A COMPARISON OF BACKPROPAGATION AND CONJUGATE GRADIENT ALGORITHMS FOR EFFICIENT TRAINING OF MULTI-LAYER PERCEPTRON NETWORKS

By

Guoping Miao

Bachelor of Science

Wuhan Urban Construction Institute

Wuhan, Hubei, P.R. of China

1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2002

A COMPARISON OF BACKPROPAGATION AND CONJUGATE GRADIENT ALGORITHMS FOR EFFICIENT TRAINING OF MULTI-LAYER PERCEPTRON NETWORKS

Thesis Approved:

J Chandler

Thesis Advisor

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

PREFACE

The popularity of the multi-layer perceptron neural network (MLP) has been growing rapidly and it is used in many real world applications. Although the backpropagation algorithm has successfully been used for training very many multi-layer perceptron networks, the traditional backprogation methods are considered inefficient. Recently there has been much work to let multi-layer perceptron networks take advantage of modern numerical optimization techniques. These non-linear optimization methods originated from about the early 1960s. A milestone of non-linear optimization was the publication of the paper by Fletcher and Powell. Since then, non-linear optimization technology has been extensively understood for about four decades and widely applied to both academic and industrial fields. Many numerical methods have been applied to train MLP networks. In real applications, global optimization, memory requirements, speed etc., are still problems, even though today computer technology provides the huge size of memory and capacity of high speed and parallel computing which make training MLP networks with non-linear optimization methods possible and efficient. Recently there are some new developments in the steepest descent family, such as effective backpropagation training with variable stepsize. This paper presents a comparison of backpropagation and conjugate gradient algorithms for efficient training of multi-layer perceptron networks. It will explore and analyze some related algorithms in this area and compare the different methods for speed and storage requirements.

I would like to express my appreciation to my major thesis advisor, Dr. J. P. Chandler, for his great insight and helpful guidance. When I was thinking about how good the "Efficient backpropagation training with variable stepsize" algorithm was, he said it should not be better than a good conjugate gradient method. This inspired me to choose the topic of this paper. Evidence shows he is correct no matter how steepest descent algorithms change their appearance. In addition, I appreciate his provision of software for numerical methods. I also thank Dr. B. E. Mayfield and Dr. N. Park for their assistance and time during this study.

TABLE OF CONTENTS

Chapter Page

1. INTRODUCTION.....1

1.1 General concept of neural networks.....1
1.2 Perceptron networks and their weakness.....2
1.3 Multi-layer perceptron networks and the backpropagation (BP) algorithm.....3

2. LITERATURE REVIEW AND PROBLEM STATEMENT.....5

2.1 Literature review.....5
2.1.1 Steepest descent algorithm.....5
2.1.2 The backpropagation algorithm.....5
2.1.3 Drawbacks of the BP algorithm.....7
2.1.4 Improvement of BP algorithm.....8
2.1.5 Numerical optimization methods.....8
2.2 Problem statement.....13
2.2.1 Motivation and objective.....13
2.2.2 Organization of this paper.....13

3. ANALYSIS OF THE STANDARD BP ALGORITHM.....15

4. BACKPROPAGATION WITH VARIABLE STEPSIZE (BPVS) ALGORITHM.....18

4.1 The BPVS algorithm.....18
4.2 The drawbacks of the BPVS algorithm.....21

5. NUMERICAL OPTIMIZATION AND CONJUGATE GRADIENT ALGORITHMS.....25

5.1. Line search.....25
5.1.1 Linear search.....25
5.1.2 Non-linear search and the golden section method.....26
5.1.3 Interval location.....27
5.1.4 Interval reduction.....28
5.2. Conjugate gradient method.....29
5.3. The conjugate gradient algorithm for MLP training.....31

Chapter Page

6. CASE STUDIES AND PROGRAM DESCRIPTION 35

6.1. The testing problems 35
6.1.1 The network design..................................35
6.1.2 The code of steepest descent-based algorithms..................................36
6.1.3 The existing conjugate gradient and quasi-Newton methods..................................37
6.2. Numerical examples..................................37
6.3. Components of comparison..................................39

7. RESULT ANALYSIS 40

7.1. Results..................................40
7.2. Result analysis and discussion..................................43

8. CONCLUSIONS AND FUTURE WORK..................................45

REFERENCE..................................47

NOTATIONS

1. $E(\mathbf{w})$	sum of mean square function
2. $\mathbf{W}^m$	weight vector of mth layer of MLP
3. $F(\mathbf{x})=\frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{d}\mathbf{x}+\mathbf{c}$	quadratic objective function
4. $\mathbf{v}$	transfer function (activate function)
5. $\nabla F(\mathbf{x})=\mathbf{H}\mathbf{x}+\mathbf{d}$	gradient of $F(\mathbf{x})$
6. $\nabla^2 F(x)=\mathbf{H}$	Hessian matrix of $F(x)$
7. $f(\mathbf{x})$	general continuous objective function
8. $\mathbf{g}_k=\nabla f(\mathbf{x})=\begin{bmatrix}\partial f/\partial x_1\\ \partial f/\partial x\\ \vdots\\ \partial f/\partial x_n\end{bmatrix}$	gradient of iteration k
9. k	number of iteration
10. $\mathbf{p}_k$	search direction
11. α_k	learning rate (step size) in iteration k
12. β_k	Fletcher and Reeves coefficient
13. $\mathbf{s}$	sensitivity
14. $\mathbf{x}_0$	starting point
15. $\mathbf{x}^*$	minimum point
16. $\mathbf{t}$	target vector
17. $\mathbf{a}$	output vector
18. $\mathbf{b}$	bias
19. $\mathbf{n}$	net input—sum of weighted input
20. $\mathbf{r}$	input
21. Q	number of inputs in a training set
22. N	number of output neurons
23. λ	coefficient of Levenberg-Marquardt method
24. L_k	Lipschitz constant

CHAPTER 1

INTRODUCTION

1.1. What are artificial neural networks?

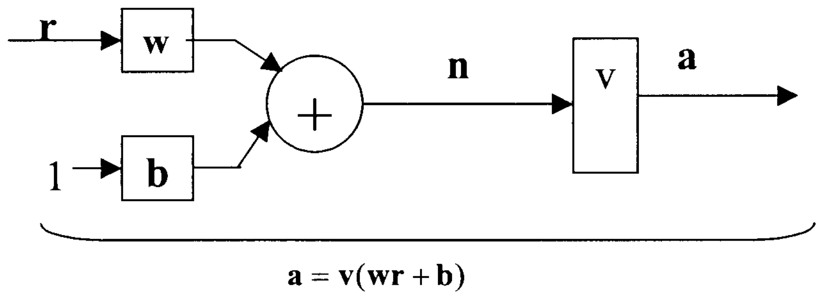
An artificial neural network (ANN) is a mathematical model, a family of parameterized functions for fitting data. The most fundamental component of ANNs is a large number of interconnected artificial neurons, which are modeled to some extent after the structure of human brain. Biological neurons and the multiple connections between them are an integral part of brain function. The structure of biological neurons is very complicated. The ANNs try to simulate only the most basic functions of the biological neurons. There are two major similarities between biological and artificial neural networks, the building blocks (processing units) and the connections (functions) between the units [1][2].

The history of ANNs is a legacy. Some fundamental and conceptual work for the field of networks appeared in the late 19th and early 20th centuries [2]. It's hard to tell who is the father of this technology. Many creative individuals from various fields contributed to the foundation of ANN. The field has grown rapidly since the beginning, but there were setback in 1960s and 1970s due to lack of technical support. This will be explained in section 2.3. In the1980s, research in neural networks recovered and increased surprisingly.

Now ANNs have been widely used in many fields, including electronics, manufacturing, medical, financial, engineering, etc. ANNs are mostly applied in prediction, classification, data association, data conceptualization, and data filtering. The computing mechanism of ANNs associated with learning rules is different from traditional methods. ANNs are considered as an important part of the advanced generation of modern computing technology. There are three principal kinds of neural networks, perceptrons, Hamming networks, and Hopfield networks. Hamming and Hopfield networks are not discussed in this paper.

1.2. Single perceptron networks and their weakness

The perceptron network is the first application of ANN. Rosenblatt introduced the perceptron network in the late 1950s [3][39]. The single perceptron network with its learning rule indicates the potential ability of ANNs to solve classification and pattern recognition problems. Although it's a simple model, it built the foundation of ANNs and triggered a great deal of interest in ANN research. The following is an abbreviated notion of a single perceptron network:

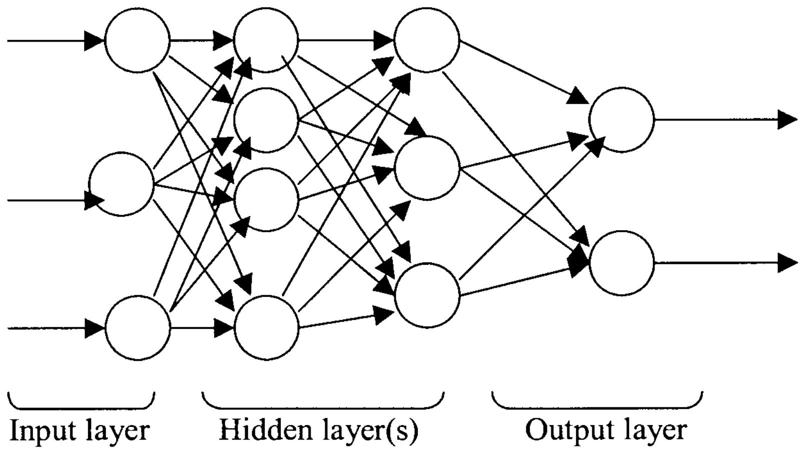


The inputs and outputs are represented by vectors **r** and **a,** respectively. Each of the inputs r_i is multiplied by a connection weight w_i. The summed products plus the bias, denoted as the net inputs (**n**), are fed to the transfer function (**v**) to generate the desired outputs.

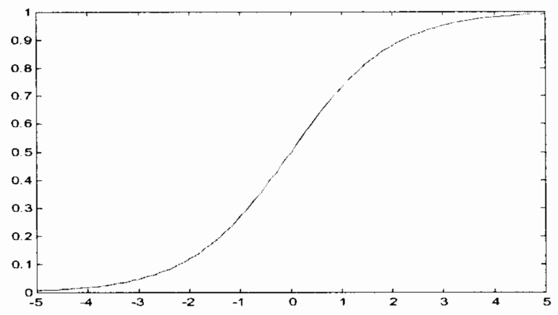
In 1960s, Widrow and Hoff introduced the ADALINE network, and a learning rule they called the LMS (Least Mean Square) algorithm [4]. Unfortunately, it was later shown that the single perceptron network could solve only linearly separable classification problems [6]. Both Rosenblatt and Widrow intended to overcome this problem and proposed multi-layer perceptron networks [4][34].

1.3. Multi-layer perceptron and the backpropagation (BP) algorithm

The structure of a multi-layer perceptron network is a combination of single perceptrons. Generally each layer takes the output of previous layers as its input and gives its output as the input to the next layer. We will take Hagan's abbreviated notation, such as $R\text{-}S^1\text{-}S^2\ldots S^n$, to represent an n-layer MLP network [2]. The following graph is an example of a four-layer, 3-4-3-2, perceptron network:



A multi-layer perceptron network contains one input layer, one output layer, and as many hidden layers as desired. One or two hidden layers are commonly used. The transfer functions could be any kind of mathematical functions. The multi-layer perceptron overcome the limitation of single perceptrons, which had no hidden layers. It can be used to solve very complicated problems. Hornik, Stinchcombe, and White have shown that three-layer networks with sigmoid transfer functions in the single hidden layer and linear transfer functions in the output layer can approximate any function to any degree of accuracy, if the hidden layer has a sufficient number of processing units [5]. The following graph depicts the sigmoid transfer function. Later in Chapter 5, I'll apply this network model to several test problems.



Sigmoid function $f(x) = 1/(1+e^{-x})$

Rosenblatt and Widrow just presented the idea of multi-layer perceptron networks. Regrettably, they were not able to show the power of this neural device since they didn't have any algorithms to train MLP networks. Unfortunately neural network research was stuck for a time. Many people doubted the capacity and the future of the ANN. They quit their research on neural networks and went to other areas [6].

In the1980s, the appearance of the backpropagation algorithm by David Rumelhart, Geoffrey Hinton and Ronald Williams [7] was a breakthrough in ANN research. ANNs stepped into a new stage. The backpropagation algorithm significantly affected the whole neural network world. Many methods based on the backpropagation algorithm have been developed in the last two to three decades.

CHAPTER 2

LITERATURE REVIEW AND PROBLEM STATEMENT

2.1 Literature review

2.1.1 Steepest descent algorithm

A very important part of the ANNs is to find algorithms to train the neural networks, in other words, to optimize the error functions of the networks iterately. One basic method is called steepest descent. Suppose there is an error function $f(\mathbf{x})$. We wish to find a value of $\mathbf{x}$ which minimizes $f(\mathbf{x})$. We start from a chosen initial guess, $\mathbf{x}_0$, and then update $\mathbf{x}_0$ in stages according to the equation

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{p}_k, \tag{2.1}$$

where α is a positive scalar (the learning rate), and $\mathbf{p}_k$ is the search direction. For our objective, we must follow the direction which makes the value of $f(\mathbf{x})$ smaller. There are many such directions. The direction in which the function decreases most rapidly is the negative of the gradient. Therefore a vector which points to the steepest descent direction is

$$\mathbf{p}_k = -\mathbf{g}_k. \tag{2.2}$$

Combining this to equation (2.1) produces the method of steepest descent

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \mathbf{g}_k$$

2.1.2 Standard backpropagation (BP) algorithm

The multi-layer perceptron networks were not able to show their potential ability without a good training algorithm until the discovery of the backpropagation (BP) algorithm. The BP algorithm was a breakthrough in training multi-layer networks. It is a generalization of the LMS (Least Mean Square) algorithm, an approximated steepest descent algorithm, in which an estimated gradient is used. In most perceptron applications, we choose to minimize the squared error,

$$E = (\mathbf{t}(k) - \mathbf{a}(k))^T (\mathbf{t}(k) - \mathbf{a}(k)) = \mathbf{e}^T(k)\mathbf{e}(k) \tag{2.3}$$

where **a** is a vector of outputs from the ANN and **t** is a vector of corresponding target outputs (output data). k is an index to a particular exemplar (data point) consisting of an input vector and an output vector $\mathbf{t}(k)$.

The approximate steepest descent algorithm is as follows:

$$\mathbf{w}_{k+1}^m = \mathbf{w}_k^m - \alpha \, \partial\mathbf{E} / \partial\mathbf{w}_k^m \tag{2.4}$$

$$\mathbf{b}_{k+1}^m = \mathbf{b}_k^m - \alpha \, \partial\mathbf{E} / \partial\mathbf{b}_k^m \tag{2.5}$$

where as before **w** is the weight vector and **b** is the vector of biases.

By applying the chain rule, we obtain

$$\mathbf{w}_{k+1}^m = \mathbf{w}_k^m - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T \tag{2.6}$$

$$\mathbf{b}_{k+1}^m = \mathbf{b}_k^m - \alpha \mathbf{s}^m \tag{2.7}$$

Here α is the learning rate, $\mathbf{s}^m = \partial\mathbf{V} / \partial\mathbf{n}^m$ is the sensitivity of **V** to a change in the net input at layer m, **V** is a vector of transfer functions, and **n** is a vector of net inputs (sum of weighted inputs).

Now by using other applications of the chain rule, we obtain equations (2.8) and (2.9) for the hidden layers and the last layer, respectively, and can compute the sensitivity of layer m:

$$\mathbf{s}^m = \dot{\mathbf{V}}^m(\mathbf{n}^m)(\mathbf{w}^{m+1})^T\mathbf{s}^{m+1} \tag{2.8}$$

$$\mathbf{s}^M = -2\dot{\mathbf{V}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}) \tag{2.9}$$

where $\dot{\mathbf{V}}$ is the first order derivatives of the transfer functions. To compute derivative of the mth layer's transfer function, we only need the weights and the sensitivity of the (m+1)th layer. The term *backpropagation* originated from this process of backward propagation of derivatives through the MLP.

BP needs one forward and one backward calculation in each iteration to update the weights and biases. Obviously, it is very straightforward and easy to calculate. This idea, along with the availability of powerful new computers, also makes scaled parallel computing possible [8].

Multi-layer perceptron networks trained with the backpropagation algorithm broke the limitation of one-layer perceptrons [7]. MLP can solve the XOR problem that the simple perceptron could not, and are the most popular form of ANN being applied today.

2.1.3 The drawbacks of the backpropagation algorithm

Despite BP's effectiveness, many researchers found this algorithm's rate of convergence was too slow for the technique to be used practically. "Although BP training has been proven to be efficient in many applications, it uses a constant stepsize, its convergence tends to be very slow and it often yields suboptimal solutions" [9]. Many other researchers believed this, and tried to make some progress on it (detailed in the next section). Li Zhang's paper [11] showed some solid examples in which the backpropagation algorithm [8] had trouble in some cases for multi-layer perceptron networks in function approximation problems. He further mentioned that BP training was very slow and sometimes had convergence problems. Due to its fixed learning rate, the training speed is limited. Why does this happen? Basically, the BP algorithm does not have a sound theoretical basis and can be inefficient and unreliable [12]. This problem will be discussed in detail in Chapter 3.

2.1.4 Improvement of the BP algorithm

During the last three decades, many researchers made efforts to overcome this kind of problem. Some progress has been made. The major two early advances were BP with momentum [13], and BP with adaptive learning rates [14]. For BP with momentum, a momentum coefficient γ, $(0 \leq \gamma \leq 1)$, is used to smooth out the oscillation of the trajectory when a larger value of learning rate α is applied. Adaptive BP tries to adjust the learning rates through different regions. If the error decreases, the learning rate is increased; if the error increases, the learning rate is reduced. But BP with momentum and BP with adaptive learning rates are not reliable due to the use of the heuristic factors. The most recent achievement is BP with variable stepsize (BPVS) [9]. It achieves the basic requirement of dynamic tuning without any heuristic factors.

2.1.5 Numerical optimization methods

Unfortunately, the improvement of BP is very limited [2][12][15][16]. In the last decade, numerical optimization techniques have been successfully applied to train multi-layer perceptron networks. These techniques include Newton's method (NT), the Gauss-Newton method (GN), the Levenberg-Marquardt method (LM), and conjugate gradient methods (CG) [21]. All of these methods significantly improved the speed of multi-layer perceptron training in different aspects, but generally speaking, each of them has advantages and disadvantages when compared to the others.

For the function $f(\mathbf{x})$ of section 2.1.1, we want to find a local minimum efficiently (Finding the global minimum is a difficult problem that can be attacked by carrying out local minimization from many pseudorandom starting points.). Based on the second-order Taylor series, Newton's method starts from point x_0 and uses the second-order Taylor series expansion of $f(\mathbf{x})$ at x_0. This yields a quadratic approximation $F_0(\mathbf{x})$ of $f(\mathbf{x})$ about x_0. Newton's method always reaches the minimum of a quadratic function in one step. If the function, $f(\mathbf{x})$, is quadratic, we have the following equation to find the minimum:

$$\mathbf{H}(\mathbf{x}_{k+1} - \mathbf{x}_k) = -\mathbf{g}_k \tag{2.10}$$

where **H** is the Hessian, the matrix of second derivatives of $f(\mathbf{x})$ with respect to **x**,

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} f(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} f(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} f(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} f(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} f(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} f(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} f(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2} f(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_n^2} f(\mathbf{x}) \end{bmatrix}$$

The step vector $\Delta\mathbf{x} = \mathbf{x}_{k+1} - \mathbf{x}_k$ is computed by solving the linear system $\mathbf{H}\Delta\mathbf{x} = -\mathbf{g}$. We don't need to compute the inverse of the Hessian matrix.

If $f(\mathbf{x})$ is not quadratic, Newton's method computes a sequence of estimates that may lead toward the minimum. The minimum point x_1 of $F_0(\mathbf{x})$ is supposed to be the next estimate of the minimum point of $f(\mathbf{x})$. Repeating this process yields successive estimates $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3} \ldots$, which should gradually approach the minimum point of $f(\mathbf{x})$, and, if all goes well, finally converge to that point with the desired accuracy. Newton's method therefore can be considered as a method based on successive minimization of the quadratic functions $F_0(\mathbf{x}), F_1(\mathbf{x}), F_2(\mathbf{x}), F_3(\mathbf{x}), \ldots$, each of which is an approximation of $f(\mathbf{x})$.

Newton's method is known as a fast method, but it has two big problems. First, unlike the method of steepest descent, convergence is not guaranteed for Newton's method unless the starting point is sufficiently near a local minimum of $f(\mathbf{x})$. It could possibly oscillate, or even diverge [2]. Second, as indicated in the Eq. (1.1), we need to compute the Hessian matrix in each iteration. For low or moderate dimensional variable space problems, this is a good tradeoff, but it is worse for high dimensionality, since it requires a lot of computation and $O(n^2)$ storage (n is the number of variables over the variable space, that is, the number of components in the **x** vector.) . Unfortunately, the number of parameters involved in a multi-layer perceptron network is sometimes very large. Often hundreds to thousands of weights and biases are required. This makes Newton's method impractical in a large scale MLP.

The Gauss-Newton method can be developed from Newton's method if the function is a sum of squares function as in equation (2.3):

$$\mathbf{E}(\mathbf{x}) = \sum_{i=1}^{N} e_i^2(\mathbf{x}) = \mathbf{e}^T(\mathbf{x})\mathbf{e}(\mathbf{x}) \tag{2.11}$$

Newton's method then would be:

$$[\nabla^2 \mathbf{e}(\mathbf{x}_k)](\mathbf{x}_{k+1} - \mathbf{x}_k) = -\nabla \mathbf{e}(\mathbf{x}_k) \tag{2.12}$$

It can be shown that

$$\nabla \mathbf{E}(\mathbf{x}) = \mathbf{J}^T(\mathbf{x})\mathbf{e}(\mathbf{x}) \tag{2.13}$$

$$\nabla^2 \mathbf{E}(\mathbf{x}) = \mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + \mathbf{S}(\mathbf{x}) \tag{2.14}$$

Where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_1(\mathbf{x})}{\partial x_1} & \frac{\partial e_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial e_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial e_2(\mathbf{x})}{\partial x_1} & \frac{\partial e_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial e_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_N(\mathbf{x})}{\partial x_1} & \frac{\partial e_N(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial e_N(\mathbf{x})}{\partial x_n} \end{bmatrix} \tag{2.15}$$

and

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} e_i(\mathbf{x})\nabla^2 e_i(\mathbf{x}) \tag{2.16}$$

For the Gauss-Newton method, the term **S(x)** is dropped. Generally, that's because if function E is expanded at a local optimum point $\mathbf{x}^*$ for those points sufficiently close to $\mathbf{x}^*$, $\mathbf{S}(\mathbf{x}) \cong 0$. However there do exist so-called "large residual least squares problem" in which S(x) is not small compared to $\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$, and for which Newton's method has a neighborhood of convergence but the Gauss-Newton method has none. In addition, the Gauss_Newton method converges only linearly, if it converges at all, while Newton's method converges quadratically. Liya Wang discussed this in more detail [15]. Then the Gauss-Newton method is:

$$\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{J}^T(\mathbf{x}_k)\mathbf{e}(\mathbf{x}_k) \tag{2.17}$$

The advantage of the Gauss-Newton method over the standard Newton's method is that it does not require the calculation of the second order derivative in **S(x)** (the

Hessian matrix **H**). It is less costly in computations. But the drawback of Newton's method that needs O(n^2) storage still remains for the Guass-Newton method. Also,there is something to worry about when we solve the linear system in (2.17), since the solution may not be unique. This problem can be overcome by the Levenberg-Marquardt method.

The Levenberg-Marquardt method is a modification of the Gauss-Newton method. In case $\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$ is singular, the Gauss-Newton method can be changed to equation (2.18) [13].

$$[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \lambda\ diag(\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k))](\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{J}^T(\mathbf{x}_k)\mathbf{e}(\mathbf{x}_k) \qquad (2.18)$$

where diag() is the diagonal part of a matrix and the parameter λ is an adjustable coefficient which avoids the problem if $\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$ is not invertible. λ is multiplied by some factor (ν>1) whenever $E(\mathbf{x})$ increases and divided by ν whenever $E(\mathbf{x})$ decreases. The Levenberg-Marquardt method removes some deficiencies of the Gauss-Newton method. The key procedure is to calculate the Jacobian matrix in which all terms are first derivatives of the error function with respect to each variable. This can be done by a simple modification to the standard backpropagation algorithm [16].

Both the Gauss-Newton (GN) and Levenberg-Marquardt (LM) backpropagation algorithms eliminate the calculation of second order derivatives, but they still need O(n^2) storage for the matrix $\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x})$. Although the rate of convergence of Newton's method, the Gauss-Newton and Levenberg-Marquardt methods are quite rapid in most cases, they do require large storage compared to conjugate gradient (CG) methods, which do not have to approximate the Hessian matrix or the Jacobian matrix. The CG methods have an obvious advantage over the Newton, Gauss-Newton and Levenberg_Marquardt methods in that there is a much lower storage requirement when we deal with a very large number of weights and biases, O(n) versus O(n^2).

2.2 Problem statement

2.2.1 Motivation and objective

The steepest descent based methods and the modern numerical optimization methods (Newton, GN, LM, and CG) are two different computing categories for training MLP networks. Due to the advantages of the conjugate gradient method against the other numerical methods (section 2.1.4), we would like to know if the performance of CG is better than BP. In addition, Magoulas, Vrahatis, and Androulakis proved that BPVS is much more efficient than BP [9]. I expect to prove that CG also has better performance than BPVS. This paper will focus on comparing the BP and BPVS algorithms with the CG algorithm. The 'restart method' proposed by Powell will be applied to the CG algorithm to make the CG method perform better. The purpose of this paper is to explore the theory and algorithms that train multi-layer perceptron networks, estimate the advantages and disadvantages of the various methods, and provide evidence and direction for future work.

2.2.2. Paper organization

Chapter 1 has already briefly reviewed some concepts of artificial neural networks, what multi-layer perceptron networks are suppose to do and why backpropagation is prevalent. These are the basics of the following chapters. Chapter 2 talks about the current research, and indicates the problem we are going to focus on.

Chapter 3 provides a detailed discussion about why the standard BP algorithm is not an efficient algorithm for MLP training.

In Chapter 4, the BPVS algorithm, and its limitations will be introduced.

Then I'll describe in Chapter 5 the CG algorithm in detail and discuss why CG is better than BP.

In Chapter 6, five different examples (the XOR problem, two function approximation problems from Li Zhang's M.S. thesis, and two real problem simulations from the UCI repository) will be implemented by computer programs. This provides the numerical data for Chapter 7, which compares and analyzes those data to make a decision.

Finally, I'll draw conclusions in Chapter 8 and predict some future work.

CHAPTER 3

ANALYSIS OF THE BACKPROPAGATION ALGORITHM

The backpropagation algorithm is the cornerstone of training MLP networks. However, many people believe it is not a good algorithm. As previously mentioned, the BP algorithm moves a suitable distance along the negative gradient in each iteration to decrease the error *E*. The intrinsic nature of steepest descent, which is also called gradient descent, determines the behavior of the BP algorithm. In the batch version of BP, which is just steepest descent, we start with some initial guess for the weight vector (which is often chosen pseudorandomly near the origin) denoted by $\mathbf{w}_0$. We then go downhill and consequently update the weight vector. The direction of the greatest rate of decrease for the error is the direction of the negative gradient. Therefore, we move a short distance along the negative gradient, at iteration k, evaluated at $\mathbf{w}_k$:

$$\Delta\mathbf{w}_k = -\alpha\, \partial\mathbf{E}/\partial\mathbf{w}_k \tag{3.1}$$

the coefficient α is called the learning rate. If the value of α is sufficiently small, the value of error E will decrease in each iteration, finally leading to a weight vector at which the following condition is satisfied:

$$\Delta\mathbf{E} \cong 0 \tag{3.2}$$

The problem is that, if the value of α is too small, it will take a long time (many iterations) to converge. We need to speed up in order to make the procedure efficient.

However, if α is too large, oscillations will occur, and the algorithm may overshoot, leading to an increase in E and possibly to divergence. The following figures illustrate the problem in an error surface E for a two-dimensional weight space. Figure 3.1 depicts the trajectory with small α. In Figure 3.2, when the value of α is too large, the trajectory gets unstable.

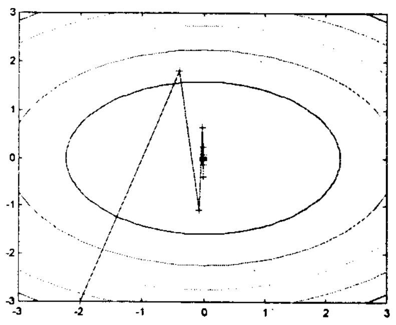


Figure 3.1

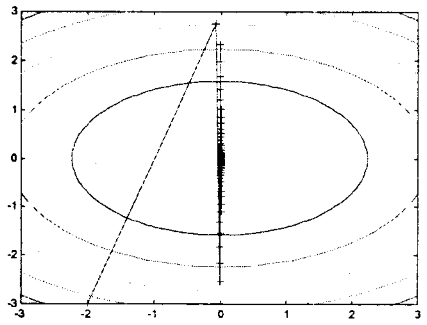


Figure 3.2

A little more should be mentioned here. In Figure 3.3, the contour line of a quadratic function is very elliptical. In other words, the eigenvalues of the Hessian matrix of E are much different, $\lambda_{max} >> \lambda_{min}$. The BP algorithm has to take many small steps to move to the minimum. (More will be discussed in Chapter 4.)

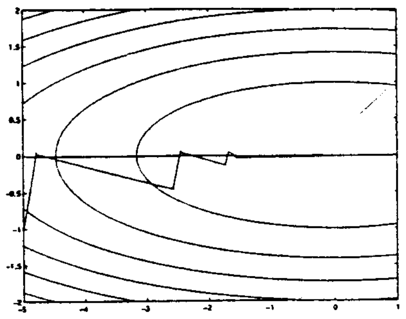


Figure 3.3

In the general case, the error functions are not quadratic. The shape of the error surface might be very contorted. Hagan shows an example in his textbook [2] in which the error function has only two parameters (Figure 3.4).

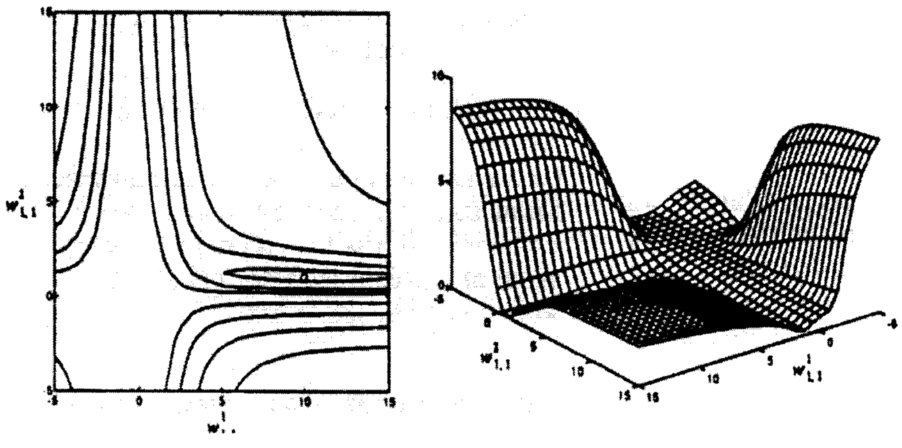


Figure 3.4

In this example, we have to choose a very small α to avoid oscillation in the steep valley, but this will take an extremely long time to go through the relatively flat region. Obviously, the BP algorithm is not able to solve this kind of problems efficiently. The ideal algorithm is to have a small α when the error surface is very steep; a large α, otherwise.

CHAPTER 4

BACKPROPAGATION WITH VARIABLE STEPSIZE ALGORITHM

4.1 Backpropagation with variable stepsize (BPVS) algorithm

The standard BP algorithm is not able to fulfill the requirement that the different performance index surface regions should have different learning rates. Many researchers tried to look for new approaches to improve the BP algorithm. Many papers have been published in recent years. The effective backpropagation training with variable stepsize algorithm [9] is a new achievement of a steepest descent based algorithm. The algorithm shows good performance without any heuristic factors and satisfies the basic requirement for MLP network training.

The BPVS method is a modified steepest descent algorithm. The idea of BPVS is to tune the learning rate dynamically in each iteration. Its convergence is guaranteed by applying estimates of the Lipschitz constant, obtained without additional error function and gradient evaluations [9].

When we train the network through a training set, we get the performance index:

$$E = \frac{1}{2}\sum_{r=1}^{R}\sum_{j=1}^{N}(a_r - t_r)^2 = \sum_{r=1}^{R} E_r \tag{4.1}$$

where $(a-t)^2$ is the squared difference between the actual output value a_r at the j^{th} output layer neuron for pattern r and the target output value t_r, and r is an index over input-output pair exemplars.

The BPVS algorithm is based on three assumptions regarding the error surface E [ref.9]. For simplicity, I omit the assumptions here (it does not further affect our discussion). We need to keep in mind that equation (4.2) must be satisfied.

$$\| \nabla E(\zeta) - \nabla E(\gamma) \| \leq K \| \zeta - \gamma \| \tag{4.2}$$

In the above equation, K is the Lipschitz constant. $\zeta, \gamma \in \mathbf{w}$, $\mathbf{w}$ is in the region $\chi(\mathbf{w}_0)$ which contains w_0, and all $E(\mathbf{w}) < E(\mathbf{w}_0)$.

We can apply theorem 1 of Armijo [9] to obtain the new weight. Then we update the equation using the Lipschitz constant.

$$\mathbf{w}_{k+1} = \mathbf{w}_k - 0.5K^{-1}\nabla E(\mathbf{w}_k),\ k = 0,1,2.... \tag{4.3}$$

where $\nabla E(\mathbf{w}_k) = \sum_{r=1}^{R} \nabla E_r(\mathbf{w}_k)$ (4.4)

This is useless because the constant $0.5K^{-1}$ is unknown before each iteration. For our purpose, we need a large $0.5K^{-1}$ to speed up convergence. But, if $0.5K^{-1}$ is too large, convergence cannot be guaranteed. To solve this problem, we always use a small stepsize at the beginning and then tune it in each epoch. This can be done by the steps based on Armijo's theorem.

Armijo's Theorem 2 [9]. Suppose that η_0 is an arbitrary assigned positive number and consider the sequence $\eta_m = \eta_0 2^{1-m}$, m=1,2.... Then the sequence of weight vectors $\{w_k\}_0^\infty$ defined by

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_{mk} \nabla E(\mathbf{w}_k), k = 0,1,2..... \tag{4.5}$$

where m_k is the smallest positive integer for which:

$$E(\mathbf{w}_k - \eta_{m_k} \nabla E(\mathbf{w}_k)) - E(\mathbf{w}_k) \le -\frac{1}{2}\eta_{m_k} \| \nabla E(\mathbf{w}_k) \|^2 \tag{4.6}$$

converges to the point w^* which minimizes the error function E.

At this point, the BPVS algorithm has been outlined. We need to make it practical to solve MLP training problems.

As mentioned before, BPVS is a modification of the steepest descent algorithm. At each iteration of the steepest descent procedure, the values of the weights are modified in the direction in which the error function, E, decreases most rapidly. Along with the direction, $-\nabla E(\mathbf{w}_k)$, BPVS uses a local approximation of the Lipschitz constant L_k to estimate the stepsize $0.5\,k^{-1}$ at each epoch. Recall the assumption we made before (equation (4.2)). We obtain the Lipschitz constant L_k as follows:

$$L_k = \| \nabla E(\mathbf{w}_k) - \nabla E(\mathbf{w}_{k-1}) \| / \| \mathbf{w}_k - \mathbf{w}_{k-1} \| \qquad (4.7).$$

This overcomes some drawbacks of Armijo's theorem 2 and reflects all the local information regarding the direction and the stepsize. But this does not mean that Armijo's theorem 2 could be ignored. We still need to apply Armijo's theorem 2 in some circumstances to guarantee convergence. The main idea is that if the stepsize $0.5\,L_k^{-1}$ is very small, we should increase the stepsize by doubling it; on the other hand, if the stepsize $0.5\,L_k^{-1}$ is too long and the successive steps in weight space do not satisfy equation (4.6), then we need to decrease the stepsize. This is the elegant point of the BPVS algorithm. It's very simple, but powerful in two aspects. First, Armijo's theorem

2 guarantees convergence. Second, $0.5\,L_k^{-1}$ is sensitive to the local shape of the error function. If it needs to, the stepsize speeds up by doubling itself. This behavior is different from other algorithms, such as the standard BP which has a fixed stepsize, or adaptive BP which increases the stepsize using some heuristic factors. BPVS is especially helpful when training with a very flat error surface region.

The BPVS algorithm is summarized as following:

Initialization: Set the epoch k=0, the weights w_0 to real pseudorandom values, the stepsize to a small value η_0, the error tolerance *tol*, the minimum stepsize $\lambda_{\min}$, the Lipschitz constant L_k=1, the number of tuning $t_k = 1$.

Step 1: Compute the error E_r and the gradient of E_r for all input-output pairs through the training set, $r \in [1, R]$. Compute the local approximation L_k of the Lipschitz constant, according to Eq (4.7). Compute $\eta_0 = 0.5 L_k^{-1}$. If $\eta_k > \lambda_{\min}$, go to the next step; otherwise set $t_k = t_k + 1$, $\eta_k = \eta_k 2^{t_k - 1}$ and go to the next step.

Step 2: If Eq. (4.6) holds, set $m_k = 1$ and go to step 4; otherwise, set $m_k = m_k + 1, t_k = 1$, and go to the next step

Step 3: Set $\eta_k = \eta_0 2^{1-m_k}$ and return to Step 2

Step 4: Update weights according to $\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla E(\mathbf{w}_k)$, where

$$\nabla E(\mathbf{w}_k) = \sum_{r=1}^{R} \nabla E_r(\mathbf{w}_k)$$

Step 5: If $E(\mathbf{w}_{k+1}) > tol$, set k= k+1, go to Step 1; otherwise

stop

4.2 Drawbacks of BPVS algorithm

The BPVS algorithm is not good enough yet. Although the performance of the BPVS algorithm has been proven to be much better than standard BP, BP with momentum, and adaptive BP [9], it is still a steepest gradient-based algorithm. The dynamic tuning approach follows the negative gradient direction to adjust the learning rate in each iteration. So, the BPVS algorithm has a more suitable learning rate than the other methods (constant learning rate, or learning rate with heuristic factors, etc.). This learning procedure is similar to steepest descent with a line search (SDLS). First, both of them follow the negative gradient direction in each iteration. Second, following the specific direction, they try to make the stepsize as large as possible. The ideas are same. But the approaches they take are totally different. SDLS uses a line search method which looks for the minimum point in each direction. We'll talk about this line search method in detail in Chapter 5. BPVS, on the other hand, evaluates the local information of direction and stepsize to estimate the optimal stepsize. This approximation is based on Armijo's theorems to guarantee convergence. It's hard to say which method is more efficient now. Basically they should have similar rates of convergence. I'll implement BPVS and SDLS on various examples to see their performance. SDLS could have more additional error function evaluations than BPVS, but only by a constant factor, the average number of evaluations per line search, which is rarely large.

Broadly speaking, the tuning behavior of BPVS and SDLS is very negative. The critical reason is the learning directions. Both BPVS and SDLS cannot avoid the limitation of the direction of the negative gradients. Return to the example we discussed in the last chapter. In Figure 3.3, the contour line is very elliptical. At most points on the performance index surface, the local gradient does not point directly toward the minimum point. The training procedure contains many small steps. Certainly, it is not an efficient way if we move along this trajectory. This is not a special case. Any general function could be locally approximated as a quadratic function. And if the curvature of this approximating quadratic function varies greatly with direction, the convergence will be very slow in that region. This essentially affects the entire performance.

There is another area in which the BPVS algorithm as given in [9] is deficient. That is the convergence criterion.

The terminating condition is a very important aspect of training neural networks. One idea that is often used is to stop the training when the value of error function is less than or equal to a given tolerance:

$$E(\mathbf{w}_{k+1}) > tol \tag{4.8}$$

where w is the vector of variables. The stopping criteria of BPVS algorithm falls into this category:

Step 5 : If $E(\mathbf{w}_{k+1}) > tol$, then continue iterating

This is a very poor idea, though. Ordinarily the user has no idea how large *tol* should be, especially for a large-scale problem, and simply wants to go to a local minimum of $E(\mathbf{w})$. A criterion such as Step 5 would never be used by a professional in the field of optimization. Instead, an absolute criterion on the step size

$$\| \mathbf{w}_{k+1} - \mathbf{w}_k \| \leq abstol \tag{4.9}$$

would be used, or a relative criterion:

$$\| \mathbf{w}_{k+1} - \mathbf{w}_k \| \leq reltol * \| \mathbf{w}_k \| \tag{4.10}$$

or a combination of the two:

$$\| \mathbf{w}_{k+1} - \mathbf{w}_k \| \leq abstol + reltol * \| \mathbf{w}_k \| \tag{4.11}$$

where , *abstol* and *reltol* stand for absolute and relative tolerance, respectively. The expression (4.11) is the most general one. If *abstol* is equal to zero, the formula is essentially a variation of (4.10). Under this circumstance, we say if the relative change in each component of the vector w is less or equal to reltol on any iteration, then convergence is assumed. Later in the testing program, this criteria will be applied. Similarly, formula (4.9) is defensible. But (4.8) is irretrievably worthless.

We have to reconsider the learning direction of the procedure in order to eliminate the limitation of the steepest gradient direction. It would be nice if most of the points on the error surface have directions as close to the local minimum as possible.

CHAPTER 5

NUMERICAL OPTIMIZATION AND CONJUGATE GRADIENT ALGORITHMS

5.1 Line search

Because the line search concept forms the basis for the conjugate gradient algorithm, let's talk about it first in this section. To train a multi-layer perceptron network, we just take a sequence of steps through the weight space. A good algorithm should consider two aspects of each of these steps. One is the direction in which we are going to move along, and the other is the pace we move in that direction. Both of these must be optimal or nearly optimal for an efficient learning algorithm. With the steepest descent backpropagation algorithm, the direction of each step is determined by the local negative gradient of the error function, and the step size is given by an arbitrary learning rate parameter (either constant or with heuristic factors). For the conjugate gradient algorithm, we need to reconsider both aspects. The line search method is applied in determining the value of the learning rate. The concept of line search comes from the procedure in which, for a particular search direction in weight space, we find the minimum of the error function along that direction.

5.1.1 Line search

Suppose that at step k in some algorithm the current weight vector is $\mathbf{w}_k$, and we consider a particular search direction $\mathbf{p}_k$ through weight space. The optimal value for the weight vector along the search direction is then given by the expression

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k \tag{5.1}$$

where the parameter α_k is chosen to minimize

$$E(\alpha_k) = E(\mathbf{w}_k + \alpha_k \mathbf{p}_k) \tag{5.2}$$

This means once we have chosen the search direction, we could get the minimum point (and also set the optimal stepsize) by evaluating the value of the error function $E(\alpha_k)$ with a single parameter α_k. Hush and Salas have shown a simple approach in their paper to complete this procedure [28]. This method is to proceed along the search direction in small steps if the error function at each new position decreases, and stop when the error starts to increase.

The derivative of Eq.(5.2) with respect to α_k, for a quadratic function E(**w**), can be shown to be

$$\frac{d}{d\alpha_k} E(\mathbf{w}_k + \alpha_k \mathbf{p}_k) = \nabla E(\mathbf{w})^T |_{w=w_k} \mathbf{p}_k + \alpha_k \mathbf{p}_k^T \nabla^2 E(\mathbf{w}) |_{w=w_k} \mathbf{p}_k \tag{5.3}$$

α_k is chosen to minimize $E(\alpha_k)$ by setting this derivative equal to zero. We obtain

$$\alpha_k = -\frac{\nabla E(\mathbf{w})^T |_{w=w_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 E(\mathbf{w}) |_{w=w_k} \mathbf{p}_k} = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{H} \mathbf{p}_k} \tag{5.4}$$

where $\mathbf{H}_k$ is the Hessian matrix evaluated at point $\mathbf{w}_k$:

$$\mathbf{H}_k = \nabla^2 E(\mathbf{w}) |_{\mathbf{w}=\mathbf{w}_k} \tag{5.5}$$

For a quadratic function, **H** is the same everywhere.

5.1.2 Non-linear search and golden section method

Non-linear optimization is concerned with methods for locating the minimum or maximum of a non-linear function of any number of independent variables. For a non-quadratic function, Eq.(5.3) won't be applied. We need to have a general procedure for locating a local minimum of a function in a specific direction. There are many books and theories on non-linear optimization. We will combine function comparison and golden section search together [21]. Each line search proceeds in two stages, interval location and interval reduction.

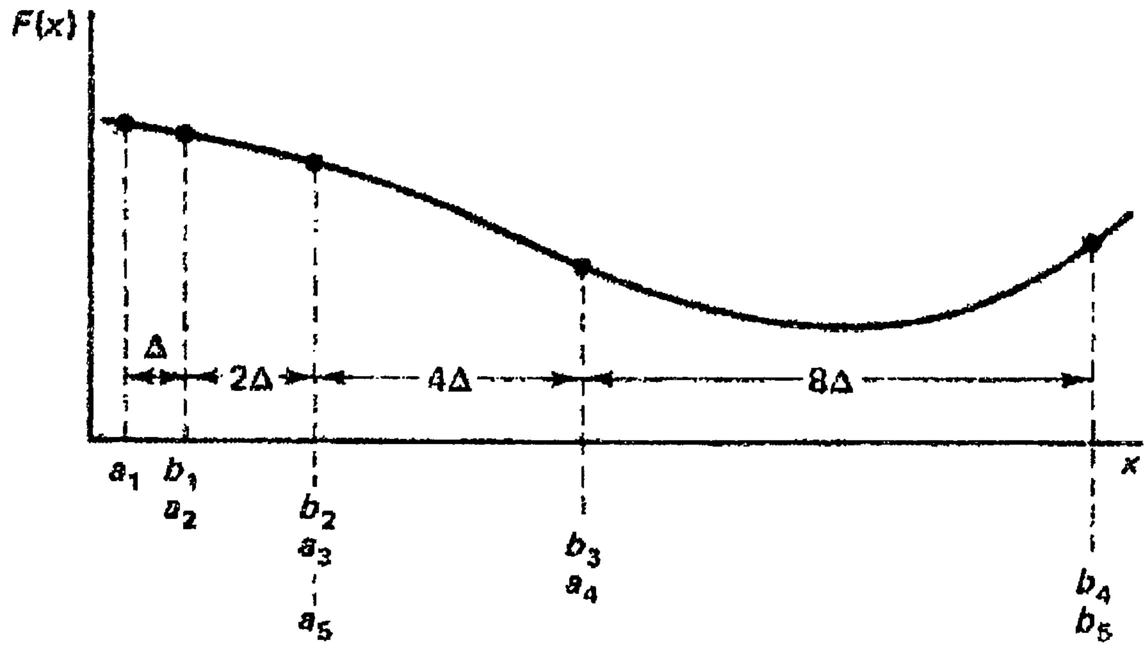


Figure 5.1

5.1.3 Interval location

The first step is to determine the interval within which the minimum is located. Since we always go downhill from the starting points, we assume that this minimum exists and the value of function from the starting point will decrease. Figure 5.1 depicts the procedure.

Suppose we start from point 'a_1'. For a given small distance, we evaluate function at the next point 'b_1'. If $F(a_1) > F(b_1)$, then we keep b_1 as a_2, go to the next point b_2 by doubling the distance, and evaluate the point 'b_2'. If $F(a_i) > F(b_i)$, repeat the same procedure until an increase in the function evaluation occurs. The minimum point should be in the last two intervals, for example, $[a_5, b_5]$ in this case [21].

5.1.4 Interval reduction

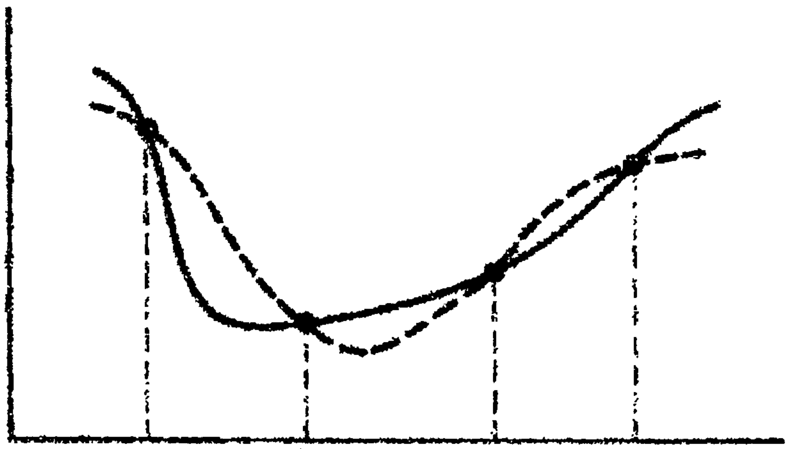


Figure 5.2

Once the interval is determined, we need to know the minimum point within this interval. Because the accuracy of this location is not satisfied by the interval location procedure, the next step is interval reduction. Namely, narrow the interval till the desired accuracy is attained and the minimum is located. Scales has described this procedure briefly and clearly. The following graphic and algorithm are directly from Scales [21].

The search algorithm has been summarized by Scales as follows:

```
Input $a_1$, $b_1$, $tol$
Set $c_1 = a_1 + (1-\tau)(b_1 - a_1)$, $F_c = F(c_1)$
    $d_1 = b_1 - (1-\tau)(b_1 - a_1), F_d = F(d_1)$
for $k = 1,2,....$ Repeat
    if $F_c < F_d$ then
        set $a_{k+1} = a_k, b_{k+1} = d_k, d_{k+1} = c_k$
          $c_{k+1} = a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$
          $F_d = F_c, F_c = F(c_{k+1})$
    else
        Set $a_{k+1} = c_k, b_{k+1} = b_k, c_{k+1} = d_k$
          $d_{k+1} = b_{k+1} - (1-\tau)(b_{k+1} - a_{k+1})$
          $F_c = F_d, F_d = F(d_{k+1})$
    end
end until $b_{k+1} - a_{k+1} < tol$
```

Procedure LOCMIN of R. P. Brent [35] is a much more efficient method of interval reduction than this one, but this will suffice for our purpose.

5.2 Conjugate gradient method

The basic idea of the line search minimization along a specific direction is to choose a suitable search direction at each stage of the algorithm. In the steepest descent method, the search directions are given by the local negative gradient at every point in the error surface. This is not the best choice though. Because the gradient at the new minimum is orthogonal to the previous search direction, choosing successive directions to be the local gradient directions can lead to the problem already illustrated in Figure 3.3, in which the search oscillates in successive directions while making little progress towards the minimum. Figure 3.3 shows a trajectory with successive search directions orthogonal. It then takes many steps to converge, even for a quadratic function. That's also why we said the steepest gradient based algorithms typically proceeded very slowly. This idea led to conjugate gradient methods.

Let's consider a quadratic function first. These two equations hold for a quadratic function:

$$\nabla F(\mathbf{x}) = \mathbf{H}x + \mathbf{d} \tag{5.6}$$

$$\nabla^2 F(\mathbf{x}) = \mathbf{H} \tag{5.7}$$

[30] A set of vectors $\{\mathbf{p}_k\}$ is said to be ***conjugate*** with respect to a positive definite Hessian matrix **H** if and only if

$$\mathbf{p}_k^T \mathbf{H} \mathbf{p}_j = 0, \quad k \neq j \tag{5.8}$$

Then the change in the gradient at iteration k+1 is

$$\Delta \mathbf{g}_k = \mathbf{g}_{k+1} - \mathbf{g}_k = (\mathbf{H}\mathbf{x}_{k+1} + \mathbf{d}) - (\mathbf{H}\mathbf{x}_k + \mathbf{d}) = \mathbf{H}\Delta\mathbf{x}_k \tag{5.9}$$

From Eq. (4.1) we have

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k \tag{5.10}$$

Then we obtain the modification of the conjugacy condition

$$\alpha_k \mathbf{p}_k^T \mathbf{H} \mathbf{p}_j = \Delta \mathbf{x}_k^T \mathbf{H} \mathbf{p}_j = \Delta \mathbf{g}_k^T \mathbf{p}_j = 0, \; k \neq j \tag{5.11}$$

This equation tells us that the search directions will be conjugate if they are orthogonal to the changes in the gradient. How does this affect our function optimization procedure? First, Scales, Gill, Murray and Wright have proved in their papers [21] that if we make a sequence of exact linear searches along any set of conjugate directions $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$, then the exact minimum of any quadratic function with n parameters, will be reached in at most n line searches. For example, if we apply this method to the 2-**D** quadratic function we illustrated in chapter 3, we need at most 2 iterations. Second, we no longer need to compute the second order derivatives (the Hessian matrix **H**). These give us a lot of benefits. Our optimization algorithm will be based on this.

5.3 The conjugate gradient algorithm for MLP training

Suppose we wish to optimize a function $F(\mathbf{x})$ and start at point $\mathbf{x}_0$. The procedure generates the sequence of points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$. For the conjugate gradient method, the first search direction, $\mathbf{p}_0$, is arbitrary. Usually we initialize $\mathbf{p}_0$ with the negative of the gradient, $-\mathbf{g}_0$ (we do not have to, though). The algorithm is described as follow:

Step 0: [Initialization]

Set k=1; **x**1 to a random point, cycle number m=0, *tol*

Step 1: [Set the negative steepest descent direction]

Calculate $\mathbf{g}_k$, set $\mathbf{p}_k = -\mathbf{g}_k$. (5.12)

Step 2: [Line search]

At step k, search along the line to determine the step length that minimize $E(\mathbf{x})$. Compute stepsize α_k using the line search methods (Interval location and Interval reduction).

Step 3: [Update the minimum point value in each iteration]

Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$. (5.13)

Step 4: [Stopping criterion]

If $\| \mathbf{x}_{k+1} - \mathbf{x}_k \| > reltol * \| \mathbf{x}_k \|$, go to step 8, otherwise go to the next step.

As mentioned in Chapter 4, for similarity, we use

$\| \mathbf{x}_{k+1} - \mathbf{x}_k \| > reltol * \| \mathbf{x}_k \|$ instead of $\| \mathbf{x}_{k+1} - \mathbf{x}_k \| > abstol + reltol * \| \mathbf{x}_k \|$,

Step 5: [Restart procedure]

If k mod $n = 0$, then set $\mathbf{p}_k = -\mathbf{g}_k$, $m = m+1, k = 1$, go to step 1, otherwise

go to the next step.

Step 6: [Direction search]

Determine the new search direction at the new minimum point,

Set $\mathbf{p}_{k+1} = -\mathbf{g}_{k+1} + \beta_{k+1}\mathbf{p}_k$ (5.14)

where β_{k+1} is calculated according to one of the three expressions (see

[21]):

$$\beta_{k+1} = \frac{\Delta g_k^T g_{k+1}}{\Delta g_k^T p_k}, \qquad \text{[Hestenes and Stiefel]} \qquad (5.15)$$

$$\beta_{k+1} = \frac{g_{k+1}^T g_k}{g_k^T g_k}, \qquad \text{[Fletcher and Reeves]} \qquad (5.16)$$

$$\beta_{k+1} = \frac{\Delta g_k^T g_{k+1}}{g_k^T g_k}, \qquad \text{[Polak and Ribiere]} \qquad (5.17)$$

Step 7: [Continue the m^{th} cycle]

Set $k = k+1$, go to step 3.

Step 8: [Stop]

Set iterations $m = m * n + k$.

These three expressions for β are actually three different conjugate gradient methods. There have been some attempts tried to determine which of these expressions for β is best, but no final conclusions have been reached [25]. Meishan Cheng declares in his M.S. thesis [20] that these various CG methods are relatively equivalent. It's believable so far. Later in our comparison, we'll apply the Fletcher-Reeves method.

The conjugate gradient algorithm is a modification of steepest descent, but it is based on a sound theoretical foundation. So it is more efficient and reliable than the BP algorithm and its variations. And CG does not have any heuristic factors or Hessian matrix computation. It is more practical.

In step 2, if a function is quadratic, step length is determined by the equation:

$$\alpha_k = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{H} \mathbf{p}_k} \tag{5.18}$$

Line search method could be avoided. But generally, the least square error function of a multi-layer perceptron network is not a quadratic function. As mentioned before, if the function is not a quadratic function, Eq. (5.18) cannot be applied to the algorithm. We need to search along the direction to get the minimum point. This procedure includes two parts: interval location and interval reduction. First, we need to search along the conjugate direction to determine the step length in each step (section 5.1.3, 5.1.4). Second, the algorithm no longer is guaranteed to converge within n iterations. The solution for the second problem is somehow uncertain. It seems that many researchers agree to have a restarting procedure for general objective functions.

When Fletcher and Reeves first applied the conjugate gradient algorithm to numerical optimization problems, they also recommended restarting. In their approach, the conjugate gradient method uses the steepest descent direction as the new search direction every n or (n+1) iterations [30]. We call the Fletcher and Reeves conjugate gradient method with their restart procedure the Fletcher-Reeves general method. Powell has shown that, without restarts, a linear rate of convergence is usual when there are more than two variables [24]. Crowder and Wolfe also gave an example to show that without restarting, the rate of convergence of traditional CG methods can be only linear [33].

Some later versions have tried to improve on the Fletcher-Reeves general method. Powell discussed restart search directions and procedures further in his paper [25]. Indeed, Powell does not restart. He generates a better search direction (details will be discussed later). We call his restart method "Powell restart" later.

For our purpose, we are going to apply the Powell restart method in our CG algorithm. We expect that this approach might have better performance than any steepest descent-based algorithm.

CHAPTER 6

CASE STUDIES AND PROGRAM DESCRIPTION

6.1. Description of the program

To support my points that have been discussed in the previous chapters, several numerical examples will be examined by a computer program. This program named BPCG is a collection of functions and subroutines written in standard FORTRAN 77. Five different algorithms will be tested in this program. Basically, I am going to focus on comparing backpropagation with a variable step size (BPVS) and conjugate gradient algorithms (CG). These two algorithms are the backbone of this paper. In addition, I'll take the backpropagation with a line search (BPLS), backpropagation with momentum (BPMOM) and quasi-Newton (QN) methods as options. This software mainly consists of three parts.

6.1.1 The network design

The strategy of neural network design is a really important factor of network training. As mentioned before, a three-layer network with a sigmoid function in the hidden layer and linear function in the output layer can solve any function approximation and classification problem. The three-layer network is the basic structure for this examination. But sometimes the functions in the hidden layer should be sigmoid functions since sometimes, for examples with normalized inputs and outputs, the network with sigmoid functions in both layers might perform better than that with sigmoid functions in the hidden layer and linear functions in the output layer. The number of nodes (neurons) in the output layer depends on the dimension of outputs of a specific example. The number of nodes in the hidden layer needs to be tested to make a decision. It's common to start from a network structure with a small number of nodes in the hidden layer. For example, in our Cancer problem which has 14 inputs and 3 outputs (see section 6.3, Numerical examples), we'll test the net structure of 14-3-3, 14-6-3,14-9-3, respectively. The code of setting up the network parameters refers to the program of Liya Wang's master's thesis [15]. But the –1 and 0 subscripts that Wang used have been eliminated by S. Nallarelli so that it can be accepted by some other FORTRAN compilers. Also, all the problems are tested with the two-layer network instead of just one output layer.

6.1.2. The code of steepest descent-based algorithms

The second part of the program is the implementation of BPVS, BPLS and BPMOM. BPVS is a steepest descent-based algorithm. We are concerned about BPVS's performance comparing not only to the conjugate method but also to the steepest descent category itself. Specifically, in this study, I'll implement BPLS and BPMOM. The comparisons to the modern numerical optimization methods will be discussed in the next section. The reason why the BPLS and BPMOM algorithms are chosen is that BPLS is the base method of backpropagation and BPMOM is a typical variation of backpropagation. BPVS adjusts the optimal step size dynamically based on a theoretical foundation. BPLS seeks the largest step size by using the line search method. The line search method involved in the BPLS algorithm is the golden section method with which to determine the optimal step size in each iteration. Since both BPVS and BPLS try to make the step size as large as possible in each iteration, they are supposed to have close performance to each other in terms of the rate of convergence, except that BPLS needs more function evaluations. For the BPMOM algorithm, we select the momentum factor $\gamma=0.8$ as is common. The step size, however, is not easy to select. It has to be tested to find an optimum value for different examples. The larger the step size is, the better, as long as it does not lead to divergence.

6.1.3. The existing numerical optimization methods

The third part is the implementation of the conjugate gradient and quasi-Newton methods. Standard, well-tested subroutines were used for these methods. The conjugate gradient algorithm uses the Fletcher and Reeves method with restarts in the negative steepest descent direction after an n+1 iterations cycle. In addition, I added the Polak-Ribiere formula to the CG method since sometimes this method works well. There are four traditional conjugate gradient methods, Fletcher-Reeves, Polak-Ribiere, Beale-Sorenson, and Perry [20]. For our purposes, we are not going to emphasize the difference among them here. Meishang Cheng has fully discussed the advantages and disadvantages of these four CG methods in his Master's thesis [20].

6.2. Numerical examples

In this chapter, I'll implement five examples to test the five different algorithms talked above. Generally speaking, they are either classification or approximation problems.

The first example is a function approximation problem. It was examined by Li Zhang in his Master's thesis [11].

$$f(x_1, x_2) = 1/(x_1^2 + x_2^2 + 1) \tag{6.1}$$

This function approximation by backpropagation with line search algorithm indicated some problems, such as inaccuracy, slow rate of convergence, etc. [11]. The information of this example is as follows:

Problem	Type of problem	Inputs	Outputs	Number of training data	Number of test data
$f(x_1, x_2)$	approximation	2	1	200	49

The next three examples come from PROBEN1 of the real data depository [26]. They were also tested by Liya Wang in his Master's thesis [15]. These three examples are listed below:

Examples	Type of examples	Inputs	Outputs	Number of training data	Number of test data
Cancer	Classification	9	2	175	125
Building	Approximation	14	3	200	150
Heart	Approximation	35	1	390	240

The last one example is the traditional classification problem, XOR. 49 uniformly spaced points are chosen on the square with vertices (-1,-1), (-1,1), (1,-1), (1,1)

Examples	Type of examples	Inputs	Outputs	Number of training data	Number of test data
XOR	Classification	2	1	49	24

6.3. The components of comparison.

For our goal, we are going to compare the performance of BP and CG algorithms. So the comparison shall focus on three major aspects, the rate of convergence, the stability, and the simulation accuracy. The rate of convergence is expressed by the number of iterations during the training phase. The accuracy is mainly measured by the value of the root mean square error (RMS).

CHAPTER 7

RESULT ANALYSIS

7.1. Results of the experiment

The following sets of data (number of updates of the weights and biases) are the test outputs of the program. Each of these tables is for one example, which is tested by five different algorithms in the same initial condition. The different initial guesses of the network parameters (weights and biases) may affect our test results. So, each example is tested under 10 different pseudorandom starting points. We will evaluate the average of them. The initial parameters are generated by a pseudorandom number generator, which is described in Liya Wang' s program [15]. Also, each example is examined for two different networks since different network architectures may affect the training results. The network with better outputs (smaller sum of square errors) is chosen as the component of our comparison. This network is probably not the best one, but it does not affect the objective of comparing the efficiency of different training methods in this paper.

Case 1: $f(x_1,x_2)=1/(x_1^2+x_2^2+1)$, Number of examples=200

Table 1.1. Function approximation with a 2-2-1 network,

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	1163	1026	1088	1007	1031	1170	1248	1105	1144	1058	1104
BPM	epoch	1010	1186	826	1103	1030	1314	1525	1227	1001	1443	1167
OM	stepsize	0.3										
BPVS	epoch	1031	1185	1059	1047	1094	1120	1314	1140	1412	1325	1173
CG	epoch	81	66	101	131	101	64	111	86	101	88	93
QN	epoch	67	75	118	150	76	61	132	106	135	84	100

Note: Global minimum of RMS=0.0303

Table 1.2. Function approximation with a 2-4-1 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	1641	1754	1424	1216	1182	1098	1196	1129	1270	1012	1292
BPM	epoch	1245	1369	1291	1086	1130	1162	1069	1094	981	1030	1146
OM	stepsize	0.3										
BPVS	epoch	1661	1714	1294	1023	1173	1165	1408	1062	972	1034	1251
CG	epoch	244	200	150	379	181	337	235	289	363	253	263
QN	epoch	176	269	165	344	288	336	311	264	371	331	285

Note: 1. Global minimum of RMS=0.0295

Case 2: Number of examples=175

Table 2.1. Cancer problem with a 9-2-2 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	1437	1091	1119	1000	1013	977	1396	1109	1549	1105	1180
BPM	epoch	1910	2027	1960	1693	1711	1824	1944	1797	1824	1981	1860
OM	stepsize	0.5										
BPVS	epoch	1175	1016	1058	1023	1022	977	944	1023	1333	1051	1062
CG	epoch	89	51	88	83	74	119	147	93	174	137	105
QN	epoch	97	63	184	113	55	94	114	89	137	173	102

Note: Global minimum of RMS=0.1491

Table 2.2. Cancer problem with a 9-4-2 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	1157	1412	1373	1158	1106	1151	1159	1473	1558	1105	1265
BPM	epoch	1498	1522	1519	1477	1459	1513	1320	1479	1417	1475	1468
OM	stepsize	0.45										
BPVS	epoch	1236	1174	1193	1204	1189	1359	1211	1210	1328	1173	1227
CG	epoch	85	103	52	123	300	80	62	71	149	60	108
QN	epoch	81	145	48	146	245	70	98	106	106	67	111

Note: Global minimum of RMS=0.1488

Case 3: Number of examples=200

Table 3.1 Building problem with a 14-3-3 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	1817	1879	2205	1813	1915	2766	1987	2941	2441	2673	2244
BPM	epoch	1741	2483	1951	2156	2043	2265	2346	3118	2732	2442	2328
OM	stepsize	0.3										
BPVS	epoch	1632	1993	2309	2143	1982	2399	2077	2977	2941	2721	2317
CG	epoch	173	407	291	175	407	408	234	233	233	175	274
QN	epoch	167	377	306	248	408	388	218	341	330	242	302

Note: 1. Global minimum of RMS=0.0330

Table 3.2 Building problem with a 14-6-3 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	2178	1938	2046	2512	2441	2563	2223	2281	3086	2722	2399
BPM	epoch	1822	1823	1841	1905	2203	2057	2186	2433	2796	2991	2205
OM	stepsize	0.3										
BPVS	epoch	2577	2140	1700	2344	3322	1906	1921	2102	3631	2669	2431
CG	epoch	337	338	473	337	225	225	345	450	449	338	351
QN	epoch	300	291	430	265	243	296	429	547	265	306	337

Note: 1. Global minimum of RMS=0.0321

Case 4: Number of examples=390

Table 4.1 Heart problem with a 35-2-1 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	2997	4136	3210	2784	3121	3070	3049	3591	3124	3123	3220
BPM	epoch	3987	3887	2986	2953	3207	3049	2703	2506	3198	3713	3219
OM	stepsize	0.3										
BPVS	epoch	3372	3086	2837	2970	2339	3662	3600	3377	3320	2701	3130
CG	epoch	239	266	198	235	195	206	305	323	232	153	235
QN	epoch	222	202	180	256	232	160	243	345	299	122	226

Note: 1. Global minimum of RMS=0.1317

Table 4.2 Heart problem with a 35-4-1 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	5061	4503	4278	4451	4322	4014	3660	4076	4411	3899	4266
BPM	epoch	4290	4976	4252	4169	3116	3981	3341	3954	3901	4584	4056
OM	stepsize	0.3										
BPVS	epoch	5393	3698	4542	4555	3820	3519	2522	4886	4313	4584	4180
CG	epoch	373	301	300	388	392	216	301	301	297	267	313
QN	epoch	328	338	248	213	316	189	311	249	253	191	264

Note: 1. Global minimum of RMS=0.1208

Case 5: Number of examples=49

Table 5.1 XOR problem with a 2-2-1 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	321	261	263	374	265	274	260	372	262	263	292
BPMOM	epoch	361	381	337	344	304	372	288	356	155	418	331
BPVS	epoch	437	206	202	246	218	218	196	276	150	176	233
CG	epoch	28	36	43	32	21	22	31	22	24	18	27
QN	epoch	29	20	22	28	22	17	20	46	27	22	25

Note: 1. Global minimum of RMS=0.001557

Table 5.2 XOR problem with a 2-4-1 network

	Seeds->	1	17	21	27	40	45	66	78	81	96	Average
BPLS	epoch	323	289	304	290	298	309	292	311	297	284	299
BPMOM	epoch	335	275	367	346	296	205	326	271	294	294	301
BPVS	epoch	266	214	254	266	244	282	202	259	303	225	252
CG	epoch	19	19	26	24	19	24	18	24	19	38	23
QN	epoch	18	10	27	18	15	10	28	25	10	36	20

Note: 1. Global minimum of RMS=0.001419

7.2. Results analysis

Obviously, the two simplest network structures for each example generate the different accuracies of approximation. For the function approximation ($f(x_1, x_2) = 1/(x_1^2 + x_2^2 + 1)$) problem, for instance, the root mean square error (RMS) of a 2-4-1 network is better than that of a 2-2-1 network. We will choose the one with the better performance as our target.

The results indicate that conjugate gradient and quasi-Newton algorithms are much faster than BPLS, BPVS, and BPMOM algorithms. For all five examples, these two numerical optimization algorithms take a few iterations (at most several tens) to converge, while those steepest descent-based algorithms need thousands of iterations. This is not an accident though. For example, the training procedure of the cancer problem shows that the error surface has a very flat valley. When BPLS, BPVS, and BPMOM algorithm fall into this valley, they have to follow the negative gradient direction and make a very little progress in each iteration, even though they try to make the step size larger. The conjugate gradient method, however, does not have to follow these directions. It goes along the conjugate gradient direction. The convergence is very fast.

Inside the steepest descent-based category itself, the BPVS and BPLS have very close rates of convergence. That's what we expected. The difference is the training procedure. The BPLS takes a long time to converge since it has to keep evaluating the error function till it gets the minimum point in that direction. The BPVS algorithm avoids so many function evaluations, requiring fewer than BPLS by approximately a constant factor. If the step size of the BPMOM is proper, this algorithm also works well. In some cases, BPMOM is even better than BPLS and BPVS.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1. Conclusions

The above simulation results provide support for what I expected. I have examined three steepest descent-based algorithms (BPVS, BPLS and BPMOM) and two numerical optimization algorithms (CG and QN). The results generally show that good numerical optimization methods have fast convergence. The newly developed algorithms in the steepest descent backpropagation family, such as the 'Effective backpropagation training with variable stepsize' are not able to approach the efficiency of the conjugate gradient algorithm. CG methods share all the desirable properties of the steepest descent method, namely low storage, ease of implementation, and parallelization. However, when properly implemented they converge far more rapidly. The advantage of the steepest descent methods over those numerical optimization methods (Newton, GN, and LM) which need to compute second order partial derivatives and therefore need $O(n^2)$ storage is the low storage. We also noticed that the quasi-Newton method trapped to local minima several times. This may be another disadvantage of the quasi-Newton method.

8.2. Future work

We compared the steepest descent based algorithms to the modern numerical optimization algorithms. How is the performance of CG and QN? In practice, CG methods generally take approximately the same number of iterations as quasi-Newton methods [22]. The results of our simulations seem to agree that CG and QN have close performance. It's interesting to investigate and answer this question.

REFERENCES

1. R. Beale and T. Jackson, *Neural Computing: An Introduction*, Adam-Hilger,1991

2. M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*, Boston: PWS Pub.,1996

3. F. Rosenblatt: "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, Vol. 65, pp. 384-408, 1958

4. B. Widrow, M. E. Hoff, "Adaptive switching circuits," *IRE WESCON Convention Record*, New York: IRE Part 4, pp. 96-104, 1960

5. K. M. Hornik, M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, Vol. 2, No. 5, pp. 359-366, 1989

6. M. Minsky and S. Papert, *Perceptrons*, Cambridge, MA: MIT Press, 1969

7. D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, Vol. 323, pp. 533-536, 1986

8. D. Rumelhart, and J. McCleland, *Parallel Distributed Processing*, Cambridge, MA: MIT, Vol. 1, 1986

9. G. D. Magoulas, M. N. Vrahatis, and G. S. Androulakis: "Effective backpropagation training with variable stepsize", *Neural Networks*, Vol. 10, No. 1, pp. 69-82, 1997

10. Pinaki RoyChowdhury, Y. P. Singh, and R. A. Chansarkar, "Dynamic tunneling technique for efficient training of multilayer perceptrons", *IEEE transactions on Neural Networks*, Vol. 10, No. 1, pp. 48-55, 1999

11. Li Zhang, "Orthogonal least squares algorithm for radial basis function networks and its comparison with multilayer perceptron networks", *M.S. Thesis*, Computer Science Department, Oklahoma State University, 1998

12. C. Charalambous, "Conjugate gradient algorithm for efficient training of artificial neural networks", *IEE Proceedings*, Vol. 139, No. 3, pp. 301-310, 1992

13. R. A. Jacobs, "Increased Rates of Convergence Through Learning Rate Adaptation", *Neural Networks*, Vol. 1, No. 4, pp. 295-308, 1988

14. T. P. Vogl, J. K. Mangis, A. K. Zigler, W. T. Zink and D. L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, Vol. 59, pp. 256-264, 1988

15. Liya Wang, "Damped Newton method--an ANN learning algorithm", *M.S. Thesis*, Computer Science Department, Oklahoma State University, 1995

16. M. T. Hagan, and M. Menhaj, "Training feedforward networks with the Marquardt algorithm", *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, pp. 989-993, 1994

17. P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences", *Ph.D. Thesis*, Harvard University, Cambridge, MA, 1974

18. Christopher M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press Inc., 1995

19. M. Hestenes, *Conjugate Direction Methods in Optimization*, Springer-Verlag, 1980

20. Meishan Cheng, "A survey and comparison for conjugate gradient methods for optimization", *M.S. Thesis*, Computer Science Department, Oklahoma State University, 1993

21. L. E. Scales, *Introduction to Non-linear Optimization*, Springer-Verlag, 1985

22. Jorge Nocedal, Stephen Wright, *Numerical Optimization*, Springer-Verlag, 1999

23. Loyce Adams, J. L. Nazareth, "Linear and nonlinear conjugate gradient-related methods", *Soc. for Industrial and Appl. Math.*, 1996

24. M. J. D. Powell, "Some convergence properties of the conjugate gradient methods", *Mathematical Programming,* 11 (1976) 42-49

25. M. J. D. Powell, "Restart procedures for the conjugate gradient methods", *Mathematical Programming*, 12 (1977) 241-254

26. D. F. Shanno: "Recent advances in numerical techniques for large-scale optimization", *Neural Networks for Control*, Miller, Sutton and Werbos, eds., Cambridge MA: MIT Press, 1990

27. P. Baldi and K. Hornik, "Neural networks and principal component analysis: learning from examples and local minima," *Neural Networks*, Vol. 2, pp. 53-58, 1989

28. D. R. Hush and J. M. Salas, "Improving the learning rate of backpropagation with the gradient reuse algorithm," *IEEE International Conference on Neural Networks*, Vol. 1, pp. 441-447, 1988

29. UCI Machine Learning, URL: http://www.ics.uci.edu/~mlearn/MLSummary.html, U. California-Irvine

30. R. Fletcher and C. M. Reeves, "Function minimization by conjugate gradients," *The Computer Journal*, Vol. 7, pp. 149-154, 1964

31. M. F. McGuire and P. Wolfe, "Evaluating a restart procedure for conjugate gradients", *Report RC-4382*, IBM Research Center, Yorktown Heights, 1973

32. Lutz Prechelt, PROBEN1-A Set of Neural Network Benchmark Problems and Benchmarking Rules, *Technical Report 21/94*, University Karlsruhe, 1994, file /pub/neuron/proben1.tar.gz, available via anonymous ftp from ftp.ira.uka.de

33. H. P. Crowder and P. Wolfe, "Linear convergence of the conjugate gradient method", *IBM Journal of Research and Development*, 16(1972) 431-433

34. F. Rosenblatt, *Principles of Neurodynamics*, Washington D. C.: Spartan Press, 1961

35. R. P. Brent, *Algorithms for Minimization Without Derivatives*, Englewood Cliffs, N. J., Prentice-Hall, 1972

36. W. C. Davidon, "Variable Metric Method for Minimization", *Technical Report ANL-5990 (Rev.)*, Argonne National Laboratory, November 1959

37. Kenneth Levenberg, "A Method for the Solution of Certain Non-linear Problems in Least Squares", *Quart. J. Math.*, 1943

38. D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters", *Soc. for Industrial and Applied Math.*, Vol. 11, No.2, 1963

39. H. D. Block, B. W. Knight, Jr. and F. Rosenblatt, "Analysis of a Four-Layer Series-Coupled Perceptron. II", *Reviews of Modern Physics*, vol. 34, pp.135-142, January 1962

VITA

Guoping Miao

Candidate for the Degree of

Masters of Science

Thesis: A COMPARISON OF BACKPROPAGATION AND CONJUGATE GRADIENT ALGORITHMS FOR EFFICIENT TRAINING OF MULTI-LAYER PERCEPTRON NETWORKS

Major Field: Computer Science

Biographical:

Education: Graduated from Wuhan Urban Construction Institute, Wuhan, China in 1990; received Bachelor of Engineering. Complete the requirements for the Masters of Science degree with a major in Computer Science at Oklahoma State University in December 2002.

Experience: Construction Engineer in Nanjing Chemical Industrial Company from 1990 to 1997; Computing and Information Service Lab Assistant, Oklahoma State University, January 1999 to October 2000; Member of Scientific Stuff, Nortel Networks Inc., Dallas from November 2002 to present.