AN ABSTRACT OF THE THESIS OF

Dingguo Chen for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on September 30, 1998. Title: Nonlinear Neural Control with Power Systems Applications

Redacted for privacy

Abstract approved: _____

Ronald R. Mohler

Extensive studies have been undertaken on the transient stability of large interconnected power systems with flexible ac transmission systems (FACTS) devices installed. Varieties of control methodologies have been proposed to stabilize the postfault system which would otherwise eventually lose stability without a proper control. Generally speaking, regular transient stability is well understood, but the mechanism of load-driven voltage instability or voltage collapse has not been well understood. The interaction of generator dynamics and load dynamics makes synthesis of stabilizing controllers even more challenging.

There is currently increasing interest in the research of neural networks as identifiers and controllers for dealing with dynamic time-varying nonlinear systems. This study focuses on the development of novel artificial neural network architectures for identification and control with application to dynamic electric power systems so that the stability of the interconnected power systems, following large disturbances, and/or with the inclusion of uncertain loads, can be largely enhanced, and stable operations are guaranteed.

The latitudinal neural network architecture is proposed for the purpose of system identification. It may be used for identification of nonlinear static/dynamic loads, which can be further used for static/dynamic voltage stability analysis. The properties associated with this architecture are investigated.

A neural network methodology is proposed for dealing with load modeling and voltage stability analysis. Based on the neural network models of loads, voltage stability analysis evolves, and modal analysis is performed. Simulation results are also provided.

The transient stability problem is studied with consideration of load effects. The hierarchical neural control scheme is developed. Trajectory-following policy is used so that the hierarchical neural controller performs as almost well for non-nominal cases as they do for the nominal cases. The adaptive hierarchical neural control scheme is also proposed to deal with the time-varying nature of loads. Further, adaptive neural control, which is based on the on-line updating of the weights and biases of the neural networks, is studied. Simulations provided on the faulted power systems with unknown loads suggest that the proposed adaptive hierarchical neural control schemes should be useful for practical power applications.

Nonlinear Neural Control with Power Systems Applications

by

Dingguo Chen

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the

degree of
Doctor of Philosophy

Completed September 30, 1998
Commencement June 1999

Doctor of Philosophy thesis of Dingguo Chen presented on September 30, 1998

APPROVED:

Redacted for privacy 14 Oct-98

Major Professor, representing Electrical and Computer Engineering

Redacted for privacy

Chair of the Department of Electrical and Computer Engineering

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Dingguo Chen, Author

ACKNOWLEDEGMENT

The author would like to express his great gratitudes to his advisor, Prof. Ronald R. Mohler of Oregon State University, for his many helpful suggestions, much encouragement and guidance throughtout the course of this work.

Sincere thanks are also due to the other members of the author's committee, Prof. Wojtek Kolodziej, Prof. Rene Spee, Prof. Larry Chen, Dr. Yu Wang, Prof. Satish Reddy, Prof. Alexander Khapalov, and Prof. Dwight J. Bushnell.

The author is grateful primarily to the National Science Foundation for its support (Grant No. ECS9301168 and ECS9530917) as well as supplemental support from Bonneville Power Administration and Electric Power Research Institution. The support by Prof. David J. Hill of The University of Sydney, Australia is highly appreciated during the author's visit for collaborative research.

The author would also like to give his thanks to those who have helped in different ways at various times with both technical and non-technical issues.

Finally, the author would like to express his deep appreciation to his wife, Xiaohui Yang, for her patience, understanding, support and encouragement throughout the course of his doctoral work.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

NOMENCLATURE

Symbol	Description
C	Capacitor
D	damping factor
E	voltage source
H	Hamiltonian function
I	branch current
M	system inertia
P	real (or active) power
Q	reactive power
s	series TCSC compensation degree
t	time
u	control variable
V	bus voltage
v	control variable
X	reactance
x	state
Y	admittance

NOMENCLATURE (Continued)

Subscripts

Symbol	Description
a	Additional value
b	power system base for conversion to per unit system
d	load demand
e	equilibrium
f	final value
g	generator
i	vector index
m	machine
n	nominal value
t	terminal
0	initial value

NOMENCLATURE (Continued)

Greek Symbols

Symbol	Description
α	multiplier coefficient
δ	rotor speed
Δ	incremental value
∞	infinite bus
λ	costate
ω	rotor speed
Ψ	final-state constraint
τ	switching time

NONLINEAR NEURAL CONTROL WITH POWER SYSTEMS APPLICATIONS

1. INTRODUCTION

1.1. Introduction

The derivation of a mathematical model from physical laws according to the use of the model, such as for control, is most basic here to determine mathematical structure. For example, it is shown in [1] that a close connection exists between dynamic identification of an environment and its control. System identification itself is a well developed area of system theory. The need of mathematical representations in many aspects of the real world dictates the importance of system identification. In a way, it may be said that identification is a link between the mathematical-model world and the real world. For characterization of the cause and effect links of an observed plant, it is often assumed that the plant can be described by a model whose structure is known, or in other words, the plant is associated with a given form of parameterization, but the values of the parameters are assumed to be unknown. The parameters of the model are tuned in such a way that the behavior of the model approximates that of the plant. Differential equations, difference equations, and state-space representations are some examples of most widely used models. An excellent treatment of system identification in theory and applications may be found in [2, 3]. Note that identification may be categorized into off-line identification and on-line identification. The former one refers to a separate procedure by which a model (usually of given structure) is constructed based on a batch of data collected from the real system. The latter one refers to a procedure by which a model (again usually of given structure)

is constructed and updated based on the most recent available data collected from the system in operation. Off-line identification may be sufficient for time-invariant systems. The need for on-line identification is seen in cases where the properties of the observed object are time-varying. For control purposes, two different approaches exist. One is the so-called indirect control by which the control action is adjusted based on the on-line identification of the plant. The other is the so-called direct control by which the control action is adjusted to improve a performance index involving implicit identification. Note that for both approaches, efforts have to be made for identification of the behavior of the plant even when control action is being taken based on the most recent available information about the plant. In a way, it may be said that control and identification are inter-dependent, which was referred to as dual control [4].

No matter what kinds of identification and adaptive control schemes are used, the basic requirement is to keep the overall system stable. Stability is always an important issue for design of adaptive control. Stability analysis of adaptive systems is still quite difficult. In general, the analytic solution of dynamic nonlinear systems is usually impossible so that indeed, general results on adaptive control of nonlinear systems are very few. It is true though that adaptive control can be designed for some general dynamic nonlinear systems, for example, feedback linearizable nonlinear systems at least in theory. On the other hand, adaptive control of linear systems was even an extensive research subject, and numerous results are available. An attempt to present a unified framework of the currently well known results for stable adaptive linear systems is made in [5]. Adaptive stabilization of nonlinear systems is overviewed in [6] where the nominal control explicitly expressed in terms of parameters is assumed. It is noted that either available results for dynamic linear systems are not adequate for real nonlinear systems or general results for dynamic nonlinear systems which are very useful are scarce. A simple class of nonlinear systems, bilinear systems which are linear in state and control but are jointly nonlinear, possess convenient structure properties, and hence make mathematical treatments possible. It is

illustrated in [7, 8, 9] that many real nonlinear systems cound be treated approximately as bilinear systems, and the control design procedures and stability analysis theories developed have played crucial roles in designing proper controls for these systems. It is shown in [7] that controllable linear systems may not be controllable with physical constraints on control while the controllability of bilinear systems of the same order could be achieved. Roughly speaking, bilinear systems are more controllable than linear systems. In many practical problems, theory of bilinear systems has found its succeful applications [9]. Recently, interest in application of bilinear system theory to power sytems was observed. It is shown in [10] that the transmission network of power systems when controlled by a variable series capacitor, (a simplified model for the thyristor-controlled series capacitor (TCSC), one kind of the popularly used flexible ac transmission systems (FACTS) devices), can be modeled as a bilinear system. Further, it has been demonstrated [11] that bilinear system models offer better approximation to the nonlinear dynamics than their linear counterparts, and moreover, that postfault power sytems which may not be stabilized via linear control could be stabilized via bilinear adaptive control. In general, it is the common consensus now that bilinear control (or more generally multiplicative control) offers a lareger horizon of stable operation of the power systems than linear control does, but of course, when the systems approach a small neighborhood of the equilibrium, linear control, which can provide better damping and local asymptotic stabilization, should take over. The idea behind bilinear control is further developed and applied to stabilization of power sytems in [12] where the total control is a weigthed sum of a number of pre-designed nominal controls. Each nominal control can conduct the system state to the system equilibrium in some "optimal" sense for a corresponding specific case. In practice, analytic forms of nominal optimal controls may not be available. Instead, through use of computational techniques [7, 13], optimal controls and optimal trajectories can be calculated, which in turn are used to train a neural controller. The weights (or called multipliers) are also obtained with another neural network yet trained with available measurements.

This leads to adaptive neural control structure, which may be applied to stabilize faulted power systems.

Aiming at the same problem, stabilization of power systems, yet with consideration of load modeling, identification and control issues, relevant to nonlinearly coupled bilinear systems, are studied in this thesis. Some aspects of identification and control in dynamic power systems are studied in this thesis. Brief description of these aspects and useful tools for dealing with identification and control are presented next.

1.2. Identification and control issue of dynamic power systems

Due to increasing electric power demand, different groups of machines are interconnected through tie-lines, and varieties of loads with various kinds of characteristics as well as a lot of protective equiment are connected to large electric transmission systems at different locations, resulting in extremely complex nonlinear dynamics. Therefore, planning, operation and control design of such systems become increasingly important.

Under ideal operational conditions, all the generators should keep synchronism. In other words, the loads should be fed at constant a.c. voltage, and fixed frequency all the time. Therefore, the variations of both voltage and frequency should be kept so small that the related equipment can normally operate at design performance. This is usually associated with the dynamic stability (or steady-state stability), which is concerned with the stability of synchronous machines under the condition of small-disturbances. Normally, synchronous machines can keep in step in some degree with the synchronizing force. Situations, however, do arise in which the synchronizing force for some machines may not be adequate such that they fall out of step, and small disturbances may cause them to lose synchronism. Fortunately, this could be handled by traditioal Power system

stabilizers (PSS), etc. with linear control design to enhance the damping of power systems. The linear control design thereof is on the basis of the linearized system model around the desired equilibrium.

While the machines may return to their original state under samll perturbations with no net power change involved, the large faults occuring in power systems may create an unbalance between the supply and demand, and thus cause the power systems to experience oscillatory transient dynamics. The oscillations are reflected as fluctuations in the power flow over the transmission lines.

The system equations for a transient stability study are usually nonlinear. The dynamics of a simple machine is characterized by the familiar swing equation:

$$\begin{cases} \dot{\delta} = \omega_b(\omega - 1) \\ \dot{\omega} = \frac{1}{2H\omega_b}(P_m - P_e - D\omega) \end{cases} \tag{1.1}$$

where δ and ω are the rotor angle and speed, both of which are the system states. P_m is the prime-mover power, P_e is the electrical power and D is the damping constant. The rotor angle is measured with respect to a synchronously rotating reference.

When there are many machines in a large interconnected power system, the above swing equation may be modified with one machine as reference, and with the relative rotor angle and speed as the state of each machine other than the reference machine.

It is observed that P_e is dependent on the network structure, and bus voltages (magnitudes and phases) as well as the loads. The nonlinearity is thus introduced. The effects of control devices are also reflected in the flow of P_e. The flow of P_e is usually associated with the supply of rective power. These observations have brought increasing interests in the use of flexible ac transmission system (FACTS) devices for purposes of increasing the power transfer capability of the transmission system and enhancing transient stability. The commonly used FACTS devices (to name a few, include thyristor-controlled series capacitors, thyristor-controlled resistors, and static var-compensators) allow for rapid manipulation of the network impedances, and affect the power flow of the systems. Mathe-

matical modeling of FACTS devices itself is a difficult task, and is not addressed in this thesis. Instead, the FACTS devices are assumed to be equivalent variable capacitors, or variable resistors, or whatsoever, which permits convenient mathematical manipulation for control purpose, and whose practical implementation is not considered.

Extensive studies have been made on the transient stability of large interconnected power systems with FACTS devices installed. Many kinds of control methodologies (to cite a few, nonlinear adaptive control, variable structure control, optimal control and artificial neural network control) have been proposed to stabilize the postfault system which would otherwise eventually lose stability without proper control.

Note that for classical transient stability study, loads are usually assumed to be either constant power consumer, or constant impedance, or at worst constant current source. The dynamics of loads are usually ignored for avoiding complexity. Since several major system failures [14] have resulted from load side voltage instability and collapse, load-driven voltage stability has now become a major concern in planning and operating electric power systems.

Load-driven stability is mainly concerned with the stability caused by load dynamics, big load build-up, etc.. Therefore, modeling of loads (including static modeling of loads and dynamic modeling of loads) is an important issue and will be studied in chapter 4.

From the viewpoint of control design, modeling of each and every component of loads in a load center is neither necessary nor practical. An aggregate load model is usually developed for power flow and transient stability study [15]. Note that in the literature voltage stability study is usually based on a static load model. As is known, the dynamics of loads play an important role in the voltage instability problems, which needs in-depth investigations. The voltage stability study and control design should include the consideration of load dynamics. This important issue on the understanding of voltage collapse mechanisms will be part of this work. Loads identification and its inclusion in the stability study and control design will be presented in chapters 4 to 6.

1.3. Neural networks as identifiers and controllers

It is observed from the discussions in the previous sections that mathematical models for systems in question (for instance, loads and power systems) are needed for design purposes in order to achieve desired performances. For these problems in discussion, analytic mathematical models are not available, and large amounts of data on the system behaviors are available. For conventional identification approaches, linear or specific nonlinear structure forms are assumed. This may be helpful in dealing with specific problems with known properties. For the addressed problems with random characters, conventional identification approaches may not be sufficient. For the problems addressed in the context of stabilization of postfault power systems, conventional, analytic methods may not yield satisfactory solutions, since either no accurate analytic model is availabe or the existing model is too complicated for use in synthesis of controllers. Thus, there is a need for novel and effective identification and control schemes. This has led to the exploration of the use of artificial neural networks (or simply neural networks). It has been shown that neural networks possess certain universal approximation properties which allow their use as identifiers and controllers for a large class of nonlinear dynamical systems.

The distributed structure of neural networks allows fast parallel computation. In addition, this kind of structure enables neural networks to perform robustly even in presence of disturbances. Due to neural networks' nice properties, they have been used in many diverse real-world applications, to cite a few, optimization [16], ill-posed inverse problem [17], image compression [18], handwritten signature recognition [19], classification [20], modeling and identification [21], and neural control [22].

Originally, artificial neural network research was motivated by the effort to model biological neurons and neural systems. It was McCulloch and Pitts in the 1940s who first represented the neuron with a methematical model [23]. The introduction of Hebbian

rule [24] makes possible the proper changing of the synaptic weights of the neuron. Rosenblatt's perceptron [25], Widrow's adaptive linear element [26], etc. aroused widespread enthusiasm about artificial neural networks. The publication of an important book [27], with exposure of serious theoretical limitations of perceptrons and in particular the pessimistic conclusions, almost gave an end to the then neural network research. The revival of neural network research is largely attributed to several researchers' famous works, such as Grossberg's ART [28], Kohonen's self-organizing mapping [29], Rumelhart's backpropagation training algorithm [30] etc.

There is currently increasing interest in the research of neural networks as identifiers and controllers for dealing with dynamic time-varying nonlinear systems.

1.4. The goals of this thesis

This thesis is mainly devoted to the theoretical aspects of neural networks and their applications as identifiers and controllers in dynamic nonlinear systems. In particular, the applications are confined to dynamic modeling and control in dynamic power systems. To be speicifc, this thesis will develop neural-net-based control design methodologies to deal with power system stability problems involving both generator dynamics and load dynamics since either generator dynamics or load dynamics, but not both, is usually considered in the literature for stability concern. This in turn leads to the tasks to be performed, namely, load modeling, transient stability study, load-side stability study, neural control design, adaptive neural control design, and stabilization of power systems which is likely to experience transient instability problems and/or voltage instability problems.

The thesis is organized as follows:

The background material on neural networks is provided in chapter 2. The backpropagation algorithm, together with its derived versions for training recurrent neural net-

works and locally recurrent neural networks, is discussed and is represented in a compact matrix-format. This is for convenience of software implementation of the backpropagation algorithm.

Chapter 3 presents some proposed neural network architectures. The proposed latitudinal neural network architectures are studied in detail. Relevent properties are further investigated.

The voltage stability of electric-power systems is discussed in chapter 4. Since voltage stability is normally associated with the load dynamics, the load modeling issue is presented first. Then use of neural networks for load modeling is addressed. Further, with the neural-network-based load model, static and dynamic voltage stability analyses are provided.

In chapter 5, the synthesis of intelligent neural controllers is addressed. First of all, the approximation of a switching manifold by a neural network is discussed. Based on such a discussion, a novel pattern recognition scheme for time-optimal control is proposed. Then a hierarchical, neural-network, control structure is proposed. Further, adaptive neural-network control is presented. These neural control schemes are justified by mathematical verification. Simulation results are presented to show the effectiveness of the proposed neural control schemes.

In chapter 6, the stabilization of multi-machine systems is addressed, together with the inclusion of dynamic load modeling by a neural network. First, the strategy developed in chapter 5 is used to stabilize the postfault multi-machine system which is represented by a set of generalized bilinear differential equaitions under some assumptions. Then the adaptive neural control is discussed. Further a control scheme is proposed to stabilize the mulit-machine systems and keep a good profile of load side voltage aiming at the study on the mechanism of voltage collapse.

Chapter 7 reviews the main contributions of this dissertation, presents the concluding remarks, and suggests future research.

2. MULTI-LAYER NEURAL NETWORK

2.1. Introduction

2.1.1. Object

This chapter is intended to present a quick review of neural networks, an overview of ongoing research topics on neural networks, and the implementation issue. Some standard materials are covered, which may be useful either for further developments in later chapters or for the interpretations of the implemented software. In addition, this chapter intentionally provides a unified compact matrix format for the backpropagation algorithms.

2.1.2. Background

The past decade has witnessed increasing interest for the use of neural networks in the identification and control of nonlinear dynamic systems. Early applications of neural networks are found to be primarily in the area of pattern recognition and classification. Function approximation by neural networks was one of the then major research subjects. These theoretical studies have laid the foundations for neural networks as a well-established discipline. Since multilayer feedfoward neural networks represent static nonlinear mappings, it was suggested in [31] that for use of neural networks for modeling and identification of dynamical systems, these neural networks have to be modified by addition of feedback connections, resulting in the so-called recurrent neural networks. Since then a tremendous growth of research and development on this subject has resulted in numerous publications. Dynamic backpropagation [32] was proposed in order to train a recurrent neural network for approximation of the system dynamics. For demonstration of use of neural networks for identification and control of dynamical systems, extensive

simulation results on identification and control and theoretical studies on controllability, observability and stability have been reported in the literature. More recently, use of locally recurrent neural networks was proposed in [33] for emulating a large class of non-linear dynamic systems. It is believed that recurrent neural networks, or a similar adaptive architecture, will be increasingly used in dealing with control design of dynamical systems in the future. It is observed, however, that neural network training based on dynamic backpropagation is intensively time-consuming, which makes impossible on-line training of such neural networks.

2.2. Feedforward neural network

It is well known that multilayer feedforward neural networks have been intended to pattern recognition and classification applications. The use of feedforward neural networks for representation of static mappings is also known. These successful applications of feedforward neural networks are mainly due to their ability to approximate a certain class of functions. It has been shown that any continuous functions with compact support can be approximated arbitrarily well by a one-hidden-layer feedforward neural network for the activation function being either sigmoidal ones [34] or radial basis functions [35]. In addition, the distributed structure of neural networks allows fast parallel computation. With proper choices of activation functions, training process can proceed conveniently for the relevant partial derivatives necessary for adjustment of weights and biases can be computed and propagated backward layer by layer whilst the outputs of each layer can be calculated easily and propagated forward. This is essentially the well-known backpropagation algorithm [30], though the very idea behind this algorithm is originated in [36]. It may be said that the appealing structural features of neural networks which allow convenient software and hardward implementations and the availability of convenient training

algorithms have made their extensive application possible. Also note that the backpropagation algorithm is merely a simple gradient method and that other methods, such as conjugate gradient, are sometimes more effective as noted in section 2.3..

2.2.1. Structure of feedforward neural networks

A feedforward neural network is composed of a number of layers, each of which in turn consists of a number of neurons. There are only connections between the neurons in one layer and its next layer (if it exists), and there are normally no connections between neurons within the same layer. Usually all neurons in the same layer have identical structure except that relevant parameters (including the connection weights and biases) take different values. The correspondence between the inputs and outputs of each layer may be viewed as a mapping from the input space to the output space. Thus, a feedforward neural network may be viewed as a composition of a number of mappings—a nonlinear finite-dimensional mapping from the input space to the output space for this neural network. In a way it may be said that a feedforward neural network realizes a parametrized nonlinear mapping. Based on this understanding, a feedforward neural network may be used to approximate, with a proper choice of the number of layers and the number of neurons in each hidden layer, some nonlinear functions with the approximation error in some sense dependent on the adaptation of the relevant parameters. The parameter optimization resulting in the optimal value of a chosen performance criterion may proceed with an iterative search guided by a learning algorithm in the training process.

2.2.1.1. A neuron: Information processing cell

An artificial neuron is characterized by its synaptic connections with connection weights w_i's, its activation function $\sigma(.)$ and the threshold b. The input signals x_i's that

a nueron receives through its corresponding synapses with connection weights w_i's are multiplied by the connection weights and summed to yield the activation of the neuron. The activation in turn produces the output y of the neuron, which can be given by

$$y = \sigma(\sum_i w_i x_i + b) = \sigma(wx + b) \tag{2.1}$$

where $w = \begin{bmatrix} w_1 & w_2 & \cdots & w_n \end{bmatrix}^\tau$ with n as the number of connections;

and $x = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^\tau$. τ designates transpose operation.

Mathematically, a neuron actually realizes a function $f : R^n \longrightarrow R$, where n is the number of the inputs.

The activation function is usually a logistic function $\sigma_l(x) = \frac{1}{1+e^{-x}}$, a tan-sigmoidal function $\sigma_t(x) = \frac{1-e^{-x}}{1+e^{-x}}$, or a similarly saturating function.

2.2.1.2. Sigmoidal functions

The activation functions that are commonly used are a special form of the so-called sigmoidal function.

A function $\sigma : R \to R$ is called sigmoidal, if it is nondecreasing and bounded, i.e., $\lim_{x\to+\infty} \sigma(x) < +\infty$, and $\lim_{x\to-\infty} \sigma(x) > -\infty$.

Besides the logistic and tan-sigmoidal activation functions, several other kinds of sigmoidal or squashing functions are defined in the following.

- Heaviside function

$$\sigma_h(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

- Soft squashing function

$$\sigma_s(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } x \in [0, 1] \\ 0 & \text{if } x < 0 \end{cases}$$

- Quadratic squashing function

$$\sigma_q(x) = \begin{cases} 1 & \text{if } x > 1 \\ x^2 & \text{if } x \in [0,1] \\ 0 & \text{if } x < 0 \end{cases}$$

2.2.1.3. Notations

An opterator $\circ$ is defined as follows: $C = A \circ B$ if for $A \in R^{m\times n}$, $B \in R^{m\times n}$, and $C \in R^{m\times n}$, $c_{ij} = a_{ij}b_{ij}$, where a_{ij}, b_{ij}, and c_{ij} are the elements in the intersection of row i and column j of A, B, and C, respectively.

$\underline{f}[A]$ is defined as

$$\underline{f}[A] = \begin{bmatrix} f(a_{11}) & f(a_{12}) & \cdots & f(a_{1n}) \\ f(a_{21}) & f(a_{22}) & \cdots & f(a_{2n}) \\ \vdots & \vdots & \vdots & \vdots \\ f(a_{m1}) & f(a_{m2}) & \cdots & f(a_{mn}) \end{bmatrix} \tag{2.2}$$

where $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \in R^{m\times n}$, and f is a scalar function.

2.2.2. Feedforward neural network composition

A feedforward neural network with L layers consists of the input layer, layer 0, of d_0 inputs, the output layer, layer L, of d_L outputs, and hidden layers with d_l neurons for each layer $l = 1, 2, \cdots, L$.

In general, the activation function for each neuron of a layer is assumed to be the same as for all others. Let the activation functions for the lth layer be σ^l for $l = 1, 2, \cdots, L$.

All the neurons in each layer are numbered from 1. A neuron n in layer l is connected to all the neurons in layer $l-1$ through d_{l-1} connections, each one associated with a weight $w_{n,j}^l$ where $j = 1, 2, \cdots, d_{l-1}$. The threshold or bias for this neuron is b_n^l. It is observed that layer l actually realizes the following function vector:
$F^l : R^{d_{l-1}} \longrightarrow R^{d_l}$, where the ith component is given by $F_i^l(x^{l-1}) = \sigma^l(\sum_{j=1}^{d_{l-1}} w_{i,j}^l x_j^{l-1} + b_i^l)$ with $x^{l-1} \in R^{d_{l-1}}$ and x_j^{l-1} is the jth component of the vector x^{l-1}, the output vector of layer $l-1$.

Therefore, a feedforward neural network with L layers results in the following compound mapping from $R^{d_0} \longrightarrow R^{d_L}$:

$$F(x^0) = (F^L \cdots F^1)(x^0) \tag{2.3}$$

For brevity, the neural network structure discussed above will be represented hereafter by $N_{\sigma_1,\sigma_2,\cdots,\sigma_L}^{d_0,d_1,\cdots,d_L}$.

2.2.3. Approximation capability

Function approximation theory in terms of neural networks has been studied extensively. Under some mild assumptions, neural networks may be used to approximate a large class of functions. To make the description simple and convenient in the following, a neural network and the function it realizes are used exchangeablely if there is no confusion arising.

It has been shown in [37] that for any continuous function f with a compact support Ω, there exists a neural network N, which approximates f arbitrarily close in L_2 sense. That is,

$$\int_\Omega ||f(x) - N(x)||^2 dx < \epsilon \tag{2.4}$$

where ϵ is a pre-specified positive number.

Moreover, it has been shown in Funahashi [34] that for a continuous function f with a compact support Ω, there exists a neural network N, which approximates f arbitarily closely in uniform topology. That is,

$$\sup_{x \in \Omega} ||f(x) - N(x)|| < \epsilon \tag{2.5}$$

More results about the approximation ability of neural networks can be found in [38, 39, 40]. Most results are for three-layered neural networks with sigmoidal activation functions used in the hidden layer. One fundamental result states that such neural networks can approximate any continuous or other kinds of functions defined on compact sets in R^n.

2.2.4. Backpropagation algorithms

Suppose the training data contain a number of patterns $s(i)$ and their corresponding targets $t(i)$ for $i = 1, 2, \cdots, P$ where P is the number of patterns and $s(i) \in R^M$, $t(i) \in R^N$. Since a neural network with m neurons in the input layer and n neurons in the output layer can be viewed as a parameterized function $F : R^M \longrightarrow R^N$. This function can be explicitly expressed as $F(., \theta)$. Note that the first argument is the input vector, and the second argument designates the parameter vector θ, composed of all the weights and biases. Once the structure of a neural network is chosen, i.e., the number of layers, the number of neurons in each layer, and the type of the activation function for each layer are specified, the remaining task is to solve the following parameter optimization problem: Find the optimal parameter vector θ^* such that a performance index $J(\theta)$ is minimized (or maximized).

Often, it is of computational advantage to use a quadratic performance index. With $s^l(i)$ designating the output of layer l ($l = 1, 2, \cdots, L$) of a neuron network $N^{d_0, d_1, \cdots, d_L}_{\sigma^1, \sigma^2, \cdots, \sigma^L}$ which is used to fit the given training data, a quadratic performance index can be ex-

pressed as $J(\theta) = \sum_{i=1}^{P} \frac{1}{2}||t(i) - s^L(i)||^2 = \sum_{i=1}^{P} \sum_{j=1}^{d_L} (t_j(i) - s_j^L(i))^2$ where the subscript j designates the jth component of a vector, and $t(i) - s^L(i)$ is the error between the actual output of the neural network and the desired ouput.

To obtain the optimal θ^*, the computation of the gradient of J with respect to θ is needed for application of a gradient-based numerical technique.

Suppose that activation functions $\sigma^l(.)$ for $l = 1, 2, \cdots, L$ are chosen properly here such that their derivatives are functions of themselves, that is, $\frac{d\sigma^l(x)}{dx} = g^l(\sigma^l(x))$.

If layer l is the output layer, that is, $l = L$, the partial derivatives can be obtained as follows.

$$\begin{aligned} \frac{\partial J}{\partial w_{n,k}^L} &= -\sum_{i=1}^{P} (t_n(i) - s_n^L(i)) \frac{\partial s_n^L(i)}{\partial w_{n,k}^L} \\ &= -\sum_{i=1}^{P} (t_n(i) - s_n^L(i)) g^L(s_n^L(i)) s_k^{L-1}(i) \end{aligned}$$

Define $\delta_n^L(i) = (t_n(i) - s_n^L(i)) g^L(s_n^L(i))$. Then

$$\frac{\partial J}{\partial w_{n,k}^L} = -\sum_{i=1}^{P} \delta_n^L(i) s_k^{L-1}(i) \tag{2.6}$$

If layer l is one of the hidden layer, then the chain rule is used to compute the partial derivatives.

$$\begin{aligned} \frac{\partial J}{\partial w_{n,k}^l} &= -\sum_{i=1}^{P} \sum_{j=1}^{d_L} (t_j(i) - s_j^L(i)) \frac{\partial s_j^L(i)}{\partial w_{n,k}^l} \\ &= -\sum_{i=1}^{P} \sum_{j=1}^{d_L} (t_j(i) - s_j^L(i)) \frac{\partial s_j^L(i)}{\partial s_n^l(i)} \frac{\partial s_n^l(i)}{\partial w_{n,k}^l} \\ &= -\sum_{i=1}^{P} \sum_{m=1}^{d_{l+1}} \sum_{j=1}^{d_L} (t_j(i) - s_j^L(i)) \frac{\partial s_j^L(i)}{\partial s_m^{l+1}(i)} \frac{\partial s_m^{l+1}(i)}{\partial s_n^l(i)} \frac{\partial s_n^l(i)}{\partial w_{n,k}^l} \end{aligned}$$

Since $\frac{\partial s_n^l(i)}{\partial w_{n,k}^l} = g^l(s_n^l(i)) s_k^{l-1}(i)$,

$$\frac{\partial J}{\partial w_{n,k}^l} = -\sum_{i=1}^{P} \sum_{j=1}^{d_L} (t_j(i) - s_j^L(i)) \frac{\partial s_j^L(i)}{\partial s_n^l(i)} \frac{\partial s_n^l(i)}{\partial w_{n,k}^l}$$

$$= -\sum_{i=1}^{P}\sum_{j=1}^{d_L}(t_j(i) - s_j^L(i))\frac{\partial s_j^L(i)}{\partial s_n^l(i)} g^l(s_n^l(i)) s_k^{l-1}(i)$$

$$= -\sum_{i=1}^{P}\delta_n^l(i) s_k^{l-1}(i) \tag{2.7}$$

where $\delta_n^l(i)$ is defined as $\delta_n^l(i) = \sum_{j=1}^{d_L}(t_j(i) - s_j^L(i))\frac{\partial s_j^L(i)}{\partial s_n^l(i)} g^l(s_n^l(i))$.

Since $\frac{\partial s_m^{l+1}(i)}{\partial s_n^l(i)} = g^{l+1}(s_m^{l+1}(i)) w_{m,n}^{l+1}$,

$$\frac{\partial J}{\partial w_{n,k}^l} = -\sum_{i=1}^{P}\sum_{m=1}^{d_{l+1}}\sum_{j=1}^{d_L}(t_j(i) - s_j^L(i))\frac{\partial s_j^L(i)}{\partial s_m^{l+1}(i)}\frac{\partial s_m^{l+1}(i)}{\partial s_n^l(i)}\frac{\partial s_n^l(i)}{\partial w_{n,k}^l}$$

$$= -\sum_{i=1}^{P}\sum_{m=1}^{d_{l+1}}\sum_{j=1}^{d_L}(t_j(i) - s_j^L(i))\frac{\partial s_j^L(i)}{\partial s_m^{l+1}(i)} g^{l+1}(s_m^{l+1}(i)) w_{m,n}^{l+1} g^l(s_n^l(i)) s_k^{l-1}(i) \tag{2.8}$$

Therefore, $\delta_n^l(i)$ may also be written as

$$\delta_n^l(i) = \sum_{m=1}^{d_{l+1}}\sum_{j=1}^{d_L}(t_j(i) - s_j^L(i))\frac{\partial s_j^L(i)}{\partial s_m^{l+1}(i)} g^{l+1}(s_m^{l+1}(i)) w_{m,n}^{l+1} g^l(s_n^l(i)) \tag{2.9}$$

Since by the definition of $\delta_n^l(i)$, $\delta_n^{l+1}(i)$ may be written as $\delta_n^{l+1}(i) = \sum_{j=1}^{d_L}(t_j(i) - s_j^L(i))\frac{\partial s_j^L(i)}{\partial s_n^{l+1}(i)} g^{l+1}(s_n^{l+1}(i))$. Therefore,

$$\delta_n^l(i) = \sum_{m=1}^{d_{l+1}}\delta_m^{l+1}(i) w_{m,n}^{l+1} g^l(s_n^l(i)) \tag{2.10}$$

Observations from equations (2.7) and (2.10) indicate that the complete determination of the gradient requires two kinds of information from two phases: feedforward phase and error backpropagation phase. To be more specific, the outputs of each layer can be computed layer by layer forward, starting from the input layer, and the error associated terms can be computed layer by layer backward, starting from the output layer. This is the main reason why this algorithm is called the backpropagation algorithm.

By defining the following matrices, the backpropagation algorithm can be written in a compact matrix format.

Define the input matrix

$$S = \begin{bmatrix} s_1(1) & s_1(2) & \cdots & s_1(P) \\ s_2(1) & s_2(2) & \cdots & s_2(P) \\ \vdots & \vdots & \vdots & \vdots \\ s_{d_0}(1) & s_{d_0}(2) & \cdots & s_{d_0}(P) \end{bmatrix},$$

the desired output matrix

$$T = \begin{bmatrix} t_1(1) & t_1(2) & \cdots & t_1(P) \\ t_2(1) & t_2(2) & \cdots & t_2(P) \\ \vdots & \vdots & \vdots & \vdots \\ t_{d_L}(1) & t_{d_L}(2) & \cdots & t_{d_L}(P) \end{bmatrix},$$

the output matrix for layer $l = 1, 2, \cdots, L$ corresponding to all the input patterns

$$S^l = \begin{bmatrix} s_1^l(1) & s_1^l(2) & \cdots & s_1^l(P) \\ s_2^l(1) & s_2^l(2) & \cdots & s_2^l(P) \\ \vdots & \vdots & \vdots & \vdots \\ s_{d_l}^l(1) & s_{d_l}^l(2) & \cdots & s_{d_l}^l(P) \end{bmatrix},$$

the weight matrix for layer $l = 1, 2, \cdots, L$

$$W^l = \begin{bmatrix} w_{1,1}^l & w_{1,2}^l & \cdots & w_{1,d_{l-1}}^l \\ w_{2,1}^l & w_{2,2}^l & \cdots & w_{2,d_{l-1}}^l \\ \vdots & \vdots & \vdots & \vdots \\ w_{d_l,1}^l & w_{d_l,2}^l & \cdots & w_{d_l,d_{l-1}}^l \end{bmatrix},$$

and the backpropagation error matrix for layer $l = 1, 2, \cdots, L$

$$\Delta^l = \begin{bmatrix} \delta_1^l(1) & \delta_1^l(2) & \cdots & \delta_1^l(P) \\ \delta_2^l(1) & \delta_2^l(2) & \cdots & \delta_2^l(P) \\ \vdots & \vdots & \vdots & \vdots \\ \delta_{d_l}^l(1) & \delta_{d_l}^l(2) & \cdots & \delta_{d_l}^l(P) \end{bmatrix}.$$

The backpropagation algorithm may be represented as the Backpropagation Algorithm I:

1. Forward phase

   $S^l = \underline{\sigma^l}[W^l S^{l-1} + B^l]$.

2. Error backpropagation

   $\Delta^L = (T - S^L) \circ \underline{g^L}[S^L]$.

   $\Delta^l = ((W^{l+1})^\tau \Delta^{l+1}) \circ \underline{g^l}[S^l]$.

3. Gradient computation and weight updating

   $\nabla_{W^l} J = -\Delta^l (S^{l-1})^\tau$.

   $W^l \longleftarrow W^l + \eta \Delta^l (S^{l-1})^\tau$.

   where $l = 1, 2, \cdots, L$; $S^0 = S$; and $B^l = \begin{bmatrix} b_1^l \\ \vdots \\ b_{d_l}^l \end{bmatrix} \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix} \in R^{d_l \times P}$;

   and η is a positive number.

Of course, the updating formula for all the biases can also be obtained directly. Alternatively, biases may be treated as weights so that the updating formula for weights and biases can be unified.

The mathematical model of an artificial neuron can be modified by associating its bias with a constant input signal 1 so that all the parameters (that is, the weights and the bias) are treated uniformly as weights. This leads to another representation of a neuron model (2.1) as

$$y = \sigma(\sum_i \hat{w}_i \hat{x}_i) \tag{2.11}$$

where $\hat{w} = \begin{bmatrix} w & b \end{bmatrix}^\tau$; and $\hat{x} = \begin{bmatrix} x & 1 \end{bmatrix}^\tau$.

The same can be done to all other neurons in a neural network. Then the orginal input signals are augmented by a constant input signal 1 to form the new inputs to the first hidden layer. The original outputs of each hidden layer (except the last hidden layer) are augmented by a constant input signal 1 to form the new inputs to the next hidden layer. Accordingly, the augmented input matrix $\hat{S}$, output matrix $\hat{S}^l$ of each layer $l = 1, 2, \cdots, L$ with $\hat{S}^L = S^L$, and weight matrix $\hat{W}^l$ for each layer $l = 1, 2, \cdots, L$ can be represented as

$\hat{S} = \begin{bmatrix} S \\ S_a \end{bmatrix}$ where S_a is a matrix of $1 \times P$ dimension with each entry being 1.

$\hat{S}^l = \begin{bmatrix} S^l \\ S_a \end{bmatrix}$ with $l = 0, 1, \cdots, L-1$. Note that $\hat{S}^0 = \hat{S}$.

$\hat{W}^l = \begin{bmatrix} W^l & b^l \end{bmatrix}$ where $b^l = \begin{bmatrix} b_1^l & b_2^l & \cdots & b_{d_l}^l \end{bmatrix}^\tau$, and $l = 1, 2, \cdots, L$.

Therefore, the backpropagation algorithm to deal with weights and biases uniformly may be represented as the Backpropagation Algorithm II:

1. Forward phase

   $\hat{S}^l = \underline{\sigma^l}[\hat{W}^l \hat{S^{l-1}}]$.

2. Error backpropagation

   $\Delta^L = (T - S^L) \circ \underline{g^L}[S^L]$.

   $\Delta^l = ((W^{l+1})^\tau \Delta^{l+1}) \circ \underline{g^l}[\hat{S}^l]$.

3. Gradient computation and weight updating

   $\nabla_{\hat{W}^l} J = -\Delta^l (\hat{S^{l-1}})^\tau$.

   $\hat{W}^l \longleftarrow \hat{W}^l + \eta \Delta^l (\hat{S^{l-1}})^\tau$.

It should be noted that updating of weights above is based on the deepest descent algorithm, which is known to be one of the slowest algorithms. It is true that backpropagation with a constant step may be very sensitive with respect to the size of the step and

may even fail to converge at all. The reasons for its poor performance might be, among others, that a constant step, if it is too large, may even increase the error to be minimized, and local minima of the error surface may make the iteration process fail to approach the global minimum.

An obvious way to improve the performance of the deepest descent method is to adjust the step size adaptively instead of using a constant step, that is, increase the step size if the previous step did not cause an overshoot around the minimum and decrease it if otherwise.

It is noted that using directional search instead of a constant step might greatly improve the convergence speed [41]. Modifications about the plain backpropation algorithm is also shown in [42]. It is known that the variable metric (or quasi Newton) method may be one of the fastest gradient optimization methods. However, its use involves the inversion of a Hessian matrix which seems impractical for a typical neural network. To aviod inversion of a Hessian matrix while converging fast, the conjugate gradient method was applied to train a neural network in [42, 43].

One popularly used modified version of the backpropagation algorithm is the so called momentum method. The weights are updated according to the current gradient as well as their previous change, that is, the weights are updated in the following way.

$$W^l \longleftarrow W^l + \eta \Delta^l (S^{l-1})^\tau + \mu \Delta W^l. \tag{2.12}$$

where ΔW^l is the previous change of the weight matrix, and μ is a positive number.

Note that the weight updating formula for the momentum method is almost the same as for the conjugate gradient method [44] except that the involved coefficients in the formula for the conjugate gradient method are computed more complicatedly and directional minimization is performed.

2.3. Recurrent neural network and dynamic backpropagation

The backpropagation algorithm is very useful and has been extensively applied when a neural net is trained to approximate a static continuous function. When a neural net is fed as inputs the previous values of its output, static backpropagation algorithm becomes ineffective in adapting the weights of this neural net. Since the current output of a neural net is dependent on the weights as well as its previous outputs recursively, the calculation of sensitivities involves a lot of complexity. The resulting neural net is called a recurrent neural net, which here is viewed as a combination of a feedforward neural net and a linear delayed feedback, shown in Figure 2.1.

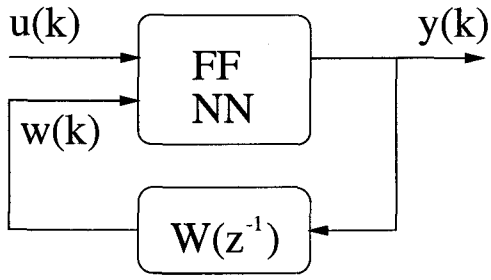


FIGURE 2.1: Block diagram for a recurrent neural network

This recurrent neural network may be described as

$$y(k) = f_N(u(k), w(k), \theta) \tag{2.13}$$

where function $f_N : R^{N_u} \times R^{N_w} \longrightarrow R^{N_y}$ is function realized by the neural net with $u(k) \in R^{N_u}$, $w(k) \in R^{N_w}$, $\theta \in R^{N_\theta}$, $y(k) \in R^{N_y}$, and z^{-1} the unit time delay operator.

Since $w(k) = W(z^{-1})y(k)$, and $W(z^{-1})$ is an affine function vector with respect to z^{-1}, $y(k)$ is dependent on $W(z^{-1})y(k)$ as well as θ. By the same equation, $W(z^{-1})y(k)$ is also dependent on θ. Therefore, we would like to denote the derivative of $y(k)$ with respect to a weight θ_j by $\frac{dy(k)}{d\theta_j}$.

After the training is completed, the weight vector θ is fixed. Then

$$y(k) = f_N(u(k), w(k)) \tag{2.14}$$

may be viewed as representative of dynamics of a plant. Usually a neural net is trained along some temporal trajectories of a plant. Let the desired trajectory be $y_d(k)$ at the instant of time k. Then the training error $e(k)$ is the difference between $y(k)$ and $y_d(k)$, i.e., $e(k) = y_d(k) - y(k)$. Let θ_j be a weight compont of vector θ. Let J be the chosen performance criterion, which is usually a functional of the training error $e(.)$. Then the sensitivity $\frac{\partial J}{\partial \theta_j}$ can be computed as

$$\frac{\partial J}{\partial \theta_j} = \frac{\partial J}{\partial e}\frac{\partial e}{\partial \theta_j} \tag{2.15}$$

However,

$$\frac{\partial e}{\partial \theta_j} = -\frac{dy(k)}{d\theta_j} \tag{2.16}$$

It is observed that

$$\frac{dy(k)}{d\theta_j} = \frac{\partial y(k)}{\partial \theta_j} + \sum_{i=1}^{N_w} \frac{\partial y(k)}{w_i(k)}\frac{\partial w_i(k)}{\partial \theta_j} \tag{2.17}$$

where $w_i(k)$ is the ith component of $w(k)$.

It is noted that since $w_i(k)$ is only the delayed version of $y(k)$, the above equation itself is a dynamic system and hence that backpropagation algorithm under this situation is called the dynamic backpropagation [31, 32]. The caculation of $\frac{dy(k)}{d\theta_j}$ can be performed as the plain backpropagation does except that there are more computations involved for the dynamic backpropagation.

Once the sensitivities are calculated, the weights updating can proceed with

$$\theta^{l+1} = \theta^l - \alpha \nabla_\theta J \tag{2.18}$$

This is the well-known gradient-based updating rule. Other updating formula can also be obtained as for a conventional feedforward neural network.

2.4. Locally recurrent multilayer neural network

A locally recurrent multilayer neural network is just a conventional multilayer neural network with local recurrency and cross-talk, by which we mean the outputs of some hidden layers are fed as inputs to these layers.

It should be noted that if on-line training is initiated to train a locally recurrent multilayer neural network, then the dynamic algorithm, such as the dynamic backpropagation algorithm, can be used effectively. Although the dynamic backpropagation requires more calculations than the plain backpropagation does, in nature calculations involved in the dynmaic backpropagation are based on the plain backpropagation. In what follows, the off-line training is focused, and the corresponding backpropagation algorithm is discussed.

The backpropagation algorithm for a conventional neural network can be somehow generalized to apply to a locally recurrent neural network. The backpropagation algorithm for a locally recurrent neural network is presented as the Backpropagation Algorithm III in the following.

The notations that are used in the Backpropagation Algorithm II are also used in the Backpropagation Algorithm III. To describe the Backpropagation Algorithm III, some more notations are needed, and are given as follows.

The original input signals are augmented, by the outputs of the first hidden layer, to form the new inputs to the first hidden layer. The original outputs of each hidden layer (except the last hidden layer) are augmented, by the outputs of the next hidden layer, to form the new inputs to the next hidden layer. Mathematically, the augmented input matrix $\tilde{S}$, output matrix $\tilde{S}^l$ of each layer $l = 1, 2, \cdots, L$ with $\tilde{S^L} = S^L$, and weight matrix

$\tilde{W}^l$ for each layer $l = 1, 2, \cdots, L$ can be represented as

$$\tilde{S} = \begin{bmatrix} \hat{S} \\ S^1 \end{bmatrix}.$$

$$\tilde{S}^l = \begin{bmatrix} \hat{S}^l \\ S^{l+1} \end{bmatrix} \text{ with } l = 0, 1, \cdots, L-1. \text{ Note that } \tilde{S}^0 = \tilde{S}.$$

$$\tilde{W}^l = \begin{bmatrix} W^l & W_r^l \end{bmatrix} \text{ where } W_r^l = \begin{bmatrix} w^l_{r_{1,1}} & w^l_{r_{1,2}} & \cdots & w^l_{r_{1,d_l}} \\ w^l_{r_{2,1}} & w^l_{r_{2,2}} & \cdots & w^l_{r_{2,d_l}} \\ \vdots & \vdots & \vdots & \vdots \\ w^l_{r_{d_l,1}} & w^l_{r_{d_l,2}} & \cdots & w^l_{r_{d_l,d_l}} \end{bmatrix}^\tau \text{ with } w^l_{r_{i,j}} \text{ repre-}$$

senting the connection strength from the jth neuron to the ith neuron in the same layer l, and $l = 1, 2, \cdots, L$.

Therefore, the Backpropagation Algorithm III may be described as

1. Forward phase

   $\tilde{S}^l = \underline{\sigma^l}[\tilde{W}^l \tilde{S^{l-1}}]$.

2. Error backpropagation

   $\Delta^L = (T - S^L) \circ \underline{g^L}[S^L]$.

   $\Delta^l = ((W^{l+1})^\tau \Delta^{l+1}) \circ \underline{g^l}[\tilde{S}^l]$.

3. Gradient computation and weight updating

   $\nabla_{\tilde{W}^l} J = -\Delta^l (\tilde{S^{l-1}})^\tau$.

   $\tilde{W}^l \longleftarrow \tilde{W}^l + \eta \Delta^l (\tilde{S^{l-1}})^\tau$.

2.5. Software implementation

Since training a typical neural network with a large set of data is very time-consuming if a MATLAB version of neural network tools is used, software is implemented

and coded in C for training a multilayer neural network. Since conventional neural networks may be viewed as a special form of recurrent neural networks just by setting the recurrent and cross-talk connection weights to be 0's, the implementation of the Backpropagation Algorithm III suffices. Note that the activation function $f^l(.)$ for each layer $l = 1, 2, \cdots, L$ may assume, for example, a form of either logistic function or tan-sigmoid function, or linear function. The momentum method is applied to update weights and biases, which is much faster and more stable than the plain backpropagation method. In addition, a proper random number generator is properly designed, which is useful for initialization of neural networks' parameters (that is, weights and biases).

All the programs for this software are listed in Appendix A, and used in the following analyses.

3. LATITUDINAL AND LONGITUDINAL NEURAL NETWORK STRUCTURES FOR FUNCTION APPROXIMATION

3.1. Introduction

One of the active areas of neural networks has been on the function approximation capabilities of neural networks, which still attracts a lot of attentions. One of the main applications of feedforward neural networks is to approximate arbitrary nonlinear continuous functions. Many research results [34, 39, 40] have been reported on the approximation capabilities. Mostly these works focus on the approximations of continuous functions by feedforward neural networks. Stronger results for approximations of functions defined on spaces of infinite dimensions may be found in [45] (see references therein). However, the theoretical results on approximations of nonlinear discontinuous functions by neural networks are considerably weaker. By applying Lusin's theorem, Hornik et al [40] showed that any measurable function may be approximated by a feedforward neural network in L^p sense. The neural networks involved in these works are non-constructive. Recently, interests have been seen in the constructive neural networks [46, 47, 48] on the purpose of overcoming the difficulties involved in training standard feedforward neural networks with backpropagation. Choi, et al [46, 47] investigated the piecewise interpolation capabilities for funciton approximations through constructive neural networks. Tessellation of a compact space was performed, and a number of neural network granules are applied. With the employment of the same kind of neural network structure, the proposed strategy makes piecewise nonlinear approximations by means of quadratic squashing functions. Very different from the traditional constructive neural networks, latitudinal neural networks are proposed to reduce the approximation error by recursively employing sub-neural

networks. This chapter is organized as follows: Section 3.2. is devoted to investigating the convergence property of the novel neural network—latitudinal neural networks, discussing function approximation with piecewise nonlinear fitting by longitudinal neural network, and presenting the relationship between these two neural network structures. The properties of latitudinal neural network architecture are further investigated and presented in section 3.3.. Finally, comments and concluding remarks are given.

3.2. Latitudinal neural network architecture

A novel neural network structure, which consists of a number of neural networks with each of those being called as a sub-neural network hereafter, is formed as such that the first sub-neural network aims at approximating the given function with given data, and then the second sub-neural network tries to, with given input data, approximate the error function from the first sub-neural network, and then the third one will devote to approximation of the error function from the second sub-neural network, and so forth. The resulting structure, called latitudinal neural network structure, shown in Figure 3.1, is intended to reduce the approximation error again and again by using a number of neural networks.

For simplifying notation, it is convenient to define the training error ratio r for a neural network as

$$r = ||f(x) - g(x)||/||f(x)|| \tag{3.1}$$

where $f(x)$ is a continuous function from R^n (or its compact subset) to R^m, and $g(x)$, which is designed to approximate $f(x)$ (also assume $g(x) \in L^2(R^n)$), is a mapping from R^n to R^m achieved by the neural network. $||.||$ is L^2 norm operation. Here,we call $f(x)$, $g(x)$ and $e(x) = f(x) - g(x)$ the target function, the actual output function and the training error function respectively.

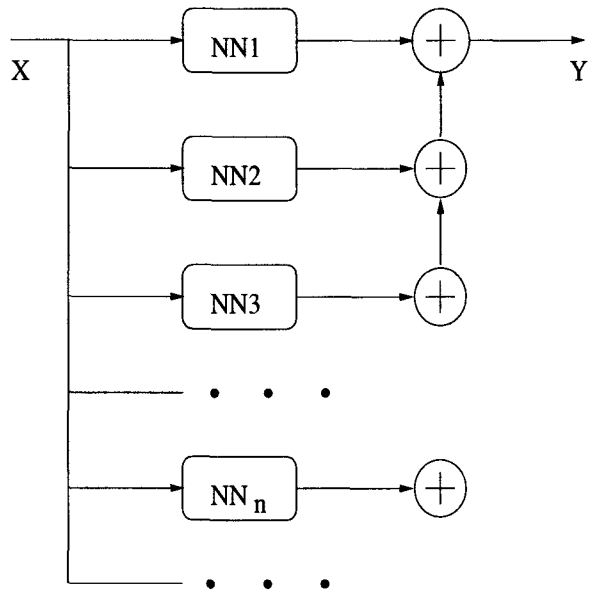


FIGURE 3.1: Latitudinal neural network architecture

Proposition 1 *$f(.)$ is a continuous function from R^n (or its compact subset) to R^m, and is approximated by latitudinal neural networks, shown in Figure 3.1. Then $f(.)$ can be approximated by a latitudinal neural network with arbitrary small error $\epsilon > 0$ in the L^2 norm sense, if*

$$\sup\{r_i, i \in Z_+\} < 1 \tag{3.2}$$

Where r_i is the training error ratio for the ith sub-neural net.

Proof: We now consider the finite section of the latitudinal neural networks, which consists of the first N ($N \in Z_+$) sub-neural networks. Let $g_i(.)$ and $e_i(.)$ be the actual output function of the ith sub-neural network and the training error function, from R^n to R^m. As is mentioned above, $e_i(.)$ is the target function for the $i+1$th sub-neural network. Here

$1 \leq i \leq N$. We know that the target function for the first sub-neural network is $f(.)$. Now let $e_0 = f$.

Then we obtain

$$r_i = ||e_{i-1} - g_i||/||e_{i-1}|| \tag{3.3}$$

where $1 \leq i \leq N$.

Note: if $||e_i||$ happens to be zero, then we don't need any more sub-neural nets for further training. Thus, in our case, $||e_i|| \neq 0$.

Then we have

$$||e_{i-1} - g_i|| = r_i||e_{i-1}|| \tag{3.4}$$

But $e_0 = f$, and $e_i = e_{i-1} - g_i$ for $1 \leq i \leq N$, then $g_i = e_{i-1} - e_i$.

Thus

$$\sum_{i=1}^{k} g_i - f = \sum_{i=1}^{k}(e_{i-1} - e_i) - e_0 = -e_k \tag{3.5}$$

where $1 \leq k \leq N$.

And $\sum_{i=1}^{k} g_i - f = g_k + \sum_{i=1}^{k-1} g_i - f = g_k - e_{k-1}$.

It turns out that

$$||e_k|| = ||\sum_{i=1}^{k} g_i - f|| = ||g_k - e_{k-1}|| \leq r_k||e_{k-1}|| \tag{3.6}$$

Therefore,

$$||\sum_{i=1}^{N} g_i - f|| \leq r_N||e_{N-1}|| \leq r_N r_{N-1}||e_{N-2}|| \leq \cdots \leq \Pi_{i=1}^{N} r_i||e_0|| \tag{3.7}$$

However, $e_0 = f \in L^2(R^n)$, and hence $||e_0|| < \infty$. Let $\gamma = \sup\{r_i, i \in Z_+\}$. By assumption, we have $r_i < \gamma < 1$ for $1 \leq i \leq N$.

Therefore, $||\sum_{i=1}^{N} g_i - f|| < \gamma^N||e_0||$. With N sufficiently large, the error between the target function and the final output of the latitudinal neural networks can be made arbitrarily small, in the sense of L^2 norm. This ends the proof.

Remark (1): Generally, a set S of data points are given in such a form that $S = \{(x_i, y_i) : x_i \in R^n, y_i \in R^m \text{and } i = 1, \cdots, N_s\}$. Then if we take the norm in the discrete

case, the above proposition still holds. This means that under some conditions, sufficiently many sub-neural networks will make the approximation error, in the discrete norm sense, sufficiently small, only on the the given data points. This does not mean, however, that the approximation error, in the continuous norm sense, will be made sufficiently small. This will explain why sometimes between two given immediate neighbor points, the actual output of the latitudinal neural networks will oscillate (sometimes even badly).

Remark (2): The structures of those neural nets in the latitudinal neural network may be either the same kind or different kinds. Thus here arises the concept of hybrid neural networks, such as the combination of wavelet neural networks and feedforward neural networks, the combination of neural networks based on radial basis function and feedforward neural networks, etc..

Remark (3): The number of sub-neural networks in the latitudinal neural network, which are needed to make a relative better approximation, will determine which part will be easily trained. If the number of sub-neural networks, required to make a good approximation, for a certain part, is less, then this part is smoother. A non-stationary part will need more sub-neural networks to give a good approximation. So the numbers of sub-neural networks are different for different parts. Then by finding the numbers of sub-neural networks, it will be easier to locate the singular points or segments. These numbers of sub-neural networks present a relative good measure for the relative singularity of the orginal mapping.

Remark (4): Generally speaking, latitudinal neural networks are trained in cascade form, and executed in parallel. But if somehow the training target for each sub-neural network can be pre-specified, then the training process can be performed in parallel (as we may can see later, this situation does occur).

Remark (5): It should be observed that even if $\sup\{r_i, i \in Z_+\} = 1$, but if with the exclusion of a finite number of r_i's being 1's, the greatest upper bound of all other r_i's is less than 1, then the Proposition still holds.

3.2.1. Longitudinal neural network structure

The idea of this kind of neural network is originated in Choi, et al's work [46, 47]. As mentioned in section 3.1., they first tessellated the compact subspace of a finite-dimensional space, then for each subset of this covering whose union covers the whole compact subpace in question, a neural network granule will be applied to approximate the specific function defined on the specific subset, which may be either a hypertriangle or a hyperrectangle. This results in a neural network structure, called a longitudinal neural network structure. In practice, piecewise linear functions are used in their work. In this part, we will develop a strategy to use piecewise nonlinear functions to approximate one-dimensional functions by introducing a new kind of sigmoidal function—a quadratic sqashing funcition.

3.2.2. Sigmoidal function

Three kinds of sigmoidal or squashing functions will be used in our neural network structures, which are defined in section 2.2.1.2. of chapter 2, and repeated in the following.

- Heaviside function $\sigma_h(x) = \begin{cases} 1 \ if \ \ x \geq 0 \\ 0 \ if \ \ x < 0 \end{cases}$
- Soft squashing function $\sigma_s(x) = \begin{cases} 1 \ if \ \ x > 1 \\ x \ if \ \ x \in [0,1] \\ 0 \ if \ \ x < 0 \end{cases}$
- Quadratic squashing function $\sigma_q(x) = \begin{cases} 1 \ if \ \ x > 1 \\ x^2 \ if \ \ x \in [0,1] \\ 0 \ if \ \ x < 0 \end{cases}$

3.2.3. Nonlinear fitting

In this part, our main focus will be on one-dimensional function approximation by using two hidden-layers of neural networks with three kinds of activation functions defined in the last part. As we know, any closed and bounded subset of the real set R will be compact and vice versa [49]. Functions defined on compact subsets of R will be our main interest. We also know that any three points in a plane, but not in a straight line, will determine a quadratic parabola. As shown in Figure 3.2, points A, B, C are arbitrary three points in the plane R^2, with coordinates $(x_1, y_1), (x_3, y_3), (x_2, y_2)$, respectively, and $x_1 < x_3 < x_2$. Without loss of generality, let f be a function from $[x_1, x_2]$ to R given by $f(x) = ax^2 + bx^2 + c$ with a, b, c constants determined by the given points A, B, and C; let g be a function from $[x_1, x_2]$ to R given by $g(x) = dx + e$ with d, e constants determined by the given points A and C. Let h be a fucntion defined on $[x_1, x_2]$ such that $h(x) = f(x) - g(x)$. By definition, we learn that $f(x_1) = y_1$; $f(x_2) = y_2$; and $g(x_1) = y_1$; $g(x_2) = y_2$. Then it follows that $h(x_1) = f(x_1) - g(x_1) = 0$ and $h(x_2) = f(x_2) - g(x_2) = 0$.

Since the function $f(x)$ is quadratic, then $h(x) = k(x - x_1)(x - x_2)$, with k being a constant which can be determined by

$$h(x_3) = f(x_3) - g(x_3) \tag{3.8}$$

However, $h(x_3) = k(x_3 - x_1)(x_3 - x_2)$, and $f(x_3) = y_3$, and $g(x_3) = \frac{y_2 - y_1}{x_2 - x_1}(x_3 - x_1) + y_1$

It turns out that

$$k = \frac{(y_3 - y_1)(x_2 - x_1) - (y_2 - y_1)(x_3 - x_1)}{(x_3 - x_1)(x_3 - x_2)(x_2 - x_1)} \tag{3.9}$$

If $y_1 = y_2 = y_3$, then $k = 0$, and hence $h(x) = 0$. This means that if three points happen to be on a straight line, then a linear function $g(x)$ is sufficient to approximate it.

Since AC is a line segment, described by function $g(x)$, it can be implemented by the neural network structure [46, 47], shown in Figure 3.3.

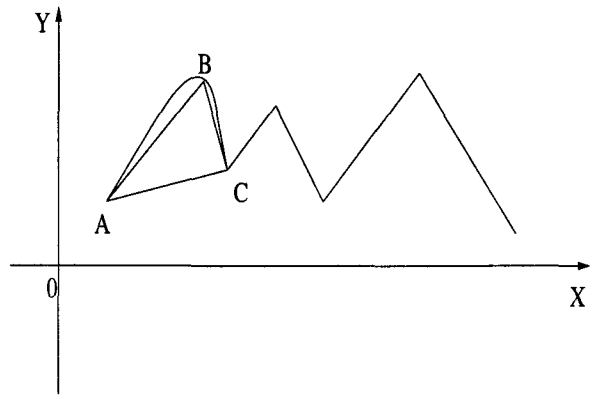


FIGURE 3.2: Piecewise quadratic fitting

There are three neurons in the first hidden layer, whose weights and biases, from left to right, are w_{11} and w_{12}, w_{21} and w_{22}, and w_1 and w_2, respectively. The activation function, is heaviside type for the first two neurons of this layer, and sqaushing type for the last neuron of this layer. The weights for all the neurons in the second hidden layer are all 1's; and the biases for them, from left to right, are -1 and -2. The last layer is the output layer, whose activation function is linear in weights, with the weights being y_{min} and $y_{max} - y_{min}$, from left to right.

The related weights are determined as follows:

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.10}$$

and

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} \bar{y_1} \\ \bar{y_2} \end{bmatrix} \tag{3.11}$$

where $\bar{y_i} = \frac{y_i - y_{min}}{y_{max} - y_{min}}$ for $i = 1, 2$ with $y_{min} = \min\{y_1, y_2\}$; and $y_{max} = \max\{y_1, y_2\}$.

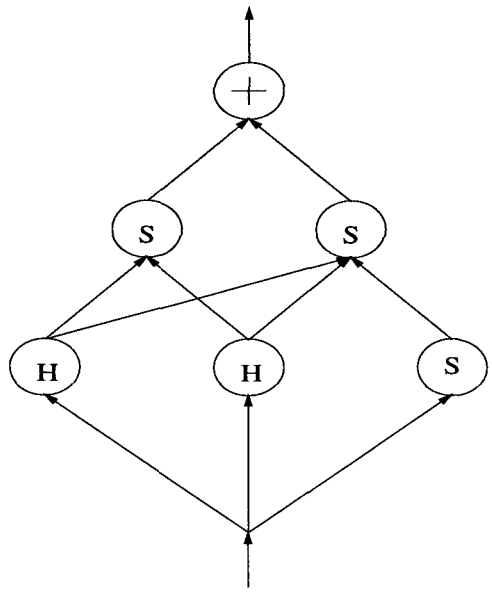


FIGURE 3.3: Piecewise linear approximation via a specific neural network. H — with Heaviside activation function; S — with Soft squashing activation function

It can be easily checked that if input $x = x_i$ for $i = 1,\ 2$, then the output of the neural network will be $y = y_i$ for $i = 1,\ 2$; if input $x \in (x_1, x_2)$, then the output y will be $y_{min} < y < y_{max}$; if input $x \notin [x_1, x_2]$, then the output y will be zero.

Next, a neural network is constructed to implement the function $h(x) = k(x - x_1)(x - x_2)$.

$h(x)$ can rewritten as

$$h(x) = k[(x - \frac{x_1 + x_2}{2})^2 - (\frac{x_1 - x_2}{2})^2] = -k(\frac{x_1 - x_2}{2})^2[1 - (\frac{x - \frac{x_1+x_2}{2}}{\frac{x_1-x_2}{2}})^2] \qquad (3.12)$$

The neural network structure, shown in Figure 3.4, are employed to approximate the function $h(x)$. In the first hidden layer, there are four neurons, with the weights and biases for the first two neurons being the same as in Figure 3.3, and with the weights and biases for the last two neurons being $\frac{2}{x_1-x_2}$ and $\frac{x_1+x_2}{x_2-x_1}$, and $\frac{2}{x_2-x_1}$ and $\frac{x_1+x_2}{x_1-x_2}$, respectively. We use

the activation function of heaviside type for the first two neurons, and quadratic squashing type for the last two neurons. In the second hidden layer, there is only one neuron with the weights being 1, 1, -1 and -1 from left to right, and with the bias being -1. The last layer is output layer with weight being $-k(\frac{x_1-x_2}{2})^2$. We now show what kind of function this neural network can produce. Let $x \in R$. If $x \in \{x_1, x_2\}$, then by Equation (3.10), both of the outputs of the leftmost two neurons in the first hidden layer will be 1's. For the third neuron in this layer, with the input, weight and bias being x, $\frac{2}{x_1-x_2}$, and $\frac{x_1+x_2}{x_2-x_1}$, then the input to the activation function will be $(x \times \frac{2}{x_1-x_2} + \frac{x_1+x_2}{x_2-x_1}) = \frac{x_1+x_2-2x}{x_2-x_1}$; if $x = x_1$, then the output of this neuron will be 1; if $x = x_2$, then the output of this neuron will be 0. Similarly, for the fourth neuron on the right, if $x = x_1$, then the output of this neuron will be 0; if $x = x_2$, then the output of this neuron will be 1. Therefore, for the neuron in the second hidden layer, the output will be 0, which in turn implies that the final output of the neural network will be 0. If $x \in (x_1, x_2)$, then the outputs of the first two neurons on the left in the first hidden layer will be 1's. For the third neuron from the left, the input to the activation function is $(x \times \frac{2}{x_1-x_2} + \frac{x_1+x_2}{x_2-x_1}) = \frac{x_1+x_2-2x}{x_2-x_1}$ while the input to the activation function of the neuron on the far right is $\frac{2x-x_1-x_2}{x_2-x_1}$. If $x = \frac{x_1+x_2}{2}$, then both of the outputs of these two neurons will be 0's. Otherwise, since $-1 < \frac{x_1+x_2-2x}{x_2-x_1} < 1$, each of these two outputs will be either 0 or $(\frac{x_1+x_2-2x}{x_2-x_1})^2$, but not both. Counting in all the inputs, weights and bias for the neuron in the second hidden layer, the output of this neuron will be $1 - (\frac{x_1+x_2-2x}{x_2-x_1})^2$. Then the output of the output layer will be $-k(\frac{x_1-x_2}{2})^2[1 - (\frac{x_1+x_2-2x}{x_2-x_1})^2] = k(x-x_1)(x-x_2) = h(x)$. If $x \notin [x_1, x_2]$, then the outputs of the two neurons on the far left of the second hidden layer, by Equation (3.10), will be either $1, 0$ or $0, 1$. By the same argument as above, the outputs of the two neurons on the far right of the second hidden layer, will be either 0 and $\min\{(\frac{x_1+x_2-2x}{x_2-x_1})^2, 1\}$ or $\min\{(\frac{x_1+x_2-2x}{x_2-x_1})^2, 1\}$ and 0. Then the input to the activation function for the neuron in the second hidden layer will be $1+0+0-\min\{(\frac{x_1+x_2-2x}{x_2-x_1})^2, 1\}-1 = -\min\{(\frac{x_1+x_2-2x}{x_2-x_1})^2, 1\} \leq 0$. Thus the output of this neuron is 0. And hence the final output of the output layer will

be 0. In a word, the neuron network shown in Figure 3.4 acts as a function that satifies the following conditions:

- if the input x is inside the given interval $[x_1, x_2]$, then the output will be $h(x)$;
- if the input x is outside the given interval $[x_1, x_2]$, then the output will be 0.

Incorporating the neural network structure shown in Figure 3.3 into the neural network structure shown in Figure 3.4 produces the new neural network shown in Figure 3.5. Thus, the output of this neuron network will be $g(x) + h(x)$, which is precisely $f(x)$, if the input x is inside the given interval $[x_1, x_2]$, while the output will be 0 if the input x is outside the given interval $[x_1, x_2]$.

3.2.4. Neural network array

In this section, various constructive neural networks will be employed to implement a continuous function $f(x)$ defined on a compact subset Ω of R. First of all, segment the given compact subset into many non-overlapping compact subsets U_i ($i \in A_n = \{i : 1 \leq i \leq n; n \in Z_+\}$), of Ω, whose union will precisely be the given compact set Ω. As we have known, the data samples are usually given in the form of $(x, f(x))$ with x being one element of the set of the boundary points of all the subsets U_i, which we may assume to be $\{x_i, i = 1, 2, \cdots, m; m \in Z_+\}$ with $x_i > x_j$ for $i > j$. Without loss of generality, assume Ω to be connected (if it is not connected, we may have the same argument for each of its connected subset Ω_i with $\cup_i \Omega_i = \Omega$). If m is odd, then we have a collection of intervals $[x_{2k+1}, x_{2k+3}]$ with k ranging from 0 to $\frac{m-3}{2}$. If m is even, then we may have an interval $[x_{m-1}, x_m]$ and a collection of intervals $[x_{2k+1}, x_{2k+3}]$ with k varying from 0 to $\frac{m-4}{2}$. For these two cases, the only difference is that for the latter case we need to employ an extra constructive neural network, with structure shown in Figure 3.3, to approximate the function defined on this interval by linear fitting. Consequently, we only

need to consider the former case. However, with $f(x_{2k+1}), f(x_{2k+2}), f(x_{2k+3})$ given, the function defined on the interval $[x_{2k+1}, x_{2k+3}]$ can be approximated by the neural network shown in Figure 3.5. Therefore, we apply a variety of neural networks to approximate the function $f(x)$ defined on the compact subset Ω, with each neural network nonlinearly approximating a specific subset of the function $f(x)$. Then we employ the structure shown in Figure 3.6 [47] to capture the final output by using analog OR operation, which essentially takes the non-zero real number from all the outputs of sub-neural nets NN_i.

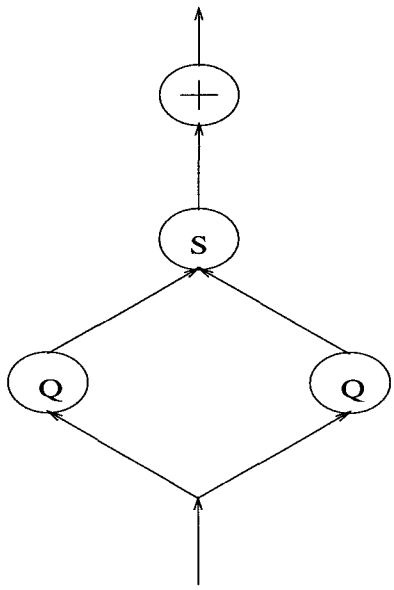


FIGURE 3.4: Piecewise quadratic approximation via a neural network. S — with Soft squashing activation function; Q — with Quadratic squashing activation function

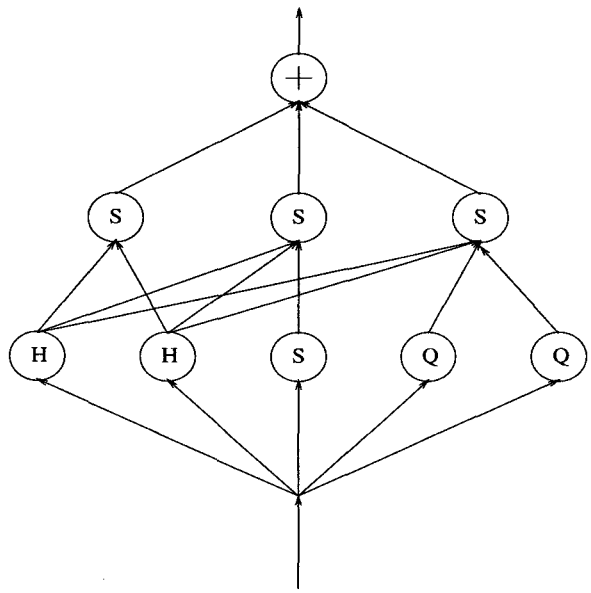


FIGURE 3.5: Nonlinear approximation via a neural network

3.2.5. Continuous function approximation with desired pricision

As we know in *Calculus*, any one-dimensional continuous function defined on a compact set can be approximated by either countably many rectangles, or by linear fitting, or by piecewise quadratic functions, with arbitrary small error in the sense of the Euclidean distance, so long as the maximum of the diameters of the resulting subsets of this compact set is sufficiently small. This fact, in turn, means that to achieve the desired precision, sufficiently many neural networks, either in the form shown in Figure 3.3, or in the form shown in Figure 3.5, may be needed.

For each sub-neural network NN_i in Figure 3.5, through use of nonlinear fitting, the number of neurons involved in this kind of structure is 9 (counting out the input neruon), while, with the use of linear fitting, two neural networks will be used with structure shown

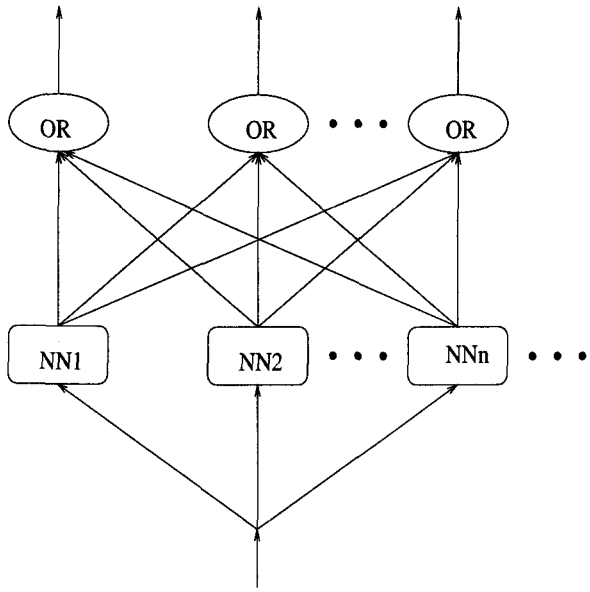


FIGURE 3.6: Neural network array

in Figure 3.3 [47], and the number of neurons is 12. Then if there need be N sub-nerual networks with nonlinear fitting, then $3N$ neurons will be saved compared with linear fitting sub-neural networks. Since for precise function approximation, many sub-neural networks will be used, a large number of neurons will be saved through nonlinear fitting other than linear fitting. Moreover, the quadratic nonlinear approximation displays more smoothness than the linear approximation.

3.2.6. Relations between latitudinal and longitudinal neural networks

As discussed before, the latitudinal neural network and the longitudinal neural network are organized in different ways with the former generally reducing the approximation

error by using more sub-neural networks based on the whole given data set, while with the latter generally performing local piecewise linear/nonlinear fittings through specific sub-neural networks based on a specific subset of the given data set. The longitudinal neural network, however, can be considered as a specific case of a latitudinal neural network if each sub-neural network of the longitudinal neural network is considered as an approximation for the whole target function. This fact can be justified by the following argument.

Without loss of generality, let $f(.)$ be a continuous function defined on a connected compact subset Ω of R^n with $f(.) \in L^2(R^n)$. To achieve a close approximation by the latitudinal neural network, the tessellation [47] of the compact set Ω has to be performed. As mentioned above, there exists a positive integer N, for any given positive real number $\epsilon > 0$ such that N or more properly constructed neural networks [47] can approximate the function $f(.)$ with approximation error less than ϵ in the L^2 norm sense. Let the compact set Ω be tessellated into N nonoverlapping compact subsets Ω_i for $i = 1, \cdots, N$ such that on each Ω_i, $f(.)$ can be approximated by a specific sub-neural network of the longitudinal neural network. As we know from [47], on the given data points, the longitudinal neural network can achieve precise representation of the target function $f(.)$. Consequently, it is then only necessary to consider function approximation on the interior points of Ω_i for $i = 1, \cdots, N$. For the funtion $f(.)$ defined on Ω_i, there is a sub-nerual network NN_i, which can give an approximate G_i on Ω_i (0 if outside Ω_i), and the approximation error is $E_i = F_i - G_i$. Here if $x \in \Omega_i$, then $F_i(x) = f(x)$; otherwise, $F_i(x) = 0$. Then from the viewpoint of the latitudinal neural network, the training is based on the whole data set, and the ith sub-neural network of the longitudinal neural network is also considered as the ith sub-neural network of a latitudinal neural network. In this context, let $f = e_0$, and g_i and e_i be the output of the ith sub-neural network and the approximation error, respectively. Since $f = \sum_{i=1}^{N} F_i$, $e_i = e_{i-1} - g_i$, and $g_i = G_i$, then $||e_i|| = ||e_{i-1} - g_i|| = ||e_{i-1} - G_i||$. However, from the above discussion, $e_{i-1} = f - \sum_{k=1}^{i-1} g_k = f - \sum_{k=1}^{i-1} G_k$. From $f = \sum_{i=1}^{N} F_i$, it follows

that, $e_{i-1} = \sum_{k=1}^{N} F_k - \sum_{k=1}^{i-1} G_k = \sum_{k=i}^{N} F_k + \sum_{k=1}^{i-1}(F_k - G_k) = \sum_{k=i}^{N} F_k + \sum_{k=1}^{i-1} E_k$, and $e_i = \sum_{k=i+1}^{N} F_k + \sum_{k=1}^{i} E_k$. Thus, $||e_i|| = ||\sum_{k=i+1}^{N} F_k + \sum_{k=1}^{i} E_k||$.

Since $F_k(x)$ and $G_k(x)$ are 0's if $x \in \Omega_i$, and hence $E_k(x)$ is also 0, then $F_k(x), G_k(x)$, and $E_k(x)$ for $k = 1, \cdots, N$, are orthogonal, respectively, if $x \in \cup_{i=1}^{N} int(\Omega_i)$. It follows that $||e_i||^2 = ||\sum_{k=i+1}^{N} F_k + \sum_{k=1}^{i} E_k||^2 = \sum_{k=i+1}^{N} ||F_k||^2 + \sum_{k=1}^{i} ||E_k||^2$. Hence,

$$\frac{||e_i||^2}{||e_{i-1}||^2} = \frac{\sum_{k=i+1}^{N} ||F_k||^2 + \sum_{k=1}^{i} ||E_k||^2}{\sum_{k=i}^{N} ||F_k||^2 + \sum_{k=1}^{i-1} ||E_k||^2} = \frac{\sum_{k=i+1}^{N} ||F_k||^2 + \sum_{k=1}^{i-1} ||E_k||^2 + ||E_i||^2}{\sum_{k=i+1}^{N} ||F_k||^2 + \sum_{k=1}^{i-1} ||E_k||^2 + ||F_i||^2} \tag{3.13}$$

Since for the ith sub-neural network (also NN_i in the context of the longitudinal neural network), the approximation error can be made sufficiently small, that is, $||E_i||^2 < ||F_i||^2$. Thus, $\frac{||e_i||^2}{||e_{i-1}||^2} < 1$, and hence, $r_i = \frac{||e_i||}{||e_{i-1}||} < 1$. Since N is finite, then there exists r such that $\max\{r_i, i = 1, \cdots, N\} < r < 1$. This means that the condition for Proposition in section 3.2. can be satisfied, and hence the convergence can be guaranteed, which in turn implies that the approximation with sufficiently small error can be made.

Remark: with the relationship between these two neural network structures established, it is possible to employ the hybrid structure of these two structures in a specific application, for example, using the longitudinal neural network structure for the non-stationary part of the given target function, while using the latitudinal neural network structure for its stationary part.

3.2.7. Comments

In this section, the concept of a new kind of neural networks (the latitudinal neural network) is proposed, its structure is discussed, and its convergence property is given. Another kind of neural networks (the longitudinal neural network) is applied by using piecewise nonlinear activation functions, a number of neurons may be saved by the strategy proposed in this section compared with existing works, and better smoothness may be achieved with the proposed method. Furthermore, it has been shown that the longitudinal

neural network can be considered as a specific case of the latitudinal neural network, and the related convergence properties is presented.

3.3. Properties of latitudinal neural networks

Neural network methodology has been widely applied. In the context of control engineering it is mostly used as a pattern identifier or a synthesized controller based on the available data or measurements, among others [31, 50]. Recently, the approximation capabilities of constructive feedforward neural networks have been studied in [47] through use of some special kinds of sigmoidal functions such as a heaviside function and a soft squashing function and the piecewise linear approximation technique. In contrast to the neural network structure thereof, the architecture of latitudinal neural networks is proposed in [51] in hope to give a better approximation to a given function, and is further studied in [52]. The convergence under some assumptions is further given. Mathematically speaking, a given function $f : R^m \longrightarrow R^n$ is approximated by a series of functions of the same dimensions f_i for $i = 1, \cdots, N$ such that when N takes some proper value the approximation error in some norm sense can be made acceptably small. The latitudinal neural network structure is further investigated in this section with the introduction of some new kinds of sigmoidal functions and their combinations. It will be shown that any continuous piecewise quadratic function or polynomial of degree m ($m \geq 3$) can be represented by a neural network, and how many neurons are needed in the hidden layer. On this basis, the development of the properties relative to latitudinal neural networks is dealt with. Finally, the concluding remarks are presented.

3.3.1. Sigmoidal functions and their combinations

In addition to the sigmoidal functions defined in section 3.2.2., a kind of somewhat more general sigmoidal function, which is defined in the following, will also be used in this section.

$$\sigma_{p_n}(x) = \begin{cases} 1 \ if \ x > 1 \\ x^n \ if \ x \in [0,1] \quad \text{where } n \geq 3. \\ 0 \ if \ x < 0 \end{cases}$$

Definition 1: Suppose $\sigma : R \to R$ is a sigmoidal function. The class of functions, which a 2-layer neural network with the activiation function σ,n neurons in the hidden layer and a linear output layer realizes, is defined as $\mathcal{F}^d_{\sigma,n} = \{f : R^d \to R | f(x) = v_0 + \sum_{i=1}^n v_i \sigma(\sum_{j=1}^d w_{ij} x_j + \theta_i)\}$.

In [51], through use of the quadratic squashing function, the piecewise quadratic approximation is suggested with the employment of a longitudainal neural network. In what follows, the idea thereof is further investigated, and more rigorous results are obtained.

Lemma 1 *Any continuous, piecewise quadratic functions $f(x) : R \to R$ with $x_0 < x_1 < \cdots < x_n$ (so-called kinks) and constant values on $(-\infty, x_0)$ and on (x_n, ∞) are in $\mathcal{F}^1_{\sigma_s,n} \bigoplus \mathcal{F}^1_{\sigma_q,n}$, where the symbol $\bigoplus$ is defined by $A \bigoplus B = \{u + v | \ u \in A, \ v \in B\}$.*

Proof: A continuous piecewise quadratic function $f(x)$ on $[x_0, x_n]$ can be written as

$$f(x) = \sum_{i=0}^{n-1} f_i(x) \mathbf{1}_{[x_i, x_{i+1}]}(x) \tag{3.14}$$

where $f_i(x)$'s are quadratic functions; and $\mathbf{1}_{[x_i,x_{i+1}]}(x) = \begin{cases} 1, & x \in [x_i, x_{i+1}] \\ 0, & x \notin [x_i, x_{i+1}] \end{cases}$

Define a continuous piecewise linear function $g : R \to R$ such that $g(x_i) = f(x_i)$ for $i = 0, 1, \cdots, n$. Then $g(x)$ can be written as

$$g(x) = \quad g(x_0) +$$

$$\sum_{i=1}^{n}\{\frac{g(x_i)-g(x_{i-1})}{x_i-x_{i-1}}(x-x_{i-1})\mathbf{1}_{[x_{i-1},x_i]}(x)+ [g(x_i)-g(x_{i-1})]\mathbf{1}_{(x_i,\infty)}(x)\} \tag{3.15}$$

That is,

$$\begin{aligned} g(x) = \quad & g(x_0)+ \\ & \sum_{i=1}^{n}[g(x_i)-g(x_{i-1})]\{\frac{x-x_{i-1}}{x_i-x_{i-1}}\mathbf{1}_{[x_{i-1},x_i]}(x)+\mathbf{1}_{(x_i,\infty)}(x)\} \end{aligned} \tag{3.16}$$

But

$$\begin{aligned} & \frac{x-x_{i-1}}{x_i-x_{i-1}}\mathbf{1}_{[x_{i-1},x_i]}(x)+\mathbf{1}_{(x_i,\infty)}(x) \\ = \quad & \frac{x-x_{i-1}}{x_i-x_{i-1}}\{\mathbf{1}_{[0,1]}(\frac{x-x_{i-1}}{x_i-x_{i-1}})+\mathbf{1}_{(1,\infty)}(\frac{x-x_{i-1}}{x_i-x_{i-1}})\} \\ = \quad & \sigma_s(\frac{x-x_{i-1}}{x_i-x_{i-1}}) \end{aligned} \tag{3.17}$$

Therefore,

$$g(x)=f(x_0)+\sum_{i=1}^{n}[f(x_i)-f(x_{i-1})]\sigma_s(\frac{x-x_{i-1}}{x_i-x_{i-1}}) \tag{3.18}$$

This implies that $g(x)\in\mathcal{F}^1_{\sigma_s,n}$.

Note that for $x\in[x_i,x_{i+1}]$, $i=0,1,\cdots,n-1$, $f(x)$ is a quadratic function, and $g(x)$ is an affine linear function, and thus $f(x)-g(x)$ is still a quadratic function. Since $f(x)-g(x)=0$ for $x=x_i,x_{i+1}$, then on $[x_i,x_{i+1}]$, $f(x)-g(x)$ can be written as

$$f(x)-g(x)=c_i(x-x_i)(x-x_{i+1}) \tag{3.19}$$

So $f(x)-g(x)=\sum_{i=0}^{n}c_i(x-x_i)(x-x_{i+1})\mathbf{1}_{[x_i,x_{i+1}]}(x)$. It can be readily shown that

$$\begin{aligned} & (x-x_i)(x-x_{i+1})\mathbf{1}_{[x_i,x_{i+1}]}(x) \\ = \quad & -(\frac{x_{i+1}-x_i}{2})^2\{1-[\sigma_q(\frac{2}{x_{i+1}-x_i}x-\frac{x_{i+1}+x_i}{x_{i+1}-x_i}) \\ & +\sigma_q(-\frac{2}{x_{i+1}-x_i}x+\frac{x_{i+1}+x_i}{x_{i+1}-x_i})]\} \end{aligned} \tag{3.20}$$

Then we have

$$f(x) - g(x) = \sum_{i=0}^{n} -c_i(\frac{x_{i+1} - x_i}{2})^2\{1 - [\sigma_q(\frac{2}{x_{i+1} - x_i}x - \frac{x_{i+1} + x_i}{x_{i+1} - x_i}) + \sigma_q(-\frac{2}{x_{i+1} - x_i}x + \frac{x_{i+1} + x_i}{x_{i+1} - x_i})]\} \tag{3.21}$$

That is,

$$\begin{aligned} f(x) = [g(x) + \sum_{i=0}^{n} -c_i(\frac{x_{i+1} - x_i}{2})^2] \\ + \sum_{i=0}^{n} -c_i(\frac{x_{i+1} - x_i}{2})^2\sigma_q(\frac{2}{x_{i+1} - x_i}x - \frac{x_{i+1} + x_i}{x_{i+1} - x_i}) \\ + \sum_{i=0}^{n} -c_i(\frac{x_{i+1} - x_i}{2})^2\sigma_q(-\frac{2}{x_{i+1} - x_i}x + \frac{x_{i+1} + x_i}{x_{i+1} - x_i}) \end{aligned} \tag{3.22}$$

Note that $g(x) \in \mathcal{F}^1_{\sigma_s,n}$, and hence the first term on the right-hand side of the above equation $[g(x) + \sum_{i=0}^{n} -c_i(\frac{x_{i+1}-x_i}{2})^2] \in \mathcal{F}^1_{\sigma_s,n}$; and note also that both the second-term and the third-term are in $\mathcal{F}^1_{\sigma_q,n}$. Thus, $f(x) \in \mathcal{F}^1_{\sigma_s,n} \bigoplus \mathcal{F}^1_{\sigma_q,n}$, concluding the proof.

Consider the approximation of a more general class of functions than what is involved in Lemma 1, we then have the following result.

Lemma 2 *Any continuous, piecewise polynomial of degree m ($m \geq 3$) functions $f : R \to R$ with kinks $x_0 < x_1 < \cdots < x_n$ and constant values in $(-\infty, x_0)$ and on (x_n, ∞) are on $\mathcal{F}^1_{\sigma_s,n} \bigoplus \mathcal{F}^1_{\sigma_q,n} \bigoplus_{i=3}^{m} \mathcal{F}^1_{\sigma_{p_i},n}$.*

Proof: The induction method is used to give the proof.

First of all, we show the conclusion holds for $m = 3$. Note that f can written as

$$f(x) = \sum_{i=0}^{n-1} f_i(x)\mathbf{1}_{[x_i,x_{i+1}]}(x) \tag{3.23}$$

where $f_i(x)$'s are polynomials of degree 3.

One can show that

$$(x^3 - x)\mathbf{1}_{[-1,1]}(x) = [\sigma_{p_3}(x) - \sigma_{p_3}(-x)] - [\sigma_s(x) - \sigma_s(-x)] \tag{3.24}$$

That is,

$$(x'^3 - x')\mathbf{1}_{[x_i,x_{i+1}]}(x) = [\sigma_{p_3}(x') - \sigma_{p_3}(-x')] - [\sigma_s(x') - \sigma_s(-x')] \tag{3.25}$$

where $x' = \frac{2}{x_{i+1}-x_i}x - \frac{x_{i+1}+x_i}{x_{i+1}-x_i}$.

Let $f_i(x) = \sum_{j=0}^{3} a_{i,j}x^j$. Then

$$\begin{aligned}
&f_i(x)\mathbf{1}_{[x_i,x_{i+1}]}(x) = a_{i,3}(\frac{x_{i+1}-x_i}{2})^3 \\
&\{[\sigma_{p_3}(x') - \sigma_{p_3}(-x')] - [\sigma_s(x') - \sigma_s(-x')]\} \\
&+\sum_{j=0}^{2} a_{i,j}x^j\mathbf{1}_{[x_i,x_{i+1}]}(x) \\
&+a_{i,3}(\frac{x_{i+1}-x_i}{2})^3(\frac{2}{x_{i+1}-x_i}x - \frac{x_{i+1}+x_i}{x_{i+1}-x_i})\mathbf{1}_{[x_i,x_{i+1}]}(x) \\
&+a_{i,3}(\frac{x_{i+1}-x_i}{2})^3\sum_{j=0}^{2}\binom{3}{j}(-\frac{x_{i+1}+x_i}{2})^{3-j}x^j\mathbf{1}_{[x_i,x_{i+1}]}(x)
\end{aligned} \tag{3.26}$$

Let

$$\begin{aligned}
h_i(x) &= \sum_{j=0}^{2} a_{i,j}x^j\mathbf{1}_{[x_i,x_{i+1}]}(x) \\
&+a_{i,3}(\frac{x_{i+1}-x_i}{2})^3(\frac{2}{x_{i+1}-x_i}x - \frac{x_{i+1}+x_i}{x_{i+1}-x_i})\mathbf{1}_{[x_i,x_{i+1}]}(x) \\
&+a_{i,3}(\frac{x_{i+1}-x_i}{2})^3\sum_{j=0}^{2}\binom{3}{j}(-\frac{x_{i+1}+x_i}{2})^{3-j}x^j\mathbf{1}_{[x_i,x_{i+1}]}(x)
\end{aligned} \tag{3.27}$$

It can be shown that $h_i(x_k) = f_i(x_k)$ for $k = i, i+1$. Let $h : R \to R$ be defined as $h(x) = \sum_{i=0}^{n-1} h_i(x)\mathbf{1}_{[x_i,x_{i+1}]}(x)$. Then $h(x)$ is a continuous, piecewise quadratic function,

and hence $h(x) \in \mathcal{F}^1_{\sigma_s,n} \bigoplus \mathcal{F}^1_{\sigma_q,n}$ by Lemma 1.

However,

$$\begin{aligned} f(x) &= h(x) + \sum_{i=0}^{n-1} a_{i,3}\{[\sigma_{p_3}(x') - \sigma_{p_3}(-x')] \\ &\quad -[\sigma_s(x') - \sigma_s(-x')]\} \\ &= h(x) \\ &\quad + \sum_{i=0}^{n-1} -a_{i,3}(\frac{x_{i+1} - x_i}{2})^3[\sigma_s(x') - \sigma_s(-x')] \\ &\quad + \sum_{i=0}^{n-1} a_{i,3}(\frac{x_{i+1} - x_i}{2})^3[\sigma_{p_3}(x') - \sigma_{p_3}(-x')] \end{aligned} \tag{3.28}$$

Note that in the above equation, the second term is in $\mathcal{F}^1_{\sigma_s,n}$ and the third term is in $\mathcal{F}^1_{\sigma_{p_3},n}$. Thus,

$$f(x) \in \mathcal{F}^1_{\sigma_s,n} \bigoplus \mathcal{F}^1_{\sigma_q,n} \bigoplus \mathcal{F}^1_{\sigma_{p_3},n} \tag{3.29}$$

Suppose that the conclusion holds for case $m = k - 1$. We now show that this implies the conclusion also holds for case $m = k$.

Note that f can written as

$$f(x) = \sum_{i=0}^{n-1} f_i(x)\mathbf{1}_{[x_i,x_{i+1}]}(x) \tag{3.30}$$

where $f_i(x)$'s are polynomials of degree k.

One can show that

$$(x^k - x)\mathbf{1}_{[-1,1]}(x) = [\sigma_{p_k}(x) - \sigma_{p_k}(-x)] - [\sigma_s(x) - \sigma_s(-x)] \tag{3.31}$$

That is,

$$(x'^k - x')\mathbf{1}_{[x_i,x_{i+1}]}(x) = [\sigma_{p_k}(x') - \sigma_{p_k}(-x')] - [\sigma_s(x') - \sigma_s(-x')] \tag{3.32}$$

where $x' = \frac{2}{x_{i+1}-x_i}x - \frac{x_{i+1}+x_i}{x_{i+1}-x_i}$.

Let $f_i(x) = \sum_{j=0}^{k} a_{i,j}x^j$. Then

$$f_i(x)\mathbf{1}_{[x_i,x_{i+1}]}(x) = a_{i,k}(\frac{x_{i+1} - x_i}{2})^k$$

$$
\begin{aligned}
&\{[\sigma_{p_k}(x') - \sigma_{p_k}(-x')] - [\sigma_s(x') - \sigma_s(-x')]\} \\
&+\sum_{j=0}^{k-1} a_{i,j} x^j \mathbf{1}_{[x_i, x_{i+1}]}(x) \\
&+a_{i,k}(\frac{x_{i+1} - x_i}{2})^k (\frac{2}{x_{i+1} - x_i} x - \frac{x_{i+1} + x_i}{x_{i+1} - x_i}) \mathbf{1}_{[x_i, x_{i+1}]}(x) \\
&+a_{i,k}(\frac{x_{i+1} - x_i}{2})^k \sum_{j=0}^{k-1} \binom{k}{j} (-\frac{x_{i+1} + x_i}{2})^{k-j} x^j \mathbf{1}_{[x_i, x_{i+1}]}(x)
\end{aligned}
\tag{3.33}
$$

Define

$$
\begin{aligned}
h_i(x) &= \sum_{j=0}^{k-1} a_{i,j} x^j \mathbf{1}_{[x_i, x_{i+1}]}(x) \\
&+a_{i,k}(\frac{x_{i+1} - x_i}{2})^k (\frac{2}{x_{i+1} - x_i} x - \frac{x_{i+1} + x_i}{x_{i+1} - x_i}) \mathbf{1}_{[x_i, x_{i+1}]}(x) \\
&+a_{i,k}(\frac{x_{i+1} - x_i}{2})^k \sum_{j=0}^{k-1} \binom{k}{j} (-\frac{x_{i+1} + x_i}{2})^{k-j} x^j \mathbf{1}_{[x_i, x_{i+1}]}(x)
\end{aligned}
\tag{3.34}
$$

and $h(x) = \sum_{i=0}^{n-1} h_i(x)$. One can show that $h(x)$ is a continuous function of degree $k-1$. With the assumption,

$$
h(x) \in \mathcal{F}^1_{\sigma_s, n} \bigoplus \mathcal{F}^1_{\sigma_q, n} \bigoplus_{i=3}^{k-1} \mathcal{F}^1_{\sigma_{p_i}, n} \tag{3.35}
$$

However, $f(x)$ can be rewritten as

$$
\begin{aligned}
f(x) &= h(x) + \sum_{i=0}^{n-1} a_{i,k} \{[\sigma_{p_k}(x') - \sigma_{p_k}(-x')] \\
&\quad -[\sigma_s(x') - \sigma_s(-x')]\} \\
&= h(x) \\
&\quad + \sum_{i=0}^{n-1} -a_{i,k} (\frac{x_{i+1} - x_i}{2})^k [\sigma_s(x') - \sigma_s(-x')] \\
&\quad + \sum_{i=0}^{n-1} a_{i,k} (\frac{x_{i+1} - x_i}{2})^k [\sigma_{p_k}(x') - \sigma_{p_k}(-x')]
\end{aligned}
\tag{3.36}
$$

Since on the right-hand-side of the above equation, the first term is in $\mathcal{F}^1_{\sigma_s, n} \bigoplus \mathcal{F}^1_{\sigma_q, n} \bigoplus_{i=3}^{k-1} \mathcal{F}^1_{\sigma_{p_i}, n}$, the second term is in $\mathcal{F}^1_{\sigma_s, n}$, and the third term is in $\mathcal{F}^1_{\sigma_{p_k}, n}$, it

follows immediately that

$$f(x) \in \mathcal{F}^1_{\sigma_s,n} \bigoplus \mathcal{F}^1_{\sigma_q,n} \bigoplus_{i=3}^{m} \mathcal{F}^1_{\sigma_{p_i},n} \tag{3.37}$$

This concludes the proof.

Regarding the number of neurons used in the hidden layer and the type of corresponding sigmoidal functions used, we have the following result.

Corollary 1 *Any continuous, piecewise polynomial of degree m ($m \geq 3$) functions $f : R \to R$ with kinks $x_0 < x_1 < \cdots < x_n$ and constant values on $(-\infty, x_0)$ and on (x_n, ∞), can be realized by a 2-layer neural network with a hidden layer and a linear output layer. The hidden layer has at most $2n$ neurons with activation function σ_{p_k} for $3 \leq k \leq m$ and σ_q, and n neurons with the activation function σ_s.*

Proof: It is observed from Equation (3.36) that the recursion process can be repeated until $h(x)$ is a continuous, piecewise quadratic function. By Lemma 1, a continuous piecewise quadratic function can be formed by a function in $\mathcal{F}^1_{\sigma_q}$ and a piecewise linear function. And the linear combination of all the continuous,piecewise linear functions interpolating the same points $(x_i, f(x_i))$ for $i = 0, 1, \cdots, n$ are still continuous and piecewise linear. Thus, in the hidden layer, at most $2n$ neurons with each of the activation functions σ_{p_k} for $3 \leq k \leq m$ and σ_q, and n neurons with the activation functions σ_s, are needed. This ends the proof.

3.3.2. Study on the properties of latitudinal neural networks

Definition 1: Suppose $f : [a, b] \to R$ is a function. The total variation norm of f is defined by $||f||_{tv} = \sup_{\{t_0,t_1,\cdots,t_M\}} \sum_{i=1}^{M} |f(x_i) - f(x_{i-1})|$ where $a = t_0 < t_1 < \cdots < t_M = b$. If $||f||_{tv} < \infty$, f is said to be of bounded total variation.

About the convergence speed of a neural network for approximation of a function with a bounded total variation norm, we have Lemma 3 [53].

Lemma 3 *Suppose $f : [a, b] \to R$ is a function of bounded variation. Then there exists a neural network $f_N(x)$ with n neurons in the hidden layer satisfying*

$$||f(x) - f_N(x)||_\infty \leq \frac{||f||_{tv}}{n+1} \tag{3.38}$$

Consider a latitudinal neural network with some assumptions, its convergence property is proven in the following.

Theorem 1 *Suppose $f : [a, b] \to R$ is a function of bounded variation and that $||f-g||_{tv} \leq C||f||_\infty$ where $g \in \mathcal{F}^1_{\sigma_s,n}$ and C is a constant. Then there exists a series of neural networks $f_{N,i}(x)$'s with n neurons in the hidden layer and $i = 1, 2, \cdots, M$ satisfying*

$$||f(x) - \sum_{i=1}^{M} f_{N,i}(x)||_\infty \leq \frac{C^{M-1}||f||_{tv}}{(n+1)^M} \tag{3.39}$$

Proof: Direct application of Lemma 3 to the neural network $f_{N,1}$ yields

$$||f(x) - f_{N,1}(x)||_\infty \leq \frac{||f||_{tv}}{n+1} \tag{3.40}$$

By use of Lemma 3 again, we obtain

$$||f(x) - f_{N,1}(x) - f_{N,2}||_\infty \leq \frac{||f - f_{N,1}||_{tv}}{n+1} \tag{3.41}$$

However, $f_{N,1} \in \mathcal{F}^1_{\sigma_s,n}$. Then with the assumption, we have

$$||f(x) - f_{N,1}(x) - f_{N,2}||_\infty \leq \frac{C||f||_{tv}{}^2}{n+1} \tag{3.42}$$

Repetition of the above procedure gives

$$||f(x) - \sum_{i=1}^{M} f_{N,i}(x)||_\infty \leq \frac{C^{M-1}||f||_{tv}}{(n+1)^M} \tag{3.43}$$

This ends the proof.

In real applications, it is of interest to investigate a fairly large family of functions which are differentiable up to some order. The following definition shows this fact while noting that an analytic function f can be considered as $f \in C^k$ with $k \to +\infty$.

Definition 2: Suppose $f : R^n \to R^m$ is a non-analytic function. If there exist a finite positive integer set $I = \{k_i \mid i = 1, 2, \cdots, M; \ M < +\infty\}$ and the I-indexed functions f_{k_i}'s such that $f = \sum_{i=1}^{M} f_{k_i}(x)$, where $f_{k_i} \in C^{k_i}$ and $f_{k_i} \not\equiv 0$, then we say that f is of finite smooth-decomposition with respect to the index set I.

It is shown in Lemma 4 [53] that differentiability of some degree helps reduce the approximation error by a neural network.

Lemma 4 *Suppose $f : [a, b] \to R$ is a C^2-function. Then there exists a neural network with n neurons in the hidden layer satisfying*

$$\sup_{a \le x \le b} |f(x) - \sum_{i=1}^{n} a_i \sigma(w_i x - \theta_i)| \le \frac{\sup_{a \le x \le b} |f''(x)|}{n^2} (b - a)^2 \tag{3.44}$$

Consider a well-defined function $f \in C^3$; then we have the following results.

Lemma 5 *Suppose $f : [a, b] \to R$ is a C^3-function. Then there exists a neural network with $3n$ neurons in the hidden layer satisfying*

$$\sup_{a \le x \le b} |f(x) - \sum_{i=1}^{3n} a_i \sigma(w_i x - \theta_i)| \le \frac{2 \sup_{a \le x \le b} |f^{(3)}(x)|}{3n^3} (b - a)^3 \tag{3.45}$$

where σ may take the form of σ_q and σ_s.

Proof: Define $\theta_i = a + \frac{i}{n}(b - a)$, $i = 0, 1, \cdots, n$. Let f_N be the piecewise quadratic curve interpolating $(\theta_0, f(\theta_0)), \cdots, (\theta_n, f(\theta_n))$. Through Lemma 1 and Corollary 1, f_N can be realized by a neural network with $2n$ neurons in the hidden layer with quadratic squashing function σ_q, and with n neurons with the soft squashing function σ_s. On $[\theta_{i-1}, \theta_i]$, $f_N(x)$ can be expressed as $f_N(x) = f(\theta_{i-1}) + f'(\theta_{i-1})(x - \theta_{i-1}) + K_i(x - \theta_{i-1})^2$ with $K_i = \frac{[f(\theta_i) - f(\theta_{i-1})] - [\theta_i - \theta_{i-1}] f'(\theta_{i-1})}{(\theta_i - \theta_{i-1})^2}$ such that $f_N(\theta_i) = f(\theta_i)$. Define functions $g_i(x) =$

$f(\theta_{i-1})+(x-\theta_{i-1})f'(\theta_{i-1})+\frac{(x-\theta_{i-1})^2}{2!}f''(\theta_{i-1})$. By Taylor's theorem, $f(x)$ on $[\theta_{i-1},\theta_i]$ can be expanded as $f(x)=f(\theta_{i-1})+(x-\theta_{i-1})f'(\theta_{i-1})+\frac{(x-\theta_{i-1})^2}{2!}f''(\theta_{i-1})+\frac{(x-\theta_{i-1})^3}{3!}f^{(3)}(\xi_i)$ with $\xi_i \in [\theta_{i-1},x] \subseteq [\theta_{i-1},\theta_i]$. Therefore, on $[\theta_{i-1},\theta_i]$, $f(x)-g_i(x)=\frac{(x-\theta_{i-1})^3}{3!}f^{(3)}(\xi_i)$. Further, $|f(x)-g_i(x)| \leq \sup_{x\in[a,b]}\frac{(x-\theta_{i-1})^3}{3!}|f^{(3)}(\xi_i)| = \frac{1}{6}(\theta_i-\theta_{i-1})^3 \sup_{x\in[a,b]}|f^{(3)}(x)|$.

Note that on $[\theta_{i-1},\theta_i]$,

$$|f_N(x)-g_i(x)| = |K_i(x-\theta_{i-1})^2 - \frac{(x-\theta_{i-1})^2}{2!}f''(\theta_{i-1})| \tag{3.46}$$

Since $K_i = \frac{[f(\theta_i)-f(\theta_{i-1}]-[\theta_i-\theta_{i-1}]f'(\theta_{i-1})}{(\theta_i-\theta_{i-1})^2}$, and by Taylor's theorem, $f(\theta_i)=f(\theta_{i-1})+f'(\theta_{i-1})(\theta_i-\theta_{i-1})+\frac{(\theta_i-\theta_{i-1})^2}{2}f''(\mu_i)$,it turns out that

$$K_i = \frac{1}{2}f''(\mu_i) \tag{3.47}$$

where $\mu_i \in [\theta_{i-1},\theta_i]$.

Thus,

$$|f_N(x)-g_i(x)| = |\frac{1}{2}f''(\mu_i)(x-\theta_{i-1})^2 - \frac{1}{2}f''(\theta_{i-1})(x-\theta_{i-1})^2| \tag{3.48}$$

Then

$$f''(\mu_i)-f''(\theta_{i-1}) = (\mu_i-\theta_{i-1})f^{(3)}(\phi_i) \tag{3.49}$$

where $\phi_i \in [\theta_{i-1},\mu_i]$.

Therefore,

$$|f_N(x)-g_i(x)| = \frac{1}{2}(x-\theta_{i-1})^2(\mu_i-\theta_{i-1})|f^{(3)}(\phi_i)| \tag{3.50}$$

Since $\mu_i-\theta_{i-1} \leq \theta_i-\theta_{i-1}$, and $|f^{(3)}(\phi_i)| \leq \sup_{x\in[a,b]}|f^{(3)}(x)|$, it then follows that

$$|f_N(x)-g_i(x)| \leq \frac{1}{2}(\theta_i-\theta_{i-1})^3 \sup_{x\in[a,b]}|f^{(3)}(x)| \tag{3.51}$$

From expressions of $|f_N(x)-g_i(x)|$ and $|f(x)-g_i(x)|$, it turns out that

$$\begin{aligned} |f_N(x)-f(x)| &\leq |f_N(x)-g_i(x)|+|f(x)-g_i(x)| \\ &\leq \frac{2}{3}(\theta_i-\theta_{i-1})^3 \sup_{x\in[a,b]}|f^{(3)}(x)| \end{aligned} \tag{3.52}$$

That is,

$$\sup_{a\le x\le b}|f(x)-\sum_{i=1}^{n}a_i\sigma(w_ix-\theta_i)|\le\frac{2\sup_{a\le x\le b}|f^{(3)}(x)|}{3n^3}(b-a)^3 \tag{3.53}$$

This completes the proof.

Now, we consider a more general case.

Theorem 2 *Suppose $f:[a,b]\to R$ is a C^k-function ($k\ge 3$). Then there exists a neural network with $(2k-3)n$ neurons in the hidden layer satisfying*

$$\begin{aligned}&\sup_{a\le x\le b}|f(x)-\sum_{i=1}^{n}a_i\sigma(w_ix-\theta_i)|\\&\le\frac{(k+1)\sup_{a\le x\le b}|f^{(k)}(x)|}{n^k k!}(b-a)^k\end{aligned} \tag{3.54}$$

where σ may take the form of σ_{p_i} for $3\le i\le k$ and σ_q and σ_s.

Proof: The idea behind the proof to Lemma 5 also applies in the present proof.

Define $\theta_i=a+\frac{i}{n}(b-a)$, $i=0,1,\cdots,n$. Let f_N be the piecewise degree-k polynomial curve interpolating $(\theta_0,f(\theta_0)),\cdots,(\theta_n,f(\theta_n))$. Through Lemma 2 and Corollary 1, f_N can be realized by a neural network with $2n$ neurons in the hidden layer with each of the activation functions σ_{p_i} for $3\le i\le k$ and σ_q, and with n neurons with the soft squashing function σ_s. On $[\theta_{i-1},\theta_i]$, $f_N(x)$ can be expressed as $f_N(x)=f(\theta_{i-1})+\sum_{j=1}^{k-2}\frac{f^{(j)}(\theta_{i-1})}{j!}(x-\theta_{i-1})^j+K_i(x-\theta_i)^{k-1}$ with $K_i=\frac{[f(\theta_i)-f(\theta_{i-1})]-\sum_{j=1}^{k-2}\frac{f^{(j)}(\theta_{i-1})}{j!}(\theta_i-\theta_{i-1})^j}{(\theta_i-\theta_{i-1})^{k-1}}$ such that $f_N(\theta_i)=f(\theta_i)$. Define functions $g_i(x)=f(\theta_{i-1})+\sum_{j=1}^{k-1}\frac{f^{(j)}(\theta_{i-1})}{j!}(x-\theta_{i-1})^j$. By Taylor's theorem, $f(x)$ on $[\theta_{i-1},\theta_i]$ can be expanded as $f(x)=f(\theta_{i-1})+\sum_{j=1}^{k-1}\frac{f^{(j)}(\theta_{i-1})}{j!}(x-\theta_{i-1})^j+\frac{(x-\theta_{i-1})^k}{k!}f^{(k)}(\xi_i)$ with $\xi_i\in[\theta_{i-1},x]\subseteq[\theta_{i-1},\theta_i]$. Therefore, on $[\theta_{i-1},\theta_i]$, $f(x)-g_i(x)=\frac{(x-\theta_{i-1})^k}{k!}f^{(k)}(\xi_i)$. Further, $|f(x)-g_i(x)|\le\sup_{x\in[a,b]}\frac{(x-\theta_{i-1})^k}{k!}|f^{(k)}(\xi_i)|=\frac{1}{k!}(\theta_i-\theta_{i-1})^k\sup_{x\in[a,b]}|f^{(k)}(x)|$.

Note that on $[\theta_{i-1},\theta_i]$,

$$\begin{aligned}|f_N(x)-g_i(x)|=\quad&|K_i(x-\theta_{i-1})^{k-1}\\&-\frac{(x-\theta_{i-1})^{k-1}}{(k-1)!}f^{(k-1)}(\theta_{i-1})|\end{aligned} \tag{3.55}$$

Since $K_i = \frac{[f(\theta_i)-f(\theta_{i-1}]-\sum_{j=1}^{k-2}\frac{f^{(j)}(\theta_{i-1})}{j!}(\theta_i-\theta_{i-1})^j}{(\theta_i-\theta_{i-1})^{k-1}}$, and by Taylor's theorem, $f(\theta_i) = f(\theta_{i-1})$ $+ \sum_{j=1}^{k-2}\frac{f^{(j)}(\theta_{i-1})}{j!}(\theta_i-\theta_{i-1})^j + \frac{(\theta_i-\theta_{i-1})^{k-1}}{(k-1)!}f^{(k-1)}(\mu_i)$, it can be readily shown that

$$K_i = \frac{1}{(k-1)!}f^{(k-1)}(\mu_i) \tag{3.56}$$

where $\mu_i \in [\theta_{i-1}, \theta_i]$.

Thus,

$$\begin{aligned} |f_N(x) - g_i(x)| = \quad & |\frac{1}{(k-1)!}f^{(k-1)}(\mu_i)(x-\theta_{i-1})^{k-1} \\ & -\frac{1}{(k-1)!}f^{(k-1)}(\theta_{i-1})(x-\theta_{i-1})^{k-1}| \end{aligned} \tag{3.57}$$

Then

$$f^{(k-1)}(\mu_i) - f^{(k-1)}(\theta_{i-1}) = (\mu_i - \theta_{i-1})f^{(k)}(\phi_i) \tag{3.58}$$

where $\phi_i \in [\theta_{i-1}, \mu_i]$.

Therefore,

$$|f_N(x) - g_i(x)| = \frac{1}{(k-1)!}(x-\theta_{i-1})^{k-1}(\mu_i - \theta_{i-1})|f^{(k)}(\phi_i)| \tag{3.59}$$

Since $\mu_i - \theta_{i-1} \leq \theta_i - \theta_{i-1}$, and $|f^{(k)}(\phi_i)| \leq \sup_{x\in[a,b]}|f^{(k)}(x)|$, it then follows that

$$|f_N(x) - g_i(x)| \leq \frac{1}{(k-1)!}(\theta_i - \theta_{i-1})^k \sup_{x\in[a,b]}|f^{(k)}(x)| \tag{3.60}$$

From expressions of $|f_N(x) - g_i(x)|$ and $|f(x) - g_i(x)|$, it turns out that

$$\begin{aligned} & |f_N(x) - f(x)| \leq |f_N(x) - g_i(x)| + |f(x) - g_i(x)| \\ & \leq \frac{k+1}{k!}(\theta_i - \theta_{i-1})^k \sup_{x\in[a,b]}|f^{(k)}(x)| \end{aligned} \tag{3.61}$$

That is,

$$\begin{aligned} & \sup_{a\leq x\leq b}|f(x) - \sum_{i=1}^{n} a_i\sigma(w_i x - \theta_i)| \\ & \leq \frac{(k+1)\sup_{a\leq x\leq b}|f^{(k)}(x)|}{n^k k!}(b-a)^k \end{aligned} \tag{3.62}$$

This completes the proof.

Theorem 3 *Suppose $f : [a,b] \rightarrow R$ is a function of finite smooth-decomposition with respect to the index set $I = \{k_i \mid i = 1,2,\cdots,M;\ M < +\infty\}$. Then there exists a latitudinal neural network structure whose each component, a sub-neural network f_{N,k_i}, either has $(2k_i - 3)n_i$ neurons for $k_i \geq 3$ in the hidden layer or has n_i neurons for $k_i = 2$ satisfying*

$$\begin{aligned}
&\sup_{a\leq x\leq b} |f(x) - \sum_{i=1}^{M} f_{N,k_i}| \\
&\leq \sum_{i=1,k_i\geq 3}^{M} \frac{(k_i+1)\sup_{a\leq x\leq b}|f^{(k_i)}(x)|}{n_i^{k_i}k_i!}(b-a)^{k_i} \\
&+ \sum_{i=1,k_i=2}^{M} \frac{\sup_{a\leq x\leq b}|f^{(k_i)}(x)|}{n_i^{k_i}}(b-a)^{k_i}
\end{aligned} \tag{3.63}$$

Proof: Applictions of Lemma 4 and Theorem 2 will immediately give the proof.

3.3.3. General results on multi-dimension cases

Consider a function $f : R^m \longrightarrow R^n$. Since this function can be viewed as n functions from $R^m \longrightarrow R$, without loss of generality, we only consider a function from $R^m \longrightarrow R$.

Suppose there exist continuous partial derivatives of up to order $n+1$ in a neighborhood of x_0 for function $f(x)$ with a compact support Ω and $x \in \Omega \subseteq R^m$. x can be written in terms of its components, that is, $x = [x^1 \cdots x^m]^T$.

First of all, suppose $f(.) \in C^2$. consider the first-order expansion of $f(x)$ at x_0. we have

$$f(x) = f(x_0) + (\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})f(x_0) + R_1 \tag{3.64}$$

where $R_1 = \frac{1}{2!}(\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})^2 f(x_0 + \theta(x - x_0))$ $(0 < \theta < 1)$ and $x_0 \in \Omega$.

Define $g(x) = f(x_0) + (\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})f(x_0)$. Note that $g(x)$ is an affine-linear function, and thus can be implemented by a constructive neural network [47]. This implementation, however, is only valid in a small neighborhood of x_0. To make a proper

approximation in the region Ω of interest, tessellation of the region is performed. Suppose that the support Ω is compact. This implies that each $x^i \in [x^i_{min},\ x^i_{max}]$. Tessellation of the support Ω generates many non-overlapping hyper-rectangles $\omega^{j_1 j_2 \cdots j_m} = [x^{1,j^1_1}, x^{1,j^2_1}] \times [x^{2,j^1_2}, x^{2,j^2_2}] \times \cdots \times [x^{m,j^1_m}, x^{m,j^2_m}] \subset \Omega$ where $x^{i,j^k_i} = x^i_{min} + (j^k_i - 1)\frac{x^i_{max} - x^i_{min}}{N_i - 1}$ and N_i is the number of grid points for the ith axis. On each of such hyper-rectangles $\omega^{j_1 j_2 \cdots j_m}$, expand function $f(.)$ around $x^{j_1 j_2 \cdots j_m} \in \omega^{j_1 j_2 \cdots j_m}$, which can be just one vertex of this rectangle. Let $g^{j_1 j_2 \cdots j_m}(.)$ define the affine linear part of $f(.)$ around $x^{j_1 j_2 \cdots j_m}$. Note that $g^{j_1 j_2 \cdots j_m}(.)$ on the support of $\omega^{j_1 j_2 \cdots j_m}$ can be implemented by a constructive neural network such that on the grid points, the neural network generates outputs which exactly match the function values of $f(.)$, and for points inside $\omega^{j_1 j_2 \cdots j_m}$, linear interpolation is performed, and further for points outside $\omega^{j_1 j_2 \cdots j_m}$, the constructive neural network generates outputs of a value 0. Piecewise linear approximation by a number of constructive neural networks can then be achieved. The approximation error can be given by

$$R_1 = \frac{1}{2!}\left(\sum_{i=1}^{m}(x^i - x^{i,j_1 j_2 \cdots j_m})\frac{\partial}{\partial x^i}\right)^2 f(x^{j_1 j_2 \cdots j_m} + \theta(x - x^{j_1 j_2 \cdots j_m})) \tag{3.65}$$

where $0 < \theta < 1$, and $x^{i,j_1 j_2 \cdots j_m}$ designates the ith component of $x^{j_1 j_2 \cdots j_m}$.

Since $x, x^{j_1 j_2 \cdots j_m} \in \omega^{j_1 j_2 \cdots j_m}$, $|x^i - x^{i,j_1 j_2 \cdots j_m}| \leq \frac{x^i_{max} - x^i_{min}}{N_i - 1}$.

Since $f(.) \in C^2$, all the second-order partial-derivatives are bounded by $B_1 = \sup_{x \in \Omega;\ 1 \leq i,j \leq m} \frac{\partial^2 f}{\partial x^i \partial x^j}$.

Thus,

$$R_1 \leq \frac{1}{2!} B_1 \left(\sum_{i=1}^{m} \frac{x^i_{max} - x^i_{min}}{N_i - 1}\right)^2 \tag{3.66}$$

Note that $\omega^{j_1 j_2 \cdots j_m}$ can be decomposed into at most $m!$ hyper-triangles on which a constructive neural network granule (NNG-t) can be applied to approximate the function $f(.)$ defined on it. Therefore, at most $\Pi^m_{i=1}(N_i - 1)m!$ constructive neural network granules are needed. As is detailed in [47], a constructive neural network granule NNG-t is a neural network with m hard-limiting neurons with an activation function of $\sigma_h(.)$ and 1 soft-limiting neuron with an activation function of $\sigma_s(.)$ in the first hidden layer, and

2 soft-limiting neurons in the second hidden neuron, and one output neuron performing linear operations.

This leads to the following conclusion:

Proposition 2 *Suppose that function* $f(.) : \Omega \subset R^m \longrightarrow R$ *with a compact support* Ω *is a* C^2 *function. Then there exists a constructive neural network* $NN(.)$ *which consists of at most* $\Pi_{i=1}^{m}(N_i - 1)m!$ *constructive neural network granules NNG-t such that*

$$|NN(x) - f(x)| \leq \frac{1}{2!}B_1(\sum_{i=1}^{m}\frac{x_{max}^i - x_{min}^i}{N_i - 1})^2 \tag{3.67}$$

where $x_{max}^i = \sup_{x\in\Omega} x^i$; $x_{min}^i = \inf_{x\in\Omega} x^i$; *and* $B_1 = \sup_{x\in\Omega;\, 1\leq i,j\leq m} \frac{\partial^2 f}{\partial x^i \partial x^j}$.

Next consider function $f(.) \in C^3$. According to Taylor's Theorem, expansion of $f(x)$ around x_0 yields

$$f(x) = f(x_0) + \sum_{j=1}^{2}\frac{1}{j!}(\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})^j f(x_0) + R_2 \tag{3.68}$$

where $R_2 = \frac{1}{3!}(\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})^3 f(x_0 + \theta(x - x_0))$ $(0 < \theta < 1)$ and $x_0 \in \Omega$.

Define $g(x) = f(x_0) + \sum_{j=1}^{2}\frac{1}{j!}(\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})^j f(x_0)$. Note that the affine linear part of $g(.)$ can be realized by a constructive neural network composed of a number of NNG-t's. We now deal with the quadratic part of $g(.)$, which is denoted by $g_q(.)$. $g_q(.)$ can be rewritten in a matrix form. That is,

$$g_q(x) = x^T G x \tag{3.69}$$

There exists an orthogonal matrix U such that $\Lambda = U^T G U$ is a diagonal matrix. Hence, by means of a transform $x = Uz$ with $z \in R^m$ function, function $g_q(x)$ can be expressed in terms of z such that $g_q(x) = z^T \Lambda z = \sum_{i=1}^{m}\lambda_i z_i^2$ where z_i is the ith component of z. The support Ω is tessellated into a number of non-overlapping hyper-rectangles $\omega^{j_1 j_2 \cdots j_m}$ defined previously. On each $\omega^{j_1 j_2 \cdots j_m}$, second-order expansion of $g(.)$ and the corresponding orthogonization transformation $U^{j_1 j_2 \cdots j_m}$ are performed.

As we discussed in the previous section, the z_i^2's terms can be implemented by means of a neual network consisting of neurons with a quadratic squashing activation function $\sigma_q(.)$. It should be noted, however, that the non-overlapping hyper-rectangles in terms of x may no longer be non-overlapping hyper-rectangles in terms of z. Therefore, a binary logic is necessary to deal with this situation such that when x is inside $\omega^{j_1 j_2 \cdots j_m}$, the binary logic turns on so that the quadratic terms are included to give a quadratic approximation but when x is outside $\omega^{j_1 j_2 \cdots j_m}$, the binary logic turns off so that the quadratic terms are not included to give a total output of 0. Fortunately, as shown for a one-dimensional case in the previous section, the output of the far left soft-limiting neuron in the second-hidden layer of an NNG-t precisely performs such a logic since the total input to the activation function $\sigma_s(.)$ of this neuron is always an integer resulting in a binary logic even though the activation function itself is not a binary logic at all. The combination of the neural network achieving affine-linear approximation and the neural network achieving the pure quadratic approximation results in a neural network granule, which is called a modified constructive neural network granule MNNG-t if this neural network granule has a desired support, hyper-triangle or MNNG-r if this neural network granule has a desired support, hyper-rectangle.

The resulting approximation error can be given by

$$R_2 = \frac{1}{3!}(\sum_{i=1}^{m}(x^i - x^{i,j_1 j_2 \cdots j_m})\frac{\partial}{\partial x^i})^3 f(x^{j_1 j_2 \cdots j_m} + \theta(x - x^{j_1 j_2 \cdots j_m})) \tag{3.70}$$

Since $x, x^{j_1 j_2 \cdots j_m} \in \omega^{j_1 j_2 \cdots j_m}$, $|x^i - x^{i,j_1 j_2 \cdots j_m}| \leq \frac{x^i_{max} - x^i_{min}}{N_i - 1}$.

Since $f(.) \in C^3$, all the third-order partial-derivatives are bounded by $B_2 = \sup_{x \in \Omega;\, 1 \leq i,j,k \leq m} \frac{\partial^3 f}{\partial x^i \partial x^j \partial x^k}$.

Thus,

$$R_2 \leq \frac{1}{3!} B_2 (\sum_{i=1}^{m} \frac{x^i_{max} - x^i_{min}}{N_i - 1})^3 \tag{3.71}$$

We obtain the following conclusion:

Proposition 3 *Suppose that function* $f(.) : \Omega \subset R^m \longrightarrow R$ *with a compact support* Ω *is a* C^3 *function. Then there exists a constructive neural network* $NN(.)$ *which consists of at most* $\Pi_{i=1}^{m}(N_i - 1)m!$ *modified constructive neural network granules MNNG-t such that*

$$|NN(x) - f(x)| \leq \frac{1}{3!}B_2(\sum_{i=1}^{m}\frac{x_{max}^i - x_{min}^i}{N_i - 1})^3 \tag{3.72}$$

where $x_{max}^i = \sup_{x\in\Omega} x^i$; $x_{min}^i = \inf_{x\in\Omega} x^i$; *and* $B_2 = \sup_{x\in\Omega;\, 1\leq i,j,k\leq m} \frac{\partial^3 f}{\partial x^i \partial x^j \partial x^k}$.

It is desired that instead of using neurons with $\sigma_q(.)$ as the activation function for approximation of the quadratic polynomials, using neurons with σ_{p_n} $(n \geq 3)$ as the activation function for approximation of higer order polynomials could be done. It turns out that such a generalization may not be easily implemented. The reason may be that use of the orthogonal transformation for quadratic polynomials can not be generalized to higher-order terms.

Note that for quadratic nonlinear approximation, there is no need for the cross-product terms to be inputs of a neural network. However, to deal with the case with higher-order expansion of a function, using the cross-product terms as inputs to a neural network may be necessary, which is illustrated next.

Suppose $f(.) \in C^{n+1}$ with a compact support Ω. According to Taylor's Theorem, expansion of $f(x)$ around x_0 yields

$$f(x) = f(x_0) + \sum_{j=1}^{n}\frac{1}{j!}(\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})^j f(x_0) + R_n \tag{3.73}$$

where $R_n = \frac{1}{(n+1)!}(\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})^{(n+1)} f(x_0 + \theta(x - x_0))$ $(0 < \theta < 1)$ and $x_0 \in \Omega$.

Define $g(x) = f(x_0) + \sum_{j=1}^{n}\frac{1}{j!}(\sum_{i=1}^{m}(x^i - x_0^i)\frac{\partial}{\partial x^i})^j f(x_0)$. Note that there are non-linear terms, cross-product terms, which can not be well approximated by NNG-t's.

As discussed before, the support Ω is tessellated into a number of non-overlapping hyper-rectangles $\omega^{j_1 j_2 \cdots j_m}$ defined previously. Each hyper-rectangle is supported by 2^m vertices. On each hyper-rectangle, a constructive neural network granule (NNG-r) can be

utilized to give a proper approximation to $g(x)$, though it requires all the cross-product terms as inputs. Refer to reference [47] for details. The resulting approximation error can be given by

$$R_n = \frac{1}{(n+1)!}(\sum_{i=1}^{m}(x^i - x^{i,j_1 j_2 \cdots j_m})\frac{\partial}{\partial x^i})^{(n+1)} f(x^{j_1 j_2 \cdots j_m} + \theta(x - x^{j_1 j_2 \cdots j_m})) \tag{3.74}$$

Since $x, x^{j_1 j_2 \cdots j_m} \in \omega^{j_1 j_2 \cdots j_m}$, $|x^i - x^{i,j_1 j_2 \cdots j_m}| \leq \frac{x^i_{max} - x^i_{min}}{N_i - 1}$.

Since $f(.) \in C^{n+1}$, all the $(n+1)$th-order partial-derivatives are bounded by $B_n = \sup_{x \in \Omega;\, 1 \leq i_1,\, i_2,\, \cdots,\, i_{n+1} \leq m} \frac{\partial^{n+1} f}{\partial x^{i_1} \partial x^{i_2} \cdots \partial x^{i_{n+1}}}$ where $1 \leq i_1,\ i_2,\ \cdots,\ i_{n+1} \leq m$.

Thus,

$$R_n \leq \frac{1}{(n+1)!} B_n (\sum_{i=1}^{m} \frac{x^i_{max} - x^i_{min}}{N_i - 1})^{n+1} \tag{3.75}$$

We obtain the following conclusion:

Proposition 4 *Suppose that function $f(.) : \Omega \subset R^m \longrightarrow R$ with a compact support Ω is a C^{n+1} function. Then there exists a constructive neural network $NN(.)$ which consists of at most $\Pi_{i=1}^{m}(N_i - 1)$ constructive neural network granules NNG-r such that*

$$|NN(x) - f(x)| \leq \frac{1}{(n+1)!} B_n (\sum_{i=1}^{m} \frac{x^i_{max} - x^i_{min}}{N_i - 1})^{n+1} \tag{3.76}$$

where $x^i_{max} = \sup_{x \in \Omega} x^i$; $x^i_{min} = \inf_{x \in \Omega} x^i$; and

$B_n = \sup_{x \in \Omega;\, 1 \leq i_1,\, i_2,\, \cdots,\, i_{n+1} \leq m} \frac{\partial^{n+1} f}{\partial x^{i_1} \partial x^{i_2} \cdots \partial x^{i_{n+1}}}$ with $1 \leq i_1,\ i_2,\ \cdots,\ i_{n+1} \leq m$.

3.3.4. Comments

This section demonstrates the relationship of neurons used in the hidden layer and the achieved precision. Further, by using a latitudinal neural network architecture, the approximation of a given function, with the finite smooth-decomposition property, can be effectively accomplished in a constructive manner. The significance of the achieved theoretical results is that it may lay a foundation for better modeling in identification and control.

3.4. Conclusions

The architecture of latitudinal neural networks is proposed, and its relevant convergence property is investigated, and further the approximation of a given function, with the finite smooth-decomposition property, can be effectively accomplished in a constructive manner. The theoretical results are generalized to multi-dimension cases. They may be useful for dynamic system modeling or static mapping.

4. LOAD MODELING AND VOLTAGE STABILITY ANALYSIS

4.1. Introduction

The power flow based static techniques still prevail on voltage stability analysis in many utilities since they are simple, fast, and convenient to use [54, 55, 56]. The quasi-static techniques (e.g.,small-disturbance analysis) have also been widely applied [57, 58, 59]. Those static or quasi-static methods are used either for estimating the static voltage stability indices or determining the robustness and stability patterns of the systems to be examined. Without question, they are very useful for on-line assessment which will give operators rich information about the current operation status of power systems. However, the disadvantages are also apparent; for instance, the static-technique based voltage stability analysis needs further confirmation by using time-domain simulation. And voltage collapse may occur well before the critical point predicted by steady-state power flow study. More importantly, with dynamic interaction of various loads with different characteristics initiated by heavy load buildup, line trip, etc., still using steady-state or quasi-steady-state analysis may give misleading results.

It should also be noted that maintaining a good voltage stability profile does not automatically guarantee voltage stability and that voltage instability need not be associated with low voltage [60]. For the former case, there are many such situations that no abnormal advance warning appears for bus voltages but all of a sudden voltage instability or even voltage collapse comes up. For the latter case, it is interesting to see that a voltage collapse occurred in Western France on January 12, 1987, but the voltages stabilized at low levels rather than completely collapsing [14].

Voltage instability covers different time frames. Transient voltage instability is usually closely associated with regular generator angle stability, which has been well studied (e.g. [61] and therein). But traditional voltage collapse or related voltage instability also may be more closely associated with loading dynamics. This kind of voltage instability has stimulated extensive research on voltage stability analysis methods involving quasi-static power flow or continuation power flow [55], snapshot method [62], modal analysis[56],load dynamics [63, 64], energy method[54, 65], static bifurcation theory's application to power systems [66, 67] and so on. By means of many methods listed above (e.g., continuation power flow), voltage collapse is viewed as a rather slow dynamic process, and can be treated as a quasi-static process or quasi-steady state process, which equivalently treat loads as constant load, constant impedance or, at worst constant current [60]. Such a viewpoint of load is very indicative in many publications [65, 68, 62] which, however, ignore load dynamics. As is argued in [69], the use of static load model for loads, combined with the dynamics of underload (or overload) tap-changer transformer, will give rise to rendered dynamics [70]. This, however, does not essentially change the feature of loads which are still static in nature. The importance of using a dynamic model for dynamic voltage stability analysis is further discussed in [71, 72, 73, 74]. A static load model incorporated into dynamic voltage stability analysis may conceivably lead to impractical results.

Therefore, static techniques and dynamic methods should be coordinated to give accurate and timely results.

To better understand the mechanisms of voltage collapses, the need for dynamic voltage stability analysis is most important [14, 63]. It is stated that voltage collapse is a dynamic phenomenon in nature, and that it is closely associated with overall (or predominant) dynamic characteristics of loads connected to a specific bus (say, a weak load bus) [14]. With complex composition of loads with different dynamics, it may be very difficult, if not impossible, to establish the time-varying dynamic interaction of all those

loads connected to a load bus. However, modeling of each and every load component is not practical. Use of aggregate models describing the overall dynamics might be possible. Motivated by this fact, the dynamics of all down-stream loads and voltage control equipment were modeled as a generic dynamic load model with many factors contributing to voltage collapse simplified or ignored [75]. Basic load dynamics and models are studied in [63, 75, 64].

It should be emphasized here again that load characteristics should be very important when voltage instability or voltage collapse is involved. How to deal with load characteristics in different stages of power systems is critical for better voltage stability analysis.

Usually the component-based load model is used in some utilities. This load modeling approach is very much dependent on the accurate statistics of various power-consuming devices. Because of its simplicity, a slightly more general form of exponential load models is widely applied. This choice, however, has no theoretical guarrantees since a combination of some exponential terms is often not likely to give a good approximation. Furthermore, dynamic loading characteristics is too complex to be expressed in a simple analytic form. Neural networks may be an appropriate choice. The use of a neural network for approximating a first-order load model is suggested in [21]. The aim of this chapter is to propose to use a recurrent neural network which is capable of approximating a high-order load model in general, and then to incorporate such a neural network model in voltage stability analysis.

In this chapter, neural networks will be applied for modeling the static load characteristics and dynamic load flow. These models will be used with the conventional power flow study. The resulting Jacobian will be used for judgement of power system stability. In a word, the whole methodology will make use of a neural network model for voltage stability analysis and assessment. In the remainder of this chapter, some representative static/quasi-static/perturbation analysis methods are presented in section 4.2.. Load

modeling through neural networks is discussed in section 4.3.. With the resulting neural network load models, dynamic and static voltage analyses are presented in section 4.4.. Finally, some concluding remarks are given.

4.2. Typical voltage stability analysis

In this section, static, quasi-steady state, and dynamic voltage stability analyses will be presented. It will be shown that simple checking with either the relevant eigenvalues or the sign of some parameters will provide information about the operation status of the system. As is mentioned before, voltage instability may occur before the predicted critical point.

4.2.1. Static voltage stability analysis

Static voltage stability analysis is important. It is particularly true when on-line dynamic voltage stability assessment is not available. Typical static techniques may involve a conventional power flow study. P-V or Q-V curves should be useful in power system planning. It is well understood that reactive power transmission is inefficient for transferring high real power, and that adequate reactive power supply nearby the heavy loading area is very helpful for maintaining good voltage profiles. The P-V or Q-V curves and their sensitivities ($\frac{\partial P}{\partial V}$ or $\frac{\partial Q}{\partial V}$) gives operators information on the relative robustness of load buses and on "how far away" (in some physical sense) a specific bus voltage is from the potential voltage collapse point. To this end, there are also many voltage stability indices calculated by estimating the distance from the current operation state to the maximum voltage stability limit point or sometimes the maximum loadability limit point, either of which may coincide with the other in some cases, and is usually called a singular point in that the Jacobian becomes singular at this point such that it might be impossible

for conventional power flow study to go further. Modified power flow study is needed. A more effective approach — modal analysis is proposed in [56], which in some cases can provide a clear indication of weak voltage areas while V-Q sensitivities method may not. A brief description of this method is given in the following.

Generally, steady state power flow at each bus k can be described by

$$\begin{cases} P_k = f(v, \theta) \\ Q_k = g(v, \theta) \end{cases} \tag{4.1}$$

where P_k's, the real power, and Q_k's, the reactive power, are functions of the magnitude vector v and phase angle vector θ of relevant bus voltages.

Conveniently, these equations can be rewritten as

$$S = h(v, \theta) \tag{4.2}$$

where $S = [P\ Q]^\tau$, $h = [f\ g]^\tau$; P is a row vector whose kth componenet is P_k; Q is a row vector whose kth componenet is Q_k.

The linearized version of Equation (4.2) is given by

$$\delta S = J \delta x \tag{4.3}$$

where $\delta S = [\delta P\ \ \delta Q]^\tau$; $x = [\delta\theta\ \ \delta v]^\tau$; $J = \frac{\partial h}{\partial x}$ is the Jacobian of function h with respect to x.

It is a common consensus that the static voltage instability is usually associated with insufficient supply of reactive power. Therefore, to relate the variations of reactive powers to the variations of voltage magnitudes, let $\delta P = 0$. It then follows that

$$\delta Q = J_R \delta v \tag{4.4}$$

where $J_R = \frac{\partial g}{\partial v} - \frac{\partial g}{\partial \theta}(\frac{\partial f}{\partial \theta})^{-1}\frac{\partial f}{\partial v}$ is the reduced Jacobian.

Diagonalizing J_R to diagnonal matrix Λ by using a similarity transform yields

$$J_R = \phi \Lambda \psi \tag{4.5}$$

where ϕ and ψ are square matrices of full rank such that $\phi\psi = I$ with I designating the unitary matrix.

That is,

$$J_R^{-1} = \phi\Lambda^{-1}\psi \tag{4.6}$$

It follows that

$$\delta v = \sum_i \frac{\phi_i\psi_i}{\lambda_i}\delta Q \tag{4.7}$$

where ϕ_i is the ith column vector of ϕ; ψ_i is the ith row vector of ψ.

If $\delta Q = \frac{\phi_i}{||\phi_i||}$, then $\delta V = \sum_j \frac{\phi_i\psi_i}{\lambda_i}\frac{\phi_i}{||\phi_i||} = \frac{\phi_i}{||\phi_i||} = \frac{\delta Q}{\lambda_i}$.

Remark: (1) Each mode corresponds to an eigenvalue λ_i. It can be seen that if $\lambda_i = 0$, then any change of reactive power vector in the dirction of the associated eigenvector will lead to infinite change of voltage magnitude vector. This would be exactly the voltage collapse point according to this linearized analysis.

(2) If all the eigenvalues λ_i's are positive (they are real since the reduced Jacobian matrix is symmetric), then the power system under investigation can be considered stable.

(3) Sensitivities can be readily shown to be $\frac{\partial V_k}{\partial Q_k} = \sum_i \frac{\phi_{ki}\psi_{ik}}{\lambda_i}$ with ϕ_{ki} and ψ_{ik} representing the kth element of ϕ_i and ψ_i, respectively.

(4) Participation factor $P_{ki} = \phi_{ki}\psi_{ik}$ determines the bus k's participation in mode i.

(5) Other sensitivities also can be obtained. That is, $\frac{\partial V_m}{\partial Q_n} = \sum_i \frac{\phi_{mi}\psi_{in}}{\lambda_i}$. These sensitivities reflect how any reactive power of one bus influences any other bus voltage.

An energy method [54, 65] is noteworthy in that it gives the energy difference between the operating point and the likely voltage collapse point. This method is associated with multiple power flow solutions. With the given initial operating conditions, it usually is possible to determine a unique stable equilibrium point (SEP) of interest while the number of possible unstable (and stable) equilibrium points (UEP) may be very large. Generally, for n bus power systems, there are 2^{n-1} possible solutions. With more practical

considerations, this large number can be reduced to $n-1$ for UEP (saddle points). It is interesting to note that at the point immediately before collapse only a pair of closely located solutions exist [54, 65]. And an algorithm proposed in [76] can be used to locate the low voltage solution paired to the high voltage solution, which requires a certain amount of time equivalent to that for convention power flow. The energy difference associated with such a pair of closely located solutions is given further in [76].

By such an energy measure, when power systems are operating at some point very close to a possible voltage collapse point, operators may know by how much more power injection the power systems may escape from such a situation. However, it does not directly answer such questions like "how much real/reactive power is needed and how are these powers distributed?".

An interesting application of neural networks to voltage stability assessment can be seen in [77]. It is based on the energy method mentioned above. A feedforward neural network with backpropagation algorithm is trained to approximate the mapping $f : R^{2n} \longrightarrow R$ with the voltage stability margin $VSM = f(v, \theta)$ where v is the voltage magnitude vector and θ is the voltage phase angle vector. Function f can be considered as composite $h(g(.))$ of function g and h. Here, functions g and h can be expressed as the following mappings.

$$g : \{(P_d, Q_d, P_g, Q_g, V_g)\} \longrightarrow \{(v^s, \alpha^s, v^u, \alpha^u)\}.$$

$$h : \{(v^s, \alpha^s, v^u, \alpha^u)\} \longrightarrow Energy\ \ Margin.$$

Here P_d,Q_d,P_g,Q_g and V_g are real power demands of loads, reactive power demands of loads, real power supplies of generators, reactive power supplies of generators, and voltages at the generator buses, respectively; v^s and α^s are the magnitudes and phase angles of voltages corresponding to a stable solution; and v^u and α^u are the magnitudes and phase angles of voltages corresponding to an unstable solution.

By calculating sensitivities through other methods beforehand, the number of input variables can be very much reduced. The results are shown to be very much the same as the case without input number reduction [77].

4.2.2. Quasi-steady state voltage stability analysis

To overcome the difficulty encountered by conventional power flow at the voltage collapse point, and to consider the slow changes in both generation and loading, a continuation power flow method [62] is applied. To reflect the slow changes in both generation and loading, a single load parameter λ is assumed so that $\lambda = 0$ corresponds to the base load flow and $\lambda = \lambda_c$ corresponds to the critical point (a saddle point). Then the generation and loading may be modified in such a way that

$P_{di} = P_{di0} + \lambda K_{di} S_{\delta base} \cos(\psi_i)$;

$Q_{di} = Q_{di0} + \lambda K_{di} S_{\delta base} \sin(\psi_i)$;

$P_{gi} = P_{gi0}(1 + \lambda K_{gi})$ where P_{di0}, Q_{di0} and P_{gi0} correspond to real power demand, reactive power demand, and real power generation in the ith bus for the base case, respectively;K_{di} and K_{gi} stand for load and generation change rate in the ith bus, respectively; ψ_i represents load change power factor angle; and $S_{\delta base}$ is a given quantity of apparent power.

Then the power flow equations can be given by

$$0 = P_{gi0}(1 + \lambda K_{gi}) - P_{di0} - \lambda(K_{di} S_{\delta base} \cos(\psi_i) - \sum_j |v_i||v_j||y_{ij}| \cos(\delta_i - \delta_j - \theta_{ij}) \quad (4.8)$$

and

$$0 = Q_{gi0} - Q_{di0} - \lambda(K_{di} S_{\delta base} \sin(\psi_i) - \sum_j |v_i||v_j||y_{ij}| \sin(\delta_i - \delta_j - \theta_{ij}) \quad (4.9)$$

The above equation can be rewritten in a compact form such that

$$F(\delta, v, \lambda) = 0 \quad (4.10)$$

where δ, v and λ represent generator angle vector, bus voltage vector and load parameter, respectively.

It follows from $dF = 0$ that

$$\begin{bmatrix} F_\delta & F_v & F_\lambda \end{bmatrix} \begin{bmatrix} d\delta \\ dv \\ d\lambda \end{bmatrix} = 0 \tag{4.11}$$

The linear prediction can be then given by

$$\begin{bmatrix} \delta^{new} \\ v^{new} \\ \lambda^{new} \end{bmatrix} = \begin{bmatrix} \delta^{old} \\ v^{old} \\ \lambda^{old} \end{bmatrix} + \alpha \begin{bmatrix} d\delta \\ dv \\ d\lambda \end{bmatrix} \tag{4.12}$$

where α is a weighting coefficient.

Another important step in continuation power flow is correction. The prediction and correction procedures alternatively proceed. When λ approaches λ_c, $d\lambda$ will approach zero, and then may change sign after the critical point is passed. Thus, the critical point can be checked out by noticing that a test of the sign of the $d\lambda$ component will reveal whether or not the critical point has been passed.

4.2.3. Dynamic voltage stability analysis

As is well known, generally there are two kinds of disturbances which may give rise to voltage instability, event driven ones and load driven ones. As is pointed out in [71], traditional voltage instability is manifested at load buses, and is mainly load driven. In [78], it is stated that event driven causes may include generator outages, short-circuits caused by lightning, sudden large load changes, or a combination of such events.

In [66], bifurcation theory and perturbation methods are used to discuss voltage stability and categorize voltage stabilities into four kinds, i.e., Type I instability, Type

II-1 instability, Type II-2S and Type II-2D instability. Power systems can sometimes be divided into two subsystems: slow response subsystem and fast response subsystem.

An underload tap-changing transformer may be an example of slow response subsystems while loads with fast response dynamics, generator and AVR may be fast response subsystems. If both the slow response subsystem and the fast response subsystem are stable, then the whole power system has the ability to restore the voltage after voltage dip and the associated voltage stability is called Type I instability. Otherwise, the resulting voltage stability may be called Type II instability. For this case, voltage collapse may occur. The voltage stability associated with slow response subsystems is called Type II-1 instability. The voltage stability associated with fast response subsystems is called Type II-2 instability, which is further categorized into two kinds, II-2S and II-2D, which are associated with static bifurcation and dynamic bifurcation, respectively. These results are very useful under some specific occasions, e.g., known load distribution. However, there is no systematic way for recognition of fast subsystems and slow subsystems, and the division of fast subsystems and slow subsystems may be changing with time or by occurrence of contingencies. Again, load dynamics is not considered in [66].

4.2.4. Comments

In section 4.2., several voltage-stability analytical methods are introduced. They are essentially either static techniques or small signal perturbation analyses. Yet the dynamics of loads has not been considered. At best, the rendered dynamics [70] was considered. As is known, the load dynamics contributes most to the load side voltage instability problems. Therefore, inclusion of load dynamics in voltage stability analysis is important. This will be addressed in more detail together with the study on the load modeling issue.

4.3. Load modeling

A load is defined in [79] as such: a load is a portion of the system that is not explicitly represented in a system model, but rather is treated as if it were a single power-consuming device connected to a bus. This indicates that an aggregate load model may be used, though the precise load model is usually not available. Use of such aggregate load models has been frequently recommended in the literature since voltage stability analysis would otherwise be made impossible. Therefore, load modeling needs to be addressed for voltage stability analysis.

4.3.1. Static load statistics

The neural network has been widely applied as a computing technique. As has been mentioned before, one standard application of neural networks is to be used as a function approximator. Therefore, they can be used to model the static load characteristics which is usually represented by polynomial models. Generally, a neural network can give a better approximation than mere polynomials.

As is well known, a static load model is usually expressed as a function of the voltage magnitude and the frequency of the voltage at a specific bus to which the load is connected. That is, static load characteristic may be given by

$$P = F(v, f) \tag{4.13}$$

and

$$Q = G(v, f) \tag{4.14}$$

where P and Q are real power and reactive power at some bus, respectively; v and f are associated voltage and frequency, respectively; $F(.)$ and $G(.)$ are generally nonlinear functions.

We are more concerned about voltage stability than frequency stability since usually voltage decays much faster than frequency does during power system instability. Thus, the above equations may be approximated for short duration by

$$P = F(v) \tag{4.15}$$

and

$$Q = G(v) \tag{4.16}$$

A feedforward neural network can be applied to model these two nonlinear functions. Since sensitivities are useful in power flow study, the calculation of the sensitivity of the output with respect to the input of the neural network should be performed, and is addressed in the following.

Let the input be $x = [x_1\ x_2\ \cdots\ x_M]^\tau \in R^M$ to a neural network $N_{\sigma_1,\sigma_2,\cdots,\sigma_L}^{d_0,d_1,\cdots,d_L}$ (refer to section 2.2.2 of chapter 2 for notations used here) with weights $w_{i,j}^l$'s for $i = 1, 2, \cdots, d_l$, $j = 1, 2, \cdots, d_{l-1}$, and $l = 1, 2, \cdots, L$, its output $y = [y_1\ y_2\ \cdots\ y_N]^\tau \in R^N$. Note that for the representation of the neural network $d_0 = M$ and $d_L = N$. Then the sensitivity $\frac{\partial y_i}{\partial x_k}$ for $1 \le i \le N$ and $1 \le k \le M$ can be computed by applying the chain-rule.

Let s^l designate the output vector of layer l ($l = 1, 2, \cdots, L-1$). Then

$$\begin{aligned}\frac{\partial y_i}{\partial x_k} &= \sum_{m_{L-1}=1}^{d_{L-1}} \frac{\partial y_i}{\partial s_{m_{L-1}}^{L-1}} \frac{\partial s_{m_{L-1}}^{L-1}}{\partial x_k} \\ &= \sum_{m_{L-1}=1}^{d_{L-1}} \sum_{m_{L-2}=1}^{d_{L-2}} \cdots \sum_{m_1=1}^{d_1} \frac{\partial y_i}{\partial s_{m_{L-1}}^{L-1}} \frac{\partial s_{m_{L-1}}^{L-1}}{\partial s_{m_{L-2}}^{L-2}} \cdots \frac{\partial s_{m_2}^2}{\partial s_{m_1}^1} \frac{\partial s_{m_1}^1}{\partial x_k}\end{aligned} \tag{4.17}$$

Let the derivative of the activation function $\sigma_l(.)$ for layer $l = 1, 2, \cdots, L$ be denoted by $g^l(\sigma_l(.))$. Note that

$$\frac{\partial y_i}{\partial s_{m_{L-1}}^{L-1}} = w_{i,m_{L-1}} g^L(y_i)$$

$$\frac{\partial s_{m_l}^l}{\partial s_{m_{l-1}}^{l-1}} = g^l(s_{m_l}^l) w_{m_l,m_{l-1}} \quad \text{for } l = L, L-1, \cdots, 2$$

and

$$\frac{\partial s_{m_1}^1}{\partial x_k} = g^1(s_{m_1}^1) w_{m_1,k}$$

Therefore, the following results

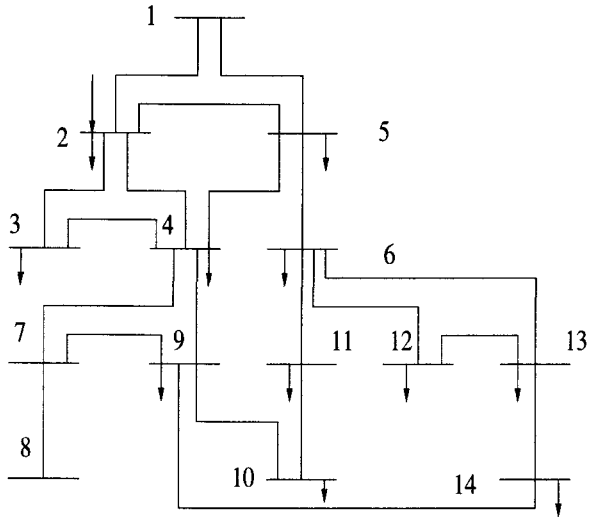
$$\frac{\partial y_i}{\partial x_k} = \sum_{m_{L-1}=1}^{d_{L-1}} \sum_{m_{L-2}=1}^{d_{L-2}} \cdots \sum_{m_1=1}^{d_1} w_{m_i,m_{L-1}} w_{m_{L-1},m_{L-2}} \cdots w_{m_2,m_1} w_{m_1,k} \\ g^L(y_i) g^{L-1}(s_{m_{L-1}}^{L-1}) \cdots g^2(s_{m_2}^2) g^1(s_{m_1}^1) \tag{4.18}$$

Note that $s_{m_l}^l = \sigma_l(\sum_{j=1}^{d_{l-1}} w_{m_l,j} s_j^{l-1} + b_{m_l}^l)$ for $l = 1, 2, \cdots, L-1$ with $s_j^0 = x_j$ for $1 \leq j \leq M$, and $b_{m_l}^l$ designating the bias for the m_lth neuron in layer $l = 1, 2, \cdots, L$.

Thus, all $s_{m_l}^l$'s for $l = 1, 2, \cdots, L-1$ are continuous functions of x, and their values for a specific x can be computed in a forward manner, as what is done for the feedforward phase in the backpropagation algorithm.

Hence with the sensitivities available for real power and reactive power of a load with respect to the voltage magnitude at the bus where the load is connected, modal analysis can be performed, and information about the relative robustness of load buses can be obtained. This will be detailed in next section, where the static and dynamic voltage stability analyses will be made.

The standard IEEE 14-bus system, shown in Figure 4.1 is used for simulation. Uniformly distributed loads are added to the original load at bus 14. The Newton-Raphson method is adopted for power flow study. For random loads, the voltage magnitudes at bus 14 can then be obtained as shown in Figure 4.2. The use of a one hidden layer feedforward neural network gives the approximation results in Figure 4.3.



1 --- Slack Bus; 2,3,6,8 --- PV Bus; all others PQ Bus

FIGURE 4.1: IEEE 14-bus system

4.3.2. Load dynamics modeling

Most physical system dynamic behaviors can not be described in terms of a static mapping from the input space to the output space. One way out may be the use of recurrent neural networks [32, 31] which can be state-feedback based, shown in Figure 4.4, or output-feedback based, shown in Figure 4.5. The complex input-output dynamics can be estimated and approximated by thses two kinds of neural networks. A natural choice of the performance criterion for such neural networks would be the weighted summation of the square of the error between the target sequence and the output sequence of the neural networks. The dynamic back-propagation algorithm [32] is very useful for training a recurrent neural network to follow a pre-specified temporal output sequence if the network is fed the pre-specified input sequence. Again, power systems and load flows are essentially

dynamic and too complex to be expressed in a simple form, but recurrent neural networks may be an appropriate option.

Power system loads or demands are dynamic in nature. Their dynamic characteristics are critically important for making predictions about the operating point of the power system and assessing the voltage stability limits. The load composition, however, is so complex and time-variant that a simple analytic form for the aggregated load dynamics is not likely by traditional methods. Due to the diversity of the dynamic characteristics of all possible power-consuming devices connected to a voltage bus, it might not be possible to obtain satisfactory results with simple linear models (either time-invariant or time varying). In [64], simplified nonlinear dynamic loads in power system are modeled by using a first-order differential equation. The model in [64] (similar model in [75]) is such that

$$T_p\dot{P}_d + P_d = P_s(V) + k_p(V)\dot{(V)} \tag{4.19}$$

where P_d and V are for power demand and bus voltage, respectively. P_s is denoted steady-state power. Such a model is proposed to give some insight into the dynamic response of the power when the voltage magnitude is suddenly reduced to a lower value. This model might not be useful for unknown load dynamics. Based on the robustness and fault-tolerance capability, and good approximation capability of neural networks, they have been applied successfully to power forecasting. It also was used to model the complex dynamics of overall load connected to a voltage bus [21]. Such model is also based on the first-order differential equation

$$\dot{x} = f(x, u) \tag{4.20}$$

A discrete version of such a model is

$$x_k = f(x_{k-1}, u_k) \tag{4.21}$$

where x stands for active power or reactive power. u denotes the voltage.

It is clear that a mathematical representation of load dynamics is critical for voltage analysis. The above models are only of a first-order approximation.

In this section, we will discuss the following more general model.

$$f(x^{(N)}, \cdots, \dot{x}, x, u^{(M)}, \cdots, \dot{u}, u) = 0 \tag{4.22}$$

where x stands for active power or reactive power; u stands for bus voltage.

For the discrete case,

$$f(x_{n-N}, \cdots, x_n, u_{n-M}, \cdots, u_{n-1}, u_n) = 0 \tag{4.23}$$

Assume x_n can be expressed in terms of other arguments in the above equations. Then

$$x_n = g(x_{n-1}, \cdots, x_{n-N}, u_n, u_{n-1}, \cdots, u_{n-M}) \tag{4.24}$$

Note that the above model is a dynamic model for modeling purpose. If a prediction model is in need, then the following model may be used.

$$x_n = g(x_{n-1}, \cdots, x_{n-N}, u_{n-1}, \cdots, u_{n-M}) \tag{4.25}$$

This is a one-step ahead prediction model, which describes the structure of neural networks shown in Figure 4.5.

For a multi-step ahead prediction model,

$$x_{n+P} = g(x_n, \cdots, x_{n-N}, u_n, \cdots, u_{n-M}) \tag{4.26}$$

where P is the prediction step.

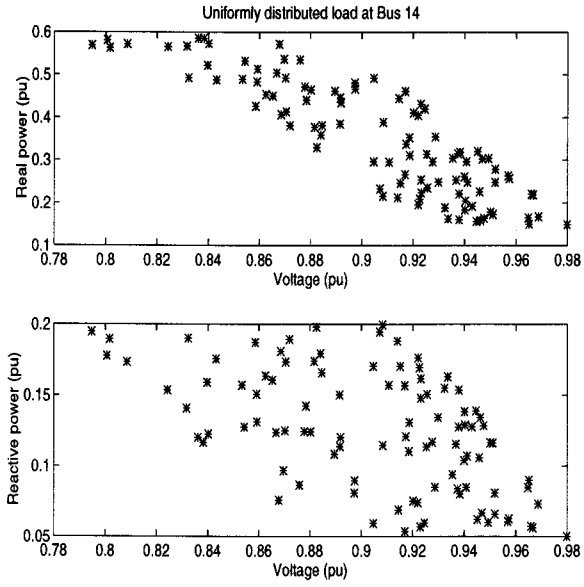


FIGURE 4.2: Real/reactive power vs voltage magnitude

Our neural network model is based on Equation (4.24). Since the output of the neural network is dependent not only on the previous voltage, but also on its own previous values, it has been called a recurrent neural network, and can be used to model complex system dynamics [80, 33].

Note that a neural network model for equation (4.24) can be obtained either by dynamic backpropagation or plain backpropagation. For the former case, the neural network is trained in parallel model; for the latter case, the neural network is trained in series-parallel model. Also both output-based recurrent neural network and locally recurrent neural network are trained over a given set of data.

Consider a locally recurrent neural network with one hidden layer. It can be viewed as a state-space model, which is illustrated in Figure 4.4.

$$x(k+1) = f(x(k), u(k)) \tag{4.27}$$

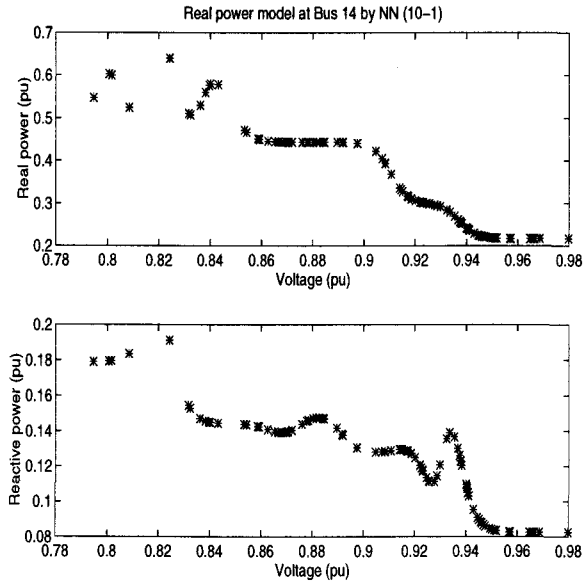


FIGURE 4.3: Neural network model for load at bus 14

$$y(k) = g(x(k)) \tag{4.28}$$

There are times at which it may be necessary to use a locally recurrent neural network with two hidden layers. Such a model is also a state-space model but with more complexity.

$$x(k+1) = f(x(k), u(k)) \tag{4.29}$$

$$z(k+1) = g(z(k), x(k)) \tag{4.30}$$

$$y(k) = h(z(k)) \tag{4.31}$$

Let $X(k) = [x(k)' \;\; z(k)']'$, then the first two equations can be combined

$$X(k+1) = F(X(k), u(k)) \tag{4.32}$$

The output of the recurrent neural network is

$$y(k) = h(X(k)) \tag{4.33}$$

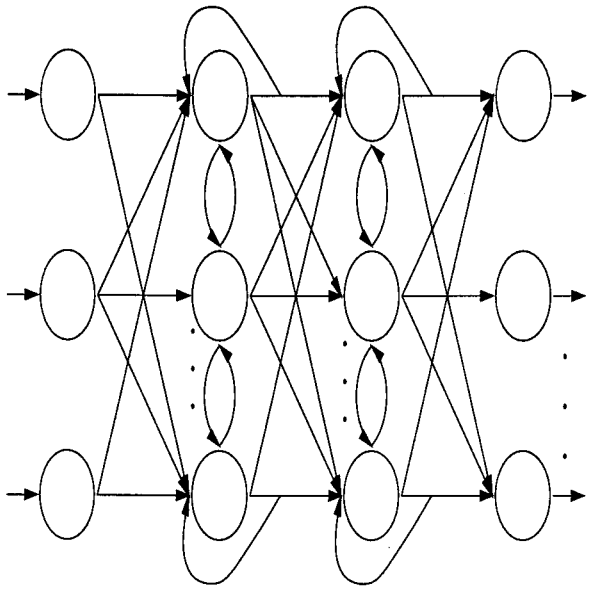


FIGURE 4.4: Recurrent neural network

A set of data [14], shown in Figure 4.6, which was recorded when voltage dip occurred first at a specific voltage bus, is used for training neural networks. Since normally bus voltages are within the pre-designated limits, say $\pm 5\%$, the training may not be so efficient if no pre-processing for the original data is performed. Under such a consideration, a standard normalization is applied befor neural network training. Let X_b be the original data. Then the post- processing data X_a is

$$X_a = (X_b - \overline{X_b})/var(X_b) \tag{4.34}$$

where $\overline{X_b}$ is the average of the given data, and $var(X_b)$ is the standard deviation. The normalized data to be used in training neural networks are shown in Figure 4.7.

First, model (4.24) is used for training a recurrent neural network through batch-mode training. Such a training involves the choice of number of layers of neural networks, and number of neurons in the hidden layers. Many different neural network configura-

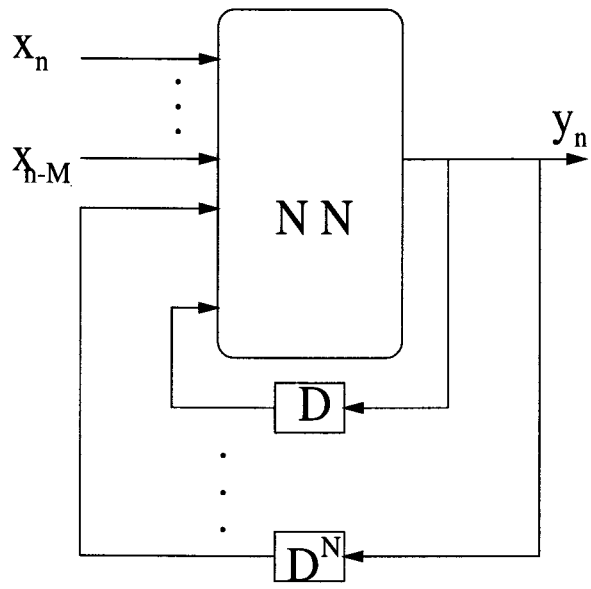


FIGURE 4.5: Output-feedback neural network

tions are tested. And the common logistic sigmoidal function and tan-sigmoidal function are used for simulations. From the experiments, it is seen that two hidden layer neural networks are more efficient that one hidden layer neural network if both kinds of neural network structures have approximately identical number of neurons (because there are more connections in general for the former, hence allowing more freedom for adjustment of parameters involved). The training error can be reduced by training more time for a fixed neural network structure. In fact, the training model is essentially input-output modeled by feeding the previous outputs into the neural network to model the dynamics of the concerned process. It should be noted that of course, the proposed latitudinal neural network structure in chapter 3 can be used for reduction of the training error.

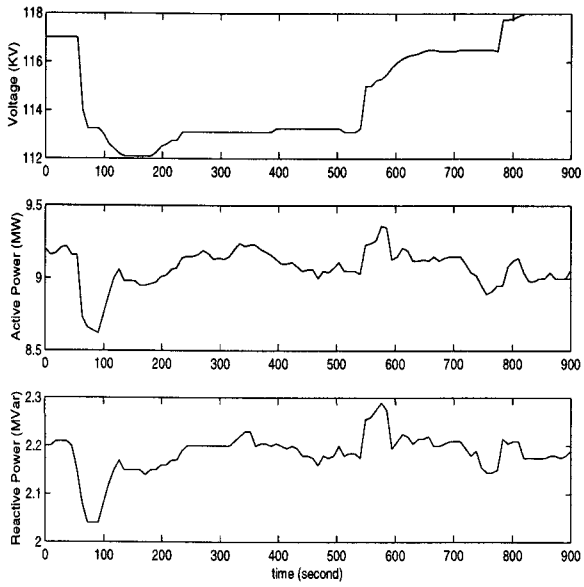


FIGURE 4.6: Original data for voltage, active/reactive power (sampling interval: 9 seconds)

Secondly, a locally recurrent neural network with one hidden layer neural is trained, in which 20 hidden neurons are used. The experimental results are shown in Figure 4.8 and Figure 4.9 after de-normalization.

It is also possible to train a locally recurrent neural network with two or more hidden layers and even with cross-talk links for modeling more complex dynamics.

The precise gradient information involved in training a recurrent neural network can be obtained only by using the dynamic backpropagation. The use of the dynamic backpropagation algorithm, however, involves intensive computational effort even for simple low-order dynamical systems. This has led to use of the plain backpropagation algorithm keeping it in mind that the step size for weights updating has to be kept small to ensure the stability of the closed-loop system.

4.4. Voltage stability analysis

Incorporation of load representations into voltage stability analysis is discussed in [75, 70]. A simple form for load representation therein is given by

$$T_p\dot{x} = P_s(v) - xP_t(v) \tag{4.35}$$

where $P_s(v)$ stands for static load model, which is usually represented by an exponential model; $P_t(v)$ stands for transient load; and T_p is load recovery time. Notice that mathematically when $t \to \infty$, x tends to a constant x_s, then $P_t(v) = P_s(v)/x_s$. In physics, when there is voltage drop, the real power that the load can draw also decreases. And within a certain amount of time, the real power recovers up to a certain amount.

Another similar load model is given in [70] by

$$T_p\dot{x} = P_s(v) - (x + P_t(v)) \tag{4.36}$$

It should be noted that these two load models are consistent with the models derived in [63, 64] and that all those models agree with the general model governed by first order differential equation (first order models for induction motors, thermostatic heating load, and tap changing transformers are typical examples).

In what follows, the loads modeled throught neural networks, addressed in the last section, are included in voltage stability analysis.

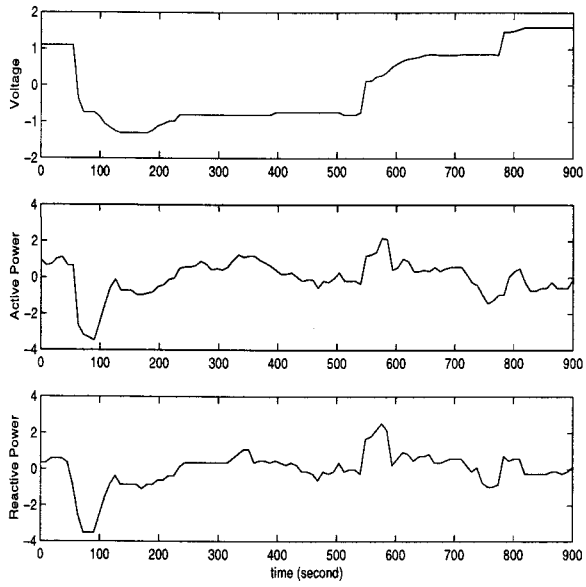


FIGURE 4.7: Normalized data for voltage, active/reactive power (sampling interval: 9 seconds)

4.4.1. Static voltage stability analysis

Consider a simple static case — a two bus system consisting of a generator and an aggregated load which is modeled by two neural networks. Let v_g, δ_g, v_d, δ_d, z and θ_z specify the generator bus voltage magnitude, the generator bus voltage pahse angle, the load bus voltage magnitude, the load bus voltage phase angle, and the impedance magnitude and phase angle of the transmission line (including the impedance on the generator side if any), respectively. And P_d and Q_d are the real and reactive power demand, whose neural network models are assumed to be $f(v_d)$ and $g(v_d)$, respectively.

It can be shown that

$$P_d = \frac{v_g v_d}{z}\cos(\delta_d - \delta_g + \theta_z) - \frac{v_d^2}{z}\cos(\theta_z) \tag{4.37}$$

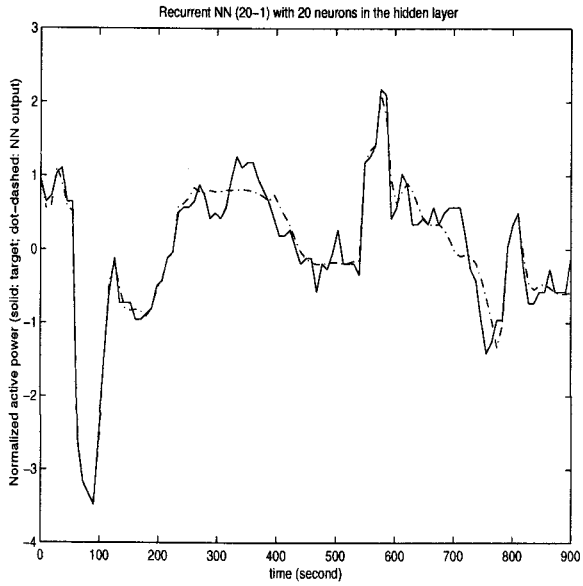


FIGURE 4.8: Target and output of the recurrent NN with 1 hidden layer (sampling interval: 9 seconds)

and

$$Q_d = \frac{v_g v_d}{z} \sin(\delta_d - \delta_g + \theta_z) - \frac{v_d^2}{z} \sin(\theta_z) \tag{4.38}$$

where δ_g may be chosen to be zero; v_g is pre-specified.

That is,

$$\frac{v_g v_d}{z} \cos(\delta_d - \delta_g + \theta_z) - \frac{v_d^2}{z} \cos(\theta_z) - f(v_d) = 0 \tag{4.39}$$

and

$$\frac{v_g v_d}{z} \sin(\delta_d - \delta_g + \theta_z) - \frac{v_d^2}{z} \sin(\theta_z) - g(v_d) = 0 \tag{4.40}$$

We must now distinguish between specified or scheduled powers and powers calculated using the above two equations. The difference is the so-called mismatch which becomes small as convergence of the iterative process for a solution is reached.

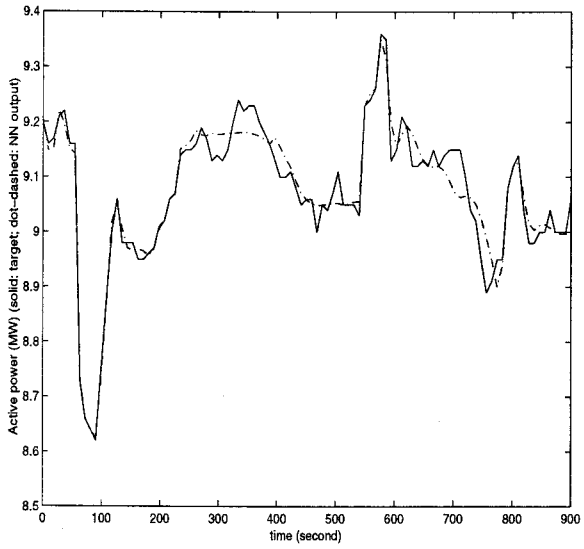


FIGURE 4.9: Target and output of the recurrent NN (sampling interval: 9 seconds)

The Newton-Raphson method may be employed to solve these two nonlinear equations as

$$J \begin{bmatrix} \Delta\delta_d \\ \Delta v_d \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \tag{4.41}$$

where ΔP and ΔQ are mismatch powers, Δv_d is the unknown load bus voltage magnitude correction, and $\Delta\delta_d$ is the unknown load bus voltage angle correction; and J is the Jacobian matrix given in the following:

$$J = \begin{bmatrix} J_{11} & J_{12} \\ J_{13} & J_{14} \end{bmatrix} \text{ with}$$

$$J_{11} = \frac{v_g}{z}\cos(\delta_d - \delta_g + \theta_z) - 2\frac{v_d}{z}\cos(\theta_z) - f'(v_d);$$

$$J_{12} = -\frac{v_g v_d}{z}\sin(\delta_d - \delta_g + \theta_z);$$

$$J_{13} = \frac{v_g}{z}\sin(\delta_d - \delta_g + \theta_z) - 2\frac{v_d}{z}\sin(\theta_z) - g'(v_d);$$

$$J_{14} = -\frac{v_g v_d}{z}\cos(\delta_d - \delta_g + \theta_z).$$

It should be noted that $f'(v_d)$ and $g'(v_d)$ can be computed by using the results derived in the last section. The system will reach its maximum loadability (sometimes called voltage stability limit though there may be the case when power systems are still stable even though the equilibrium point is located on the lower part of P-V curve) when the Jacobian tends to zero. Also, it should be noted that transient instability may occur without this condition.

The above results can be easily generalized to large interconnected power systems. For a total of N buses the voltage V_k at any bus k, where net real power P_k and reactive power Q_k are given, can be expressed as

$$V_k = \frac{1}{Y_{kk}}\left(\frac{P_k - jQ_k}{V_k^*} - \sum_{n=1,\ n\neq k}^{N} Y_{kn}V_n\right) \tag{4.42}$$

where Y_{kk}'s and Y_{kn}'s are just mutual-admittances of nodes k and n, and self-admittances of nodes k's.

With $V_k = v_k \exp(-j\delta_k)$, $V_n = v_n \exp(-j\delta_n)$, $Y_{kn} = y_{kn}\exp(-j\theta_{kn})$, the above equation can be rewritten in polar form.

$$P_k = \sum_{n=1}^{n=N} v_k v_n y_{kn} \cos(\theta_{kn} + \delta_n - \delta_k) \tag{4.43}$$

and

$$Q_k = -\sum_{n=1}^{n=N} v_k v_n y_{kn} \sin(\theta_{kn} + \delta_n - \delta_k) \tag{4.44}$$

The sensitivity $\frac{\partial P_k}{\partial V_n}$ can be easily obtained.

$$\begin{aligned}
\frac{\partial P_k}{\partial v_n} &= v_k y_{kn} \cos(\theta_{kn} + \delta_n - \delta_k) \ \text{ if } n \neq k \\
\frac{\partial P_k}{\partial v_n} &= 2v_k y_{kk} \cos(\theta_{kk}) \\
&\quad + \sum_{i=1,\ i\neq k}^{N} v_i y_{ki} \cos(\theta_{ki} + \delta_i - \delta_k) \ \text{ if } n = k
\end{aligned}$$

Similarly, other sensitivities $\frac{\partial P_k}{\partial \delta_n}$, $\frac{\partial Q_k}{\partial V_n}$ and $\frac{\partial Q_k}{\partial \delta_n}$ can also be obtained and given in the following:

$$\begin{aligned}
\frac{\partial P_k}{\partial \delta_n} &= -v_k v_n y_{kn} \sin(\theta_{kn} + \delta_n - \delta_k) \text{ if } n \neq k \\
\frac{\partial P_k}{\partial v_n} &= + \sum_{i=1,\ i\neq k}^{N} v_k v_i y_{ki} \sin(\theta_{ki} + \delta_i - \delta_k) \text{ if } n = k
\end{aligned}$$

$$\begin{aligned}
\frac{\partial Q_k}{\partial v_n} &= -v_k y_{kn} \sin(\theta_{kn} + \delta_n - \delta_k) \text{ if } n \neq k \\
\frac{\partial Q_k}{\partial v_n} &= -2 v_k y_{kk} \sin(\theta_{kk}) \\
&\quad - \sum_{i=1,\ i\neq k}^{N} v_i y_{ki} \sin(\theta_{ki} + \delta_i - \delta_k) \text{ if } n = k
\end{aligned}$$

$$\begin{aligned}
\frac{\partial Q_k}{\partial \delta_n} &= -v_k v_n y_{kn} \cos(\theta_{kn} + \delta_n - \delta_k) \text{ if } n \neq k \\
\frac{\partial Q_k}{\partial v_n} &= + \sum_{i=1,\ i\neq k}^{N} v_k v_i y_{ki} \cos(\theta_{ki} + \delta_i - \delta_k) \text{ if } n = k
\end{aligned}$$

The involved sensitivities at some specific buses need to be modified if the loads at these buses are modeled by neural networks. For instance, assume bus k is one of such buses. Then from equations (4.43) and (4.44), we have

$$Pg_k - Pd_k = \sum_{n=1}^{n=N} v_k v_n y_{kn} \cos(\theta_{kn} + \delta_n - \delta_k) \tag{4.45}$$

and

$$Qg_k - Qd_k = - \sum_{n=1}^{n=N} v_k v_n y_{kn} \sin(\theta_{kn} + \delta_n - \delta_k) \tag{4.46}$$

where Pd_k and Qd_k are real and reactive power demands at bus k modeled by neural networks, and Pg_k and Qg_k are generated real and reactive powers at bus k.

Thus, the sensitivities neeed to be modified at bus k should be $\frac{\partial P_k}{\partial v_k} + Pd_k'(v_k)$ and $\frac{\partial Q_k}{\partial v_k} + Qd_k'(v_k)$ if they are $\frac{\partial P_k}{\partial v_k}$ and $\frac{\partial Q_k}{\partial v_k}$ originally . However, $Pd_k'(v_k)$ and $Qd_k'(v_k)$ can

be obtained through the technique developed in last section. Therefore, the modified Jacobian matrix, with some related elements modified, still takes the general form

$$J = \begin{bmatrix} P_v & P_\delta \\ Q_v & Q_\delta \end{bmatrix} \tag{4.47}$$

where P_v is the partial derivative matrix resulting from differentiating the real power vector with respect to voltage magnitude vector; P_δ, Q_v and Q_δ can be explained similarly.

This matrix can be applied either for power flow study by the Newton-Raphason method, or for eigenvalue analysis. The eigenvalues of the reduced Jacobian matrix display the possible modes of voltage stability [56].

The resulting neural network load model for bus 14 in the IEEE 14-bus system is combined with the Newton-Raphson method for use in the load flow study. The results are shown in Table 4.1.

It can be observed from the results that if the randomly added loads are ignored, the results may be optimistic; if the random loads are considered to be their average values, the resulted power flow study may be over-simplified, and that the neural networks used present an approximately accurate representation of the nonlinear relation between the random loads and the 14th bus voltage magnitude, and thus give reasonable results. The eigenvalues in Table 4.1 are all positive and, through modal analysis, suggest that the power system is still stable.

4.4.2. Dynamic voltage stability analysis

Dynamic load models can be used for dynamic voltage stability analysis. Such a model, detailed in the last section can be trained with available data recorded in credible contingencies, large load buildup, and unfavorable load dynamics, etc. Also those data can be used to train neural networks. Through the trained neural network, the loading patterns can be recogized. Dynamic voltage stability analysis should be employed whenever there

is such a need indicated by neural networks. For application of the Newton-Raphson method, equations (4.24) and (4.42) should be iteratively used. It should be noted that equation (4.24) describes the dynamic relation of real load power and reactive load power on voltage magnitude. It should also be noticed that equation (4.42) characterizes the whole power system. With use of previous real/reactive load power and previous voltage magnitude, current real/reactive load power at a specific bus can be computed through equation (4.24) (Keep it in mind that there should be each dynamic model for either dynamic real load or reactive load at a specific bus). With the resulting values and the given conditions, equation (4.42) is applied to give the possible solution. These steps can be repeated. Of course, the state-space model (4.27) and (4.28), combined with equation (4.42), can also be used for dynamic voltage stability analysis. The modal analysis can also be performed through time. The involved sensitivities for modal analysis may be obtained in the way the static voltage stability analysis is made. Without question, dynamic voltage stability analysis is very time-consuming.

4.5. Conclusions and outlooks

This chapter presents a neural network methodology for dealing with static and dynamic load modeling. The loading patterns are classified by feedforward neural networks. Based on the static load model and dynamic load model, either static voltage stability analysis or dynamic voltage stability analysis can be made. The sensitivities involved in neural network models for loads are derived, and are then used in the Jacobian matrix, and further for the modal analysis. The neural network methodology is tested either on the IEEE-14 bus system or real field data. Since static load model may not be suitable for loading dynamics while dynamic voltage stability analysis is too time-consuming, which will affect its effective on-line use, voltage stability indices (e.g. margin for operating point

to reach the saddle point) may be very suitable for on-line use. Static voltage stability indices may not be sufficient. Dynamic voltage stability indices are needed.

	NN load model	Original load	Expected load
Real load	0.2268	0.1490	0.3725
Reactive load	0.1053	0.0500	0.1250
Voltage magnitude	0.9458	0.9798	0.9076
Eigenvalues	63.8474	64.2885	63.2086
	38.5836	38.6527	38.3830
	30.6897	31.2209	30.0822
	27.2015	27.6361	26.6971
	17.2232	17.4309	16.9932
	0.5126	0.5412	0.4750
	15.2683	15.4484	15.0381
	3.8089	3.8762	3.7308
	5.4869	5.5675	5.9188
	11.2959	11.5020	11.0658
	6.0411	6.1986	5.3601

TABLE 4.1: Comparison of different load models

5. SYNTHESIS OF ADAPTIVE HIERARCHICAL CONTROLLERS APPLIED TO DYNAMIC POWER SYSTEMS

5.1. Introduction

The concern for maintaining bus voltages stability has been growing. Many voltage instability incidents have occurred around the world (e.g. Japan, France, Belgium, and USA). Some of these incidents even caused partial or complete blackouts (voltage collapses). The detailed descriptions of these events can be obtained in [14, 81]. There are various causes which might lead to these severe system failures. According to the available data [14], the initial causes may be AC line trip caused by a ground fault, generator loss, immediate heavy load buildup, special load with unfavorable load dynamic characteristics, etc. The real problem, however, is one of interaction of very complex nonlinear dynamics and lack of control.

Transient voltage stability is usually closely associated with regular generator angle stability ([61] and therein). Longer-term voltage stability, however, is more closely associated with loading dynamics. This kind of voltage instability has stimulated extensive research on voltage stability analysis methods involving quasi-static power flow or continuation power flow [55], snapshot method [62], modal analysis [56],load dynamics [63, 64], energy method [54, 65], static bifurcation application to power systems [66, 67] etc. However, voltage stability and rotor angle stability are more or less interlinked, and their mechanisms can be difficult to separate. It is usually assumed that if voltage collapse occurs in a transmission system far away from load centers, it is mainly a rotor angle instability problem; if voltage collapse occurs in load areas, it is primarily a voltage instability problem—but hardly proven.

The Flexible AC Transmission Systems (FACTS) are utilized to enhance power transfer capability over existing transmission lines and greatly improve stability characteristics of power systems. Some FACTS devices are already in wide use (for instance, Thyristor Controlled Series Capacitors (TCSC) and Static VAR Compensators (SVC)), while some others are still under development (for instance, Unified Power Controllers (UPC)). One of the main roles that FACTS devices play is to adjust the reactive power flow, correct the massive power imbalance, and re-establish the equilibrium in case of occurance of large faults. Therefore, the proper manipulation of the FACTS devices in place by means of some properly designed control mechanism is then crucial in maintaining power system stability.

As is known, the mechanism of the transient stability is well understood while the voltage collapse mechanism is much more complicated that might be associated with either the rotor angle stability or load-driven stability or both, and needs many in-depth investigations. Since voltage instability is closely associated with the loading patterns, as a first attemp to exploration of the voltage collapse mechanism, the transient stability issue is investigated while considering the effect of the load at the same time. Such an effort is significant—in that: (1) since the TCSC installed on the tie line is intended to help dampen the inter-area mode oscillations between the two subsystems connected through a tie line, it is shown in [82] that such a system can be simplified as a SMIB system with a TCSC and time-varying parameters under some assumptions; and (2) the insights gained and the techniques developed from the study of a SMIB with a load may help develop techniques which are useful for preventing voltage instability problems in multi-machine systems with various kinds of loading characteristics.

Note that the random changes in operation conditions and possible faults in power systems result in uncertain dynamic systems, which call for high-performance robust nonlinear controllers to enhance and ensure the system transient stability.

A number of studies, involving bilinear adaptive control scheme [10, 11], variable-structure control [83], robust control [84], and neural network control [12], have been done on the controller design which may stabilize the postfault power systems. Nonlinear control strategies have been effective on a case to case basis. Note that adaptive control, robust control and variable-structure control are typically considered model-based schemes, and that neural control is considered to be data based. This difference makes neural control perhaps superior to the others in the case of unmodeled plants since the off-line generated optimal trajectories or the desired trajectories are available and may help train a proper neural controller. However, it should be noted that the neural network structure, of nonlinearly coupled bilinear systems, is similar to the basic nonlinear structure of FACTS where the parametric control allows improved controllability and transient stabilization. In order to handle the uncertainty that exists in practical systems, a control-switching scheme is introduced in [85] in order to generate the intelligent control. The idea thereof was further developed in [12], resulting in a multiplicative control scheme, which is essentially a convex interpolation of the nominal controllers, designed for specific cases, instead of control switching. A similar hierarchical control structure [86] is approximately the same as that for a fuzzy control ([87] and therein) except that the control weights, i.e., the membership values in the context of fuzzy logic, are determined by a set of fuzzy rules. The application to jet aircraft engines of this kind of hierarchical structure was also investigated in [88].

As is known, it is desired to stabilize the postfault power systems as quickly as possible by means of the constrained control. The time-optimal control policy, or near-time-optimal control policy in a more practical sense, is studied in the context of control design. Note that for nonlinear power systems, generally it is still very difficult to solve for optimal feedback controls. However, the off-line generated optimal trajectories may help train a neural network control which sufficiently approximates the optimal control which exists and whose analytic form is often quite difficult to obtain. Regarding the SMIB

system, yet with the consideration of the effect of the load, a number of novel techniques are developed to stabilize the transients incurred by occurance of large faults to the single-machine infinite-bus system with an uncertain load. Note that the uncertainty of the load makes the whole power system uncertain, and thus a somewhat "intelligent" controller is then necessary. The techniques proposed mainly include: (a) tessellation schemes which help synthesize reliable controllers with respect to an uncertain load, which can be modeled by a feedforward/dynamic neural network as discussed in chapter 4, and large faults; (b) a couple of pattern recognition schemes which are intended to well approximate the switching curve in the context of time-optimal control; (c) a hierarchical control structure which consists of two levels of neural networks, with the lower level neural networks trained for specific cases, and the upper level neural networks associated with some sort of comparator properly assigning the multipliers so that the resulting multiplicative control is still a bang-bang type. (d) A combination of a hierarchical neural control and a linear control by the latter of which the system can be driven from some neighborhood of the equilibrium to the exact equilibrium more effectively and thus it is required that the system be driven by means of the former only until it comes to some neighborhood of the equilibrium and then the latter takes over, thereby accomodating the possible errors in the available calculated optimal trajectories and also avoiding the so-called "chattering" phenomenon [83, 8, 9]. Further, some theoretical justifications of the proposed techniques are presented. The techniques developed, however, can be generally applied to more complex nonlinear systems.

This chapter is organized as follows: Section 5.2. formulates the problems that will be studied in detail later on, and shows that the explicit analytic solution to the time-optimal control problems are not available. In section 5.3., the formulated problems are transformed and nonlinear systems linear in control result. Numerical solutions can then be obtained by means of the switching-time-variation method (STVM) [7, 13] which makes effective use of the linearity in control. Based on the available optimal trajectories, the

pattern recoginition schemes are developed and used to stabilize the postfault power system. The hierarchical neural control is discussed in section 5.4.. Then section 5.5. presents some theoretical justification for the proposed methods. Some illustrative simulations are shown in section 5.6.. The proposed methods are generalized to more general systems in section 5.7.. Finally, some comments and conclusions are presented.

5.2. Time-optimal control for SMIB with a load

As is well known, a SMIB system with a FACTS device TCSC installed can be described by

$$\begin{cases} \dot{\delta} = \omega_b(\omega - 1) \\ \dot{\omega} = \frac{1}{M}(P_m - D(\omega - 1) - \frac{V_t V_\infty}{X_d + (1-s)Xe} \sin\delta) \end{cases} \tag{5.1}$$

where

δ - rotor angle (rad);

ω - rotor speed (p.u.);

ω_b - synchronous speed as base (rad/sec);

P_m - mechanical power input assumed to be constant (p.u.);

D - damping factor;

M - system inertia referenced to the base power;

V_t - terminal bus voltage (p.u.);

V_∞ - infinity bus voltage (p.u.);

X_d - transient reactance of the generator (p.u.);

X_e - transmission reactance (p.u.);

s - series compensation degree ($-sXe$ is the reactance of the TCSC, and often $0 < s < 1$);
The system is desired to be driven, after a transient period, to its equilibrium (δ_e, ω_e) by the admissible control $s \in [s_{min}, s_{max}]$ and stay in the equilibrium thereafter by the fixed compensation $s_e \in [s_{min}, s_{max}]$.

With the translation transform $\omega = \omega + 1$, it follows that

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = \frac{1}{M}(P_m - D\omega - \frac{V_t V_\infty}{X_d + (1-s)Xe} \sin\delta) \end{cases}$$

Note that the equilibrium for rotor speed is translated from 1 to 0.

To make the later derivations convenient, a nonlinear transformation is introduced as follows:

$$Y(u) = Y_0 + Y_a u$$

where Y_0 corresponds to the total admittance (i.e., $\frac{1}{X_d + X_e}$) under no compensation ($s = 0$ and also $u = 0$); Y_a is the resulting additional admittance for $u = 1$ (and also $s = 1$) due to TCSC; $Y(u)$ is the resulting total admittance, $Y(u) = \frac{1}{X_d + (1-s)Xe}$.

It then follows that

$$\begin{aligned} Y_0 &= \frac{1}{X_d + X_e} \\ Y_a &= \frac{X_e}{X_d + X_e} \\ s &= \frac{u(X_d + X_e)}{X_d + X_e u} \\ u &= \frac{X_d s}{X_d + (1-s)X_e} \end{aligned} \tag{5.2}$$

Note that the mapping from s ($s \in [0, 1]$) to u ($u \in [0, 1]$) is one-to-one correspondence and onto, and monotonical.

By the above transformation, the swing equation can then be rewritten as follows:

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = \frac{1}{M}(P_m - D\omega - (V_t V_\infty)(Y_0 + Y_a u) \sin\delta) \end{cases} \tag{5.3}$$

Further, it may be convenient sometimes to transform the above system in such a manner that its equilibrium is translated into the origin and that the control range is converted to [-1,1].

Note that $0 < s_{min} < s_e < s_{max} < 1$. Hence $0 < u_{min} < u_e < u_{max} < 1$, where u_{min}, u_e, and u_{max} are associated with s_{min}, s_e, and s_{max}, respectively, by Equation (5.2). Let $u = u' + u_e$ and $\delta = \delta' + \delta_e$. Then Equation (5.3) can be rewritten as

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = \frac{1}{M}(P_m - D\omega - (V_t V_\infty)(Y_0 + Y_a(u + u_e)) \sin(\delta + \delta_e)) \end{cases} \tag{5.4}$$

Note that in the above equation, the admissible control $u \in [u_{min} - u_e,\ u_{max} - u_e]$ and that the equilibrium is now at the origin.

Let $u = \frac{u_{max} - u_{min}}{2} v + \frac{u_{max} + u_{min}}{2} - u_e$. Then $v \in [-1,\ 1]$. Substitution of u in terms of v into Equation (5.4) and some algebraic manipulations yield

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = \frac{1}{M}(P_m - D\omega - (V_t V_\infty)(Y_0 + Y_a(\frac{u_{max} - u_{min}}{2} v + \frac{u_{max} + u_{min}}{2})) \sin(\delta + \delta_e)) \end{cases}$$

That is,

$$\begin{bmatrix} \dot{\delta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \omega_b \omega \\ \frac{1}{M}(P_m - D\omega - V_t V_\infty (Y_0 + \frac{(u_{max} + u_{min}) Y_a}{2}) \sin(\delta + \delta_e)) \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{M} V_t V_\infty \frac{(u_{max} - u_{min}) Y_a}{2} \sin(\delta + \delta_e) \end{bmatrix} v \tag{5.5}$$

Define $c_1 = \frac{P_m}{M}$, $c_2 = \frac{D}{M}$, $c_3 = \frac{V_t V_\infty (Y_0 + \frac{(u_{max} + u_{min}) Y_a}{2})}{M}$, and $c_4 = \frac{V_t V_\infty \frac{(u_{max} - u_{min}) Y_a}{2}}{M}$. Then we have

$$\begin{bmatrix} \dot{\delta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \omega_b \omega \\ c_1 - c_2 \omega - c_3 \sin(\delta + \delta_e) \end{bmatrix} + \begin{bmatrix} 0 \\ -c_4 \sin(\delta + \delta_e) \end{bmatrix} v \tag{5.6}$$

From Equation (5.6) the role of parametric control (and bilinear control for small δ) is apparent—making FACTS so effective. Likewise, the general nonlinearly coupled bilinear system structure, and the assumed neural network structure of approximation may be recognized.

5.2.1. SMIB with a load

On the study of power system stability, the load is usually assumed to be a constant in the literature. In this section, the SMIB system with a load is considered for stability concern and control design while the load is assumed to have some properties but its parameters or itself needs to be identified. Several different system models are formulated and discussed.

Case I: The load is assumed to be fixed but is unknown.

The SMIB system with a constant load $P_l = P_0$ can be described as follows:

$$\begin{cases} \dot{\delta} = \omega_b(\omega - 1) \\ \dot{\omega} = \frac{1}{M}(P_m - P_0 - D(\omega - 1) - \frac{V_t V_\infty}{X_d + (1-s)Xe} \sin\delta) \end{cases} \tag{5.7}$$

By using the same transformations introduced before, the above equation can be transformed as follows:

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = c_1 - c_{10} - c_2\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta)v \end{cases} \tag{5.8}$$

where $c_{10} = \frac{P_0}{M}$; and all other coefficients are defined as before.

Case II: The load is assumed to be an affine function of the frequency.

The SMIB system with such a load $P_l = P_0 + C\omega$ can be described as follows:

$$\begin{cases} \dot{\delta} = \omega_b(\omega - 1) \\ \dot{\omega} = \frac{1}{M}(P_m - P_0 - C\omega - D(\omega - 1) - \frac{V_t V_\infty}{X_d + (1-s)Xe} \sin\delta) \end{cases} \tag{5.9}$$

The above equation can also be transformed as follows:

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = c_1 - c_{10} - (c_{20} + c_2)\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta)v \end{cases} \tag{5.10}$$

where $c_{10} = \frac{P_0 + C}{M}$; $c_{20} = \frac{C}{M}$; and all other coefficients are defined as before.

Note that for case I, the parameter c_{10} is an unknown constant and has to be identified in order to proceed with a proper control; and for case II, the parameters c_{10} and c_{20} are unknown constants and have to be identified for the same purpose.

Case III: the analytic model for the load P_l is not known, but may be identified with previous data.

The SMIB system with such a load P_l can be described as follows:

$$\begin{cases} \dot{\delta} = \omega_b(\omega - 1) \\ \dot{\omega} = \frac{1}{M}(P_m - P_l - D(\omega - 1) - \frac{V_t V_\infty}{X_d + (1-s)Xe} \sin\delta) \end{cases} \tag{5.11}$$

The above equation can also be transformed as follows:

$$\begin{cases} \dot{\delta} = \omega_b\omega \\ \dot{\omega} = c_1 - c_{10} - c_2\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta)v \end{cases} \tag{5.12}$$

where $c_{10} = \frac{P_l}{M}$.

Note that with the formulation of these problems, the goal is to design neural controllers which perform well for each case. Since for case I and II, once the parameters involved are determined, the models take the same form as Equation (5.6) except that the fixed parameters take different values, the techniques used for computing the optimal control and trajectories can be used for cases I and II. Calculation of optimal control for system Equation (5.6) is actually to locate the switching manifold. Unfortunately, the switching manifold can not be given explicitly. It may, however, be determined approximately by using some numerical methods, for example, the switching-times-variation method (STVM), detailed in section 5.3.. The parameter spaces for cases I and II are tessellated into many sub-regions. For those parameters which correspond to the vertices of the tessellated sub-regions, optimal trajectories and optimal control are computed in the region of stability interest. The computed optimal trajectories and optimal control are used for training a neural network which approximately characterizes the switching manifold in a way described in later sections 5.4. and 5.5.. The synthesis of neural controllers for cases I and II, which requires the identification of some parameters, is also described in section 5.4..

Once the neural controller for cases I and II are synthesized, the neural controller for case III may be synthesized in a way detailed also section 5.4.. The theoretical support for these techniques is provided in section 5.5..

5.2.2. Minimal time control

Consider Equation (5.3) for minimal time control.

The optimal time performance index can be expressed as

$$J = \int_{t_0}^{T} 1dt \tag{5.13}$$

Define the Hamiltonian function as

$$H(x,u,t) = 1 + \lambda^{\tau} f = 1 + \lambda_1(\omega_b\omega) + \lambda_2(\frac{1}{M}(P_m - D\omega - (V_t V_\infty)(Y_0 + Y_a u)\sin\delta)) \tag{5.14}$$

where $x^{\tau} = [\delta \quad \omega]$; $\lambda^{\tau} = [\lambda_1 \quad \lambda_2]$; and $f(x,u,t)^{\tau} = [\omega_b\omega \quad \frac{1}{M}(P_m - D\omega - (V_t V_\infty(Y_0 + Y_a u)\sin\delta))]$.

The final-state constraint is $\Psi(x(T),T) = x(T) - x_e = 0$, or

$$\begin{cases} \delta(T) = \delta_e \\ \omega(T) = \omega_e = 0 \end{cases} \tag{5.15}$$

where $x_e^{\tau} = [\delta_e \quad \omega_e]$ is the desired equilibrium point; $x_0^{\tau} = [\delta_0 \quad \omega_0]$ is the initial state.

The state equation can be expressed as

$$\dot{x} = \frac{\partial H}{\partial \lambda} = f(x,u), \quad t \geq t_0 \tag{5.16}$$

The costate equation can be written as

$$-\dot{\lambda} = \frac{\partial H}{\partial x} = \frac{\partial f}{\partial x}^{\tau} \lambda, \quad t \leq T \tag{5.17}$$

where τ designates transpose.

That is,

$$\begin{cases} \dot{\lambda_1} = \frac{1}{M} V_t V_\infty (Y_0 + Y_a u)\lambda_2 \cos\delta \\ \dot{\lambda_2} = -\omega_b \lambda_1 + \frac{D}{M}\lambda_2 \end{cases} \tag{5.18}$$

The Pontryagin minimum principle is applied in order to derive the optimal control [94]. That is,

$$H(x^*, u^*, \lambda^*, t) \leq H(x^*, u, \lambda^*, t) \text{ for all admissible } u \tag{5.19}$$

It turns out that

$$-\frac{1}{M}\lambda_2^* \sin\delta^* Y_a u^* \leq -\frac{1}{M}\lambda_2^* \sin\delta^* Y_a u \tag{5.20}$$

Since M,Y_a are assumed positive constants,

$$\lambda_2^* \sin\delta^* u^* \geq \lambda_2^* \sin\delta^* u \tag{5.21}$$

Thus, the time-optimal control satisfies the following condition:

$$u^* = \begin{cases} u_{max}, & \lambda_2 \sin\delta > 0 \\ u_{min}, & \lambda_2 \sin\delta < 0 \end{cases} \tag{5.22}$$

Note that the possibility of a singular solution, i.e., $\lambda_2(t)\sin\delta(t) \equiv 0$ for some finite time interval, can be excluded, which is shown in Appendix B.

The terminal boundary conditon can be given by

$$(\Psi_t^T \mu + H)|_T = 0 \tag{5.23}$$

or

$$H|_T = 1 + \lambda_2(T)\frac{1}{M}[P_m - V_t V_\infty (Y_0 + Y_a u(T))\sin\delta(T)] = 0 \tag{5.24}$$

That is,

$$\lambda_2(T) = -\frac{M}{P_m - V_t V_\infty (Y_0 + Y_a u(T))\sin\delta(T)} \tag{5.25}$$

It is observed that the resulting Hamiltonian system is a coupled twopoint boundary-value problem, and its analytic solution is not available, to our best knowledge.

5.3. Switching-time-variation method (STVM)

In section 5.2.2., though the necessary conditions for the time-optimal control are given, the analytic solution is not available. In order to get the time-optimal control, the switching-times-variation method (STVM) is used [7, 13].

It is observed that the system described by Equation (5.6) is a nonlinear system but linear in control. In the following, the STVM applies to the SMIB system for calculation of time-optimal control.

Define the performance index by

$$J = \int_{t_0}^{t_f} 1dt + \frac{\rho}{2}(\delta_f^2 + \omega_f^2) \tag{5.26}$$

Or

$$\begin{aligned} J = \quad & \int_{t_0}^{t_f} [1 + \rho(\delta\dot{\delta} + \omega\dot{\omega})]dt \\ & \int_{t_0}^{t_f} \{1 + \rho(\omega_b\omega\delta + \omega(c_1 - c_2\omega - c_3\sin(\delta + \delta_e) - c_4\sin(\delta + \delta_e)v))\}dt \end{aligned} \tag{5.27}$$

where ρ is a positive real number. Note that for equations (5.26) and (5.27) there is only a constant difference.

Define

$$\dot{x}_0 = 1 + \rho(\omega_b\omega\delta + \omega(c_1 - c_2\omega - c_3\sin(\delta + \delta_e) - c_4\sin(\delta + \delta_e)v)) \tag{5.28}$$

Then $J = x_0(t_f)$.

Define the augmented state vector as $\underline{x} = [x_0 \ \ \delta \ \ \omega]^\tau$. And the augmented system is then the following:

$$\dot{\underline{x}} = A(\underline{x}) + B(\underline{x})v \tag{5.29}$$

where $A(\underline{x}) = \begin{bmatrix} 1 + \rho\omega(\omega_b\delta + c_1 - c_2\omega - c_3\sin(\delta + \delta_e)) \\ \omega_b\omega \\ c_1 - c_2\omega - c_3\sin(\delta + \delta_e) \end{bmatrix}$,

$$B(\underline{x}) = \begin{bmatrix} -\rho c_4 \omega \sin(\delta + \delta_e) \\ 0 \\ -c_4 \sin(\delta + \delta_e) \end{bmatrix}, \text{ and } \underline{x}(t_0) = [0 \;\; \delta_0 \;\; \omega_0]^\tau.$$

Define the adjoint equation as follows

$$\dot{\underline{\lambda}} = -\frac{\partial}{\partial \underline{x}}[A(\underline{x}) + B(\underline{x})v]^\tau \underline{\lambda} \tag{5.30}$$

where $\underline{\lambda} = [\lambda_0 \; \lambda_1 \; \lambda_2]^\tau$ and $\underline{\lambda}(t_f) = \frac{\partial J}{\partial \underline{x}}|_{\underline{x}(t_f)} = [1 \; \rho\delta \; \rho\omega]^\tau|_{\underline{x}(t_f)}$.

That is,

$$\begin{bmatrix} \dot{\lambda}_0 \\ \dot{\lambda}_1 \\ \dot{\lambda}_2 \end{bmatrix} = - \begin{bmatrix} 0 & 0 & 0 \\ \rho\omega[\omega_b - (c_3 + c_4 v)\cos(\delta + \delta_e)] & 0 & -(c_3 + c_4 v)\cos(\delta + \delta_e) \\ \rho[\omega_b \delta + c_1 - 2c_2\omega - (c_3 + c_4 v)\sin(\delta + \delta_e)] & \omega_b & -c_2 \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{bmatrix}$$

Observation that $\lambda_0 = 1$ reduces the above equation to the following

$$\begin{aligned} \begin{bmatrix} \dot{\lambda}_1 \\ \dot{\lambda}_2 \end{bmatrix} = & - \begin{bmatrix} 0 & -(c_3 + c_4 v)\cos(\delta + \delta_e) \\ \omega_b & -c_2 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} \\ & - \begin{bmatrix} \rho\omega[\omega_b - (c_3 + c_4 v)\cos(\delta + \delta_e)] \\ \rho[\omega_b \delta + c_1 - 2c_2\omega - (c_3 + c_4 v)\sin(\delta + \delta_e)] \end{bmatrix} \end{aligned} \tag{5.31}$$

The switching function is given by

$$\begin{aligned} \phi(t) &= 2\underline{\lambda}^\tau B(\underline{x}) \\ &= -2c_4 \sin(\delta + \delta_e)(\rho\omega + \lambda_2) \end{aligned} \tag{5.32}$$

Suppose the number of the optimal switching times is N (including the variable terminal time). Let the switching vector be $\underline{\tau} = [\tau_1 \; \cdots \; \tau_{N-1} \; t_f]^\tau$. According to [7, 13], the gradient of the cost function with respect to the switching vector can be given by

$$\begin{aligned} \nabla_{\underline{\tau}}^J &= -\underline{\phi}^N \\ &= [\phi(\tau_1) \; \cdots \; (-1)^{N-2}\phi(\tau_{N-1}) \; \dot{J}(t_f)]^\tau \end{aligned} \tag{5.33}$$

where $\underline{\phi}^N = [-\phi(\tau_1) \; \cdots \; (-1)^{N-1}\phi(\tau_{N-1}) \; -\dot{J}(t_f)]^\tau$ and $\dot{J}(t_f) = 1 + \rho\{\omega_b\omega\delta + \omega[c_1 - c_2\omega - c_3\sin(\delta + \delta_e) - c_4\sin(\delta + \delta_e)v]\}|_{t_f}$.

The optimal switching vector can then be obtained by using a gradient-based method through iterations.

$$\underline{\tau}_{i+1} = \underline{\tau}_i + K_i \underline{\phi}_i^N \tag{5.34}$$

where $\underline{\tau}_i$ is the switching vector, , $\underline{\phi}_i^N$ is the negative gradient vector, and K_i is a properly-chosen $N \times N$-dimensional diagonal matrix with non-negative entries for the ith iteration.

Note that the time-optimal solution can also be obtained by considering the orginal variable terminal time problem as the limit of a sequence of fixed terminal time problems. Mathematically, let t_f be the solution to the variable terminal time problem, and t_f^i the solution to the ith fixed variable terminal problem, then

$$t_f = \inf_i \{t_f^i : \text{ solution exists}\} \tag{5.35}$$

Details are available in [7, 13], where the solution is shown to converge to the minimum-principle solution in general.

It is observed that the STVM is also applicable to Equation (5.8) and (5.10) for calculation of optimal trajectories for different initial conditions.

5.4. Synthesis of a neural controller as a power system stabilizer

Artifical neural networks have been widely applied in many diverse real-world applications, such as speech processing, image processing, computer vision, pattern classification and recognition, system control, and robotics [89]. The great function approximation capabilities of neural networks and the gradient-based back-propagation algorithms [90] have made possible their various applications. In the context of control engineering applications, neural networks are often trained either to approximate the forward and/or inverse input-output relations of nonlinear systems [91] and are further used in different applications, or to approximate the analytically unobtainable mappings by means of

available data [92]. As discussed before, the state based feedback optimal control is not analytically available, which is often the case for nonlinear power systems. Therefore, the trajectory following approach is to be used to synthesize a state feedback optimal neural control. This will be detailed in what follows.

5.4.1. Time-optimal neural control

The design of optimal feedback controllers for general nonlinear systems is usually untractable. Yet by means of a numerical method, the optimal trajectories and optimal controls can be computed. The information inherent in these outputs and controls help establish the link between them. This link actually leads to a closed-loop feedback to generate approximately an optimal policy. To put this more specifically for time-optimal control, the link thereof can be completely characterized by the associated switching curve in state space. Suppose the switching curve is represented by $S(x) = 0$ where x is the state vector of the system of interest. Note that as shown previously, the control range can be converted to $[-1, \ 1]$. Then the optimal control u of a bang-bang type can be given by

$$u = sgn(S(x)) \tag{5.36}$$

or

$$u = -sgn(S(x)) \tag{5.37}$$

where the function $sgn(.)$ is defined by $sgn(S) = \begin{cases} 1, & if \ \ S > 0 \\ -1, & if \ \ S < 0 \end{cases}$

Denote $sgn(S(.))$ by g. Then $u = g(x)$.

The off-line optimal trajectories and optimal controls may be computed by a gradient based numerical method, say switching-time-variation method [7, 13], which makes effective use of the linearity in control of the nonlinear system. The function g can then be approximated by training a neural network.

Denote by Ω the region of stability interest, which is assumed to be compact. Then the switching curve divides the region into two parts. On one side of the switching curve (or manifold in general), the optimal control takes one extremal value of the confined control while on the other side the optimal control takes the other control limit. From this observation a pattern recognition scheme is proposed as follows. Note that the true state feedback control $u = g(x)$ displays a discontinuity on the switching curve (or manifold in general). This may lead to the training of a neural network a difficult job. Also note that sufficiently many trajectories uniformly distributed in the region of interest may ensure the desired approximation with respect to Lebesgue measure. Let the function realized by a neural network be denoted by $u = NN(x)$. With a number of off-line generated trajectories which are approximately distributed in the region of interest, the trained neural network tends to produce the outputs closely approximating the optimal control on both sides of the switching curve while it is likely that some mismatch error may occur in some neighborhood containing the switching curve and that the output of the neural network tends to take a positive (or negative) value when the desired control takes the positive (or accordingly negative) limit. Since the time-optimal control only takes the extremal values, the control by means of the trained neural network can be formed as

$$u = sgn(NN(x)) \tag{5.38}$$

or

$$u = -sgn(NN(x)) \tag{5.39}$$

Therefore, a new neural network may be formed by means of a conventional neural network followed by a neuron with a heaviside sigmoidal function. This is shown in Figure 5.1.

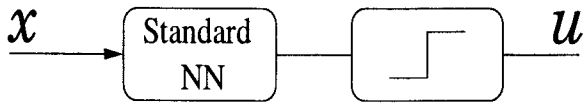


FIGURE 5.1: Neural-net-based time-optimal state feedback control

Remark (1): This structure can recognize the optimal control pattern which is characterized by a switching curve. The output of the new neural network precisely matches the optimal control on both sides of the switching curve except in some small neighborhood containing the switching curve. The training patterns are in the form of (x, u).

Remark (2): This structure will be a component of the lower level neural networks in the hierarchical neural network architecture which is detailed next.

Remark (3): The optimal controls, as the desired output of a neural network can be scaled by a positive factor, say β. There might be some β's such that the neural network is trained satisfactorily well. Then the activation of the heaviside function gives out a control of a bang-bang type.

Remark (4): The above method can be applied to system (5.6) or system (5.8) as long as the parameter is specified or system (5.10) with the parameters specified. The resulting time-optimal neural controllers for different cases form the lower level neural controllers in the hierarchical neural network architecture which will be studied.

Remark (5): The alternative for inputs to the neural network, for system (5.6), (5.8) or (5.10), may be the rotor angle and its previous value since they can be used to reconstruct a state approximation, and since it may affect the training since the deviated rotor speed is too small relative to the rotor angle.

5.4.2. Near time-optimal hierarchical neural control

For parameterized systems with fixed parameters, for instance, system (5.6) (system (5.8) or (5.10) exactly takes the same form as system (5.6) does once related parameters are given), the transient stabilization of power systems can be done by means of time-optimal neural control even though the explicit analytic form of the state based feedback control is not available. The same task, however, may become more challenging if the parameters of the load are not known, which calls for somewhat robust control. Note that the range for the parameters can often be assumed to be specified without loss of generality. The parameter (or parameter vector) space can be tessellated into a number of sub-intervals (or rectangles, or rectangles on high-dimension case, simply called rectangles hereafter). For each case on which the parameter (or parameter vector) corresponding to an endpoint (or vertex) is specified, the case-specific time-optimal neural controller can be trained. For simplicity, these cases may be termed as nominal cases. It is then expected that a near time-optimal control may be synthesized by making use of the information extracted from each nominal case. However, the information abouth those nominal cases, for which the unknown parameter (or parameter vector) lies within the sub-interval (or rectangle) determined by the corresponding endpoints (or vertices), may be more relevant than that about other nominal cases. Therefore, a rough estimate of the parameter (or parameter vector) is necessary in order to determine which sub-interval (or rectangle) it is within. After such an identification, a control u corresponding to the unknown parameters can be synthesized as

$$u = \sum_{i=1}^{M} \alpha_i u_i \tag{5.40}$$

where the time-optimal neural control u_i corresponds to a specific parameter case; α_i is the multiplier for control u_i; and $\sum_{i=1}^{M} \alpha_i = 1$ and $\alpha_i \geq 0$. The resulting hierarchical near time-optimal neural controller is shown in Figure 5.2.

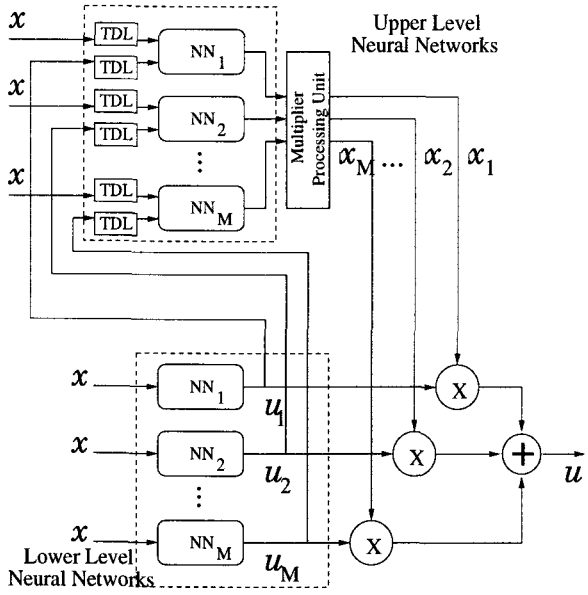


FIGURE 5.2: Hierarchical time-optimal neural control

Remark (1): TDL stands for tapped delay line such that the previous values and also the current value are available to feed into the input of neural networks.

Remark (2): Note that the outputs of lower level neural networks as lower level controls take a value of either -1 or +1. For state $x(k-1)$ at instant $(k-1)T$ (T is the sampling period), it can be driven to $x(k)$ by means of an optimal control $u_i(k-1)$. Also note that for some optimal trajectories and corresponding optimal controls, they are related in the second equation of system (5.6), (5.8) or (5.10). This means that the multiplier $\alpha_i(k)$ may be somehow related to $u_i(k)$. It is then expected that $(x(k-1), x(k), u_i(k-1), u_i(k))$ may characterize the behavior specific to the ith case. Therefore, the training patterns are in such a form $((x(k-1), x(k), u_i(k-1), u_i(k)), \alpha_i(k))$. As pointed out previously, the state $x(k)$ can be replaced by the rotor angle and its previous values. It is desired that the multiplier α_i may take 1 if the trajectories and controls are specific to

the ith case and for other cases the multiplier α_i may take 0. Based on these arguments, the upper level neural networks can be trained.

Remark (3): Since $u_i \in [-1, \ +1]$, then $u = \sum_{i=1}^{M} \alpha_i u_i \in [-1, \ +1]$, too. And the outputs of the upper level neural networks need normalization (still denote them by α_i's) such that $\sum_{i=1}^{M} \alpha_i = 1$. Note that all u_i's only take either +1 or -1. Thus, if all u_i's have the same sign, then from $u = \sum_{i=1}^{M} \alpha_i u_i$, $u = u_i$. For other situations, let $A_1 = \sum_{i \in \{i: \ u_i=1, i \in [1, \ M]\}} \alpha_i$; let $A_2 = \sum_{i \in \{i: \ u_i=-1, i \in [1, \ M]\}} \alpha_i$. Then if $A_1 > A_2$, $u = 1$; if $A_1 < A_2$, $u = -1$. These operations are mainly done by the multiplier processing unit, which is shown in Figure 5.2. Therefore, the resulting control is still a bang-bang type, which drives the system to a neighborhood of the equilibrium in near optimal time.

Remark (4): To accomodate the possible computation errors involved in obtaining the off-line optimal trajectories and to avoid the so-called chattering phenomenon, the linear feedback controller is necessary, which also performs very effectively around the equilibrium. These will be illustrated through simulations in section 5.6..

It is pointed out that the proposed control scheme can be applied to cases I and II, described in section 5.2.1..

5.4.3. Adaptive near time-optimal hierarchical neural control

Consider the system Equation (5.12), where the load P_l, hence c_{10}, is a continuous nonlinear function of ω, but its explicit form may not be known. This is usually the case in reality, which accounts for the fluctuation and random nature of loads. The modeling of an aggregate load is usually useful from available data and measurements. As is known, load modeling is an important issue for voltage stability analysis. Load modeling was studied in [93], where static and dynamic load modeling through neural networks and corresponding voltage stability analyses were discussed. Here again, a neural

network is trained with available data to approximate the nonlinear load such that the approximation error is uniformly bounded. That is, the load can be modeled as $\hat{P}_l = NN(\omega)$, or equivalently $c\hat{_{10}} = NN(\omega)$. And the approximation error can be expressed as $e_1 = c\hat{_{10}} - c_{10}$ with $|e_1| < \epsilon_{e_1}$, where ϵ_{e_1} is a pre-specified positive number.

Note that $c\hat{_{10}}$ is a continuous function of ω with a compact support since the maximum value and minimum value for ω are usually physically determined for stability concern. $c\hat{_{10}}$ can then be approximated by a piece-wise linear function $c\tilde{_{10}}$ such that the approximation error $e_2 = c\tilde{_{10}} - c\hat{_{10}}$ is uniformly bounded by a positive number ϵ_{e_2}. That is, $|e_2| < \epsilon_{e_2}$.

It then follows that $|c\tilde{_{10}} - c_{10}| = |c\tilde{_{10}} - c\hat{_{10}} + c\hat{_{10}} - c_{10}| < |c\tilde{_{10}} - c\hat{_{10}}| + |c\hat{_{10}} - c_{10}| < \epsilon_{e_1} + \epsilon_{e_2}$.

It is then reasonable to assume that for the region of stability interest, the sufficiently small disturbance in c_{10} would not bring about a significant change in the resulting trajectories. Since $c\tilde{_{10}}$ is a piece-wise linear function, the near time-optimal hierarchical neural control scheme applies. And since c_{10} can be approximated by a piece-wise linear function with a uniformly bounded small error, the near time-optimal hierarchical neural control scheme also applies with some adaptations addressed in the following.

At sampling instant kT (T is the sampling period), from the measurements about the state, the load is estimated by the trained neural network $c\hat{_{10k}}$. The previous estimate $c\hat{_{10k-1}}$ and the current estimate $c\hat{_{10k}}$ can be used to identify the coefficients α_1 and α_2 involved in an affine approximation $c\tilde{_{10}} = \alpha_1 + \alpha_2\omega$ for the period of time $[(k-1)T,\ kT]$. Note that α_1 and α_2 are time-varying. Since for the period of time $[(k-1)T,\ kT]$, the load is approximately identified as an affine function of ω, the near time-optimal hierarchical neural control structure then applies, which first locates the parameter vector $(\alpha_1,\ \alpha_2)$ in the tessellated parameter vector space, and enables the proper lower-level time-optimal neural controllers. Since α_1 and α_2 are time-varying, then on-line estimation of the load helps achieve a piece of affine approximation for the specific period of time, and initiates the corresponding lower-level time-optimal neural controllers.

Remark: As the system is stabilized and is gradually driven toward its equilibrium, the rotor speed is approaching a constant, the load must also approach a constant. Therefore, the range for the parameters of the load thereafter shrinks, and enables only the lower-level time-optimal neural controllers correpsonding to the vertices of the sub-region encircling the almost fixed parameter vector involved in the load. As the system is brought to some small neighborhood of its equilibrium, then a linearized controller is enabled to take over, and quickly drive the system to the equilibrium.

5.5. Theoretical justification

This section provides theoretical support for the control schemes developed in the last section.

5.5.1. Switching manifold approximation

Consider the system Equation (5.6)

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = c_1 - c_2\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta)v \end{cases} \tag{5.41}$$

Let $x = [\delta \ \ \omega]^\tau$, $a(x) = [\omega_b\omega \ \ c_1 - c_2\omega - c_3 \sin(\delta_e + \delta)]^\tau$, $B(x) = [0 \ \ c_4 \sin(\delta_e + \delta)]^\tau$, and $f(x) = a(x) + B(x)v$. The above equation can be rewritten as

$$\dot{x} = a(x) + B(x)v \tag{5.42}$$

Note that $v \in [-1, \ +1]$.

Suppose that with the initial condition $x(t_0) = x_0$, a proper bang-bang control $v(t)$ for $t \in [t_0, \ \ t_f]$ can be found such that the state can be driven to the origin at the instant t_f. Note that the control can be completely specified by giving the switching vector $\tau^N = [\tau_1^N \ \ \tau_2^N \ \ \cdots \tau_N^N]^\tau$ (where N designates the number of switching times) and

the first initialized control, and that the corresponding trajectory is a function of x_0 and the switching times, and can be denoted by $x(x_0, t)$. For our interest, without loss of generality, the initial control is assumed to be positive.

Suppose there is a perturbation in the initial state while this does not cause a change of the sign of the initial control and a change of the number of the switching either. Let the perturbed initial state be $y_0 \notin x^*(x_0, t)$ (the optimal trajectory starting from x_0), and the resulting switching vector $\hat{\tau}^N = [\hat{\tau}_1^N \ \hat{\tau}_2^N \ \cdots \hat{\tau}_N^N]^\tau$.

In what follows, it is shown that for the small change in the initial state, the switching times may make an accordingly small change in order to drive the final state to the origin.

Integrating the system equation from t_0 to t_f yields

$$\begin{aligned} x(x_0, t_f) = x_0 \ &+ \ \int_{t_0}^{\tau_1^N} [a(x(x_0,t)) + B(x(x_0,t))]dt + \cdots \\ &+ \ \int_{\tau_i^N}^{\tau_{i+1}^N} [a(x(x_0,t)) + (-1)^i B(x(x_0,t))]dt + \cdots \\ &+ \ \int_{\tau_N^N}^{t_f} [a(x(x_0,t)) + (-1)^N B(x(x_0,t))]dt \end{aligned} \tag{5.43}$$

Observation from the above equation indicates that if x_0 is fixed, then $x(x_0, t)$ is a continuous function of the switching vector. Since $f(x)$ satisfies the Lipschitz condition [94], that is, a constant k exists such that $||f(x_1) - f(x_2)|| \leq k||x_1 - x_2||$ for all x_1,x_2 in the region of interest, the solution $x(x_0, t)$ is unique. In a word, $x(x_0, t)$ is a continuous function of x_0 and the switching times.

Note that $\frac{\partial x(x_0,t_f)}{\partial \tau_i^N} = 2(-1)^i B(x(x_0, \tau_i^N))$ for $i = 1, \cdots, N$; and $\frac{\partial x(x_0,t_f)}{\partial t_f} = a(x(x_0, t_f)) + (-1)^N B(x(x_0, t_f))$.

It follows that

$$\begin{aligned} dx(x_0, t_f) = \ & dx_0 + \sum_{i=1}^{N} 2(-1)^i B(x(x_0, \tau_i^N))d\tau_i^N \\ &+ \ [a(x(x_0, t_f)) + (-1)^N B(x(x_0, t_f))]dt_f \end{aligned}$$

$$+ \int_{t_0}^{\tau_1^N} \frac{\partial[a(x(x_0,t)) + B(x(x_0,t))]}{\partial x_0} dx_0 dt + \cdots$$
$$+ \int_{\tau_i^N}^{\tau_{i+1}^N} \frac{\partial[a(x(x_0,t)) + (-1)^i B(x(x_0,t))]}{\partial x_0} dx_0 dt + \cdots$$
$$+ \int_{\tau_N^N}^{t_f} \frac{\partial[a(x(x_0,t)) + (-1)^N B(x(x_0,t))]}{\partial x_0} dx_0 dt \tag{5.44}$$

That is,

$$dx(x_0,t_f) = \sum_{i=1}^{N} 2(-1)^i B(x(x_0,\tau_i^N))d\tau_i^N + [a(x(x_0,t_f)) + (-1)^N B(x(x_0,t_f))]dt_f$$
$$+ \{I + \int_{t_0}^{\tau_1^N} \frac{\partial[a(x(x_0,t)) + B(x(x_0,t))]}{\partial x_0} dt + \cdots$$
$$+ \int_{\tau_i^N}^{\tau_{i+1}^N} \frac{\partial[a(x(x_0,t)) + (-1)^i B(x(x_0,t))]}{\partial x_0} dt + \cdots$$
$$+ \int_{\tau_N^N}^{t_f} \frac{\partial[a(x(x_0,t)) + (-1)^N B(x(x_0,t))]}{\partial x_0} dt\} dx_0 \tag{5.45}$$

Define $C(x_0,\tau^N,t_f) = I + \int_{t_0}^{\tau_1^N} \frac{\partial[a(x(x_0,t))+B(x(x_0,t))]}{\partial x_0} dt + \cdots +$
$\int_{\tau_i^N}^{\tau_{i+1}^N} \frac{\partial[a(x(x_0,t))+(-1)^i B(x(x_0,t))]}{\partial x_0} dt + \cdots + \int_{\tau_N^N}^{t_f} \frac{\partial[a(x(x_0,t))+(-1)^N B(x(x_0,t))]}{\partial x_0} dt.$

Notice that $dx(x_0,t_f) = 0$ since the desired final state is the origin. Then we have

$$\sum_{i=1}^{N} 2(-1)^i B(x(x_0,\tau_i^N))d\tau_i^N + [a(x(x_0,t_f)) + (-1)^N B(x(x_0,t_f))]dt_f = -C(x_0,\tau^N,t_f)dx_0 \tag{5.46}$$

It can be readily shown that $C(x_0,\tau^N,t_f)$ is bounded. Then it follows that there may be some freedom for $d\tau_i^N$ and dt_f to take some small values. Hence, for any positive small number ϵ, there exists a positive small number δ such that $if ||x_0^2 - x_0^1|| < \delta$, $||\tau^{N,2} - \tau^{N,1}|| < \epsilon$ where x_0^1 and x_0^2 are different initial conditions; and $\tau^{N,1}$ and $\tau^{N,2}$ are corresponding switching vectors.

It should be noted from Equation (5.45) that since all the first-order partial derivatives are bounded, any small perturbations to both the switching times and the initial state only cause small change to the final state.

Therefore, some conclusions naturally follow.

Conclusion 1: Suppose Ω is a compact region where with proper control the optimal trajectories starting in the compact region will still remain in it. That is, for any initial state $x_0 \in \Omega$, there exists an optimal control $v = g(x)$ which is a state feedback control such that the state can be driven to the origin in a finite amount of time. Let the switching curve (or manifold) be S. Let a region $D \subset \Omega$ surrounding the switching curve be defined as $D = \{x : ||x - y|| < \epsilon \;\; y \in S;\; x \in \Omega\}$. Then a neural controller $u = NN(x)$ which only takes -1 or +1 with x being the state, can be trained such that if $x \in \Omega - D$, $||u(x) - v(x)|| = 0$. Then for any $\epsilon_1 > 0$ and $\epsilon_2 > 0$, there exists $\epsilon_3 > 0$ such that if $||x_0^{'} - x_0|| < \epsilon_3$, there exists the terminal time t_f such that $|t_f - t_f^*| < \epsilon_1$,and $||x(x_0^{'}, t_f) - x^*(x_0, t_f)|| < \epsilon_2$ where t_f^* is the optimal terminal time for the initial state x_0; and $x^*(x_0, t)$ is the optimal trajectory starting form x_0.

Remark: First of all, the optimal control $v = g(x)$ with $x \in \Omega$ is a discontinuous function only on $x \in S$. It can be approximated with a continuous function,say $v = h(x)$, with the same support with sufficiently small error $\gamma > 0$ such that $h = g$ if $x \in \Omega - D$, and $|h(.) - g(.)| < \gamma$ for $x \in D$. Then a neural network $NN1(.)$ can be trained to approximate the function $h(.)$ such that $|NN1(.) - h(.)| < \gamma_1$ with γ_1 being an arbitrarily small positive number. Note that $h(.)$ takes a value of -1 or +1 if $x \in \Omega - D$. Then $1 - \gamma_1 < NN1(.) < 1 + \gamma_1$ or $-1 - \gamma_1 < NN1(.) < -1 + \gamma_1$ for $x \in \Omega - D$. As long as γ is chosen such that $\gamma < 1$, then $sgn(NN1(.)) = h(.)$ for $x \in \Omega - D$. But $sgn(NN1(.))$ is another neural network. Thus, the existance of such a neural controller is assured.

Conclusion 2: Suppose Ω is a compact region where with proper control the optimal trajectories starting in the compact region will still remain in it. That is, for any initial state $x_0 \in \Omega$, there exists an optimal control $v = g(x)$ which is a state feedback control such that the state can be driven to the origin in a finite amount of time. Let the switching curve (or manifold) be S. Let S_l be the piecewise linear approximation of the switching curve. Suppose that $x \in S$ and $x^{'} \in S_l$ where x and x' are points of an optimal

trajectory. Then for any $\epsilon_1 > 0$ and $\epsilon_2 > 0$, there exists $\epsilon_3 > 0$ and $\epsilon_4 > 0$, such that if $\sup_{x \in S;\, x' \in S_l} ||x - x'|| < \epsilon_3$, and $||x_0' - x_0|| < \epsilon_4$, there exists the terminal time t_f such that $|t_f - t_f^*| < \epsilon_1$,and $||x(x_0', t_f) - x^*(x_0, t_f)|| < \epsilon_2$ where t_f^* is the optimal terminal time for the initial state x_0; and $x^*(x_0, t)$ is the optimal trajectory starting form x_0.

Remark: Such a piecewise linear approximation of the switching manifold may be realized by a constructive neural network.

5.5.2. Support for construction of hierarchical neural controllers

Consider again the system equation

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = c_1 - c_2\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta)v \end{cases} \tag{5.47}$$

Note that since a load is considered in the SMIB case, c_1 is now a fixed unknown scalar. c_1 is assumed to be within the interval $I_c = [c_{1min},\ \ c_{1max}]$. Thus, the above system equation is equivalent to Equation (5.8).

For any fixed c_1, the control can be designed to stabilize the system in a near optimal manner. The question then arises how an effective control can be designed for the fixed but unknown parameter. One natural solution would be identifying the parameter first and then activating the according control. One other alternative is to use all available specific controllers corresponding to specific cases, and make a combination of them. In what follows, the theoretical aspects for the latter case are investigated about how such a combined controller can be synthesized and how well such a synthesized controller performs.

For any given initial state x_0, and the corresponding optimal switching vector τ^N and the final time t_f, if c_1 is a variable, then the state $x = [\delta \ \ \omega]^\tau$ will be a continuous function of the switching vector, t_f and c_1. Suppose there is an increment dc_1 in c_1, and

suppose that this variation in c_1 does not cause the structure change in the system (which means the behavior of the system does not change much), and the goal is still to drive the system state to the origin. Suppose that this will cause some increments in the switching vector τ^N and t_f. Therefore, we have the following equation by means of perturbation analysis.

$$\begin{aligned} 0 = & \sum_{i=1}^{N} 2(-1)^i B(x(x_0, \tau_i^N; c_1)) d\tau_i^N + [a(x(x_0, t_f; c_1)) + (-1)^N B(x(x_0, t_f; c_1))] dt_f \\ & + \int_{t_0}^{t_f} [0 \;\; 1]^\tau dc_1 dt \\ & + \int_{t_0}^{\tau_1^N} \frac{\partial [a(x(x_0, t; c_1)) + B(x(x_0, t; c_1))]}{\partial x} \frac{\partial x}{c_1} dc_1 dt + \cdots \\ & + \int_{\tau_i^N}^{\tau_{i+1}^N} \frac{\partial [a(x(x_0, t; c_1)) + (-1)^i B(x(x_0, t; c_1))]}{\partial x} \frac{\partial x}{c_1} dc_1 dt + \cdots \\ & + \int_{\tau_N^N}^{t_f} \frac{\partial [a(x(x_0, t; c_1)) + (-1)^N B(x(x_0, t; c_1))]}{\partial x} \frac{\partial x}{c_1} dc_1 dt \end{aligned} \tag{5.48}$$

Define $C(c_1, \tau^N, t_f) = \int_{t_0}^{\tau_1^N} \frac{\partial [a(x(x_0,t;c_1)) + B(x(x_0,t;c_1))]}{\partial x} \frac{\partial x}{c_1} dt +$
$\int_{\tau_i^N}^{\tau_{i+1}^N} \frac{\partial [a(x(x_0,t;c_1)) + (-1)^i B(x(x_0,t;c_1))]}{\partial x} \frac{\partial x}{c_1} dt$
$+ \int_{\tau_N^N}^{t_f} \frac{\partial [a(x(x_0,t;c_1)) + (-1)^N B(x(x_0,t;c_1))]}{\partial x} \frac{\partial x}{c_1} dt$

And $C(c_1, \tau^N, t_f)$ can be further expressed as

$$C(c_1, \tau^N, t_f) = \int_{t_0}^{t_f} \begin{bmatrix} 0 & \omega_b \\ (-c_3 - c_4)\cos(\delta_e + \delta) & -c_2 \end{bmatrix} \frac{\partial x}{c_1} dt$$
$$+ \sum_{i=1}^{N-1} \int_{\tau_i^N}^{\tau_{i+1}^N} \begin{bmatrix} 0 & \omega_b \\ (-c_3 - (-1)^i c_4)\cos(\delta_e + \delta) & -c_2 \end{bmatrix} \frac{\partial x}{c_1} dt$$
$$+ \int_{\tau_N^N}^{t_f} \begin{bmatrix} 0 & \omega_b \\ (-c_3 - c_4)\cos(\delta_e + \delta) & -c_2 \end{bmatrix} \frac{\partial x}{c_1} dt.$$

It follows that

$$\begin{aligned} 0 = & \sum_{i=1}^{N} 2(-1)^i B(x(\tau_i^N)) d\tau_i^N + [a(x(t_f)) + (-1)^N B(x(t_f))] dt_f \\ & + \{(t_f - t_0)[0 \;\; 1]^\tau + C(c_1, \tau^N, t_f)\} dc_1 \end{aligned} \tag{5.49}$$

Since we assume that the variation in c_1 does not cause any structure change in the system, for any t, $\frac{\partial x}{c_1}$ is bounded. Thus, $C(c_1, \tau^N, t_f)$ is bounded. Since $\frac{\partial x}{\partial \tau_i^N}$ for $i = 1, \cdots, N$ and $\frac{\partial x}{\partial t_f}$ are bounded, any small change dc_1 in c_1 will cause small changes in the switching vector and the terminal time.

Based on the above discussion, the interval I_c is devided into $M-1$ parts such that $c_{1min} = c_1^1 < c_1^2 < \cdots < c_1^M = c_{1max}$. Since $c_1 \in I_c$, c_1 must be in some interval $[c_1^i,\ c_1^{i+1}]$. Further suppose there exist a number of optimal controllers $u^i(x)$, corresponding to the parameter c_1^i, where i ranges from 1 to M , for the above system. For any $c_1 \in I_c$, define the combined controller by $u(x) = \sum_{j=1}^{M} \lambda_j u^j(x)$ where $0 \leq \lambda_j \leq 1$ for $j = 1, \cdots, M$ and $\sum_{j=1}^{M} \lambda_j = 1$. But since there exists i such that $c_1 \in [c_1^i,\ c_1^{i+1}]$, it is then reasonable to use, for synthesis of a control corresponding to c_1, only the information about the system and control corresponding to both cases where the parameter takes a value of c_1^i and c_1^{i+1},respectively.

Therefore, a controller can be synthesized in two stages that follow.

First, identify the sub-interval $[c_1^i,\ c_1^{i+1}]$ that c_1 is likely within.

Secondly, construct the control by means of a combination of the pre-designed controller $u^{i*}(x)$ and $u^{i+1*}(x)$. That is, $u(x) = \sum_{j=i}^{i+1} \lambda_j u^{j*}(x)$.

Remark (1): since the analytic form for u^{j*} may not be available, it is then therefore necessary to use the method described before to train a neural controller $NN^j(x)$ for each case.

Remark (2): since λ_j is not known, it has to be identified. This can be done by using the available optimal trajectories to train another neural network. That is, $\lambda_j = NN_s^j(x, \dot{x})$.

Remark (3): to identify the parameter c_1, first feed the initial state to all M neural controllers. Let the number of the resulting controls taking $+1$ be M_1. If $M_1 \geq M/2$, then the control for the first cycle takes $+1$; otherwise it takes -1. With the control for the first cycle, the measurement can then be used to determine the parameter c_1. That

is, $c_1 = \dot{\omega} + c_2\omega + c_3\sin(\delta) + c_4\sin(\delta)v$. Note δ is now the rotor angle not the rotor angle deviation from the equilibrium, because for each c_1, the equilibrium is different from the other.

It should be noted that for system (5.10) with a couple of unknown parameters similar theoretical results and implementation procedures can be readily obtained.

5.5.3. Approximate time-optimal adaptive neural controller

In the following, it will be shown that the procedures used for synthesis of neural controllers for cases I and II, described in section 5.2., can be used for case III.

Gronwall-Bellman Inequality [95]: Suppose that $\phi(t)$, $\Psi(t)$ and $\mu(t)$ are real, continuous functions with $\mu(t) \geq 0$ for all $t \geq t_0$. Then the implicit inequality

$$\phi(t) \leq \Psi(t) + \int_{t_0}^{t} \mu(\sigma)\phi(\sigma)d\sigma, \quad t \geq t_0 \tag{5.50}$$

implies the explicit inequality

$$\phi(t) \leq \Psi(t) + \int_{t_0}^{t} \mu(\sigma)\Psi(\sigma)e^{\int_{\sigma}^{t}\mu(\tau)d\tau}d\sigma, \quad t \geq t_0 \tag{5.51}$$

In the following, it will be shown for case III that a bounded error involved in the identification of the load only results in a bounded deviation from the desired trajectory. Here,

$$\begin{cases} \dot{\delta} = \omega_b\omega \\ \dot{\omega} = c_1 - c_{10} - c_2\omega - c_3\sin(\delta_e + \delta) - c_4\sin(\delta_e + \delta)v \end{cases} \tag{5.52}$$

where $c_{10} = \frac{P_l}{M}$, and c_1, c_2, c_3, c_4, δ_e, and ω_b are all constants.

A neural network is trained with available data to approximate the nonlinear load such that the approximation error is uniformly bounded. That is, the load can be modeled as $\hat{P}_l = NN(\omega)$, or equivalently $\hat{c_{10}} = NN(\omega)$ (mathematically, $\hat{c_{10}}$ should be written as $\hat{c_{10}}(\omega)$; for brevity, the argument is dropped if no confusion arises). And the approximation

error can be expressed as $e_1 = \hat{c_{10}} - c_{10}$ with $|e_1| < \epsilon_{e_1}$, where ϵ_{e_1} is a pre-specified positive number.

Note that $\hat{c_{10}}$ is a continuous function of ω with a compact support since the maximum value and minimum value for ω are usually physically determined for stability concern. $\hat{c_{10}}$ can then be approximated by a piece-wise linear function $\tilde{c_{10}}$ such that the approximation error $e_2 = \tilde{c_{10}} - \hat{c_{10}}$ is uniformly bounded by a positive number ϵ_{e_2}. That is, $|e_2| < \epsilon_{e_2}$.

It then follows that $|\tilde{c_{10}} - c_{10}| = |\tilde{c_{10}} - \hat{c_{10}} + \hat{c_{10}} - c_{10}| < |\tilde{c_{10}} - \hat{c_{10}}| + |\hat{c_{10}} - c_{10}| < \epsilon_{e_1} + \epsilon_{e_2}$.

Therefore, c_{10} can be expressed as

$$c_{10} = \tilde{c_{10}} + e \tag{5.53}$$

where $\| e(.) \| < \epsilon$, and ϵ is a positive number.

It follows that

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = c_1 - \tilde{c_{10}} - e - c_2\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta)v \end{cases} \tag{5.54}$$

Note that the optimal control can be obtained for

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = c_1 - \tilde{c_{10}} - c_2\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta)v \end{cases} \tag{5.55}$$

through the method described before.

Define $x = [\delta \ \omega]^\tau$; $a(x) = [\omega_b \omega \ \ c_1 - \tilde{c_{10}} - c_2\omega - c_3 \sin(\delta_e + \delta)]^\tau$; and $B(x) = [0 \ \ -c_4 \sin(\delta_e + \delta)]^\tau$.

Then the above two equations can be writen compactly as

$$\dot{x} = a(x) + Ce + B(x)v \tag{5.56}$$

where $C = [0 \ 1]^\tau$.

and

$$\dot{x} = a(x) + B(x)v \tag{5.57}$$

Since the optimal control exists for $\dot{x} = a(x) + B(x)v$, with the given initial condition $x(t_0) = x_0$, we have, by integration of the above two equations from t_0 to t

$$x_1(t) = x_1(t_0) + \int_{t_0}^{t} [a(x_1(s)) + Ce + B(x_1(s))v(s)]ds \tag{5.58}$$

and

$$x_2(t) = x_2(t_0) + \int_{t_0}^{t} [a(x_2(s)) + B(x_2(s))v(s)]ds \tag{5.59}$$

By noting that $x_1(t_0) = x_2(t_0) = x_0$, substraction of the above two equations yields

$$x_1(t) - x_2(t) = \int_{t_0}^{t} \{a(x_1(s)) - a(x_2(s)) + Ce + [B(x_1(s)) - B(x_2(s))]v(s)\}ds \tag{5.60}$$

Note that, by Taylor's theorem, $a(x_1(s)) - a(x_2(s)) = a_T(x_1(s) - x_2(s))$ and $B(x_1(s)) - B(x_2(s)) = B_T(x_1(s) - x_2(s))$, where

$$a_T = \begin{bmatrix} 0 & \omega_b \\ -c_3 \sin(\delta + \delta_e) & -c_2 \end{bmatrix} \text{ with } \delta \text{ lying between } \delta_1 \text{ and } \delta_2;$$

and

$$B_T = \begin{bmatrix} 0 & 0 \\ -c_4 \sin(\delta + \delta_e) & 0 \end{bmatrix} \text{ with } \delta \text{ lying between } \delta_1 \text{ and } \delta_2.$$

Define $\Delta x(t) = x_1(t) - x_2(t)$. Then we have

$$\Delta x(t) = \int_{t_0}^{t} Ceds + \int_{t_0}^{t} [a_T(x(s))\Delta x(s) + B_T(x(s))\Delta x(s)v(s)]ds \tag{5.61}$$

If the appropriate norm of both sides of the above equation is taken and the triangle inequality is applied to it, the following is obtained:

$$||\Delta x(t)|| \leq \int_{t_0}^{t} ||Ce||ds + || \int_{t_0}^{t} [a_T(x(s))\Delta x(s) + B_T(x(s))\Delta x(s)v(s)]ds|| \tag{5.62}$$

Note that e is uniformly bounded (i.e., $|e| < \epsilon$), $|v(t)| \leq 1$, $||a_T|| = \sup_{x \in \Omega} a_T(x) < \infty$, and $||B_T|| = \sup_{x \in \Omega} B_T(x) < \infty$.

It follows that

$$\begin{aligned} ||\Delta x(t)|| &\leq \epsilon(t - t_0) + \int_{t_0}^{t} |||[a_T(x(s))\Delta x(s) + B_T(x(s))\Delta x(s)v(s)]|||ds \\ &\leq \epsilon(t - t_0) + (||a_T|| + ||B_T||) \int_{t_0}^{t} ||\Delta x(s)||ds \end{aligned} \tag{5.63}$$

Application of *Gronwall-Bellman* Inequality yields

$$\begin{aligned} ||\Delta x(t)|| &\leq \epsilon(t-t_0) + \int_{t_0}^{t} (||a_T|| + ||B_T||)\epsilon(s-t_0)\exp\{\int_{s}^{t} (||a_T|| + ||B||)d\sigma\}ds \\ &\leq \epsilon(t-t_0) + \epsilon(||a_T|| + ||B_T||)\frac{(t-t_0)^2}{2}\exp\{(||a_T|| + ||B_T||)(t-t_0)\} \\ &\leq K\epsilon \end{aligned} \tag{5.64}$$

where $K = (t-t_0)[1 + (||a_T|| + ||B_T||)\frac{(t-t_0)}{2}\exp\{(||a_T|| + ||B_T||)(t-t_0)\}$, and $K < \infty$, for all $t \in [t_0,\ t_f]$.

Roughly speaking, as long as the identified load, by means of an affine function for each period of time, is close to the actual load, through the control corresponding to the identified load, the resulting trajectory corresponding to the actual load is close to the trajectory corresponding to the identified load. Therefore, the procedures developed for synthesis of neural controllers for case I and II, suitably apply for case III.

5.6. Simulations

A SMIB system with a load P_l, described by Equation (5.9), is considered for simulations with the parameters and some related data given as

$\omega_b = 2\pi \times 60$; $M = 3.5$; $P_m = 0.3665$; $D = 2.0$; $V_t = 1.0$; $V_\infty = 0.9$; $X_d = 2.0$; $X_e = 0.35$; $s_e = 0.4$; $s_{max} = 0.75$; $s_{min} = 0.2$;

A controller is to be synthesized to stabilize the postfault SMIB system in minimal time, taking into account the unknown load.

Due to the uncertainty of the load, this can be achieved by application of a somewhat "intelligent" time-optimal control to drive the system to a small neighborhood of the equilibrium in near optimal time and thereafter a linearized feedback controller to take over and maintain the equilibrium.

For brevity, Equation (5.10) is preferred to Equation (5.9). The linearized version of the SMIB system (5.10) around the origin can be obtained as follows:

$$\begin{bmatrix} \Delta\dot{\delta} \\ \Delta\dot{\omega} \end{bmatrix} = \begin{bmatrix} 0 & \omega_b \\ -(c_3 + c_4 v_e)\cos\delta_e & -(c2 + c_{20}) \end{bmatrix} \begin{bmatrix} \Delta\delta \\ \Delta\omega \end{bmatrix} + \begin{bmatrix} 0 \\ -c_4 \sin\delta_e \end{bmatrix} \Delta v \quad (5.65)$$

With the substitution of the parameters, it follows that

$$\begin{bmatrix} \Delta\dot{\delta} \\ \Delta\dot{\omega} \end{bmatrix} = \begin{bmatrix} 0 & 376.9911 \\ -0.0507 & -0.5714 - c_{20} \end{bmatrix} \begin{bmatrix} \Delta\delta \\ \Delta\omega \end{bmatrix} + \begin{bmatrix} 0 \\ -0.0047 \end{bmatrix} \Delta v \quad (5.66)$$

For $c_{20} = 0$, the eigenvalues are $-0.2857 \pm j4.3625$. This indicates that the equilibrium is lowly damped. It can be seen that even for $c_{20} \neq 0$, this low-damping nature will hardly change as long as c_{20} is around the level of c_2 or less. In case of disturbances, this equilibrium may experience the oscillation. Therefore, a linear state-feedback controller is useful to enhance stability around the equilibrium of the SMIB power system.

Let the feedback gain vector be $K = [k_1 \;\; k_2]^\tau$. Then $\Delta v = k_1\Delta\delta + k_2\Delta\omega$.

Note that due to the calculation error involved in computing the off-line time-optimal trajectories and other practical reasons mentioned previously, the system is expected to be driven to a small neighborhood of the equilibrium. This neighborhood can be chararcterized by an elliptic region $(\frac{\Delta\delta}{\epsilon_\delta})^2 + (\frac{\Delta\omega}{\epsilon_\omega})^2 = 1$ where ϵ_δ and ϵ_ω are two positive numbers, and ϵ_δ is much larger than ϵ_ω. The feedback gain k_1 can not take very large values for small-signal analysis. Further, the resulting feedback system matrix becomes $\begin{bmatrix} 0 & 376.9911 \\ -0.0507 - 0.0047k_1 & -0.5714 - 0.0047k_2 \end{bmatrix}$. In order for the oscillation to be suppressed, k_1 must take a large value with a negative sign. To make a trade-off, let k_1 be 0. However, k_2 can take a relative larger value. But k_2 can not take too large a value in order to keep Δv small enough so that the small signal analysis is validated. Let $k_2 = 1000$. The eigenvalues of the resulting feedback system now become $-2.6357 \pm j3.4881$, which ensures much stronger stability of the SMIB system.

In what follows, approximation of switching manifolds, synthesis of neural controllers, and synthesis of a hierarchical neural controller are demonstrated.

Note that the load in system (5.9), denoted by $P_l(P_0, C)$, is parameterized, and that the each of the parameters thereof can be generally assumed to lie within some range. That is, assume $P_0 \in [P_{0_{min}},\ P_{0_{max}}]$ and $C \in [C_{min},\ C_{max}]$. Tessellate the region spanned by P_0 and C into small sub-regions $A_{i,j}$'s whose vertices are $(P_0^i,\ C^j)$,$(P_0^{i+1},\ C^j)$,$(P_0^i,\ C^{j+1})$ and $(P_0^{i+1},\ C^{j+1})$, where $i = 1, 2, \cdots, N_p - 1$, $j = 1, 2, \cdots, N_c - 1$, $P_{0_{min}} = P_0^1 < P_0^2 < \cdots < P_0^{N_p} = P_{0_{max}}$ and $C_{min} = C^1 < C^2 < \cdots < C^{N_c} = P_{0_{max}}$. Note that once the sub-region $A_{i,j}$ within which the load parameter vector may lie is roughly identified, only the pre-designed lower level time-optimal neural controllers corresponding to the loads $P_l(P_0^i,\ C^j)$, $P_l(P_0^{i+1},\ C^j)$,$P_l(P_0^i,\ C^{j+1})$ and $P_l(P_0^{i+1},\ C^{j+1})$, respectively, can be enabled to synthesize a controller, with the structure proposed in section 5.4., applicable to this unknown load case. For simplicity, let $C = 0$ while P_0 may range from 0 to $P_{0_{max}}$.

By means of the efficient switching-time-variation-method, in the region of stability interest, a number of optimal trajectories, corresponding to different initial conditions as well as the load $P_l = 0$, are generated and shown in Figure 5.3. Note that in Figure 5.3, the equilibrium is the origin, and the rotor angle and the rotor speed are translated from the equlibrium rotor angle and rotor speed, respectively. To make a distinction, the rotor angle and the rotor speed thereof are called the rotor angle deviation and the rotor speed deviation. Hereafter, this will be done for similar cases. Since for different sets of parameters for a load, the corresponding equilibrium points are different, the specific equilibrium will, therefore, be mentioned in case the confusion arises. The trajectories shown in Figure 5.3 are computed in a continuous time setup. For the sake of neural network training, the time continuous trajectories are sampled at a rate of 1 sample per cycle. As pointed out previously, the inputs to a lower-level neural network include the rotor angle and its previous value. The computed trajectories in terms of the rotor angle and its previous value are shown in Figure 5.4. The lower-level neural networks are trained

and neural controllers are obtained in the configuration shown in Figure 5.1. The learned pattern in terms of the rotor angle and its previous value by the neural network is shown in Figure 5.5. Accordingly, the learned pattern in terms of the system state is shown in Figure 5.6. Comparisons between Figure 5.4 and Figure 5.5, and between Figure 5.3 and Figure 5.6, indicate that the lower-level neural network in the proposed configuration performs satisfactorily in terms of the given pattern. For an initial condition corresponding to an off-line calculated optimal trajectory, the resulting trajectory by means of the trained time-optimal neural controller is shown in Figure 5.7. It can be seen that the resulting trajectory and the corresponding off-line optimal trajectory are almost indistinguishable. (Note that a linearized controller further brings the system to the exact equilibrium; this will not be mentioned hereafter unless the confusion arises). The system after experiencing a severe short-circuit fault, from which the resulting initial condition is not trained, loses its stability with the fixed compensation s_e (or v_e). By application of the trained neural controller, the system can be stabilized in near optimal time. The resulting trajectory is shown in Figure 5.8, which is very close to the computed optimal trajectory by means of the STVM method.

Similarly, for a load $P_l = P_m \times 10\%$, the off-line calculated optimal trajectories in the region of interest are shown in Figure 5.9. The corresponding trajectories in terms of the rotor angle and its previous value are shown in Figure 5.10. The learned patterns by a time-optimal neural controller, in terms of the state and in terms of the rotor angle and its previous value, are shown in Figure 5.11 and Figure 5.12, respectively.For an initial condition corresponding to an off-line calculated optimal trajectory, the resulting trajectory by means of the trained time-optimal neural controller is shown in Figure 5.13. It can be seen that the resulting trajectory and the corresponding off-line optimal trajectory are nearly indistinguishable. With employment of the trained neural controller, the system after experiencing a severe short-circuit fault can be brought to a pre-designated small neighborhood of the equilibrium in near optimal time, and further brought to the

equilibrium by a linearized controller, discussed above. The resulting trajectory is shown in Figure 5.14, which is also very close to the computed optimal trajectory by means of the STVM method.

Similarly, other lower-level time-optimal neural controllers can be obtained corresponding to the case on which the load $P_l = P_m \times 20\%, P_m \times 30\%, \cdots$, etc.

Next the training of the upper-level neural networks is addressed. As discussed before, the inputs include the current rotor angle $\delta(k)$ and its previous values ($\delta(k-1)$ and $\delta(k-2)$), and the current control and its previous value ($v(k)$ and $v(k-1)$) coming from the corresponding lower-level neural controller. Each of those neural network can be described by $\alpha_n = UNN_n(\delta(k), \delta(k-1), \delta(k-2), v(k), v(k-1))$. With these upper-level neural networks trained, the outputs of the upper-level neural networks are fed into the multiplier processing unit. The sum of the lower-level controllers multiplied by the resulting multipliers forms the current control to the power system.

The proposed hierarchical neural control, in the configuration shown in Figure 5.2, is examined for a severe short-circuit fault for an unknown load ($P_l = P_m \times 5\%$) , for which the related optimal trajectory are not used for neural network training. Following the proposed identification and control procedures described in section 5.5., the resulting trajectory is shown in Figure 5.15, and is also very close to the off-line calculated time-optimal trajectory.

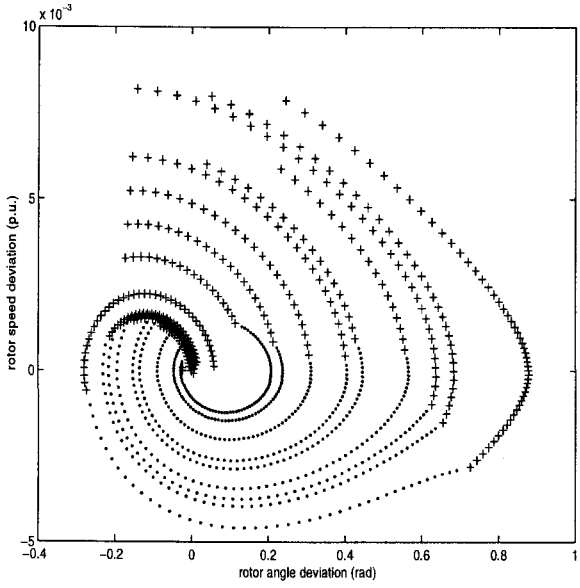


FIGURE 5.3: Time-optimal trajectories calculated by STVM for case $P_l = 0$

5.7. Generalization to more general systems

The results obtained for simplifed power systems with loads can be generalized to more general nonlinear systems. Consider a parameterized nonlinear system described by

$$\dot{x} = f(x, u; c) \tag{5.67}$$

where $x \in R^n$ is the system state; $u \in R^m$ is the admissible control vector; and $c \in R^l$ is the parameter vector confined within some sub-space. Assume for the region Ω of interest the existance of a bang-bang type of control is assured. Through tessellation of the parameter vector space, lower-level time-optimal controllers can be designed corresponding to the vertices of each sub-region. The upper-level neural networks in the proposed hierarchical neural control structure can also be trained in the way discussed previously. Then for an unknown parameter vector, its identification based on the measurements of the state helps

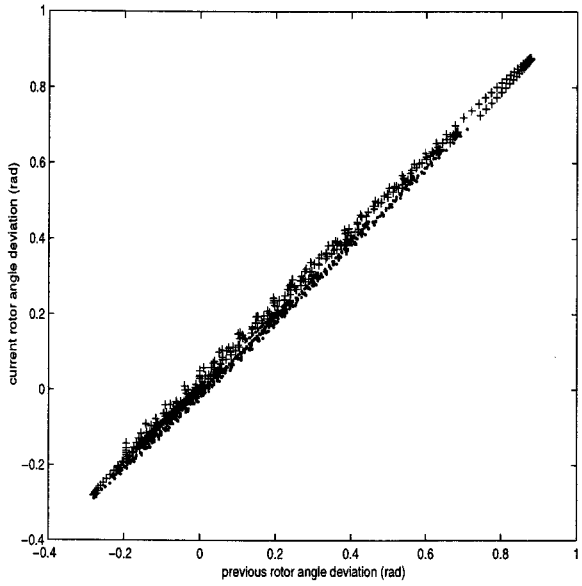


FIGURE 5.4: Rotor angle deviation data for neural net training for case $P_l = 0$

locate which sub-region it is likely within, and then the lower-level neural controllers are enabled corresponding to the vertices of this sub-region. With the state and its previous value feeding in, the upper-level neural networks and additional multiplier processing unit, calculate and process the proper multipliers. The "intelligent" control signal is the sum of the modulated control by the multipliers.

5.8. Conclusions

For efficient utilization of the existing high voltage transmission networks, FACTS devices are installed in order to enhance power system stability. The proper manipulation of the installed FACTS devices are crucial for maintaining power system stability while improving the power transfer capability of transmission networks. The transients initiated

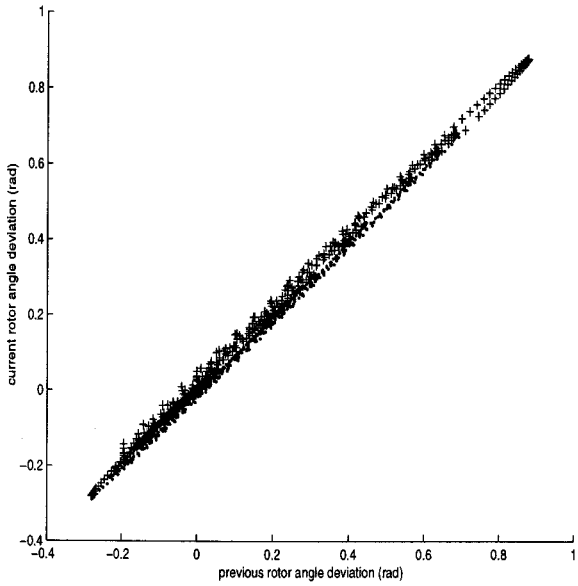


FIGURE 5.5: Learned pattern of rotor angle deviation for case $P_l = 0$

by large faults may interact with the loads, and further cause voltage instability problems. The transient stabilization of power systems is addressed while the load effect is also considered. The uncertainty of the load makes the faulted power system a nonlinear uncertain dynamic system, which is a challenge calling for robust and intelligent control design.

The explicit analytic optimal feedback control is generally not available, but it can be obtained numerically. The numerically obtained optimal trajectories can help produce an approximate near optimal control by training a neural network. The time-optimal control, more specifically, the bang-bang control, is achieved by a neural network such that the feedback control pattern is recognized. This is essential to approximate the switching curve (or manifold). Related theoretical justification is presented. When the load taking a parameterized form is unknown with its parameters fixed but needing identification,

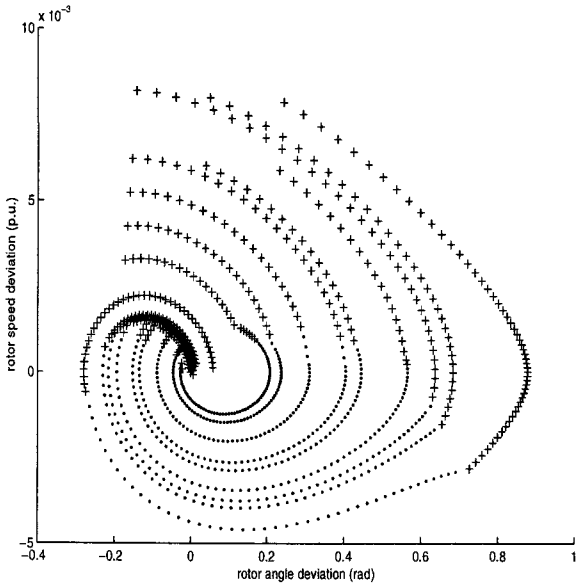


FIGURE 5.6: Learned pattern about time-optimal trajectories for case $P_l = 0$

a reasonable control is synthesized by means of a hierarchical neural network structure. Here, the lower level neural networks are designed so that for a given parametrized load, a corresponding lower level neural network will work well enough to approximate the minimal time control, whose upper level neural networks assign corresponding values to the "weights", or membership values in the context of fuzzy control, to the associated lower level control. Since the possible control values that the optimal control may take are discrete, say -1 or +1, the current lower level controls can then be fed into the input of the upper level neural networks. This is motivated by noting the fact that the the proper "weights" have something to do with the corresponding lower level control. The "weights" for controls are not simply operated to obtain a weighted sum — the resulting control. Rather they are somehow operated so that the resulting control is also a bang-bang type, which may avoid the otherwise longer duration for transient stabilization. The tessellation

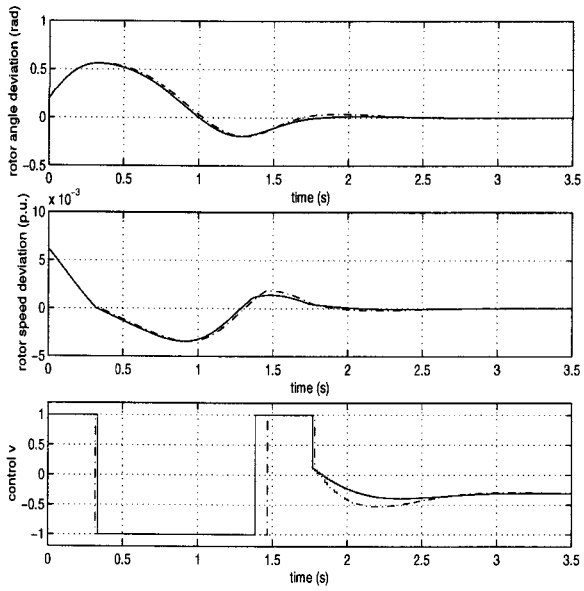


FIGURE 5.7: Training performance for case $P_l = 0$; solid—the resulting trajectory from the neural controller; dashed—the off-line calculated trajectory

scheme and rough identification of the parameters involved help identify the sub-interval (or rectangle) that the parameters (or parameter vector) are within. The idea behind these schemes should be readily generalized to more general systems.

The linear controller around an equilibrium is designed so that it is sufficient to drive the system into a pre-designed neighborhood of the equilibrium by means of a time-optimal control. This may accomodate some slight differences between the actual switching and the optimal switching.

As a subsequent effort, more complex power systems will be studied along with the dynamics of connected loads in order to further investigate the mechanism of voltage collapse, which will be studied in chapter 6.

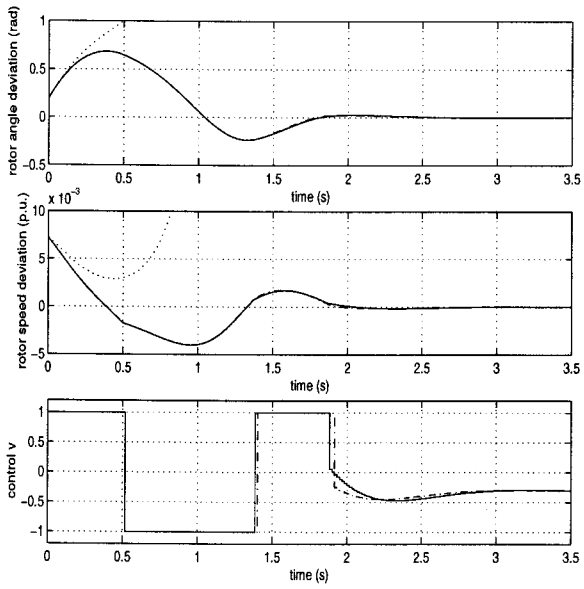


FIGURE 5.8: Performance of the neural controller for untrained case for case $P_l = 0$; solid—the resulting trajectory from the neural controller; dashed—the off-line calculated trajectory; dotted—the trajectory resulted from fixed compensation

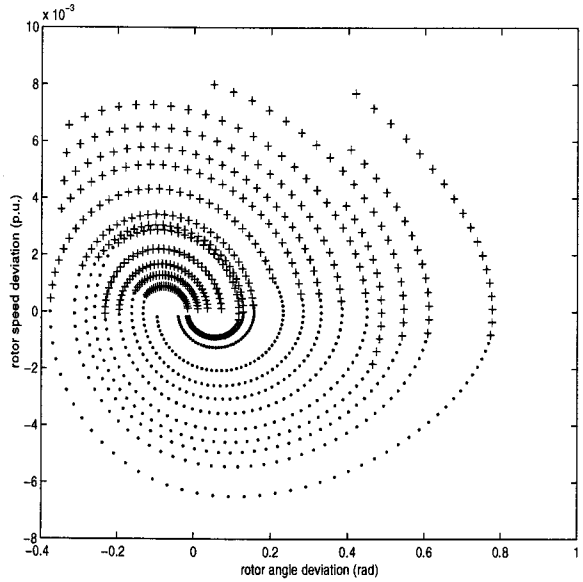


FIGURE 5.9: Time-optimal trajectories calculated by STVM for case $P_l = 10\%P_m$

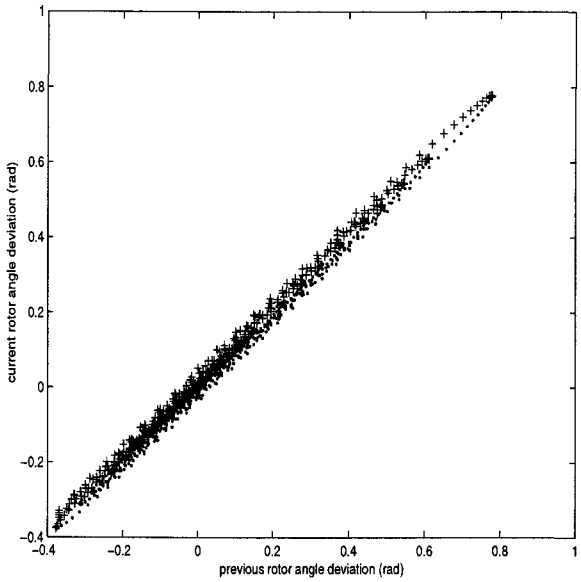


FIGURE 5.10: Rotor angle deviation data for neural net training for case $P_l = 10\%P_m$

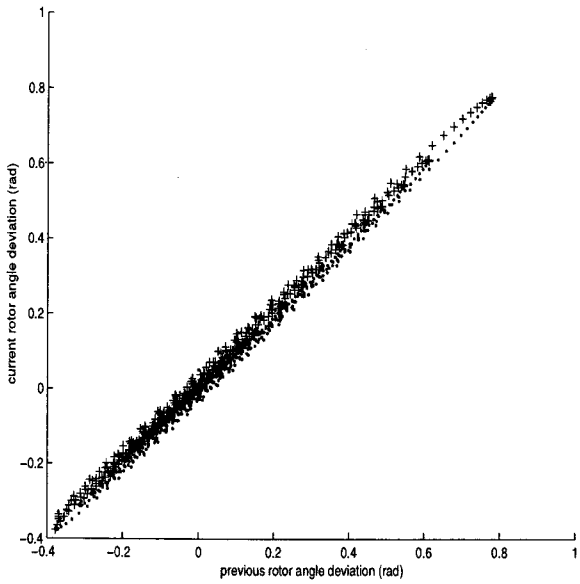


FIGURE 5.11: Learned pattern of rotor angle deviation for case $P_l = 10\%P_m$

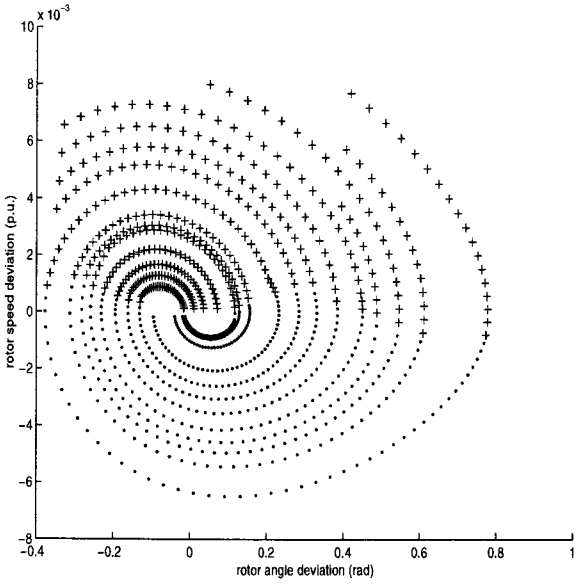


FIGURE 5.12: Learned pattern about time-optimal trajectories for case $P_l = 10\%P_m$

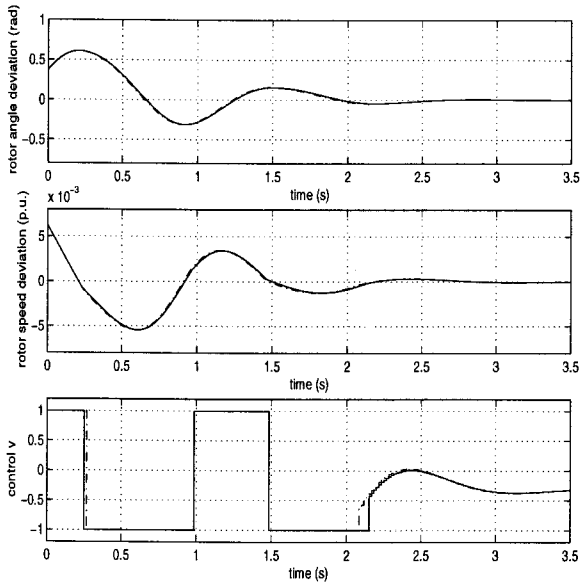


FIGURE 5.13: Training performance for case $P_l = 10\%P_m$; solid—the resulting trajectory from the neural controller; dashed—the off-line calculated trajectory

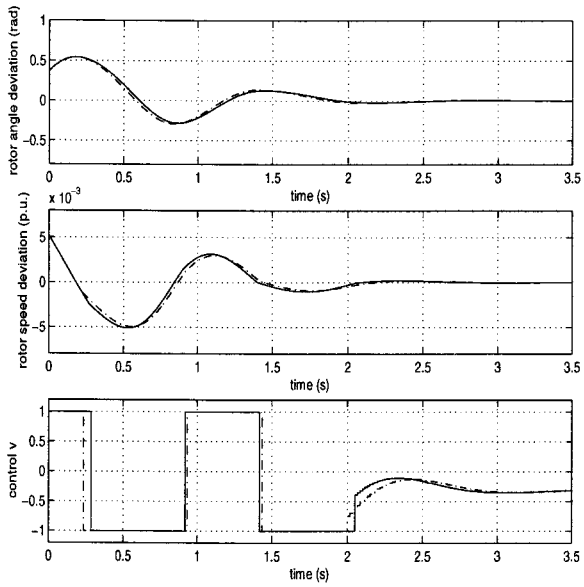


FIGURE 5.14: Performance of the neural controller for untrained case for case $P_l = 10\%P_m$; solid—the resulting trajectory from the neural controller; dashed—the off-line calculated trajectory

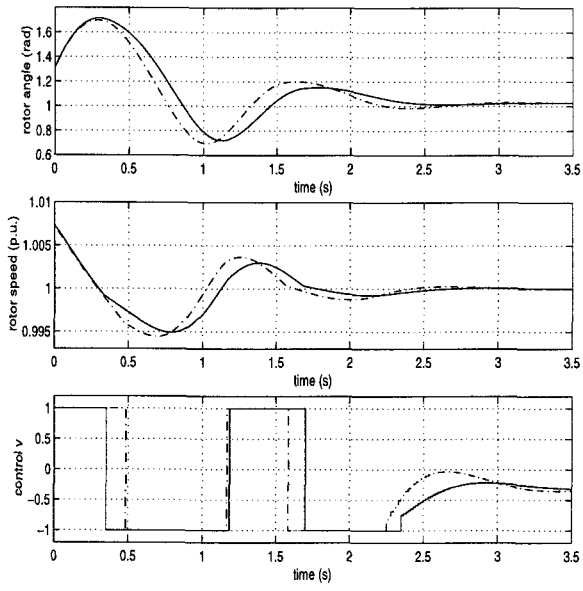


FIGURE 5.15: Performance of the hierarchical neural controller for SMIB with an unknown load after experiencing a short-circuit fault; solid—the resulting trajectory from the neural controller; dashed—the off-line calculated trajectory

6. NONLINEAR ADAPTIVE NEURAL CONTROL WITH APPLICATION TO PREVENTION OF VOLTAGE COLLAPSE

6.1. Introduction

Since the concept of a bilinear system [7] was proposed, bilinear systems have been studied extensively. The developed bilinear system theory has found many practical applications. It has been shown by both theoretical study and real applications ([7] and therein) that bilinear control or more generally multiplicative control can be more effective than linear control. One approach is to use some kind of adaptation law for adaptively changing the multipliers so that the resulting multiplicative control performs properly.

It is well known that bilinear systems comprise one of the simplest classes of nonlinear systems, and have appealing structural properties. Bilinear systems may typify adaptive or variable structure systems as well as general nonlinear systems.

Affine nonlinear systems, as a generalized version of bilinear systems, have also been investigated extensively. For a parameterized affine nonlinear system, a proper conventional controller may be designed with the help of the developed theory provided the parameters of the system to be studied are known. If the parameters are unknown to the controller designer or the parameters vary in some way, it is desired that the controller should be properly designed such that the whole system performs at least still the same in the presense of parameter disturbances. In this sense, the control may be termed as an adaptive control. Indeed, the neural-net controls, here, may be viewed as nonlinearly coupled bilinear systems.

An excellent survey paper [6] has overviewed the current status of adaptive control, and proposed a unified and generalized framework to address the adaptive stabilization

problems of nonlinear systems. It is observed that most available adaptive theoretical results are obtained without constraints on the control. The main reason is perhaps that imposing constraints on the control may lead to tremendous difficulties for mathematical treatments. And it is also noted that in adaptive stabilization problems of nonlinear systems the control corresponding to the true parameter, possessing some analytic form, is usually assumed available in some way. In some simple cases, it might be true. For more general cases, however, such a control is usually not available analytically. Instead, numerical techniques may have to be employed. The available calculated controls and temporal trajectories can be then used to train a neural network, which is well known to be capable of approximating properly both static nonlinear functions and dynamics of nonlinear systems. The concept of dynamic neural networks, accordingly the dynamic backpropagation algorithm for training such a neural network, was proposed in [31] as a natural extension of static neural networks in the context of dynamical system control. The combination of a static neural network with some feedback forms a dynamic neural network, which may involve online weights updating. Weights updating by means of the so-called dynamic backpropagation algorithm, however, requires intensive computation and a large amount of storage memory, which is not practical in the foreseeable future. Instead, simple adaptation laws for weights updating are sought. Extensive studies have been in progress in this respect. Still stability is an overwhelming issue. General results are very few, and most often depend on some fundamental assumptions such as the matching condition and the separation principle [6]. Some results have been obtained for affine nonlinear systems linear in parameters. For general dynamic nonlinear systems containing dynamic neural networks, general results are quite difficult to obtain. Rather, it is most often assumed that the nonlinear systems to be dealt with take specific forms and/or dynamic neural networks take specific forms. For example, with the assumption that the dynamic neural network that models a very special nonlinear system is a one hidden layer neural network without crosstalk feedback links among other assumptions, it is

demonstrated that the use of dynamic neural networks is efficient, as long as there are no constraints imposed on the control, for adaptive regulation of unknown dynamical nonlinear systems [96, 97]. Some other results are also available for dealing with the control of linear systems, nonlinear systems which can be feedback linearizable, and so on. Motivated by the fact that desirable control can be synthesized by training a neural network off-line with available optimal controls and optimal trajectories, our approach is then to synthesize a proper multiplicative control by off-line available nominal neural network controllers with updating the according multipliers on-line. This will become clear later. The systems to be studied are affine nonlinear systems which are linear in parameters. It is demonstrated in section 6.2. that several commonly-used power system models belong to the affine systems which are also affine in parameters. It is then natural to extend to these systems the hierarchical intelligent control schemes developed in chapter 5. Further, the conventional adaptive control combined with the hierarchical intelligent control schemes is studied in the following aspects: First of all, adaptive control is studied for time-invariant unknown parameters; for time-varying unknown parameters, adaptive controllers are synthesized; and the relevant system stability issues are studied. Finally, the simulations are provided for a typical model exhibiting voltage collapse phenomena to demonstrate the performance of adaptive hierarchical neural control.

6.2. Models and neural control of FACTS-equipped power systems

A few examples of power systems equipped with FACTS devices are presented in this section and their models are formulated/developed to illustrate the fact that many real systems may belong to a parameterized system affine in both control and parameters, for which the hierarchical neural-control scheme may be applied. Such a formulation also leads to the investigation of adaptive stabilization of affine systems which are also affine

in parameters via neural control. The adaptive stabilization of affine systems via neural control is presented in section 6.3..

6.2.1. Formulation of compound power systems

As is known, FACTS devices are popular for their rapid response, which should be properly manipulated. Models of power systems with FACTS devices are useful for control purpose. In what follows, a SMIB system with a load, developed in chapter 5, is described here again as one of illustrative examples; then a typical system model for voltage collapse study is presented to show that such a model is also an affine system; and further a four-machine sytem is presented and its model is developed with some reasonable assumptions, with which similar models can be formulated for general power systems equipped with FACTS devices.

6.2.1.1. Single-machine infinite-bus system with a load

As studied in chapter 5, one of the models for a SMIB system with a load is given by equation (5.10), and repeated in the following.

$$\begin{cases} \dot{\delta} = \omega_b \omega \\ \dot{\omega} = c_1 - c_{10} - (c_{20} + c_2)\omega - c_3 \sin(\delta_e + \delta) - c_4 \sin(\delta_e + \delta) v \end{cases} \tag{6.1}$$

Note that the parameters are c_{10} and c_{20}. It is evident that this system is affine in both control v and parameters.

6.2.1.2. A typical system model for voltage collapse study

The voltage collapse mechanism is not well understood yet. A simple system model in Figure 6.1, proposed in [98], was shown to display complex system behaviors typified for voltage collapse circumstances. Inclusion of this model is for designing a proper control

which is capable of preventing the occurrence of potential voltage collapse. Such a study is important in that it demonstrates how a specific potential voltage collapse problem could be solved by means of proper control, whose design is not clear until later sections. This study may also represent an important step towards a general approach to general voltage collapse problems.

The system model is given by

$$\begin{aligned}
\dot{\delta_m} &= \omega \\
M\dot{\omega} &= -d_m\omega + P_m + E_mY_mV\sin(\delta - \delta_m - \theta_m) + E_m^2Y_m\sin\theta_m \\
K_{q\omega}\dot{\delta} &= -K_{qv2}V^2 - K_{qv}V + Q(\delta, V) - Q_0 - Q_1 \\
TK_{q\omega}K_{pv}\dot{V} &= K_{p\omega}K_{qv2}V^2 + (K_{p\omega}K_{qv} - K_{q\omega}K_{pv})V \\
&\quad + K_{q\omega}(P(\delta, V) - P_0 - P_1) \\
&\quad - K_{p\omega}(Q(\delta, V) - Q_0 - Q_1) \qquad (6.2)
\end{aligned}$$

where $P(\delta,V) = -E_0'Y_0'V\sin(\delta+\theta_0) - E_mY_mV\sin(\delta-\delta_m+\theta_m) + (Y_0'\sin\theta_0' + Y_m\sin\theta_m)V^2$ and $Q(\delta,V) = E_0'Y_0'V\cos(\delta+\theta_0) + E_mY_mV\cos(\delta-\delta_m+\theta_m) - (Y_0'\cos\theta_0' + Y_m\cos\theta_m)V^2$, with $E_0' = \frac{E_0}{(1+C^2Y_0^{-2}-2CY_0^{-1}\cos\theta_0)^{1/2}}$, $Y_0' = Y_0(1 + C^2Y_0^{-2} - 2CY_0^{-1}\cos\theta_0)^{1/2}$, and $\theta_0' = \theta_0 + \tan^{-1}(\frac{CY_0^{-1}\sin\theta_0}{1-CY_0^{-1}\cos\theta_0})$.

Through some algebra, it can be readily shown that $Y_0'\sin\theta_0' = Y_0\sin\theta_0$, $Y_0'\cos\theta_0' = Y_0\cos\theta_0 - C$, and $E_0'Y_0' = E_0Y_0$.

Therefore, $P(\delta,V)$ and $Q(\delta,V)$ can rewritten as $P(\delta,V) = -E_0Y_0V\sin(\delta+\theta_0) - E_mY_mV\sin(\delta-\delta_m+\theta_m) + (Y_0\sin\theta_0 + Y_m\sin\theta_m)V^2$ and $Q(\delta,V) = E_0Y_0V\cos(\delta+\theta_0) + E_mY_mV\cos(\delta-\delta_m+\theta_m) - (Y_0\cos\theta_0 - C + Y_m\cos\theta_m)V^2$.

Define $a_1 = -\frac{d_m}{M}$, $a_2 = \frac{E_mY_m}{M}$, and $a_0 = \frac{P_m+E_m^2Y_m\sin\theta_m}{M}$;
$b_1 = -\frac{K_{qv2}+Y_0\cos\theta_0+Y_m\cos\theta_m}{K_{q\omega}}$, $b_2 = -\frac{K_{qv}}{K_{q\omega}}$, $b_3 = \frac{E_0Y_0}{K_{q\omega}}$, $b_4 = \frac{E_mY_m}{K_{q\omega}}$, $b_5 = \frac{1}{K_{q\omega}}$, $b_0 = -\frac{Q_0}{K_{q\omega}}$;
$c_1 = \frac{K_{p\omega}K_{qv2}+K_{q\omega}(Y_0\sin\theta_0+Y_m\sin\theta_m)+K_{p\omega}(Y_0\cos\theta_0+Y_m\cos\theta_m)}{TK_{q\omega}K_{pv}}$, $c_2 = \frac{K_{p\omega}K_{qv}-K_{q\omega}K_{pv}}{TK_{q\omega}K_{pv}}$,
$c_3 = -\frac{K_{q\omega}E_0Y_0}{TK_{q\omega}K_{pv}}$, $c_4 = -\frac{K_{q\omega}E_mY_m}{TK_{q\omega}K_{pv}}$, $c_5 = -\frac{K_{p\omega}E_0Y_0}{TK_{q\omega}K_{pv}}$, $c_6 = -\frac{K_{p\omega}E_mY_m}{TK_{q\omega}K_{pv}}$, $c_7 = -\frac{K_{p\omega}}{TK_{q\omega}K_{pv}}$,

$c_0 = -\frac{K_{q\omega}(P_0+P_1)-K_{p\omega}Q_0}{TK_{q\omega}K_{pv}}$; and $u = C$.

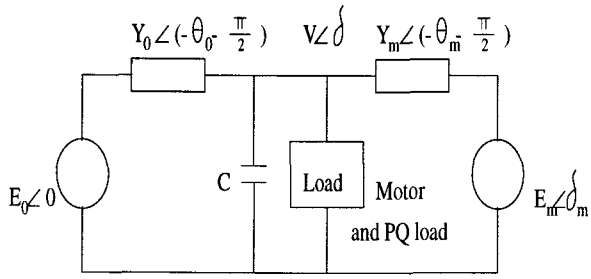


FIGURE 6.1: A power system for voltage collapse study

Then the system model can be rewritten in a simpler form as follows:

$$\begin{aligned}
\dot{\delta_m} &= \omega \\
\dot{\omega} &= a_1\omega + a_2 V\sin(\delta - \delta_m - \theta_m) + a_0 \\
\dot{\delta} &= b_1V^2 + [b_2 + b_3\cos(\delta+\theta_0) + b_4\cos(\delta - \delta_m + \theta_m)]V + b_5V^2u - b_5Q_1 + b_0 \\
\dot{V} &= c_1V^2 + [c_2 + c_3\sin(\delta+\theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\
&\quad + c_5\cos(\delta+\theta_0) + c_6\cos(\delta - \delta_m + \theta_m)]V + c_7V^2u - c_7Q_1 + c_0
\end{aligned} \tag{6.3}$$

Note that in the above equation, the control is C, which conceptually consists of two parts, i.e., C_n, the nominal value, which is relevant to the regulation of the voltage magnitude, and ΔC, the adjustable part, which is related to dynamic stability. In practice, C_n is usually implemented by traditional switching capacitors, and ΔC is implemented by a FACTS device, which can be operated rapidly for stability purposes. Note again that the resulting system is affine in control and also affine in parameters.

Let $C = C_n + \Delta C$. The above system can be rewritten as

$$\begin{aligned}
\dot{\delta_m} &= \omega \\
\dot{\omega} &= a_1\omega + a_2 V \sin(\delta - \delta_m - \theta_m) + a_0 \\
\dot{\delta} &= b_1' V^2 + [b_2 + b_3\cos(\delta + \theta_0) + b_4\cos(\delta - \delta_m + \theta_m)]V + b_5 V^2 u - b_5 Q_1 + b_0 \\
\dot{V} &= c_1' V^2 + [c_2 + c_3\sin(\delta + \theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\
&\quad + c_5\cos(\delta + \theta_0) + c_6\cos(\delta - \delta_m + \theta_m)]V + c_7 V^2 u - c_7 Q_1 + c_0
\end{aligned} \tag{6.4}$$

where $b_1' = b_1 + b_5 C_n$, $c_1' = c_1 + c_7 C_n$, and $u = \Delta C$.

6.2.1.3. Multi-machine power systems

Consider a four-machine power system model shown in Figure 6.2, which is also described in [11].

The network equation can be expressed as

$$\begin{bmatrix}
y_{17}^b & 0 & 0 & 0 & 0 & 0 & -y_{17}^b & 0 \\
0 & y_{25}^b & 0 & 0 & -y_{25}^b & 0 & 0 & 0 \\
0 & 0 & y_{38}^b & 0 & 0 & 0 & 0 & -y_{38}^b \\
0 & 0 & 0 & y_{46}^b & 0 & -y_{46}^b & 0 & 0 \\
0 & -y_{25}^b & 0 & 0 & y_{55} & -y_{56}^b & -y_{57}^b & -y_{58}^b \\
0 & 0 & 0 & -y_{46}^b & -y_{56}^b & y_{66} & 0 & -y_{68}^b \\
-y_{17}^b & 0 & 0 & 0 & -y_{57}^b & 0 & y_{77} & 0 \\
0 & 0 & -y_{83}^b & 0 & -y_{85}^b & -y_{86}^b & 0 & y_{88}
\end{bmatrix}
\begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \\ V_7 \\ V_8 \end{bmatrix}
=
\begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{6.5}$$

where $y_{55} = y_{25}^b + y_{56}^b + y_{57}^b + y_{58}^b + y_{55}^b$; $y_{66} = y_{46}^b + y_{56}^b + y_{68}^b + y_{66}^b$; $y_{77} = y_{17}^b + y_{57}^b + y_{77}^b$; $y_{88} = y_{83}^b + y_{85}^b + y_{86}^b + y_{88}^b$.

Note that if the transient reactances from the generators are also considered, the Nodes 1 to 4 have to be moved to between the generators and their corresponding transient reactances. The form of the network equation still holds except that y_{17}^b, y_{25}^b, y_{38}^b,

and y_{46}^b are formed as follow:

$y_{17}^b = \frac{1}{z_{17}^b + z_{g1}}$; $y_{25}^b = \frac{1}{z_{25}^b + z_{g2}}$; $y_{38}^b = \frac{1}{z_{38}^b + z_{g3}}$; $y_{46}^b = \frac{1}{z_{46}^b + z_{g4}}$;

where z_{17}^b, z_{25}^b, z_{38}^b, and z_{46}^b are the branch impedances; z_{g1},z_{g2},z_{g3},and z_{g4} are the corresponding generator transient reactances.

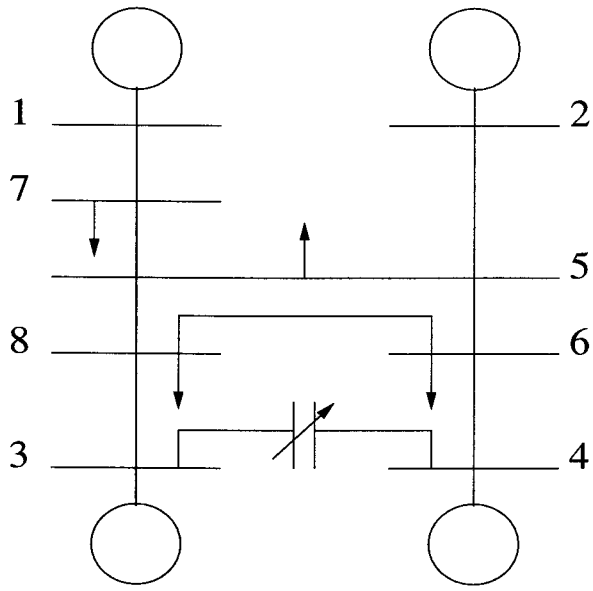


FIGURE 6.2: Four-machine power system

The network equation can then be written as

$$\begin{bmatrix} y_{17}^b & 0 & 0 & 0 & 0 & 0 & -y_{17}^b & 0 \\ 0 & y_{25}^b & 0 & 0 & -y_{25}^b & 0 & 0 & 0 \\ 0 & 0 & y_{38}^b & 0 & 0 & 0 & 0 & -y_{38}^b \\ 0 & 0 & 0 & y_{46}^b & 0 & -y_{46}^b & 0 & 0 \\ 0 & -y_{25}^b & 0 & 0 & y_{55} & -y_{56}^b & -y_{57}^b & -y_{58}^b \\ 0 & 0 & 0 & -y_{46}^b & -y_{56}^b & y_{66} & 0 & -y_{68}^b \\ -y_{17}^b & 0 & 0 & 0 & -y_{57}^b & 0 & y_{77} & 0 \\ 0 & 0 & -y_{83}^b & 0 & -y_{85}^b & -y_{86}^b & 0 & y_{88} \end{bmatrix} \begin{bmatrix} Eg_1 \\ Eg_2 \\ Eg_3 \\ Eg_4 \\ V_5 \\ V_6 \\ V_7 \\ V_8 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{6.6}$$

The admittance matrix can be partitioned as

$$Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \tag{6.7}$$

Elimination of V_7 and V_8 yields the reduced admittance matrix as

$$Yr = Y_{11} - Y_{12}Y_{22}^{-1}Y_{21} \tag{6.8}$$

where $Y_{12}Y_{22}^{-1}Y_{21} = \begin{bmatrix} \frac{y_{17}^2}{y_{77}} & 0 & 0 & 0 & \frac{y_{17}y_{57}}{y_{77}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{y_{38}^2}{y_{88}} & 0 & \frac{y_{38}y_{58}}{y_{88}} & \frac{y_{38}y_{68}}{y_{88}} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{y_{17}y_{57}}{y_{77}} & 0 & \frac{y_{38}y_{58}}{y_{88}} & 0 & \frac{y_{57}^2}{y_{77}} + \frac{y_{58}^2}{y_{88}} & \frac{y_{58}y_{68}}{y_{88}} \\ 0 & 0 & \frac{y_{38}y_{68}}{y_{88}} & 0 & \frac{y_{58}y_{68}}{y_{88}} & \frac{y_{68}^2}{y_{88}} \end{bmatrix}$.

Substitution of Y_{11} and $Y_{12}Y_{22}^{-1}Y_{21}$ yields the following:

$$Yr = \begin{bmatrix} y_{17}^b - \frac{y_{17}^2}{y_{77}} & 0 & 0 & 0 & -\frac{y_{17}y_{57}}{y_{77}} & 0 \\ 0 & y_{25}^b & 0 & 0 & -y_{25}^b & 0 \\ 0 & 0 & y_{38}^b - \frac{y_{38}^2}{y_{88}} & 0 & -\frac{y_{38}y_{58}}{y_{88}} & -\frac{y_{38}y_{68}}{y_{88}} \\ 0 & 0 & 0 & y_{46}^b & 0 & -y_{46}^b \\ -\frac{y_{17}y_{57}}{y_{77}} & -y_{25}^b & -\frac{y_{38}y_{58}}{y_{88}} & 0 & y_{55} - \frac{y_{57}^2}{y_{77}} - \frac{y_{58}^2}{y_{88}} & -y_{56}^b - \frac{y_{58}y_{68}}{y_{88}} \\ 0 & 0 & -\frac{y_{38}y_{68}}{y_{88}} & -y_{46}^b & -y_{56}^b - \frac{y_{58}y_{68}}{y_{88}} & y_{66} - \frac{y_{68}^2}{y_{88}} \end{bmatrix} \tag{6.9}$$

The reduced network equation can be written as

$$Yr \begin{bmatrix} Eg_1 \\ Eg_2 \\ Eg_3 \\ Eg_4 \\ V_5 \\ V_6 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ 0 \\ 0 \end{bmatrix} \tag{6.10}$$

The reduced admittance matrix Yr can be partitioned as

$$Yr = \begin{bmatrix} Yr_{11} & Yr_{12} \\ Yr_{21} & Yr_{22} \end{bmatrix} \tag{6.11}$$

Let $Vc = [V_5\ V_6]^\tau$ and $Eg = [Eg_1\ Eg_2\ Eg_3\ Eg_4]^\tau$. Then

$$\begin{cases} Yr_{11}E_g + Yr_{12}Vc = Ig \\ Yr_{21}E_g + Yr_{22}Vc = 0 \end{cases} \tag{6.12}$$

Note that the complete information about the TCSC is contained in Yr_{22}. And Yr_{22} can be rewritten as $Yr_{22} = Yr_{22}^0 + ju \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$.
Here u is the additional compensation around the fixed compensation, and Yr_{22}^0 is the admittance matrix related to the TCSC structure with fixed compensation.

Let $Ya = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$.

Then elimination of Vc yields the following:

$$\{Yr_{11} - Yr_{12}[Yr_{22}^0 + juYa]^{-1}Yr_{21}\}Eg = Ig \tag{6.13}$$

However, $[Yr_{22}^0 + juYa]^{-1}$ can be rewritten as

$$[Yr_{22}^0 + juYa]^{-1} = {Yr_{22}^0}^{-1} - juYr_{22}^{0\,-1}YaYr_{22}^{0\,-1} + O(u^2) \tag{6.14}$$

where $O(u^2)$ represents the second-order and higher-order terms of u.

If the system dynamics introduced by the nonlinear term $O(u^2)$ can be compensated for in some manner, the following analysis is simplified.

In such a case, it follows that

$$\{Yr_{11} - Yr_{12}\{{Yr_{22}^0}^{-1} - juYr_{22}^{0\,-1}YaYr_{22}^{0\,-1}\}Yr_{21}\}Eg = Ig \tag{6.15}$$

Define $Yrr = Yr_{11} - Yr_{12}{Yr_{22}^0}^{-1}Yr_{21}$ and $Yra = jYr_{12}Yr_{22}^{0\,-1}YaYr_{22}^{0\,-1}Yr_{21}$. Then

$$(Yrr + uYra)Eg = Ig \tag{6.16}$$

where Yrr represents the reduced admittance matrix with fixed compensation.

The electrical power drawn from each generator can be expressed as

$$\begin{aligned} Pe_i &= \mathbf{Re}\{Eg_i Ig_i^*\} = \mathbf{Re}\{\sum_j (Yrr^{i,j} + uYra^{i,j})^* Eg_j^* Eg_i\} \\ &= \sum_j |Eg_i||Eg_j||Yrr^{i,j}|\cos(\delta_{ij} - \theta_{ij}) + |Eg_i||Eg_j||Yra^{i,j}|u\cos(\delta_{ij} - \theta a_{ij}) \end{aligned} \tag{6.17}$$

where i ranges from 1 to the number of the generators. Note that the absolute value symbol will be dropped for brevity unless the confusion arises.

Given the mechanical power Pm_i for each generator, we have the following swing equation:

$$\begin{cases} \dot{\delta}_i = \omega_b(\omega_i - 1) \\ M_i\dot{\omega}_i = Pm_i - Pe_i - D_i(\omega_i - 1) \end{cases} \tag{6.18}$$

That is,

$$\begin{cases} \dot{\delta}_i = \omega_b(\omega_i - 1) \\ M_i\dot{\omega}_i = Pm_i - D_i(\omega_i - 1) - \sum_j Eg_iEg_jYrr^{i,j}\cos(\delta_{ij} - \theta_{ij}) + Eg_iEg_jYra^{i,j}u\cos(\delta_{ij} - \theta a_{ij}) \end{cases} \tag{6.19}$$

Translation of the equilibrium of the above system to the origin yields the following:

$$\begin{cases} \dot{\delta}_i = \omega_b\omega_i \\ \dot{\omega}_i = p_i - d_i\omega_i - \sum_j r_{ij}\cos(\delta_{ij} + \delta_{eij} - \theta_{ij}) + a_{ij}u\cos(\delta_{ij} + \delta_{eij} - \theta a_{ij}) \end{cases} \tag{6.20}$$

where $p_i = \frac{Pm_i}{M_i}$, $d_i = \frac{D_i}{M_i}$, $r_{ij} = \frac{Eg_iEg_jYrr^{i,j}}{M_i}$, $a_{ij} = \frac{Eg_iEg_jYra^{i,j}}{M_i}$, and $\delta_{eij} = \delta_{ei} - \delta_{ej}$.

The above equation can be rewritten in matrix form as

$$\dot{x} = Ax + B(F(x) + G(x)u) \tag{6.21}$$

wher $x = [\delta_1\ \delta_2\ \delta_3\ \delta_4\ \omega_1\ \omega_2\ \omega_3\ \omega_4]^\tau$; $A = \begin{bmatrix} 0_{4\times4} & I_{4\times4} \\ 0_{4\times4} & 0_{4\times4} \end{bmatrix}$; $B = \begin{bmatrix} 0_{4\times4} \\ I_{4\times4} \end{bmatrix}$;

$$F(x) = \begin{bmatrix} p_1 - d_1\omega_1 - \sum_{j=1}^4 r_{1j}\cos(\delta_{1j} + \delta_{e1j} - \theta_{1j}) \\ p_2 - d_2\omega_2 - \sum_{j=1}^4 r_{2j}\cos(\delta_{2j} + \delta_{e2j} - \theta_{2j}) \\ p_3 - d_3\omega_3 - \sum_{j=1}^4 r_{3j}\cos(\delta_{3j} + \delta_{e3j} - \theta_{3j}) \\ p_4 - d_4\omega_4 - \sum_{j=1}^4 r_{4j}\cos(\delta_{4j} + \delta_{e4j} - \theta_{4j}) \end{bmatrix}; \text{ and}$$

$$G(x) = \begin{bmatrix} -\sum_{j=1}^4 a_{1j}\cos(\delta_{1j} + \delta_{e1j} - \theta a_{1j}) \\ -\sum_{j=1}^4 a_{2j}\cos(\delta_{2j} + \delta_{e2j} - \theta a_{2j}) \\ -\sum_{j=1}^4 a_{3j}\cos(\delta_{3j} + \delta_{e3j} - \theta a_{3j}) \\ -\sum_{j=1}^4 a_{4j}\cos(\delta_{4j} + \delta_{e4j} - \theta a_{4j}) \end{bmatrix}.$$

Note that Equation (6.21) is affine in control u.

It should be noted that similar results can be obtained for general multi-machine systems equipped with FACTS devices.

6.2.1.4. Generalization: affine nonlinear systems

The previously presented systems belong to a class of affine nonlinear systems in general. A parameterized affine-in-control nonlinear system may be described as follows.

$$\dot{x} = f(x,p) + g(x,p)u \tag{6.22}$$

where $x \in R^n$ is the state; $p \in R^l$ is the parameter vector; and $u \in R^m$ is the control vector.

Notice that those example systems also belong to a class of affine-in-parameter nonlinear system, which may be described by

$$\dot{x} = \phi(x,u) + \psi(x,u)p \tag{6.23}$$

As an intersection of the class of systems (6.22) and the class of systems (6.23), a special class of parameterized nonlinear systems is given by

$$\dot{x} = \phi(x,u) + \psi(x)p \tag{6.24}$$

where $\phi(x,u)$ is affine in u.

The hierarchical neural control of systems (6.22) is discussed in section 6.2.2.. Through use of on-line hierarchical neural control (by which we mean relevant weights are updated in real time), the stabilization and adaptive control of systems (6.24) are presented in section 6.3..

6.2.2. Neural control of affine systems

Since systems (6.22) are affine in control, the so-called STVM—the computational algorithm developed in [8, 13], may be employed to calculate efficiently the time-optimal

trajectories and optimal controls. The techniques in addition to the hierarchical neural control structure developed in chapter 5, may be used to design an adaptive near time-optimal neural controller.

Of course, neural-network-based quadratic performance index optimal control may also be synthesized with the same techniques for near time-optimal control except for some minor modifications, which are clarified later.

First of all, quadratic-performance-index-based optimal trajectories and controls are derived in the following. The goal is to bring the state x very close to the equilibrium within a specified period of time $[t_0, T]$. Often the starting point t_0 is assumed to be 0 without loss of generality.

The quadratic performance index can be then expressed as

$$J(t_0) = \frac{1}{2}(x(T) - r(T))^T S(T)(x(T) - r(T)) + \frac{1}{2}\int_{t_0}^{T} (x^T Q x + (u - u_e)^T R(u - u_e))dt \quad (6.25)$$

where $S(T) \geq 0$, $Q \geq 0$, $R > 0$. The desired final state $r(T)$ is specified as the equilibrium x_e. u_e is the equilibrium control.

The Hamiltonian function can be defined as

$$H(x, u, t) = \frac{1}{2}(x^T Q x + (u - u_e)^T R(u - u_e)) + \lambda^T (f + gu) \quad (6.26)$$

The state equation is given by

$$\dot{x} = \frac{\partial H}{\partial \lambda} = f + gu \quad (6.27)$$

The costate equation can be given by

$$-\dot{\lambda} = \frac{\partial H}{\partial x} = \frac{\partial (f + gu)^T}{\partial x}\lambda + Qx \quad (6.28)$$

The stationarity equation gives

$$0 = \frac{\partial H}{\partial u} = R(u - u_e) + \frac{\partial (f + gu)^T}{\partial u}\lambda \quad (6.29)$$

Since $\frac{\partial(f(x,p)+g(x,p)u)}{\partial u} = g(x,p)$, which is independent of u, the optimal control u can be solved out as

$$u = -R^{-1}g(x,p)^{\tau}\lambda + u_e \tag{6.30}$$

Substitution of u into the state equation yields

$$\dot{x} = f(x,p) + g(x,p)(-R^{-1}g(x,p)^{\tau}\lambda + u_e) \tag{6.31}$$

Substitution of u into the costate equation yields

$$\dot{\lambda} = \frac{\partial[f(x,p) + g(x,p)(-R^{-1}g(x,p)^{\tau}\lambda + u_e]^{\tau}}{\partial x}\lambda + Qx \tag{6.32}$$

By choosing $R = I$, the control u can then be written as

$$u = -g(x,p)^{\tau}\lambda + u_e \tag{6.33}$$

Further, the boundary condition can be given by

$$\lambda(T) = S(T)(x(T) - r(T)) \tag{6.34}$$

Notice that for the Hamiltonian system which is composed of the state and costate equations, the initial condition for the state equation is given while for the costate equation there are constraints for the final costate value.

It is observed that the Hamiltonian system is a set of nonlinear ordinary differential equations in $x(t)$ and $\lambda(t)$ which develop forward and backward in time, respectively. Generally, it is not possible to obtain the analytic closed-form solution to such a two-point boundary-value problem (TPBVP). Numerical methods have to be employed to solve for the Hamiltonian system. One simple method, called shooting method [99] may be used. There are other methods like the "shooting to a fixed point" method, and relaxation method, etc.

The idea for the shooting method is as follows:

1. first make a guess for the initial values for the costate.

2. integrate the Hamiltonian system forward.
3. evaluate the mismatch on the final constaints.
4. find the sensitivity Jacobian for the final state and costate with respect to the initial costate value.
5. Using Newton-Raphson method to determine the change on the initial costate value.
6. repeat the loop steps 2 through 5 until the mismatch is close enough to zero if convergence indeed is assured.

For hierarchical neural control, the lower-level nominal neural controllers may be trained by using the computed optimal trajectories, and the upper-level multipliers may also be trained as for the neural-network-based time-optimal control case except that the multiplier processing unit in Figure 5.2 only performs the normalization of these multipliers.

6.3. Adaptive neural control design

The previous section presents development of models of FACTS-equipped power systems and control of affine-in-control nonlinear systems via hierarchical neural control. The parameters (i.e., weights and biases) relevant to neural networks therein are kept unchanged once the training is completed. In this section, we take a different approach to the same problem—adaptive stabilization of nonlinear systems. This approach is a combination of the conventional adaptive control design strategy [6] and the neural control design strategy developed in chapter 5. It is different from the neural control design strategy developed in chapter 5 since it allows for the on-line updating of weights and biases of applied neural networks as well as updating of estimates of the true parameters in the systems. For convenience of the subsequent derivations involved in the proposed

control design, some basic assumptions and useful lemmas are presented; then adaptive control of affine systems with unknown fixed parameters is discussed; further adaptive control of affine systems with time-varying parameters is discussed. It should be noted that some assumptions in the following (e.g., SBO and SCO) are made as usual in the literature (e.g., [6] and therein). However, there have been researches on stabilizability of affine nonlinear systems as well as power systems by employment of different assumptions (e.g., [100] and therein).

6.3.1. Definitions, assumptions and lemmas

Definition [5]: For any fixed $s \in [1, \infty)$, $f : R_+ \longrightarrow R$ is said to belong to L^s iff f is locally integrable and $||f||_s = (\int_0^\infty |f(t)|^s dt)^{1/s} < \infty$. When $s = \infty$, $f \in L^\infty$ iff $||f||_\infty = \sup_{t \leq 0} |f(t)| < \infty$.

Several basic assumptions [6] relevant to the adaptive stabilization problem of nonlinear systems are stated in the following, and will be assumed to hold throughout this chapter by default unless otherwise claimed.

Assumption of State Boundedness Observability (SBO): Let π be an open subset of R^l and Ω be an open neighborhood of $x_e \in R^n$. There exists a function: $h : \Omega \longrightarrow R_+$ of class C^2, such that there exist an open neighborhood Ω_0 of x_e in Ω and a strictly positive constant α_0 such that for all real numbers α, $0 < \alpha < \alpha_0$, all compact subsets $\mathcal{K}$ of π and all vectors $x_0 \in \Omega_0$, we can find a compact subset Γ of Ω such that, for any C^1 time functions $\hat{p} : R_+ \longrightarrow \pi$ and $u : R_+ \longrightarrow R^m$ and any solution $x(t)$ of $\dot{x}(t) = \phi(x, u) + \psi(x, u)p^*$, $x(0) = x_0 \in \Omega_0$ defined on $[0,\ T)$, we have the following implications:

$h(x) \leq \alpha$ and $\hat{p}(t) \in \mathcal{K}\ \forall t \in [0,\ T)$ imply $x(t) \in \Gamma\ \forall t \in [0,\ T)$.

The SBO assumption guarantees the boundedness of the state if an observation function is bounded.

Assumption of State Convergence Observability (SCO): For any bounded C^1 time function $\hat{p} : R_+ \longrightarrow \pi$ and $u : R_+ \longrightarrow R^m$ with $\dot{\hat{p}}$ also bounded and for any solution $x(t)$ of $\dot{x}(t) = \phi(x,u) + \psi(x,u)p^*$ defined on $[0,\ \infty)$, we have the following implication:

if $\lim_{t\to\infty} h(x) = 0$, and for $\forall t \in [0,\ \infty)$, $x(t) \in \Omega$ is bounded, then $\lim_{t\to\infty} x(t) = x_e$, where x_e is a desired equilibrium.

The SCO assumption guarantees the convergence of the state to a desired point if an observation function converges to zero in addition to the boundedness of the state.

To show that an observation function is bounded by zero, and the convergence of the state, in the context of adaptive stabilization of affine nonlinear systems, the following lemmas are considered for convenience.

Lemma 1 *(Barbalat's Lemma) [5]: If $f \in L^2 \cap L^\infty$, and $\dot{f}$ is bounded, then* $\lim_{t\to\infty} f(t) = 0$.

Lemma 2 *[6]: Let X be a C^1 time function defined on $[0,\ T)$ $(0 < T \leq \infty)$, satisfying*

$$\dot{X} \leq -cX + \sum_i \vartheta_i(t)X(t) + \sum_j \varpi_j(t) \tag{6.35}$$

where c is a strictly positive constant, $\sum_i$ and $\sum_j$ are finite sums and ϑ_i, and ϖ_j are positive time functions satisfying: $\int_0^T \vartheta_i^{\sigma_i} \leq S_{1i}$ and $\int_0^T \varpi_j^{\zeta_j} \leq S_{2j}$, where $\sigma_i \geq 1$ and $\zeta_j \geq 1$. Then $X(t)$ is bounded from above on $[0,\ T)$, and $X(t) \leq K_1 X(0) + K_2$ $\forall t \in [0,\ T)$, with K_1 and K_2 depending only on σ_i, ζ_j, S_{1i} and S_{2j}. Moreover, if T is infinite, then

$$\lim_{t\to\infty} \sup X(t) \leq 0 \tag{6.36}$$

6.3.2. Adaptive neural control for stabilization of nonlinear systems

Consider a parametrerized affine in control nonlinear system given by

$$\dot{x} = f(x,p) + g(x,p)u \tag{6.37}$$

where $x \in \mathcal{M} \subseteq R^n$ with $\mathcal{M}$ designating a manifold, $u \in U \subseteq R^m$ with U designating an admissible control set, and $p \in \pi \subseteq R^l$ with π designating a parameter set; $f : \mathcal{M} \times \pi \longrightarrow \mathcal{M}$ is a C^1 vector field; and $g : \mathcal{M} \times \pi \longrightarrow \mathcal{M}_{nm}$ is a C^1 vector field.

The goal is to design an "adaptive" controller u such that the system can be stabilized regardless of the parameter (p) disturbances.

As is illustrated in the previous sections, there are many practical systems which may be approximately affine in control systems which are also affine in parameters. Thus, the above system can be further written as

$$\begin{aligned} \dot{x} &= [f_0(x) + \sum_{i=1}^{P} f_p^i(x)p_i] + [g_0(x) + \sum_{i=1}^{P} g_p^i(x)p_i]u \\ &= [f_0(x) + g_0(x)u] + \sum_{i=1}^{P} [f_p^i(x) + g_p^i(x)u]p_i \end{aligned} \tag{6.38}$$

where $f(x,p) = f_0(x) + \sum_{i=1}^{P} f_p^i(x)p_i$, and $g(x,p) = g_0(x) + \sum_{i=1}^{P} g_p^i(x)p_i u$.

Define $\phi(x,u) = f_0(x) + g_0(x)u$ and $\psi(x,u)p = \sum_{i=1}^{P}[f_p^i(x) + g_p^i(x)u]p_i$. Then

$$\dot{x} = \phi(x,u) + \psi(x,u)p \tag{6.39}$$

Note that system (6.37) and system (6.39) will be used exchangeably for convenience if no confusion arises.

For the system parameterized with true parameters, we have

$$\dot{x} = \phi(x,u) + \psi(x,u)p^* \tag{6.40}$$

It is reasonable to assume that for the true parameter vector p^*, and for any initial state $x_0 \in \Omega$, there exists a proper state-based feedback control $u(x,p^*) \in U$ such that for any t, $x(t) \in \Omega$ and $\lim_{t\to\infty} x(t) = x_e$ where x_e is a desired equilibrium which may depend on p^*.

Since the true parameter vector p^* is usually not known, a control can only be synthesized based on an estimate $\hat{p}$ of p^* or available measurements, which is expected to

be able to stabilize the system regardless of the parameter disturbance. In this sense, the synthesized controller may be called "adaptive".

More formally, as formulated in [6], the adaptive stabilization problem may be formalized as follows:

Find an integer ν and two functions $\mu_1 : R^n \times R^\nu \longrightarrow R^\nu$ and $\mu_2 : R^n \times R^\nu \longrightarrow R^m$ such that there exists an open subset $D \subseteq R^n \times R^\nu$ with the following property:

The solution $(x(t), \chi(t))$ to the augmented system:

$$\begin{cases} \dot{x} = \phi(x,u) + \psi(x,u)p^* \\ \dot{\chi} = \mu_1(x,\chi) \end{cases} \tag{6.41}$$

and the state-feedback controller $u = \mu_2(x,\chi)$ with $(x(0), \chi(0)) \in D$ satisfies the following assumptions:

- As1: $(x(t), \chi(t))$ and u are well-defined, unique and bounded on $[0, \infty)$.
- As2: $\lim_{t\to\infty} x(t) = x_e$ where x_e is a desired equilibrium point which may depend on p^*.

Since the true parameter vector p^* is not available, an estimate $\hat{p}$ of p^* may be obtained and the control can be synthesized simultaneously based on the separation principle if indeed it holds. To be specific, use a parameter estimator to get an estimate $\hat{p}$ of p^*, and at the same time the control $u_n(x,\hat{p})$ instead of $u_n(x,p^*)$ is applied.

It is observed that this design scheme depends on the assumption that for the nominal control its explicit dependence on the state x and the parameter vector p is known. For the time being, it is assumed so; and later we will deal with the case where this explicit dependence is not known.

Consider system (6.41) but with the parameter vector time varying. In order to study the adaptive stabilization problem of this system, we make the following assumptions.

Assumption I: For any initial state $x_0 \in \Omega$, and for any $p \in \pi$, there exists a control $u(x,p)$ such that the system

$$\dot{x} = \phi(x, u(x,p)) + \psi(x, u(x,p))p \tag{6.42}$$

is stabilizable to a desirable equilibrium point.

Assumption II: For the same system as in Assumption I but with the parameter vector time-varying, for any initial state $x_0 \in \Omega$, for $p \in \pi$, and for all compact subsets $\mathcal{K} \in \pi$, there exists a convex tessellation of $\mathcal{K}$ such that for any any $p \in \mathcal{K}_i \subseteq \mathcal{K}$ ($\mathcal{K}_i$ is a tessellated sub-region of $\mathcal{K}$), the corresponding control $u(x,p)$ can be expressed as a linear combination of the nominal controls corresponding to the vertices of $\mathcal{K}_i$. That is,

$$u(x,p) = \sum_j \alpha^j(x) u(x,p_j) \tag{6.43}$$

where the multipliers $\alpha^j(x) : \mathcal{M}(R^n) \longrightarrow R_+$ is a non-negative C^1 function satisfying $\sum_j \alpha^j(x) = 1$, and $u(x,p_j)$ is the nominal control corresponding to the jth vertex of $\mathcal{K}_i$.

Note that in order to fulfill a proper control, the multipliers $\alpha^j(x)$'s must be identified and updated in a proper manner.

For a parameter vector in a small neighborhood of one of the vertices, the adaptive control $u(x,p^*)$ is used to stabilize the system

$$\dot{x} = \phi(x,u) + \psi(x,u)p \tag{6.44}$$

where p is time-varying and satisfies $|p-p^*| < \epsilon$ where ϵ is a pre-specified positive number.

The following conclusion can be obtained. (Since any convex region may be approximately represented by a union of many non-overlapping hyper-rectangles, it is hereafter assumed that the true parameter vector is within a hyper-rectangle unless otherwise claimed.)

Proposition 1 *System (6.44) is stabilized by the control $u(x,p^*)$ if the following assumptions, together with the assumpitons of SBO and SCO, are met. There exists a function $h : \mathcal{M}(R^n) \longrightarrow R_+$ such that*

- *As1:* $\frac{\partial h}{\partial x}[\phi(x,u(x,p^*)) + \psi(x,u(x,p^*))p] \leq -ch$ *where constant* $c > 0$.

Proof: According to assumption 1, we have $\dot{h} = \frac{\partial h}{\partial x}[\phi(x,u) + \psi(x,u)p] \leq -ch$. Application of Lemma 2 immediately yields the following:

$$\lim_{t\to\infty} \sup h \leq 0 \tag{6.45}$$

However, $h \geq 0$. Thus, $0 \leq \lim_{t\to\infty} h \leq \lim_{t\to\infty} \sup h \leq 0$. This implies that $\lim_{t\to\infty} h = 0$.

Note also that from $\dot{h} \leq -ch \leq 0$, it follows that for $\forall t \geq 0$, $h(t) \in L^\infty$. Then, use of the assumption SBO leads to the boundedness of $x(t)$.

The boundedness of $x(t)$, and $\lim_{t\to\infty} \hat{h} = 0$, together with the assumption SCO, leads to the convergence of the state, i.e., $\lim_{t\to\infty} x(t) = x_e$. This completes the proof.

Note that the condition $\frac{\partial h}{\partial x}[\phi(x,u(x,p^*)) + \psi(x,u(x,p^*))p^*] \leq -ch$, instead of the assumption 1 in the above Proposition is usually assumed in the context of adaptive control. However, for $p(t)$ in a small neighborhood of p^*, it is reasonable to assume that $\frac{\partial h}{\partial x}[\phi(x,u(x,p^*)) + \psi(x,u(x,p^*))p] \leq -ch$ may hold.

Conclusion 1 *System (6.44) with* $p(t)$ *fixed, is stabilized by the control* $u(x,p^*)$ *if the following assumptions, together with the assumpitons of SBO and SCO, are met. There exists a function* $h : \mathcal{M}(R^n) \longrightarrow R_+$ *such that*

- *As1:* $\frac{\partial h}{\partial x}[\phi(x,u(x,p^*)) + \psi(x,u(x,p^*))p^*] \leq -ch$ *where constant* $c > 0$.

Next, consider the more general adaptive stabilization problem described by equation (6.39) but with the assumption that $\psi(x,u)$ is independent of u. As a matter of fact, the previously described application systems all belong to this class. The goal is to estimate the parameters, and find a proper updating law for the multipliers so that the nonlinear system can be stabilized.

The system equation can be given by

$$\dot{x} = \phi(x,u) + \psi(x)p^* \tag{6.46}$$

For the true parameter vector p^*, the desired control u is $u(x,p^*)$, which can be expressed as $u(x,p^*) = \sum_j \alpha^{*j} u(x,p_j^*)$ with p^* is inside the region whose vertices are p_j^*'s. Therefore, the above system can be rewritten as

$$\dot{x} = \phi(x, \sum_j \alpha^{*j} u(x,p_j^*)) + \psi(x)p^* \tag{6.47}$$

Since p^* and α^* whose jth component is α^{*j} are not known, to fulfill a proper control u, an estimate $\hat{\alpha}$ of α^* has to be used. Since p^* is not available either, its estimation has to be made, too.

Proposition 2 *The adaptive stabilization of* $\dot{x} = \phi(x,u) + \psi(x)p^*$, *is generated by the control as* $u(x,p^*) = \sum_{j=1}^{J} \alpha^{*j} u(x,p_j^*)$, *if the following assumptions, together with assumptions of SBO and SCO, are met. There exists a function* $h : \mathcal{M}(R^n) \longrightarrow R_+$ *such that*

- *As1:* $\frac{\partial h}{\partial x}[\phi(x,\hat{u}(x,p^*)) + \psi(x)p^*] \leq -ch$ *where constant* $c > 0$, *and* $\hat{u}(x,p^*) = \sum_{j=1}^{J} \hat{\alpha}^j(x) u(x,p_j^*)$ *with* $\hat{\alpha}^j(x)$ *being the non-negative estimate of the true multiplier* $\alpha^{*j}(x)$ *and* $\sum_{j=1}^{J} \hat{\alpha}^j(x) = 1$.

Proof: $\dot{h} = \frac{\partial h}{\partial x}[\phi(x, \sum_j \alpha^{*j} u(x,p_j^*)) + \psi(x)p^*]$.

Define $\nu = \frac{\partial h}{\partial x}[\phi(x, \sum_j \hat{\alpha}^j u(x,p_j^*)) + \psi(x)\hat{p}] - \dot{h}$.

Let $\tilde{p} = \hat{p} - p^*$, and $\tilde{\alpha} = \hat{\alpha} - \alpha^*$. Since $\phi(x,u) = f_0(x) + g_0(x)u$, then we have the following:

$\nu = \frac{\partial h}{\partial x} g_0(x) \sum_j \tilde{\alpha}^j u(x,p_j^*) + \frac{\partial h}{\partial x}\psi(x)\tilde{p}$.

Using the error filtering technique, we have $\dot{e} + re = \nu$.

Define a Lyapunov-like function [6] $W = \frac{1}{2}e^\tau e + \frac{1}{2}\tilde{p}^\tau \tilde{p} + \frac{1}{2}\tilde{\alpha}^\tau \tilde{\alpha}$.

The derivative of W with respect to time t can be computed as follows:

$$\begin{aligned} \dot{W} &= e^\tau \dot{e} + \tilde{p}^\tau \dot{\tilde{p}} + \tilde{\alpha}^\tau \dot{\tilde{\alpha}} \\ &= e^\tau [-re + \frac{\partial h}{\partial x} g_0(x) \sum_j \tilde{\alpha}^j u(x, p_j^*) + \frac{\partial h}{\partial x} \psi(x) \tilde{p}] + \tilde{p}^\tau \dot{\hat{p}} + \tilde{\alpha}^\tau \dot{\hat{\alpha}} \end{aligned} \tag{6.48}$$

Let $e^\tau \frac{\partial h}{\partial x} g_0(x) \sum_j \tilde{\alpha}^j u(x, p_j^*) + \tilde{\alpha}^\tau \dot{\hat{\alpha}} = 0$ and $e^\tau \frac{\partial h}{\partial x} \psi(x) \tilde{p} + \tilde{p}^\tau \dot{\hat{p}} = 0$. We end up with the following results:

$$\begin{aligned} \dot{\hat{p}} &= -[e^\tau \frac{\partial h}{\partial x} \psi(x)]^\tau \\ \dot{\hat{\alpha}} &= -[e^\tau \frac{\partial h}{\partial x} g_0(x) \overline{u}(x, \{p_j^*\})]^\tau \end{aligned} \tag{6.49}$$

where $\overline{u}(x, \{p_j^*\}) = [u(x, p_1^*)\ u(x, p_2^*)\ \cdots\ u(x, p_J^*)]$ with J designating the number of the vertices of a tessellated sub-region $\mathcal{K}_i$.

and

$$\dot{W} = -r|e|^2 \tag{6.50}$$

Note that to confine the estimate of the parameter vector within a convex set, the updating law has to be modified. Note also that the defaulted convex set is a hyper-rectangle. Let this convex set be denoted by $\pi = \Pi_{i=1}^{l} \times [p_{min}^i,\ p_{max}^i]$. Then the modified updating law for $\hat{p}(t)$ can be given by

$$\dot{\hat{p}}^i = \begin{cases} -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \leq 0 \\ 0 & \text{if } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i > 0 \\ -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \geq 0 \\ 0 & \text{if } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i < 0 \end{cases} \tag{6.51}$$

where for $i = 1, \cdots, l$, $\hat{p}^i$ desigates the ith component of $\hat{p}$, and similarly $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$ the ith component of $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$.

Similarly, the modified updating law for $\hat{\alpha}(t)$ can be given by

$$\dot{\hat{\alpha}}^k = \begin{cases} -z^k & \text{if } 0 < \hat{\alpha}^k < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \\ & \text{or } \hat{\alpha}^k = 0 < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \text{ and } z^k \leq 0 \\ 0 & \text{if } \hat{\alpha}^k = 0 < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \text{ and } z^k > 0 \\ -z^k & \text{if } 0 < \hat{\alpha}^k < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \\ & \text{or } \hat{\alpha}^k = 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j > 0 \text{ and } z^k > 0 \\ 0 & \text{if } \hat{\alpha}^k = 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j > 0 \text{ and } z^k \leq 0 \\ 0 & \text{if } \hat{\alpha}^k = 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j = 0 \end{cases} \tag{6.52}$$

where $\hat{\alpha}^{-1} = \hat{\alpha}^0 \equiv 0$; and $z^k = \{[e^\tau \frac{\partial h}{\partial x} g_0(x) \overline{u}(x, \{p_j^*\})]^\tau\}^k$ designates the kth component of $\{[e^\tau \frac{\partial h}{\partial x} g_0(x) \overline{u}(x, \{p_j^*\})]^\tau\}$.

It can be shown in Appendix C that for $\forall t \geq 0$, $\hat{p}(t) \in \pi$, and $0 \leq \hat{\alpha}^k(t) \leq 1$ for $k = 1, 2, \cdots, J$ with $\sum_k \hat{\alpha}^k(t) = 1$, and that with the modified updating law, the quantity $\dot{W}$ can be made even more negative provided that $\hat{p}(0) \in \pi$, $p^* \in \pi$, and $0 \leq \hat{\alpha}^k(0) \leq 1$ for $k = 1, 2, \cdots, J$ with $\sum_k \hat{\alpha}^k(0) = 1$.

Therefore, $\dot{W} \leq -\gamma |e|^2$, hence $W \in L^\infty$, which in turn implies $e \in L^\infty$, $|\tilde{\alpha}| \in L^\infty$, and $|\tilde{p}| \in L^\infty$. Since $|e|^2 \leq -\frac{1}{r}\dot{W}$, i.e., $\int_0^\infty |e|^2 dt \leq \frac{1}{r}[W(0) - W(\infty)]$, then $e \in L^2$.

With assumption 1, $\frac{\partial h}{\partial x}[\phi(x, \sum_j \hat{\alpha}^j u(x, p_j^*)) + \psi(x) p^*] \leq -ch$ with $c \geq 0$. That is, $\dot{h} \leq -ch$. Application of Lemma 2 yields $\lim_{t \to \infty} \sup h = 0$.

However, $h \geq 0$. Thus, $0 \leq \lim_{t \to \infty} h \leq \lim_{t \to \infty} \sup h \leq 0$. This implies that $\lim_{t \to \infty} h = 0$.

Note also that from $\dot{h} \leq -ch \leq 0$, it follows that for $\forall t \geq 0$, $h(t) \in L^\infty$. Use of the assumption SBO leads to the boundedness of $x(t)$.

Since $e \in L^2 \cap L^\infty$, $\tilde{p} \in L^\infty$, $\tilde{\alpha} \in L^\infty$, and $\dot{e} + re = \nu = \frac{\partial h}{\partial x} \psi(x, u) \tilde{p} + \frac{\partial h}{\partial x} g_0(x) \sum_j \tilde{\alpha}^j u(x, p_j^*)$, then $\dot{e} \in L^\infty$. Through application of Barbalat's Lemma, $\lim_{t \to \infty} e(t) = 0$ immediately follow from the facts that $e \in L^2 \cap L^\infty$ and $\dot{e} \in L^\infty$. Therefore, the boundedness of all the quantities involved is assured.

The boundedness of $x(t)$, $\hat{\alpha}(t)$ and $\hat{p}(t)$, and $\lim_{t\to\infty}\dot{\hat{h}} = 0$, together with the assumption SCO, leads to the convergence of the state, i.e., $\lim_{t\to\infty} x(t) = x_e$. This completes the proof.

Note that the previous Proposition deals with the case where the multipliers are fixed. In the following, it will be considered that the multipliers are functions of the state, which consititute the upper-level controllers with respect to the nominal controllers in the context of hierarchical control.

Each multiplier is assumed to be a weighted sum of some known functions of state while the weights are unknown. Note that such a multiplier may be viewed as a one-hidden layer neural network.

Proposition 3 *The adaptive stabilization of* $\dot{x} = \phi(x,u) + \psi(x)p^*$, *is generated by the control* $u(x,p^*) = \sum_{j=1}^{J}\alpha^{*j}(x)u(x,p_j^*)$ *where* $\alpha^{*j}(x) = \sum_{n=0}^{N_j} w_{jn}^* s_n(x)$ *with* $s_n(x)$*'s being known functions, if the following assumptions, together with assumptions of SBO and SCO, are met. There exist a function* $h : \mathcal{M}(R^n) \longrightarrow R_+$ *such that*

- *As1:* $\frac{\partial h}{\partial x}[\phi(x,\hat{u}(x,p^*)) + \psi(x)p^*] \le -ch$ *where constant* $c > 0$,
  and $\hat{u}(x,p^*) = \sum_{j=1}^{J}\hat{\alpha}^j(x)u(x,p_j^*) = \sum_{j=1}^{J}\sum_{n=0}^{N_j}\hat{w}_{jn}s_n(x)u(x,p_j^*)$
  if each $\hat{w}_{jn}$, $w_{jn}^* \in [w_{jn,min},\ w_{jn,max}]$.

Proof: $\dot{h} = \frac{\partial h}{\partial x}[\phi(x,\sum_{j=1}^{J}\alpha^{*j}u(x,p_j^*)) + \psi(x)p^*]$.

Define $\nu = \frac{\partial h}{\partial x}[\phi(x,\sum_j \hat{\alpha}^j u(x,p_j^*)) + \psi(x)\hat{p}] - \dot{h}$.

Let $\tilde{p} = \hat{p} - p^*$, and $\tilde{\alpha} = \hat{\alpha} - \alpha^*$. Note here that $\hat{\alpha} = \sum_{n=0}^{N_j}\hat{w}_{jn}s_n(x)$. Then $\tilde{\alpha} = \sum_{n=0}^{N_j}\tilde{w}_{jn}s_n(x)$ where $\tilde{w}_{jn} = \hat{w}_{jn} - w_{jn}^*$.

Since $\phi(x,u) = f_0(x) + g_0(x)u$, then we have the following:

$\nu = \frac{\partial h}{\partial x}g_0(x)\sum_{j=1}^{J}\tilde{\alpha}^j u(x,p_j^*) + \frac{\partial h}{\partial x}\psi(x)\tilde{p} = \frac{\partial h}{\partial x}g_0(x)\sum_{j=1}^{J}\sum_{n=0}^{N_j}\hat{w}_{jn}s_n(x)u(x,p_j^*) + \frac{\partial h}{\partial x}\psi(x)\tilde{p}$.

Using the error filtering technique, we have $\dot{e} + re = \nu$.

Define a Lyapunov-like function $W = \frac{1}{2}e^\tau e + \frac{1}{2}\tilde{p}^\tau\tilde{p} + \frac{1}{2}\sum_{j=1}^{J}\sum_{n=0}^{N_j}\tilde{w}_{jn}^2$.

The derivative of W with respect to time t can be computed as follows:

$$\begin{aligned}\dot{W} &= e^\tau\dot{e} + \tilde{p}^\tau\dot{\tilde{p}} + \sum_{j=1}^{J}\sum_{n=0}^{N_j}\tilde{w}_{jn}\dot{\tilde{w}}_{jn} \\ &= e^\tau[-re + \frac{\partial h}{\partial x}g_0(x)\sum_{j=1}^{J}\sum_{n=0}^{N_j}\tilde{w}_{jn}s_n(x)u(x,p_j^*) + \frac{\partial h}{\partial x}\psi(x)\tilde{p}] + \tilde{p}^\tau\dot{\hat{p}} + \sum_{j=1}^{J}\sum_{n=0}^{N_j}\tilde{w}_{jn}\dot{\hat{w}}_{jn}\end{aligned}$$

Let $e^\tau\frac{\partial h}{\partial x}g_0(x)\sum_{j=1}^{J}\sum_{n=0}^{N_j}\tilde{w}_{jn}s_n(x)u(x,p_j^*) + \sum_{j=1}^{J}\sum_{n=0}^{N_j}\tilde{w}_{jn}\dot{\hat{w}}_{jn} = 0$ and $e^\tau\frac{\partial h}{\partial x}\psi(x)\tilde{p} + \tilde{p}^\tau\dot{\hat{p}} = 0$. We end up with the following results:

$$\begin{aligned}\dot{\hat{p}} &= -[e^\tau\frac{\partial h}{\partial x}\psi(x)]^\tau \\ \dot{\hat{w}}_{jn} &= -e^\tau\frac{\partial h}{\partial x}g_0(x)u(x,p_j^*)\end{aligned} \tag{6.53}$$

and

$$\dot{W} = -r|e|^2 \tag{6.54}$$

Note that to confine the estimate of the parameter vector within a convex set, the updating law has to be modified. Note also that the defaulted convex set is a hyper-rectangle. Let this convex set be denoted by $\pi = \Pi_{i=1}^{l} \times [p_{min}^i,\ p_{max}^i]$. Then the modified updating law for $\hat{p}(t)$ can be given by

$$\dot{\hat{p}}^i = \begin{cases} -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \le 0 \\ 0 & \text{if } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i > 0 \\ -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \ge 0 \\ 0 & \text{if } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i < 0 \end{cases} \tag{6.55}$$

where for $i = 1, \cdots, l$, $\hat{p}^i$ desigates the ith component of $\hat{p}$, and similarly $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$ the ith component of $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$.

Similarly, the modified updating law for $\hat{w}_{jn}$ can be given by

$$\dot{\hat{w}}_{jn} = \begin{cases} -e^T \frac{\partial h}{\partial x} g_0(x) s_n(x) u(x, p_j^*) & \text{if } \hat{w}_{jn} \in (w_{jn,min},\ w_{jn,max}) \\ & \text{or } \hat{w}_{jn} = w_{jn,min} \text{ and } e^T \frac{\partial h}{\partial x} g_0(x) s_n(x) u(x, p_j^*) \leq 0 \\ 0 & \text{if } \hat{w}_{jn} = w_{jn,min} \text{ and } e^T \frac{\partial h}{\partial x} g_0(x) s_n(x) u(x, p_j^*) > 0 \\ -e^T \frac{\partial h}{\partial x} g_0(x) s_n(x) u(x, p_j^*) & \text{if } \hat{w}_{jn} \in (w_{jn,min},\ w_{jn,max}) \\ & \text{or } \hat{w}_{jn} = w_{jn,max} \text{ and } e^T \frac{\partial h}{\partial x} g_0(x) s_n(x) u(x, p_j^*) \geq 0 \\ 0 & \text{if } \hat{w}_{jn} = w_{jn,max} \text{ and } e^T \frac{\partial h}{\partial x} g_0(x) s_n(x) u(x, p_j^*) < 0 \end{cases} \tag{6.56}$$

Following the same procedures as shown in Appendix C, it can be shown that for $\forall t \geq 0$, $\hat{p}(t) \in \pi$, and $w_{jn}(t) \in [w_{jn,min},\ w_{jn,max}]$ for $j = 1, 2, \cdots, J;\ \ n = 0, 1, \cdots, N_j$, and that with the modified updating law, the quantity $\dot{W}$ can be made even more negative provided that $\hat{p}(0) \in \pi$, $p^* \in \pi$, and $\hat{w}_{jn}(0) \in [w_{jn,min},\ w_{jn,max}]$, $w_{jn}^* \in [w_{jn,min},\ w_{jn,max}]$ for $j = 1, 2, \cdots, J;\ \ n = 0, 1, \cdots, N_j$.

Therefore, $\dot{W} \leq -\gamma |e|^2$, hence $W \in L^\infty$, which in turn implies $e \in L^\infty$, $|\tilde{w}_{jn}| \in L^\infty$, and $|\tilde{p}| \in L^\infty$. Since $|e|^2 \leq -\frac{1}{r}\dot{W}$, i.e., $\int_0^\infty |e|^2 dt \leq \frac{1}{r}[W(0) - W(\infty)]$, then $e \in L^2$.

With assumption 1, $\frac{\partial h}{\partial x}[\phi(x, \sum_j \hat{\alpha}^j u(x, p_j^*)) + \psi(x) p^*] \leq -ch$ with $c \geq 0$. That is, $\dot{h} \leq -ch$. Application of Lemma 2 yields $\lim_{t\to\infty} \sup h = 0$.

However, $h \geq 0$. Thus, $0 \leq \lim_{t\to\infty} h \leq \lim_{t\to\infty} \sup h \leq 0$. This implies that $\lim_{t\to\infty} h = 0$.

Note also that from $\dot{h} \leq -ch \leq 0$, it follows that for $\forall t \geq 0$, $h(t) \in L^\infty$. Use of the assumption SBO leads to the boundedness of $x(t)$.

Since $e \in L^2 \cap L^\infty$, $\tilde{p} \in L^\infty$, $\tilde{w}_{jn} \in L^\infty$, and $\dot{e} + re = \nu$ and $\nu = \frac{\partial h}{\partial x}\psi(x, u)\tilde{p} + \frac{\partial h}{\partial x} g_0(x) \sum_{j=1}^{J} \sum_{n=1}^{N_j} \tilde{w}_{jn} u(x, p_j^*)$, then $\dot{e} \in L^\infty$. Through application of Barbalat's Lemma, $\lim_{t\to\infty} e(t) = 0$ immediately follow from the facts that $e \in L^2 \cap L^\infty$ and $\dot{e} \in L^\infty$. Therefore, the boundedness of all the quantities involved is assured.

The boundedness of $x(t)$, $\hat{\alpha}(t)$ and $\hat{p}(t)$, and $\lim_{t\to\infty}\dot{\hat{h}} = 0$, together with the assumption SCO, leads to the convergence of the state, i.e., $\lim_{t\to\infty} x(t) = x_e$. This completes the proof.

The previous Proposition deals with the case where the unknown fixed parameter vector is within a hyper-rectangle. In the following, adaptive stabilization of nonlinear systems with time-varying parameters is considered.

Proposition 4 *The adaptive stabilization of* $\dot{x} = \phi(x,u) + \psi(x)p(t)$, *is generated by the control* $u(x,p) = \sum_{j=1}^{J} \alpha^{*j}(x)u(x,p_j^*)$, *if the following assumptions, together with assumptions of SBO and SCO, are met.*

- *As1:* $p^i(t) = p^{*i} + p_a^i \exp(-\beta^i t)$ *where* $i = 1,2,\cdots,l$, $p^* = [p^{*1}\ p^{*2}\ \cdots\ p^{*l}]^\tau$ *is a known constant vector, and* p_a^i*'s and* β^i*'s are unknown positive constants. But there exist known positive constants* $p^{0,i}$*'s and* $\beta^{0,i}$*'s such that* $p^{0,i} \le p_a^i$ *and* $\beta^{0,i} \ge \beta^i$;

- *As2:* $\alpha^*(x) = [\alpha^{*1}(x)\ \alpha^{*2}(x)\ \cdots\ \alpha^{*J}(x)]^\tau$ *corresponds to the system with fixed parameter* p^*, *whose variation rate with respect to time* t *is measureable;*

- *As3:* $\frac{\partial h}{\partial x}[\phi(x,\hat{u}(x,p^*)) + \psi(x)p^*] \le -ch$ *where constant* $c > 0$, *and* $\hat{u}(x,p^*) = \sum_{j=1}^{J} \hat{\alpha}^j(x)u(x,p_j^*)$ *with* $\hat{\alpha}^j(x)$ *being the non-negative estimate of the true multiplier* $\alpha^{*j}(x)$ *and* $\sum_{j=1}^{J} \hat{\alpha}^j(x) = 1$.

Proof: $\dot{h} = \frac{\partial h}{\partial x}[\phi(x,\sum_j \alpha^{*j}u(x,p_j^*)) + \psi(x)p(t)]$.

Define $\nu = \frac{\partial h}{\partial x}[\phi(x,\sum_j \hat{\alpha}^j u(x,p_j^*)) + \psi(x)\hat{p}] - \dot{h}$.

Let $\tilde{p} = \hat{p} - p$, and $\tilde{\alpha} = \hat{\alpha} - \alpha^*$. Since $\phi(x,u) = f_0(x) + g_0(x)u$, then we have the following:

$\nu = \frac{\partial h}{\partial x} g_0(x) \sum_j \tilde{\alpha}^j u(x,p_j^*) + \frac{\partial h}{\partial x}\psi(x)\tilde{p}$.

Using the error filtering technique, we have $\dot{e} + re = \nu$.

Define a Lyapunov-like function $W = \frac{1}{2}e^\tau e + \frac{1}{2}\tilde{p}^\tau\tilde{p} + \frac{1}{2}\tilde{\alpha}^\tau\tilde{\alpha}$.

The derivative of W with respect to time t can be computed as follows:

$$\begin{aligned}\dot{W} &= e^\tau \dot{e} + \tilde{p}^\tau \dot{\tilde{p}} + \tilde{\alpha}^\tau \dot{\tilde{\alpha}} \\ &= e^\tau[-re + \frac{\partial h}{\partial x} g_0(x) \sum_j \tilde{\alpha}^j u(x, p_j^*) + \frac{\partial h}{\partial x}\psi(x)\tilde{p}] + \tilde{p}^\tau(\dot{\hat{p}} - \dot{p}) + \tilde{\alpha}^\tau(\dot{\hat{\alpha}} - \dot{\alpha}^*)\end{aligned}$$

From $p^i(t) = p^{*i} + p_a^i \exp(-\beta^i t)$, we have $\dot{p}^i(t) = -\beta^i p_a^i \exp(-\beta^i t) = -\beta^i(p^i(t) - p^{*i})$. Thus, $\tilde{p}^\tau(\dot{\hat{p}} - \dot{p}) = \tilde{p}^\tau \dot{\hat{p}} - \sum_{i=1}^l \beta^i(\hat{p}^i - p^i(t))(p^i(t) - p^{*i})$. However, $-\sum_{i=1}^l \beta^i(\hat{p}^i - p^i(t))(p^i(t) - p^{*i}) = -\sum_{i=1}^l \beta^i|\hat{p}^i - p^i(t)|^2 + \beta^i(\hat{p}^i - p^i(t))(\hat{p}^i - p^{*i})$. Hence, $-\sum_{i=1}^l \beta^i(\hat{p}^i - p^i(t))(p^i(t) - p^{*i}) \leq \sum_{i=1}^l \beta^i(\hat{p}^i - p^i(t))(\hat{p}^i - p^{*i})$. Notice that if each $\hat{p}^i$ is updated so that it is confined between p^{*i} and $p^i(t)$, then $-\sum_{i=1}^l \beta^i(\hat{p}^i - p^i(t))(p^i(t) - p^{*i}) \leq 0$. But $p(t)$ is not known. Fortunately, $p^{0,i}$'s and $\beta^{0,i}$'s are known. Let $q^i(t) = p^{*i} + p^{0,i}\exp(-\beta^{0,i}t)$. Then $p^{*i} \leq q^i(t) \leq p^i(t)$. Note that $q(t) = [q^1(t)\ q^2(t)\ \cdots\ q^l(t)]^\tau$ and p^* are known, if each $\hat{p}^i$ is updated so that $p^{*i} \leq \hat{p}^i \leq q^i(t)$, then each $\hat{p}^i$ is confined between p^{*i} and $p^i(t)$. This can be done and is shown later.

Since each $p^{*i} \leq \hat{p}^i \leq p^i(t)$, $\tilde{p}^\tau(\dot{\hat{p}} - \dot{p}) \leq \tilde{p}^\tau \dot{\hat{p}}$. Thus,

$$\dot{W} \leq e^\tau[-re + \frac{\partial h}{\partial x} g_0(x) \sum_{j=1}^J \tilde{\alpha}^j u(x, p_j^*) + \frac{\partial h}{\partial x}\psi(x)\tilde{p}] + \tilde{p}^\tau \dot{\hat{p}} + \tilde{\alpha}^\tau(\dot{\hat{\alpha}} - \dot{\alpha}^*) \tag{6.57}$$

Let $e^\tau \frac{\partial h}{\partial x} g_0(x) \sum_j \tilde{\alpha}^j u(x, p_j^*) + \tilde{\alpha}^\tau(\dot{\hat{\alpha}} - \dot{\alpha}^*) = 0$ and $e^\tau \frac{\partial h}{\partial x}\psi(x)\tilde{p} + \tilde{p}^\tau \dot{\hat{p}} = 0$. We end up with the following results:

$$\begin{aligned}\dot{\hat{p}} &= -[e^\tau \frac{\partial h}{\partial x}\psi(x)]^\tau \\ \dot{\hat{\alpha}} &= -[e^\tau \frac{\partial h}{\partial x} g_0(x)\overline{u}(x, \{p_j^*\})]^\tau + \dot{\alpha}^*\end{aligned} \tag{6.58}$$

where $\overline{u}(x, \{p_j^*\}) = [u(x, p_1^*)\ u(x, p_2^*)\ \cdots\ u(x, p_J^*)]$.

and

$$\dot{W} \leq -r|e|^2 \tag{6.59}$$

It should be noted that with assumption 2, $\dot{\alpha}^*$ is measureable.

To confine the estimate of the parameter vector within a convex set, the updating law has to be modified. Let this convex set be denoted by $\pi = \Pi_{i=1}^{l} \times [p_{min}^i,\ p_{max}^i]$ with $p_{min}^i = p^{*i}$ and $p_{max}^i = q^i$. Then the modified updating law for $\hat{p}(t)$ can be given by

$$\dot{\hat{p}}^i = \begin{cases} -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \leq 0 \\ 0 & \text{if } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i > 0 \\ -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \geq 0 \\ 0 & \text{if } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i < 0 \end{cases} \tag{6.60}$$

where for $i = 1, \cdots, l$, $\hat{p}^i$ desigates the ith component of $\hat{p}$, and similarly $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$ the ith component of $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$.

Similarly, the modified updating law for $\hat{\alpha}(t)$ can be given by

$$\dot{\hat{\alpha}}^k = \begin{cases} -z^k & \text{if } 0 < \hat{\alpha}^k < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \\ & \text{or } \hat{\alpha}^k = 0 < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \text{ and } z^k \leq 0 \\ 0 & \text{if } \hat{\alpha}^k = 0 < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \text{ and } z^k > 0 \\ -z^k & \text{if } 0 < \hat{\alpha}^k < 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j \\ & \text{or } \hat{\alpha}^k = 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j > 0 \text{ and } z^k > 0 \\ 0 & \text{if } \hat{\alpha}^k = 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j > 0 \text{ and } z^k \leq 0 \\ 0 & \text{if } \hat{\alpha}^k = 1 - \sum_{j=0}^{k-1} \hat{\alpha}^j = 0 \end{cases} \tag{6.61}$$

where $z^k = \{[e^\tau \frac{\partial h}{\partial x} g_0(x)\overline{u}(x, \{p_j^*\})]^\tau + \dot{\alpha}^*\}^k$ designates the kth component of the vector $\{[e^\tau \frac{\partial h}{\partial x} g_0(x)\overline{u}(x, \{p_j^*\})]^\tau + \dot{\alpha}^*\}$.

Following the same procedures as shown in Appendix C, it can be shown that for $\forall t \geq 0$, $\hat{p}(t) \in \pi$, and $0 \leq \hat{\alpha}^k(t) \leq 1$ for $k = 1, 2, \cdots, J$ with $\sum_k \hat{\alpha}^k(t) = 1$, and that with the modified updating law, the quantity $\dot{W}$ can be made even more negative provided that $\hat{p}(0) \in \pi$, $p^* \in \pi$, and $0 \leq \hat{\alpha}^k(0) \leq 1$ for $k = 1, 2, \cdots, J$ with $\sum_k \hat{\alpha}^k(0) = 1$.

Therefore, $\dot{W} \leq -\gamma|e|^2$, hence $W \in L^\infty$, which in turn implies $e \in L^\infty$, $|\tilde{\alpha}| \in L^\infty$, and $|\tilde{p}| \in L^\infty$. Since $|e|^2 \leq -\frac{1}{r}\dot{W}$, i.e., $\int_0^\infty |e|^2 dt \leq \frac{1}{r}[W(0) - W(\infty)]$, then $e \in L^2$.

Note also that from assumption 3, i.e., $\dot{h} \leq -ch \leq 0$, it follows that for $\forall t \geq 0$, $h(t) \in L^\infty$. Use of the assumption SBO leads to the boundedness of $x(t)$.

Again from assumption 3, $\dot{h} \leq -ch \leq 0$. Application of Lemma 2 yields $\lim_{t\to\infty} \sup h = 0$.

However, $h \geq 0$. Thus, $0 \leq \lim_{t\to\infty} h \leq \lim_{t\to\infty} \sup h \leq 0$. This implies that $\lim_{t\to\infty} h = 0$.

Since $e \in L^2 \cap L^\infty$, $\tilde{p} \in L^\infty$, $\tilde{\alpha} \in L^\infty$, $\dot{e} + re = \nu$,

and $\nu = \frac{\partial h}{\partial x}\psi(x,u)\tilde{p} + \frac{\partial h}{\partial x}g_0(x)\sum_j \tilde{\alpha}^j u(x,p_j^*)$, then $\dot{e} \in L^\infty$. Through application of Barbalat's Lemma, $\lim_{t\to\infty} e(t) = 0$ immediately follow from the facts that $e \in L^2 \cap L^\infty$ and $\dot{e} \in L^\infty$. Therefore, the boundedness of all the quantities involved is assured.

The boundedness of $x(t)$, $\hat{\alpha}(t)$ and $\hat{p}(t)$, and $\lim_{t\to\infty} \hat{h} = 0$, together with the assumption SCO, leads to the convergence of the state, i.e., $\lim_{t\to\infty} x(t) = x_e$. This completes the proof.

In what follows, we consider adaptive stabilization of nonlinear systems with parameters time-varying within a hyper-rectangle through use of neural networks. Note that as mentioned before, the nominal controllers (i.e., the lower-level neural controllers in the context of hierarchical control) are trained neural controllers. The multipliers are actually the upper-level neural controllers for hierarchical control. These upper-level neural controllers are assumed to be independent of the variation of the parameter vector $p(t)$ as long as $p(t)$ is within some tessellated sub-region.

Proposition 5 *The adaptive stabilization of* $\dot{x} = \phi(x,u) + \psi(x)p(t)$, *is generated by the control* $u(x,p^*) = \sum_{j=1}^{J} \alpha^{*j}(x)u(x,p_j^*)$ *where each* $\alpha^{*j}(x)$ *is a mapping achieved by a one-hidden neural network, and can be expressed as* $\alpha^{*j}(x) = \sum_{n=0}^{N_j} w_{jn}^* s_n(x)$ *with* $s_n(x)$*'s being known sigmoidal functions, if the following assumptions, together with assumptions*

of SBO and SCO, are met.

- *As1:* $p(t) \in \pi$, *and its variation rate* $\dot{p}(t)$ *with respect to time* t *is measureable. Here* π *is a tessellated hyper-rectangle.*

- *As2: There exist a function* $h : \mathcal{M}(R^n) \longrightarrow R_+$ *such that* $\frac{\partial h}{\partial x}[\phi(x, \hat{u}(x, p^*)) + \psi(x)p^*] \leq -ch$ *where constant* $c > 0$, *and* $\hat{u}(x, p^*) = \sum_{j=1}^{J} \hat{\alpha}^j(x) u(x, p_j^*) = \sum_{j=1}^{J} \sum_{n=0}^{N_j} \hat{w}_{jn} s_n(x) u(x, p_j^*)$ *if each* $\hat{w}_{jn}$, $w_{jn}^* \in [w_{jn,min}, w_{jn,max}]$.

Proof: The proof can be given in the same way as for Proposition 3 except a minor modification for the updating law for $\hat{p}$, which is given in the following.

$$\dot{\hat{p}}^i = \begin{cases} -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i + \dot{p}^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i - \dot{p}^i \leq 0 \\ 0 & \text{if } \hat{p}^i = p_{min}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i - \dot{p}^i > 0 \\ -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i + \dot{p}^i & \text{if } \hat{p}^i \in (p_{min}^i,\ p_{max}^i) \\ & \text{or } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i - \dot{p}^i \geq 0 \\ 0 & \text{if } \hat{p}^i = p_{max}^i \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i - \dot{p}^i < 0 \end{cases} \tag{6.62}$$

where for $i = 1, \cdots, l$, $\hat{p}^i$ and $\dot{p}^i$ desigates the ith component of $\hat{p}$ and $\dot{p}(t)$, respectively, and similarly $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$ the ith component of $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$. Note that $\dot{p}(t)$ is assumed to be measureable.

When the parameter vector $p(t)$ is not confined to be within a specific sub-region (a hyper-rectangle), then perhaps information about which sub-region $p(t)$ is within is in need. If this can be done by a known indicator $I : \cup_i K_i \rightarrow Z_+$ where $\cup_i K_i$ is a tessellation of the parameter region of interest and Z_+ designates the set of non-negative integers, then corresponding lower-level nominal controllers are enabled, and all others are disabled. Once such indications are done, Proposition 5 applies.

6.4. Simulations

Consider the system model (6.4) for simulations. The relevant parameters are given next. According to [98], the load parameter values are given as $K_{p\omega} = 0.4$, $K_{pv} = 0.3$, $K_{q\omega} = -0.03$, $K_{qv} = -2.8$, $K_{qv2} = 2.1$, $T = 8.5$, $P_0 = 0.6$, $Q_0 = 1.3$, $P_1 = 0.0$, and the network and generator parameter values are given as $Y_0 = 20.0$, $\theta_0 = -5.0$, $E_0 = 1.0$, $Y_m = 5.0$, $\theta_m = -5.0$, $E_m = 1.0$, $P_m = 1.0$, $d_m = 0.05$, and $M = 0.3$. According to [101], when the parameter Q_1 varies slowly, this system exhibits a qualitative change in its behavior. For instance, when $C_n = 12$, the node-saddle bifurcation point is corresponding to $Q_1 = 11.41$, where equilibrium is lost. And further, $Q_1 = 10.98$ corresponds to a subcritical Hopf bifurcation point, and $Q_1 = 11.38$ corresponds to a supercritical Hopf bifurcation point. This indicates that for $Q_1 \in [10.98, 11.38]$, the equilibrium is oscillatorily unstable. Therefore, it is necessary to design a controller such that for any $Q_1 < 11.41$, the system can be stabilized. Note that application of the results developed in previous sections requires that the assumptions must be met, which is usually hard to check for practical problems. In the following, both conventional control design and adaptive neural control design are discussed; further simulations on the performance of the adaptive neural control for the voltage collapse problem are demonstrated.

6.4.1. Lyapunov-analysis-based control design

Note that there are some features for this model, which can be utilized to design a state-feedback nonlinear adaptive controller.

Define a function $v : \mathcal{M} \longrightarrow R$ where $\mathcal{M}$ is a state manifold. This function is given by $v = V^2 u - Q_1$. Since u, the feedback control, is a well-defined function from $\mathcal{M} \longrightarrow R$, then v is well-defined.

Define a Lyapunov function $W = \frac{1}{2}[r_m(\delta_m - \delta_{me})^2 + r_\omega(\omega - \omega_e)^2 + r_\delta(\delta - \delta_e)^2 + r_V(V - V_e)^2]$ where r_m, r_ω, r_δ and r_V are all positive numbers, and $(\delta_{me}, \omega_e, \delta_e, V_e)$ is a desired equilibrium, which is dependent on Q_1. At the equilibrium, the control is assumed to be u_e. Differentiating the Lyapunov function W along the system (6.4) yields the following:

$$\begin{aligned}
\dot{W} &= r_m(\delta_m - \delta_{me})\dot{\delta}_m + r_\omega(\omega - \omega_e)\dot{\omega} + r_\delta(\delta - \delta_e)\dot{\delta} + r_V(V - V_e)\dot{V} \\
&= r_m(\delta_m - \delta_{me})\omega + r_\omega(\omega - \omega_e)[a_1\omega + a_2 V \sin(\delta - \delta_m - \theta_m) + a_0] \\
&\quad + r_\delta(\delta - \delta_e)[b_1 V^2 + (b_2 + b_3\cos(\delta + \theta_0) + b_4\cos(\delta - \delta_m + \theta_m))V + b_5 v + b_0] \\
&\quad + r_V(V - V_e)[c_1 V^2 + (c_2 + c_3\sin(\delta + \theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\
&\quad + c_5\cos(\delta + \theta_0) + c_6\cos(\delta - \delta_m + \theta_m))V + c_7 v + c_0] \\
&= -[s_m(\delta_m - \delta_{me})^2 + s_\omega(\omega - \omega_e)^2 + s_\delta(\delta - \delta_e)^2 + s_V(V - V_e)^2] \\
&\quad + [r_\delta(\delta - \delta_e)b_5 + r_V(V - V_e)c_7]v \\
&\quad + r_m(\delta_m - \delta_{me})\omega + r_\omega(\omega - \omega_e)[a_1\omega + a_2 V \sin(\delta - \delta_m - \theta_m) + a_0] \\
&\quad + r_\delta(\delta - \delta_e)[b_1 V^2 + (b_2 + b_3\cos(\delta + \theta_0) + b_4\cos(\delta - \delta_m + \theta_m))V + b_0] \\
&\quad + r_V(V - V_e)[c_1 V^2 + (c_2 + c_3\sin(\delta + \theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\
&\quad + c_5\cos(\delta + \theta_0) + c_6\cos(\delta - \delta_m + \theta_m))V + c_0] \\
&\quad + [s_m(\delta_m - \delta_{me})^2 + s_\omega(\omega - \omega_e)^2 + s_\delta(\delta - \delta_e)^2 + s_V(V - V_e)^2] \qquad (6.63)
\end{aligned}$$

where s_m, s_ω, s_δ and s_V are all positive numbers.

Define a function $g(\delta_m, \omega, \delta, V) : \mathcal{M} \longrightarrow R$ as follows:

$$\begin{aligned}
g(\delta_m, \omega, \delta, V) &= r_m(\delta_m - \delta_{me})\omega + r_\omega(\omega - \omega_e)[a_1\omega + a_2 V \sin(\delta - \delta_m - \theta_m) + a_0] \\
&\quad + r_\delta(\delta - \delta_e)[b_1 V^2 + (b_2 + b_3\cos(\delta + \theta_0) + b_4\cos(\delta - \delta_m + \theta_m))V + b_0] \\
&\quad + r_V(V - V_e)[c_1 V^2 + (c_2 + c_3\sin(\delta + \theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\
&\quad + c_5\cos(\delta + \theta_0) + c_6\cos(\delta - \delta_m + \theta_m))V + c_0] \\
&\quad + [s_m(\delta_m - \delta_{me})^2 + s_\omega(\omega - \omega_e)^2 + s_\delta(\delta - \delta_e)^2 + s_V(V - V_e)^2] \qquad (6.64)
\end{aligned}$$

Then $\dot{W}$ can be rewritten as

$$\begin{aligned}\dot{W} &= -[s_m(\delta_m-\delta_{me})^2+s_\omega(\omega-\omega_e)^2+s_\delta(\delta-\delta_e)^2+s_V(V-V_e)^2] \\ &\quad +[r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7]v+g(\delta_m,\omega,\delta,V) \end{aligned} \tag{6.65}$$

Note that $g(\delta_m,\omega,\delta,V)$ is a well-defined continuous function.

Choose v as follows:

$$v=\begin{cases} -\frac{g(\delta_m,\omega,\delta,V)}{r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7} & \text{if } |r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7|\geq\epsilon_1 \\ -\frac{[r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7]g(\delta_m,\omega,\delta,V)}{[r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7]^2+\epsilon_2} & \text{if } |r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7|<\epsilon_1 \text{ and } g(\delta_m,\omega,\delta,V)\leq 0 \\ 0 & \text{if } |r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7|<\epsilon_1 \text{ and } g(\delta_m,\omega,\delta,V)>0 \end{cases} \tag{6.66}$$

where ϵ_1 and ϵ_2 are pre-specified positive numbers. It is pointed out that the above control may not perform robustly in practice.

Note that for $|r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7|\geq\epsilon_1$, $\dot{W}=-[s_m(\delta_m-\delta_{me})^2+s_\omega(\omega-\omega_e)^2+s_\delta(\delta-\delta_e)^2+s_V(V-V_e)^2]$; and for $|r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7|\geq\epsilon_1$ and $g(\delta_m,\omega,\delta,V)\leq 0$, it follows that

$$\begin{aligned}\dot{W} &= -[s_m(\delta_m-\delta_{me})^2+s_\omega(\omega-\omega_e)^2+s_\delta(\delta-\delta_e)^2+s_V(V-V_e)^2] \\ &\quad -\frac{[r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7]^2g(\delta_m,\omega,\delta,V)}{[r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7]^2+\epsilon_2}+g(\delta_m,\omega,\delta,V) \\ &= -[s_m(\delta_m-\delta_{me})^2+s_\omega(\omega-\omega_e)^2+s_\delta(\delta-\delta_e)^2+s_V(V-V_e)^2] \\ &\quad +\frac{\epsilon_2 g(\delta_m,\omega,\delta,V)}{[r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7]^2+\epsilon_2}\end{aligned}$$

It is clear that $\dot{W}$ is negative. However, when $g(\delta_m,\omega,\delta,V)>0$, v can only take a value which can make $[r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7]v$ as negative as possible. That is, $v=-v_{max}sgn(r_\delta(\delta-\delta_e)b_5+r_V(V-V_e)c_7)$ where $sgn(.)$ is defined by $sgn(x)=$

$$\begin{cases} 1 & \text{if } x>0 \\ -1 & \text{if } x<0 \\ 0 & \text{if } x=0 \end{cases}.$$

Note that even if v is such chosen, it is still possible that $\dot{W}$ may be positive. Therefore, for the worst case, such a synthesized controller can drive the system to this

switching surface $r_\delta(\delta - \delta_e)b_5 + r_V(V - V_e)c_7 = 0$; for the best case, such a synthesized controller may drive the system into a small neighborhood of a desired equilibrium.

6.4.2. Optimal control design

In this section, the goal is to bring the system from its transient period to the equilibrium through an optimal control within a specified period of time $[t_0, T]$. Often the starting point t_0 is 0, which corresponds to the begining of the postfault transient period. Note that for the region of interest, the system exhibits both stable and unstable modes. Therefore, when the system approaches an unstable equilibrium, some other kind of control must be designed to stabilize the equilibrium. In a way it may be said that the former case is mainly a transient stability problem whereas the latter case is mainly a steady-state stability problem.

Let the state be denoted by $x^\tau = [\delta_m \ \omega \ \delta \ V]$. Let the initial state be $x_0^\tau = [\delta_{m0} \ \omega_0 \ \delta_0 \ V_0]$, and the equilibrium $x_e^\tau = [\delta_{me} \ \omega_e \ \delta_e \ V_e]$.

The system can be rewritten as

$$\dot{x} = F(x, u) \tag{6.67}$$

where

$$F(x,u) = \begin{bmatrix} \omega \\ a_1\omega + a_2 V \sin(\delta - \delta_m - \theta_m) + a_0 \\ b_1' V^2 + [b_2 + b_3\cos(\delta + \theta_0) + b_4\cos(\delta - \delta_m + \theta_m)]V + b_5 V^2 u - b_5 Q_1 + b_0 \\ c_1' V^2 + [c_2 + c_3\sin(\delta + \theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\ +c_5\cos(\delta + \theta_0) + c_6\cos(\delta - \delta_m + \theta_m)]V + c_7 V^2 u - c_7 Q_1 + c_0 \end{bmatrix}.$$

Case I: Quadratic-performance-index-based optimal control

The quadratic performance index is defined by

$$J(t_0) = \frac{1}{2}(x(T) - r(T))^\tau S(T)(x(T) - r(T)) + \frac{1}{2}\int_{t_0}^{T} (x^\tau Q x + (u - u_e)^\tau R(u - u_e))dt \quad (6.68)$$

where $S(T) \geq 0$, $Q \geq 0$, $R > 0$. The desired final state $r(T)$ is specified as the equilibrium x_e. u_e is the control at the equilibrium, and equal to zero for this simulated system model.

The Hamiltonian function can be defined as

$$H(x, u, t) = \frac{1}{2}(x^\tau Q x + (u - u_e)^\tau R(u - u_e)) + \lambda^\tau F \quad (6.69)$$

The state equation is given by

$$\dot{x} = \frac{\partial H}{\partial \lambda} = F \quad (6.70)$$

The costate equation is given by

$$-\dot{\lambda} = \frac{\partial H}{\partial x} = \frac{\partial F^\tau}{\partial x}\lambda + Qx \quad (6.71)$$

where

$$\frac{\partial F}{\partial x}^\tau = \begin{bmatrix} 0 & -a_2 V\cos(\delta - \delta_m - \theta_m) & b_4 V\sin(\delta - \delta_m + \theta_m) & [-c_4\cos(\delta - \delta_m + \theta_m) + \\ & & & c_6\sin(\delta - \delta_m + \theta_m)]V \\ 1 & a_1 & 0 & 0 \\ 0 & a_2 V\cos(\delta - \delta_m - \theta_m) & [-b_3\sin(\delta + \theta_0) - b_4\sin(\delta - \delta_m + \theta_m)]V & [c_3\cos(\delta + \theta_0) + c_4\cos(\delta - \delta_m + \theta_m)]V \\ & & & -[c_5\sin(\delta + \theta_0) + c_6\sin(\delta - \delta_m + \theta_m)]V \\ 0 & a_2\sin(\delta - \delta_m - \theta_m) & 2b_1' V + b_2 + b_3\cos(\delta + \theta_0) & 2c_1' V + c_2 + c_3\sin(\delta + \theta_0) \\ & & +b_4\cos(\delta - \delta_m + \theta_m) + 2b_5 V u & +c_4\sin(\delta - \delta_m + \theta_m) \\ & & & +c_5\cos(\delta + \theta_0) \\ & & & +c_6\cos(\delta - \delta_m + \theta_m) + 2c_7 V u \end{bmatrix}$$

The stationarity equation gives

$$0 = \frac{\partial H}{\partial u} = R(u - u_e) + \frac{\partial F^\tau}{\partial u}\lambda \quad (6.72)$$

Since $\frac{\partial F}{\partial u} = [0\ 0\ b_5 V^2\ c_7 V^2]^\tau$, is independent of u, the optimal control u can be solved out as

$$u = -R^{-1}\frac{\partial F^\tau}{\partial u}\lambda + u_e \quad (6.73)$$

Substitution of u into the state equation yields

$$\dot{x} = f(x, -R^{-1}\frac{\partial F}{\partial u}^{\tau}\lambda + u_e) \tag{6.74}$$

Substitution of u into the costate equation yields

$$\dot{\lambda} = \frac{\partial F(x, -R^{-1}\frac{\partial F}{\partial u}^{\tau}\lambda + u_e)}{\partial x}^{\tau}\lambda + Qx \tag{6.75}$$

By choosing $R = I$, the control u can then be written as

$$u = b_5 V^2 \lambda_3 + c_7 V^2 \lambda_4 + u_e \tag{6.76}$$

From the general boundary condition $(\phi_x + \psi_x^{\tau}\mu - \lambda)^{\tau}|_T dx(T) + (\phi_t + \psi_t^{\tau}\mu + H)|_T dT = 0$ with $\phi(x(T), T)$ being part of the performance index, and $\psi(x(T), T)$ the final state constraint, the boundary condition for the problem under study can be obtained $\lambda(T) = S(T)[x(T) - r(T)] = S(T)[x(T) - x_e]$.

Notice that for the Hamiltonian system, the initial condition for the state equation is given while for the costate equation there are constraints for the final costate value. The analytic solution to the two-point boundary-value (TPBV) problem is not available in general, and the numerical solution may be obtained through numerical techniques.

Case II: Time-optimal control

Translating the system equilibrium from x_e to the origin yields the following equation.

$$\dot{x} = F(x + x_e, u) = F_1(x, u) \tag{6.77}$$

where

$$F_1(x,u) = \begin{bmatrix} \omega \\ a_1\omega + a_2(V+V_e)\sin(\delta - \delta_m - \theta_m^{'}) + a_0 \\ b_1^{'}(V+V_e)^2 + [b_2 + b_3\cos(\delta + \theta_0^{'}) + b_4\cos(\delta - \delta_m + \theta_m^{''})](V+V_e) + b_5(V+V_e)^2 u - b_5 Q_1 + b_0 \\ c_1^{'}(V+V_e)^2 + [c_2 + c_3\sin(\delta + \theta_0^{'}) + c_4\sin(\delta - \delta_m + \theta_m^{''}) \\ +c_5\cos(\delta + \theta_0^{'}) + c_6\cos(\delta - \delta_m + \theta_m^{''})](V+V_e) + c_7(V+V_e)^2 u - c_7 Q_1 + c_0 \end{bmatrix},$$

$\theta_m^{'} = \theta_m - \delta_e + \delta_{me}$, $\theta_0^{'} = \theta_0 + \delta_e$, and $\theta_m^{''} = \theta_m + \delta_e - \delta_{me}$.

Define the performance index as $J = \int_{t_0}^{t_f} dt + \frac{\rho_1}{2}\delta_{mf}^2 + \frac{\rho_2}{2}\omega_f^2 + \frac{\rho_3}{2}\delta_f^2 + \frac{\rho_4}{2}V_f^2$, where the subscript $_f$ designates for the value at the final terminal time t_f. By omitting the constant terms related to the initial conditions, the performance index can be expressed as $J = \int_{t_0}^{t_f}(1 + \rho_1\delta_m\dot{\delta}_m + \rho_2\omega\dot{\omega} + \delta\dot{\delta} + \rho_4 V\dot{V})dt$.

Define $\dot{x_0}$ in the following:

$$
\begin{aligned}
\dot{x_0} \;=\; & 1 + \rho_1\delta_m\omega + \rho_2\omega[a_1\omega + a_2(V+V_e)\sin(\delta - \delta_m - \theta_m^{'}) + a_0] \\
& +\rho_3\delta[b_1^{'}(V+V_e)^2 + (b_2 + b_3\cos(\delta + \theta_0^{'}) + b_4\cos(\delta - \delta_m + \theta_m^{''}))(V+V_e) + b_5(V+V_e)^2u - b_5Q_1 + b_0] \\
& +\rho_4 V[c_1^{'}(V+V_e)^2 + (c_2 + c_3\sin(\delta + \theta_0^{'}) + c_4\sin(\delta - \delta_m + \theta_m^{''}) \\
& +c_5\cos(\delta + \theta_0^{'}) + c_6\cos(\delta - \delta_m + \theta_m^{''}))(V+V_e) + c_7(V+V_e)^2u - c_7Q_1 + c_0]
\end{aligned}
$$

Define the augmented state $x^{'} = [x^{\tau}\ x_0]^{\tau}$. The augmented system is then the following:

$$\dot{x}^{'} = F^{'}(x^{'}) + G(x^{'})u \tag{6.78}$$

where

$$
F^{'}(x^{'}) = \begin{bmatrix}
\omega \\
a_1\omega + a_2(V+V_e)\sin(\delta - \delta_m - \theta_m^{'}) + a_0 \\
b_1^{'}(V+V_e)^2 + [b_2 + b_3\cos(\delta + \theta_0^{'}) + b_4\cos(\delta - \delta_m + \theta_m^{''})](V+V_e) - b_5Q_1 + b_0 \\
c_1^{'}(V+V_e)^2 + [c_2 + c_3\sin(\delta + \theta_0^{'}) + c_4\sin(\delta - \delta_m + \theta_m^{''}) \\
+c_5\cos(\delta + \theta_0^{'}) + c_6\cos(\delta - \delta_m + \theta_m^{''})](V+V_e) - c_7Q_1 + c_0 \\
1 + \rho_1\delta_m\omega + \rho_2\omega[a_1\omega + a_2(V+V_e)\sin(\delta - \delta_m - \theta_m^{'}) + a_0] \\
+\rho_3\delta[b_1^{'}(V+V_e)^2 + (b_2 + b_3\cos(\delta + \theta_0^{'}) + b_4\cos(\delta - \delta_m + \theta_m^{''}))(V+V_e) - b_5Q_1 + b_0] \\
+\rho_4 V[c_1^{'}(V+V_e)^2 + (c_2 + c_3\sin(\delta + \theta_0^{'}) + c_4\sin(\delta - \delta_m + \theta_m^{''}) \\
+c_5\cos(\delta + \theta_0^{'}) + c_6\cos(\delta - \delta_m + \theta_m^{''}))(V+V_e) - c_7Q_1 + c_0]
\end{bmatrix},
$$

$$
G^{'}(x^{'}) = \begin{bmatrix}
0 \\
0 \\
b_5(V+V_e)^2 \\
c_7(V+V_e)^2 \\
b_5(V+V_e)^2\rho_3\delta + c_7(V+V_e)^2\rho_4 V
\end{bmatrix},
$$

and $x^{'}(t_0) = [x_0\ 0]^{\tau}$.

The adjoint state equation can be written as

$$\dot{\lambda} = -\frac{\partial}{\partial x^{'}}[F^{'}(x^{'}) + G^{'}(x^{'})u]^{\tau}\lambda \tag{6.79}$$

Define $D = \frac{\partial}{\partial x'}[F'(x') + G'(x')u]^\tau$. $\frac{\partial}{\partial x'}F'(x')$ and $\frac{\partial}{\partial x'}G'(x')$ can be computed in the following.

$$\frac{\partial}{\partial x'}F'(x')^\tau = \left[\begin{array}{ccccc} \frac{\partial}{\partial \delta_m}F' & \frac{\partial}{\partial \omega}F' & \frac{\partial}{\partial \delta}F' & \frac{\partial}{\partial V}F' & \frac{\partial}{\partial x_0}F' \end{array} \right]^\tau \text{ with}$$

$$\frac{\partial}{\partial \delta_m}F' = \left[\begin{array}{l} 0 \\ -a_2(V+V_e)\cos(\delta-\delta_m-\theta'_m) \\ b_4\sin(\delta-\delta_m+\theta''_m)(V+V_e) \\ (V+V_e)[-c_4\cos(\delta-\delta_m+\theta''_m)+c_6\sin(\delta-\delta_m+\theta''_m)] \\ \rho_1\omega-\rho_2a_2\omega(V+V_e)\cos(\delta-\delta_m-\theta'_m)+\rho_3\delta[b_4(V+V_e)\sin(\delta-\delta_m+\theta''_m)] \\ +\rho_4V[-c_4\cos(\delta-\delta_m+\theta''_m)+c_6\sin(\delta-\delta_m+\theta''_m)](V+V_e) \end{array} \right];$$

$$\frac{\partial}{\partial \omega}F' = \left[\begin{array}{l} 1 \\ a_1 \\ 0 \\ 0 \\ \rho_1\delta_m+\rho_2\omega[2a_1+a_2(V+V_e)\sin(\delta-\delta_m-\theta'_m)+a_0] \end{array} \right];$$

$$\frac{\partial}{\partial \delta}F' = \left[\begin{array}{l} 0 \\ a_2(V+V_e)\cos(\delta-\delta_m-\theta'_m) \\ (V+V_e)[-b_3\sin(\delta+\theta'_0)+b_4\sin(\delta-\delta_m+\theta''_m)] \\ (V+V_e)[c_3\cos(\delta+\theta'_0)+c_4\cos(\delta-\delta_m+\theta''_m)-c_5\sin(\delta+\theta'_0)-c_6\sin(\delta-\delta_m+\theta''_m)] \\ \rho_2a_2\omega(V+V_e)\cos(\delta-\delta_m-\theta'_m)+\rho_3[b'_1(V+V_e)^2+(b_2+b_3\cos(\delta+\theta'_0) \\ +b_4\cos(\delta-\delta_m+\theta''_m))(V+V_e)+b_5(V+V_e)^2u-b_5Q_1+b_0] \\ +\rho_3\delta(-b_3\sin(\delta+\theta'_0)-b_4\sin(\delta-\delta_m+\theta''_m))(V+V_e)+\rho_4V[c_3\cos(\delta+\theta'_0) \\ +c_4\cos(\delta-\delta_m+\theta''_m)-c_5\sin(\delta+\theta'_0)-c_6\sin(\delta-\delta_m+\theta''_m)](V+V_e) \end{array} \right];$$

$$\frac{\partial}{\partial V}F' = \left[\begin{array}{l} 0 \\ a_2\sin(\delta-\delta_m-\theta'_m) \\ 2b'_1(V+V_e)+b_2+b_3\cos(\delta+\theta'_0)+b_4\cos(\delta-\delta_m+\theta''_m) \\ 2c'_1(V+V_e)+c_2+c_3\sin(\delta+\theta'_0)+c_4\sin(\delta-\delta_m+\theta''_m) \\ +c_5\cos(\delta+\theta'_0)+c_6\cos(\delta-\delta_m+\theta''_m) \\ \rho_2a_2\omega\sin(\delta-\delta_m-\theta'_m)+\rho_3\delta[2b'_1(V+V_e)+b_2+b_3\cos(\delta+\theta'_0) \\ +b_4\cos(\delta-\delta_m+\theta''_m)]+\rho_4[c'_1(V+V_e)^2+(c_2+c_3\sin(\delta+\theta'_0) \\ +c_4\sin(\delta-\delta_m+\theta''_m)+c_5\cos(\delta+\theta'_0)+c_6\cos(\delta-\delta_m+\theta''_m))(V+V_e)-c_7Q_1+c_0] \\ +\rho_4V[2c'_1(V+V_e)+c_2+c_3\sin(\delta+\theta'_0)+c_4\sin(\delta-\delta_m+\theta''_m) \\ +c_5\cos(\delta+\theta'_0)+c_6\cos(\delta-\delta_m+\theta''_m)] \end{array} \right];$$

$$\frac{\partial}{\partial x_0}F' = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

$$\frac{\partial}{\partial x'}G'(x')^\tau = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & b_5(V+V_e)^2\rho_3 \\ 0 & 0 & 2b_5(V+V_e) & 2c_7(V+V_e) & (V+V_e)[2b_5\delta\rho_3 + 2c_7V\rho_4 + c_7(V+V_e)\rho_4] \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Therefore, the adjoint equation can be written as

$$\dot{\lambda} = -D\lambda \tag{6.80}$$

and the terminal condition is $\lambda(t_f) = \frac{\partial J}{\partial x'}|_{t_f} = [\rho_1\delta_{mf}\ \rho_2\omega_f\ \rho_3\delta_f\ \rho_4 V_f\ 1]^\tau$.

Suppose the number of the optimal switching times is N (including the variable terminal time). Let the switching vector be $\underline{\tau} = [\tau_1\ \cdots\ \tau_{N-1}\ t_f]^\tau$. Then the switching function can be given as follows

$\phi(\tau_j) = 2\lambda(t)^\tau G'(x'(\tau_j))$ for $j = 1, 2, \cdots, N-1$; and $\phi^N = -\dot{J}|_{t_f} = -1 - (\rho_1\delta_m\dot{\delta}_m + \rho_2\omega\dot{\omega} + \rho_3\delta\dot{\delta} + \rho_4 V\dot{V})|_{t_f}$.

The gradient of the cost function with respect to the switching vector can be given by

$$\begin{aligned} \nabla^J_{\underline{\tau}} &= -\underline{\phi}^N \\ &= [\phi(\tau_1)\ \cdots\ (-1)^{N-2}\phi(\tau_{N-1})\ \dot{J}(t_f)]^\tau \end{aligned} \tag{6.81}$$

where $\underline{\phi}^N = [-\phi(\tau_1)\ \cdots\ (-1)^{N-1}\phi(\tau_{N-1})\ -\dot{J}(t_f)]^\tau$.

The optimal switching vector can then be obtained by using a gradient-based method through iterations [7, 13].

$$\underline{\tau}_{i+1} = \underline{\tau}_i + K_i\underline{\phi}^N_i \tag{6.82}$$

where $\underline{\tau}_i$ is the switching vector, , $\underline{\phi}^N_i$ is the negative gradient vector, and K_i is a properly-chosen $N \times N$-dimensional diagonal matrix with non-negative entries for the ith iteration.

6.4.3. Equilibrium stabilization

For some values of Q_1, the equilibrium of the system (6.4) are unstable. It is necessary to stabilize the equilibrium. Of course, the state-feedback nonlinear control is also able to stabilize the equilibrium after it supresses the transient and brings the system to a small neighborhood of the desired equilibrium. However, it is more often effective to employ a linear control around the equilibrium to enhance the damping.

Around the equilibrium x_e with $u_e = 0$, the linearized version of the system (6.4) is obtained in the following:

$$\begin{aligned}
\Delta\dot{\delta}_m &= \Delta\omega \\
\Delta\dot{\omega} &= -a_2 V_e \cos(\delta_e - \delta_{me} - \theta_m)\Delta\delta_m + a_1\Delta\omega + a_2 V_e \cos(\delta_e - \delta_{me} - \theta_m)\Delta\delta \\
&\quad + a_2 \sin(\delta_e - \delta_{me} - \theta_m)\Delta V \\
\Delta\dot{\delta} &= b_4 V_e \sin(\delta_e - \delta_{me} + \theta_m)\Delta\delta_m + V_e[-b_3 \sin(\delta_e + \theta_0) - b_4 \sin(\delta_e - \delta_{me} + \theta_m)]\Delta\delta \\
&\quad + [2b_1' V_e + b_2 + b_3 \cos(\delta_e + \theta_0) + b_4 \cos(\delta_e - \delta_{me} + \theta_m)]\Delta V + b_5 V_e^2 \Delta u \\
\Delta\dot{V} &= V_e[-c_4 \cos(\delta_e - \delta_{me} + \theta_m) + c_6 \sin(\delta_e - \delta_{me} + \theta_m)]\Delta\delta_m \\
&\quad + V_e[c_3 \cos(\delta_e + \theta_0) + c_4 \cos(\delta_e - \delta_{me} + \theta_m) \\
&\quad - c_5 \sin(\delta_e + \theta_0) - c_6 \sin(\delta_e - \delta_{me} + \theta_m)]\Delta\delta \\
&\quad + [2c_1' V_e + c_2 + c_3 \sin(\delta_e + \theta_0) + c_4 \sin(\delta_e - \delta_{me} + \theta_m) \\
&\quad + c_5 \cos(\delta_e + \theta_0) + c_6 \cos(\delta_e - \delta_{me} + \theta_m)]\Delta V + c_7 V_e^2 \Delta u
\end{aligned} \tag{6.83}$$

Since $\theta_m' = -\delta_e + \delta_{me} + \theta_m$, $\theta_0' = \delta_e + \theta_0$, and $\theta_m'' = \delta_e - \delta_{me} + \theta_m$, the above equation can be simplified as follows:

$$\begin{aligned}
\Delta\dot{\delta}_m &= \Delta\omega \\
\Delta\dot{\omega} &= -a_2 V_e \cos(\theta_m')\Delta\delta_m + a_1\Delta\omega + a_2 V_e \cos(\theta_m')\Delta\delta - a_2 \sin(\theta_m')\Delta V \\
\Delta\dot{\delta} &= b_4 V_e \sin(\theta_m'')\Delta\delta_m + V_e[-b_3 \sin(\theta_0') - b_4 \sin(\theta_m'')]\Delta\delta \\
&\quad + [2b_1' V_e + b_2 + b_3 \cos(\theta_0') + b_4 \cos(\theta_m'')]\Delta V + b_5 V_e^2 \Delta u
\end{aligned}$$

$$\begin{aligned}\Delta\dot{V} &= V_e[-c_4\cos(\theta_m^{''}) + c_6\sin(\theta_m^{''})]\Delta\delta_m \\ &\quad + V_e[c_3\cos(\theta_0^{'}) + c_4\cos(\theta_m^{''}) - c_5\sin(\theta_0^{'}) - c_6\sin(\theta_m^{''})]\Delta\delta \\ &\quad + [2c_1^{'}V_e + c_2 + c_3\sin(\theta_0^{'}) + c_4\sin(\theta_m^{''}) + c_5\cos(\theta_0^{'}) + c_6\cos(\theta_m^{''})]\Delta V + c_7V_e^2\Delta u\end{aligned}$$

6.4.4. Simulation results

First of all, both the quasi-steady-state stabilization and the transient stabilization, via gain scheduling, of the system (6.4) are simulated.

The dependence of the load-bus voltage magnitude V upon the reactive power demand Q_1 is demonstrated in Figure 6.3. It is apparent that the voltage magnitude decreases as the reactive power demand increases until the voltage collapses at the bifurcation point.

As Q_1 varies slowly with time in practice, the equilibrium of the system varies slowly accordingly until it loses the equilibrium. Since the moving equilibrium demonstrates unstable modes, a feedback control must be imposed. The state-feedback control is designed for some specific values of Q_1. The resulting 131 patterns of the gain vector vs Q_1 are used to train a one-hidden neural network with 15 neurons (with a logistic sigmoidal function as the activation function) in the hidden-layer.

To illustrate the effectiveness of the trained neural-net-based controller with respect to the variation in the reactive power demand Q_1 and the small disturbance in the state, the reactive power demand Q_1 is supposed to vary slowly, as shown in Figure 6.4, and the postfault initial state is assumed every several seconds. The transients to the state are assumed in such a way that $x_+ = x_- + [0.1\ 0.02\ 0.1\ 0.1]^\tau$, where x_+ and x_- represent the pre-fault state value and post-fault state value, respectively.

It is observed from Figure 6.5 that the whole system is stabilized with the transients suppressed and the equilibrium stabilized. The neural-net-based gain is shown in Figure 6.6.

Next, the transient stabilization, via nonlinear control, of the system (6.4) is simulated. Note that in reality the control of the generation-side and the control of the load-side are often considered separately due to the fact that the control on the generation side has relatively weak impact on the load-side, and vice versa. A variable resistor is installed on the generator side. The resulting system may be written as

$$\begin{aligned}
\dot{\delta_m} &= \omega \\
\dot{\omega} &= a_1\omega + a_2 V\sin(\delta - \delta_m - \theta_m) + a_0 + a_3 v \\
\dot{\delta} &= b_1 V^2 + [b_2 + b_3\cos(\delta + \theta_0) + b_4\cos(\delta - \delta_m + \theta_m)]V + b_5 V^2 u - b_5 Q_1 + b_0 \\
\dot{V} &= c_1 V^2 + [c_2 + c_3\sin(\delta + \theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\
&\quad + c_5\cos(\delta + \theta_0) + c_6\cos(\delta - \delta_m + \theta_m)]V + c_7 V^2 u - c_7 Q_1 + c_0
\end{aligned} \tag{6.84}$$

where $a_3 = -E_m^2/M$ and v represents the conductance of the resistor.

First of all, for the convenient design of a proper control v, the generator dynamics is considered while the load side voltage dynamics is ignored. The original fourth-order system reduces to a second-order system. With the application of the hierarchical neural control design method developed in chapter 5, the lower-level neural controllers, corresponding to cases $Q_1 = 11.00,\ 11.15,\ 11.30$, are trained based on the data obtained through use of the Switching-Time-Variation Method for calculation of time-optimal control. The upper-level neural networks, acting as multipliers, are also trained for these nominal cases. Once these trainings are done, the hierarchical neural controller is tested for untrained cases. For $Q_1 = 11.10$, the behavior of the controlled system is shown in Figure 6.7. For $Q_1 = 11.20$, behavior of the controlled system is shown in Figure 6.8. It can be seen that even for the untrained cases, the synthesized hierarchical neural controller for generator dynamics performs reasonably well.

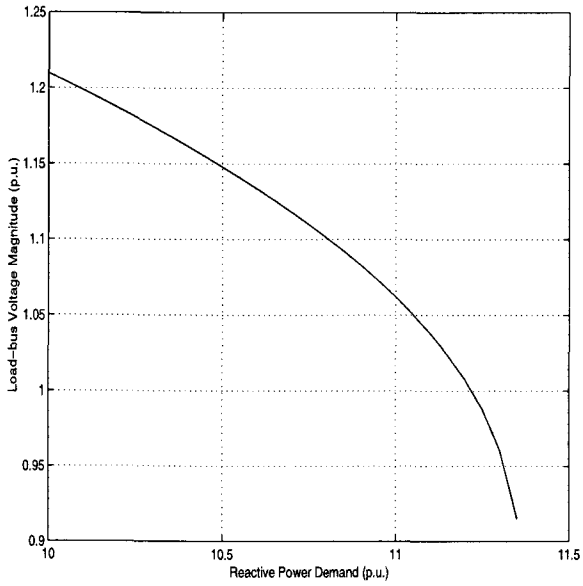


FIGURE 6.3: QV curve

Similarly, for the convenient design of a proper control u, the load side voltage dynamics is considered while the generator dynamics is ignored. The original fourth-order system also reduces to a second-order system. With the appication of the hierarchical neural control design method developed in chapter 5, the lower-level neural controllers, corresponding to cases $Q_1 = 11.00,\ 11.15,\ 11.30$, are trained based on the data obtained through use of the Switching-Time-Variation Method for calculation of time-optimal control. The upper-level neural networks, acting as multipliers, are also trained for these nominal cases. Once these trainings are done, the hierarchical neural controller is tested for untrained cases. For $Q_1 = 11.10$, the behavior of the controlled system is shown in Figure 6.9. For $Q_1 = 11.20$, behavior of the controlled system is shown in Figure 6.10. It can be seen that even for the untrained cases, the synthesized hierarchical neural controller for load side voltage dynamics also performs reasonably well.

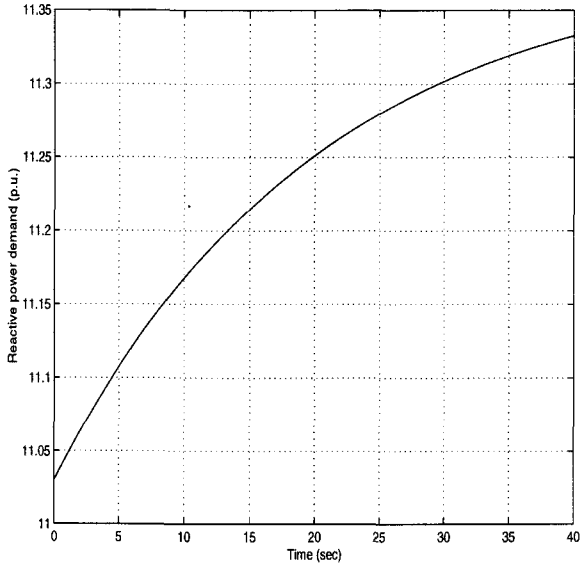


FIGURE 6.4: Reactive power demand Q_1 varying with time

Finally, the hierarchical neural controllers for both generator dynamics and load side voltage dynamics are applied for stabilizing the whole fourth-order system. The hierarchical neural controllers designed and trained for generator dynamics and load side voltage dynamics are put together to form the control for the complete fourth-order system. This control strategy is based on the observations that in power system practice control of generator dynamics and load side dynamics is usually considered separately though there are cases where generator dynamics is interacting with the load side dynamics. If there is strong interaction between generator dynamics and load side dynamics, the installation of TCSC may be a possible solution for control purpose, and its design can just follow the same design procedures for control of either generator dynamics or load side dynamics with the only difference that for the former case the considered system is of higher order.

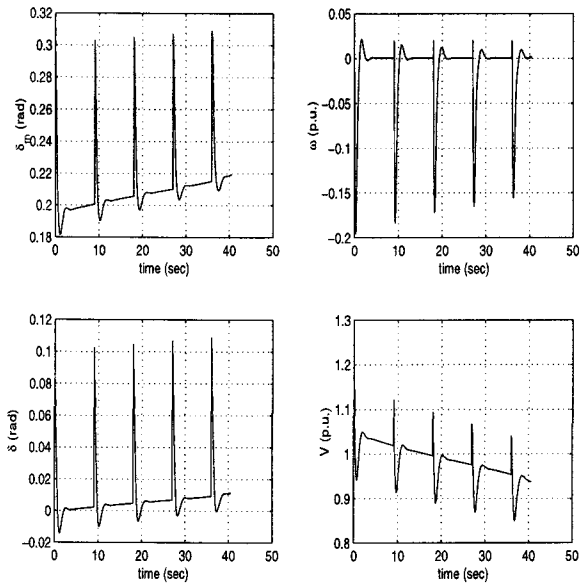


FIGURE 6.5: Quasi-steady-state and transient stabilization via neural control

Consider again the system (6.84)

$$\begin{aligned}
\dot{\delta_m} &= \omega \\
\dot{\omega} &= a_1\omega + a_2 V\sin(\delta - \delta_m - \theta_m) + a_0 + a_3 v \\
\dot{\delta} &= b_1 V^2 + [b_2 + b_3\cos(\delta + \theta_0) + b_4\cos(\delta - \delta_m + \theta_m)]V + b_5 V^2 u - b_5 Q_1 + b_0 \\
\dot{V} &= c_1 V^2 + [c_2 + c_3\sin(\delta + \theta_0) + c_4\sin(\delta - \delta_m + \theta_m) \\
&\quad + c_5\cos(\delta + \theta_0) + c_6\cos(\delta - \delta_m + \theta_m)]V + c_7 V^2 u - c_7 Q_1 + c_0
\end{aligned}$$

As before, translating the system equilibrium into the origin yields the following:

$\dot{\delta_m} = \omega$

$\dot{\omega} = a_1\omega + a_2(V + V_e)\sin(\delta - \delta_m - \theta_m') + a_0 + a_3 v$

$\dot{\delta} = b_1'(V + V_e)^2 + [b_2 + b_3\cos(\delta + \theta_0') + b_4\cos(\delta - \delta_m + \theta_m'')](V + V_e) + b_5(V + V_e)^2 u - b_5 Q_1 + b_0$

$\dot{V} = c_1'(V + V_e)^2 + [c_2 + c_3\sin(\delta + \theta_0') + c_4\sin(\delta - \delta_m + \theta_m'')$

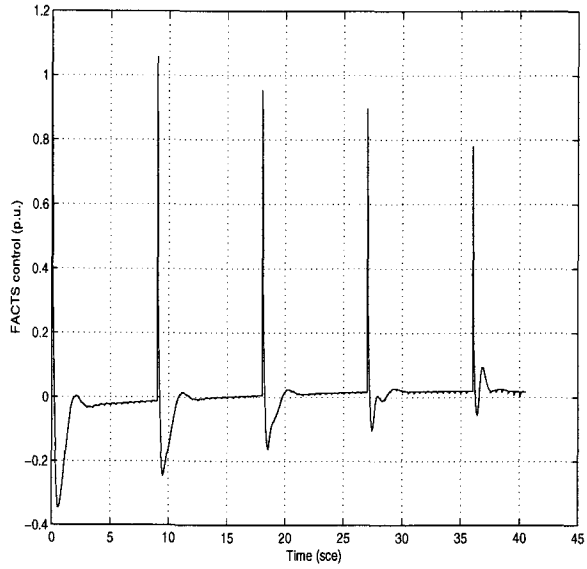


FIGURE 6.6: Neural-net-based feedback generated control

$+ c_5 \cos(\delta + \theta_0') + c_6 \cos(\delta - \delta_m + \theta_m'')](V + V_e) + c_7(V + V_e)^2 u - c_7 Q_1 + c_0$

Note that the load side dynamics has effect on the generator dynamics by observing the first two equations above, and the generator dynamics has effect on the load side dynamics, too, by observing the last two equations above. This indicates that the generator dynamics interacts with the load side dynamics. However, the design of the hierarchical neural controller for generator dynamics is based on the first two equations above with V and δ forced to zeros, and similarly the design of the hierarchical neural control for load side dynamics is based on the last two equations above with δ_m and ω forced to zeros. To justify such a design, the control variable v is split into two parts, namely v_1 and v_2. v_1 functions just as designed to be a neural controller. v_2 is designed to reject the possible disturbance caused by the load side dynamics, and can be given by $v_2 = \frac{a_2}{a_3}[V_e \sin(-\delta_m - \theta_m') - (V + V_e)\sin(\delta - \delta_m - \theta_m')]$.

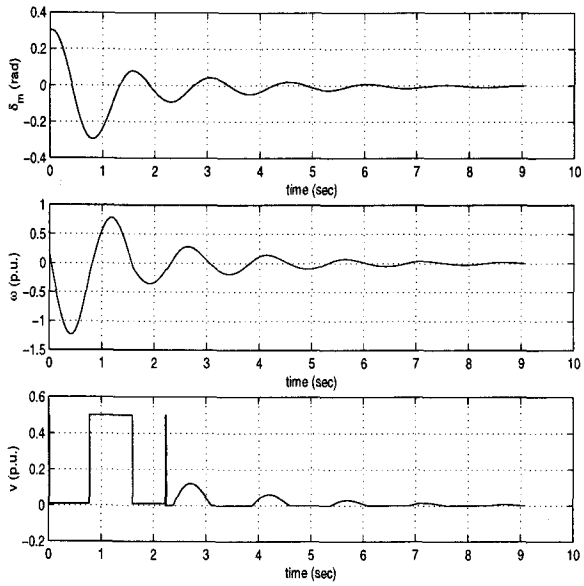


FIGURE 6.7: Performance of the hierarchical neural controller for generator dynamics; $Q_1 = 11.10$;the equilibrium is translated to the origin.

On the other hand, since only δ_m has some impact on the load side dynamics, and it appears in either sin(.) terms or cos(.) terms, the impact of its variation can be suppressed by the designed neural controller for load side dynamics, as is illustrated in what follows. For $Q_1 = 11.20$, with a quite large disturbance, the behavior of the controlled system and the imposed control are all shown in Figure 6.11. It can be seen that the synthesized hierarchical neural controller for the complete fourth-order system performs satisfactorily.

It should be noted that the design of v_2 is dependent on the complete knowledge of the plant dynamics, which is often not the case in reality. Instead, we will not split v into v_1 and v_2. In other words, the control on the generator side is considered to be v as in the system (6.84). The design of control v is not dependent on the availability of the exact information about the generator dynamics. The above designed hierarchical neural controller is tested against a large disturbance for $Q_1 = 11.20$. The behavior of

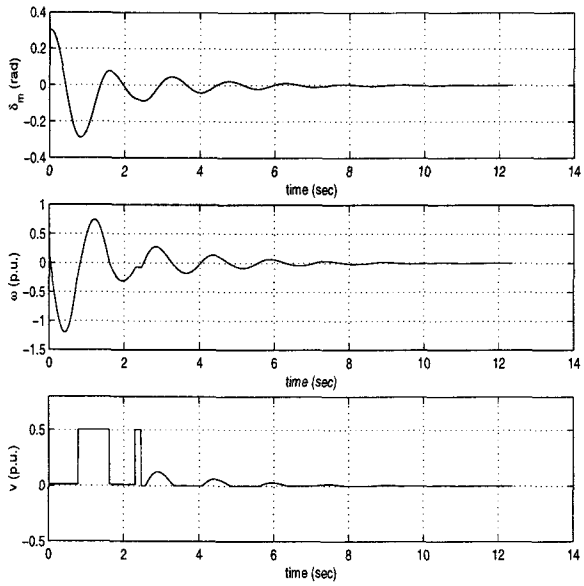


FIGURE 6.8: Performance of the hierarchical neural controller for generator dynamics; $Q_1 = 11.20$;the equilibrium is translated to the origin.

the controlled system and the imposed control are all shown in Figure 6.12. It can be seen again that the synthesized hierarchical neural controller for the complete fourth-order system performs satisfactorily.

6.5. Conclusions

First of all, a few examples are given to demonstrate the fact that under (or even sometimes without) some mild assumptions, many power systems may be modeled as affine systems which are affine in both control and parameters. Proper control of power systems is crucial for maintaining their transient and/or steady-state stability. Since the parameters (for instance, the load demand) are time-varying, this requires that the

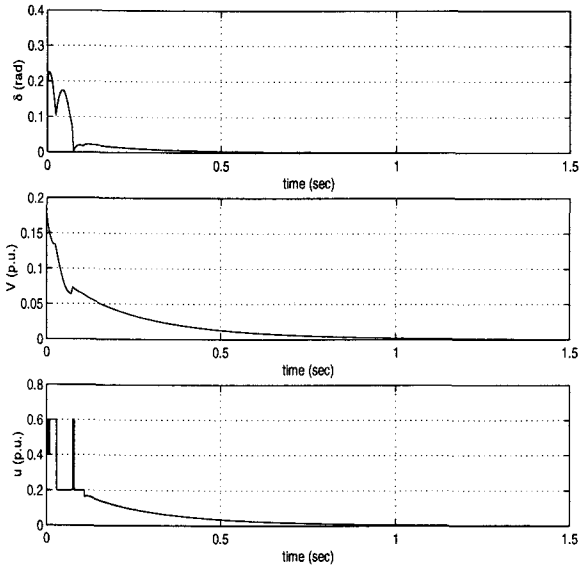


FIGURE 6.9: Performance of the hierarchical neural controller for load side voltage dynamics; $Q_1 = 11.10$;the equilibrium is translated to the origin.

designed controller must be somehow robust with respect to the variation of parameters, or adaptive based on the estimated parameters.

The issue on adaptive control of nonlinear systems in general is still quite open. Very few general results are availabe. On the other hand, some results on adaptive control of affine systems which are affine in both control and parameters are available. However, most available results are dependent on quite strong assumptions. For instance, one of such assumptions is that the analytic form of the desired control corresponding to a specified parameters' setting is available.

Motivated by the fact that desirable control can be synthesized by training a neural network off-line with available optimal control and optimal trajectories, our approach is to synthesize a proper multiplicative control by off-line available nominal neural network controllers by updating the according multipliers on-line.

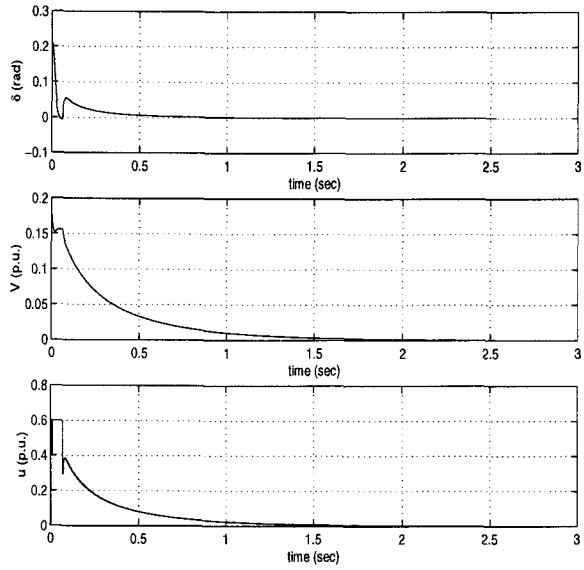


FIGURE 6.10: Performance of the hierarchical neural controller for load side voltage dynamics; $Q_1 = 11.20$;the equilibrium is translated to the origin.

With the employment of the hierarchical control architecture, the adaptive stabilization of affine systems is investigated. First of all, neural-net-based adaptive hierarchical control is studied for time-invariant unknown parameters. Then for time-varying unknown parameters, neural-net-based adaptive hierarchical controllers are synthesized. The relevant system stability issues are studied.

Finally, for a typical model for study of voltage collapse mechanism, several different control approaches are discussed, which include the Lyapunov-analysis based control design, time optimal control, quadratic-performance-index based optimal control, and linear control. The simulation shows that with proper design of a neural network as a pattern identifier, and proper design of lower level nominal linear controllers corresponding to different parameter values, the neural network can properly synthesize a linear controller, which helps stabilize the postfault power system and maintain stability at a desired equi-

librium. Further simulations are also conducted to demonstrate the performances of the adaptive hierarchical neural control, a nonlinear adaptive control strategy for suppressing big transients and stabilizing the system equilibrium.

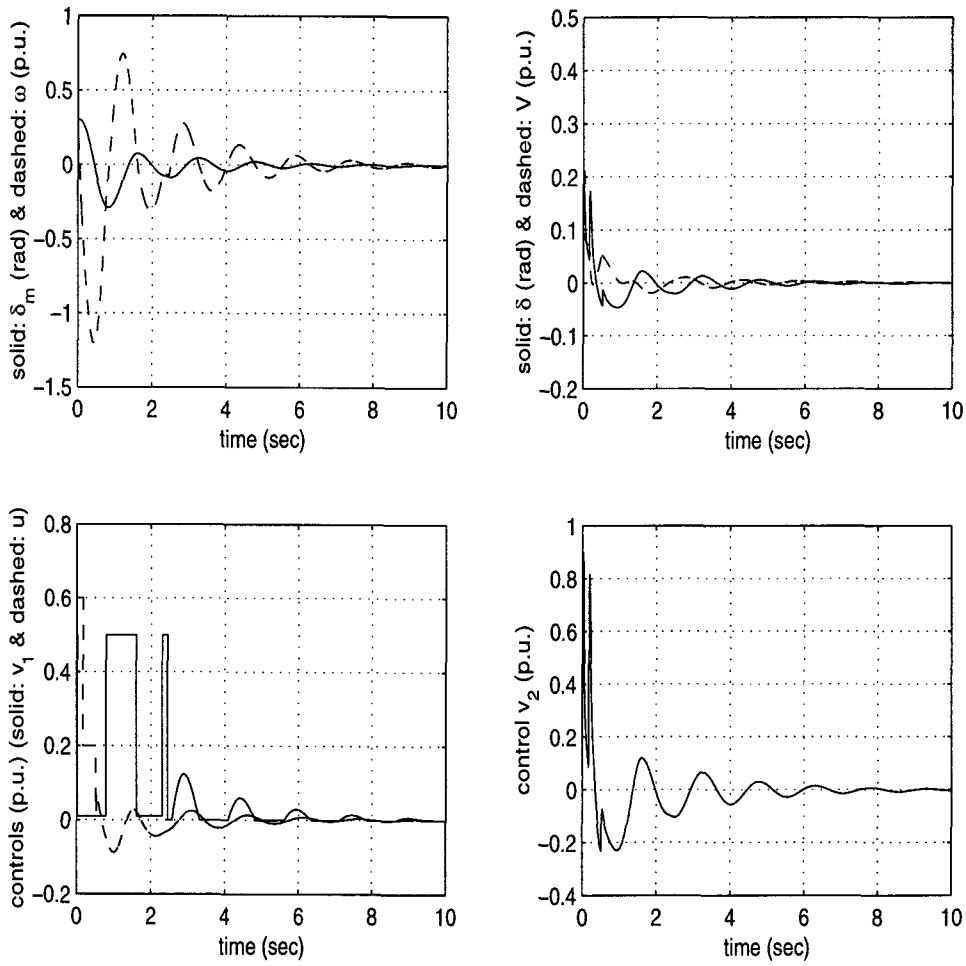


FIGURE 6.11: Performance of the hierarchical neural controller for the whole system; $Q_1 = 11.20$ (with control design for partial system dynamics cancellation); the equilibrium is translated to the origin.

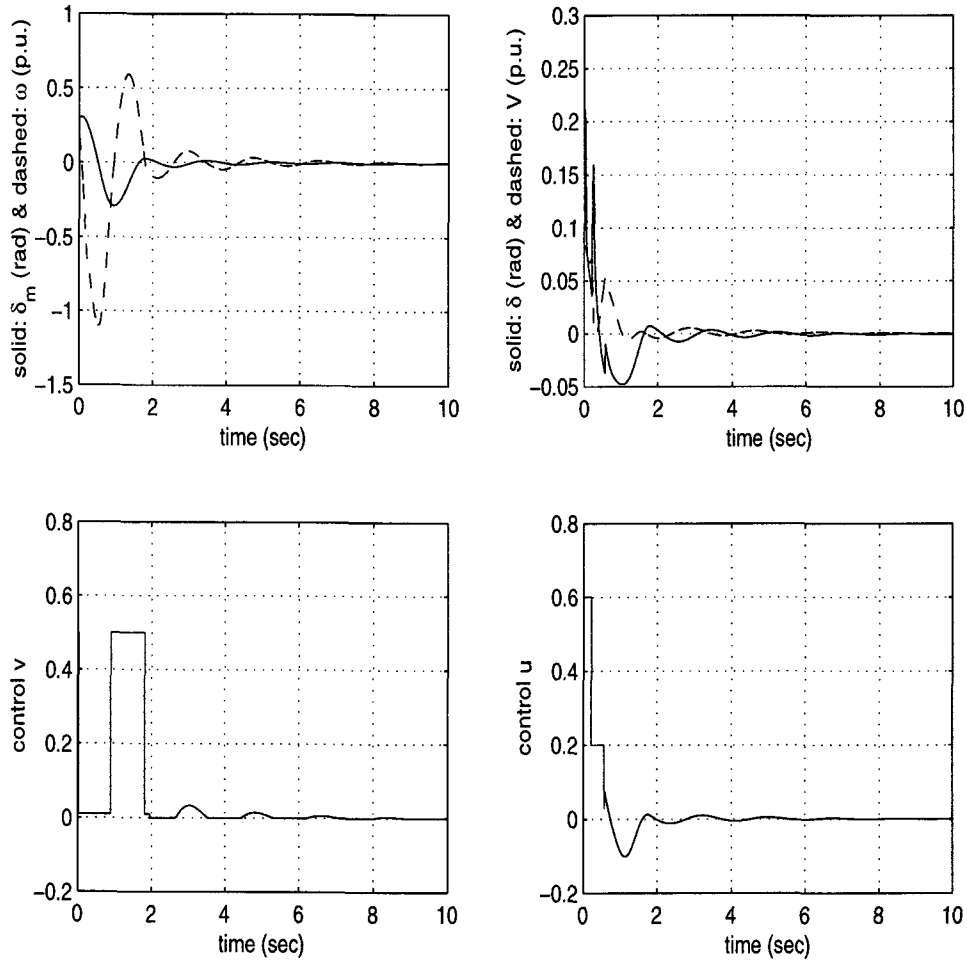


FIGURE 6.12: Performance of the hierarchical neural controller for the whole system; $Q_1 = 11.20$;the equilibrium is translated to the origin.

7. SUMMARY, CONCLUSIONS AND FUTURE RESEARCH

7.1. Summary

This work is devoted to studies on the stabilization of dynamic power systems through use of controlling FACTS devices. Power system stability is a big concern in power engineering. Transient stability and load-driven voltage stability are equally important issues for normal operation of large interconnected power systems. To study load-driven voltage stability, load modeling must be properly handled. Power systems are highly nonlinear and dynamic in general, which calls for high-performance control designs, leading to problems usually intractable. Data-based trajectory-following policy may be used. Artificial neural networks, with rigorous theoretical support and some successful control applications, may offer the potential for wide-spread application to practical dynamic nonlinear systems. Thus, development of novel neural network methodologies for identification and control are not only of academic interest, but also practically significant. Centering on this theme, this manuscript mainly addresses the following problems:

- Development of new artificial neural network methodologies.
- Load modeling through artificial neural networks and voltage stability analysis.
- Stabilization of interconnected power systems following large disturbances, using thyristor-controlled series-capacitor (TCSC), static var compensator (SVC), and braking resistor.
- Development of hierarchical neural-network control architectures.
- Development of adaptive neural-network control design methodologies.

- Studies on the mechanism of voltage collapse and its prevention by proper design of adaptive neural control.

A review of artificial neural networks with application to system identification and control is made in chapter 2. The backpropagation algorithm is discussed and is represented in a compact matrix-format. This is for convenience of software implementation of the backpropagation algorithm.

The theoretical aspects of neural networks are studied in chapter 3. The architecture of latitudinal neural networks is proposed, and its relevant convergence properties are investigated, and further the approximation of a given function, with the finite smooth-decomposition property, can be effectively accomplished in a constructive manner. Moreover, the theoretical results are generalized to multi-dimension cases. They are maybe useful for dynamic system modeling or static mapping.

A neural network methodology is presented in chapter 4 for dealing with static and dynamic load modeling. Further, with the neural-network-based load model, static and dynamic voltage stability analyses are provided. The sensitivities involved in neural network models for loads are derived, and are then used in the Jacobian matrix, and further for the modal analysis. The neural network methodology is tested either on an IEEE 14-bus system or real field data.

The synthesis of intelligent neural controllers is addressed in chapter 5. The approximation of a switching manifold by a neural network is discussed, and a novel pattern recognition scheme for time-optimal control is proposed. To stabilize uncertain dynamic power systems, a hierarchical neural-network control structure is proposed. Further, adaptive neural-network control is presented to deal with the time-varying nature of practical dynamic power systems. These neural control schemes are justified by mathematical verification. Simulations are conducted to demonstrate the performances of the proposed neural control schemes.

The stabilization of postfault multi-machine systems is addressed in chapter 6, together with the inclusion of the time-varying exogenous load model. First of all it is shown that under some mild assumptions, many power systems may be modeled as affine systems which are affine in both control and parameters. Adaptive control designs may benefit from such a simplification.

The issue on adaptive control of nonlinear systems in general is still quite open. Very few general results are available. On the other hand, results on adaptive control of affine systems which are affine in both control and parameters are much richer. However, most available results are dependent on quite strong assumptions. For instance, one of such assumptions is that the analytic form of the desired control corresponding to a specified parameters' setting is available.

Motivated by the fact that desirable control can be synthesized by training a neural network off-line with available optimal control and optimal trajectories, our approach is to synthesize a proper multiplicative control by off-line available nominal neural network controllers with updating the according multipliers on-line.

With the employment of the hierarchical control architecture, the adaptive stabilization of affine systems is investigated. First of all, neural-net-based adaptive hierarchical control is studied for time-invariant unknown parameters. Then for time-varying unknown parameters, neural-net-based adaptive hierarchical controllers are synthesized. The relevant system stability issues are studied.

Finally, for a typical model for study of voltage collapse mechanisms, several different control approaches are discussed. The simulation shows that with proper design of a neural network as a pattern identifier, and proper design of lower level nominal linear controllers corresponding to different parameter values, the neural network can properly synthesize a linear controller, which help stabilize the postfault power system and maintain stability at a desired equilibrium. Further simulations are conducted to demonstrate the performances of the nonlinear adaptive hierarchical neural control, involving the controlling braking

resistor on the generator side and the controlling SVC on the load side. The satisfactory performances of the designed adaptive neural controllers indicate that they may be useful for practical power systems.

7.2. Conclusions and future research

Power systems are complex, nonlinear dynamic systems, for which stability is an overwhelming issue. Regular generator rotor angle stability is well understood, but load-driven voltage instability (or even voltage collapse) needs more in-depth investigations. The interaction of generator dynamics and load dynamics make control design and stability analysis even more difficult. In the literature, generator dynamics and load dynamics are usually dealt with separately, which makes control design and stability analysis much easier, but may give misleading results. This work deals with power system stability issues with considering generator dynamics as well as load effects, and the methods proposed and developed can be applied to complex nonlinear systems in general. Simulations are conducted on simplified power system models, and have indicated that the proposed methods may be useful for real power systems. For implementation of the proposed control design strategies, future research that may be conducted should include the following aspects:

- perform simulations on complex multi-machine systems.
- perform simulations with inclusion of dynamic neural network model for loads.
- develop control design methodologies to deal with hybrid nonlinear systems composed of both time-continuous and discrete models.

REFERENCES

1. E. Laszlo, *The relevance of general systems theory*, George Braziller, Inc., New York, 1972.

2. L. Ljung and T. Söderström, *Theory and practice of recursive identification*, the MIT Press, Mass., 1983.

3. L. Ljung, *System identification: Theory for the user*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

4. A. A. Fel'dbaum, *Optimal control systems*, Academic Press, New York, 1965.

5. K. S. Narendra and A. M. Annaswamy, *Stable Adaptive Systems*, Prentice Hall, Englewood Cliffs, New Jersery, 1989.

6. L. Praly, G. Bastin, J. B. Pomet, and Z. P. Jiang, " Adaptive stabilization of nonlinear systems," in *Foundations of Adaptive Control* (P. V. Kokotovic Ed.), 347-433, Springer-Verlag, Berlin, 1991.

7. R. Mohler,*Bilinear control processes: with applications to engineering, ecology, and medicine*, Academic Press, New York, 1973.

8. R. R. Mohler, *Nonlinear Systems Volume I, Dynamics and Control*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

9. R. R. Mohler, *Nonlinear Systems Volume II, Applications to Bilinear Control*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

10. V. Rajkumar and R. R. Mohler, "Bilinear generalized predictive control using the thyristor-controlled series capacitor," Pre-Print, IEEE PES Winter Meeting, 94 WM 194-1 PWRS, New York, 1994.

11. V. Rajkumar, *Nonlinear Control Applied to Power Systems*, Ph.D. dissertation, Oregon State University, 1994.

12. R. R. Zakrzewski, R. R. Mohler, and W. J. Kolodziej, "Hierarchical intelligent control with flexible AC transmission system application," *IFAC J. Control Engineering Practice*, vol. 2, 979-987, 1994.

13. Sang Fi Moon, *Optimal Control of Bilinear Systems and Systems Linear in Control*, Ph.D. dissertation, The University of New Mexico, 1969.

14. C. W. Taylor, *Power system voltage stability*, McGraw-Hill, Inc., New York, 1994.

15. W. Price, K. Wirgau, et al, "Load modeling for power flow and transient stability computer studies," *IEEE Trans. on Power Systems*, vol. 3, 180-187, 1988.

16. J. J. Hopfield and D. W. Tank, "Neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, 141-152, 1985.

17. Y. T. Zhou, R. Chellappa, A. Vaid and B. K. Jenkins, "Image restoration using a neural network," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 36, 1141-1151, 1988.

18. G. W. Cottrell, P. Munro and D. Zipser, "Learning internal representations from gray-scale images: An example of extensional programming," Proc. Ninth Conf. Cognitive Science Society, 462-473, Erlbaum, Hillsdale, 1987.

19. D. A. Mighell, T. S. Wilkinson and J. W. Goodman, "Backpropagation and its application to handwritten signature verification," In *Advances in Neural Information Processing Systems 1*, D. S. Touretzky, eds., 340-347, Morgan Kaufmann, San Mateo, California, 1989.

20. R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley, New York, 1973.

21. B. Ku, R. J. Thomas, C. Chiou and C. Lin, "Power system dynamic load modelling using artificial neural networks," *IEEE Trans. on Power Systems*, vol. 9, 1868-1874, 1994.

22. A. U. Levin and K. S. Narendra, "Control of nonlinear dynamical systems using neural networks: controllability and stabiliztion," *IEEE Trans. on Neural Networks*, vol. 4, 192-206, 1993.

23. W. S. McCollough and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, 115-133, 1943.

24. D. Hebb, *The organization of behavior*, John Wiley and Sons, New York, 1949.

25. F. Rosenblatt, *Principles of neurodynamics*, Spartan Books, Washington, 1961.

26. B. Widrow, "Generalization and information storage in networks of Adaline 'neurons'," In *Self-Organizing Systems*, M. Yovitz, G. Jacobi, G. Goldstein, eds., 435-461, Spartan Books, Washington, 1962.

27. M. L. Minsky and S. A. Papert, *Perceptrons*. MIT Press, Mass., 1969.

28. S. Grossberg, "Competitive learning: From interactive activation to adaptive resonance," *Cognitive Science*, vol. 11, 23-63, 1987.

29. T. Kohonen, "Correlation matrix memories," *IEEE Trans. on Computers*, vol. 21, 353-359, 1972.

30. D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing*, vol. 1-2, MIT Press, Mass., 1986.

31. K. S. Narendra and K. Parthasarathy,"Identification and control of dynamical systems using neural networks," *IEEE Trans. Neural Networks*, vol. 1, 4-27, 1990.

32. K. S. Narendra and K. Parthasarathy,"Gradient methods for the optimization of dynamical systems containing neural networks," *IEEE Trans. Neural Networks*, vol. 2, 252-261, 1991.

33. A. G. Parlos, K. T. Chong, and A. F. Atiya,"Application of the recurrent multilayer perceptron in modeling complex process dynamics," *IEEE Trans. Neural Networks*, vol. 5, 255-266, 1994.

34. K. Funahashi,"On the approximate realization of continuous mappings by neural networks," *Neural Networks*, vol. 2, 183-192, 1989.

35. E. J. Hartman, J. D. Keeler, and J. M. Kowalski," Layered neural networks with Gaussian hidden units as universal approximations," *Neural Computation*, vol. 2, 210-215, 1990.

36. P. J. Werbos, *New tools for prediction and analysis in the behavioral sciences*, Ph.D. Dissertation, Harvard University, Cambridge, 1974.

37. R. Hecht-Nielsen, "Theory of the backpropagation neural network," Proc. IJCNN, vol. 1, 593-605, Washington, 1989.

38. K.Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, 251-257, 1991.

39. K. Hornik, M. Stinchcombe and H. WHite, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, 359-366, 1989.

40. G. Cybenko,"Approximation by superpositions of a sigmoidal function," *Math. Contr., Signals, Syst.*, vol. 2, 303-314, 1989.

41. E. D. Dahl ,"Accelerated learning using the generalized delta rule," Proc. IEEE ICNN, vol. 2, 523-530, San Diego, 1987.

42. D. R. Hush and J. M. Salas, "Improving the learning rate of backpropagation with the gradient reuse technique," Proc. IEEE ICNN, vol. 1, 383-389, San Diego, 1988.

43. T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink and D. L. Alkan, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, vol. 59, 257-263, 1988.

44. D. F. Shanno, "Conjugate gradient mehtods with inexact searches," *Math. of Operations Research*, vol. 3, 244-256, 1978.

45. T. Chen and H. Chen, "Approximations of continuous functionals by neural networks with application to dynamic systems," *IEEE Trans. Neural Networks*, vol. 4, 910-918, 1993.

46. C. H. Choi and J. Y. Choi, "Construction of neural networks to approximate arbitrary continuous functions of one variable," *Elect. Lett.*, vol. 28, 151-153, 1992.

47. C. H. Choi and J. Y. Choi, "Constructive neural networks with piecewise interpolation capabilities for function approximations," *IEEE Trans. on Neural Networks*, vol. 5, 936-944, 1994.

48. E. K. Blum and L. K. Li, "Approximation theory and feedforward networks," *Neural Networks*, vol. 4, 511-515, 1991.

49. J. R. Munkres, *Topology: A first course*, Prentice-Hall, Englewood-Cliffs, New Jersey, 1974.

50. A. U. Levin and K. S. Narendra, "Control of nonlinear dynamical systems using neural networks— Part II: Observability, Identification and Control", *IEEE Trans. Neural Networks*, vol. 7, 30-42, 1996.

51. D. Chen and R. Mohler, "Latitudinal and Longitudinal Neural Network Structures for Function Approximations", Proc. IEEE Conf. Electronics, Circuits, and System, Greece, 1996.

52. D. Chen and R. Mohler, "The properties of latitudinal neural networks with potential power system applications", Proc. American Control Conference, Philadelphia, 1998.

53. R. M. Burton and H. G. Dehling, *Mathematical Aspects of Neural Computing*, Preprint, Department of Mathematics, Oregon State University, 1996.

54. T. J. Overbye and C. L. Demarco, "Voltage security enhancement using energy based sensitivities," *IEEE Trans. Power Systems*, vol. 6, 1196-1202, 1991.

55. V. Ajjarapu and C. Christy, "The continuation power flow: A tool for steady state voltage stability analysis," *IEEE Trans. Power Systems*, vol. 7, 416-423, 1992.

56. B. Gao, G. K. Morison and P. Kundur, "Voltage stability evaluation using modal analysis," *IEEE Trans. Power Systems*, vol. 7, 1529-1536, 1992.

57. C. D. Vournas, "Voltage stability and controllability indices for multimachine power systems," *IEEE Trans. Power Systems*, vol. 10, 1183-1191, 1995.

58. T. V. Cutsem,"An approach to corrective control of voltage instability using simulation and sensitivity," *IEEE Trans. Power Systems*, vol. 10, 616-622, 1995.

59. G. K. Morison, B. Gao and P. Kundur, "Voltage stability analysis using static and dynamic approaches," *IEEE Trans. Power Systems*, vol. 8, 1159-1165, 1993.

60. M. K. Pal, "Voltage stability conditions considering load characteristics," *IEEE Trans. Power Systems*, vol. 7, 243-249, 1992.

61. OSU-ECE-FACTS 9401 Report on Intelligent Control of Complex Nonlinear Systems with Electric Power Applications, 1994.

62. G. C. Ejebe et al, "Methods for contingency screening and ranking for voltage stability analysis of power systems," *IEEE Trans. Power Systems*, vol. 11, 350-356, 1996.

63. D. J. Hill, "Nonlinear dynamic load models with recovery for voltage stability studies," *IEEE Trans. Power Systems*, vol. 8, 166-172, 1993.

64. D. Karlsson and D. J. Hill, "Modelling and identification of nonlinear dynamic loads in power systems," *IEEE Trans. Power Systems*, vol. 9, 157-163, 1994.

65. T. J. Overbye and C. L. Demarco, "Improved techniques for power system voltage stability assessment using energy methods," *IEEE Trans. on Power Systems*, vol. 6, 1446-1452, 1991.

66. N. Yorino et al, "An investigation of voltage instability problems," *IEEE Trans. Power Systems*, vol. 7, 600-607, 1992.

67. B. Lee and V. Ajjarapu, "A piecewise global small-disturbance voltage-stability analysis of structure-preserving power system models," *IEEE Trans. Power Systems*, vol. 10, 1963-1968, 1995.

68. H. Chiang and R. Jean-Jumeau, "Toward a practical performance index for predicting voltage collapse in elctric power systems," *IEEE Trans. Power Systems*, vol. 10, 584-590, 1995.

69. Discussion of "An investigation of voltage instability problems," *IEEE Trans. Power Systems*, vol. 7, 608-611, 1992.

70. Discussion of "Voltage stability analysis using generic dynamic load models," *IEEE Trans. Power Systems*, vol. 9, 487-493, 1994.

71. Discussion of "Voltage stability and controllability indices for multimachine power systems," *IEEE Trans. Power Systems*, vol. 10, 1191-1194, 1995.

72. Discussion of "Voltage stability analysis in transient and mid-term time scales," *IEEE Trans. Power Systems*, vol. 11, 153-154, 1996.

73. Discussion of "Voltage stability analysis using static and dynamic approaches," *IEEE Trans. Power Systems*, vol. 8, 1166-1171, 1993.

74. Discussion of "A piecewise global small-disturbance voltage-stability analysis of structure-preserving power system models," *IEEE Trans. Power Systems*, vol. 10, 1969-1971, 1995.

75. W. Xu and Y. Mansour, "Voltage stability analysis using generic dynamic load models," *IEEE Trans. Power Systems*, vol. 9, 479-486, 1994.

76. K. Iba et al.,"A method for finding a pair of multiple load flow solutions in bulk power systems," Proc. PICA, 98-104, Seattle, 1989.

77. A. A. EI-Keib and X. Ma, "Application of artificial neural networks in voltage stability assessment," *IEEE Trans. Power Systems*, vol. 10, 1890-1895, 1995.

78. Closure of "Toward a practical performance index for predicting voltage collapse in elctric power systems," *IEEE Trans. Power Systems*, vol. 10, 591-592, 1995.

79. IEEE Committee Report, "Load representation for dynamic performance studies," IEEE/PES Winter Meeting, WM 126-3 PWRS, 1992.

80. J. T. Connor, R. D. Martin, and L. E. Atlas, "Recurrent neural networks and robust time series prediction," *IEEE Trans. Neural Networks*, vol. 5, 240-254, 1994.

81. IEEE Committee Report, *Voltage Stability of Power Systems: Concept, Analytical Tools, and Industry Experience*, IEEE publication 90TH0358-2-PWR.

82. Nonlinear Control and Operation of FACTS: Methodologies and Basic Concepts, EPRI TR-103398, Palo Alto, 1995.

83. Y. Wang et al, "Variable structure FACTS controllers for power system transient stability," *IEEE Trans. Power Systems*, Vol. 7, 307-313, 1992

84. D. Kosterev and W. Kolodziej, "Robust control for power system transient stabilization," Proc. American Control Conf., San Francisco, 1993.

85. K. Narendra and S. Mukhopadhyay, "Intelligent control using neural networks," *IEEE Control Systems Magazine*, vol. 12, 11-18, 1992.

86. M. Herman, J. Albus, and T. Hong, "Intelligent control for multiple autonomous undersea vehicles," In *Neural Networks for Control* (W. T. Miller, R. S. Sutton and P. J. Werbos, eds.), 427-474, MIT Press, Mass., 1990.

87. K. Tanaka et al, "Fuzzy control of a vehicle with two trailers," Proc. American Control Conf., Albuquerque, 1997.

88. T. W. Long et al, "A neural network based receding horizon optimal controller," Proc. American Control Conf., Albuquerque, 1997.

89. N. B. Karayiannis and A. N. Venetsanopoulos, *Artificial Neural Networks: Learning Algorithms, Performance Evaluation, and Applications*, Kluwer Academic Publishers, Boston, 1993.

90. P. J. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*, John Wiley & Sons, New York, 1994.

91. M. Kawato, "Computational schemes and neural network models for formation and control of multijoint arm trajectory," In *Neural Networks for Control* (W. T. Miller, R. S. Sutton and P. J. Werbos, eds.), 197-228, MIT Press, Mass., 1990.

92. C. Atkeson and D. Reinkensmeyer, "Using associative content-addressable memories to control robots," In *Neural Networks for Control* (W. T. Miller, R. S. Sutton and P. J. Werbos, eds.), 255-285, MIT Press, Mass.,1990.

93. D. Chen and R. Mohler, "Load modelling and voltage stability analysis by neural networks," Proc. American Control Conf., Albuquerque, 1997.

94. E. B. Lee and L. Markus, *Foundations of Optimal Control Theory*, Wiley, New York, 1967.

95. Wilson J. Rugh, *Linear System Theory*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

96. G. A. Rovithakis and M. A. Christodoulou, "Direct adaptive regulation of unknown nonlinear dynamical systems via dynamic neural networks," *IEEE Trans. Syst. Man Cyber.*, vol. 25, 1578-1594, 1995.

97. G. A. Rovithakis and M. A. Christodoulou, "Adaptive control of unknown plants using dynamical neural networks," *IEEE Trans. Syst. Man Cyber.*, vol. 24, 400-412, 1994.

98. I. Dobson and H. Chiang,"Towards a theory of voltage collapse in electric power systems," *System & Control Letters*, vol. 13, 253-262, 1989.

99. A. E. Bryson and Y. C. Ho, *Applied optimal control : optimization, estimation, and control*, Hemisphere Pub. Corp., distributed by Halsted Press, Washington, 1975.

100. A. Y. Khapalov and R. R. Mohler, "Asymptotic stabilization of the bilinear time-invariant system via piecewise constant feedback," Systems & Control Letters, vol. 33, 47-54, 1998.

101. S. A. Shahrestani and D. J. Hill, "Hierarchical control of bifurcating power systems," Preprint, Univ. of Sydney, 1998.

APPENDICES

A C Programs

In the following, all the programs coded in C for implementation of the training algorithm for neural networks are listed.

```
/* C Programs are listed in the following */

/******************** Begin of ../../C_nnet/mynn.h ************************/

/* This header file is specific to neural network application */

void system_error(char error_message[]);
int *allocate_integer_vector(int l, int u);
float *allocate_real_vector(int l, int u);
int **allocate_integer_matrix(int lr, int ur, int lc, int uc);
float **allocate_real_matrix(int lr, int ur, int lc, int uc);
void free_integer_vector(int *v, int l);
void free_real_vector(float *v, int l);
void free_integer_matrix(int **m, int lr, int ur, int lc);
void free_real_matrix(float **m, int lr, int ur,int lc);

float rand_num(long *idum);
void myrand(float **w,int w_row,int w_col,float *b);
void mynormr(float **w, int w_row, int w_col);
void myrands(float **w, int w_row,int w_col,float *b);
void mynwtan(int s,float ** p,int p_row,int p_col,float ** w,int w_row,
             int w_col,float *b);
void mynwlog(int s,float ** p,int p_row,int p_col,float ** w,int w_row,
             int w_col,float *b);
void myinitff(float ** p, int p_row, int p_col,
              int s1, char *f1,
              int s2, char *f2,
              int s3, char *f3,
              float ** w1, int w1_row, int w1_col, float *b1,
              float ** w2, int w2_row, int w2_col, float *b2,
              float ** w3, int w3_row, int w3_col, float *b3);

void mysimuff(float **p, int p_row, int p_col,
              float **w1, int w1_row, int w1_col, float *b1,
              float **w2, int w2_row, int w2_col, float *b2,
              float **w3, int w3_row, int w3_col, float *b3,
              void (*f1)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*f2)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*f3)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              float **a1, int a1_row, int a1_col,
              float **a2, int a2_row, int a2_col,
              float **a3, int a3_row, int a3_col
             );

void mytbpx3(float **w1, int row1, int col1, float *b1,
             float **w2, int row2, int col2, float *b2,
             float **w3, int row3, int col3, float *b3,
             float **p,  int p_row, int p_col,
             float **t,  int t_row, int t_col,
             float *tp,
```

```
            void (*f1)(float **w,int row,int col,float **x,
            int x_row,int x_col, float *b,float **a),

            void (*f2)(float **w,int row,int col,float **x,
            int x_row,int x_col, float *b,float **a),

            void (*f3)(float **w,int row,int col,float **x,
            int x_row,int x_col, float *b,float **a),
            void (*df1)(float **a,int a_row,int a_col,float **d,
            int d_row,int d_col,float **w,int w_row,int w_col,float **r),
            void (*df2)(float **a,int a_row,int a_col,float **d,
            int d_row,int d_col,float **w,int w_row,int w_col,float **r),
            void (*df3)(float **a,int a_row,int a_col,float **d,
            int d_row,int d_col,float **w,int w_row,int w_col,float **r)
           );

void mytbpx2(float **w1, int w1_row, int w1_col, float *b1,
            float **w2, int w2_row, int w2_col, float *b2,
            float **p,  int p_row, int p_col,
            float **t,  int t_row, int t_col,
            float *tp,
            void (*f1)(float **w,int row,int col,float **x,
            int x_row,int x_col, float *b,float **a),

            void (*f2)(float **w,int row,int col,float **x,
            int x_row,int x_col, float *b,float **a),

            void (*df1)(float **a,int a_row,int a_col,float **d,
            int d_row,int d_col,float **w,int w_row,int w_col,float **r),
            void (*df2)(float **a,int a_row,int a_col,float **d,
            int d_row,int d_col,float **w,int w_row,int w_col,float **r)
           );

float mysumsqr(float **e,int e_row,int e_col);
void mylearnbpm(float **p,int p_row,int p_col,
                float **d,int d_row,int d_col,
                float lr,float mc,
                float **dw,int dw_row,int dw_col,
                float *db);

/************* This is the version for standard Backpropagation ****
float ** mypurelin(float **w,int row,int col,
                   float **x,int x_row,int x_col,
                   float *b);
float ** mylogsig(float **w,int row,int col,
                  float **x,int x_row,int x_col,
                  float *b);
float ** mytansig(float **w,int row,int col,
                  float **x,int x_row,int x_col,
                  float *b);
float ** mydeltalin(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col);
float ** mydeltalog(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col);
float ** mydeltatan(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col);
*********************************************************************/

/************* This is the version for recurrent Backpropagation ****/
```

```
void mypurelin(float **w,int row,int col,
                  float **x,int x_row,int x_col,
                  float *b,float **a);
void mylogsig(float **w,int row,int col,
                 float **x,int x_row,int x_col,
                 float *b,float **a);
void mytansig(float **w,int row,int col,
                 float **x,int x_row,int x_col,
                 float *b,float **a);
void mydeltalin(float **a,int a_row,int a_col,
                   float **d,int d_row,int d_col,
                   float **w,int w_row,int w_col,float **r);
void mydeltalog(float **a,int a_row,int a_col,
                   float **d,int d_row,int d_col,
                   float **w,int w_row,int w_col,float **r);
void mydeltatan(float **a,int a_row,int a_col,
                   float **d,int d_row,int d_col,
                   float **w,int w_row,int w_col,float **r);
/*********************************************************************/


/********* The above is for standard neural network **********/

/********* The following is for recurrent nerual network          ******/
/********* But it can be used for standard neural network          *****/
/********* since the programs are such coded to accomodate both cases **/

void myinitelm(float ** p, int p_row, int p_col,
               int s1, char *f1, int local_recur1,
                                    /* 1/0: yes/no local recurr */
               int s2, char *f2, int local_recur2,
                                    /* 1/0: yes/no local recurr */
               int s3, char *f3, int local_recur3,
                                    /* 1/0: yes/no local recurr */
               float ** w1, int w1_row, int w1_col, float * b1,
               float ** w2, int w2_row, int w2_col, float * b2,
               float ** w3, int w3_row, int w3_col, float * b3);

void mysimuelm(float **p, int p_row, int p_col,
              float **w1, int w1_row, int w1_col, float *b1,int local_recur1,
                                                    /* yes/no: 1/0 */
              float **w2, int w2_row, int w2_col, float *b2,int local_recur2,
                                                    /* yes/no: 1/0 */
              float **w3, int w3_row, int w3_col, float *b3,int local_recur3,
                                                    /* yes/no: 1/0 */
              void (*f1)(float **w,int row,int col,float **x,
                         int x_row,int x_col, float *b,float **a),
              void (*f2)(float **w,int row,int col,float **x,
                         int x_row,int x_col, float *b,float **a),
              void (*f3)(float **w,int row,int col,float **x,
                         int x_row,int x_col, float *b,float **a),
              float **a1, int a1_row, int a1_col,
              float **a2, int a2_row, int a2_col,
              float **a3, int a3_row, int a3_col
             );

void mytrbpx3(float **w1, int w1_row, int w1_col, float *b1, int local_recur1,
              float **w2, int w2_row, int w2_col, float *b2, int local_recur2,
              float **w3, int w3_row, int w3_col, float *b3, int local_recur3,
              float **p,  int p_row, int p_col,
              float **t,  int t_row, int t_col,
              float *tp,
              void (*f1)(float **w,int row,int col,float **x,
```

```
                    int x_row,int x_col, float *b,float **a),
                void (*f2)(float **w,int row,int col,float **x,
                    int x_row,int x_col, float *b,float **a),
                void (*f3)(float **w,int row,int col,float **x,
                    int x_row,int x_col, float *b,float **a),
                void (*df1)(float **a,int a_row,int a_col,float **d,
                int d_row,int d_col,float **w,int w_row,int w_col,float **r),
                void (*df2)(float **a,int a_row,int a_col,float **d,
                int d_row,int d_col,float **w,int w_row,int w_col,float **r),
                void (*df3)(float **a,int a_row,int a_col,float **d,
                int d_row,int d_col,float **w,int w_row,int w_col,float **r)
               );

void mytrbpx2(float **w1, int w1_row, int w1_col, float *b1, int local_recur1,
                float **w2, int w2_row, int w2_col, float *b2, int local_recur2,
                float **p,  int p_row, int p_col,
                float **t,  int t_row, int t_col,
                float *tp,
                void (*f1)(float **w,int row,int col,float **x,
                int x_row,int x_col, float *b,float **a),
                void (*f2)(float **w,int row,int col,float **x,
                int x_row,int x_col, float *b,float **a),
                void (*df1)(float **a,int a_row,int a_col,float **d,
                int d_row,int d_col,float **w,int w_row,int w_col,float **r),
                void (*df2)(float **a,int a_row,int a_col,float **d,
                int d_row,int d_col,float **w,int w_row,int w_col,float **r)
               );

/********** END of mynn.h ***************************/

/********************** Begin of myinitff.c *****************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mynn.h"

void myinitff(float ** p, int p_row, int p_col,
                int s1, char *f1,
                int s2, char *f2,
                int s3, char *f3,
                float ** w1, int w1_row, int w1_col, float *b1,
                float ** w2, int w2_row, int w2_col, float *b2,
                float ** w3, int w3_row, int w3_col, float *b3)

{
  int ii,jj;
  float **tmp2,**tmp3;
  int tmp2_row,tmp2_col,tmp3_row,tmp3_col;

  /* First of all, dimension consistency check */
  if (strcmp(f2,"mypurelin") ==0) {/* only 1 hidden layer case */
    if (w1_col != p_row || w2_col != w1_row || s1 !=w1_row || s2 !=w2_row) {
      printf("Dimension inconsistency error in myinitff\n");
      printf("Exiting from myinitff now.");
      exit(1);
    }
  }

  if (strcmp(f3,"mypurelin") ==0) {/* only 2 hidden layer case */
    if (w1_col!=p_row || w2_col!=w1_row || w3_col!=w2_row || s1!=w1_row
                      || s2!=w2_row || s3!=w3_row)
    {
```

```
    printf("Dimension inconsistency error in myinitff\n");
    printf("Exiting from myinitff now.");
    exit(1);
  }
}

/********* Case 1: first layer mylogsig **********/
if (strcmp(f1,"mylogsig") == 0) {
  mynwlog(s1,p,p_row,p_col,w1,w1_row,w1_col,b1);
  tmp2 = allocate_real_matrix(1,s1,1,2);
                                /* the output range of the 1st layer */
  for (ii=1;ii<=s1;ii++) {
    tmp2[ii][1]=0;
    tmp2[ii][2]=1;
  }
  tmp2_row=s1;
  tmp2_col=2;
  if (strcmp(f2,"mylogsig") ==0 ) {
    mynwlog(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);
    tmp3 = allocate_real_matrix(1,s2,1,2);
                              /* the output range of the 2nd layer */
    for (ii=1;ii<=s2;ii++) {
      tmp3[ii][1]=0;
      tmp3[ii][2]=1;
    }
    tmp3_row=s2;
    tmp3_col=2;
    if (strcmp(f3,"mylogsig") == 0) {
      mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mytansig") == 0) {
      mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mypurelin") == 0) {
      myrands(w3,w3_row,w3_col,b3);
    }
    free_real_matrix(tmp2,1,tmp2_row,1);
    free_real_matrix(tmp3,1,tmp3_row,1);
  }

  if (strcmp(f2,"mytansig") == 0) {
    mynwtan(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);
    tmp3 = allocate_real_matrix(1,s2,1,2);
    for (ii=1;ii<=s2;ii++) {
      tmp3[ii][1]=-1;
      tmp3[ii][2]=1;
    }
    tmp3_row=s2;
    tmp3_col=2;
    if (strcmp(f3,"mylogsig") == 0) {
      mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mytansig") == 0) {
      mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mypurelin") == 0) {
      myrands(w3,w3_row,w3_col,b3);
    }
    free_real_matrix(tmp2,1,tmp2_row,1);
    free_real_matrix(tmp3,1,tmp3_row,1);
  }
```

```
  if (strcmp(f2,"mypurelin") == 0) {
                        /* if layer 2 is linear layer, then no more layers */
    myrands(w2,w2_row,w2_col,b2);
    free_real_matrix(tmp2,1,tmp2_row,1);
  }

}

/***** Case 2: first layer mytansig *********/
if (strcmp(f1,"mytansig") == 0) {
  mynwtan(s1,p,p_row,p_col,w1,w1_row,w1_col,b1);
  tmp2 = allocate_real_matrix(1,s1,1,2);
                        /* the output range of the 1st layer */
  for (ii=1;ii<=s1;ii++) {
    tmp2[ii][1]=-1;
    tmp2[ii][2]=1;
  }
  tmp2_row=s1;
  tmp2_col=2;
  if (strcmp(f2,"mytansig") ==0 ) {
    mynwtan(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);
    tmp3 = allocate_real_matrix(1,s2,1,2);
                        /* the output range of the 2nd layer */
    for (ii=1;ii<=s2;ii++) {
      tmp3[ii][1]=-1;
      tmp3[ii][2]=1;
    }
    tmp3_row=s2;
    tmp3_col=2;
    if (strcmp(f3,"mytansig") == 0) {
      mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mylogsig") == 0) {
      mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mypurelin") == 0) {
      myrands(w3,w3_row,w3_col,b3);
    }
    free_real_matrix(tmp2,1,tmp2_row,1);
    free_real_matrix(tmp3,1,tmp3_row,1);
  }

  if (strcmp(f2,"mylogsig") == 0) {
    mynwlog(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);
    tmp3 = allocate_real_matrix(1,s2,1,2);
    for (ii=1;ii<=s2;ii++) {
      tmp3[ii][1]=0;
      tmp3[ii][2]=1;
    }
    tmp3_row=s2;
    tmp3_col=2;
    if (strcmp(f3,"mylogsig") == 0) {
      mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mytansig") == 0) {
      mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mypurelin") == 0) {
      myrands(w3,w3_row,w3_col,b3);
    }
    free_real_matrix(tmp2,1,tmp2_row,1);
    free_real_matrix(tmp3,1,tmp3_row,1);
  }
```

```
    if (strcmp(f2,"mypurelin") == 0) {
      myrands(w2,w2_row,w2_col,b2);
      free_real_matrix(tmp2,1,tmp2_row,1);
    }
  }

}


/************** END of myinitff.c ***************/

/************** BEGIN of mytrainff.c ************/
/* This subroutine is for training a standard neural network;
   include mytbpx3 and mytbpx2 < * 2 hidden layers and 1 hidden layer case *>
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

void mytbpx3(float **w1, int w1_row, int w1_col, float *b1,
             float **w2, int w2_row, int w2_col, float *b2,
             float **w3, int w3_row, int w3_col, float *b3,
             float **p,  int p_row, int p_col,
             float **t,  int t_row, int t_col,
             float *tp,
             void (*f1)(float **w,int row,int col,float **x,
             int x_row,int x_col, float *b,float **a),

             void (*f2)(float **w,int row,int col,float **x,
             int x_row,int x_col, float *b,float **a),

             void (*f3)(float **w,int row,int col,float **x,
             int x_row,int x_col, float *b,float **a),
             void (*df1)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col,float **r),
             void (*df2)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col,float **r),
             void (*df3)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col,float **r)
            )

{

  float mysumsqr(float **e,int e_row,int e_col);
  void mylearnbpm(float **p,int p_row,int p_col,
                  float **d,int d_row,int d_col,
                  float lr,float mc,
                  float **dw,int dw_row,int dw_col,
                  float *db);
  float ** allocate_real_matrix(int,int,int,int);
  float * allocate_real_vector(int,int);

  int df,me; /* epoches for display, default = 25
                max number of epochs to train, default = 1000 */
  float eg,lr,im,dm,mc,er; /* sum-square error goal, default = 0.02;
                              learning rate, default = 0.01;
                              learning rate increse, default = 1.05;
                              learning rate decrease, default = 0.7;
                              momentum constant, default = 0.9;
                              maximum error ratio, default = 1.04     */
```

```
float MC;
float **dw1, **dw2, **dw3;
float *db1,*db2,*db3;

float **new_w1, **new_w2, **new_w3;
float *new_b1,*new_b2,*new_b3;

float **a1,**a2,**a3,**e;
float **new_a1,**new_a2,**new_a3,**new_e;

float **d1,**d2,**d3;

float SSE, new_SSE;

float **tr; /* training record */
int ii,jj,kk;

printf("Welcome to the neural net training program\n");
df=tp[1];
me=tp[2];
eg=tp[3];
lr=tp[4];
im=tp[5];
dm=tp[6];
mc=tp[7];
er=tp[8];

/* use memo_man.c in ~/PS_simu/Copt to allocate memory */
tr    =allocate_real_matrix(1,2,1,me+1);

dw1   =allocate_real_matrix(1, w1_row, 1, w1_col);
                        /* w1_row: number of neurons in the present layer  */
                        /* w1_col: number of neurons in the previous layer */
                        /* all others: with the same interpretation      */
new_w1=allocate_real_matrix(1, w1_row, 1, w1_col);
dw2   =allocate_real_matrix(1, w2_row, 1, w2_col);
new_w2=allocate_real_matrix(1, w2_row, 1, w2_col);
dw3   =allocate_real_matrix(1, w3_row, 1, w3_col);
new_w3=allocate_real_matrix(1, w3_row, 1, w3_col);

db1   =allocate_real_vector(1, w1_row);
new_b1=allocate_real_vector(1, w1_row);
db2   =allocate_real_vector(1, w2_row);
new_b2=allocate_real_vector(1, w2_row);
db3   =allocate_real_vector(1, w3_row);
new_b3=allocate_real_vector(1, w3_row);

for (ii=1;ii<=w1_row;ii++) {
  db1[ii]=0.0;
  for (jj=1;jj<=w1_col;jj++)
    dw1[ii][jj]=0.0;
}

for (ii=1;ii<=w2_row;ii++) {
  db2[ii]=0.0;
  for (jj=1;jj<=w2_col;jj++)
    dw2[ii][jj]=0.0;
}

for (ii=1;ii<=w3_row;ii++) {
  db3[ii]=0.0;
  for (jj=1;jj<=w3_col;jj++)
```

```
      dw3[ii][jj]=0.0;
  }

  MC=0;

  a1    =allocate_real_matrix(1, w1_row, 1, p_col);
  new_a1=allocate_real_matrix(1, w1_row, 1, p_col);
  a2    =allocate_real_matrix(1, w2_row, 1, p_col);
  new_a2=allocate_real_matrix(1, w2_row, 1, p_col);
  a3    =allocate_real_matrix(1, w3_row, 1, p_col);
  new_a3=allocate_real_matrix(1, w3_row, 1, p_col);

  d1    =allocate_real_matrix(1, w1_row, 1, p_col);
  d2    =allocate_real_matrix(1, w2_row, 1, p_col);
  d3    =allocate_real_matrix(1, w3_row, 1, p_col);
  e     =allocate_real_matrix(1, w3_row, 1, p_col);
  new_e =allocate_real_matrix(1, w3_row, 1, p_col);

  printf("Memory allocation ready\n");

/*********** Presentation Phase ****************************/
  (*f1)(w1,w1_row,w1_col,p,p_row,p_col,b1,a1);
      /* note: a1 is a matrix with dimension w1_row x p_col */
  (*f2)(w2,w2_row,w2_col,a1,w1_row,p_col,b2,a2);
      /* note: a2 is a matrix with dimension w2_row x p_col */
  (*f3)(w3,w3_row,w3_col,a2,w2_row,p_col,b3,a3);
      /* note: a3 is a matrix with dimension w3_row x p_col */

  for (ii=1;ii<=w3_row;ii++)
    for (jj=1;jj<=p_col;jj++)
      e[ii][jj]=t[ii][jj]-a3[ii][jj];

  SSE = mysumsqr(e,w3_row,p_col);
  printf("Presentation Phase finished\n");
  printf("Initial SSE=%f\n",SSE);

/*********** BackPropagation Phase *************************/
  (*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL,d3);
                                        /* e, a3, d3 are same dimensional */
  printf("Any problem with NULL (pointer) use?\n");
  (*df2)(a2,w2_row,p_col,d3,w3_row,p_col,w3,w3_row,w3_col,d2);
                                        /* d2,a2 same dimension */
  printf("Any problem with df2?\n");
  (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col,d1);
                                        /* d1,a1 same dimension */

  printf("Error calculations ready\n");
  for (ii=1;ii<=me;ii++) {
    if (SSE < eg) { /* CHECK PHASE */
      ii--;
      break;
    }

    /* LEARNING PHASE */
    mylearnbpm(p,p_row,p_col,d1,w1_row,p_col,lr,MC,dw1,w1_row,w1_col,db1);
    mylearnbpm(a1,w1_row,p_col,d2,w2_row,p_col,lr,MC,dw2,w2_row,w2_col,db2);
    mylearnbpm(a2,w2_row,p_col,d3,w3_row,p_col,lr,MC,dw3,w3_row,w3_col,db3);

    MC=mc;
    for (jj=1;jj<=w1_row;jj++) { /* updating w1,b1 */
      new_b1[jj]=b1[jj]+db1[jj];
      for (kk=1;kk<=w1_col;kk++)
```

```
    new_w1[jj][kk]=w1[jj][kk]+dw1[jj][kk];
}

for (jj=1;jj<=w2_row;jj++) { /* updating w2,b2 */
  new_b2[jj]=b2[jj]+db2[jj];
  for (kk=1;kk<=w2_col;kk++)
    new_w2[jj][kk]=w2[jj][kk]+dw2[jj][kk];
}

for (jj=1;jj<=w3_row;jj++) { /* updating w3,b3 */
  new_b3[jj]=b3[jj]+db3[jj];
  for (kk=1;kk<=w3_col;kk++)
    new_w3[jj][kk]=w3[jj][kk]+dw3[jj][kk];
}

/* PRESENTATION PHASE */
(*f1)(new_w1,w1_row,w1_col,p,p_row,p_col,new_b1,new_a1);
  /* note: new_a1 (as a1) is a matrix with dimension w1_row x p_col */
(*f2)(new_w2,w2_row,w2_col,new_a1,w1_row,p_col,new_b2,new_a2);
  /* note: new_a2 (as a2) is a matrix with dimension w2_row x p_col */
(*f3)(new_w3,w3_row,w3_col,new_a2,w2_row,p_col,new_b3,new_a3);
  /* note: new_a3 (as a3) is a matrix with dimension w3_row x p_col */

for (jj=1;jj<=w3_row;jj++)
  for (kk=1;kk<=p_col;kk++)
    new_e[jj][kk]=t[jj][kk]-new_a3[jj][kk];


new_SSE=mysumsqr(new_e,w3_row,p_col);
                      /* new_e (as e) with dimension w3_row x p_col */

/* Momentum and adaptive learning rate phase */
if (new_SSE > SSE * er) {
  lr=lr*dm;
  MC=0;
}
else {
  if (new_SSE < SSE) {
    lr=lr*im;
  }
  for (jj=1;jj<=w1_row;jj++) {
    b1[jj]=new_b1[jj];
    for (kk=1;kk<=w1_col;kk++) {
      w1[jj][kk]=new_w1[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a1[jj][kk]=new_a1[jj][kk];
    }
  }
  for (jj=1;jj<=w2_row;jj++) {
    b2[jj]=new_b2[jj];
    for (kk=1;kk<=w2_col;kk++) {
      w2[jj][kk]=new_w2[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a2[jj][kk]=new_a2[jj][kk];
    }
  }
  for (jj=1;jj<=w3_row;jj++) {
    b3[jj]=new_b3[jj];
    for (kk=1;kk<=w3_col;kk++) {
      w3[jj][kk]=new_w3[jj][kk];
    }
```

```
        for (kk=1;kk<=p_col;kk++) {
          a3[jj][kk]=new_a3[jj][kk];
          e[jj][kk] =new_e[jj][kk];
        }
      }

      SSE=new_SSE;

      /* BACKPROPAGATION PHASE */
      (*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL,d3);
                                          /* e, a3, d3 are same dimensional */
      (*df2)(a2,w2_row,p_col,d3,w3_row,p_col,w3,w3_row,w3_col,d2);
                                          /* d2,a2 same dimension */
      (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col,d1);
                                          /* d1,a1 same dimension */
    }

    /* TRAINING RECORDS */
    tr[1][ii+1]=SSE;
    tr[2][ii+1]=lr;

    if ((ii % df) == 0)
      printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);

  } /* end ii --- "for" loop */

  if ((ii % df) !=0) {/* This is for last training epoch printing
                         in case me is not multiples of df        */
    printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);
  }

  if ( SSE > eg) {
    printf("Tranibpx: network error did not reach the error goal;\n");
    printf("Future training may be necessary or try different    \n");
    printf("initial weights and biases and/or more hidden neurons.\n");
  }
}

/*********** End of mytbpx3 ***********/

void mytbpx2(float **w1, int w1_row, int w1_col, float *b1,
             float **w2, int w2_row, int w2_col, float *b2,
             float **p,  int p_row, int p_col,
             float **t,  int t_row, int t_col,
             float *tp,
             void (*f1)(float **w,int row,int col,float **x,
             int x_row,int x_col, float *b,float **a),

             void (*f2)(float **w,int row,int col,float **x,
             int x_row,int x_col, float *b,float **a),

             void (*df1)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col,float **r),
             void (*df2)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col,float **r)
            )

{

  float mysumsqr(float **e,int e_row,int e_col);
  void mylearnbpm(float **p,int p_row,int p_col,
                  float **d,int d_row,int d_col,
                  float lr,float mc,
```

```
                float **dw,int dw_row,int dw_col,
                float *db);
float ** allocate_real_matrix(int,int,int,int);
float * allocate_real_vector(int,int);

int df,me; /* epoches for display, default = 25
              max number of epochs to train, default = 1000 */
float eg,lr,im,dm,mc,er; /* sum-square error goal, default = 0.02;
                            learning rate, default = 0.01;
                            learning rate increse, default = 1.05;
                            learning rate decrease, default = 0.7;
                            momentum constant, default = 0.9;
                            maximum error ratio, default = 1.04     */

float MC;
float **dw1, **dw2;
float *db1,*db2;

float **new_w1, **new_w2;
float *new_b1,*new_b2;

float **a1,**a2,**e;
float **new_a1,**new_a2,**new_e;

float **d1,**d2;

float SSE, new_SSE;

float **tr; /* training record */
int ii,jj,kk;

printf("Welcome to the neural net training program\n");
df=tp[1];
me=tp[2];
eg=tp[3];
lr=tp[4];
im=tp[5];
dm=tp[6];
mc=tp[7];
er=tp[8];

/* use memo_man.c in ~/PS_simu/Copt to allocate memory */
tr    =allocate_real_matrix(1,2,1,me+1);

dw1   =allocate_real_matrix(1, w1_row, 1, w1_col);
                        /* w1_row: number of neurons in the present layer  */
                        /* w1_col: number of neurons in the previous layer */
                        /* all others: with the same interpretation      */
new_w1=allocate_real_matrix(1, w1_row, 1, w1_col);
dw2   =allocate_real_matrix(1, w2_row, 1, w2_col);
new_w2=allocate_real_matrix(1, w2_row, 1, w2_col);

db1   =allocate_real_vector(1, w1_row);
new_b1=allocate_real_vector(1, w1_row);
db2   =allocate_real_vector(1, w2_row);
new_b2=allocate_real_vector(1, w2_row);

for (ii=1;ii<=w1_row;ii++) {
  db1[ii]=0.0;
  for (jj=1;jj<=w1_col;jj++)
    dw1[ii][jj]=0.0;
}
```

```
  for (ii=1;ii<=w2_row;ii++) {
    db2[ii]=0.0;
    for (jj=1;jj<=w2_col;jj++)
      dw2[ii][jj]=0.0;
  }

  MC=0;

  a1    =allocate_real_matrix(1, w1_row, 1, p_col);
  new_a1=allocate_real_matrix(1, w1_row, 1, p_col);
  a2    =allocate_real_matrix(1, w2_row, 1, p_col);
  new_a2=allocate_real_matrix(1, w2_row, 1, p_col);

  d1    =allocate_real_matrix(1, w1_row, 1, p_col);
  d2    =allocate_real_matrix(1, w2_row, 1, p_col);
  e     =allocate_real_matrix(1, w2_row, 1, p_col);
  new_e =allocate_real_matrix(1, w2_row, 1, p_col);

  printf("Memory allocation ready\n");

/********** Presentation Phase ***************************/
  (*f1)(w1,w1_row,w1_col,p,p_row,p_col,b1,a1);
      /* note: a1 is a matrix with dimension w1_row x p_col */
  (*f2)(w2,w2_row,w2_col,a1,w1_row,p_col,b2,a2);
      /* note: a2 is a matrix with dimension w2_row x p_col */

  for (ii=1;ii<=w2_row;ii++)
    for (jj=1;jj<=p_col;jj++)
      e[ii][jj]=t[ii][jj]-a2[ii][jj];

  SSE = mysumsqr(e,w2_row,p_col);
  printf("Presentation Phase finished\n");
  printf("Initial SSE=%f\n",SSE);

/********** BackPropagation Phase ************************/
  (*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL,d2);
                                        /* e, a3, d3 are same dimensional */
  (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col,d1);
                                        /* d1,a1 same dimension */

  for (ii=1;ii<=me;ii++) {
    if (SSE < eg) { /* CHECK PHASE */
      ii--;
      break;
    }

    /* LEARNING PHASE */
    mylearnbpm(p,p_row,p_col,d1,w1_row,p_col,lr,MC,dw1,w1_row,w1_col,db1);
    mylearnbpm(a1,w1_row,p_col,d2,w2_row,p_col,lr,MC,dw2,w2_row,w2_col,db2);

    MC=mc;
    for (jj=1;jj<=w1_row;jj++) { /* updating w1,b1 */
      new_b1[jj]=b1[jj]+db1[jj];
      for (kk=1;kk<=w1_col;kk++)
        new_w1[jj][kk]=w1[jj][kk]+dw1[jj][kk];
    }

    for (jj=1;jj<=w2_row;jj++) { /* updating w2,b2 */
      new_b2[jj]=b2[jj]+db2[jj];
      for (kk=1;kk<=w2_col;kk++)
        new_w2[jj][kk]=w2[jj][kk]+dw2[jj][kk];
    }
```

```
  /* PRESENTATION PHASE */
  (*f1)(new_w1,w1_row,w1_col,p,p_row,p_col,new_b1,new_a1);
    /* note: new_a1 (as a1) is a matrix with dimension w1_row x p_col */
  (*f2)(new_w2,w2_row,w2_col,new_a1,w1_row,p_col,new_b2,new_a2);
    /* note: new_a2 (as a2) is a matrix with dimension w2_row x p_col */

  for (jj=1;jj<=w2_row;jj++)
    for (kk=1;kk<=p_col;kk++)
      new_e[jj][kk]=t[jj][kk]-new_a2[jj][kk];


  new_SSE=mysumsqr(new_e,w2_row,p_col);
                        /* new_e (as e) with dimension w3_row x p_col */

  /* Momentum and adaptive learning rate phase */
  if (new_SSE > SSE * er) {
    lr=lr*dm;
    MC=0;
  }
  else {
    if (new_SSE < SSE) {
      lr=lr*im;
    }
    for (jj=1;jj<=w1_row;jj++) {
      b1[jj]=new_b1[jj];
      for (kk=1;kk<=w1_col;kk++) {
        w1[jj][kk]=new_w1[jj][kk];
      }
      for (kk=1;kk<=p_col;kk++) {
        a1[jj][kk]=new_a1[jj][kk];
      }
    }
    for (jj=1;jj<=w2_row;jj++) {
      b2[jj]=new_b2[jj];
      for (kk=1;kk<=w2_col;kk++) {
        w2[jj][kk]=new_w2[jj][kk];
      }
      for (kk=1;kk<=p_col;kk++) {
        a2[jj][kk]=new_a2[jj][kk];
        e[jj][kk] =new_e[jj][kk];
      }
    }

    SSE=new_SSE;

    /* BACKPROPAGATION PHASE */
    (*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL,d2);
                                        /* e, a3, d3 are same dimensional */
    (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col,d1);
                                        /* d1,a1 same dimension */
  }

  /* TRAINING RECORDS */
  tr[1][ii+1]=SSE;
  tr[2][ii+1]=lr;

  if ((ii % df) == 0)
    printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);

} /* end ii --- "for" loop */

if ((ii % df) !=0) {/* This is for last training epoch printing
```

```
                    in case me is not multiples of df        */
    printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);
  }

  if ( SSE > eg) {
    printf("Tranibpx: network error did not reach the error goal;\n");
    printf("Future training may be necessary or try different    \n");
    printf("initial weights and biases and/or more hidden neurons.\n");
  }
}


/*************** END of mytbpx2 ***********************/

/*************** END of mytrainff.c *******************/

/*************** BEGIN of mysimuff.c ******************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

void mysimuff(float **p, int p_row, int p_col,
              float **w1, int w1_row, int w1_col, float *b1,
              float **w2, int w2_row, int w2_col, float *b2,
              float **w3, int w3_row, int w3_col, float *b3,
              void (*f1)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*f2)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*f3)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              float **a1, int a1_row, int a1_col,
              float **a2, int a2_row, int a2_col,
              float **a3, int a3_row, int a3_col
             )

{
  int ii,jj;
  float **A1,**A2,**A3;

  /* Dimension consistency check */
  if ((a3) == NULL) { /* only 1 hidden layer case */
    if (w1_col!=p_row || w2_col!=w1_row) {
      printf("Dimension inconsistency in mysimuff\n");
      printf("Exiting from mysimuff\n");
      exit(1);
    }
    if (a1_row!=w1_row || a1_col!=p_col || a2_row!=w2_row || a2_col!=p_col) {
      printf("Dimension inconsistency in mysimuff\n");
      printf("Exiting from mysimuff\n");
      exit(1);
    }
    A1=allocate_real_matrix(1,a1_row,1,a1_col);
    A2=allocate_real_matrix(1,a2_row,1,a2_col);
    (*f1)(w1,w1_row,w1_col,p,p_row,p_col,b1,A1);
    (*f2)(w2,w2_row,w2_col,A1,w1_row,p_col,b2,A2);
         /* Note:_____________!!!_____________________________*/
         /* Note here: a1,a2 are float ***, so need use (*a1) and (*a2)
            to assure the consistence of the pointer type */
    for (ii=1;ii<=a1_row;ii++)
```

```
      for (jj=1;jj<=a1_col;jj++)
        a1[ii][jj]=A1[ii][jj];

     for (ii=1;ii<=a2_row;ii++)
      for (jj=1;jj<=a2_col;jj++)
        a2[ii][jj]=A2[ii][jj];

  }

  if ((a3) != NULL) { /* 2 hidden layer case */
    if (w1_col!=p_row || w2_col!=w1_row || w3_col!=w2_row) {
      printf("Dimension inconsistency in mysimuff\n");
      printf("Exiting from mysimuff\n");
      exit(1);
    }
    if (a1_row!=w1_row || a1_col!=p_col || a2_row!=w2_row ||
        a2_col!=p_col || a3_row!=w3_row || a3_col!=p_col) {
      printf("Dimension inconsistency in mysimuff\n");
      printf("Exiting from mysimuff\n");
      exit(1);
    }

    A1=allocate_real_matrix(1,a1_row,1,a1_col);
    A2=allocate_real_matrix(1,a2_row,1,a2_col);
    A3=allocate_real_matrix(1,a3_row,1,a3_col);

    (*f1)(w1,w1_row,w1_col,p,p_row,p_col,b1,A1);
                                /* note: a1: w1_row x p_col */
    (*f2)(w2,w2_row,w2_col,A1,w1_row,p_col,b2,A2);
                                /* note: a2: w2_row x p_col */
    (*f3)(w3,w3_row,w3_col,A2,w2_row,p_col,b3,A3);
                                /* note: a3: w3_row x p_col */

    for (ii=1;ii<=a1_row;ii++)
      for (jj=1;jj<=a1_col;jj++)
        a1[ii][jj]=A1[ii][jj];

     for (ii=1;ii<=a2_row;ii++)
      for (jj=1;jj<=a2_col;jj++)
        a2[ii][jj]=A2[ii][jj];

    for (ii=1;ii<=a3_row;ii++)
      for (jj=1;jj<=a3_col;jj++)
        a3[ii][jj]=A3[ii][jj];
    /*
    for (ii=1;ii<=a3_row;ii++)
      for (jj=1;jj<=a3_col;jj++)
        printf("%f\n",(a3)[ii][jj]);
    */
  }
  /*return ;*/
}

/*************** END of mysimuff.c ******************/

/*************** BEGIN of myinitelm.c ***************/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mynn.h"

void myinitelm(float ** p, int p_row, int p_col,
               int s1, char *f1, int local_recur1,
```

```
                                /* 1/0: yes/no local recurr */
                int s2, char *f2, int local_recur2,
                                /* 1/0: yes/no local recurr */
                int s3, char *f3, int local_recur3,
                                /* 1/0: yes/no local recurr */
                float ** w1, int w1_row, int w1_col, float * b1,
                float ** w2, int w2_row, int w2_col, float * b2,
                float ** w3, int w3_row, int w3_col, float * b3)

{
  int ii,jj;
  float **tmp1,**tmp2,**tmp3;
  int tmp1_row,tmp1_col,tmp2_row,tmp2_col,tmp3_row,tmp3_col;

  tmp1_row=p_row+s1*local_recur1;
  tmp1_col=2;
  tmp1 = allocate_real_matrix(1,p_row+s1*local_recur1,1,2);
                                          /* convert matrix p into tmp1
                                             so that it may accomodate
                                             the recurrent part from the
                                             output of the 1st hidden layer
                                             if applicable */

  for (ii=1;ii<=p_row;ii++) {
    tmp1[ii][1]=p[ii][1];
    tmp1[ii][2]=p[ii][1];
    for (jj=2;jj<=p_col;jj++) {
      if (p[ii][jj] < tmp1[ii][1])
        tmp1[ii][1]=p[ii][jj];
      if (p[ii][jj] > tmp1[ii][2])
        tmp1[ii][2]=p[ii][jj];
    }
  }

  /* First of all, dimension consistency check */
  if (strcmp(f2,"mypurelin") ==0) {/* only 1 hidden layer case */
    if (w1_col != (p_row+s1*local_recur1) || w2_col != (w1_row+s2*local_recur2)
       || s1 !=w1_row || s2 !=w2_row) {
      printf("Dimension inconsistency error in myinitff\n");
      printf("Exiting from myinitff now.");
      exit(1);
    }
  }

  if (strcmp(f3,"mypurelin") ==0) {/* only 2 hidden layer case */
    if (w1_col!=(p_row+s1*local_recur1) || w2_col!=(w1_row+s2*local_recur2)
        || w3_col!=(w2_row+s3*local_recur3)
        || s1!=w1_row || s2!=w2_row || s3!=w3_row) {
      printf("Dimension inconsistency error in myinitff\n");
      printf("Exiting from myinitff now.");
      exit(1);
    }
  }

  printf("Dimension check ok\n");

/************ Read the following lines carefully and make modifications ***/
  /********* Case 1: first layer mylogsig **********/
  if (strcmp(f1,"mylogsig") == 0) {
    if (local_recur1 == 1) {
      for (ii=1;ii<=s1;ii++) {
        tmp1[p_row+ii][1]=0;
```

```
    tmp1[p_row+ii][2]=1;
  }
}
/*if (local_recur1 == 0) ; doing nothing; doing-so to remind logic */

mynwlog(s1,tmp1,tmp1_row,tmp1_col,w1,w1_row,w1_col,b1);
tmp2 = allocate_real_matrix(1,s1+s2*local_recur2,1,2);
                          /* the output range of the 2nd layer */
for (ii=1;ii<=s1;ii++) {
  tmp2[ii][1]=0;
  tmp2[ii][2]=1;
}
tmp2_row=s1+s2*local_recur2;
tmp2_col=2;
if (strcmp(f2,"mylogsig") ==0 ) {
  if (local_recur2 == 1) {
    for (ii=1;ii<=s2;ii++) {
      tmp2[s1+ii][1]=0;
      tmp2[s1+ii][2]=1;
    }
  }
  /*if (local_recur2 == 0) ; doing nothing; doing-so to remind logic */
  mynwlog(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);

  tmp3 = allocate_real_matrix(1,s2+s3*local_recur3,1,2);
                          /* the output range of the 2nd layer */
  for (ii=1;ii<=s2;ii++) {
    tmp3[ii][1]=0;
    tmp3[ii][2]=1;
  }
  tmp3_row=s2+s3*local_recur3;
  tmp3_col=2;
  if (strcmp(f3,"mylogsig") == 0) {
    if (local_recur3 == 1) {
      for (ii=1;ii<=s3;ii++) {
        tmp3[s2+ii][1]=0;
        tmp3[s2+ii][2]=1;
      }
    }
    /*if (local_recur3 == 0) ; doing nothing; doing-so to remind logic */
    mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
  }
  if (strcmp(f3,"mytansig") == 0) {
    if (local_recur3 == 1) {
      for (ii=1;ii<=s3;ii++) {
        tmp3[s2+ii][1]=-1;
        tmp3[s2+ii][2]=1;
      }
    }
    /*if (local_recur3 == 0) ; doing nothing; doing-so to remind logic */

    mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
  }
  if (strcmp(f3,"mypurelin") == 0) {

    myrands(w3,w3_row,w3_col,b3);
  }
  free_real_matrix(tmp1,1,tmp1_row,1);
  free_real_matrix(tmp2,1,tmp2_row,1);
  free_real_matrix(tmp3,1,tmp3_row,1);
}
```

```
  if (strcmp(f2,"mytansig") == 0) {
    if (local_recur2 == 1) {
      for (ii=1;ii<=s2;ii++) {
        tmp2[s1+ii][1]=-1;
        tmp2[s1+ii][2]=1;
      }
    }
    /*if (local_recur2 == 0) ;  doing nothing; doing-so to remind logic */
    mynwtan(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);

    tmp3 = allocate_real_matrix(1,s2+s3*local_recur3,1,2);
    for (ii=1;ii<=s2;ii++) {
      tmp3[ii][1]=-1;
      tmp3[ii][2]=1;
    }
    tmp3_row=s2+s3*local_recur3;
    tmp3_col=2;
    if (strcmp(f3,"mylogsig") == 0) {
      if (local_recur3 == 1) {
        for (ii=1;ii<=s3;ii++) {
          tmp3[s2+ii][1]=0;
          tmp3[s2+ii][2]=1;
        }
      }
      /*if (local_recur3 == 0) ; doing nothing; doing-so to remind logic */
      mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mytansig") == 0) {
      if (local_recur3 == 1) {
        for (ii=1;ii<=s3;ii++) {
          tmp3[s2+ii][1]=-1;
          tmp3[s2+ii][2]=1;
        }
      }
      /*if (local_recur3 == 0) ; doing nothing; doing-so to remind logic */
      mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
    }
    if (strcmp(f3,"mypurelin") == 0) {
      myrands(w3,w3_row,w3_col,b3);
    }
    free_real_matrix(tmp1,1,tmp1_row,1);
    free_real_matrix(tmp2,1,tmp2_row,1);
    free_real_matrix(tmp3,1,tmp3_row,1);
  }

  if (strcmp(f2,"mypurelin") == 0) {
                      /* if layer 2 is linear layer, then no more layers */
    myrands(w2,w2_row,w2_col,b2);
    free_real_matrix(tmp1,1,tmp1_row,1);
    free_real_matrix(tmp2,1,tmp2_row,1);
  }

}

/***** Case 2: first layer mytansig *********/
if (strcmp(f1,"mytansig") == 0) {
  if (local_recur1 == 1) {
    for (ii=1;ii<=s1;ii++) {
      tmp1[p_row+ii][1]=0;
      tmp1[p_row+ii][2]=1;
    }
  }
```

```
/* if (local_recur1 == 0) ;  doing nothing; doing-so to remind logic */

printf("ready to do mynwtan\n");
mynwtan(s1,tmp1,tmp1_row,tmp1_col,w1,w1_row,w1_col,b1);

printf("mynwtan ok\n");

tmp2 = allocate_real_matrix(1,s1+s2*local_recur2,1,2);
                          /* the output range of the 1st layer */
for (ii=1;ii<=s1;ii++) {
  tmp2[ii][1]=-1;
  tmp2[ii][2]=1;
}
tmp2_row=s1+s2*local_recur2;
tmp2_col=2;
if (strcmp(f2,"mytansig") ==0 ) {
  if (local_recur2 == 1) {
    for (ii=1;ii<=s2;ii++) {
      tmp2[s1+ii][1]=-1;
      tmp2[s1+ii][2]=1;
    }
  }
  /*if (local_recur2 == 0) ; doing nothing; doing-so to remind logic */
  mynwtan(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);


  tmp3 = allocate_real_matrix(1,s2+s3*local_recur3,1,2);
                            /* the output range of the 2nd layer */
  for (ii=1;ii<=s2;ii++) {
    tmp3[ii][1]=-1;
    tmp3[ii][2]=1;
  }
  tmp3_row=s2+s3*local_recur3;
  tmp3_col=2;
  if (strcmp(f3,"mytansig") == 0) {
    if (local_recur3 == 1) {
      for (ii=1;ii<=s3;ii++) {
        tmp3[s2+ii][1]=-1;
        tmp3[s2+ii][2]=1;
      }
    }
    /*if (local_recur3 == 0) ; doing nothing; doing-so to remind logic */
    mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
  }
  if (strcmp(f3,"mylogsig") == 0) {
    if (local_recur3 == 1) {
      for (ii=1;ii<=s3;ii++) {
        tmp3[s2+ii][1]=0;
        tmp3[s2+ii][2]=1;
      }
    }
    /*if (local_recur3 == 0) ; doing nothing; doing-so to remind logic */
    mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
  }
  if (strcmp(f3,"mypurelin") == 0) {

    myrands(w3,w3_row,w3_col,b3);
  }
  free_real_matrix(tmp1,1,tmp1_row,1);
  free_real_matrix(tmp2,1,tmp2_row,1);
  free_real_matrix(tmp3,1,tmp3_row,1);
}
```

```
    if (strcmp(f2,"mylogsig") == 0) {
      if (local_recur2 == 1) {
        for (ii=1;ii<=s2;ii++) {
          tmp2[s1+ii][1]=0;
          tmp2[s1+ii][2]=1;
        }
      }
      /* if (local_recur2 == 0) ;  doing nothing; doing-so to remind logic */

      printf("ready to do mynwlog\n");
      mynwlog(s2,tmp2,tmp2_row,tmp2_col,w2,w2_row,w2_col,b2);

      printf("mynwlog ok\n");

      tmp3 = allocate_real_matrix(1,s2+s3*local_recur3,1,2);
      for (ii=1;ii<=s2;ii++) {
        tmp3[ii][1]=0;
        tmp3[ii][2]=1;
      }
      tmp3_row=s2+s3*local_recur3;
      tmp3_col=2;
      if (strcmp(f3,"mylogsig") == 0) {
        if (local_recur3 == 1) {
          for (ii=1;ii<=s3;ii++) {
            tmp3[s2+ii][1]=0;
            tmp3[s2+ii][2]=1;
          }
        }
        /* if (local_recur3 == 0) ;  doing nothing; doing-so to remind logic */
        mynwlog(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
      }
      if (strcmp(f3,"mytansig") == 0) {
        if (local_recur3 == 1) {
          for (ii=1;ii<=s3;ii++) {
            tmp3[s2+ii][1]=-1;
            tmp3[s2+ii][2]=1;
          }
        }
        /*if (local_recur3 == 0) ;  doing nothing; doing-so to remind logic */
        mynwtan(s3,tmp3,tmp3_row,tmp3_col,w3,w3_row,w3_col,b3);
      }
      if (strcmp(f3,"mypurelin") == 0) {
        myrands(w3,w3_row,w3_col,b3);
        printf("myrands for mypurelin ok\n");
      }
      free_real_matrix(tmp1,1,tmp1_row,1);
      free_real_matrix(tmp2,1,tmp2_row,1);
      free_real_matrix(tmp3,1,tmp3_row,1);
    }

    if (strcmp(f2,"mypurelin") == 0) {
      myrands(w2,w2_row,w2_col,b2);
      free_real_matrix(tmp1,1,tmp1_row,1);
      free_real_matrix(tmp2,1,tmp2_row,1);
    }
  }
/************* END of case 2 ******************/

}

/************* END of mysimuelm.c ******************/
```

```
/************* BEGIN of mytrainelm.c ***************/

/* This subroutine is used to train a recurrent nerual network */
/* Using standard backpropagation, not dynamic backpropagation */
/* local recurrency not global recurrency                      */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

/* Train Recurrent using BackPropagation with fast algorithms */
/* thus,... ===> trbpx3                                       */
void mytrbpx3(float **w1, int w1_row, int w1_col, float *b1, int local_recur1,
              float **w2, int w2_row, int w2_col, float *b2, int local_recur2,
              float **w3, int w3_row, int w3_col, float *b3, int local_recur3,
              float **p,  int p_row, int p_col,
              float **t,  int t_row, int t_col,
              float *tp,
              /*float ** (*f1)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b),
              float ** (*f2)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b),
              float ** (*f3)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b),
              float ** (*df1)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col),
              float ** (*df2)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col),
              float ** (*df3)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col)
              */
              void (*f1)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*f2)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*f3)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*df1)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col,float **r),
              void (*df2)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col,float **r),
              void (*df3)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col,float **r)
             )
{
  int df,me; /* epoches for display, default = 25
                max number of epochs to train, default = 1000 */
  float eg,lr,im,dm,mc,er; /* sum-square error goal, default = 0.02;
                              learning rate, default = 0.01;
                              learning rate increse, default = 1.05;
                              learning rate decrease, default = 0.7;
                              momentum constant, default = 0.9;
                              maximum error ratio, default = 1.04     */

  float MC;
  float **dw1, **dw2, **dw3;
  float *db1,*db2,*db3;

  float **new_w1, **new_w2, **new_w3;
  float *new_b1,*new_b2,*new_b3;

  float **a1,**a2,**a3,**e;
```

```
float **new_a1,**new_a2,**new_a3,**new_e;

float **d1,**d2,**d3;

float SSE, new_SSE;

float **tr; /* training record */

float **tmp_p ,**tmp_a1,**tmp_a2;
                        /* in case of local recurrency,temp storage */
float **tmp_w2,**tmp_w3; /* local recurrency, part of w2, w3 */

int ii,jj,kk,mm;
float **tmp_new_p,**tmp_new_a1,**tmp_new_a2,**tmp_new_a3;
float **tmp_tmp_new_a1,**tmp_tmp_new_a2;

tmp_new_p=allocate_real_matrix(1,p_row+w1_row*local_recur1,1,1);
tmp_new_a1=allocate_real_matrix(1,w1_row,1,1);
tmp_new_a2=allocate_real_matrix(1,w2_row,1,1);
tmp_tmp_new_a1=allocate_real_matrix(1,w1_row+w2_row*local_recur2,1,1);
tmp_tmp_new_a2=allocate_real_matrix(1,w2_row+w3_row*local_recur3,1,1);
tmp_new_a3=allocate_real_matrix(1,w3_row,1,1);

if (   w1_col!=( p_row+w1_row*local_recur1)
    || w2_col!=(w1_row+w2_row*local_recur2)
    || w3_col!=(w2_row+w3_row*local_recur3)) {
  printf("Dimension inconsistency in mytrainelm\n");
  printf("Exiting from mytrainelm\n");
  exit(1);
} /* Note: the number of neurons is the same
     as that of rows for the weight matrix.
     ===> # neurons in layer 1 = w1_row; etc */

printf("Welcome to the Recurrent Neural Net training program\n");
df=tp[1];
me=tp[2];
eg=tp[3];
lr=tp[4];
im=tp[5];
dm=tp[6];
mc=tp[7];
er=tp[8];

/* use memo_man.c in ~/PS_simu/Copt to allocate memory */
tr    =allocate_real_matrix(1,2,1,me+1);

dw1   =allocate_real_matrix(1, w1_row, 1, w1_col);
                        /* w1_row: number of neurons in the present layer  */
                        /* w1_col: number of neurons in the previous layer */
                        /* all others: with the same interpretation        */
new_w1=allocate_real_matrix(1, w1_row, 1, w1_col);
dw2   =allocate_real_matrix(1, w2_row, 1, w2_col);
new_w2=allocate_real_matrix(1, w2_row, 1, w2_col);
dw3   =allocate_real_matrix(1, w3_row, 1, w3_col);
new_w3=allocate_real_matrix(1, w3_row, 1, w3_col);

db1   =allocate_real_vector(1, w1_row);
new_b1=allocate_real_vector(1, w1_row);
db2   =allocate_real_vector(1, w2_row);
new_b2=allocate_real_vector(1, w2_row);
db3   =allocate_real_vector(1, w3_row);
new_b3=allocate_real_vector(1, w3_row);
```

```
  for (ii=1;ii<=w1_row;ii++) {
    db1[ii]=0.0;
    for (jj=1;jj<=w1_col;jj++)
      dw1[ii][jj]=0.0;
  }

  for (ii=1;ii<=w2_row;ii++) {
    db2[ii]=0.0;
    for (jj=1;jj<=w2_col;jj++)
      dw2[ii][jj]=0.0;
  }

  for (ii=1;ii<=w3_row;ii++) {
    db3[ii]=0.0;
    for (jj=1;jj<=w3_col;jj++)
      dw3[ii][jj]=0.0;
  }

  MC=0;

  a1    =allocate_real_matrix(1, w1_row, 1, p_col);
  new_a1=allocate_real_matrix(1, w1_row, 1, p_col);
  a2    =allocate_real_matrix(1, w2_row, 1, p_col);
  new_a2=allocate_real_matrix(1, w2_row, 1, p_col);
  a3    =allocate_real_matrix(1, w3_row, 1, p_col);
  new_a3=allocate_real_matrix(1, w3_row, 1, p_col);

  d1    =allocate_real_matrix(1, w1_row, 1, p_col);
  d2    =allocate_real_matrix(1, w2_row, 1, p_col);
  d3    =allocate_real_matrix(1, w3_row, 1, p_col);
  e     =allocate_real_matrix(1, w3_row, 1, p_col);
  new_e =allocate_real_matrix(1, w3_row, 1, p_col);

/*********** Presentation Phase ****************************/

  mysimuelm(p,  p_row,  p_col,
            w1, w1_row, w1_col, b1, local_recur1,/* yes/no: 1/0 */
            w2, w2_row, w2_col, b2, local_recur2,/* yes/no: 1/0 */
            w3, w3_row, w3_col, b3, local_recur3,/* yes/no: 1/0 */
            (*f1),
            (*f2),
            (*f3),
            a1, w1_row, p_col, /* a1_row=w1_row; a1_col=p_col; */
            a2, w2_row, p_col, /* a2_row=w2_row; a2_col=p_col; */
            a3, w3_row, p_col  /* a3_row=w3_row; a3_col=p_col; */
           );

  for (ii=1;ii<=w3_row;ii++)
    for (jj=1;jj<=p_col;jj++)
      e[ii][jj]=t[ii][jj]-a3[ii][jj];

  SSE = mysumsqr(e,w3_row,p_col);

  tmp_w3=allocate_real_matrix(1,w3_row,1,w3_col-w3_row*local_recur3);
  tmp_w2=allocate_real_matrix(1,w2_row,1,w2_col-w2_row*local_recur2);
  tmp_p =allocate_real_matrix(1, p_row+w1_row*local_recur1,1,p_col);
  tmp_a1=allocate_real_matrix(1,w1_row+w2_row*local_recur2,1,p_col);
  tmp_a2=allocate_real_matrix(1,w2_row+w3_row*local_recur3,1,p_col);

  printf("Initial SSE=%f\n",SSE);

/*********** BackPropagation Phase *************************/
```

```
/* ALL the following lines must be modified, which are copied from mytbpx3.c */
  for (jj=1;jj<=w3_row;jj++)
    for (kk=1;kk<=(w3_col-w3_row*local_recur3);kk++)
      tmp_w3[jj][kk]=w3[jj][kk];

  for (jj=1;jj<=w2_row;jj++)
    for (kk=1;kk<=(w2_col-w2_row*local_recur2);kk++)
      tmp_w2[jj][kk]=w2[jj][kk];
  /*
  d3=(*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL);
  d2=(*df2)(a2,w2_row,p_col,d3,w3_row,p_col,tmp_w3,w3_row,
            w3_col-w3_row*local_recur3);
  d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
            w2_col-w2_row*local_recur2);
  */
  (*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL,d3);
  (*df2)(a2,w2_row,p_col,d3,w3_row,p_col,tmp_w3,w3_row,
         w3_col-w3_row*local_recur3,d2);
  (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
         w2_col-w2_row*local_recur2,d1);


  /*** Training LOOP begins ***/
  for (ii=1;ii<=me;ii++) {
    if (SSE < eg) { /* CHECK PHASE */
      ii--;
      break;
    }

    /* LEARNING PHASE */
    for (jj=1;jj<=p_row;jj++)
      for (kk=1;kk<=p_col;kk++)
        tmp_p[jj][kk]=p[jj][kk];
    if (local_recur1 == 1) {
      for (jj=1;jj<=w1_row;jj++) {
        tmp_p[p_row+jj][1]=0.0;
        for (kk=2;kk<=p_col;kk++)
          tmp_p[p_row+jj][kk]=a1[jj][kk-1];
      }
    }
    /*if (local_recur1 == 0);*/

    /*printf("Augmented input for the 1st hidden layer\n");*/

    for (jj=1;jj<=w1_row;jj++)
      for (kk=1;kk<=p_col;kk++)
        tmp_a1[jj][kk]=a1[jj][kk];
    if (local_recur2 == 1) {
      for (jj=1;jj<=w2_row;jj++) {
        tmp_a1[w1_row+jj][1]=0.0;
        for (kk=2;kk<=p_col;kk++)
          tmp_a1[w1_row+jj][kk]=a2[jj][kk-1];
      }
    }
    /*if (local_recur2 == 0);    */

    /*printf("Augmented input for the 2nd hidden layer\n");*/

    for (jj=1;jj<=w2_row;jj++)
      for (kk=1;kk<=p_col;kk++)
        tmp_a2[jj][kk]=a2[jj][kk];
    if (local_recur3 == 1) {
```

```
  for (jj=1;jj<=w3_row;jj++) {
    tmp_a2[w2_row+jj][1]=0.0;
    for (kk=2;kk<=p_col;kk++)
      tmp_a2[w2_row+jj][kk]=a3[jj][kk-1];
  }
}
/*if (local_recur3 == 0); */

mylearnbpm(tmp_p,w1_col,p_col,
           d1,w1_row,p_col,lr,MC,dw1,w1_row,w1_col,db1);
mylearnbpm(tmp_a1,w2_col,p_col,
           d2,w2_row,p_col,lr,MC,dw2,w2_row,w2_col,db2);
mylearnbpm(tmp_a2,w3_col,p_col,
           d3,w3_row,p_col,lr,MC,dw3,w3_row,w3_col,db3);

MC=mc;
for (jj=1;jj<=w1_row;jj++) { /* updating w1,b1 */
  new_b1[jj]=b1[jj]+db1[jj];
  for (kk=1;kk<=w1_col;kk++)
    new_w1[jj][kk]=w1[jj][kk]+dw1[jj][kk];
}

for (jj=1;jj<=w2_row;jj++) { /* updating w2,b2 */
  new_b2[jj]=b2[jj]+db2[jj];
  for (kk=1;kk<=w2_col;kk++)
    new_w2[jj][kk]=w2[jj][kk]+dw2[jj][kk];
}

for (jj=1;jj<=w3_row;jj++) { /* updating w3,b3 */
  new_b3[jj]=b3[jj]+db3[jj];
  for (kk=1;kk<=w3_col;kk++)
    new_w3[jj][kk]=w3[jj][kk]+dw3[jj][kk];
}

/* PRESENTATION PHASE */
/********* Use (*f1),(*f2),(*f3) instead of mysimuelm ***********/
mysimuelm(p,  p_row,  p_col,
          new_w1, w1_row, w1_col, new_b1, local_recur1,
          new_w2, w2_row, w2_col, new_b2, local_recur2,
          new_w3, w3_row, w3_col, new_b3, local_recur3,
          (*f1),
          (*f2),
          (*f3),
          new_a1, w1_row, p_col,
          new_a2, w2_row, p_col,
          new_a3, w3_row, p_col
         );
/*****************************************************************/
/*****************************************************************
for (mm=1;mm<=p_row;mm++)
  tmp_new_p[mm][1]=p[mm][1];

if (local_recur1 == 1) {
  for (mm=1;mm<=w1_row;mm++)
    tmp_new_p[p_row+mm][1]=0;
}

(tmp_new_a1)=(*f1)(new_w1,w1_row,w1_col,tmp_new_p,
                   p_row+w1_row*local_recur1,1,b1);

for (mm=1;mm<=w1_row;mm++) {
  tmp_tmp_new_a1[mm][1]=tmp_new_a1[mm][1];
```

```
  new_a1[mm][1]=tmp_new_a1[mm][1];
}
if (local_recur2 == 1) {
  for (mm=1;mm<=w2_row;mm++)
    tmp_tmp_new_a1[w1_row+mm][1]=0;
}

(tmp_new_a2)=(*f2)(new_w2,w2_row,w2_col,tmp_tmp_new_a1,
                   w1_row+w2_row*local_recur2,1,b2);
for (mm=1;mm<=w2_row;mm++) {
  tmp_tmp_new_a2[mm][1]=tmp_new_a2[mm][1];
  new_a2[mm][1]=tmp_new_a2[mm][1];
}
if (local_recur3 == 1) {
  for (mm=1;mm<=w3_row;mm++)
    tmp_tmp_new_a2[w2_row+mm][1]=0;
}

(tmp_new_a3)=(*f3)(new_w3,w3_row,w3_col,tmp_tmp_new_a2,
                   w2_row+w3_row*local_recur3,1,b3);

for (mm=1;mm<=w3_row;mm++)
  new_a3[mm][1]=tmp_new_a3[mm][1];

for (jj=2;jj<=p_col;jj++) {
  for (mm=1;mm<=p_row;mm++)
    tmp_new_p[mm][1]=p[mm][jj];

  if (local_recur1 == 1) {
    for (mm=1;mm<=w1_row;mm++)
      tmp_new_p[p_row+mm][1]=new_a1[mm][jj-1];
  }

  (tmp_new_a1)=(*f1)(new_w1,w1_row,w1_col,tmp_new_p,
                     p_row+w1_row*local_recur1,1,b1);

  for (mm=1;mm<=w1_row;mm++) {
    tmp_tmp_new_a1[mm][1]=tmp_new_a1[mm][1];
    new_a1[mm][jj]=tmp_new_a1[mm][1];
  }
  if (local_recur2 == 1) {
    for (mm=1;mm<=w2_row;mm++)
      tmp_tmp_new_a1[w1_row+mm][1]=new_a2[mm][jj-1];
  }

  (tmp_new_a2)=(*f2)(new_w2,w2_row,w2_col,tmp_tmp_new_a1,
                     w1_row+w2_row*local_recur2,1,b2);

  for (mm=1;mm<=w2_row;mm++) {
    tmp_tmp_new_a2[mm][1]=tmp_new_a2[mm][1];
    new_a2[mm][jj]=tmp_new_a2[mm][1];
  }
  if (local_recur3 == 1) {
    for (mm=1;mm<=w3_row;mm++)
      tmp_tmp_new_a2[w2_row+mm][1]=new_a3[mm][jj-1];
  }

  (tmp_new_a3)=(*f3)(new_w3,w3_row,w3_col,tmp_tmp_new_a2,
                     w2_row+w3_row*local_recur3,1,b3);

  for (mm=1;mm<=w3_row;mm++)
    new_a3[mm][jj]=tmp_new_a3[mm][1];
```

```
}
************************************************************/
/*****************  end of (*f1),(*f2),(*f3)   **************/

for (jj=1;jj<=w3_row;jj++)
  for (kk=1;kk<=p_col;kk++)
    new_e[jj][kk]=t[jj][kk]-new_a3[jj][kk];


new_SSE=mysumsqr(new_e,w3_row,p_col);
                      /* new_e (as e) with dimension w3_row x p_col */

/* Momentum and adaptive learning rate phase */
if (new_SSE > SSE * er) {
  lr=lr*dm;
  MC=0;
}
else {
  if (new_SSE < SSE) {
    lr=lr*im;
  }
  for (jj=1;jj<=w1_row;jj++) {
    b1[jj]=new_b1[jj];
    for (kk=1;kk<=w1_col;kk++) {
      w1[jj][kk]=new_w1[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a1[jj][kk]=new_a1[jj][kk];
    }
  }
  for (jj=1;jj<=w2_row;jj++) {
    b2[jj]=new_b2[jj];
    for (kk=1;kk<=w2_col;kk++) {
      w2[jj][kk]=new_w2[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a2[jj][kk]=new_a2[jj][kk];
    }
  }
  for (jj=1;jj<=w3_row;jj++) {
    b3[jj]=new_b3[jj];
    for (kk=1;kk<=w3_col;kk++) {
      w3[jj][kk]=new_w3[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a3[jj][kk]=new_a3[jj][kk];
      e[jj][kk] =new_e[jj][kk];
    }
  }

  SSE=new_SSE;

  /* BACKPROPAGATION PHASE */
  for (jj=1;jj<=w3_row;jj++)
    for (kk=1;kk<=(w3_col-w3_row*local_recur3);kk++)
      tmp_w3[jj][kk]=w3[jj][kk];

  for (jj=1;jj<=w2_row;jj++)
    for (kk=1;kk<=(w2_col-w2_row*local_recur2);kk++)
      tmp_w2[jj][kk]=w2[jj][kk];
  /*
  d3=(*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL);
```

```
      d2=(*df2)(a2,w2_row,p_col,d3,w3_row,p_col,tmp_w3,w3_row,
                w3_col-w3_row*local_recur3);
      d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
                w2_col-w2_row*local_recur2);
      */
      (*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL,d3);
      (*df2)(a2,w2_row,p_col,d3,w3_row,p_col,tmp_w3,w3_row,
             w3_col-w3_row*local_recur3,d2);
      (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
             w2_col-w2_row*local_recur2,d1);

    }

    /* TRAINING RECORDS */
    tr[1][ii+1]=SSE;
    tr[2][ii+1]=lr;

    if ((ii % df) == 0)
      printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);

  } /* end ii --- "for" loop */

  if ((ii % df) !=0) {/* This is for last training epoch printing
                         in case me is not multiples of df        */
    printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);
  }

  if ( SSE > eg) {
    printf("Trainrbpx: network error did not reach the error goal;\n");
    printf("Future training may be necessary or try different    \n");
    printf("initial weights and biases and/or more hidden neurons.\n");
  }

}

/************** END of mytrbpx3() ************************/

void mytrbpx2(float **w1, int w1_row, int w1_col, float *b1, int local_recur1,
              float **w2, int w2_row, int w2_col, float *b2, int local_recur2,
              float **p,  int p_row, int p_col,
              float **t,  int t_row, int t_col,
              float *tp,
              void (*f1)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*f2)(float **w,int row,int col,float **x,
              int x_row,int x_col, float *b,float **a),
              void (*df1)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col,float **r),
              void (*df2)(float **a,int a_row,int a_col,float **d,
              int d_row,int d_col,float **w,int w_row,int w_col,float **r)
             )

{
  int df,me; /* epoches for display, default = 25
                max number of epochs to train, default = 1000 */
  float eg,lr,im,dm,mc,er; /* sum-square error goal, default = 0.02;
                              learning rate, default = 0.01;
                              learning rate increse, default = 1.05;
                              learning rate decrease, default = 0.7;
                              momentum constant, default = 0.9;
                              maximum error ratio, default = 1.04     */

  float MC;
```

```
float **dw1, **dw2;
float *db1,*db2;

float **new_w1, **new_w2;
float *new_b1,*new_b2;

float **a1,**a2,**e;
float **new_a1,**new_a2,**new_e;

float **d1,**d2;

float SSE, new_SSE;

float **tr; /* training record */

float **tmp_p ,**tmp_a1; /* in case of local recurrency,temp storage */
float **tmp_w2; /* in case of local recurrency, part of w2 */

int ii,jj,kk;

if (   w1_col!=( p_row+w1_row*local_recur1)
    || w2_col!=(w1_row+w2_row*local_recur2)) {
  printf("Dimension inconsistency in mytrainelm\n");
  printf("Exiting from mytrainelm\n");
  exit(1);
} /* Note: the number of neurons is the same
     as that of rows for the weight matrix.
     ===> # neurons in layer 1 = w1_row; etc */

printf("Welcome to the Recurrent Neural Net training program\n");
df=tp[1];
me=tp[2];
eg=tp[3];
lr=tp[4];
im=tp[5];
dm=tp[6];
mc=tp[7];
er=tp[8];

/* use memo_man.c in ~/PS_simu/Copt to allocate memory */
tr    =allocate_real_matrix(1,2,1,me+1);

dw1   =allocate_real_matrix(1, w1_row, 1, w1_col);
                       /* w1_row: number of neurons in the present layer  */
                       /* w1_col: number of neurons in the previous layer */
                       /* all others: with the same interpretation        */
new_w1=allocate_real_matrix(1, w1_row, 1, w1_col);
dw2   =allocate_real_matrix(1, w2_row, 1, w2_col);
new_w2=allocate_real_matrix(1, w2_row, 1, w2_col);

db1   =allocate_real_vector(1, w1_row);
new_b1=allocate_real_vector(1, w1_row);
db2   =allocate_real_vector(1, w2_row);
new_b2=allocate_real_vector(1, w2_row);

for (ii=1;ii<=w1_row;ii++) {
  db1[ii]=0.0;
  for (jj=1;jj<=w1_col;jj++)
    dw1[ii][jj]=0.0;
}

for (ii=1;ii<=w2_row;ii++) {
  db2[ii]=0.0;
```

```
    for (jj=1;jj<=w2_col;jj++)
      dw2[ii][jj]=0.0;
  }


  MC=0;

  a1    =allocate_real_matrix(1, w1_row, 1, p_col);
  new_a1=allocate_real_matrix(1, w1_row, 1, p_col);
  a2    =allocate_real_matrix(1, w2_row, 1, p_col);
  new_a2=allocate_real_matrix(1, w2_row, 1, p_col);

  d1    =allocate_real_matrix(1, w1_row, 1, p_col);
  d2    =allocate_real_matrix(1, w2_row, 1, p_col);
  e     =allocate_real_matrix(1, w2_row, 1, p_col);
  new_e =allocate_real_matrix(1, w2_row, 1, p_col);

/********** Presentation Phase ****************************/

  mysimuelm(p,  p_row,  p_col,
            w1, w1_row, w1_col, b1, local_recur1,/* yes/no: 1/0 */
            w2, w2_row, w2_col, b2, local_recur2,/* yes/no: 1/0 */
            NULL, NULL, NULL, NULL, NULL,
            (*f1),
            (*f2),
            NULL,
            a1, w1_row, p_col, /* a1_row=w1_row; a1_col=p_col; */
            a2, w2_row, p_col, /* a2_row=w2_row; a2_col=p_col; */
            NULL, NULL, NULL   /* a3_row=w3_row; a3_col=p_col; */
           );

  for (ii=1;ii<=w2_row;ii++)
    for (jj=1;jj<=p_col;jj++)
      e[ii][jj]=t[ii][jj]-a2[ii][jj];

  SSE = mysumsqr(e,w2_row,p_col);
  printf("Presentation Phase finished\n");
  printf("Initial SSE=%f\n",SSE);

/********** BackPropagation Phase *************************/
/* ALL the following lines must be modified, which are copied from mytbpx3.c */
  tmp_w2=allocate_real_matrix(1,w2_row,1,w2_col-w2_row*local_recur2);

  for (jj=1;jj<=w2_row;jj++)
    for (kk=1;kk<=(w2_col-w2_row*local_recur2);kk++)
      tmp_w2[jj][kk]=w2[jj][kk];
  /*
  d2=(*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL);
  d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
            w2_col-w2_row*local_recur2);
  */
  (*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL,d2);
  (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
         w2_col-w2_row*local_recur2,d1);

  tmp_p =allocate_real_matrix(1, p_row+w1_row*local_recur1,1,p_col);
  tmp_a1=allocate_real_matrix(1,w1_row+w2_row*local_recur2,1,p_col);

  /*** Training LOOP begins ***/
  for (ii=1;ii<=me;ii++) {
    if (SSE < eg) { /* CHECK PHASE */
      ii--;
      break;
```

```
}

/* LEARNING PHASE */
for (jj=1;jj<=p_row;jj++)
  for (kk=1;kk<=p_col;kk++)
    tmp_p[jj][kk]=p[jj][kk];
if (local_recur1 == 1) {
  for (jj=1;jj<=w1_row;jj++) {
    tmp_p[p_row+jj][1]=0.0;
    for (kk=2;kk<=p_col;kk++)
      tmp_p[p_row+jj][kk]=a1[jj][kk-1];
  }
}
/*if (local_recur1 == 0) ;*/

for (jj=1;jj<=w1_row;jj++)
  for (kk=1;kk<=p_col;kk++)
    tmp_a1[jj][kk]=a1[jj][kk];
if (local_recur2 == 1) {
  for (jj=1;jj<=w2_row;jj++) {
    tmp_a1[w1_row+jj][1]=0.0;
    for (kk=2;kk<=p_col;kk++)
      tmp_a1[w1_row+jj][kk]=a2[jj][kk-1];
  }
}
/*if (local_recur2 == 0) ;    */

mylearnbpm(tmp_p,p_row+w1_row*local_recur1,p_col,
           d1,w1_row,p_col,lr,MC,dw1,w1_row,w1_col,db1);
mylearnbpm(tmp_a1,w1_row+w2_row*local_recur2,p_col,
           d2,w2_row,p_col,lr,MC,dw2,w2_row,w2_col,db2);

MC=mc;
for (jj=1;jj<=w1_row;jj++) { /* updating w1,b1 */
  new_b1[jj]=b1[jj]+db1[jj];
  for (kk=1;kk<=w1_col;kk++)
    new_w1[jj][kk]=w1[jj][kk]+dw1[jj][kk];
}

for (jj=1;jj<=w2_row;jj++) { /* updating w2,b2 */
  new_b2[jj]=b2[jj]+db2[jj];
  for (kk=1;kk<=w2_col;kk++)
    new_w2[jj][kk]=w2[jj][kk]+dw2[jj][kk];
}

/* PRESENTATION PHASE */
mysimuelm(p,  p_row,  p_col,
          new_w1, w1_row, w1_col, new_b1, local_recur1,/* yes/no: 1/0 */
          new_w2, w2_row, w2_col, new_b2, local_recur2,/* yes/no: 1/0 */
          NULL, NULL, NULL, NULL, NULL,
          (*f1),
          (*f2),
          NULL,
          new_a1, w1_row, p_col, /* a1_row=w1_row; a1_col=p_col; */
          new_a2, w2_row, p_col, /* a2_row=w2_row; a2_col=p_col; */
          NULL, NULL, NULL
         );

for (jj=1;jj<=w2_row;jj++)
  for (kk=1;kk<=p_col;kk++)
    new_e[jj][kk]=t[jj][kk]-new_a2[jj][kk];
```

```
  new_SSE=mysumsqr(new_e,w2_row,p_col);
                        /* new_e (as e) with dimension w3_row x p_col */

  /* Momentum and adaptive learning rate phase */
  if (new_SSE > SSE * er) {
    lr=lr*dm;
    MC=0;
  }
  else {
    if (new_SSE < SSE) {
      lr=lr*im;
    }
    for (jj=1;jj<=w1_row;jj++) {
      b1[jj]=new_b1[jj];
      for (kk=1;kk<=w1_col;kk++) {
        w1[jj][kk]=new_w1[jj][kk];
      }
      for (kk=1;kk<=p_col;kk++) {
        a1[jj][kk]=new_a1[jj][kk];
      }
    }
    for (jj=1;jj<=w2_row;jj++) {
      b2[jj]=new_b2[jj];
      for (kk=1;kk<=w2_col;kk++) {
        w2[jj][kk]=new_w2[jj][kk];
      }
      for (kk=1;kk<=p_col;kk++) {
        a2[jj][kk]=new_a2[jj][kk];
        e[jj][kk] =new_e[jj][kk];
      }
    }

    SSE=new_SSE;

    /* BACKPROPAGATION PHASE */
    for (jj=1;jj<=w2_row;jj++)
      for (kk=1;kk<=(w2_col-w2_row*local_recur2);kk++)
        tmp_w2[jj][kk]=w2[jj][kk];

    /*
    d2=(*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL);
    d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
              w2_col-w2_row*local_recur2);
    */
    (*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL,d2);
    (*df1)(a1,w1_row,p_col,d2,w2_row,p_col,tmp_w2,w2_row,
           w2_col-w2_row*local_recur2,d1);

  }

  /* TRAINING RECORDS */
  tr[1][ii+1]=SSE;
  tr[2][ii+1]=lr;

  if ((ii % df) == 0)
    printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);

} /* end ii --- "for" loop */

if ((ii % df) !=0) {/* This is for last training epoch printing
                       in case me is not multiples of df        */
  printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);
}
```

```
  if ( SSE > eg) {
    printf("Trainrbpx: network error did not reach the error goal;\n");
    printf("Future training may be necessary or try different    \n");
    printf("initial weights and biases and/or more hidden neurons.\n");
  }

}

/*************** END of mytrbpx2() ***********************/

/*************** END of mytrainelm.c *********************/

/*************** BEGIN of mysimuelm.c ********************/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mynn.h"

/********** To simulate a recurrent neural network ************/
void mysimuelm(float **p, int p_row, int p_col,
               float **w1, int w1_row, int w1_col, float *b1,int local_recur1,
                                                    /* yes/no: 1/0 */
               float **w2, int w2_row, int w2_col, float *b2,int local_recur2,
                                                    /* yes/no: 1/0 */
               float **w3, int w3_row, int w3_col, float *b3,int local_recur3,
                                                    /* yes/no: 1/0 */
               void (*f1)(float **w,int row,int col,float **x,
               int x_row,int x_col, float *b,float **a),
               void (*f2)(float **w,int row,int col,float **x,
               int x_row,int x_col, float *b,float **a),
               void (*f3)(float **w,int row,int col,float **x,
               int x_row,int x_col, float *b,float **a),
               float **a1, int a1_row, int a1_col,
               float **a2, int a2_row, int a2_col,
               float **a3, int a3_row, int a3_col
              )

{
  int ii,jj;
  float **tmp_p,**tmp_a1,**tmp_a2,**tmp_a3;
  float **tmp_tmp_a1,**tmp_tmp_a2;

  /* Dimension consistency check */
  if ((a3) == NULL) { /* only 1 hidden layer case */
    if (w1_col!=(p_row+w1_row*local_recur1) ||
        w2_col!=(w1_row+w2_row*local_recur2)) {
      printf("Dimension inconsistency in mysimuelm\n");
      printf("Exiting from mysimuelm\n");
      exit(1);
    } /* Note: the number of neurons is the same
         as that of rows for the weight matrix.
         ===> # neurons in layer 1 = w1_row; etc */

    if (a1_row!=w1_row || a1_col!=p_col || a2_row!=w2_row || a2_col!=p_col) {
      printf("Dimension inconsistency in mysimuelm\n");
      printf("Exiting from mysimuelm\n");
      exit(1);
    }

    tmp_p=allocate_real_matrix(1,p_row+w1_row*local_recur1,1,1);
    for (ii=1;ii<=p_row;ii++)
      tmp_p[ii][1]=p[ii][1];
```

```
if (local_recur1 == 1) { /* local recurrency */
  for (ii=1;ii<=w1_row;ii++)
    tmp_p[p_row+ii][1]=0; /* Note: here the init_a1 assumed to be 0's;
                                   and may use other non-zero values for
                                   initial conditions                  */

}

/*if (local_recur1 == 0) ; no local recurrency; doing nothing */
tmp_a1=allocate_real_matrix(1,w1_row,1,1);
/*(tmp_a1)=(*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1);*/
(*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1,tmp_a1);

tmp_tmp_a1=allocate_real_matrix(1,w1_row+w2_row*local_recur2,1,1);
for (ii=1;ii<=w1_row;ii++) {
  tmp_tmp_a1[ii][1]=tmp_a1[ii][1];
  a1[ii][1]=tmp_a1[ii][1];
}
if (local_recur2 == 1) {
  for (ii=1;ii<=w2_row;ii++)
    tmp_tmp_a1[w1_row+ii][1]=0;
}

/*if (local_recur2 == 0) ;*/
tmp_a2=allocate_real_matrix(1,w2_row,1,1);
/*(tmp_a2)=(*f2)(w2,w2_row,w2_col,tmp_tmp_a1,
                 w1_row+w2_row*local_recur2,1,b2);*/
(*f2)(w2,w2_row,w2_col,tmp_tmp_a1,w1_row+w2_row*local_recur2,1,b2,tmp_a2);
for (ii=1;ii<=w2_row;ii++)
  a2[ii][1]=tmp_a2[ii][1];

/* calculate the output column by column starting from the 2nd column */
for (jj=2;jj<=p_col;jj++) {
  for (ii=1;ii<=p_row;ii++)
    tmp_p[ii][1]=p[ii][jj];

  if (local_recur1 == 1) { /* local recurrency */
    for (ii=1;ii<=w1_row;ii++)
      tmp_p[p_row+ii][1]=a1[ii][jj-1];
  }

  /*if (local_recur1 == 0) ; no local recurrency; doing nothing */
  /*(tmp_a1)=(*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1);*/
  (*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1,tmp_a1);
  for (ii=1;ii<=w1_row;ii++) {
    tmp_tmp_a1[ii][1]=tmp_a1[ii][1];
    a1[ii][jj]=tmp_a1[ii][1];
  }
  if (local_recur2 == 1) {
    for (ii=1;ii<=w2_row;ii++)
      tmp_tmp_a1[w1_row+ii][1]=a2[ii][jj-1];
  }

  /*if (local_recur2 == 0) ;*/

  /*(tmp_a2)=(*f2)(w2,w2_row,w2_col,tmp_tmp_a1,
                   w1_row+w2_row*local_recur2,1,b2);*/
  (*f2)(w2,w2_row,w2_col,tmp_tmp_a1,w1_row+w2_row*local_recur2,1,b2,tmp_a2);
  for (ii=1;ii<=w2_row;ii++)
    a2[ii][jj]=tmp_a2[ii][1];

} /* End of loop jj */
```

```
  free_real_matrix(tmp_p,1,p_row+w1_row*local_recur1,1);
  free_real_matrix(tmp_a1,1,w1_row,1);
  free_real_matrix(tmp_tmp_a1,1,w1_row+w2_row*local_recur2,1);
  free_real_matrix(tmp_a2,1,w2_row,1);


} /* End of 1 hidden layer case */

/******* Case 2: 2 hidden layer case ******/
if ((a3) != NULL) { /* 2 hidden layer case */
  if (w1_col!=(p_row+w1_row*local_recur1) ||
              w2_col!=(w1_row+w2_row*local_recur2)
      || w3_col!=(w2_row+w3_row*local_recur3)) {
    printf("Dimension inconsistency in mysimuff\n");
    printf("Exiting from mysimuff\n");
    exit(1);
  }
  if (a1_row!=w1_row || a1_col!=p_col || a2_row!=w2_row || a2_col!=p_col ||
      a3_row!=w3_row || a3_col!=p_col) {
    printf("Dimension inconsistency in mysimuff\n");
    printf("Exiting from mysimuff\n");
    exit(1);
  }

  tmp_p=allocate_real_matrix(1,p_row+w1_row*local_recur1,1,1);
  tmp_a1=allocate_real_matrix(1,w1_row,1,1);
  tmp_a2=allocate_real_matrix(1,w2_row,1,1);
  tmp_tmp_a1=allocate_real_matrix(1,w1_row+w2_row*local_recur2,1,1);
  tmp_tmp_a2=allocate_real_matrix(1,w2_row+w3_row*local_recur3,1,1);
  tmp_a3=allocate_real_matrix(1,w3_row,1,1);

  for (ii=1;ii<=p_row;ii++)
    tmp_p[ii][1]=p[ii][1];

  if (local_recur1 == 1) { /* local recurrency */
    for (ii=1;ii<=w1_row;ii++)
      tmp_p[p_row+ii][1]=0; /* Note: here the init_a1 assumed to be 0's;
                               and may use other non-zero values for
                               initial conditions                    */

  }

  /*if (local_recur1 == 0) ; no local recurrency; doing nothing */
  /*tmp_a1=allocate_real_matrix(1,w1_row,1,1);          */
  /*(tmp_a1)=(*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1);*/
  (*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1,tmp_a1);

  /*tmp_tmp_a1=allocate_real_matrix(1,w1_row+w2_row*local_recur2,1,1);*/
  for (ii=1;ii<=w1_row;ii++) {
    tmp_tmp_a1[ii][1]=tmp_a1[ii][1];
    a1[ii][1]=tmp_a1[ii][1];
  }
  if (local_recur2 == 1) {
    for (ii=1;ii<=w2_row;ii++)
      tmp_tmp_a1[w1_row+ii][1]=0;
  }

  /*if (local_recur2 == 0) ;*/
  /*tmp_a2=allocate_real_matrix(1,w2_row,1,1);        */
  /*(tmp_a2)=(*f2)(w2,w2_row,w2_col,tmp_tmp_a1,
                   w1_row+w2_row*local_recur2,1,b2);*/
  (*f2)(w2,w2_row,w2_col,tmp_tmp_a1,w1_row+w2_row*local_recur2,1,b2,tmp_a2);
```

```
/*tmp_tmp_a2=allocate_real_matrix(1,w2_row+w3_row*local_recur3,1,1);*/
for (ii=1;ii<=w2_row;ii++) {
  tmp_tmp_a2[ii][1]=tmp_a2[ii][1];
  a2[ii][1]=tmp_a2[ii][1];
}
if (local_recur3 == 1) {
  for (ii=1;ii<=w3_row;ii++)
    tmp_tmp_a2[w2_row+ii][1]=0;
}

/*if (local_recur3 == 0) ;*/

/*tmp_a3=allocate_real_matrix(1,w3_row,1,1);*/
/*(tmp_a3)=(*f3)(w3,w3_row,w3_col,tmp_tmp_a2,
                w2_row+w3_row*local_recur3,1,b3);*/
(*f3)(w3,w3_row,w3_col,tmp_tmp_a2,w2_row+w3_row*local_recur3,1,b3,tmp_a3);

for (ii=1;ii<=w3_row;ii++)
  a3[ii][1]=tmp_a3[ii][1];

/*** Calculating the outputs column by column starting from 2nd column ***/
for (jj=2;jj<=p_col;jj++) {
  for (ii=1;ii<=p_row;ii++)
    tmp_p[ii][1]=p[ii][jj];

  if (local_recur1 == 1) { /* local recurrency */
    for (ii=1;ii<=w1_row;ii++)
      tmp_p[p_row+ii][1]=a1[ii][jj-1];
  }

  /*if (local_recur1 == 0) ; no local recurrency; doing nothing */
  /*(tmp_a1)=(*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1);*/
  (*f1)(w1,w1_row,w1_col,tmp_p,p_row+w1_row*local_recur1,1,b1,tmp_a1);

  for (ii=1;ii<=w1_row;ii++) {
    tmp_tmp_a1[ii][1]=tmp_a1[ii][1];
    a1[ii][jj]=tmp_a1[ii][1];
  }
  if (local_recur2 == 1) {
    for (ii=1;ii<=w2_row;ii++)
      tmp_tmp_a1[w1_row+ii][1]=a2[ii][jj-1];
  }

  /*if (local_recur2 == 0) ;*/
  /*(tmp_a2)=(*f2)(w2,w2_row,w2_col,tmp_tmp_a1,
                  w1_row+w2_row*local_recur2,1,b2);*/
  (*f2)(w2,w2_row,w2_col,tmp_tmp_a1,w1_row+w2_row*local_recur2,1,b2,tmp_a2);

  for (ii=1;ii<=w2_row;ii++) {
    tmp_tmp_a2[ii][1]=tmp_a2[ii][1];
    a2[ii][jj]=tmp_a2[ii][1];
  }
  if (local_recur3 == 1) {
    for (ii=1;ii<=w3_row;ii++)
      tmp_tmp_a2[w2_row+ii][1]=a3[ii][jj-1];
  }

  /*if (local_recur3 == 0) ;*/

  /*(tmp_a3)=(*f3)(w3,w3_row,w3_col,tmp_tmp_a2,
                  w2_row+w3_row*local_recur3,1,b3);*/
  (*f3)(w3,w3_row,w3_col,tmp_tmp_a2,w2_row+w3_row*local_recur3,1,b3,tmp_a3);
```

```
      for (ii=1;ii<=w3_row;ii++)
        a3[ii][jj]=tmp_a3[ii][1];

    }
    free_real_matrix(tmp_p,1,p_row+w1_row*local_recur1,1);
    free_real_matrix(tmp_a1,1,w1_row,1);
    free_real_matrix(tmp_tmp_a1,1,w1_row+w2_row*local_recur2,1);
    free_real_matrix(tmp_a2,1,w2_row,1);
    free_real_matrix(tmp_a3,1,w3_row,1);
    free_real_matrix(tmp_tmp_a2,1,w2_row+w3_row*local_recur3,1);

  }

}

/********************* END of mysimuelm.c ******************************/

/********************* BEGIN of mytbpx2.c ******************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

void mytbpx2(float **w1, int w1_row, int w1_col, float *b1,
             float **w2, int w2_row, int w2_col, float *b2,
             float **p,  int p_row, int p_col,
             float **t,  int t_row, int t_col,
             float *tp,
             float ** (*f1)(float **w,int row,int col,float **x,
             int x_row,int x_col, float *b),
             float ** (*f2)(float **w,int row,int col,float **x,
             int x_row,int x_col, float *b),
             float ** (*df1)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col),
             float ** (*df2)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col)
            )

{

  float mysumsqr(float **e,int e_row,int e_col);
  void mylearnbpm(float **p,int p_row,int p_col,
                  float **d,int d_row,int d_col,
                  float lr,float mc,
                  float **dw,int dw_row,int dw_col,
                  float *db);
  float ** allocate_real_matrix(int,int,int,int);
  float * allocate_real_vector(int,int);

  int df,me; /* epoches for display, default = 25
                max number of epochs to train, default = 1000 */
  float eg,lr,im,dm,mc,er; /* sum-square error goal, default = 0.02;
                              learning rate, default = 0.01;
                              learning rate increse, default = 1.05;
                              learning rate decrease, default = 0.7;
                              momentum constant, default = 0.9;
                              maximum error ratio, default = 1.04     */

  float MC;
  float **dw1, **dw2;
  float *db1,*db2;

  float **new_w1, **new_w2;
```

```
float *new_b1,*new_b2;

float **a1,**a2,**e;
float **new_a1,**new_a2,**new_e;

float **d1,**d2;

float SSE, new_SSE;

float **tr; /* training record */
int ii,jj,kk;

printf("Welcome to the neural net training program\n");
df=tp[1];
me=tp[2];
eg=tp[3];
lr=tp[4];
im=tp[5];
dm=tp[6];
mc=tp[7];
er=tp[8];

/* use memo_man.c in ~/PS_simu/Copt to allocate memory */
tr    =allocate_real_matrix(1,2,1,me+1);

dw1   =allocate_real_matrix(1, w1_row, 1, w1_col);
                      /* w1_row: number of neurons in the present layer  */
                      /* w1_col: number of neurons in the previous layer */
                      /* all others: with the same interpretation      */
new_w1=allocate_real_matrix(1, w1_row, 1, w1_col);
dw2   =allocate_real_matrix(1, w2_row, 1, w2_col);
new_w2=allocate_real_matrix(1, w2_row, 1, w2_col);

db1   =allocate_real_vector(1, w1_row);
new_b1=allocate_real_vector(1, w1_row);
db2   =allocate_real_vector(1, w2_row);
new_b2=allocate_real_vector(1, w2_row);

for (ii=1;ii<=w1_row;ii++) {
  db1[ii]=0.0;
  for (jj=1;jj<=w1_col;jj++)
    dw1[ii][jj]=0.0;
}

for (ii=1;ii<=w2_row;ii++) {
  db2[ii]=0.0;
  for (jj=1;jj<=w2_col;jj++)
    dw2[ii][jj]=0.0;
}

MC=0;

a1    =allocate_real_matrix(1, w1_row, 1, p_col);
new_a1=allocate_real_matrix(1, w1_row, 1, p_col);
a2    =allocate_real_matrix(1, w2_row, 1, p_col);
new_a2=allocate_real_matrix(1, w2_row, 1, p_col);

d1    =allocate_real_matrix(1, w1_row, 1, p_col);
d2    =allocate_real_matrix(1, w2_row, 1, p_col);
e     =allocate_real_matrix(1, w2_row, 1, p_col);
new_e =allocate_real_matrix(1, w2_row, 1, p_col);

printf("Memory allocation ready\n");
```

```
/********** Presentation Phase ****************************/
  a1=(*f1)(w1,w1_row,w1_col,p,p_row,p_col,b1);
      /* note: a1 is a matrix with dimension w1_row x p_col */
  a2=(*f2)(w2,w2_row,w2_col,a1,w1_row,p_col,b2);
      /* note: a2 is a matrix with dimension w2_row x p_col */

  for (ii=1;ii<=w2_row;ii++)
    for (jj=1;jj<=p_col;jj++)
      e[ii][jj]=t[ii][jj]-a2[ii][jj];

  SSE = mysumsqr(e,w2_row,p_col);
  printf("Presentation Phase finished\n");
  printf("Initial SSE=%f\n",SSE);

/********** BackPropagation Phase *************************/
  d2=(*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL);
                                    /* e, a3, d3 are same dimensional */
  d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col);
                                    /* d1,a1 same dimension */

  for (ii=1;ii<=me;ii++) {
    if (SSE < eg) { /* CHECK PHASE */
      ii--;
      break;
    }

    /* LEARNING PHASE */
    mylearnbpm(p,p_row,p_col,d1,w1_row,p_col,lr,MC,dw1,w1_row,w1_col,db1);
    mylearnbpm(a1,w1_row,p_col,d2,w2_row,p_col,lr,MC,dw2,w2_row,w2_col,db2);

    MC=mc;
    for (jj=1;jj<=w1_row;jj++) { /* updating w1,b1 */
      new_b1[jj]=b1[jj]+db1[jj];
      for (kk=1;kk<=w1_col;kk++)
        new_w1[jj][kk]=w1[jj][kk]+dw1[jj][kk];
    }

    for (jj=1;jj<=w2_row;jj++) { /* updating w2,b2 */
      new_b2[jj]=b2[jj]+db2[jj];
      for (kk=1;kk<=w2_col;kk++)
        new_w2[jj][kk]=w2[jj][kk]+dw2[jj][kk];
    }

    /* PRESENTATION PHASE */
    new_a1=(*f1)(new_w1,w1_row,w1_col,p,p_row,p_col,new_b1);
      /* note: new_a1 (as a1) is a matrix with dimension w1_row x p_col */
    new_a2=(*f2)(new_w2,w2_row,w2_col,new_a1,w1_row,p_col,new_b2);
      /* note: new_a2 (as a2) is a matrix with dimension w2_row x p_col */

    for (jj=1;jj<=w2_row;jj++)
      for (kk=1;kk<=p_col;kk++)
        new_e[jj][kk]=t[jj][kk]-new_a2[jj][kk];


    new_SSE=mysumsqr(new_e,w2_row,p_col);
                        /* new_e (as e) with dimension w3_row x p_col */

    /* Momentum and adaptive learning rate phase */
    if (new_SSE > SSE * er) {
      lr=lr*dm;
      MC=0;
    }
```

```
    else {
      if (new_SSE < SSE) {
        lr=lr*im;
      }
      for (jj=1;jj<=w1_row;jj++) {
        b1[jj]=new_b1[jj];
        for (kk=1;kk<=w1_col;kk++) {
          w1[jj][kk]=new_w1[jj][kk];
        }
        for (kk=1;kk<=p_col;kk++) {
          a1[jj][kk]=new_a1[jj][kk];
        }
      }
      for (jj=1;jj<=w2_row;jj++) {
        b2[jj]=new_b2[jj];
        for (kk=1;kk<=w2_col;kk++) {
          w2[jj][kk]=new_w2[jj][kk];
        }
        for (kk=1;kk<=p_col;kk++) {
          a2[jj][kk]=new_a2[jj][kk];
          e[jj][kk] =new_e[jj][kk];
        }
      }

      SSE=new_SSE;

      /* BACKPROPAGATION PHASE */
      d2=(*df2)(a2,w2_row,p_col,e,w2_row,p_col,NULL,NULL,NULL);
                                      /* e, a3, d3 are same dimensional */
      d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col);
                                      /* d1,a1 same dimension */
    }

    /* TRAINING RECORDS */
    tr[1][ii+1]=SSE;
    tr[2][ii+1]=lr;

    if ((ii % df) == 0)
      printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);

  } /* end ii --- "for" loop */

  if ((ii % df) !=0) {/* This is for last training epoch printing
                         in case me is not multiples of df        */
    printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);
  }

  if ( SSE > eg) {
    printf("Tranibpx: network error did not reach the error goal;\n");
    printf("Future training may be necessary or try different    \n");
    printf("initial weights and biases and/or more hidden neurons.\n");
  }
}


/*************** END of mytbpx2.c ***********************/

/*************** BEGIN of mytbpx3.c *********************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"
```

```
void mytbpx3(float **w1, int w1_row, int w1_col, float *b1,
             float **w2, int w2_row, int w2_col, float *b2,
             float **w3, int w3_row, int w3_col, float *b3,
             float **p,  int p_row, int p_col,
             float **t,  int t_row, int t_col,
             float *tp,
             float ** (*f1)(float **w,int row,int col,float **x,
                            int x_row,int x_col, float *b),
             float ** (*f2)(float **w,int row,int col,float **x,
                            int x_row,int x_col, float *b),
             float ** (*f3)(float **w,int row,int col,float **x,
                            int x_row,int x_col, float *b),
             float ** (*df1)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col),
             float ** (*df2)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col),
             float ** (*df3)(float **a,int a_row,int a_col,float **d,
             int d_row,int d_col,float **w,int w_row,int w_col)
            )

{

  float mysumsqr(float **e,int e_row,int e_col);
  void mylearnbpm(float **p,int p_row,int p_col,
                  float **d,int d_row,int d_col,
                  float lr,float mc,
                  float **dw,int dw_row,int dw_col,
                  float *db);
  float ** allocate_real_matrix(int,int,int,int);
  float * allocate_real_vector(int,int);

  int df,me; /* epoches for display, default = 25
                max number of epochs to train, default = 1000 */
  float eg,lr,im,dm,mc,er; /* sum-square error goal, default = 0.02;
                              learning rate, default = 0.01;
                              learning rate increse, default = 1.05;
                              learning rate decrease, default = 0.7;
                              momentum constant, default = 0.9;
                              maximum error ratio, default = 1.04     */

  float MC;
  float **dw1, **dw2, **dw3;
  float *db1,*db2,*db3;

  float **new_w1, **new_w2, **new_w3;
  float *new_b1,*new_b2,*new_b3;

  float **a1,**a2,**a3,**e;
  float **new_a1,**new_a2,**new_a3,**new_e;

  float **d1,**d2,**d3;

  float SSE, new_SSE;

  float **tr; /* training record */
  int ii,jj,kk;

  printf("Welcome to the neural net training program\n");
  df=tp[1];
  me=tp[2];
  eg=tp[3];
  lr=tp[4];
```

```
im=tp[5];
dm=tp[6];
mc=tp[7];
er=tp[8];

/* use memo_man.c in ~/PS_simu/Copt to allocate memory */
tr    =allocate_real_matrix(1,2,1,me+1);

dw1   =allocate_real_matrix(1, w1_row, 1, w1_col);
                        /* w1_row: number of neurons in the present layer  */
                        /* w1_col: number of neurons in the previous layer */
                        /* all others: with the same interpretation      */
new_w1=allocate_real_matrix(1, w1_row, 1, w1_col);
dw2   =allocate_real_matrix(1, w2_row, 1, w2_col);
new_w2=allocate_real_matrix(1, w2_row, 1, w2_col);
dw3   =allocate_real_matrix(1, w3_row, 1, w3_col);
new_w3=allocate_real_matrix(1, w3_row, 1, w3_col);

db1   =allocate_real_vector(1, w1_row);
new_b1=allocate_real_vector(1, w1_row);
db2   =allocate_real_vector(1, w2_row);
new_b2=allocate_real_vector(1, w2_row);
db3   =allocate_real_vector(1, w3_row);
new_b3=allocate_real_vector(1, w3_row);

for (ii=1;ii<=w1_row;ii++) {
  db1[ii]=0.0;
  for (jj=1;jj<=w1_col;jj++)
    dw1[ii][jj]=0.0;
}

for (ii=1;ii<=w2_row;ii++) {
  db2[ii]=0.0;
  for (jj=1;jj<=w2_col;jj++)
    dw2[ii][jj]=0.0;
}

for (ii=1;ii<=w3_row;ii++) {
  db3[ii]=0.0;
  for (jj=1;jj<=w3_col;jj++)
    dw3[ii][jj]=0.0;
}

MC=0;

a1    =allocate_real_matrix(1, w1_row, 1, p_col);
new_a1=allocate_real_matrix(1, w1_row, 1, p_col);
a2    =allocate_real_matrix(1, w2_row, 1, p_col);
new_a2=allocate_real_matrix(1, w2_row, 1, p_col);
a3    =allocate_real_matrix(1, w3_row, 1, p_col);
new_a3=allocate_real_matrix(1, w3_row, 1, p_col);

d1    =allocate_real_matrix(1, w1_row, 1, p_col);
d2    =allocate_real_matrix(1, w2_row, 1, p_col);
d3    =allocate_real_matrix(1, w3_row, 1, p_col);
e     =allocate_real_matrix(1, w3_row, 1, p_col);
new_e =allocate_real_matrix(1, w3_row, 1, p_col);

printf("Memory allocation ready\n");

/*********** Presentation Phase ****************************/
 a1=(*f1)(w1,w1_row,w1_col,p,p_row,p_col,b1);
```

```
    /* note: a1 is a matrix with dimension w1_row x p_col */
  a2=(*f2)(w2,w2_row,w2_col,a1,w1_row,p_col,b2);
    /* note: a2 is a matrix with dimension w2_row x p_col */
  a3=(*f3)(w3,w3_row,w3_col,a2,w2_row,p_col,b3);
    /* note: a3 is a matrix with dimension w3_row x p_col */

  for (ii=1;ii<=w3_row;ii++)
    for (jj=1;jj<=p_col;jj++)
      e[ii][jj]=t[ii][jj]-a3[ii][jj];

  SSE = mysumsqr(e,w3_row,p_col);
  printf("Presentation Phase finished\n");
  printf("Initial SSE=%f\n",SSE);

/********** BackPropagation Phase *************************/
  d3=(*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL);
                                    /* e, a3, d3 are same dimensional */
  printf("Any problem with NULL (pointer) use?\n");
  d2=(*df2)(a2,w2_row,p_col,d3,w3_row,p_col,w3,w3_row,w3_col);
                                    /* d2,a2 same dimension */
  printf("Any problem with df2?\n");
  d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col);
                                    /* d1,a1 same dimension */

  printf("Error calculations ready\n");
  for (ii=1;ii<=me;ii++) {
    if (SSE < eg) { /* CHECK PHASE */
      ii--;
      break;
    }

    /* LEARNING PHASE */
    mylearnbpm(p,p_row,p_col,d1,w1_row,p_col,lr,MC,dw1,w1_row,w1_col,db1);
    mylearnbpm(a1,w1_row,p_col,d2,w2_row,p_col,lr,MC,dw2,w2_row,w2_col,db2);
    mylearnbpm(a2,w2_row,p_col,d3,w3_row,p_col,lr,MC,dw3,w3_row,w3_col,db3);

    MC=mc;
    for (jj=1;jj<=w1_row;jj++) { /* updating w1,b1 */
      new_b1[jj]=b1[jj]+db1[jj];
      for (kk=1;kk<=w1_col;kk++)
        new_w1[jj][kk]=w1[jj][kk]+dw1[jj][kk];
    }

    for (jj=1;jj<=w2_row;jj++) { /* updating w2,b2 */
      new_b2[jj]=b2[jj]+db2[jj];
      for (kk=1;kk<=w2_col;kk++)
        new_w2[jj][kk]=w2[jj][kk]+dw2[jj][kk];
    }

    for (jj=1;jj<=w3_row;jj++) { /* updating w3,b3 */
      new_b3[jj]=b3[jj]+db3[jj];
      for (kk=1;kk<=w3_col;kk++)
        new_w3[jj][kk]=w3[jj][kk]+dw3[jj][kk];
    }

    /* PRESENTATION PHASE */
    new_a1=(*f1)(new_w1,w1_row,w1_col,p,p_row,p_col,new_b1);
      /* note: new_a1 (as a1) is a matrix with dimension w1_row x p_col */
    new_a2=(*f2)(new_w2,w2_row,w2_col,new_a1,w1_row,p_col,new_b2);
      /* note: new_a2 (as a2) is a matrix with dimension w2_row x p_col */
    new_a3=(*f3)(new_w3,w3_row,w3_col,new_a2,w2_row,p_col,new_b3);
      /* note: new_a3 (as a3) is a matrix with dimension w3_row x p_col */
```

```
for (jj=1;jj<=w3_row;jj++)
  for (kk=1;kk<=p_col;kk++)
    new_e[jj][kk]=t[jj][kk]-new_a3[jj][kk];


new_SSE=mysumsqr(new_e,w3_row,p_col);
                      /* new_e (as e) with dimension w3_row x p_col */

/* Momentum and adaptive learning rate phase */
if (new_SSE > SSE * er) {
  lr=lr*dm;
  MC=0;
}
else {
  if (new_SSE < SSE) {
    lr=lr*im;
  }
  for (jj=1;jj<=w1_row;jj++) {
    b1[jj]=new_b1[jj];
    for (kk=1;kk<=w1_col;kk++) {
      w1[jj][kk]=new_w1[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a1[jj][kk]=new_a1[jj][kk];
    }
  }
  for (jj=1;jj<=w2_row;jj++) {
    b2[jj]=new_b2[jj];
    for (kk=1;kk<=w2_col;kk++) {
      w2[jj][kk]=new_w2[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a2[jj][kk]=new_a2[jj][kk];
    }
  }
  for (jj=1;jj<=w3_row;jj++) {
    b3[jj]=new_b3[jj];
    for (kk=1;kk<=w3_col;kk++) {
      w3[jj][kk]=new_w3[jj][kk];
    }
    for (kk=1;kk<=p_col;kk++) {
      a3[jj][kk]=new_a3[jj][kk];
      e[jj][kk] =new_e[jj][kk];
    }
  }

  SSE=new_SSE;

  /* BACKPROPAGATION PHASE */
  d3=(*df3)(a3,w3_row,p_col,e,w3_row,p_col,NULL,NULL,NULL);
                                      /* e, a3, d3 are same dimensional */
  d2=(*df2)(a2,w2_row,p_col,d3,w3_row,p_col,w3,w3_row,w3_col);
                                      /* d2,a2 same dimension */
  d1=(*df1)(a1,w1_row,p_col,d2,w2_row,p_col,w2,w2_row,w2_col);
                                      /* d1,a1 same dimension */
}

/* TRAINING RECORDS */
tr[1][ii+1]=SSE;
tr[2][ii+1]=lr;

if ((ii % df) == 0)
```

```
      printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);

  } /* end ii --- "for" loop */

  if ((ii % df) !=0) {/* This is for last training epoch printing
                         in case me is not multiples of df        */
    printf("Trainbpx: %d Epochs, lr=%f, SSE=%f\n",ii,lr,SSE);
  }

  if ( SSE > eg) {
    printf("Tranibpx: network error did not reach the error goal;\n");
    printf("Future training may be necessary or try different    \n");
    printf("initial weights and biases and/or more hidden neurons.\n");
  }
}


/********* mysumsqr *****************/
float mysumsqr(float **e,int e_row,int e_col)
{
  int ii,jj;
  float tmp;
  tmp=0;
  for (ii=1;ii<=e_row;ii++)
    for (jj=1;jj<=e_col;jj++)
      tmp += e[ii][jj] * e[ii][jj];

  return (tmp);
}

/********* mylearnbpm ***************/
void mylearnbpm(float **p,int p_row,int p_col,
                float **d,int d_row,int d_col,
                float lr,float mc,
                float **dw,int dw_row,int dw_col,
                float *db)

{
  float ** allocate_real_matrix(int,int,int,int);
  float **x;
  int ii,jj,kk;
  float tmp,tmp1;

  x=allocate_real_matrix(1, d_row, 1, d_col);
  if ((p_col==d_col) & (d_row==dw_row) & (dw_col==p_row)) {
    for (ii=1;ii<=d_row;ii++) /* x=(1-mc)*lr*d */
      for (jj=1;jj<=d_col;jj++)
        x[ii][jj]=(1-mc)*lr*d[ii][jj];

    for (ii=1;ii<=dw_row;ii++) { /* dw=mc*dw+x*p' */
      for (jj=1;jj<=dw_col;jj++) {
        tmp=0.0;
        for (kk=1;kk<=d_col;kk++) {
          tmp += x[ii][kk]*p[jj][kk];
        }
        dw[ii][jj]=mc*dw[ii][jj]+tmp;
      }

      tmp1=0.0;
      for (kk=1;kk<=d_col;kk++) { /* db=mc*db+x*ones(Q,1) */
        tmp1 += x[ii][kk]*1;
      }
```

```
      db[ii]=mc*db[ii] + tmp1;
    } /* end ii-loop */
    free_real_matrix(x,1,d_row,1);
  }
  else {
    printf("Error ------ Dimensions are not consistent\n");
    printf("Quiting from the subroutine mylearnbpm\n");
    free_real_matrix(x,1,d_row,1);
    exit(1);
  }
}

/********* END of mytbpx3.c ***********/

/********* BEGIN of rand_gen.c ********/
/* These subroutines are for generating random numbers
   and doing some normalizing operations, etc.        */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

void mynwlog(int s,float ** p,int p_row,int p_col,float ** w,
             int w_row,int w_col,float *b)
{
  int ii,jj;
  float magw,tmp;
  float *tmp_min,*tmp_max;
  float *rng,*mid;
  tmp_min=allocate_real_vector(1,p_row);
  tmp_max=allocate_real_vector(1,p_row);
  rng    =allocate_real_vector(1,p_row);
  mid    =allocate_real_vector(1,p_row);
  for (ii=1;ii<=p_row;ii++) {
    tmp_min[ii]=p[ii][1];
    tmp_max[ii]=p[ii][1];
    for (jj=2;jj<=p_col;jj++) {
      if (p[ii][jj] < tmp_min[ii])
        tmp_min[ii]=p[ii][jj];
      if (p[ii][jj] > tmp_max[ii])
        tmp_max[ii]=p[ii][jj];
    }
  }

  magw=2.8*pow(s,1/p_row);
/*************************  Replaced
  myrandnr(w,w_row,w_col);
  myrands(b,s,1);
*************************/
  myrands(w,w_row,w_col,b);
  mynormr(w,w_row,w_col);   /* normalize row */

  for (ii=1;ii<=p_row;ii++) {
    rng[ii]=tmp_max[ii]-tmp_min[ii];
    mid[ii]=(tmp_max[ii]+tmp_min[ii])/2.0;
  }

  for (ii=1;ii<=w_row;ii++)
    for (jj=1;jj<=w_col;jj++)
      w[ii][jj]=2*magw*w[ii][jj] / rng[jj];

  for (ii=1;ii<=w_row;ii++) {
```

```
    tmp=0.0;
    for (jj=1;jj<=w_col;jj++)
      tmp += w[ii][jj]*mid[jj];
    b[ii] = magw*b[ii] - tmp;
  }

  free_real_vector(tmp_min,1);
  free_real_vector(tmp_max,1);
  free_real_vector(rng,1);
  free_real_vector(mid,1);
}
/************ mynwtan ***************/
void mynwtan(int s,float ** p,int p_row,int p_col,float ** w,
             int w_row,int w_col,float *b)
{
  int ii,jj;
  float magw,tmp;
  float *tmp_min,*tmp_max;
  float *rng,*mid;
  tmp_min=allocate_real_vector(1,p_row);
  tmp_max=allocate_real_vector(1,p_row);
  rng    =allocate_real_vector(1,p_row);
  mid    =allocate_real_vector(1,p_row);
  for (ii=1;ii<=p_row;ii++) {
    tmp_min[ii]=p[ii][1];
    tmp_max[ii]=p[ii][1];
    for (jj=2;jj<=p_col;jj++) {
      if (p[ii][jj] < tmp_min[ii])
        tmp_min[ii]=p[ii][jj];
      if (p[ii][jj] > tmp_max[ii])
        tmp_max[ii]=p[ii][jj];
    }
  }

  magw=0.7*pow(s,1/p_row);
/************************** Replaced
  myrandnr(w,w_row,w_col);
  myrands(b,s,1);
**************************/
  myrands(w,w_row,w_col,b);
  mynormr(w,w_row,w_col);  /* normalize row */

  for (ii=1;ii<=p_row;ii++) {
    rng[ii]=tmp_max[ii]-tmp_min[ii];
    mid[ii]=(tmp_max[ii]+tmp_min[ii])/2.0;
  }

  for (ii=1;ii<=w_row;ii++)
    for (jj=1;jj<=w_col;jj++)
      w[ii][jj]=2*magw*w[ii][jj] / rng[jj];

  for (ii=1;ii<=w_row;ii++) {
    tmp=0.0;
    for (jj=1;jj<=w_col;jj++)
      tmp += w[ii][jj]*mid[jj];
    b[ii] = magw*b[ii] - tmp;
  }

  free_real_vector(tmp_min,1);
  free_real_vector(tmp_max,1);
  free_real_vector(rng,1);
  free_real_vector(mid,1);
}
```

```
/*********** myrands ***************/
void myrands(float **w, int w_row,int w_col,float *b)
{
  int ii,jj;
  myrand(w,w_row,w_col,b);    /* generate random number in [ 0,+1] */
  for (ii=1;ii<=w_row;ii++) {/* generate random number in [-1,+1] */
    for (jj=1;jj<=w_col;jj++)
      w[ii][jj]=2*w[ii][jj]-1;
    b[ii]=2*b[ii]-1;
  }
}

void mynormr(float **w, int w_row, int w_col) /* row normalization */
{
  int ii,jj;
  float tmp;
  for (ii=1;ii<=w_row;ii++) {
    tmp=0.0;
    for (jj=1;jj<=w_col;jj++)
      tmp += w[ii][jj]*w[ii][jj];
    for (jj=1;jj<=w_col;jj++)
      w[ii][jj]=w[ii][jj]/sqrt(tmp);
  }
}

void myrand(float **w,int w_row,int w_col,float *b)
          /* generate random number [ 0,+1] */
{
  int ii,jj;
  static long idum=-11; /* idum can be changed for initialization */


  for (ii=1;ii<=1000;ii++) rand_num(&idum);

  for (ii=1;ii<=w_row;ii++) {
    for (jj=1;jj<=w_col;jj++)
      w[ii][jj]=rand_num(&idum);
    b[ii]=rand_num(&idum);
  }
}

/****************** rand_num ***************************/
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float rand_num(long *idum)
/*************************************************************************
   Minimal random number generator of Park and Miller with Bays-Durham shuffle
   and added safeguards. Returns a uniform random deviate between 0.0 and 1.0
   (exclusive of the endpoint values). Call with a negative integer to
   initialize; thereafter, do not alter idum between successive deviates in a
   sequence. RNMX should approximate the largest floating value that is less
   than 1.
*************************************************************************/
{
```

```
  int j;
  long k;
  static long iy=0;
  static long iv[NTAB];
  float temp;

  if (*idum <=0 || !iy) { /* Initialize */
    if (-(*idum) < 1) *idum=1; /* Be sure to prevent idum=0 */
    else *idum=-(*idum);
    for (j=NTAB+7;j>=0;j--) {
      k=(*idum)/IQ;
      *idum=IA*(*idum-k*IQ)-IR*k;
      if (*idum < 0) *idum += IM;
      if (j < NTAB) iv[j] = *idum;
    }
    iy=iv[0];
  }
  k=(*idum)/IQ; /* start here when not initializing */
  *idum=IA*(*idum-k*IQ)-IR*k; /* compute idum=(IA*idum) % M without overflows
                                 by Schrage's method */
  if (*idum < 0) *idum += IM;
  j=iy/NDIV; /* Will be in the range 0..NTAB-1. */
  iy=iv[j];  /* Output previously stored value and refill the shuffle table */
  iv[j]= *idum;
  if ((temp=AM*iy) > RNMX) return RNMX;
                          /* Because users don't expect endpoint values */
  else return temp;
}

/***************** END of rand_gen.c ***************/

/***************** BEGIN of sigm_deriv.c ************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

/* sigmoidal functions: pure-linear function,logistic sigmoid,tan-sigmoid */
float ** mypurelin(float **w,int row,int col,
                   float **x,int x_row,int x_col,
                   float *b)
                   /* a = mypurelin(w*x,b) */
                   /* w*x - SxQ Matrix of weighted input (column) vectors. */
           /* b   - Sx1 Bias (column) vector. */
{
  float **a;
  float tmp;
  int ii,jj,kk;
  if (col != x_row) {
    printf("Inconsistent dimensions in mypurelin\n");
    printf(" Exiting from mypurelin\n");
    exit(1);
  }
  a=allocate_real_matrix(1,row,1,x_col);
  for (ii=1;ii<=row;ii++) {
    for (jj=1;jj<=x_col;jj++) {
      tmp=0;
      for (kk=1;kk<=col;kk++) {
        tmp += w[ii][kk]*x[kk][jj];
      }
      a[ii][jj]=tmp + b[ii];
                /* n=w*x;                */
                /* n=n+b*ones(1,x_col); */
```

```
    }
  }
  return (a);
}


float ** mylogsig(float **w,int row,int col,
                  float **x,int x_row,int x_col,
                  float *b)
                   /* a = mylogsig(w*x,b) */
                   /* w*x - SxQ Matrix of weighted input (column) vectors. */
          /* b   - Sx1 Bias (column) vector. */
{
  float **a;
  float tmp;
  int ii,jj,kk;
  if (col != x_row) {
    printf("Inconsistent dimensions in mylogsig\n");
    printf(" Exiting from mylogsig\n");
    exit(1);
  }
  a=allocate_real_matrix(1,row,1,x_col);
  for (ii=1;ii<=row;ii++) {
    for (jj=1;jj<=x_col;jj++) {
      tmp=0;
      for (kk=1;kk<=col;kk++) {
        tmp += w[ii][kk]*x[kk][jj];
      }
      a[ii][jj]=tmp + b[ii];
      a[ii][jj]=1/(1+exp(-a[ii][jj]));
                           /* n=n+b*ones(1,x_col); a=1./(1+exp(-n)); */
    }
  }
  return (a);
}

float ** mytansig(float **w,int row,int col,
                  float **x,int x_row,int x_col,
                  float *b)
                   /* a = mytansig(w*x,b) */
                   /* w*x - SxQ Matrix of weighted input (column) vectors. */
           /* b   - Sx1 Bias (column) vector. */
{
  float **a;
  float tmp;
  int ii,jj,kk;
  if (col != x_row) {
    printf("Inconsistent dimensions in mytansig\n");
    printf(" Exiting from mytansig\n");
    exit(1);
  }
  a=allocate_real_matrix(1,row,1,x_col);
  for (ii=1;ii<=row;ii++) {
    for (jj=1;jj<=x_col;jj++) {
      tmp=0;
      for (kk=1;kk<=col;kk++) {
        tmp += w[ii][kk]*x[kk][jj];
      }
      a[ii][jj]=tmp + b[ii];
      tmp=a[ii][jj];
      a[ii][jj]=2.0/(1.0+exp(-2.0*tmp))-1.0;
                              /* n=n+b*ones(1,x_col); a=2./(1+exp(-2*n))-1; */
    }
```

```
  }
  return (a);
}


/* Derivatives for pure-linear functions, logistic sigmoid, tan-sigmoid */
float ** mydeltalin(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col)
{
  float ** r;
  int ii,jj,kk;

  if (d==NULL && w==NULL) {
    r=allocate_real_matrix(1,a_row,1,a_col);
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=1;
  }

  if (d != NULL && w==NULL) {
    r=allocate_real_matrix(1,d_row,1,d_col);
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=d[ii][jj];
  }

  if (d != NULL && w != NULL) {
    if (d_row !=w_row) {
      printf(" Inconsistent dimensions in mydeltalin\n");
      printf(" Exiting from mydeltalin\n");
      exit(1);
    }
    r=allocate_real_matrix(1,w_col,1,d_col);
    for (ii=1;ii<=w_col;ii++)   /* r = w' * d */
      for (jj=1;jj<=d_col;jj++) {
        r[ii][jj]=0.0;
        for (kk=1;kk<=w_row;kk++)
          r[ii][jj] += w[kk][ii]*d[kk][jj];
      }
  }

  return (r);
}

float ** mydeltalog(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col)
{
  float ** r;
  int ii,jj,kk;

  if (d==NULL && w==NULL) {
    r=allocate_real_matrix(1,a_row,1,a_col);
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=a[ii][jj] * (1 - a[ii][jj]);
  }

  if (d != NULL && w==NULL) {
    r=allocate_real_matrix(1,a_row,1,a_col);
    for (ii=1;ii<=a_row;ii++)
```

```
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=a[ii][jj]*(1-a[ii][jj])*d[ii][jj];
  }

  if (d != NULL && w != NULL) {
    if (d_row !=w_row) {
      printf(" Inconsistent dimensions in mydeltalog\n");
      printf(" Exiting from mydeltalog\n");
      exit(1);
    }
    r=allocate_real_matrix(1,a_row,1,a_col);
    for (ii=1;ii<=w_col;ii++)   /* r = a.*(1-a).*(w' * d) */
      for (jj=1;jj<=d_col;jj++) {
        r[ii][jj]=0.0;
        for (kk=1;kk<=w_row;kk++)
          r[ii][jj] += w[kk][ii]*d[kk][jj];
        r[ii][jj] *= a[ii][jj]*(1-a[ii][jj]);
      }
  }

  return (r);
}


float ** mydeltatan(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col)
{
  float ** r;
  int ii,jj,kk;

  if (d==NULL && w==NULL) {
    r=allocate_real_matrix(1,a_row,1,a_col);
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=1 - a[ii][jj]*a[ii][jj];
  }

  if (d != NULL && w==NULL) {
    r=allocate_real_matrix(1,a_row,1,a_col);
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=(1-a[ii][jj]*a[ii][jj])*d[ii][jj];
  }

  if (d != NULL && w != NULL) {
    if (d_row !=w_row) {
      printf(" Inconsistent dimensions in mydeltatan\n");
      printf(" Exiting from mydeltatan\n");
      exit(1);
    }

    r=allocate_real_matrix(1,a_row,1,a_col);

    for (ii=1;ii<=a_row;ii++)   /* r = (1-a.*a).*(w' * d) */
      for (jj=1;jj<=a_col;jj++) {
        r[ii][jj]=0.0;
        for (kk=1;kk<=w_row;kk++)
          r[ii][jj] += w[kk][ii]*d[kk][jj];
        r[ii][jj] *= (1-a[ii][jj]*a[ii][jj]);
      }
  }
  return (r);
```

```
}


/***** END of sigmoidal functions and its derivatives (sigm_deriv.c) *******/

/***** BEGIN of sigm_deriv_rnn.c ***********************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

/* sigmoidal functions: pure-linear function,logistic sigmoid,tan-sigmoid */
void mypurelin(float **w,int row,int col,
                   float **x,int x_row,int x_col,
                   float *b,float **a)
                   /* a = mypurelin(w*x,b) */
                   /* w*x - SxQ Matrix of weighted input (column) vectors. */
           /* b   - Sx1 Bias (column) vector. */
{
  /*float **a;*/
  float tmp;
  int ii,jj,kk;
  if (col != x_row) {
    printf("Inconsistent dimensions in mypurelin\n");
    printf(" Exiting from mypurelin\n");
    exit(1);
  }
  /*a=allocate_real_matrix(1,row,1,x_col);*/
  for (ii=1;ii<=row;ii++) {
    for (jj=1;jj<=x_col;jj++) {
      tmp=0;
      for (kk=1;kk<=col;kk++) {
        tmp += w[ii][kk]*x[kk][jj];
      }
      a[ii][jj]=tmp + b[ii];
                /* n=w*x;               */
                /* n=n+b*ones(1,x_col); */
    }
  }
  /*return (a);*/
}


void mylogsig(float **w,int row,int col,
                   float **x,int x_row,int x_col,
                   float *b,float **a)
                    /* a = mylogsig(w*x,b) */
                    /* w*x - SxQ Matrix of weighted input (column) vectors. */
           /* b   - Sx1 Bias (column) vector. */
{
  /*float **a;*/
  float tmp;
  int ii,jj,kk;
  if (col != x_row) {
    printf("Inconsistent dimensions in mylogsig\n");
    printf(" Exiting from mylogsig\n");
    exit(1);
  }
  /*a=allocate_real_matrix(1,row,1,x_col);*/
  for (ii=1;ii<=row;ii++) {
    for (jj=1;jj<=x_col;jj++) {
```

```
      tmp=0;
      for (kk=1;kk<=col;kk++) {
        tmp += w[ii][kk]*x[kk][jj];
      }
      a[ii][jj]=tmp + b[ii];
      a[ii][jj]=1/(1+exp(-a[ii][jj]));
                          /* n=n+b*ones(1,x_col); a=1./(1+exp(-n)); */
    }
  }
  /*return (a);  */
}

void mytansig(float **w,int row,int col,
                  float **x,int x_row,int x_col,
                  float *b,float **a)
                   /* a = mytansig(w*x,b) */
                   /* w*x - SxQ Matrix of weighted input (column) vectors. */
           /* b   - Sx1 Bias (column) vector. */
{
  /*float **a;*/
  float tmp;
  int ii,jj,kk;
  if (col != x_row) {
    printf("Inconsistent dimensions in mytansig\n");
    printf(" Exiting from mytansig\n");
    exit(1);
  }
  /*a=allocate_real_matrix(1,row,1,x_col);*/
  for (ii=1;ii<=row;ii++) {
    for (jj=1;jj<=x_col;jj++) {
      tmp=0;
      for (kk=1;kk<=col;kk++) {
        tmp += w[ii][kk]*x[kk][jj];
      }
      a[ii][jj]=tmp + b[ii];
      tmp=a[ii][jj];
      a[ii][jj]=2.0/(1.0+exp(-2.0*tmp))-1.0;
                       /* n=n+b*ones(1,x_col); a=2./(1+exp(-2*n))-1; */
    }
  }
  /*return (a);  */
}


void mydeltalin(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col,float **r)

{
  /*float ** r;*/
  int ii,jj,kk;

  if (d==NULL && w==NULL) {
    /*r=allocate_real_matrix(1,a_row,1,a_col);*/
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=1;
  }

  if (d != NULL && w==NULL) {
    /*r=allocate_real_matrix(1,d_row,1,d_col);*/
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
```

```
      r[ii][jj]=d[ii][jj];
  }

  if (d != NULL && w != NULL) {
    if (d_row !=w_row) {
      printf(" Inconsistent dimensions in mydeltalin\n");
      printf(" Exiting from mydeltalin\n");
      exit(1);
    }
    /*r=allocate_real_matrix(1,w_col,1,d_col);*/
    for (ii=1;ii<=w_col;ii++)   /* r = w' * d */
      for (jj=1;jj<=d_col;jj++) {
        r[ii][jj]=0.0;
        for (kk=1;kk<=w_row;kk++)
          r[ii][jj] += w[kk][ii]*d[kk][jj];
      }
  }

  /*return (r);*/
}

void mydeltalog(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col,float **r)
{
  /*float ** r;*/
  int ii,jj,kk;

  if (d==NULL && w==NULL) {
    /*r=allocate_real_matrix(1,a_row,1,a_col);*/
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=a[ii][jj] * (1 - a[ii][jj]);
  }

  if (d != NULL && w==NULL) {
    /*r=allocate_real_matrix(1,a_row,1,a_col);*/
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=a[ii][jj]*(1-a[ii][jj])*d[ii][jj];
  }

  if (d != NULL && w != NULL) {
    if (d_row !=w_row) {
      printf(" Inconsistent dimensions in mydeltalog\n");
      printf(" Exiting from mydeltalog\n");
      exit(1);
    }
    /*r=allocate_real_matrix(1,a_row,1,a_col);*/
    for (ii=1;ii<=w_col;ii++)   /* r = a.*(1-a).*(w' * d) */
      for (jj=1;jj<=d_col;jj++) {
        r[ii][jj]=0.0;
        for (kk=1;kk<=w_row;kk++)
          r[ii][jj] += w[kk][ii]*d[kk][jj];
        r[ii][jj] *= a[ii][jj]*(1-a[ii][jj]);
      }
  }

  /*return (r);*/
}
```

```
void mydeltatan(float **a,int a_row,int a_col,
                    float **d,int d_row,int d_col,
                    float **w,int w_row,int w_col,float **r)
{
  /*float ** r;*/
  int ii,jj,kk;

  if (d==NULL && w==NULL) {
    /*r=allocate_real_matrix(1,a_row,1,a_col);*/
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=1 - a[ii][jj]*a[ii][jj];
  }

  if (d != NULL && w==NULL) {
    /*r=allocate_real_matrix(1,a_row,1,a_col);*/
    for (ii=1;ii<=a_row;ii++)
      for (jj=1;jj<=a_col;jj++)
        r[ii][jj]=(1-a[ii][jj]*a[ii][jj])*d[ii][jj];
  }

  if (d != NULL && w != NULL) {
    if (d_row !=w_row) {
      printf(" Inconsistent dimensions in mydeltatan\n");
      printf(" Exiting from mydeltatan\n");
      exit(1);
    }

    /*r=allocate_real_matrix(1,a_row,1,a_col);*/

    for (ii=1;ii<=a_row;ii++)   /* r = (1-a.*a).*(w' * d) */
      for (jj=1;jj<=a_col;jj++) {
        r[ii][jj]=0.0;
        for (kk=1;kk<=w_row;kk++)
          r[ii][jj] += w[kk][ii]*d[kk][jj];
        r[ii][jj] *= (1-a[ii][jj]*a[ii][jj]);
      }
  }
  /*return (r);*/
}



/***** END of sigmoidal functions and its derivatives (sigm_deriv_rnn.c) *****/

/***** BEGIN of memo_man.c ************************************************/
/* These utilities are used for dynamic memory management */

#include <stdio.h>
#include <stdlib.h>
#include "mynn.h"

void system_error(char error_message[])
{
  void exit(int);
  printf("%s",error_message);
  exit(1);
}

int *allocate_integer_vector(int l, int u)
{
  /* allocates an integer vector of range [l..u] */
```

```
  void system_error(char *);
  int *p;

  p=(int *)malloc((unsigned) (u-l+1)*sizeof(int));
  if (!p) system_error("Failure in allocate_integer_vector().");
  return p-l;
}

float *allocate_real_vector(int l, int u)
{
  /* allocate a real vector of range [l..u] */
  void system_error(char *);
  float *p;

  p=(float *)malloc((unsigned) (u-l+1)*sizeof(float));
  if (!p) system_error("Failure in allocate_real_vector().");
  return p-l;
}

int **allocate_integer_matrix(int lr, int ur, int lc, int uc)
{
  /* allocate an integer matrix of range [lr..ur][lc..uc] */
  void system_error(char *);
  int i, **p;

  p=(int **)malloc((unsigned) (ur-lr+1)*sizeof(int *));
  if (!p) system_error("Failure in allocate_integer_matrix().");
  p -= lr;

  for (i=lr;i<=ur;i++) {
    p[i]=(int *)malloc((unsigned) (uc-lc+1)*sizeof(int));
    if (!p[i]) system_error("Failure in allocate_integer_matrix().");
    p[i] -= lc;
  }
  return p;
}

float **allocate_real_matrix(int lr, int ur, int lc, int uc)
{
  /* allocate a real matrix of range [lr..ur][lc..uc] */
  void system_error(char *);
  int i;
  float **p;

  p=(float **)malloc((unsigned) (ur-lr+1)*sizeof(float *));
  if (!p) system_error("Failure in allocate_real_matrix().");
  p -= lr;

  for (i=lr;i<=ur;i++) {
    p[i]=(float *)malloc((unsigned) (uc-lc+1)*sizeof(float));
    if (!p[i]) system_error("Failure in allocate_real_matrix().");
    p[i] -= lc;
  }
  return p;
}

void free_integer_vector(int *v, int l)
{
  /* free an integer vector of range [l..u] */
  free((char *) (v+l));
}

void free_real_vector(float *v, int l)
```

```
{
  /* free a real vector of range [l..u] */
  free((char *) (v+l));
}

void free_integer_matrix(int **m, int lr, int ur, int lc)
{
  /* free an integer matrix of range [lr..ur][lc..uc] */
  int i;

  for (i=ur; i>=lr; i--) free((char *) (m[i]+lc));
  free((char *) (m+lr));
}

void free_real_matrix(float **m, int lr, int ur,int lc)
{
  /* free a real matrix of range [lr..ur][lc..uc]. */
  int i;

  for (i=ur; i>=lr; i--) free((char *) (m[i]+lc));
  free((char *) (m+lr));
}

\input{../../C_nnet/learn_bp.c}
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mynn.h"

/********* mysumsqr ******************/
float mysumsqr(float **e,int e_row,int e_col)
{
  int ii,jj;
  float tmp;
  tmp=0;
  for (ii=1;ii<=e_row;ii++)
    for (jj=1;jj<=e_col;jj++)
      tmp += e[ii][jj] * e[ii][jj];

  return (tmp);
}

/********* mylearnbpm ****************/
void mylearnbpm(float **p,int p_row,int p_col,
                float **d,int d_row,int d_col,
                float lr,float mc,
                float **dw,int dw_row,int dw_col,
                float *db)

{
  /*float ** allocate_real_matrix(int,int,int,int);*/
  float **x;
  int ii,jj,kk;
  float tmp,tmp1;

  /*x=allocate_real_matrix(1, d_row, 1, d_col);*/
  if ((p_col==d_col) && (d_row==dw_row) && (dw_col==p_row)) {
    x=allocate_real_matrix(1, d_row, 1, d_col);
    for (ii=1;ii<=d_row;ii++) /* x=(1-mc)*lr*d */
      for (jj=1;jj<=d_col;jj++)
        x[ii][jj]=(1-mc)*lr*d[ii][jj];
```

```
    for (ii=1;ii<=dw_row;ii++) { /* dw=mc*dw+x*p' */
      for (jj=1;jj<=dw_col;jj++) {
        tmp=0.0;
        for (kk=1;kk<=d_col;kk++) {
          tmp += x[ii][kk]*p[jj][kk];
        }
        dw[ii][jj]=mc*dw[ii][jj]+tmp;
      }

      tmp1=0.0;
      for (kk=1;kk<=d_col;kk++) { /* db=mc*db+x*ones(Q,1) */
        tmp1 += x[ii][kk]*1;
      }
      db[ii]=mc*db[ii] + tmp1;
    } /* end ii-loop */
    free_real_matrix(x,1,d_row,1);
  }
  else {
    printf("Error ------ Dimensions are not consistent\n");
    printf("Quiting from the subroutine mylearnbpm\n");
    free_real_matrix(x,1,d_row,1);
    exit(1);
  }
}

/********* END of learn_bp.c ***********/
```

B About the singular solution

It is shown in the following that there is no singular solution to the Hamiltonian system consisting of Equations (5.3) and (5.18).

Proof: Suppose otherwise. That is, there exists a time interval $I(T_0, T_1) = [T_0,\ T_0 + T_1]$ such that $\lambda_2(t)\sin\delta(t) \equiv 0$ for $\forall t \in I(T_0, T_1)$ where T_0 and T_1 are two finite positive numbers.

Note again the time-optimal control (5.22) $u^* = \begin{cases} u_{max}, & \lambda_2 \sin\delta > 0 \\ u_{min}, & \lambda_2 \sin\delta < 0 \end{cases}$. Since for $\lambda_2 \sin\delta = 0$, it does not matter what value u^* may take, let us assume $u^* = 0$.

Let $u = u^*$ in Equation (5.3). It follows immediately that the time solution $(\delta(t),\ \omega(t))$ is continuous. Similarly the time solutioin $(\lambda_1(t),\ \lambda_2(t))$ to Equation (5.18) is also continuous. These facts and the assumption $\lambda_2(t)\sin\delta(t) \equiv 0$ for $\forall t \in I(T_0,\ T_1)$ imply that there must exist an interval $I(T_2, T_3) = [T_2,\ T_2 + T_3] \subseteq I(T_0, T_1)$ such that either $\lambda_2(t) \equiv 0$ or $\sin\delta(t) \equiv 0$ for $\forall t \in I(T_2, T_3)$.

Consider the case $\lambda_2(t) \equiv 0$ for $\forall t \in I(T_2, T_3)$. It is apparent that $\lambda_1(t) \equiv 0$ for $T_2 < t < T_3$. It follows from Equation (5.18) that $(\lambda_1(t),\ \lambda_2(t)) \equiv (0,\ 0)$ for $T_2 < t < T_3$.

Notice that $(0,\ 0)$ for all t is also a solution. Since Equation (5.18) satisfies the Lipschitz condition, the solution must be unique. This in turn leads to the following implication:

$$(\lambda_1(t),\ \lambda_2(t)) \equiv (0,\ 0) \text{ for } T_2 < t < T_3 \Longrightarrow (\lambda_1(t),\ \lambda_2(t)) \equiv (0,\ 0) \text{ for } T_3 \leq t.$$

In particular, $\lambda_2(T) = 0$ where T is the finite terminal time. This, however, contradicts Equation (5.25). Therefore, $\lambda_2(t)$ must not be zero for any finite time interval.

Consider the case $\sin\delta(t) \equiv 0$ for $\forall t \in I(T_2, T_3)$. Since $\delta(t)$ is continuous, it follows immediately that $\delta(t) \equiv C$ for $\forall t \in I(T_2, T_3)$ where C is a constant value. This in turn

leads Equation (5.3) to the following contradiction:

$$\begin{cases} 0 = \omega_b \omega \\ \dot{\omega} = \frac{1}{M}(P_m - D\omega) \end{cases} \tag{7.1}$$

Therefore, $\sin\delta(t)$ must not be zero for any finite time interval.

By contradiction, we have shown that $\lambda_2(t)\sin\delta(t)$ must not be zero for any finite time interval, which implies that there is no singular solution to the Hamiltonian system in question. This completes the proof.

C About parameters updating

It is shown in the following that with the modified updating laws (6.51) and (6.52), for $\forall t \geq 0$, $\dot{W} \leq -\gamma|e|^2$, and that $\hat{p}(t) \in \pi$, $0 \leq \hat{\alpha}(t) \leq 1$ with $\sum_{j=1}^{J} \hat{\alpha}(t) = 1$ provided that $\hat{p}(0) \in \pi$, $p(t) \in \pi$, $0 \leq \hat{\alpha}(0) \leq 1$ with $\sum_{j=1}^{J} \hat{\alpha}(0) = 1$, and $0 \leq \hat{\alpha}^* \leq 1$ with $\sum_{j=1}^{J} \hat{\alpha}^* = 1$.

Proof: First, we show that $\hat{p}(t) \in \pi$ provided that $\hat{p}(0) \in \pi$ and $p(t) \in \pi$.

To show that $\hat{p}(t) \in \pi$, we only need to show that for $\forall t \geq 0$, $\hat{p}^i \in [p^i_{min}, p^i_{max}]$, which can be verified by examination of the sign of $\dot{\hat{p}}^i$ when $\hat{p}^i$ reaches the boundary of $[p^i_{min}, p^i_{max}]$.

For $\hat{p}^i = p^i_{min}$, the updating law is given by

$$\dot{\hat{p}}^i = \begin{cases} -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i & \text{if } \hat{p}^i = p^i_{min} \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \leq 0 \\ 0 & \text{if } \hat{p}^i = p^i_{min} \text{ and } [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i > 0 \end{cases} \tag{7.1}$$

Hence when $\hat{p}^i = p^i_{min}$, $\dot{\hat{p}}^i \geq 0$, which implies that $\hat{p}^i$ is directed towards the interior of $[p^i_{min}, p^i_{max}]$.

Similarly, it can be readily shown that for $\hat{p}^i = p^i_{max}$, $\dot{\hat{p}}^i \leq 0$, which implies that $\hat{p}^i$ is directed towards the interior of $[p^i_{min}, p^i_{max}]$.

For $\hat{p}^i(0) \in (p^i_{min}, p^i_{max})$, the updating law keeps the $\hat{p}^i(t)$ still in the region $[p^i_{min}, p^i_{max}]$ before it reaches the boundary.

Therefore, if $\hat{p}^i(0) \in [p^i_{min}, p^i_{max}]$ and $\hat{p}^{*i} \in [p^i_{min}, p^i_{max}]$, then for $\forall t \geq 0$, $\hat{p}^i(t) \in [p^i_{min}, p^i_{max}]$. This in turn implies that if $\hat{p}(0) \in \pi$ and $p^* \in \pi$, $\hat{p} \in \pi$.

In the same way, one can show that if $0 \leq \hat{\alpha}(0) \leq 1$ with $\sum_{j=1}^{J} \hat{\alpha}(0) = 1$, and $0 \leq \hat{\alpha}^* \leq 1$ with $\sum_{j=1}^{J} \hat{\alpha}^* = 1$, then for $\forall t \geq 0$, $0 \leq \hat{\alpha}(t) \leq 1$ with $\sum_{j=1}^{J} \hat{\alpha}(t) = 1$.

Next, we prove that the modified updating law (6.51) and (6.52) can only make $\dot{W}$ more negative. Note that since for $\forall t \geq 0$, $\hat{p}^i(t) \in [p^i_{min}, p^i_{max}]$, then Equation (6.48) holds. First, we show that the modified updating law (6.51) can only make $\dot{W}$ more negative.

Consider Equation (6.48).

If $\hat{p}^i \in (p^i_{main},\ p^i_{max})$, or $\hat{p}^i = p^i_{min}$ and $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i \leq 0$, or $\hat{p}^i = p^i_{max}$ and $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$

≥ 0, from the updating law $\dot{\hat{p}}^i = -[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i$, we have $\{\dot{\hat{p}}^i + [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\}\tilde{p}^i = 0$.

If $\hat{p}^i = p^i_{min}$ and $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i > 0$, from the updating law $\dot{\hat{p}}^i = 0$ and $\tilde{p}^i = \hat{p}^i - p^i(t)$, we have $\{\dot{\hat{p}}^i + [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\}\tilde{p}^i = (\hat{p}^i - p^i(t))\{[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\} = (\hat{p}^i_{min} - p^i(t))\{[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\} \leq 0$.

If $\hat{p}^i = p^i_{max}$ and $[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i < 0$, from the updating law $\dot{\hat{p}}^i = 0$ and $\tilde{p}^i = \hat{p}^i - p^i(t)$, we have $\{\dot{\hat{p}}^i + [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\}\tilde{p}^i = (\hat{p}^i - p^i(t))\{[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\} = (\hat{p}^i_{max} - p^i(t))\{[\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\} \leq 0$.

In a word, $\{\dot{\hat{p}}^i + [\gamma(\frac{\partial h}{\partial x}\psi(x,u))^\tau e]^i\}\tilde{p}^i \leq 0$.

Thus, $\dot{W} \leq e^\tau[-re + \frac{\partial h}{\partial x}g_0(x)\sum_{j=1}^J \tilde{\alpha}^j u(x,p^*_j)] + \tilde{\alpha}^\tau\dot{\tilde{\alpha}}$.

Similarly, it can be shown that with the modified updating law (6.52), the following inequality holds, that is, $e^\tau\frac{\partial h}{\partial x}g_0(x)\sum_{j=1}^J \tilde{\alpha}^j u(x,p^*_j) + \tilde{\alpha}^\tau\dot{\tilde{\alpha}} \leq 0$.

Therefore, $\dot{W} \leq -r|e|^2$. This completes the proof.