. and L.A. Feldkamp (1994), Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks, *IEEE Transactions on Neural Networks*, Vol.5(2), 279-297.

[91] Reed R. (1993), Pruning algorithms – A survey, *IEEE Transactions on Neural Networks*, Vol.4(5), 740-747.

[92] Rosenblatt F., *Principles of Neurodynamics : Perceptrons and the Theory of Brain Mechanisms.* Spartan Press. 1962.

[93] Ruck D.W., S.K. Rogers, M. Kabrisky, P.S. Maybeck and M.E. Oxley (1992), Comparative analysis of backpropagation and the extended Kalman filter for training multilayer perceptrons, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.14(6), 686-691.

[94] Rumelhart D.E., G.E. Hinton and R.J. Williams (1986). *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, Vol.1. MIT Press, Cambridge, Mass.

[95] Shah S., F. Palmeieri and M. Datum (1992), Optimal filtering algorithms for fast learning in feedforward neural networks, *Neural Networks*, Vol.5, 779-787.

[96] Siegelmann, Hava T., Bill G. Horne and C. Lee Giles, Computational capabilities of recurrent NARX neural networks, *IEEE Transactions on Systems, Man and Cybernetics – Part B: Cybernetics*, Vol. 27(2), 208, 1997.

[97] Singhal S. and L. Wu (1989). Training multilayer perceptrons with the extended Kalman algorithm, in *Advances in Neural Information Processing Systems I*, D.S.Touretzky Ed., 133-140.

[98] Sjöberg J. and L. Ljung (1995), Overtraining, regularization and searching for a minimum, with application to neural networks. *Int. J. Control*, **62**, 1391-1407.

[99] T. Soderstrom, *Discrete-Time Stochastic Systems: estimation and control.* Prentice Hall, 1994.

[100] Sontag E.D., Neural network for control. In *Essays on Control: Perspectives in the Theory and its Applications* (H.L. Trentelman and J.C. Willems, eds), 339-380, Birkhauser, Boston, 1993.

[101] Sontag E.D., Recurrent neural networks: Some systems-theoretic aspects. In Dealing with Complexity: a Neural Network Approach (M. Karny, K. Warwick, and V. Kurkova, eds.), Springer-Verlag, London, 1997, to appear.

[102] Soucek B., *Neural and Concurrent Real-Time systems*, John Wiley & Sons, Inc., 1989.

[103] Stankovic J. and K. Ramamritham, *Hard Real-Time Systems*, IEEE Computer Society Press, 1988.

[104] Sum J., C.S. Leung and L.W. Chan (1996) Extended Kalman filter in recurrent neural network training and pruning. *Technical report* **CS-TR-96-05**, Department of Computer Science and Engineering, The Chinese University of Hong Kong, June 1996.

[105] Sum J., C. Leung, L. Chan, W. Kan and G.H. Young, On the Kalman filtering method in neural network training and pruning, to appear in *IEEE Transactions on Neural Networks.*

[106] Sum J., C. Leung, L. Chan, W. Kan and G.H. Young, An adaptive Bayesian pruning for neural network in non-stationary environment, to appear in *Neural Computation.*

[107] Sum J. and L.W. Chan, On the approximation property of recurrent neural network. To appear in the *Proceedings of World Multiconference on Systemics, Cybernetics and Informatics*, Caracas, Venezuela July 7-11, 1997.

[108] Sum J., W. Kan and G.H. Young, Hypercube recurrent neural network, submitted.

[109] Sum J., W. Kan and G.H. Young, Note on some pruning algorithms for recurrent neural network, unpublished manuscript.

[110] Sum J., G.H. Young and W. Kan, Imprecise neural computation, to be presented in *International Conference in Theoretical Computer Science*, Hong Kong, 1998.

[111] Sum J., G.H. Young and W. Kan, Parallel algorithm for the realization of recursive least square based training and pruning using SIMD machine, submitted to *PDPTA'98* Les Vegas.

[112] John Sum, Gilbert H. Young and Wing-kay Kan, Imprecise Neural Computation in Real-Time Neural System Design, to be presented in *Workshop of Real-Time Programming, Shatou, China.*

[113] Suykens J., B. De Moor and J.Vandewalle (1995). Nonlinear system identification using neural state space models, applicable to robust control design. *International Journal of Control.* Vol.62(1), 129-152.

[114] Tikhonov A.N., Incorrect problems of linear algebra and a stable method for their solution, *Doklady* Vol.163(3), 988-991. 1965.

[115] van Tilborg A. and G. Koob, *Foundations of Real-Time Computing : Scheduling and Resource Management*, Kulwer Academic Publishers, 1991.

[116] Tresp V., R. Neuneier and H. Zimmermann (1996). Early brain damage. Presented in NIPS'96.

[117] A. Tsoi and A. Back, Locally recurrent globally feedforward networks: A critical review of architectures, *IEEE Transactions on Neural Networks*, Vol.5(2), 229-239, 1994.

[118] Wahba G., A survey of some smoothing problems and the method of generalized cross-validation for solving them, in P.R.Frishnaish (ed.) *Applications of Statistics*, North-Holland, pp.507-523, 1977.

[119] Watanabe K. T. Fukuda and S.G. Tzafestas. Learning algorithms of layered neural networks via extended Kalman filters. *International Journal of Systems Science*, Vol.22(4), 753-768, 1991.

[120] Wan E.A. (1993). *Finite Impulse Response Neural Networks with Applications in Time Series Prediction.* PhD Dissertation, Standford University, Nov.

[121] Wan E.A. and A.T. Nelson (1996). Dual Kalman filtering methods for nonlinear prediction, smoothing, and estimation. To appear in *NIPS'96.*

[122] Weigend A.S. *et al.* (1991), Generalization by weight-elimination applied to currency exchange rate prediction. *Proceeding of IJCNN'91*, Vol.I, 837-841.

[123] Williams R.J. (1992), Training recurrent networks using the extended Kalman filter, *Proceedings of the IJCNN'92 Baltimore*, Vol.IV, 241-246.

[124] Williams R.J. and D. Zipser (1989). A learning algorithm for continually running fully recurrent neural networks, *Neural Computation*, **1**(2), 270-280.

[125] With Pedersen M. and L.K. Hansen (1995), Recurrent networks: Second order properties and pruning, *Advances in Neural Information Processing Systems 7*, (G. Tesauro *et al.* ed.). 673-680, MIT Press.

[126] With Pedersen *et al.* (1996), Pruning with generalization based weight saliencies: γOBD and γOBS. To appear in *Advances in Neural Information Processing Systems 8*, edited by D.S. Touretzky *et al.*. MIT Press.

[127] With Pedersen M. *Optimization of Recurrent Neural Networks for Time Series Modeling*, P.D. Thesis, Department of Mathematical Modelling, Technical University of Denmark, 1997.

[128] Wu L. and J. Moody (1996), A smoothing regularizer for feedforward and recurrent neural networks, *Neural Computation*, **8**, 461-489.

[129] Yang B. and J.F. Bohme, CORDIC processor arrays for adaptive least squares algorithms, in R. Eckmiller *et al.* (eds), *Parallel Processing in Neural Systems and Computers*, Elsevier Science Publishers, 1990.

[130] E.F.Y. Young and L.W. Chan, Locally connected recurrent neural network, Department of Computer Science Technical Report, CUHK, 1995.