FITTING FUNCTIONS AND THEIR DERIVATIVES

WITH NEURAL NETWORKS

By

ARJPOLSON PUKRITTAYAKAMEE

Bachelor of Engineering
Chulalongkorn University
Bangkok, Thailand
1997

Master of Sciences
Oklahoma State University
Stillwater, Oklahoma
2001

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
July, 2009

FITTING FUNCTIONS AND THEIR DERIVATIVES

WITH NEURAL NETWORKS

Dissertation Approved:

Dr. Martin T. Hagan

Dissertation Adviser

Dr. Lionel M. Raff

Dr. Carl D. Latino

Dr. George Scheets

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGEMENT

I would like to express my deep gratitude to my advisor, Professor Dr. Martin Hagan, for his dedication, support, guidance, patience and friendship throughout my graduate studies. His invaluable feedback and suggestions infinitely contributed to my research. I also would like to thank my other committee members, Professor Dr. Lionel Raff, Professor Dr. George Scheets and Professor Dr. Carl Latino, for their useful comments and suggestions.

I also thank Professor Dr. Ranga Komanduri, Professor Dr. Satish Bukkapatnam, Professor Dr. Paras Agrawal and Professor Dr. Mark Rockley for providing helpful comments and all supports possible to the research. My special appreciation goes to Professor Dr. Lionel Raff for his dedication to teaching me the fundamental of quantum mechanics and molecular dynamics.

The research is funded by a grant (DMI-0457663) from the National Science Foundation, Division of Mechanical and Manufacturing Innovation (CMMI). I thank Dr. Jocelyn Harrison, Program Director for Materials Processing and Manufacturing, for the interest and support of this work.

Most importantly, I especially thank and dedicate this dissertation to my parents, my grandmother and my sister for their loving support to all my endeavors and their strong encouragement at difficult times.

I also would like to take this opportunity to express my gratitude to all of my teachers at Deves Thampirak School, St. Gabriel's College, Patumwan Demonstration School and Triam Udom Suksa School, and Professors at Chulalongkorn University and Oklahoma State University for their immeasurable dedication, support and strong encouragement to my learning.

I would like to thank Dr. Prapaporn Kiattikulwattana for special care, friendship, enthusiastic support and substantial encouragement throughout my doctoral study. My thanks and appreciation are extended to the following people: Rathachai Kaewlai (MD.), Chokeanan Wechaprukpitak, Dr. Manisra Baramichai, Dr. Nipapan Klunngien, Chainart Vongsittipornrung, Dr. Wisit Kumphai, Kultarika Kwosurat, Rungnapa and Dr. Somsak Kittipiyakul, Pornphan Dulyakarn, Dr. Chanvit Chantrasrisalai, Chayanuch Jakmanon, Arthit Phadungsilp, Xiaochen Hu, Dr. Jakkrit Kunthong, Arttaporn Komolhiran and Katherine McCollom for the enduring friendship, care, support and encouragement; Reza Jafari, Milind Malshe and Rutu Narulkar for being great colleagues and friends as well as providing strong supports to the research.

My gratitude also goes to Professor Dr. Dusit Kruangam, Dr. Porponth Sichanugrist, Dr. Pavan Siamchai and Dr. Sakon Kittivatcharapong for giving me opportunities to acquire precious experiences and supports to pursue the doctoral degree. Finally, I thank friends at Chulalongkorn University, colleagues at Thaicom Satellite and Thai students at Oklahoma State University for the supportive friendship.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Overview

Multilayer feedforward neural networks (MFNN) have been used in many nonlinear regression problems. They have been shown (both mathematically and practically) to be a powerful tool in approximating functions, based on a set of examples. In addition, it has been theoretically proven that their derivatives are capable of approximating the underlying derivatives of the functions. In this research, we focus on the development and the derivation of training algorithms for MFNNs to approximate both functions and their first-order derivatives.

Multilayer feedforward neural networks are capable of simultaneously approximating both a function and its derivatives, as has been proven in [HoSt90], [Horn91], [Ito93], [Li96] and [Pink99]. However, there has not been a large amount of research into the development of algorithms for training multilayer feedforward neural networks to simultaneously approximate both a function and its first derivatives. This will be the focus of our research.

There have been a few methods proposed in the past to train MFNNs for approximating both a function and its derivatives. All of these methods assume noise-free environment. One approach, called algebraic training, was proposed in [FeSt05]. This method

algebraically obtains the network parameters in one function approximation step. The derivative approximation is carried out in the second stage, where the algorithm iteratively adjusts the network parameters to improve the derivative approximation of the neural network. In [BaEn99], the derivative approximation was performed by adding an extra output unit for each partial derivative to the regular structure of the feedforward neural network that was used to approximate the function. The standard backpropagation procedure was then used to train the proposed network structure. In [HaZa99], an additional network was created to approximate the derivatives. The derivative approximation obtained from the extra network was then combined with the network output used for function approximation using the Taylor series expansion.

In our research, a different technique will be used. We will *not* modify the network structure or create an additional network for derivative approximation. We will use the same network for derivative approximation that we use for function approximation. In addition, unlike the algebraic training, the derivative approximation in our method will occur simultaneously with the function approximation. However, like the other methods above, we also assume that the data for this research are noise-free.

Objectives

As previously mentioned, our research goal is to develop new algorithms for training MFNNs to simultaneously approximate both a function and its first derivatives (in a noise-free environment). However, we will also investigate both function and derivative approximation accuracy for three existing algorithms. These existing algorithms will be called the standard methods in this research, and they are:

1. Broyden-Fletcher-Goldfarb-Shanno with Early Stopping ($BFGS-ES$),

2. Levenberg-Marquardt with Early Stopping ($LM-ES$), and

3. Gauss-Newton Approximation to Bayesian Regularization ($GNBR$).

This evaluation is similar to the work of [GaWh92], though we will use different training methods. The approximation accuracy and the execution time of the new algorithms will be compared with the standard methods.

In this research, five new algorithms are developed. They are

1. Levenberg-Marquardt with Early Stopping using a modified validation measure ($LM-ES1$),

2. Levenberg-Marquardt with Early Stopping using the derivative information of the training set ($LM-ES2$),

3. Combined Function and Derivative Approximation using Broyden-Fletcher-Goldfarb-Shanno optimization ($CFDA-BFGS$), and

4. Combined Function and Derivative Approximation with Levenberg-Marquardt optimization ($CFDA-LM$).

5. Combined Function and Derivative Approximation methods with a network-pruning algorithm.

The $LM-ES1$ and $LM-ES2$ methods only change the validation performance measure, while standard training algorithms are used to train the neural networks. For $CFDA-BFGS$ and $CFDA-LM$, the proposed change is in the training performance index. Therefore, the standard calculations cannot be applied. The $CFDA$ methods can be

incorporated with the proposed network-pruning algorithm, thus resulting in the fifth algorithm.

The results of the standard methods and the new algorithms will be compared on several analytic and real-world problems. The real-world application we focus on in this research is molecular dynamics, where the motion of atoms is simulated. The potential energy of the atoms is a function of atomic configuration, and the forces acting on the atoms are the negative first derivatives of the potential energy. Therefore, molecular dynamics is a good example where both the function and its first-order derivative approximation are of interest.

Outline

In Chapter 2, the mathematical notation and the general concepts of MFNNs will be introduced. The use of the neural networks in function approximation will be briefly reviewed. Three standard training algorithms for function approximation will be discussed, i.e. $BFGS-ES$, $LM-ES$ and $GNBR$. The conditions under which the neural networks can simultaneously and uniformly approximate both a function and its derivatives will be provided.

In Chapter 3, two new validation-related methods will be introduced, i.e. $LM-ES1$ and $LM-ES2$. The comparison of the approximation accuracy between these methods and $LM-ES$ on analytic problems will be compared and discussed.

In Chapter 4, the $CFDA$ training performance index is proposed. Two approaches (i.e. batch mode and memory-save method) for calculating the gradient of the $CFDA$ per-

formance index (which works with any gradient-based optimization) are proposed and derived. The execution time for computing the $CFDA$ gradient and the standard gradient, under several scenarios, is compared.

In Chapter 5, the $CFDA$ method under the Levenberg-Marquardt framework, named $CFDA-LM$, is discussed. Two approaches (i.e. batch mode and memory-save method) for minimizing the $CFDA$ performance index are proposed and derived. The comparison of the execution time for the $CFDA-LM$ method and the standard Levenberg-Marquardt algorithm is illustrated.

In Chapter 6, two new types of overfitting in a two-layer network with one output, trained with any $CFDA$ method are discussed. A network-pruning algorithm to mitigate the overfitting is proposed. A pseudo code for the algorithm is given.

In Chapter 7, we introduce a procedure to test the approximation accuracy of neural networks in four analytic problems. We discuss a way to assign a value to the unknown parameter in the $CFDA$ performance index. We choose $CFDA-BFGS$, for the gradient-based $CFDA$. The pruning algorithm is applied to $CFDA-BFGS$ and $CFDA-LM$. We denote $CFDA-BFGS$ with the pruning method and $CFDA-LM$ with the pruning method: $CFDA-BFGS_p$ and $CFDA-LM_p$, respectively. We test the approximation accuracy of neural networks trained by the standard methods and the $CFDA$ methods (i.e. $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$). The results are shown and discussed. Examples showing the $CFDA$ overfitting and how the pruning algorithm removes it are demonstrated.

In Chapter 8, the background material for molecular dynamics is reviewed. The use of neural networks in molecular dynamics is discussed. The comparison in approximation accuracy of neural networks trained by $GNBR$ and the $CFDA$ methods is illustrated and discussed for several problems. An example showing the $CFDA$ overfitting in a molecular dynamics problem and how the pruning algorithm removes it is demonstrated.

In Chapter 9, the summary of the research and the future work is provided.

CHAPTER 2

APPROXIMATION USING NEURAL NETWORKS

Introduction

The objective of this research is to use multilayer feedforward neural networks (MFNNs) for approximating functions and their first-order derivatives. This chapter serves three purposes. First, the notation used throughout the research will be introduced. Second, we will discuss the concept and the background material of MFNNs, including a review of three existing training algorithms for multilayer feedforward neural networks. These algorithms are Broyden-Fletcher-Goldfarb-Shanno with Early Stopping $(BFGS-ES)$, Levenberg-Marquardt with Early Stopping $(LM-ES)$, and Gauss-Newton approximation to the Bayesian regularization $(GNBR)$. Finally, the conditions under which multilayer feedforward neural networks can simultaneously approximate both a function and its first-order derivatives will be briefly discussed and reviewed.

Notation

This section will introduce mathematical notation that will be used throughout the research. The following definitions provide the meaning of three mathematical operators, see [HoJo94] and [MaNe99].

Definition (3) Let $\mathbf{A}$ *be an* $m \times n$ *matrix and* $\mathbf{B}$ *a* $p \times q$ *matrix. The* $mp \times nq$ *matrix defined by*

$$\begin{bmatrix} a_{1,1}\mathbf{B} & \dots & a_{1,n}\mathbf{B} \\ \dots & & \dots \\ a_{m,1}\mathbf{B} & \dots & a_{m,n}\mathbf{B} \end{bmatrix} \tag{1}$$

is called the Kronecker product of $\mathbf{A}$ *and* $\mathbf{B}$*, and written* $\mathbf{A} \otimes \mathbf{B}$.

Definition (4) If $\mathbf{A} = [a_{i,j}]$ *and* $\mathbf{B} = [b_{i,j}]$ *are matrices of the same order, say* $m \times n$*, then the Hadamard product of* $\mathbf{A}$ *and* $\mathbf{B}$ *is the* $m \times n$ *matrix*

$$\mathbf{A} \bullet \mathbf{B} = [a_{i,j} b_{i,j}]. \tag{2}$$

Definition (5) Let $\mathbf{A}$ *be an* $m \times n$ *matrix and* $\mathbf{a}_j$ *its* j^{th} *column; then* $vec\mathbf{A}$ *is the* $mn \times 1$ *vector*

$$vec\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \dots \\ \mathbf{a}_n \end{bmatrix}. \tag{3}$$

We will use the following notation, from [MaNe99], for representing the derivatives of a function:

Definition (6) Let f *be a differentiable scalar function of a scalar variable* x. *Then the derivative of* f *is denoted* $\dot{f}(x) = \frac{\partial f(x)}{\partial x}$.

Definition (7) Let f *be a differentiable scalar function of a* $p \times 1$ *vector* $\mathbf{x}$. *Then the derivative of* f *is the* $1 \times p$ *array*

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}^T} = \frac{\partial f}{\partial \mathbf{x}^T} = \left[\frac{\partial f(\mathbf{x})}{\partial x_1} \cdots \frac{\partial f(\mathbf{x})}{\partial x_n} \right]. \tag{4}$$

Definition (8) Let $\mathbf{f}$ *be a differentiable* $m \times 1$ *real vector function of a* $p \times 1$ *vector* $\mathbf{x}$, *then the derivative of* $\mathbf{f}$ *is the* $m \times p$ *matrix*

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}^T} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^T} = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}^T} \\ \cdots \\ \frac{\partial f_m(\mathbf{x})}{\partial \mathbf{x}^T} \end{bmatrix} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \cdots \frac{\partial \mathbf{f}}{\partial x_p} \right]. \tag{5}$$

And the derivative of the function f_i, *where* $i = 1, 2, \ldots, m$, *with respect to the scalar variable* x_j, *where* $j = 1, 2, \ldots, p$ *is denoted by*

$$\frac{\partial f_i(\mathbf{x})}{\partial x_j} = \frac{\partial f_i}{\partial x_j}. \tag{6}$$

Definition (9) Let $\mathbf{F}$ *be a differentiable* $m \times n$ *matrix function of a* $p \times q$ *matrix of real variables* $\mathbf{X}$. *Then, the derivative of* $\mathbf{F}$ *with respect to* $\mathbf{X}$ *is the* $mn \times pq$ *matrix*

$$\frac{\partial vec\mathbf{F}}{\partial (vec\mathbf{X})^T} = \begin{bmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1^T} & \cdots & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_q^T} \\ \cdots & & \cdots \\ \frac{\partial \mathbf{f}_n}{\partial \mathbf{x}_1^T} & \cdots & \frac{\partial \mathbf{f}_n}{\partial \mathbf{x}_q^T} \end{bmatrix}. \tag{7}$$

We will also use the notation $\mathbf{1}_{m \times n}$ to denote the $m \times n$ matrix, all of whose elements are ones. The $m \times m$ identity matrix will be denoted by the notation $\mathbf{I}_m$.

The next section will discuss some notation and background material for MFNNs. A brief discussion of how the MFNN can be used for function approximation will also be presented.

Multilayer Feedforward Neural Networks

We will divide this section into three parts. The first part will briefly discuss the general concept of MFNNs. The notation for MFNNs will be introduced in the second part. Finally, we will describe how MFNNs can be used for function approximation.

Background material

The term neural networks has been used to refer to a structure consisting of connected nodes (or neurons) in any formation (i.e. parallel, series or both). The inspiration of inventing neural networks was initially based on how a brain works. However, neural networks are now considered a mathematical and statistical model for information and sig-

nal processing. Neural networks have been theoretically and practically proven to be a powerful tool in many applications, such as function approximation, classification, pattern recognition, novelty detection, filtering, etc. Neural networks have been categorized into several types usually based on how neurons are connected. One of the most widely-used types for function approximation is the MFNN. In this research, we will focus on using MFNNs for function approximation. For ease of reference, the term neural network (or just network) will be used throughout this document to refer to MFNNs.

A neural network consists of neurons connected in parallel and series. The structure of a neuron is shown in the following figure.

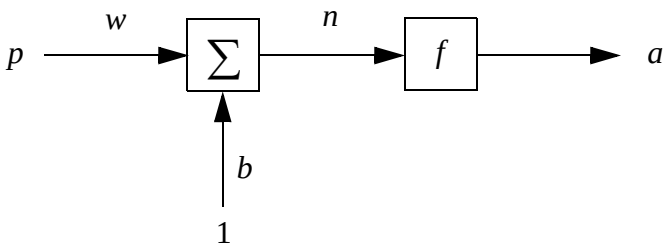

Figure 1) A single neuron

From Figure 1), a neuron consists of several components. They are the neuron input p, the weight w, the bias b, the summer, the net input n, the transfer function f, and the neuron output a. The net input is computed as $n = wp + b$. The net input is fed to the transfer function f to produce the neuron output a, i.e. $a = f(n)$. The weight w and the bias b are called the "network parameters".

When neurons are connected in parallel and cascade, a more complicated structure is obtained. In the parallel structure (layer), each neuron receives the same inputs as the other neurons do but independently produces its own output. When neurons are in the cascade

structure, each neuron output of a preceding layer is distributed to be a neuron input for every neuron in the layer following it. The general structure is referred to as an $M-layer$ neural network. An example of a $3-layer$ neural network is illustrated in the following figure.

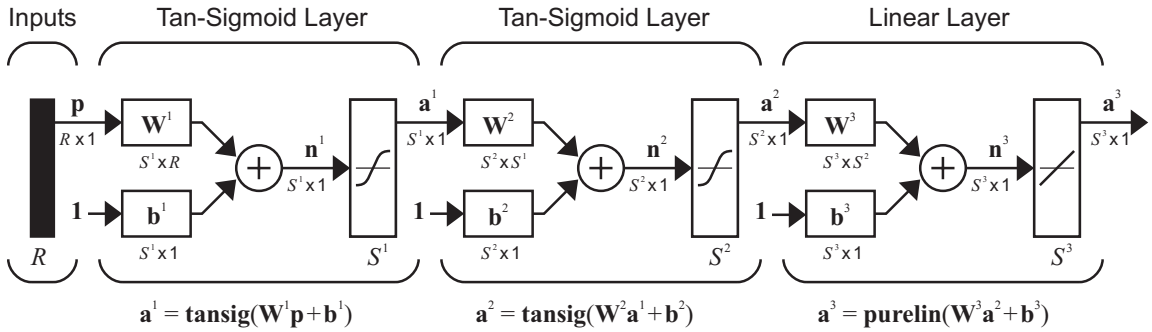

Figure 2) The $3-layer$ feedforward neural network

Since the complexity drastically increases as we have more neurons and layers, we need notation to refer to a specific neuron. In the next part, we will introduce the notation for the general structure of a multilayer feedforward neural networks that will be used throughout this research.

Notations for neural networks

An $M-layer$ feedforward neural network structure can be denoted $R-S^1-S^2-\ldots-S^M$. This structure notation corresponds to the statement that the $M-layer$ neural network consists of R inputs, S^1 neurons in the first layer, S^2 neurons in the second layer and so on, until S^M neurons in layer M, i.e. the output layer. The layers 1 through $M-1$ are called hidden layers. Layer M is called the output layer. We can consider the network inputs as the neuron outputs of layer $m = 0$.

Suppose we have an $R - S^1 - S^2 - \ldots - S^M$ neural network and Q input/target pairs in the training set. We write p_r to denote element r of the input vector, and write $p_{r,q}$ to denote element r in the q^{th} input vector. We write $w^m_{i,j}$ to denote the weight of neuron i at layer m, that connects from the output of neuron j at layer $m-1$. The bias for neuron i at layer m is denoted by b^m_i. The weight $w^1_{i,j}$ is the weight, which connects from the j^{th} element of the input vector to neuron i in the first layer. The net input of neuron i at layer m is denoted by n^m_i and the notation a^m_i denotes the output of neuron i at layer m. At the output layer, a^M_k is also denoted by a_k ($a^M_k = a_k$). When the q^{th} input vector is applied to the network (i.e. $p_r = p_{r,q}$ for $r = 1, 2, \ldots, R$), the values of n^m_i, a^m_i and a_k are denoted by $n^m_{i,q}$, $a^m_{i,q}$ and $a_{k,q}$, respectively. We assume that the transfer function is the same for each neuron in the same layer; thus using f^m to denote the transfer function at layer m.

The notation we just introduced can also be used in matrix form. A couple of examples are provided: the notation $p_{r,q}$ means this element is at row r of the $R \times 1$ input vector $\mathbf{p}_q$ and it is also the element at row r and column q of the $R \times Q$ matrix $\mathbf{P}$. The notation $a^m_{k,q}$ means this element is at row k of the $S^m \times 1$ vector $\mathbf{a}^m_q$, and it is the element at row k and column q of the $S^m \times Q$ matrix $\mathbf{A}^m$. The notation b^m_i is at row i of the $S^m \times 1$ vector $\mathbf{b}^m$. The notation $w^m_{i,j}$ is at row i of the $S^m \times 1$ vector $\mathbf{w}^m_j$, and it is at the row i and

column j of the $S^m \times S^{m-1}$ matrix $\mathbf{W}^m$. The following examples illustrate how the notations $p_{r,q}$, $a^m_{k,q}$, b^m_i and $w^m_{i,j}$ can be expressed in matrix form:

$$\mathbf{p}_q^T = \left[p_{1,q}\ p_{2,q}\ \cdots\ p_{R,q}\right] \text{ and } \mathbf{P} = \left[\mathbf{p}_1\ \mathbf{p}_2\ \cdots\ \mathbf{p}_Q\right], \tag{8}$$

$$(\mathbf{a}_q^m)^T = \left[a^m_{1,q}\ a^m_{2,q}\ \cdots\ a^m_{S^m,q}\right] \text{ and } \mathbf{A}^m = \left[\mathbf{a}_1^m\ \mathbf{a}_2^m\ \cdots\ \mathbf{a}_Q^m\right], \tag{9}$$

$$(\mathbf{b}^m)^T = \left[b^m_1\ b^m_2\ \cdots\ b^m_{S^m}\right], \tag{10}$$

$$(\mathbf{w}_j^m)^T = \left[w_{1,j}\ w_{2,j}\ \cdots\ w_{S^m,j}\right] \text{ and } \mathbf{W}^m = \left[\mathbf{w}_1^m\ \mathbf{w}_2^m\ \cdots\ \mathbf{w}_{S^{m-1}}^m\right]. \tag{11}$$

Note that we use the notation ${}_r\mathbf{p}^T$ to denote the r^{th} row vector of the matrix $\mathbf{P}$. Similarly, the notation $({}_i\mathbf{w}^m)^T$ is to denote the i^{th} row vector of the matrix $\mathbf{W}^m$. The following equations illustrate how these are related to the notations previously introduced:

$${}_r\mathbf{p}^T = \left[p_{r,1}\ p_{r,2}\ \cdots\ p_{r,Q}\right] \text{ and } \mathbf{P} = \begin{bmatrix} {}_1\mathbf{p}^T \\ {}_2\mathbf{p}^T \\ \cdots \\ {}_R\mathbf{p}^T \end{bmatrix}, \tag{12}$$

$$({}_i\mathbf{w}^m)^T = \left[w_{i,1}\ w_{i,2}\ \cdots\ w_{i,S^{m-1}}\right] \text{ and } \mathbf{W}^m = \begin{bmatrix} ({}_1\mathbf{w}^m)^T \\ ({}_2\mathbf{w}^m)^T \\ \cdots \\ \left({}_{S^m}\mathbf{w}^m\right)^T \end{bmatrix}. \tag{13}$$

Given the input signal $\mathbf{p}_q$, the output of neuron i at layer m, i.e. $a_{i,q}^m$, can be computed as

$$a_{i,q}^m = f^m(n_{i,q}^m) \text{ and } n_{i,q}^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_{j,q}^{m-1} + b_i^m , \tag{14}$$

for $i = 1, 2, \ldots, S^m$. At the first layer $(m = 1)$, the i^{th} neuron output can be calculated as

$$a_{i,q}^1 = f^1(n_{i,q}^1) \text{ and } n_{i,q}^1 = \sum_{r=1}^{R} w_{i,r}^1 p_{r,q} + b_r^1 , \tag{15}$$

for $i = 1, 2, \ldots, S^1$. Eq. (14) and Eq. (15) can be written in matrix form as

$$\mathbf{a}_q^m = \mathbf{f}^m(\mathbf{n}_q^m) = \begin{bmatrix} f^m(n_{1,q}^m) \\ f^m(n_{2,q}^m) \\ \cdots \\ f^m\left(n_{S^m,q}^m\right) \end{bmatrix} \text{ and } \mathbf{n}_q^m = \mathbf{W}^m \mathbf{a}_q^{m-1} + \mathbf{b}^m , \tag{16}$$

$$\mathbf{a}_q^1 = \mathbf{f}^1(\mathbf{n}_q^1) = \begin{bmatrix} f^1(n_{1,q}^1) \\ f^1(n_{2,q}^1) \\ \cdots \\ f^1\left(n_{S^1,q}^1\right) \end{bmatrix} \text{ and } \mathbf{n}_q^1 = \mathbf{W}^1 \mathbf{p}_q + \mathbf{b}^1 . \tag{17}$$

In the batch mode when all of the inputs (i.e. $q = 1, 2, \ldots, Q$) are presented to the network at the same time, Eq. (14) can be expressed as:

$$\mathbf{A}^m = \mathbf{F}^m(\mathbf{N}^m) = \left[\mathbf{f}^m(\mathbf{n}_1^m)\ \mathbf{f}^m(\mathbf{n}_2^m)\ \ldots\ \mathbf{f}^m(\mathbf{n}_Q^m)\right], \text{ where} \tag{18}$$

$$\mathbf{N}^m = \mathbf{W}^m \mathbf{A}^{m-1} + 1_{1 \times Q} \otimes \mathbf{b}^m . \tag{19}$$

Eq. (15) can be expressed in the batch mode as:

$$\mathbf{A}^1 = \mathbf{F}^1(\mathbf{N}^1) \text{ and } \mathbf{N}^1 = \mathbf{W}^1 \mathbf{P} + 1_{1 \times Q} \otimes \mathbf{b}^1 . \tag{20}$$

Since the objective of the research focuses on the first-order derivative approximation, we will need to provide the notation for the first-order derivatives of neural networks. By Definition (6), we write the derivative of a_k with respect to p_r, evaluated at $p_r = p_{r,q}$, as

$$\frac{\partial a_{k,q}}{\partial p_{r,q}} \equiv \left. \frac{\partial a_k}{\partial p_r} \right|_{p_r = p_{r,q}} . \tag{21}$$

By Definition (7), the derivative of a_k with respect to the input $\mathbf{p}$, evaluated at $\mathbf{p} = \mathbf{p}_q$, as

$$\frac{\partial a_{k,q}}{\partial \mathbf{p}_q^T} \equiv \left. \frac{\partial a_k}{\partial \mathbf{p}^T} \right|_{\mathbf{p} = \mathbf{p}_q} . \tag{22}$$

The derivative of $\mathbf{a}$ with respect to p_r, evaluated at $p_r = p_{r,q}$, is denoted by

$$\frac{\partial \mathbf{a}_q}{\partial p_{r,q}} \equiv \left. \frac{\partial \mathbf{a}}{\partial p_r} \right|_{p_r = p_{r,q}} . \tag{23}$$

By Definition (8), the derivative of $\mathbf{a}$ with respect to the input $\mathbf{p}$, evaluated at $\mathbf{p} = \mathbf{p}_q$, is written

$$\frac{\partial \mathbf{a}_q}{\partial \mathbf{p}_q^T} \equiv \left. \frac{\partial \mathbf{a}}{\partial \mathbf{p}^T} \right|_{\mathbf{p} = \mathbf{p}_q} . \tag{24}$$

A couple of more examples: the derivative of a_j^m with respect to the net input n_j^m, evaluated at $n_j^m = n_{j,q}^m$, is expressed as

$$\frac{\partial a_{j,q}^m}{\partial n_{j,q}^m} \equiv \left. \frac{\partial a_j^m}{\partial n_j^m} \right|_{n_j^m = n_{j,q}^m} . \tag{25}$$

The derivative of $\mathbf{a}^m$ with respect to the net input $\mathbf{n}^m$, evaluated at $\mathbf{n}^m = \mathbf{n}_q^m$, is denoted by

$$\frac{\partial \mathbf{a}_q^m}{\partial (\mathbf{n}_q^m)^T} \equiv \left. \frac{\partial \mathbf{a}^m}{\partial (\mathbf{n}^m)^T} \right|_{\mathbf{n}^m = \mathbf{n}_q^m} . \tag{26}$$

For batch mode, first suppose the input vectors $\mathbf{p}_q$ for all $q = 1, 2, \ldots, Q$ are distinct. By Definition (9), we write the derivative of $\mathbf{a}$ with respect to the input p_r, evaluated at $p_r = p_{r,q}$ for all $q = 1, 2, \ldots, Q$, as

$$\frac{\partial vec\mathbf{A}}{\partial_r \mathbf{p}^T} = \begin{bmatrix} \frac{\partial \mathbf{a}_1}{\partial p_{r,1}} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{a}_2}{\partial p_{r,2}} & \ldots & \mathbf{0} \\ \ldots & \ldots & & \ldots \\ \mathbf{0} & \mathbf{0} & \ldots & \frac{\partial \mathbf{a}_Q}{\partial p_{r,Q}} \end{bmatrix} . \tag{27}$$

Note that the blocks off the diagonal are zero since p_{r,q_1} is not related to p_{r,q_2}. The derivative of $\mathbf{a}$ with respect to the input $\mathbf{p}$, evaluated at $\mathbf{p} = \mathbf{p}_q$ for all $q = 1, 2, \ldots, Q$, is denoted by

$$\frac{\partial vec\mathbf{A}}{\partial (vec\mathbf{P})^T} = \begin{bmatrix} \frac{\partial \mathbf{a}_1}{\partial \mathbf{p}_1^T} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{a}_2}{\partial \mathbf{p}_2^T} & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial \mathbf{a}_Q}{\partial \mathbf{p}_Q^T} \end{bmatrix}, \tag{28}$$

where, again, the blocks off the diagonal are zeros because each input vector $\mathbf{p}_q$ is distinct.

The derivative of $\mathbf{a}^m$ with respect to the input $\mathbf{p}$, evaluated at $\mathbf{p} = \mathbf{p}_q$ for all $q = 1, 2, \ldots, Q$, is denoted by

$$\frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{P})^T} = \begin{bmatrix} \frac{\partial \mathbf{a}_1^m}{\partial \mathbf{p}_1^T} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{a}_2^m}{\partial \mathbf{p}_2^T} & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial \mathbf{a}_Q^m}{\partial \mathbf{p}_Q^T} \end{bmatrix}. \tag{29}$$

One more example: the derivative of $\mathbf{a}^m$ with respect to the net input $\mathbf{n}^m$, evaluated at $\mathbf{n}^m = \mathbf{n}_q^m$ for all $q = 1, 2, \ldots, Q$, is denoted by

$$\frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{N}^m)^T} = \begin{bmatrix} \frac{\partial \mathbf{a}_1^m}{\partial (\mathbf{n}_1^m)^T} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{a}_2^m}{\partial (\mathbf{n}_2^m)^T} & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial \mathbf{a}_Q^m}{\partial (\mathbf{n}_Q^m)^T} \end{bmatrix}. \quad (30)$$

In the next section, we will briefly review how neural networks have been used in function approximation problems.

Neural networks and function approximation

A MFNN can be used as a function approximator. This means that the network outputs will be estimates of the response of an unknown function $\mathbf{g}$. Given the vector function $\mathbf{g}$, we write g_k to denote the k^{th} element. We will use the notation $g_{k,q}$ to denote the k^{th} element of the function response to the q^{th} input vector. Note that we also call $g_{k,q}$ the target value. As in previous matrix notation, $\mathbf{g}_q$ and $\mathbf{G}$ are the column vector and batch mode representations, respectively.

Now, suppose we want a neural network to approximate a function $\mathbf{g}$ mapping from a subset in $\Re^R$ to a subset in $\Re^{S^M}$. Also suppose that a set of examples were drawn from function $\mathbf{g}$, where an example represents a pair of function inputs and corresponding function responses; i.e. $(\mathbf{p}_q, \mathbf{g}_q)$. Given a sufficient number of neurons and a set of examples

(the training set), an $R - S^1 - S^2 - \ldots - S^M$ neural network can be used to approximate a function $\mathbf{g}$ over a subset in $\Re^R$. In fact, [HoSt89] theoretically showed that, with a sufficient number of neurons in the hidden layer, $2 - layer$ networks are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy; meaning that the networks are a class of universal approximators.

When a neural network is trained, its weights and biases are adjusted so as to minimize some performance index (or objective function), which usually involves the mean square error between the network outputs and the target values. An optimization method is used to find the network parameters that minimize the performance index. The combination of the performance index and the optimization method makes up the training algorithm. There have been many training algorithms developed for neural networks. However, we will only review three algorithms: Broyden-Fletcher-Goldfarb-Shanno with Early Stopping $(BFGS - ES)$, Levenberg-Marquardt with Early Stopping $(LM - ES)$, and Gauss-Newton approximation to the Bayesian Regularization $(GNBR)$.

Training Algorithms

In this section, we will review three existing neural network training algorithm. We will first discuss the $BFGS - ES$, followed by the $LM - ES$, and finally the $GNBR$ training algorithm.

Suppose we want to approximate a function $\mathbf{g}$, which maps from a subset in $\Re^R$ to a subset in $\Re^{S^M}$ using an $R - S^1 - S^2 - \ldots - S^M$ neural network. Assume that the number of data (or examples) in the training set is Q. We use the $n \times 1$ vector $\mathbf{x}$ to denote the column vector containing all of the network parameters. The total number of the network parameters (i.e. the number of elements in the vector $\mathbf{x}$) is

$$n = S^1(R+1) + S^2(S^1+1) + \ldots + S^M(S^{M-1}+1). \tag{31}$$

BFGS-ES training algorithm

The performance index for this algorithm is the sum square of the difference between the network outputs and the target values, i.e. the sum square function error. Its performance index is written as:

$$J_f = \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \{a_{k,q} - g_{k,q}\}^2. \tag{32}$$

Minimizing the performance index J_f (with respect to the network parameters) is equivalent to forcing the neural network to approximate the function over a subset in $\Re^R$. For this training algorithm, the $BFGS$ optimization method, see [GiMu81], will be used to minimize the performance index. The steps for the $BFGS$ training algorithm are described in the following section. Note that we use the notation $J_f(\mathbf{x}_k)$ and $\nabla J_f(\mathbf{x}_k)$ to denote the performance index and the gradient of the performance index with respect to $\mathbf{x}$, evaluated at $\mathbf{x}_k$, respectively. The notation $\|\mathbf{x}\|$ denotes the 2-norm of the vector $\mathbf{x}$.

Steps for BFGS training algorithm

1. Set $k = 0$. Initialize the network parameter vector $\mathbf{x}_0$ by the method in [NgWi90]. Present the training set to the network. Compute the gradient of the performance index J_f with respect to the network parameters $\nabla J_f(\mathbf{x}_0)$. Set the initial search direction $\mathbf{p}_0 = -\nabla J_f(\mathbf{x}_0)$. Initialize the approximated Hessian matrix $\mathbf{B}_0 = \mathbf{I}_n$.

2. Minimize the performance index along the search direction, i.e. determine α_k such that $\mathbf{x}_k + \alpha_k \mathbf{p}_k$ minimizes the performance index J_f.

3. Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$, and evaluate $J_f(\mathbf{x}_{k+1})$. Also compute the gradient $\nabla J_f(\mathbf{x}_{k+1})$. Terminate the process if $J_f(\mathbf{x}_{k+1})$ or $\|\nabla J_f(\mathbf{x}_{k+1})\|$ are less than their pre-defined thresholds.

4. Calculate the change in the gradient $\mathbf{y}_k = \nabla J_f(\mathbf{x}_{k+1}) - \nabla J_f(\mathbf{x}_k)$, and compute the new approximated Hessian matrix:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{[\nabla J_f(\mathbf{x}_k)][\nabla J_f(\mathbf{x}_k)]^T}{[\nabla J_f(\mathbf{x}_k)]^T \mathbf{p}_k} + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\alpha_k \mathbf{y}_k^T \mathbf{p}_k}. \tag{33}$$

5. Compute the new search direction $\mathbf{p}_{k+1}$, which is the solution of

$$\mathbf{B}_{k+1} \mathbf{p}_{k+1} = -\nabla J_f(\mathbf{x}_{k+1}). \tag{34}$$

6. Set $k = k + 1$ and iterate step 2) to 6).

Note that we will use the algorithm in [NgWi90] to initialize the network parameters for all training algorithms in this research. The backpropagation process is used to compute the gradient of the performance index with respect to the network parameters $\nabla J_f(\mathbf{x})$.

The details of the backpropagation process for training a neural network can be found in several books, such as [HaDe96]. Note also that there are several methods to perform the line search in step 2. We will use the backtracking algorithm [DeSc83].

From Eq. (32), we can see that the performance index J_f is only the sum square of the errors, and thus it is an unregularized performance index. When using an unregularized performance index, it is possible to overfit the training data and then fail to generalize. There are several available techniques that could be used to prevent overfitting. For example, [HeKr91] proposed an approach to perform weight elimination based on the magnitude of the parameters. [SiDo91] proposed a method which adds noise to the function inputs, and [Bish95] showed that this technique is equivalent to Tikhonov regularization [TiAr77]. Another well known technique called Early Stopping, abbreviated ES, can be also used to prevent overfitting, and it will be the technique we will use in the research.

Early stopping is a widely-used technique to prevent overfitting by monitoring the approximation error on a set of data that is not in the training set. This set of data is called the "validation set". The approximation error on both the training and validation set initially decreases, until at some point the error on the validation set starts to increase while the error on the training set still keeps going lower. When the error over the validation set increases for a certain number of iterations, early stopping terminates the training process and it returns the network parameters at the point just before the increase of the validation error occurs. An example below illustrates how early stopping technique works.

Suppose we want to approximate a function $g(p) = \sin(\pi p)$ for $-1 \le p \le 1$. The following picture shows the graph of the function. Suppose a set of data points drawn from

the graph of the function were provided. We divided the set of data points into two groups: training set and validation set. Data points in the training and validation set are also shown in the figure.

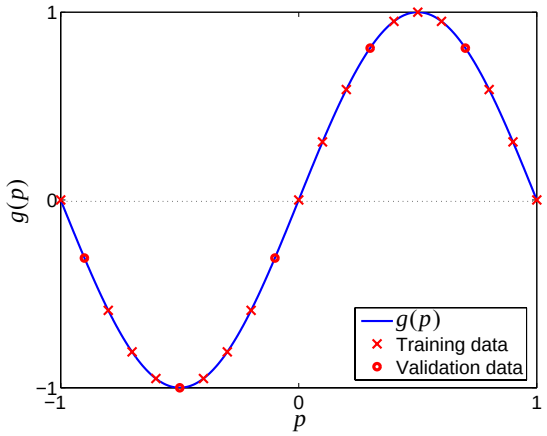

Figure 3) Training and validation set

Now, suppose a $1 - 5 - 1$ network was used to approximate the function through the $BFGS$ training algorithm. Figure 4) shows the sum square function error, i.e. J_f , on the training and validation set at each iteration.

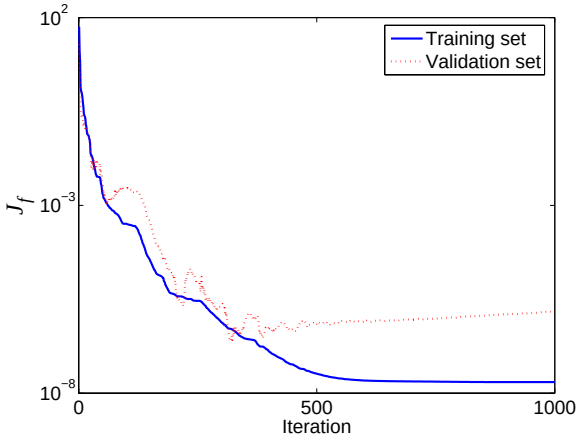

Figure 4) Sum square function error on training and validation set

From Figure 4), we can see that the validation error initially decreased as the training errors decreased. However, at some point around iteration 400, the validation error started increasing, while the training error continued to decrease. In order to prevent overfitting, it is desirable to use the network trained up to iteration 400. Therefore, if early stopping was used, it would terminate the training process at some iteration after 400 and use the network trained until iteration 400.

The validation error may fluctuate, so we do not stop the training at the first instance of an increase in the validation error. Instead, we monitor the error for a specified number of iterations to be sure that the error does not go back down. For the experiments described in this report, we monitored the error for 500 iterations after a minimum was reached before stopping the training.

We use $BFGS-ES$ to refer to the $BFGS$ training algorithm with early stopping. It is one of the three training algorithms we will use in the research. In the next section, we will review the second training method, which is the $LM-ES$ algorithm.

LM-ES training algorithm

The performance index for this algorithm is the same as in the $BFGS-ES$ training algorithm, i.e. Eq. (32). However, to minimize the performance index, the Levenberg-Marquardt optimization method [Leve44] [Marq63], abbreviated LM, will be used. The LM optimization algorithm is a combination of the Gauss-Newton algorithm and the method of gradient descent. It was designed to approximate Newton's method. To understand the concept of the LM optimization, let us begin with Newton's method. The update equation in Newton's method is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\nabla^2 J_f(\mathbf{x}_k)]^{-1} \nabla J_f(\mathbf{x}_k), \tag{35}$$

where $\nabla^2 J_f(\mathbf{x}_k)$ is the Hessian matrix evaluated at $\mathbf{x}_k$. This means we need to compute the gradient and the Hessian matrix of the performance index J_f.

To compute the gradient and the Hessian matrix, first suppose the performance index is in the form of

$$F(\mathbf{x}) = \mathbf{z}^T(\mathbf{x}) \times \mathbf{z}(\mathbf{x}), \tag{36}$$

then the j^{th} element of the gradient $\nabla F(\mathbf{x})$ would be

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = 2 \sum_{i=1}^{N} z_i(\mathbf{x}) \frac{\partial z_i(\mathbf{x})}{\partial x_j}, \tag{37}$$

where N is the number of elements in the vector $\mathbf{z}(\mathbf{x})$. The gradient can be written in matrix form:

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{z}(\mathbf{x}), \tag{38}$$

where $\mathbf{J}(\mathbf{x}) = \partial \mathbf{z}(\mathbf{x}) / \partial \mathbf{x}^T$ is the Jacobian matrix.

Next, we need to find the Hessian matrix. The k, j element of the Hessian matrix would be

$$[\nabla^2 F(\mathbf{x})]_{k,j} = \frac{\partial^2 F(\mathbf{x})}{\partial x_k \partial x_j} = 2 \sum_{i=1}^{N} \left\{ \frac{\partial z_i(\mathbf{x})}{\partial x_k} \frac{\partial z_i(\mathbf{x})}{\partial x_j} + z_i(\mathbf{x}) \frac{\partial^2 z_i(\mathbf{x})}{\partial x_k \partial x_j} \right\}. \tag{39}$$

The Hessian matrix can then be expressed in matrix form:

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x}), \tag{40}$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} z_i(\mathbf{x})\nabla^2 z_i(\mathbf{x}). \tag{41}$$

If we assume $\mathbf{S}(\mathbf{x})$ is small, the approximated Hessian matrix becomes

$$\nabla^2 F(\mathbf{x}) \cong 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}). \tag{42}$$

By substituting Eq. (38) and Eq. (42) into Eq. (35), we obtain the Gauss-Newton method:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{z}(\mathbf{x}_k), \tag{43}$$

where $\mathbf{z}(\mathbf{x}_k)$ and $\mathbf{J}(\mathbf{x}_k)$ are the vector $\mathbf{z}(\mathbf{x})$ and the Jacobian matrix $\mathbf{J}(\mathbf{x})$, evaluated at $\mathbf{x}_k$, respectively.

One problem with the Gauss-Newton method is that the approximate Hessian matrix $\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$ may not be invertible. This problem can be overcome by using the matrix $\tilde{\mathbf{H}} = \mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + \mu\mathbf{I}_n$ in place of the matrix $\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$, where $\mu > 0$. Increasing the parameter μ will make the matrix $\tilde{\mathbf{H}}$ become positive definite, and thus invertible. This leads to the Levenberg-Marquardt algorithm:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k\mathbf{I}_n]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{z}(\mathbf{x}_k). \tag{44}$$

The *LM* optimization algorithm requires us to calculate two terms, which are the vector $\mathbf{z}(\mathbf{x})$ and the Jacobian matrix $\mathbf{J}(\mathbf{x})$. By equating the performance index in Eq. (36) and Eq. (32), we have the vector $\mathbf{z}(\mathbf{x})$

$$\mathbf{z}(\mathbf{x}) = vec\mathbf{E}, \text{ where } \mathbf{E} = \mathbf{A} - \mathbf{G}, \tag{45}$$

with $N = S^M Q$. Obtaining $\mathbf{z}(\mathbf{x})$ is thus simply the feedforward propagation. Computing the Jacobian matrix $\mathbf{J}(\mathbf{x})$ in neural networks is, however, more complicated and it was originally discussed in detail in [HaMe94]. Calculating the Jacobian matrix involves the calculation of the *Marquardt sensitivity* [HaDe96], using the backpropagation process. The following is a summary of the LM training algorithm.

Steps for LM training algorithm

1. Set $k = 0$. Initialize the network parameter vector $\mathbf{x}_0$. Present the training set to the network. Initialize μ_0 to a small value, e.g. 0.001. Set μ_{max} to a large value, e.g. 10^{10}. Set $\vartheta > 1$, e.g. 10.

2. Compute $\mathbf{z}(\mathbf{x}_k)$ and the Jacobian matrix $\mathbf{J}(\mathbf{x}_k)$ using Eq. (45) and Eq. (38), respectively. Also compute $J_f(\mathbf{x}_k)$. Terminate the process if $J_f(\mathbf{x}_k)$ or $\|\mathbf{J}(\mathbf{x}_k)\mathbf{z}(\mathbf{x}_k)\|$ is less than its predefined threshold, or if $\mu_k \geq \mu_{max}$.

3. While $\mu_k < \mu_{max}$, do the following:

a). Compute

$$\mathbf{x}_{k+1}^{temp} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}_n]^{-1} \mathbf{J}^T(\mathbf{x}_k)\mathbf{z}(\mathbf{x}_k). \tag{46}$$

b). Compute $\mathbf{z}(\mathbf{x}_{k+1}^{temp})$ and the Jacobian matrix $\mathbf{J}(\mathbf{x}_{k+1}^{temp})$. Compute $J_f(\mathbf{x}_{k+1}^{temp})$.

c). If $J_f(\mathbf{x}_{k+1}^{temp}) \geq J_f(\mathbf{x}_k)$, set $\mu_k = \vartheta \mu_k$ and go back to Step 3. Otherwise, go to the next step.

d). If $J_f(\mathbf{x}_{k+1}^{temp}) < J_f(\mathbf{x}_k)$, set $\mu_{k+1} = \mu_k / \vartheta$ and set $\mathbf{x}_{k+1} = \mathbf{x}_{k+1}^{temp}$. Then, set $k = k + 1$ and go back to Step 2.

4. Terminate the process (in this case it is due to $\mu_k \geq \mu_{max}$).

As the performance index J_f is only the sum square of the function errors, early stopping will again be used to prevent overfitting. We will denote this algorithm, which minimizes the performance index J_f using the LM optimization with early stopping, as the $LM-ES$ method. In the next section, we will review the Gauss-Newton approximation to Bayesian Regularization ($GNBR$) training algorithm.

GNBR training algorithm

Unlike the performance index for the $BFGS-ES$ and the $LM-ES$ training algorithms, the performance index of the $GNBR$ training method is a regularized performance index, as shown below:

$$F = \beta \sum_{q=1}^{Q} \sum_{k=1}^{S^M} (g_{k,q} - a_{k,q})^2 + \alpha \sum_{i=1}^{n} x_i^2, \tag{47}$$

where x_i is a network weight or bias, n is the total number of weights and biases, β and α are scalar values weighting the importance of the two terms. The regularized term in the performance index is $\sum_{i=1}^{n} x_i^2$, which is the sum of squares of the network weights and biases.

Regularization is another technique used to prevent overfitting. Although its purpose is the same as early stopping, it works in a different way. The regularized performance index forces the magnitudes of the network parameters to be small, which causes the network output to be smooth. That is, the regularization prevents steep fluctuations in the network response.

A problem for using the regularized performance index is that the weighting factors β and α are unknown and problem-dependent. If α is too small relative to β, then we would still observe overfitting as there is too little impact from the regularized term. In contrast, if α is too large relative to β, then the network output would be too smooth and would not approximate the function. To overcome this problem, David J. C. MacKay proposed a method in [MacK92] using Bayes' rule to automatically choose the weighting factors to balance between the function approximation capability and the smoothness of the network output. [FoHa97] later combined MacKay's method with the Levenberg-Marquardt framework to obtain the Gauss-Newton Approximation to Bayesian Regularization, denoted as *GNBR*. The concept of this algorithm can be explained in two parts: first, minimizing the performance index in Eq. (47), and second, choosing the values of β and α.

From Eq. (36) and Eq. (47), we can rewrite the performance index as:

$$F = \beta \mathbf{z}^T(\mathbf{x})\mathbf{z}(\mathbf{x}) + \alpha \mathbf{x}^T\mathbf{x} = \beta E_D + \alpha E_W. \tag{48}$$

Now, from Eq. (38), the gradient of the performance index can be written

$$\nabla F = \beta \nabla E_D + \alpha \nabla E_W = 2\beta \mathbf{J}_D^T(\mathbf{x})\mathbf{z}(\mathbf{x}) + 2\alpha \mathbf{J}_E^T(\mathbf{x})\mathbf{x}. \tag{49}$$

Since E_D is in fact J_f in Eq. (32), the Jacobian matrix $\mathbf{J}_D(\mathbf{x}) = \mathbf{J}(\mathbf{x})$, which can be ob-

tained by the same backpropagation process as in the Levenberg-Marquardt training algorithm. Since the Jacobian matrix $\mathbf{J}_E(\mathbf{x}) = \partial\mathbf{x}/\partial\mathbf{x}^T$, this becomes $\mathbf{J}_E(\mathbf{x}) = \mathbf{I}_n$. Therefore, Eq. (49) turns out to be

$$\nabla F = 2\beta\mathbf{J}^T(\mathbf{x})\mathbf{z}(\mathbf{x}) + 2\alpha\mathbf{x}. \tag{50}$$

Using Eq. (42) and Eq. (49), the Hessian can be approximated and written in matrix form:

$$\nabla^2 F(\mathbf{x}) = \beta\nabla^2 E_D + \alpha\nabla^2 E_W \cong 2\beta\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\alpha\mathbf{I}_n. \tag{51}$$

By substituting Eq. (50) and Eq. (51) into Eq. (35), the update equation becomes

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\beta_k\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + (\alpha_k + \mu_k)\mathbf{I}_n]^{-1}[\beta_k\mathbf{J}^T(\mathbf{x}_k)\mathbf{z}(\mathbf{x}_k) + \alpha_k\mathbf{x}_k]. \tag{52}$$

Now, we have an update equation for the performance index in Eq. (47). Next, we will explain MacKay's method to choose the values of β and α.

Under the Bayesian framework of MacKay [MacK92], the network parameters are considered random variables. The posterior density P of the network parameters $\mathbf{x}$ can be written according to the Bayes' rule:

$$P(\mathbf{x}|D, \beta, \alpha, M) = \frac{P(D|\mathbf{x}, \beta, M)P(\mathbf{x}|\alpha, M)}{P(D|\beta, \alpha, M)}, \tag{53}$$

where D represents the data set presented to the network, and M is the network model. The term $P(D|\mathbf{x}, \beta, M)$ is the likelihood function, the term $P(\mathbf{x}|\alpha, M)$ is the prior density, and the term $P(D|\beta, \alpha, M)$ is named evidence. The parameter β is related to the variance of the likelihood and α is related to the variance of the prior density. If the noise in the model output and the prior density are assumed to follow Gaussian distributions, then the likeli-

hood function and the prior density become

$$P(D|\mathbf{x}, \beta, M) = \frac{1}{Z_D(\beta)} \exp(-\beta E_D), \text{ and} \tag{54}$$

$$P(\mathbf{x}|\alpha, M) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W), \tag{55}$$

respectively. The term $Z_D(\beta) = (\pi/\beta)^{N/2}$ and $Z_W(\alpha) = (\pi/\alpha)^{n/2}$, where $N = S^M Q$.

Considering the evidence as a normalization factor, and substituting Eq. (54) and Eq. (55) into Eq. (53), we obtain the posterior density

$$P(\mathbf{x}|D, \beta, \alpha, M) = \frac{1}{Z_F(\beta, \alpha)} \exp(-F(\mathbf{x})). \tag{56}$$

From Eq. (56), we can see that maximizing the posterior density is equivalent to minimizing the regularized objective function F.

Now, given the data, we are interested in what value β and α should be. This means we need to consider $P(\beta, \alpha|D, M)$. By using Bayes' rule, it becomes

$$P(\beta, \alpha|D, M) = \frac{P(D|\beta, \alpha, M)P(\beta, \alpha|M)}{P(D|M)}. \tag{57}$$

Suppose the prior density $P(\beta, \alpha|M)$ is uniform, which corresponds to the statement that we do not know what value β and α should be. Then, maximizing the posterior $P(\beta, \alpha|D, M)$ is equivalent to maximizing the likelihood function $P(D|\beta, \alpha, M)$. However, note that the likelihood function in Eq. (57) is the evidence, which is the normalization factor, in Eq. (53). Therefore, the evidence in Eq. (53) can be solved:

$$P(D|\beta, \alpha, M) = \frac{P(D|\mathbf{x}, \beta, M)P(\mathbf{x}|\alpha, M)}{P(\mathbf{x}|D, \beta, \alpha, M)}. \tag{58}$$

By substituting Eq. (54) to Eq. (56) into Eq. (58), we obtain

$$P(D|\beta, \alpha, M) = \frac{Z_F(\beta, \alpha)}{Z_D(\beta) Z_W(\alpha)}. \quad (59)$$

From Eq. (59), the only term we do not know is $Z_F(\beta, \alpha)$. However, we can estimate it from a Taylor series expansion, by assuming the objective function has a quadratic shape in a small region around the minimum point $\mathbf{x}^{MP}$. At the minimum point, the gradient of the function is zero. Thus, the objective function approximated around $\mathbf{x}^{MP}$ is written as

$$F(\mathbf{x}) \cong F(\mathbf{x}^{MP}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{MP})^T \mathbf{H}^{MP} (\mathbf{x} - \mathbf{x}^{MP}), \quad (60)$$

where $\mathbf{H} = \beta \nabla^2 E_D + \alpha \nabla^2 E_W$ is the Hessian matrix of $F(\mathbf{x})$, and $\mathbf{H}^{MP}$ is the Hessian matrix evaluated at $\mathbf{x}^{MP}$. Therefore, the posterior density in Eq. (56) can be written as

$$P(\mathbf{x}|D, \beta, \alpha, M) \cong \frac{1}{Z_F(\beta, \alpha)} \exp\left(-\left\{F(\mathbf{x}^{MP}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{MP})^T \mathbf{H}^{MP} (\mathbf{x} - \mathbf{x}^{MP})\right\}\right), \quad (61)$$

which is rewritten as

$$P(\mathbf{x}|D, \beta, \alpha, M) \cong \left\{\frac{1}{Z_F(\beta, \alpha)} \exp(-F(\mathbf{x}^{MP}))\right\} \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}^{MP})^T \mathbf{H}^{MP} (\mathbf{x} - \mathbf{x}^{MP})\right). \quad (62)$$

The multivariate Gaussian density with mean $\mathbf{x}^{MP}$ and the covariance matrix $(\mathbf{H}^{MP})^{-1}$ is expressed as:

$$P(\mathbf{x}) = \frac{1}{(2\pi)^{n/2} \left|(\mathbf{H}^{MP})^{-1}\right|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}^{MP})^T \mathbf{H}^{MP} (\mathbf{x} - \mathbf{x}^{MP})\right). \quad (63)$$

By equating Eq. (62) and Eq. (63), we can solve for

$$Z_F(\beta, \alpha) \cong [(2\pi)^n \left|(\mathbf{H}^{MP})^{-1}\right|]^{1/2} \exp(-F(\mathbf{x}^{MP})). \quad (64)$$

Substitute Eq. (64) into Eq. (59) along with the values of $Z_D(\beta)$ and $Z_W(\alpha)$ from Eq. (54) and Eq. (55), take the derivative with respect to each of the log in Eq. (59) and set them to zero. This produces

$$\beta^{MP} = \frac{N-\gamma}{2E_D(\mathbf{x}^{MP})} \text{ and } \alpha^{MP} = \frac{\gamma}{2E_W(\mathbf{x}^{MP})}, \quad (65)$$

where $\gamma = N - 2\alpha^{MP} tr(\mathbf{H}^{MP})^{-1}$ is called the effective number of parameters. The parameter γ is a measure of how many parameters in the network are effectively used in reducing the error function, and it can range from zero to n.

Since the estimation for β and α requires the calculation of the Hessian matrix of the performance index $F(\mathbf{x})$ at the minimum point $\mathbf{x}^{MP}$, [FoHa97] proposed using the approximated Hessian readily available under the Levenberg-Marquardt framework, i.e. Eq. (51), leading to the *GNBR* training algorithm. The algorithm is summarized below.

Steps for GNBR training algorithm

1. Initialization: Same as the Levenberg-Marquardt training algorithm. Initialize $\alpha_0 = 0$, $\beta_0 = 1$.

2. Take one step of the *LM* training algorithm to minimize the objective function in Eq. (47), by using Eq. (52).

3. Compute the effective number of parameters $\gamma_k = n - 2\alpha_k tr(\mathbf{H}_k)$, where the Hessian matrix $\mathbf{H}$ can be approximated by Eq. (51), and $\mathbf{H}_k$ is the Hessian evaluated at $\mathbf{x}_k$.

4. Compute the new estimates for β and α:

$$\beta_{k+1} = \frac{N - \gamma_k}{2E_D(\mathbf{x}_k)} \text{ and } \alpha_{k+1} = \frac{\gamma_k}{2E_W(\mathbf{x}_k)}. \quad (66)$$

Then, set $k = k + 1$.

5. Iterate step 2) to 5) until $\mu_k \geq \mu_{max}$, or $F(\mathbf{x}_k)$ is less than its predefined threshold.

As the *GNBR* algorithm penalizes the magnitude of the network parameters as a technique to reduce the overfit problem, we will not have a validation data set in this algorithm.

Recall that the goal of this research is to approximate both a function and its first-order derivatives using neural networks. [HoSt89] showed that multilayer feedforward neural networks can approximate any Borel measurable function. In the next section, a theoretical discussion of function and derivative approximations with neural networks will be reviewed.

Conditions for Function and Derivative Approximation

Recall that our goal is to approximate both a function and its first-order derivatives. This section will briefly discuss the theoretical conditions under which neural networks can simultaneously approximate both a function and its derivatives. There are several discussions on the conditions, such as [HoSt90], [Horn91], [Ito93] or [Pink99]; however, we will follow [Li96].

We first introduce notation. We let Z_+^R denote the lattice of non-negative multi-integers in $\Re^R$. For $\mathbf{m} = (m_1, m_2, \ldots, m_R) \in Z_+^R$, we set $|\mathbf{m}| = m_1 + m_2 + \ldots + m_R$ and

$$\mathbf{D}^{\mathbf{m}} = \frac{\partial^{|\mathbf{m}|}}{\partial p_1^{m_1} \partial p_2^{m_2} \ldots \partial p_R^{m_R}}. \tag{67}$$

We also write $\mathbf{m}^1 \le \mathbf{m}^2$ if $m_r^1 \le m_r^2$ for all $r = 1, 2, \ldots, R$. Given an open set Ω of $\Re^R$ (probably $\Omega = \Re^R$), we write $C^{\mathbf{m}}(\Omega)$ to denote the set consisting of functions with all $\mathbf{k}^{th}$ order continuous partial derivatives in Ω, for $\mathbf{k} \in Z_+^R$ and $\mathbf{k} \le \mathbf{m}$. We write $f \in \hat{C}^{\mathbf{m}}(K)$ to imply, for a compact set K of $\Re^R$, there is an open set Ω such that $K \subset \Omega$ and $f \in C^{\mathbf{m}}(\Omega)$. We write M to denote a $2-layer$ neural network with the network output a, the transfer function f^1 in the hidden layer and the linear transfer function in the output layer, i.e. f^2 is linear.

Given a compact subset K of $\Re^R$, and a function $g \in \hat{C}^{\mathbf{m}}(K)$ for $\mathbf{m} \in Z_+^R$, [Li96] showed that if $f^1 \in C^{|\mathbf{m}|}(\Re)$ and f^1 is not a polynomial, then $\mathbf{D}^{\mathbf{k}} a$ of a network M can uniformly and simultaneously approximate $\mathbf{D}^{\mathbf{k}} g$, for $\mathbf{k} \in Z_+^R$ and $\mathbf{k} \le \mathbf{m}$.

For example, [Li96] considered a function on $\Re^2$, given by:

$g(p_1, p_2) = (p_1 p_2)^{11/3} \sin[1/(p_1 p_2)]$ if $p_1, p_2 \ne 0$; otherwise $g(p_1, p_2) = 0$. We can verify that $\partial^2 g / \partial p_1^2$ and $\partial^2 g / \partial p_2^2$ are discontinuous at the origin $(0, 0)$; however, g,

$\partial g/\partial p_1$, $\partial g/\partial p_2$, and $\partial^2 g/\partial p_1 \partial p_2$ are continuous on $\Re^2$. Therefore, $g \in C^{(1,1)}(\Re^2)$. In this situation, a network M can simultaneously and uniformly approximate g, $\partial g/\partial p_1$, $\partial g/\partial p_2$, and $\partial^2 g/\partial p_1 \partial p_2$ on any compact set K containing $(0, 0)$, provided that the transfer function f^1 in the hidden layer is non-polynomial and $f^1 \in C^2(\Re)$.

The typical transfer functions in the hidden layers and the output layer of neural networks used for function approximation are sigmoid and linear functions, respectively. Sigmoid functions are non-polynomial and they are of class C^∞ (infinitely differentiable or smooth functions), i.e. all derivative orders are continuous. Therefore, given that the function $\mathbf{g}$ and its first-order partial derivatives exist and are continuous, the standard two-layer neural network has the ability to simultaneously approximate the function $\mathbf{g}$ and its first-order derivatives.

Next, we will introduce notation for first-order derivative values. Similar to the notation defined in Eq. (21) to Eq. (24), the derivative of the scalar function g_k with respect to p_r and with respect to $\mathbf{p}$, evaluated at $p_r = p_{r,q}$ and $\mathbf{p} = \mathbf{p}_q$, is written, respectively,

$$\frac{\partial g_{k,q}}{\partial p_{r,q}} \equiv \left.\frac{\partial g_k}{\partial p_r}\right|_{p_r = p_{r,q}}, \quad \frac{\partial g_{k,q}}{\partial \mathbf{p}_q^T} \equiv \left.\frac{\partial g_k}{\partial \mathbf{p}^T}\right|_{\mathbf{p} = \mathbf{p}_q}. \tag{68}$$

The derivative of the vector function $\mathbf{g}$ with respect to p_r and $\mathbf{p}$, evaluated at $p_r = p_{r,q}$ and $\mathbf{p} = \mathbf{p}_q$, is denoted, respectively, by:

$$\frac{\partial \mathbf{g}_q}{\partial p_{r,\,q}} \equiv \left.\frac{\partial \mathbf{g}}{\partial p_r}\right|_{p_r = p_{r,\,q}}, \quad \frac{\partial \mathbf{g}_q}{\partial \mathbf{p}_q^T} \equiv \left.\frac{\partial \mathbf{g}}{\partial \mathbf{p}^T}\right|_{\mathbf{p} = \mathbf{p}_q}. \tag{69}$$

For batch mode, similar to Eq. (27) and Eq. (28), we will use the following notation:

$$\frac{\partial vec\mathbf{G}}{\partial ({}_r\mathbf{p})^T} = \begin{bmatrix} \frac{\partial \mathbf{g}_1}{\partial p_{r,\,1}} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{g}_2}{\partial p_{r,\,2}} & \dots & \mathbf{0} \\ \dots & \dots & & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \frac{\partial \mathbf{g}_Q}{\partial p_{r,\,Q}} \end{bmatrix}, \text{ and } \frac{\partial vec\mathbf{G}}{\partial (vec\mathbf{P})^T} = \begin{bmatrix} \frac{\partial \mathbf{g}_1}{\partial \mathbf{p}_1^T} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{g}_2}{\partial \mathbf{p}_2^T} & \dots & \mathbf{0} \\ \dots & \dots & & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \frac{\partial \mathbf{g}_Q}{\partial \mathbf{p}_Q^T} \end{bmatrix}. \tag{70}$$

Summary

In this chapter, we first stated the objective of the research: to develop procedures for approximating functions and their derivatives. Then, we introduced the operators and notation that will be used throughout the research. The notation and background material for the multilayer feedforward neural network were provided. A neural network learns to approximate a function through an optimization process. A combination of the objective function and the optimization process defines a distinct training algorithm. We discussed three existing training algorithms: $BFGS-ES$, $LM-ES$ and $GNBR$. Each is capable of forcing a neural network to approximate a function in a different way. The $BFGS-ES$ and $LM-ES$ methods use early stopping as a technique to prevent overfitting, while the $GNBR$ algorithm uses the Bayesian regularizer.

Finally, the conditions under which a neural network can uniformly and simultaneously approximate a function and its derivatives were discussed. One of the conditions requires that the function and its first-order derivatives be continuous. A second condition requires that the transfer function in the hidden layer of the neural network be sufficiently differentiable but not be a polynomial. For example, the typical sigmoid transfer function would be satisfactory.

CHAPTER 3

VALIDATION-RELATED METHODS

Introduction

Recall that our objective is to use a neural network to approximate a function and its first-order derivatives. In this chapter, we will discuss two simple methods for accomplishing this task. These methods are similar to the standard training algorithms discussed in Chapter 2, but some modifications are applied to the early stopping technique. As the Levenberg-Marquardt optimization is chosen to work with these two modified early stopping techniques, the two proposed methods are:

1. Modified validation performance measure ($LM-ES1$), and

2. Early stopping using the derivative information on the training set ($LM-ES2$).

As we describe each method, the idea behind it will be discussed. Then, the simulation results for each method, based on the benchmark tests that are described in Chapter 7, will be shown. The results in this chapter can be compared with the simulation results obtained using the standard early stopping technique, which are shown in Chapter 7.

Modified validation performance measure

For the standard early stopping method (discussed in Chapter 2), the validation performance is measured by the sum squared function error. We proposed using a combination

of the sum squared function error and the sum squared derivative error. In other words, the validation performance will be determined by

$$E_{V_d} = \sum_{q_V=1}^{Q_V} \sum_{k=1}^{S^M} (a_{k,q_V} - g_{k,q_V})^2 + \rho_d \sum_{q_V=1}^{Q_V} \sum_{k=1}^{S^M} \sum_{r=1}^{R} \left(\frac{\partial a_{k,q_V}}{\partial p_{r,q_V}} - \frac{\partial g_{k,q_V}}{\partial p_{r,q_V}} \right)^2, \qquad (71)$$

where Q_V is the number of examples in the validation set, and ρ_d is a scalar factor.

If the validation error E_{V_d} increases for a certain number of iterations, then the training will be terminated. The unknown parameter in the equation is ρ_d, which we will vary to investigate its consequences to the approximation of neural networks. Another purpose of this parameter is to account for the fact that the scale of the derivative values can be much different from the function values. The procedure for training a neural network using this method is exactly the same as the standard training algorithm with the standard early stopping technique (e.g. $BFGS-ES$ or $LM-ES$), with two exceptions. First, we need to compute the derivative of the network output with respect to the network inputs; i.e. $\partial \mathbf{a}_q / \partial \mathbf{p}_q^T$, for all $q = 1, 2, \ldots, Q_V$, and this calculation is shown in Chapter 4. Second, the performance measure in the standard early stopping technique is now replaced by the new measure in Eq. (71). In this research, we choose $LM-ES$ to work with the modified validation performance measure. We denote this method $LM-ES1$. Note that when $\rho_d = 0$, $LM-ES1$ is $LM-ES$.

The approximation accuracy obtained from $LM-ES1$ for the simple analytic functions (which are introduced in Chapter 7) are shown in the next section. Note again that the procedure to perform the simulation is described in Chapter 6.

Simulation results

The approximation accuracy obtained from $LM-ES1$ for the simple analytic functions are shown in the following tables. The results in Table 1, Table 2 and Table 3 can be compared with the results obtained from other algorithms in Table 9, Table 10 and Table 11 in Chapter 7, respectively. For ease of reference, we put the results obtained from $LM-ES$ with the standard early stopping (i.e. $\rho_d = 0$). Note that the definitions of $RMSE_F^{md}$ and $RMSE_D^{md}$ are defined in the simulation procedure in Chapter 7.

ρ_d in $LM-ES1$	Problem 1			
	$RMSE_F^{md}$		$RMSE_D^{md}$	
	Training	Test	Training	Test
0	2.48E-06	8.90E-04	6.42E-03	1.74E-02
1E-06	2.48E-06	8.90E-04	6.42E-03	1.77E-02
1E-04	2.50E-06	8.90E-04	6.42E-03	1.74E-02
1E-02	2.52E-06	8.90E-04	5.71E-03	1.38E-02
1E+00	3.00E-06	9.96E-04	6.51E-03	1.73E-02
1E+02	3.00E-06	9.96E-04	6.51E-03	1.73E-02

Table 1 Approximation accuracy on problem 1

ρ_d in $LM-ES1$	Problem 2			
	$RMSE_F^{md}$		$RMSE_D^{md}$	
	Training	Test	Training	Test
0	1.14E-05	1.16E-02	1.09E+00	1.94E+00
1E-06	1.14E-05	1.14E-02	1.09E+00	1.94E+00
1E-04	4.18E-07	7.44E-03	5.52E-01	1.45E+00
1E-02	9.04E-07	8.67E-03	9.69E-01	1.54E+00
1E+00	9.04E-07	8.67E-03	9.69E-01	1.54E+00
1E+02	9.04E-07	8.67E-03	9.69E-07	1.54E+00

Table 2 Approximation accuracy on problem 2

ρ_d in $LM-ES1$	Problem 3			
	$RMSE_F^{md}$		$RMSE_D^{md}$	
	Training	Test	Training	Test
0	6.72E-06	2.09E-04	5.67E-03	1.12E-02
1E-06	6.72E-06	2.09E-04	5.67E-03	1.12E-02
1E-04	9.93E-06	2.80E-04	1.41E-02	2.69E-02
1E-02	1.14E-05	2.96E-04	1.38E-02	2.69E-02
1E+00	1.14E-05	2.97E-04	1.38E-02	2.69E-02
1E+02	1.14E-05	2.97E-04	1.38E-02	2.69E-02

Table 3 Approximation accuracy on problem 3

From these results, we can see that the approximation accuracy obtained from $LM-ES1$ is similar to the results from $LM-ES$. We conclude that adding the derivative error term into the standard validation performance measure does not improve the approximation accuracy.

In the next section, the simulation results obtained from the LM training algorithm with early stopping using the derivative information of the training set, i.e. $LM-ES2$, will be discussed.

Early stopping using the derivative information of the training set

In regular early stopping, the training process will be terminated when the validation performance increases. Recall that the standard validation performance measure is

$$E_V = \sum_{q_V = 1}^{Q_V} \sum_{k = 1}^{S^M} (a_{k, q_V} - g_{k, q_V})^2 . \tag{72}$$

The function error term $e_{k, q_V} \equiv a_{k, q_V} - g_{k, q_V}$ can be approximated by the first-order Taylor series expansion at the point $\mathbf{p}_{q_V}$, and this becomes:

$$e_{k, q_V} \approx e_{k, t_V} + \sum_{r = 1}^{R} \frac{\partial e_{k, t_V}}{\partial p_{r, t_V}} (p_{r, t_V} - p_{r, q_V}) \text{, where } \frac{\partial e_{k, t_V}}{\partial p_{r, t_V}} = \frac{\partial a_{k, t_V}}{\partial p_{r, t_V}} - \frac{\partial g_{k, t_V}}{\partial p_{r, t_V}}, \tag{73}$$

where $\mathbf{p}_{t_V}$ is the nearest training point to $\mathbf{p}_{q_V}$. By inserting Eq. (73) into Eq. (72), we obtain

$$E_V \approx \sum_{q_V} \sum_{k} \left\{ e_{k, t_V}^2 + 2 e_{k, t_V} \left(\sum_{r} \frac{\partial e_{k, t_V}}{\partial p_{r, t_V}} (p_{r, t_V} - p_{r, q_V}) \right) + \left(\sum_{r} \frac{\partial e_{k, t_V}}{\partial p_{r, t_V}} (p_{r, t_V} - p_{r, q_V}) \right)^2 \right\} . \tag{74}$$

We may ignore the second term in Eq. (74) if we assume equal distribution around zero of the training error terms, i.e. e_{k, t_V} and $\partial e_{k, t_V} / \partial p_{r, t_V}$ for all $k = 1, 2, \ldots, S^M$, $t_V = 1, 2, \ldots, Q_V$ and $r = 1, 2, \ldots, R$. Thus the sum of these error terms is approximately zero. The third term in Eq. (74) can be further expanded:

$$\left(\sum_{r} \frac{\partial e_{k, t_V}}{\partial p_{r, t_V}} (p_{r, t_V} - p_{r, q_V}) \right)^2 = \sum_{r} \left(\frac{\partial e_{k, t_V}}{\partial p_{r, t_V}} (p_{r, t_V} - p_{r, q_V}) \right)^2 + \sum_{r} \sum_{\substack{r' \\ r' \neq r}} \left(\frac{\partial e_{k, t_V}}{\partial p_{r, t_V}} (p_{r, t_V} - p_{r, q_V}) \right) \left(\frac{\partial e_{k, t_V}}{\partial p_{r', t_V}} (p_{r', t_V} - p_{r', q_V}) \right) . \tag{75}$$

From Eq. (75), the validation performance measure E_V could then be approximated by:

$$\begin{aligned} E_V &\approx \sum_{q_V}\sum_{k} e_{k,\,t_V}^2 + \sum_{q_V}\sum_{k}\left(\sum_{r}\frac{\partial e_{k,\,t_V}}{\partial p_{r,\,t_V}}(p_{r,\,t_V}-p_{r,\,q_V})\right)^2 \\ &= \sum_{q_V}\sum_{k} e_{k,\,t_V}^2 + \sum_{q_V}\sum_{k}\sum_{r}\left(\frac{\partial e_{k,\,t_V}}{\partial p_{r,\,t_V}}(p_{r,\,t_V}-p_{r,\,q_V})\right)^2 + \\ &\quad \sum_{q_V}\sum_{k}\sum_{r}\sum_{\substack{r' \\ r'\neq r}}\left(\frac{\partial e_{k,\,t_V}}{\partial p_{r,\,t_V}}(p_{r,\,t_V}-p_{r,\,q_V})\right)\left(\frac{\partial e_{k,\,t_V}}{\partial p_{r',\,t_V}}(p_{r',\,t_V}-p_{r',\,q_V})\right). \end{aligned} \tag{76}$$

It should be noted that the differences of the two inputs; i.e. $p_{r,\,t_V}-p_{r,\,q_V}$ for all $r = 1, 2, \ldots, R$, are constant. In addition, by the assumption of equal distribution around zero of the training error terms, the last term in Eq. (76) is then approximately zero. Thus, Eq. (76) reduces to

$$E_V \approx \sum_{q_V}\sum_{k} e_{k,\,t_V}^2 + \sum_{q_V}\sum_{k}\sum_{r}\left(\frac{\partial e_{k,\,t_V}}{\partial p_{r,\,t_V}}(p_{r,\,t_V}-p_{r,\,q_V})\right)^2. \tag{77}$$

Again, since the differences of the two inputs are constant, the validation performance measure E_V is proportional to:

$$E_V \propto \sum_{t_V}\sum_{k} e_{k,\,t_V}^2 + \sum_{t_V}\sum_{k}\sum_{r}\left(\frac{\partial e_{k,\,t_V}}{\partial p_{r,\,t_V}}\right)^2, \tag{78}$$

where the summation index $q_V = 1, 2, \ldots, Q_V$ can be replaced by $t_V = 1, 2, \ldots, Q_V$.

From Eq. (78), we can see that the first term is getting smaller, while the network is being trained. Therefore, the increase in the performance measure E_V may be estimated by the increase of the second term, which is the sum squared derivative errors in the training

set. This seems to make sense as large derivative errors in training data will result in large function errors in the validation data. Therefore, if we define a new validation performance measure whose value depends only on the derivative error of the training set, its increase should imply overfitting and we would then want to terminate the training process. The new validation measure is written as

$$\tilde{E}_V = \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \sum_{r=1}^{R} \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \right)^2 , \tag{79}$$

where Q is the number of examples available - the number of data in the training and validation sets. Since the new validation measure uses the derivative information of all data, we can use all the function information for training. This means that we use all the data in the training process, while overfitting is prevented by using the new validation measure $\tilde{E}_V$. An advantage of having more training examples is we have better generalization, see [GaWh92] and [AtPa97]. The following summarizes the method.

Steps for early stopping using the derivative information of the training set

1. Change the validation performance measure for early stopping so that it follows Eq. (79).

2. Use all of the available data for the training process, which can be performed by any standard training algorithm, e.g. $BFGS$ or LM, etc. In other words, there is no data division into training and validation set.

3. The training process is terminated when the new validation measure, i.e. Eq. (79), consecutively increases for a certain number of iterations.

We choose the Levenberg-Marquardt training algorithm (LM) to work with the modified validation performance measure. We denote the method $LM-ES2$. The method requires the calculation of the derivative of the network with respect to the network inputs ($\partial \mathbf{a}_q / \partial \mathbf{p}_q^T$ for all $q = 1, 2, \ldots, Q$), and the calculation is shown in Chapter 4. In the next section, we will provide the simulation results obtained from $LM-ES2$ on the simple analytic problems (introduced in Chapter 7). Recall again that the procedure to perform the simulation is provided in Chapter 7.

Simulation results

In this section, the simulation results for $LM-ES2$ on the simple analytic problems are presented in Table 4, Table 5 and Table 6. For ease of reference, we also put the results obtained from $LM-ES$ in the tables. The results in Table 4, Table 5 and Table 6 can be compared with Table 1, Table 2 and Table 3 for $LM-ES1$, and with Table 9, Table 10 and Table 11 in Chapter 7 for other training algorithms.

Training Algorithm	Problem 1			
	$RMSE_F^{md}$		$RMSE_D^{md}$	
	Training	Test	Training	Test
$LM-ES$	2.48E-06	8.90E-04	6.42E-03	1.74E-02
$LM-ES2$	2.24E-05	3.67E-04	4.78E-03	8.43E-03

Table 4 Approximation accuracy on problem 1

Training Algorithm	Problem 2			
	$RMSE_F^{md}$		$RMSE_D^{md}$	
	Training	Test	Training	Test
$LM-ES$	1.14E-05	1.16E-02	1.09E+00	1.94E+00
$LM-ES2$	1.76E-05	6.44E-03	2.40E-01	8.23E-01

Table 5 Approximation accuracy on problem 2

Training Algorithm	Problem 3			
	$RMSE_F^{md}$		$RMSE_D^{md}$	
	Training	Test	Training	Test
$LM-ES$	6.72E-06	2.09E-04	5.67E-03	1.12E-02
$LM-ES2$	1.16E-05	1.82E-04	7.66E-03	1.41E-02

Table 6 Approximation accuracy on problem 3

From the results, we can see that $LM-ES2$ seems to provide better results than the standard algorithm $LM-ES$ for both function and the derivatives for problem 1 and problem 2. However, it is not the case for problem 3, where the derivative error obtained from $LM-ES2$ was larger than $LM-ES$.

We will analyze the algorithm in the following section. This is to understand why $LM-ES2$ does not consistently yield better results over $LM-ES$, even though the training set for $LM-ES2$ is relatively larger than for $LM-ES$.

Algorithm analysis

Recall that, for $LM-ES2$, the overfitting is prevented by terminating the training process once the derivative error of all training data increases. An advantage of this method is the number of examples in the training process increases from including examples in the

validation set into the training set. The increase in the training examples would lead to a better generalization (see [GaWh92] and [AtPa97]).

We analyzed the $LM-ES2$ method by training a network with the $LM-ES$ algorithm. However, we also monitored the values of E_V (see Eq. (72)) and $\tilde{E}_V$ (see Eq. (79)) along the optimization process. The following figure shows the training records.

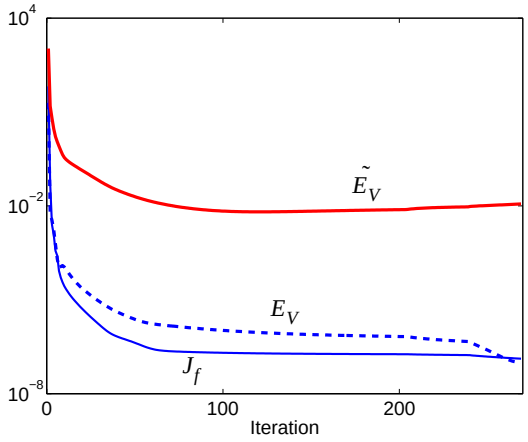

Figure 5) A training record showing $\tilde{E}_V$ versus E_V

From Figure 5), we can see that the derivative error of the training set $\tilde{E}_V$ increased (at around iteration 125^{th}) sooner than the validation performance measure E_V did. This means that the increase in $\tilde{E}_V$ does not directly imply an increase in E_V. This contradicts the assumption we made in the previous section, where we assumed the increase in E_V could be estimated by the increase in $\tilde{E}_V$. The reason behind this lies in Eq. (78).

In Eq. (78), we can see that the regular validation measure E_V is proportional to two terms; the first is the square training function errors, and $\tilde{E}_V$ is the second. This means that

even though the value of $\tilde{E}_V$ increases, if the reduction in the training function error counts more, the regular validation measure E_V will still be lower. From this fact, we conclude that the regular validation measure E_V cannot exactly be estimated by the training derivative error, since one more factor (i.e. the training function error) also counts.

Summary

Recall that our objective is to approximate a function and its first-order derivatives using neural networks. Two new methods were proposed in this chapter: $LM-ES1$ and $LM-ES2$. These methods are similar to the standard training algorithms. However, changes were made in the early stopping technique.

In the $LM-ES1$ method, the validation performance measure was changed from the squared function error to a combination of the squared function and derivative errors. The simulation results showed that the effectiveness of this new validation measure is similar to that of the standard early stopping.

In the second method ($LM-ES2$), the validation performance measure was changed to the squared derivative error of the training set. In this method, data division into training and validation sets is no longer needed, unlike in the standard early stopping. The simulation results showed that the new validation measure sometimes terminates the training process too soon (i.e. sooner than using the standard early stopping), causing worse approximation.

CHAPTER 4

GRADIENT-BASED COMBINED FUNCTION AND DERIVATIVE APPROXIMATION

Introduction

Recall that the objective of this research is to use neural networks for approximating functions and their first-order derivatives. In this chapter, we will propose a new training algorithm to perform this function. This algorithm is designed to work with any gradient-based optimization method (e.g. steepest descent, conjugate gradient, $BFGS$, etc.). We call this algorithm the Combined Function and Derivative Approximation ($CFDA$) algorithm.

We will start this chapter by introducing the performance index used in the $CFDA$ method and will derive two approaches for gradient calculation. At the end of the chapter, we will provide some examples to illustrate how fast the new training algorithm is in comparison with the standard algorithm.

Gradient-Based Combined Function and Derivative Approximation

In Chapter 2, the conditions under which the neural network can approximate both a function and its first-order derivatives were discussed. We will introduce the performance index for the $CFDA$ method, assuming these conditions are satisfied. Then, we will focus on the derivation of the gradient of the performance index, which is required for any gradi-

ent-based optimization method. We will develop two approaches for gradient calculation. The first approach assumes the gradient computation is in batch mode, i.e. data are used at the same time, and this is performed by arranging information in matrices. This may be necessary in some programming languages in order to speed up the calculation. The second method, however, offers a trade-off between the computation time and the required memory.

Performance Index

Assume we want to approximate the function $\mathbf{g}$, which maps a subset in $\Re^R$ to a subset in $\Re^{S^M}$, and its first-order derivatives, by an $M-layer$ neural network (with the conditions discussed in Chatper 2). Also suppose $g_k \in C^1$ for all $k = 1, 2, \ldots, S^M$. The proposed performance index is written as:

$$\begin{aligned} J &= J_f + \rho J_d \\ &= \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\{a_{k,q}-g_{k,q}\}^2 + \frac{\rho}{Q_d S_d^M R_d}\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\sum_{r=1}^{R}\varphi_{k,r}(q)\left(\frac{\partial a_{k,q}}{\partial p_{r,q}}-\frac{\partial g_{k,q}}{\partial p_{r,q}}\right)^2, \end{aligned} \quad (80)$$

where the term ρJ_d is added to form the new performance index. ρ is a scalar value controlling how important the term J_d is, relative to the term J_f. If $\rho = 0$, the new performance index reduces to the standard performance index we discussed in Chapter 2. The function $\varphi_{k,r}(q)$ in the performance index is introduced to cope with the situation when the terms $\partial g_{k,q}/\partial p_{r,q}$ are not available. The function is defined below:

$$\varphi_{k,r}(q) = \begin{cases} 1 & ; \frac{\partial g_{k,q}}{\partial p_{r,q}} \text{ is available.} \\ 0 & ; \text{otherwise.} \end{cases} \tag{81}$$

Minimizing the performance index will force the neural network to simultaneously approximate both the function **g** and its first-order derivatives. We will present two approaches for calculating the gradient of the performance index. The first puts all the calculations in matrices for batch operation, and the second offers a trade-off between the computation speed and the required memory. The derivations for these two approaches will be shown in the next section.

Gradient Calculation: Batch Operation

The batch mode operation will be discussed in this section. To minimize the performance index using gradient-based optimization methods, we need to compute the derivative of the performance index with respect to each network parameter. This means that we need to compute $\partial J / \partial w_{i,j}^m$ and $\partial J / \partial b_i^m$, where $i = 1, 2, \ldots, S^m$ and $j = 1, 2, \ldots, S^{m-1}$. Note that we will show only how to compute $\partial J_d / \partial w_{i,j}^m$ and $\partial J_d / \partial b_i^m$, since the terms $\partial J_f / \partial w_{i,j}^m$ and $\partial J_f / \partial b_i^m$ can be computed using the standard backpropagation algorithm [HaDe96].

From Eq. (80), by taking the derivative of J_d with respect to $w_{i,j}^m$ (note that the constant term $1/(Q_d S_d^M R_d)$ is temporarily dropped from the term J_d for simplicity), we have:

$$\frac{\partial J_d}{\partial w_{i,j}^m} = \frac{\partial}{\partial w_{i,j}^m}\left(\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\sum_{r=1}^{R}\varphi_{k,r}(q)\left(\frac{\partial a_{k,q}}{\partial p_{r,q}} - \frac{\partial g_{k,q}}{\partial p_{r,q}}\right)^2\right)$$
$$= 2\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\sum_{r=1}^{R}\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial w_{i,j}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) . \tag{82}$$

where

$$\frac{\partial e_{k,q}}{\partial p_{r,q}} \equiv \varphi_{k,r}(q)\left(\frac{\partial a_{k,q}}{\partial p_{r,q}} - \frac{\partial g_{k,q}}{\partial p_{r,q}}\right). \tag{83}$$

Since the term $\partial e_{k,q} / \partial p_{r,q}$ can be computed with standard backpropagation, we will focus on the computation of the term $\frac{\partial}{\partial w_{i,j}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right)$. Note that

$$\frac{\partial}{\partial w_{i,j}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial w_{i,j}^m}\right). \tag{84}$$

From Eq. (84), we can use the chain rule of calculus to compute the term $\partial e_{k,q} / \partial w_{i,j}^m$. This is also a part of the standard backpropapgation [HaDe96], which can be computed as follows:

$$\frac{\partial e_{k,q}}{\partial w_{i,j}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} . \tag{85}$$

The term $n_{i,q}^m$ can be calculated using Eq. (14). Thus, Eq. (85) becomes

$$\frac{\partial e_{k,q}}{\partial w_{i,j}^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times a_{j,q}^{m-1}. \tag{86}$$

From Eq. (86), Eq. (84) becomes

$$\begin{aligned}\frac{\partial}{\partial w_{i,j}^{m}}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) &= \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial w_{i,j}^{m}}\right) \\ &= \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial n_{i,q}^{m}} \times a_{j,q}^{m-1}\right) \\ &= \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial n_{i,q}^{m}}\right) \times a_{j,q}^{m-1} + \frac{\partial e_{k,q}}{\partial n_{i,q}^{m}} \times \frac{\partial a_{j,q}^{m-1}}{\partial p_{r,q}} \quad .\end{aligned} \tag{87}$$

Using Eq. (87), Eq. (82) becomes

$$\begin{aligned}\frac{\partial J_d}{\partial w_{i,j}^{m}} = 2\Bigg\{ &\sum_{q}\sum_{k}\sum_{r}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}}\left\{\frac{\partial e_{k,q}}{\partial n_{i,q}^{m}}\right\} \times a_{j,q}^{m-1}\right) \\ &+ \sum_{q}\sum_{k}\sum_{r}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n_{i,q}^{m}} \times \frac{\partial a_{j,q}^{m-1}}{\partial p_{r,q}}\right)\Bigg\} \quad .\end{aligned} \tag{88}$$

By rearranging the summations, we obtain

$$\begin{aligned}\frac{\partial J_d}{\partial w_{i,j}^{m}} = 2\Bigg\{ &\sum_{q}\left(a_{j,q}^{m-1}\sum_{k}\sum_{r}\left\{\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial n_{i,q}^{m}}\right)\right\}\right) \\ &+ \sum_{q}\sum_{r}\left(\frac{\partial a_{j,q}^{m-1}}{\partial p_{r,q}}\sum_{k}\left\{\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n_{i,q}^{m}}\right\}\right)\Bigg\} \quad .\end{aligned} \tag{89}$$

To further compute Eq. (89), define

$$v_{i,q}^{m} \equiv \sum_{k}\sum_{r}\left\{\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial n_{i,q}^{m}}\right)\right\} \text{ , and} \tag{90}$$

$$u_{q_{i,r}}^{m} \equiv \sum_{k}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n_{i,q}^{m}}\right). \tag{91}$$

By using Eq. (90) and Eq. (91), Eq. (89) becomes (with the term $1/(Q_d S_d^M R_d)$ returned):

$$\frac{\partial J_d}{\partial w_{i,j}^m} = \frac{2}{Q_d S_d^M R_d}\left(\sum_q \{ a_{j,q}^{m-1} v_{i,q}^m \} + \sum_q \sum_r \left\{ \frac{\partial a_{j,q}^{m-1}}{\partial p_{r,q}} u_{q_{i,r}}^m \right\}\right). \tag{92}$$

In matrix form, Eq. (92) can also be written as

$$\frac{\partial J_d}{\partial \mathbf{W}^m} = \frac{2}{Q_d S_d^M R_d} \sum_q \left\{ \mathbf{v}_q^m (\mathbf{a}_q^{m-1})^T + \mathbf{U}_q^m \left(\frac{\partial \mathbf{a}_q^{m-1}}{\partial \mathbf{p}_q^T} \right)^T \right\}, \tag{93}$$

where the $S^m \times 1$ vector $\mathbf{v}_q^m$ consists of the terms $v_{i,q}^m$, for all $i = 1, 2, \ldots, S^m$. The $S^m \times R$ matrix $\mathbf{U}_q^m$ consists of the terms $u_{q_{i,r}}^m$, for all $i = 1, 2, \ldots, S^m$ and $r = 1, 2, \ldots, R$.

To write Eq. (92) in batch mode, we first define the $S^m Q \times RQ$ matrix $\mathbf{U}^m$:

$$\mathbf{U}^m \equiv \begin{bmatrix} \mathbf{U}_1^m & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2^m & \ldots & \mathbf{0} \\ \ldots & \ldots & & \ldots \\ \mathbf{0} & \mathbf{0} & \ldots & \mathbf{U}_Q^m \end{bmatrix}. \tag{94}$$

By Eq. (94), Eq. (92) can be rewritten as

$$\begin{aligned} \frac{\partial J_d}{\partial \mathbf{W}^m} = \frac{2}{Q_d S_d^M R_d} \Bigg\{ & \mathbf{V}^m (\mathbf{A}^{m-1})^T + \\ & (\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^m}) \times \mathbf{U}^m \left(\frac{\partial vec\mathbf{A}^{m-1}}{\partial (vec\mathbf{P})^T} \right)^T \times (\mathbf{1}_{Q\times 1} \otimes \mathbf{I}_{S^{m-1}}) \Bigg\}, \end{aligned} \tag{95}$$

where the $S^m \times Q$ matrix $\mathbf{V}^m$ is

$$\mathbf{V}^m = \left[\mathbf{v}_1^m \ \mathbf{v}_2^m \ \ldots \ \mathbf{v}_Q^m \right]. \tag{96}$$

Note that

$$(\mathbf{1}_{1\times Q}\otimes \mathbf{I}_{S^m})\times \mathbf{U}^m \;=\; \left[\mathbf{U}_1^m\;\mathbf{U}_2^m\;\ldots\;\mathbf{U}_Q^m\right], \text{ and} \tag{97}$$

$$\left(\frac{\partial vec\mathbf{A}^{m-1}}{\partial (vec\mathbf{P})^T}\right)^T \times (\mathbf{1}_{Q\times 1}\otimes \mathbf{I}_{S^{m-1}}) \;=\; \left[\frac{\partial \mathbf{a}_1^{m-1}}{\partial \mathbf{p}_1^T}\;\frac{\partial \mathbf{a}_2^{m-1}}{\partial \mathbf{p}_2^T}\;\ldots\;\frac{\partial \mathbf{a}_Q^{m-1}}{\partial \mathbf{p}_Q^T}\right]^T, \tag{98}$$

are the $S^m \times RQ$ and $RQ \times S^{m-1}$ matrices, respectively. Note further that Eq. (83) can be written in batch mode as:

$$\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{P})^T} \;=\; \Phi \bullet \left\{\frac{\partial vec\mathbf{A}}{\partial (vec\mathbf{P})^T} - \frac{\partial vec\mathbf{G}}{\partial (vec\mathbf{P})^T}\right\} \;=\; \begin{bmatrix} \frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{e}_2}{\partial \mathbf{p}_2^T} & \ldots & \mathbf{0} \\ \ldots & \ldots & & \ldots \\ \mathbf{0} & \mathbf{0} & \ldots & \frac{\partial \mathbf{e}_Q}{\partial \mathbf{p}_Q^T} \end{bmatrix}. \tag{99}$$

The $S^M Q \times RQ$ matrix Φ is defined as:

$$\Phi \equiv \begin{bmatrix} \Phi_1 & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \Phi_2 & \ldots & \mathbf{0} \\ \ldots & \ldots & & \ldots \\ \mathbf{0} & \mathbf{0} & \ldots & \Phi_Q \end{bmatrix}, \tag{100}$$

where Φ_q is the $S^M \times R$ matrix consisting of the elements $\varphi_{k,r}(q)$, for all $k = 1, 2, \ldots, S^M$ and $r = 1, 2, \ldots, R$.

In addition to $\partial J_d / \partial w_{i,j}^m$, we also need to compute the derivative of the performance index with respect to the biases, i.e. $\partial J_d / \partial b_i^m$. In Eq. (82), the term $\partial J_d / \partial w_{i,j}^m$ is changed to $\partial J_d / \partial b_i^m$, and using the fact that

$$\frac{\partial}{\partial b_i^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial b_i^m}\right), \tag{101}$$

thus

$$\frac{\partial J_d}{\partial b_i^m} = \frac{2}{Q_d S_d^M R_d} \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \sum_{r=1}^{R} \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}}\left\{\frac{\partial e_{k,q}}{\partial b_i^m}\right\}\right). \tag{102}$$

As in Eq. (85), the term $\partial e_{k,q} / \partial b_i^m$ can be computed as

$$\frac{\partial e_{k,q}}{\partial b_i^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial n_{i,q}^m}{\partial b_i^m}, \tag{103}$$

and by Eq. (14), this reduces to

$$\frac{\partial e_{k,q}}{\partial b_i^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m}. \tag{104}$$

Thus, Eq. (102) becomes

$$\frac{\partial J_d}{\partial b_i^m} = \frac{2}{Q_d S_d^M R_d} \sum_q \sum_k \sum_r \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}}\left\{\frac{\partial e_{k,q}}{\partial n_{i,q}^m}\right\}\right). \tag{105}$$

From Eq. (90), Eq. (105) reduces to

$$\frac{\partial J_d}{\partial b_i^m} = \frac{2}{Q_d S_d^M R_d} \sum_q v_{i,q}^m. \tag{106}$$

Therefore, the $S^m \times 1$ vector $\partial J_d / \partial \mathbf{b}^m$ can be written as

$$\frac{\partial J_d}{\partial \mathbf{b}^m} = \frac{2}{Q_d S_d^M R_d} \sum_q \mathbf{v}_q^m = \frac{2}{Q_d S_d^M R_d} \{ \mathbf{V}^m \times \mathbf{1}_{Q \times 1} \} . \tag{107}$$

From Eq. (95) and Eq. (107), we need to calculate the elements of the matrices $\mathbf{V}^m$, $\mathbf{U}^m$ and $\partial vec\mathbf{A}^m / \partial (vec\mathbf{P})^T$. We will first show how to compute $\partial vec\mathbf{A}^m / \partial (vec\mathbf{P})^T$, followed by $\mathbf{U}^m$, and finally $\mathbf{V}^m$. We will break these calculations into three sections.

I. Calculation of $\partial vec\mathbf{A}^m / \partial (vec\mathbf{P})^T$

An element of this matrix is $\partial a_{j,q}^m / \partial p_{r,q}$. From Eq. (14), by using the chain rule of calculus, we obtain

$$\frac{\partial a_{j,q}^m}{\partial p_{r,q}} = \frac{\partial a_{j,q}^m}{\partial n_{j,q}^m} \times \frac{\partial n_{j,q}^m}{\partial p_{r,q}} . \tag{108}$$

Using Eq. (14), Eq. (108) becomes

$$\frac{\partial a_{j,q}^m}{\partial p_{r,q}} = \dot{f}^m (n_{j,q}^m) \sum_{l=1}^{S^{m-1}} \left(w_{j,l}^m \times \frac{\partial a_{l,q}^{m-1}}{\partial p_{r,q}} \right) , \tag{109}$$

where

$$\dot{f}^m (n_{j,q}^m) = \frac{\partial a_{j,q}^m}{\partial n_{j,q}^m} . \tag{110}$$

Eq. (109) can be written in matrix form as

$$\frac{\partial \mathbf{a}_q^m}{\partial \mathbf{p}_q^T} = \dot{\mathbf{F}}^m (\mathbf{n}_q^m) \mathbf{W}^m \frac{\partial \mathbf{a}_q^{m-1}}{\partial \mathbf{p}_q^T} , \tag{111}$$

where $\dot{\mathbf{F}}^m(\mathbf{n}_q^m)$ is the $S^m \times S^m$ matrix defined below:

$$\dot{\mathbf{F}}^m(\mathbf{n}_q^m) \equiv \frac{\partial \mathbf{a}_q^m}{\partial (\mathbf{n}_q^m)^T} = \begin{bmatrix} \dot{f}^m(n_{1,q}^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_{2,q}^m) & \dots & 0 \\ \dots & \dots & & \dots \\ 0 & 0 & \dots & \dot{f}^m\left(n_{S^m,q}^m\right) \end{bmatrix}. \tag{112}$$

In batch mode, we have

$$\frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{N}^m)^T} = \begin{bmatrix} \dot{\mathbf{F}}^m(\mathbf{n}_1^m) & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \dot{\mathbf{F}}^m(\mathbf{n}_2^m) & \dots & \mathbf{0} \\ \dots & \dots & & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \dot{\mathbf{F}}^m(\mathbf{n}_Q^m) \end{bmatrix}. \tag{113}$$

Thus, from Eq. (111), the matrix $\partial vec\mathbf{A}^m / \partial (vec\mathbf{P})^T$ can be computed as

$$\frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{P})^T} = \frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^m) \times \frac{\partial vec\mathbf{A}^{m-1}}{\partial (vec\mathbf{P})^T}. \tag{114}$$

From Eq. (114), we can see that computing $\partial vec\mathbf{A}^m / \partial (vec\mathbf{P})^T$ requires the calculation of $\partial vec\mathbf{A}^{m-1} / \partial (vec\mathbf{P})^T$. Thus, we need to initialize $\partial vec\mathbf{A}^0 / \partial (vec\mathbf{P})^T$ (i.e. $m = 0$). From the fact that $a_{j,q}^0$ is the input $p_{j,q}$, thus $S^0 = R$ and we can compute $\partial a_{j,q}^0 / \partial p_{r,q}$ as follows:

$$\frac{\partial a_{j,q}^0}{\partial p_{r,q}} = \begin{cases} 1 & ; \text{if } j = r. \\ 0 & ; \text{if } j \neq r. \end{cases} \tag{115}$$

Thus, the $R \times R$ matrix $\partial \mathbf{a}_q^0 / \partial \mathbf{p}_q^T$ is

$$\frac{\partial \mathbf{a}_q^0}{\partial \mathbf{p}_q^T} = \frac{\partial \mathbf{p}_q}{\partial \mathbf{p}_q^T} = \mathbf{I}_R \ . \tag{116}$$

Therefore, in batch mode, the $RQ \times RQ$ matrix $\partial vec\mathbf{A}^0 / \partial (vec\mathbf{P})^T$ becomes

$$\frac{\partial vec\mathbf{A}^0}{\partial (vec\mathbf{P})^T} = \begin{bmatrix} \mathbf{I}_R & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_R & \dots & \mathbf{0} \\ \dots & \dots & & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{I}_R \end{bmatrix} = \mathbf{I}_{RQ} \ . \tag{117}$$

This completes the calculation for the matrix $\partial vec\mathbf{A}^m / \partial (vec\mathbf{P})^T$. Next, we will show the calculation for $\mathbf{U}^m$.

II. Calculation of $\mathbf{U}^m$

We begin by expanding the term in the term $\partial e_{k,q} / \partial n_{i,q}^m$ in Eq. (91). Using the chain rule of calculus:

$$\frac{\partial e_{k,q}}{\partial n_{i,q}^m} = \sum_{l=1}^{S^{m+1}} \left(\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}} \right) \left(\frac{\partial n_{l,q}^{m+1}}{\partial a_{i,q}^m} \right) \left(\frac{\partial a_{i,q}^m}{\partial n_{i,q}^m} \right) , \tag{118}$$

From Eq. (14), we obtain

$$\frac{\partial e_{k,q}}{\partial n_{i,q}^m} = \sum_l \left(\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}} \right) w_{l,i}^{m+1} \dot{f}^m (n_{i,q}^m) \ . \tag{119}$$

If we substitute this expression into Eq. (91) and rearrange the summation, we find

$$
\begin{aligned}
u_{q_{i,r}}^{m} &= \sum_{k}\left\{\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \sum_{l}\left(\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}}\right) w_{l,i}^{m+1} \dot{f}^{m}(n_{i,q}^{m})\right\} \\
&= \dot{f}^{m}(n_{i,q}^{m}) \sum_{l}\left\{ w_{l,i}^{m+1} \sum_{k}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}}\right)\right\} .
\end{aligned}
\tag{120}
$$

Using Eq. (91), Eq. (120) becomes

$$
u_{q_{i,r}}^{m} = \dot{f}^{m}(n_{i,q}^{m}) \sum_{l}\left\{ w_{l,i}^{m+1} u_{q_{l,r}}^{m+1} \right\} . \tag{121}
$$

Eq. (121) can be written in matrix form as

$$
\mathbf{U}_{q}^{m} = \dot{\mathbf{F}}^{m}(\mathbf{n}_{q}^{m})(\mathbf{W}^{m+1})^{T} \mathbf{U}_{q}^{m+1} . \tag{122}
$$

From Eq. (94) and Eq. (113), the batch matrix $\mathbf{U}^{m}$ can be expressed as

$$
\mathbf{U}^{m} = \frac{\partial vec\mathbf{A}^{m}}{\partial (vec\mathbf{N}^{m})^{T}} \times \left\{ \mathbf{I}_{Q} \otimes (\mathbf{W}^{m+1})^{T} \right\} \times \mathbf{U}^{m+1} . \tag{123}
$$

We need to initialize this equation with $\mathbf{U}^{M}$. From Eq. (91) with $m = M$, we have

$$
u_{q_{i,r}}^{M} = \sum_{k}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n_{i,q}^{M}}\right) . \tag{124}
$$

Since $e_{k,q} = a_{k,q} - g_{k,q}$ and $a_{k,q} = a_{k,q}^{M}$,

$$
\frac{\partial e_{k,q}}{\partial n_{i,q}^{M}} = \frac{\partial a_{k,q}}{\partial n_{i,q}^{M}} = \begin{cases} \dot{f}^{M}(n_{i,q}^{M}) & ; \text{if } i = k \\ 0 & ; \text{if } i \neq k \end{cases} . \tag{125}
$$

Therefore, Eq. (124) reduces to

$$
u_{q_{i,r}}^{M} = \frac{\partial e_{i,q}}{\partial p_{r,q}} \dot{f}^{M}(n_{i,q}^{M}) . \tag{126}
$$

From Eq. (126), $\mathbf{U}_q^M$ can be expressed as

$$\mathbf{U}_q^M = \dot{\mathbf{F}}^M(\mathbf{n}_q^M)\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T} \ . \tag{127}$$

Therefore, by Eq. (94) and Eq. (99), the batch matrix $\mathbf{U}^M$ is written as

$$\mathbf{U}^M = \frac{\partial vec\mathbf{A}}{\partial (vec\mathbf{N}^M)^T} \times \frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{P})^T} \ . \tag{128}$$

Note that if f^M is the linear function, then $\dot{\mathbf{F}}^M(\mathbf{n}_q^M) = \mathbf{I}_{S^M}$, and the batch matrix

$\partial vec\mathbf{A} / \partial (vec\mathbf{N}^M)^T = \mathbf{I}_{S^M Q}$.

This completes the calculation for $\mathbf{U}^m$. Next, we will compute $\mathbf{V}^m$.

III. Calculation of $\mathbf{V}^m$

From Eq. (90), by using the chain rule of calculus for the term $\partial e_{k,q} / \partial n_{i,q}^m$ which is presented in Eq. (119), we obtain

$$v_{i,q}^m = \sum_r \sum_k \frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}} \left\{ \sum_{l=1}^{S^{m+1}} \left(\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}} \right) w_{l,i}^{m+1} \dot{f}^m(n_{i,q}^m) \right\} \ . \tag{129}$$

Since the last term in Eq. (129), $\dot{f}^m(n_{i,q}^m)$, does not depend on l, it can be brought outside the summation $\sum_l$. Then, Eq. (129) becomes

$$
\begin{aligned}
v_{i,q}^{m} &= \sum_{r}\sum_{k}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}}\left\{\dot{f}^{m}(n_{i,q}^{m})\sum_{l}\left(\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}}\right) w_{l,i}^{m+1}\right\}\right) \\
&= \sum_{r}\sum_{k}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial \dot{f}^{m}(n_{i,q}^{m})}{\partial p_{r,q}}\sum_{l}\left\{\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}} \times w_{l,i}^{m+1}\right\}\right) + \\
&\quad \sum_{r}\sum_{k}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \dot{f}^{m}(n_{i,q}^{m})\frac{\partial}{\partial p_{r,q}}\left\{\sum_{l}\left(\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}} \times w_{l,i}^{m+1}\right)\right\}\right) \quad .
\end{aligned}
\tag{130}
$$

By rearranging the terms in Eq. (130), this results in

$$
\begin{aligned}
v_{i,q}^{m} &= \sum_{r}\left(\frac{\partial \dot{f}^{m}(n_{i,q}^{m})}{\partial p_{r,q}}\sum_{l}\left\{w_{l,i}^{m+1}\sum_{k}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}}\right)\right\}\right) + \\
&\quad \dot{f}^{m}(n_{i,q}^{m})\left(\sum_{l}\left(w_{l,i}^{m+1}\sum_{r}\sum_{k}\left\{\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}}\right)\right\}\right)\right) \quad .
\end{aligned}
\tag{131}
$$

From Eq. (90) and Eq. (91), Eq. (131) reduces to

$$
v_{i,q}^{m} = \sum_{r}\left(\frac{\partial \dot{f}^{m}(n_{i,q}^{m})}{\partial p_{r,q}}\sum_{l}\left\{w_{l,i}^{m+1} u_{q_{l,r}}^{m+1}\right\}\right) + \dot{f}^{m}(n_{i,q}^{m})\sum_{l}\{w_{l,i}^{m+1} v_{l,q}^{m+1}\} . \tag{132}
$$

To write Eq. (132) for batch mode, first consider the term $\partial \dot{f}^{m}(n_{i,q}^{m}) / \partial p_{r,q}$. The term $\dot{f}^{m}(n_{i,q}^{m})$ depends only on the term $f^{m}(n_{i,q}^{m})$, i.e. $\dot{f}^{m}(n_{i,q}^{m})$ does not depend on $f^{m}(n_{j,q}^{m})$ when $i \neq j$ for $i, j = 1, 2, \ldots, S^{m}$. Thus, $\dot{f}^{m}(n_{i,q}^{m})$ can be considered the i^{th} element of the $S^{m} \times 1$ vector function:

$$\mathbf{df}^m(\mathbf{n}_q^m) \equiv \begin{bmatrix} \dot{f}^m(n_{1,q}^m) \\ \dot{f}^m(n_{2,q}^m) \\ \cdots \\ \dot{f}^m\left(n_{S^m,q}^m\right) \end{bmatrix}, \tag{133}$$

which are the diagonal elements of the matrix $\dot{\mathbf{F}}^m(\mathbf{n}_q^m)$. In other words,

$$\mathbf{df}^m(\mathbf{n}_q^m) \;=\; \dot{\mathbf{F}}^m(\mathbf{n}_q^m) \times \mathbf{1}_{S^m \times 1}. \tag{134}$$

From Eq. (133), further define the $S^m \times Q$ batch matrix:

$$\mathbf{DF}^m(\mathbf{N}^m) \equiv \left[\mathbf{df}^m(\mathbf{n}_1^m)\ \mathbf{df}^m(\mathbf{n}_2^m)\ \ldots\ \mathbf{df}^m(\mathbf{n}_Q^m)\right]. \tag{135}$$

It can be obtained from the matrix $\partial vec\mathbf{A}^m / \partial (vec\mathbf{N}^m)^T$ by

$$\begin{aligned} &(\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^m}) \times \frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{I}_Q \otimes \mathbf{1}_{S^m \times 1}) \\ &= \left[\dot{\mathbf{F}}^m(\mathbf{n}_1^m)\ \dot{\mathbf{F}}^m(\mathbf{n}_2^m)\ \ldots\ \dot{\mathbf{F}}^m(\mathbf{n}_Q^m)\right] \times (\mathbf{I}_Q \otimes \mathbf{1}_{S^m \times 1}) \\ &= \mathbf{DF}^m(\mathbf{N}^m)\ . \end{aligned} \tag{136}$$

The batch derivative of the matrix $\mathbf{DF}^m(\mathbf{N}^m)$ with respect to the input is:

$$\frac{\partial vec\mathbf{DF}^m(\mathbf{N}^m)}{\partial (vec\mathbf{P})^T} = \begin{bmatrix} \frac{\partial \mathbf{df}^m(\mathbf{n}_1^m)}{\partial \mathbf{p}_1^T} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{df}^m(\mathbf{n}_2^m)}{\partial \mathbf{p}_2^T} & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial \mathbf{df}^m(\mathbf{n}_Q^m)}{\partial \mathbf{p}_Q^T} \end{bmatrix}, \tag{137}$$

We can now represent Eq. (132) in vector form:

$$\mathbf{v}_q^m = \left(\frac{\partial \mathbf{df}^m(\mathbf{n}_q^m)}{\partial \mathbf{p}_q^T} \bullet \left\{ (\mathbf{W}^{m+1})^T \mathbf{U}_q^{m+1} \right\} \right) \mathbf{1}_{R \times 1} + \dot{\mathbf{F}}^m(\mathbf{n}_q^m)(\mathbf{W}^{m+1})^T \mathbf{v}_q^{m+1}. \tag{138}$$

Using Eq. (135) and Eq. (137), this can be put in batch matrix form:

$$\mathbf{V}^m = (\mathbf{1}_{1 \times Q} \otimes \mathbf{I}_{S^m}) \times \left\{ \frac{\partial vec\mathbf{DF}^m(\mathbf{N}^m)}{\partial (vec\mathbf{P})^T} \bullet \left(\left\{ \mathbf{I}_Q \otimes (\mathbf{W}^{m+1})^T \right\} \mathbf{U}^{m+1} \right) \right\} \times (\mathbf{I}_Q \otimes \mathbf{1}_{R \times 1}) + \mathbf{DF}^m(\mathbf{N}^m) \bullet \{ (\mathbf{W}^{m+1})^T \mathbf{V}^{m+1} \}. \tag{139}$$

$\mathbf{V}^M$ is needed to initialize Eq. (139). From Eq. (90), we have

$$v_{i,q}^M = \sum_k \sum_r \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}} \left\{ \frac{\partial e_{k,q}}{\partial n_{i,q}^M} \right\} \right). \tag{140}$$

From Eq. (125), this reduces to

$$v_{i,q}^M = \sum_r \left(\frac{\partial e_{i,q}}{\partial p_{r,q}} \times \frac{\partial \dot{f}_{i,q}^M}{\partial p_{r,q}} \right). \tag{141}$$

Therefore, $\mathbf{v}_q^M$ can be written as

$$\mathbf{v}_q^M = \left(\frac{\partial \mathbf{df}^M(\mathbf{n}_q^M)}{\partial \mathbf{p}_q^T} \bullet \frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T} \right) \mathbf{1}_{R \times 1} \ . \tag{142}$$

Using Eq. (99) and Eq. (137), the batch matrix $\mathbf{V}^M$ can be then expressed as

$$\mathbf{V}^M = (\mathbf{1}_{1 \times Q} \otimes \mathbf{I}_{S^M}) \times \left(\frac{\partial vec\mathbf{DF}^M(\mathbf{N}^M)}{\partial (vec\mathbf{P})^T} \bullet \frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{P})^T} \right) \times (\mathbf{I}_Q \otimes \mathbf{1}_{R \times 1}) . \tag{143}$$

It now remains to compute $\partial vec\mathbf{DF}^M(\mathbf{N}^M) / \partial (vec\mathbf{P})^T$. Consider one element of this matrix, i.e. $\partial \dot{f}^m(n_{i,q}^m) / \partial p_{r,q}$. Using the chain rule of calculus:

$$\frac{\partial \dot{f}^m(n_{i,q}^m)}{\partial p_{r,q}} = \frac{\partial \dot{f}^m(n_{i,q}^m)}{\partial a_{i,q}^m} \times \frac{\partial a_{i,q}^m}{\partial p_{r,q}} \ . \tag{144}$$

It may seem unusual that we are using $a_{i,q}^m$ as the itermediate variable here. In fact, it is often true that $\dot{f}^m(n_{i,q}^m)$ can be written as a function of $a_{i,q}^m$. For example, if f^m is the hyperbolic tangent sigmoid function, i.e.

$$f^m(n) = \frac{e^n - e^{-n}}{e^n + e^{-n}} \ , \tag{145}$$

then the derivative of f^m with respect to n, i.e. $\dot{f}^m(n)$, is

$$\dot{f}^m(n) = \frac{\partial f^m}{\partial n} = 1 - \left(\frac{e^n - e^{-n}}{e^n + e^{-n}} \right)^2 = 1 - (f^m(n))^2 = 1 - (a^m)^2 . \tag{146}$$

With Eq. (144), the $S^m \times R$ matrix $\partial \mathbf{df}^m(\mathbf{n}_q^m) / \partial \mathbf{p}_q^T$ can be expressed as

$$\frac{\partial \mathbf{df}^m(\mathbf{n}_q^m)}{\partial \mathbf{p}_q^T} = \frac{\partial \mathbf{df}^m(\mathbf{n}_q^m)}{\partial (\mathbf{a}_q^m)^T} \times \frac{\partial \mathbf{a}_q^m}{\partial \mathbf{p}_q^T} \quad , \tag{147}$$

where, the $S^m \times S^m$ matrix $\partial \mathbf{df}^m(\mathbf{n}_q^m)/\partial(\mathbf{a}_q^m)^T$ is

$$\frac{\partial \mathbf{df}^m(\mathbf{n}_q^m)}{\partial (\mathbf{a}_q^m)^T} = \begin{bmatrix} \frac{\partial \dot{f}^m(n_{1,q}^m)}{\partial a_{1,q}^m} & 0 & \cdots & 0 \\ 0 & \frac{\partial \dot{f}^m(n_{2,q}^m)}{\partial a_{2,q}^m} & \cdots & 0 \\ \cdots & \cdots & & \cdots \\ 0 & 0 & \cdots & \frac{\partial \dot{f}^m\left(n_{S^m,q}^m\right)}{\partial a_{S^m,q}^m} \end{bmatrix} . \tag{148}$$

From Eq. (147), the batch matrix $\partial vec\mathbf{DF}^m(\mathbf{N}^m)/\partial(vec\mathbf{P})^T$ can be decomposed to

$$\frac{\partial vec\mathbf{DF}^m(\mathbf{N}^m)}{\partial (vec\mathbf{P})^T} = \frac{\partial vec\mathbf{DF}^m(\mathbf{N}^m)}{\partial (vec\mathbf{A}^m)^T} \times \frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{P})^T} \quad , \tag{149}$$

At the output layer (i.e. $m = M$), Eq. (144), Eq. (147) and Eq. (149) also apply, with $m = M$ and $a_{i,q}^M = a_{i,q}$, $\mathbf{a}_q^M = \mathbf{a}_q$ and $\mathbf{A}^M = \mathbf{A}$.

This completes the derivation of the batch form of gradient of the performance index J_d with respect to the network parameters $w_{i,j}^m$ and b_i^m. In some programming languages (e.g. MATLAB) batch algorithms are more efficient than computing element by element. However, performing the calculation in batch mode requires sufficient memory to

hold all of the matrix elements at once. Therefore, the larger the matrices, the more memory is needed. The next section will derive an algorithm that is designed to save memory.

Before going to the derivation of the memory-save method, let's summarize the algorithm in this section.

STEPS TO COMPUTE $\partial J/\partial \mathbf{W}^m$ AND $\partial J/\partial \mathbf{b}^m$ (BATCH MODE)

1. Given $\mathbf{P}$, compute $\mathbf{A}^m$, and $\partial vec\mathbf{A}^m/\partial(vec\mathbf{P})^T$, by using Eq. (114) and Eq. (117).

2. Given $\partial vec\mathbf{G}/\partial(vec\mathbf{P})^T$, compute the derivative of the errors $\partial vec\mathbf{E}/\partial(vec\mathbf{P})^T$, using Eq. (99).

3. Compute $\mathbf{U}^M$ and $\mathbf{V}^M$, using Eq. (128) and Eq. (143), respectively.

4. Backpropagate $\mathbf{U}^m$ and $\mathbf{V}^m$, by Eq. (123) and Eq. (139), respectively.

5. Compute $\partial J_d/\partial \mathbf{W}^m$ and $\partial J_d/\partial \mathbf{b}^m$ by Eq. (95) and Eq. (107), respectively. Then, compute $\partial J/\partial \mathbf{W}^m$ and $\partial J/\partial \mathbf{b}^m$.

6. Update the weights and biases using any gradient-based optimization technique.

Gradient Calculation: Memory-Save Method

In this section, another approach to compute the gradient of J_d with respect to the network parameters will be discussed. As previously mentioned, this approach will be useful when the machine's memory is insufficient for the batch mode operation. Mainly, the method will break down some matrices into smaller matrices. We will first briefly discuss the concept of how to break down matrices. This will be followed by the derivation of the gradient.

From Eq. (80), we can see that, when comparing with the typical performance index J_f, the new term J_d has the additional summation $\sum_{r=1}^{R}$. This extra summation, when manipulated for batch mode, leads to larger matrices than those matrices in the regular backpropagation. The idea is then to form smaller matrices whose sizes do not depend on the extra summation $\sum_{r}$.

First, from Eq. (90) and Eq. (91), redefine

$$\tilde{v}^m_{r_{i,q}} \equiv \sum_k \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial}{\partial p_{r,q}} \left\{ \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \right\} \right), \text{ and} \tag{150}$$

$$\tilde{u}^m_{r_{i,q}} \equiv \sum_k \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \right). \tag{151}$$

Thus, $\tilde{v}^m_{r_{i,q}}$ and $\tilde{u}^m_{r_{i,q}}$ are the elements at row i and column q of $\tilde{\mathbf{V}}^m_r$ and $\tilde{\mathbf{U}}^m_r$, respectively. From Eq. (90) and Eq. (150), this implies that

$$v_{i,q}^{m} = \sum_{r=1}^{R} \tilde{v}_{r_{i,q}}^{m} \quad . \tag{152}$$

For the term $\partial a_{j,q}^{m-1} / \partial p_{r,q}$, rather than expressing it as an element of $\partial \mathbf{a}_q^{m-1} / \partial \mathbf{p}_q^T$, we express it as the element at row j of $\partial \mathbf{a}_q^{m-1} / \partial p_{r,q}$. It is also the element at row l (note: $l = (q-1)S^{m-1} + j$) and column q of the matrix $\partial vec\mathbf{A}^{m-1} / \partial_r \mathbf{p}^T$ (see Eq. (27) for the notation). Note that

$$(\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^{m-1}}) \times \frac{\partial vec\mathbf{A}^{m-1}}{\partial_r \mathbf{p}^T} = \left[\frac{\partial \mathbf{a}_1^{m-1}}{\partial p_{r,1}} \; \frac{\partial \mathbf{a}_2^{m-1}}{\partial p_{r,2}} \; \cdots \; \frac{\partial \mathbf{a}_Q^{m-1}}{\partial p_{r,Q}} \right]. \tag{153}$$

By Eq. (153), the following matrices immediately follow:

$$(\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{A}}{\partial_r \mathbf{p}^T} = \left[\frac{\partial \mathbf{a}_1}{\partial p_{r,1}} \; \frac{\partial \mathbf{a}_2}{\partial p_{r,2}} \; \cdots \; \frac{\partial \mathbf{a}_Q}{\partial p_{r,Q}} \right], \tag{154}$$

$$(\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{G}}{\partial_r \mathbf{p}^T} = \left[\frac{\partial \mathbf{g}_1}{\partial p_{r,1}} \; \frac{\partial \mathbf{g}_2}{\partial p_{r,2}} \; \cdots \; \frac{\partial \mathbf{g}_Q}{\partial p_{r,Q}} \right], \text{ and} \tag{155}$$

$$(\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{E}}{\partial_r \mathbf{p}^T} = \left[\frac{\partial \mathbf{e}_1}{\partial p_{r,1}} \; \frac{\partial \mathbf{e}_2}{\partial p_{r,1}} \; \cdots \; \frac{\partial \mathbf{e}_Q}{\partial p_{r,1}} \right], \tag{156}$$

where

$$\frac{\partial vec\mathbf{E}}{\partial_r \mathbf{p}^T} = \tilde{\Phi}_r \bullet \left(\frac{\partial vec\mathbf{A}}{\partial_r \mathbf{p}^T} - \frac{\partial vec\mathbf{G}}{\partial_r \mathbf{p}^T} \right). \tag{157}$$

The matrix $\tilde{\Phi}_r$ is defined:

$$\tilde{\Phi}_r \equiv \begin{bmatrix} \phi_r(1) & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \phi_r(2) & \dots & \mathbf{0} \\ \dots & \dots & & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \phi_r(Q) \end{bmatrix}, \tag{158}$$

where $\phi_r(q)$ is the r^{th} column vector of the matrix Φ_q, see Eq. (100).

By Eq. (150) and Eq. (151), we can rewrite Eq. (92) as

$$\frac{\partial J_d}{\partial w_{i,j}^m} = \frac{2}{Q_d S_d^M R_d} \left(\sum_r \left\{ \sum_q (a_{j,q}^{m-1} \tilde{v}_{r_{i,q}}^m) + \sum_q \left(\frac{\partial a_{j,q}^{m-1}}{\partial p_{r,q}} \tilde{u}_{r_{i,q}}^m \right) \right\} \right), \tag{159}$$

which can be further expressed as:

$$\frac{\partial J_d}{\partial \mathbf{W}^m} = \frac{2}{Q_d S_d^M R_d} \sum_{r=1}^{R} \left\{ \tilde{\mathbf{V}}_r^m (\mathbf{A}^{m-1})^T + \tilde{\mathbf{U}}_r^m \left(\frac{\partial vec\mathbf{A}^{m-1}}{\partial_r \mathbf{p}^T} \right)^T \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^{m-1}}) \right\} . \tag{160}$$

It should be noted that the transpose of the matrix in Eq. (153) produces the matrices in Eq. (160), i.e.

$$\left((\mathbf{1}_{1 \times Q} \otimes \mathbf{I}_{S^{m-1}}) \times \frac{\partial vec\mathbf{A}^{m-1}}{\partial_r \mathbf{p}^T} \right)^T = \left(\frac{\partial vec\mathbf{A}^{m-1}}{\partial_r \mathbf{p}^T} \right)^T \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^{m-1}}) . \tag{161}$$

The sizes of the matrices in Eq. (160) are less than or equal to the matrix sizes in Eq. (95), which was for batch mode.

For calculating the term $\partial J_d / \partial \mathbf{b}^m$, we can directly use Eq. (107) since $\mathbf{V}^m$ can be easily obtained by using Eq. (152):

$$\mathbf{V}^m = \sum_{r=1}^{R} \tilde{\mathbf{V}}_r^m . \tag{162}$$

Alternatively, from Eq. (107) and Eq. (162), $\partial J_d / \partial \mathbf{b}^m$ can also be directly expressed in the form of $\tilde{\mathbf{V}}_r^m$ as follows:

$$\frac{\partial J_d}{\partial \mathbf{b}^m} = \frac{2}{Q_d S_d^M R_d} \sum_r (\tilde{\mathbf{V}}_r^m \times \mathbf{1}_{Q \times 1}). \tag{163}$$

From Eq. (160) and Eq. (163), we need to compute $\tilde{\mathbf{V}}_r^m$, $\tilde{\mathbf{U}}_r^m$ and $\partial vec\mathbf{A}^m / \partial_r \mathbf{p}^T$.

We will start with $\partial vec\mathbf{A}^m / \partial_r \mathbf{p}^T$, followed by $\tilde{\mathbf{U}}_r^m$ and $\tilde{\mathbf{V}}_r^m$.

I. Calculation of $\partial vec\mathbf{A}^m / \partial_r \mathbf{p}^T$

From Eq. (109), by using Eq. (112), we can express $\partial \mathbf{a}_q^m / \partial p_{r,q}$ as

$$\frac{\partial \mathbf{a}_q^m}{\partial p_{r,q}} = \dot{\mathbf{F}}^m(\mathbf{n}_q^m)\mathbf{W}^m \frac{\partial \mathbf{a}_q^{m-1}}{\partial p_{r,q}}. \tag{164}$$

Therefore, by Eq. (113) and Eq. (153), $\partial vec\mathbf{A}^m / \partial_r \mathbf{p}^T$ becomes

$$\frac{\partial vec\mathbf{A}^m}{\partial_r \mathbf{p}^T} = \frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^m) \times \frac{\partial vec\mathbf{A}^{m-1}}{\partial_r \mathbf{p}^T}. \tag{165}$$

Since this is a forward propagation process, we need to initialize at $m = 0$. Recall that $S^0 = R$. From Eq. (115), we then have the vector

$$\frac{\partial \mathbf{a}_q^0}{\partial p_{r,q}} = \frac{\partial \mathbf{p}_q}{\partial p_{r,q}} = \begin{bmatrix} 0 \\ \cdots \\ 0 \\ 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix}, \tag{166}$$

where one appears at row r. Thus, the matrix

$$\frac{\partial vec\mathbf{A}^0}{\partial_r \mathbf{p}^T} = \mathbf{I}_Q \otimes \frac{\partial \mathbf{p}_q}{\partial p_{r,q}}, \text{ for any } q. \tag{167}$$

Next, we will derive $\tilde{\mathbf{U}}_r^m$.

II. Calculation of $\tilde{\mathbf{U}}_r^m$

From Eq. (151), using Eq. (119), we have

$$\tilde{u}_{r_{i,q}}^m = \dot{f}^m(n_{i,q}^m) \sum_l \left\{ w_{l,i}^{m+1} \sum_k \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \times \frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}} \right) \right\}, \tag{168}$$

and by Eq. (151), it reduces to

$$\tilde{u}_{r_{i,q}}^m = \dot{f}^m(n_{i,q}^m) \sum_l \left\{ w_{l,i}^{m+1} \tilde{u}_{r_{l,q}}^{m+1} \right\}. \tag{169}$$

Eq. (169) can be expressed in the form of $\tilde{\mathbf{U}}_r^m$, using Eq. (136), as

$$\tilde{\mathbf{U}}_r^m = \mathbf{DF}^m(\mathbf{N}^m) \bullet \{ (\mathbf{W}^{m+1})^T \tilde{\mathbf{U}}_r^{m+1} \}. \tag{170}$$

Since this is a backpropagation process, we need to initialize $\tilde{\mathbf{U}}_r^M$. From Eq. (125), Eq. (151) reduces to

$$\tilde{u}_{r_{i,q}}^{M} = \frac{\partial e_{i,q}}{\partial p_{r,q}} \dot{f}^{M}(n_{i,q}^{M}) \quad . \tag{171}$$

From Eq. (171), $\tilde{\mathbf{U}}_r^M$ can be written, using Eq. (156), as

$$\tilde{\mathbf{U}}_r^M = \mathbf{DF}^M(\mathbf{N}^M) \bullet \left\{ (\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{E}}{\partial_r \mathbf{p}^T} \right\} \quad . \tag{172}$$

Next, we will illustrate how to calculate $\tilde{\mathbf{V}}_r^m$.

III. Calculation of $\tilde{\mathbf{V}}_r^m$

From Eq. (150), by similarly following Eq. (129) to Eq. (132), we obtain

$$\tilde{v}_{r_{i,q}}^{m} = \frac{\partial \dot{f}^{m}(n_{i,q}^{m})}{\partial p_{r,q}} \sum_{l} \left\{ w_{l,i}^{m+1} \tilde{u}_{r_{l,q}}^{m+1} \right\} + \dot{f}^{m}(n_{i,q}^{m}) \sum_{l} \left\{ w_{l,i}^{m+1} \tilde{v}_{r_{l,q}}^{m+1} \right\} \quad . \tag{173}$$

Eq. (173) can be expressed in the form of $\tilde{\mathbf{v}}_{r_q}^m$ as

$$\tilde{\mathbf{v}}_{r_q}^{m} = \frac{\partial \mathbf{df}^{m}(\mathbf{n}_q^m)}{\partial p_{r,q}} \bullet \left\{ (\mathbf{W}^{m+1})^T \tilde{\mathbf{u}}_{r_q}^{m+1} \right\} + \dot{\mathbf{F}}^{m}(\mathbf{n}_q^m)(\mathbf{W}^{m+1})^T \tilde{\mathbf{v}}_{r_q}^{m+1}, \tag{174}$$

where $\tilde{\mathbf{u}}_{r_q}^{m+1}$ is the q^{th} column vector of $\tilde{\mathbf{U}}_r^{m+1}$. Then, by Eq. (136), $\tilde{\mathbf{V}}_r^m$ can be expressed as

$$\begin{aligned} \tilde{\mathbf{V}}_r^m = & \left\{ (\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^m}) \times \frac{\partial vec\mathbf{DF}^m(\mathbf{N}^m)}{\partial_r \mathbf{p}^T} \right\} \bullet \{ (\mathbf{W}^{m+1})^T \tilde{\mathbf{U}}_r^{m+1} \} \\ & + \mathbf{DF}^m(\mathbf{N}^m) \bullet \{ (\mathbf{W}^{m+1})^T \tilde{\mathbf{V}}_r^{m+1} \} \quad . \end{aligned} \tag{175}$$

Next, we need to initialize at $m = M$. From Eq. (141), the term $\tilde{v}_{r_{i,q}}^M$ can be computed as follows:

$$\tilde{v}^{M}_{r_{i,q}} = \frac{\partial \dot{f}^{M}(n^{M}_{i,q})}{\partial p_{r,q}} \times \frac{\partial e_{i,q}}{\partial p_{r,q}} \quad . \tag{176}$$

Thus, $\tilde{\mathbf{V}}^{M}_{r}$ can be expressed as

$$\tilde{\mathbf{V}}^{M}_{r} = (\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \left(\frac{\partial vec\mathbf{DF}^{M}(\mathbf{N}^{M})}{\partial_r \mathbf{p}^{T}} \bullet \frac{\partial vec\mathbf{E}}{\partial_r \mathbf{p}^{T}} \right). \tag{177}$$

It now remains to calculate $\partial vec\mathbf{DF}^{M}(\mathbf{N}^{M})/\partial_r \mathbf{p}^{T}$. Consider one element of this matrix, which is $\partial \dot{f}^{m}(n^{m}_{i,q})/\partial p_{r,q}$. From Eq. (144) and Eq. (147), we obtain

$$\frac{\partial \mathbf{df}^{m}(\mathbf{n}^{m}_{q})}{\partial p_{r,q}} = \frac{\partial \mathbf{df}^{m}(\mathbf{n}^{m}_{q})}{\partial (\mathbf{a}^{m}_{q})^{T}} \times \frac{\partial \mathbf{a}^{m}_{q}}{\partial p_{r,q}} \quad . \tag{178}$$

Then, by Eq. (137) and Eq. (149), $\partial vec\mathbf{DF}^{M}(\mathbf{N}^{M})/\partial_r \mathbf{p}^{T}$ can be decomposed to

$$\frac{\partial vec\mathbf{DF}^{m}(\mathbf{N}^{m})}{\partial_r \mathbf{p}^{T}} = \frac{\partial vec\mathbf{DF}^{m}(\mathbf{N}^{m})}{\partial (vec\mathbf{A}^{m})^{T}} \times \frac{\partial vec\mathbf{A}^{m}}{\partial_r \mathbf{p}^{T}} \quad . \tag{179}$$

We have derived an algorithm to compute the gradient of the new performance index J_d with respect to the network parameters $\mathbf{W}^{m}$ and $\mathbf{b}^{m}$ such that the calculation process requires less machine memory than the batch mode operation. This algorithm is summarized below.

STEPS TO COMPUTE $\partial J / \partial \mathbf{W}^m$ AND $\partial J / \partial \mathbf{b}^m$ (MEMORY SAVED)

1. Given Q inputs in $\mathbf{P}$, obtain $\mathbf{A}^m$. Initialize $r = 1$.

2. Obtain $\partial vec\mathbf{A}^m / \partial_r \mathbf{p}^T$ by Eq. (165) and Eq. (167).

3. With $\partial vec\mathbf{G} / \partial_r \mathbf{p}^T$, compute $\partial vec\mathbf{E} / \partial_r \mathbf{p}^T$ using Eq. (157).

4. Calculate $\tilde{\mathbf{U}}_r^M$ and $\tilde{\mathbf{V}}_r^M$, using Eq. (172) and Eq. (177), respectively.

5. Backpropagate to obtain $\tilde{\mathbf{U}}_r^m$ and $\tilde{\mathbf{V}}_r^m$, using Eq. (170) and Eq. (175), respectively.

6. Compute $\partial J_d / \partial \mathbf{W}^m$ and $\partial J_d / \partial \mathbf{b}^m$ by Eq. (160) and Eq. (163), respectively.

7. Set $r = r + 1$, and repeat step $2 - 7$ until $r > R$.

8. Compute $\partial J / \partial \mathbf{W}^m$ and $\partial J / \partial \mathbf{b}^m$. Update the weights and biases using any gradient-based optimization technique.

In this section, we have shown two approaches for computing the terms $\partial J_d / \partial \mathbf{W}^m$ and $\partial J_d / \partial \mathbf{b}^m$. The first approach is performed in batch mode, thus requiring sufficient machine memory to simultaneously hold all numeric elements. This approach takes advantages of faster computation for some programming languages, e.g. MATLAB. The second approach, however, breaks down the matrices in the batch mode operation so that their sizes are smaller. This approach is useful when the machine's memory is insufficient to perform in batch mode, thereby compromising between the computation speed and the required memory.

Speed Test

In this section, we create a network structure, i.e. an $R - 25 - 1$ network (R will be varied), to test the speed of the $CFDA$ algorithm. We will compare the computation time between the two approaches we previously derived, i.e. batch mode and memory-save approach.

The following figure shows the relative one-iteration execution times (averaged over 10 runs) for the batch and memory-save methods for computing the gradient of J_d, when compared to the time to compute the gradient of J_f. Note that we assume the number of training points is fixed at $Q = 75,000$. These tests were run on a computer with processor speed of 2.0GHz, and the memory size of 512MB.

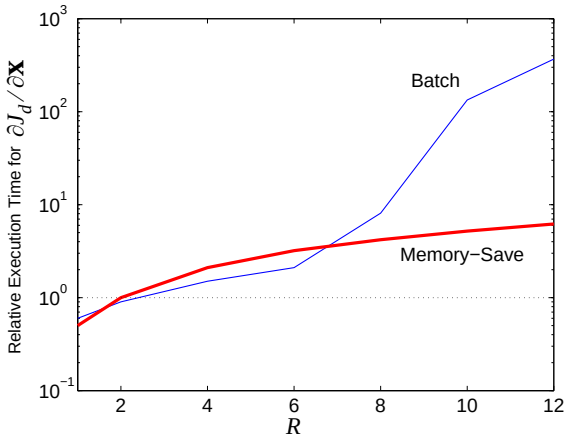

Figure 6) Relative execution time (compared to $\partial J_f/\partial \mathbf{x}$) for computing $\partial J_d/\partial \mathbf{x}$

We expected that time for computing the gradient of J_d would be more than that of J_f. However, from Figure 6), we can see that, initially, the gradient calculation for J_d took less time than that for J_f. This is because of the overhead in MATLAB codes. Once the input dimension increased, the time for computing the gradient of J_d was now more than that for J_f. The interesting part, however, occurred when the input dimension was more than seven. For these cases, the memory requirement caused the existing PC's RAM to overflow, which required data to be sent to disk. Thus, the time for computing the gradient in batch mode was more than that for the memory-save approach. These results show that the memory-save approach is useful when the data storage requirements exhaust existing RAM.

Summary

In this chapter, a new training algorithm for approximating a function and its first-order derivatives, called gradient-based $CFDA$, was proposed. We proposed the new per-

formance index, which includes not only the squared function errors but also the squared derivative errors. Two approaches for gradient calculation to minimize the performance index were presented. The first method arranges every numeric element into matrices for batch mode operation, in order to expedite the gradient calculation time in some programming languages. This approach, however, requires sufficient memory to simultaneously hold all elements. The other approach, called the memory-save method, compromises between the computation time and the required memory. The computation time under different conditions were measured. The results showed that the computation time for the squared derivative error term defined in the $CFDA$ method was longer than the standard backpropagation. This is expected as the gradient computation in the $CFDA$ method is more complicated than that in the standard backpropagation algorithm. The results also illustrated that the memory-save approach is useful in cases where computer RAM is insufficient to perform the gradient calculation in batch mode.

CHAPTER 5

COMBINED FUNCTION AND DERIVATIVE APPROXIMATION WITH LEVENBERG-MARQUARDT

Introduction

In this chapter, we present a Levenberg-Marquardt algorithm for minimizing the $CFDA$ performance index. Recall from Chapter 2 that optimization with the Levenberg-Marquardt algorithm requires the calculation of the Jacobian matrix. This will be the core work in this chapter.

We will begin this chapter with a discussion of the Levenberg-Marquardt framework for $CFDA$. We will then present two approaches for computing the Jacobian matrix. The first approach performs the calculation in batch mode. The second approach (i.e. the memory save method) compromises between the execution time and the required memory. The measured execution time under different conditions will be illustrated at the end of the chapter.

CFDA with Levenberg-Marquardt

Consider a performance index $F(\mathbf{x})$ in the form:

$$F(\mathbf{x}) = \rho_1 F_1(\mathbf{x}) + \rho_2 F_2(\mathbf{x}), \tag{180}$$

where

$$F_1(\mathbf{x}) = \sum_{i=1}^{N_1} z_i^2(\mathbf{x}) = \mathbf{z}^T(\mathbf{x}) \times \mathbf{z}(\mathbf{x}) \text{ and } F_2(\mathbf{x}) = \sum_{i=1}^{N_2} \tilde{z}_i^2(\mathbf{x}) = \tilde{\mathbf{z}}^T(\mathbf{x}) \times \tilde{\mathbf{z}}(\mathbf{x}). \quad (181)$$

The j^{th} element of the gradient is

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = \rho_1 \frac{\partial F_1(\mathbf{x})}{\partial x_j} + \rho_2 \frac{\partial F_2(\mathbf{x})}{\partial x_j}. \quad (182)$$

The gradient can be written in matrix form:

$$\nabla F(\mathbf{x}) = \rho_1 \nabla F_1(\mathbf{x}) + \rho_2 \nabla F_2(\mathbf{x}), \quad (183)$$

and, from Eq. (38) in Chapter 2, this becomes

$$\nabla F(\mathbf{x}) = 2\{\rho_1 \mathbf{J}^T(\mathbf{x})\mathbf{z}(\mathbf{x}) + \rho_2 \tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{z}}(\mathbf{x})\}, \quad (184)$$

where the Jacobian matrices are

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{z}(\mathbf{x})}{\partial \mathbf{x}^T} \text{ and } \tilde{\mathbf{J}}(\mathbf{x}) = \frac{\partial \tilde{\mathbf{z}}(\mathbf{x})}{\partial \mathbf{x}^T}. \quad (185)$$

Next, we want to find the Hessian matrix. The k, j element of the Hessian matrix would be

$$[\nabla^2 F(\mathbf{x})]_{k,j} = \frac{\partial^2 F(\mathbf{x})}{\partial x_k \partial x_j} = \rho_1 \frac{\partial^2 F_1(\mathbf{x})}{\partial x_k \partial x_j} + \rho_2 \frac{\partial^2 F_2(\mathbf{x})}{\partial x_k \partial x_j}. \quad (186)$$

From Eq. (39) in Chapter 2, this turns out to be

$$\begin{aligned}[\nabla^2 F(\mathbf{x})]_{k,j} = {} & 2\rho_1 \sum_{i=1}^{N_1} \left\{ \frac{\partial z_i(\mathbf{x})}{\partial x_k} \frac{\partial z_i(\mathbf{x})}{\partial x_j} + z_i(\mathbf{x}) \frac{\partial^2 z_i(\mathbf{x})}{\partial x_k \partial x_j} \right\} + \\ & 2\rho_2 \sum_{i=1}^{N_1} \left\{ \frac{\partial \tilde{z}_i(\mathbf{x})}{\partial x_k} \frac{\partial \tilde{z}_i(\mathbf{x})}{\partial x_j} + z_i(\mathbf{x}) \frac{\partial^2 \tilde{z}_i(\mathbf{x})}{\partial x_k \partial x_j} \right\}. \end{aligned} \quad (187)$$

Using Eq. (40), the Hessian can then be expressed in matrix form

$$\nabla^2 F(\mathbf{x}) = 2\rho_1 \left\{ \mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + \mathbf{S}(\mathbf{x}) \right\} + 2\rho_2 \{ \tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x}) + \tilde{\mathbf{S}}(\mathbf{x}) \} \quad , \tag{188}$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N_1} z_i(\mathbf{x})\nabla^2 z_i(\mathbf{x}) \text{ and } \tilde{\mathbf{S}}(\mathbf{x}) = \sum_{i=1}^{N_2} \tilde{z}_i(\mathbf{x})\nabla^2 \tilde{z}_i(\mathbf{x}) \quad . \tag{189}$$

If we assume $\mathbf{S}(\mathbf{x})$ is small. In this scenario, if we assume that both $\mathbf{S}(\mathbf{x})$ and $\tilde{\mathbf{S}}(\mathbf{x})$ are small, relative to the terms $\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$ and $\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x})$, then the Hessian matrix can be approximated as

$$\begin{aligned} \nabla^2 F(\mathbf{x}) &\cong 2\rho_1 \mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\rho_2 \tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x}) \\ &= 2\{ \rho_1 \mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + \rho_2 \tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x}) \} \quad . \end{aligned} \tag{190}$$

Therefore, by Eq. (184) and Eq. (190), the Levenberg-Marquardt update is

$$\begin{aligned} \Delta\mathbf{x}_{k+1} &= \mathbf{x}_{k+1} - \mathbf{x}_k \\ &= -[\rho_1 \mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \rho_2 \tilde{\mathbf{J}}^T(\mathbf{x}_k)\tilde{\mathbf{J}}(\mathbf{x}_k) + \mu_k \mathbf{I}_n]^{-1} \times [\rho_1 \mathbf{J}^T(\mathbf{x}_k)\mathbf{z}(\mathbf{x}_k) + \rho_2 \tilde{\mathbf{J}}^T(\mathbf{x}_k)\tilde{\mathbf{z}}(\mathbf{x}_k)]. \end{aligned} \tag{191}$$

Note that μ_k is adjusted using the typical Levenberg-Marquardt algorithm, which was presented in Chapter 2.

Now, recall from Eq. (80) that the performance index in the *CFDA* method consists of two terms (i.e. J_f and J_d). This is in the same form as the performance index in Eq. (180), if we assign

$$F_1(\mathbf{x}) \equiv \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \{a_{k,q} - g_{k,q}\}^2, \tag{192}$$

$$F_2(\mathbf{x}) \equiv \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \sum_{r=1}^{R} \varphi_{k,r}(q) \left(\frac{\partial a_{k,q}}{\partial p_{r,q}} - \frac{\partial g_{k,q}}{\partial p_{r,q}} \right)^2, \tag{193}$$

where $\rho_1 \equiv 1/(QS^M)$ and $\rho_2 \equiv \rho/(Q_d S_d^M R_d)$.

Since the algorithm for computing $\mathbf{J}(\mathbf{x})$ and $\mathbf{z}(\mathbf{x})$ is already known (see [HaMe94] and [HaDe96]), we will focus on the calculation of $\tilde{\mathbf{z}}(\mathbf{x})$ and $\tilde{\mathbf{J}}(\mathbf{x})$. The calculations for these terms are given in the following two sections. The next section will show the batch calculation and the following ssection will show the memory-save calculation.

Batch calculation

In this section, we will compute the vector $\tilde{\mathbf{z}}(\mathbf{x})$ and the Jacobian matrix $\tilde{\mathbf{J}}(\mathbf{x})$. Recall from Eq. (193) that we have

$$F_2(\mathbf{x}) = \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \sum_{r=1}^{R} \varphi_{k,r}(q) \left(\frac{\partial a_{k,q}}{\partial p_{r,q}} - \frac{\partial g_{k,q}}{\partial p_{r,q}} \right)^2. \tag{194}$$

We can rewrite Eq. (194) as

$$F_2(\mathbf{x}) = \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \sum_{r=1}^{R} \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \right)^2 = \sum_{q=1}^{Q} \left(vec \frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T} \right)^T \times vec \frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}. \tag{195}$$

Define the following matrix

$$\mathbf{DE_P} \equiv (\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{P})^T} = \left[\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T} \frac{\partial \mathbf{e}_2}{\partial \mathbf{p}_2^T} \cdots \frac{\partial \mathbf{e}_Q}{\partial \mathbf{p}_Q^T} \right], \tag{196}$$

where the matrix $\partial vec\mathbf{E} / \partial (vec\mathbf{P})^T$ is defined in Eq. (99). Then, Eq. (195) can be expressed as

$$J_d = (vec\mathbf{DE_P})^T \times vec\mathbf{DE_P}, \tag{197}$$

where

$$vec\mathbf{DE_P} = \begin{bmatrix} vec\dfrac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T} \\ vec\dfrac{\partial \mathbf{e}_2}{\partial \mathbf{p}_2^T} \\ \cdots \\ vec\dfrac{\partial \mathbf{e}_Q}{\partial \mathbf{p}_Q^T} \end{bmatrix}. \tag{198}$$

Now, by equating Eq. (197) to Eq. (181), we can see that

$$\tilde{\mathbf{z}}(\mathbf{x}) = vec\mathbf{DE_P}. \tag{199}$$

For the calculation for the Jacobian matrix $\tilde{\mathbf{J}}(\mathbf{x})$, we need to have the vector $\mathbf{x}$ containing all of the network parameters. There are several ways to define $\mathbf{x}$. However, the definition we use is

$$\mathbf{x}^T \equiv \left[(\mathbf{x}^1)^T (\mathbf{x}^2)^T \cdots (\mathbf{x}^M)^T \right], \tag{200}$$

where the value of n is shown in Eq. (31). The vector $\mathbf{x}^m$ is defined as

$$\mathbf{x}^m \equiv \begin{bmatrix} vec\mathbf{W}^m \\ \mathbf{b}^m \end{bmatrix}. \tag{201}$$

Therefore, from Eq. (185), the Jacobian matrix $\tilde{\mathbf{J}}(\mathbf{x})$ can be written as

$$\tilde{\mathbf{J}}(\mathbf{x}) = \frac{\partial vec\mathbf{DE_P}}{\partial \mathbf{x}^T} = \begin{bmatrix} \frac{\partial}{\partial \mathbf{x}^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T} \right) \\ \frac{\partial}{\partial \mathbf{x}^T}\left(vec\frac{\partial \mathbf{e}_2}{\partial \mathbf{p}_2^T} \right) \\ \cdots \\ \frac{\partial}{\partial \mathbf{x}^T}\left(vec\frac{\partial \mathbf{e}_Q}{\partial \mathbf{p}_Q^T} \right) \end{bmatrix}. \tag{202}$$

Using Eq. (200) and Eq. (201), $\partial vec\mathbf{DE_P}/\partial \mathbf{x}^T$ can be written

$$\tilde{\mathbf{J}}(\mathbf{x}) = \frac{\partial vec\mathbf{DE_P}}{\partial \mathbf{x}^T} = \begin{bmatrix} \frac{\partial vec\mathbf{DE_P}}{\partial (\mathbf{x}^1)^T} & \frac{\partial vec\mathbf{DE_P}}{\partial (\mathbf{x}^2)^T} & \cdots & \frac{\partial vec\mathbf{DE_P}}{\partial (\mathbf{x}^M)^T} \end{bmatrix}, \tag{203}$$

where

$$\frac{\partial vec\mathbf{DE_P}}{\partial (\mathbf{x}^m)^T} = \begin{bmatrix} \frac{\partial vec\mathbf{DE_P}}{\partial (vec\mathbf{W}^m)^T} & \frac{\partial vec\mathbf{DE_P}}{\partial (\mathbf{b}^m)^T} \end{bmatrix}. \tag{204}$$

Therefore, by Eq. (202) through Eq. (204), $\partial vec\mathbf{DE_P}/\partial \mathbf{x}^T$ can be written

$$\tilde{\mathbf{J}}(\mathbf{x}) = \frac{\partial vec\mathbf{DE_P}}{\partial \mathbf{x}^T} = \begin{bmatrix} \frac{\partial}{\partial(\mathbf{x}^1)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) & \frac{\partial}{\partial(\mathbf{x}^2)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) & \cdots & \frac{\partial}{\partial(\mathbf{x}^M)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) \\ \frac{\partial}{\partial(\mathbf{x}^1)^T}\left(vec\frac{\partial \mathbf{e}_2}{\partial \mathbf{p}_2^T}\right) & \frac{\partial}{\partial(\mathbf{x}^2)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) & \cdots & \frac{\partial}{\partial(\mathbf{x}^M)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) \\ \cdots & \cdots & & \cdots \\ \frac{\partial}{\partial(\mathbf{x}^1)^T}\left(vec\frac{\partial \mathbf{e}_Q}{\partial \mathbf{p}_Q^T}\right) & \frac{\partial}{\partial(\mathbf{x}^2)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) & \cdots & \frac{\partial}{\partial(\mathbf{x}^M)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) \end{bmatrix}, \quad (205)$$

where

$$\frac{\partial}{\partial(\mathbf{x}^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right) = \left[\frac{\partial}{\partial(vec\mathbf{W}^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right) \quad \frac{\partial}{\partial(\mathbf{b}^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right)\right]. \quad (206)$$

To obtain $\tilde{\mathbf{J}}(\mathbf{x})$, we need to compute $\partial vec\mathbf{DE_P}/\partial(\mathbf{x}^m)^T$. However, it is fairly complicated to derive the entire $\partial vec\mathbf{DE_P}/\partial(\mathbf{x}^m)^T$ at once. Therefore, $\frac{\partial}{\partial(\mathbf{x}^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right)$ will be considered first. Then, the batch matrix $\partial vec\mathbf{DE_P}/\partial(\mathbf{x}^m)^T$ will be expressed.

Each element in the matrix $\frac{\partial}{\partial(\mathbf{x}^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right)$ is $\frac{\partial}{\partial x_l^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right)$, where x_l^m is at element l of $\mathbf{x}^m$. We will first consider the case when x is an element in the weight matrix $\mathbf{W}^m$ and then the case when x_l^m is an element of the bias vector $\mathbf{b}^m$.

Consider when $x_l^m = w_{i,j}^m$. From Eq. (87) in Chapter 4 and using the fact that

$$\frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial n_{i,q}^m}\right) = \frac{\partial}{\partial n_{i,q}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right), \tag{207}$$

we have

$$\frac{\partial}{\partial w_{i,j}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \frac{\partial}{\partial n_{i,q}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) \times a_{j,q}^{m-1} + \frac{\partial e_{k,q}}{\partial n_{i,q}^m} \times \frac{\partial a_{j,q}^{m-1}}{\partial p_{r,q}} \quad . \tag{208}$$

Next, consider when x_l^m is an element of $\mathbf{b}^m$, i.e. $x_l^m = b_i^m$. From Eq. (101), Eq. (104) and Eq. (207), we have

$$\frac{\partial}{\partial b_i^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \frac{\partial}{\partial n_{i,q}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right). \tag{209}$$

Eq. (208) and Eq. (209) can be written in matrix form:

$$\begin{aligned}\frac{\partial}{\partial (vec\mathbf{W}^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right) = &\left\{\mathbf{1}_{1\times S^{m-1}} \otimes \frac{\partial}{\partial (\mathbf{n}_q^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right)\right\} \bullet \left\{(\mathbf{a}_q^{m-1})^T \otimes \mathbf{1}_{S^M R\times S^m}\right\} \\ &+ \left\{\mathbf{1}_{R\times S^{m-1}} \otimes \frac{\partial \mathbf{e}_q}{\partial (\mathbf{n}_q^m)^T}\right\} \bullet \left\{\left(\frac{\partial \mathbf{a}_q^{m-1}}{\partial \mathbf{p}_q^T}\right)^T \otimes \mathbf{1}_{S^M\times S^m}\right\},\end{aligned} \tag{210}$$

and

$$\frac{\partial}{\partial (\mathbf{b}^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right) = \frac{\partial}{\partial (\mathbf{n}_q^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right). \tag{211}$$

In batch mode for Eq. (210), the batch matrix $\partial vec\mathbf{DE_p}/\partial (vec\mathbf{W}^m)^T$ and can be expressed as

$$\frac{\partial vec\mathbf{DE_P}}{\partial (vec\mathbf{W}^m)^T} = \left(\left\{ \mathbf{1}_{1 \times S^{m-1}} \otimes \left(\frac{\partial vec\mathbf{DE_P}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^m}) \right) \right\} \bullet \left\{ (\mathbf{A}^{m-1})^T \otimes \mathbf{1}_{S^M R \times S^m} \right\} \right) + \left(\left\{ \mathbf{1}_{R \times S^{m-1}} \otimes \left(\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^m}) \right) \right\} \bullet \left\{ \left((\mathbf{1}_{1 \times Q} \otimes \mathbf{I}_{S^{m-1}}) \times \frac{\partial vec\mathbf{A}^{m-1}}{\partial (vec\mathbf{P})^T} \right)^T \otimes \mathbf{1}_{S^M \times S^m} \right\} \right) . \tag{212}$$

The batch matrix for Eq. (211) can be written as

$$\frac{\partial vec\mathbf{DE_P}}{\partial (\mathbf{b}^m)^T} = \frac{\partial vec\mathbf{DE_P}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^m}) . \tag{213}$$

To clarify the expressions in Eq. (212) and Eq. (213), first note, from Eq. (196), that

$$\frac{\partial vec\mathbf{DE_P}}{\partial (vec\mathbf{N}^m)^T} = \begin{bmatrix} \frac{\partial}{\partial (\mathbf{n}_1^m)^T} \left(vec \frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T} \right) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial}{\partial (\mathbf{n}_2^m)^T} \left(vec \frac{\partial \mathbf{e}_2}{\partial \mathbf{p}_2^T} \right) & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial}{\partial (\mathbf{n}_Q^m)^T} \left(vec \frac{\partial \mathbf{e}_Q}{\partial \mathbf{p}_Q^T} \right) \end{bmatrix}, \tag{214}$$

and

$$\frac{\partial vec\mathbf{DE_P}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q\times 1} \otimes \mathbf{I}_{S^m}) = \begin{bmatrix} \frac{\partial}{\partial (\mathbf{n}_1^m)^T}\left(vec\frac{\partial \mathbf{e}_1}{\partial \mathbf{p}_1^T}\right) \\ \frac{\partial}{\partial (\mathbf{n}_2^m)^T}\left(vec\frac{\partial \mathbf{e}_2}{\partial \mathbf{p}_2^T}\right) \\ \cdots \\ \frac{\partial}{\partial (\mathbf{n}_Q^m)^T}\left(vec\frac{\partial \mathbf{e}_Q}{\partial \mathbf{p}_Q^T}\right) \end{bmatrix}. \quad (215)$$

In addition, we have

$$\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^m)^T} = \begin{bmatrix} \frac{\partial \mathbf{e}_1}{\partial (\mathbf{n}_1^m)^T} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{e}_2}{\partial (\mathbf{n}_2^m)^T} & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial \mathbf{e}_Q}{\partial (\mathbf{n}_Q^m)^T} \end{bmatrix}, \text{ and} \quad (216)$$

$$\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q\times 1} \otimes \mathbf{I}_{S^m}) = \begin{bmatrix} \frac{\partial \mathbf{e}_1}{\partial (\mathbf{n}_1^m)^T} \\ \frac{\partial \mathbf{e}_2}{\partial (\mathbf{n}_2^m)^T} \\ \cdots \\ \frac{\partial \mathbf{e}_Q}{\partial (\mathbf{n}_Q^m)^T} \end{bmatrix}. \quad (217)$$

We can also write

$$\left((\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^{m-1}}) \times \frac{\partial vec\mathbf{A}^{m-1}}{\partial (vec\mathbf{P})^T} \right)^T = \left(\frac{\partial vec\mathbf{A}^{m-1}}{\partial (vec\mathbf{P})^T} \right)^T \times (\mathbf{1}_{Q\times 1} \otimes \mathbf{I}_{S^{m-1}}), \qquad (218)$$

and the result is shown in Eq. (98).

From Eq. (212) and Eq. (213), we will need to calculate $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$, and $\partial vec\mathbf{DE_P} / \partial (vec\mathbf{N}^m)^T$. (We previously provided the equations for the other terms.) We will first show the derivation for computing the matrix $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$, followed by $\partial vec\mathbf{DE_P} / \partial (vec\mathbf{N}^m)^T$. Note that the computation for the term $\mathbf{A}^m$ is in Chapter 2, Eq. (18), and for the calculation for the term $\partial vec\mathbf{A}^m / \partial (vec\mathbf{P})^T$ is in Chapter 4, Eq. (114).

I. Calculation of $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$

Consider an element of $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$, which is $\partial e_{k,q} / \partial n_{i,q}^m$. This element is known as the *Marquardt sensitivity*, [HaDe96]. This term was computed in Eq. (118) and Eq. (119) in Chapter 4. Hence, $\partial \mathbf{e}_q / \partial (\mathbf{n}_q^m)^T$ can be expressed as

$$\frac{\partial \mathbf{e}_q}{\partial (\mathbf{n}_q^m)^T} = \frac{\partial \mathbf{e}_q}{\partial (\mathbf{n}_q^{m+1})^T} \mathbf{W}^{m+1} \dot{\mathbf{F}}^m (\mathbf{n}_q^m), \qquad (219)$$

where $\dot{\mathbf{F}}^m (\mathbf{n}_q^m)$ is defined in Eq. (112).

Thus, from Eq. (216) and Eq. (219), the batch matrix $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$ can be expressed as

$$\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^m)^T} = \frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^{m+1})^T} \times \{ \mathbf{I}_Q \otimes \mathbf{W}^{m+1} \} \times \frac{\partial vec\mathbf{A}^m}{\partial (vec\mathbf{N}^m)^T}, \qquad (220)$$

where the term $\partial vec\mathbf{A}^m / \partial (vec\mathbf{N}^m)^T$ is computed as shown in Eq. (113).

Since Eq. (220) is a backpropagation process, we need to initialize it at the output layer, i.e. $m = M$. From Eq. (125), we can write

$$\frac{\partial \mathbf{e}_q}{\partial (\mathbf{n}_q^M)^T} = \frac{\partial \mathbf{a}_q}{\partial (\mathbf{n}_q^M)^T} = \dot{\mathbf{F}}^M(\mathbf{n}_q^M) \quad . \tag{221}$$

We can also write Eq. (221) in the batch mode:

$$\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^M)^T} = \frac{\partial vec\mathbf{A}}{\partial (vec\mathbf{N}^M)^T} \quad . \tag{222}$$

This completes the computation of $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$. Next, the calculation of $\partial vec\mathbf{DE_P} / \partial (vec\mathbf{N}^m)^T$ will be derived.

II. Calculation of $\partial vec\mathbf{DE_P} / \partial (vec\mathbf{N}^m)^T$

Recall that an element of $\partial vec\mathbf{DE_P} / \partial (vec\mathbf{N}^m)^T$ is $\frac{\partial}{\partial n_{i,q}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right)$. From Eq. (207) along with Eq. (119), we have

$$\frac{\partial}{\partial n_{i,q}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \frac{\partial}{\partial p_{r,q}}\left(\dot{f}^m(n_{i,q}^m)\sum_l \left\{ w_{l,i}^{m+1}\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}}\right\}\right). \tag{223}$$

By taking the derivative inside the parenthesis and using Eq. (207), this turns out to be

$$\frac{\partial}{\partial n_{i,q}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \frac{\partial \dot{f}^m(n_{i,q}^m)}{\partial p_{r,q}}\sum_l \left\{ w_{l,i}^{m+1}\frac{\partial e_{k,q}}{\partial n_{l,q}^{m+1}}\right\} + \dot{f}^m(n_{i,q}^m)\sum_l \left\{ w_{l,i}^{m+1}\frac{\partial}{\partial n_{l,q}^{m+1}}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right)\right\}. \tag{224}$$

Therefore, using Eq. (133) and Eq. (134), Eq. (224) can be written in matrix form:

$$\frac{\partial}{\partial(\mathbf{n}_q^m)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right) = \left\{\left(\frac{\partial \mathbf{df}^m(\mathbf{n}_q^m)}{\partial \mathbf{p}_q^T}\right)^T \otimes \mathbf{1}_{S^M \times 1}\right\} \bullet \left\{\mathbf{1}_{R\times 1} \otimes \left(\frac{\partial \mathbf{e}_q}{\partial(\mathbf{n}_q^{m+1})^T}\mathbf{W}^{m+1}\right)\right\}$$
$$+ \left\{(\mathbf{df}^m(\mathbf{n}_q^m))^T \otimes \mathbf{1}_{S^M R \times 1}\right\} \bullet \left\{\frac{\partial}{\partial(\mathbf{n}_q^{m+1})^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right)\mathbf{W}^{m+1}\right\}, \tag{225}$$

where $\partial \mathbf{df}^m(\mathbf{n}_q^m)/\partial \mathbf{p}_q^T$ can be computed using Eq. (147) and Eq. (148). From Eq. (214) and Eq. (225), the batch matrix $\partial vec\mathbf{DE_P}/\partial(vec\mathbf{N}^m)^T$ can be expressed as

$$\frac{\partial vec\mathbf{DE_P}}{\partial(vec\mathbf{N}^m)^T} = \left\{\left(\frac{\partial vec\mathbf{DF}^m(\mathbf{N}^m)}{\partial(vec\mathbf{P})^T}\right)^T \otimes \mathbf{1}_{S^M \times 1}\right\}$$
$$\bullet \left\{(\mathbf{I}_Q \otimes \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M}) \times \left(\frac{\partial vec\mathbf{E}}{\partial(vec\mathbf{N}^{m+1})^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^{m+1})\right)\right\} +$$
$$\left\{\left(\left\{\mathbf{1}_{1\times Q} \otimes (\mathbf{DF}^m(\mathbf{N}^m))^T\right\} \bullet \{\mathbf{I}_Q \otimes \mathbf{1}_{1\times S^m}\}\right) \otimes \mathbf{1}_{S^M R \times 1}\right\} \tag{226}$$
$$\bullet \left\{\frac{\partial vec\mathbf{DE_P}}{\partial(vec\mathbf{N}^{m+1})^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^{m+1})\right\},$$

where $\mathbf{DF}^m(\mathbf{N}^m)$ is defined in Eq. (135) and $\partial vec\mathbf{DF}^m(\mathbf{N}^m)/\partial(vec\mathbf{P})^T$ can be computed using Eq. (149). To clarify Eq. (226), note that

$$\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^{m+1})^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^{m+1})$$

$$= \begin{bmatrix} \frac{\partial \mathbf{e}_1}{\partial (\mathbf{n}_1^{m+1})^T} \times \mathbf{W}^{m+1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{e}_2}{\partial (\mathbf{n}_2^{m+1})^T} \times \mathbf{W}^{m+1} & \dots & \mathbf{0} \\ \dots & \dots & & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \frac{\partial \mathbf{e}_Q}{\partial (\mathbf{n}_Q^{m+1})^T} \times \mathbf{W}^{m+1} \end{bmatrix}, \quad (227)$$

$$\mathbf{I}_Q \otimes \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M} = \begin{bmatrix} \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M} & \dots & \mathbf{0} \\ \dots & \dots & & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M} \end{bmatrix}. \quad (228)$$

The product of the two matrices in Eq. (227) and Eq. (228) results in the following matrix:

$$(\mathbf{I}_Q \otimes \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M}) \times \left(\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^{m+1})^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^{m+1}) \right)$$

$$= \begin{bmatrix} \mathbf{1}_{R\times 1} \otimes \left(\frac{\partial \mathbf{e}_1}{\partial (\mathbf{n}_1^{m+1})^T} \times \mathbf{W}^{m+1} \right) & \dots & \mathbf{0} \\ \dots & & \dots \\ \mathbf{0} & \dots & \mathbf{1}_{R\times 1} \otimes \left(\frac{\partial \mathbf{e}_Q}{\partial (\mathbf{n}_Q^{m+1})^T} \times \mathbf{W}^{m+1} \right) \end{bmatrix}. \quad (229)$$

In addition, note further, in Eq. (226), that

$$\mathbf{1}_{1\times Q} \otimes (\mathbf{DF}^m(\mathbf{N}^m))^T = \left[(\mathbf{DF}^m(\mathbf{N}^m))^T \; (\mathbf{DF}^m(\mathbf{N}^m))^T \; \dots \; (\mathbf{DF}^m(\mathbf{N}^m))^T \right], \quad (230)$$

and

$$\mathbf{I}_Q \otimes \mathbf{1}_{1 \times S^m} = \begin{bmatrix} \mathbf{1}_{1 \times S^m} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{1}_{1 \times S^m} & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{1}_{1 \times S^m} \end{bmatrix}. \tag{231}$$

Thus, the Hadamard product of the two expressions in Eq. (230) and Eq. (231) results in

$$\begin{aligned} &\left\{ \mathbf{1}_{1 \times Q} \otimes (\mathbf{DF}^m(\mathbf{N}^m))^T \right\} \bullet \{ \mathbf{I}_Q \otimes \mathbf{1}_{1 \times S^m} \} \\ &= \begin{bmatrix} (\mathbf{df}^m(\mathbf{n}_1^m))^T & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & (\mathbf{df}^m(\mathbf{n}_2^m))^T & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & (\mathbf{df}^m(\mathbf{n}_Q^m))^T \end{bmatrix}. \end{aligned} \tag{232}$$

Since Eq. (226) is a backpropagation process, the computation at the output layer where $m = M$ needs to be evaluated. From Eq. (223), we have

$$\frac{\partial}{\partial n_{i,q}^M}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial e_{k,q}}{\partial n_{i,q}^M}\right) = \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial a_{k,q}}{\partial n_{i,q}^M}\right). \tag{233}$$

Using Eq. (125), it becomes

$$\frac{\partial}{\partial n_{i,q}^M}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right) = \begin{cases} \dfrac{\partial \dot{f}^M(n_{i,q}^M)}{\partial p_{r,q}} & ; \text{if } i = k. \\ 0 & ; \text{if } i \neq k. \end{cases} \tag{234}$$

Eq. (234) can be expressed in matrix form:

$$\frac{\partial}{\partial(\mathbf{n}_q^M)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right) = \begin{bmatrix} \frac{\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)}{\partial p_{1,q}} \\ \frac{\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)}{\partial p_{2,q}} \\ \cdots \\ \frac{\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)}{\partial p_{R,q}} \end{bmatrix}, \tag{235}$$

where, by Eq. (112), the matrix $\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)/\partial p_{r,q}$ is

$$\frac{\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)}{\partial p_{r,q}} = \begin{bmatrix} \frac{\partial \dot{f}^M(n_{1,q}^M)}{\partial p_{r,q}} & 0 & \cdots & 0 \\ 0 & \frac{\partial \dot{f}^M(n_{2,q}^M)}{\partial p_{r,q}} & \cdots & 0 \\ \cdots & \cdots & & \cdots \\ 0 & 0 & \cdots & \frac{\partial \dot{f}^M\left(n_{S^M,q}^M\right)}{\partial p_{r,q}} \end{bmatrix}. \tag{236}$$

From Eq. (144), we then have

$$\frac{\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)}{\partial p_{r,q}} = \frac{\partial \mathbf{df}^M(\mathbf{n}_q^M)}{\partial \mathbf{a}_q^T} \bullet \left\{\frac{\partial \mathbf{a}_q}{\partial p_{r,q}} \otimes \mathbf{1}_{1\times S^M}\right\}. \tag{237}$$

Therefore, Eq. (235) can be written as

$$\frac{\partial}{\partial(\mathbf{n}_q^M)^T}\left(vec\frac{\partial \mathbf{e}_q}{\partial \mathbf{p}_q^T}\right) = \left\{\mathbf{1}_{R\times 1} \otimes \frac{\partial \mathbf{df}^M(\mathbf{n}_q^M)}{\partial \mathbf{a}_q^T}\right\} \bullet \left\{vec\frac{\partial \mathbf{a}_q}{\partial \mathbf{p}_q^T} \otimes \mathbf{1}_{1\times S^M}\right\}. \tag{238}$$

From Eq. (214) and Eq. (238), the batch matrix $\partial vec\mathbf{DE_P}/\partial(vec\mathbf{N}^M)^T$ can be written as

$$\frac{\partial vec\mathbf{DE_P}}{\partial (vec\mathbf{N}^M)^T} = \left\{ (\mathbf{I}_Q \otimes \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{DF}^M(\mathbf{N}^M)}{\partial (vec\mathbf{A})^T} \right\} \bullet$$
$$\left\{ \left(\left\{ \mathbf{1}_{1\times Q} \otimes vec\left((\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{A}}{\partial (vec\mathbf{P})^T} \right) \right\} \right. \tag{239}$$
$$\left. \bullet \{ \mathbf{I}_Q \otimes \mathbf{1}_{S^M R\times 1} \} \right) \otimes \mathbf{1}_{1\times S^M} \} \ .$$

To clarify Eq. (239), note that the first expression is

$$(\mathbf{I}_Q \otimes \mathbf{1}_{R\times 1} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{DF}^M(\mathbf{N}^M)}{\partial (vec\mathbf{A})^T}$$
$$= \begin{bmatrix} \mathbf{1}_{R\times 1} \otimes \frac{\partial \mathbf{df}^M(\mathbf{n}_1^M)}{\partial \mathbf{a}_1^T} & \cdots & \mathbf{0} \\ \cdots & & \cdots \\ \mathbf{0} & \cdots & \mathbf{1}_{R\times 1} \otimes \frac{\partial \mathbf{df}^M(\mathbf{n}_Q^M)}{\partial \mathbf{a}_Q^T} \end{bmatrix}. \tag{240}$$

In addition,

$$vec\left((\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{A}}{\partial (vec\mathbf{P})^T} \right) = vec \left[\frac{\partial \mathbf{a}_1}{\partial \mathbf{p}_1^T} \frac{\partial \mathbf{a}_2}{\partial \mathbf{p}_2^T} \cdots \frac{\partial \mathbf{a}_Q}{\partial \mathbf{p}_Q^T} \right] = \begin{bmatrix} vec\frac{\partial \mathbf{a}_1}{\partial \mathbf{p}_1^T} \\ vec\frac{\partial \mathbf{a}_2}{\partial \mathbf{p}_2^T} \\ \cdots \\ vec\frac{\partial \mathbf{a}_Q}{\partial \mathbf{p}_Q^T} \end{bmatrix} \tag{241}$$

Thus, the second expression in Eq. (239) yields

$$\left\{ \mathbf{1}_{1\times Q} \otimes vec\left((\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{A}}{\partial (vec\mathbf{P})^T} \right) \right\} \bullet \{ \mathbf{I}_Q \otimes \mathbf{1}_{S^M R\times 1} \}$$

$$= \begin{bmatrix} vec\frac{\partial \mathbf{a}_1}{\partial \mathbf{p}_1^T} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & vec\frac{\partial \mathbf{a}_2}{\partial \mathbf{p}_2^T} & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & vec\frac{\partial \mathbf{a}_Q}{\partial \mathbf{p}_Q^T} \end{bmatrix} . \tag{242}$$

This completes the calculation of $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$ and $\partial vec\mathbf{DE_P} / \partial (vec\mathbf{N}^m)^T$, which are needed for $\partial vec\mathbf{DE_P} / \partial (vec\mathbf{W}^m)^T$ and $\partial vec\mathbf{DE_P} / \partial (\mathbf{b}^m)^T$, i.e. Eq. (212) and Eq. (213). Then, by Eq. (204), we obtain $\partial vec\mathbf{DE_P} / \partial (\mathbf{x}^m)^T$. By concatenating the matrix $\partial vec\mathbf{DE_P} / \partial (\mathbf{x}^m)^T$ for all $m = 1, 2, \ldots, M$ as in Eq. (203), we finally have the Jacobian matrix $\tilde{\mathbf{J}}(\mathbf{x})$. Along with Eq. (199) for $\tilde{\mathbf{z}}(\mathbf{x})$, the Levenberg-Marquardt update for the *CFDA* method can be performed, using Eq. (191).

Before going to the next section, in which we will present the computation for the Jacobian matrix in the memory save approach, the summary of the batch algorithm is shown as follows.

Summary of Batch Calculation

STEPS FOR CFDA WITH LEVENBERG-MARQUARDT

(BATCH MODE)

1. Given Q inputs in $\mathbf{P}$, compute $\mathbf{A}^m$ and $\partial vec\mathbf{A}^m/\partial(vec\mathbf{P})^T$ by Eq. (114) and Eq. (117).

2. Given the derivative information in $\partial vec\mathbf{G}/\partial(vec\mathbf{P})^T$, compute $\mathbf{DE_P}$ by Eq. (196) and obtain $\tilde{\mathbf{z}}(\mathbf{x})$ using Eq. (199).

3. Compute $\partial vec\mathbf{E}/\partial(vec\mathbf{N}^M)^T$ and $\partial vec\mathbf{DE_P}/\partial(vec\mathbf{N}^M)^T$ using Eq. (222) and Eq. (239), respectively.

4. Backpropagate for $\partial vec\mathbf{E}/\partial(vec\mathbf{N}^m)^T$ and $\partial vec\mathbf{DE_P}/\partial(vec\mathbf{N}^m)^T$ by Eq. (220) and Eq. (226), respectively.

5. Compute $\partial vec\mathbf{DE_P}/\partial(vec\mathbf{W}^m)^T$ and $\partial vec\mathbf{DE_P}/\partial(\mathbf{b}^m)^T$, using Eq. (212) and Eq. (213), respectively. Then obtain $\partial vec\mathbf{DE_P}/\partial(\mathbf{x}^m)^T$ by Eq. (204).

6. Obtain the Jacobian matrix $\tilde{\mathbf{J}}(\mathbf{x})$, using Eq. (203).

7. Update the network parameters by Eq. (191).

Memory-save Calculation

In this section, we will present a procedure for computing the Jacobian matrix that uses less memory than the batch algorithm presented in the previous section.

From Eq. (180), write the performance index $F_2(\mathbf{x})$ in the form:

$$F_2(\mathbf{x}) = \sum_{r=1}^{R} F_{2_r}(\mathbf{x}) = \sum_{r=1}^{R} \tilde{\mathbf{z}}_r^T(\mathbf{x}) \times \tilde{\mathbf{z}}_r(\mathbf{x}), \tag{243}$$

where $\tilde{\mathbf{z}}_r(\mathbf{x})$ is a column vector. The gradient and the Hessian of the performance index $F_2(\mathbf{x})$, i.e. $\nabla F_2(\mathbf{x})$ and $\nabla^2 F_2(\mathbf{x})$, can then be computed as

$$\nabla F_2(\mathbf{x}) = \sum_{r=1}^{R} \nabla F_{2_r}(\mathbf{x}) \tag{244}$$

and

$$\nabla^2 F_2(\mathbf{x}) = \sum_{r=1}^{R} \nabla^2 F_{2_r}(\mathbf{x}). \tag{245}$$

From Eq. (244), by following Eq. (182) to Eq. (190), the term $\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{z}}(\mathbf{x})$ and $\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x})$ can be computed as

$$\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{z}}(\mathbf{x}) = \sum_{r=1}^{R} \tilde{\mathbf{J}}_r^T(\mathbf{x}) \times \tilde{\mathbf{z}}_r(\mathbf{x}) \text{ and } \tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x}) = \sum_{r=1}^{R} \tilde{\mathbf{J}}_r^T(\mathbf{x}) \times \tilde{\mathbf{J}}_r(\mathbf{x}), \tag{246}$$

where

$$\tilde{\mathbf{J}}_r(\mathbf{x}) = \frac{\partial \tilde{\mathbf{z}}_r(\mathbf{x})}{\partial \mathbf{x}^T}. \tag{247}$$

Therefore, Eq. (191) is the Levenberg-Marquardt update for this memory-save approach; with the term $\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{z}}(\mathbf{x})$ and $\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x})$ replaced by Eq. (246).

To compute $\tilde{\mathbf{z}}_r(\mathbf{x})$ and $\tilde{\mathbf{J}}_r(\mathbf{x})$, first consider Eq. (243). By equating it with Eq. (193), we obtain

$$F_{2_r}(\mathbf{x}) = \sum_{q=1}^{Q}\sum_{k=1}^{S^M} \varphi_{k,r}(q)\left(\frac{\partial a_{k,q}}{\partial p_{r,q}} - \frac{\partial g_{k,q}}{\partial p_{r,q}}\right)^2, \tag{248}$$

which can be also written as

$$F_{2_r}(\mathbf{x}) = \sum_{q=1}^{Q}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)^T \times \frac{\partial \mathbf{e}_q}{\partial p_{r,q}} \ . \tag{249}$$

Define the $S^M \times Q$ matrix in Eq. (156):

$$\mathbf{DE}_{r\mathbf{p}} \equiv (\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^M}) \times \frac{\partial vec\mathbf{E}}{\partial_r \mathbf{p}^T} = \left[\frac{\partial \mathbf{e}_1}{\partial p_{r,1}} \ \frac{\partial \mathbf{e}_2}{\partial p_{r,2}} \ \cdots \ \frac{\partial \mathbf{e}_Q}{\partial p_{r,Q}}\right]. \tag{250}$$

By Eq. (249) and Eq. (250), $F_2(\mathbf{x})$ in Eq. (193) can be written as

$$F_2(\mathbf{x}) = \sum_{r=1}^{R} (vec\mathbf{DE}_{r\mathbf{p}})^T \times vec\mathbf{DE}_{r\mathbf{p}} \, . \tag{251}$$

Equating Eq. (251) with Eq. (243), this implies

$$\tilde{\mathbf{z}}_r(\mathbf{x}) = vec\mathbf{DE}_{r\mathbf{p}} \, . \tag{252}$$

To compute the Jacobian matrix $\tilde{\mathbf{J}}_r^T(\mathbf{x})$, consider Eq. (247), which implies

$$\tilde{\mathbf{J}}_r(\mathbf{x}) = \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial \mathbf{x}^T} = \begin{bmatrix} \frac{\partial}{\partial \mathbf{x}^T}\left(\frac{\partial \mathbf{e}_1}{\partial p_{r,1}}\right) \\ \frac{\partial}{\partial \mathbf{x}^T}\left(\frac{\partial \mathbf{e}_2}{\partial p_{r,2}}\right) \\ \cdots \\ \frac{\partial}{\partial \mathbf{x}^T}\left(\frac{\partial \mathbf{e}_Q}{\partial p_{r,Q}}\right) \end{bmatrix}. \tag{253}$$

By Eq. (200), $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial \mathbf{x}^T$ can also be written

$$\tilde{\mathbf{J}}_r(\mathbf{x}) = \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial \mathbf{x}^T} = \begin{bmatrix} \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (\mathbf{x}^1)^T} & \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (\mathbf{x}^2)^T} & \cdots & \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (\mathbf{x}^M)^T} \end{bmatrix}, \tag{254}$$

where, by Eq. (201),

$$\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (\mathbf{x}^m)^T} = \begin{bmatrix} \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (vec\mathbf{W}^m)^T} & \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (\mathbf{b}^m)^T} \end{bmatrix}. \tag{255}$$

From Eq. (200) and Eq. (253), the matrix $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial \mathbf{x}^T$ can also be written as

$$\tilde{\mathbf{J}}_r(\mathbf{x}) = \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial \mathbf{x}^T} = \begin{bmatrix} \frac{\partial}{\partial (\mathbf{x}^1)^T}\left(\frac{\partial \mathbf{e}_1}{\partial p_{r,1}}\right) & \frac{\partial}{\partial (\mathbf{x}^2)^T}\left(\frac{\partial \mathbf{e}_1}{\partial p_{r,1}}\right) & \cdots & \frac{\partial}{\partial (\mathbf{x}^M)^T}\left(\frac{\partial \mathbf{e}_1}{\partial p_{r,1}}\right) \\ \frac{\partial}{\partial (\mathbf{x}^1)^T}\left(\frac{\partial \mathbf{e}_2}{\partial p_{r,2}}\right) & \frac{\partial}{\partial (\mathbf{x}^2)^T}\left(\frac{\partial \mathbf{e}_2}{\partial p_{r,2}}\right) & \cdots & \frac{\partial}{\partial (\mathbf{x}^M)^T}\left(\frac{\partial \mathbf{e}_2}{\partial p_{r,2}}\right) \\ \cdots & \cdots & & \cdots \\ \frac{\partial}{\partial (\mathbf{x}^1)^T}\left(\frac{\partial \mathbf{e}_Q}{\partial p_{r,Q}}\right) & \frac{\partial}{\partial (\mathbf{x}^2)^T}\left(\frac{\partial \mathbf{e}_Q}{\partial p_{r,Q}}\right) & \cdots & \frac{\partial}{\partial (\mathbf{x}^M)^T}\left(\frac{\partial \mathbf{e}_Q}{\partial p_{r,Q}}\right) \end{bmatrix}, \tag{256}$$

where

$$\frac{\partial}{\partial(\mathbf{x}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) = \left[\frac{\partial}{\partial(vec\mathbf{W}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) \quad \frac{\partial}{\partial(\mathbf{b}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)\right]. \tag{257}$$

To obtain $\tilde{\mathbf{J}}_r(\mathbf{x})$, we need $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial(vec\mathbf{W}^m)^T$ and $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial(\mathbf{b}^m)^T$. However, it will be easier if we first consider $\frac{\partial}{\partial(vec\mathbf{W}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)$ and $\frac{\partial}{\partial(\mathbf{b}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)$.

Consider one element in $\frac{\partial}{\partial(vec\mathbf{W}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)$, which is $\frac{\partial}{\partial w_{i,j}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right)$. Therefore, from Eq. (208), $\frac{\partial}{\partial(vec\mathbf{W}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)$ can be expressed as:

$$\begin{aligned}\frac{\partial}{\partial(vec\mathbf{W}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) = &\left\{\mathbf{1}_{1\times S^{m-1}} \otimes \frac{\partial}{\partial(\mathbf{n}_q^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)\right\} \bullet \left\{(\mathbf{a}_q^{m-1})^T \otimes \mathbf{1}_{S^M \times S^m}\right\} \\ &+ \left\{\mathbf{1}_{1\times S^{m-1}} \otimes \frac{\partial \mathbf{e}_q}{\partial(\mathbf{n}_q^m)^T}\right\} \bullet \left\{\left(\frac{\partial \mathbf{a}_q^{m-1}}{\partial p_{r,q}}\right)^T \otimes \mathbf{1}_{S^M \times S^m}\right\}.\end{aligned} \tag{258}$$

The elements of $\frac{\partial}{\partial(\mathbf{b}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)$ are $\frac{\partial}{\partial b_i^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right)$. By using Eq. (209), $\frac{\partial}{\partial(\mathbf{b}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right)$ can be written as

$$\frac{\partial}{\partial(\mathbf{b}^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) = \frac{\partial}{\partial(\mathbf{n}_q^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right). \tag{259}$$

From Eq. (258), the batch matrix $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial(vec\mathbf{W}^m)^T$ can be expressed as

$$\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (vec\mathbf{W}^m)^T} = \left(\left\{ \mathbf{1}_{1 \times S^{m-1}} \otimes \left(\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^m}) \right) \right\} \bullet \left\{ (\mathbf{A}^{m-1})^T \otimes \mathbf{1}_{S^M \times S^m} \right\} \right) + \left(\left\{ \mathbf{1}_{1 \times S^{m-1}} \otimes \left(\frac{\partial vec\mathbf{E}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^m}) \right) \right\} \bullet \left\{ \left((\mathbf{1}_{1 \times Q} \otimes \mathbf{I}_{S^{m-1}}) \times \frac{\partial vec\mathbf{A}^{m-1}}{\partial_r \mathbf{p}^T} \right)^T \otimes \mathbf{1}_{S^M \times S^m} \right\} \right) . \tag{260}$$

From Eq. (259), the batch matrix $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial (\mathbf{b}^m)^T$ can be written as

$$\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (\mathbf{b}^m)^T} = \frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q \times 1} \otimes \mathbf{I}_{S^m}) . \tag{261}$$

To clarify Eq. (260) and Eq. (261), note that

$$\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (vec\mathbf{N}^m)^T} = \begin{bmatrix} \frac{\partial}{\partial (\mathbf{n}_1^m)^T} \left(\frac{\partial \mathbf{e}_1}{\partial p_{r,1}} \right) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial}{\partial (\mathbf{n}_2^m)^T} \left(\frac{\partial \mathbf{e}_2}{\partial p_{r,2}} \right) & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial}{\partial (\mathbf{n}_Q^m)^T} \left(\frac{\partial \mathbf{e}_Q}{\partial p_{r,Q}} \right) \end{bmatrix} , \tag{262}$$

and, hence

$$\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial (vec\mathbf{N}^m)^T} \times (\mathbf{1}_{Q\times 1} \otimes \mathbf{I}_{S^m}) = \begin{bmatrix} \frac{\partial}{\partial (\mathbf{n}_1^m)^T}\left(\frac{\partial \mathbf{e}_1}{\partial p_{r,1}}\right) \\ \frac{\partial}{\partial (\mathbf{n}_2^m)^T}\left(\frac{\partial \mathbf{e}_2}{\partial p_{r,2}}\right) \\ \cdots \\ \frac{\partial}{\partial (\mathbf{n}_Q^m)^T}\left(\frac{\partial \mathbf{e}_Q}{\partial p_{r,Q}}\right) \end{bmatrix}. \tag{263}$$

Note further that the product of $(\mathbf{1}_{1\times Q} \otimes \mathbf{I}_{S^{m-1}})$ and $\partial vec\mathbf{A}^{m-1}/\partial_r \mathbf{p}^T$ in Eq. (260) is already shown in Eq. (153).

Now, from Eq. (260) and Eq. (261), $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial (vec\mathbf{N}^m)^T$ is the only term we have not yet computed. We will show its computation next.

Calculation of $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial (vec\mathbf{N}^m)^T$

Consider one element of $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial (vec\mathbf{N}^m)^T$, which is $\frac{\partial}{\partial n_{i,q}^m}\left(\frac{\partial e_{k,q}}{\partial p_{r,q}}\right)$. From Eq. (224), we find

$$\begin{aligned} \frac{\partial}{\partial (\mathbf{n}_q^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) = & \left\{\left(\frac{\partial \mathbf{df}^m(\mathbf{n}_q^m)}{\partial p_{r,q}}\right)^T \otimes \mathbf{1}_{S^M\times 1}\right\} \bullet \left\{\frac{\partial \mathbf{e}_q}{\partial (\mathbf{n}_q^{m+1})^T} \times \mathbf{W}^{m+1}\right\} + \\ & \left\{(\mathbf{df}^m(\mathbf{n}_q^m))^T \otimes \mathbf{1}_{S^M\times 1}\right\} \bullet \left\{\frac{\partial}{\partial (\mathbf{n}_q^{m+1})^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) \times \mathbf{W}^{m+1}\right\}, \end{aligned} \tag{264}$$

where $\mathbf{df}^m(\mathbf{n}_q^m)$ is defined in Eq. (133) and $\partial \mathbf{df}^m(\mathbf{n}_q^m)/\partial p_{r,q}$ can be computed by Eq.

(178) in Chapter 4. Note that Eq. (264) can be also expressed as:

$$\frac{\partial}{\partial(\mathbf{n}_q^m)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) = \frac{\partial \mathbf{e}_q}{\partial(\mathbf{n}_q^{m+1})^T} \times \mathbf{W}^{m+1} \times \frac{\partial \dot{\mathbf{F}}^m(\mathbf{n}_q^m)}{\partial p_{r,q}} + \frac{\partial}{\partial(\mathbf{n}_q^{m+1})^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) \times \mathbf{W}^{m+1} \times \dot{\mathbf{F}}^m(\mathbf{n}_q^m) \; . \tag{265}$$

However, we will use Eq. (264) as part of our computation for two reasons. First, the notations in Eq. (264) is in a similar form as Eq. (225). Second, we have already developed a notation for the term $\partial \dot{f}^m(n_{i,q}^m)/\partial p_{r,q}$, i.e. $\partial vec\mathbf{DF}^m(\mathbf{N}^m)/\partial_r \mathbf{p}^T$, thus it is more convenient to use Eq. (264).

From Eq. (262) and Eq. (264), $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial(vec\mathbf{N}^m)^T$ can be computed by:

$$\begin{aligned}\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial(vec\mathbf{N}^m)^T} &= \left\{\left(\frac{\partial vec\mathbf{DF}^m(\mathbf{N}^m)}{\partial_r \mathbf{p}^T}\right)^T \otimes \mathbf{1}_{S^M \times 1}\right\} \bullet \left\{\frac{\partial vec\mathbf{E}}{\partial(vec\mathbf{N}^{m+1})^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^{m+1})\right\} \\ &+ \left\{\left(\left\{\mathbf{1}_{1\times Q} \otimes (\mathbf{DF}^m(\mathbf{N}^m))^T\right\} \bullet \{\mathbf{I}_Q \otimes \mathbf{1}_{1\times S^m}\}\right) \otimes \mathbf{1}_{S^M \times 1}\right\} \\ &\bullet \left\{\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial(vec\mathbf{N}^{m+1})^T} \times (\mathbf{I}_Q \otimes \mathbf{W}^{m+1})\right\} \; ,\end{aligned} \tag{266}$$

where the term $\partial vec\mathbf{DF}^m(\mathbf{N}^m)/\partial_r \mathbf{p}^T$ can be computed by Eq. (179). The second term in Eq. (266) can be clarified as follow:

$$\mathbf{1}_{1\times Q} \otimes (\mathbf{DF}^m(\mathbf{N}^m))^T = \left[(\mathbf{DF}^m(\mathbf{N}^m))^T \; (\mathbf{DF}^m(\mathbf{N}^m))^T \; \ldots \; (\mathbf{DF}^m(\mathbf{N}^m))^T\right], \text{ and} \tag{267}$$

$$\mathbf{I}_Q \otimes \mathbf{1}_{1\times S^m} = \begin{bmatrix} \mathbf{1}_{1\times S^m} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{1}_{1\times S^m} & \cdots & \mathbf{0} \\ & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{1}_{1\times S^m} \end{bmatrix}. \tag{268}$$

Therefore, the Hadamard product of the two matrices results in

$$\begin{aligned} &\left\{\mathbf{1}_{1\times Q} \otimes (\mathbf{DF}^m(\mathbf{N}^m))^T\right\} \bullet \{\mathbf{I}_Q \otimes \mathbf{1}_{1\times S^m}\} \\ &= \begin{bmatrix} (\mathbf{df}^m(\mathbf{n}_1^m))^T & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & (\mathbf{df}^m(\mathbf{n}_2^m))^T & \cdots & \mathbf{0} \\ \cdots & \cdots & & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & (\mathbf{df}^m(\mathbf{n}_Q^m))^T \end{bmatrix}. \end{aligned} \tag{269}$$

Since Eq. (266) is a backpropagation process, we need to initialize it at the output layer, i.e. $m = M$. From Eq. (234), the matrix

$$\frac{\partial}{\partial(\mathbf{n}_q^M)^T}\left(\frac{\partial \mathbf{e}_q}{\partial p_{r,q}}\right) = \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial \mathbf{e}_q}{\partial(\mathbf{n}_q^M)^T}\right) = \frac{\partial}{\partial p_{r,q}}\left(\frac{\partial \mathbf{a}_q}{\partial(\mathbf{n}_q^M)^T}\right) = \frac{\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)}{\partial p_{r,q}}, \tag{270}$$

where $\partial \dot{\mathbf{F}}^M(\mathbf{n}_q^M)/\partial p_{r,q}$ can be computed by Eq. (237). Thus, from Eq. (237) and Eq. (262), $\partial vec\mathbf{DE}_{r\mathbf{p}}/\partial(vec\mathbf{N}^M)^T$ can be decomposed to:

$$\frac{\partial vec\mathbf{DE}_{r\mathbf{p}}}{\partial(vec\mathbf{N}^M)^T} = \frac{\partial vec\mathbf{DF}^M(\mathbf{N}^M)}{\partial(vec\mathbf{A})^T} \bullet \left\{\frac{\partial vec\mathbf{A}}{\partial_r \mathbf{p}^T} \otimes \mathbf{1}_{1\times S^M}\right\}. \tag{271}$$

This completes the derivation for calculating the Jacobian matrix $\tilde{\mathbf{J}}(\mathbf{x})$ and the vector $\tilde{\mathbf{z}}(\mathbf{x})$ for the *CFDA* performance index with the memory save approach. In the next section, we will compare the execution time for computing the gradient and the approximated Hessian between the standard backpropagation and the *CFDA* method. We summarize the *CFDA* method with Levenberg-Marquardt using the memory save approach below.

Summary of memory-save calculation

STEPS FOR CFDA WITH LEVENBERG-MARQUARDT

(MEMORY SAVE)

1. Given Q inputs $\mathbf{P}$, obtain $\mathbf{A}^m$. Initialize $r = 1$.

2. Obtain $\partial vec\mathbf{A}^m / \partial_r \mathbf{p}^T$ by Eq. (165) and Eq. (167).

3. With $\partial vec\mathbf{G} / \partial_r \mathbf{p}^T$, compute $\mathbf{DE}_{r\mathbf{p}}$ by Eq. (157) and Eq. (250). Obtain $\tilde{\mathbf{z}}_r(\mathbf{x})$ by Eq. (252).

4. Compute $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^M)^T$ and $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial (vec\mathbf{N}^M)^T$, using Eq. (222) and Eq. (271), respectively.

5. Backpropagate $\partial vec\mathbf{E} / \partial (vec\mathbf{N}^m)^T$ and $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial (vec\mathbf{N}^m)^T$ by Eq. (220) and Eq. (266), respectively.

6. Compute $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial (vec\mathbf{W}^m)^T$ and $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial (\mathbf{b}^m)^T$, using Eq. (260) and Eq. (261), respectively. Then, obtain $\partial vec\mathbf{DE}_{r\mathbf{p}} / \partial (\mathbf{x}^m)^T$ by Eq. (255).

7. Obtain $\tilde{\mathbf{J}}_r^T(\mathbf{x})$ by Eq. (254). Set $r = r + 1$, and repeat step $2 - 7$ until $r > R$.

8. Obtain $\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{z}}(\mathbf{x})$ and $\tilde{\mathbf{J}}^T(\mathbf{x})\tilde{\mathbf{J}}(\mathbf{x})$ by Eq. (246).

9. Update the network parameters by Eq. (191).

Speed Test

In this section, we will compare the execution time for computing the gradient and the approximate Hessian matrix under the Levenberg-Marquardt framework in one iteration in three different scenarios. These three scenarios are (see Eq. (192) and Eq. (193)):

1. The sum squared function error, i.e. $F_1(\mathbf{x})$, using the batch mode,

2. The sum squared derivative error, i.e. $F_2(\mathbf{x})$, using the batch mode, and

3. The sum squared derivative error, using the memory-save approach.

The following figure shows the relative execution time (averaging over ten runs) for computing the gradient and the Jacobian for $F_2(\mathbf{x})$ in batch and memory-save approaches, using the $R-25-1$ networks, compared to the time for computing the gradient and the Jacobian for $F_1(\mathbf{x})$. Note that the number of training examples in this case was $Q = 5,000$.

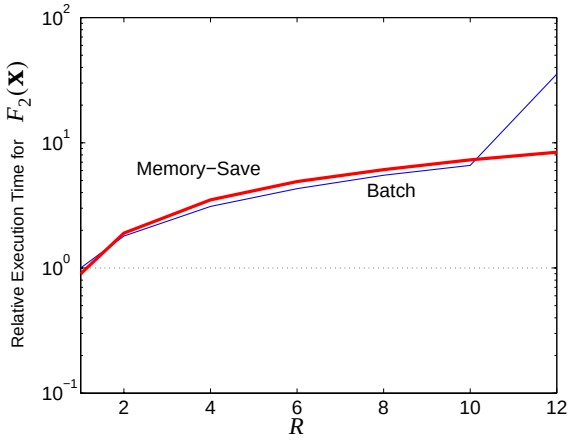

Figure 7) Relative execution time for computing the gradient and Jacobian of $F_2(\mathbf{x})$, compared to $F_1(\mathbf{x})$

From Figure 7), we can see that computing the gradient and Jacobian for $F_2(\mathbf{x})$ took more time than for $F_1(\mathbf{x})$ (which was the standard training method). In batch mode, computing the gradient and the approximated Hessian for $F_2(\mathbf{x})$ took slightly less time than that in memory-save approach. More operations (e.g. kronecker products) in batch mode made it just slightly more efficient than in the memory-save approach. Once the input dimension was greater than ten, the computer's RAM overflowed, thus requiring data to be sent to disk. The execution time in batch mode for this case thus was greater than that in memory-save approach.

Summary

In this chapter, the algorithm for the $CFDA$ method under the Levenberg-Marquardt framework was derived. Two approaches were presented. The first method performs the calculation in batch mode. The second method is the memory-save approach, which intends to make the matrix size independent of the input dimension R (same concept as the memory-save approach in Chapter 4). The computation times under different conditions were measured. The results showed that computing the gradient and approximated Hessian for the term $F_2(\mathbf{x})$ in the $CFDA$ method was longer than that in standard backpropagation. This is expected since its computation is more complicated and it involves larger matrices than the calculation in standard backpropagation. The results also illustrated that the memory-save approach is useful in cases where computer RAM is insufficient to perform the calculation in batch mode.

CHAPTER 6

A NETWORK PRUNING ALGORITHM FOR CFDA

Introduction

One of the major problems in nonlinear fitting is overfitting. In Chapter 2, we briefly mentioned some common techniques to prevent overfitting in function approximation. One of the most common techniques was to prune the network (see [SiDo88] or [Reed93]). Many methods have been proposed to efficiently prune the network. For example, optimal brain surgeon [HaSt93] computes the product of the weight squared and the second-order derivative of the function error with respect to the weights. A neuron is pruned if the product is sufficiently small. [Lo99] proposed a statistical-based method, which computes the covariance of the weights and uses the $z-statistics$ to decide which neurons to prune. The method proposed in [Karn90] measures the sensitivity of the function error with respect to the removal of each weight. The weights with low sensitivity are pruned. [Enge01] proposed another statistical-based method, which prunes the weight with the variance in the sensitivity not significantly different from zero over all the training data. The sensitivity is based on the calculation of the derivatives of the network output with respect to the network parameters. [LaFo06] analyzed the Fourier decomposition of the variance of the network output with respect to each weight. This information is used to assess which weight will be eliminated. Some other methods are also discussed in [SeGa00], [WaHi00] or [HuSe05].

When fitting both a function and its first-order derivatives through $CFDA$ training algorithms, we expect less overfitting. That is, in the local neighborhoods of training inputs, the function response of the neural network is more accurate since the derivatives at the training points are forced to be correct. However, we have observed new types of overfitting. In this chapter, we will discuss these new types of overfitting as well as propose a method to mitigate the problems. We will focus only on the case of fitting an $R - S^1 - 1$ network with the $CFDA$ training algorithms, where the transfer function of the network is hyperbolic tangent sigmoid.

We will start this chapter by discussing the impact of the network parameters on the function and derivative response. Then, we describe new types of overfitting for networks trained by the $CFDA$ algorithms. Finally, we will propose a pruning method to remove these types of overfitting from the network responses.

Two-Layer Network Response

In this section, we will discuss the impact of the network parameters in a $R - S^1 - 1$ network on the function and derivative responses. Given a $R - S^1 - 1$ network, the function response of the network can be expressed as:

$$a_1^2 = \sum_{i=1}^{S^1} w_{1,i}^2 a_i^1 + b^2, \tag{272}$$

where

$$a_i^1 = f^1(n_i^1) \text{ and } n_i^1 = \sum_{r=1}^{R} w_{i,r}^1 p_r + b_i^1 . \tag{273}$$

By taking the derivative of Eq. (272) with respect to the input p_r, we obtain

$$\frac{\partial a_1^2}{\partial p_r} = \sum_{i=1}^{S^1} w_{1,i}^2 \frac{\partial a_i^1}{\partial p_r} . \tag{274}$$

Using the chain rule of calculus, we can further express the term $\partial a_i^1 / \partial p_r$ as:

$$\frac{\partial a_i^1}{\partial p_r} = \frac{\partial a_i^1}{\partial n_i^1} \times \frac{\partial n_i^1}{\partial p_r} = \dot{f}^1(n_i^1) \times w_{i,r}^1 . \tag{275}$$

Substituting Eq. (275) into Eq. (274), the derivative response can be written as

$$\frac{\partial a_1^2}{\partial p_r} = \sum_{i=1}^{S^1} w_{1,i}^2 w_{i,r}^1 \dot{f}^1(n_i^1) . \tag{276}$$

For ease of reference in this chapter, we also introduce the notation:

$$y_i = w_{1,i}^2 f^1(n_i^1) \text{ and } \frac{\partial y_i}{\partial p_r} = w_{1,i}^2 w_{i,r}^1 \dot{f}^1(n_i^1) . \tag{277}$$

Therefore, Eq. (272) and Eq. (276) can also be written as

$$a_1^2 = \sum_{i=1}^{S^1} y_i + b^2 \text{ and } \frac{\partial a_1^2}{\partial p_r} = \sum_{i=1}^{S^1} \frac{\partial y_i}{\partial p_r} . \tag{278}$$

From Eq. (278), the terms y_i and $\partial y_i / \partial p_r$ can be viewed as the function and derivative responses of the i^{th} neuron. The network's function response a_1^2 and derivative response

$\partial a_1^2 / \partial p_r$ are linear combinations of the terms y_i and $\partial y_i / \partial p_r$, respectively.

To illustrate how Eq. (278) works, we start with the case of a single input and assume that the transfer function is hyperbolic tangent sigmoid. Observe that there are four types of parameters contributing to the function and derivative responses: the first-layer bias b_i^1, the second-layer bias b^2, the first-layer weight w_i^1 and the second-layer weight $w_{1,i}^2$, for $i = 1, \ldots, S^1$. It is clear from Eq. (278) that the term b^2 shifts the entire function response a_1^2 up or down, while it does not contribute to the derivative response $\partial a_1^2 / \partial p_r$. Next, we will consider the impact of the other three parameters on the network responses. Specifically, we will focus on the impact of the three parameters to the terms y_i and $\partial y_i / \partial p_r$. First recall that the terms y_i and $\partial y_i / \partial p_r$ depend on the terms $f^1(n_i^1)$ and $\dot{f}^1(n_i^1)$, respectively. With the transfer function being hyperbolic tangent sigmoid, the term $\dot{f}^1(n_i^1)$ can be computed by

$$\dot{f}^1(n_i^1) = 1 - (a_i^1)^2. \tag{279}$$

The sketches of the term $f^1(n_i^1)$ and $\dot{f}^1(n_i^1)$ are shown in Figure 8).

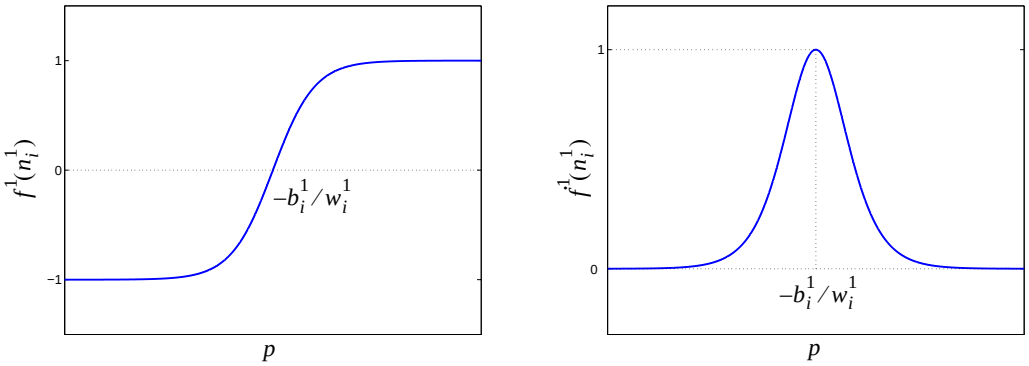

Figure 8) Sketches of $f^1(n_i^1)$ and $\dot{f}^1(n_i^1)$

The effect of the first-layer bias b_i^1 on the terms $f^1(n_i^1)$ and $\dot{f}^1(n_i^1)$ can be seen from Figure 8). It controls the center of the neuron response. The center at $p = -b_i^1/w_i^1$ is obtained by solving $n_i^1 = w_i^1 p + b_i^1 = 0$. The effect of the first-layer weight w_i^1 on the terms $f^1(n_i^1)$ and $\dot{f}^1(n_i^1)$ is shown in Figure 9).

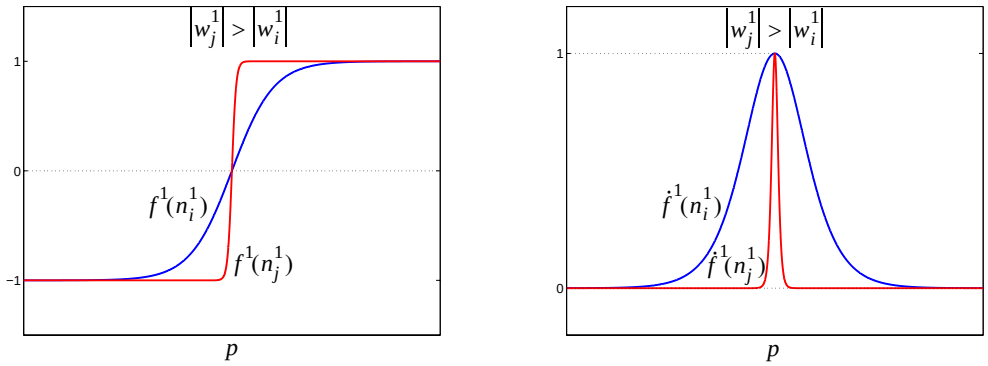

Figure 9) Effect of the first-layer weight

From Figure 9), we can see that, as w_i^1 increases, the terms $f^1(n_i^1)$ and $\dot{f}^1(n_i^1)$ change more rapidly. For the term $\dot{f}^1(n_i^1)$, we see that the width of the response is narrower as the

magnitude of w_i^1 increases. In addition, from Eq. (277), w_i^1 affects the term $\partial y_i / \partial p_r$ by also scaling the response of $\dot{f}^1(n_i^1)$.

For the second-layer weight $w_{1,i}^2$, its impact on the terms y_i and $\partial y_i / \partial p_r$ can be seen from Eq. (277), where it scales both the values of $f^1(n_i^1)$ and $\dot{f}^1(n_i^1)$. Figure 10) illustrates the impact of $w_{1,i}^2$ when its value increases.

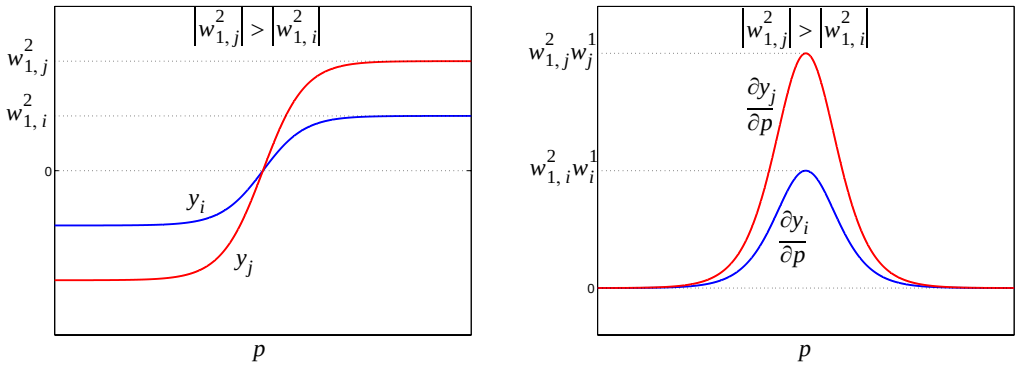

Figure 10) Effect of the second-layer weight

For the single-input case, we can see from Eq. (277) and Figure 10) that the maximum magnitude of $\partial y_i / \partial p_r$, which occurs at the neuron center, equals $\left| w_{1,i}^2 w_i^1 \right|$.

Now, consider the case of multiple inputs. Recall in the single-input case that the neuron center is obtained by solving $n_i^1 = w_i^1 p + b_i^1 = 0$. For multiple inputs, the center of the neuron is governed by the equation:

$$n_i^1 = \sum_{r=1}^{R} w_{i,r}^1 p_r + b_i^1 = 0. \tag{280}$$

That is, the center of the neuron becomes a line for two dimensional inputs, a plane for three

dimensional inputs or a hyperplane for higher dimensions. The impact of the network parameters on the neuron's responses for multiple inputs is similar to the single-input case. For example, consider a case of two dimensional inputs. If we assume $b_i^1 = 0$, $b^2 = 0$, $({}_i\mathbf{w}^1)^T = \begin{bmatrix} 5 & -1 \end{bmatrix}$ and $w_{1,i}^2 = 1$, the terms y_i, $\partial y_i / \partial p_1$, and $\partial y_i / \partial p_2$ in the region of $-1 \le p_1, p_2 \le 1$ are illustrated below.

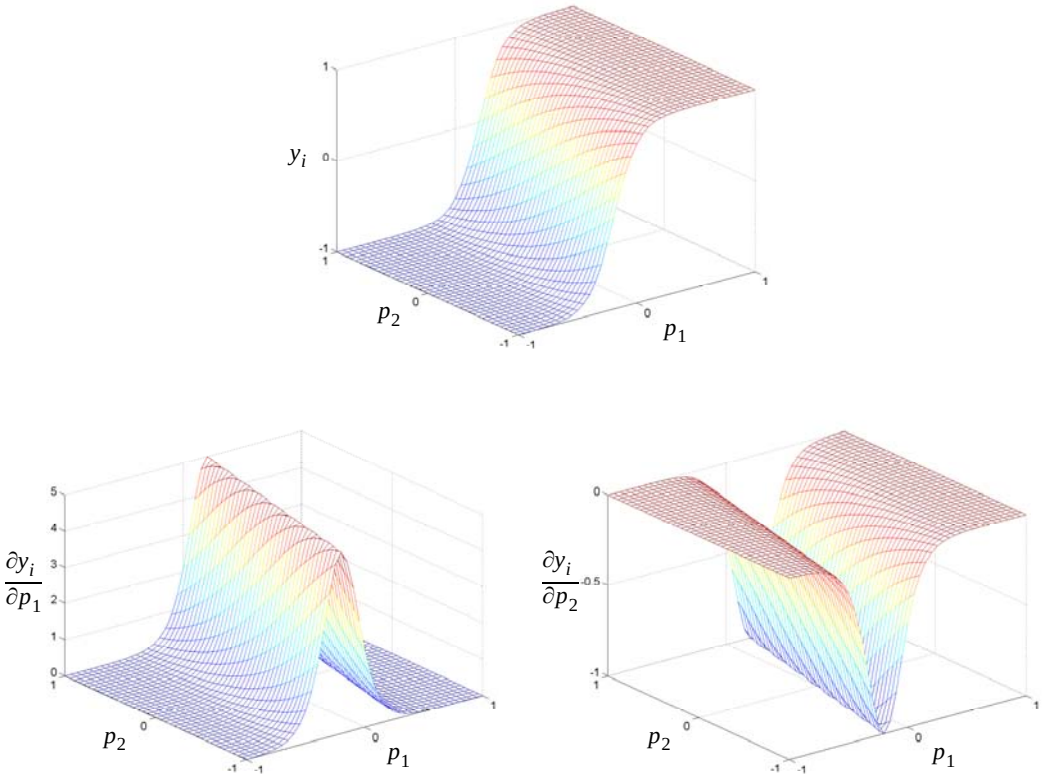

Figure 11) Neuron's function and derivative responses in two dimensions

In the example, the neuron center occurs along the line $5p_1 - p_2 = 0$. The widths of the terms $\partial y_i / \partial p_1$ and $\partial y_i / \partial p_2$ are controlled by the first-layer weights, which are now composed of two elements (i.e. $\left|w_{i,1}^1\right| = 5$ and $\left|w_{i,2}^1\right| = 1$). With $\left|w_{i,1}^1\right| > \left|w_{i,2}^1\right|$, we can see

that the responses change along p_1 much more rapidly than along p_2. To illustrate this, let us plot $\partial y_i / \partial p_1$ along the line $p_2 = 0$ (and plot $\partial y_i / \partial p_2$ along the line $p_1 = 0$). The result is shown in Figure 12).

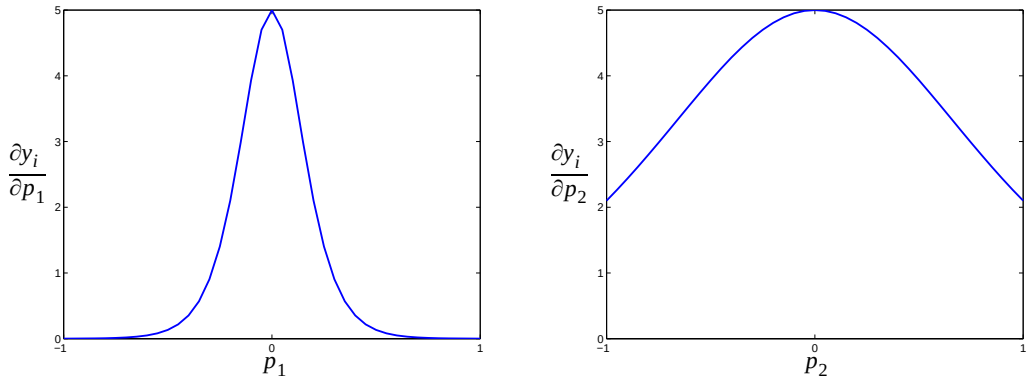

Figure 12) A cross section of the neuron's derivative responses

From Figure 11), we see that the extreme values for $\partial y_i / \partial p_1$ and $\partial y_i / \partial p_2$ equal $w^2_{1,i} w^1_{i,1} = 5$ and $w^2_{1,i} w^1_{i,2} = -1$, respectively, and they occur along the neuron center. The magnitude of $\partial y_i / \partial \mathbf{p}^T$ equals

$$\left\| \frac{\partial y_i}{\partial \mathbf{p}^T} \right\| = \sqrt{\sum_{r=1}^{R} \left(\frac{\partial y_i}{\partial p_r} \right)^2} = \left| w^2_{1,i} \right| \times \left\| {}_i\mathbf{w}^1 \right\| \times \left| \dot{f}^1(n^1_i) \right| . \tag{281}$$

Thus, at the neuron center, the magnitude is $\left| w^2_{1,i} \right| \left\| {}_i\mathbf{w}^1 \right\|$.

In this section, we provided the fundamental concepts of how each network parameter contributes to the function and derivative responses of a $R - S^1 - 1$ network. With this background, we are ready to introduce $CFDA$ overfitting in the next section.

CFDA Overfitting

We will introduce two types of $CFDA$ overfitting, $Type-A$ and $Type-B$. For $Type-A$ overfitting, the network responses produce accurate results at every training point. However, both the function and derivative responses are inaccurate at some points outside the training set. For $Type-B$ overfitting, the network's derivative responses produce accurate results at all training points. However, the network's function response has small errors at some training points. The two types of overfitting, as well as the guidelines of the method to eliminate them, are discussed as follows.

Type-A

For the first type of overfitting, the network responses are accurate over the training set. However, at some spaces between training data, the network performs very poorly, causing very large approximation errors on both function and the first derivatives. The problem comes from a group of neurons (greater than one), whose responses are cancelled at all training points. That is, their responses, when combined together, produce insignificant contribution to the fitting over the training data. Unfortunately, their responses may not cancel at some spaces between training points, thus yielding inaccurate responses at these locations.

To illustrate the idea, consider a $1-S^1-1$ network. Assume that two neurons of the network yield their function and derivative responses as shown in Figure 13). Note that the symbol $\times$ in the figure represents the location of training data.

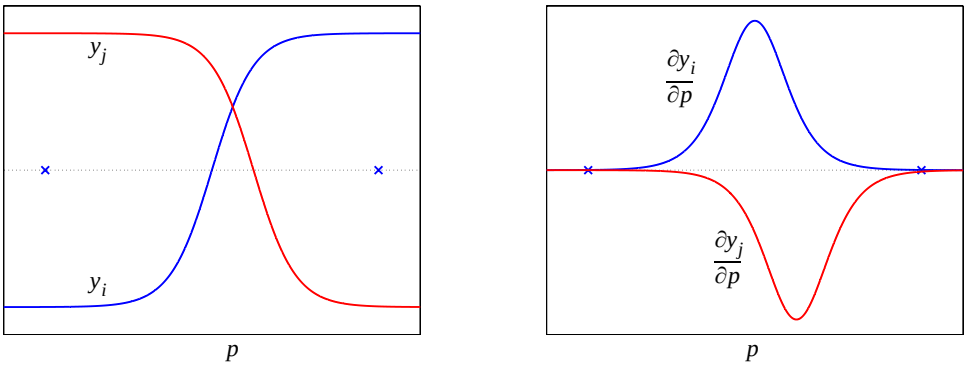

Figure 13) Responses of two neurons

If we combine the responses of these two neurons together, we obtain Figure 14).

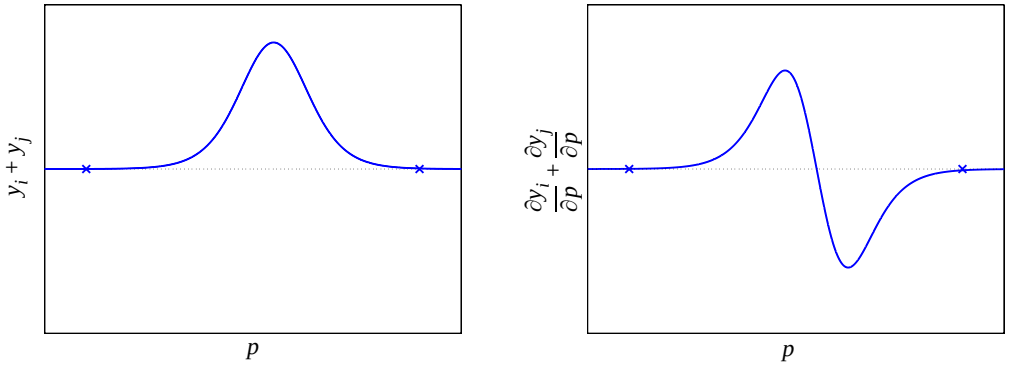

Figure 14) Type-A Overfitting

From Figure 14), we can see that the responses from the two neurons cancel almost exactly at training points, but not at the points in between. The remainder after cancellation is $Type - A$ overfitting. If these two neurons are removed, it will not produce a significant change to the original fitting (since their responses yield almost no contribution to the training data). However, it will eliminate the overfitting. The basic idea of the pruning method is discussed below.

From Figure 13) and Figure 14), we can see that the severity of the overfitting (which depends on the magnitude of the remainder) in the derivative is associated with the

magnitude of the responses of the two neurons. Recall from the previous section that, for the single-input case, the maximum magnitude of $\partial y_i / \partial p$ equals the weight product $\left| w^2_{1,i} w^1_i \right|$. For multiple inputs, it equals $\left| w^2_{1,i} \right| \left\| {}_i\mathbf{w}^1 \right\|$. Therefore, the higher the weight products are, the worse the overfitting could be.

The weight product of a neuron is defined as "high" if it is larger than the actual derivative of the training point closest to the neuron center. (The reason we select the training point closest to the neuron center is that it is the point to which the neuron response most contributes.) The distance between a training point p_q and the i^{th} neuron center (at $p = -b^1_i / w^1_i$) can be easily computed in the case of single input. For multiple inputs, it requires the calculation illustrated in Figure 15).

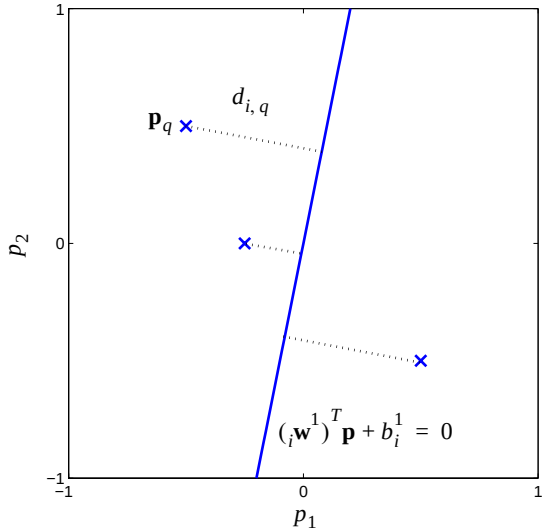

Figure 15) Distance between a point to a line

From geometry, the distance from the training point $\mathbf{p}_q$ to the i^{th} neuron's center (which

is $({}_i\mathbf{w}^1)^T\mathbf{p} + b_i^1 = 0$) is

$$d_{i,q} = \frac{\left|({}_i\mathbf{w}^1)^T\mathbf{p}_q + b_i^1\right|}{\left\|{}_i\mathbf{w}^1\right\|} = \frac{\left|n_{i,q}^1\right|}{\left\|{}_i\mathbf{w}^1\right\|}. \tag{282}$$

The neuron with high weight product will be selected as a candidate that yields $Type-A$ overfitting. Once the candidate is chosen, we need to search for other neurons, whose responses will cancel the response of the candidate over the training data. Practically, it is impossible to validate every possible combination of neurons, since the number of combinations could be extremely large. It is also not useful to verify every possible combination of neurons, since neurons with centers far from the candidate's center would not provide the response cancellation. Therefore, checking only a set of neurons whose centers are close to the candidate's center would be sufficient. We thus need to define the "neighbors" of the neuron candidate.

From Figure 13) and Figure 14), we can see that the largest distance between the center of two neurons that cause overfitting is the distance between the two training points. If the distance between the two centers is greater than or equal to the distance between the two training points, it implies that each neuron's response contributes to the fitting of the two training data. Therefore, we use the maximum distance between neighboring training points to be the threshold for deciding which neurons should be tested for response cancellation. That is, any neurons whose centers are closer to the candidate center than the maximum distance between neighboring data points are considered the candidate's neighbors. In the case of single input, it is easy to calculate the distance between weight centers. For multiple inputs, more consideration is needed.

The definition of a neighbor for multiple inputs requires that a neuron center be close to the candidate center, and that the two centers be parallel. Theoretically, the probability of having two centers in parallel is zero. An angle tolerance is applied in practice. Therefore, neighbors in the case of multiple inputs are those neurons whose centers are parallel within a tolerance and are close to the candidate center. We fix the angle tolerance at one degree. To compute the distance between the neuron centers, we further assume that those centers (whose angles are within one degree) are parallel. By geometry, it can be shown that the distance between two parallel hyperplanes, $({}_i\mathbf{w}^1)^T\mathbf{p} + b_i^1 = 0$ and $({}_j\mathbf{w}^1)^T\mathbf{p} + b_j^1 = 0$, is

$$d_{H_{i,j}} = \left| \frac{b_i^1}{\left\| {}_i\mathbf{w}^1 \right\|} - \frac{b_j^1}{\left\| {}_j\mathbf{w}^1 \right\|} \right|, \tag{283}$$

if ${}_i\mathbf{w}^1$ and ${}_j\mathbf{w}^1$ are in the same direction, or

$$d_{H_{i,j}} = \left| \frac{b_i^1}{\left\| {}_i\mathbf{w}^1 \right\|} + \frac{b_j^1}{\left\| {}_j\mathbf{w}^1 \right\|} \right|, \tag{284}$$

if the directions of ${}_i\mathbf{w}^1$ and ${}_j\mathbf{w}^1$ are opposite. We can use Eq. (284) and Eq. (285) to compute the distance between two parallel (within a tolerance) centers. Then, we compare the distance against the distance threshold, specified by the maximum distance between neighboring training points.

Once having a candidate and its neighbors, combinations of these neurons can be tested. We need to verify if cancellation (over the training points) occurs for any of these

combinations. If cancellation occurs, the combination of neurons is pruned. Note that there could exist more than one combination yielding an insignificant contribution. In this situation, the combination with highest number of neurons is pruned.

When verifying cancellation for a combination of neurons, their combined derivative response is measured over the training set. Then, the maximum magnitude of the derivative response (chosen among the training points) will be compared with the true derivative magnitude evaluated at the training point. If the maximum derivative response from these neurons is much less than the true derivative value, we say that the derivative response from these neurons contributes insignificantly to the fitting. Thus, these neurons are pruned. The mathematical procedure will be described in detail after we finish discussing the other type of $CFDA$ overfitting.

In the next section, we will introduce $Type-B$ overfitting, as well as the method to remove it.

Type-B

For $Type-B$ overfitting, the network produces accurate derivative response over the training set. However, the function response has small errors for some training data. At some locations between training points that produce accurate function response and training points whose function response is inaccurate, the network response changes rapidly. This produces a large derivative response between training points. This problem is caused by a local minimum in the $CFDA$ performance surface. Although there is a very small function error, the derivative error is minimized. This type of overfitting could be generated by a single neuron, or by multiple neurons.

To demonstrate the $Type-B$ overfitting, consider a $1-S^1-1$ network. Assume that one neuron of the network yields the function and derivative responses as shown in Figure 16).

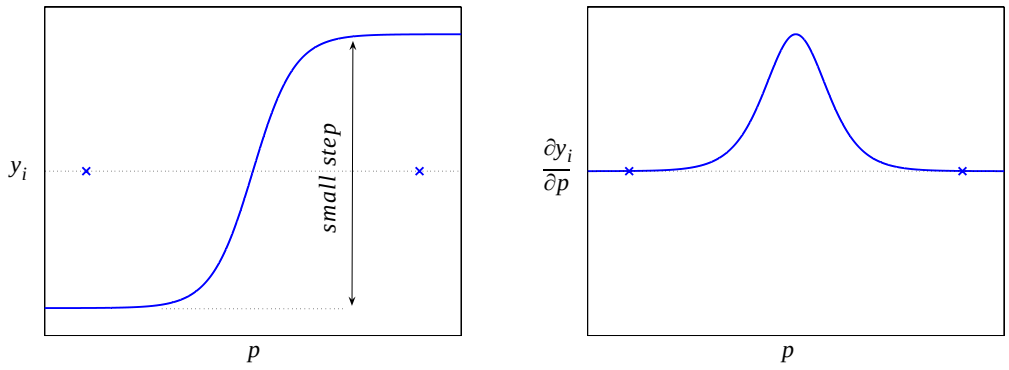

Figure 16) Type-B Overfitting

From Figure 16), it should be noted that the step in the function response y_i must be very small. (Otherwise, the function errors at the training points would be large and, thus, the training process could further reduce these function errors.) A small step in y_i corresponds to a small magnitude for $\partial y_i / \partial p$, i.e. a small weight product $\left| w^2_{1,i} w^1_i \right|$. From the figure, the response $\partial y_i / \partial p$ does not contribute to the derivative fitting at the training points, but it causes a small fluctuation at points between the training data. If this neuron is eliminated from the network, it will not produce a significant impact on the overall derivative responses at training data. However, it would automatically improve the function response, as well as eliminate the small fluctuation in the derivative response. The basic idea for the pruning method is discussed next.

To get rid of $Type-B$ overfitting, we want to remove neurons whose derivative response is narrower than the distance between training points. Consider the derivative response in Figure 17) and the corresponding buffer zone. If there are no training points within the buffer zone, then that neuron may not contribute to the derivative response at the training points. The neuron can then be further tested for potential removal.

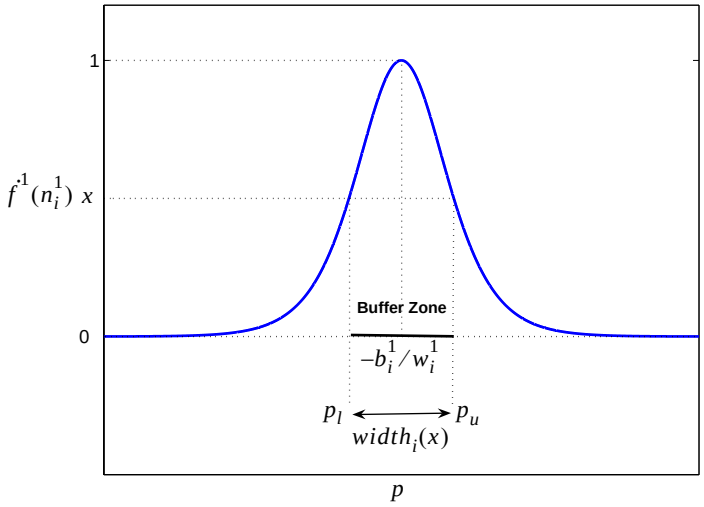

Figure 17) Definition of Buffer Zone

Note that since the combined responses from more than one neuron could also cause $Type-B$ overfitting, the neighbor search procedure (discussed earlier) is also needed. In addition, the process of validating whether the response from a combination of neurons significantly contributes to the derivative fitting is the same as when dealing with $type-A$ overfitting.

In order to proceed with this pruning concept, we first need to define the buffer zone. From Figure 9) and Figure 17), we can see that the width of the buffer zone depends on the magnitude of the first-layer weight w_i^1 and the term $\dot{f}^1(n_i^1)$. That is, the width of

the zone is a function of w_i^1 and $\dot{f}^1(n_i^1)$. To find the relationship, first let's fix the value of $\dot{f}^1(n_i^1)$ at x. From Eq. (279), we have $\dot{f}^1(n_i^1) = x = 1-(a_i^1)^2$. Solve for a_i^1 and recall that $a_i^1 = f^1(n_i^1) = \tanh(w_i^1 p + b_i^1)$. Thus, we obtain

$$\tanh(w_i^1 p + b_i^1) = \pm\sqrt{1-x}. \tag{285}$$

If $w_i^1 > 0$, we have $\tanh(w_i^1 p_u + b_i^1) = \sqrt{1-x}$ and $\tanh(w_i^1 p_l + b_i^1) = -\sqrt{1-x}$. Solve for p_u and p_l. Thus, the width of the buffer zone, $width_i(x)$, is

$$\begin{aligned} width_i(x) &= p_u - p_l \\ &= \frac{\operatorname{atanh}(\sqrt{1-x}) - b_i^1}{w_i^1} - \frac{\operatorname{atanh}(-\sqrt{1-x}) - b_i^1}{w_i^1} \end{aligned} \tag{286}$$

Using the fact that the inverse of hyperbolic tangent sigmoid is an odd function, i.e. $\operatorname{atanh}(-x) = -\operatorname{atanh}(x)$, Eq. (286) becomes

$$width_i(x) = \frac{2\operatorname{atanh}(\sqrt{1-x})}{w_i^1}. \tag{287}$$

If $w_i^1 < 0$, we have $p_u = [\operatorname{atanh}(-\sqrt{1-x}) - b_i^1]/w_i^1$ and $p_l = [\operatorname{atanh}(\sqrt{1-x}) - b_i^1]/w_i^1$. Therefore, the width, $width_i(x)$, is

$$width_i(x) = \frac{-2\operatorname{atanh}(\sqrt{1-x})}{w_i^1}. \tag{288}$$

From Eq. (287) and Eq. (288), we finally obtain

$$width_i(x) = \frac{2\operatorname{atanh}(\sqrt{1-x})}{\left|w_i^1\right|} \; ; \; w_i^1 \neq 0. \tag{289}$$

For multiple inputs, by changing Eq. (285) from $w_i^1 p + b_i^1$ to $({}_i\mathbf{w}^1)^T \mathbf{p} + b_i^1$ and solving, we obtain two hyperplanes making up the boundary of the buffer zone. These two hyperplanes are parallel to the weight center, and they are

$$({}_i\mathbf{w}^1)^T \mathbf{p} + b_i^1 = \operatorname{atanh}(\sqrt{1-x}) \text{ and } ({}_i\mathbf{w}^1)^T \mathbf{p} + b_i^1 = \operatorname{atanh}(-\sqrt{1-x}). \quad (290)$$

To know the width of the buffer zone, we need to know the distance between the two hyperplanes. Therefore, from Eq. (283), the distance between the two hyperplanes in Eq. (290) is

$$width_i(x) = \frac{2\operatorname{atanh}(\sqrt{1-x})}{\left\| {}_i\mathbf{w}^1 \right\|} \; ; \; {}_i\mathbf{w}^1 \neq \mathbf{0} \; . \quad (291)$$

The range of x is $0 \leq x \leq 1$. For pruning, the value of x could be set, for instance, at 0.5.

Once knowing the width of the buffer zone, we will be able to identify which training points are inside or outside the buffer zone, by comparing with the distance from the training points to the weight center. Then, by counting how many training points satisfying the condition:

$$d_{i,q} < \frac{width_i(x)}{2} \; ; \; \forall q, \quad (292)$$

we will be able to tell whether there exist training points inside the zone. If there are none in the zone, the neuron becomes a candidate for $Type-B$ overfitting.

To allow no training points inside the buffer zone is a very strict condition. In practice, it is more promising to allow a small number of training points inside the zone. That is, if the neuron response is fitting only a small fraction of the entire training set, we also

presume it to be a candidate for $Type - B$ overfitting. Therefore, the rule is now to list a neuron as a candidate if the number of training data located inside the buffer zone is less than a small fraction of the total number of training points. For example, the fraction could be set at 1%. Figure 17) illustrates an example showing a few training points inside the buffer zone for a two-dimensional case.

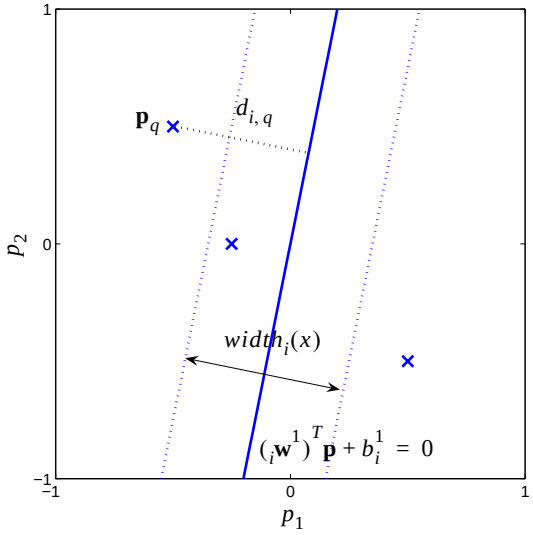

Figure 18) Training points and the buffer zone in a two-dimensional case

In this section, we introduced two types of overfitting produced by the $CFDA$ training algorithms. The general concept of the pruning algorithm that eliminates these types of overfitting was discussed. In the next section, the steps of the algorithm will be provided in more detail.

Pruning Algorithm

In this section, we will provide more details for the steps of the pruning algorithm. The section will be divided into two parts. First, the description of the steps of the pruning algorithm will be provided. Second, the pseudo code of the pruning algorithm will be given.

Description of the pruning algorithm

The steps of the pruning algorithm will be described here. We divide this section into six parts. The first part will provide an overview of the pruning method and how it interfaces with the $CFDA$ training algorithms. The last five parts are the body of the algorithm itself, which consists of:

A. Initialization,

B. Candidate selection,

C. Neighbor search,

D. Contribution check, and

E. Pruning and adjusting.

Overview

Assume we have a neural network NN trained by a $CFDA$ training method. Once the training is converged, the pruning method will be executed. If the pruning method indicates that the network NN is subjected to pruning, we will prune the network. Then, the pruned network will be retrained by the $CFDA$ training algorithm. This process will be repeated until the pruning algorithm indicates no further pruning. The flowchart of this process is illustrated below.

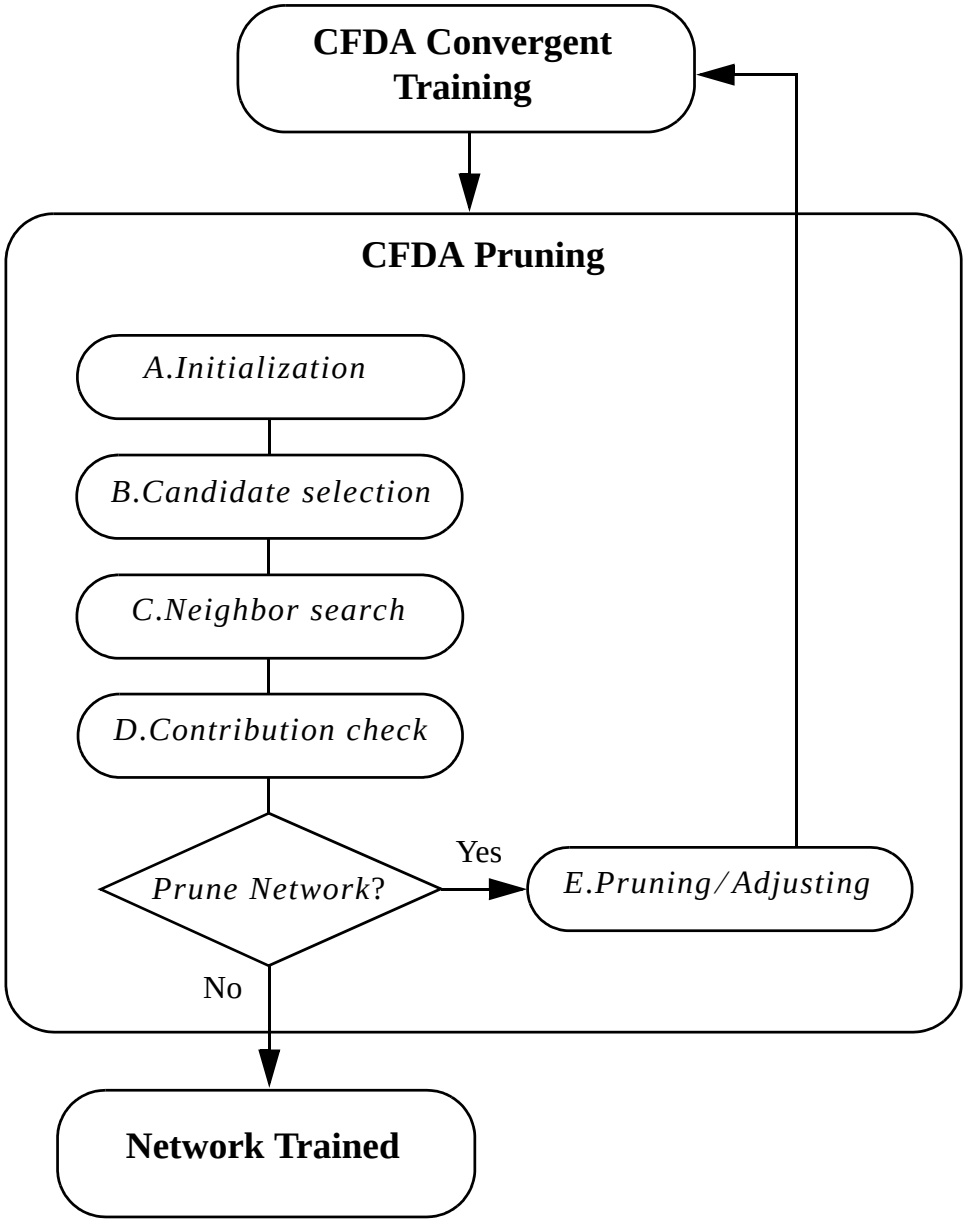

Figure 19) Overview for the pruning algorithm

Next, we will describe the pruning algorithm, consisting of five processes: Initialization, Candidate Selection, Neighbor Search, Contribution Check and Pruning and Adjusting.

A. Initialization

We initialize the pruning algorithm here. Assume we have a $R - S^1 - 1$ network NN, completely trained by a $CFDA$ training algorithm.

1. Initialize the set $\overline{Q} = \{1, 2, \ldots, Q\}$ and the set $\overline{S}^1 = \{1, 2, \ldots, S^1\}$.

2. Specify the thresholds:

a) T_1 is the threshold for the number of training points inside a buffer zone. We choose a value relative to the total number of training points (e.g. 1%: $T_1 = 0.01$).

b) T_2 is the threshold for the distance between the centers of neurons. This is to consider whether a neuron is a neighbor to the other neuron. As previously mentioned, we choose the maximum of the minimum distance between the training points for T_2:

$$T_2 = max\{min(\|\mathbf{p}_{q'} - \mathbf{p}_q\| ; \forall q' \in \overline{Q} \setminus \{q\}) ; \forall q \in \overline{Q}\}. \quad (293)$$

c) $\tilde{T}_2$ is the threshold for the angle between the centers of neurons. In the case of multiple inputs, any two neurons will be neighbors if their centers are parallel (within the tolerance $\tilde{T}_2$) and the distance between them is smaller than the threshold T_2. We choose a fixed value for $\tilde{T}_2$ of one degree: $\tilde{T}_2 = 1$.

d) T_3 is the threshold for the contribution. The measure for contribution is computed as the ratio of the maximum magnitude of the derivative response to the actual derivative magnitude evaluated at the training point. If the ratio is less than T_3, we assume the contribution is not significant. We choose this value to be 10%: i.e. $T_3 = 0.1$.

3. Initialize the set $P = \varnothing$. The set P contains the neurons that will be pruned.

4. Initialize the set $V = \varnothing$. The elements of the set V are the neurons already visited for candidate selection.

5. Go to step **B**.1 .

B. Candidate selection

1. If $\bar{S}^1 \setminus \{P \cup V\} \notin \varnothing$, go to step **B**.2 . Otherwise, go to step **E**.1 . This step is to let the algorithm proceed if there are neurons that have not been visited for candidate selection or marked for being pruned.

2. Find the smallest element i such that $i \in \bar{S}^1 \setminus \{P \cup V\}$. Set $V = V \cup \{i\}$. Note that once neurons are pruned, we will not check them again.

3. Find the training point closest to the weight center: $\tilde{q}_i$ such that $d_{i,\tilde{q}_i} \leq d_{i,q}$, $\forall q$.

4. If $\left|w^2_{1,i}\right| \left\|{}_i\mathbf{w}^1\right\| > \left\|\partial a^2_{1,\tilde{q}_i} / \partial \mathbf{p}^T_{\tilde{q}_i}\right\|$, mark neuron i and go to step **C**.1 . Otherwise, go to step **B**.5 .

5. Compute $\left|w^2_{1,i}\right| \left\|{}_i\mathbf{w}^1\right\|$ and $width_i(x)$ using Eq. (291).

6. Compute the distance $d_{i,q}$ using Eq. (282), $\forall q$.

7. Count how many training points are inside the buffer zone using Eq. (292). Store the number in bz_i.

8. If $bz_i \leq T_1$, mark neuron i as a candidate and go to step **C**.1 . Otherwise, go to step **B**.1 .

C. Neighbor search

Once neuron i is selected to be a candidate, perform the following steps.

1. Initialize the set $NB_i = \varnothing$. This set contains the neighbors of the candidate neuron i.

2. Find its neighbors:

a) For a single input, find neurons whose centers are within the maximum of the minimum distance between neighboring training points from the candidate center. That is, find neuron l such that

$$\left| \frac{b_l^1}{w_l^1} - \frac{b_i^1}{w_i^1} \right| \le T_2, \text{ for all } l \in \bar{S}^1 \setminus \{i\}. \tag{294}$$

b) For multiple inputs, neighbors are those neurons with parallel centers (within a tolerance) and close to the candidate center. The angle between neuron l and i is computed by:

$$angle_{l,i} = \frac{1}{\pi} \text{acos}\left(\frac{\langle {}_l\mathbf{w}^1, {}_i\mathbf{w}^1 \rangle}{\left\| {}_l\mathbf{w}^1 \right\| \left\| {}_i\mathbf{w}^1 \right\|} \right), \tag{295}$$

where $\langle {}_l\mathbf{w}^1, {}_i\mathbf{w}^1 \rangle$ is the inner product of ${}_l\mathbf{w}^1$ and ${}_i\mathbf{w}^1$. The distance between two hyperplanes can be computed from Eq. (283) and Eq. (284). Thus, we have two cases. First, find neurons whose centers are pointed in the same direction as and close to the candidate center. That is, find neuron l such that

$$angle_{l,i} \le \tilde{T}_2 \text{ and } \left| \frac{b_l^1}{\left\| {}_l\mathbf{w}^1 \right\|} - \frac{b_i^1}{\left\| {}_i\mathbf{w}^1 \right\|} \right| \le T_2, \text{ for all } l \in \bar{S}^1 \setminus \{i\}. \tag{296}$$

Second, find neurons whose centers are pointed in the opposite direction and close to the candidate center. In other words, find neuron l such that

$$\pi - angle_{l,i} \le \tilde{T}_2 \text{ and } \left| \frac{b_l^1}{\|{}_l\mathbf{w}^1\|} + \frac{b_i^1}{\|{}_i\mathbf{w}^1\|} \right| \le T_2, \text{ for all } l \in \bar{S}^1 \setminus \{i\}. \quad (297)$$

For any neuron $l \in \bar{S}^1 \setminus \{i\}$ satisfying the condition in Eq. (296) or Eq. (297), go to step **C**.3.

3. Put l into the set NB_i. Then, go to step **D**.1.

D. Contribution check

Assume neuron i and its neighbors NB_i are established (note: NB_i could be an empty set). We need to validate whether their response significantly contributes to the derivative fitting. Perform the following steps.

1) Initialize k with n_{NB_i} (the number of elements in NB_i): $k = n_{NB_i}$. Initialize the combinadic index $j = 0$. Initialize $quit = 0$.

2. Generate elements of $P(NB_i)$, starting with the combinadic with highest number of elements: $P(NB_i)_{k,j} = psgen(NB_i, k, j)$. For the details of this step, see Appendix A.

3. Include neuron i into the set $P(NB_i)_{k,j}$: $G_i = \{i\} \cup P(NB_i)_{k,j}$. Now, we want to validate the response contribution of the neurons in the set G_i. Perform the following steps:

a) Create a copy of the neural network: $\tilde{NN} = NN$.

b) Set the weights of the neurons not in G_i for the network $\tilde{NN}$ to be zero:

$({}_l\tilde{\mathbf{w}}^1)^T = \mathbf{0}$, $\tilde{b}_l^1 = 0$, $\tilde{w}_{1,l}^2 = 0$ and $\tilde{b}^2 = 0$, for all $l \in \bar{S}^1 \setminus G_i$.

c) Compute the magnitude of the derivative response from the network $\tilde{NN}$ over the training data: $\left\| \partial \tilde{a}_{1,q}^2 / \partial \mathbf{p}_q^T \right\|$, for all $q \in \bar{Q}$.

d) Find the training point yielding the maximum magnitude of the derivative response: $q^* = \arg\max_q (\left\| \partial \tilde{a}_{1,q}^2 / \partial \mathbf{p}_q^T \right\| \ ; \forall q \in \bar{Q})$.

e) Compute the ratio of the response magnitude to the magnitude of the actual derivative at the training point: $r = \left\| \partial \tilde{a}_{1,q^*}^2 / \partial \mathbf{p}_{q^*}^T \right\| / \left\| \partial g_{q^*} / \partial \mathbf{p}_{q^*}^T \right\|$.

f) If $r < T_3$, set $P = P \cup G_i$, quit this process (set $quit = 1$) and go to step **B**.1 . Otherwise, go to step g). Note that this step means if $r < T_3$ (the contribution of neurons in G_i is not significant), the neurons in G_i are included in the set P). If $r \geq T_3$ (the contribution of neurons in G_i is significant), we will form another combination of neurons.

g) If $j < \binom{n_{NB_i}}{k} - 1$, set $j = j + 1$ and go to step **D**.2 (we will create another combination of neurons, where the number of neurons in the new combination will be the same, i.e. k). Otherwise, go to step h).

h) If $k > 0$, set $k = k - 1$ and go to step **D**.2 (we will create another combination of neurons, but the number of neurons in the new combination is reduced by one). Otherwise, quit the process by setting $quit = 1$ and go to step **B**.1 .

E. Pruning & Adjusting

Once the set P, which contains the neurons to be pruned, is completely formed from step **A** to **D**, one more calculation is needed. This is to calculate the function response of the neurons stored in P. From Eq. (278), we can rewrite the derivative response of the network evaluated at the training point $\mathbf{p}_q$ as:

$$\frac{\partial a^2_{1,q}}{\partial p_{r,q}} = \sum_{i \notin P} \frac{\partial y_{i,q}}{\partial p_{r,q}} + \sum_{j \in P} \frac{\partial y_{j,q}}{\partial p_{r,q}}. \tag{298}$$

This corresponds to the function response:

$$a^2_{1,q} = \sum_{i \notin P} y_{i,q} + \sum_{j \in P} y_{j,q} + b^2. \tag{299}$$

The notation $y_{j,q}$ and $\partial y_{j,q} / \partial p_{r,q}$ denotes the term $y_{j,q}$ and $\partial y_{j,q} / \partial p_{r,q}$ evaluated at the input point $\mathbf{p}_q$, respectively. After the neurons in P are pruned, it will not have a significant impact on the derivative response $\partial a^2_{1,q} / \partial p_{r,q}$ for all $q \in \overline{Q}$ due to the contribution check (i.e. $\sum_{j \in P} \partial y_{j,q} / \partial p_{r,q}$ for all $q \in \overline{Q}$ is small). However, it may dramatically change the network's function response $a^2_{1,q}$ for some $q \in \overline{Q}$, since $\sum_{j \in P} y_{j,q}$ for some $q \in \overline{Q}$ could be large. An simple example is when the set P contains a neuron with very large step in its function response and its center is outside the training set. Pruning this neuron would produce no impact on the derivative response, but it would cause a shift up/down to the entire function response over the training set.

To compensate for the response $\sum_{j \in P} y_j$ (that will disappear with the neurons in P), we compute its average value computed over the entire training set: i.e.

$$b_P = \frac{1}{Q} \sum_{q=1}^{Q} \sum_{j \in P} y_{j,q} \ , \tag{300}$$

Then, the compensation can be done by inserting b_P to the second-layer bias b^2.

Now, we are ready to provide the process of pruning and adjusting.

1. If $P = \varnothing$, quit the pruning algorithm and return the network NN as the final network. Otherwise, go to step $\mathbf{E}.2$.

2. Prune the neurons contained in P from NN.

3. Set the second-layer bias of the network to $b^2 = b^2 + b_P$. Quit the pruning algorithm and return the pruned network for a $CFDA$ retraining.

We will provide the pseudo code for the algorithm in the next section.

Pseudo Code

We will provide the pseudo code in this section. The variables in the pseudo code were already introduced in the previous section.

A. INITIALIZATION

Initialize $\bar{Q} = \{1, 2, \ldots, Q\}$ and $\bar{S}^1 = \{1, 2, \ldots, S^1\}$. Set $P = \varnothing$ and $V = \varnothing$. Specify the thresholds: T_1, T_2, $\tilde{T}_2$, T_3 and x.

B. CANDIDATE SELECTION

While $\bar{S}^1 \setminus \{P \cup V\} \notin \varnothing$

Find the smallest element i such that $i \in \bar{S}^1 \setminus \{P \cup V\}$.

Compute $width_i(x) = 2\,\text{atanh}(\sqrt{1-x}) / \left\| {}_i\mathbf{w}^1 \right\|$.

Compute $d_{i,q} = \left| ({}_i\mathbf{w}^1)^T \mathbf{p}_q + b_i^1 \right| / \left\| {}_i\mathbf{w}^1 \right\|$; $\forall q \in \bar{Q}$.

Set bz_i to be the number of $q \in \bar{Q}$ satisfying $d_{i,q} < width_i(x)/2$.

Find $\tilde{q}_i \in \bar{Q}$ such that $d_{i,\tilde{q}_i} \le d_{i,q}$, $\forall q \in \bar{Q}$.

If $\left| w_{1,i}^2 \right| \left\| {}_i\mathbf{w}^1 \right\| > \left\| \partial a_{1,\tilde{q}_i}^2 / \partial \mathbf{p}_{\tilde{q}_i}^T \right\|$ or $bz_i \le T_1$

C. NEIGHBOR SEARCH

Initialize $NB_i = \varnothing$.

For all $l \in \bar{S}^1 \setminus \{i\}$

If $R = 1$

If $\left| \frac{b_l^1}{w_l^1} - \frac{b_i^1}{w_i^1} \right| \le T_2$, add l to the set NB_i.

Else

Compute $angle_{l,i} = \frac{1}{\pi}\text{acos}\left(\frac{\langle {}_l\mathbf{w}^1, {}_i\mathbf{w}^1 \rangle}{\left\| {}_l\mathbf{w}^1 \right\| \left\| {}_i\mathbf{w}^1 \right\|} \right)$.

If $angle_{l,i} < \tilde{T}_2$ and $\left| \frac{b_l^1}{\left\| {}_l\mathbf{w}^1 \right\|} - \frac{b_i^1}{\left\| {}_i\mathbf{w}^1 \right\|} \right| \le T_2$

Add l to the set NB_i.

ElseIf $\pi - angle_{l,i} < \tilde{T}_2$ and $\left| \frac{b_l^1}{\left\| {}_l\mathbf{w}^1 \right\|} + \frac{b_i^1}{\left\| {}_i\mathbf{w}^1 \right\|} \right| \le T_2$

Add l to the set NB_i.

EndIf $\tilde{T}_2$ and T_2

EndIf R

EndFor l

D. CONTRIBUTION CHECK

Initialize $k = n_{NB_i}$, $j = 0$ and $quit = 0$.

While $quit \neq 1$ and $k \geq 0$

While $quit \neq 1$ and $j \leq \binom{n_{NB_i}}{k} - 1$

Set $P(NB_i)_{k,j} = psgen(NB_i, k, j)$.

Set $G_i = \{i\} \cup P(NB_i)_{k,j}$.

Create a network $\tilde{NN} = NN$.

For all $l \in \bar{S}^1 \setminus G_i$

Set $({}_l\tilde{\mathbf{w}}^1)^T = \mathbf{0}$, $\tilde{b}_l^1 = 0$, $\tilde{w}_{1,l}^2 = 0$ and $\tilde{b}^2 = 0$.

EndFor l

Compute $\left\| \partial \tilde{a}_{1,q}^2 / \partial \mathbf{p}_q^T \right\|$, $\forall q \in \bar{Q}$.

Set $q^* = \arg\max_q \left(\left\| \partial \tilde{a}_{1,q}^2 / \partial \mathbf{p}_q^T \right\| ; \forall q \in \bar{Q} \right)$.

Compute $r = \left\| \partial \tilde{a}_{1,q^*}^2 / \partial \mathbf{p}_{q^*}^T \right\| / \left\| \partial g_{q^*} / \partial \mathbf{p}_{q^*}^T \right\|$.

If $r < T_3$

Set $P = P \cup G_i$ and $quit = 1$.

EndIf r

Set $j = j + 1$.

EndWhile $quit$ and j

Set $k = k - 1$.

EndWhile $quit$ and k

EndIf T_1

Set $V = V \cup \{i\}$.

EndWhile $\bar{S}^1 \setminus \{P \cup V\}$

E. PRUNING & ADJUSTING

Compute $b_P = \frac{1}{Q} \sum_{q=1}^{Q} \sum_{j \in P} y_{j,q}$.

Create a network $\tilde{NN} = NN$.

Prune the network $\tilde{NN}$, using P.

Set $\tilde{b}_2 = b_2 + b_P$.

Return $\tilde{NN}$.

Summary

In this chapter, we introduced two new types of overfitting, i.e. $Type-A$ and $Type-B$. These types of overfitting are produced by the neural networks trained by a $CFDA$ training algorithm. For $Type-A$ overfitting, the network produces accurate function and derivative response over the entire training set. However, at some points between training points, the network responses (both function and derivative) are inaccurate. We demonstrated that $Type-A$ overfitting comes from two or more neurons whose centers are close together and whose responses cancel over the entire training set. However, the response cancellation does not occur betweeen some training points. We proposed a way to detect this problem by first selecting neurons with high weight products. If the weight product of a neuron is higher than the magnitude of the actual derivative evaluated at the training point closest to the neuron center, the neuron is marked as a candidate for producing $Type-A$ overfitting.

For $Type-B$ overfitting, the network produces accurate derivative response over the training set. However, the function response has small errors on some training data. This problem is caused by a local minimum in the $CFDA$ training surface. We proposed a way to detect a neuron producing this type of overfitting by counting how many training points are inside the neuron's buffer zone. If there are less than a small number of training points in the zone, the neuron is marked as a candidate for generating $Type-B$ overfitting.

Once a candidate is selected, we search for its neighbors. The neighbors are defined as those neurons whose centers are close to the candidate center within the maximum of the minimum distance between training points. After locating the candidate's neighbors, com-

binations of neurons from the set are formed. The derivative response from these combinations are checked to see whether or not their contribution to the derivative fitting is significant. We measure the contribution by computing the ratio of the maximum magnitude of the derivative response (computed at training points) to the magnitude of the true derivative evaluated at that training point. If the ratio is smaller than the threshold (we chose 0.1), we define the contribution as insignificant. Therefore, if a combination of neurons produces insignificant contribution, this combination is pruned. After pruning the network, we showed that the second-layer bias of the network must be adjusted to compensate for the function response of the pruned neurons.

In the appendix, we provided the steps of an algorithm to form combinations of neurons. In this chapter, we described the interface of the algorithm to the $CFDA$ training methods, as well as the steps of the pruning algorithm. Finally, the pseudo code of the pruning algorithm was provided.

CHAPTER 7

TRAINING RESULTS ON SIMPLE PROBLEMS

Introduction

This chapter serves four purposes. First, it describes the procedure for evaluating and comparing the approximation accuracy of various training algorithms, for simple problems. Comparisons will be made among four groups of algorithms:

1. The standard training algorithms introduced in Chapter 2: $BFGS-ES$, $LM-ES$, and $GNBR$.
2. The gradient-based $CFDA$ method (Chapter 4),
3. The $CFDA$ method with Levenberg-Marquardt (Chapter 5), and
4. The $CFDA$ methods (i.e. gradient-based and Levenberg-Marquardt) with the pruning algorithm (Chapter 6).

Second, the choice of the parameter ρ in the $CFDA$ method will be analyzed. Third, the approximation accuracy obtained from each training algorithm for these simple examples will be shown and compared. Finally, some examples illustrating the $CFDA$ overfitting and demonstrating how the pruning algorithm eliminates the overfitting will be shown.

Evaluation Procedure

This section will describe the procedure for evaluating the approximation accuracy of each training algorithm. It consists of two parts. First, the problem definitions will be introduced. Second, the simulation steps will be presented.

Problem definition

We will use neural networks to approximate the four different functions (and their derivatives) defined in Table 7. The table also shows the input ranges for generating the training and testing data sets. The table also shows the number of training and testing data used for each function. Each data set contains function inputs, the corresponding function outputs and the associated first-order derivatives,

Problem	Function	Training Region	No. of Training	Test Region	No. of Testing
1	$\sin(0.5\pi p)$	$-1 \le p \le 1$	40	$-0.8 \le p \le 0.8$	321
2	$1.2e^{-3p}\sin(8\pi p)$	$0 \le p \le 1$	200	$0.1 \le p \le 0.9$	801
3	$\cos(4\pi p)$	$-1 \le p \le 1$	200	$-0.8 \le p \le 0.8$	1601
4	$\frac{5\sin(10\sqrt{p_1^2+p_2^2})}{10\sqrt{p_1^2+p_2^2}}$	$-1 \le p_1 \le 1$, $-1 \le p_2 \le 1$	300	$-0.8 \le p_1 \le 0.8$, $-0.8 \le p_2 \le 0.8$	100K

Table 7 Description of test functions

The network structure used for each problem is defined in Table 8. Note that Figure 20) illustrates the graph of the functions defined in Table 7.

Problem	Network Structure
1	1 – 10 – 1
2	1 – 60 – 1
3	1 – 40 – 1
4	2 – 100 – 1

Table 8 Network structure for each problem

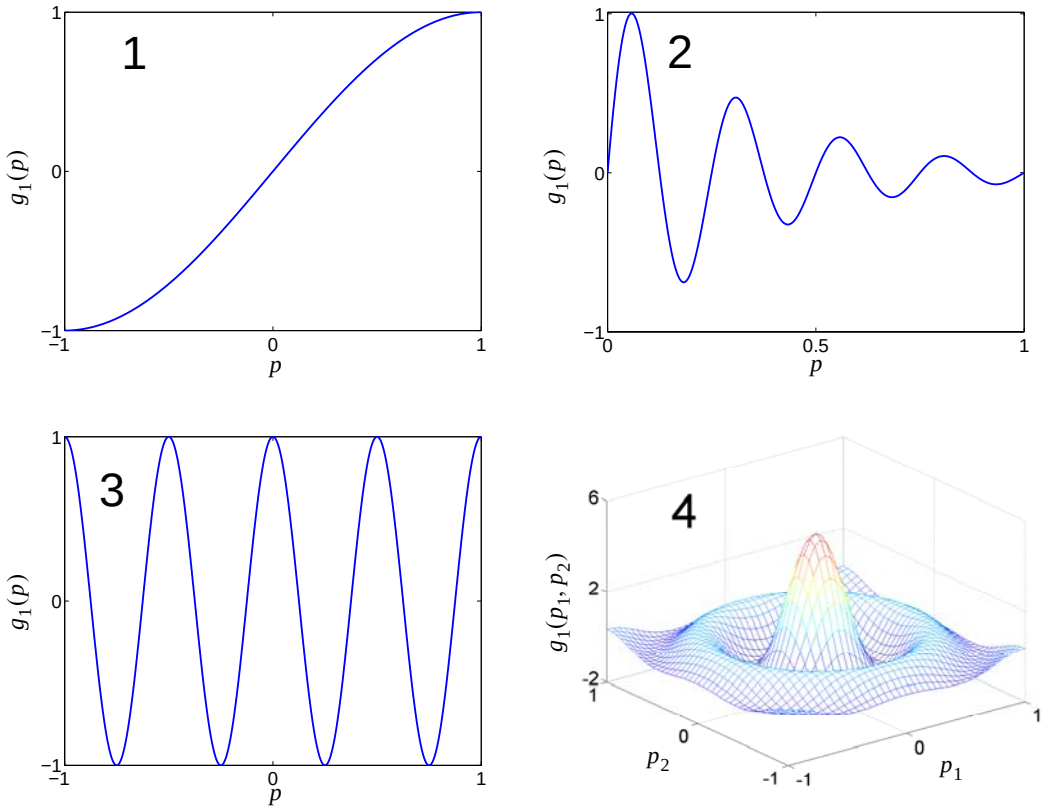

Figure 20) Graph of the four test functions

Experiment design

The following steps will be repeated on each test problem for every training algorithm.

Simulation Steps

1. Randomly generate the testing data.
2. Randomly generate the training data.
3. If the training algorithm uses the early stopping technique, randomly divide the training data in the previous step into two parts: training and validation. The new training set will contain 80% of the data, and the validation set will contain 20% of the data. If the training algorithm does not use early stopping, the training set remains the same. Note that the data sets will be reused for every training algorithm.
4. Create a neural network with the structure defined in Table 8. The network parameters are initialized by Nguyen-Widrow algorithm [NgWi90]. Note that this initialized network will be reused for every training algorithm.
5. Train the network until the algorithm is terminated.
6. Compute Root Mean Square Error (RMSE) for the function approximation, the first derivative approximation and the second derivative approximation on both the training and test set. We denote

$$RMSE_F \equiv \sqrt{\frac{1}{QS^M} \sum_{q=1}^{Q} \sum_{k=1}^{S^M} e_{k,q}^2}, \tag{301}$$

$$RMSE_D \equiv \sqrt{\frac{1}{QS^M R} \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \sum_{r=1}^{R} \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \right)^2}, \text{ and} \tag{302}$$

$$RMSE_{D^2} \equiv \sqrt{\frac{1}{QS^M N} \sum_{q=1}^{Q} \sum_{k=1}^{S^M} \sum_{r'=r}^{R} \sum_{r=1}^{R} \left\{ \frac{\partial}{\partial p_{r',q}} \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \right) \right\}^2}, \tag{303}$$

where $N = (R+1)R/2$. The notation $RMSE_F$ is the function RMSE, $RMSE_D$ is the first-order derivative RMSE, $RMSE_{D^2}$ is the second-order derivative RMSE, and Q is the number of examples in a data set. The calculation of $\frac{\partial}{\partial p_{r',q}} \left(\frac{\partial e_{k,q}}{\partial p_{r,q}} \right)$ requires the computation of $\frac{\partial}{\partial p_{r',q}} \left(\frac{\partial a_{k,q}}{\partial p_{r,q}} \right)$. This is presented in Appendix B.

7. Repeat step 2) to 6) K times for Monte Carlo simulation.
8. Report the sample median statistic of $RMSE_F$ and $RMSE_D$, denoted $RMSE_F^{md}$ and $RMSE_D^{md}$ respectively, on both training and testing sets, among the K Monte Carlo samples. Report the sample median statistic of $RMSE_{D^2}$, denoted $RMSE_{D^2}^{md}$ on testing set, among the K Monte Carlo samples.

The median statistic of $RMSE_F$ and $RMSE_D$ is the measure of the approximation accuracy for each training algorithm. Note that the median statistic is used, rather than the average, because it is less sensitive to outliers (e.g. poor approximation because the training process is stuck in a local minimum). We set $K = 25$ for Problem 1, 2 and 3, and $K = 10$ for Problem 4. We used a lower number for Problem 4 because of the significant computation time involved.

For the gradient-based $CFDA$ method, we selected Quasi-Newton $BFGS$ optimization (with backtracking line search). See Chapter 2 for more details of the $BFGS$ optimization. We refer to this training method as $CFDA-BFGS$, while $CFDA-LM$ is the $CFDA$ method with Levenberg-Marquardt optimization. In the $CFDA$ methods, early stopping is not used. This is because standard overfitting does not occur when fitting both a function and its derivatives at the training points. We will discuss this in more detail later in this document. For the $CFDA$ methods with pruning, we denote $CFDA-BFGS_p$ and $CFDA-LM_p$ to indicate $CFDA-BFGS$ and $CFDA-LM$ with the pruning algorithm, respectively.

For problems 1, 2 and 3, we will evaluate the approximation accuracy for $BFGS-ES$, $LM-ES$, $GNBR$, $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$. However, for problem 4, the evaluation will be performed only for $GNBR$, $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$. In the next section, the parameter ρ in the $CFDA$ performance index will be discussed.

Parameters in CFDA

Recall that the $CFDA$ performance index contains a parameter ρ:

$$\begin{aligned} J &= J_f + \rho J_d \\ &= \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\{a_{k,q}-g_{k,q}\}^2 + \frac{\rho}{Q_d S_d^M R_d}\sum_{q=1}^{Q}\sum_{k=1}^{S^M}\sum_{r=1}^{R}\varphi_{k,r}(q)\left(\frac{\partial a_{k,q}}{\partial p_{r,q}}-\frac{\partial g_{k,q}}{\partial p_{r,q}}\right)^2. \end{aligned} \quad (304)$$

In this section, we will discuss how the value of ρ is selected. First, we define ρ to be:

$$\rho = \frac{\lambda}{\eta^2}\ , \tag{305}$$

where

$$\eta = \frac{max(|\partial g_{k,q} / \partial p_{r,q}|; \forall k, \forall r, \forall q)}{max(|g_{k,q}|; \forall k, \forall q)}. \tag{306}$$

The terms $g_{k,q}$ and $\partial g_{k,q} / \partial p_{r,q}$ in Eq. (306) are the target function values and the first-order derivative values in the training set. In this way, η will account for the scale difference between the target function values and the first-order derivative values. The value of λ will then be varied to see its impact on the approximation accuracy. The following figures show the impact of λ on the approximation accuracy in RMSE for problem 1, 2 and 3 using the $CFDA-BFGS$ algorithm.

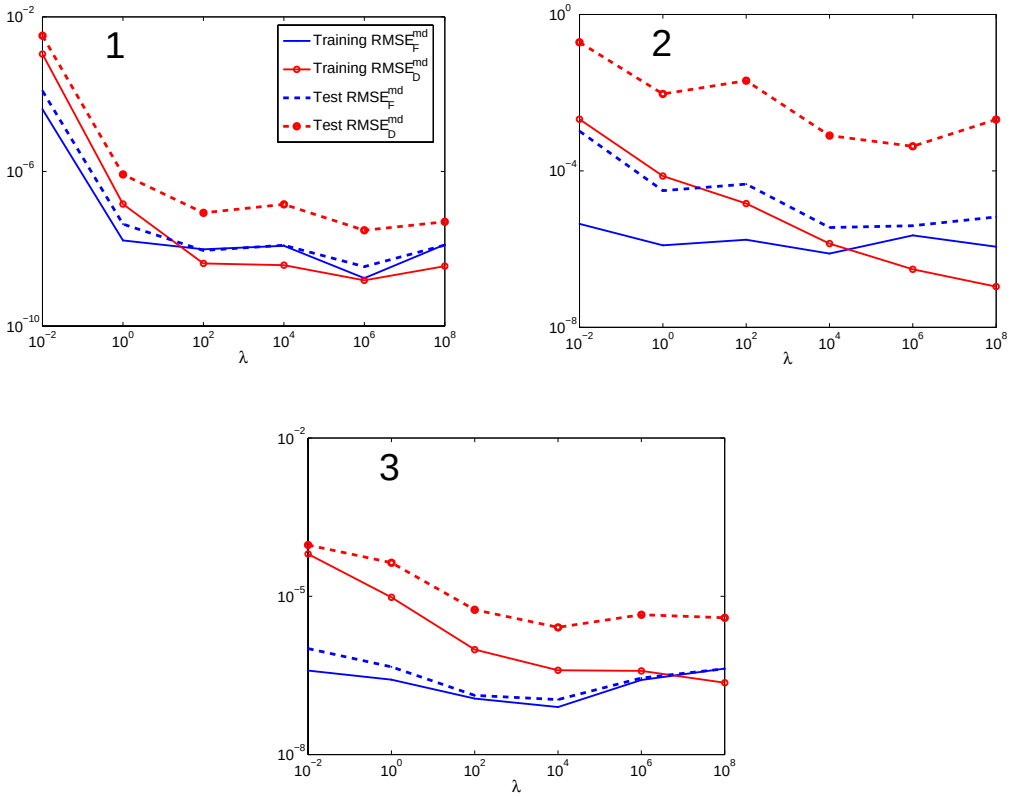

Figure 21) Impact of λ on the approximation accuracy

From these results, we determined that a robust value for λ would be 10^4. This value will be used for all of the subsequent cases. In the next section, the simulation results showing the approximation accuracy in RMSE on each problem for each training algorithm will be illustrated.

Simulation Results

We divide this section into two parts. The approximation accuracy in RMSE for each problem obtained from every training algorithm is shown in the first part. The second part is dedicated to illustrating how the *CFDA* pruning algorithm works.

Approximation accuracy

We show the approximation accuracy for two sets of K Monte Carlo samples in this section. We used the first set to study the thresholds for the $CFDA$ pruning algorithm. Then, the thresholds were applied to the second set to validate its efficiency. Note that, for $CFDA-BFGS_p$ and $CFDA-LM_p$, the retraining process (i.e. after pruning) is terminated either when the $CFDA$ performance index is convergent or it reaches the same level achieved before pruning.

First Set

Table 9 to Table 12 show the approximation accuracy obtained by each training algorithm. Note again that, for the $CFDA$ methods, the value of λ is set at 10^4. Note also that RMSE values reported in the tables represent the median statistic among the K Monte Carlo samples. The thresholds for the pruning algorithm are $T_1 = 0.01$, $\tilde{T}_2 = 1$, $T_3 = 0.1$ and $x = 0.5$.Table 13 shows the median number of neurons of the final pruned networks for each case. The values following the " / " are the numbers of neuron in the networks before pruning.

Training Algorithm	Problem 1				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$BFGS-ES$	4.29E-06	1.12E-03	7.38E-03	1.74E-02	4.54E-01
$LM-ES$	2.48E-06	8.90E-04	6.42E-03	1.74E-02	3.88E-01
$GNBR$	3.37E-07	1.91E-06	2.38E-05	3.45E-05	7.52E-04
$CFDA-BFGS$	1.19E-08	1.23E-08	3.75E-09	1.41E-07	6.02E-06
$CFDA-BFGS_p$	2.86E-09	2.89E-09	3.64E-09	1.60E-08	2.69E-07
$CFDA-LM$	7.98E-09	8.09E-09	2.70E-09	7.59E-08	3.51E-06
$CFDA-LM_p$	3.67E-09	3.83E-09	2.92E-09	1.73E-08	2.58E-07

Table 9 Approximation accuracy on problem 1 (First set)

Training Algorithm	Problem 2				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$BFGS-ES$	5.76E-04	1.94E-02	2.45E+00	3.24E+00	9.23E+02
$LM-ES$	1.14E-05	1.16E-02	1.09E+00	1.94E+00	5.73E+02
$GNBR$	3.68E-07	4.19E-04	4.02E-02	7.79E-02	1.95E+01
$CFDA-BFGS$	7.70E-07	3.57E-06	1.38E-06	8.01E-04	3.43E-01
$CFDA-BFGS_p$	6.42E-08	6.47E-08	1.16E-06	1.94E-06	3.20E-04
$CFDA-LM$	2.03E-07	2.92E-07	1.28E-07	4.55E-05	1.33E-02
$CFDA-LM_p$	5.68E-08	5.64E-08	5.60E-07	1.49E-06	2.08E-04

Table 10 Approximation accuracy on problem 2 (First set)

Training Algorithm	Problem 3				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$BFGS-ES$	6.41E-05	2.28E-03	4.96E-02	1.59E-01	1.83E+01
$LM-ES$	6.72E-06	2.09E-04	5.67E-03	1.12E-02	1.11E+00
$GNBR$	6.33E-07	7.55E-06	3.83E-04	5.51E-04	5.94E-02
$CFDA-BFGS$	7.94E-08	1.10E-07	3.96E-07	2.58E-06	1.98E-04
$CFDA-BFGS_p$	7.14E-08	7.05E-08	3.94E-07	7.42E-07	6.97E-05
$CFDA-LM$	6.20E-08	7.42E-08	1.41E-07	9.64E-07	9.45E-05
$CFDA-LM_p$	3.03E-08	2.97E-08	1.80E-07	3.50E-07	2.85E-05

Table 11 Approximation accuracy on problem 3 (First set)

Training Algorithm	Problem 4				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	1.93E-06	4.26E-04	3.19E-03	5.12E-03	8.72E-02
$CFDA-BFGS$	1.73E-04	2.06E-04	4.55E-04	2.54E-03	3.24E-01
$CFDA-BFGS_p$	9.36E-05	1.12E-04	4.30E-04	1.16E-03	2.31E-02
$CFDA-LM$	2.39E-05	3.01E-05	5.47E-05	4.64E-04	1.35E-02
$CFDA-LM_p$	2.03E-05	2.61E-05	6.21E-05	3.37E-04	6.71E-03

Table 12 Approximation accuracy on problem 4 (First set)

Training Algorithm	S^1			
	Problem 1	Problem 2	Problem 3	Problem 4
$CFDA-BFGS_p$	8/10	26/60	35/40	92/100
$CFDA-LM_p$	8/10	29/60	31/40	96/100

Table 13 Number of neurons after pruning (First set)

From Table 9 to Table 11, we can see that among the three *standard* training algorithms, the $GNBR$ method yielded best approximation accuracy on both training and test

set. However, both $CFDA$ methods provided lower function errors and much lower first derivative errors on the test set than any of the standard methods. The results show that the $CFDA$ methods provide improved generalization capabilities, without the need of a validation set. The $CFDA-LM$ yielded the smallest test error on both function and the derivatives, followed by $CFDA-BFGS$ and $GNBR$.

It is interesting to note that, the $CFDA$ methods yielded much smaller second derivative errors than $GNBR$ in every problem, except problem 4. In problem 4, the second derivative errors obtained from $CFDA-BFGS$ were larger than $GNBR$. However, after pruning the networks, the errors became smaller. For $CFDA-BFGS_p$ and $CFDA-LM_p$, the approximation accuracy was better on both function and derivatives (i.e. first and second orders) than the regular $CFDA$ methods. However, we will validate these results again, using the second set. Among the three standard training algorithms, $GNBR$ consistently provided the smallest approximation error in every problem. Therefore, only $GNBR$ will be used to compare against the two $CFDA$ methods (with and without the pruning algorithm) for the remainder of this chapter.

Second Set

For each problem, a new set of K Monte Carlo samples is simulated. The purpose is to validate the efficiency of the thresholds of the $CFDA$ pruning algorithm we specified in the first set. Table 14 to Table 17 show the approximation accuracy obtained for each

training algorithm. Table 18 shows the median number of neurons of the final pruned networks for each case.

Training Algorithm	Problem 1				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	6.72E-08	3.39E-07	4.45E-06	6.08E-06	1.71E-04
$CFDA-BFGS$	1.98E-09	3.58E-09	1.47E-09	2.55E-08	7.51E-07
$CFDA-BFGS_p$	7.58E-10	7.91E-10	1.75E-09	5.74E-09	1.21E-07
$CFDA-LM$	3.67E-09	4.37E-09	1.59E-09	4.23E-08	1.00E-06
$CFDA-LM_p$	1.02E-09	1.17E-09	1.57E-09	1.07E-08	1.73E-07

Table 14 Approximation accuracy on problem 1 (Second set)

Training Algorithm	Problem 2				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	7.18E-07	2.11E-04	8.75E-03	3.40E-02	7.11E+00
$CFDA-BFGS$	2.38E-06	6.35E-06	3.84E-06	5.43E-04	1.80E-01
$CFDA-BFGS_p$	1.96E-07	2.40E-07	2.87E-06	7.43E-06	9.48E-04
$CFDA-LM$	1.42E-07	2.06E-07	1.01E-07	2.81E-05	8.21E-03
$CFDA-LM_p$	2.16E-08	2.15E-08	3.17E-07	8.72E-07	1.22E-04

Table 15 Approximation accuracy on problem 2 (Second set)

Training Algorithm	Problem 3				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	7.98E-07	2.77E-06	1.95E-04	2.26E-04	2.04E-02
$CFDA-BFGS$	3.22E-07	3.44E-07	8.24E-07	4.62E-06	3.68E-04
$CFDA-BFGS_p$	1.73E-07	1.76E-07	8.24E-07	2.53E-06	1.75E-04
$CFDA-LM$	8.66E-08	8.63E-08	1.06E-07	1.91E-06	2.14E-04
$CFDA-LM_p$	2.24E-08	2.13E-08	1.12E-07	2.87E-07	2.37E-05

Table 16 Approximation accuracy on problem 3 (Second set)

Training Algorithm	Problem 4				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	1.92E-06	3.54E-04	3.52E-03	5.01E-03	7.58E-02
$CFDA-BFGS$	1.39E-04	1.66E-04	3.60E-04	3.13E-03	2.40E-01
$CFDA-BFGS_p$	7.71E-05	1.04E-04	3.47E-04	1.21E-03	2.75E-02
$CFDA-LM$	1.15E-05	1.29E-05	2.80E-05	1.80E-04	4.16E-03
$CFDA-LM_p$	1.12E-05	1.29E-05	2.66E-05	1.73E-04	3.86E-03

Table 17 Approximation accuracy on problem 4 (Second set)

Training Algorithm	S^1			
	Problem 1	Problem 2	Problem 3	Problem 4
$CFDA-BFGS_p$	9/10	26/60	36/40	93/100
$CFDA-LM_p$	8/10	30/60	32/40	98/100

Table 18 Number of neurons after pruning (Second set)

From Table 14 to Table 17, we can see that the results are consistent with the results in the first set. More importantly, the results showed that the pruning thresholds also worked well for this new set. The methods $CFDA-BFGS_p$ and $CFDA-LM_p$ consistently provided lower approximation errors on both function and derivatives than the reg-

ular $CFDA$ methods. In addition, for problem 4, although $CFDA-BFGS$ produced higher second derivative errors than $GNBR$, the pruning algorithm eventually made the errors smaller.

We show the RMSEs for each Monte Carlo sample in Figure 22) to Figure 25), i.e. for $GNBR$, $CFDA-LM$ and $CFDA-LM_p$ in problem 1 to 3, and for $GNBR$, $CFDA-BFGS$ and $CFDA-BFGS_p$ in problem 4. We will also show the second derivative errors for each Monte Carlo network in problem 1 and problem 4.

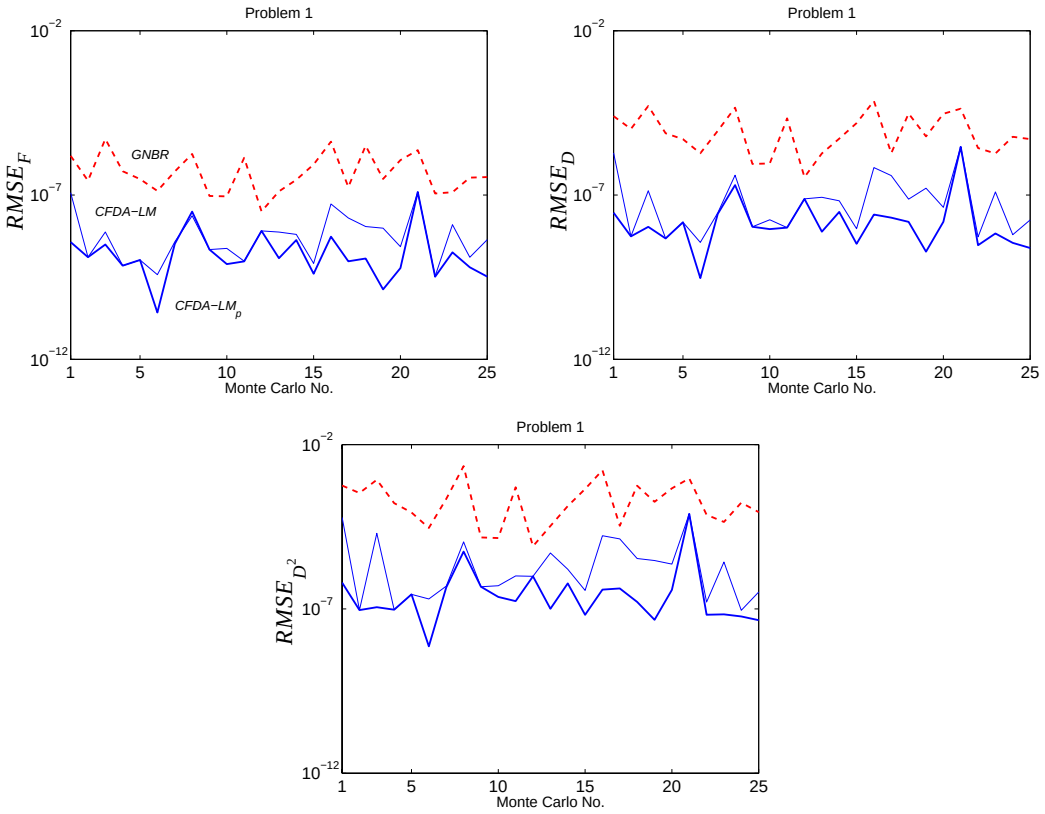

Figure 22) Error comparison for problem 1

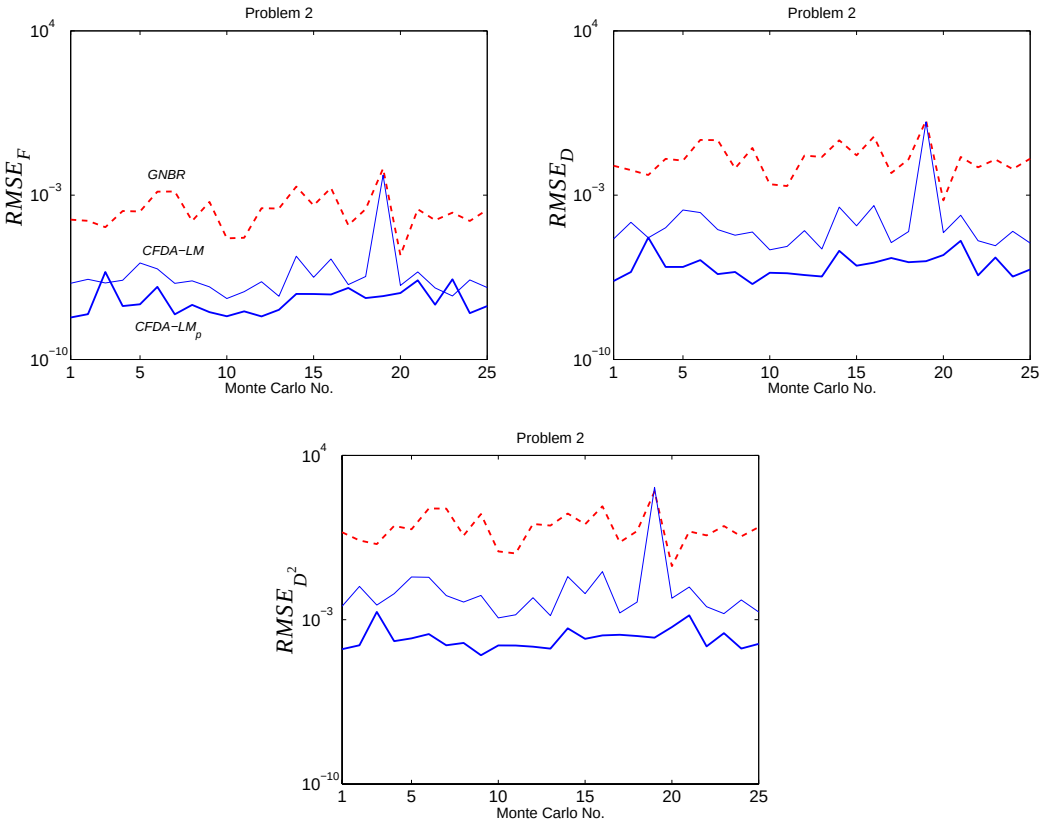

Figure 23) Error comparison for problem 2

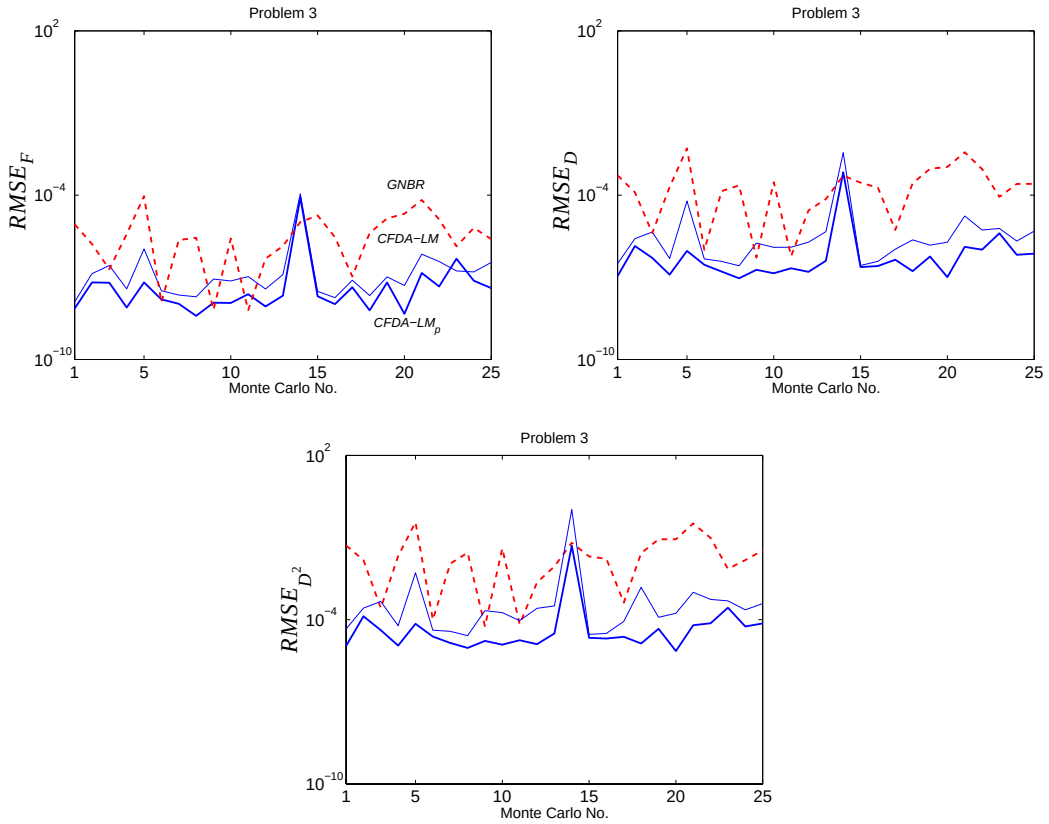

Figure 24) Error comparison for problem 3

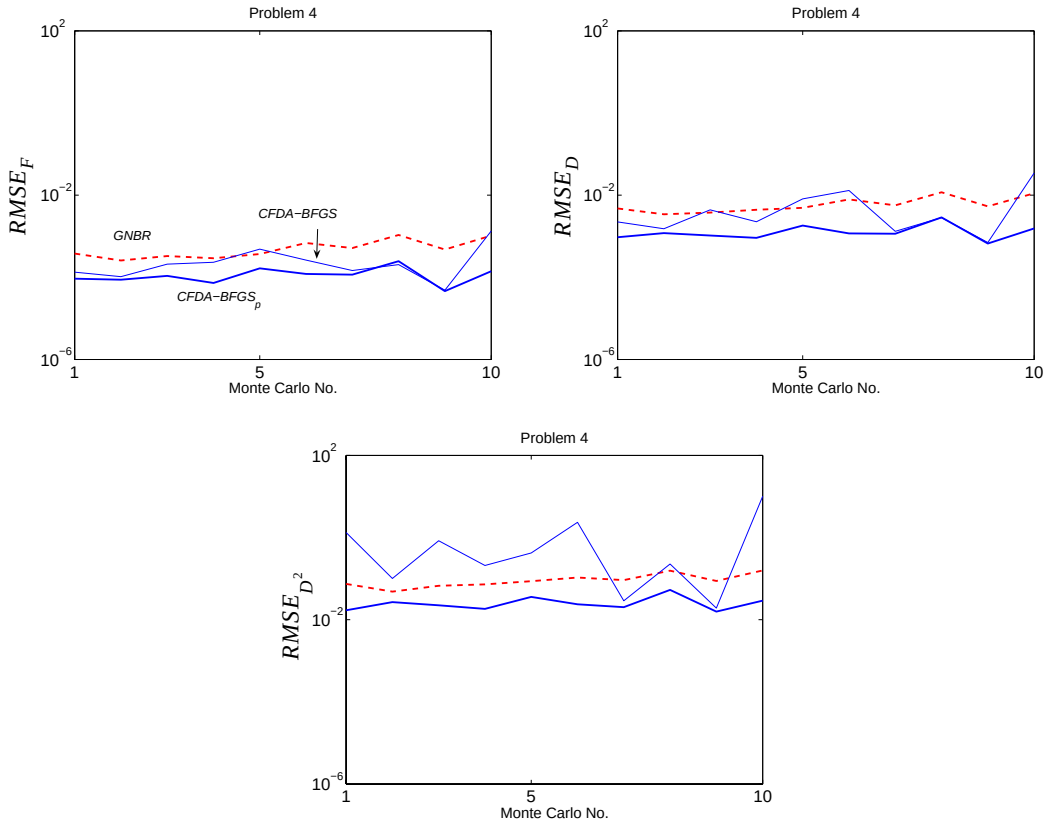

Figure 25) Error comparison for problem 4

From Figure 22) to Figure 25), we can see that the *CFDA* methods with the pruning algorithm yielded lower approximation errors than the regular *CFDA* algorithms in almost every Monte Carlo sample. Only a few networks showed slightly worse approximation errors after pruning. Two reasons could explain this. First, it could have occurred by chance (i.e. the test errors could slightly fluctuate up and down, even without pruning, when continuing training around the local minimum). Second, the retraining process yielded a new local minimum. Recall that one of the criteria to terminate the retraining process is when the *CFDA* performance index reaches the same value as before the network is pruned. This means that the function errors could be higher, while the first derivative errors could be lower than those before pruning (or vice versa). It is worth noting that, in problem 4,

$CFDA-BFGS$ produced worse second derivative errors than $GNBR$ in almost every Monte Carlo trial. However, the pruning algorithm made the errors smaller for every Monte Carlo trial.

The results in this section indicated that the $CFDA$ methods with the pruning algorithm yielded the most accurate approximation, followed by the regular $CFDA$ methods and the standard training algorithms. Of all the methods tested, $CFDA-LM$ with pruning yielded the most accurate networks. In the next section, we will demonstrate some examples, which were selected from this section, to show the $CFDA$ overfitting and how the pruning algorithm eliminates them.

Elimination of CFDA Overfitting

This section provides some examples that demonstrate the $CFDA$ overfitting and show how the pruning algorithm removes them. The plots are generated from some networks selected from one of the two sets of the K Monte Carlo networks. There are two parts in this section. Each part will illustrate one type of $CFDA$ overfitting and demonstrate how the pruning algorithm removes it. We will start with $Type-A$ and then $Type-B$.

Type A

We will show two examples. First is the overfitting that occurs in problem 2 with a network trained by $CFDA-LM$. The second is from problem 4 with a network trained by $CFDA-BFGS$.

For the overfitting in problem 2, Figure 26) shows the true function, its first derivative and the function and derivative responses of the network trained by $CFDA-LM$.

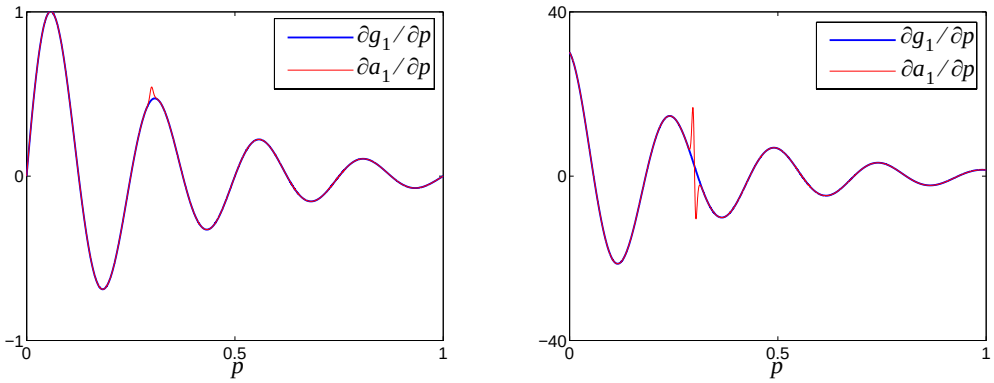

Figure 26) Overfitting in the function and derivative responses

The function and derivative errors are shown in Figure 27). We can see that the errors were small everywhere, except over a very tiny region. The large errors occur between training data. Since the area in the input space having the overfitting is very small, a validation set would not be able to detect this. The symbol $\times$ represents the location of training points.

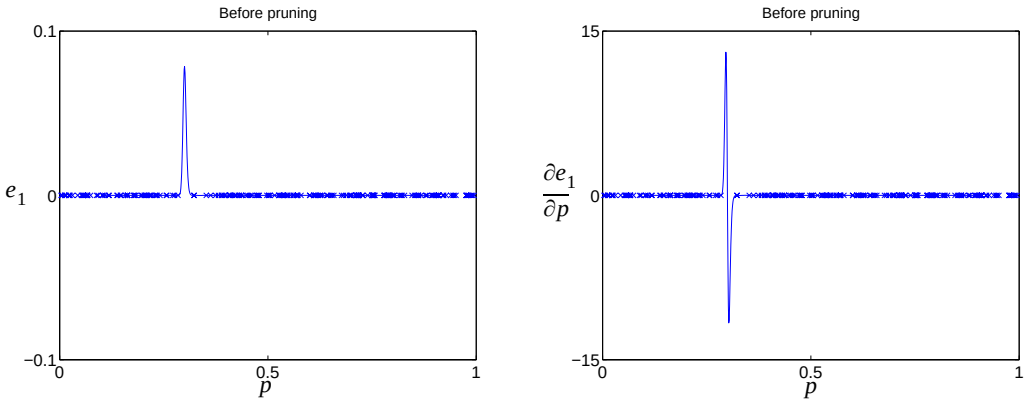

Figure 27) Function and first derivative errors

Figure 28) and Figure 29) show how the overfitting occurred. The pruning algorithm indicated that the overfitting was produced by three neurons with responses shown in Figure 28). We can see that two of the three neurons have extremely large slope, while the other one is smaller to compensate the difference between the two. Notice also that the centers of these neurons are close together.

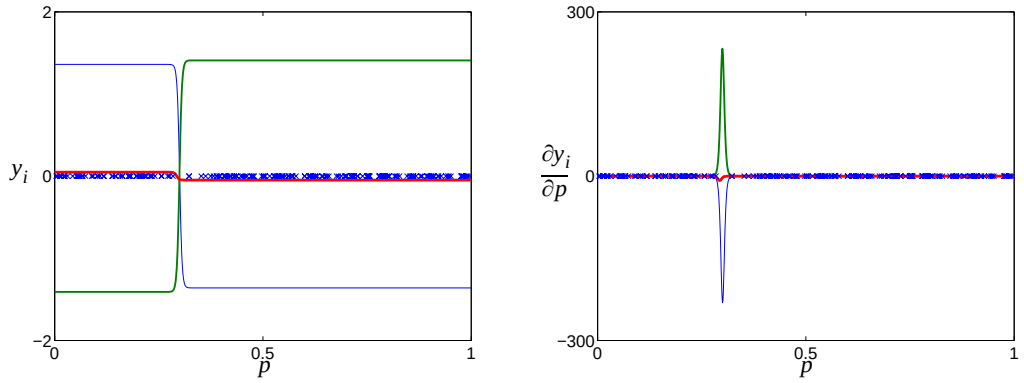

Figure 28) Responses of the three neurons

When combining the responses of these three neurons, we obtain Figure 29).

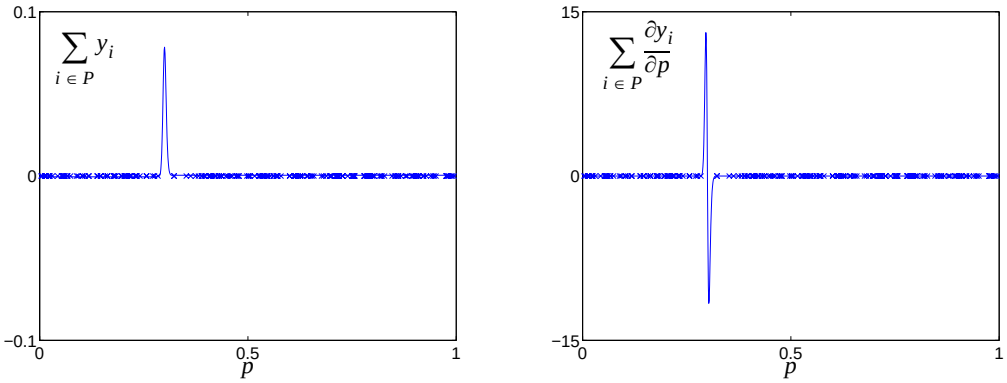

Figure 29) The combined responses of the three neurons

We can see from Figure 29) that the combined response is very small everywhere, except where the overfitting occurs. In that region, the responses of the three neurons did not cancel, thus yielding a large residual. By removing these three neurons, as indicated by the pruning algorithm, the function and derivative errors right after pruning (without any $CFDA$ retraining) are shown in Figure 30). The figure has two rows, both are the same but different in scale. The first row uses the same scale as Figure 27), while the second row uses a smaller scale.

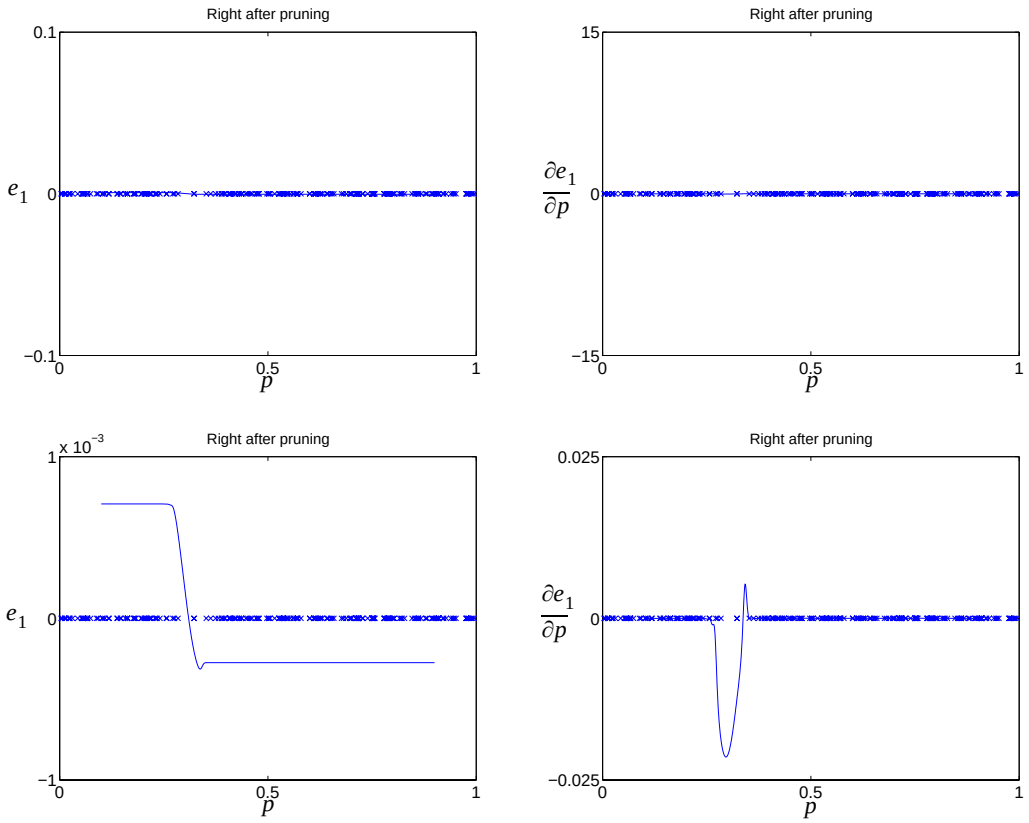

Figure 30) Function and derivative errors, right after pruning (with no retraining)

From Figure 30) in the first row, we can see that the pruning algorithm got rid of the overfitting. However, pruning the network caused worse approximation at some training points. Therefore, a retraining process after pruning is needed. Figure 31) shows the function and derivative errors of the final retrained network. The figure shows the errors in the same scale as the figure in the second row of Figure 30).

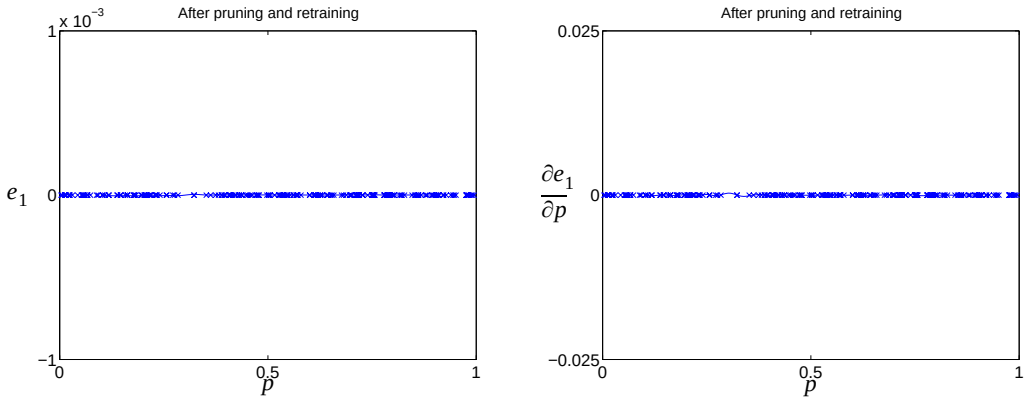

Figure 31) Function and derivative errors of the final network

We can see that we obtained much lower errors after the network was pruned. The approximation was smoother, and the second derivative errors were also lower after pruning.

Next, we will demonstrate the overfitting in problem 4, which has two inputs. Figure 32) illustrates the function error and the first derivative error with respect to each input of the network trained by $CFDA-BFGS$.

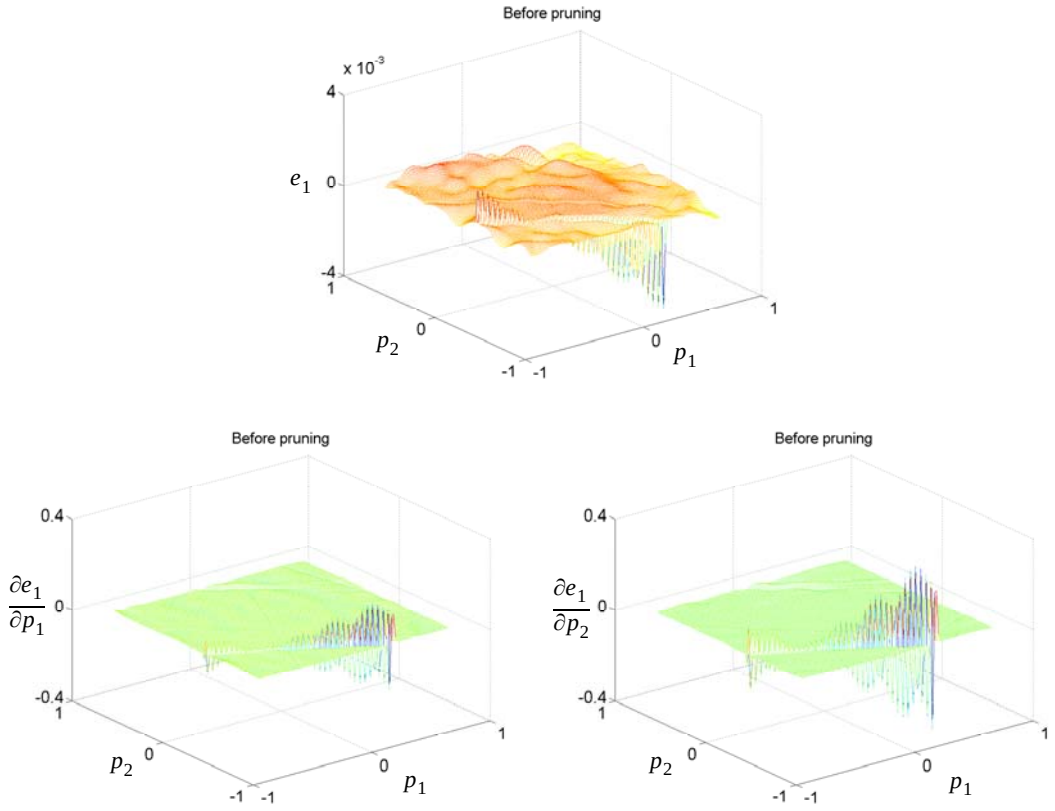

Figure 32) Function and derivative errors

We can see from Figure 32) that the errors were small everywhere, except where the over-fitting occurred (which was along a line). We emphasize again that, since the regions where the large errors occur is very small, the use of a validation set would not be able to detect this. The pruning algorithm indicated that two neurons whose centers are almost parallel caused the overfitting. Their responses are shown in Figure 33).

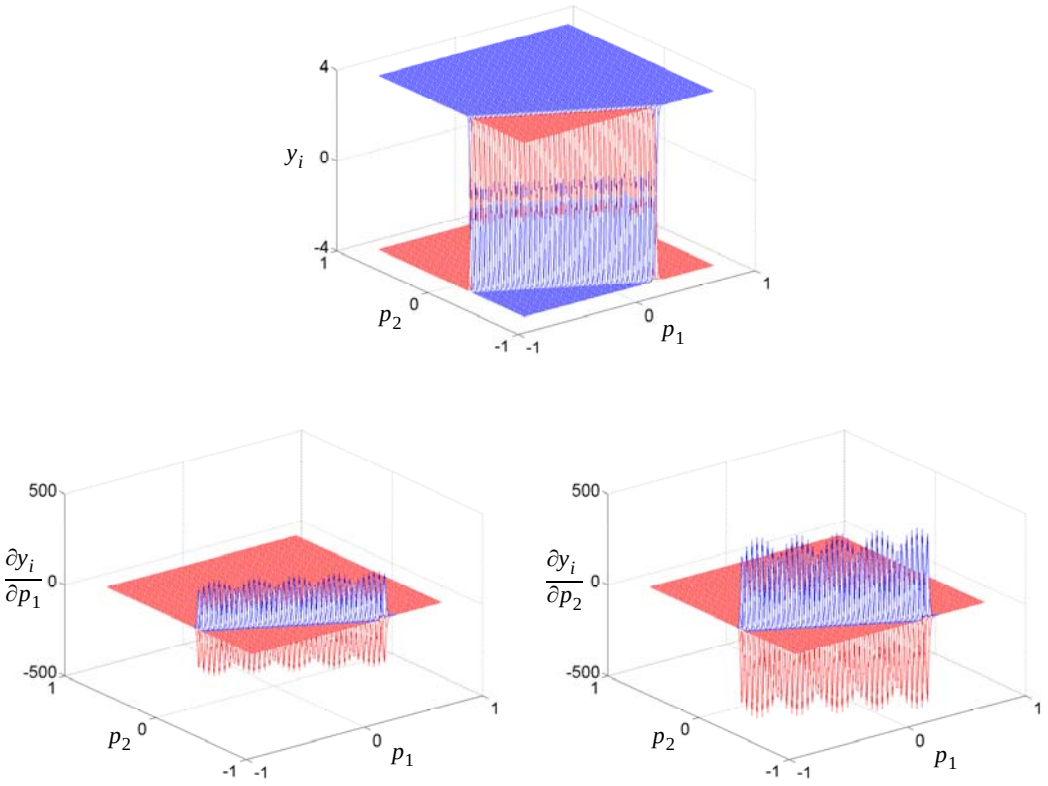

Figure 33) Responses of the two neurons

From Figure 33), their slopes were very large, almost the same size with opposite sign.

When combining their responses, we obtain Figure 34).

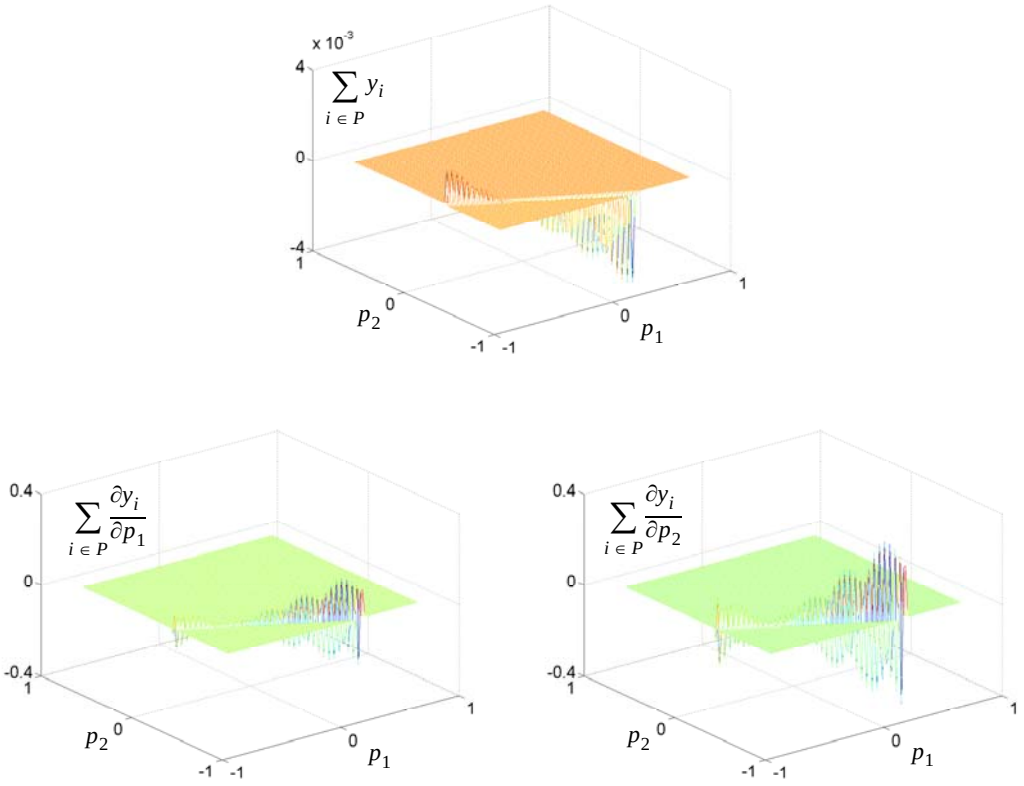

Figure 34) The combined responses of the two neurons

Figure 34) shows that the responses of the two neurons cancel everywhere, except in the region close to the neurons' centers, which is the region where the overfitting occurred. The function and derivative errors right after pruning these two neurons from the network (without a $CFDA$ retraining) are shown in Figure 35).

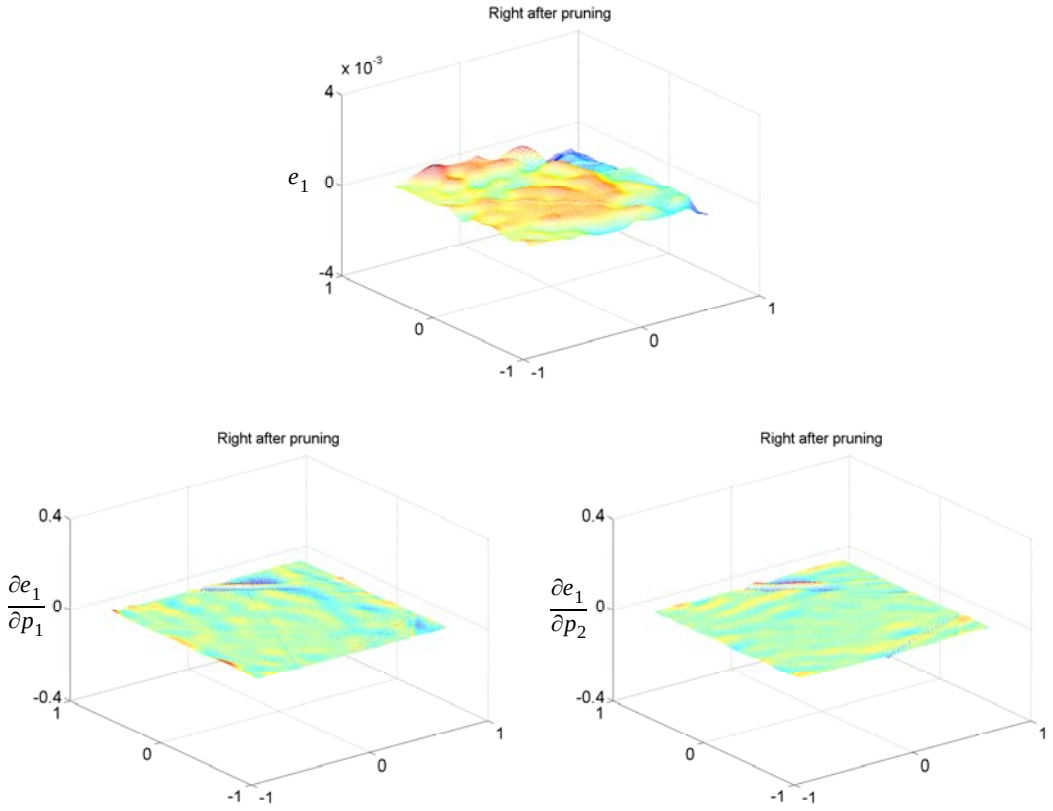

Figure 35) Function and derivative errors, right after pruning (with no retraining)

From Figure 35), we can see that the overfitting disappeared (even without retraining). This is because the pruning algorithm correctly identified and removed the neurons producing the overfitting. Figure 36) shows the errors of the final trained network.

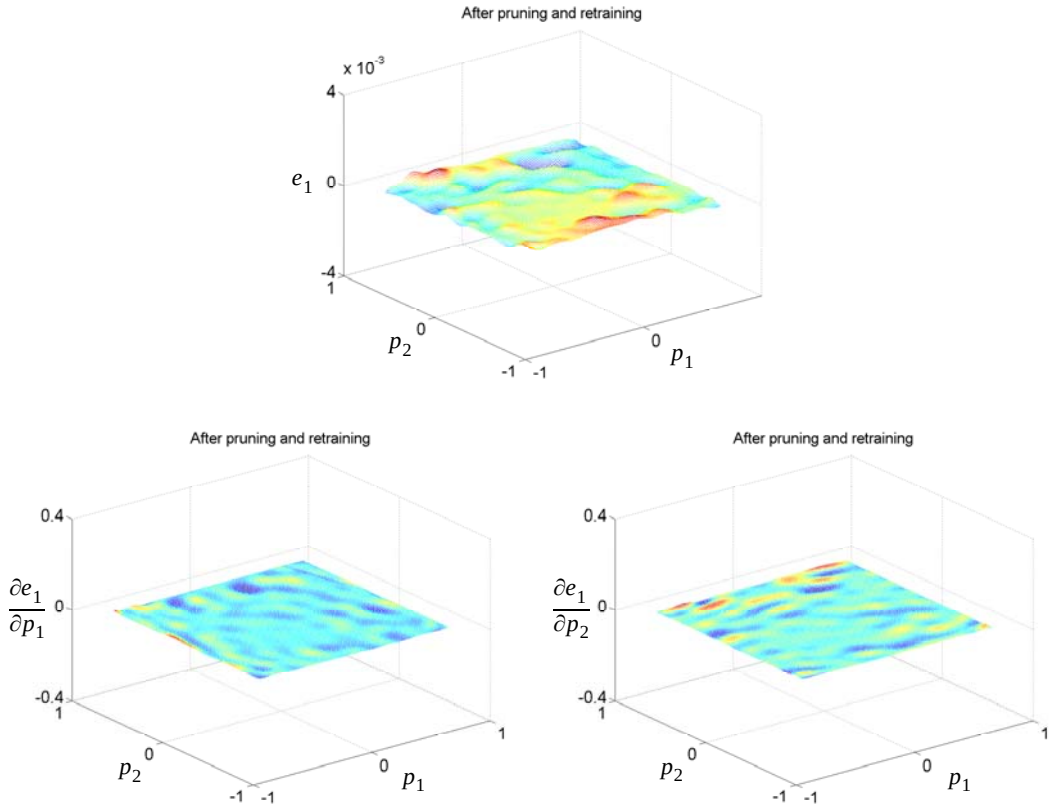

Figure 36) Function and derivative errors of the final network

With retraining after pruning, the final network produced a very smooth response on both the function and its derivatives. The generalization ability of the network after pruning was improved.

Next, some examples with $Type-B$ overfitting will be demonstrated.

Type B

Two examples will be provided. First is from a network for problem 3. The second network is for problem 4 (which has two input variables). Both networks were trained by $CFDA-BFGS$.

For the network with $Type-B$ overfitting in problem 3, Figure 37) shows the function and derivative errors.

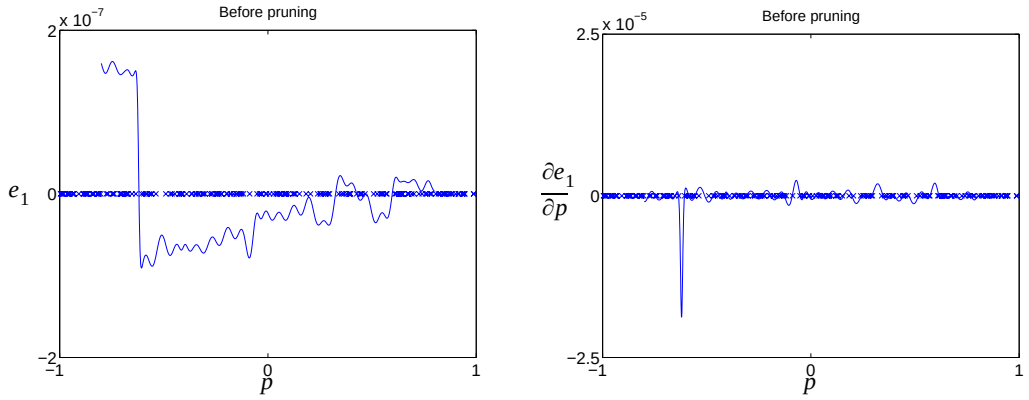

Figure 37) Function and derivative errors

We can see from the figure that the function error had a step at a point, whereas the derivative error spiked. The step in the function error demonstrated that, even on training points where $p < -0.6$, the training process misfits the function. The pruning algorithm indicated that there was one neuron causing the overfitting. Its responses are shown in Figure 38).

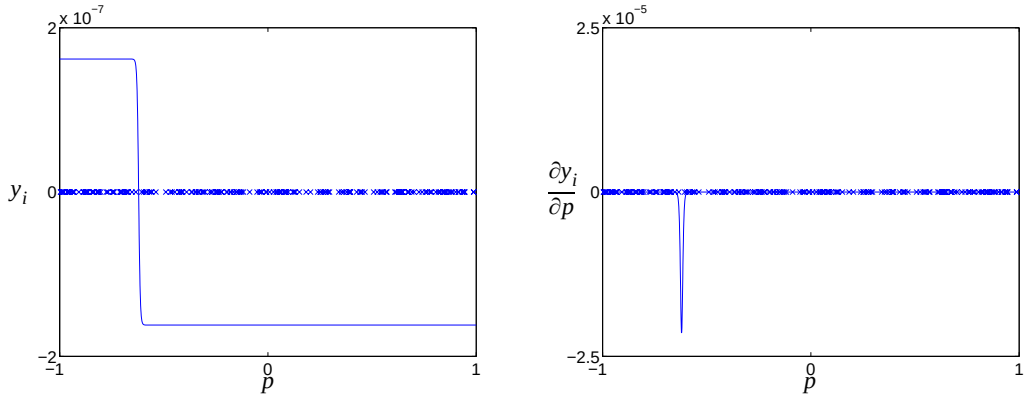

Figure 38) Responses of the neuron causing the overfitting

From Figure 38), we can see that the neuron produced a very sharp step in the function response, with a small step size. This implies the first-layer weight is large, while the second-layer weight is small. Figure 39) shows the function and derivative errors right after pruning the neuron (without a $CFDA$ retraining).

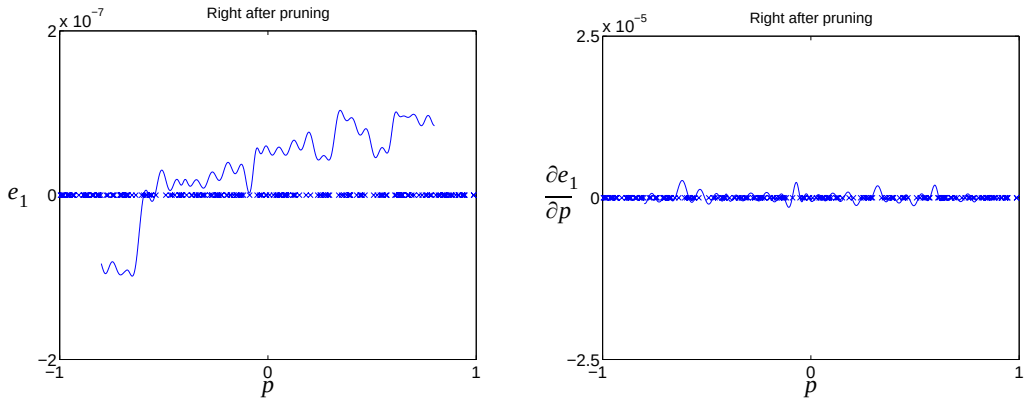

Figure 39) Function and derivative errors, right after pruning (with no retraining)

From Figure 39), we can see that right after pruning the neuron, the network responses were automatically improved. That is, the step in the function response observed in Figure 37) was smaller, while the spike in the derivative error disappeared. Figure 40) shows the errors of the final trained network.

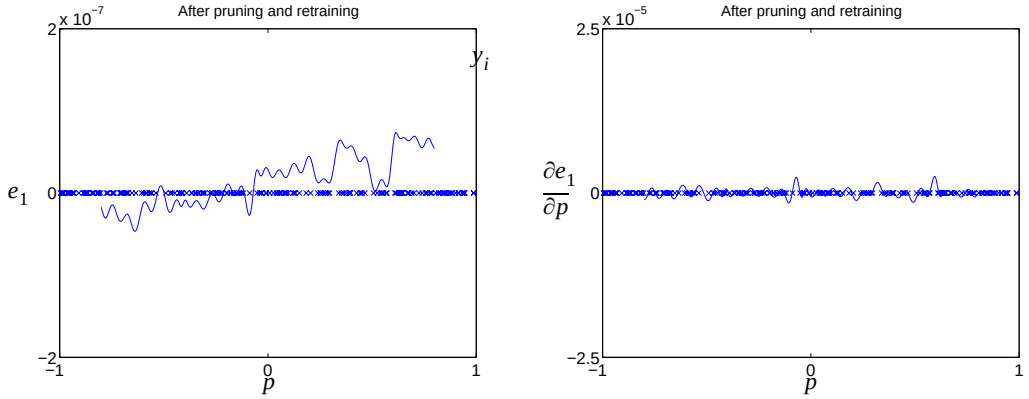

Figure 40) Function and derivative errors of the final network

As shown in Figure 40), the responses of the final network were more accurate on both the function and its derivatives.

Next, we will illustrate $type-B$ overfitting in problem 4. Figure 41) shows the function and derivative errors of the network, trained by $CFDA-BFGS$.

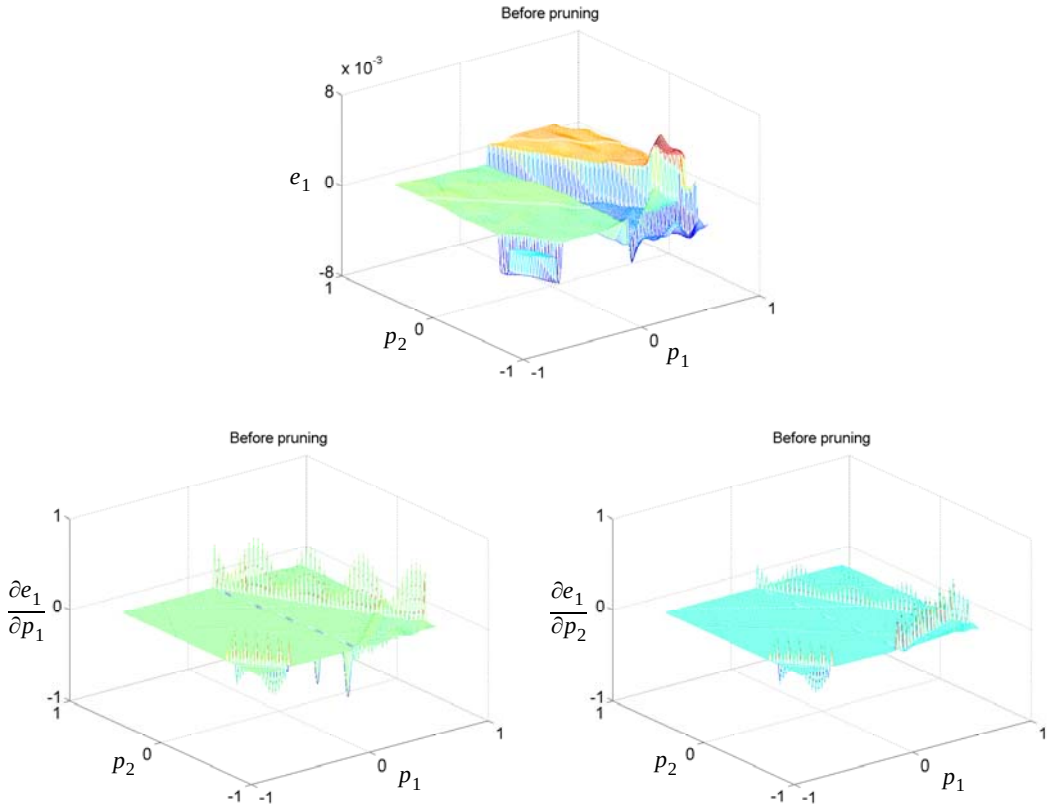

Figure 41) Function and derivative errors

From Figure 41), we can see that the errors were, although small, very jagged. The pruning algorithm indicated that 15 neurons were involved in the overfitting. We will, however, show only the response of three neurons, causing overfitting at three different locations, in Figure 42).

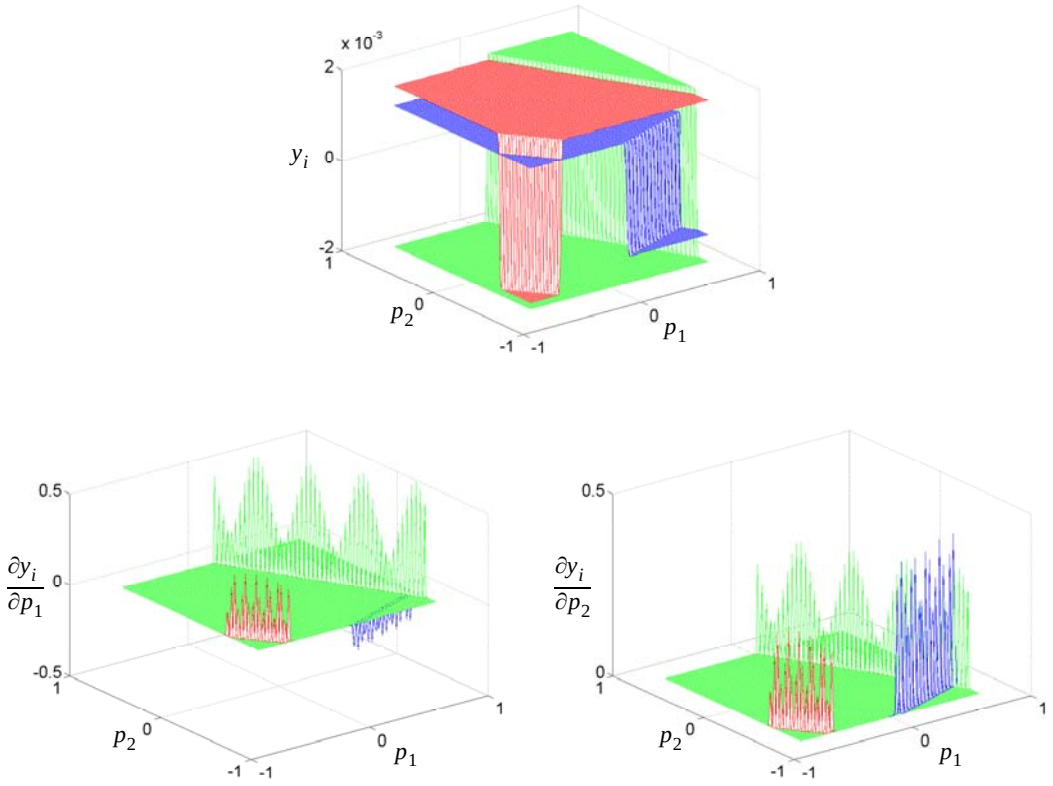

Figure 42) Responses of the three neurons

From Figure 42), we can see that these three neurons had small but very sharp steps in the function response. This corresponds to the small spikes in the derivative response. The combined responses of these 15 neurons are shown in Figure 43).

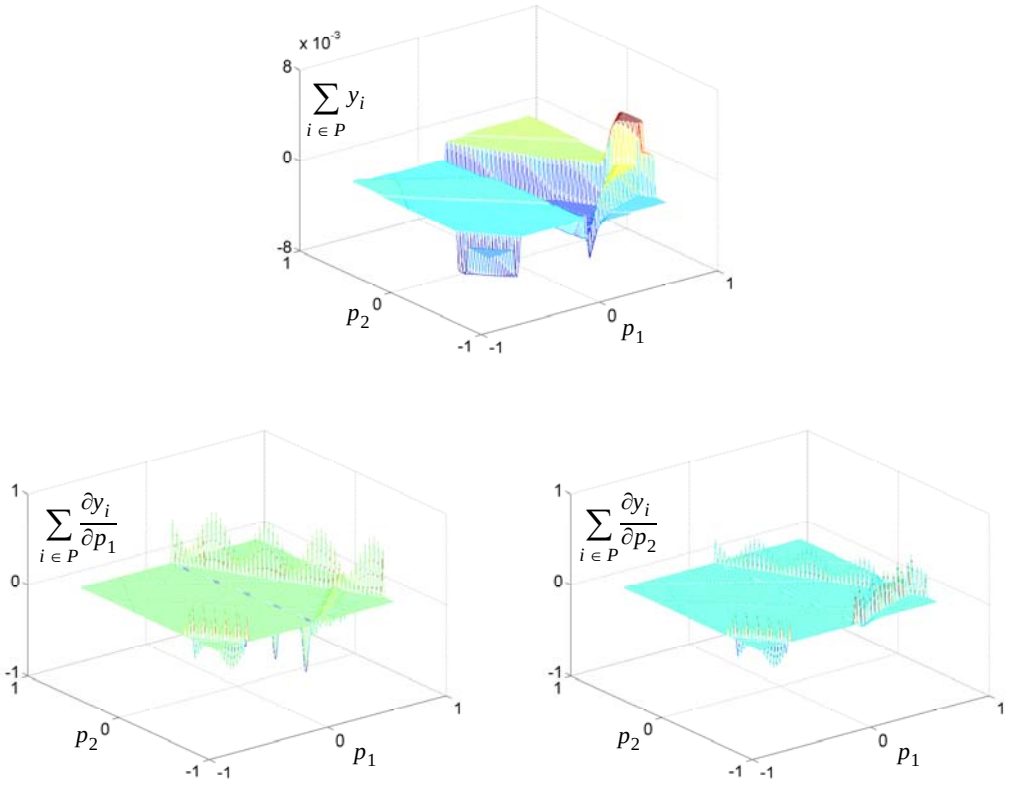

Figure 43) The combined responses of the 15 neurons

We can see from Figure 43) that the combined response of these neurons generated the jagged network responses. By removing these 15 neurons from the network, the function and derivative errors after pruning (without a $CFDA$ retraining) are shown in Figure 44).

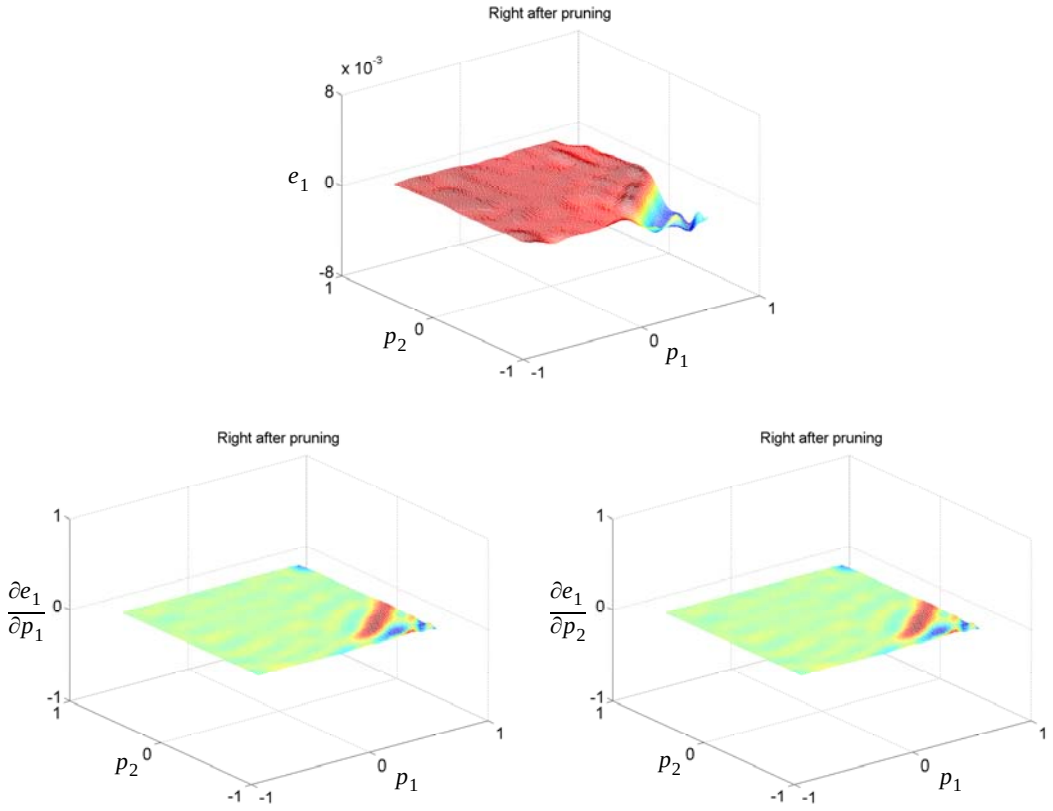

Figure 44) Function and derivative errors, right after pruning (with no retraining)

We can see from Figure 44) that the responses after pruning the network were smoother (even without a retraining), with all the jagged response removed. The responses of the final trained network are shown in Figure 45).

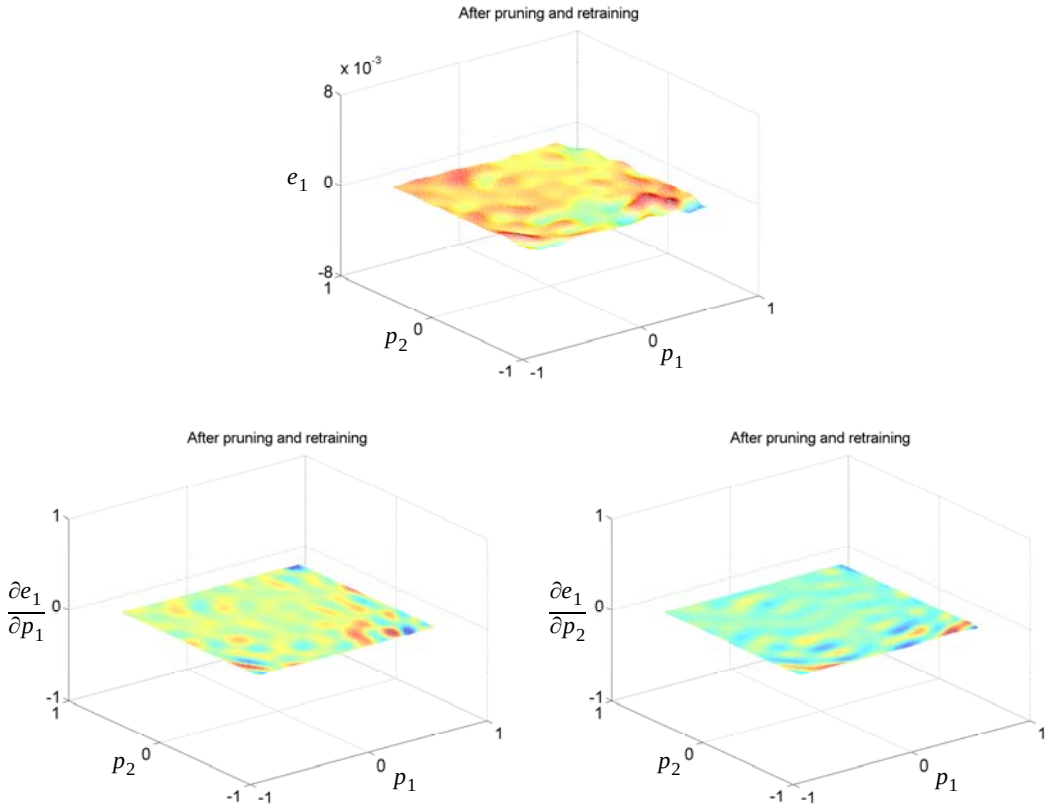

Figure 45) Function and derivative errors of the final network

Figure 45) shows that the final network produced much smoother responses than the network before pruning.

We will provide a summary of this chapter next.

Summary

In this chapter, seven different algorithms for training neural networks were tested on four simple problems. The seven training algorithms consist of $BFGS-ES$, $LM-ES$, $GNBR$, $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$. The goal was to compare the approximation accuracy obtained from each training algorithm. We proposed to measure the approximation accuracy by using RMSE on functions and their

first and the second derivatives. The effect of the parameter ρ in the $CFDA$ method was also analyzed, and ρ was defined so as to account for the scale difference between the values of the function and the values of the first-order derivatives. An automated procedure was developed so that ρ could be automatically set for any problem.

The test results showed that, among the three standard training algorithms, $GNBR$ yielded best approximation accuracy on both training and test sets. The $CFDA-BFGS$ and $CFDA-LM$ method provided even better approximation accuracy on the test set than the other three standard training methods. In fact, the derivative approximation errors were usually one or two orders of magnitude smaller when using the $CFDA$ methods.

The results also showed that the $CFDA$ training algorithms with pruning (i.e. $CFDA-BFGS_p$ and $CFDA-LM_p$) yielded even more precise approximation than the regular $CFDA$ algorithms. The pruning algorithm not only provided more accurate function and the first derivative approximation, but also it produced smoother responses, as the second derivative errors were significantly reduced. We also provided some examples showing the two types of $CFDA$ overfitting and how the pruning algorithm eliminates them, for both single and multiple inputs.

Among the training algorithms tested, $CFDA-LM_p$ produced the best approximation accuracy.

CHAPTER 8

A REAL-WORLD APPLICATION

Introduction

This chapter serves two purposes. First, we will introduce a real-world application where neural networks can be used to approximate both a function and its first-order derivatives. This application is in the field of Molecular Dynamics. We will review the general concept of molecular dynamics, followed by a description of how neural networks can be used in molecular dynamics. Second, the approximation accuracy of neural networks trained by five training algorithms: $GNBR$, $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$ will be shown for three molecular-dynamics problems. An example showing the $CFDA$ overfitting (discussed in Chapter 6) in molecular dynamics and how the pruning algorithm works will also be illustrated.

Molecular Dynamics with Neural Networks

In Molecular Dynamics (MD), the motion of atoms and molecules in a material under a given force are simulated, using known laws of physics to calculate the forces on individual atoms. This section will provide a basic review of molecular dynamics. For further details on molecular dynamics, books such as [Raff01] and [Stei85] are recommended. We will start by reviewing Hamilton's equations of motion in classical mechanics. Then, the

well-known stationary-state Schrödinger equation of quantum mechanics will be briefly described. An additional assumption, the Born-Oppenheimer approximation, which is used to numerically solve the stationary-state Schrödinger equation, will be also reviewed. Finally, we will discuss a general framework for molecular dynamics.

Classical mechanics: Hamilton's equation of motion

For any isolated system consisting of K particles, the classical Newtonian equations of motion can be used to describe the behavior of all the particles. From the three fundamental Newtonian postulates, the behavior of any particle in the system is described by the Newtonian equations of motion. For example, if the system is referred in the Cartesian coordinate system, the Newtonian equations of motion for particle k are in the form:

$$m_k \frac{\mathrm{d}^2 x_k}{\mathrm{d}t^2} + \frac{\partial V}{\partial x_k} = 0,\ m_k \frac{\mathrm{d}^2 y_k}{\mathrm{d}t^2} + \frac{\partial V}{\partial y_k} = 0 \text{ and } m_k \frac{\mathrm{d}^2 z_k}{\mathrm{d}t^2} + \frac{\partial V}{\partial z_k} = 0, \tag{307}$$

where m_k is the mass of particle k, and t denotes time. The variables x_k, y_k and z_k are the positions of particle k along the x, y and z directions, respectively. The term V is the potential energy of the system, and it is a function of the position of all particles, i.e. $V(x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_K, y_K, z_K)$. The potential energy is related to the force field acting on particle k in the x, y and z direction by:

$$F_{x_k} = -\frac{\partial V}{\partial x_k},\ F_{y_k} = -\frac{\partial V}{\partial y_k} \text{ and } F_{z_k} = -\frac{\partial V}{\partial z_k}, \tag{308}$$

respectively.

It is useful to express the Newtonian equations in the Hamiltonian form:

$$T(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_K) + V(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_K) = E\,, \tag{309}$$

where E is the total energy of the system, which consists of the kinetic energy T and the potential energy V. The kinetic energy depends on the velocities of the K particles, and the potential energy depends on the positions of the particles, where $\mathbf{v}_k$ and $\mathbf{s}_k$ are the velocity and the position of particle k.

For example, in Cartesian coordinates, the kinetic energy is in the form:

$$T = \sum_{k=1}^{K} \frac{1}{2} m_k (v_{x_k}^2 + v_{y_k}^2 + v_{z_k}^2) = \sum_{k=1}^{K} \frac{1}{2m_k} (p_{x_k}^2 + p_{y_k}^2 + p_{z_k}^2)\,, \tag{310}$$

where p_{x_k}, p_{y_k}, p_{z_k} represent the momentum of particle k with mass m_k and the velocities v_{x_k}, v_{y_k}, v_{z_k}, along the x, y and z coordinates, respectively. From Eq. (309) and Eq. (310), we obtain the classical Hamilton's equations of motion for particle k:

$$\frac{\partial E}{\partial p_{x_k}} = \frac{p_{x_k}}{m_k} = v_{x_k} = \frac{\mathrm{d}x_k}{\mathrm{d}t}\,, \tag{311}$$

$$\frac{\partial E}{\partial p_{y_k}} = \frac{p_{y_k}}{m_k} = v_{y_k} = \frac{\mathrm{d}y_k}{\mathrm{d}t}\,, \tag{312}$$

$$\frac{\partial E}{\partial p_{z_k}} = \frac{p_{z_k}}{m_k} = v_{z_k} = \frac{\mathrm{d}z_k}{\mathrm{d}t}\,, \tag{313}$$

$$\frac{\partial E}{\partial x_k} = \frac{\partial V}{\partial x_k} = -F_{x_k} = -\frac{\mathrm{d}p_{x_k}}{\mathrm{d}t}, \tag{314}$$

$$\frac{\partial E}{\partial y_k} = \frac{\partial V}{\partial y_k} = -F_{y_k} = -\frac{\mathrm{d}p_{y_k}}{\mathrm{d}t}, \tag{315}$$

$$\frac{\partial E}{\partial z_k} = \frac{\partial V}{\partial z_k} = -F_{z_k} = -\frac{\mathrm{d}p_{z_k}}{\mathrm{d}t}. \tag{316}$$

It should be noted that, although written in Cartesian coordinates, Eq. (311) to Eq. (316) hold regardless of the coordinate system used [Raff01], see the proof in [Arya90].

If we know the positions, the velocities of all particles at a particular time and the functional form of V, we can compute the entire future and past behavior of the system (i.e. $x_k(t)$, $y_k(t)$ and $z_k(t)$) by solving the Newtonian or the Hamilton's equations of motion.

Quantum mechanics: The Schrödinger equation

In an isolated molecular system, electrons and nuclei are considered particles. The classical Hamilton's equations of motion, however, fail to explain the behavior of the particles, e.g. the behavior that the electron can stay in its orbital in a Hydrogen atom. Quantum theories were then developed, in order to provide a better explanation of the particles' behavior. The theories are based upon the fundamental postulates of quantum mechanics. The postulates allow the existence of the wave function that follows certain mathematical properties. The wave function becomes a mathematical tool to provide a complete description of how the particles behave. In 1926, Erwin Schrödinger showed how the wave function can evolve over time. Similar to the classical Hamiltonian, Schrödinger showed that the quantum mechanical Hamiltonian is the operator acting upon the wave function:

$$\mathcal{H}\psi(\mathbf{q}, t) = \mathcal{E}\psi(\mathbf{q}, t), \tag{317}$$

where $\psi(\mathbf{q}, t)$ is the wave function of a molecular system, $\mathbf{q}$ denotes the positions of all the particles in the coordinates used. Eq. (317) is called the *time-dependent* Schrödinger equation. The quantum mechanical Hamiltonian $\mathcal{H}$ and the total energy $\mathcal{E}$ operators are:

$$\mathcal{H} = \mathcal{T} + V(\mathbf{q}, t) \text{ and } \mathcal{E} = i\tilde{h}\frac{\partial}{\partial t}. \tag{318}$$

The notation i is the imaginary unit, $\tilde{h} = h/2\pi$ and h is the Planck's constant. The term $V(\mathbf{q}, t)$ is the time-dependent electric potential energy. The kinetic energy operator $\mathcal{T}$ is in the form (in SI unit):

$$\mathcal{T} = \sum_{k=1}^{K} -\frac{\tilde{h}^2}{2m_k}\nabla_k^2 \ , \tag{319}$$

where ∇_k^2 is the Laplacian operator (i.e. the second-order partial derivatives with respect to the coordinates for particle k). For example, in Cartesian coordinates, we then have $\mathbf{q} = (x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_K, y_K, z_K)$ and $\mathcal{T}$ is in the form:

$$\mathcal{T} = \sum_{k=1}^{K} -\frac{\tilde{h}^2}{2m_k}\left(\frac{\partial^2}{\partial x_k^2} + \frac{\partial^2}{\partial y_k^2} + \frac{\partial^2}{\partial z_k^2}\right). \tag{320}$$

From Eq. (317), when solved, it yields the wave function $\psi(\mathbf{q}, t)$.

For many problems, the electric potential energy V does not depend on time, written $V(\mathbf{q})$. In addition, assume that the wave function can be separable and written as

$$\psi(\mathbf{q}, t) = \phi(\mathbf{q})\varphi(t). \tag{321}$$

By substituting Eq. (321) into Eq. (317) and solving it, the wave function is obtained:

$$\psi(\mathbf{q}, t) = C\phi(\mathbf{q})\exp\left(-\frac{iEt}{\tilde{h}}\right), \tag{322}$$

where C is a constant and the real constant E is

$$E = \frac{\mathcal{H}\phi(\mathbf{q})}{\phi(\mathbf{q})}. \tag{323}$$

By using Eq. (322), it has been shown that the constant E is the expected total energy of the system operating on the wave function, denoted by $\langle \mathcal{E} \rangle$:

$$\langle \mathcal{E} \rangle = \frac{\int \psi^*(\mathbf{q}, t)\mathcal{E}\psi(\mathbf{q}, t)d\tau}{\int \psi^*(\mathbf{q}, t)\psi(\mathbf{q}, t)d\tau} = E, \tag{324}$$

where $d\tau$ is the volume over the coordinates $\mathbf{q}$, e.g. $d\tau = dxdydz$ if the Cartesian coordinate is used. $\psi^*(\mathbf{q}, t)$ is the complex conjugate of $\psi(\mathbf{q}, t)$. Since E is the expected total energy of the system, it is customary to write Eq. (323) in the form:

$$\mathcal{H}\phi(\mathbf{q}) = E\phi(\mathbf{q}), \tag{325}$$

and this is called the *stationary-state* Schrödinger equation. When Eq. (325) is solved, it yields the wave function $\phi(\mathbf{q})$. Then, by Eq. (322), the time-dependent wave function $\psi(\mathbf{q}, t)$ is finally obtained.

The stationary-state Schrödinger equation can be analytically solved only in very simple problems, such as the system consisting of one Hydrogen atom (see [Raff01] for details), or the problem of a particle in a box (see [Matt93] for details and more problems). It is impossible to analytically solve Eq. (325) for more complex systems. An approximation to the solution is needed.

To obtain approximated solutions of Eq. (325) for a general system, further assumptions and numerical methods are needed. Before introducing the assumptions, let us rewrite the stationary-state Schrödinger equation in a more general form. Consider an N-particle system consisting of L electrons and K nuclei, i.e. $N = L + K$. Let $\mathbf{r}_l$ be the vector representing the position of electron l, and $\mathbf{s}_k$ be the vector representing the position of nucleus k. The Hamiltonian operator in Eq. (325) can be rewritten as:

$$\mathcal{H} = \mathcal{T}_e + \mathcal{T}_n + V(\mathbf{r}, \mathbf{s}) , \tag{326}$$

where, from Eq. (319),

$$\mathcal{T}_e = -\sum_{l=1}^{L} \frac{\tilde{h}^2}{2m_l}\nabla_l^2 \text{ and } \mathcal{T}_n = -\sum_{k=1}^{K} \frac{\tilde{h}^2}{2M_k}\nabla_k^2 \tag{327}$$

are the kinetic energy operator of all electrons and that of all nuclei, respectively. The term m_l is the mass of electron l, and M_k is the mass of nucleus k. The Laplacian operator in $\mathcal{T}_e$ and $\mathcal{T}_n$ is with respect to $\mathbf{r}_l$ and $\mathbf{s}_k$, respectively. The electric potential energy $V(\mathbf{r}, \mathbf{s})$ can be computed, by the Coulomb's law, in SI unit as:

$$\begin{aligned} V(\mathbf{r}, \mathbf{s}) &= V_{ne}(\mathbf{r}, \mathbf{s}) + V_{ee}(\mathbf{r}) + V_{nn}(\mathbf{s}) \\ &= -\sum_{k=1}^{K}\sum_{l=1}^{L} \frac{Z_k e^2}{4\pi\varepsilon_0 \|\mathbf{s}_k - \mathbf{r}_l\|} + \frac{1}{2}\sum_{l=1}^{L-1}\sum_{l'=l+1}^{L} \frac{e^2}{4\pi\varepsilon_0 \|\mathbf{r}_l - \mathbf{r}_{l'}\|} \\ &\quad + \frac{1}{2}\sum_{k=1}^{K-1}\sum_{k'=k+1}^{K} \frac{Z_k Z_{k'} e^2}{4\pi\varepsilon_0 \|\mathbf{s}_k - \mathbf{s}_{k'}\|} , \end{aligned} \tag{328}$$

where ε_0 is the permittivity of the vacuum, and e is the charge magnitude (electrons and

protons have equal charge magnitudes that are opposite in sign). The term $V_{ne}(\mathbf{r}, \mathbf{s})$ is electron-nuclei attraction, the term $V_{ee}(\mathbf{r})$ is electron-electron repulsion, and the term $V_{nn}(\mathbf{s})$ is nuclei-nuclei repulsion. The notation $\|\mathbf{x}\|$ is the 2-norm of the vector $\mathbf{x}$, or the distance between the two vectors in the 2-norm. The notation Z_k is the atomic number of nucleus k. In atomic units, Eq. (327) reduces to:

$$\mathcal{T}_e = -\sum_{l=1}^{L} \frac{1}{2}\nabla_l^2 \text{ and } \mathcal{T}_n = -\sum_{k=1}^{K} \frac{1}{2M_k}\nabla_k^2 \ , \tag{329}$$

while Eq. (328) reduces to

$$V(\mathbf{r}, \mathbf{s}) = -\sum_{k=1}^{K}\sum_{l=1}^{L} \frac{Z_k}{\|\mathbf{s}_k - \mathbf{r}_l\|} + \frac{1}{2}\sum_{l=1}^{L-1}\sum_{l'=l+1}^{L} \frac{1}{\|\mathbf{r}_l - \mathbf{r}_{l'}\|} + \frac{1}{2}\sum_{k=1}^{K-1}\sum_{k'=k+1}^{K} \frac{Z_k Z_{k'}}{\|\mathbf{s}_k - \mathbf{s}_{k'}\|} \ . \tag{330}$$

To numerically solve Eq. (325), a well-known assumption, called the Born-Oppenheimer approximation, is first needed. We will discuss the Born-Oppenheimer approximation in the next section.

Born-Oppenheimer Approximation

To solve Eq. (325) numerically, Max Born and Robert J. Oppenheimer first assumed that the wave function $\phi(\mathbf{r}, \mathbf{s})$ can be separable to

$$\phi(\mathbf{r}, \mathbf{s}) \approx \phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}), \tag{331}$$

where the electronic wave function $\phi_e(\mathbf{r};\mathbf{s})$ is a function of electron coordinates $\mathbf{r}$ and it depends *parametrically* on $\mathbf{s}$. This means that, at a fixed nuclear position $\mathbf{s}$, the electronic wave function depends only on the coordinates $\mathbf{r}$; however, the shape of the electronic

wave function is changed as the nuclei's position $\mathbf{s}$ changes. The term $\phi_n(\mathbf{s})$ is the nuclear wave function, depending only on the position of all nuclei. From Eq. (325) to Eq. (331), Eq. (325) can be written as:

$$[\mathcal{T}_e + \mathcal{T}_n + V_{ne}(\mathbf{r}, \mathbf{s}) + V_{ee}(\mathbf{r}) + V_{nn}(\mathbf{s})]\phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}) = E\phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}). \tag{332}$$

From Eq. (332), first note that, since $\mathcal{T}_e$ is the operator only on $\mathbf{r}$, then

$$\mathcal{T}_e\phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}) = \phi_n(\mathbf{s})\mathcal{T}_e\phi_e(\mathbf{r};\mathbf{s}). \tag{333}$$

In addition, recall from Eq. (329) that $\mathcal{T}_n$ is the second order derivative operator, thus

$$\mathcal{T}_n\phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}) = \phi_e(\mathbf{r};\mathbf{s})\mathcal{T}_n\phi_n(\mathbf{s}) - \left(\sum_k \frac{1}{2M_k}\{2\nabla_k\phi_n(\mathbf{s})\nabla_k\phi_e(\mathbf{r};\mathbf{s}) + \phi_n(\mathbf{s})\nabla_k^2\phi_e(\mathbf{r};\mathbf{s})\}\right). \tag{334}$$

Here comes the second approximation. Born and Oppenheimer used that fact that nuclei are much heavier than electrons, i.e. $M_k >> m_l$. This assumption makes the bracketed term in Eq. (334) negligible (see [Stei85]); resulting in

$$\mathcal{T}_n\phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}) \approx \phi_e(\mathbf{r};\mathbf{s})\mathcal{T}_n\phi_n(\mathbf{s}). \tag{335}$$

Plugging Eq. (333) and Eq. (335) into Eq. (332), and rearranging the terms, we obtain

$$\phi_e(\mathbf{r};\mathbf{s})\mathcal{T}_n\phi_n(\mathbf{s}) + V_{nn}(\mathbf{s})\phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}) + \phi_n(\mathbf{s})[\mathcal{T}_e + V_{ne}(\mathbf{r}, \mathbf{s}) + V_{ee}(\mathbf{r})]\phi_e(\mathbf{r};\mathbf{s}) = E\phi_e(\mathbf{r};\mathbf{s})\phi_n(\mathbf{s}). \tag{336}$$

For a *fixed* position of nuclei $\tilde{\mathbf{s}}$, the third term in Eq. (336) depends only on the electron coordinates $\mathbf{r}$. This forms the *electronic* Schrödinger equation:

$$[\mathcal{T}_e + V_{ne}(\mathbf{r};\tilde{\mathbf{s}}) + V_{ee}(\mathbf{r})]\phi_e(\mathbf{r};\tilde{\mathbf{s}}) = \mathcal{H}_e\phi_e(\mathbf{r};\tilde{\mathbf{s}}) = E_e(\tilde{\mathbf{s}})\phi_e(\mathbf{r};\tilde{\mathbf{s}}), \tag{337}$$

where the real constant $E_e(\tilde{\mathbf{s}})$ is the expected electronic energy at the given nuclear position $\tilde{\mathbf{s}}$. It should be noted that, although written as a function of nuclear position, this energy value changes as the nuclear configuration (i.e. distances and/or angles between nuclei) changes. This means that two different nuclear positions with the same nuclear configuration produce the same value of the electronic energy. We will mention this point again when we want to rewrite the energy as the function of nuclear configuration, rather than the nuclear position. Similar to Eq. (324), the term $E_e(\tilde{\mathbf{s}})$ can then be computed by:

$$E_e(\tilde{\mathbf{s}}) = \frac{\int \phi_e^*(\mathbf{r};\tilde{\mathbf{s}}) \mathcal{H}_e \phi_e(\mathbf{r};\tilde{\mathbf{s}}) d\mathbf{r}}{\int \phi_e^*(\mathbf{r};\tilde{\mathbf{s}}) \phi_e(\mathbf{r};\tilde{\mathbf{s}}) d\mathbf{r}} \quad . \tag{338}$$

By substituting Eq. (337) into Eq. (336), we obtain

$$\begin{aligned} \phi_e(\mathbf{r};\tilde{\mathbf{s}}) \mathcal{T}_n \phi_n(\tilde{\mathbf{s}}) + V_{nn}(\tilde{\mathbf{s}}) \phi_e(\mathbf{r};\tilde{\mathbf{s}}) \phi_n(\tilde{\mathbf{s}}) + \phi_n(\tilde{\mathbf{s}}) E_e(\tilde{\mathbf{s}}) \phi_e(\mathbf{r};\tilde{\mathbf{s}}) \\ = E(\tilde{\mathbf{s}}) \phi_e(\mathbf{r};\tilde{\mathbf{s}}) \phi_n(\tilde{\mathbf{s}}) \quad , \end{aligned} \tag{339}$$

and rearranging Eq. (339) yields the *nuclear* Schrödinger equation:

$$[\mathcal{T}_n + V_{nn}(\tilde{\mathbf{s}}) + E_e(\tilde{\mathbf{s}})] \phi_n(\tilde{\mathbf{s}}) = \mathcal{H}_n \phi_n(\tilde{\mathbf{s}}) = E(\tilde{\mathbf{s}}) \phi_n(\tilde{\mathbf{s}}) , \tag{340}$$

where the term $E(\tilde{\mathbf{s}})$ is the expected total energy at the fixed nuclear position $\tilde{\mathbf{s}}$. Eq. (340) implies that the nuclei are considered to be moving in the potential generated by the moving electrons $E_e(\tilde{\mathbf{s}})$ and the nuclei repulsion $V_{nn}(\tilde{\mathbf{s}})$. Note again that the energy $E(\tilde{\mathbf{s}})$ and the nuclei repulsion $V_{nn}(\tilde{\mathbf{s}})$ change as the nuclear configuration changes.

The steps for solving Eq. (325) using the Born-Oppenheimer approximation can be summarized as follows. First, assume Eq. (331). Second, the nuclei are considered to be

fixed at a configuration $\tilde{\mathbf{s}}$. Third, solve the electronic Schrödinger equation, i.e Eq. (337), and obtain $E_e(\tilde{\mathbf{s}})$ from Eq. (338). Fourth, use $E_e(\tilde{\mathbf{s}})$ and solve the nuclear Schrödinger equation, i.e. Eq. (340). Finally, infinitesimally change the nuclear configuration and repeat the procedure.

With the assistance of the Born-Oppenheimer approximation, we are now ready to explain molecular dynamics.

Molecular Dynamics

In molecular dynamics, the motion of atoms is of interest. By the Born-Oppenheimer approximation, this is equivalent to changing nuclei position in the potential of the moving electrons and nuclei repulsion. However, to perform molecular dynamics, we need further approximations. First, recall that we need to solve $E_e(\tilde{\mathbf{s}})$ using Eq. (338). Unfortunately, we cannot analytically solve it, since the electronic wave function $\phi_e(\mathbf{r};\tilde{\mathbf{s}})$ is unknown. One way to overcome this problem is to create a predefined mathematical model for the electronic wave function. Let $\Upsilon(\mathbf{r}, \Omega;\mathbf{s})$ represent the mathematical model used in approximating the true electronic wave function $\phi_e(\mathbf{r};\mathbf{s})$. The model $\Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}})$ is a function of all of the electrons' position $\mathbf{r}$ and a set of model parameters Ω, at the fixed nuclei position $\tilde{\mathbf{s}}$. We use to notation $\hat{E}_e(\tilde{\mathbf{s}}, \Omega)$ to represent the expected electronic energy of the system using the guessed model $\Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}})$. Similar to Eq. (340), $\hat{E}_e(\tilde{\mathbf{s}}, \Omega)$ is then computed by:

$$\hat{E}_e(\tilde{\mathbf{s}}, \Omega) = \frac{\int \Upsilon^*(\mathbf{r}, \Omega;\tilde{\mathbf{s}}) \mathcal{H}_e \Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}}) d\mathbf{r}}{\int \Upsilon^*(\mathbf{r}, \Omega\tilde{\mathbf{s}}) \Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}}) d\mathbf{r}} \quad . \tag{341}$$

Nature always adjusts the electrons in the most stable orbitals possible, thus its true energy is lowest. Therefore, for any values of Ω,

$$\hat{E}_e(\tilde{\mathbf{s}}, \Omega) \geq E_e(\tilde{\mathbf{s}}), \tag{342}$$

and the equality holds when $\Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}}) = \phi_e(\mathbf{r};\tilde{\mathbf{s}})$. This inequality is called *Rayleigh-Ritz Variational Principle*. Hence, given the nuclei position $\tilde{\mathbf{s}}$, we adjust the parameters Ω in the model $\Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}})$ so as to minimize the energy $\hat{E}_e(\tilde{\mathbf{s}}, \Omega)$. This is an optimization process, with respect to the parameters Ω in the model $\Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}})$. Once minimized, we obtain Ω^*, i.e. Ω that produces the minimum value of $\hat{E}_e(\tilde{\mathbf{s}}, \Omega)$. We use $\Upsilon(\mathbf{r};\tilde{\mathbf{s}})$ and $\hat{E}_e(\tilde{\mathbf{s}})$ to denote the model $\Upsilon(\mathbf{r}, \Omega;\tilde{\mathbf{s}})$ and the value of $\hat{E}_e(\tilde{\mathbf{s}}, \Omega)$, evaluated at Ω^*. The energy $\hat{E}_e(\tilde{\mathbf{s}})$ approximates the true electronic energy of the system $E_e(\tilde{\mathbf{s}})$. Examples of models that have been used for approximating the electronic wave function are Slater type orbitals (STO) and Gaussian type orbitals (GTO). There are several methods that have been used to obtain Ω^* and $\hat{E}_e(\tilde{\mathbf{s}})$, such as the Hartree-Fock method and the electron correlation methods (e.g. M∅ller-Plesset Perturbation theory (MP) [MoPl34], and the density functional theory (DFT) [HoKo64]). The details of the models and methods can be found in Chapter 14 in [Raff01]. To solve $\hat{E}_e(\tilde{\mathbf{s}})$ using Eq. (341), some models provide the solution in analytic form. For the models that do not provide the analytic form of solution, numerical integra-

tion is needed.

In addition to the electronic energy approximation, we also need one more approximation to perform molecular dynamics. Recall that once obtaining the electronic energy $E_e(\tilde{\mathbf{s}})$ (which is now estimated by $\hat{E}_e(\tilde{\mathbf{s}})$), we need to solve Eq. (340). However, in molecular dynamics, we rather assume that the nuclei motion is treated using classical Newtonian equations of motion. From Eq. (307) and Eq. (340), this means that we treat the nuclei as classical particles in the potential energy $V_{nn}(\mathbf{s}) + \hat{E}_e(\mathbf{s})$. For nucleus k, this takes the form:

$$M_k \frac{\mathrm{d}^2 \mathbf{s}_k}{\mathrm{d}t^2} + \frac{\partial}{\partial \mathbf{s}_k}\{ V_{nn}(\mathbf{s}) + \hat{E}_e(\mathbf{s})\} \;=\; 0\,. \tag{343}$$

The first term is simply $M_k \mathbf{a}_k$, where $\mathbf{a}_k$ is the acceleration of nucleus k in the coordinates used. The computation for the term $\partial V_{nn}(\mathbf{s})/\partial \mathbf{s}_k$ is straightforward, using Eq. (330). The term $\partial \hat{E}_e(\mathbf{s})/\partial \mathbf{s}_k$ can be computed by considering at a fixed nuclei position $\tilde{\mathbf{s}}$ and calculating

$$\frac{\partial}{\partial \mathbf{s}_k}\hat{E}_e(\tilde{\mathbf{s}}) \equiv \frac{\partial}{\partial \mathbf{s}_k}\hat{E}_e(\mathbf{s})\bigg|_{\mathbf{s}\,=\,\tilde{\mathbf{s}}} \;=\; \frac{\int \Upsilon^*(\mathbf{r};\tilde{\mathbf{s}})\left(\dfrac{\partial}{\partial \mathbf{s}_k} V_{ne}(\mathbf{r};\tilde{\mathbf{s}})\right)\Upsilon(\mathbf{r};\tilde{\mathbf{s}})\,d\mathbf{r}}{\int \Upsilon^*(\mathbf{r};\tilde{\mathbf{s}})\Upsilon(\mathbf{r};\tilde{\mathbf{s}})\,d\mathbf{r}}\,, \tag{344}$$

where

$$\frac{\partial}{\partial \mathbf{s}_k} V_{ne}(\mathbf{r};\tilde{\mathbf{s}}) \equiv \frac{\partial}{\partial \mathbf{s}_k} V_{ne}(\mathbf{r};\mathbf{s})\bigg|_{\mathbf{s}\,=\,\tilde{\mathbf{s}}}\,. \tag{345}$$

Again, numerical integration is needed to solve Eq. (344), unless the analytic form of solu-

tion exists (and this depends on the model used). Similar to Eq. (308), the force field acting on the nucleus k, at the nuclei position $\tilde{\mathbf{s}}$, is

$$\mathbf{F}_k(\tilde{\mathbf{s}}) = -\frac{\partial}{\partial \mathbf{s}_k}\bar{V}(\mathbf{s})\Big|_{\mathbf{s}=\tilde{\mathbf{s}}}, \tag{346}$$

where $\bar{V}(\mathbf{s})$ is the nuclear potential energy:

$$\bar{V}(\mathbf{s}) = V_{nn}(\mathbf{s}) + \hat{E}_e(\mathbf{s}). \tag{347}$$

Before summarizing the steps for molecular dynamics, first consider Eq. (343). Integrating the equation yields the velocity and position of each nucleus (or equivalently each atom). Several well-known algorithms have been used to perform the numerical integration, for example Verlet algorithm [Verl67], Leapfrog algorithm [Finc92], velocity Verlet algorithm [SwAn82], Beeman's algorithm [Beem76].

The steps of molecular dynamics can be summarized as follow, for a system consisting of K nuclei (or atoms) and L electrons. Suppose Beeman's algorithm is used for numerically integrating the classical Newtonian equation of motion.

1. Initialize, for atom k for all $k = 1, 2, \ldots, K$, the position $\mathbf{s}_k^{(0)}$ and velocity $\mathbf{v}_k^{(0)}$ over the coordinates used. Initialize $\mathbf{a}_k^{(-1)} = 0$. Set $n = 0$ and time $t = 0$. Choose small time step Δt (e.g. $\Delta t = 10^{-7}$).

2. Given $\mathbf{s}^{(n)}$, i.e. the position $\mathbf{s}_k^{(n)}$ for all k, we obtain $\hat{E}_e(\mathbf{s}^{(n)})$ by minimizing Eq. (341) with respect to Ω.

3. Compute $V_{nn}(\mathbf{s}^{(n)})$ using Eq. (328). Then, obtain the nuclear potential energy $\bar{V}(\mathbf{s}^{(n)})$ using Eq. (347).

4. Calculate the force field $\mathbf{F}_k(\mathbf{s}^{(n)})$, for all k, by Eq. (344) and Eq. (346).

5. The acceleration $\mathbf{a}_k^{(n)}$ can be computed by $\mathbf{a}_k^{(n)} = \mathbf{F}_k(\mathbf{s}^{(n)})/M_k$, where M_k is the mass of nucleus k.

6. For all k, move atom k, using Beeman's algorithm, by:

$$\mathbf{s}_k^{(n+1)} = \mathbf{s}_k^{(n)} + \mathbf{v}_k^{(n)}\Delta t + \frac{2}{3}\mathbf{a}_k^{(n)}(\Delta t)^2 - \frac{1}{6}\mathbf{a}_k^{(n-1)}(\Delta t)^2. \tag{348}$$

7. Perform step 2) to step 5) with $\mathbf{s}_k^{(n+1)}$ given, and obtain $\mathbf{a}_k^{(n+1)}$ (this is a part of Beeman's algorithm).

8. Update the velocity for all k, by Beeman's algorithm:

$$\mathbf{v}_k^{(n+1)} = \mathbf{v}_k^{(n)} + \frac{1}{3}\mathbf{a}_k^{(n+1)}\Delta t + \frac{5}{6}\mathbf{a}_k^{(n)}\Delta t - \frac{1}{6}\mathbf{a}_k^{(n-1)}\Delta t. \tag{349}$$

9. Move time forward: $t = t + \Delta t$. Set $n = n + 1$.

10. Repeat step 2) to step 9) as long as we need.

Note that, it is common to write the potential energy as a function of nuclear configuration, rather than the nuclei position (i.e. the potential value is the same for the same configuration, regardless where the system is). This means that we can write the nuclear potential energy as $\bar{V}(\mathbf{h}^{(n)})$, for the nuclear configuration $\mathbf{h}^{(n)}$ (i.e. distances and/or angles between nuclei) associated with the nuclei position $\mathbf{s}^{(n)}$. The force field acting on the nucleus k, as

calculated by Eq. (346), can then be computed using the chain rule of calculus:

$$\frac{\partial}{\partial \mathbf{s}_k}\bar{V}(\mathbf{s}^{(n)}) = \frac{\partial}{\partial \mathbf{s}_k}\bar{V}(\mathbf{h}^{(n)}) = -\frac{\partial}{\partial \mathbf{h}}\bar{V}(\mathbf{h}^{(n)}) \times \left.\frac{\partial \mathbf{h}}{\partial \mathbf{s}_k}\right|_{\mathbf{s} = \mathbf{s}^{(n)}} . \tag{350}$$

Thus, we write the force field acting on the system:

$$\mathbf{F}(\mathbf{h}^{(n)}) = -\frac{\partial}{\partial \mathbf{h}}\bar{V}(\mathbf{h}^{(n)}) = -\left.\frac{\partial}{\partial \mathbf{h}}\bar{V}(\mathbf{h})\right|_{\mathbf{h} = \mathbf{h}^{(n)}} . \tag{351}$$

This molecular dynamics procedure is generally called *ab initio quantum dynamics* (or direct dynamics). Since it involves an optimization process in every iteration, the method is very time-consuming. Although involving with many approximations to obtain the nuclear potential energy $\bar{V}(\mathbf{h})$ and the force field $\mathbf{F}(\mathbf{h})$, they are currently considered to be the most accurate simulations. Note that the process to obtain the nuclear potential energy, i.e. Eq. (330), Eq. (341) and Eq. (347), and the force field, i.e. Eq. (344) and Eq. (351), at a certain nuclear configuration, is called *ab initio quantum calculation.*

Another method, which is much faster than direct dynamics, is to create an analytic empirical nuclear potential energy surface. There are several well-known surfaces available, e.g. the Tersoff model for Carbon [Ters88] and for Silicon [Ters89], the Finnis-Sinclair model for metals and alloys [FiSi84], or the Morse potential for covalently bonded diatomic molecules [Mors29]. Molecular dynamics is then performed on the empirical surface, where the forces can be obtained by calculating the analytic derivatives of the surface with respect to the distances and/or angles between nuclei.

Another approach is to evaluate the nuclear potential energy $\bar{V}$ and the force field $\mathbf{F}$ only at the nuclear configurations of interest. The energy surface can be then fitted based

on the data. Subsequently, the molecular dynamics can be performed on the fitted surface. However, it is not easy to determine which part of the configuration hyperspace we should evaluate. This problem can be overcome by first performing the molecular dynamics process on an analytic empirical surface, then select only the configurations of interest. After obtaining the configurations of interest, the quantum calculation is then performed only for these configurations. Then, based on the data from the quantum calculation, the energy surface can be created by nonlinear regression. Once the surface is created, the dynamics can be performed. It should be noted that the accuracy of the dynamics depends on the force field $\mathbf{F}$, since the force determines the future configuration of the system.

Since the MFNN is a universal approximator [HoSt89], it can be used to create the energy surface based on the data from quantum calculations. Let $\bar{V}_{NN}(\mathbf{h})$ denote the energy surface formed by a neural network, whose inputs are the nuclear configuration. Several authors, for example [RaMa05], [AgSa05] and [AgRa06], used neural networks trained by a standard training algorithm to produce the energy surface, by fitting only on the potential energy. To perform dynamics, the force field in [AgRa06] was approximated by numerical differentiation on the surface $\bar{V}_{NN}(\mathbf{h})$.

In this research, we are using neural networks to fit not only the potential energy, but also the force field simultaneously. The conditions that allow us to do this were discussed in Chapter 2. We will then compare the approximation accuracy of the energy surface obtained from neural networks trained by $GNBR$, $CFDA-BFGS$ and $CFDA-LM$, where the force field is computed by differentiating the energy surface network $\bar{V}_{NN}(\mathbf{h})$

with respect to the distances and/or angles, i.e. $\mathbf{F}_{NN}(\mathbf{h}) = -\partial \bar{V}_{NN}(\mathbf{h}) / \partial \mathbf{h}$. See Eq. (114) in Chapter 4 for how to differentiate the networks. This means that we are going to compare the accuracy of neural networks trained by the three algorithms in approximating the potential energy (i.e. function) and the force fields (i.e. the negative of the first-order derivatives). Note that we will not perform dynamics in this research, as the goal is to approximate the energy surfaces by neural networks. To perform dynamics on the created energy surfaces, the potential energy can be obtained by evaluating $\bar{V}_{NN}(\mathbf{h})$. The forces can be computed by using Eq. (350) and Eq. (351), with $\bar{V}(\mathbf{h})$ replaced by $\bar{V}_{NN}(\mathbf{h})$ and $\mathbf{F}(\mathbf{h})$ replaced by $\mathbf{F}_{NN}(\mathbf{h})$. It should be noted that the accuracy of dynamics depends on the approximation accuracy of the energy surface (i.e. the potential energy and the force fields). In this experiment, we will approximate the energy surfaces of three compounds: Si_3, Si_5 and H_2Br.

In the next section, we will illustrate the accuracy of the potential energy and the force field approximated by neural networks trained with $GNBR$, $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$ for the three different molecular systems: Si_3, Si_5 and H_2Br.

Simulation results

This section is divided into two main parts. First, the approximation accuracy of the energy surface and the force field produced by neural networks trained by five training algorithms will be compared on three molecular dynamics problems, which are Si_3, Si_5 and H_2Br. The five training algorithms are $GNBR$ (see Chapter 2), $CFDA-BFGS$ (see

Chapter 4), $CFDA-LM$ (see Chapter 5) and the $CFDA$ methods with the pruning algorithm discussed in Chapter 6, i.e. $CFDA-BFGS_p$ and $CFDA-LM_p$. As mentioned in Chapter 7, the unknown factor ρ in the $CFDA$ performance index is set according to Eq. (305) and Eq. (306) with $\lambda = 10^4$. Note that the $CFDA$ retraining process (i.e. after pruning) is terminated either when the total performance index is convergent or it reaches the same level achieved before pruning. Second, we will illustrate an example to show the $CFDA$ overfitting and how the pruning algorithm eliminates it.

Approximation accuracy

Compound Si_3

For Si_3, a molecular system consists of three silicon atoms. Since it is a three-body system, the coordinates specifying the system's configuration could be the bond distances and angle between the three atoms, i.e. r_1, r_2 and θ. Therefore, the energy surface $\overline{V}$ for this compound is a function of these three input variables. The following figure shows a silicon molecule, along with the distances and angle used in defining a configuration.

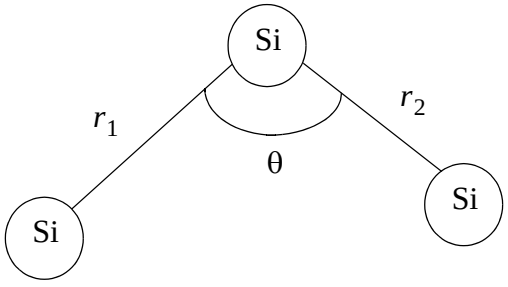

Figure 46) Configuration of three-body silicon

The configurations of interest were first drawn by performing molecular dynamics on the Tersoff model [Ters89]. Then, a novelty sampling method [RaMa05] is used to collect

more configurations. Once obtaining the final set of configurations, the potential energy and the force field are computed from the Tersoff model. The total number of configurations in this case is 10,000. Note that, in this case, since the data were obtained from the empirical potential energy surface (not from *ab initio* quantum calculations), this is equivalent to fitting neural networks to an analytic function.

We created two test conditions. The first case randomly samples $Q = 3,000$ configurations for the training set, and the rest is for testing. The second case randomly samples $Q = 1,500$ configurations for training, and the rest is for testing. Since we are using neural networks to approximate a function of three input variables, the input vector entering the network is three. The network structure for this compound is chosen to be $3 - 100 - 1$. The following tables show the approximation accuracy in RMSE of the energy surface approximated by neural networks trained by three different algorithms. Note that, in each case, the network trained by each training algorithm was initialized with the same network parameters, using the method proposed in [NgWi90]. Recall again that $RMSE_F$ is the function (or the potential energy) RMSE, and $RMSE_D$ is the first-order derivative (or the force field) RMSE. The $RMSE_F$ is given in electron volts (eV). However, there is no unit for $RMSE_D$, since some derivatives have units of electron volts per angstrom ($eV/\mathring{A}$) and other derivatives have units of electron volts per radian (eV/rad). Note also that the range of the potential energy is in between -5.31 and -4.74 eV, whereas the range of the force fields is between -1.73 and 2.28 $eV/\mathring{A}$, and between -1.06 and 0.40 eV/rad.

Training Algorithm	$Q = 3,000$			
	$RMSE_F$		$RMSE_D$	
	Training	Testing	Training	Testing
$GNBR$	2.86E-07	2.97E-07	4.08E-06	4.74E-06
$CFDA-BFGS$	2.91E-07	2.87E-07	3.97E-07	4.53E-07
$CFDA-BFGS_p$	2.91E-07	2.87E-07	3.92E-07	4.53E-07
$CFDA-LM$	2.91E-07	2.87E-07	2.90E-07	3.05E-07
$CFDA-LM_p$	2.91E-07	2.87E-07	2.90E-07	3.03E-07

Table 19 Approximation RMSE of three-body silicon, 3000 training points

Training Algorithm	$Q = 1,500$			
	$RMSE_F$		$RMSE_D$	
	Training	Testing	Training	Testing
$GNBR$	2.68E-07	3.69E-07	1.09E-05	1.68E-05
$CFDA-BFGS$	2.84E-07	2.91E-07	4.42E-07	1.23E-06
$CFDA-BFGS_p$	2.83E-07	2.91E-07	4.46E-07	9.77E-07
$CFDA-LM$	2.83E-07	2.90E-07	2.81E-07	3.63E-07
$CFDA-LM_p$	2.82E-07	2.90E-07	2.81E-07	3.16E-07

Table 20 Approximation RMSE of three-body silicon, 1500 training points

Training Algorithm	S^1	
	$Q = 3,000$	$Q = 1,500$
$CFDA-BFGS_p$	64/100	69/100
$CFDA-LM_p$	63/100	69/100

Table 21 Number of neurons after pruning, for three-body silicon

From the results shown in Table 19 and Table 20, we can see that both of the $CFDA$ methods, i.e. $CFDA-BFGS$ and $CFDA-LM$, produced smaller approximation errors than $GNBR$ on the test set for both the function and the derivatives, for both cases.

The derivative test errors obtained from the $CFDA$ methods, for both cases, were at least one order of magnitude less than the errors from $GNBR$. The pruning algorithm yielded slightly lower approximation errors than the regular $CFDA$ methods. The small difference in errors between the regular $CFDA$ and the $CFDA$ with pruning indicates two phenomena. First, there was no severe overfitting. Second, the overfitting may occur, but we do not have test points over the region of overfitting (since that the area of overfitting is tiny). Table 21 shows the number of neurons after pruning the network for each case. We conclude that the $CFDA$ methods (with or without pruning) provided more accurate energy surfaces and force fields than $GNBR$. In addition, $CFDA - LM_p$ is the most accurate method.

Compound Si_5

The Si_5 system consists of five silicon atoms. For the system of five atoms, a configuration is defined by the four bond distances (i.e. r_{12}, r_{13}, r_{14} and r_{15}), the three angles (i.e. θ_{312}, θ_{412} and θ_{512}), and the two dihedral angles (i.e. Θ_{4123} and Θ_{5124}). Note that θ_{312} is the angle between the $3-1$ bond and the $1-2$ bond, and the dihedral angle Θ_{4123} is the angle between the plane formed by the atoms $4-1-2$ and the plane formed by the atoms $1-2-3$. The configurations of interest were first collected by performing molecular dynamics on the Tersoff model [Ters89]. After the modified novelty sampling technique [RaMa05] is used, the final set of configurations is obtained. Then, the potential energy and the forces were computed from *ab initio* quantum calculation for these configurations, using the Gaussian 03 program [FrTr04]. The total number of configurations in this case is

10,000. The following figure shows the structure of the five-body silicon system, where each circle represents a silicon atom.

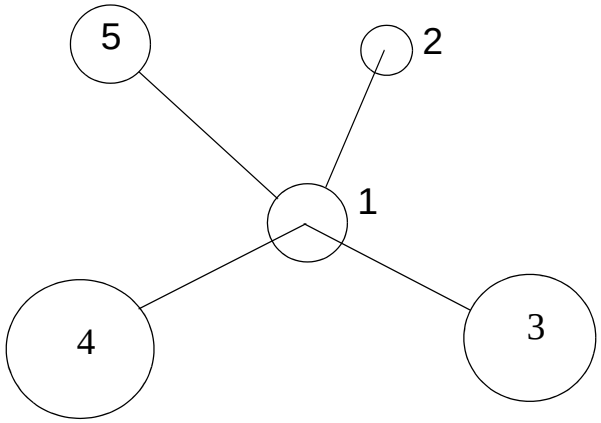

Figure 47) Configuration of five-body silicon

We created two test conditions. The first case randomly samples $Q = 3,000$ configurations for the training set, and the rest is for testing. The second case randomly samples $Q = 1,500$ configurations for training, and the rest is for testing. Since we are using neural networks to approximate a function of nine input variables, the input vector entering the network is nine. The network structure for this compound is chosen to be $9 - 45 - 1$, following [RaMa05]. The network trained by each algorithm was initialized to have the same network parameters, using the method in [NgWi90]. The following tables show the approximation accuracy in RMSE of the potential energy (in eV). Again, there is no unit for the RMSE of the force fields (as $eV/\mathring{A}$ and eV/deg were mixed in the RMSE computation).

Note that the range of the potential energy is between -15.02 and $-14.20\ eV$. The range of the force fields is -1.97 to $1.38\ eV/\mathring{A}$, and -9.86×10^{-2} to $8.23 \times 10^{-2}\ eV/deg$.

Training Algorithm	$Q = 3,000$			
	$RMSE_F$		$RMSE_D$	
	Training	Testing	Training	Testing
$GNBR$	8.18E-05	1.00E-04	1.75E-03	1.76E-03
$CFDA-BFGS$	1.23E-04	1.23E-04	8.09E-04	8.20E-04
$CFDA-BFGS_p$	1.23E-04	1.23E-04	8.08E-04	8.19E-04
$CFDA-LM$	1.16E-04	1.15E-04	7.16E-04	7.27E-04
$CFDA-LM_p$	1.18E-04	1.18E-04	7.36E-04	7.50E-04

Table 22 Approximation RMSE of five-body silicon, 3000 training points

Training Algorithm	$Q = 1,500$			
	$RMSE_F$		$RMSE_D$	
	Training	Testing	Training	Testing
$GNBR$	7.40E-05	1.49E-04	2.31E-03	2.45E-03
$CFDA-BFGS$	1.36E-04	1.37E-04	9.69E-04	1.01E-03
$CFDA-BFGS_p$	1.36E-04	1.36E-04	9.63E-04	1.01E-03
$CFDA-LM$	1.13E-04	1.21E-04	7.57E-04	8.16E-04
$CFDA-LM_p$	1.17E-04	1.23E-04	7.56E-04	8.10E-04

Table 23 Approximation RMSE of five-body silicon, 1500 training points

Training Algorithm	S^1	
	$Q = 3,000$	$Q = 1,500$
$CFDA-BFGS_p$	44/45	40/45
$CFDA-LM_p$	43/45	42/45

Table 24 Number of neurons after pruning, for five-body silicon

From the results shown in Table 22 and Table 23, we can see that the function test errors obtained from all training algorithms were similar, for both cases. However, the derivative test errors obtained from the $CFDA$ methods were lower than the errors from $GNBR$, for both cases. The approximation errors from the pruning method were also in the same level as the regular $CFDA$ methods. Again, this implies that we may not have overfitting, or the overfitting may occur but the test points were not over the overfitting areas. Table 24 shows the number of neurons after pruning the network for each case. We conclude that the $CFDA$ methods (with and without pruning) yield more accurate energy surfaces than $GNBR$. For this compound, $CFDA-LM$ yielded the most accurate function approximation, whereas $CFDA-LM_p$ produced the most accurate derivative approximation.

Compound H_2Br

The H_2Br molecular system is a three-body system. It consists of two Hydrogen atoms and one Bromine atom. As a three-body system, the coordinates defining a configuration could be either two bond distances and one angle, like Figure 46), or three bond distances between the three atoms, i.e. r_1, r_2 and r_3 as shown in Figure 48). (Note that the conversion between the two coordinate systems is performed through the law of cosines: $r_3^2 = r_1^2 + r_2^2 - 2r_1r_2\cos\theta$.)

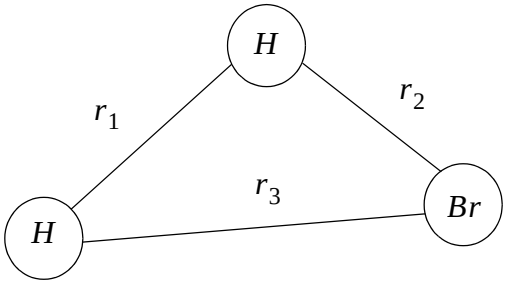

Figure 48) Configuration of H_2Br

We opted for the three bond distances as the coordinates, since the H_2Br empirical potential energy $\bar{V}$, proposed by Kuntz *et. al.* [KuNe66], is a function of the three bond distances. The parameters in the Kuntz's model were proposed by [SuRa84]. (The detail of the Kuntz's model is in Appendix C.)

A main problem in generating H_2Br configurations from the Kuntz's model is that the model is of class C^0 (see the proof in Appendix C). This violates the conditions to use neural networks for simultaneously and uniformly approximating both a function and its first derivatives (see Chapter 2). To avoid sampling configurations near the discontinuity in the first-order derivatives, the configurations of interests were first generated by the approach in [PuMa09]. Using the approach, we generated approximately 470,000 configurations. Then, the potential energy and the force field associated with the generated configurations are computed from the model.

Four test conditions are created; each uses a different number of configurations for training, Q. We set $Q = 3,000$, $Q = 1,500$, $Q = 750$ and $Q = 375$ for the four cases. In each case, a Monte Carlo simulation is performed with M trials. We set $M = 10$. In each trial, we randomly sample Q configurations for training and use the rest for testing.

The network structure for this problem is chosen to be $3 - 150 - 1$. The network parameters are then initialized by Nguyen-Widrow algorithm [NgWi90]. The initialized network will be reused for every training algorithm. Once the training for a trial is completed, the approximation accuracy in terms of $RMSE_F$, $RMSE_D$ and $RMSE_{D^2}$ will be computed. The $RMSE_F$ is given in electron volts (eV). The $RMSE_D$ is provided in electron volts per angstrom $(eV/\mathring{A})$, while the unit of $RMSE_{D^2}$ is electron volts per angstrom squared $(eV/\mathring{A}^2)$. The median statistic is used to report the approximation accuracy over the K trials. The range of the potential energy is from -4.73 to -2.72 eV, while the range of the force fields is from -4.61 to 7.67 $eV/\mathring{A}$. The range of the second derivatives of the model is from -6.73 to 76.83 $eV/\mathring{A}^2$. Table 25 to Table 28 show the approximation accuracy of the neural networks trained by the five training algorithms for the four test conditions. Table 29 shows the median number of neurons after pruning the networks for each case.

Training Algorithm	$Q = 3,000$				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	1.01E-04	4.65E-04	1.55E-02	7.71E-03	1.10E-01
$CFDA-BFGS$	2.33E-04	2.71E-04	1.29E-03	2.76E-03	4.95E-02
$CFDA-BFGS_p$	2.80E-04	2.76E-04	1.29E-03	2.51E-03	4.54E-02
$CFDA-LM$	1.39E-04	1.52E-04	5.31E-04	1.69E-03	2.76E-02
$CFDA-LM_p$	1.91E-04	1.64E-04	5.76E-04	1.53E-03	2.67E-02

Table 25 Approximation RMSE of H_2Br, 3000 training points

Training Algorithm	$Q = 1,500$				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	5.16E-05	1.27E-03	8.48E-03	1.39E-02	1.09E-01
$CFDA-BFGS$	2.25E-04	1.12E-03	1.27E-03	7.24E-03	8.18E-02
$CFDA-BFGS_p$	2.52E-04	9.14E-04	1.27E-03	7.13E-03	8.22E-02
$CFDA-LM$	2.07E-04	9.82E-04	8.92E-04	7.49E-03	5.65E-02
$CFDA-LM_p$	3.68E-04	1.05E-03	8.92E-04	7.24E-03	5.70E-02

Table 26 Approximation RMSE of H_2Br, 1500 training points

Training Algorithm	$Q = 750$				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	3.44E-05	3.52E-03	1.68E-02	2.28E-02	2.00E-01
$CFDA-BFGS$	4.76E-04	5.87E-03	1.30E-03	1.93E-02	1.46E-01
$CFDA-BFGS_p$	4.24E-04	3.82E-03	1.30E-03	1.47E-02	1.09E-01
$CFDA-LM$	2.99E-04	2.22E-03	6.94E-04	1.19E-02	9.81E-02
$CFDA-LM_p$	2.46E-04	3.57E-03	6.94E-04	1.40E-02	9.20E-02

Table 27 Approximation RMSE of H_2Br, 750 training points

Training Algorithm	$Q = 375$				
	$RMSE_F^{md}$		$RMSE_D^{md}$		$RMSE_{D^2}^{md}$
	Training	Test	Training	Test	
$GNBR$	8.81E-05	2.13E-02	9.08E-02	1.03E-01	7.72E-01
$CFDA-BFGS$	2.20E-03	3.57E-02	7.04E-04	2.75E-01	3.11E+00
$CFDA-BFGS_p$	1.33E-03	1.41E-02	8.38E-04	9.54E-02	1.28E+00
$CFDA-LM$	3.28E-03	1.45E-02	5.93E-04	8.22E-02	7.91E-01
$CFDA-LM_p$	2.19E-03	1.16E-02	7.56E-04	6.53E-02	6.42E-01

Table 28 Approximation RMSE of H_2Br, 375 training points

Training Algorithm	S^1			
	$Q = 3,000$	$Q = 1,500$	$Q = 750$	$Q = 375$
$CFDA-BFGS_p$	136/150	141/150	136/150	108/150
$CFDA-LM_p$	138/150	144/150	137/150	126/150

Table 29 Number of neurons after pruning, for H_2Br

From Table 25 to Table 27, the approximation errors obtained from $CFDA$ methods were lower than $GNBR$. In addition, the errors of the pruned networks were even lower than the $CFDA$ networks with no pruning. These results were consistent with the previous cases. However, one may observe that, in the case of $Q = 375$ (see Table 28), the errors from $CFDA-BFGS$ and the second derivative errors from $CFDA-LM$ were higher, compared to $GNBR$. We investigated these results and found that the major contribution to larger approximation errors in the $CFDA$ methods come from some data points outside the region of training data, i.e. extrapolation. Neural networks have no capability to correctly predict the function characteristics outside the region of training data, and the approximation errors at extrapolation is unpredictably large.

Thus far, there have been methods developed for detecting extrapolation. However, it is not the main objective of this research to have the best method to detect extrapolation. We only want to rule out some extrapolation effects that may mislead the result comparison, like Table 28. We detected and removed extrapolations by visual inspection, using the following steps:

1. Collect the test points yielding large first derivative errors (we define "large" as first derivative errors greater than $0.5\ eV/\mathring{A}$) from a network trained by $GNBR$. Put these points into the set X_{BR}^{k}.

2. Plot the location of the test points in X_{BR}^{k}. Visually inspect the locations of the test points to see whether or not they are outside the training region. If there are test points inside the training region, exclude them from X_{BR}^{k}.

3. Repeat step 1) and 2) for the networks trained by $CFDA-BFGS$ (create the set X_{BFGS}^{k}) and $CFDA-LM$ (create the set X_{LM}^{k}).

4. Combine the three sets, i.e. X_{BR}^{k}, X_{BFGS}^{k} and X_{LM}^{k}. Put them in the set X^{k}.

5. Remove the test points in X^{k} from the original test set for this trial.

Keep in mind that these five steps do not guarantee the removal of every single extrapolation point. It only eliminates some test points that may mislead the comparison of the approximation accuracy for interpolation (i.e. points inside the training region). An example of the extrapolation points in a trial, using these five steps, is illustrated in Figure 49). The symbol $\times$ represents the training points for this trial. The symbol $\bullet$ represents the points with large derivative errors.

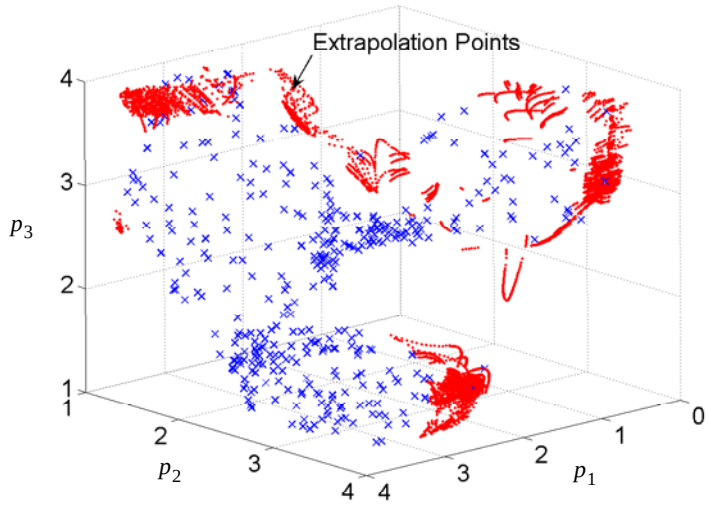

Figure 49) Extrapolation in a Monte Carlo trial, with $Q = 375$

After performing the steps to eliminate some extrapolation, we recalculated the approximation errors on the new test set. Table 30 shows the median approximation test errors after removing extrapolation (for $Q = 375$). Compare this with Table 28.

Training Algorithm	$Q = 375$		
	$RMSE_F^{md}$	$RMSE_D^{md}$	$RMSE_{D^2}^{md}$
$GNBR$	6.99E-03	5.19E-02	6.27E-01
$CFDA-BFGS$	3.10E-03	2.54E-02	5.54E-01
$CFDA-BFGS_p$	1.24E-03	1.16E-02	2.12E-01
$CFDA-LM$	4.24E-03	3.07E-02	3.96e-01
$CFDA-LM_p$	1.59E-03	1.28E-02	1.88E-01

Table 30 Approximation RMSE of H_2Br, after some extrapolation elimination

We can see from Table 30 that the errors from the $CFDA$ methods are now lower than $GNBR$. They were higher in Table 28. Another interesting issue is the pruning method improved the approximation accuracy, indicating that the error improvement also occurs at in-

terpolation points. The results in Table 30 are now consistent with the other problems. In this case, the $CFDA-BFGS_p$ produced the lowest function and first derivative approximation errors, while $CFDA-LM_p$ yielded the most accurate second derivative approximation. Figure 50) illustrates the error comparison for every Monte Carlo trial between $GNBR$, $CFDA-LM$ and $CFDA-LM_p$ for $Q = 375$, after extrapolation removal.

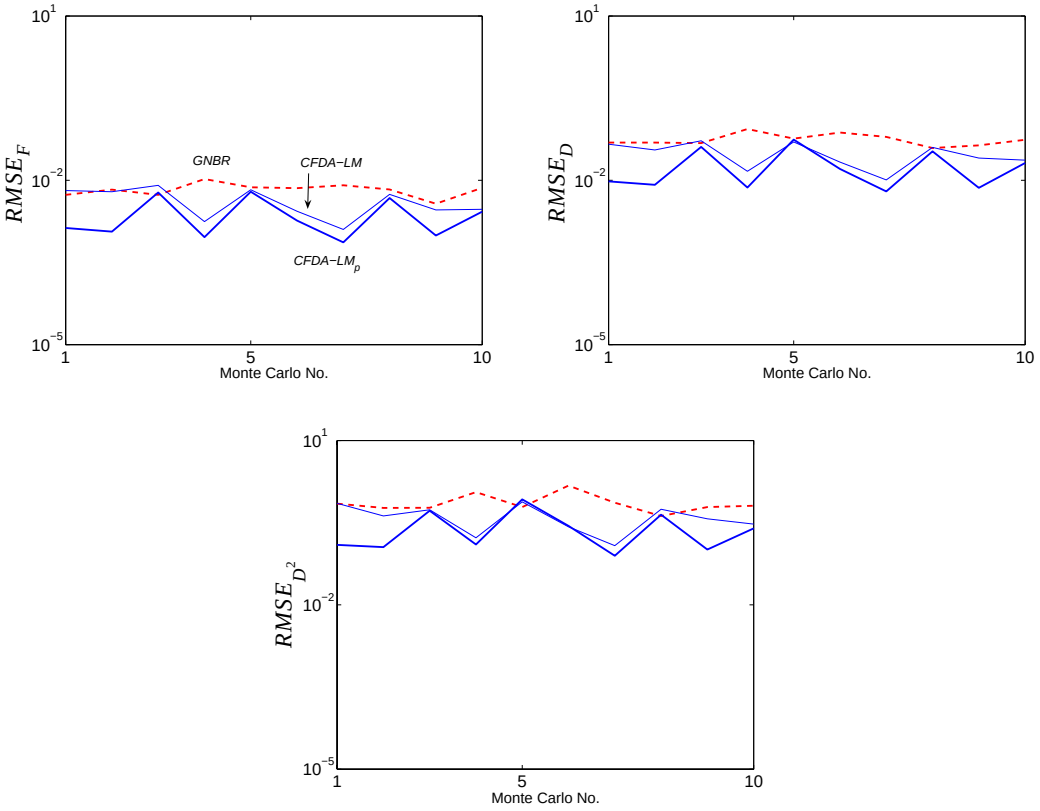

Figure 50) Error comparison for H_2Br with $Q = 375$

From Figure 50), we can see that the approximation errors from the $CFDA$ method were consistently lower than $GNBR$ for almost every trial. In addition, the pruning algorithm also improved the accuracy for the $CFDA$ method in every trial.

In the next section, we will show an example illustrating the $CFDA$ overfitting in molecular dynamics and how the pruning algorithm gets rid of it.

Elimination of CFDA Overfitting

We will illustrate an example of the $CFDA$ overfitting and how the pruning algorithm eliminates it in H_2Br problem. This example is selected from a network (with $Q = 375$) trained with the $CFDA-BFGS$ algorithm.

The pruning algorithm indicated that 21 neurons are to be pruned in this network. However, we will demonstrate the two types of overfitting developed from only three neurons. Figure 51) shows the input spaces in a three-dimensional plot, where $\times$ represents the training set. In the case of three-dimensional inputs, the neuron center $({}_i\mathbf{w}^1)^T\mathbf{p} + b_i^1 = 0$ becomes a plane. The centers of the three neurons are also presented in the figure.

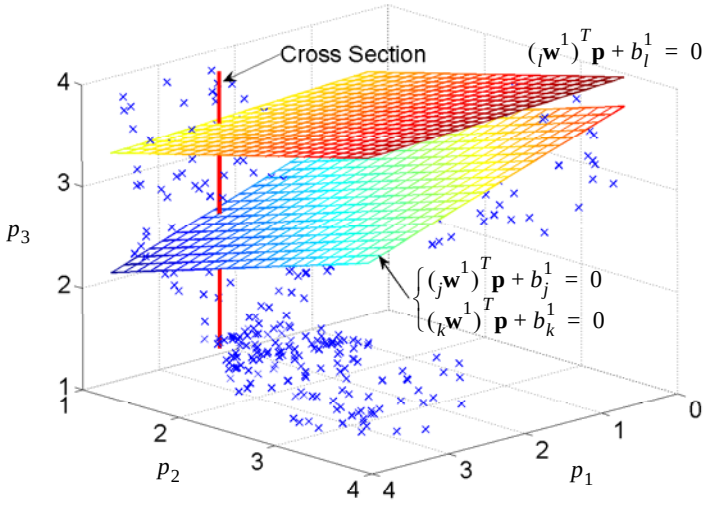

Figure 51) The three neuron centers

Neuron j and k develop $Type-A$ overfitting, while neuron l causes $Type-B$. In order

to easily visualize the overfitting, we will take a cross section of the network response along the cross-section line shown in Figure 51). The cross section is at $p_1 = 2.73$, $p_2 = 1.42$ and $1.27 \le p_3 \le 4$ angstroms (Note that this is associated with $p_1 = 0.25$, $p_2 = -0.9$ and $-1 \le p_3 \le 1$ in the normalized input space). We will first illustrate $Type-A$ overfitting, followed by $Type-B$ overfitting, using the three neurons. Then, we will show the final outcome.

Type A

The network function and derivative responses (with respect to p_3) of neuron j and k along the cross-section line are shown in Figure 52).

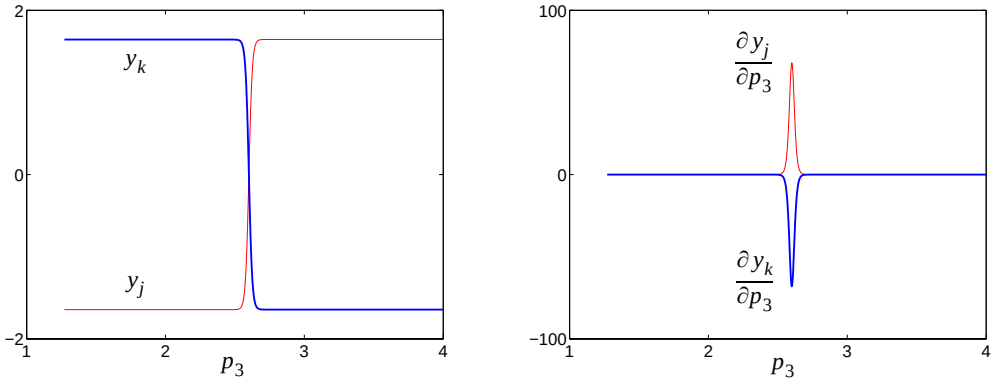

Figure 52) Function and derivative responses of the two neurons along the cross section

When combining the responses of these two neurons, we obtain Figure 53).

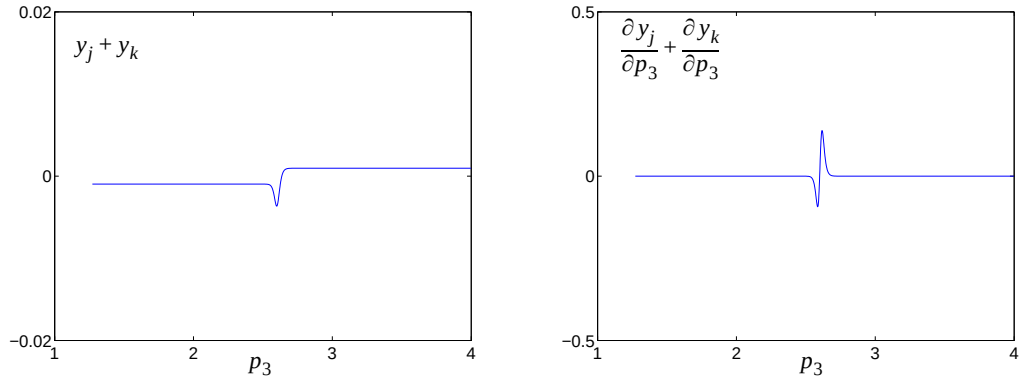

Figure 53) The combined responses

From Figure 53), we can see that the responses from the two neurons cancel almost everywhere, except at a tiny region close to their centers. Figure 54) illustrates the function and derivative errors (with respect to p_3) before and right after pruning the network (without retraining).

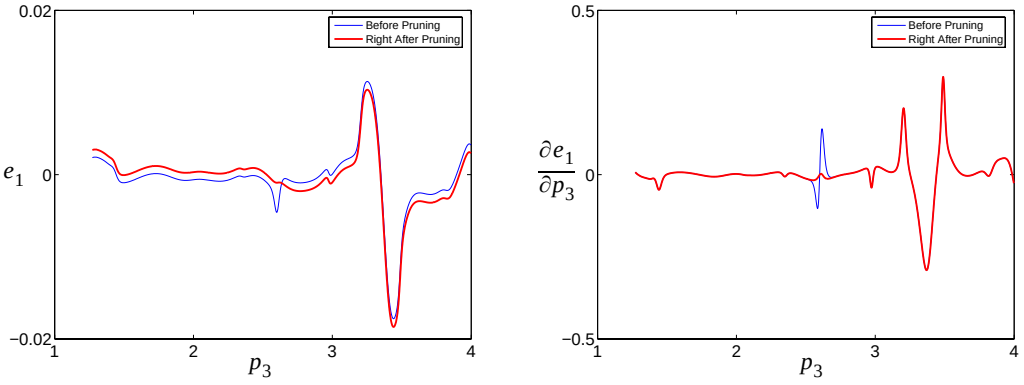

Figure 54) Errors before and right after pruning the two neurons

Figure 54) shows that the spikes of the function and derivative errors, which were observed before pruning the neurons, are eliminated.

Next, we will illustrate $Type - B$ overfitting.

Type B

Along the cross section, the function and derivative responses of neuron l are shown in Figure 55).

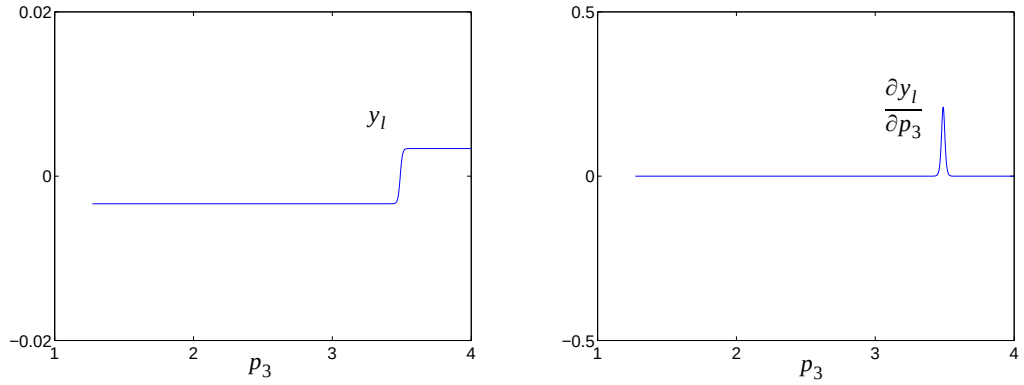

Figure 55) Function and derivative response of the neuron

The function and derivative errors before and right after pruning the neuron (with no re-training) are illustrated in Figure 56).

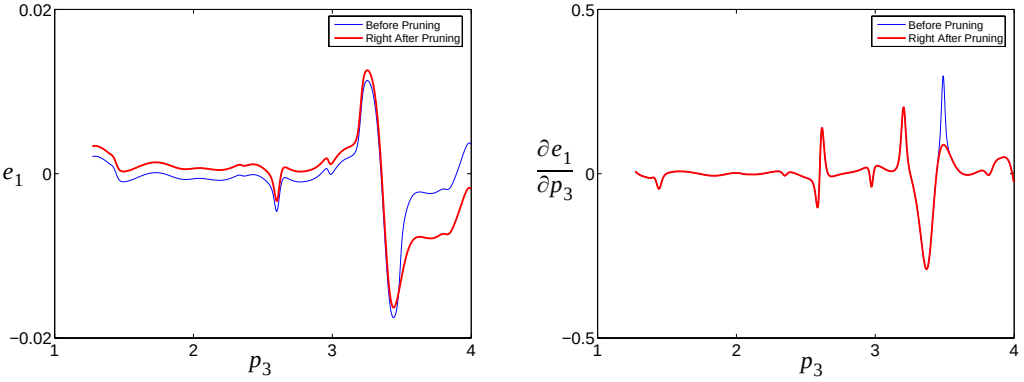

Figure 56) Errors before and right after pruning the neuron

We can see from Figure 56) that the spike in the derivative errors, which were observed in the original errors, disappears once pruning the neuron.

Next, we will show the final outcome.

Final outcome

The pruning algorithm indicated the elimination of 21 neurons. Figure 57) compares the errors before and right after pruning all of the 21 neurons.

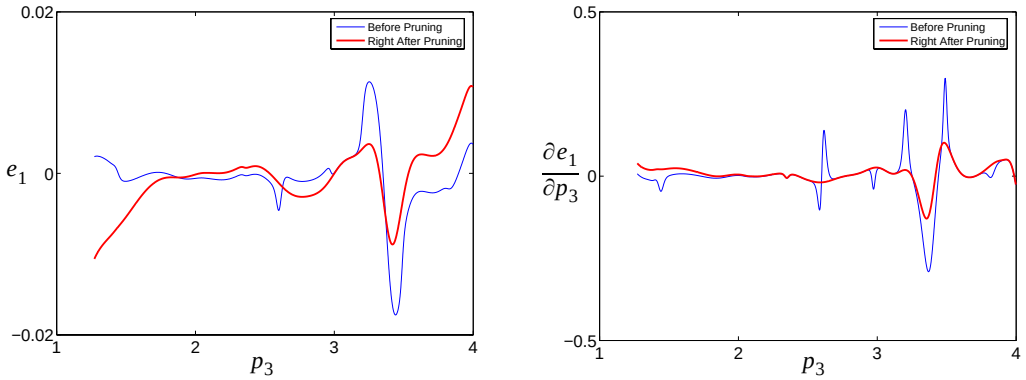

Figure 57) Errors before and right after pruning the 21 neurons

After pruning the 21 neurons and $CFDA$ retraining the network, the errors before and after applying the pruning algorithm are shown in Figure 58).

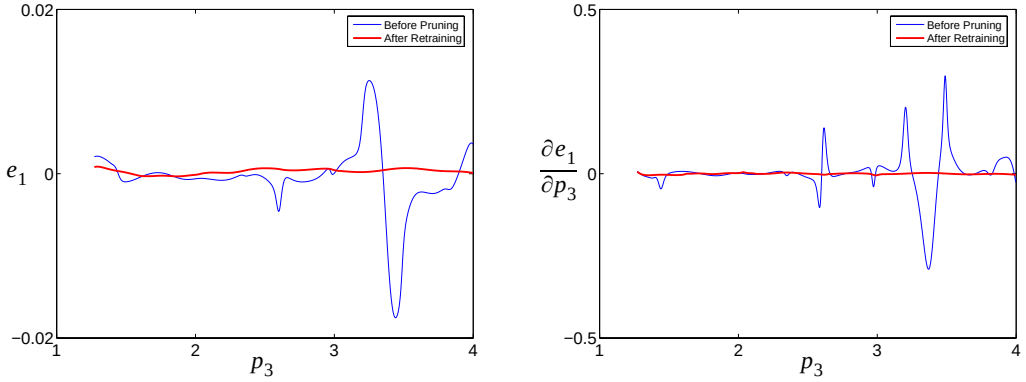

Figure 58) Errors before and after performing the pruning algorithm

From Figure 58), it is clear that the function and derivative errors with the pruning algorithm are much smaller and smoother. The error plots in Figure 58) imply that the second derivative approximation from the pruned network would be also more accurate. This is another example illustrating that the pruning algorithm successfully improves the approxima-

tion accuracy for the function, first derivative and second derivatives of neural networks trained by the *CFDA* methods.

Summary

In this chapter, we reviewed the general concept of molecular dynamics. This is an application in which it is important to approximate both a function and its first-order derivatives. We started the review by describing classical mechanics. The Newtonian or Hamiltonian equations of motion are used to determine the motion of particles.

However, to explain the behavior of atomic-level particles, the Schrödinger equation is used. For time-independent problems including molecular dynamics, the stationary-state Schrödinger equation is used, where the wave function and the potential energy are independent of time. It is impossible to obtain analytic solutions for a general system. To obtain approximate solutions, the Born-Oppenheimer approximation is used. The electronic energy is first solved at a fixed nuclear configuration. Then, given the electronic energy, the nuclear energy at the nuclear configuration is solved. To solve for the nuclear energy, the nuclear wave function must be known. Unfortunately, since the nuclear wave function is unknown, we replace it with a mathematical model. By using the Rayleigh-Ritz variational principle, we obtain the approximated nuclear energy at a fixed nuclear configuration.

In molecular dynamics, we treat the motion of nuclei using classical mechanics. To calculate the motion of nuclei using the Newtonian equations of motion, we need to know the force fields acting on the nuclei. This can be computed by taking the negative of the

first-order derivatives of the potential energy with respect to the nuclei position. Once obtaining the forces, the acceleration on the nuclei can be computed. Subsequently, the velocity and the position of the nuclei can be updated by performing numerical integration. We summarized the general procedure of molecular dynamics, using the Beeman's algorithm for numerically integrating the Newtonian equations of motion.

Then, we discussed how neural networks can be used in molecular dynamics. We explained that, since the force is the negative first-order derivatives of the potential energy, neural networks can be used to predict both the potential energy and the force fields. We compared the approximation accuracy of the potential energy and the force fields obtained by neural networks trained by five algorithms: $GNBR$, $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$, in three molecular dynamics problems: Si_3, Si_5 and H_2Br. The results showed that the $CFDA$ methods produced more accurate potential energy surfaces and force fields than $GNBR$. In addition, the approximation accuracy in the regular $CFDA$ methods could be improved by applying the pruning algorithm. The most accurate method was $CFDA-LM_p$. Finally, we illustrated the $CFDA$ overfitting in molecular dynamics and how the pruning algorithm removed it, through an example. The results in this chapter also indicate that $CFDA$ overfitting occurs only rarely in molecular dynamics problems. $CFDA$ overfitting appears to occur less often as the dimension of the input space increases.

CHAPTER 9

CONCLUSIONS

Objective

In this chapter, we will present the summary of the study, and also discuss the future work.

Summary

The objective of this research was to develop training algorithms for neural networks to fit both a function and its first derivatives. We also wanted to compare the approximation accuracy obtained from the new training methods with the standard training methods (which are used to fit only the function). The new training algorithms are $LM-ES1$ and $LM-ES2$ (Chapter 3), $CFDA-BFGS$ (Chapter 4), $CFDA-LM$ (Chapter 5), and the $CFDA$ methods with the pruning algorithm, i.e. $CFDA-BFGS_p$ and $CFDA-LM_p$ (Chapter 6). The standard training methods are $BFGS-ES$, $LM-ES$ and $GNBR$.

We first reviewed the general concept of neural networks and the three standard training algorithms used for function approximation. Then, the conditions under which neural networks can simultaneously and uniformly approximate a function and its first derivatives were presented. The conditions require that the true function and its first deriva-

tives be continuous, while the transfer function in the hidden layer of the neural networks be sufficiently differentiable and the transfer function in the output layer be linear. These materials were discussed in Chapter 2.

To approximate both a function and its first derivatives, we first introduced two validation-related methods in Chapter 3. These two methods use the standard backpropagation algorithm with early stopping. The first method, i.e. $LM-ES1$, modified the validation performance measure so that it includes two terms: the function error and the derivative error of the validation set. The derivative error term is also multiplied by a weighting factor. With different values of the weighting factor, the simulation results on approximating analytic functions showed that there was no improvement in the approximation over the standard training methods. In the second method ($LM-ES2$), the validation performance measure is changed from the function errors in the validation set to the derivative errors in the training set. The simulation results on approximating the analytic functions showed that the new validation measure sometimes terminates the training process sooner than the standard early stopping, causing worse approximation.

Another proposed method, to fit both a function and its first derivatives, changes the training performance index so that it contains the squared function errors and the squared derivative errors. The squared derivative error is multiplied by the weighting factor ρ. This method is called Combined Function and Derivative Approximation, or $CFDA$. To minimize the $CFDA$ performance index using any gradient-based optimization method, the gradient of the performance index with respect to the network parameters is required. In Chapter 4, we derived two approaches to calculate the gradient: the batch mode and the

memory-save method. The execution time for computing the gradient of the $CFDA$ performance index under several scenarios was measured, and the results showed that the batch mode took from 0.5 to 370 times as long as the standard method. The memory-save approach took from 0.5 to 6.2 times as long as the standard method. The batch mode performed faster than the memory-save approach, unless the computer's memory overflowed. In this research, we chose $BFGS$ optimization as the gradient-based method of choice. We call this method $CFDA-BFGS$.

The $CFDA$ performance index can also be minimized by the Levenberg-Marquardt algorithm. However, extra calculations for the Jacobian matrix of the derivative error term are needed. In Chapter 5, we derived two approaches to obtain the new Jacobian matrix: the batch mode and the memory-save method. The execution times for computing the gradient and the Jacobian matrix of the derivative error term, under several scenarios, were measured, and the results showed that the batch mode took from 0.9 to 35.4 times as long as the standard Levenberg-Marquardt method. The memory-save approach took from 0.9 to 8.4 times as long as the standard Levenberg-Marquardt method. The batch mode performed slightly faster than the memory-save approach, unless the computer's memory overflowed. We name this method $CFDA-LM$.

Although the $CFDA$ methods force the first derivatives of neural networks to the correct values, new overfitting has been observed. In Chapter 6, we proposed two new types of overfitting: $Type-A$ and $Type-B$. The $Type-A$ overfitting is developed from the responses of more than one neuron. The responses cancel each other at training points, but not the points in between. The $Type-B$ overfitting can be developed from one neuron

or more, and it is caused by a local minimum in the $CFDA$ training surface. We introduced an algorithm to prune neurons producing the overfitting. The pruning algorithm can be applied to any $R-S^1-1$ networks, trained by the $CFDA$ methods with the hyperbolic tangent sigmoid transfer function. By incorporating the pruning algorithm to the two $CFDA$ training methods above, we name them $CFDA-BFGS_p$ and $CFDA-LM_p$.

In Chapter 7, we proposed to set the weighting factor in the $CFDA$ performance index to be $\rho = \lambda/\eta^2$, where η is the ratio of the maximum absolute derivative value to the maximum absolute function value in the training set. We chose $\lambda = 10^4$, since it yielded robust results for many problems. The simulation results on approximating four analytic functions showed that the regular $CFDA$ methods provided more accurate results and better generalization, for the both function and its first derivatives, than the standard training methods. The $CFDA$ methods with pruning produced even better approximation accuracy than the regular $CFDA$ methods. The improvement (for the function, first derivatives and second derivatives) was as high as several orders of magnitude for some problems, where the overfitting was extreme. The results showed that $CFDA-LM_p$ yielded the most accurate approximation and the best generalization (for both the function, first derivatives and second derivatives) among the other tested methods, which are $BFGS-ES$, $LM-ES$, $GNBR$, $LM-ES1$, $LM-ES2$, $CFDA-BFGS$, $CFDA-LM$ and $CFDA-BFGS_p$.

Molecular dynamics is an application where neural networks can be used to fit both a function and its first-order derivatives. Neural networks can be used to construct the po-

tential energy surface, while the negative of the first-order derivatives are the forces. The *ab initio* quantum calculations are used to compute the true potential energy and the forces. Unfortunately, it is very time-consuming to evaluate the true potential energy and the forces for a given input, as it involves an optimization process. Thus, the advantage of using neural networks instead of the *ab initio* function is the reduction in computation time, which can be several orders of magnitude. In Chapter 8, we compared the approximation accuracy of neural networks trained by five training algorithms: $GNBR$, $CFDA-BFGS$, $CFDA-LM$, $CFDA-BFGS_p$ and $CFDA-LM_p$ for three molecular dynamics problems. The simulation results showed that the regular $CFDA$ methods yielded better generalization than $GNBR$. The $CFDA$ algorithms with pruning consistently improved the approximation accuracy when compared to the regular $CFDA$ methods. The outcomes of the three molecular dynamics problems showed that the $CFDA-LM_p$ method provided the most promising results in terms of approximation accuracy (for the function, first derivatives and second derivatives) among the five training methods.

Examples illustrating the $CFDA$ overfitting and how the pruning algorithm eliminates it were demonstrated in both Chapter 7 (for simple analytic functions) and Chapter 8 (for molecular dynamics).

Future work

There is more work to be done. The most critical assumption required to produce the excellent results shown in this research is the availability of the correct values of the function and its first derivatives. Although several real-world applications follow this pre-

sumption, the interest would be even broader if the assumption is more relaxed. Many applications do not assume having the correct values of the function and its first derivatives, since the values may come from imperfect measurements. That is, noise can corrupt the data. However, it is currently not clear how the noise levels in the function and its first derivatives are associated. Understanding this may lead to constructing an appropriate objective function for training neural networks to approximate both the function and its first derivatives under high noise situations.

Although the pruning algorithm produced promising results, it can only be applied to a two-layer network with one output. Generalizing the method so that it can be used with any multilayer feedforward networks with any number of outputs would be desirable. To achieve this, more work is needed to analyze how the overfitting in a layer affects the next layer and how the overfitting is developed in multiple outputs.

APPENDIX

A. PSGEN algorithm

The $psgen$ algorithm is used in the pruning method to form combinations of neurons after a candidate and its neighbors are created. The derivative response for each of these combinations will be evaluated to verify whether or not it provides a significant contribution. We will describe the $psgen$ algorithm in such a way that it solves a general problem, not specific to the pruning method. However, at the end of this section, we will describe how to apply this algorithm to the pruning method. Note that the algorithm was proposed by [McCa06].

Given a set S with n_S distinct elements (the set S could be an empty set), the $psgen$ algorithm generates an element of the power set of S, denoted by $P(S)$. There exist 2^{n_S} elements in $P(S)$. To generate all of these elements, we first assign an index i_S to each element of S. The index is an integer starting from zero to $n_S - 1$, i.e. $0 \le i_S \le n_S - 1$. For

example, assume the set $S = \{a, b, c\}$. We define the index to each element according to the table below:

Element of S	Element Index i_S
a	0
b	1
c	2

Table 31 Elements of S and their index i_S

For the given set S, the power set of S is

$$P(S) = \{\{\varnothing\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\},$$

where $\varnothing$ denotes the empty set. To represent each element of $P(S)$, we can use a binary sequence with the length of n_S (where the least significant bit is the zeroth and the most significant bit is the $(n_S - 1)^{th}$). This results in 2^{n_S} different sequences, where each will be used to represent an element of $P(S)$. The mapping between a sequence and an element of $P(S)$ can be done using the index i_S and a new set, called the *combinadic*.

A combinadic is a set containing the bit location of the ones in the binary sequence. (Thus, elements in a combinadic range from zero to $n_S - 1$.) Once we have a combinadic, the associated element of $P(S)$ can be realized by using the assigned index i_S. In other words, the elements in a combinadic are the indices i_S. Therefore, by mapping the index i_S with the element of S, the element of $P(S)$ can be generated. For our example, Table

32 illustrates the mapping from the binary sequence to the associated elements of $P(S)$ through the combinadic, using the index i_S defined in Table 31.l

Binary Sequence	Combinadic	Element of $P(S)$
000	$\{\varnothing\}$	$\{\varnothing\}$
001	$\{0\}$	$\{a\}$
010	$\{1\}$	$\{b\}$
011	$\{1, 0\}$	$\{a, b\}$
100	$\{2\}$	$\{c\}$
101	$\{2, 0\}$	$\{a, c\}$
110	$\{2, 1\}$	$\{b, c\}$
111	$\{2, 1, 0\}$	$\{a, b, c\}$

Table 32 The binary sequences, combinadics and the associated elements of $P(S)$

Therefore, once the indices i_S of elements in S are defined, we are instantly able to generate all possible elements of $P(S)$.

Another problem is raised when one wants to generate only some elements of $P(S)$, given that they have exactly k elements from the set S (where $0 \le k \le n_S$). This corresponds to generating the combinadics that contain exactly k elements. Given this condition, there are totally $\binom{n_S}{k}$ possible combinadics. To specify each of these combinadics, we first assign an index i_c to each of them starting from zero to $\binom{n_S}{k} - 1$, i.e. $0 \le i_c \le \binom{n_S}{k} - 1$. The zeroth index is assigned to the combinadic with the smallest value of its associated binary sequence. The largest index is assigned to the combinadic with the highest value of its associated binary sequence. For our example, assume we are interested in generating only

the combinadics with two elements, i.e. $\{1, 0\}$, $\{2, 0\}$ and $\{2, 1\}$. Given the binary sequence in Table 32, Table 33 demonstrates the combinadics and their index i_c,

Combinadic	Combinadic Index i_c
$\{1, 0\}$	0
$\{2, 0\}$	1
$\{2, 1\}$	2

Table 33 Combinadics and their index i_c

With the index i_c, we are also able to introduce a notation to represent each combinadic. We use $M_{(n_S, k)}(i_c)$ to represent the combinadic with k elements, i.e. $\{m_{k-1}, m_{k-2}, \ldots, m_0\}$, assigned by index i_c, generated for the set S (which has n_S elements). Thus, given the binary sequences and combinadics in Table 32, the associated notations $M_{(n_S, k)}(i_c)$ are shown in Table 34.

Binary Sequence	Combinadic	Notation $M_{(n_S, k)}(i_c)$
000	$\{\varnothing\}$	$M_{(3, 0)}(0)$
001	$\{0\}$	$M_{(3, 1)}(0)$
010	$\{1\}$	$M_{(3, 1)}(1)$
011	$\{1, 0\}$	$M_{(3, 2)}(0)$
100	$\{2\}$	$M_{(3, 1)}(2)$
101	$\{2, 0\}$	$M_{(3, 2)}(1)$
110	$\{2, 1\}$	$M_{(3, 2)}(2)$
111	$\{2, 1, 0\}$	$M_{(3, 3)}(0)$

Table 34 Notations for the combinadics

Given a set S, the values of k and i_c, the $psgen$ algorithm can generate the combinadic $M_{(n_S, k)}(i)$. Consequently, once obtaining $M_{(n_S, k)}(i)$, we can easily convert it to the associated elements of $P(S)$ (by mapping from the index i_S to the element of S). We use $P(S)_{k, i}$ to denote the associated element of $P(S)$. Now, we are ready to provide the steps of the $psgen$ algorithm.

Steps for psgen

1. Given the set S, define index i_S to each element of S, where $0 \le i_S \le n_S - 1$. (For example, if the elements of S are numeric, we can first sort these elements in ascending order. Then, assign index zero to the smallest element and index $n_S - 1$ to the largest element.)

2. Initialize $i = 1$. Define $\binom{n}{k} \equiv 0$ if $n < k$. This definition will be used only in step 3 to step 6.

3. Find the largest element m_{k-i}, where $0 \le m_{k-i} \le n_S - 1$ such that $\binom{m_{k-i}}{k} \le i_c$.

4. Set $n_S = m_{k-i}$ and $i_c = i_c - \binom{m_{k-i}}{k}$.

5. If $i_c < 0$, set $i_c = 0$.

6. Set $k = k - 1$.

7. If $k > 0$, set $i = i + 1$ and go back to step 3). Otherwise, go to step 8 with $M_{(n_S, k)}(i_c) = \{m_{k-1}, m_{k-2}, \ldots, m_0\}$.

8. By using the index i_S assigned in step 1 to map from the elements in $M_{(n_S, k)}(i_c)$ to the elements of S, obtain and return $P(S)_{k, i_c}$.

Note that we will not provide the detail of $psgen$ algorithm in pseudo code for the pruning method. However, wherever the $psgen$ algorithm is needed, we will use $psgen(S, k, i_c)$ to denote the function that takes the set S, the values of k and i_c, performs these eight steps, and returns $P(S)_{k, i_c}$.

As previously mentioned, we use the $psgen$ algorithm when we want to form combinations of neurons. That is, once the candidate neuron and its neighbors are established, we will put the neighbors into the set S. By specifying the values of k and i_c, the $psgen$ algorithm will provide the corresponding set of neighbor neurons. Then, a combination of neurons are formed by including the candidate neuron with the set of neighbor neurons. Then, the derivative response generated from this combination of neurons will be evaluated to see whether or not its contribution is significant. We can repeat this process for different values of k and i_c, if necessary.

B. Calculation for the second derivatives of neural networks

We will show the calculation for the second derivatives of neural networks, with respect to the inputs, evaluated at $\mathbf{p}_q$, i.e. $\frac{\partial}{\partial p_{r',q}}\left(\frac{\partial a_{k,q}}{\partial p_{r,q}}\right)$. In Chapter 4, we have shown the calculation for the first derivatives of neural networks with respect to the inputs. That is, recall from Eq. (109) in Chapter 4 that

$$\frac{\partial a_{j,q}^m}{\partial p_{r,q}} = \dot{f}^m(n_{j,q}^m)\sum_{l=1}^{S^{m-1}}\left(w_{j,l}^m \times \frac{\partial a_{l,q}^{m-1}}{\partial p_{r,q}}\right).$$

By taking the derivative of Eq. (109) with respect to $p_{r'}$, we obtain

$$\frac{\partial}{\partial p_{r'}}\left(\frac{\partial a_{j,q}^m}{\partial p_{r,q}}\right) = \frac{\partial \dot{f}^m(n_{j,q}^m)}{\partial p_{r'}}\sum_l\left(w_{j,l}^m \times \frac{\partial a_{l,q}^{m-1}}{\partial p_{r,q}}\right) + \dot{f}^m(n_{j,q}^m)\sum_l\left\{w_{j,l}^m \times \frac{\partial}{\partial p_{r'}}\left(\frac{\partial a_{l,q}^{m-1}}{\partial p_{r,q}}\right)\right\}. \quad (352)$$

From Eq. (352), the only term we have not shown the calculation is $\frac{\partial}{\partial p_{r'}}\left(\frac{\partial a_{l,q}^{m-1}}{\partial p_{r,q}}\right)$. (Note that the calculation for the term $\partial \dot{f}^m(n_{j,q}^m)/\partial p_{r'}$ is expressed in Eq. (144).) Therefore, we will focus on the calculation of the term $\frac{\partial}{\partial p_{r'}}\left(\frac{\partial a_{l,q}^{m-1}}{\partial p_{r,q}}\right)$.

First note that the calculation in Eq. (352) is a forward propagation. Therefore, we need to initialize $\frac{\partial}{\partial p_{r'}}\left(\frac{\partial a_{j,q}^0}{\partial p_{r,q}}\right)$. We have the calculation for $\partial a_{j,q}^0/\partial p_{r,q}$ in Eq. (115):

$$\frac{\partial a_{j,\,q}^{0}}{\partial p_{r,\,q}} = \begin{cases} 1 & ;\text{ if } j = r\,. \\ 0 & ;\text{ if } j \neq r\,. \end{cases}$$

By taking the derivative of Eq. (115) with respect to $p_{r'}$, we obtain

$$\frac{\partial}{\partial p_{r'}}\left(\frac{\partial a_{j,\,q}^{0}}{\partial p_{r,\,q}}\right) = 0\,. \tag{353}$$

Recall that $a_{k,\,q} = a_{k,\,q}^{M}$. Therefore, we obtain the second derivatives of the network $\frac{\partial}{\partial p_{r',\,q}}\left(\frac{\partial a_{k,\,q}}{\partial p_{r,\,q}}\right)$ by propagating Eq. (352) until $m = M$.

C. Class of H_2Br model

We will provide a proof showing that the H_2Br model is of class C^0. That is, the function itself is continuous, while the first-order derivatives are not continuous. The location of the discontinuity in the first-order derivatives will also be specified.

According to [KuNe66] and [SuRa84], the potential energy surface of H_2Br can be written as:

$$\begin{aligned}\bar{V}(r_1, r_2, r_3) \;=\; & Q(r_1) + Q(r_2) + Q(r_3) - [J^2(r_1) + J^2(r_2) + J^2(r_3) \\ & - J(r_1)J(r_2) - J(r_2)J(r_3) - J(r_1)J(r_3)]^{1/2},\end{aligned} \tag{354}$$

where

$$Q(r_i) \;=\; \frac{1}{2}\left\{E_1(r_i) + \frac{(1-a_i)}{(1+a_i)}E_2(r_i)\right\} \text{ and } J(r_i) \;=\; \frac{1}{2}\left\{E_1(r_i) - \frac{(1-a_i)}{(1+a_i)}E_2(r_i)\right\}. \tag{355}$$

The function $E_1(r_i)$ is

$$E_1(r_i) \;=\; D_i[\exp(-2\alpha_i\{r_i - r_{c_i}\}) - 2\exp(-\alpha_i\{r_i - r_{c_i}\})], \tag{356}$$

and the function $E_2(r_i)$ is

$$E_2(r_i) \;=\; \frac{1}{2}D_i[\exp(-2\alpha_i\{r_i - r_{c_i}\}) + 2\exp(-\alpha_i\{r_i - r_{c_i}\})]. \tag{357}$$

The condition for the input space is $r_i > 0$, $\forall i$. The parameters a_i, α_i, D_i and r_{c_i} are specified below.

a_1	a_2, a_3	α_1	α_2, α_3	D_1	D_2, D_3	r_{c_1}	r_{c_2}, r_{c_3}
0.26	0.06	1.027	0.9588	4.7466	3.918	1.402	2.673

Table 35 Parameters of the model

Note that the units for D_i are electron volts, the units for α_i are au^{-1}, and the units for r_{c_i} are au, where au (or atomic unit) equals 0.529 angstroms.

Two major properties of the function are of interest, i.e. continuity and differentiability. We will show that the function $\bar{V}$ is continuous, but it is not differentiable. However, the derivatives $\partial\bar{V}/\partial r_i;\ \forall i$ are neither continuous nor differentiable. First, for ease of reference, define

$$T_Q(\mathbf{r}) \equiv Q(r_1) + Q(r_2) + Q(r_3), \text{ and} \tag{358}$$

$$T_J(\mathbf{r}) \equiv J^2(r_1) + J^2(r_2) + J^2(r_3) - J(r_1)J(r_2) - J(r_2)J(r_3) - J(r_1)J(r_3)\ . \tag{359}$$

Therefore,

$$\bar{V}(\mathbf{r}) \ = \ T_Q(\mathbf{r}) + T_J^{1/2}(\mathbf{r})\,. \tag{360}$$

By taking the derivative of Eq. (360) with respect to r_i, we obtain

$$\frac{\partial\bar{V}}{\partial r_i} \ = \ \frac{\partial T_Q}{\partial r_i} + \frac{1}{2}T_J^{-1/2}\frac{\partial T_J}{\partial r_i}\,. \tag{361}$$

For the second derivatives, we take the derivative of Eq. (361) with respect to r_i, and obtain

$$\frac{\partial^2\bar{V}}{\partial r_i\partial r_j} \ = \ \frac{\partial^2 T_Q}{\partial r_i\partial r_j} - \frac{1}{4}T_J^{-3/2}\frac{\partial T_J}{\partial r_i}\frac{\partial T_J}{\partial r_j} + \frac{1}{2}T_J^{-1/2}\frac{\partial^2 T_J}{\partial r_i\partial r_j}\,. \tag{362}$$

From Eq. (361) and Eq. (362), we can see the locations where we should focus on the continuity and differentiability of the function are at the points yielding $T_J(\mathbf{r}) \ = \ 0$. The inputs $\mathbf{r}$ yielding $T_J(\mathbf{r}) \ = \ 0$ are denoted $\tilde{\mathbf{r}}$. Excluding $\tilde{\mathbf{r}}$, the function $\bar{V}$, $\partial\bar{V}/\partial r_i$ and

$\partial^2 \bar{V} / \partial r_i \partial r_j$ are continuous and differentiable.

We will divide the proof into four parts. First, we will show that $\bar{V}$ is continuous. Second, we will locate $\tilde{\mathbf{r}}$. Third, we will prove that $\partial \bar{V} / \partial r_i$ is not continuous at $\tilde{\mathbf{r}}$. Finally, the differentiability of $\bar{V}$ and $\partial \bar{V} / \partial r_i$ at $\tilde{\mathbf{r}}$ will be discussed.

Proof

A. Continuity of $\bar{V}$

It is easy to prove that $\bar{V}$ is continuous. Since $\bar{V}$ is a function composed of several continuous functions (i.e. exponential functions), consequently $\bar{V}$ is continuous, following a fundamental theorem of calculus. Note that, at $\tilde{\mathbf{r}}$, the values of $\bar{V}$ are simply $T_Q(\tilde{\mathbf{r}})$ (since $T_J(\tilde{\mathbf{r}}) = 0$).

B. Location of $\tilde{\mathbf{r}}$

We will locate $\tilde{\mathbf{r}}$. At $\tilde{\mathbf{r}}$, recall that $T_J(\tilde{\mathbf{r}}) = 0$. Therefore, from Eq. (359), we can write

$$J^2(\tilde{r}_1) + J^2(\tilde{r}_2) + J^2(\tilde{r}_3) - J(\tilde{r}_1)J(\tilde{r}_2) - J(\tilde{r}_2)J(\tilde{r}_3) - J(\tilde{r}_1)J(\tilde{r}_3) = 0. \qquad (363)$$

A possibility to make Eq. (363) hold is when

$$J(\tilde{r}_1) = J(\tilde{r}_2) = J(\tilde{r}_3). \qquad (364)$$

From Table 35, we can see that the parameters for r_2 and r_3 are the same, thus causing $J(r_2) = J(r_3)$ whenever $r_2 = r_3$. Therefore, we can conclude that $\tilde{r}_2 = \tilde{r}_3$ is a condi-

tion that yields $J(\tilde{r}_2) = J(\tilde{r}_3)$. Now, we need to find the condition for r_1 that produces $J(\tilde{r}_1)$ satisfying Eq. (363). Given $J(\tilde{r}_1) = J(\tilde{r}_2)$ and Eq. (355), we can write it as

$$J(\tilde{r}_1) = \frac{1}{2}\left\{E_1(\tilde{r}_1) - \frac{(1-a_1)}{(1+a_1)}E_2(\tilde{r}_1)\right\} = J(\tilde{r}_2)\ . \tag{365}$$

By substituting the terms $E_1(\tilde{r}_1)$ and $E_2(\tilde{r}_1)$ in Eq. (365), using Eq. (356) and Eq. (357), and rearranging all of the terms, we obtain

$$k_1 x^2 + k_2 x - 2J(\tilde{r}_2) = 0\ , \tag{366}$$

where

$$k_1 = \left[D_1 - \frac{1}{2}D_1\frac{(1-a_1)}{(1+a_1)}\right]\exp(2\alpha_1 r_{c_1}),\ k_2 = -\left[2D_1 + D_1\frac{(1-a_1)}{(1+a_1)}\right]\exp(\alpha_1 r_{c_1}), \tag{367}$$

and

$$x = \exp(-\alpha_1 \tilde{r}_1). \tag{368}$$

From Eq. (366), we can solve for x. By using Eq. (368), we obtain the value of $\tilde{r}_1$:

$$\tilde{r}_1 = -\frac{1}{\alpha_1}\ln\left(-\frac{k_2}{2k_1} \pm \frac{1}{2}\sqrt{\left(\frac{k_2}{k_1}\right)^2 + \frac{8J(\tilde{r}_2)}{k_1}}\ \right). \tag{369}$$

Therefore, the location of $\tilde{\mathbf{r}}$ is at when $r_2 = r_3$ and r_1 satisfying Eq. (369).

C. Continuity of $\partial\bar{V}/\partial r_i$

We will show that $\partial\bar{V}/\partial r_i\ ;\ \forall i$, is not continuous at $\tilde{\mathbf{r}}$. We will prove this by showing that the points $\tilde{\mathbf{r}}$ are a jump discontinuity. There are two parts of proof in this section.

First, we need to show that $\partial T_J(\mathbf{r})/\partial r_i = 0$ and $\frac{\partial^2 T_J}{\partial r_i \partial r_j} \neq 0\,;\ \forall i, j$ at $\tilde{\mathbf{r}}$. By taking the derivative of Eq. (359) with respect to r_i, we have

$$\frac{\partial T_J}{\partial r_i} = [2J(r_i) - J(r_j) - J(r_k)]\frac{\partial J(r_i)}{\partial r_i}\ ;\ \forall i,\ j, k \neq i \text{ and } j \neq k. \tag{370}$$

However, at $\tilde{\mathbf{r}}$, we have $J(\tilde{r}_1) = J(\tilde{r}_2) = J(\tilde{r}_3)$. Therefore the term $2J(\tilde{r}_i) - J(\tilde{r}_j) - J(\tilde{r}_k)$ in Eq. (370) is zero. Thus, we obtain $\partial T_J(\mathbf{r})/\partial r_i = 0\,;\ \forall i$, at $\tilde{\mathbf{r}}$. By taking the derivative of Eq. (370) with respect to r_j, we obtain, if $i \neq j$,

$$\frac{\partial^2 T_J}{\partial r_i \partial r_j} = [2J(r_i) - J(r_j) - J(r_k)]\frac{\partial}{\partial r_j}\left(\frac{\partial J(r_i)}{\partial r_i}\right) - \frac{\partial J(r_j)}{\partial r_j}\frac{\partial J(r_i)}{\partial r_i}, \tag{371}$$

and, if $i = j$,

$$\frac{\partial^2 T_J}{\partial r_i^2} = [2J(r_i) - J(r_j) - J(r_k)]\frac{\partial^2 J(r_i)}{\partial r_i^2} + 2\left(\frac{\partial J(r_i)}{\partial r_i}\right)^2. \tag{372}$$

At $\tilde{\mathbf{r}}$, since $2J(\tilde{r}_i) - J(\tilde{r}_j) - J(\tilde{r}_k) = 0$, then Eq. (371) and Eq. (372) reduce to

$$\frac{\partial^2 T_J}{\partial r_i \partial r_j} = \begin{cases} 2\left(\dfrac{\partial J(r_i)}{\partial r_i}\right)^2 & ;\text{ if } i = j\,. \\ -\dfrac{\partial J(r_j)}{\partial r_j}\dfrac{\partial J(r_i)}{\partial r_i} & ;\text{ if } i \neq j\,. \end{cases} \tag{373}$$

From Eq. (373), the terms $\frac{\partial^2 T_J}{\partial r_i \partial r_j}\,;\ \forall i, j$, are not necessarily zero at $\tilde{\mathbf{r}}$.

Second, from Eq. (361), the value $\partial \bar{V} / \partial r_i$ at $\tilde{\mathbf{r}}$ is in an indeterminate form because both $T_J(\mathbf{r})$ and $\partial T_J(\mathbf{r}) / \partial r_i$ are zero at $\tilde{\mathbf{r}}$. To find the values of $\partial \bar{V} / \partial r_i$ at $\tilde{\mathbf{r}}$ using L'Hopital's rule, we need to find $\lim_{\mathbf{r} \to \tilde{\mathbf{r}}} y$, where

$$y \equiv \frac{\partial T_J / \partial r_i}{T_J^{1/2}} . \tag{374}$$

Unfortunately, the denominator $T_J^{1/2}$ of y is not differentiable at $\tilde{\mathbf{r}}$, since its derivative

$$\frac{\partial T_J^{\,1/2}}{\partial r_i} = \frac{1}{2} T_J^{-1/2} \frac{\partial T_J}{\partial r_i} \tag{375}$$

does not exist (as its value is $0/0$). Thus, L'Hopital's rule cannot be directly used to find $\lim_{\mathbf{r} \to \tilde{\mathbf{r}}} y$. To overcome this problem, we find $\lim_{\mathbf{r} \to \tilde{\mathbf{r}}} y^2$. Now, both the numerator and denominator of y^2 are both differentiable, and their values at $\tilde{\mathbf{r}}$ are both zero. Therefore,

$$\lim_{\mathbf{r} \to \tilde{\mathbf{r}}} y^2 = \lim_{\mathbf{r} \to \tilde{\mathbf{r}}} \frac{(\partial T_J / \partial r_i)^2}{T_J} . \tag{376}$$

From L'Hopital's rule, by taking the derivative of the numerator and the denominator in Eq. (376) with respect to r_i, we obtain

$$\lim_{\mathbf{r} \to \tilde{\mathbf{r}}} y^2 = \lim_{\mathbf{r} \to \tilde{\mathbf{r}}} \frac{2(\partial T_J / \partial r_i)(\partial^2 T_J / \partial r_i^2)}{\partial T_J / \partial r_i} = \lim_{\mathbf{r} \to \tilde{\mathbf{r}}} 2(\partial^2 T_J / \partial r_i^2) . \tag{377}$$

By Eq. (373), Eq. (377) becomes

$$\lim_{\mathbf{r} \to \tilde{\mathbf{r}}} y^2 = 4\left(\frac{\partial J(r_i)}{\partial r_i} \right)^2 . \tag{378}$$

Therefore, we obtain

$$\lim_{\mathbf{r}\to\tilde{\mathbf{r}}} y \;=\; \pm 2\frac{\partial J(r_i)}{\partial r_i}\bigg|_{\mathbf{r}=\tilde{\mathbf{r}}} . \tag{379}$$

Thus, from Eq. (361) and Eq. (379), we have

$$\lim_{\mathbf{r}\to\tilde{\mathbf{r}}} \frac{\partial \bar{V}}{\partial r_i} \;=\; \left(\frac{\partial T_Q}{\partial r_i} \pm \frac{\partial J(r_i)}{\partial r_i}\right)\bigg|_{\mathbf{r}=\tilde{\mathbf{r}}} . \tag{380}$$

This implies that $\lim_{\mathbf{r}\to\tilde{\mathbf{r}}} \partial\bar{V}/\partial r_i\,;\ \forall i$, do not exist, as the limit converges to two values. Thus, $\partial\bar{V}/\partial r_i$ is not continuous at $\tilde{\mathbf{r}}$. From Eq. (380), we conclude that the points $\tilde{\mathbf{r}}$ are a jump discontinuity in $\partial\bar{V}/\partial r_i$. As $\mathbf{r}\to\tilde{\mathbf{r}}$, the values of $\partial\bar{V}/\partial r_i$ converge to $\frac{\partial T_Q}{\partial r_i} + \frac{\partial J(r_i)}{\partial r_i}$ at one end, and $\frac{\partial T_Q}{\partial r_i} - \frac{\partial J(r_i)}{\partial r_i}$ at the other end.

D. Differentiability of $\bar{V}$ and $\partial\bar{V}/\partial r_i$

It is easy to prove that the function $\bar{V}$ (which is continuous) is not differentiable. A continuous function is differentiable if its derivatives exist and are continuous. However, we proved that $\partial\bar{V}/\partial r_i$ does not exist and it is not continuous at $\tilde{\mathbf{r}}$. Therefore $\bar{V}$ is not differentiable at $\tilde{\mathbf{r}}$. Finally, we want to show that $\partial\bar{V}/\partial r_i$ is not differentiable. Since $\partial\bar{V}/\partial r_i$ is not continuous at $\tilde{\mathbf{r}}$, thus it is not differentiable at $\tilde{\mathbf{r}}$. (Another proof: At $\tilde{\mathbf{r}}$, we have

$\partial^2 T_J / \partial r_i^2 \neq 0$ from Eq. (373) but $T_J(\tilde{\mathbf{r}}) = 0$. Thus, $\lim_{\mathbf{r} \to \tilde{\mathbf{r}}} \partial^2 \bar{V} / \partial r_i^2$ in Eq. (362) converges to infinity. Consequently, $\partial \bar{V} / \partial r_i$ is not differentiable at $\tilde{\mathbf{r}}$.)

Summary of the proof: $\bar{V}(\mathbf{r})$ is a continuous function, but the first derivatives $\partial V / \partial r_i$; $\forall i$, are discontinuous at $\tilde{\mathbf{r}}$. Therefore, we conclude that the function $\bar{V}(\mathbf{r})$; $\forall \mathbf{r} > \mathbf{0}$, is of class C^0. If $\tilde{\mathbf{r}}$ are excluded from the input domain, the function $\bar{V}$ and $\partial \bar{V} / \partial r_i$ are both continuous and differentiable.

D. Scaled and true derivatives

When training neural networks, the input space and/or the output space of the training data are sometimes normalized to the range between -1 and 1. Consequently, this produces an impact on the scales of the derivatives. The conversion between the scaled and the true derivatives will be discussed. We will focus only on the conversion of the first and second derivatives, since they are the orders discussed in this research.

We use $g^n_{k,\,q}$ to denote the normalized k^{th} function value evaluated at the q^{th} input, associated to the true function value $g_{k,\,q}$. The notations $g_{k_{mn}}$ and $g_{k_{mx}}$ denote the minimum and the maximum values of g_k for all the training points, respectively. That is, for $\overline{Q} = \{1, 2, \ldots, Q\}$,

$$g_{k_{mn}} = min(g_{k,\,q}\,; \forall q \in \overline{Q}) \text{ and } g_{k_{mx}} = max(g_{k,\,q}\,; \forall q \in \overline{Q}). \tag{381}$$

The relationship between $g_{k,\,q}$ and $g^n_{k,\,q}$ is

$$g_{k,\,q} = \frac{1}{2}(g^n_{k,\,q} + 1)(g_{k_{mx}} - g_{k_{mn}}) + g_{k_{mn}}, \tag{382}$$

or, equivalently,

$$g^n_{k,\,q} = \frac{2(g_{k,\,q} - g_{k_{mn}})}{g_{k_{mx}} - g_{k_{mn}}} - 1. \tag{383}$$

For the conversion of the r^{th} input space, the relationship between $p_{r,\,q}$ and $p^n_{r,\,q}$ is realized by simply replacing every g_k with p_r in Eq. (382) and Eq. (383). (That is, change $g^n_{k,\,q}$

to $p^n_{r,\,q}$, change $g_{k,\,q}$ to $p_{r,\,q}$, change $g_{k_{mn}}$ to $p_{r_{mn}}$ and change $g_{k_{mx}}$ to $p_{r_{mx}}$.)

We will first discuss the conversion between the scaled and the true values for the first derivatives, followed by the second derivatives.

First derivatives

In this section, we are interested in the conversion between $\partial g^n_{k,\,q} / \partial p^n_{r,\,q}$ to $\partial g_{k,\,q} / \partial p_{r,\,q}$. By taking the derivative of Eq. (382) with respect to p_r, we obtain

$$\frac{\partial g_{k,\,q}}{\partial p_{r,\,q}} = \frac{1}{2}(g_{k_{mx}} - g_{k_{mn}})\frac{\partial g^n_{k,\,q}}{\partial p_{r,\,q}} \ . \tag{384}$$

By using the chain rule of calculus to the term $\partial g^n_{k,\,q} / \partial p_{r,\,q}$, we have

$$\frac{\partial g^n_{k,\,q}}{\partial p_{r,\,q}} = \frac{\partial g^n_{k,\,q}}{\partial p^n_{r,\,q}} \times \frac{\partial p^n_{r,\,q}}{\partial p_{r,\,q}}. \tag{385}$$

By the relationship in Eq. (383), we can compute $\partial p^n_{r,\,q} / \partial p_{r,\,q}$:

$$\frac{\partial p^n_{r,\,q}}{\partial p_{r,\,q}} = \frac{2}{p_{r_{mx}} - p_{r_{mn}}} \ . \tag{386}$$

Thus, Eq. (384) becomes

$$\frac{\partial g_{k,\,q}}{\partial p_{r,\,q}} = \left(\frac{g_{k_{mx}} - g_{k_{mn}}}{p_{r_{mx}} - p_{r_{mn}}}\right) \times \frac{\partial g^n_{k,\,q}}{\partial p^n_{r,\,q}} \ . \tag{387}$$

Next, we will discuss the conversion for the second derivatives.

Second derivatives

In this section, we want to know the conversion between $\frac{\partial}{\partial p_{r',q}}\left(\frac{\partial g_{k,q}}{\partial p_{r,q}}\right)$ and $\frac{\partial}{\partial p^n_{r',q}}\left(\frac{\partial g^n_{k,q}}{\partial p^n_{r,q}}\right)$. By taking the derivative of Eq. (387) with respect to $p_{r'}$, we obtain

$$\frac{\partial}{\partial p_{r',q}}\left(\frac{\partial g_{k,q}}{\partial p_{r,q}}\right) = \left(\frac{g_{k_{mx}} - g_{k_{mn}}}{p_{r_{mx}} - p_{r_{mn}}}\right) \times \frac{\partial}{\partial p_{r',q}}\left(\frac{\partial g^n_{k,q}}{\partial p^n_{r,q}}\right) \quad . \tag{388}$$

If the network is trained with the normalized inputs $\mathbf{p}^n$, the term $\partial g^n_{k,q} / \partial p^n_{r,q}$ is a function of $\mathbf{p}^n$. By using the chain rule of calculus to the last term in Eq. (388), we have

$$\frac{\partial}{\partial p_{r',q}}\left(\frac{\partial g^n_{k,q}}{\partial p^n_{r,q}}\right) = \frac{\partial}{\partial p^n_{r',q}}\left(\frac{\partial g^n_{k,q}}{\partial p^n_{r,q}}\right) \times \frac{\partial p^n_{r',q}}{\partial p_{r',q}} \quad . \tag{389}$$

By substituting Eq. (386) into Eq. (389), Eq. (388) becomes

$$\frac{\partial}{\partial p_{r',q}}\left(\frac{\partial g_{k,q}}{\partial p_{r,q}}\right) = \frac{2(g_{k_{mx}} - g_{k_{mn}})}{(p_{r_{mx}} - p_{r_{mn}})(p_{r'_{mx}} - p_{r'_{mn}})} \times \frac{\partial}{\partial p^n_{r',q}}\left(\frac{\partial g^n_{k,q}}{\partial p^n_{r,q}}\right) \quad . \tag{390}$$

REFERENCES

[AgRa06] Agrawal, P. M., Raff, L. M., Hagan, M. T., and Komanduri, R., "Molecular dynamics investigations of the dissociation of SiO_2 on ab initio potential energy surface obtained using neural network methods," *The Journal of Chemical Physics*, vol. 124, 134306, 2006.

[AgSa05] Agrawal, P. M., Samadh, A. N. A., Raff, L. M., Hagan, M., Bukkapatnam, S. T., and Komanduri, R., "Prediction of molecular-dynamics simulation results using feedforward neural networks: Reaction of a C2 dimer with an activated diamond (100) surface," *The Journal of Chemical Physics*, vol. 123, 224711, 2005.

[Arya90] Arya, A. P., *Introduction to Classical Mechanics*, Allyn and Bacon, Needham Haights, MA, 1990.

[AtPa97] Attali, J. G., and Pages, G., "Approximation of functions by a multilayer perceptron: a new approach," *Neural Networks*, vol. 10, pp. 1069-1081, 1997.

[BaEn99] Basson, E. and Engelbrecht, A. P., "Approximation of a function and its derviatives in feedforward neural networks," *International Joint Conference on Neural Networks*, Washington, Vol. 1, pp. 419-421, July 1999.

[Beem76] Beeman, D., "Some multistep methods for use in molecular dynamics calculations," *Journal of Computational Physics*, vol. 20, pp. 130-139, 1976.

[Bish95] Bishop, C. M., "Training with noise is equivalent to Tikhonov regularization," *Neural Computation*, Vol. 7, No. 1, pp. 108-116, 1995.

[ChHa99] Chen, D., and Hagan, M. T., "Optimal use of regularization and cross-validation in neural network modeling," *International Joint Conference on Neural Networks*, Washington, paper no. 323, July 1999.

[DeSc83] Dennis, J. E., and Schnabel, R. B., *Numerical Methods for Uncontrained Optimization and Nonlinear Equations*, Englewood Cliffs, Prentice-Hall, 1983.

[DyLo99] Dyck, D., Lowther, D. A., Malik, Z., Spence, R., and Nelder, J., "Response surface models of electromagnetic devices and their application to design," *IEEE Transaction on Magnetics*, vol. 35, no. 3, pp. 1821-1824, May, 1999.

[Enge01] Engelbrecht, A. P., "A new pruning heuristic based on variance analysis of sensitivity information," *IEEE Transaction on neural networks*, vol. 12, no. 6, pp. 1386-1399, November, 2001.

[FeSt05] Ferrari, S., and Stengel, R. F., "Smooth function approximation using neural networks," *IEEE Transactions on Neural Networks*, Vol. 16, No. 1, pp. 24-38, January 2005.

[Finc92] Fincham, D., "Leapfrog rotational algorithms," *Mol. Simul*, vol. 8, pp. 165-178, 1992.

[FiSi84] Finnis, M. W., and Sinclair, J. E., "A simple empirical N-body potential for transition metals," *Philos. Mag. A*, 50, 45-55, 1984.

[FoHa97] Foresee, D., and Hagan, M. T., "Gauss-Newton approximation to Bayesian learning," *Proceedings of the 1997 International Joint Conference on Neural Networks*, pp. 1930-1935, 1997.

[FrTr04] Frisch, M. J., Trucks, G. W., Schlegel, H. B., *et al.*, GAUSSIAN 03 (Revision C.02), Gaussian, Inc., Wallingford, CT, 2004.

[GaWh92] Gallant, A. R., and White, H., "On learning the derivatives of an unknown mapping with multilayer feedforward neural networks," *Neural Networks*, vol. 6, pp. 12-138, 1992.

[GiMo92] Gilbert, J. R., Moler, C., and Schreiber, R., "Sparse matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications*, Vol. 13, issue 1, pp. 333-356, 1992.

[GiMu81] Gill, P., Murray, W., and Wright, M. H., *Practical Optimization*, Academic Press, London, U.K., 1981.

[GiPo76] Gibbs, N. E., Poole, W. G., and Stockmeyer, P. K. Jr., "A comparison of several bandwidth and profile reduction algorithms," *ACM Transactions on Mathematical Software*, vol. 2, issue 4, pp. 322-330, 1976.

[HaDe96] Hagan, M. T., Demuth, H. B., and Beale, M., *Neural Network Design*, PWS Publishing Co., Boston, MA, 1996.

[HaMe94] Hagan, M. T., and Menhaj, M., "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989-993, November 1994.

[HaSt93] Hassibi, B., Stork, D. G., and Wolff, G. J., "Optimal brain surgeon and general network pruning," *IEEE International Conference on Neural networks*, San Francisco, CA, vol. 1, pp. 293-299, 1993.

[HaZa99] Hanselmann, T., Zaknich, A., and Attikiouzel, Y., "Learning functions and their derivatives using Taylor series and neural networks," *International Joint Conference on Neural Networks*, Washington, vol. 1, pp. 409-412, July 1999.

[HeKr91] Hertz, J., Krogh, A., and Plamer, R. G., *Introduction to the Theory of Neural Computation*, Addison Wesley, Redwood City, CA, 1991.

[HoJo94] Horn, R., and Johnson, C. R., *Topics in Matrix Analysis*, Cambridge, MA, 1994.

[HoKo64] Hohenberg, P., and Kohn, W., "Inhomogenous electron gas," *Phys. Rev.*, 136, B864, 1964.

[Horn91] Hornik, K., "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, pp. 1069-1072, 1991.

[HoSt89] Hornik, K., Stinchcombe, M. and White, H., "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359-366, 1989.

[HoSt90] Hornik, K., Stinchcombe, M. and White, H., "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks," *Neural Networks*, vol. 3, pp. 551-560, 1990.

[HuSe05] Huynh, T. Q., and Setiono, R., "Effective neural network pruning using cross-validation," *Proceedings of International Joint Conference on Neural Networks*, Montreal, Canada, 2005.

[Ito93] Ito, Y., "Approximation of differentiable functions and their derivatives on compact sets by neural networks," *Math. Scient.*, 18, pp. 11-19, 1993.

[Karn90] Karnin, E. D., "A simple procedure for pruning back-propagation trained neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 2, pp. 239-242, 1990.

[KuNe66] Kuntz, P. J., Nemeth, E. M., Polanyi, J. C., Rosner, S. D., and Young, C. E., *The Journal of Chemical Physics*, vol. 44, 1168, 1966.

[LaFo06] Lauret, P., Fock, E., and Mara T. A., "A node pruning algorithm based on a Fourier amplitude sensitivity test method," *IEEE Transactions on Neural Networks*, vol. 17, no. 2, pp. 273-293, 2006.

[Lee07] Lee, Kun-Chou, “Application of neural network and its extension of derivative to scattering from a nonlinearly loaded antenna,” *IEEE Transaction on Antenna and Propagation*, vol. 55, no. 3, pp. 990-993, 2007.

[Leve44] Levenberg, K., “A method for the solution of certain non-linear problems in least squares”, *The Quarterly of Applied Mathematics*, vol. 2, 1944, pp. 164-168

[Li96] Li, X., “Simultaneuous approximations of multivariate functions and their derivatives by neural networks with one hidden layer,” *Neurocomputing*, 12, pp. 327-343, 1996.

[Lo99] Lo, J. T., “Statistical method of pruning neural networks,” *Proceedings of international joint conference on neural networks*, pp. 1678-1680, July, 1999.

[Marq63] Marquardt, D. W., “An algorithm for least-square estimation of nonlinear parameters”, *SIAM Journal on Numerical Analysis*, vol. 11, no. 2, 1963.

[McCa06] McCaffrey, J., *.NET Test Automation Recipes: A Problem-solution Approach*, Apress, Inc., Berkeley, CA, 2006.

[MacK92] MacKay, D. J. C., “Bayesian Interpolation,” *Neural Computation*, vol. 4, pp. 415-447, 1992.

[MaDy99] Malik, Z., Dyck, D., Nelder, J., Spence, J., and Lowther, D., “Modelling with gradient information,” Proc Instn Mech Engrs - Part B, vol. 213, pp. 209-213, 1999.

[MaNe99] Magnus, J. R., and Neudecker, H., *Matrix differential calculus with applications in statistics and econometrics*, Revised edition, Wiley, Chickester, U.K., 1999.

[Matt93] Mattis, D. C., *The Many-Body Problem: An Encyclopedia of Exactly Solved Models in One Dimension*, World Scientific, 1993.

[MoPl34] M∅ller, C., and Plesset, M. S., “Note on an approximation treatment for many-electron systems,” *Phys. Rev.*, 46, 618, 1934.

[Mors29] Morse, P. M., “Diatomic molecules according to the wave mechanics II vibrational levels,” *Phys. Rev*., 34, 57-64, 1929.

[NgTr99] Nguyen-Thien, T., and Tran-Cong, T., “Approximation of functions and their derivatives: A neural network implementation with applications,” *Appl. Math. Modelling*, vol. 23, pp. 687-704, 1999.

[NgWi90] Nguyen, D., and Widrow, B., “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights,” *International Joint Conference on Neural Networks*, San Diego, CA, vol. 3, pp. 21-26, 1990.

[OgHe07] Ogunniyi, A. J., Henriquez, S. L., Karangu, C. W., Dickens, C., and White, C., “Accurate modelling of drain current derivatives of MES-FET/HEMT devices for intermodulation analysis,” *IEEE International Symposium on Circuits and Systems*, pp. 1013-1016, May, 2007.

[Pink99] Pinkus, A., “Approximation theory of the MLP model in neural networks,” *Acta Numerica*, 8, pp. 143-195, 1999.

[PuMa09] Pukrittayakamee, A., Malshe, M., Hagan, M., Raff, L., Bukkapatnam, S., and Komanduri, R., “Simultaneous fitting of a potential-energy surface and its corresponding force fields using feedforward neural networks,” *The Journal of Chemical Physics*, 130, 134101, 2009.

[Raff01] Raff, L. M., *Principles of Physical Chemistry*, Prentice Hall, New Jersy, 2001.

[RaMa05] Raff, L.M., Malshe, M., Hagan, M., Doughan, D.I., Rockley, M.G., and Komanduri, R., “Ab initio potential-energy surfaces for complex, multi-channel systems using modified novelty sampling and feedforward neural networks,” *The Journal of Chemical Physics*, vol. 122, 084104, 2005.

[Reed93] Reed, R., “Pruning algorithm - A survey,” *IEEE Transaction on Neural Networks*, vol. 4, no. 5, pp. 740-747, September 1993.

[SeGa00] Setiono, R., and Gaweda, A., “Neural network pruning for function approximation,” *Proceedings of the International Joint Conference on Neural Networks*, vol. 6, 2000.

[SiDo88] Sietsma, J., and Dow, R J F., “Neural net pruning - why and how,” *IEEE International Conference on Neural Networks*, San Diego, CA, vol. 1, pp. 325-333, 1988.

[SiDo91] Sietsma, J., and Dow, R J F., “Creating aritificial neural networks that gneralize,” *Neural Networks*, vol. 4, pp. 67-69, 1991.

[Stei85] Steinfeld, J. I., *Molecules and Radiation: An Introduction to Modern Molecular Spectroscopy*, second edition, MIT Press, Cambridge, MA, 1985.

[SuRa84] Sudhakaran, M. P., and Raff, L. M., “Quasiclassical trajectory studies of HD + HBr(DBr) abstraction and exachange reactions,” *The Journal of Chemical Physics*, vol, 95, pp. 165-177, 1985.

[SwAn82] Swope, W. C., Andersen, H. C., Berens, P. H., and Wilson, K. R., "A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters," *The Journal of Chemical Physics*, vol. 76, pp. 637-649, 1982.

[Ters88] Tersoff, J., "*Empirical interatomic potential for Carbon, with applications to amorphous Carbon*," Phys. Rev. Lett 61, pp. 2879, 1988.

[Ters89] Tersoff, B., "*Modelling soild-state chemistry: Interatomic potentials for multicomponent systems*", Phys. Rev. B, vol. 39, pp. 5566 - 5568, 1989.

[TiAr77] Tikhonov, A. N., and Arsenin, V. Y., *Solutions of Ill-posed Problems*, V H Winston and sons, Washington D.C., 1977.

[Verl67] Verlet, L., "Computer experiments on classical fluids," *Phys. Rev.*, vol. 165, pp. 201-214, 1967.

[WaHi00] Wan, W., Hirasawa K., Hu, J., and Jin, C., "A new method to prune the neural network," *Proceedings of the International Joint Conference on Neural Networks*, vol. 6, 2000.

[Werb88] Werbos. P., "Backpropagation: Past and future," *Proceedings of the IEEE International Conference on Neural Networks*, pp. 343-353. IEEE Press, 1988.

[XuYa03] Xu, J., Yagoub, M. C. E., Ding, R., and Zhang, Q. J., "Exact adjoint sensitivity analysis for neural-based microwave modeling and desing," *IEEE Transaction on Microwave Theory and Techniques*, vol. 51, no. 1, January, 2003.

VITA

Arjpolson Pukrittayakamee

Candidate for the Degree of

Doctor of Philosophy

Thesis: FITTING FUNCTIONS AND THEIR DERIVATIVES WITH NEURAL NETWORKS

Major Field: Electrical and Computer Engineering

Biographical:

Personal Data: Born in Bangkok, Thailand, on June 10, 1977, the son of Ongarj Pukrittayakamee and Boontida Pukrittayakamee.

Education: Graduated from Triam Udom Suksa School, Bangkok, Thailand, in April 1993; received Bachelor of Engineering degree in Electrical Engineering from Chulalongkorn University, Bangkok, Thailand, in May 1997; received Master of Sciences degree in Electrical and Computer Engineering from Oklahoma State University, Stillwater, Oklahoma, in December 2001. Completed the requirements for the Doctor of Philosophy degree in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma, in July 2009.

Experience: Employed by National Electronics and Computer Technology Center as an Assistant Researcher from 1997 to 1999; employed by Oklahoma State University as a Research Assistant from 2000 to 2002; employed by Thaicom Satellite Plc. as an Engineering Specialist from 2002 to 2005; employed by Oklahoma State University as a Research Associate from 2005 to present.

Professional Status and Membership: Member of Institute of Electrical and Electronic Engineers, and International Neural Network Society.

Name: Arjpolson Pukrittayakamee Date of Degree: July, 2009

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: FITTING FUNCTIONS AND THEIR DERIVATIVES WITH NEURAL NETWORKS

Pages in Study: 250 Candidate for the Degree of Doctor of Philosophy

Major Field: Electrical and Computer Engineering

Scope and Method of Study: The objective of this work was to study methods of simultaneously approximating functions and their first-order derivatives using multilayer feedforward neural networks. There are a few methods proposed today to simultaneously approximate both functions and their first derivatives, but they require modifications to the network structure. The new method works with any multilayer feedfoward neural network, by forming a new performance index that combines both the function error and the first derivative error. We tested and analyzed the results of the proposed method on both analytic and real-world problems.

Findings and Conclusions: We selected two optimization procedures for the new performance index. The first procedure was for any gradient-based optimization, while the other was implemented under the Levenberg-Marquardt framework. For each procedure, extra backpropagation calculations were derived to force the first derivative response of the neural network to match the desired derivative target. Moreover, we discovered two new types of overfitting from neural networks trained with the proposed performance index. We analyzed and illustrated how the overfitting develops. A network pruning algorithm was proposed to eliminate these types of overfitting. The simulation results tested on four analytic problems and three systems in Molecular Dynamics consistently showed that the approximation accuracy of neural networks trained by the new performance index significantly outperformed the use of standard training methods. In addition, the network generalization was even further improved with the incorporation of the pruning algorithm. We found that the most promising method yielding the most accurate approximation and the best generalization was to optimize the new performance index under the Levenberg-Marquardt framework along with the use of the pruning algorithm.

ADVISER'S APPROVAL: Dr. Martin T. Hagan