A FAMILY OF ROBUST SECOND ORDER TRAINING ALGORITHMS

by

SANJEEV SREENIVASA RAO MALALUR

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2009

To my Parents, for the right *initialization* and to all my family and friends.

Without them I would not be able to *learn* and *generalize* in life.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Michael T. Manry. He has been a constant source behind my work, motivating me, challenging me and inspiring me. I am indebted to him for the countless hours he put into my work. He was always available for discussions, be it week day or weekends, always willing to address any problems and always providing the right guidance. Discussions with Dr. Manry was always a pleasure and the invaluable knowledge I gained is something I will treasure for life.

I would like to thank Dr. Jean Gao, Dr. Frank Lewis, Dr. Qilian Liang and Dr. Saibun Tjuatja for taking the time to serve on my comprehensive and dissertation committee.

I would like to thank my parents, my uncle and aunt, my brother, my fiancée, my future in-laws and all my friends for their support.

July 16, 2009

ABSTRACT

A FAMILY OF ROBUST SECOND ORDER TRAINING ALGORITHMS

SANJEEV SREENIVASA RAO MALALUR, Ph.D.

The University of Texas at Arlington, 2009

Supervising Professor: Michael T. Manry

Starting with the concept of *equivalent networks*, a framework for analyzing the effect of linear dependence on training of a multi-layer perceptron is established. Detailed mathematical analyses are carried out to show that training using backpropagation and Newton's method is different under the presence of linear dependence.

Two effective batch training algorithms are developed for the multilayer perceptron. First, the optimal input gain algorithm is presented, which computes an optimal gain coefficient for each input, used to update the input weights. The motivation for this algorithm comes from using equivalent networks to analyze the effect of input transformation. It is shown that the use of a non-orthogonal, non-singular diagonal transformation matrix is equivalent to altering the input gains in the network. Newton's method is used to simultaneously solve for the input gains and an optimal learning factor. In several examples, it is shown that the final algorithm is a reasonable compromise between first order training methods and Levenburg-Marquardt.

Second, a multiple optimal learning factor algorithm, that assigns a separate learning factor for each hidden unit is developed. The idea stems from relating a

single optimal learning factor to Newton's method. It is then extended to estimate separate optimal learning factors for each hidden unit. In several examples, this method performs as well as or better than Levenberg-Marquardt.

Both methods yield a smaller Hessian compared to Newton's method for updating input weights. The Hessian matrix thus computed is less susceptible to linear dependence and displays fast convergence. It is shown that the elements of the Hessian matrix for both methods are formed by some weighted combinations of the elements from the total network's Hessian.

When used with backpropagation-type learning, the two proposed methods are limited by the presence of dependent inputs. However, when used with hidden weight optimization technique, it is shown that both methods over come the presence of dependent inputs and completely ignore them during training. This improvement results in two highly robust second order learning algorithms, which are less heuristic, less susceptible to ill-conditioned Hessian, immune to linear dependencies, faster than LM and superior to standard first order training methods.

In the last part, a new approach for modeling simple discontinuous functions is developed. This two-stage approach, trains separate networks, one for a continuous function and another for discrete step function, in the first stage and *fuses* the two trained networks in the second stage to obtain the final network capable of modeling the discontinuous function. Results of using our proposed second order methods to train and fuse networks to model simple discontinuous sine and ramp functions are presented.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Feed-forward Neural Networks: Evolution and Applications

From its inception, early work on artificial neural networks was inspired by the functioning of the human cognitive system. McCulloch and Pitts [1] laid the foundation for neural networks with their pioneering work in the early 1940s. Several inspired works followed, but the next breakthrough came 15 years later from Rosenblatt [2] and his work on the *perceptron* which included the so-called *perceptron convergence theorem*. The earliest form of trainable layered neural network architecture with multiple adaptive elements can be traced to Widrow's work on the *Madaline* [3]. Among all the research work that followed, the most influential publication related to feed-forward networks is that on *backpropagation* (BP) by Rumelhart, Hilton and Williams [4]. BP has emerged as the most popular learning algorithm for training the multi-layer perceptron. Other training algorithms found in the literature such as Newton's method, *Levenberg-Marquardt* algorithm [5][6], etc., have their roots in *classical optimization theory*. Haykin [7] provides a more detailed account on the historic development of neural networks.

Neural networks have not succeeded as a model of the human brain. Without doubt, the learning, generalization, memorization and prediction capabilities of the human cognitive system is the most advanced system known to man. Moreover, significant advances in neural networks have followed more of an experimental path in spite of some supporting theoretical proofs. Hence, most methods have more of a heuristic origin than a theoretical one. This does not mean neural networks can be

discarded as some ad-hoc method that just happens to work. Neural networks have emerged as powerful statistical tools capable of learning and generalization. They have been used in the in the areas of parameter estimation [8] [9], document analysis and recognition [10], finance and manufacturing [11] and data mining [12]. Specific applications of neural networks include target recognition [13, 14], power load forecasting [15, 16], ZIP code recognition [17, 18], prognostics [19], face recognition [20], image retrieval [21] and speaker recognition [22] among several others.

1.2 Neural Network Architecture

The primary component that governs the functional aspect of a neural network is the *neuron*. The neurons form the basic computing elements. Different neural networks architectures are a result of varying either the *type* of neuron or the neuron *arrangement* or the neuron *adaptation*. Figure 1.1 shows a simple model of the neuron.

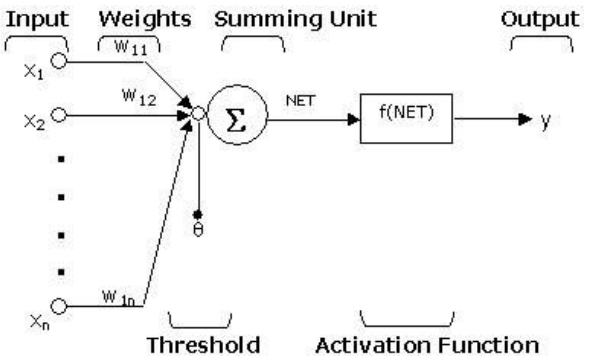


Figure 1.1. Model of an artificial neuron.

It has a set of weights connecting the inputs to the summing node. The summing node feeds into an activation function for limiting the output of the neuron.

The activation function is usually non-linear and the most popular one is the sigmoid [23]. There are other activations also, such as piecewise-linear functions [24] and trigonometric functions [25]. Neural networks derive the non-linearity based on the activation type.

The *structure* defines the connections between the neurons and how the information is processed. The network architecture decides how the inputs are mapped to the outputs. Without getting into too much detail, some of the popular neural network architectures include the multi-layer perceptron, radial basis function network [26], piecewise-linear network [27] and self-organizing maps [28].

The learning algorithm determines how the network's connections are adapted, to best carry out the desired task. Learning methods can be broadly classified as *supervised* or *unsupervised* learning. Supervised learning usually involves minimizing a *cost* function and adjusting the various *network parameters* accordingly. Some well known supervised learning techniques include backpropagation (BP), output weight optimization-backpropagation (OWO-BP) [29], output weight optimization - hidden weight optimization (OWO-HWO) [30] and Levenberg-Marquardt (LM) algorithm.

The tasks of a neural network can be broadly categorized as *regression*-type tasks or *classification*-type tasks. Regression and classification can be seen as particular cases of function approximation. Both look for some form of a mapping between a set of inputs to a set of outputs, which form the training data set. In a classification problem, the task is to assign the inputs to one of a number of discrete classes or categories. In this case, the outputs can be considered as being discrete. However, there are problems in which the outputs are continuous variables. Regression problems seek a function which best approximates the continuous mapping between the inputs and outputs.

1.3 The Multilayer Perceptron

The multilayer perceptron (MLP) forms a very important class of neural networks. A typical MLP structure consists of a set of input units that constitute the *input layer*, one or more layer of neurons forming the *hidden layers* and an *output layer* for producing the actual output. All the layers are connected by weights and the signal travels from the input through the hidden layers, to the output layer. Such a network is shown below.

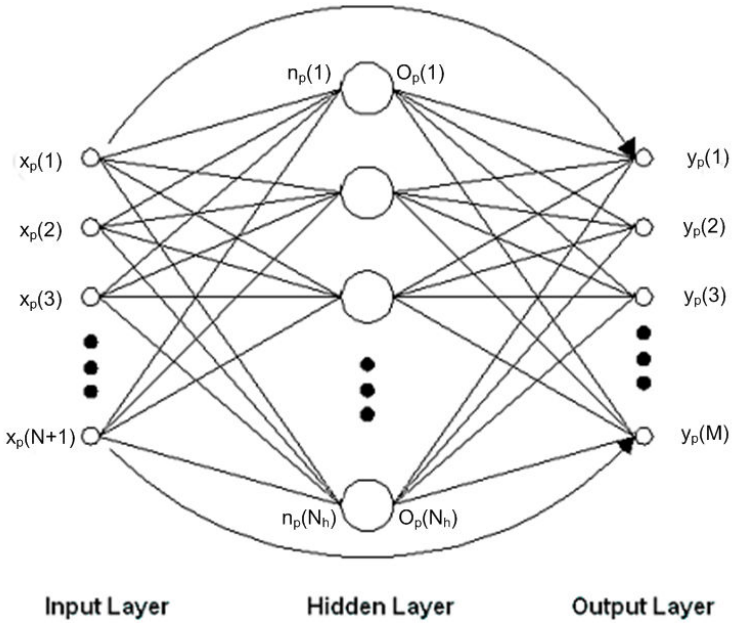


Figure 1.2. Structure of a Multilayer Perceptron.

The MLP shown in figure 1.2 has N_h nonlinear hidden units. Each unit in the hidden layer is equivalent to the single neuron in figure 1.1. The threshold θ from figure 1.1 appears as an additional constant input, $x_p(N+1)$, such that $x_p(N+1) = 1$. The output of the summing node, *NET*, appears here as $n_p(k)$ and the output of the block *f(NET)*, is represented as $O_p(k)$, where $1 \leq k \leq N_h$. It has been proved that a multilayer perceptron with sufficiently many nonlinear units (neurons)

in a single hidden layer can work as universal function *approximators* [31]. Other favorable features include the MLP's ability to mimic Bayes discriminant [32] and MAP estimation [33]. The MLP computes its outputs as a weighted sum of the inputs and the hidden unit outputs. The weights form the *unknowns*, which are typically found by minimizing the mean squared error between the actual and desired outputs. This is well known problem in *classical optimization theory*. We are interested in a specific branch of it, called *smooth, unconstrained nonlinear optimization*.

The MLP provides a physical model to encode the empirical knowledge represented by the input data into a corresponding set of synaptic weights, $\mathbf{w}$. Training the MLP involves adapting its weights $\mathbf{w}$ by minimizing an error criteria. Some classical training algorithms are backpropagation [4], conjugate-gradient method [34], the classic Newton's method and Levenberg-Marquardt [5][6].

1.4 Research Focus

This dissertation aims to make theoretical and practical contributions in the following areas related to training MLP networks.

1.4.1 Effect of Linear Dependence on Learning

Considerable research has been done on MLP training and several powerful algorithms have been proposed and analyzed. An obvious, yet surprisingly less explored aspect is the effect of linear dependence on MLP training algorithms. Given the structure of the MLP, dependence can occur at any layer, between any units and can often have a negative impact on learning. Guo [35] presented one of the early works on mathematical analysis of the dynamics of MLP. Hirose [36] and Murray [37] identified two types of undesired minima that can occur during training an MLP. In [38], Annema et al., introduce a vector decomposition method to simplify

the mathematical analysis of occurrence of temporary minima during learning using backpropagation. Yi-Jen Wang [39] provides an analysis of Newton's method used for training an MLP, where it is proved that the Hessian matrix of the MLP is always singular.

Linear dependence in data or network parameters is an important issue as it can significantly affect learning by slowing down convergence or getting stuck at a local minima. It would also result in a redundant network structure. Current literature on modeling and analyzing the effect of linear dependence on learning is rather sparse.

1.4.2 Input Transformation

Given the statistical nature involved in the learning of MLP, many data preprocessing techniques have been used in the past to improve the performance of MLP. Feature de-correlation [40], whitening transformation [41], un-biasing or normalization are some of the frequently used strategies. Yu [42] analyzed the effect of data preprocessing on several training methods and showed that while some transforms are useful, orthogonal transforms in general do not help in reducing the overall training error, i.e. orthogonal transforms have no effect on learning. Pre-processing techniques are nothing but linear transformations of the data performed in an effort to reduce the training error. They are data dependent, i.e. there is no one transformation that is guaranteed to reduce the error for all types of data. Given the data, researchers usually try several preprocessing schemes and use the one that works best with their algorithm.

1.4.3 Learning with Multiple Learning Factors

A common heuristic employed in training MLP is to use a single learning factor to adjust all the weights in the network. For instance, backpropagation finds the

gradient and determines the *step size* to take along the direction of the negative gradient to update the weights. While this sometimes works well in practice, there is no reason to not use multiple learning factors during training in order to speed up convergence. The delta-bar-delta method [43] was one of the popular early methods to adapt each weight independently. Moody [44] proposed a network with locally tuned processing units. Silva [45] proposes an acceleration technique for backpropagation based on the individual adaptation of learning rates for each weight. Unfortunately, these methods are heuristic and their performance relies on the settings of some user chosen parameters. Also, using standard gradients makes them slow to converge. Newton's method can be viewed as a second order method that assigns a learning rate to every weight in the network. For quadratic error functions, Newton's method converges in one step if the Hessian matrix is nonsingular. However, this is usually not true in practice [39], so Levenberg-Marquardt [46] (LM) and other methods are used instead.

1.4.4 Modeling Discontinuous Functions

The *universal approximation theorem* [31] says that any continuous function can be approximated over a compact input space, but there are several real-world instances where the signal is discontinuous, either due to the nature of the problem or mere truncation during data acquisition. Such signals can be discontinuous in certain parts of the *data space* and continuous elsewhere. Traditional training methods such as backpropagation do not perform so well on discontinuous data. In [47], Selmic and Lewis propose a network architecture that uses a mixture of both sigmoid and *jump* activations to model the overall function. Another method proposed in [48], incrementally add units as discontinuity is encountered.

1.4.5 Convergence Proof

A variety of algorithms exist for training a multilayer perceptron with improvements in either the overall training mean square error (MSE) or training time or compactness (less memory for implementation). As mentioned in section 1.1, developments of neural network training algorithms have followed more of a heuristic path. While these algorithms work very well in practice, it is also important to have theoretical proofs that support their behavior. Bounds on performance, proof of convergence, etc. are a few important indicators of an algorithms overall capability. However, there are several algorithms that lack the theoretical foundation.

In general, MLP training algorithms are negatively impacted by sensitivity to the choice of initial weights, linear dependencies among inputs and hidden units, non-zero means in the inputs, computational complexity and the difficulties in approximating discontinuous functions.

1.5 Research Objectives and Dissertation Organization

The main objectives of this dissertation are to provide a detailed mathematical analysis of linear dependency and its effects on training, develop faster second order training methods that can overcome dependencies, develop a training algorithm that can model discontinuous functions and develop a convergence proof of the hidden weight optimization algorithm.

Chapter 2 introduces a notation for MLP training that is used throughout this dissertation and presents a brief review of some important training algorithms, relevant to this dissertation. Chapter 3 begins by establishing the concept of equivalent networks and extends it to analyze the effects of linear dependency on network training (including first and second order methods). Problems addressed in this disserta-

tion along with goals and tasks are covered in chapter 4. Chapters 5 and 6 introduce two new learning algorithms called the *optimal input gain* algorithm and the *multiple optimal learning factor* algorithm, along with analyses, performance evaluations and comments on their limitations. Chapter 7 presents improvements to overcome the limitations of the two algorithms, making them highly robust.

The data files used for performance evaluation are common to all algorithms and are listed in appendix A. A proof of convergence for the hidden weight optimization algorithm is given in appendix B.

CHAPTER 2

PRELIMINARIES

This chapter introduces the notation for MLP that is used throughout this dissertation. It then briefly describes popular first and second order methods for training MLP, all of which are relevant to this dissertation.

2.1 MLP Notation

A fully connected MLP is shown in figure 1.2. It consists of an input layer, a single hidden layer consisting of nonlinear processing elements, also called as neurons, and an output layer. All the layers are connected by weights.

Input weight $w(k,n)$ connects the n^{th} input to the k^{th} hidden unit. Output weight $w_{oh}(m,k)$ connects the k^{th} hidden unit's activation $O_p(k)$ to the m^{th} actual output $y_p(m)$, which has a linear activation. The bypass weight $w_{oi}(m,n)$ connects the n^{th} input to the m^{th} output.

The training data, described by the set $\{\mathbf{x}_p, \mathbf{t}_p\}$ consists of N-dimensional input vectors $\mathbf{x}_p$ and M-dimensional desired output vectors $\mathbf{t}_p$. The pattern number p varies from 1 to N_v, where N_v denotes the number of training vectors present in the data set.

In order to handle thresholds in the hidden and output layers, the input vectors are augmented by an extra element $x_p(N+1)$ where $x_p(N+1) = 1$, so $\mathbf{x}_p = [x_p(1), x_p(2), \cdots x_p(N+1)]^T$. For the p^{th} pattern, the k^{th} hidden unit's net function, $n_p(k)$ is given by

$$n_p(k) = \sum_{n=1}^{N+1} w(k,n) x_p(n) \tag{2.1}$$

which can be summarized in matrix notation as

$$\mathbf{n}_p = \mathbf{W} \cdot \mathbf{x}_p \tag{2.2}$$

Here $\mathbf{n}_p$ denotes the N_h-dimensional column vector of net function values and $\mathbf{W}$ is N_h by $(N+1)$. For the p^{th} pattern, the k^{th} hidden unit's activation output is denoted as $O_p(k)$ where $O_p(k) = f(n_p(k))$ and $f(\cdot)$ denotes the hidden layer activation.

The actual output of the network $y_p(m)$ is computed as

$$y_p(m) = \sum_{n=1}^{N+1} w_{oi}(m,n)x_p(n) + \sum_{k=1}^{N_h} w_{oh}(m,k)O_p(k) \tag{2.3}$$

which can be written in matrix notation as

$$\mathbf{y}_p = \mathbf{W}_{oi} \cdot \mathbf{x}_p + \mathbf{W}_{oh} \cdot \mathbf{O}_p \tag{2.4}$$

where $\mathbf{O}_p$ is the N_h-dimensional hidden unit activation vector. The last rows of $\mathbf{W}$ and $\mathbf{W}_{oi}$ respectively store the hidden unit and output unit threshold values. The weights form the *unknowns* which are found by minimizing a *cost* or *error* function. A typical error function used in training the MLP is the mean-squared error (MSE) described as

$$E = \frac{1}{N_v} \sum_{p=1}^{N_v} \sum_{m=1}^{M} [t_p(m) - y_p(m)]^2 \tag{2.5}$$

Training an MLP involves minimizing the MSE and solving for the unknown weights over several iterations. In the following sections, some well known MLP training methods are reviewed.

2.2 Output Weight Optimization

Output weight optimization (OWO) is a technique to solve for weights connected to the actual outputs of the network (this would be the output weights, $\mathbf{W}_{oh}$ and by-pass weights, $\mathbf{W}_{oi}$). Since the outputs have linear activation, finding the

weights connected to the outputs is equivalent to solving a system of linear equations. Popular candidates for OWO are conjugate gradient [34] and the orthogonal least squares methods [49].

2.3 Backpropagation Algorithm

The popular backpropagation (BP) algorithm is a *first order* method that uses gradient information to update the weights in the network. In full batch mode, the (BP) algorithm updates the input weights and thresholds, $\mathbf{W}$, as

$$w(k,n) = w(k,n) + Z\left(\frac{-\partial E}{\partial w(k,n)}\right) \tag{2.6}$$

for all $1 \leq k \leq N_h$ and all $1 \leq n \leq (N+1)$. For the p^{th} pattern, we use BP to get the partial derivative of E_p (error for the p^{th} pattern) as

$$-\frac{\partial E_p}{\partial w(k,n)} = -\frac{\partial E_p}{\partial n_p(k)} \cdot \frac{\partial n_p(k)}{\partial w(k,n)} \tag{2.7}$$

For the p^{th} pattern, hidden layer delta function is found as

$$\delta_p(k) \;\; = \;\; O'_p(k)) \sum_{m=1}^{M} \delta_{po}(m) w_{oh}(m,k) \tag{2.8}$$

where, $\delta_{po}(m) = 2(t_p(m) - y_p(m))$

Now, the negative gradient of E is

$$g(k,n) = -\frac{\partial E}{\partial w_{kn}} = \frac{1}{N_v} \sum_{p=1}^{N_v} \delta_p(k) x_p(n) \tag{2.9}$$

The matrix of negative partial derivatives can be written as

$$\mathbf{G} = \frac{1}{N_v} \sum_{p=1}^{N_v} \boldsymbol{\delta}_p \mathbf{x}_p^T \tag{2.10}$$

where $\boldsymbol{\delta}_p = [\delta_p(1), \delta_p(2), \cdots, \delta_p(N_h)]^T$. If steepest descent is used to modify the hidden weights, $\mathbf{W}$ is updated in a given iteration as

$$\mathbf{W} \leftarrow \mathbf{W} + z \cdot \mathbf{G} \tag{2.11}$$

so

$$\triangle \mathbf{W} = z \cdot \mathbf{G} \tag{2.12}$$

where z is the learning factor.

First order methods are generally easier to implement and also require the least computation per training iteration. They are also guaranteed to converge to a global or local minimum. However, these benefits are largely outweighed by the fact that the method's rate of convergence is often very slow [50].

2.4 Output Weight Optimization-Backpropagation

One option to train an MLP would be to divide the weight adaptation into two separate stages: (i) train all weights, $\mathbf{W}_{oh}, \mathbf{W}_{oi}$ connected to the actual network outputs and (ii) train all the input weights, $\mathbf{W}$. During either stage, the other weights are not updated. This approach combines the two previously described approaches and is called output weight optimization-backpropagation (OWO-BP). This method is attractive for several reasons. First, the training is faster, since training weights connected to the outputs is equivalent to solving for linear equations. Second, it helps us avoid some local minima. Third, the method exhibits improved training performance.

2.4.1 Convergence of OWO-BP

To show convergence of OWO-BP, we have to show that both OWO and BP stages are convergent. The backpropagation weight update is given by equation 2.12. It is possible to find an *optimal* z, that will minimize the mean square error, E, in a given iteration by solving for $\partial(E + z \cdot \mathbf{G})/\partial z = 0$. Since $\mathbf{G}$ is the negative Jacobian matrix, and we are finding a minimum, z will have to be non-negative.

Every time the input weights are updated using BP, the output weights must be re-calculated and updated. As mentioned before, OWO finds output weights by solving a system of linear equations and in any training iteration the error after OWO is guaranteed to be less than or at least equal to the error in the previous iteration.

Let E_k denote the error at the k^{th} step of OWO-BP training. For k odd, E_k denotes the error after an OWO stage and for k even, E_k denotes the error after a BP step. Since z is positive and optimal, the BP step can only decrease E or leave it unchanged. Similarly, OWO steps can only decrease E or leave it the same. If OWO-BP is run for N_{it} iterations, then error E_k for every step forms a *monotone series*, i.e. a series of non-increasing, non-negative numbers such that $E_{k+1} \leq E_k$. Such a series is guaranteed to converge [51] as $N_{it} \rightarrow \infty$.

2.5 Newton's Method

One approach to produce a significant improvements in the convergence performance of an MLP training is to use *higher order information* [7]. Generally, second order methods exhibit superior performance to first order methods.

Newton's method is a classic, iterative unconstrained optimization technique that aims to minimize E with respect to the weights in the network. In Newton's method, we calculate the Hessian matrix $\mathbf{H}$, whose elements are second partial derivatives of E with respect to the network weights. Hence $\mathbf{H}$ is N_w by N_w, where N_w is the number of weights in the network. The weight update vector $\triangle\mathbf{W}$ for all N_w weights is found as

$$\triangle\mathbf{W} = -\mathbf{H}^{-1}\mathbf{g} \tag{2.13}$$

where $\mathbf{g}$ is the gradient of E.

If the error function is quadratic, then Newton's method converges to the optimum solution in one iteration. However, practical application of this method to train an MLP is limited for several reasons. Excess storage and multiplies per iteration are two among the many difficulties which prevent the implementation of Newton's algorithm for training the MLP. One approach for remedying these problems is to apply Newton's algorithm to the input weights only, in the first half of a given iteration. The second part of the iteration would be to solve linear equations for the output weights, which include the bypass weights, as in OWO.

Equation (2.13) is then rewritten as

$$\triangle \mathbf{W}_R = -\mathbf{H}_R^{-1}\mathbf{g}_R \tag{2.14}$$

where the reduced size Hessian $\mathbf{H}_R$ has N_{iw} rows where $N_{iw} = (N+1)N_h$, is the number of input weights. When Gauss-Newton [50] updates are used, elements of $\mathbf{H}_R$ are computed as

$$\frac{\partial^2 E}{\partial w(j,i)\partial w(k,n)} = \frac{2}{N_v} u(j,k) \sum_{p=1}^{N_v} x_p(i) x_p(n) O'_p(j) O'_p(k) \tag{2.15}$$

$$u(j,k) = \sum_{m=1}^{M} w_{oh}(m,j) w_{oh}(m,k)$$

Elements of the gradient vector $\mathbf{g}_R$ are computed as

$$\frac{\partial E}{\partial w(k,n)} = \frac{2}{N_v} \sum_{p=1}^{N_v} O'_p(k) x_p(n) \sum_{m=1}^{M} (t_p(m) - y_p(m)) \cdot w_{oh}(m,k) \tag{2.16}$$

2.6 Levenberg-Marquardt Method

The widely used Levenberg-Marquardt method [5][6] cleverly combines gradient descent and Newton's method to update the weights. The weight update, similar to Newton's method (refer equation (2.13)) is given by,

$$\triangle \mathbf{W} = -[\mathbf{H} + \lambda \mathbf{I}]^{-1}\mathbf{g} \tag{2.17}$$

where λ is a user chosen parameter that is scaled according to the mean square error in a training iteration. If the error decreases following an update, then λ is decreased by, say a factor of 10, whereas, if the error increases, then λ is increased by the same factor.

LM works well in practice and has good convergence. The λ parameter makes the Hessian diagonally dominant, so an inverse exists. Also, it can be seen from equation (2.17) that LM follows Newton's method for small λ and gradient descent for large values of λ. However, calculation and inversion of the Hessian is still required in every iteration. Also, the addition of the parameter λ and the method for determining it are completely heuristic.

2.7 Output Weight Optimization-Hidden Weight Optimization

Output weight optimization-hidden weight optimization (OWO-HWO) [30] is very similar to OWO-BP. However, HWO minimizes the objective function

$$E_\delta(j) = \sum_{p=1}^{N_v} \left[\delta_p(j) - \sum_{n=1}^{N+1} g_{hwo}(k,n) x_p(n) \right]^2 \tag{2.18}$$

for $0 \leq i \leq N_h$, by solving for linear equations of the form

$$\sum_{n=1}^{N+1} g_{hwo}(k,n) r(n,m) = \frac{-\partial E}{\partial w(j,m)} \tag{2.19}$$

In matrix notation,

$$\mathbf{G}_{hwo} \cdot \mathbf{R} = \mathbf{G}_{bp} \tag{2.20}$$

where, $\mathbf{G}_{bp}$ is the backpropagation gradient, $\mathbf{R}$ is the input auto-correlation matrix and $\mathbf{G}_{hwo}$ is the HWO weight changes. The weights are updated as,

$$w(k,n) \leftarrow w(k,n) + z \cdot g_{hwo}(k,n)$$

where z is the optimal learning factor. The linear equation in 2.20 can be solved for $\mathbf{G}_{hwo}$ using conjugate gradient, orthogonal least squares (OLS) or matrix inversion

using the singular value decomposition (SVD). It is shown in appendix B that OWO-HWO is equivalent to linearly transforming the training data and then performing OWO-BP. The weights connected to the outputs are adapted using OWO mentioned in 2.2.

2.8 Discussion

Having reviewed the basics of MLP training, we can make the following comments.

(i) First order methods are slow to converge

(ii) Second order methods are slow between training iterations and require a lot of computational resource

(iii) Both first and second order methods are negatively impacted by presence of linear dependence

We now move on to modeling and analyzing the effect of linear dependence on MLP training.

CHAPTER 3

EFFECT OF LINEAR DEPENDENCE ON LEARNING

Presence of linear dependence can affect the learning ability of feed-forward networks. A simple unifying theory is required to (i) model the dependence and (ii) analyze its effect on MLP training (both first and second order methods). In this chapter we present a detailed mathematical analysis of the effect of linear dependencies on MLP training. In section 1.3 we looked at the layered architecture of the MLP. Linear dependence can occur in any of these layers and can affect the overall training. We have identified and categorized these dependencies as follows:

1. Linearly dependent inputs
2. Linearly dependent hidden units
   - 2.1 Hidden units dependent upon inputs
   - 2.2 Hidden units dependent upon other hidden units

We begin by establishing the concept of *equivalent networks*. Then, for each case listed above, we model for the linear dependence and use the concept of equivalent networks to analyze the effect of dependency. We will examine the effects of linear dependence on backpropagation and Newton's method. In general, these results can be extended to any training algorithm that uses gradient information for training.

3.1 Equivalent Networks: A Concept

The concept of equivalent networks [42] is very important for the analysis of linear dependency, which will be the focus of this section. Feed forward neural networks, such as the MLP are sensitive to the choice of initial weights [52]. Given a

training data set and a fixed architecture for the MLP, the final mean square error after training is not guaranteed to be the same every time the network is trained. This is because the training is largely dependent on the initialization of the weights in the network.

Let $\{\mathbf{x}_p, \mathbf{t}_p\}_{p=1}^{N_v}$ denote the original training data and let $\mathbf{A}$ be a matrix of rank $(N+1)$ that transforms input vector $\mathbf{x}_p$ to $\mathbf{x}'_p$ as $\mathbf{x}'_p = \mathbf{A} \cdot \mathbf{x}_p$. $\{\mathbf{x}'_p, \mathbf{t}_p\}_{p=1}^{N_v}$ denotes the transformed data. In order to analyze the effect of the transformation, we define the concept of equivalent networks.

Definition 3-1 *A network trained with* $\{\mathbf{x}_p, \mathbf{t}_p\}$ *and one trained with* $\{\mathbf{x}'_p, \mathbf{t}_p\}$ *are equivalent if the output vectors,* $\mathbf{y}_p$ *and* $\mathbf{y}'_p$ *are identical.*

We must ensure that the networks trained on the original and transformed data have the exact same starting point. This translates to ensuring the two networks have identical hidden unit activations and identical outputs for every training pattern. The concept of equivalent networks is graphically illustrated below.

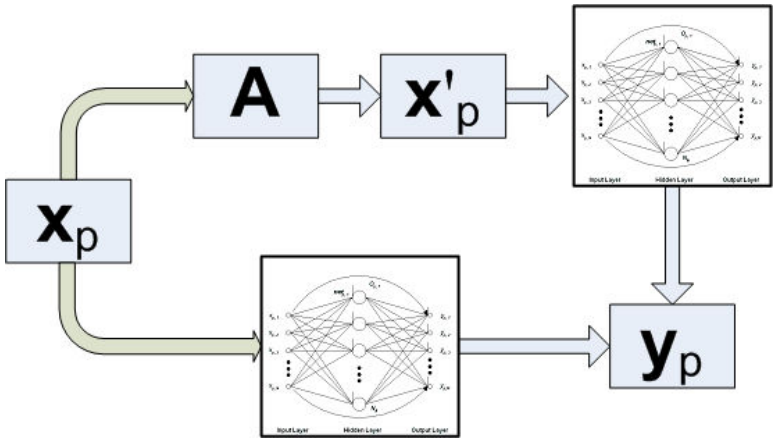


Figure 3.1. Illustration: Concept of Equivalent Networks.

If we have to analyze the effect of a transformation $\mathbf{A}$ on the training of a network, then it is logical to start by initializing a network that will train on the

transformed data. Subsequently,we can absorb this initialization into another equivalent network that will train on the original data so as to ensure common starting points. The concept of weight absorption also makes for a fair comparison [53]. The concept of equivalent networks, though simple is very effective in

(i) analyzing linear dependency

(ii) detecting the point of deviation of two comparable schemes

(iii) providing a framework for a fair comparison

In the following sections, we use the concept of equivalent networks and present a detailed analysis of the effects of linear dependence on network training. We first look at how linear dependence affects backpropagation, subsequently, we will do a similar analysis on Newton's method.

3.2 Effect of Dependence on Backpropagation

This section analyzes the effect of dependent inputs on learning using BP-type algorithm. The expression for steepest descent gradient is derived for the case of dependent and independent inputs. Equivalent networks concept is used to analyze the effect on learning. We start by asking the question, given an MLP with linearly independent inputs, what is the effect of adding some dependent inputs on training?

3.2.1 Linearly Dependent Inputs

3.2.1.1 Modeling Dependent Inputs

Let the $(N+1)$ elements in the input vector $\mathbf{x}_p$ be linearly independent. Let $(K-1)$ additional linearly dependent inputs be added as

$$x_p(N+1+k) = \sum_{m=1}^{N+1} a(N+1+k, m) x_p(m) \tag{3.1}$$

where k varies from $1 \leq k \leq (K-1)$. Let the first $(N+1)$ rows of the matrix $\mathbf{A}$ have diagonal elements equal to 1, with the off-diagonal elements equal to 0. $\mathbf{A}$ is $(N+K)$ by $(N+1)$ and has rank $(N+1)$. The input vector augmented with the dependent elements is then

$$\mathbf{x}'_p = \mathbf{A} \cdot \mathbf{x}_p \tag{3.2}$$

where the first $(N+1)$ elements of $\mathbf{x}'_p$ satisfy $\mathbf{x}'_p = \mathbf{x}_p(n)$. For the case where $N = 2$ and $K = 2$, the matrix $\mathbf{A}$ is

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ a_{41} & a_{42} & a_{43} \end{bmatrix} \tag{3.3}$$

3.2.1.2 An Equivalent Compact Network

Our goal is to analyze the effects of these linearly dependent inputs on MLP training. First, assume that a dependent network with augmented input vectors $\mathbf{x}'_p$ is being trained. For the p^{th} pattern, the hidden unit net function vector $\mathbf{n}_p$ is expressed as

$$\mathbf{n}_p = \mathbf{W}' \cdot \mathbf{x}'_p \tag{3.4}$$

Next, we want to construct an equivalent network, with input vectors $\mathbf{x}_p$, whose hidden unit activations and outputs take on the same values as those in the augmented network, pattern by pattern. Starting with the equation above we have

$$\begin{aligned} n_p(k) &= \sum_{n=1}^{N+K} w'(k,n)x'_p(n) \\ &= \sum_{n=1}^{N+1} w'(k,n)x'_p(n) + \sum_{n=N+2}^{N+K} w'(k,n)x'_p(n) \\ &= \sum_{n=1}^{N+1} w'(k,n)x_p(n) + \sum_{n=N+2}^{N+K} w'(k,n)x'_p(n) \end{aligned} \tag{3.5}$$

Rewriting and expanding the second term, we have

$$\begin{aligned} \sum_{m=N+2}^{N+K} w'(k,m)x'_p(m) &= \sum_{m=N+2}^{N+K} w'(k,m) \sum_{n=1}^{N+1} a(m,n)x_p(n) \\ &= \sum_{n=1}^{N+1}[\sum_{m=N+2}^{N+K} w'(k,m)a(m,n)]x_p(n) \end{aligned} \tag{3.6}$$

The weights of this compact, equivalent network are now

$$w(k,n) = w'(k,n) + \sum_{m=N+2}^{N+K} w'(k,m)a(m,n) \tag{3.7}$$

Since the compact network is equivalent to the dependent one, we can infer that

$$\mathbf{n}_p = \mathbf{W}' \cdot \mathbf{x}'_p = \mathbf{W} \cdot \mathbf{x}_p \tag{3.8}$$

Unsurprisingly, it is easy to find a compact, equivalent network to one which has linearly dependent inputs. Substituting equation 3.2 into 3.4, we can get an expression for the weights in the equivalent network as

$$\mathbf{W}' \cdot \mathbf{A} = \mathbf{W} \tag{3.9}$$

3.2.1.3 Training the Dependent Network

Here, we want to compare training of the dependent net with that of the compact equivalent net. Modifying equations 2.2 and 2.10 for the dependent network, we get

$$g'(k,n) \equiv \frac{-\partial E}{\partial w'_{kn}} = \frac{1}{N_v} \sum_{p=1}^{N_v} \delta_p(k) x'_p(k) \tag{3.10}$$

and

$$\mathbf{G}' = \frac{1}{N_v} \sum_{p=1}^{N_v} \boldsymbol{\delta}_p (\mathbf{x}'_p)^T \tag{3.11}$$

Using equation 3.2 in 3.11, we get

$$\mathbf{G}' = \mathbf{G} \cdot \mathbf{A}^T$$

So the gradient matrix for the dependent matrix is that of the equivalent network times $\mathbf{A}^T$. Using 3.9, we can now map $\mathbf{G}'$ back to the equivalent network as

$$\mathbf{G}'' = \mathbf{G} \cdot \mathbf{A}^T \mathbf{A} \tag{3.12}$$

$\mathbf{G}''$ denotes $\mathbf{G}'$ after it is mapped back to the compact equivalent network. Some interesting lemmas result from 3.7.

Lemma 3-1: If we are at a local minimum in the weight space of the dependent network, we are also at a local minimum in the weight space of the compact equivalent network.

Lemma 3-2: If backpropagation (BP) is used to train the dependent network's input weight matrix $\mathbf{W}'$, this is not equivalent to applying BP to the compact network's weight matrix $\mathbf{W}$.

In other words, BP in a dependent network is not equivalent to BP in the compact equivalent network. Training in the dependent network may not converge to a local minimum equivalent to that in the compact network.

3.2.1.4 Illustration

We present some illustrations for the case of linearly dependent inputs. In all the simulations, the network to be trained on the dependent data was initialized first and then the network to be trained on the independent data was initialized using the weights from the dependent network, as explained above.

The figure 3.2 is for training on the F-17 data [30]. K here represents the number of dependent inputs generated in each case, i.e., the dependent network is trained on K additional inputs while the independent network is trained on the original data without dependent inputs. However, both network are initialized to be equivalent in the first training iteration.

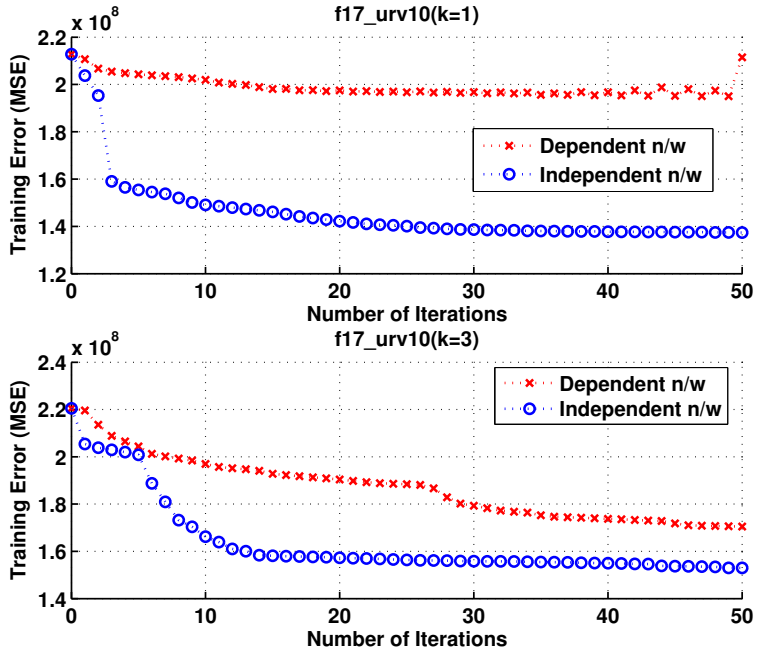


Figure 3.2. Dependent and independent network performance on F-17 data.

The plot in figure 3.3 results from training on the FM-train data file [54]. In all

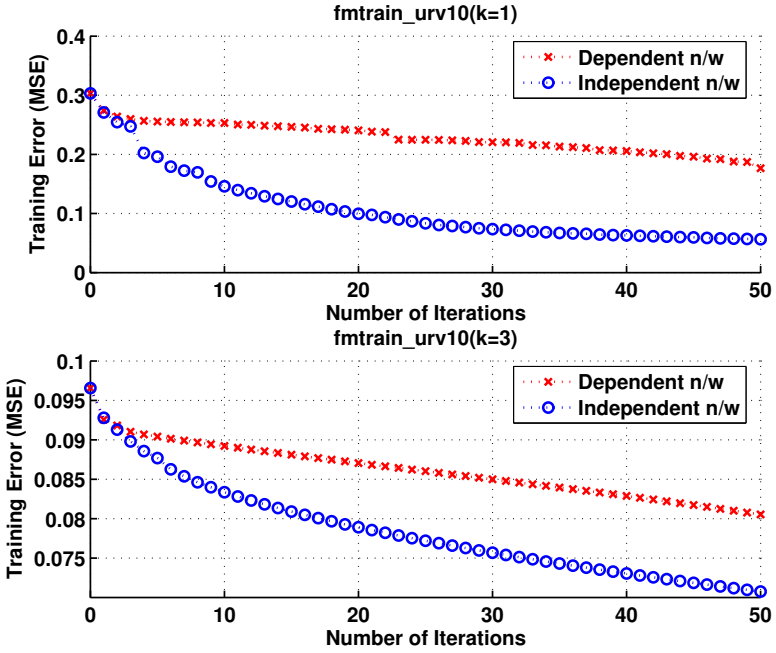


Figure 3.3. Dependent and independent network performance on FMTRAIN data.

plots, the identical initialization of the two networks, using the concept of equivalent networks is evident in the error being the exact same for the first iteration.

Despite starting at the same point, the two networks train very differently. Contrary to intuition, linear dependency is not always bad as can be seen in figure 3.3. A network trained on linearly independent data is not guaranteed to be better. Sometimes linear dependence seems to help. However, one thing is for certain, the training is different for a network with dependent inputs and for the one without.

3.2.2 Hidden Units Dependent Upon Inputs

In this subsection we are interested in analyzing BP when some hidden units' outputs are linearly dependent upon the inputs. This special case can occur when the net function in equation 3.4 falls in the linear region of the sigmoid activation.

This would make the hidden unit output to be 0.5 plus a constant multiple of the net function. For our dependent network, the assumptions are as follows:

A1 Hidden unit activations $O_p(1)$ through $O_p(N_h)$ are not linear1y dependent on the inputs.

A2 Hidden unit activations $O_p(N_h+1)$ through $O_p(N_h+K)$ are linear1y dependent on the inputs.

A3 The bypass weight in the dependent network, which maps the n^{th} input to the i^{th} output, is denoted by $w_{oi}(i,n)$.

3.2.2.1 Modeling Input-dependent Hidden Units

For k between 1 and K, assumption A2 can be expressed as

$$O_p(N_h+k) = 0.5 + c_k n_p(N_h+k) \quad = c_k \sum_{n=1}^{N+1} w'(N_h+k,n) x_p(n) \tag{3.13}$$

c_k denotes the a constant that multiplies the net function.

In a compact network equivalent to the dependent one, the bypass weights from input to output are

$$w_{oi}(i,n) = w'_{oi}(i,n) + \sum_{k=1}^{K} c_k \cdot w'_{oh}(i,N_h+k) w'(N_h+k,n) \tag{3.14}$$

3.2.2.2 Training the Dependent Network

If BP is applied to bypass weights in the compact equivalent network, we get weight changes as

$$z.g_{oi}(i,n) \equiv z \cdot \frac{-\partial E_p}{\partial w_{oi}(i,n)} = z \cdot \delta_{po}(i) x_p(n) \tag{3.15}$$

In the dependent network, we have

$$g'_{oi}(i,n) \equiv \frac{-\partial E_p}{\partial w'_{oi}(i,n)} = \delta_{po}(i)x_p(n) \tag{3.16}$$

$$g'_{oh}(i,N_h+k) \equiv \frac{-\partial E_p}{\partial w'_{oh}(i,N_h+k)} = \delta_{po}(i)c_n n_p(N_h+k) \tag{3.17}$$

$$g'(N_h+k,n) \equiv \frac{-\partial E_p}{\partial w'(N_h+k,n)} = \delta_p(N_h+k)x_p(n) \tag{3.18}$$

Assuming that the learning factor used in the dependent network is z_1, the resulting weight change in the equivalent network is

$$\begin{aligned}\triangle w_{oi}(i,n) = {} & z_1 \cdot g'_{io}(i,n) \\ & + \sum_{c=1}^{K} c_k[z_1 \cdot g'_{oh}(i,N_h+k)w'(N_h+k,n) \\ & + z_1 \cdot w'_{oh}(i,N_h+k)g'(N_h+k,n) \\ & + (z_1)^2 \cdot g'_{oh}(,N_h+k) \cdot g'(N_h+k,n)]\end{aligned} \tag{3.19}$$

3.2.2.3 Analysis

We see that the first term in equation 3.19 is similar to equation 3.15, but that the other terms are very different. This leads to the following theorem.

Theorem 3-1: If BP is used to train the dependent network's hidden units that are linearly dependent on the inputs, this is not equivalent to applying BP to the compact network's bypass weights.

Proof: The right hand sides of equations 3.15 and 3.19 are not equal.

In other words, BP in a dependent network is not equivalent to steepest descent in the compact equivalent network. Training in the dependent network may not converge to a local minimum equivalent to that in the compact network.

3.2.3 Linearly Dependent Hidden Units

In this subsection we are interested in analyzing BP when some hidden units' outputs are linearly dependent upon the outputs of other hidden units. This is different from the case discussed in section 3.2.2 in that the outputs of the hidden units are all non-linear functions of the input, but it does not guarantee that outputs of the hidden units are linearly independent. For our dependent network, the assumptions are as follows:

A1 Hidden unit activations $O_p(1)$ through $O_p(N_h)$ are not linear1y dependent on each other.

A2 Hidden unit activations $O_p(N_h+1)$ through $O_p(N_h+K)$ are linear1y dependent on hidden unit activations $O_p(1)$ through $O_p(N_h)$.

A3 Weights from hidden units to output units are denoted by $w'_{oh}(i,k)$.

3.2.3.1 Modeling Dependent Hidden Units

For k between 1 and K, assumption A2 can be expressed as

$$O_p(N_h+k) = \sum_{n=1}^{N_h} b(N_h+k,n) O_p(n) \tag{3.20}$$

where the unknown coefficients $b(m,n)$ are the elements of the matrix $\mathbf{B}$ of dimension $(N_h+K)\times N_h$, similar to $\mathbf{A}$ defined in 3.3. A compact network equivalent to the dependent one, has N_h hidden units identical to the first N_h hidden units in the dependent network. The weights from the hidden layer to the output layer however would be

$$w_{oh}(i,k) = w'(_{oh}(i,k) + \sum_{m=N_h+1}^{N_h+K} w'_{oh}(i,m) b(m,k) \tag{3.21}$$

where k varies from 1 to N_h and i varies from 1 to M.

3.2.3.2 Training the Dependent Network's Output Weights

Here, we want to compare training of the dependent network with training of the compact equivalent network. While training the equivalent network, the weight change for the p^{th} pattern for $w_{oh}(i,k)$ is proportional to

$$\frac{-\partial E_p}{\partial w_{oh}(i,k)} = \delta_{po}(i) O_p(k) \tag{3.22}$$

where k varies from 1 to N_h. For the dependent network, the first N_h hidden units are identical to those in the equivalent network. So, we have

$$\frac{-\partial E_p}{\partial w'_{oh}(i,k)} = \delta_{po}(i) O_p(k) \tag{3.23}$$

where k varies from 1 to $N_h + K$. Based upon equation 3.21, the p^{th} pattern's weight change in the dependent network, when mapped to an equivalent network, is proportional to

$$\begin{aligned}
&\frac{-\partial E_p}{\partial w'_{oh}(i,k)} + \sum_{m=N_h+1}^{N_h+K} \frac{-\partial E_p}{\partial w'_{oh}(i,m)} b(m,k) \\
&\quad = \delta_{po}(i) O_p(k) + \sum_{m=N_h+1}^{N_h+K} \delta_{po}(i) O_p(m) b(m,k) \\
&\quad = \delta_{po}(i) [O_p(k) + \sum_{m=N_h+1}^{N_h+K} \sum_{n=1}^{N_h} b(m,n) O_p(n) b(m,k)] \\
&\quad = \delta_{po}(i) [O_p(k) + \sum_{n=1}^{N_h} O_p(n) \sum_{m=N_h+1}^{N_h+K} b(m,n) b(m,k)]
\end{aligned} \tag{3.24}$$

If we define

$$c(n,k) \equiv \sum_{m=N_h+1}^{N_h+K} b(m,n) b(m,k) \tag{3.25}$$

we have,

$$\begin{aligned}
g''_{oh}(i,k) &= g_{oh}(i,k) + \sum_{n=1}^{N_h} g_{oh}(i,n) c(n,k) \\
\mathbf{G''_{oh}} &= \mathbf{G_{oh}}(\mathbf{I} + \mathbf{C}) = \mathbf{G_{oh}}(\mathbf{I} + \mathbf{B^T B})
\end{aligned} \tag{3.26}$$

Comparing the last line of equation 3.26 with the right hand side of equation 3.22, we see that they are different. This leads to the following theorem.

Theorem 3-2: If BP is used to train the output weights of the dependent network's hidden units, this is not equivalent to applying BP to the compact equivalent network's hidden unit output weights.

Proof: See equation 3.26.

As in the previous cases, BP in a dependent network is not equivalent to BP in a compact equivalent network. Training in the dependent network may not converge to a local minimum equivalent to that in the compact network.

3.2.3.3 Training the Dependent Network's Input Weights

Here, we want to compare training of the dependent net's input weights with training of the compact equivalent net's input weights. Starting with equation 2.9, the negative gradient of the dependent network's input weights is

$$g'(k,n) = \frac{\partial E}{\partial w'_{kn}} = \frac{1}{N_v}\delta'_p(k)x'_p(n) \tag{3.27}$$

where k varies from 1 to $N_h + K$. Substituting equation 2.8 into equation 3.27, we get

$$\begin{aligned} g'(k,n) &= \sum_{i=1}^{M} w'_{oh}(i,k)u_k(n,i) \\ u_k(n,i) &= \frac{1}{N_v}\sum_{p=1}^{N_v} f'(n_p(k))\delta_{po}(i)x_p(n) \end{aligned} \tag{3.28}$$

Assume that equation 3.21 maps the dependent network's output weights to a compact network's output weights $w_{oh}(i,k)$. For k between 1 and N_h, the compact

network's hidden units have identical input weights, net functions, and activations. Equations 3.27 and 3.28 for this network are rewritten as

$$\begin{aligned} g(k,n) &= \frac{\partial E}{\partial w_{kn}} = \frac{1}{N_v}\delta_p(k)x_p(n) \\ &= \sum_{i=1}^{M} w'_{oh}(i,k)u_k(n,i) + \sum_{i=1}^{M}\sum_{m=N_h+1}^{N_h+K} w'_{oh}(i,m)b(m,k)u_m(n,i) \\ &= g'(k,n) + \sum_{m=N_h+1}^{N_h+K} b(m,k)g'(m,n) \end{aligned} \tag{3.29}$$

Example:Given a compact network with N_h hidden units, we can construct a dependent network for the $K = 1$ case by copying hidden unit N_h onto hidden unit $N_h + 1$. The output weights satisfy

$$w'_{oh}(i,N_h) = w'_{oh}(i,N_h+1) = \frac{1}{2}w_{oh}(i,N_h) \tag{3.30}$$

The only non-zero b coefficient is $b(N_h, N_h + 1)$, which equals 1. Considering the output weights, we have

$$g''_{oh}(i,N_h) = 2 \cdot g_{oh}(i,N_h) \tag{3.31}$$

because of 3.31. Equation 3.26 gives the same result. For the input weights, we get

$$g'(N_h,n) = \frac{1}{2} \cdot g(N_h,n) \tag{3.32}$$

because the output weights of the two identical hidden units are half-size. This leads to hidden unit delta functions that are half-size. Using the fact that $b(N_h, N_h+1) = 1$ in equation 3.29, we get the same result as equation 3.32. The following theorem follows from equation 3.29.

Theorem 3-3: If BP is used to train the input weights of the dependent network's hidden units, this is not equivalent to applying BP to the compact equivalent network's input weights.

Proof: See the derivation of 3.29.

We have presented a rigorous treatment of the effects of linear dependencies on the training of the network. We have shown that the dependencies can occur in any layer of the network and can have an effect on the training. Experimental results show that linearly dependent signals often have a detrimental effect on neural network training.

3.3 Effect of Linear Dependence on Newton's Method

Even when the reduced size Hessian $\mathbf{H}_R$ is used, Newton's method can use excessive memory, since the computational expense for inverting $\mathbf{H}_R$ is $O(N_{iw}^3)$. In addition, $\mathbf{H}$ and $\mathbf{H}_R$ are usually ill-conditioned or rank deficient [50], as shown in this subsection.

3.3.1 Linearly Dependent Inputs

First, consider the effect of a linearly dependent input on $\mathbf{H}_R$, modeled as

$$x_p(N+2) = \sum_{n=1}^{N+1} b(n) x_p(n)$$

Equation (2.15) can be re-written for the dependent input as,

$$\begin{aligned}
\frac{\partial^2 E}{\partial w(j,N+2)\partial w(u,v)} &= \frac{2}{N_v} v(j,u) \sum_{p=1}^{N_v} \sum_{n=1}^{N+1} b(n) x_p(n) x_p(u) O_p'(j) O_p'(u) \\
&= \sum_{n=1}^{N+1} b(n) \left(\frac{2}{N_v} v(j,u) \sum_{p=1}^{N_v} x_p(n) x_p(u) O_p'(j) O_p'(u) \right) \\
&= \sum_{n=1}^{N+1} b(n) \frac{\partial^2 E}{\partial w(j,n)\partial w(u,v)}
\end{aligned}$$

For a fully connected MLP, this implies that for a fixed j, the $(N+2)^{nd}$ row is a linear combination of $(N+1)$ other rows. The index j represents a hidden unit and varies from 1 to N_h. Expanding on this result, each dependent input generates N_h dependent rows and columns in the Hessian $\mathbf{H}_R$.

3.3.2 Linearly Dependent Hidden Units

Next, consider the effect of a linearly dependent hidden unit on $\mathbf{H}_R$, modeled as

$$O_p(N_h + 1) = \sum_{k=1}^{N_h} c(k) O_p(k)$$

If orthogonal least squares [55] is used to solve for output weights, then the weights from the dependent hidden unit to all outputs would be set to zero, i.e. $w_{oh}(i, N_h + 1) = 0, 1 \leq i \leq M$. From the expression for $u(j, u)$ above, it is clear that $u(j, N_h+1) = u(N_h + 1, u) = 0$. Equation (2.15) for the dependent hidden unit is,

$$\frac{\partial^2 E}{\partial w(N_h + 1, m) \partial w(u, v)} = \frac{\partial^2 E}{\partial w(j, m) \partial w(N_h + 1, v)} = 0$$

Each linearly dependent hidden unit results in $(N + 1)$ zero-valued rows and $(N+1)$ zero-valued columns in $\mathbf{H}_R$. The above analyses show that a single dependent basis function can affect multiple rows and columns in the Hessian, indicating its *sensitivity* to different training conditions.

In this chapter we have provided a thorough mathematical analysis of the effect of linear dependence on MLP trained using backpropagation and Newton's method. Using equivalent networks we showed that in the presence of linear dependence (i) the gradient information in backpropagation will be different and (ii) the Hessian in Newton's method will have linearly dependent rows and columns.

CHAPTER 4

PROPOSED WORK

In this chapter we identify the problem areas and outline the tasks and goals of this dissertation.

4.1 Problems

4.1.1 Computational Complexity of Second Order Methods

Second order methods such as the Newton's method, and its variants, involve computing and storing a giant Hessian in every training iteration in order to update the weights in the network. The Hessian is usually ill-conditioned and LM tries to counter that by heuristically scaling the diagonal elements of the Hessian. Though LM works well in practice, it is extremely slow and resource hungry. The power of LM-type algorithms come at the cost of greatly increased computational loads. Also, second order methods are not free from the effects of linear dependence.

4.1.2 Effects of Input Transformations Are Poorly Understood

Data pre-processing techniques can be characterized as transformations of the inputs. These are mostly ad-hoc and depends on the application and the learning algorithm. It is known that some transformations are beneficial and help reducing the training error, while some other have no effect on reducing the overall error.

4.1.3 Primitive Methods For Accelerating Convergence

Early work on accelerating input weight convergence, such as the delta-bar-delta [43], did so by adapting individual learning factors for each weight. The methods introduce additional user chosen parameters, with no clear guidance or technique on how to choose the best set of parameters. The performance of the methods are dependent on these parameters making it hard to scale them to different applications or make them *generally applicable*.

4.1.4 Training Affected By Dependencies

In chapter 3, we have thoroughly analyzed the effect of linear dependence on training. In general, presence of linear dependence causes the training to be different, slow to converge and even unpredictable at times. Existing algorithms work best with linearly independent training data. However, several real world applications of neural networks are required to handle large amounts of data not guaranteed to be linearly independent. An obvious solution is to perform feature selection or use data pre-processing to detect and eliminate the linear dependence. A better solution would be to develop resilient learning algorithms immune to linear dependence.

4.1.5 Inability to Model Discontinuous Functions

MLPs were made popular by the universal approximation theorem [31], which in essence states that a single layer MLP, with sufficiently large number of nonlinear units can approximate any real continuous function to an arbitrary degree of accuracy. What it does not mention is the fact that MLP training algorithms perform poorly on discontinuous functions. This is in spite of the fact that sigmoidal units can easily be designed by hand to approximate step functions arbitrarily well, simply by increasing the input weight and adjusting the threshold.

4.2 Proposed Goals and Tasks

In this section, we set a goal for each problem listed in section 4.1. We also propose a list of tasks to be carried out in order to achieve these goals.

4.2.1 Towards a Positive Definite Hessian

Equation 2.15 (reproduced here for convenience), gives the expression for the elements of Hessian for all the input weights in the network.

$$\frac{\partial^2 E}{\partial w(j,i)\partial w(k,n)} = \frac{2}{N_v} u(j,k) \sum_{p=1}^{N_v} x_p(i) x_p(n) O'_p(j) O'_p(k)$$

$$u(j,k) = \sum_{m=1}^{M} w_{oh}(m,j) w_{oh}(m,k)$$

where (j,i) specify the row number and (k,n) specify the column number. Using lexicographic ordering for example, the row number m could be calculated from (j,i) as $m(j,i) = j + (i-1)N_h$. Similarly the column number l could be $l(k,n) = k + (n-1)N_h$. The Hessian is N_{iw} by N_{iw} where N_{iw} is $(N+1)N_h$. If some rows and columns in $\mathbf{H}_R$ are linearly dependent upon others, it may be desirable to delete them before using in Newton's algorithm. Let K be the set of linearly independent rows of $\mathbf{H}_R$. Let $\mathbf{S}_K$ be an N_{iw} by N_{iw} identity matrix, after rows not enumerated in set K are deleted. The Hessian for the remaining linearly independent rows and columns is

$$\mathbf{H}_d = \mathbf{S}_K \mathbf{H}_R \mathbf{S}_K^T \tag{4.1}$$

A more general case is

$$\mathbf{H}_T = \mathbf{T} \mathbf{H}_R \mathbf{T}^T \tag{4.2}$$

where $\mathbf{T}$ is any L by N_{iw} matrix with $L < N_{iw}$. Elements of $\mathbf{H}_T$ are weighted sums of elements of $\mathbf{H}_R$. In this case, if $\mathbf{H}_R$ is decomposed as $\mathbf{V}^T\mathbf{V}$, then

$$\mathbf{H}_T = \mathbf{U}^T \mathbf{U} \tag{4.3}$$

where

$$\mathbf{U} = \mathbf{V}\mathbf{T}^T \tag{4.4}$$

Lemma 4-1: $\mathbf{H}_T$ is nonnegative definite. This follows from equations 4.3 and 4.4.

Lemma 4-2: $\mathbf{H}_T$ is positive definite for some matrices $\mathbf{T}$, even if $\mathbf{H}_R$ is singular. This follows if $\mathbf{T} = \mathbf{S}_K$.

Our primary goal is to find matrices $\mathbf{T}$, as in 4.2, such that $\mathbf{H}_T$ is small and positive definite and can be used in Newton's algorithm.

4.2.2 An Optimal Input Transformation

We present an optimal input gain algorithm that in effect is a diagonal transformation that best transforms each input to guarantee a reduction in the training error. We show that orthogonal transforms are useless when it comes to reducing the training error and proceed to derive an optimal non-singular, non-orthogonal diagonal transform that will scale each input individually in order to reduce the overall error. We present detailed analysis that relates the method to Newton's method and compare it with existing algorithms using several publicly available data sets.

4.2.3 Learning using Multiple Learning Factors

We present an algorithm that assigns individual optimal learning factor to each hidden unit. The algorithm uses Newton's method to simultaneously compute the multiple optimal learning factors in every iteration. Compared to the traditional backpropagation method that uses a single optimal learning factor and LM, the proposed algorithm displays a superior performance in terms of overall training error and speed of operation, as will be evident in the results.

4.2.4 Countering Dependencies During Training

We extend the optimal input gain and the multiple optimal learning factor algorithms to OWO-HWO type algorithm and show that these algorithms are resilient in the presence of linearly dependent inputs and/or hidden units. This makes them very powerful and robust, with a built-in ability to counter dependence that is absent in most training algorithms. Both algorithms use Newton-type approach to compute the learning parameters. We will relate the structure of Hessian that reveals the overall saving in computation for the two methods.

4.2.5 Approximating Discontinuous Functions

Experiments with MLP trained using BP, OWO-BP and classic Newton's method indicate that they perform poorly in approximating simple discontinuous functions. LM provides some hope as it is able to model a discontinuous step function, but it quickly fades when extended to other discontinuous functions. We propose to demonstrate a method for training separate networks and fusing them to form a network with locally adapted units that can model simple discontinuities.

CHAPTER 5

THE OPTIMAL INPUT GAIN ALGORITHM

In this chapter, an effective batch training algorithm is developed for the multilayer perceptron. First, the effects of input transforms are reviewed and explained, using the concept of equivalent networks. Next, a non-singular diagonal transform matrix for the inputs is proposed. Use of this transform is equivalent to altering the input gains in the network. Newton's method is used to solve for the input gains and an optimal learning factor. In several examples, it is shown that the final algorithm is a reasonable compromise between first order training methods and Levenburg-Marquardt.

Non-singular input transforms are reviewed in section 5.1. It is shown that nonsingular orthogonal transforms of inputs are useless. In section 5.2.3, the emphasis is therefore on non-orthogonal transforms. The simplest such transform, consisting of a diagonal matrix, is introduced. It is shown to be equivalent to modifying the gains on the network inputs. A combination of the optimal input gains (OIGs) and optimal learning factor (OLF) are then found, using Newton's method. In section 5.4, the computational burdens of the OIG and other training algorithms are presented for comparison. Numerical results and discussions are presented in sections 5.5 and 5.7.

5.1 Linear Transformation of Inputs

Neural network training algorithms are sensitive to initialization and also data dependent. In the past, researchers have proposed pre-processing techniques in order to accelerate the training and make it less data dependent.

In equation 3.12 of section 3.2, we saw that

$$\mathbf{G}'' = \mathbf{G} \cdot \mathbf{A}^T \cdot \mathbf{A}$$

where $\mathbf{G}$ is the input weight gradient matrix for the original network, and $\mathbf{G}''$ denotes the gradient matrix of an equivalent network with transformed inputs, mapped back to the original network.

Lemma 5-1 If the transformation matrix $\mathbf{A}$ is orthogonal, then training a network using OWO-BP is equivalent to training an equivalent network with transformed inputs and mapping the weight changes back to the original network.

The above lemma implies that it is useless to apply a transformation matrix $\mathbf{A}$ if it is orthogonal, since $\mathbf{A}^T\mathbf{A}$ will be an identity matrix. Therefore orthogonal transforms do not help reduce the overall mean square error during training.

So far, it is clear that $\mathbf{A}$ should not be orthogonal, and should not have more rows than columns. In this section, we discuss the utility of non-singular, non-orthogonal $\mathbf{A}$ matrices, with $N' = (N + 1)$.

5.1.1 A Useful Non-orthogonal Transform Matrix

Suppose that our input vectors $\mathbf{x}_p$ are biased such that $E[\mathbf{x}_p] = \mathbf{m}$. A zero-mean version of $\mathbf{x}_p$ is $\mathbf{x}'_p$ which satisfies

$$\mathbf{x}_p = \mathbf{x}'_p + \mathbf{m} \tag{5.1}$$

For the zero-mean data, equation 2.11 may now be written as

$$\mathbf{G}' = \frac{1}{N_v} \sum_{p=1}^{N_v} \boldsymbol{\delta}_p (\mathbf{x}'_p)^T \tag{5.2}$$

For the non-zero-mean data $\mathbf{x}_p$ in (5.1), we have

$$\begin{aligned}\mathbf{G} &= \frac{1}{N_v}\sum_{p=1}^{N_v}\boldsymbol{\delta}_p(\mathbf{x}'_p+\mathbf{m})^T \\ &= \left(\frac{1}{N_v}\sum_{p=1}^{N_v}\boldsymbol{\delta}_p\right)\cdot\mathbf{m}^T+\mathbf{G}'\end{aligned} \tag{5.3}$$

where $\mathbf{G}$ is the negative Jacobian matrix of the input weights for an equivalent network trained on biased input vectors $\mathbf{x}_p$. Note that all of the useful information in $\mathbf{x}_p$ resides in $\mathbf{x}'_p$. As elements of $\mathbf{m}$ become larger however, the first term of $\mathbf{G}$ dominates the expression, and $\mathbf{G}$ has little useful information. Training then has no effect on the network. Consequently, several authors have pointed out the need to make MLP inputs zero-mean [56].

Now assume that the unbiased input vectors $\mathbf{x}_p$ are transformed as $\mathbf{x}'_p = A\mathbf{x}_p$ where $\mathbf{A}$ is defined as

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & m_1 \\ 0 & 1 & 0 & 0 & \dots & m_2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & m_N \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

Then we have

$$\mathbf{x}'_p = \mathbf{A}\cdot\mathbf{x}_p = \begin{bmatrix} x_p(1)+m_1 \\ x_p(2)+m_2 \\ \cdot \\ \cdot \\ x_p(N)+m_N \\ 1 \end{bmatrix}$$

Mapping the gradient back to a network with inputs $\mathbf{x}$ using 3.12, we get

$$\mathbf{G}'' = \mathbf{G} \cdot (\mathbf{A}^T \cdot \mathbf{A})$$

where

$$\mathbf{A}^T \cdot \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & -m_1 \\ 0 & 1 & 0 & 0 & \dots & -m_2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & -m_N \\ -m_1 & -m_2 & -m_3 & \dots & -m_N & \alpha \end{bmatrix}$$

and

$$\alpha = 1 + \sum_{n=1}^{N} m_n^2$$

Since multiplication by $\mathbf{A}$ removes the bias from $\mathbf{x}_p$, we see that a non-orthogonal transform matrix can be useful.

5.2 Optimal Input Gain Algorithm

In this subsection, we discuss the utility of non-singular, non-orthogonal $\mathbf{A}$ matrices, with $N' = (N + 1)$.

5.2.1 A Diagonal Transform Matrix

The simplest non-orthogonal, nonsingular transform matrix $\mathbf{A}$ is diagonal. For this case, let $a(k)$ initially denote the k^{th} diagonal element of $\mathbf{A}^T\mathbf{A}$. Also, the elements of $\mathbf{x}'_p$ are simply scaled versions of $\mathbf{x}_p$. Following (3.12) we get

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix} a(1) & 0 & \dots & 0 & 0 \\ 0 & a(1) & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & a(N) & 0 \\ 0 & 0 & \dots & 0 & a(N+1) \end{bmatrix} \tag{5.4}$$

Instead of using the negative Jacobian elements $g(k,n)$ in training the network, we use $g(k,n) \cdot a(n)$. Note also that the optimal learning factor (OLF) z can be absorbed into the gains $a(n)$.

5.2.2 Derivation of the Optimal Gain Coefficients

Assume that the MLP is being trained using OWO-BP. Given the negative Jacobian $\mathbf{G}$ of dimension N_h by $(N+1)$, the output of the network can be written in terms of unknown gain coefficients as

$$y_p(i) = \sum_{n=1}^{N+1} w_{oi}(i,n)x_p(n) + \sum_{k=1}^{N_h} w_{oh}(i,k) \cdot f\left(\sum_{n=1}^{N+1}(w(k,n) + a(n) \cdot g(k,n))x_p(n)\right) \tag{5.5}$$

where, $f(\cdot)$ is the activation function of the hidden unit, modified to reflect the addition of OIG coefficients. The error function being minimized with respect to the $a(n)$'s is given in (2.5). The first partial of E with respect to $a(m)$ is

$$\begin{aligned} g_{ig}(m) &\equiv \frac{\partial E}{\partial a(m)} \\ &= \frac{-2}{N_v} \sum_{p=1}^{N_v} x_p(m) \sum_{i=1}^{M} \left[t'_p(i) - \sum_{k=1}^{N_h} w_{oh}(i,k) f(n_p(k)) \right] \\ &\cdot \sum_{k=1}^{N_h} w_{oh}(i,k) f'(n_p(k)) g(k,m) \\ &= \frac{-2}{N_v} \sum_{p=1}^{N_v} x_p(m) \sum_{i=1}^{M} [t_p(i) - y_p(i)] \cdot v(i,m) \end{aligned} \tag{5.6}$$

Here, $g(k,m)$ is an element of the negative Jacobian matrix $\mathbf{G}$ in equation (2.10), and $f'(n_p(k))$ denotes the derivative of $f(n_p(k))$ with respect to its net function. Then,

$$\begin{aligned} t'_p(i) &= t_p(i) - \sum_{n=1}^{N+1} w_{oi}(i,n) x_p(n) \\ n_p(m) &= \sum_{n=1}^{N+1} (w(m,n) + a(n) \cdot g(m,n)) x_p(n) \\ v(i,m) &= \sum_{k=1}^{N_h} w_{oh}(i,k) f'(n_p(k)) g(k,m) \end{aligned} \tag{5.7}$$

The second partial derivatives are given by

$$\begin{aligned} h_{ig}(m,m) &\equiv \frac{\partial^2 E}{\partial a(m)^2} \\ &= \frac{2}{N_v} \sum_{p=1}^{N_v} x_p^2(m) \sum_{i=1}^{M} v^2(i,m) \end{aligned} \tag{5.8}$$

$$\begin{aligned} h_{ig}(m,u) &\equiv \frac{\partial^2 E}{\partial a(m) \partial a(u)} \\ &= \frac{2}{N_v} \sum_{p=1}^{N_v} x_p(m) x_p(u) \sum_{i=1}^{M} v(i,m) v(i,u) \end{aligned} \tag{5.9}$$

5.2.3 Implementation Steps

Given $\mathbf{g}$ and the Hessian $\mathbf{H}_{ig}$, we minimize E with respect to the vector $\mathbf{a}$ using Newton's method. Now we have a choice. In each iteration, we can (i) use $\mathbf{A}^T\mathbf{A}$ to transform the gradient matrix as in (3.12), or we can (ii) decompose $\mathbf{A}^T\mathbf{A}$ to find $\mathbf{A}$ using an SVD approach. We can then transform the input data and use OWO-BP with the optimal learning factor. However, this latter approach is too inefficient to consider, even when $\mathbf{A}$ is diagonal. We use the first approach.

In each iteration of the training algorithm, the steps are as follows:

(i) Calculate the input weight Jacobian $\mathbf{G}$ using BP.

(ii) Calculate the OLF-input gain products $a(n)$

(iii) Update the input weights as

$$w(k,n) \leftarrow w(k,n) + a(n)g(k,n)$$

(iv) Solve linear equations for all output weights

Here, the optimal input gain (OIG) procedure has been inserted into the OWO-BP algorithm. It can be inserted into other algorithms as well, including standard BP.

5.3 OIG Analyses

The equation for OIG Hessian can be re-written as,

$$h_{ig}(m,u) = \sum_{k=1}^{N_h}\sum_{j=1}^{N_h}\left[\frac{2}{N_v}\sum_{p=1}^{N_v} x_p(m)x_p(u)f'(n_p(k))f'(n_p(k))\sum_{i=1}^{M} w_{oh}(i,k)w_{oh}(i,j)\right] \cdot g(k,m)\cdot g(j,u)$$

The term within the square brackets is nothing but an element from the Hessian of Newton's method for updating input weights. Hence,

$$h_{ig}(m,u) = \sum_{k=1}^{N_h}\sum_{j=1}^{N_h}\left[\frac{\partial^2 E}{\partial w(k,m)\partial w(j,u)}\right] g(k,m)\cdot g(j,u) \tag{5.10}$$

For fixed (m,u), the above equation can be expressed in vector notation as,

$$\begin{aligned} h_{ig}(m,u) &= \sum_{k=1}^{N_h} g_m(k) \sum_{j=1}^{N_h} h_N^{m,u}(k,j)\cdot g_u(j) \\ &= \mathbf{g}_m^T \mathbf{H}_N^{mu} \mathbf{g}_u \end{aligned} \tag{5.11}$$

where, $\mathbf{g}_m$ is the m^{th} column of the gradient matrix $\mathbf{G}$ and $\mathbf{H}_N^{m,u}$ is the matrix formed by choosing elements from the Newton's Hessian for weights connecting inputs (m,u) to all hidden units.

Equation 5.11 gives the expression for one element of the OIG Hessian. Each element of the OIG Hessian combines the information from N_h rows and columns of the Newton Hessian. This can be seen as compressing the original Newton Hessian of dimension $N_h \times (N+1)$ to simply $(N+1)$. OIG effectively *encodes* the information from the Newton Hessian into a smaller dimension. This makes OIG less sensitive to input conditions and facilitates faster computation.

From equation 5.10, we see that the Hessian from Newton's method uses four indices (j,m,u,k) and can be viewed as a 4-dimensional array, represented by $\boldsymbol{\mathcal{H}}_N^4 \in \mathbb{R}^{N_h\times(N+1)\times(N+1)\times N_h}$. Using this representation, we can express a 4-dimensional OIG Hessian as

$$\boldsymbol{\mathcal{H}}_{ig}^4 = \mathbf{G}^T \boldsymbol{\mathcal{H}}_N^4 \mathbf{G} \tag{5.12}$$

where elements of $\boldsymbol{\mathcal{H}}_{ig}^4$ are defined as,

$$h_{ig}^4(m,u,n,l) = \sum_{j=1}^{N_h}\sum_{k=1}^{N_h} h_N(j,m,u,k) g(j,n) g(k,l) \tag{5.13}$$

Comparing 5.13 and 5.10, we see that $h_{ig}(m,u) = h^4_{ig}(m,u,m,u)$, i.e., the 4-dimensional $\boldsymbol{\mathcal{H}}^4_{ig}$ is transformed into the 2-dimensional Hessian, $\mathbf{H}_{ig}$, by setting $n = m$ and $l = u$. To make this idea clear, consider a matrix, $\mathbf{Q}$, then $p(n) = q(n,n)$ is a vector, $\mathbf{p}$, of all diagonal elements of $\mathbf{Q}$. Similarly, the OIG Hessian $\mathbf{H}_{ig}$ is formed by a weighted combination of elements from $\boldsymbol{\mathcal{H}}^4_N$. From 5.11 and 5.12, we have successfully expressed a reduced size Hessian in a manner similar to 4.2.

5.4 Computational Burden

In this section, we describe the computational burden for using the training algorithms described so far. Let $N_u = (N + N_h + 1)$ denote the number of weights connected to each output. The total number of weights in the network is denoted as $N_w = M(N + N_h + 1) + N_h(N + 1)$. The number of multiplies required to solve for output weights using the Orthogonal Least Squares [49] is M_{ols}, which is described by

$$M_{ols} = N_u(N_u + 1)\left[M + \frac{1}{6}N_u(2N_u + 1) + \frac{3}{2}\right] \tag{5.14}$$

The numbers of multiplies required for training using BP, OWO-BP, OIG and LM are respectively given by

$$\begin{aligned} M_{bp} =& N_{it}\{N_v[MN_u + 2N_h(N + 1) \\ &+ M(N + 6N_h + 4)] + N_w\} \end{aligned} \tag{5.15}$$

$$\begin{aligned} M_{owo-bp} =& N_{it}\{N_v[2N_h(N + 2) + M(N_u + 1) \\ &+ \frac{N_u(N_u + 1)}{2} + M(N + 6N_h + 4)] \\ &+ M_{ols} + N_h(N + 1)\} \end{aligned} \tag{5.16}$$

$$\begin{aligned} M_{oig} =& M_{owo-bp} + N_{it}\{N_v[(N+1)(3MN_h + MN \\ & + 2(M+N) + 3) - M(N + 6N_h + 4) \\ & - N_h(N+1)] + (N+1)^3\} \end{aligned} \tag{5.17}$$

$$\begin{aligned} M_{lm} =& M_{bp} + N_{it}\{Nv[MN_u(N_u + 3N_h(N+1)) \\ & + 4N_h^2(N+1)^2] + N_w^3 + N_w^2\} \end{aligned} \tag{5.18}$$

where N_{it} is the number of training iterations.

Note that M_{oig} consists of M_{owo-bp} plus the required multiplies for calculating optimal input gains. Similarly, M_{lm} consists of M_{bp} plus the required multiplies for calculating and inverting the Hessian matrix. The above equations for the number of multiplies per iteration will be used to evaluate performance in the following section.

5.5 Results

Here we present the results for the OWO-BP algorithm, modified using the optimal input gain method. We compare its performance with BP, OWO-BP and LM, where optimal learning factors (OLFs) were used in the latter three algorithms. In BP and LM, all weights are varied in each iteration. In OWO-BP, we alternately use BP for the input weights (with the OLF) and solve linear equations for the output weights.

For a given network, we obtain the training error and the number of multiplies required for each training iteration. We also obtain the validation error for a fully trained network. This information is used to subsequently generate the plots and compare performances.

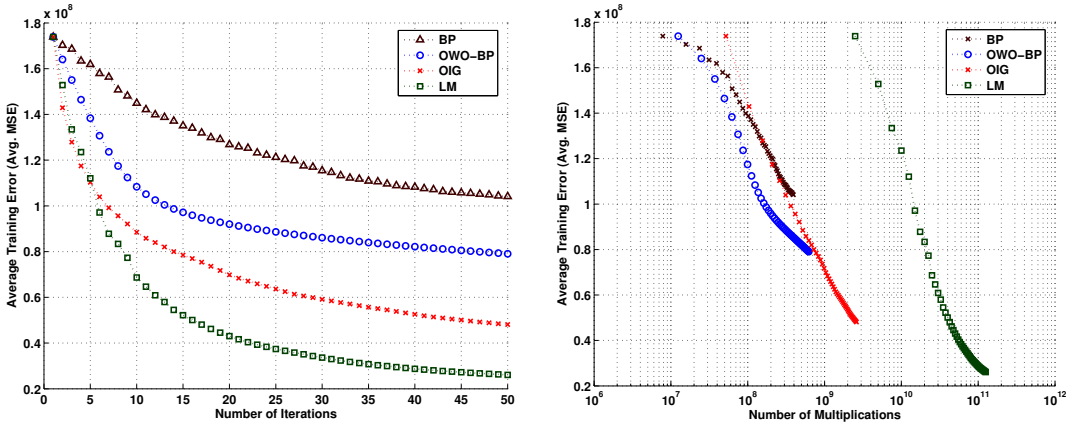
We use the *k-fold* cross-validation procedure to obtain the training and validation errors. Given a data set, we split the set into k non-overlapping parts of equal

size, and use $(k-1)$ parts for training and the remaining one part for validation. The procedure is repeated till we have exhausted all k combinations ($k = 10$ for our simulations).

All the data sets used for simulation are publicly available. In all data sets, the inputs have been normalized to be zero-mean and unit variance. For a detailed description of each data set, refer to appendix A.

5.5.1 Prognostics Data Set

This data file is available on the Image Processing and Neural Networks Lab repository [57]. It consists of parameters that are available in the Bell Helicopter health usage monitoring system (HUMS), which performs flight load synthesis, which is a form of prognostics [58].



(a) (b)

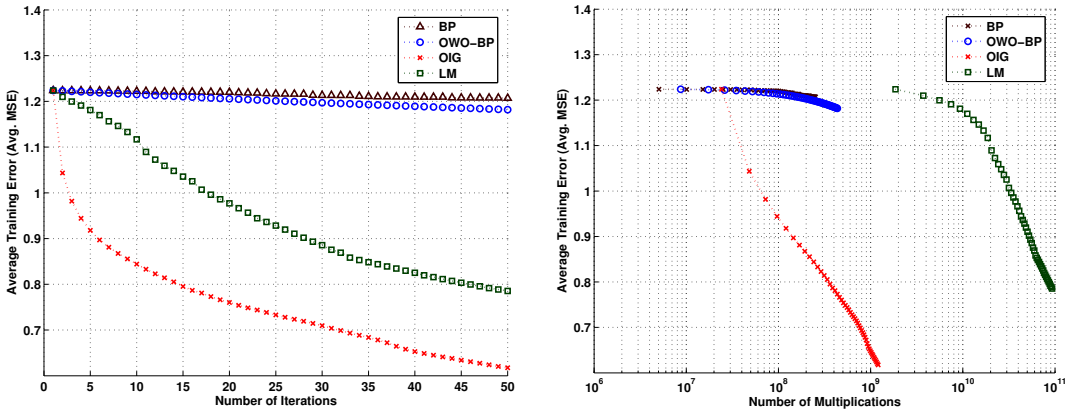
Figure 5.1. Prognostics Data: average error vs. (a) iterations and (b) multiplies.

For this data file, which is called *F17*, we trained an MLP having 15 hidden units. In Fig. 5.1-a, the average mean square error (MSE) for training from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 5.1-

b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies (shown on a log_{10} scale). From Fig. 5.1-a and Fig. 5.1-b, the proposed optimal input gain algorithm converges faster than BP or OWO-BP, and it is much faster than LM.

5.5.2 Remote Sensing Data Set

This data file is available on the Image Processing and Neural Networks Lab repository [57]. It consists of 16 inputs and 3 outputs and represents the training set for inversion of surface permittivity, the normalized surface rms roughness, and the surface correlation length found in back scattering models from randomly rough dielectric surfaces [59].



(a) (b)

Figure 5.2. Remote Sensing Data: average error vs. (a) iterations and (b) multiplies.

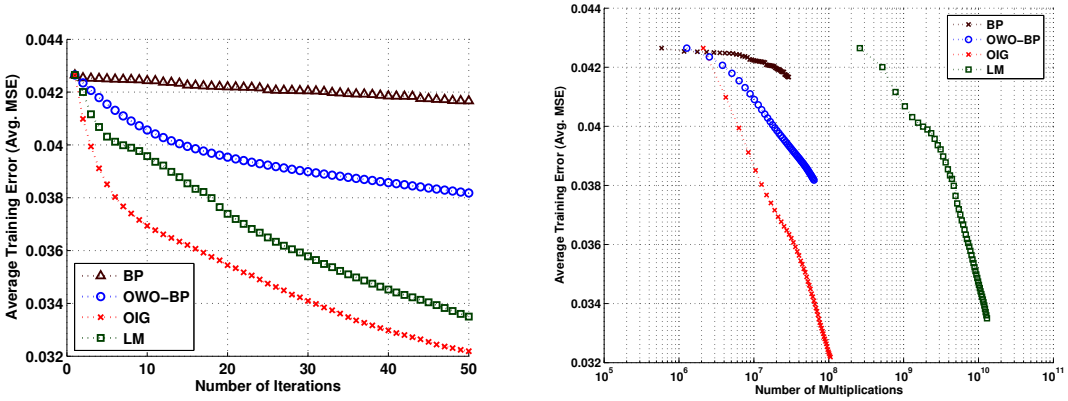
For this data file, which is called *Single2*, we trained an MLP having 15 hidden units. In Fig. 5.2-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 5.2-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.

From the plots, the optimal input gain algorithm again converges faster than BP and OWO-BP, and it has smaller training error. In this example, it trains better than LM, with almost two orders of magnitude fewer multiplies.

5.5.3 Federal Reserve Economic Data Set

This file contains some economic data for the USA from 01/04/1980 to 02/04/2000 on a weekly basis. From the given features, the goal is to predict the 1-Month CD Rate [60].

It has 15 inputs and one output per pattern, with a total of 1049 patterns. For this data file, which is called *TR*, we trained an MLP having 15 hidden units. In Fig. 5.3-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 5.3-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.



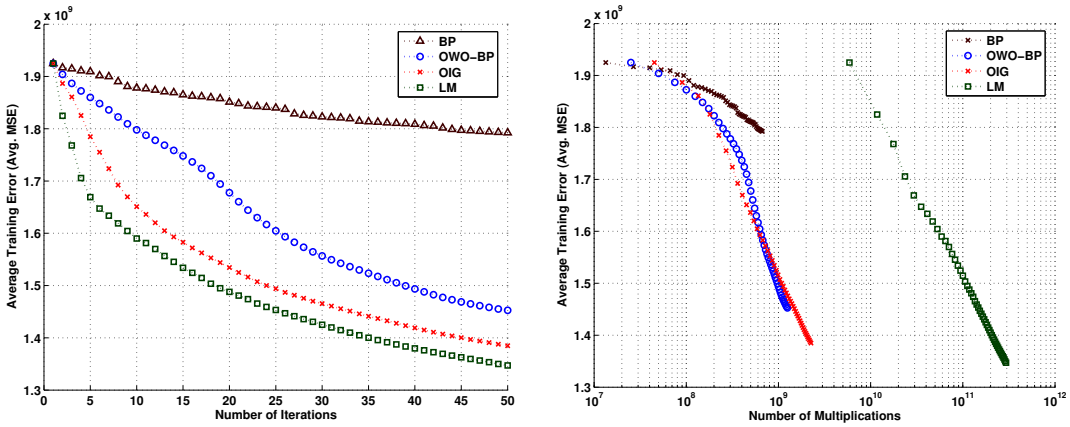
(a) (b)

Figure 5.3. Economic Data: Average error vs. (a) iterations and (b) multiplies.

From Fig. 5.3-a and Fig. 5.3-b, the optimal input gain algorithm has a training error less than that of LM, with far fewer multiplies per iteration.

5.5.4 Housing Data Set

This data file is available on the DELVE data set repository [61].



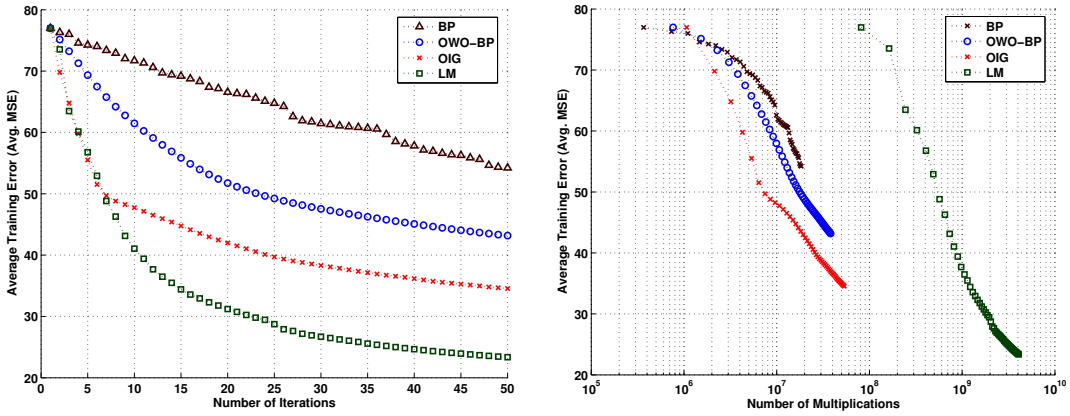
(a) (b)

Figure 5.4. Housing Data: Average error vs. (a) iterations and (b) multiplies.

The training data consists of 16 inputs and 1 output per pattern, with a total of 22,784 patterns. For this data file, we trained an MLP having 15 hidden units. In Fig. 5.4-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 5.4-b, the MSE from 10-fold validation is plotted versus the required number of multiplies. From the plots, the optimal input gain algorithm has a training error close to that of LM, with far fewer multiplies per iteration.

5.5.5 Concrete Compressive Strength Data Set

This data file is available on the UCI Machine Learning Repository [62]. The data set consists of 8 inputs and one output per pattern, with a total of 1030 patterns. For this data file, we trained an MLP having 15 hidden units. In Fig. 5.5-a, the average training MSE from 10-fold validation is plotted versus the number of iterations



(a) (b)

Figure 5.5. Concrete Data: Average error vs. (a) iterations and (b) multiplies.

for each algorithm. In Fig. 5.5-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies. From Figs. 5.5-a and 5.5-b, the optimal input gain algorithm has a training error close to that of LM, with far fewer multiplies per iteration.

Table 5.1 compares the average training and validation errors of the proposed OIG algorithm with BP, OWO-BP and LM on different data sets. For each data set, the training and validation errors again come from 10-fold cross validation.

We can see that the proposed OIG algorithm sometimes has a performance comparable to or better than the popular LM algorithm.

5.6 Limitations on OIG

As mentioned before, the OIG algorithm developed in this chapter has been used to modify OWO-BP type training. When there are no dependent inputs, the OIG algorithm finds the optimal gain coefficients for each input that reduces the overall mean squared training error. Singular value decomposition (SVD) or similar

Table 5.1. Average 10-fold training and validation error

Data set		BP	OWO-BP	OIG	LM
F-17	E_{trn}	1.0410E8	7.9042E7	4.8109E7	2.6089E7
	E_{val}	1.0748E8	8.2064E7	5.1440E7	2.9399E7
Single2	E_{trn}	1.207082	1.181896	0.617487	0.785390
	E_{val}	1.296914	1.277067	0.777170	0.863154
Treasury	E_{trn}	0.041667	0.038182	0.032190	0.033500
	E_{val}	0.047801	0.044630	0.041435	0.040850
Housing	E_{trn}	1.7923E9	1.4526E9	1.3849E9	1.3472E9
	E_{val}	1.7974E9	1.4976E9	1.4386E9	1.3917E9
Concrete	E_{trn}	54.2286	43.1730	34.5438	23.3564
	E_{val}	60.8905	50.6084	41.5764	31.9576

algorithm can be used for inverting the OIG Hessian. In this section we analyze the performance of OIG in the presence of linearly dependent inputs.

Any time there are dependent or identical inputs present, it is easy to show that the input autocorrelation matrix, $\mathbf{R}$ and the gradient matrix $\mathbf{G}$ have dependent or identical columns. However, the key to analyzing the effect of identical or dependent inputs on OIG's performance lies in examining the structure of the Hessian for each case. If indeed the input dependence is reflected in the Hessian being singular (having dependent rows and columns), then the SVD should pick up the dependency. This would force the gain coefficient corresponding to the dependent input to be zero, hence eliminating them during training. Next, we analyze the structure of Hessian for identical inputs, followed by linearly dependent inputs.

5.6.1 Identical Inputs

Consider the case of identical inputs. Assume that input $(N+2)$ is identical to input $(N+1)$. In this case the corresponding gradient elements are also identical, i.e.,

$$g(k, N+2) = b \cdot g(k, N+1) \quad \forall \quad k \in (1, N_h)$$

Now, the element of the Hessian can be written as,

$$\begin{aligned} h_{ig}(m,u) &\equiv \frac{\partial^2 E}{\partial a(m) \partial a(u)} \\ &= \frac{2}{N_v} \sum_{p=1}^{N_v} x_p(m) x_p(u) \sum_{i=1}^{M} v(i,m) v(i,u) \end{aligned}$$

where,$v(i,m)$ is described in equation 5.7. For the case of identical input, we can write,

$$v(i, N+2) = b \cdot v(i, N+1)$$

and the corresponding element in the Hessian would be

$$\begin{aligned} h_{ig}(N+2,u) &= b^2 \frac{2}{N_v} \sum_{p=1}^{N_v} x_p(N+1) x_p(u) \sum_{i=1}^{M} v(i,N+1) v(i,u) \\ &= h_{ig}(N+1,u) \end{aligned}$$

From the above analysis and based on the fact that the Hessian is symmetric, the $(N+2)^{th}$ row and column of the Hessian would be identical to the $(N+1)^{th}$ row and column. Clearly, the Hessian is singular. If the SVD is used for inversion, then the corresponding row and column of the inverted Hessian would be zero. This in turn sets the gain coefficient $a(N+2)$ to be zero, effectively eliminating any contribution of the identical input $x_p(N+2)$ during training.

We have shown how identical inputs reflect in the Hessian having identical rows and columns, making it easy to detect and eliminate the identical inputs.

5.6.2 Dependent Inputs

We have seen for the case of identical inputs that the Hessian is singular. Next we examine the case of more general dependent inputs. We can model the $(N+2)^{th}$ dependent input as,

$$x_p(N+2) = \sum_{j=1}^{N+1} b(j) x_p(j)$$

and the corresponding elements of the gradient matrix $\mathbf{G}$ and $\mathbf{V}$ are given by,

$$g(k, N+2) = \sum_{j=1}^{N+1} g(k,j) b(j)$$

$$v(i, N+2) = \sum_{j=1}^{N+1} v(i,j) b(j)$$

Similarly, the Hessian element is given by,

$$\begin{aligned} h_{ig}(N+2, u) &= \frac{2}{N_v} \sum_{p=1}^{N_v} x_p(N+2) x_p(u) \sum_{i=1}^{M} v(i, N+2) v(i,u) \\ &= \sum_{n=1}^{N+1} b(n) \sum_{j=1}^{N+1} b(j) \frac{2}{N_v} \sum_{p=1}^{N_v} x_p(n) x_p(u) \sum_{i=1}^{M} v(i,j) v(i,u) \\ &\neq \sum_{j=1}^{N+1} h_{ig}(i,j) b(j) \end{aligned} \tag{5.19}$$

Clearly, the Hessian is nonsingular in the case of linearly dependent input as the corresponding row and column of the Hessian will not be a linear sum of the other rows and columns. This means that the Hessian can still be inverted and the linearly dependent input, $x_p(N+2)$, will have a non-zero gain $a(N+2)$. This nonsingular *impure* Hessian cannot be used to detect and eliminate the linearly dependent inputs. This could cause OIG to have a suboptimal performance and possibly poor convergence. One strategy to overcome this limitation of OIG is to use the autocorrelation matrix for an early detection and elimination of linearly dependent inputs. This *in-place preprocessing technique* would be done only once before training begins. The

OIG thus trains only on independent inputs. Another strategy would be to use the negative Jacobian matrix for input weights, **G**. This would have to be done for every training iteration as the elements of **G** change for every iteration. The end result is that the Hessian is free from dependent rows and columns and the OIG algorithm would train on linearly independent inputs only.

5.7 Discussion

We have derived a second order method for simultaneously optimizing input gains and the OLF. The method has been successfully demonstrated on five data sets. Results show that this approach performs much better than two common first order algorithms with comparable complexity, namely BP and OWO-BP. It comes close to LM in terms of the training error, but with orders of magnitude less computation. This is evident in all of the plots of training error versus the required number of multiplies and also from the expressions for the numbers of multiplies.

Although LM works very well in practice it has a high computational burden and is sub-optimal in the way it handles the 'scaling' factor, λ. OIG on the other hand uses a Newton type update and combines the optimal learning factor, leaving little room for heuristics.

We discussed the limitations of the OIG algorithm. We have shown that Hessian used in calculating the input gain coefficients can be non-singular in the presence of linearly dependent inputs, leading to poor training. The Hessian is hence not usable to detect or eliminate dependent inputs. We have suggested two strategies for in-place detection and elimination of dependent inputs, either of which could be implemented to improve OIG's performance in the presence of dependent inputs.

CHAPTER 6

A MULTIPLE OPTIMAL LEARNING FACTOR ALGORITHM

This chapter presents a new learning algorithm called the multiple optimal learning factor (MOLF) algorithm, that calculates an optimal learning factor for every hidden unit, in order to increase the speed of learning and overall convergence.

The multiple optimal learning factor is a batch training algorithm for feed-forward networks which uses Newton's method to estimate a separate optimal learning factor for each hidden unit's input weights. Linear equations are then solved for the network's output weights. The primary motivation is discussed in section 6.1. The new algorithm is described in section 6.2. Elements of the new method's Hessian matrix are shown to be weighted sums of elements from the total network's Hessian. Results and discussion are presented in sections 6.6 and 6.8. In several examples, the new method performs as well as or better than Levenberg-Marquardt.

6.1 Motivation For Multiple Learning Factors

In the past, researchers have used multiple learning rates and/or momentum terms in order to speed up the learning process [43], [45]. Unfortunately, these methods are mostly heuristic, and their performance relies on the settings of some user chosen parameters. Also, using standard gradients, makes them slow to converge. The Newton's method can be viewed as a second order method to assign a learning rate to every weight in the network.

6.1.1 First Order Algorithm with Second Order Learning Factor

As suggested earlier, we can alternately update input weights and solve linear equations for the output weights located in arrays $\mathbf{W}_{oh}$ and $\mathbf{W}_{oi}$. Instead of using Newton's method or even LM to modify the input weights, we can use BP. Let $\mathbf{W}$ again denote the N_h by $(N+1)$ input weight array. In every iteration, $\mathbf{W}$ can be updated as

$$\mathbf{W} \leftarrow \mathbf{W} + z \cdot \mathbf{G} \tag{6.1}$$

where z is the learning factor or the step length and $\mathbf{G}$ is the negative input weight Jacobian. This first order training algorithm is called output weight optimization-backpropagation (OWO-BP) [29].

Using a Taylor's series for E, a non-heuristic, optimal learning factor (OLF) can be derived [63] as,

$$z = \frac{-\partial E/\partial z}{\partial^2 E/\partial z^2} \tag{6.2}$$

where the numerator and denominator derivatives are evaluated at $z = 0$.

The expressions for the first and second derivatives of the error with respect to the OLF is found using equation (2.5) as,

$$\frac{\partial E}{\partial z} = \frac{-2}{N_v} \sum_{p=1}^{N_v} \sum_{m=1}^{M} (t_p(m) - y_p(m)) \cdot \frac{\partial y_p(m)}{\partial z} \tag{6.3}$$

and

$$\frac{\partial^2 E}{\partial z^2} = \frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{m=1}^{M} \left[\left(\frac{\partial y_p(m)}{\partial z} \right)^2 \right] \tag{6.4}$$

Equation (6.4) can be expanded as

$$\begin{aligned}\frac{\partial^2 E}{\partial z^2} &= \sum_{k=1}^{N_h}\sum_{j=1}^{N_h}\sum_{n=1}^{N+1}\sum_{i=1}^{N+1} g(k,n)\left[\frac{2}{N_v}v(k,j)\sum_{p=1}^{N_v} x_p(i)x_p(n)\cdot O'_p(k)O'_p(j)\right] g(j,i) \\ &= \sum_{k=1}^{N_h}\sum_{j=1}^{N_h}\sum_{n=1}^{N+1} g(k,n)\sum_{i=1}^{N+1}\frac{\partial^2 E}{\partial w(k,n)\partial w(j,i)} g(j,i) \qquad (6.5) \\ &= \sum_{k=1}^{N_h}\sum_{j=1}^{N_h} \mathbf{g}_k^T \mathbf{H}_R^{k,j} \mathbf{g}_j\end{aligned}$$

where column vector $\mathbf{g}_k$ contains $\mathbf{G}$ elements $g(k,n)$ for all values of n, and where the $(N+1)$ by $(N+1)$ matrix $\mathbf{H}_R^{k,j}$ contains elements of $\mathbf{H}_R$ for weights connected to the j^{th} and k^{th} hidden units. Comparing with equation (2.15), it is clear that the optimal learning factor can be computed as a weighted sum of the elements in the Hessian $\mathbf{H}_R$. This relation shows that (i) the OLF can be obtained from elements of the Hessian, (ii) the Hessian contains useful information even when it is singular, and (iii) a smaller non-singular Hessian ($\partial^2 E/\partial z^2$) can be constructed using $\mathbf{H}_R$. Note the similarity between equations 4.2 and 6.5

There should be algorithms, in addition to optimal input gain, that produce nonsingular N_1 by N_1 Hessian matrices, where $1 \leq N_1 \leq N_{iw}$. Such algorithms could average rows and columns of $\mathbf{H}_R$ or even eliminate dependent rows and columns.

6.2 Multiple Optimal Learning Factor Algorithm

Here, each hidden unit in section 6.1.1 is given its own OLF. The result is called the multiple optimal learning factor (MOLF) training method.

6.2.1 Derivation of Multiple Optimal Learning Factors

Assume that an MLP is being trained using OWO-BP. However, also assume that a separate OLF z_k is being used to update each hidden unit's input weights,

$w(k,n)$, where $1 \leq k \leq N_h$ and $1 \leq n \leq (N+1)$. The error function being minimized with respect to the z_k is given by equation (2.5). The predicted output $y_p(m)$ is given by,

$$y_p(m) = \sum_{n=1}^{N+1} w_{oi}(m,n)x_p(n) + \sum_{k=1}^{N_h} w_{oh}(m,k) f\left(\sum_{n=1}^{N+1}(w(k,n) + z_k \cdot g(k,n))x_p(n)\right)$$

where, $g(k,n)$ is an element of the negative Jacobian matrix $\mathbf{G}$. The first partial of E with respect to z_j is

$$g_{molf}(j) \equiv \frac{\partial E}{\partial z_j} = \frac{-2}{N_v}\sum_{p=1}^{N_v}\sum_{m=1}^{M}\left[t'_p(m) - \sum_{k=1}^{N_h} w_{oh}(m,k)O_p(z_k)\right] \cdot w_{oh}(m,j)O'_p(j)\triangle n_p(j) \tag{6.6}$$

where

$$t'_p(m) = t_p(m) - \sum_{n=1}^{N+1} w_{oi}(m,n)x_p(n)$$

$$\triangle n_p(j) = \sum_{n=1}^{N+1} x_p(n) \cdot g(j,n)$$

$$O_p(z_k) = f\left(\sum_{n=1}^{N+1}(w(k,n) + z \cdot g(k,n))x_p(n)\right)$$

Using Gauss-Newton updates, the second partial derivative elements of the Hessian $\mathbf{H}_{molf}$ are

$$\frac{\partial^2 E}{\partial z_l \partial z_j} = \frac{2}{N_v}\sum_{m=1}^{M} w_{oh}(m,l)w_{oh}(m,j)\sum_{p=1}^{N_v} O'_p(l)O'_p(j)\triangle n_p(l)\triangle n_p(j) \tag{6.7}$$

6.2.2 MOLF Implementation

The Gauss-Newton update guarantees that $\mathbf{H}_{molf}$ is non-negative definite. Given the negative gradient vector, $\mathbf{g}_{molf} = [-\partial E/\partial z_1, -\partial E/\partial z_2 \ldots, -\partial E/\partial z_{N_h}]^T$ and the Hessian $\mathbf{H}_{molf}$, we minimize E with respect to the vector $\mathbf{z}$ using Newton's method. In each iteration of the training algorithm, the steps are as follows:

(i) Calculate the negative input weight Jacobian $\mathbf{G}$ using BP.

(ii) Calculate the MOLF z_k using Newton's method and update the input weights as

$$w(k,n) \leftarrow w(k,n) + z_k \cdot g(k,n) \tag{6.8}$$

(iii) Solve linear equations for all output weights.

Here, the MOLF procedure has been inserted into the OWO-BP algorithm. It can be inserted into other algorithms as well, including standard BP. An obvious limitation of the MOLF is expressed in the following lemma.

Lemma 6-1 If $N_h = 1$, then MOLF training as described in this subsection is identical to OWO-BP with optimal learning factor.

6.3 MOLF Analyses

If $\mathbf{H}_{molf}$ and $\mathbf{g}_{molf}$ are the Hessian and gradient, respectively, of the error with respect to $\mathbf{z}$, then the multiple optimal learning factors are computed as,

$$\Delta \mathbf{z} = \mathbf{H}_{molf}^{-1} \cdot \mathbf{g}_{molf} \tag{6.9}$$

Re-writing equation (6.7), for the element of the Hessian as,

$$\frac{\partial^2 E}{\partial z_l \partial z_j} = \sum_{i=1}^{N+1} \sum_{n=1}^{N+1} \left[\frac{2}{N_v} \sum_{p=1}^{N_v} x_p(i) x_p(n) O'_p(l) O'_p(j) \sum_{m=1}^{M} w_{oh}(m,l) w_{oh}(m,j) \right] g(l,i) \cdot g(j,n)$$

The term within the square brackets is nothing but an element of the reduced Hessian, $\mathbf{H}_R$ from Gauss-Newton method for updating input weights (equation (2.15)). Hence,

$$\frac{\partial^2 E}{\partial z_l \partial z_j} = \sum_{i=1}^{N+1} \sum_{n=1}^{N+1} \left[\frac{\partial^2 E}{\partial w(l,i) \partial w(j,n)} \right] g(l,i) \cdot g(j,n)$$

Lemma 6-2: For fixed (l,j), $h_{molf}(l,j)$ can be expressed in vector notation as,

$$h_{molf}(l,j) \equiv \frac{\partial^2 E}{\partial z_l \partial z_j} = \sum_{i=1}^{N+1} g_l(i) \sum_{n=1}^{N+1} h_R^{l,j}(i,n) \cdot g_j(n) = \mathbf{g}_l^T \mathbf{H}_R^{l,j} \mathbf{g}_j \tag{6.10}$$

where column vector $\mathbf{g}_l$ contains $\mathbf{G}$ elements $g(l,n)$ for all values of n, and the $(N+1)$ by $(N+1)$ matrix $\mathbf{H}_R^{l,j}$ contains elements of $\mathbf{H}_R$ for weights connected to the l^{th} and j^{th} hidden units. Each element of the MOLF Hessian combines the information from $(N+1)$ rows and columns of the reduced Hessian, $\mathbf{H}_R^{l,j}$. This can be seen as compressing the original Hessian of size N_{iw} by N_{iw} to simply N_h by N_h. The MOLF effectively *encodes* the information from the Hessian into a smaller matrix. This makes MOLF less sensitive to input conditions. Note the similarities between (6.5) and (6.10).

From equation 6.10, the reduced Hessian, $\mathbf{H}_R$ uses four indices (l,i,j,n) and can be viewed as a 4-dimensional array, represented by $\boldsymbol{\mathcal{H}}_R^4 \in \mathbb{R}^{(N+1)\times N_h \times N_h \times (N+1)}$. Using this representation, we can express the 4-dimensional MOLF Hessian as

$$\boldsymbol{\mathcal{H}}_{molf}^4 = \mathbf{G}\boldsymbol{\mathcal{H}}_R^4\mathbf{G}^T \tag{6.11}$$

In 6.11, we express an element of $\boldsymbol{\mathcal{H}}_{molf}^4$ as

$$h_{molf}^4(l,j,m,u) = \sum_{i=1}^{N+1}\sum_{n=1}^{N+1} h_R(i,l,j,n)g(m,i)g(u,n) \tag{6.12}$$

Comparing 6.12 and 6.10, we see that $h_{molf}(l,j) = h_{molf}^4(l,j,l,j)$, i.e., the 4-dimensional $\boldsymbol{\mathcal{H}}_{molf}^4$ is transformed into the 2-dimensional $\mathbf{H}_{molf}$, by setting $m=l$ and $u=j$. To make this idea clear, consider a matrix, $\mathbf{Q}$, then $p(n) = q(n,n)$ is a vector, $\mathbf{p}$, of all diagonal elements of $\mathbf{Q}$. Similarly, the MOLF Hessian $\mathbf{H}_{molf}$ is formed by a weighted combination of elements of $\boldsymbol{\mathcal{H}}_R^4$. From 6.10 and 6.11, we have again succeeded in expressing a reduced size Hessian in a manner similar to our goal in 4.2.

As a side note, comparing the matrix triple product expressions 6.11 and 5.12, we can say that $\boldsymbol{\mathcal{H}}_R^4$ contains the same elements as $\boldsymbol{\mathcal{H}}_N^4$, but ordered differently. This is due to the fact that the gradient matrix $\mathbf{G}$ in both expressions is exactly the same, but it appears differently in the triple products.

6.4 Effect of Dependent Inputs

Going back to the case of dependent input, let the $(N+2)^{th}$ input be dependent on some others, as modeled in (3.2.1.1). Let $\bar{\mathbf{H}}_{molf}$ be the Hessian, when the extra dependent input $x_p(N+2)$ is included.

$$\bar{h}_{molf}(l,j) = h_{molf}(l,j) + \frac{2}{N_v} u(l,j) \sum_{p=1}^{N_v} x_p(N+2) O'_p(l) O'_p(j) \Big[g(l,N+2) \sum_{n=1}^{N+1} x_p(n) g(j,n) + g(j,N+2) \sum_{i=1}^{N+1} x_p(i) g(l,i) + x_p(N+2) g(l,N+2) g(j,N+2) \Big] \quad (6.13)$$

Lemma 6-3: Linearly dependent inputs, when added to the network, do not force $\bar{\mathbf{H}}_{molf}$ to be singular. As seen in (3.2.1.1) and (6.13), each $\bar{h}_{molf}(m,j)$ simply gains some first and second degree terms in the variables $b(n)$.

Assume that some hidden unit activations are linearly dependent upon others, as in equation (3.20). Further assume that OLS is used to solve for output weights. We have the following lemma.

Lemma 6-4: For each dependent hidden unit, the corresponding row and column of $\mathbf{H}_{molf}$ is zero-valued. This follows from (6.7). The zero-valued rows and columns in $\mathbf{H}_{molf}$ need not cause any difficulties, if (6.9) is rewritten as $\mathbf{H}_{molf} \cdot \mathbf{z} = \mathbf{g}_{molf}$ [64] and solved using OLS. This is due to the fact that the zero rows or columns in the Hessian will result in the corresponding orthonormal coefficients to be zero and when the coefficients are mapped from the orthonormal space, back to the original weight space, the learning factor corresponding to the dependent hidden unit will be zero.

6.5 Computational Cost

The proposed MOLF algorithm involves inverting a Hessian. However, compared to Newton's method or LM, the size of the Hessian is much smaller. Updating

input weights using Newton's method or LM, requires a Hessian with $N_h(N+1)$ rows and columns, whereas the Hessian used in the proposed MOLF has only N_h rows and columns.

The total number of weights in the network is denoted as $N_w = M(N_u) + N_h(N+1)$. The number of multiplies required to solve for output weights using the Orthogonal Least Squares [49] is given by

$$M_{ols} = N_u(N_u+1)\left[M + \frac{1}{6}N_u(2N_u+1) + \frac{3}{2}\right] \tag{6.14}$$

The numbers of multiplies per training iteration for OWO-BP, LM and MOLF are given below

$$\begin{aligned} M_{owo-bp} =& N_v[2N_h(N+2) + M(N_u+1) + \frac{N_u(N_u+1)}{2} + M(N+6N_h+4)] \\ &+ M_{ols} + N_h(N+1) \end{aligned} \tag{6.15}$$

$$\begin{aligned} M_{lm} =& N_v[MN_u + 2N_h(N+1) + M(N+6N_h+4) + MN_u(N_u+3N_h(N+1)) \\ &+ 4N_h^2(N+1)^2] + N_w^3 + N_w^2 \end{aligned} \tag{6.16}$$

$$M_{molf} = M_{owo-bp} + N_v[N_h(N+4) - M(N+6N-h+4)] + (N_h)^3 \tag{6.17}$$

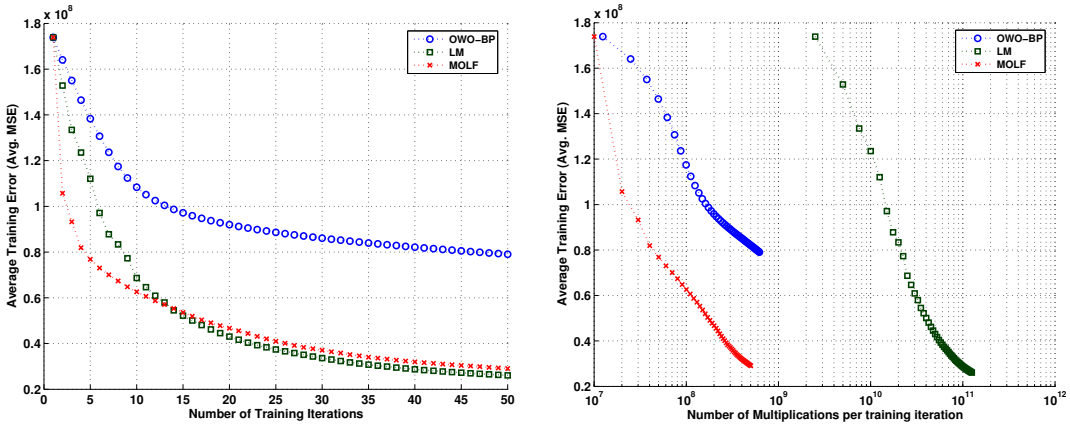
Note that M_{molf} consists of M_{owo-bp} plus the required multiplies for calculating optimal learning factors.

6.6 Results

In this section, the performance of the OWO-BP algorithm, modified using the MOLF algorithm is compared with regular OWO-BP and LM (both using a single OLF). LM updates all the weights in each iteration. OWO-BP alternately uses BP for input weights and solves linear equations for output weights.

For each algorithm, the mean square error in every training iteration, the approximate number of multiplications required per training iteration and the overall validation error of a fully trained network is stored. This information is used to subsequently generate the plots and compare performances. The *10-fold* cross-validation procedure is used to obtain the training and validation errors. Given a data set, it is split into K non-overlapping parts of equal size. $(K-1)$ parts is used for training and the remaining one part for validation. The procedure is repeated till all K combinations are exhausted ($K = 10$ for all simulations). All the data sets used for simulation are publicly available.

6.6.1 Prognostics Data Set



(a) (b)

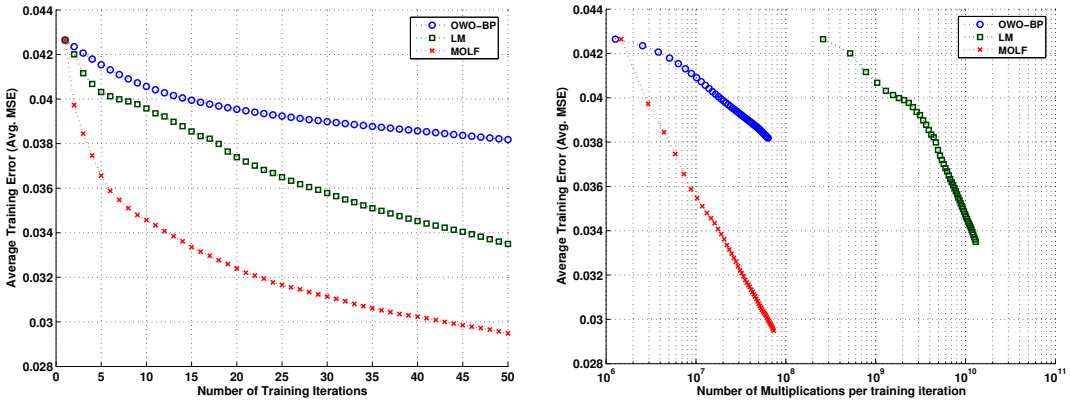
Figure 6.1. Prognostics Data: Average error vs. (a) iterations and (b) multiplies.

We trained an MLP having 15 hidden units. In Fig. 6.1-a, the average mean square error (MSE) for training from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 6.1-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies (shown on a log_{10} scale).

From Fig. 6.1-a and Fig. 6.1-b, the proposed multiple optimal learning factor algorithm converges faster than OWO-BP, and it is much faster than LM. The performance of MOLF is similar to that of LM.

6.6.2 Federal Reserve Economic Data Set

For this data file, we trained an MLP having 15 hidden units. In Fig. 6.2-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 6.2-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies. From Fig. 6.2-a and Fig. 6.2-b, the MOLF algorithm has a training error less than that of LM, with far fewer multiplies per iteration.

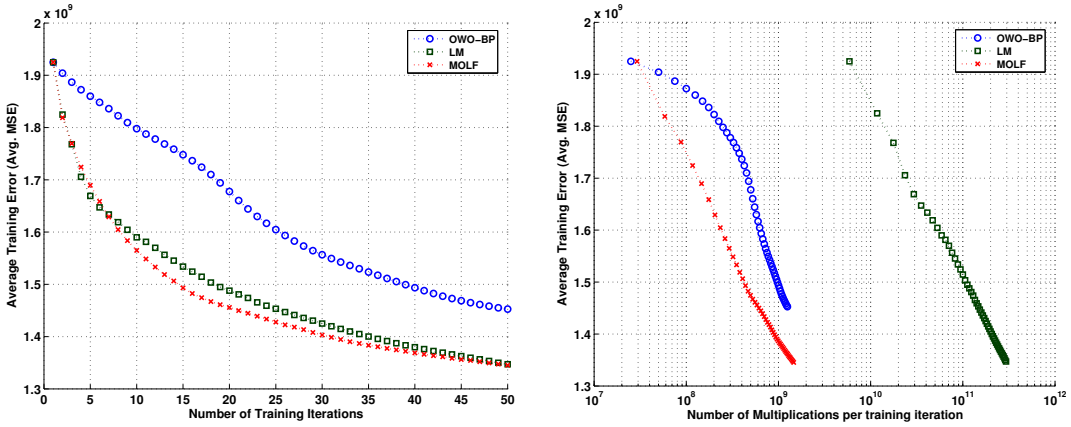


(a) (b)

Figure 6.2. Federal reserve data: average error vs. (a) iterations and (b) multiplies.

6.6.3 Housing Data Set

We trained an MLP having 15 hidden units. In Fig. 6.3-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 6.3-b, the MSE from 10-fold validation is plotted versus the required



(a) (b)

Figure 6.3. Housing data: average error vs. (a) iterations and (b) multiplies.

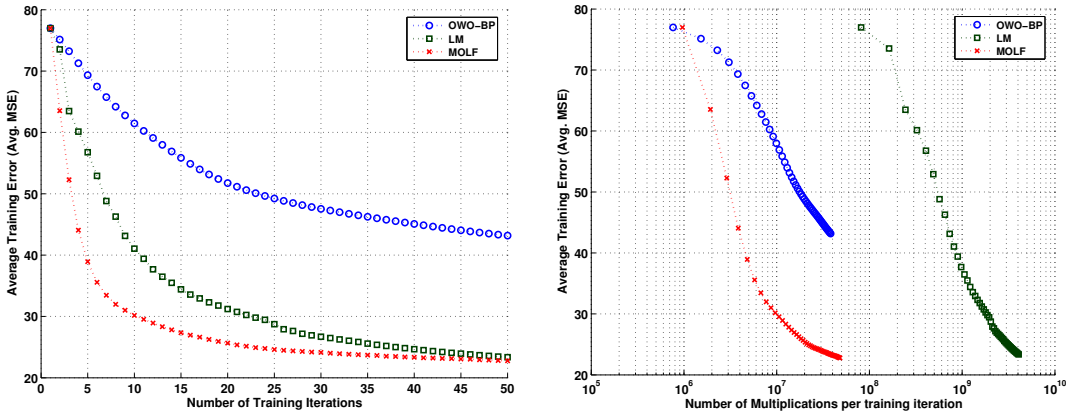
number of multiplies. From Fig. 6.3-a and Fig. 6.3-b, the MOLF algorithm has a training error close to that of LM, with far fewer multiplies per iteration.

6.6.4 Concrete Compressive Strength Data Set

We trained an MLP having 15 hidden units. In Fig. 6.4-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 6.4-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies. From the figures, the MOLF algorithm has a training error close to that of LM, with far fewer multiplies per iteration.

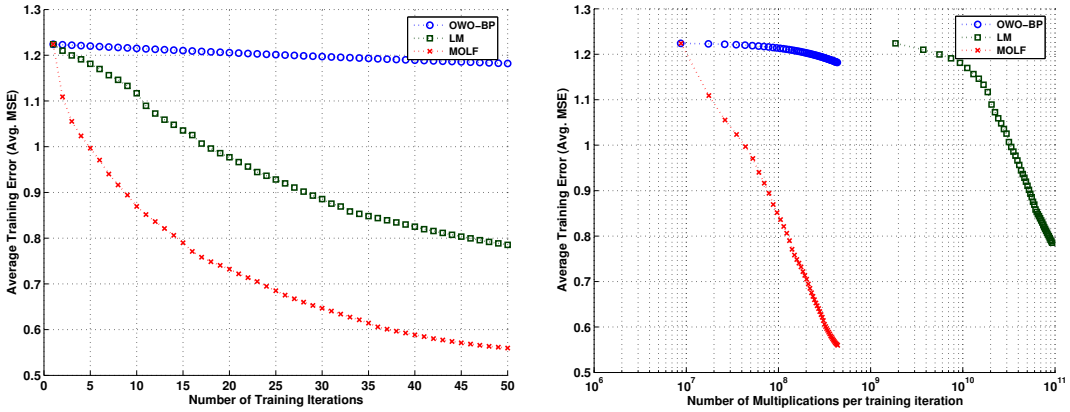
6.6.5 Remote Sensing Data Set

We trained an MLP having 15 hidden units. In Fig. 6.5-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 6.5-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.



(a) (b)

Figure 6.4. Concrete data: average error vs. (a) iterations and (b) multiplies.



(a) (b)

Figure 6.5. Remote sensing data: average error vs. (a) iterations and (b) multiplies.

From Fig. 6.5-a and Fig. 6.5-b, the MOLF algorithm again converges faster than OWO-BP, and it has smaller training error. In this example, it trains as well as LM, with almost two orders of magnitude fewer multiplies.

Table 6.1 compares the average training and validation errors of the proposed MOLF algorithm with BP, OWO-BP and LM on different data sets. For each data set, the training and validation errors again come from 10-fold cross validation. We can see that the proposed MOLF algorithm often has a performance comparable to or better than the LM.

Table 6.1. Average 10-fold training and validation error

Data set	MSE	BP	OWO-BP	MOLF	LM
Prognostics	E_{trn}	1.0410E8	7.9042E7	2.9057E7	2.6089E7
	E_{val}	1.0748E8	8.2064E7	3.2475E7	2.9399E7
Federal Reserve	E_{trn}	0.041667	0.038182	0.029490	0.033500
	E_{val}	0.047801	0.044630	0.039010	0.040850
Housing	E_{trn}	1.7923E9	1.4526E9	1.3453E9	1.3472E9
	E_{val}	1.7974E9	1.4976E9	1.4126E9	1.3917E9
Concrete	E_{trn}	54.2286	43.1730	22.7746	23.3564
	E_{val}	60.8905	50.6084	30.7310	31.9576
Remote Sensing	E_{trn}	1.207082	1.181896	0.559648	0.785390
	E_{val}	1.296914	1.277067	0.665496	0.863154

6.7 Limitations of MOLF Algorithm

The effectiveness of the proposed MOLF applied to OWO-BP is evident in the results. It performs as well as LM with far fewer multiplies per iteration.

A constant theme through out this dissertation is the analyses of linear dependence on training. In this section, we analyze the performance of MOLF applied to OWO-BP, in the presence of linear dependence.

6.7.1 Dependence in the Hidden Layer

The weights in the network are updated during every training iteration and it is quite possible that this could cause some hidden units to be dependent. The dependence could manifest in hidden units being identical or a linear combination of other hidden unit outputs or simply a weighted sum of the inputs (if the hidden unit is *saturated*).

Consider one of the hidden units to be dependent. The autocorrelation matrix will undoubtedly be singular and since we are using OLS to solve for output weights, all the weights connecting the dependent hidden unit to the outputs will be forced to zero. This will ensure that the dependent hidden unit does not contribute to the output. To see if this has any impact on learning in MOLF we can look at the expression for the gradient and Hessian, given by 6.6 and 6.7 respectively. Both equations have a sums of product terms and since OWO sets the output weights for the dependent hidden unit to be zero, this will also set the corresponding gradient and Hessian elements to be zero. In general, any dependence in the hidden layer will cause the corresponding learning factor to be zero and will not affect the performance of MOLF.

Next we will look at the how a dependent input would affect the performance of MOLF.

6.7.2 Dependence in the Input Layer

A linearly dependent input can be modeled as

$$x_p(N+2) = \sum_{n=1}^{N+1} b(n) x_p(n)$$

During OWO, the weights from the dependent input, feeding the outputs will be set to zero and the output weight adaptation will not be affected. During the input weight adaptation, the expression for gradient given by 6.6 can be re-written as,

$$\begin{aligned} \frac{\partial E}{\partial z_j} = \frac{-2}{N_v} \sum_{p=1}^{N_v} \sum_{m=1}^{M} \Big(t'_p(m) - \sum_{k=1}^{N_h} w_{oh}(m,k) O_p(z_k) \Big) \cdot \\ w_{oh}(m,j) O'_p(j) \left[\triangle n_p(j) + g(k, N+2) \sum_{n=1}^{N+1} b(n) x_p(n) \right] \end{aligned} \tag{6.18}$$

and the expression for an element of the Hessian can be re-written as 6.7

$$\frac{\partial^2 E}{\partial z_l \partial z_j} = \frac{2}{N_v} \sum_{m=1}^{M} w_{oh}(m,l) w_{oh}(m,j) \sum_{p=1}^{N_v} O'_p(l) O'_p(j) \Bigg[\triangle n_p(l) \triangle n_p(j) + \triangle n_p(l) \\ g(j, N+2) \sum_{n=1}^{N+1} b(n) x_p(n) + \triangle n_p(j) g(l, N+2) \sum_{i=1}^{N+1} b(i) x_p(i) + \\ g(j, N+2) g(l, N+2) \sum_{n=1}^{N+1} \sum_{i=1}^{N+1} b(i) x_p(i) b(n) x_p(n) \Bigg] \tag{6.19}$$

Comparing 6.6 with 6.18 and 6.7 with 6.19, we see some additional terms that appear within the square brackets in the expressions for gradient and Hessian in the presence of linearly dependent input. Clearly, these *parasitic* cross-terms will cause the training using MOLF to be different for the case of linearly dependent inputs. The $\mathbf{H}_{molf}$ thus obtained is not guaranteed to be singular.

6.8 Discussion

The sensitivity of Newton's method to dependent inputs and hidden units is illustrated. A non-heuristic second order learning algorithm is presented that calculates an optimal learning factor for every hidden unit using Newton's method. The proposed method makes it easier to detect dependent hidden units during inversion. Compared to LM, the proposed method is completely non-heuristic as it uses an optimal learning factor for every hidden unit. It is also faster since it uses a smaller Hessian. The reduced Hessian is shown to be a weighted sum of the elements of the Hessian for the entire network.

For the data sets investigated, using *10-fold*, the proposed algorithm has a better overall performance, in terms of the training error and the number of multiplies per training iteration, than LM.

The success of the MOLF algorithm is limited only by the presence of linearly dependent inputs. As in the case of OIG, the Hessian derived in the MOLF algorithm

will be non-singular in the presence of linearly dependent inputs and will be useless in detecting and eliminating the dependent inputs. A common approach applicable to both OIG and MOLF is desired that will make them immune to dependencies. This will the topic of discussion in the following chapter.

CHAPTER 7

IMPROVEMENTS TO OIG AND MOLF ALGORITHMS

The hidden weight optimization (HWO) technique was introduced in section 2.7. In this chapter we first show that HWO is immune to presence of linearly dependent inputs during training. We then replace the BP component in both OIG and MOLF with HWO. The resulting improved versions of OIG and MOLF are shown to be unaffected by linearly dependent inputs and better than using BP.

7.1 Effect of Dependence on HWO

HWO finds the input weight update by solving for a system of linear equations as given by equation (2.20), reproduced below for convenience.

$$\mathbf{G}_{hwo} \cdot \mathbf{R} = \mathbf{G}_{bp} \tag{7.1}$$

If one of the inputs to the network is linearly dependent, clearly it would cause the input auto-correlation matrix, $\mathbf{R}$ to be singular. In such a situation, using the conjugate gradient (C-G) algorithm to solve (7.1) would lead to poor training, since the convergence of C-G is affected by the presence of linearly dependent inputs. However, using OLS or inversion methods could prove useful in detecting and eliminating the linearly dependent input, as analyzed in the following subsections.

7.1.1 Orthogonal Least Squares

Using OLS to solve for $\mathbf{G}_{hwo}$ in (7.1) involves computing the orthonormal weight update matrix, $\mathbf{G}'_{hwo}$, as

$$\mathbf{G}'_{hwo} = \mathbf{G}_{bp} \cdot \mathbf{C}^T$$

where $\mathbf{C}$ is a lower triangular matrix of orthonormal coefficients of dimension $(N+1)$. The orthonormal weight update can be mapped to the original weight update as

$$\begin{aligned}\mathbf{G}_{hwo} &= \mathbf{G}'_{hwo} \cdot \mathbf{C} \\ &= \mathbf{G}_{bp} \cdot \mathbf{C}^T\mathbf{C} \end{aligned} \tag{7.2}$$

Assume $x_p(N+2)$ was linearly dependent. This would cause the $(N+2)^{th}$ row and column of $\mathbf{R}$ to be linearly dependent. During OLS, a singular auto-correlation matrix transforms to the $(N+2)^{th}$ row of $\mathbf{C}$ to be zero.

The expression for $\mathbf{G}_{hwo}$ contains $\mathbf{C}^T\mathbf{C}$, which will be a square, symmetric matrix with zeros for the $(N+2)^{th}$ row and column. This would reflect in $\mathbf{G}_{hwo}$ having zeros for the $(N+2)^{th}$ column. The implication is that the weight update vector computed for all input weights connected to the dependent input $(N+2)$ is zero. These weights are not updated during training, effectively *freezing* them. This is highly desirable, as the dependent input is not contributing any new information. Thus HWO-type update using OLS is perfectly capable of picking up linearly dependent inputs, leading to a very robust training algorithm.

7.1.2 Matrix Inversion using SVD

An alternate way of solving equation 2.20 is to use matrix inversion as,

$$\mathbf{G}_{hwo} = \mathbf{G}_{bp} \cdot \mathbf{R}^{-1}$$

Since the auto-correlation matrix needs to be computed only once for all inputs, the inversion also needs to be computed only for the first iteration.

Singular value decomposition (SVD) is a very powerful technique for solving linear least squares problem. Given a matrix, $\mathbf{B}$, of size $M \times N$ SVD, decomposes it as,

$$\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

$\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices of size $M \times N$ and $N \times N$, $\mathbf{\Sigma}$ is a diagonal matrix of singular values of size $N \times N$.

Calculating $\mathbf{B}^{-1}$ using the SVD is equivalent to computing the matrix product,

$$\mathbf{B}^{-1} = \mathbf{V}\left[diag(1/\sigma_n)\right]\mathbf{U}^T$$

Clearly, if $\mathbf{B}$ is ill-conditioned, the diagonal matrix $\mathbf{\Sigma}$ will have some singular values close to zero which transforms to the corresponding rows and columns being zero in the inverted matrix.

The overall effect of using the SVD for inversion of a singular auto-correlation matrix would be similar to using OLS. It would set the weights update corresponding to the dependent input to be zero. Either OLS or SVD can be used to solve for the HWO weight update. HWO lends itself to detecting and eliminating linearly dependent inputs that can result in poor learning. Hence, training algorithms using a HWO-type update, with either OLS or SVD can be extremely robust and immune to input dependence.

7.2 Improvements to OIG Algorithm

We have shown in section 7.1 that the HWO algorithm is immune to dependent inputs. However, the classic OIG algorithm still has some room for improvements. As a first improvement, we replace BP in OIG with HWO. The resulting algorithm called *OIG-HWO1* will be unaffected by dependencies. A second improvement, which we will refer to as *OIG-HWO2*, is to use a non-optimal factor in addition to the optimal input gain. This heuristic improvement is inspired by the success of Levenberg-Marquardt method introduced in section 2.6, which uses λ as a heuristic scale factor during training. This leads to a heuristic scaling of the gradients used to update the input weights and could possibly improve convergence speed and avoid local minima.

However, there is also a danger of *overshoot*, which will cause the error to be larger after OWO, in which case, we use *backtracking* to revert back to the best previous state and continue training. Using backtracking also ensures the convergence of the algorithm does not change due to the introduction of the non-optimal learning factor.

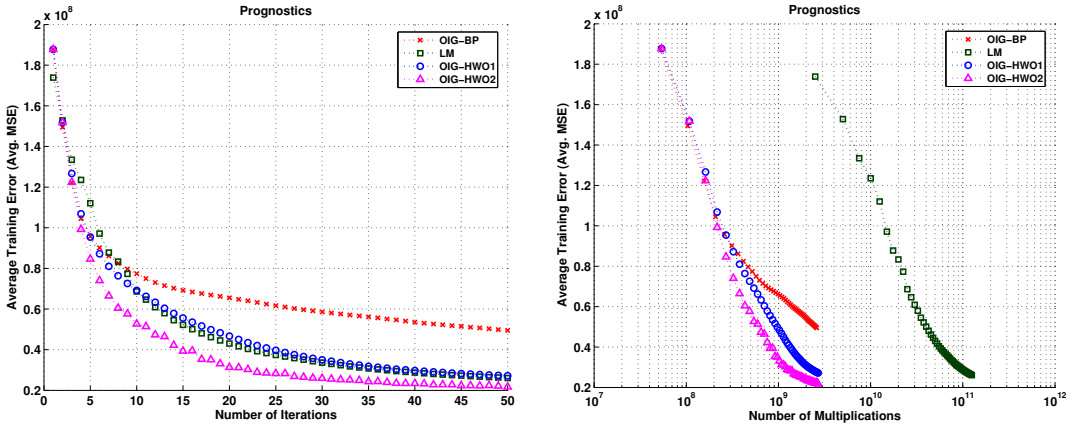
7.2.1 Effect of Linear Dependence on Improved OIG

In subsection 5.6.2, it was shown that in presence of linearly dependent inputs, the Hessian for OIG applied to OWO-BP is not guaranteed to be singular. OWO-HWO on the other hand, picks up the linearly dependent input and the resulting weight update matrix will be singular, as shown in section 7.1.

Revisiting equation 5.19 we can see that when HWO replaces BP in OIG, $h_{ig}(N+2,u) = 0 \quad \forall\ u \in (1, N+2)$. Since the Hessian is symmetric, this will force the $(N+2)^{th}$ row and column of the OIG Hessian to be zero. Similarly, the $(N+2)^{th}$ element of $\mathbf{g}_{ig}$ will be zero. Together, this translates to $a(N+2) = 0$, i.e. the input weights from the dependent input are not updated during training using OIG, as mentioned in section 5.2.3. Using OIG with OWO-HWO in place of BP, guarantees that the Hessian will be singular in the presence of linearly dependent inputs. This is highly desirable since the singular Hessian can be used to detect the dependent inputs and eliminate them during training as shown by *lemma 6-4*. This overcomes the limitations of OIG applied to OWO-BP and gives an improved algorithm that is resilient to dependent inputs.

Next, we repeat the experiments in 5.5 for OIG applied to OWO-HWO type learning. We compare the performance of *OIG-HWO1*, *OIG-HWO2*, LM and OIG applied to OWO-BP, that was developed in chapter 5. We use the *k-fold* cross-validation procedure to obtain the average training and validation errors.

7.2.2 Prognostics Data Set



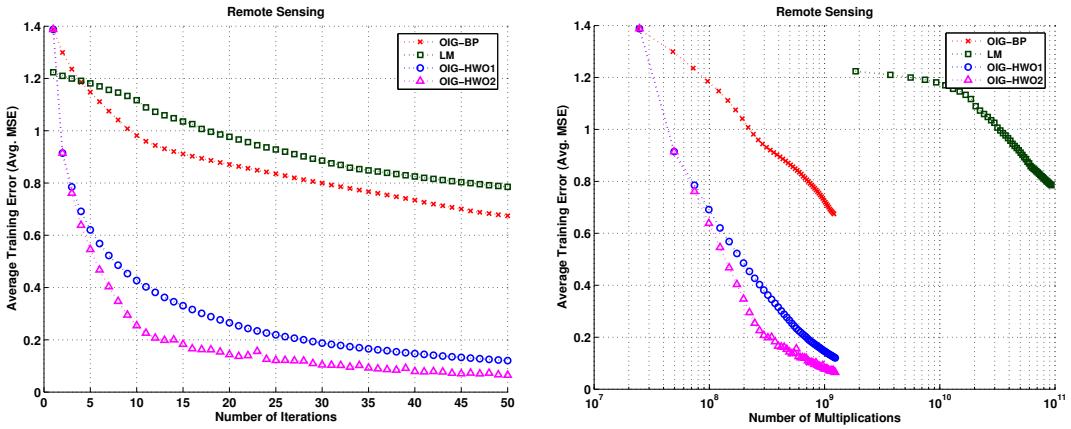
(a) (b)

Figure 7.1. Prognostics Data: average error vs. (a) iterations and (b) multiplies.

For this data file, which is called *F17*, we trained an MLP having 15 hidden units. In Fig. 7.1-a, the average mean square error (MSE) for training from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.1-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies (shown on a log_{10} scale).

7.2.3 Remote Sensing Data Set

For this data file, which is called *Single2*, we trained an MLP having 15 hidden units. In Fig. 7.2-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.2-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.

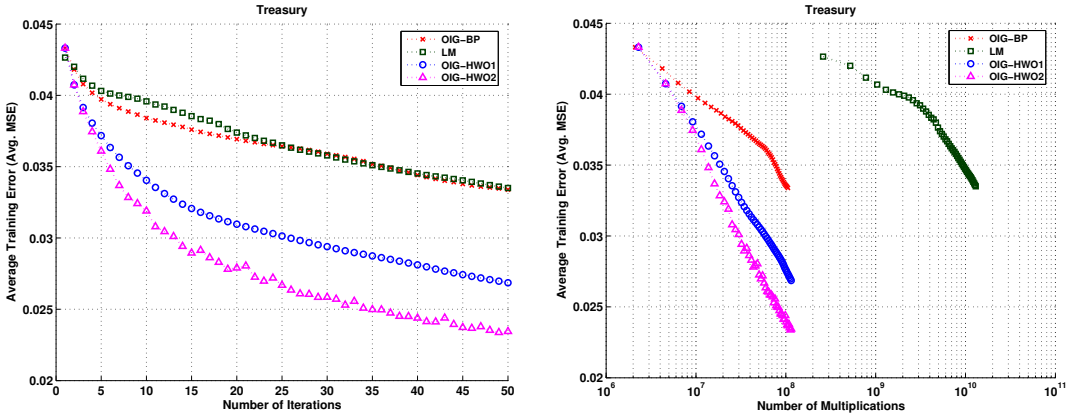


(a) (b)

Figure 7.2. Remote Sensing Data: average error vs. (a) iterations and (b) multiplies.

7.2.4 Federal Reserve Economic Data Set

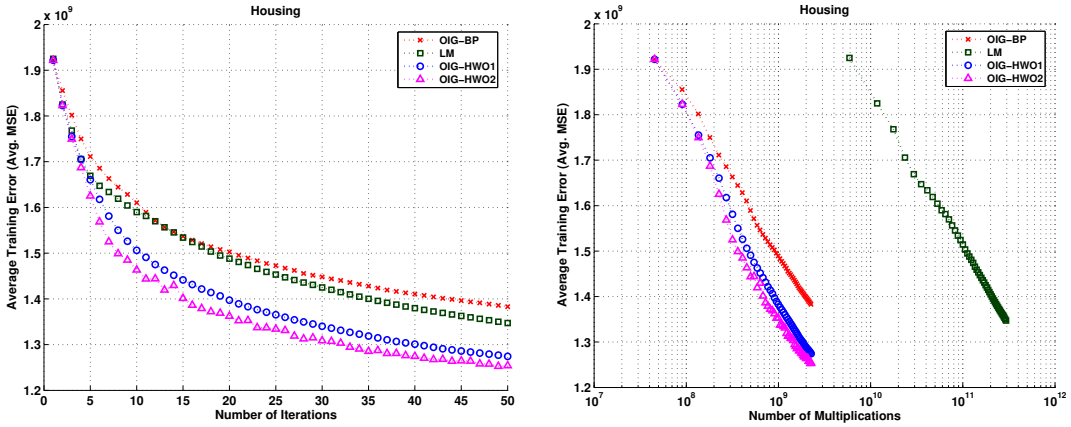
For this data file called *TR*, we trained an MLP having 15 hidden units. In Fig. 7.3-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.3-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.



(a) (b)

Figure 7.3. Remote Sensing Data: average error vs. (a) iterations and (b) multiplies.

7.2.5 Housing Data Set



(a) (b)

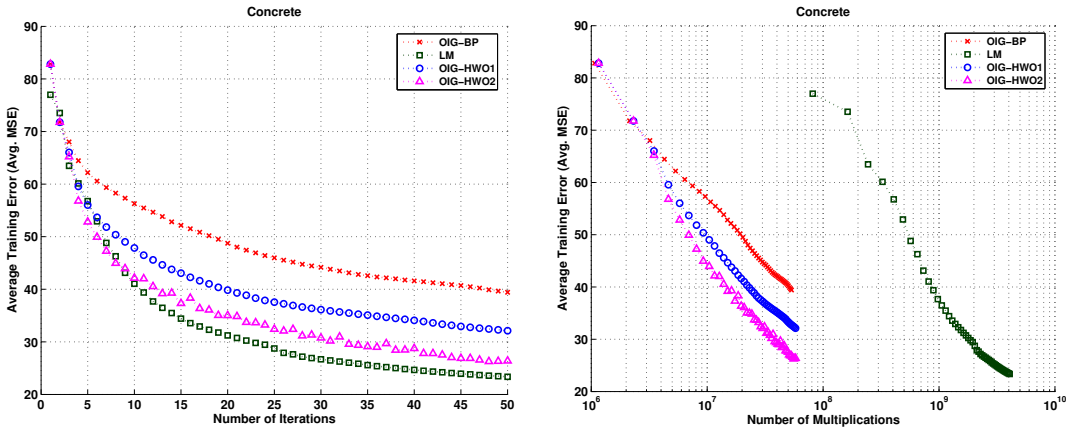
Figure 7.4. Housing Data: Average error vs. (a) iterations and (b) multiplies.

For this data file, we trained an MLP having 15 hidden units. In Fig. 7.4-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.4-b, the MSE from 10-fold validation is plotted versus the required number of multiplies.

7.2.6 Concrete Compressive Strength Data Set

For this data file, we trained an MLP having 15 hidden units. In Fig. 7.5-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.5-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.

Table 7.1 compares the average training and validation errors of the proposed OIG algorithm with BP, OWO-BP and LM on different data sets. For each data set, the training and validation errors again come from 10-fold cross validation.



(a) (b)

Figure 7.5. Concrete Data: Average error vs. (a) iterations and (b) multiplies.

From the plots, we see that the two suggested improvements are effective in reducing the error compared to the original OIG version applied to OWO-BP. The non-optimal learning factor seems to help and consistently has a better performance over the rest.

7.3 Improvement to MOLF Algorithm

The improvements suggested in section 7.2 can be easily extended to the MOLF algorithm too. In subsection 6.7.2, using 6.18 and 6.19 it was shown that the presence of a linearly dependent input reflects in additional *parasitic* terms appearing in the expression for the gradient and Hessian for MOLF applied to OWO-BP. The additional terms will affect the learning factor calculation, causing the training to be different. Since OWO-HWO is better equipped to handle linearly dependent inputs, replacing BP by HWO will force the parasitic terms in 6.19 to be zero and the optimal learning factors thus computed would completely ignore any contributions from the dependent input. This overcomes the limitations of MOLF applied to OWO-BP and gives an improved algorithm that is resilient to dependent inputs.

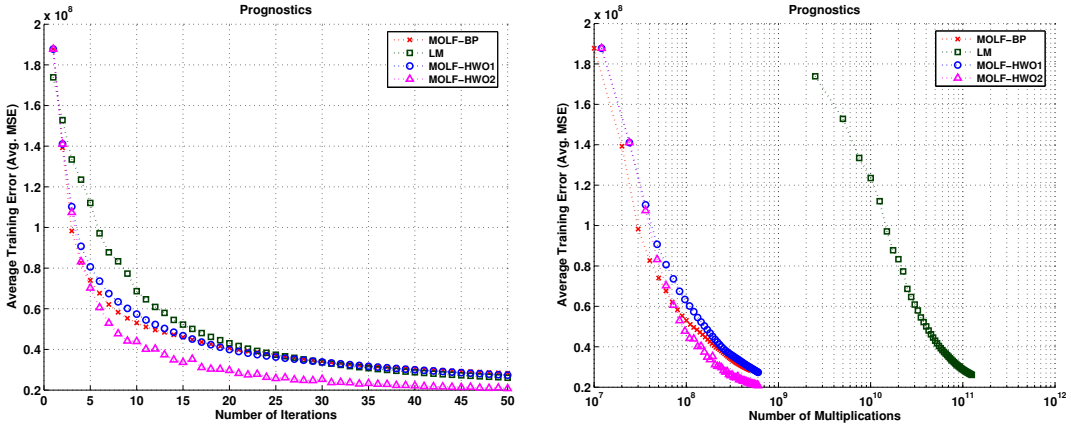
Table 7.1. Average 10-fold training and validation error

Data set		OIG-BP	OIG-HWO1	OIG-HWO2	LM
Prognostics	E_{trn}	4.8109E7	2.7160E7	2.17142E7	2.6089E7
	E_{val}	5.1440E7	3.0997E7	2.5239E7	2.9399E7
Remote Sensing	E_{trn}	0.617487	0.120338	0.064930	0.785390
	E_{val}	0.777170	0.187501	0.218238	0.863154
Treasury	E_{trn}	0.032190	0.026859	0.023448	0.033500
	E_{val}	0.041435	0.040166	0.038320	0.040850
Housing	E_{trn}	1.3849E9	1.2742E9	1.2541E9	1.3472E9
	E_{val}	1.4386E9	1.3611E9	1.3431E9	1.3917E9
Concrete	E_{trn}	34.5438	32.1205	26.385839	23.3564
	E_{val}	41.5764	45.448286	41.855400	31.9576

Here we present the results of replacing BP by HWO in MOLF and call this *MOLF-HWO1*. As a separate improvement, similar to the one mentioned in section 7.2, we use a non-optimal learning factor in addition to MOLF for updating the weights and call this method *MOLF-HWO2*. Results are compared to those of with MOLF in OWO-BP and LM. LM updates all the weights in each iteration. OWO-BP alternately uses BP for input weights and solves linear equations for output weights.

7.3.1 Prognostics Data Set

We trained an MLP having 15 hidden units. In Fig. 7.6-a, the average mean square error (MSE) for training from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.6-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies (shown on a log_{10} scale).

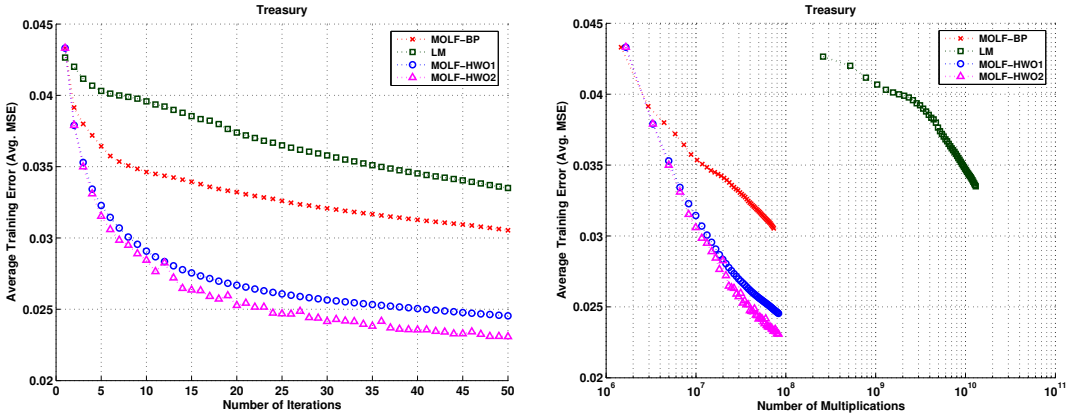


(a) (b)

Figure 7.6. Prognostics Data: Average error vs. (a) iterations and (b) multiplies.

7.3.2 Federal Reserve Economic Data Set

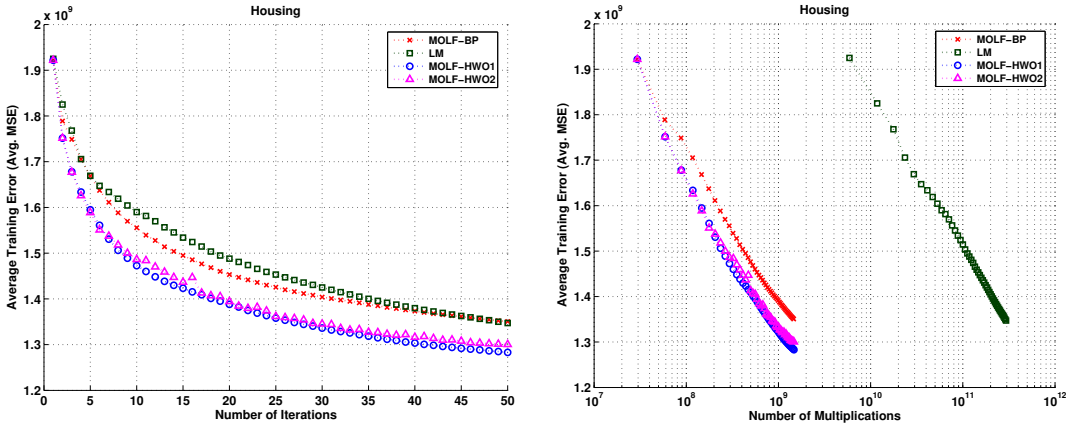
For this data file, we trained an MLP having 15 hidden units. In Fig. 7.7-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.7-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.



(a) (b)

Figure 7.7. Federal reserve data: average error vs. (a) iterations and (b) multiplies.

7.3.3 Housing Data Set



(a) (b)

Figure 7.8. Housing data: average error vs. (a) iterations and (b) multiplies.

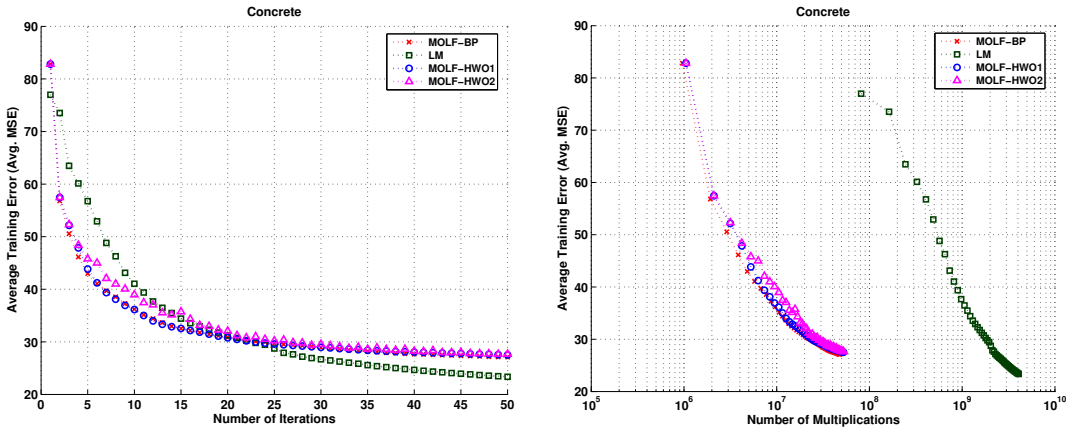
We trained an MLP having 15 hidden units. In Fig. 7.8-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.8-b, the MSE from 10-fold validation is plotted versus the required number of multiplies.

7.3.4 Concrete Compressive Strength Data Set

We trained an MLP having 15 hidden units. In Fig. 7.9-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each algorithm. In Fig. 7.9-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.

7.3.5 Remote Sensing Data Set

We trained an MLP having 15 hidden units. In Fig. 7.10-a, the average training MSE from 10-fold validation is plotted versus the number of iterations for each



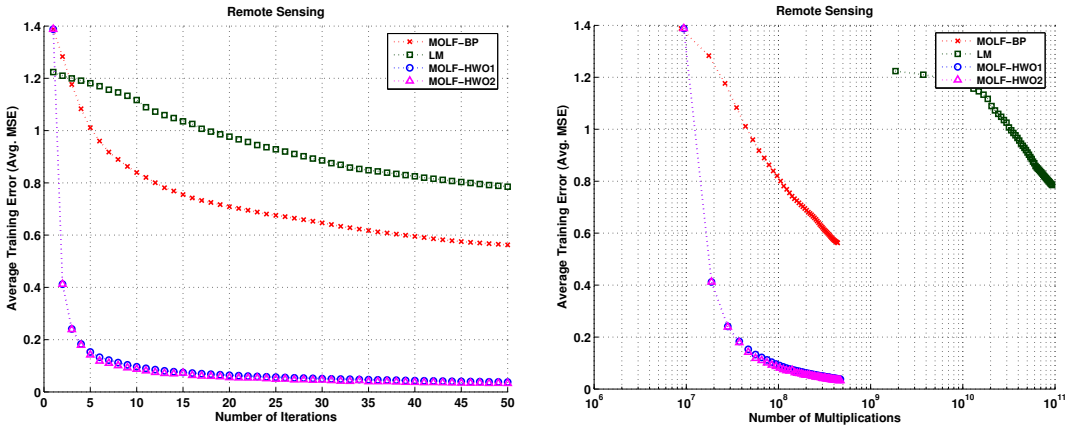
(a) (b)

Figure 7.9. Concrete data: average error vs. (a) iterations and (b) multiplies.

algorithm. In Fig. 7.10-b, the average training MSE from 10-fold validation is plotted versus the required number of multiplies.

Table 7.2 compares the average training and validation errors of the proposed MOLF algorithm with BP, OWO-BP and LM on different data sets. For each data set, the training and validation errors again come from 10-fold cross validation. We can see that the proposed MOLF algorithm often has a performance comparable to or better than the LM.

We can see from the plots that while HWO consistently makes the MOLF algorithm better, the same cannot be said about using the non-optimal learning factor. It helps in some cases, but not always. This is a common theme to methods that employ heuristics to improve the speed of operation.



(a) (b)

Figure 7.10. Remote sensing data: average error vs. (a) iterations and (b) multiplies.

Table 7.2. Average 10-fold training and validation error

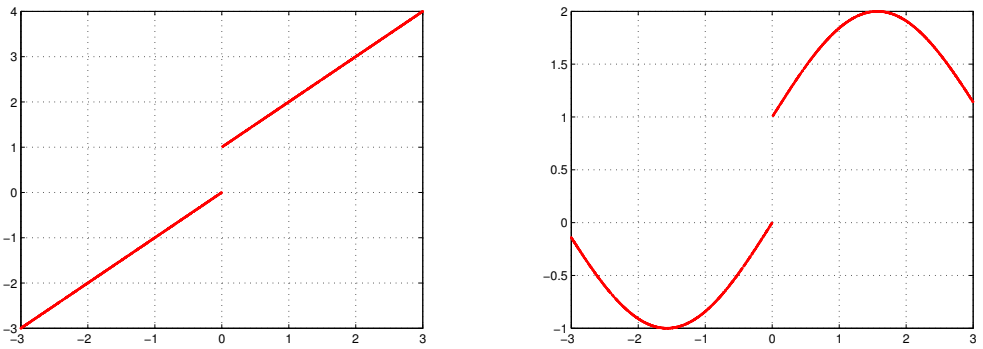
Data set	MSE	MOLF-BP	MOLF-HWO1	MOLF-HWO2	LM
Prognostics	E_{trn}	2.9057E7	2.0779E7	2.0771E7	2.6089E7
	E_{val}	3.2475E7	3.0968E7	2.4154E7	2.9399E7
Federal Reserve	E_{trn}	0.029490	0.023109	0.023088	0.033500
	E_{val}	0.039010	0.040929	0.036753	0.040850
Housing	E_{trn}	1.3453E9	1.3027E9	1.3005E9	1.3472E9
	E_{val}	1.4126E9	1.3892E9	1.4001E9	1.3917E9
Concrete	E_{trn}	22.7746	28.146295	27.660962	23.3564
	E_{val}	30.7310	38.086012	38.936990	31.9576
Remote Sensing	E_{trn}	0.559648	0.031619	0.032912	0.785390
	E_{val}	0.665496	0.056386	0.050364	0.863154

CHAPTER 8

MODELING SIMPLE DISCONTINUOUS FUNCTIONS

In section 4.2.5, we stated that OWO-BP type training algorithms for the MLP cannot model discontinuous functions. In this chapter, we first illustrate the problem and extend the OIG algorithm developed in this dissertation to model simple discontinuous functions.

8.1 Discontinuous Function



(a) (b)

Figure 8.1. Simple Discontinuous Functions: (a) Ramp and (b) Sine.

A discontinuous or a piecewise continuous function, is any function that is not continuous for all values of the dependent parameter, i.e. some function $y = f(x)$, which is dependent on the continuous variable x may not have values defined for some values of x. A simple example is a step function. The step function $u(x)$ is $+1$ for

$x > 0$ and 0 for $x < 0$, but is not defined around zero. Simple discontinuous functions, similar to the ones used in [47] are shown in figure 8.1.

8.1.1 Problem Illustration

Discontinuities in training data can render the approximation by an MLP inaccurate. To illustrate this problem, we trained an MLP with one hidden unit using OWO-BP and LM. Theoretically, the sigmoid activation can be adapted to model a

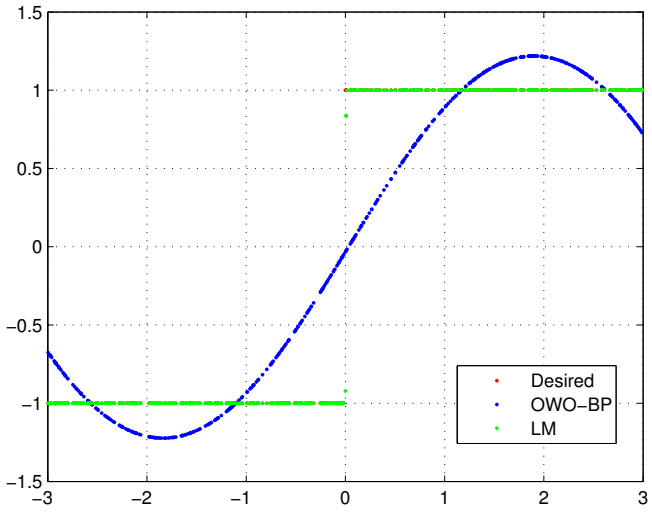


Figure 8.2. Result of using OWO-BP and LM to model a step function.

simple step function. The result of using OWO-BP and LM to model a step function is shown in figure 8.2. We see that OWO-BP fails to model the step function, while LM has better success. As seen in figure 8.3, this advantage does not carry through when LM is used to model the discontinuous sine function.

Both LM and OWO-BP seem to settle into an *average* value at the boundary of discontinuity. In general, OWO-BP is faster and more efficient than the LM, but

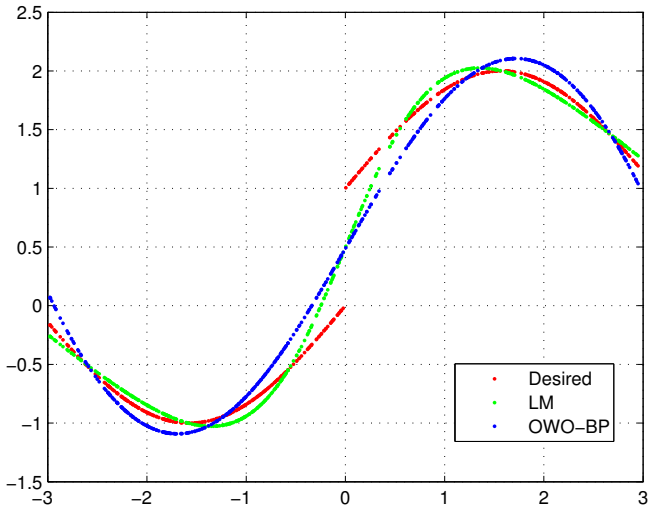


Figure 8.3. Result of using OWO-BP and LM to model a step function.

seems ill-equipped to handle simple discontinuous functions and while LM shows some promise on step discontinuity, it fails to model the discontinuous sine function, hence making it not so reliable.

8.2 A Fusion Approach to Model Discontinuous Functions

In this section we present a method to model simple discontinuous functions, like the ones shown in figure 8.1. The basic idea is that a discontinuous function can be split into a continuous function and a step function, with the step function appearing at the boundary of discontinuity.

In our approach, we divide the training data into separate continuous and step data files. For each data set we trained a separate network with one hidden unit. The improved OIG algorithm, *OIG-HWO2* was used for training the two networks. This procedure is illustrated in figure 8.4. Figures 8.5 and 8.6 show the output of the two networks. After training, the two networks were *fused*, i.e. the weights from the

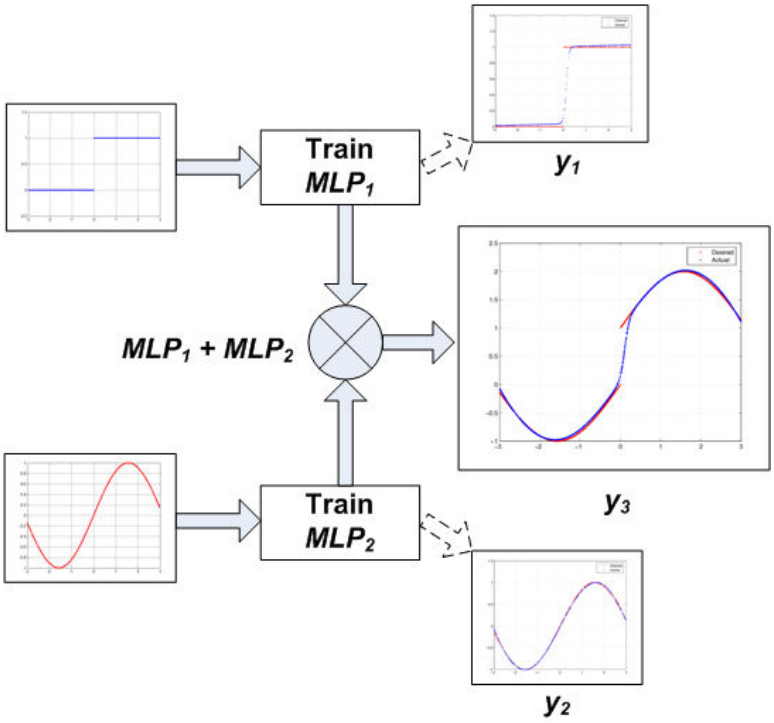


Figure 8.4. Block diagram of the fusion approach.

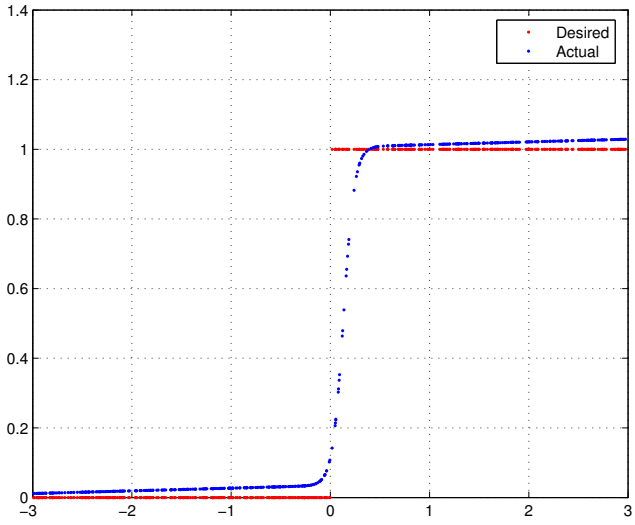


Figure 8.5. MLP-1: Trained using improved OIG on step data.

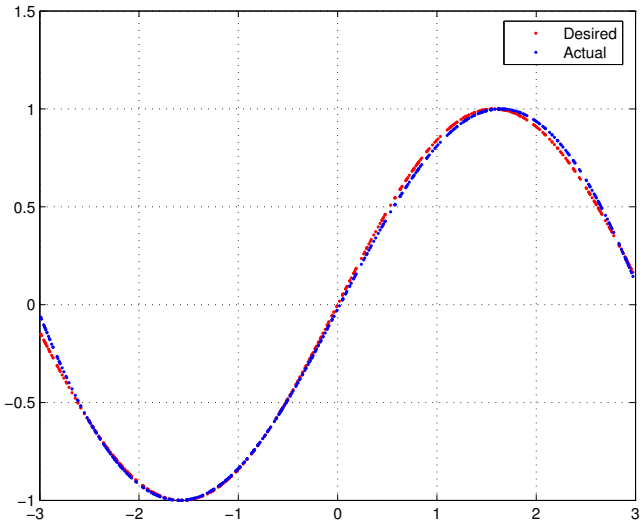


Figure 8.6. MLP-2: Trained using improved OIG on continuous sine data.

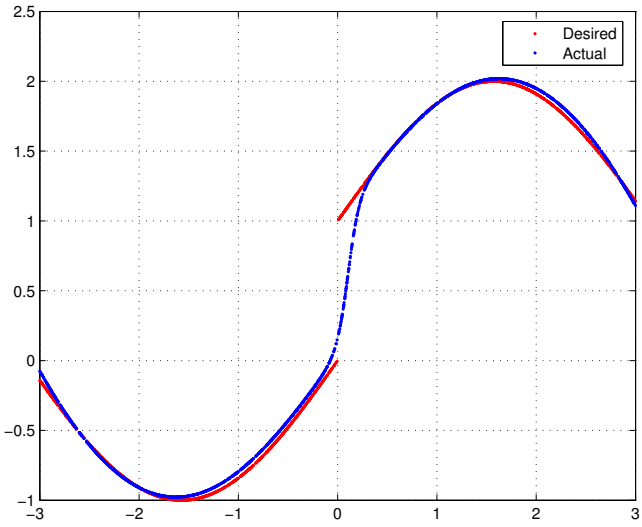


Figure 8.7. Output of fused network (MLP1+MLP2) for discontinuous sine data.

two networks were combined to form a single network. The fused network contains one input, two hidden units, one trained on step and the other trained on the sine data, and one output. A third data set that was generated by adding the outputs of step and sine data to form a discontinuous sine function. The fused network was now used to process the newly formed discontinuous sine data. The output of the fused network is shown in figure 8.7.

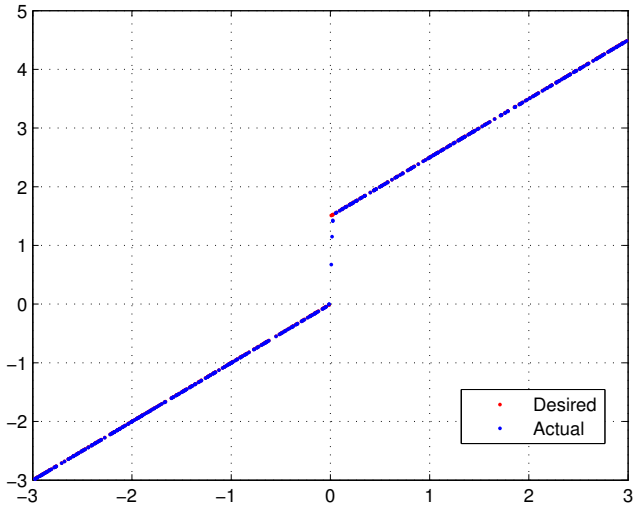


Figure 8.8. Output of fused network (MLP1+MLP2) for discontinuous ramp data.

We carried out a similar procedure to model the discontinuous ramp function, with equal success as shown in figure 8.8.

8.3 Discussion

The basic idea is to represent a discontinuity as the sum of a continuous functions and a step function. Fusing networks which individually model the continuous and the step data, works with simple discontinuities. The implications are that *global*

learning techniques used in OWO-BP do not work well for modeling discontinuous functions. The suggested fusion approach is equivalent to training a network with units that are locally tuned and capable of modeling both the continuous function and the discontinuity. The results are promising for some simple discontinuous functions. The problem now is to find a method to separate the data into a continuous and a step function. This will require further exploration.

CHAPTER 9

CONTRIBUTIONS AND FUTURE WORK

This dissertation addresses several interesting issues related to MLP training. In this chapter, we summarize the work done and suggest possible future improvements.

9.1 Contributions

9.1.1 Mathematical Analysis of Linear Dependence on Training

By establishing the concept of equivalent networks and developing simple models for linearly dependent inputs and hidden units, we have developed a theoretical framework which brings out the effect of linear dependence on first and second order learning methods.

9.1.2 Optimal Input Gain Algorithm

We have derived a second order method for simultaneously optimizing input gains and the OLF. The method has been successfully demonstrated on five data sets. Results show that this approach performs much better than two common first order algorithms with comparable complexity, namely BP and OWO-BP. It comes close to LM in terms of the training error, but with orders of magnitude less computation. This is evident in all of the plots of training error versus the required number of multiplies and also from the expressions for the numbers of multiplies.

Although LM works very well in practice it has a high computational burden and is sub-optimal in the way it handles the 'scaling factor', λ. OIG on the other

hand uses a Newton type update and combines the optimal learning factor, leaving little room for heuristics.

9.1.3 Multiple Optimal Learning Factor Algorithm

The sensitivity of Newton's method to dependent inputs and hidden units is illustrated. A non-heuristic second order learning algorithm is presented that calculates an optimal learning factor for every hidden unit using Newton's method. The proposed method makes it easier to detect dependent hidden units during the Hessian inversion. Compared to LM, the proposed method is completely non-heuristic as it uses an optimal learning factor for every hidden unit. It is also faster since it uses a smaller Hessian. The reduced Hessian is shown to be a weighted sum of the elements of the Hessian for the entire network.

For the data sets investigated, using *10-fold* validation, the proposed algorithm has a better overall performance, in terms of the training error and the number of multiplies per training iteration, than LM.

9.1.4 Improvements to OIG and MOLF Algorithms

We presented an analysis which shows that the presence of dependent inputs negatively impacts the OIG and MOLF algorithms when used with the OWO-BP learning method. However, when HWO replaces BP in OIG and MOLF, both algorithms completely negate any effect of the linearly dependent input and the learning is identical to the case of linearly independent inputs. This improvement is significant as it equips the two algorithms to handle dependence internally, during learning, by completely suppressing it. The other suggested performance improvement was the use an additional non-optimal learning factor during the weight update stage for which, we observed a slight improvement in the speed of convergence.

9.1.5 A Fusion-based Approach for Modeling Discontinuous Functions

We presented a technique to model simple discontinuous functions, which involved breaking up the discontinuous function into a continuous function and a step function. Two separate networks were trained, one on the continuous function, the other on the step function and fused together to model the discontinuous function. Results of using improved OIG on discontinuous sine and ramp functions are encouraging. However, the problem of splitting data into a continuous function and a discontinuous function needs investigation.

9.1.6 Convergence Theorem for HWO

Another theoretical contribution is the proof of convergence for OWO-HWO given in appendix B. We extend the proof of convergence for traditional OWO-BP and show that each iteration of OWO-HWO is equivalent to an iteration of OWO-BP on transformed data and hence guaranteed to converge. This forms the missing piece in the theory of OWO-HWO which should increase its popularity as a training algorithm.

9.2 Future Work

Some improvements and extensions of the current work in this dissertation include

(i) Developing a non-diagonal version of $\mathbf{A}^T\mathbf{A}$ in OIG. Instead of using only a diagonal transformation, it will be interesting to see how the other elements of $\mathbf{A}^T\mathbf{A}$ affect training.

(ii) Combining OIG and MOLF could lead to a more powerful training algorithm.

(iii) Extending of OIG and MOLF for classification would be useful.

(iv) Extending the proposed fusion approach to model more complicated discontinuities.

APPENDIX A

DATA SETS

A.1 Prognostics Data Set

This data file is available on the Image Processing and Neural Networks Lab repository [57]. It consists of parameters that are available in the Bell Helicopter health usage monitoring system (HUMS), which performs flight load synthesis, which is a form of prognostics [58]. The data was obtained from the M430 flight load level survey conducted in Mirabel Canada in early 1995. The seventeen input features include cockpit available signals including CG F/A load factor, pitch attitude, roll attitude, yaw rate, and several others. The nine desired outputs are loads on various mechanical components.

A.2 Remote Sensing Data Set

This data file is available on the Image Processing and Neural Networks Lab repository [57]. It consists of 16 inputs and 3 outputs and represents the training set for inversion of surface permittivity, the normalized surface rms roughness, and the surface correlation length found in back scattering models from randomly rough dielectric surfaces [59].

A.3 Federal Reserve Economic Data Set

This file contains some economic data for the USA from 01/04/1980 to 02/04/2000 on a weekly basis. From the given features, the goal is to predict the 1-Month CD Rate [60].

A.4 Housing Data Set

This data file is available on the DELVE data set repository [61]. It was designed on the basis of data provided by the US Census Bureau (under Lookup Access:

Summary Tape File 1). The data were collected as part of the 1990 US census. These are mostly counts cumulated at different survey levels. For the purpose of this data set a level State-Place was used. Data from all states was obtained. Most of the counts were changed into appropriate proportions [65].These are all concerned with predicting the median price of houses in a region based on demographic composition and the state of the housing market in the region. For Low task difficulty, more correlated attributes were chosen as signified by univariate smooth fit of that input on the target. Tasks with high difficulty have had their attributes chosen to make the modeling more difficult due to higher variance or lower correlation of the inputs to the target. The training data consists of 16 inputs and 1 output per pattern, with a total of 22,784 patterns.

A.5 Concrete Compressive Strength Data Set

This data file is available on the UCI Machine Learning Repository [62]. It contains the actual concrete compressive strength (MPa) for a given mixture under a specific age (days) determined from laboratory. The concrete compressive strength is a highly nonlinear function of age and ingredients. These ingredients include cement, blast furnace slag, fly ash, water, super plasticizer, coarse aggregate, and fine aggregate. The data set consists of 8 inputs and one output per pattern, with a total of 1030 patterns.

APPENDIX B

CONVERGENCE PROOF FOR HWO ALGORITHM

This document proves the convergence of the error function for hidden weight optimization (HWO) algorithm. This does not necessarily translate to global convergence of the algorithm, it simply means that the mean square error for a training algorithm using HWO will be convergent.

The classic backpropagation (BP) algorithm is introduced briefly, followed by a brief description of the HWO algorithm. Using concepts of equivalent networks, it is shown HWO is equivalent to BP on a *transformed* network. In general, BP is equivalent to steepest descent and is guaranteed to converge, provided that the learning rate is globally convergent. Hence in order to prove convergence of HWO, it would suffice if it is shown that HWO is equivalent to BP on a transformed network.

B.1 Backpropagation

A typical error function used in training the MLP is the mean-squared error (MSE) described as

$$E = \frac{1}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} [t_p(i) - y_p(i)]^2 \tag{B.1}$$

In full batch mode BP, the hidden unit weights and thresholds are updated using the steepest descent approach as

$$w(k,j) = w(k,j) + z(\frac{-\partial E}{\partial w(k,j)}) \tag{B.2}$$

for all $1 \leq k \leq N_h$ and all $1 \leq j \leq (N+1)$.

The negative gradient of E is

$$g_{bp}(k,n) = -\frac{\partial E}{\partial w(k,n)} = \frac{1}{N_v} \sum_{p=1}^{N_v} \delta_p(k) x_p(n) \tag{B.3}$$

Using matrix notations, the negative gradients can be written as

$$\mathbf{G}_{bp} = \frac{1}{N_v} \sum_{p=1}^{N_v} \boldsymbol{\delta}_p \mathbf{x}_p^T \tag{B.4}$$

where $\boldsymbol{\delta}_p = [\delta_p(1), \delta_p(2), \cdots, \delta_p(N_h)]^T$. If steepest descent is used to modify the hidden weights, $\mathbf{W}$ is updated in a given iteration as

$$\mathbf{W} \leftarrow \mathbf{W} + z \cdot \mathbf{G}_{bp} \tag{B.5}$$

so

$$\triangle\mathbf{W} = z \cdot \mathbf{G}_{bp} \tag{B.6}$$

where z is the optimal learning factor.

As mentioned in chapter 2, BP is equivalent to steepest descent and for an optimal learning factor, is guaranteed to converge to a global or local minimum.

B.2 Hidden Weight Optimization

Hidden weight optimization (HWO) is an alternate to classic backpropagation method for obtaining weight changes. The HWO algorithm utilizes separate error functions for each hidden unit. The aim of the classic HWO algorithm is to minimize the objective function defined as

$$E_\delta(j) = \sum_{p=1}^{N_v} \left[\delta_p(j) - \sum_{n=1}^{N+1} g_{hwo}(k,n) x_p(n) \right]^2 \tag{B.7}$$

for $0 \leq i \leq N_h$, by solving for linear equations of the form

$$\sum_{n=1}^{N+1} g_{hwo}(k,n) r(n,m) = \frac{-\partial E}{\partial w(j,m)} \tag{B.8}$$

where, $-\partial E/\partial w(j,m)$ is the BP gradient as computed in equation (B.3).

In matrix notation,

$$\mathbf{G}_{hwo} \cdot \mathbf{R} = \mathbf{G}_{bp} \tag{B.9}$$

where, $\mathbf{R}$ is the input auto-correlation matrix and $\mathbf{G}_{hwo}$ is the HWO weight changes. Solving for the elements of $\mathbf{G}_{hwo}$, the input weights are updated as,

$$w(k,n) \leftarrow w(k,n) + z \cdot g_{hwo}(k,n)$$

where z is the optimal learning factor.

It can be shown that HWO is equivalent to linearly transforming the training data and then performing BP which will be discussed in the next sections.

B.3 Linear Transformation of Inputs and Equivalent Networks

Let $\mathbf{A}$ be a non-singular, non-orthogonal transformation matrix, that transforms the input vector $\mathbf{x}_p$ to $\mathbf{x}'_p$ as,

$$\mathbf{x}'_p = \mathbf{A} \cdot \mathbf{x}_p \tag{B.10}$$

If we have to analyze the effect of the transformation $\mathbf{A}$ on training a network, then the idea of *equivalent network* is very important.

Definition: *Two networks, one for training on original data* $\{\mathbf{x}_p, \mathbf{t}_p\}_{p=1}^{N_v}$ *and the other for training on data with linearly transformed input,* $\{\mathbf{x}'_p, \mathbf{t}_p\}_{p=1}^{N_v}$ *are equivalent if they are initialized such that the output vector,* $\mathbf{y}_p$*, before training is identical.*

Equivalency translates to the networks being initialized in such a way that they have identical hidden unit activations and outputs for every training pattern. It should be noted this is only an initial condition to ensure that the networks have the same starting point. Past the equivalent initialization, each network can train independently and any deviation or lack thereof due to the transformation should be evident.

For the p^{th} pattern, the hidden unit net function vector, $\mathbf{n}'_p$, for the network trained on transformed input can be expressed as

$$\mathbf{n}'_p = \mathbf{W}' \cdot \mathbf{x}'_p \tag{B.11}$$

where, $\mathbf{W}'$ is the input weight matrix for the transformed network. If this has to be equal to the net function of the network trained on the original input, then

$$\mathbf{n}'_p = \mathbf{n}_p$$

$$\mathbf{W}' \cdot \mathbf{x}'_p = \mathbf{W} \cdot \mathbf{x}_p$$

From equation (B.10), the weights for the network trained on original data can be obtained as,

$$\mathbf{W}' \cdot \mathbf{A} = \mathbf{W} \tag{B.12}$$

$$\Delta\mathbf{W}' \cdot \mathbf{A} = \Delta\mathbf{W} \tag{B.13}$$

Modifying equation (B.4) for the transformed network,

$$\mathbf{G}'_{bp} = \frac{1}{N_v} \sum_{p=1}^{N_v} \boldsymbol{\delta}_p (\mathbf{x}'_p)^T \tag{B.14}$$

Using equation (B.10) and (B.4) in (B.14)

$$\mathbf{G}'_{bp} = \mathbf{G}_{bp} \cdot \mathbf{A}^T$$

So the gradient matrix for the transformed network is that of the equivalent network times $\mathbf{A}^T$. Using equation(B.10), we can now map $\mathbf{G}'$ back to the equivalent network as

$$\mathbf{G}'' = \mathbf{G}_{bp} \cdot \mathbf{A}^T \mathbf{A} \tag{B.15}$$

$\mathbf{G}''$ denotes $\mathbf{G}'_{bp}$ after it is mapped back to a equivalent network.

B.4 Convergence of HWO

As mentioned, HWO solves linear equations of the form indicated in equation (B.9). One way of solving equation (B.9) is to use matrix inversion as,

$$\mathbf{G}_{hwo} = \mathbf{G}_{bp} \cdot \mathbf{R}^{-1}$$

The inverse of the auto-correlation matrix can be computed using the singular value decomposition (SVD). The SVD decomposes the auto-correlation matrix as

$$\mathbf{R} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices and $\mathbf{\Sigma}$ is a diagonal matrix of singular values. Calculating $\mathbf{R}^{-1}$ using the SVD is equivalent to computing the matrix product,

$$\mathbf{R}^{-1} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T$$

Factoring $\mathbf{\Sigma}$, the above equation can be re-written as,

$$\mathbf{R}^{-1} = \left(\mathbf{V}\left[diag(1/\sqrt{\sigma_n})\right]\right) \cdot \left(\left[diag(1/\sqrt{\sigma_n})\right]\mathbf{U}^T\right)$$
$$\mathbf{R}^{-1} = \mathbf{A}^T\mathbf{A}$$

where $\mathbf{A} = \left(\left[diag(1/\sqrt{\sigma_n})\right]\mathbf{U}^T\right)$ and since $\mathbf{R}$ is symmetric, $\mathbf{U} \equiv \mathbf{V}$.

We can now re-write equation (B.9) as,

$$\mathbf{G}_{hwo} = \mathbf{G}_{bp} \cdot \mathbf{A}^T\mathbf{A} \tag{B.16}$$

Comparing this with equation (B.15) and based on the analysis in the previous section, it is clear that performing OWO-HWO is equivalent to performing OWO-BP on transformed data. Since OWO-BP was proved to converge in chapter 2, it is clear that OWO-HWO converges as well.

REFERENCES

[1] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[2] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[3] B. Widrow, "Generalization and information storage in networks of adeline 'neurons'," in *Self-Organizing Systems*, M. C. Yovitz, G. T. Jacobi, and G. D. Goldstein, Eds. Spartan Books, 1962, pp. 435–461.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*, D. E. Rumelhart and J. L. Mcclelland, Eds. Cambridge, Massachusetts: The MIT Press, 1986, vol. I.

[5] K. Levenberg, "A method for the solution of certain problems in least squares," *The Quarterly of Applied Mathematics*, vol. 2, pp. 164–168, 1944.

[6] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM Journal on Applied Mathematics*, vol. 11, p. 431441, 1963.

[7] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Pearson Education, 2004.

[8] R. C. Odom, P. Pavlakos, S. Diocee, S. M. Bailey, D. M. Zander, and J. J. Gillespie, "Shaly sand analysis using density-neutron porosities from a cased-hole pulsed neutron system," in *SPE Rocky Mountain regional meeting proceedings: Society of Petroleum Engineers*, 1999, pp. 467–476.

[9] A. Khotanzad, M. H. Davis, A. Abaye, and D. J. Maratukulam, "An artificial neural network hourly temperature forecaster with applications in load forecasting," *IEEE Transactions on Power Systems*, vol. 11, no. 2, pp. 870–876, May 1996.

[10] S. Marinai, M. Gori, and G. Soda, "Artificial neural networks for document analysis and recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 1, pp. 23–35, 2005.

[11] R. B. J. Kamruzzaman, R. A. Sarker, *Artificial Neural Networks: Applications in Finance and Manufacturing.* Idea Group Inc (IGI), 2006.

[12] L. Wang and X. Fu, *Data Mining With Computational Intelligence.* Springer-Verlag, 2005.

[13] G. Edwards and J. P. Tate, "Target recognition and classification using neural networks," in *Proceedings of MILCOM 2002*, vol. 2, Oct 2002, pp. 1439–1442.

[14] E. F. M. Filho and A. C. P. L. de Carvalho, "Target recognition using evolutionary neural networks," in *Proceedings of Vth Brazilian Symposium on Neural Networks, 1998*, Dec 1998, pp. 226–231.

[15] K. Liu, S. Subbarayan, R. R.Shoults, M. T. Manry, C. Kwan, F. L. Lewis, and J.Naccarino, "Comparison of very short-term load forecasting techniques," *IEEE Transactions on Power Systems*, vol. 11, no. 2, pp. 877–882, May 1996.

[16] M. T. Manry, R. Shoults, and J. Naccarino, "An automated system for developing neural network short term load forecasters," in *Proceedings of the 58th American Power Conference*, Apr 1996, pp. 237–241.

[17] Y. Saifullah and M. T. Manry, "Classification based segmentation of zip codes," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 5, pp. 1437–1443, Sep/Oct 1993.

[18] Y. LeCun, B. Boser, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, pp. 541–551, 1989.

[19] M. T. Manry, C.-H. Hsieh, and H. Chandrasekaran, "Near-optimal flight load synthesis using neural nets," in *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop*, 1999, pp. 535–544.

[20] P. Hong, Z. Wen, and T. S. Huang, "Real-time speech-driven face animation with expressions using neural networks," *IEEE Transaction on Neural Networks*, vol. 13, no. 4, pp. 916–927, July 2002.

[21] P. Muneesawant and L. Guan, "Automatic machine interaction for content-based image retrieval using a self organizing tree map structure," *IEEE Transaction on Neural Networks*, vol. 13, no. 4, pp. 821–834, July 2002.

[22] I. Lapidot, H. Gunterman, and A. Cohen, "Unsupervised speaker recognition based on competition between self-organizing maps," *IEEE Transaction on Neural Networks*, vol. 13, no. 4, pp. 877–887, July 2002.

[23] A. Mennon, K. Mehrotra, C. K. Mohan, and S. Ranka, "Characterization of a class of sigmoid functions with applications to neural networks," *Neural Networks*, vol. 9, pp. 819–835, 1996.

[24] D. Hush and B. Horne, "Efficient algorithms for function approximation with piecewise linear sigmoidal networks," *IEEE Transactions on Neural Networks*, vol. 9, pp. 1129–1141, 1998.

[25] I. I. Sakhnini, M. T. Manry, and H. Chandrasekarn, "Iterative improvement of trigonometric networks," in *International Joint Conference on Neural Networks*, 1999.

[26] D. S. Broomhead and D. Lowe, "Multivariable functional interpolation and adaptive networks," *Complex Systems*, vol. 2, pp. 321–355, 1988.

[27] J. N. Lin and R. Unbehauen, "Canonical piecewise-linear networks," *IEEE Transactions on Neural Networks*, vol. 6, no. 1, January 1995.

[28] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, pp. 59–69, 1982.

[29] M. T. Manry, S. J. Apollo, L. S. Allen, W. D. Lyle, W. Gong, M. S. Dawson, and A. K. Fung, "Fast training of neural networks for remote sensing," *Remote Sensing Reviews*, vol. 9, pp. 77–96, 1994.

[30] H. H. Chen, M. T. Manry, and H. Chandrasekaran, "A neural network training algorithm utilizing multiple sets of linear equations," *Neurocomputing*, vol. 25, no. 1-3, pp. 55–72, April 1999.

[31] K. H. M. Stinchcombe and H. White, "Universal approximation of an unknown mapping and its derivatives using multilayer feed-forward networks," *Neural Networks*, vol. 3, no. 5, pp. 551 – 560, 1990.

[32] D. W. Ruck, "The multi-layer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, no. 4, 1990.

[33] Q. Yu, S. J. Apollo, and M. T. Manry, "Map estimation and the multi-layer perceptron," *Proceedings of the 1993 IEEE Workshop on Neural Networks for Signal Processing*, pp. 30–39, September 1993.

[34] M. R. Hestenes and E. Steifel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[35] H. Guo and S. B. Gelfand, "Analysis of gradient descent learning algorithms for multilayer feed-forward neural networks," in *Proceedings of the 29th IEEE Conference on Decision and Control*, vol. 3, Dec 1990, pp. 1751–1756.

[36] Y. Hirose, K. Yamashita, and S. Hijiya, "Back-propagation algorithm which varies the number of hidden units," *Neural Networks*, vol. 4, no. 1, pp. 61–66, 1991.

[37] A. F. Murray, "Analog vlsi and multi-layer perceptrons - accuracy, noise and on-chip learning," *Neurocomputing*, vol. 4, no. 4, pp. 301–310, 1992.

[38] A.-J. Annema, K. Hoen, and H. Wallinga, "Learning behavior and temporary minima of two-layer neural networks," *Neural Networks*, vol. 7, no. 9, pp. 1387–1404, 1994.

[39] Y.-J. Wang and C.-T. Lin, "A second-order learning algorithm for multilayer networks based on block hessian matrix," *Neural Networks*, vol. 11, no. 9, pp. 1607–1622, 1998.

[40] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks: Tricks of the Trade*, 1996, pp. 9–50.

[41] S. Raudys, *Statistical and Neural Classifiers: An Integrated Approach to Design*. Springer-Verlag, 2001.

[42] C. Yu, M. T. Manry, and J. Li, "Effects of nonsingular preprocessing on feeforward network training," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 19, no. 2, pp. 217–247, 2005.

[43] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, pp. 295–307, 1988.

[44] J. Moody and C. J. Darken, "Fast learning in networks of locally-tuned processing units," *Neural Comput.*, vol. 1, no. 2, pp. 281–294, 1989.

[45] F. M. Silva and L. B. Almeida, "Acceleration techniques for the backpropagation algorithm," in *Lecture Notes In Computer Science*, 1990, vol. 412, pp. 110–119.

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

[47] R. R. Selmic and F. L. Lewis, "Neural-network approximation of piecewise continuous functions: Application to friction compensation," *IEEE Transactions on Neural Networks*, vol. 13, no. 3, pp. 745–751, 2002.

[48] H. Lee, K. Mehrotra, C. Mohan, and S. Ranka, "An incremental network construction algorithm for approximating discontinuous functions," in *IEEE International Conference on Neural Networks*, ser. 27, vol. 4. IEEE, July 1994, pp. 2191–2196.

[49] F. J. Maldonado, M. T. Manry, and T.-H. Kim, "Finding optimal neural network basis function subsets using the schmidt procedure," in *Proceedings of the International Joint Conference on Neural Networks*, ser. 20-24, vol. 1, July 2003, pp. 444 – 449.

[50] A. J. Shepherd, *Second-Order Methods for Neural Networks*, ser. Perspectives in Neural Computing. Springer, 1997.

[51] J. Yeh, *Real Analysis: Theory of Measure and Integration*. World Scientific Publishing Company, Incorporated, 2006.

[52] W. H. Delashmit, "Multilayer perceptron structured initialization and separating mean processing," Dissertation, University of Texas at Arlington, December 2003.

[53] C. Yu, "Analyses and training of nonlinear networks with linear pre-processors," Dissertation, University of Texas at Arlington, December 2004.

[54] K. Rohani and M. T. Manry, "The design of multi-layer perceptrons using building blocks," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, 1991, pp. 497–502.

[55] W. Kaminski and P. Strumillo, "Kernel orthonormalization in radial basis function neural networks," *IEEE Transactions on Neural Networks*, vol. 8, no. 5, 1997.

[56] B. Iglewicz, *Understanding Robust and Exploratory Data Analysis.* New York: Wiley, 1983, ch. Robust scale estimators and confidence intervals for location.

[57] U. of Texas at Arlington, "Training data files."

[58] M. T. Manry, H. Chandrasekaran, and C. H. Hsieh, *Handbook of neural network signal processing.* CRC Press, 2001, ch. Signal Processing Applications of the Multilayer Perceptron.

[59] A. K. Fung, Z. Li, and K. S. Chen, "Back scattering from a randomly rough dielectric surface," *IEEE Transactions Geoscience and Remote Sensing*, vol. 30, no. 2, March 1992.

[60] U. C. Bureau, "[http://www.census.gov] (under lookup access [http://www.census.gov/cdrom/lookup]: Summary tape file 1)."

[61] U. of Toronto, "Delve data sets."

[62] U. of California Irvine, "Machine learning repository."

[63] R. Fletcher, *Practical Methods of Optimization*, 2nd ed. New York: Wiley Publications, 1987.

[64] R. Battiti, "First- and second-order methods for learning: Between steepest descent and newton's method," *Neural Computation*, vol. 4, pp. 141–166, 1992.

[65] "Source: http://wwwstls.frb.org/fred/index.html."

BIOGRAPHICAL STATEMENT

Sanjeev S. Malalur holds a PhD in Electrical Engineering from The University of Texas at Arlington. He obtained a Master's degree also from The University of Texas at Arlington in August 2004 and completed his Bachelor of Engineering in Electronics and Communications from the Bangalore University, India in November 2000. He has authored or co-authored several peer-reviewed publications. His current research interests include neural networks, image processing, machine learning, pattern recognition and biometrics. He is also a member of IEEE, INNS and SIAM.