Advanced Learning Algorithms of Neural Networks

by

Hao Yu

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 12, 2011

Keywords: Artificial Neural Networks, Levenberg Marquardt Algorithm, Neuron-by-Neuron Algorithm, Forward-Only Algorithm, Improved Second Order Computation

Approved by

Bogdan M Wilamowski, Chair, Professor of Electrical and Computer Engineering
Hulya Kirkici, Professor of Electrical and Computer Engineering
Vishwani D. Agrawal, Professor of Electrical and Computer Engineering
Vitaly Vodyanoy, Professor of Anatomy Physiology and Pharmacy

Abstract

The concept of “learn to behave” gives very vivid description of functionalities of neural networks. Specifically, a group of observations, each of which consists of inputs and desired outputs, are directly applied to neural networks, and the networks parameters (called “weights”) are adjusted iteratively according with the differences (called “error”) between desired network outputs and actual network output. The parameter adjustment process is called “learning” or “training”. After the errors converging to expected accuracy, the trained networks can be used to analyze the input dataset which are in the same range of observations, for classification, recognition and prediction.

In neural network realm, network architectures and learning algorithms are the major research topics, and both of them are essential in designing well-behaved neural networks. In the dissertation, we are focused on the computational efficiency of learning algorithms, especially second order algorithms. Two algorithms are proposed to solve the memory limitation problem and computational redundancy problem in second order computations, including the famous Hagan and Menhaj Levenberg Marquardt algorithm and the recently developed neuron-by-neuron algorithm.

The dissertation consists of seven chapters. The first chapter demonstrates the attractive properties of neural network with two examples, by comparing with several other methods of computational intelligence and human beings. The second chapter introduces background of neural networks, including the history of neural networks, basic concepts, network architectures,

learning algorithms, generalization ability and the recently developed neuron-by-neuron algorithm. The third chapter discusses the current problems in second order algorithms. The fourth chapter describes another way of gradient vector and quasi Hessian matrix computation for implementing Levenberg Marquardt algorithm. With the similar computational complexity, the improved second order computation solves the memory limitation in second order algorithms. The fifth chapter presents the forward-only algorithm. By replacing the backpropagation process with extra calculation in forward process, the forward-only algorithm improves the training efficiency, especially for networks with multiple outputs. Also, the forward-only algorithm can handle networks consisting of arbitrarily connected neurons. The sixth chapter introduces the computer software implementation of neural networks, using C++ based on Visual C++ 6.0 platform. All the algorithms introduced in the dissertation are implemented in the software. The seventh chapter concludes the dissertation and also introduces our recent work.

Acknowledgments

First of all, I would like to sincerely appreciate my honorific supervisor, Prof. Bogdan M Wilamowski, for his great patience and knowledgeable guidance during the past three years Ph.D study. His professional research experience teaches me how to be creative, how to find problems and solve them. His active attitude of life encourages me working hard towards my destination. His kindness and great sense of humor makes me feel warm and happy. All the things I have learnt from him are marked deeply in my memory and will benefit the rest of my life. Without his help, I could not have finished my dissertation and Ph.D study successfully. Besides, I would like to express my special appreciation to both Mr. Bogdan Wilamowski and his wife, Mrs. Barbara Wilamowski, for their kindness, caring about me and letting me feel like studying at home.

Special thanks are also given to my committee members, Prof. Hulya Kirkici, Prof. Vishwani D. Agrawal and Prof. Vitaly Vodyanoy, and the outside reader Prof. Weikuan Yu. From their critical and valuable comments, I noticed the weakness in my dissertation and made the necessary improvements according to their suggestions.

I would like to express my appreciation to my good friends who have helped me with my studying and living in Auburn. They are Joel Hewlett, Nam Pham, Nicholas Cotton, Pradeep Dandamudi, Steven Surgnier, Yuehai Jin, Haitao Zhao, Hua Mu, Jinho Hyun, Qing Dai, Yu Zhang, Chao Han, Xin Jin, Jia Yao, Pengcheng Li, Fang Li and Jiao Yu. I am very lucky to be their friend.

Special thanks to Charles Ellis, Prof. David Irwin and Prof. Michael Hamilton, for their professional guidance on the projects and papers we worked together. It was my great honor to have worked with them. I also would like to thank Prof. John Hung, Prof. Fa Foster Dai, Prof. Hulya Kirkici, Prof. Vishwani Agrawal, Prof. Stanley Reeves, Prof. Adit Singh, Prof. Bogdan Wilamowski and Prof. Thomas Baginski, for their excellent teaching skills and professional knowledge in their courses.

Last but not least, I am greatly indebted to my wife, Dr. Tiantian Xie, my newborn daughter, Amy X Yu, and my parents and my parents-in-law. They are the backbone and origin of my happiness. Being both a father and mother while I was struggling with my dissertation was not an easy thing for my wife. Without her support and encouragement, I could never finish my Ph.D study successfully. I owe my every achievement to my family.

Thanks to everyone.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

WHY NEURAL NETWORKS

1.1 Introduction

As rapid development of computational intelligence, the tendency becomes more and more apparent that human kind is going to be replaced by intelligent systems. Various algorithms of computational intelligence have been well-developed based on different biological or statistic models [1-4], and they are paid great attentions in both scientific research and industrial applications, such as nonlinear compensations [5-7], motor control [8-12], dynamic distribution systems [13], robotic manipulators [14-16], pattern recognition [17-19] and fault diagnosis [20-21].

Artificial neural networks (ANNs) were extracted from the complicated interconnections of biological neurons and inherit the learning and reasoning properties of human brains. It was proven that neural networks could be considered as a general model being capable of building arbitrary linear/nonlinear relationships between stimulus and response [22]. It is still unknown about the internal computations of neural networks, so it is hard to design them directly; instead, researchers have developed smart algorithms to train neural networks. Error back propagation (EBP) algorithm [23], developed by David E. Rumelhart, is the first algorithm which has ability to train multilayer perceptron (MLP) networks. Levenberg Marquardt (LM) algorithm [24-25] is regarded as one of the most efficient algorithms for neural network learning. Recently developed second order neuron-by-neuron (NBN) algorithm [26-27] is capable of training arbitrarily

connected neural (ACN) networks which could be more efficient and powerful than traditional MLP networks. Fault tolerance and generalization ability are improved, when efficient network architectures are applied for training [28].

Fuzzy inference systems were designed based on fuzzy logical rules [29]. All parameters for designing fuzzy inference systems can be extracted from problems themselves, and the training process is not required. However, the tradeoff of the very simple design process is the accuracy of approximation. Some hybrid architectures [30], inherited from both neural networks and fuzzy inference systems, are proposed to improve the performance of fuzzy inference systems. Another disadvantage of fuzzy inference systems is that, as the increase of input dimensions, the computation cost increases exponentially.

Support vector machines (SVMs) were developed from statistical learning theory [31] to solve data classification problems. The concept of SVMs is very similar with the three-layer MLP networks. Differently, the layer-by-layer architecture in SVMs is organized based on Cover's theorem and each layer performs different computation. Unlike other learn-by-examples systems, SVMs do not face local minima problem and they can find optimized solutions by constrained learning process. Later improvements [32] make SVMs also proper for solving function approximation problems.

Other methods of computational intelligence, such self-organizing maps (SOMs) [33], principal component analysis (PCA) [34], particle swarm optimization [35], ant colony optimization [36] and genetic algorithm [37], also attracts great interests in solving special optimization problems. These methods are often combined with training algorithms so as to improve their performance [38-40].

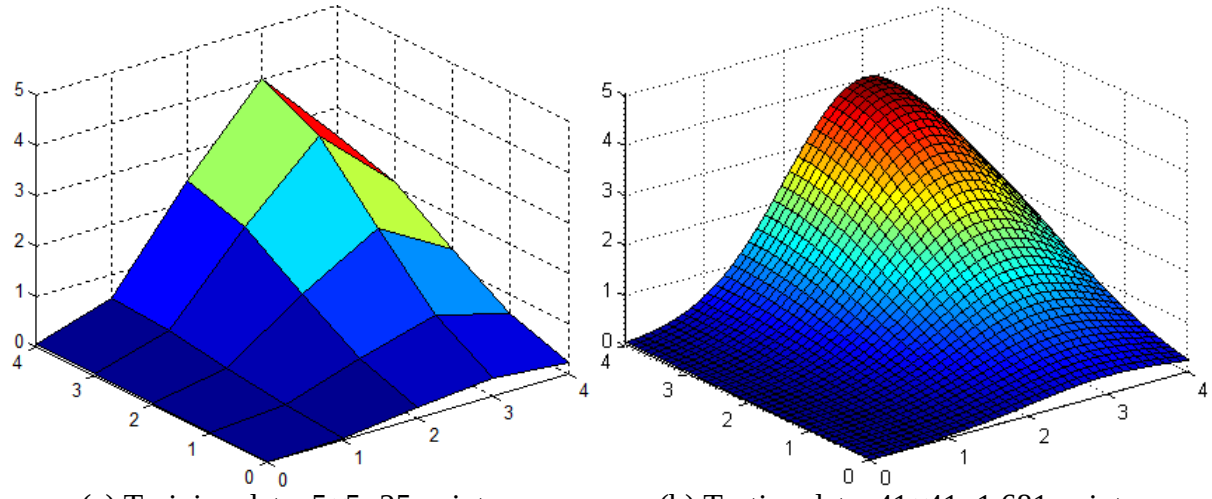
In the followed two sections, we will have two examples to illustrate the potential

advantages of neural networks over (1) several other methods for function approximation, and (2) human beings for image recognition.

1.2 Comparison of Different Nonlinear Approximators

In this section, different methods of computational intelligence, including fuzzy inference systems, neuro-fuzzy systems and support vector machines, interpolation and neural networks are compared based on a nonlinear surface approximation problem. The purpose of the problem is that, using the given 5×5=25 points (Fig. 1-1a, uniformly distributed in [0, 4] in both *x* and *y* directions) to approximate the 41×41=1,681 points (Fig. 1-1b) in the same input range. All the training/testing points are obtained by equation (1-1) and visualized in Fig. 1-1. The approximation will be evaluated by sum square error (SSE).

$$z = 4\exp\left(-0.15(x-4)^2 - 0.5(y-3)^2\right) + 10^{-9} \tag{1-1}$$



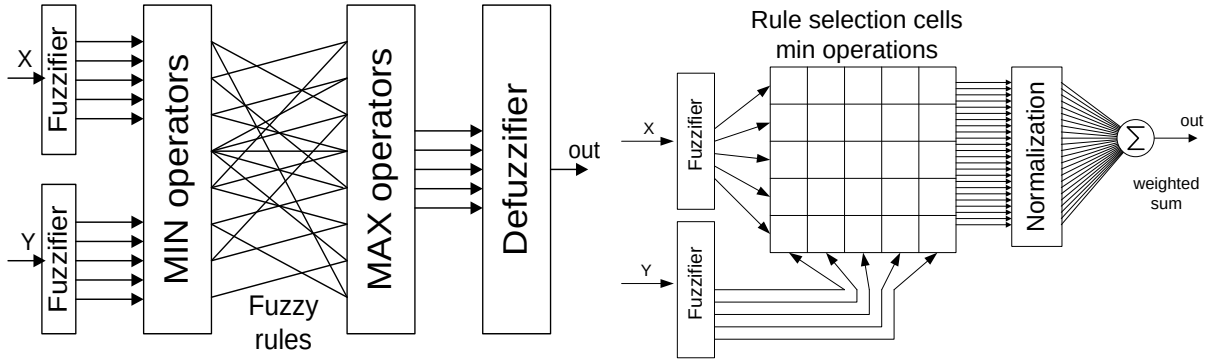
(a) Training data, 5×5=25 points (b) Testing data, 41×41=1,681 points

Fig. 1-1 Surface approximation problem

1.2.1 *Fuzzy Inference Systems*

The most commonly used architectures for fuzzy system development are the Mamdani fuzzy system [41] and TSK (Takagi, Sugeno and Kang) fuzzy system [42]. Both of them consist of

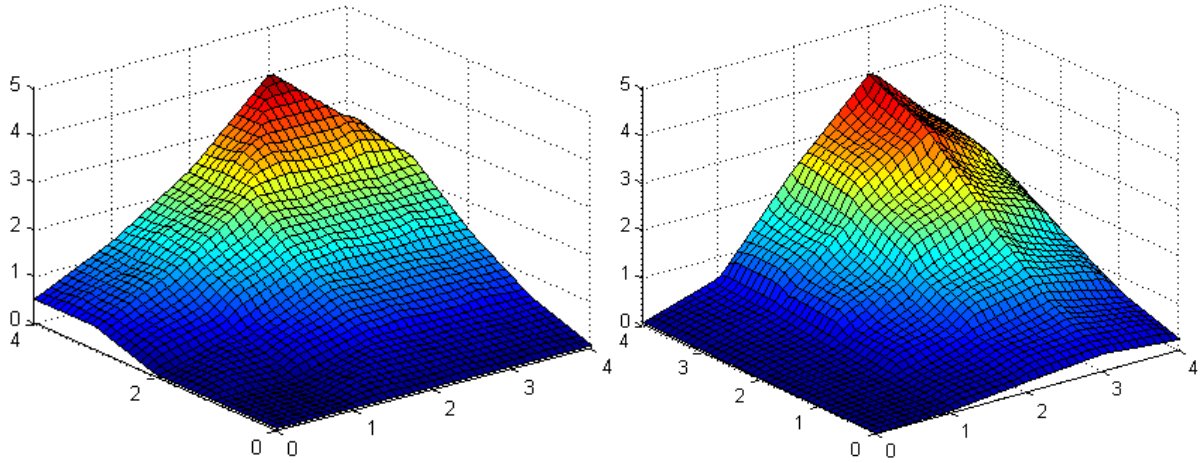
three blocks: fuzzification block, fuzzy rule block and defuzzification/normalization block, as shown in Fig. 1-2 below.



(a) Mamdani fuzzy system (b) TSK fuzzy system

Fig. 1-2 Block diagram of the two types of fuzzy systems

For the given surface approximation problem, with 5 triangular membership functions in each direction, two different fuzzy inference systems can obtain the approximated surfaces as shown in Fig. 1-3.



(a) Mamdani fuzzy system, SSE=319.7334 (b) TSK fuzzy system, SSE=35.1627

Fig. 1-3 Result surfaces obtained using fuzzy inference systems

The rawness of control surfaces (Fig. 1-3) in fuzzy controllers leads to raw control and instabilities [43]. Therefore, for resilient control systems fuzzy controllers are not used directly in the control loop. Instead, traditional PID controllers [44-45] are often used and fuzzy inference

systems are just applied to automatically adjust parameters of PID controllers [46].

1.2.2 *Neuro-Fuzzy Systems*

The neuro-fuzzy system, as shown in Fig. 1-4, attempts to present fuzzy inference system in form of neural network [47]. It consists of four layers: fuzzification, multiplication, summation and division. Notice that, in the second layer, product operations are performed among fuzzy variables (from first layer), instead of the fuzzy rules (MIN/MAX operations) in classic fuzzy inference systems. The multiplication process improves the performance of neuro-fuzzy system, but it is more difficult for hardware implementation.

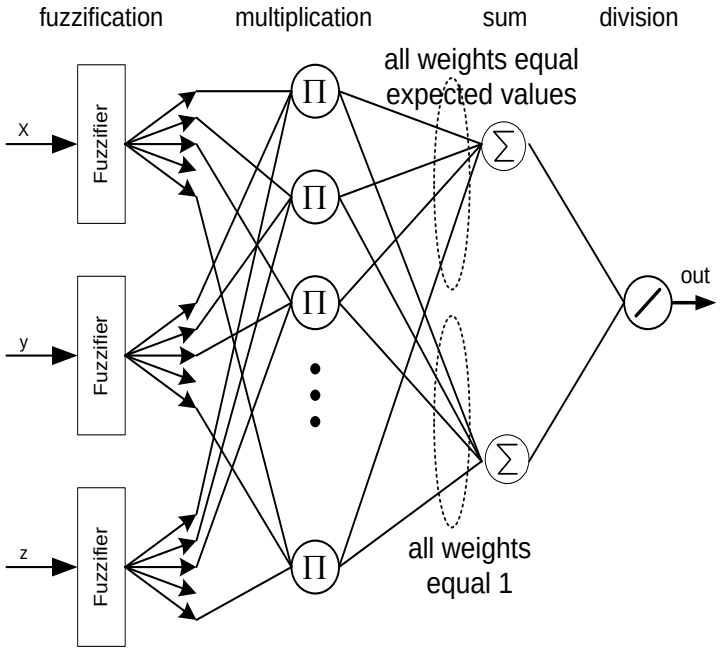


Fig. 1-4 Neuro-Fuzzy System

For the given problem, with the same membership functions chosen for fuzzy inference system design in section 1.2.1, Fig. 1-5 shows the approximation result of the neuro-fuzzy system.

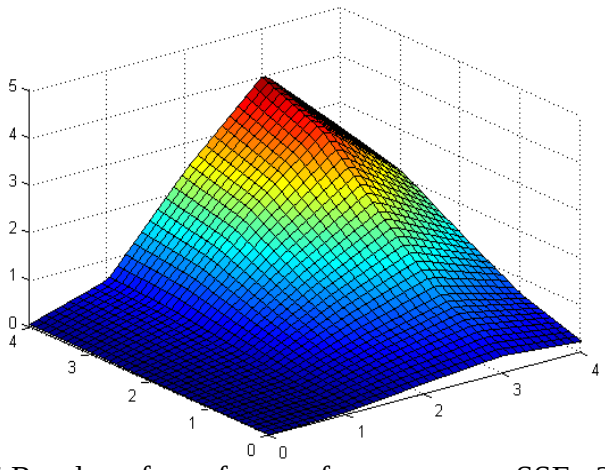
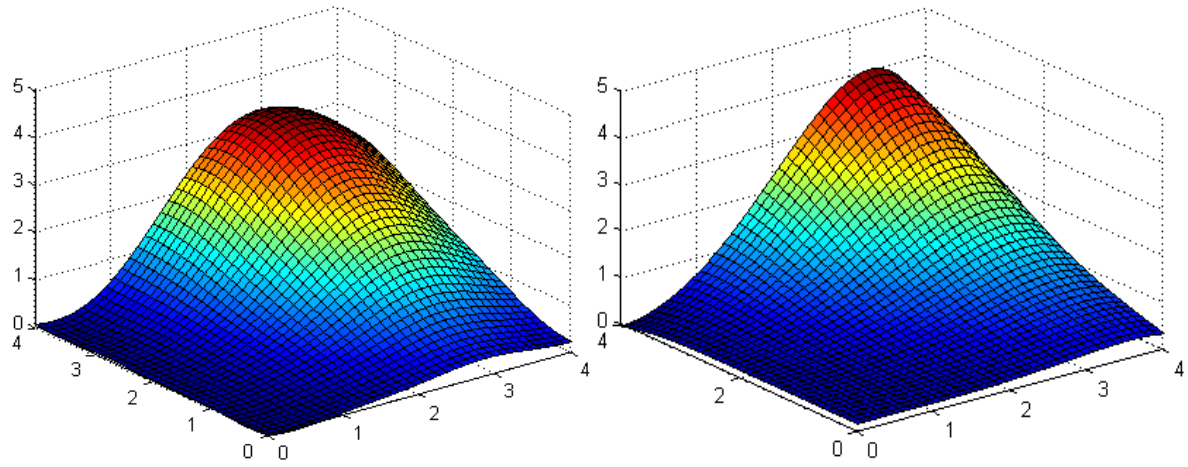


Fig. 1-5 Result surface of neuro-fuzzy systems, SSE= 27.3356

1.2.3 *Support Vector Machines (SVMs)*

For the given problem, with the software LIBSVM [48], the best results (as we tried) obtained using radial basis function kernel ($\exp(-\gamma|\boldsymbol{u}-\boldsymbol{v}|^2)$ with γ=0.7) and polynomial kernel ($((0.1\boldsymbol{u}'\times\boldsymbol{v}+0.1)^n$ with n=7) separately are shown in Fig. 1-6. For other kernels, such as linear and sigmoid, the SVM does not work at all.

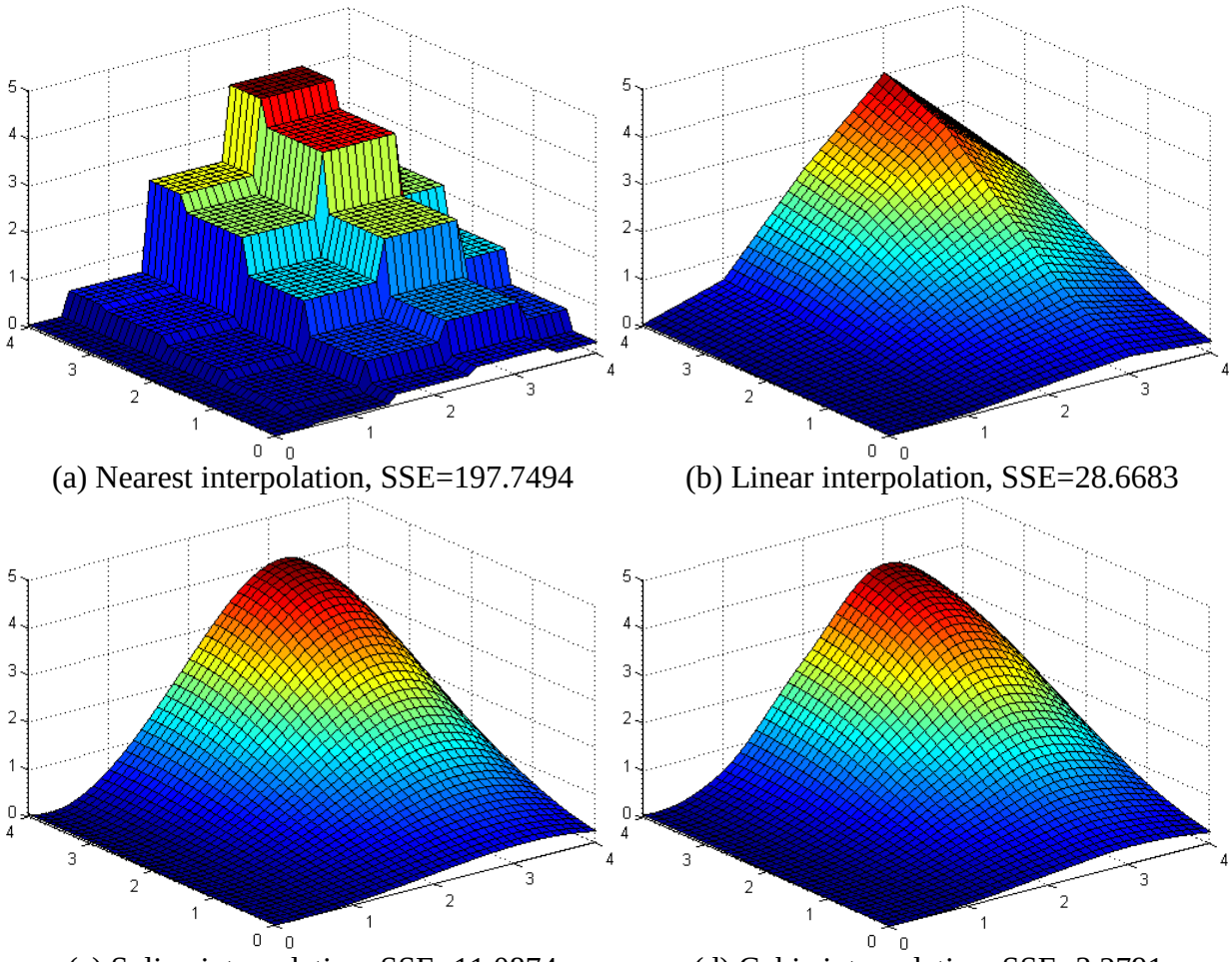


(a) Radial basis function kernel, SSE=28.9595 (b) Polynomial kernel, SSE=176.1520
Fig. 1-6 Result surfaces obtained using support vector machine

1.2.4 *Interpolation*

Interpolation is considered as a commonly used method for function approximation. MATLAB

provides the function "*interp2*" for two-dimension interpolation and there are four approximation methods used in this function: nearest (nearest neighbor interpolation), linear (bilinear interpolation), spline (spline interpolation) and cubic (bicubic interpolation as long as the data is uniformly distributed). Fig. 1-7 presents the approximation results using the four different ways of interpolation.



(a) Nearest interpolation, SSE=197.7494 (b) Linear interpolation, SSE=28.6683

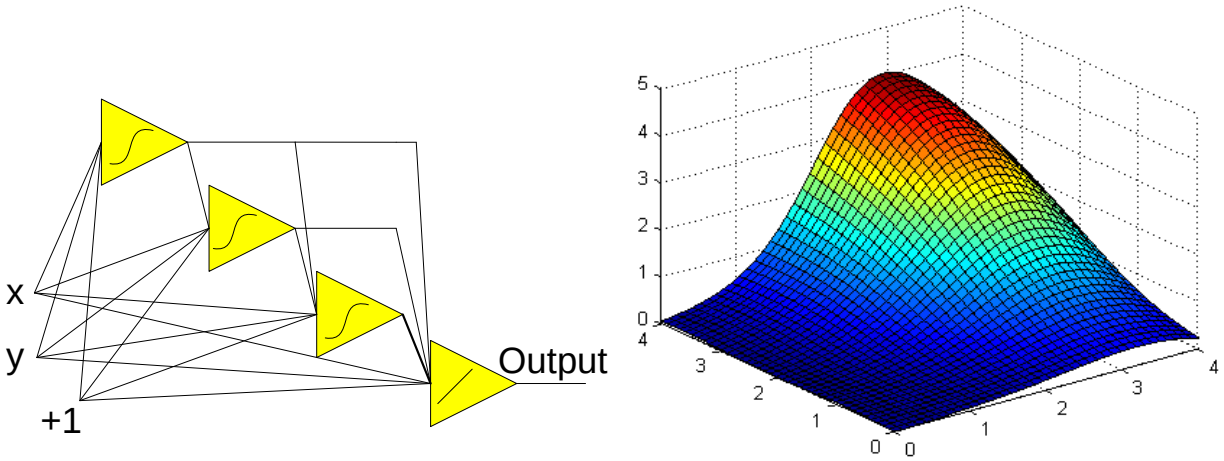
(c) Spline interpolation, SSE=11.0874 (d) Cubic interpolation, SSE=3.2791

Fig. 1-7 Result surfaces obtained using interpolation methods

1.2.5 *Neural Networks*

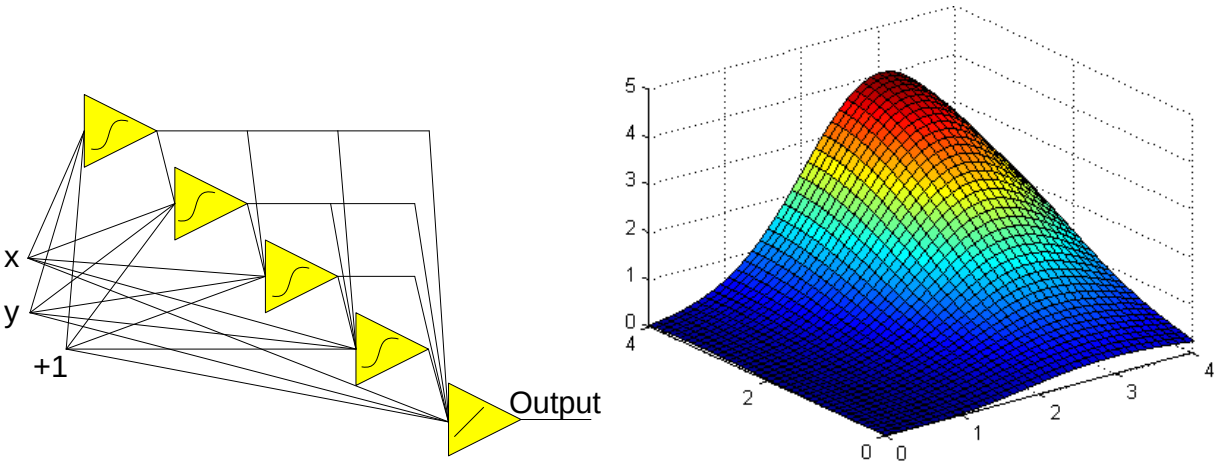
For the given problem in Fig. 1-1, Figs. 1-8 and 1-9 show the result surfaces using different number of neurons with fully connected cascade (FCC) networks. All the hidden neurons use

unipolar sigmoidal activation functions and the output neuron is linear. The software NBN 2.0 [49-51] was used in the experiment and the neuron-by-neuron (NBN) algorithm [52-53] in the software was selected for training.



(a) Four neurons in FCC network (b) Result surface with SSE=2.3628

Fig. 1-8 Neural network architecture and related testing result



(a) Five neurons in FCC network (b) Result surface with SSE=0.4648

Fig. 1-9 Neural network architecture and related testing result

Table 1-1 concludes the experimental results of different nonlinear approximators. One may notice that, from the point of approximating accuracy, neural networks can be the best choice for the problem.

Table 1-1 Comparison of approximation accuracy using different methods of computational intelligence

Methods of Computational Intelligence	***Sum Square Errors***
Fuzzy inference system – Mamdani	319.7334
Fuzzy inference system – TSK	35.1627
Neuron – fuzzy system	27.3356
Support vector machine – RBF kernel	28.9595
Support vector machine – polynomial kernel	176.1520
Interpolation – nearest	197.7494
Interpolation – linear	28.6683
Interpolation – spline	11.0874
Interpolation – cubic	3.2791
Neural network – 4 neurons in FCC network	2.3628
Neural network – 5 neurons in FCC network	0.4648

1.3 Neural Networks for Image Recognition

It is common knowledge that computers are much superior to human beings in numerical computation; however, it is still believed that human beings are superior to computers in areas of image processing. In this part, an example is used to show the expertise of special designed neural networks for recognizing noised images which cannot be handled by normal people.

The experiment was carried out in the following scheme. As shown in Fig. 1-10, for each column, there are 10 digit images, from "0" to "9", each of which consists of 8×7=56 pixels with normalized Jet degree between -1 and 1 (-1 for blue and 1 for red). The first column is the original image data without noise; for the noised data from the 2nd column to the 8th column, the strength of noise is increased according with equation (1-2):

$$NP_i = P_0 + i \times \delta \tag{1-2}$$

Where: P_0 is the original image data (the 1st column); NP_i is the image data with i-th level noise; i is the noise level; δ is the randomly generated noise between [-0.5, 0.5].

The purpose of this problem is to design the neural networks based on the image data in the 1st column and then test the generalization ability of the designed neural networks using the

noised image data, from the 2nd column to the 8th column. For each noise level, the testing will be repeated for 100 times with randomly generated noise, in order to statistically obtain the recognition success rate.

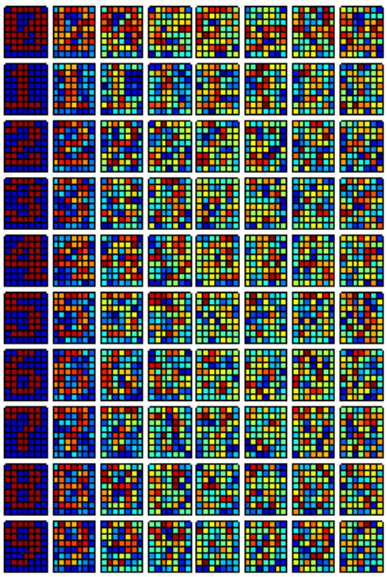


Fig. 1-10 Digit images with different noise levels from 0 to 7 in left-to-right order (one data in 100 groups)

Using the enhanced counterpropagation neural network [54] as shown in Fig. 1-11, the testing results are presented in Table 1-2 below. One may notice that the recognition error appears when patterns with level three noises are applied.

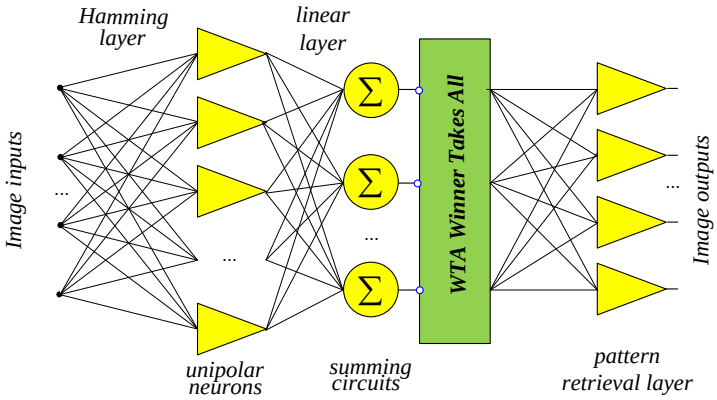


Fig. 1-11 The designed counterpropagation neural network architecture for the digit image recognition problem

Table 1-2 Success rates of the designed counterpropagation neural network for digit image recognition

Data / Digit	*Noise level 1*	*Noise level 2*	*Noise level 3*	*Noise level 4*	*Noise level 5*	*Noise level 6*	*Noise level 7*
Digit 0	100%	100%	100%	100%	100%	96%	97%
Digit 1	100%	100%	100%	100%	100%	100%	94%
Digit 2	100%	100%	100%	95%	91%	77%	82%
Digit 3	100%	100%	99%	92%	88%	84%	65%
Digit 4	100%	100%	100%	100%	100%	98%	96%
Digit 5	100%	100%	100%	100%	100%	95%	93%
Digit 6	100%	100%	100%	100%	92%	91%	88%
Digit 7	100%	100%	100%	100%	100%	98%	88%
Digit 8	100%	100%	99%	98%	83%	76%	67%
Digit 9	100%	100%	100%	100%	94%	91%	72%

Comparing human beings and computers in recognition of those noisy characters, Fig. 1-12 presents the retrieval results of 7th level noised digit images. Obviously it is totally a gamble for human beings to retrieve most of those images, but the designed counterpropagation neural networks can do the job correctly.

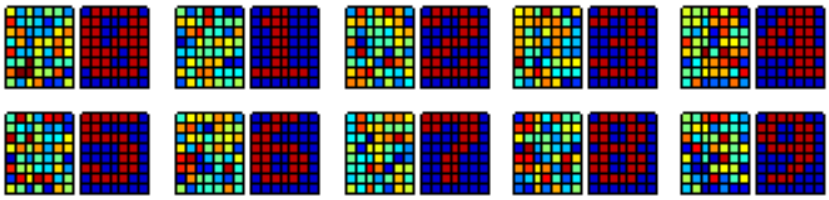


Fig. 1-12 Retrieval results of 7th level noised digit images

1.4 conclusion

The two examples above show the potentially good performance of neural networks in function approximation and pattern recognition problems. Because of the attractive and powerful nonlinear mapping ability, we are very interested in the research of neural networks, including both network architectures and learning algorithms. Besides, for better understanding of neural networks, we have also extended our research scope to several other methods of computational intelligence, such as fuzzy inference systems and radial basis function neural networks. Our recent publications (at the end of the dissertation), as listed at the end of the dissertation,

somehow prove our achievement in these realms.

In the dissertation, we will discuss how to design efficient and powerful algorithms for neural network learning. Especially, we will focus on the second order algorithms considering their high training efficiency and powerful search ability over first order algorithms. Our recently developed improved second order computation and the forward-only algorithm will be introduced as the recommended solutions to memory limitation problem and the computation redundancy problem, respectively, in second order algorithms.

CHAPTER 2

BACKGROUND

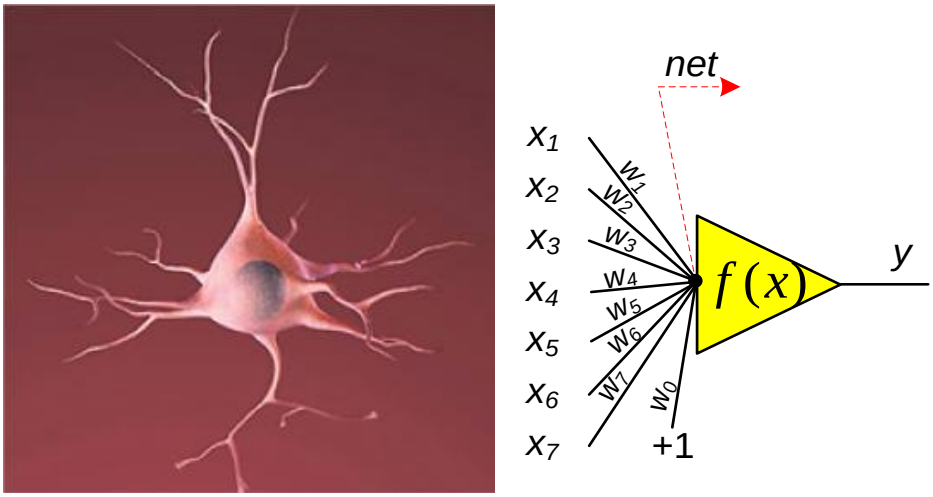
2.1 History

The history of the neural networks can be traced back to 1942, when Warren McCulloch and Walter Pitts proposed McCulloch-Pitts model, named Threshold Logic Unit (TLU) [55]. Originally, TLU was designed to perform simple logic operations, such as "&" and "|". In 1949, Donald Hebb mentioned the concept "synaptic modification" in his book "The organization of behavior" [56]. This concept was considered as a milestone during the development of neural networks. It is very similar with the analytical neuron models used today. In 1956, Albert Uttley reported that he successfully solved simple binary pattern classification problems using neural networks [57]. In 1958, Frank Rosenblatt introduced the important concept "Perceptron"; in the following four years, Frank Rosenblatt designed several learning algorithms for the perceptron model, in order to do binary pattern classification [58]. As another milestone, in 1960, Bernard Widrow and his student Ted Hoff proposed "ADALINE" model which consisted of linear neurons. Least mean squares method was designed to adjust the parameters of ADALINE model. Two years later (1962), as the expansion of ADALINE, Widrow and Hoff introduced "MADALINE" model which had two-layer architecture: multiple ADALINE units arranged in parallel as input layer and a single processor as output layer [59]. Based on ADALINE and MADALINE models, neural networks attracted lots of researchers and went through very fast development. Until 1969, Marvin Minsky and Seymour Papert proved the very limited power of

neural networks in their book “Perceptron” [60]. They showed that the single layer perceptron model was only capable of classifying the patterns which were linearly separable; for linearly inseparable patterns, such as the very simple XOR problem, the single layer perceptron model would be helpless. The theory proposed by Minsky and Papert stopped the development of neural networks for almost 10 years, until 1986, the invention of error backpropagation algorithm, proposed by David E. Rumelhart [61]. The error backpropagation algorithm dispersed the dark clouds on the field of neural networks and could be regarded as one of the most significant breakthroughs in neural network training. By using the sigmoidal shape activation function, such as tangent hyperbolic function, and incorporating with the gradient descent concept in numerical methods, the error backpropagation algorithm enhanced the power of neural networks significantly. Neural networks can not only be used for classifying binary linear patterns, but also be applied to approximate any nonlinear relationships. In the following 10 years, various learning algorithms [62-68] and network models [69-70] came out like the bamboo shoot after spring rain. Currently, error backpropagation (EBP) algorithms and multiplayer perceptron (MLP) networks are still the most popular learning algorithm and network architecture in practical applications.

2.2 Basic Concepts

As the basic unit of human brain, neural cells play the roles of signal transmission and storage. A neural cell mainly consists of cell body with lots of synapses around as shown in Fig. 2-1a. Extracting from the human brain model, a single neuron is made up of the linear/nonlinear activation function $f(x)$ (like cell body) and weighted connections (like synapses), as shown in Fig. 2-1b.



(a) Neural cell [71] (b) Neural model

Fig. 2-1 Neural cell in human brain and its simplified model in neural networks

Taking the neuron in Fig. 2-1b as an example, the two fundamental operations in a single neuron can be described as:

- Calculate the *net* value as sum of weighted input signals

$$net = \sum_{i=1}^{7} x_i w_i + w_0 \tag{2-1}$$

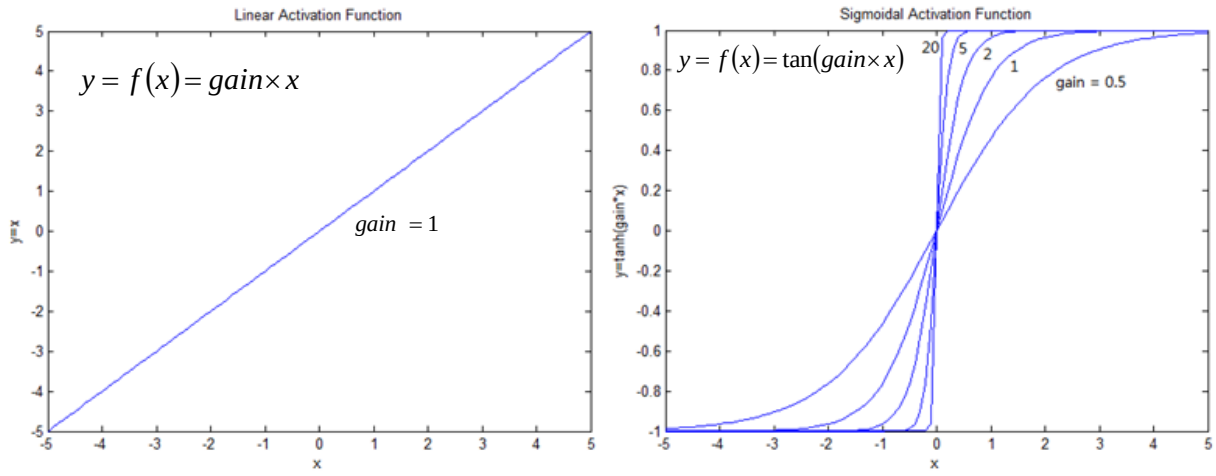
- Calculate the output *y*

$$y = f(net) \tag{2-2}$$

The activation function *f*(*x*) in equation (2-2) can be either linear function (equation 2-3) or sigmoidal shape function (equation 2-4), as shown in Fig. 2-2.

$$y = gain \times x \tag{2-3}$$

$$y = \tanh(gain \times x) = \frac{2}{1+\exp(-2 \times gain \times x)} - 1 \tag{2-4}$$



(a) Linear function (b) Sigmoidal function

Fig. 2-2 Different types of activation functions

It is quite straightforward that linear neurons (Fig. 2-2a), such as ADALINE model, have very limited power and can only handle patterns which are linear separable. On the other hand, sigmoidal shape functions (Fig. 2-2b), such as tangent hyperbolic function (equation 2-4), can be applied for nonlinear situations. It can be also noticed that, for sigmoidal shape functions, when the *gain* value becomes larger, the function behaves more like a step function.

For more than one neuron interconnected together, the two basic computations in equations (2-1) and (2-2) for each neuron remain the same as for a single neuron. The only difference is that the inputs of a neuron could be either network inputs or the outputs of neurons from the previous layers.

2.3 Network Architectures

Neural networks consist of neurons and their interconnections. Technically, the interconnections among neurons can be arbitrary. In the dissertation, we only discuss the feedforward neural networks where signals are propagated from input layer to output layer without feedback.

In this section, different types of neural network architectures are studied and compared

from the point of network efficiency, based on parity problems. The *N*-bit parity function can be interpreted as a mapping (defined by 2^N binary vectors) that indicates whether the sum of the *N* elements of every binary vector is odd or even. Parity-*N* problem is also considered to be one of the most difficult problems in neural network training [72-74].

2.3.1 *Simplified Patterns for Parity Problems*

One may notice that, in parity problems, input patterns which have the same sum of each input are going to have the same output. Therefore, considering all the weights on network inputs as "1", the number of training patterns of parity-*N* problem can be reduced from 2^N to *N*+1.

Fig. 2-3 shows both the original 8 training patterns and the simplified 4 training patterns in parity-3 problem. The two types of training patterns are identical.

Input	Sum of Inputs	Output
0 0 0	0	0
0 0 1	1	1
0 1 0	1	1
0 1 1	2	0
1 0 0	1	1
1 0 1	2	0
1 1 0	2	0
1 1 1	3	1

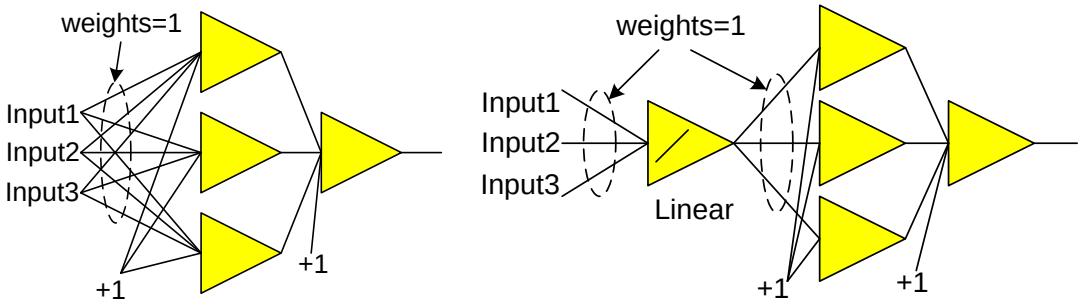
(a) Original patterns

Input	Output
0	0
1	1
2	0
3	1

(b) Simplified patterns

Fig. 2-3 Training patterns simplification for parity-3 problem

Based on this pattern simplification strategy, for parity-3 problem, instead of the network architecture in Fig. 2-4a, a linear neuron (with slope equal to 1) can be used as the network input (see Fig. 2-4b). The linear neuron works as a summator and it does not have bias input. All weights connected to the linear neuron, including input weights and output weights, are fixed as "1".



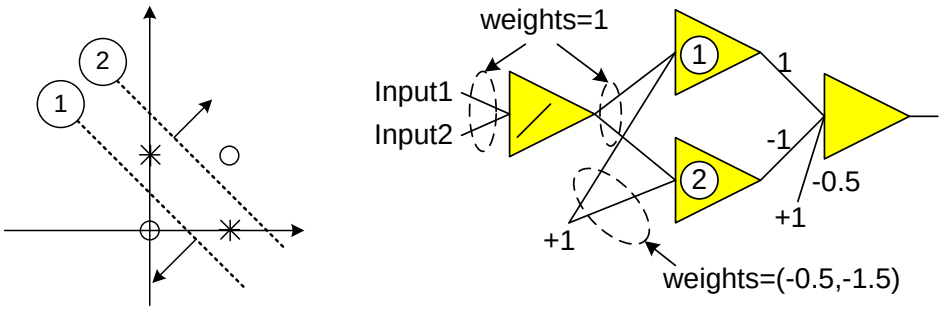
(a) Original parity-3 inputs (b) Simplified linear neuron inputs

Fig. 2-4 Two equivalent networks for parity-3 problem

2.3.2 *MLP Networks with One Hidden Layer*

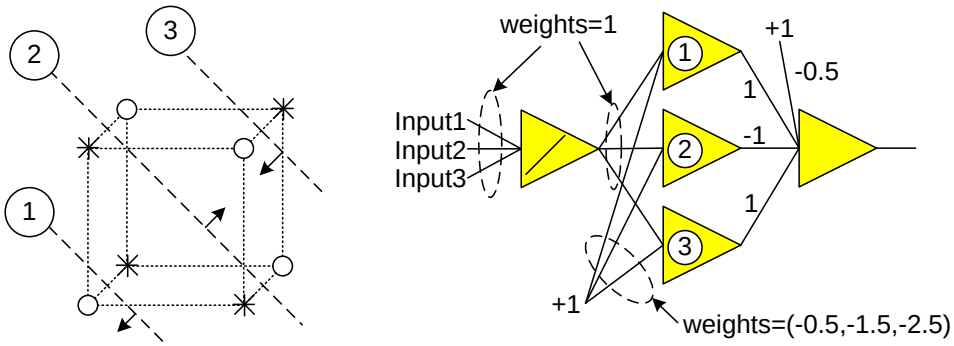
Multilayer perceptron (MLP) networks are the most popular networks because they are regularly formed and easy for programming. In MLP networks, neurons are organized layer by layer and there are no connections across layers.

Both parity-2 (XOR) and parity-3 problems can be visually illustrated in two and three dimensions respectively, as shown in Figs. 2-5 and 2-6. For parity-2 problem, each hidden neuron in Fig. 2-5b works as a separating **line** as shown in Fig. 2-5a and the output unit decides the values of separation area. Similarly, for parity-3 problem, each hidden unit in Fig. 2-6b represents a separating **plane** in Fig. 2-6a and the values of separation area are determined by the output unit.



(a) Graphical interpretation of separation (b) Designed neural network

Fig. 2-5 Analytical solution of parity-2 problem

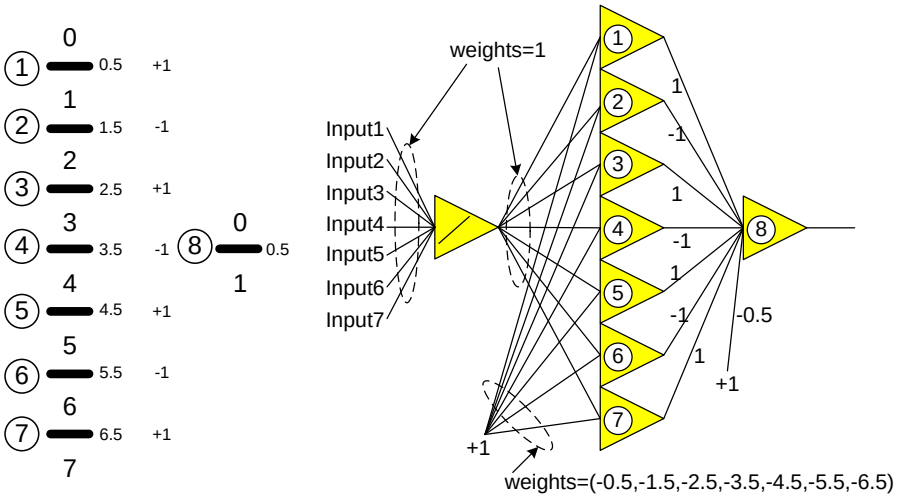


(a) Graphical interpretation of separation (b) Designed neural network

Fig. 2-6 Analytical solution of parity-3 problem

Using MLP networks with one hidden layer to solve the parity-7 problem, there could be at least 7 neurons in the hidden layer to separate the 8 training patterns (using the pattern simplification strategy described in Figs. 2-3 and 2-4), as shown in Fig. 2-7a.

In Fig. 2-7a, 8 patterns {0, 1, 2, 3, 4, 5, 6, 7} are separated by 7 neurons (bold line). The thresholds of the hidden neurons are {0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5}. Then summing the outputs of hidden neurons weighted by {1, -1, 1, -1, 1, -1, 1}, the net inputs at the output neurons could be only {0, 1}, which can be separated by the neuron with threshold 0.5. Therefore, parity-7 problem can be solved by the architecture shown in Fig. 2-7b.



(a) Analysis (b) Architecture

Fig. 2-7 Solving Parity-7 problem using MLP network with one hidden layer

Generally, if there are *n* neurons in MLP networks with single hidden layer, the largest possible parity-*N* problem that can be solved is

$$N = n - 1 \tag{2-5}$$

Where: *n* is the number of neurons and *N* is the number of dimensions of the parity problem.

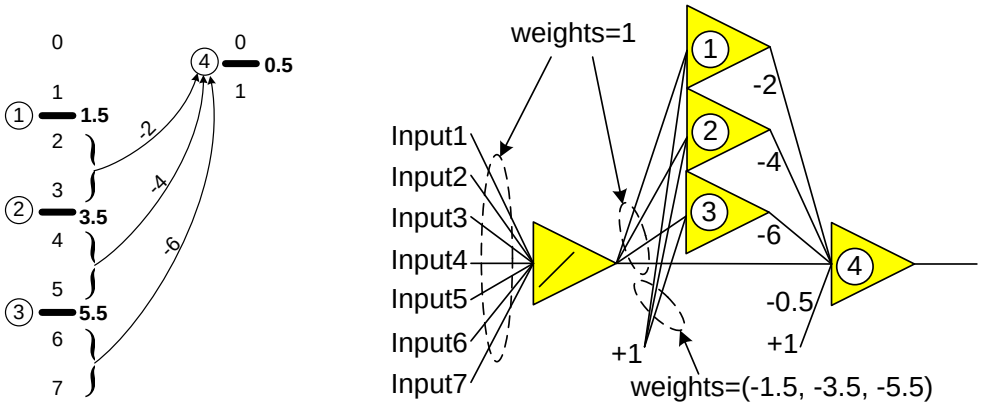
2.3.3 *BMLP Networks*

In MLP networks, if connections across layers are permitted, then networks have bridged multilayer perceptron (BMLP) topologies. BMLP networks are more powerful than traditional MLP networks if the number of neurons is the same.

2.3.3.1 BMLP Networks with One Hidden Layer

Considering BMLP networks with only one hidden layer, all network inputs are connected to both of the hidden neurons and the output neuron or neurons.

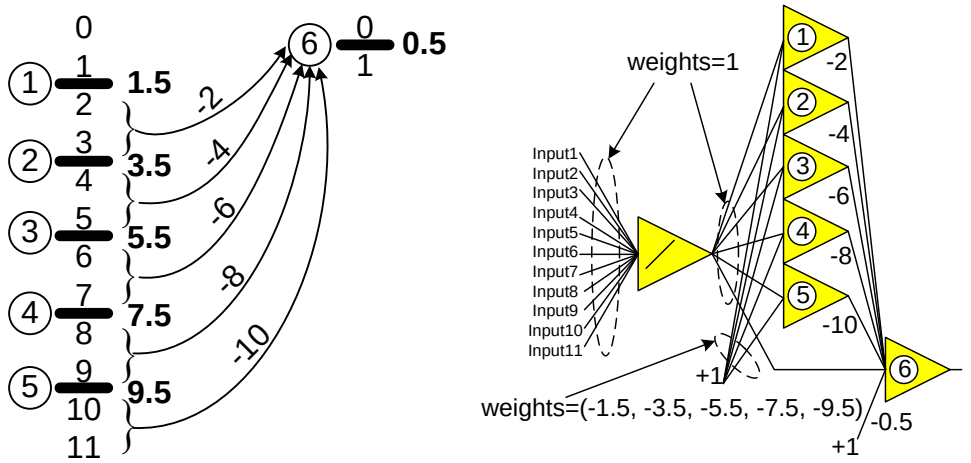
For parity-7 problem, the 8 simplified training patterns can be separated by 3 neurons to four sub patterns {0, 1}, {2, 3}, {4, 5} and {6, 7}. The threshold of the hidden neurons should be {1.5, 3.5, 5.5}. In order to transfer all sub patterns to the unique pattern {0, 1} for separation, patterns {2, 3}, {4, 5} and {6, 7} should be reduce by 2, 4 and 6 separately, which determines the weight values on connections between hidden neurons and output neurons. After pattern transformation, the unique pattern {0, 1} can be separated by the output neuron with threshold 0.5. The design process is shown in Fig. 2-8a and the corresponding solution architecture is shown in Fig. 2-8b.



(a) Analysis (b) Architecture

Fig. 2-8 Solve parity-7 problem using BMLP networks with one hidden layer

For parity-11 problem, similar analysis and related BMLP networks with single hidden layer solution architecture are presented in Fig. 2-9.



(a) Analysis (b) Architecture

Fig. 2-9 Solve parity-11 problem using BMLP networks with single hidden layer

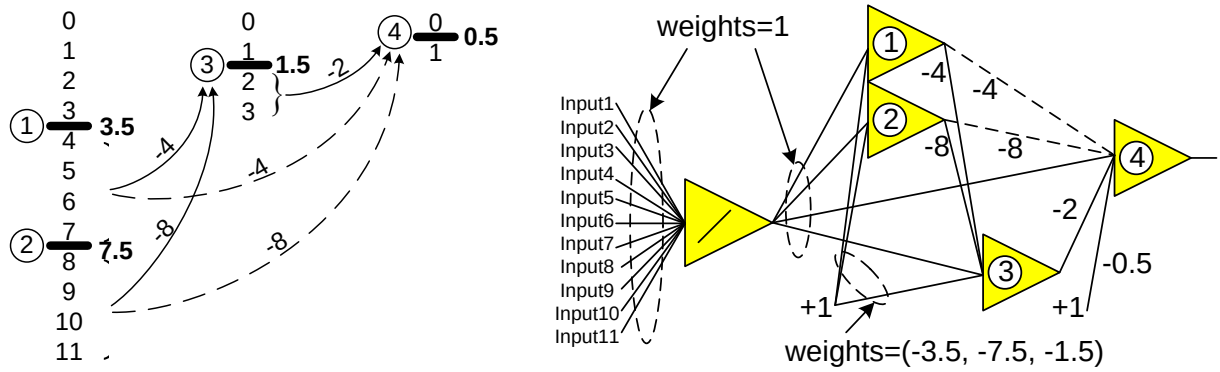
Generally, for *n* neurons in BMLP networks with one hidden layer, the largest parity-*N* problem that can be possibly solved is:

$$N = 2n - 1 \tag{2-6}$$

2.3.3.2 BMLP Networks with Multiple Hidden Layer

If BMLP networks have more than one hidden layers, then the further reduction of the number of neurons are possible, for solving the same problem.

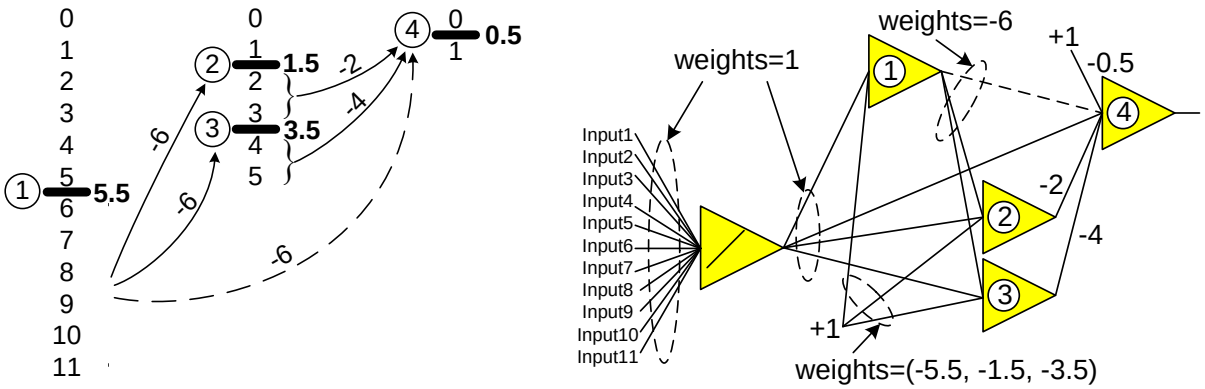
For parity-11 problem, using 4 neurons, in both 11=2=1=1 and 11=1=2=1 architectures, can find solutions. Considering the 11=2=1=1 network, the 12 simplified training patterns would be separated by two neurons at first, into {0, 1, 2, 3}, {4, 5, 6, 7} and {8, 9, 10 11}; the thresholds of the two neurons are 3.5 and 7.5 separately. Then, sub patterns {4, 5, 6, 7} and {8, 9, 10, 11} are transformed to {0, 1, 2, 3} by subtracting -4 and -8 separately, which determines the weight values on connections between the first hidden layer and followed layers. In the second hidden layer, one neuron is introduced to separate {0, 1, 2, 3} into {0, 1} and {2, 3}, with threshold 1.5. After that, sub pattern {2, 3} is transferred to {0, 1} by setting weight value as -2 on the connection between the second layer and the output layer. At last, output neuron with threshold 0.5 separates the pattern {0, 1}. The whole procedure is presented in Fig. 2-10 below.



(a) Analysis (b) Architecture

Fig. 2-10 Solve parity-11 problem using BMLP networks with two hidden layers, 11=2=1=1

Fig. 2-11 shows the 11=1=2=1 BMLP network with two hidden layers, for solving parity-11 problem.



(a) Analysis (b) Architecture

Fig. 2-11 Solve parity-11 problem using BMLP networks with two hidden layers, 11=1=2=1

Generally, considering the BMLP network with two hidden layers, the largest parity-*N* problem can be possibly solved is:

$$N = 2(m+1)(n+1)-1 \tag{2-7}$$

Where: *m* and *n* are the numbers of neurons in the two hidden layers, respectively.

For further derivation, one may notice that if there are *k* hidden layers and n_i is the number of neurons in the *i*-th hidden layer, where *i* is ranged from 1 to *k*, then

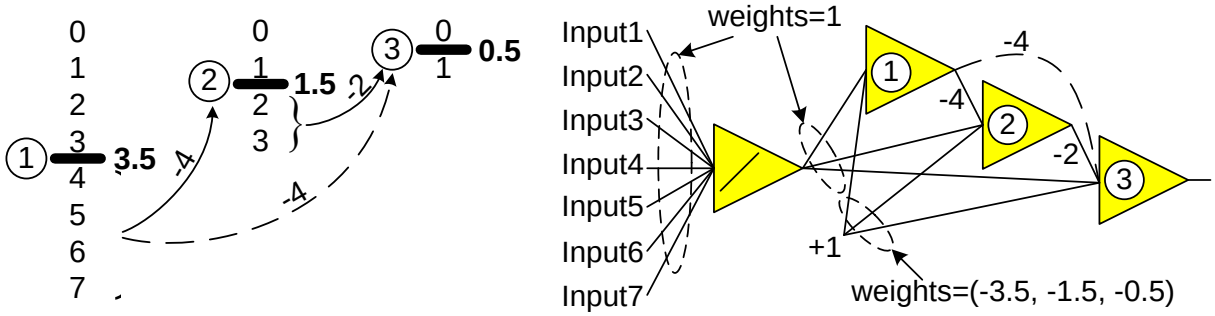
$$N = 2(n_1+1)(n_2+1)\cdots(n_i+1)\cdots(n_k+1)-1 \tag{2-8}$$

2.3.4 *FCC Networks*

Fully connected cascade (FCC) networks can solve problems using the smallest possible number of neurons. In the FCC networks, all possible routines are weighted, and each neuron contributes to a layer.

For parity-7 problem, the simplified 8 training patterns are divided by one neuron at first, as {0, 1, 2, 3} and {4, 5, 6, 7}; the threshold of the neuron is 3.5. Then the sub pattern {4, 5, 6, 7} is transferred to {0, 1, 2, 3} by weights equal to -4, connected to the followed neurons. Again, by

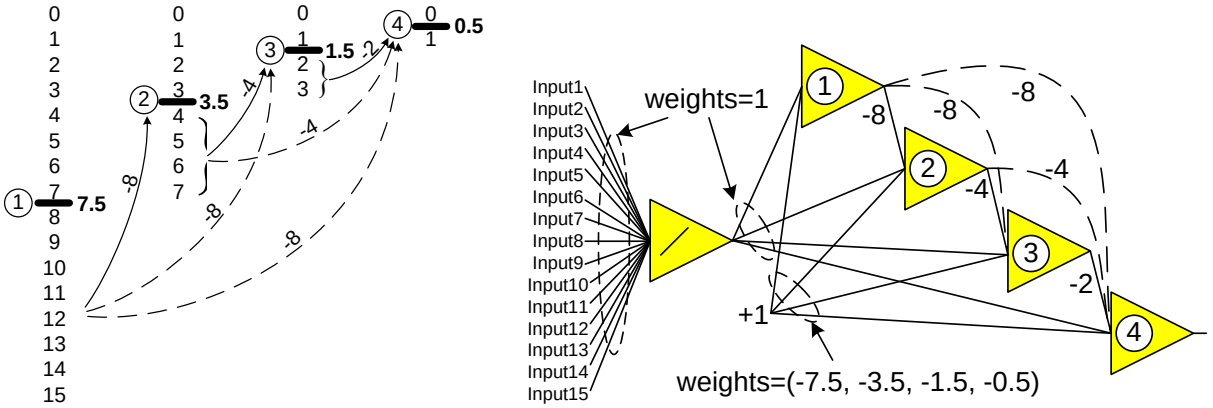
using another neuron, the patterns in the second hidden layer {0, 1, 2, 3} can be separated as {0, 1} and {2, 3}; the threshold of the neuron is 1.5. In order to transfer the sub pattern {2, 3} to {1, 2}, 2 should be subtracted from sub pattern {2, 3}, which determines that the weight between the second layer and the output layer is -2. At last, output neurons with threshold 0.5 is used to separate the pattern {0, 1}, see Fig. 2-12.



(a) Analysis (b) Architecture

Fig. 2-12 Solve parity-7 problem using FCC networks

Fig. 2-13 shows the solution of parity-15 problem using FCC networks.



(a) Analysis (b) Architecture

Fig. 2-13 Solve parity-15 problem using FCC networks

Considering the FCC networks as special BMLP networks with only one neuron in each hidden layer, for *n* neurons in FCC networks, the largest *N* for parity-*N* problem can be derived from equation (2-8) as:

$$N = 2\underbrace{(1+1)(1+1)\cdots(1+1)(1+1)}_{n-1} - 1 \tag{2-9}$$

or

$$N = 2^n - 1 \tag{2-10}$$

2.3.5 *Comparison of Different Topologies*

Table 2-1 concludes the analysis of network efficiency above and the largest parity-*N* problem that can be solved with a given network structure. For example, with 5 neurons: the MLP network with only one hidden layer can solve parity-4 problem (4-4-1 network); BMLP network with a single hidden layer can solve parity-11 problem (11=4=1 network); BMLP network with two hidden layers can solve parity-15 problem (15=3=1=1 network or 15=1=3=1 network) or parity-17 problem (17=2=2=1 network); FCC network can solve parity-31 problem at most (31=1=1=1=1=1 network).

Table 2-1 Different architectures for solving parity-*N* problem

Network Architectures	***Parameters***	***Parity-N Problem***
MLP with single hidden layer	*n* neurons	$n-1$
BMLP with single hidden layer	*n* neurons	$2n+1$
BMLP with multiple hidden layer	*k* hidden layers, each with n_i neurons	$2(n_1+1)(n_2+1)\cdots(n_{k-1}+1)(n_k+1)-1$
FCC	*n* neurons	2^n-1

Based on the comparison results shown in Table 2-1, one may draw the conclusion that, with more connections across layers, the networks become more powerful. The FCC architecture is the most powerful and can solve problems with much less number of neurons.

2.4 Learning Algorithms

Many methods have already been developed for neural networks training [62-68]. In this dissertation, we will focus on the gradient descent based optimization methods.

2.4.1 *Introduction*

Steepest descent algorithm, also known as error backpropagation algorithm [61], is the most popular algorithm for neural network training; however, it is also known as an inefficient algorithm because of its slow convergence.

There are two main reasons for the slow convergence: the first reason is that its step sizes should be adequate to the gradients as shown in Fig. 2-14. Logically, small step size should be taken where the gradient is steep, so as not to rattle out of the required minima (because of oscillation). So if the step size is a constant, it needs to be chosen small. Then, in the place where the gradient is gentle, the training process would be very slow. The second reason is that the curvature of the error surface may not be the same in all directions, such as the Rosenbrock function, so the classic "error valley" problem [75] may exist and may result in the slow convergence.

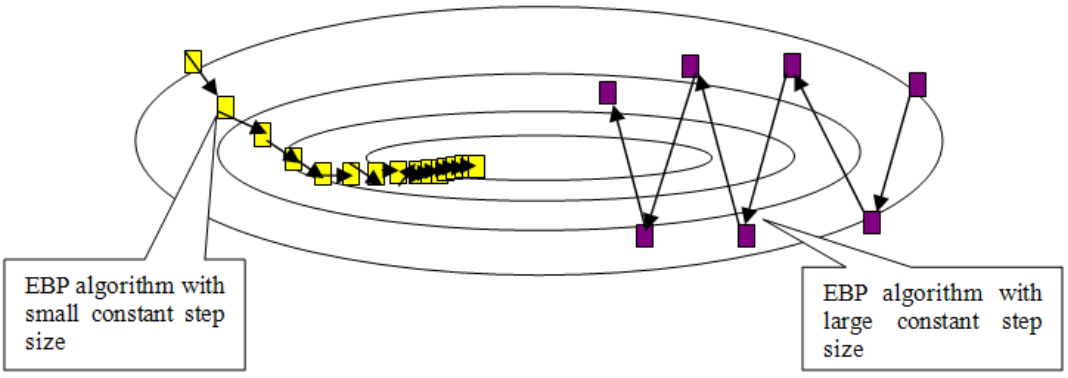


Fig. 2-14 Searching process of the steepest descent method with different learning constants: yellow trajectory (left) is for small learning constant which leads to slow convergence; purple trajectory (right) is for large learning constant which causes oscillation (divergence)

The slow convergence of the steepest descent method can be greatly improved by Gauss-Newton algorithm [75]. Using second order derivatives of error function to "naturally" evaluate the curvature of error surface, The Gauss-Newton algorithm can find proper step sizes for each direction and converge very fast. Especially, if the error function has a quadratic surface, it can converge directly in the first iteration. But this improvement only happens when the quadratic approximation of error function is reasonable. Otherwise, Gauss-Newton algorithm would be mostly divergent.

Levenberg Marquardt algorithm [24-25][76] blends the steepest descent method and Gauss-Newton algorithm. Fortunately, it inherits the speed advantage of the Gauss-Newton algorithm and the stability of the steepest descent method. It's more robust than the Gauss-Newton algorithm, because in many cases it can converge well even if the error surface is much more complex than quadratic situation. Although Levenberg Marquardt algorithm tends to be a bit slower than Gauss-Newton algorithm (in convergent situation), it converges much faster than the steepest descent method.

The basic idea of Levenberg Marquardt algorithm is that it performs a combined training process: around the area with complex curvature, Levenberg Marquardt algorithm switches to steepest descent algorithm, until the local curvature is proper to make a quadratic approximation; then it approximately becomes Gauss-Newton algorithm which can speed up the convergence significantly.

In the following sections, the four basic gradient descent methods will be introduced, including (1) steepest descent method; (2) Newton method; (3) Gaussian-Newton algorithm and (4) Levenberg Marquardt algorithm.

Sum square error (SSE) E is defined to evaluate the training process, as the object

function. For all training patterns and network outputs, it is calculated by

$$E(\boldsymbol{x},\boldsymbol{w}) = \frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M} e_{p,m}^2 \tag{2-11}$$

Where: $\boldsymbol{x}$ and $\boldsymbol{w}$ are the input vector and weight vector respectively; p is the index of training patterns, from 1 to P, where P is the number of training patterns; m is the index of outputs, from 1 to M, where M is the number of outputs; $e_{p,m}$ is the training error at output m when applying pattern p and it is defined as

$$e_{p,m} = d_{p,m} - o_{p,m} \tag{2-12}$$

Where: $\boldsymbol{d}$ is the desired output vector and $\boldsymbol{o}$ is the actual output vector.

2.4.2 *Steepest Descent Algorithm*

Steepest descent algorithm is a first order algorithm. It uses the first order derivative of total error function to find the minima in error space. Normally, gradient $\boldsymbol{g}$ is defined as the first order derivative of total error function (2-11)

$$\boldsymbol{g} = \frac{\partial E(\boldsymbol{x},\boldsymbol{w})}{\partial \boldsymbol{w}} = \left[\frac{\partial E}{\partial w_1} \quad \frac{\partial E}{\partial w_2} \quad \cdots \quad \frac{\partial E}{\partial w_N}\right]^T \tag{2-13}$$

Where: N is the number of weights.

With the definition of gradient $\boldsymbol{g}$ in (2-13), the update rule of steepest descent algorithm could be written as:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \boldsymbol{g}_k \tag{2-14}$$

Where: α is the learning constant (step size) and k is the index of training iterations.

The training process of steepest descent algorithm is asymptotic convergence so it never reaches the minima. Around the solution, all the elements of gradient vector $\boldsymbol{g}$ would be very

small and there would be very tiny weight changing.

2.4.3 *Newton Method*

Newton method assumes that all the gradient components $\boldsymbol{g}=\{g_1, g_2 \ldots g_N\}$ are function of weights and all weights are linearly independent:

$$\begin{cases} g_1 = F_1(w_1, w_2 \cdots w_N) \\ g_2 = F_2(w_1, w_2 \cdots w_N) \\ \cdots \\ g_N = F_N(w_1, w_2 \cdots w_N) \end{cases} \tag{2-15}$$

Where: $\{F_1, F_2 \ldots F_N\}$ are nonlinear relationships between weights and related gradient components.

Unfold each g_i $(i=1, 2 \ldots N)$ in equations (2-15) by Taylor series and take the first order approximation:

$$\begin{cases} g_1 \approx g_{1,0} + \dfrac{\partial g_1}{\partial w_1}\Delta w_1 + \dfrac{\partial g_1}{\partial w_2}\Delta w_2 + \cdots + \dfrac{\partial g_1}{\partial w_N}\Delta w_N \\ g_2 \approx g_{2,0} + \dfrac{\partial g_2}{\partial w_1}\Delta w_1 + \dfrac{\partial g_2}{\partial w_2}\Delta w_2 + \cdots + \dfrac{\partial g_2}{\partial w_N}\Delta w_N \\ \cdots \\ g_N \approx g_{N,0} + \dfrac{\partial g_N}{\partial w_1}\Delta w_1 + \dfrac{\partial g_N}{\partial w_2}\Delta w_2 + \cdots + \dfrac{\partial g_N}{\partial w_N}\Delta w_N \end{cases} \tag{2-16}$$

By combining the definition of gradient vector $\boldsymbol{g}$ in (2-13), it could be determined that

$$\frac{\partial g_i}{\partial w_j} = \frac{\partial\left(\dfrac{\partial E}{\partial w_j}\right)}{\partial w_j} = \frac{\partial^2 E}{\partial w_i \partial w_j} \tag{2-17}$$

Where: i and j are the indices of weights, from 1 to N.

By inserting equation (2-17) to (2-16):

$$\begin{cases} g_1 \approx g_{1,0} + \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ g_2 \approx g_{2,0} + \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ g_N \approx g_{N,0} + \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \tag{2-18}$$

Comparing with the steepest descent method, the second order derivatives of the total error function need to be calculated for each component of gradient vector.

In order to get the minima of total error function *E*, each element of the gradient vector should be zero. Therefore, left sides of the equations (2-18) are all zero, then

$$\begin{cases} 0 \approx g_{1,0} + \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ 0 \approx g_{2,0} + \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ 0 \approx g_{N,0} + \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \tag{2-19}$$

By combining equation (2-13) with (2-19)

$$\begin{cases} -\dfrac{\partial E}{\partial w_1} = -g_{1,0} \approx \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ -\dfrac{\partial E}{\partial w_2} = -g_{2,0} \approx \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ -\dfrac{\partial E}{\partial w_N} = -g_{N,0} \approx \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \tag{2-20}$$

There are N equations for N parameters so that all Δw_i can be calculated. With the solutions, the weight space can be updated iteratively.

Equations (2-20) can be also written in matrix form

$$\begin{bmatrix} -g_1 \\ -g_2 \\ \cdots \\ -g_N \end{bmatrix} = \begin{bmatrix} -\dfrac{\partial E}{\partial w_1} \\ -\dfrac{\partial E}{\partial w_2} \\ \cdots \\ -\dfrac{\partial E}{\partial w_N} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial^2 E}{\partial w_1^2} & \dfrac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_1 \partial w_N} \\ \dfrac{\partial^2 E}{\partial w_2 \partial w_1} & \dfrac{\partial^2 E}{\partial w_2^2} & \cdots & \dfrac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial^2 E}{\partial w_N \partial w_1} & \dfrac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \times \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \cdots \\ \Delta w_N \end{bmatrix} \tag{2-21}$$

Where: the square matrix is Hessian matrix $\boldsymbol{H}$ ($N \times N$):

$$\boldsymbol{H} = \begin{bmatrix} \dfrac{\partial^2 E}{\partial w_1^2} & \dfrac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_1 \partial w_N} \\ \dfrac{\partial^2 E}{\partial w_2 \partial w_1} & \dfrac{\partial^2 E}{\partial w_2^2} & \cdots & \dfrac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial^2 E}{\partial w_N \partial w_1} & \dfrac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \tag{2-22}$$

By Combining equations (2-13) and (2-22) with equation (2-21)

$$-\boldsymbol{g} = \boldsymbol{H}\Delta\boldsymbol{w} \tag{2-23}$$

So

$$\Delta\boldsymbol{w} = -\boldsymbol{H}^{-1}\boldsymbol{g} \tag{2-24}$$

Therefore, update rule for Newton method is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k \tag{2-25}$$

As the second order derivatives of total error function, Hessian matrix $\boldsymbol{H}$ gives the proper evaluation on the change of gradient vector. By comparing equations (2-14) and (2-25), one may notice that well-matched step sizes are given by the inverted Hessian matrix.

2.4.4 *Gaussian-Newton Algorithm*

If Newton method is applied for weight updating, in order to get Hessian matrix $\boldsymbol{H}$, the second order derivatives of total error function have to be calculated and it could be very complicated. In order to simplify the calculating process, Jacobian matrix $\boldsymbol{J}$ is introduced as

$$\boldsymbol{J}=\begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \cdots & \frac{\partial e_{1,1}}{\partial w_N} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \cdots & \frac{\partial e_{1,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1,M}}{\partial w_1} & \frac{\partial e_{1,M}}{\partial w_2} & \cdots & \frac{\partial e_{1,M}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P,1}}{\partial w_1} & \frac{\partial e_{P,1}}{\partial w_2} & \cdots & \frac{\partial e_{P,1}}{\partial w_N} \\ \frac{\partial e_{P,2}}{\partial w_1} & \frac{\partial e_{P,2}}{\partial w_2} & \cdots & \frac{\partial e_{P,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P,M}}{\partial w_1} & \frac{\partial e_{P,M}}{\partial w_2} & \cdots & \frac{\partial e_{P,M}}{\partial w_N} \end{bmatrix} \tag{2-26}$$

By integrating equations (2-11) and (2-13), elements of gradient vector can be calculated as

$$g_i=\frac{\partial E}{\partial w_i}=\frac{\partial\left(\frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M}e_{p,m}^2\right)}{\partial w_i}=\sum_{p=1}^{P}\sum_{m=1}^{M}\left(\frac{\partial e_{p,m}}{\partial w_i}e_{p,m}\right) \tag{2-27}$$

Combining equations (2-26) and (2-27), the relationship between Jacobian matrix $\boldsymbol{J}$ and gradient vector $\boldsymbol{g}$ would be

$$\mathbf{g}=\mathbf{J}^T\mathbf{e} \tag{2-28}$$

Where: error vector $\boldsymbol{e}$ has the form

$$\boldsymbol{e} = \begin{bmatrix} e_{1,1} \\ e_{1,2} \\ \cdots \\ e_{1,M} \\ \cdots \\ e_{P,1} \\ e_{P,2} \\ \cdots \\ e_{P,M} \end{bmatrix} \tag{2-29}$$

Inserting equation (2-11) into (2-22), the element at *i*-th row and *j*-th column of Hessian matrix can be calculated as

$$h_{i,j} = \frac{\partial^2 E}{\partial w_i \partial w_j} = \frac{\partial^2 \left(\frac{1}{2} \sum_{p=1}^{P} \sum_{m=1}^{M} e_{p,m}^2 \right)}{\partial w_i \partial w_j} = \sum_{p=1}^{P} \sum_{m=1}^{M} \frac{\partial e_{p,m}}{\partial w_i} \frac{\partial e_{p,m}}{\partial w_j} + S_{i,j} \tag{2-30}$$

Where: $S_{i,j}$ is equal to

$$S_{i,j} = \sum_{p=1}^{P} \sum_{m=1}^{M} \frac{\partial^2 e_{p,m}}{\partial w_i \partial w_j} e_{p,m} \tag{2-31}$$

As the basic assumption of Newton's method is that $S_{i,j}$ is closed to zero and the relationship between Hessian matrix $\boldsymbol{H}$ and Jacobian matrix $\boldsymbol{J}$ can be rewritten as

$$\boldsymbol{H} \approx \boldsymbol{J}^T \boldsymbol{J} \tag{2-32}$$

By combining equations (2-25), (2-28) and (2-32), the update rule of Gaussian-Newton algorithm is presented as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T \mathbf{J}_k \right)^{-1} \mathbf{J}_k^T \mathbf{e}_k \tag{2-33}$$

Obviously, the advantage of Gaussian-Newton algorithm over the standard Newton

method (equation 2-25) is that the former one doesn't require the calculation of second order derivatives of the total error function, by introducing Jacobian matrix $\boldsymbol{J}$ instead. However, Gaussian-Newton algorithm still faces the same convergent problem like Newton algorithm for complex error space optimization. Mathematically, the problem can be interpreted as: matrix $\boldsymbol{J}^T\boldsymbol{J}$ may be not invertible.

2.4.5 *Levenberg Marquardt Algorithm*

In order to make sure that the approximated Hessian matrix $\boldsymbol{J}^T\boldsymbol{J}$ is invertible, Levenberg Marquardt algorithm introduces another approximation to Hessian matrix

$$\boldsymbol{H} \approx \boldsymbol{J}^T\boldsymbol{J} + \mu \boldsymbol{I} \tag{2-34}$$

Where: μ is always positive, called combination coefficient and $\boldsymbol{I}$ is the identity matrix.

From equation (2-34), one may notice that the elements on the main diagonal of the approximated Hessian matrix will be larger than zero. Therefore, with this approximation (equation 2-34), it can be sure that matrix $\boldsymbol{H}$ is always invertible.

By combining equations (2-33) and (2-34), the update rule of Levenberg Marquardt algorithm can be presented as:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k + \mu\mathbf{I}\right)^{-1}\mathbf{J}_k^T\mathbf{e}_k \tag{2-35}$$

As the combination of steepest descent algorithm and Gaussian-Newton algorithm, Levenberg Marquardt algorithm switches between the two algorithms during the training process. When combination coefficient μ is very small (nearly zero), equation (2-35) is approaching to equation (2-33) and Gaussian-Newton algorithm is used. When combination coefficient μ is very large, equation (2-35) approximates to equation (2-14) and the steepest descent method is used.

If the combination coefficient μ in equation (2-35) is very big, it can be interpreted as learning coefficient in the steepest descent method (2-14):

$$\alpha = \frac{1}{\mu} \tag{2-36}$$

2.4.6 *Comparison of Different Algorithms*

Table 2-2 summarizes the update rules and their properties of the four algorithms above.

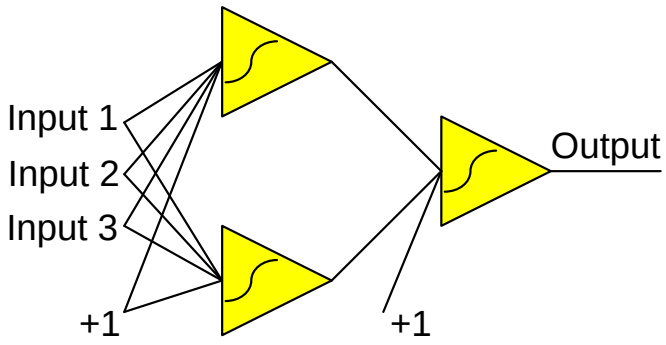
Table 2-2 Specifications of different learning algorithms

Learning Algorithms	***Update Rules***	***Convergent Rate***	***Computation Complexity***
EBP algorithm	$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \boldsymbol{g}_k$	Stable, slow	Gradient
Newton algorithm	$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k$	Unstable, fast	Gradient and Hessian
Gaussian-Newton algorithm	$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T \mathbf{J}_k\right)^{-1} \mathbf{J}_k^T \mathbf{e}_k$	Unstable, fast	Jacobian
Levenberg Marquardt algorithm	$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T \mathbf{J}_k + \mu \mathbf{I}\right)^{-1} \mathbf{J}_k^T \mathbf{e}_k$	Stable, fast	Jacobian

In order to compare the behavior of different learning algorithms, let us use the parity-3 problem as an example. The training patterns of pariry-3 problem are shown in Fig. 2-15a.

Inputs			Outputs
-1	-1	-1	-1
-1	-1	1	1
-1	1	-1	1
-1	1	1	-1
1	-1	-1	1
1	-1	1	-1
1	1	-1	-1
1	1	1	1

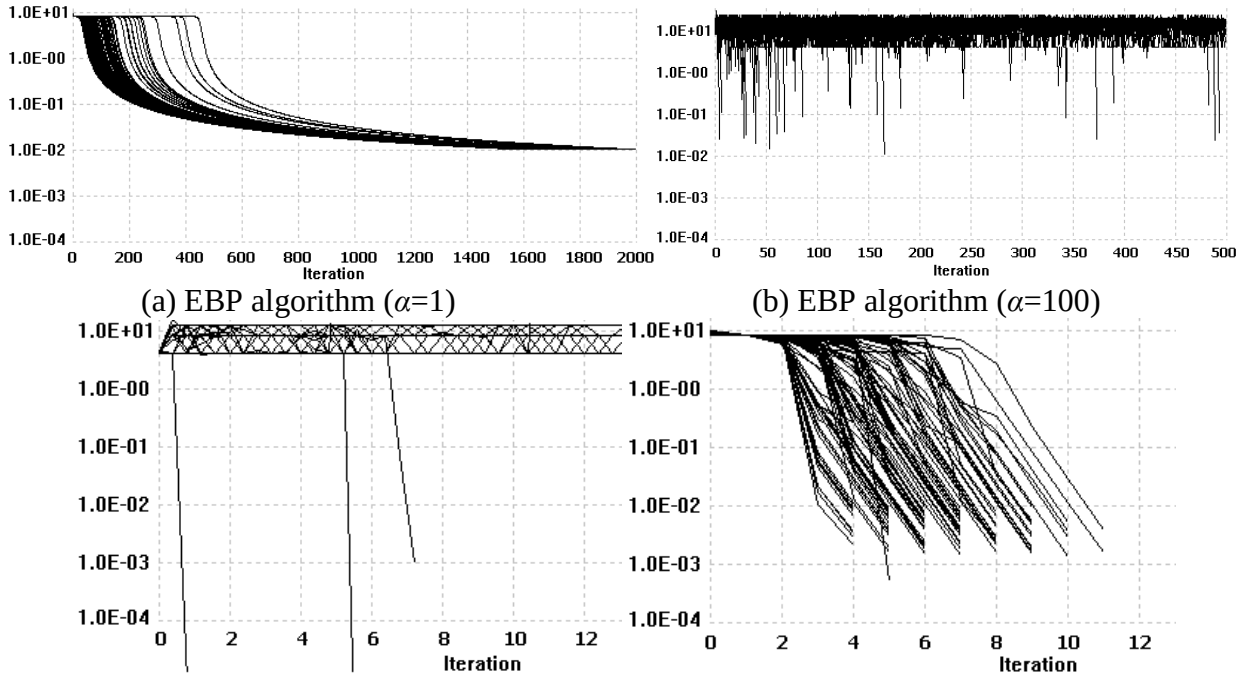
(a) Training patterns



(b) MLP network 3-2-1

Fig. 2-15 Parity-3 data and network architecture

Three neurons in 3-2-1 MLP network, as shown in Fig. 2-15b, are used for training and the required training error is 0.01. Convergent rates are tested by repeating each case for 100 trials with randomly generated initial weights.



(a) EBP algorithm (α=1) (b) EBP algorithm (α=100)

(c) Gaussian-Newton algorithm (d) Levenberg Marquardt algorithm

Fig. 2-16 Training results of parity-3 problem

Table 2-3 Comparison among different learning algorithms for parity-3 problem

Algorithms	***Convergence Rate***	***Average Iteration***	***Average Time (ms)***
EBP algorithm (α=1)	100%	1646.52	320.6
EBP algorithm (α=100)	79%	171.48	36.5
Gauss-Newton algorithm	3%	4.33	1.2
LM algorithm	100%	6.18	1.6

The training results are shown in Fig. 2-16 and the comparison is presented in Table 2-3. It can be concluded that:

- For EBP algorithm, the larger the training constant α is, the faster and less stable the training process will be (Figs. 2-16a and 2-16b);
- Gaussian-Newton algorithm computes very fast, but it seldom converges (Fig. 2-16c);

- Levenberg Marquardt algorithm is much faster than EBP algorithm and more stable than Gaussian-Newton algorithm (Fig. 2-16d).

For more complex parity-*N* problems, Gaussian-Newton algorithm cannot converge at all, and EBP algorithm also becomes more time-consuming and harder to find solutions; while Levenberg Marquardt algorithm can still perform successful training.

Another example is the two-spiral classification problem [77] which is often considered as a very complex benchmark to evaluate the efficiency of learning algorithms and network architectures. As shown in Fig. 2-17, the two-spiral problem is purposed to separate two groups of twisted points (red circles and blue stars).

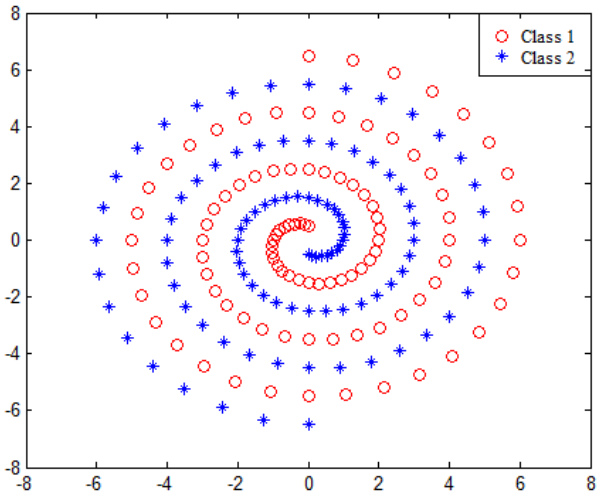
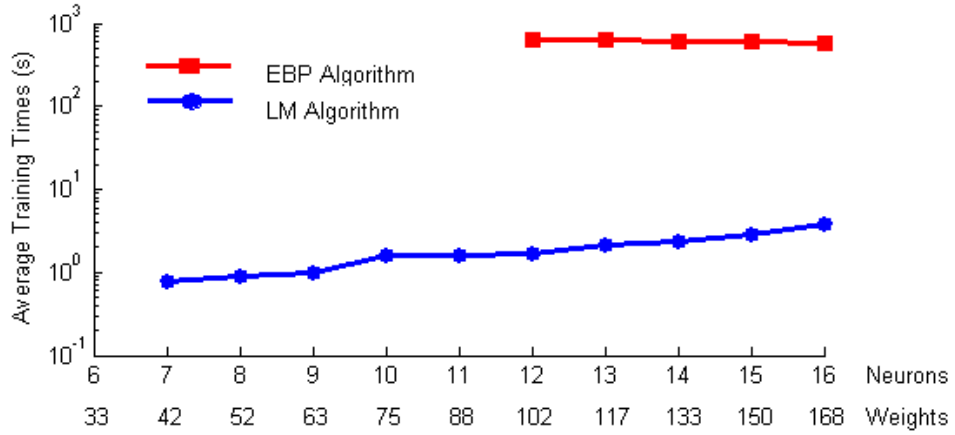


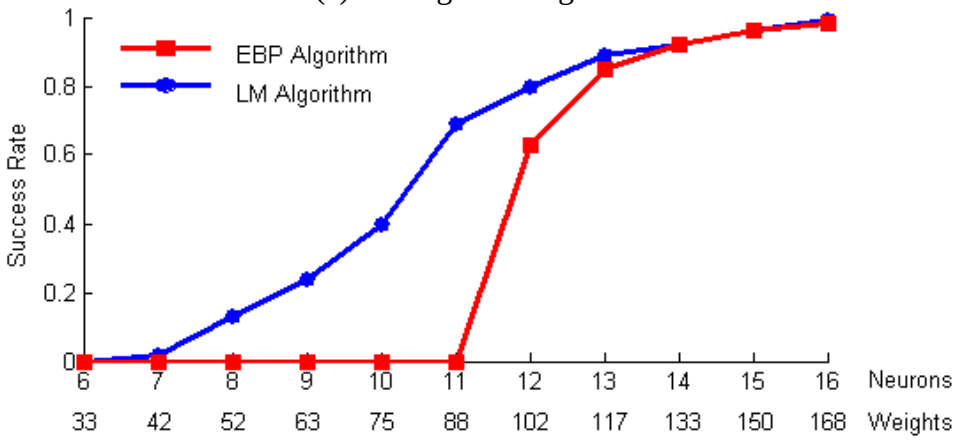
Fig. 2-17 Two-spiral problem: separation of two groups of points

Fig. 2-18 presents the training results the two-spiral problem, using EBP and LM algorithms. In both cases, fully connected cascade (FCC) networks were used; the desired sum squared error was 0.01; the maximum number of iteration was 1,000,000 for EBP algorithm and 1,000 for LM algorithm. The LM algorithm was implemented by NBN algorithm [78-79], so as to be able to handle FCC networks. EBP algorithm not only requires much more time than LM algorithm (Fig. 2-18a), but also is not able to solve the problem unless excessive number of

neurons is used. EBP algorithm requires at least 12 neurons and the second order algorithm can solve it in much smaller networks, such as 7 neurons (Fig .2-18b).



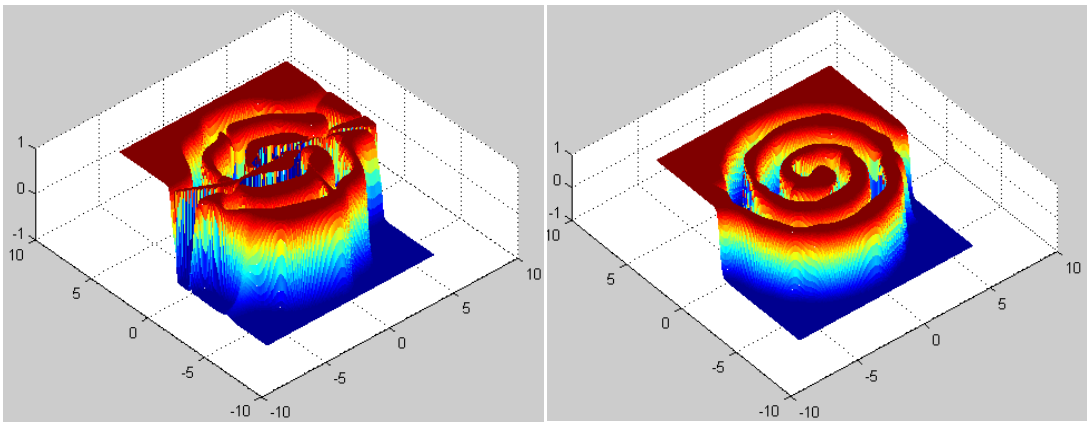
(a) Average training time



(b) Success rate

Fig. 2-18 Comparison between EBP algorithm and LM algorithm, for different number of neurons in fully connected cascade networks

Fig. 2-19 shows the training results of the two-spiral problem, using 16 neurons in fully connected cascade network, for both EBP algorithm and LM algorithm. One may notice that, with the same topology, LM algorithm is able to find better solutions than those found using EBP algorithm.



(a) EBP algorithm (b) LM algorithm

Fig. 2-19 Training results of the two-spiral problem with 16 neurons in fully connected cascade network

By conclusion, Levenberg Marquardt algorithm is the most efficient gradient based algorithm and it is recommended for neural network learning; however, it needs much more challenging computation than first order gradient methods.

2.5 Generalization Ability

Neural networks can work as universal approximator [22], but it happens only after successful training/learning process. The generalization is defined to evaluate the ability of trained neural networks to successfully handle new patterns which are not used for training. In order to obtain neural networks with good generalization ability, the over-fitting problems [28] should be avoided during the training process.

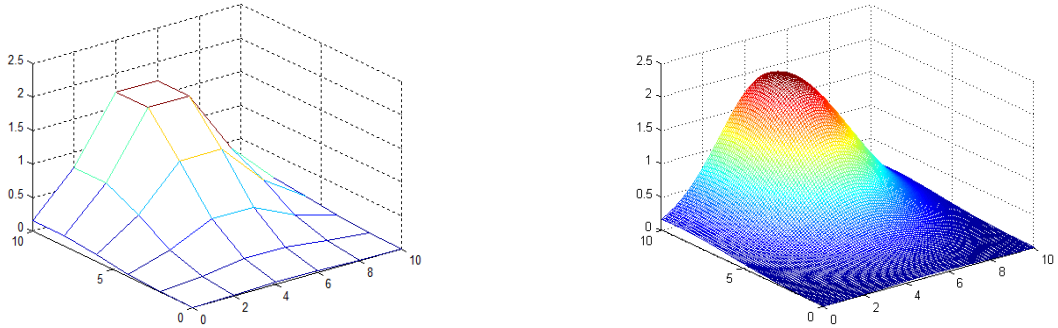
2.5.1 *The Over-fitting Problem*

The over-fitting problem is critical for designing neural networks with good generalization ability. When over-fitting happens, the trained neural networks can fit the training patterns very preciously, but they response poorly for new patterns which are not used for training.

Let us have an example to illustrate the existence of the over-fitting problems in neural network training. The purpose of the example is to approximate the function below

$$f(x,y) = 2\exp\left(-0.05(x-9)^2 - 0.1(y-5)^2 + 10^{-9}\right) \tag{2-37}$$

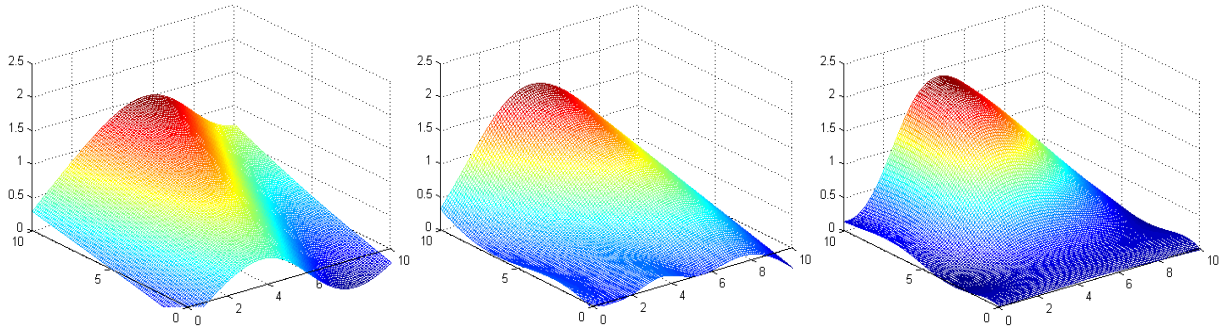
As shown in the Fig. 2-20, the training patterns consist of 6×6=36 points (Fig. 2-20a) uniformly distributed in sampling range $x \in [0, 10]$ and $y \in [0, 10]$. After training, another 101×101=10,201 points (Fig. 2-20b, also uniformly distributed) in the same sampling range are applied to test the trained neural networks.



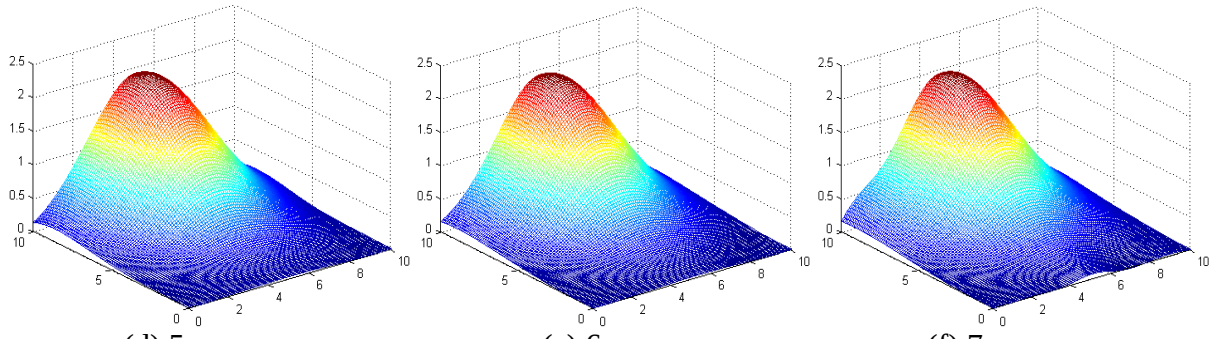
(a) Training patterns, 6×6=36 points (b) Testing patterns, 101×101=10,201 points

Fig. 2-20 Function approximation problem

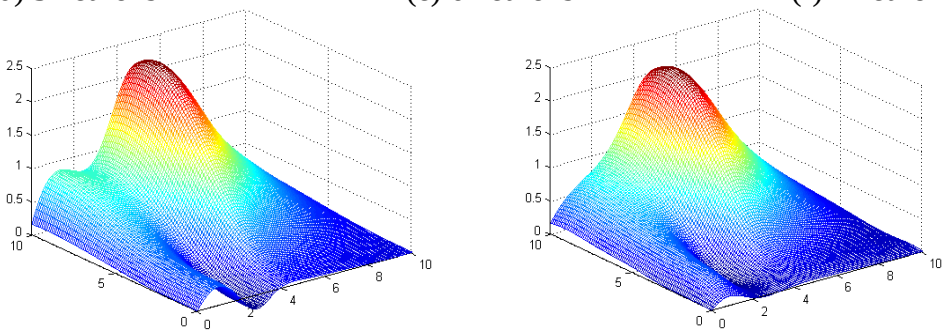
Using the most powerful neural network architecture (as analyzed in section 2.3.5), fully connected cascade (FCC) networks, the testing results of trained networks consisting of different number of neurons are shown in Fig. 2-21.



(a) 2 neurons (b) 3 neurons (c) 4 neurons



(d) 5 neurons (e) 6 neurons (f) 7 neurons



(g) 8 neurons (h) 9 neurons

Fig. 2-21 Approximation results of FCC networks with different number of neurons

Table 2-4 presents the training and testing sum square errors (SSEs) of FCC networks with different number of neurons.

Table 2-4 Training/testing SSEs of different sizes of FCC networks

Number of Neurons	***Training SSEs***	***Testing SSEs***
2	2.43055	678.7768
3	1.17843	346.0761
4	0.13682	49.6832
5	0.00591	1.7712
6	0.00022	0.2809
7	0.00008	7.3590
8	0.00003	249.3378
9	0.00000008	142.3883

From the results presented in Fig. 2-21 and Table 2-4, one may notice that, as the network size increases, the training errors keep decreasing stably; however, the testing errors decrease at first (when the number of neurons is less than 6) and they turned to become

increasing and unpredictable when more neurons are added. When the FCC networks consist of 5 and 6 neurons, very good approximation results are obtained.

2.5.2 *Analytical Solutions*

Based on the experiment above, one may notice that the basic reason of the over-fitting problem in neural network design can be ascribed as the mismatch between the size of training patterns and the size of networks. Normally, using improperly large size networks to train very simple patterns may result in over-fitting. From another way of speaking, in order to reduce the probability of occurrence of the over-fitting in neural network design, there are two very straightforward methods:

- Increase the size of training patterns
- Decrease the size of neural networks

For the first method, it is always good to get as many training patterns as possible; however, this strategy is only proper in practical applications when extra measurement can be performed.

For the second method, it could be notice that in order to preserve the generalization abilities of neural networks, the size of the networks should be as small as possible. From this point of view, EBP algorithm is not a good choice for design compact neural networks because of its slow convergence and poor search ability. In order to overcome the two main disadvantages of EBP algorithms, networks with much larger than optimal size are often applied for training.

Levenberg Marquardt (LM) algorithm is very efficient for neural network training and has much powerful search ability. With these properties, LM algorithm is proper to design

compact neural networks in practical applications. However, the most famous implementation of LM algorithm, Hagan and Menhaj LM algorithm [80], is only for MLP networks which perform much less efficiently than networks with connections across layers, such as BMLP networks and FCC networks.

The recently developed neuron-by-neuron (NBN) algorithm [27] solves the network limitation in Hagan and Menhaj LM algorithm, and can handle arbitrarily connected neural networks using second order update rule. Therefore, the combination of NBN algorithm and BMLP/FCC networks is recommended in literature [28] for designing compact neural networks, so as to reduce the probability of occurrence of the over-fitting problem.

2.6 Neuron-by-Neuron Algorithm

The neuron-by-neuron (NBN) algorithm [27] was proposed to solve the network architecture limitation in Hagan and Menhaj LM algorithm, so that second order algorithms can be applied to train very efficient network architectures with connections across layers [74].

The NBN algorithm adopts the index technology used in SPICE problem, and it consists of two steps, forward computation and backward computation, to gather the information required for Jacobian matrix computation in equation (2-26), using

$$j_{p,m.n} = \frac{\partial e_{p,m}}{\partial w_n} = \frac{\partial\left(d_{p,m} - o_{p,m}\right)}{\partial w_n} = -\frac{\partial o_{p,m}}{\partial w_n} = -\frac{\partial o_{p,m}}{\partial o_i}\frac{\partial o_i}{\partial net_i}\frac{\partial net_i}{\partial w_n} = -\delta_{p.m.,i} s_i y_{i,n} \qquad (2\text{-}38)$$

Where: $j_{p,m,n}$ is the element of Jacobian matrix in (2-26) related with pattern p, output m and weight n. Equation (2-38) is derived from (2-12) and (2-26), using the chain rule of differentiation. Vector $\boldsymbol{\delta}$ is defined to measure the error backpropagation process [81] and vector $\boldsymbol{y_i}$ consists of the inputs of neuron i which may be either the network inputs or the outputs of

other neurons. s_i is the slope (derivative of activation function) of the given neuron i. i is the index of neurons.

In the forward computation, neurons are organized according to the direction of signal propagation; while in backward computation, the analysis will follow the error backpropagation procedure like in first order algorithms. In order to illustrate the computation process of NBN algorithm, let us consider the network architecture with arbitrary connections as shown in Fig. 2-22.

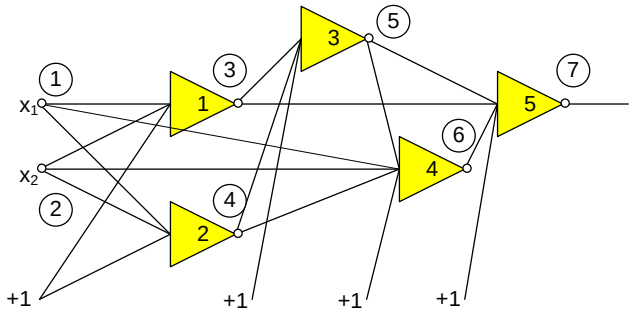


Fig. 2-22 Arbitrarily connected neural network indexed by NBN algorithm

For the network in Fig. 2-22, using the NBN algorithm, the network topology can be described as

N_1 3 1 2
N_2 4 1 2
N_3 5 3 4
N_4 6 1 2 4 5
N_5 7 3 5 6

Notice that each line represents the connections to a given neuron. The first part, from N_1 to N_5, is the neuron index. Followed, the first digit of each line is the node index of the neuron. The rest of the digits of each line represent the nodes connected to the specified neuron. With these rules, one may notice that, for each neuron, the input nodes must have smaller indices than the index of itself.

In the forward computation, the neurons connected to the network inputs are first

processed so that their outputs can be used as inputs to the subsequent neurons. The following neurons are then processed as all their input values become available. In other words, the selected computing sequence has to follow the concept of feedforward signal propagation. If a signal reaches the inputs of several neurons at the same time, then these neurons can be processed in any sequence. In the example in Fig. 2-22, there are two possible ways in which neurons can be processed in forward direction: $N_1N_2N_3N_4N_5$ or $N_2N_1N_3N_4N_5$. The two procedures have different computing processes, but lead to exactly the same results. When the forward computation is done, both of the vector $\boldsymbol{y}$ and the derivative vector $\boldsymbol{s}$ in equation (2-38) are obtained.

The sequence of the backward computation is opposite to the forward computation sequence. The process starts with the last neuron and continues toward to the inputs. In the case of the network in Fig. 2-22, there are two possible backpropagation paths: $N_5N_4N_3N_2N_1$ and $N_5N_4N_3N_1N_2$. Again, different paths will lead to the same results. In this example, let us use the $N_5N_4N_3N_2N_1$ sequence to illustrate how to calculate the vector $\boldsymbol{\delta}$ in the backward computation. Notice that, the vector $\boldsymbol{\delta}$ represents signal propagating from a network output to the inputs of all other neurons, so the size of the vector $\boldsymbol{\delta}$ is equal to the number of neurons. For the output neuron N_5, it is initialed as δ_5=1. For the neuron N_4, δ_5 is propagated by the slope of neuron N_5 and then propagated by the weight $w_{4,5}$ connected between neurons N_4 and N_5, so as to obtain $\delta_4=\delta_5 s_5 w_{4,5}$. For the neuron N_3, both of the parameters δ_4 and δ_5 will be propagated in two separated paths to the output of neuron N_3 and then summed together, as $\delta_3= \delta_4 s_4 w_{3,4}+\delta_5 s_5 w_{3,5}$. Following the same rule, it can be obtained that $\delta_2=\delta_3 s_3 w_{2,3}+\delta_4 s_4 w_{2,4}$ and $\delta_1=\delta_3 s_3 w_{1,3}+\delta_5 s_5 w_{1,5}$. After the backward computation, all the elements of vector $\boldsymbol{\delta}$ in equation (2-38) are calculated.

With the forward and backward computation, all the neuron outputs $\boldsymbol{y}$ and slope $\boldsymbol{s}$, and

vector $\boldsymbol{\delta}$ are calculated. Then using equation (2-38), all the elements of Jacobian matrix can be obtained.

In the NBN computation above, neurons are analyzed one-by-one, following the specified sequence which is decided by the network architectures. This property makes the NBN algorithm capable of handling networks consisting of arbitrarily connected neurons.

CHAPTER 3

PROBLEMS IN SECOND ORDER ALGORITHMS

The very efficient second order Levenberg Marquardt (LM) algorithm [24-25] was adopted for neural network training by Hagan and Menhaj [80], and later was implemented in MATLAB Neural Network tool box [82]. The LM algorithm uses significantly more parameters describing the error surface than just gradient elements as in the EBP algorithm. As a consequence the LM algorithm is not only fast but also it can train neural networks for which the EBP algorithm has difficulty to converge [28]. Many researchers now are using the Hagan and Menhaj LM algorithm for neural network training, but this implementation has several disadvantages:

(1) The Hagan and Menhaj LM algorithm requires the inversion of quasi Hessian matrix of size $nw \times nw$ in every iteration, where nw is the number of weights. Because of the necessity of matrix inversion in every iteration the speed advantage of LM algorithm over the EBP algorithm is less evident as the network size increases.

(2) The Hagan and Menhaj LM algorithm was developed only for multilayer perceptron (MLP) neural networks. Therefore, much more powerful and efficient networks, such as fully connected cascade (FCC) or bridged multilayer perceptron (BMLP) architectures cannot be trained.

(3) The Hagan and Menhaj LM algorithm cannot be used for the problems with many training patterns because the Jacobian matrix become prohibitively too large.

(4) The implementation of the Hagan and Menhaj LM algorithm calculated elements of

Jacobian matrix using basically the same routines as in the EBP algorithm. The different is that the error backpropagation process (for Jacobian matrix computation) must be carried on not only for every pattern but also for every output separately. So for network with multiple outputs, the backpropagation process has to be repeated for each output.

The problem (1) inherits property of the original Levenberg marquardt algorithm and it is still unsolved so that the LM algorithm can be used only for small and medium size neural networks. Considering that LM algorithm often solves problems with very efficient networks, so that the problem (1) is somehow compensated by this powerful search ability.

The problem (2) was solved by the recently developed neuron-by-neuron (NBN) algorithm, as discussed in chapter 2.6, but this algorithm requires very complex computation. The NBN algorithm also inherits the problems (3) and (4) in Hagan and Menhaj LM algorithm.

The problem (3) is called memory limitation, which makes the second order algorithms not proper for problems with large-sized patterns. This is a fatal issue for second order algorithms, since in practical problems, the size of training patterns is very large and it is encouraged to be as large as possible.

The problem (4) is also called computational redundant, which makes second order algorithms relatively complicated and inefficient for training networks with multiple outputs. Also, it is easier to handle the networks with arbitrarily connected neurons, when there is no need for backward computation process in problem (4).

In the followed two chapters, we will introduce the two methods, improved second order computation and the forward-only algorithm, as the potential solutions to memory limitation in the problem (3) and computational redundant in the problem (4), respectively.

CHAPTER 4

IMPROVED SECOND ORDER COMPUTATION

The improved second order computation presented in this chapter is aimed to optimize the neural networks learning process using Levenberg Marquardt (LM) algorithm. Quasi Hessian matrix and gradient vector are computed directly, without Jacobian matrix multiplication and storage. The memory limitation problem for LM training is solved. Considering the symmetry of quasi Hessian matrix, only elements in its upper/lower triangular array need to be calculated. Therefore, training speed is improved significantly, not only because of the smaller array stored in memory, but also the reduced operations in quasi Hessian matrix calculation. The improved memory and time efficiencies are especially true for large-sized patterns training.

In this chapter, firstly, computational fundamentals of LM algorithm are introduced to address the memory problem. Secondly, the improved computations for both quasi Hessian matrix and gradient vector are described in details. Thirdly, a simple problem is applied to illustrate the implementation of the improved computation. Finally, several experimental results are presented as the memory and training time comparison between the traditional computation and the improved computation.

4.1 Problem Description

Derived from steepest descent method and Newton algorithm, the update rule of Levenberg Marquardt algorithm is [76]

$$\Delta \boldsymbol{w} = \left(\boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{J} + \mu \boldsymbol{I} \right)^{-1} \boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{e} \tag{4-1}$$

Where: $\boldsymbol{w}$ is weight vector, $\boldsymbol{I}$ is identity matrix, μ is combination coefficient, $(P \times M) \times N$ Jacobian matrix $\boldsymbol{J}$ and $(P \times M) \times 1$ error vector $\boldsymbol{e}$ are defined as

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_N} \\ \frac{\partial e_{12}}{\partial w_1} & \frac{\partial e_{12}}{\partial w_2} & \cdots & \frac{\partial e_{12}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1M}}{\partial w_1} & \frac{\partial e_{1M}}{\partial w_2} & \cdots & \frac{\partial e_{1M}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P1}}{\partial w_1} & \frac{\partial e_{P1}}{\partial w_2} & \cdots & \frac{\partial e_{P1}}{\partial w_N} \\ \frac{\partial e_{P2}}{\partial w_1} & \frac{\partial e_{P2}}{\partial w_2} & \cdots & \frac{\partial e_{P2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{PM}}{\partial w_1} & \frac{\partial e_{PM}}{\partial w_2} & \cdots & \frac{\partial e_{PM}}{\partial w_N} \end{bmatrix} \quad \boldsymbol{e} = \begin{bmatrix} e_{11} \\ e_{12} \\ \cdots \\ e_{1M} \\ \cdots \\ e_{P1} \\ e_{P2} \\ \cdots \\ e_{PM} \end{bmatrix} \tag{4-2}$$

Where: P is the number of training patterns, M is the number of outputs and N is the number of weights. Elements in error vector $\boldsymbol{e}$ are calculated by

$$e_{pm} = d_{pm} - o_{pm} \tag{4-3}$$

Where: d_{pm} and o_{pm} are the desired output and actual output respectively, at network output m when training pattern p.

Traditionally, Jacobian matrix $\boldsymbol{J}$ is calculated and stored at first; then Jacobian matrix multiplications are performed for weight updating using (4-1). For small and median size patterns training, this method may work smoothly. However, for large-sized patterns, there is a memory limitation for Jacobian matrix $\boldsymbol{J}$ storage.

For example, the pattern recognition problem in MNIST handwritten digit database [83] consists of 60,000 training patterns, 784 inputs and 10 outputs. Using only the simplest possible neural

network with 10 neurons (one neuron per each output), the memory cost for the entire Jacobian matrix storage is nearly 35 gigabytes. This huge memory requirement cannot be satisfied by any 32-bit Windows compliers, where there is a 3 gigabytes limitation for single array storage. At this point, with traditional computation, one may conclude that Levenberg Marquardt algorithm cannot be used for problems with large number of patterns.

4.2 Improved Computation

In the following derivation, sum squared error (SSE) is used to evaluate the training process.

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M} e_{pm}^2 \tag{4-4}$$

Where: e_{pm} is the error at output m obtained by training pattern p, defined by (4-3).

The $N \times N$ Hessian matrix $\boldsymbol{H}$ is

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_N} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 E}{\partial w_N \partial w_1} & \frac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \tag{4-5}$$

Where: N is the number of weights.

Combining (4-4) and (4-5), elements of Hessian matrix $\boldsymbol{H}$ can be obtained as

$$\frac{\partial^2 E}{\partial w_i \partial w_j} = \sum_{p=1}^{P}\sum_{m=1}^{M}\left(\frac{\partial e_{pm}}{\partial w_i}\frac{\partial e_{pm}}{\partial w_j} + \frac{\partial^2 e_{pm}}{\partial w_i \partial w_j} e_{pm}\right) \tag{4-6}$$

Where: i and j are weight indexes.

For LM algorithm, equation (4-6) is approximated as [76]

$$\frac{\partial^2 E}{\partial w_i \partial w_j} \approx \sum_{p=1}^{P}\sum_{m=1}^{M}\left(\frac{\partial e_{pm}}{\partial w_i}\frac{\partial e_{pm}}{\partial w_j}\right) = q_{ij} \tag{4-7}$$

Where: q_{ij} is the element of quasi Hessian matrix in row i and column j.

Combining (4-2) and (4-7), quasi Hessian matrix $\boldsymbol{Q}$ can be calculated as an approximation of Hessian matrix

$$\boldsymbol{H} \approx \boldsymbol{Q} = \boldsymbol{J}^T \boldsymbol{J} \tag{4-8}$$

$N \times 1$ gradient vector $\boldsymbol{g}$ is

$$\boldsymbol{g} = \left[\frac{\partial E}{\partial w_1} \quad \frac{\partial E}{\partial w_2} \quad \cdots \quad \frac{\partial E}{\partial w_N}\right]^T \tag{4-9}$$

Inserting (4-4) into (4-9), elements of gradient can be calculated as

$$g_i = \frac{\partial E}{\partial w_i} = \sum_{p=1}^{P}\sum_{m=1}^{M}\left(\frac{\partial e_{pm}}{\partial w_i} e_{pm}\right) \tag{4-10}$$

From (4-2) and (4-10), the relationship between gradient vector $\boldsymbol{g}$ and Jacobian matrix $\boldsymbol{J}$ is

$$\boldsymbol{g} = \boldsymbol{J}^T \boldsymbol{e} \tag{4-11}$$

Combining (4-8), (4-11) and (4-1), the update rule of Levenberg Marquardt algorithm can be rewritten

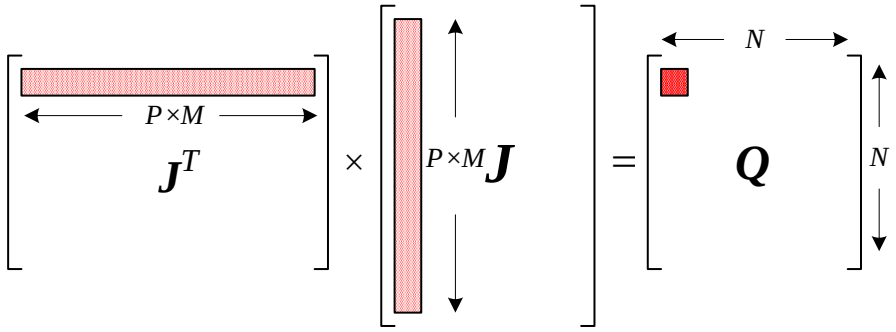
$$\Delta \boldsymbol{w} = \left(\boldsymbol{Q} + \mu \boldsymbol{I}\right)^{-1} \boldsymbol{g} \tag{4-12}$$

One may notice that the sizes of quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$ are proportional to number of weights in networks, but they are not associated with the number of training patterns and outputs.

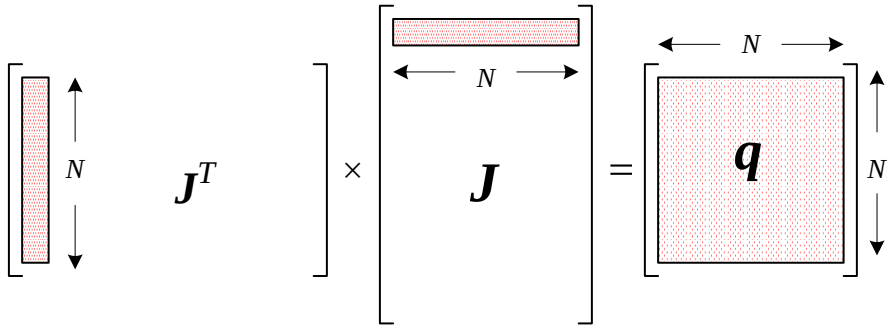
Equations (4-1) and (4-12) are producing identical results for weight updating. The major difference is that in (4-12), quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$ are calculated directly without necessity to calculate and to store Jacobian matrix $\boldsymbol{J}$ as it is done in (4-1).

4.2.1 *Review of Matrix Algebra*

There are two ways to multiply rows and columns of two matrixes. If the row of first matrix is multiplied by the column of the second matrix, then we obtain a scalar, as shown in Fig. 4-1a. When the column of the first matrix is multiplied by the row of the second matrix then the result is a partial matrix $\boldsymbol{q}$ (Fig. 4-1b) [84]. The number of scalars is $N \times N$, while number of partial matrices $\boldsymbol{q}$, which later have to be summed is $P \times M$.



(a) Row-column multiplication results in a scalar



(b) Column-row multiplication results in a partial matrix $\boldsymbol{q}$

Fig. 4-1 Two ways of multiplying matrixes

When $\boldsymbol{J}^{\mathrm{T}}$ is multiplied by $\boldsymbol{J}$ using routine shown in Fig. 4-1b, at first, partial matrices $\boldsymbol{q}$ (size: $N \times N$) need to be calculated $P \times M$ times, then all of $P \times M$ matrices $\boldsymbol{q}$ must be summed

together. The routine of Fig. 4-1b seems complicated therefore almost all matrix multiplication processes use the routine of Fig. 4-1a, where only one element of resulted matrix is calculated and stored at each time.

Even the routine of Fig. 4-1b seems to be more complicated and it is used very seldom, after detailed analysis, one may conclude that the number of numerical multiplications and additions is exactly the same as that in Fig. 4-1a, but they are performed in different order. The computation cost analysis is presented in Table 4-1.

Table 4-1 Computation cost analysis

$\boldsymbol{J}^T\boldsymbol{J}$ Computation	*Addition*	*Multiplication*
Original LM	$(P \times M) \times N \times N$	$(P \times M) \times N \times N$
Improved LM	$N \times N \times (P \times M)$	$N \times N \times (P \times M)$

In a specific case of neural network training, only one row (N elements) of Jacobian matrix $\boldsymbol{J}$ (or one column of $\boldsymbol{J}^T$) is calculated, when each pattern is applied. Therefore, if routine from Fig. 4-1b is used then the process of creation of quasi Hessian matrix can start sooner without necessity of computing and storing the entire Jacobian matrix for all patterns and all outputs.

Table 4-2 Memory cost analysis

Multiplication Methods	***Elements for storage***
Row-column (Fig. 4-1a)	$(P \times M) \times N + N \times N + N$
Column-row (Fig. 4-1b)	$N \times N + N$
Difference	$(P \times M) \times N$

P is the number of training patterns, M is the number of outputs and N is the number of weights.

The analytical results in Table 4-2 show that the column-row multiplication (Fig. 4-1b) can save a lot of memory.

4.2.2 *Improved Quasi Hessian Matrix Computation*

Let us introduce quasi Hessian sub matrix $\boldsymbol{q_{pm}}$ (size: $N \times N$)

$$\boldsymbol{q}_{pm} = \begin{bmatrix} \left(\frac{\partial e_{pm}}{\partial w_1}\right)^2 & \frac{\partial e_{pm}}{\partial w_1}\frac{\partial e_{pm}}{\partial w_2} & \cdots & \frac{\partial e_{pm}}{\partial w_1}\frac{\partial e_{pm}}{\partial w_N} \\ \frac{\partial e_{pm}}{\partial w_2}\frac{\partial e_{pm}}{\partial w_1} & \left(\frac{\partial e_{pm}}{\partial w_2}\right)^2 & \cdots & \frac{\partial e_{pm}}{\partial w_2}\frac{\partial e_{pm}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{pm}}{\partial w_N}\frac{\partial e_{pm}}{\partial w_1} & \frac{\partial e_{pm}}{\partial w_N}\frac{\partial e_{pm}}{\partial w_2} & \cdots & \left(\frac{\partial e_{pm}}{\partial w_N}\right)^2 \end{bmatrix} \tag{4-13}$$

Using (4-7) and (4-13), the $N \times N$ quasi Hessian matrix $\boldsymbol{Q}$ can be calculated as the sum of sub matrices $\boldsymbol{q_{pm}}$

$$\boldsymbol{Q} = \sum_{p=1}^{P}\sum_{m=1}^{M}\boldsymbol{q}_{pm} \tag{4-14}$$

By introducing $1 \times N$ vector $\boldsymbol{j_{pm}}$

$$\boldsymbol{j}_{pm} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} & \frac{\partial e_{pm}}{\partial w_2} & \cdots & \frac{\partial e_{pm}}{\partial w_N} \end{bmatrix} \tag{4-15}$$

sub matrices $\boldsymbol{q_{pm}}$ in (4-13) can be also written in the vector form (Fig. 4-1b)

$$\boldsymbol{q}_{pm} = \boldsymbol{j}_{pm}^{T}\boldsymbol{j}_{pm} \tag{4-16}$$

One may notice that for the computation of sub matrices $\boldsymbol{q_{pm}}$, only N elements of vector $\boldsymbol{j_{pm}}$ need to be calculated and stored. All the sub matrixes can be calculated for each pattern p and output m separately, and summed together, so as to obtain quasi Hessian matrix $\boldsymbol{Q}$.

Considering the independence among all patterns and outputs, there is no need to store all the quasi Hessian sub matrices $\boldsymbol{q_{pm}}$. Each sub matrix can be summed to a temporary matrix after its computation. Therefore, during the direct computation of quasi Hessian matrix $\boldsymbol{Q}$ using (4-14),

only memory for N elements is required, instead of that for the whole Jacobian matrix with $(P \times M) \times N$ elements (Table 4-2).

From equation (4-13), one may notice that all the sub matrixes $\boldsymbol{q}_{pm}$ are symmetrical. With this property, only upper (or lower) triangular elements of those sub matrixes need to be calculated. Therefore, during the improved quasi Hessian matrix $\boldsymbol{Q}$ computation, multiplication operations in (4-16) and sum operations in (4-14) can be both reduced by half approximately.

4.2.3 *Improved Gradient Vector Computation*

Gradient sub vector $\boldsymbol{\eta}_{pm}$ (size: $N \times 1$) is

$$\boldsymbol{\eta}_{pm} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} e_{pm} \\ \frac{\partial e_{pm}}{\partial w_2} e_{pm} \\ \cdots \\ \frac{\partial e_{pm}}{\partial w_N} e_{pm} \end{bmatrix} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} \\ \frac{\partial e_{pm}}{\partial w_2} \\ \cdots \\ \frac{\partial e_{pm}}{\partial w_N} \end{bmatrix} \times e_{pm} \tag{4-17}$$

Combining (4-10) and (4-17), gradient vector $\boldsymbol{g}$ can be calculated as the sum of gradient sub vector $\boldsymbol{\eta}_{pm}$

$$\boldsymbol{g} = \sum_{p=1}^{P} \sum_{m=1}^{M} \boldsymbol{\eta}_{pm} \tag{4-18}$$

Using the same vector $\boldsymbol{j}_{pm}$ defined in (4-15), gradient sub vector can be calculated using

$$\boldsymbol{\eta}_{pm} = \boldsymbol{j}_{pm}^{T} e_{pm} \tag{4-19}$$

Similarly, gradient sub vector $\boldsymbol{\eta}_{pm}$ can be calculated for each pattern and output separately, and summed to a temporary vector. Since the same vector $\boldsymbol{j}_{pm}$ is calculated during quasi Hessian matrix computation above, there is only an extra scalar e_{pm} need to be stored.

With the improved computation, both quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$ can be computed directly, without Jacobian matrix storage and multiplication. During the process, only a temporary vector $\boldsymbol{j_{pm}}$ with N elements needs to be stored; in other words, the memory cost for Jacobian matrix storage is reduced by $(P \times M)$ times. In the MINST problem mentioned in section 4.1, the memory cost for the storage of Jacobian elements could be reduced from more than 35 gigabytes to nearly 30.7 kilobytes.

4.2.4 *Simplified ∂epm/∂wi computation*

The key point of the improved computation above for quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$ is to calculate vector $\boldsymbol{j_{pm}}$ defined in (4-15) for each pattern and output. This vector is equivalent of one row of Jacobian matrix $\boldsymbol{J}$.

The elements of vector $\boldsymbol{j_{pm}}$ can be calculated by

$$\frac{\partial e_{\boldsymbol{pm}}}{\partial w_i} = \frac{\partial\left(o_{pm} - d_{pm}\right)}{\partial w_i} = \frac{\partial o_{pm}}{\partial net_{pn}} \frac{\partial net_{pn}}{\partial w_i} \tag{4-20}$$

Where: $\boldsymbol{d}$ is the desired output and $\boldsymbol{o}$ is the actual output; net_{pn} is the sum of weighted inputs at neuron n described as

$$net_{pn} = \sum_{i=1}^{I} x_{pi} w_i \tag{4-21}$$

Where: x_{pi} and w_i are the inputs and related weights respectively at neuron n; I is the number of inputs at neuron n.

Inserting (4-20) and (4-21) into (4-15), vector $\boldsymbol{j_{pm}}$ can be calculated by

$$\boldsymbol{j}_{pm} = \left[\frac{\partial o_{pm}}{\partial net_{p1}} \begin{bmatrix} x_{p1,1} & \cdots & x_{p1,i} & \cdots \end{bmatrix} \cdots \frac{\partial o_{pm}}{\partial net_{pn}} \begin{bmatrix} x_{pn,1} & \cdots & x_{pn,i} & \cdots \end{bmatrix} \cdots \right] \tag{4-22}$$

Where: $x_{pn,i}$ is the i-th input of neuron n, when training pattern p.

Using the neuron by neuron computation [27], elements $x_{pn,i}$ in (4-22) can be calculated in the forward computation, while $\partial o_{pm}/\partial net_{pn}$ are obtained in the backward computation. Again, since only one vector $\boldsymbol{j_{pm}}$ needs to be stored for each pattern and output in the improved computation, the memory cost for all those temporary parameters can be reduced by $(P \times M)$ times. All matrix operations are simplified to vector operations.

4.3 Implementation

In order to better illustrate the direct computation process for both quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$, let us analyze parity-2 problem as a simple example.

Parity-2 problem is also known as XOR problem. It has 4 training patterns, 2 inputs and 1 output. See Fig. 4-2.

patterns	Inputs		outputs
1	-1	-1	1
2	-1	1	-1
3	1	-1	-1
4	1	1	1

Fig. 4-2 Parity-2 problem: 4 patterns, 2 inputs and 1 output

The structure, 3 neurons in MLP topology (see Fig. 4-3), is used.

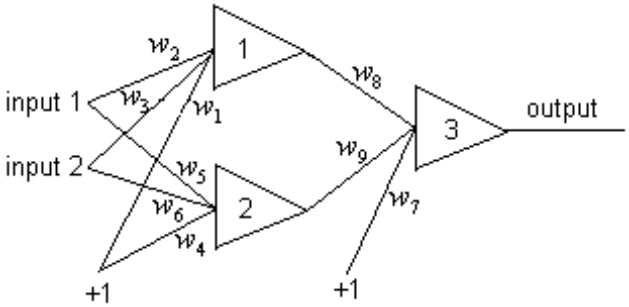


Fig. 4-3 Three neurons in MLP network used for training parity-2 problem; weight and neuron indexes are marked in the figure

As shown in Fig. 4-3 above, all weight values are initialed as the vector $\boldsymbol{w}=\{w_1,w_2,w_3,w_4,w_5,w_6,w_7,w_8,w_9\}$. All elements in both quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$ are set to "0".

For the first pattern (-1, -1), the forward computation is:

a) $net_{11}=1\times w_1+(-1)\times w_2+(-1)\times w_3$

b) $o_{11}=f(net_{11})$

c) $net_{12}=1\times w_4+(-1)\times w_5+(-1)\times w_6$

d) $o_{12}=f(net_{12})$

e) $net_{13}=1\times w_7+o_{11}\times w_8+o_{12}\times w_9$

f) $o_{13}=f(net_{13})$

g) $e_{11}=1-o_{13}$

Then the backward computation is performed to calculate $\partial e_{11}/\partial net_{11}$, $\partial e_{11}/\partial net_{12}$ and $\partial e_{11}/\partial net_{13}$ in following steps:

h) With results of steps *(f)* and *(g)*, it can be calculated

$$s_3=\frac{\partial e_{11}}{\partial net_{13}}=\frac{\partial(1-f(net_{13}))}{\partial net_{13}}=-\frac{\partial f(net_{13})}{\partial net_{13}} \tag{4-23}$$

i) With results of step *(b)* to step *(g)*, using the chain-rule in differential, one can obtain

$$s_2=\frac{\partial e_{11}}{\partial net_{12}}=-\frac{\partial f(net_{12})}{\partial net_{12}}\times w_9\times\frac{\partial f(net_{13})}{\partial net_{13}} \tag{4-24}$$

$$s_1=\frac{\partial e_{11}}{\partial net_{11}}=-\frac{\partial f(net_{11})}{\partial net_{11}}\times w_8\times\frac{\partial f(net_{13})}{\partial net_{13}} \tag{4-25}$$

In this example, using (4-22), the vector $\boldsymbol{j_{11}}$ is calculated as

$$\boldsymbol{j}_{11}=\left[\frac{\partial e_{11}}{\partial net_{11}}\times\begin{bmatrix}1 & -1 & -1\end{bmatrix}\ \ \frac{\partial e_{11}}{\partial net_{12}}\times\begin{bmatrix}1 & -1 & -1\end{bmatrix}\ \ \frac{\partial e_{11}}{\partial net_{13}}\times\begin{bmatrix}1 & o_{11} & o_{12}\end{bmatrix}\right] \tag{4-26}$$

With (4-16) and (4-19), sub matrix $\boldsymbol{q_{11}}$ and sub vector $\boldsymbol{\eta_{11}}$ can be calculated separately

$$\boldsymbol{q}_{11}=\begin{bmatrix} s_1^2 & -s_1^2 & -s_1^2 & \cdots & s_1s_3o_{11} & s_1s_3o_{12} \\ 0 & s_1^2 & s_1^2 & \cdots & -s_1s_3o_{11} & -s_1s_3o_{12} \\ 0 & 0 & s_1^2 & \cdots & -s_1s_3o_{11} & -s_1s_3o_{12} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \cdots & s_3^2o_{11}o_{12} \\ 0 & 0 & 0 & \cdots & 0 & s_3^2o_{12}^2 \end{bmatrix} \tag{4-27}$$

$$\boldsymbol{\eta}_{11}=\begin{bmatrix}s_1 & -s_1 & -s_1 & \cdots & s_3o_{11} & s_3o_{12}\end{bmatrix}\times e_{11} \tag{4-28}$$

One may notice that only upper triangular elements of sub matrix $\boldsymbol{q_{11}}$ are calculated, since all sub matrixes are symmetrical. This can save nearly half of computation.

The last step is to add sub matrix $\boldsymbol{q_{11}}$ and sub vector $\boldsymbol{\eta_{11}}$ to quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$.

The analysis above is only for training the first pattern. For other patterns, the computation process is almost the same. During the whole process, there is no Jacobian matrix computation; only the derivatives and outputs of activation functions are required to be computed. All the temporary parameters are stored in vectors which have no relationship with the number of patterns and outputs.

Generally, for the problem with *P* patterns and *M* outputs, the improved computation can be organized as the pseudo code shown in Fig. 4-4.

```
% Initialization
Q=0;
g =0
% Improved computation
for p=1:P              % Number of patterns
  % Forward computation
  ...
  for m=1:M            % Number of outputs
    % Backward computation
    ...
    calculate vector j_pm;          % Eq. (4-22)
    calculate sub matrix q_pm;      % Eq. (4-16)
    calculate sub vector η_pm;      % Eq. (4-19)
    Q=Q+q_pm;                       % Eq. (4-14)
    g=g+η_pm;                       % Eq. (4-18)
  end;
end;
```

Fig. 4-4 Pseudo code of the improved computation for quasi Hessian matrix and gradient vector

The same quasi Hessian matrices and gradient vectors are obtained in both traditional computation (equations 4-8 and 4-11) and the proposed computation (equations 4-14 and 4-18). Therefore, the proposed computation does not affect the success rate.

4.4 Experiments

Several experiments are designed to test the memory and time efficiencies of the improved computation, comparing with traditional computation. They are divided into two parts: (1) Memory comparison and (2) Time comparison.

4.4.1 *Memory Comparison*

Three problems, each of which has huge number of patterns, are selected to test the memory cost of both the traditional computation and the improved computation. LM algorithm is used for

training and the test results are shown Tables 4-3 and 4-4. In order to make more precise comparison, memory cost for program code and input files were not used in the comparison.

Table 4-3 Memory comparison for parity problems

Parity-N Problems	*N=14*	*N=16*
Patterns	16,384	65,536
Structures*	15 neurons	17 neurons
Jacobian matrix sizes	5,406,720	27,852,800
Weight vector sizes	330	425
Average iteration	99.2	166.4
Success Rate	13%	9%
Algorithms	*Actual memory cost*	
Traditional LM	79.21Mb	385.22Mb
Improved LM	3.41Mb	4.30Mb

*All neurons are in fully connected cascade networks

Table 4-4 Memory comparison for MINST problem

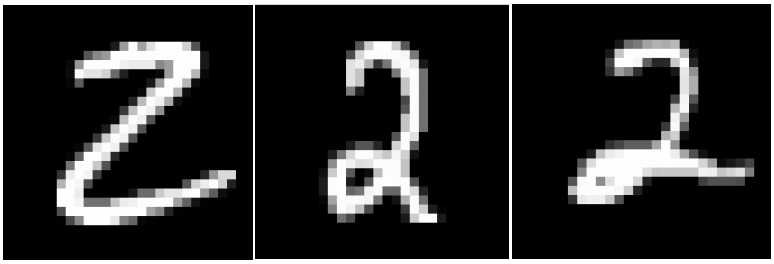
Problem	*MINST*
Patterns	60,000
Structures	784=1 single layer network*
Jacobian matrix sizes	47,100,000
Weight vector sizes	785
Algorithms	*Actual memory cost*
Traditional LM	385.68Mb
Improved LM	15.67Mb

*In order to perform efficient matrix inversion during training, only one of ten digits is classified each time.

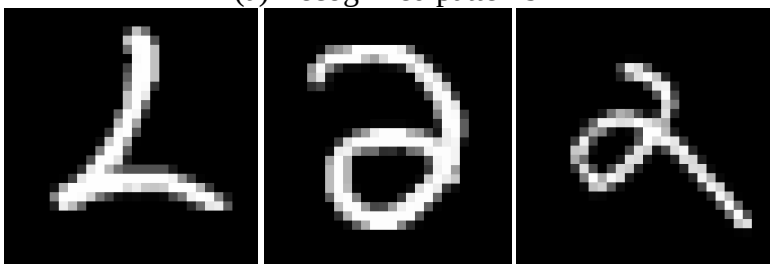
From the test results in Tables 4-3 and 4-4, it is clear that memory cost for training is significantly reduced in the improved computation.

In the MNIST problem [82], there are 60,000 training patterns, each of which is a digit (from 0 to 9) image made up of grayed 28 by 28 pixels. And also, there are another 10,000 patterns used to test the training results. With the trained network, our testing error rate for all the digits is 7.68%. In this result, for compressed, stretched and moved digits, the trained neural

network can classify them correctly (see Fig. 4-5a); for seriously rotated or distorted images, it is hard to recognize them (see Fig. 4-5b).



(a) Recognized patterns



(b) Unrecognized patterns

Fig. 4-5 Some testing results for digit "2" recognition

4.4.2 *Time Comparison*

Parity-N problems are presented to test the training time for both traditional computation and the improved computation using LM algorithm. The structures used for testing are all fully connected cascade networks. For each problem, the initial weights and training parameters are the same.

Table 4-5 Time comparison for parity problems

Parity-N Problems	*N=9*	*N=11*	*N=13*	*N=15*
Patterns	512	2,048	8,192	32,768
Neurons	10	12	14	16
Weights	145	210	287	376
Average Iterations	38.51	59.02	68.08	126.08
Success Rate	58%	37%	24%	12%
Algorithms	*Averaged training time (s)*			
Traditional LM	0.78	68.01	1508.46	43,417.06
Improved LM	0.33	22.09	173.79	2,797.93

From Table 4-5, one may notice that the improved computation can not only handle much larger problems, but also computes much faster than traditional one, especially for large-sized patterns training. The larger the pattern size is, the more time efficient the improved computation will be.

Obviously, the simplified quasi Hessian matrix computation is the one reason for the improved computing speed (nearly two times faster for small problems). Significant computation reductions obtained for larger problems are most likely due to the simpler way of addressing elements in vectors, in comparison to addressing elements in huge matrices.

With the presented experimental results, one may notice that the improved computation is much more efficient than traditional computation for training with Levenberg Marquardt algorithm, not only on memory requirements, but also training time.

4.5 Conclusion

In this chapter, the improved computation is introduced to increase the training efficiency of Levenberg Marquardt algorithm. The proposed method does not require to store and to multiply large Jacobian matrix. As a consequence, memory requirement for quasi Hessian matrix and gradient vector computation is decreased by $(P \times M)$ times, where P is the number of patterns and M is the number of outputs. Additional benefit of memory reduction is also a significant reduction in computation time. Based on the proposed computation, calculating process of quasi Hessian matrix is further simplified using its symmetrical property. Therefore, the training speed of the improved algorithm becomes much faster than traditional computation.

In the proposed computation process, quasi Hessian matrix can be calculated on fly when training patterns are applied. Moreover, the proposed method has special advantage for

applications which require dynamically changing the number of training patterns. There is no need to repeat the entire multiplication of $\boldsymbol{J}^T\boldsymbol{J}$, but only add to or subtract from quasi Hessian matrix. The quasi Hessian matrix can be modified as patterns are applied or removed.

Second order algorithms have lots of advantages, but they require at each iteration solution of large set of linear equations with number of unknowns equal to number of weights. Since in the case of first order algorithms, computing time is only proportional to the problem size, first order algorithms (in theory) could be more useful for large neural networks. However, as discussed in the previous chapters, first order algorithm (EBP algorithm) is not able to solve some problems unless excessive number of neurons is used. But with excessive number of neurons, networks lose their generalization ability and as a result, the trained networks will not respond well for new patterns, which are not used for training.

One may conclude that both first order algorithms and second order algorithms have their disadvantages and the problem of training extremely large networks with second order algorithms is still unsolved. The method presented in this chapter at least solved the problem of training neural networks using second order algorithm with basically unlimited number of training patterns.

CHAPTER 5

FORWARD-ONLY ALGORITHM

Following the neuron-by-neuron (NBN) computation procedure [27], the forward-only algorithm [78] is introduced in this chapter also allows for training arbitrarily connected neural networks; therefore, more powerful network architectures with connections across layers, such as bridged multilayer perceptron (BMLP) networks and fully connected cascade (FCC) networks, can be efficiently trained. A further advantage of the proposed forward-only algorithm is that the learning process requires only forward computation without the necessity of the backward computations. Information needed for gradient vector (for first order algorithms) and Jacobian or Hessian matrix (for second order algorithms) is obtained during forward computation. This way the forward-only method, in many cases, may also lead to the reduction of the computation time, especially for networks with multiple outputs.

In this chapter, we firstly introduce the traditional gradient vector and Jacobian matrix computation to address the computational redundancy problem for networks with multiple outputs. Then, the forward-only algorithm is proposed to solve the problem by removing backward computation process. Thirdly, both analytical and experimental comparisons are performed between the proposed forward-only algorithm and Hagan and Menhaj Levenberg Marquardt algorithm. Experimental results also show the ability of the forward-only algorithm to train networks consisting of arbitrarily connected neurons.

5.1 Computational Fundamentals

Before the derivation, let us introduce some commonly used indices in this chapter:

- *p* is the index of patterns, from 1 to *np*, where *np* is the number of patterns;
- *m* is the index of outputs, from 1 to *no*, where *no* is the number of outputs;
- *j* and *k* are the indices of neurons, from 1 to *nn*, where *nn* is the number of neurons;
- *i* is the index of neuron inputs, from 1 to *ni*, where *ni* is the number of inputs and it may vary for different neurons.

Other indices will be explained in related places.

Sum square error (SSE) *E* is defined to evaluate the training process. For all patterns and outputs, it is calculated by

$$E = \frac{1}{2}\sum_{p=1}^{np}\sum_{m=1}^{no} e_{p,m}^2 \tag{5-1}$$

Where: $e_{p,m}$ is the error at output *m* defined as

$$e_{p,m} = o_{p,m} - d_{p,m} \tag{5-2}$$

Where: $d_{p,m}$ and $o_{p,m}$ are desired output and actual output, respectively, at network output *m* for training pattern *p*.

In all training algorithms, the same computations are being repeated for one pattern at a time. Therefore, in order to simplify notations, the index *p* for patterns will be skipped in the following derivations, unless it is essential.

5.1.1 *Review of Basic Concepts in Neural Network Training*

Let us consider neuron *j* with *ni* inputs, as shown in Fig. 5-1. If neuron *j* is in the first layer, all its inputs would be connected to the inputs of the network; otherwise, its inputs can be connected to

outputs of other neurons or to networks' inputs if connections across layers are allowed.

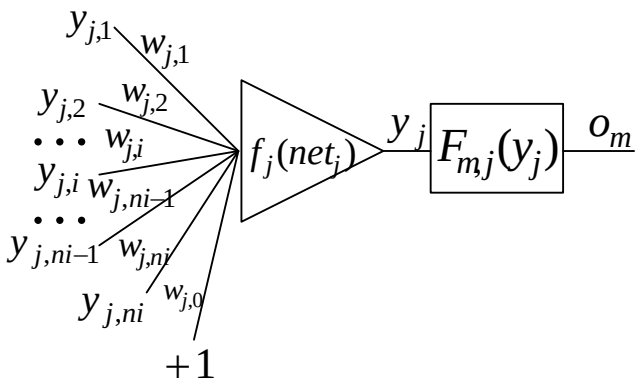


Fig. 5-1 Connection of a neuron j with the rest of the network. Nodes $y_{j,i}$ could represents network inputs or outputs of other neurons. $F_{m,j}(y_j)$ is the nonlinear relationship between the neuron output node y_j and the network output o_m

Node y is an important and flexible concept. It can be $y_{j,i}$, meaning the i-th input of neuron j. It also can be used as y_j to define the output of neuron j. In this chapter, if node y has one index then it is used as a neuron output node, but if it has two indices (neuron and input), it is a neuron input node.

Output node of neuron j is calculated using

$$y_j = f_j\left(net_j\right) \tag{5-3}$$

Where: f_j is the activation function of neuron j and net value net_j is the sum of weighted input nodes of neuron j

$$net_j = \sum_{i=1}^{ni} w_{j,i} y_{j,i} + w_{j,0} \tag{5-4}$$

Where: $y_{j,i}$ is the i-th input node of neuron j, weighted by $w_{j,i}$, and $w_{j,0}$ is the bias weight.

Using (5-4) one may notice that derivative of net_j is:

$$\frac{\partial net_j}{\partial w_{j,i}} = y_{j,i} \tag{5-5}$$

and slope s_j of activation function f_j is:

$$s_j = \frac{\partial y_j}{\partial net_j} = \frac{\partial f_j(net_j)}{\partial net_j} \tag{5-6}$$

Between the output node y_j of a hidden neuron j and network output o_m there is a complex nonlinear relationship (Fig. 5-1):

$$o_m = F_{m,j}(y_j) \tag{5-7}$$

Where: o_m is the m-th output of the network.

The complexity of this nonlinear function $F_{m,j}(y_j)$ depends on how many other neurons are between neuron j and network output m. If neuron j is at network output m, then $o_m=y_j$ and $F'_{m,j}(y_j)=1$, where $F'_{m,j}$ is the derivative of nonlinear relationship between neuron j and output m.

5.1.2 *Gradient Vector and Jacobian Matrix Computation*

For every pattern, in EBP algorithm only one backpropagation process is needed, while in second order algorithms the backpropagation process has to be repeated for every output separately in order to obtain consecutive rows of the Jacobian matrix (Fig. 5-2). Another difference in second order algorithms is that the concept of back propagating of δ parameter [81] has to be modified. In EBP algorithm, output errors are parts of δ parameter

$$\delta_j = s_j \sum_{m=1}^{no} F'_{m,j} e_m \tag{5-8}$$

In second order algorithms, the δ parameters are calculated for each neuron j and each output m separately. Also, in the backpropagation process [80] the error is replaced by a unit value

$$\delta_{m,j} = s_j F'_{m,j} \tag{5-9}$$

Knowing $\delta_{m,j}$, elements of Jacobian matrix are calculated as

$$\frac{\partial e_{p,m}}{\partial w_{j,i}} = y_{j,i}\delta_{m,j} = y_{j,i}s_j F'_{m,j} \tag{5-10}$$

In EBP algorithm, elements of gradient vector are computed as

$$g_{j,i} = \frac{\partial E}{\partial w_{j,i}} = y_{j,i}\ \delta_j \tag{5-11}$$

Where: δ_j is obtained with error back-propagation process. In second order algorithms, gradient can be obtained from partial results of Jacobian calculations

$$g_{j,i} = y_{j,i} \sum_{m=1}^{no} \delta_{m,j} e_m \tag{5-12}$$

Where: m indicates a network output and $\delta_{m,j}$ is given by (5-9).

The update rule of EBP algorithm is

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \alpha\, \boldsymbol{g}_n \tag{5-13}$$

Where: n is the index of iterations, $\boldsymbol{w}$ is weight vector, α is learning constant, $\boldsymbol{g}$ is gradient vector.

Derived from Newton algorithm and steepest descent method, the update rule of Levenberg Marquardt (LM) algorithm is [80]

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \left(\boldsymbol{J}_n^T \boldsymbol{J}_n + \mu\, \boldsymbol{I}\right)^{-1} \boldsymbol{g}_n \tag{5-14}$$

Where: μ is the combination coefficient, $\boldsymbol{I}$ is the identity matrix and $\boldsymbol{J}$ is Jacobian matrix shown in Fig. 5-2.

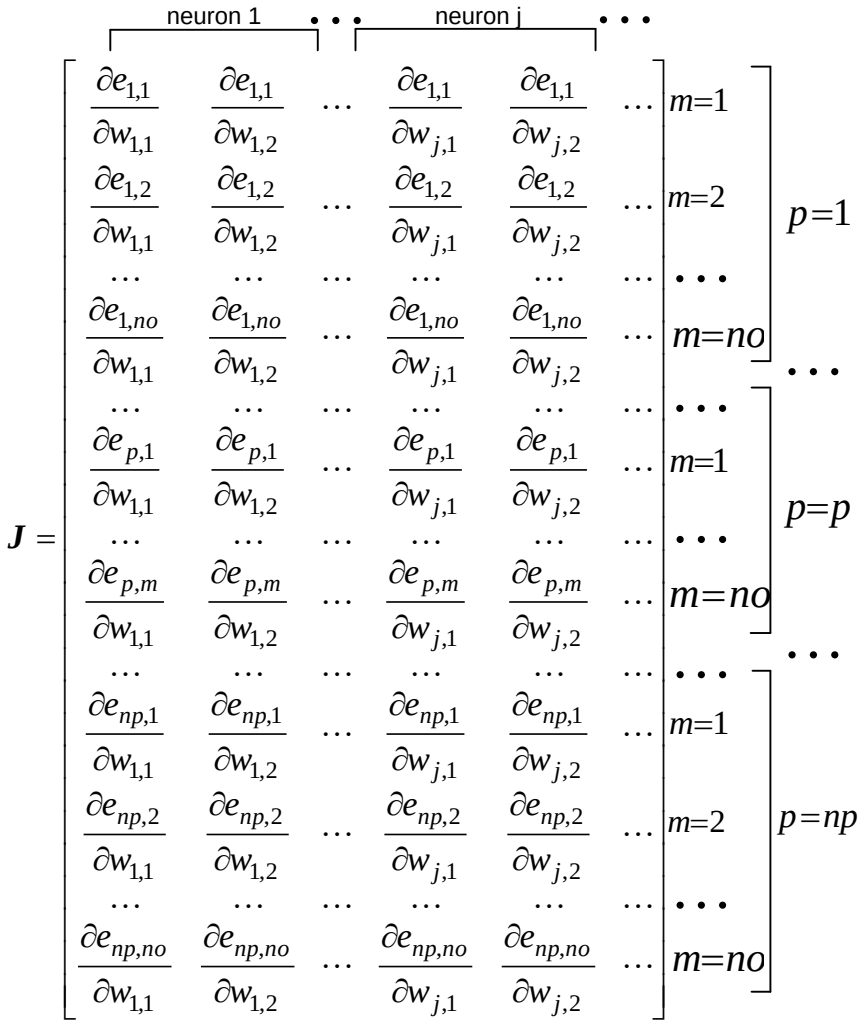


Fig. 5-2 Structure of Jacobian matrix: (1) the number of columns is equal to the number of weights; (2) each row is corresponding to a specified training pattern *p* and output *m*

From Fig. 5-2, one may notice that, for every pattern *p*, there are *no* rows of Jacobian matrix where *no* is the number of network outputs. The number of columns is equal to number of weights in the networks and the number of rows is equal to $np \times no$.

Traditional backpropagation computation, for delta matrix ($np \times no \times nn$) computation in second order algorithms, can be organized as shown in Fig. 5-3.

```
for all patterns
% Forward computation
  for all neurons (nn)
    for all weights of the neuron (nx)
      calculate net;        % Eq. (5-4)
    end;
    calculate neuron output;   % Eq. (5-7)
    calculate neuron slope;    % Eq. (5-6)
  end;
  for all outputs (no)
    calculate error;          % Eq. (5-2)
%Backward computation
    initial delta as slope;
    for all neurons starting from output neurons (nn)
      for the weights connected to other neurons (ny)
        multiply delta through weights
        sum the backpropagated delta at proper nodes
      end;
      multiply delta by slope (for hidden neurons);
    end;
  end;
end;
```

Fig. 5-3 Pseudo code using traditional backpropagation of delta in second order algorithms (code in bold will be removed in the proposed computation)

5.2 Forward-Only Computation

The proposed forward-only method is designed to improve the efficiency of Jacobian matrix computation, by removing the backpropagation process.

5.2.1 *Derivation*

The concept of $\delta_{m,j}$ was defined in equation (5-9). One may notice that $\delta_{m,j}$ can be interpreted also as a signal gain between net input of neuron j and the network output m. Let us extend this concept to gain coefficients between all neurons in the network (Fig. 5-4 and Fig. 5-6). The notation of $\delta_{k,j}$ is extension of equation (5-9) and can be interpreted as signal gain between neurons j and k and it is given by

$$\delta_{k,j} = \frac{\partial F_{k,j}(y_j)}{\partial net_j} = \frac{\partial F_{k,j}(y_j)}{\partial y_j}\frac{\partial y_j}{\partial net_j} = F'_{k,j} s_j \tag{5-15}$$

Where: k and j are indices of neurons; $F_{k,j}(y_j)$ is the nonlinear relationship between the output node of neuron k and the output node of neuron j. Naturally in feedforward networks, $k \geq j$. If $k=j$, then $\delta_{k,k}=s_k$, where s_k is the slope of activation function (5-6). Fig 5-4 illustrates this extended concept of $\delta_{k,j}$ parameter as a signal gain.

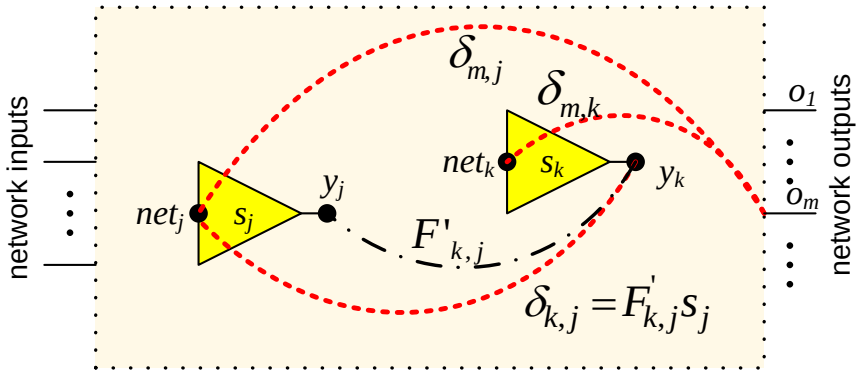


Fig. 5-4 Interpretation of $\delta_{k,j}$ as a signal gain, where in feedforward network neuron j must be located before neuron k

The matrix $\boldsymbol{\delta}$ has a triangular shape and its elements can be calculated in the forward only process. Later, elements of gradient vector and elements of Jacobian can be obtained using equations (5-10) and (5-12) respectively, where only the last rows of matrix $\boldsymbol{\delta}$ associated with network outputs are used. The key issue of the proposed algorithm is the method of calculating of $\delta_{k,j}$ parameters in the forward calculation process and it will be described in the next section.

5.2.2 *Calculation of $\boldsymbol{\delta}$ Matrix for Fully Connected Cascade Architectures*

Let us start our analysis with fully connected neural networks (Fig. 5-5). Any other architecture could be considered as a simplification of fully connected neural networks by eliminating

connections (setting weights to zero). If the feedforward principle is enforced (no feedback), fully connected neural networks must have cascade architectures.

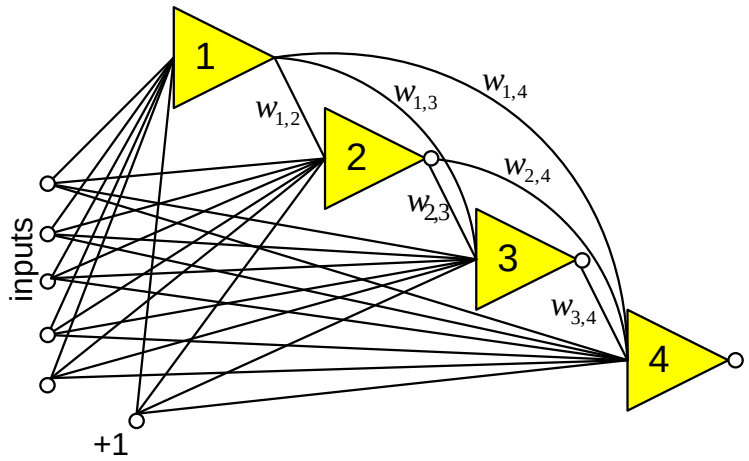


Fig. 5-5 Four neurons in fully connected neural network, with 5 inputs and 3 outputs

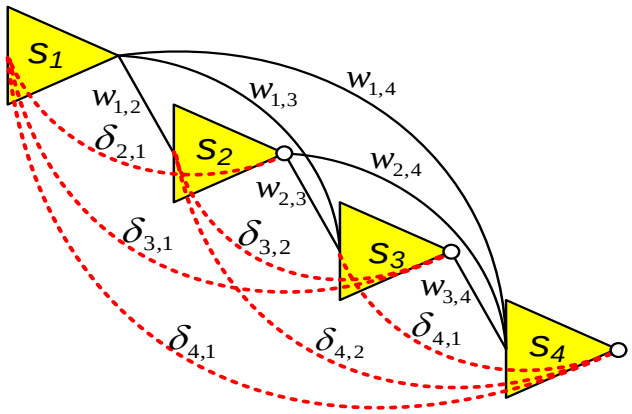


Fig. 5-6 The $\delta_{k,j}$ parameters for the neural network of Fig. 5-5. Input and bias weights are not used in the calculation of gain parameters

Slops of neuron activation functions s_j can be also written in form of δ parameter as $\delta_{j,j}=s_j$. By inspecting Fig. 5-6, $\boldsymbol{\delta}$ parameters can be written as:

For the first neuron there is only one δ parameter

$$\delta_{1,1} = s_1 \tag{5-16}$$

For the second neuron there are two δ parameters

$$\begin{aligned}\delta_{2,2} &= s_2 \\ \delta_{2,1} &= s_2 w_{1,2} s_1\end{aligned} \tag{5-17}$$

For the third neuron there are three δ parameters

$$\begin{aligned}\delta_{3,3} &= s_3 \\ \delta_{3,2} &= s_3 w_{2,3} s_2 \\ \delta_{3,1} &= s_3 w_{1,3} s_1 + s_3 w_{2,3} s_2 w_{1,2} s_1\end{aligned} \tag{5-18}$$

One may notice that all δ parameters for third neuron can be also expressed as a function of δ parameters calculated for previous neurons. Equations (5-18) can be rewritten as

$$\begin{aligned}\delta_{3,3} &= s_3 \\ \delta_{3,2} &= \delta_{3,3} w_{2,3} \delta_{2,2} \\ \delta_{3,1} &= \delta_{3,3} w_{1,3} \delta_{1,1} + \delta_{3,3} w_{2,3} \delta_{2,1}\end{aligned} \tag{5-19}$$

For the fourth neuron there are four δ parameters

$$\begin{aligned}\delta_{4,4} &= s_4 \\ \delta_{4,3} &= \delta_{4,4} w_{3,4} \delta_{3,3} \\ \delta_{4,2} &= \delta_{4,4} w_{2,4} \delta_{2,2} + \delta_{4,4} w_{3,4} \delta_{3,2} \\ \delta_{4,1} &= \delta_{4,4} w_{1,4} \delta_{1,1} + \delta_{4,4} w_{2,4} \delta_{2,1} + \delta_{4,4} w_{3,4} \delta_{3,1}\end{aligned} \tag{5-20}$$

The last parameter $\delta_{4,1}$ can be also expressed in a compacted form by summing all terms connected to other neurons (from 1 to 3)

$$\delta_{4,1} = \delta_{4,4} \sum_{i=1}^{3} w_{i,4} \delta_{i,1} \tag{5-21}$$

The universal formula to calculate $\delta_{k,j}$ parameters using already calculated data for previous neurons is

$$\delta_{k,j} = \delta_{k,k} \sum_{i=j}^{k-1} w_{i,k} \delta_{i,j} \tag{5-22}$$

Where: in feedforward network neuron j must be located before neuron k, so $k \geq j$; $\delta_{k,k} = s_k$ is the slop of activation function of neuron k; $w_{j,k}$ is weight between neuron j and neuron k; and $\delta_{k,j}$ is a signal gain through weight $w_{j,k}$ and through other part of network connected to $w_{j,k}$.

In order to organize the process, the $nn \times nn$ computation table is used for calculating signal gains between neurons, where nn is the number of neurons (Fig. 5-7). Natural indices (from 1 to nn) are given for each neuron according to the direction of signals propagation. For signal gains computation, only connections between neurons need to be concerned, while the weights connected to network inputs and biasing weights of all neurons will be used only at the end of the process. For a given pattern, a sample of the $nn \times nn$ computation table is shown in Fig. 5-7. One may notice that the indices of rows and columns are the same as the indices of neurons. In the followed derivation, let us use k and j, used as neurons indices, to specify the rows and columns in the computation table. In feed forward network, $k \geq j$ and matrix $\boldsymbol{\delta}$ has triangular shape.

Neuron Index	1	2	⋯	j	⋯	k	⋯	nn
1	$\delta_{1,1}$	$w_{1,2}$	⋯	$w_{1,j}$	⋯	$w_{1,k}$	⋯	$w_{1,nn}$
2	$\delta_{2,1}$	$\delta_{2,2}$	⋯	$w_{2,j}$	⋯	$w_{2,k}$	⋯	$w_{2,nn}$
⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯
j	$\delta_{j,1}$	$\delta_{j,2}$	⋯	$\delta_{j,j}$	⋯	$w_{j,k}$	⋯	$w_{j,nn}$
⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯
k	$\delta_{k,1}$	$\delta_{k,2}$	⋯	$\delta_{k,j}$	⋯	$\delta_{k,k}$	⋯	$w_{k,nn}$
⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯
nn	$\delta_{nn,1}$	$\delta_{nn,2}$	⋯	$\delta_{nn,j}$	⋯	$\delta_{nn,k}$	⋯	$\delta_{nn,nn}$

Fig. 5-7 The $nn \times nn$ computation table; gain matrix $\boldsymbol{\delta}$ contains all the signal gains between neurons; weight array $\boldsymbol{w}$ presents only the connections between neurons, while network input weights and biasing weights are not included

The computation table consists of three parts: weights between neurons in upper triangle, vector of slopes of activation functions in main diagonal and signal gain matrix $\boldsymbol{\delta}$ in lower

triangle. Only main diagonal and lower triangular elements are computed for each pattern. Initially, elements on main diagonal $\delta_{k,k}=s_k$ are known as slopes of activation functions and values of signal gains $\delta_{k,j}$ are being computed subsequently using equation (5-22).

The computation is being processed neuron by neuron starting with the neuron closest to network inputs. At first the row number one is calculated and then elements of subsequent rows. Calculation on row below is done using elements from above rows using (5-22). After completion of forward computation process, all elements of $\boldsymbol{\delta}$ matrix in the form of the lower triangle are obtained.

In the next step elements of gradient vector and Jacobian matrix are calculated using equation (5-10) and (5-12). In the case of neural networks with one output only the last row of $\boldsymbol{\delta}$ matrix is needed for gradient vector and Jacobian matrix computation. If networks have more outputs *no* then last *no* rows of $\boldsymbol{\delta}$ matrix are used. For example, if the network shown in Fig. 5-5 has 3 outputs the following elements of $\boldsymbol{\delta}$ matrix are used

$$\begin{bmatrix} \delta_{2,1} & \delta_{2,2}=s_2 & \delta_{2,3}=0 & \delta_{2,4}=0 \\ \delta_{3,1} & \delta_{3,2} & \delta_{3,3}=s_3 & \delta_{3,4}=0 \\ \delta_{4,1} & \delta_{4,2} & \delta_{4,3} & \delta_{4,4}=s_4 \end{bmatrix} \tag{5-23}$$

and then for each pattern, the three rows of Jacobian matrix, corresponding to three outputs, are calculated in one step using (5-10) without additional propagation of

$$\begin{bmatrix} \delta_{2,1}\times\{\boldsymbol{y}_1\} & s_2\times\{\boldsymbol{y}_2\} & 0\times\{\boldsymbol{y}_3\} & 0\times\{\boldsymbol{y}_4\} \\ \delta_{3,1}\times\{\boldsymbol{y}_1\} & \delta_{3,2}\times\{\boldsymbol{y}_2\} & s_3\times\{\boldsymbol{y}_3\} & 0\times\{\boldsymbol{y}_4\} \\ \underbrace{\delta_{4,1}\times\{\boldsymbol{y}_1\}}_{neuron\ 1} & \underbrace{\delta_{4,2}\times\{\boldsymbol{y}_2\}}_{neuron\ 2} & \underbrace{\delta_{4,3}\times\{\boldsymbol{y}_3\}}_{neuron\ 3} & \underbrace{s_4\times\{\boldsymbol{y}_4\}}_{neuron\ 4} \end{bmatrix} \tag{5-24}$$

Where: neurons' input vectors $\boldsymbol{y_1}$ through $\boldsymbol{y_4}$ have 6, 7, 8 and 9 elements respectively (Fig. 5-5), corresponding to number of weights connected. Therefore, each row of Jacobian matrix has

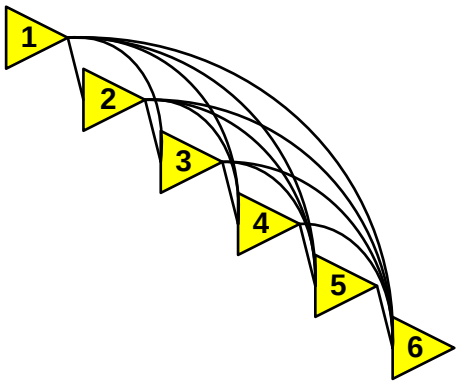
6+7+8+9=30 elements. If the network has 3 outputs, then from 6 elements of $\boldsymbol{\delta}$ matrix and 3 slops, 90 elements of Jacobian matrix are calculated. One may notice that the size of newly introduced $\boldsymbol{\delta}$ matrix is relatively small and it is negligible in comparison to other matrixes used in calculation.

The proposed method gives all the information needed to calculate both gradient vector (5-12) and Jacobian matrix (5-10), without backpropagation process; instead, δ parameters are obtained in relatively simple forward computation (see equation (5-22)).

5.2.3 *Training Arbitrarily Connected Neural Networks*

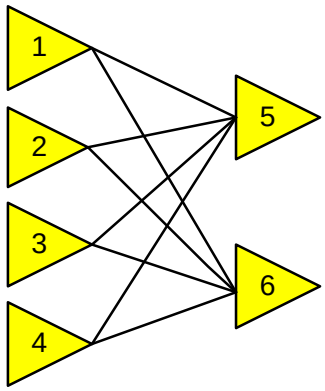
The forward-only computation above was derived for fully connected neural networks. If network is not fully connected, then some elements of the computation table are zero. Fig. 5-8 shows computation tables for different neural network topologies with 6 neurons each. Please notice zero elements are for not connected neurons (in the same layers). This can further simplify the computation process for popular MLP topologies (Fig. 5-8b).

Index	1	2	3	4	5	6
1	s_1	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	$w_{1,6}$
2	$\delta_{2,1}$	s_2	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	$w_{2,6}$
3	$\delta_{3,1}$	$\delta_{3,2}$	s_3	$w_{3,4}$	$w_{3,5}$	$w_{3,6}$
4	$\delta_{4,1}$	$\delta_{4,2}$	$\delta_{4,3}$	s_4	$w_{4,5}$	$w_{4,6}$
5	$\delta_{5,1}$	$\delta_{5,2}$	$\delta_{5,3}$	$\delta_{5,4}$	s_5	$w_{5,6}$
6	$\delta_{6,1}$	$\delta_{6,2}$	$\delta_{6,3}$	$\delta_{6,4}$	$\delta_{6,5}$	s_6



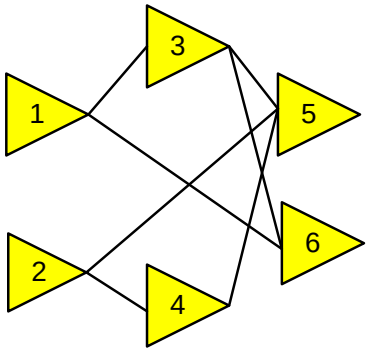
(a) Fully connected cascade network

Index	1	2	3	4	5	6
1	s_1	0	0	0	$w_{1,5}$	$w_{1,6}$
2	0	s_2	0	0	$w_{2,5}$	$w_{2,6}$
3	0	0	s_3	0	$w_{3,5}$	$w_{3,6}$
4	0	0	0	s_4	$w_{4,5}$	$w_{4,6}$
5	$\delta_{5,1}$	$\delta_{5,2}$	$\delta_{5,3}$	$\delta_{5,4}$	s_5	0
6	$\delta_{6,1}$	$\delta_{6,2}$	$\delta_{6,3}$	$\delta_{6,4}$	0	s_6



(b) Multilayer perceptron network

Index	1	2	3	4	5	6
1	s_1	0	$w_{1,3}$	0	0	$w_{1,6}$
2	0	s_2	0	$w_{2,4}$	$w_{2,5}$	0
3	$\delta_{3,1}$	0	s_3	0	$w_{3,5}$	$w_{3,6}$
4	0	$\delta_{4,2}$	0	s_4	$w_{4,5}$	0
5	$\delta_{5,1}$	$\delta_{5,2}$	$\delta_{5,3}$	$\delta_{5,4}$	s_5	0
6	$\delta_{6,1}$	0	$\delta_{6,3}$	0	0	s_6



(c) Arbitrarily connected neural network

Fig. 5-8 Three different architectures with 6 neurons

Most of used neural networks have many zero elements in the computation table (Fig. 5-8). In order to reduce the storage requirements (do not store weights with zero values) and to reduce computation process (do not perform operations on zero elements), a part of the NBN algorithm [27] in chapter 5 was adopted for forward computation.

In order to further simplify the computation process, the equation (5-22) is completed in two steps

$$x_{k,j} = \sum_{i=j}^{k-1} w_{i,k}\delta_{i,j} \tag{5-25}$$

and

$$\delta_{k,j} = \delta_{k,k} x_{k,j} = s_k x_{k,j} \tag{5-26}$$

The complete algorithm with forward-only computation is shown in Fig. 5-9. By adding two additional steps using equations (5-25) and (5-26) (highlighted in bold in Fig. 5-9), all computation can be completed in the forward only computing process.

```
for all patterns (np)
% Forward computation
  for all neurons (nn)
    for all weights of the neuron (nx)
      calculate net;                          % Eq. (5-4)
    end;
    calculate neuron output;                  % Eq. (5-3)
    calculate neuron slope;                   % Eq. (5-6)
    set current slope as delta;
    for weights connected to previous neurons (ny)
      for previous neurons (nz)
          multiply delta through weights then sum;  % Eq. (5-24)
        end;
        multiply the sum by the slope;                % Eq. (5-25)
    end;
    related Jacobian elements computation;    % Eq. (5-12)
  end;
  for all outputs (no)
    calculate error;        % Eq. (5-2)
  end;
end;
```

Fig. 5-9 Pseudo code of the forward-only computation, in second order algorithms

5.3 Computation Comparison

The proposed forward-only computation removes the backpropagation part, but it includes an additional calculation in the forward computation (the bold part in Fig. 5-9). Let us compare the computation cost of forward part and backward part for each method, in LM algorithm. Naturally such comparison can be done only for traditional MLP architectures, which can be handled by both algorithms.

As is shown in Fig. 5-3 and Fig. 5-9, computation cost of traditional computation and the forward-only computation depends on the neural network topology. In order to do the analytical comparison, for each neuron, let us consider:

- *nx* as the average number of weights

$$nx = \frac{nw}{nn} \tag{5-27}$$

- *ny* as the average number of weights between neurons

$$ny = \frac{nh \times no}{nn} \tag{5-28}$$

- *nz* as the average number of previous neurons

$$nz = \frac{nh}{nn} \tag{5-29}$$

Where: *nw* is the number of weights; *nn* is the number of neurons; *no* is the number of outputs; *nh* is the number of hidden neurons. The estimation of *ny* depends on network structures. Equation (5-28) gives the *ny* value for MLP networks with one hidden layer. The comparison below is for training one pattern.

From the analytical results in Table 5-1, one may notice that, for the backward part, time cost in backpropagation computation is tightly associated with the number of outputs; while in the forward-only computation, the number of outputs is almost irrelevant.

Table 5-1 Analysis of computation cost in Hagan and Menhaj LM algorithm and forward-only computation

	Hagan and Menhaj Computation	
	Forward Part	Backward Part
+/-	$nn \times nx + 3nn + no$	$no \times nn \times ny$
×/÷	$nn \times nx + 4nn$	$no \times nn \times ny + no \times (nn - no)$
exp*	nn	0
	Forward-only computation	
	Forward	Backward
+/-	$nn \times nx + 3nn + no + nn \times ny \times nz$	0
×/÷	$nn \times nx + 4nn + nn \times ny + nn \times ny \times nz$	0
exp	nn	0
	Subtraction forward-only from traditional	
+/-	$nn \times ny \times (no - 1)$	
×/÷	$nn \times ny \times (no - 1) + no \times (nn - no) - nn \times ny \times nz$	
exp	0	

*Exponential operation.

Table 5-2 shows the computation cost for the neural network which will be used for the ASCII problem in section 5.4, using the equations of Table 5-1.

In typical PC computer with arithmetic coprocessor, based on the experimental results, if the time cost for "+/-" operation is set as unit "1", then "×/÷" and "exp" operations will cost nearly 2 and 65 respectively.

Table 5-2 Comparison for ASCII problem

	Hagan and Menhaj computation		Forward-only computation	
	Forward	Backward	Forward	Backward
+/-	4,088	175,616	7,224	0
×/÷	4,144	178,752	8,848	0
exp	7,280	0	7,280	0
Total	552,776		32,200	
Relative time	100%		5.83%	

*Network structure: 112 neurons in 8-56-56 MLP network

For the computation speed testing in the next section, the analytical relative times are presented in Table 5-3.

Table 5-3 Analytical relative time of the forward-only computation of problems

Problems	nn	no	nx	ny	nz	Relative time
ASCII conversion	112	56	33	28	0.50	5.83%
Error correction	42	12	18.1	8.57	2.28	36.96%
Forward kinematics	10	3	5.9	2.10	0.70	88.16%

For MLP network with one hidden layer topologies, using the estimation rules in Table 5-1, computation cost of both the forward-only method and traditional forward-backward method is compared in Fig. 5-10. All networks have 20 inputs.

Based on the analytical results, it could be seen that, in LM algorithm, for single output networks, the forward-only computation is similar with the traditional computation; while for networks with multiple outputs, the forward-only computation is supposed to be more efficient.

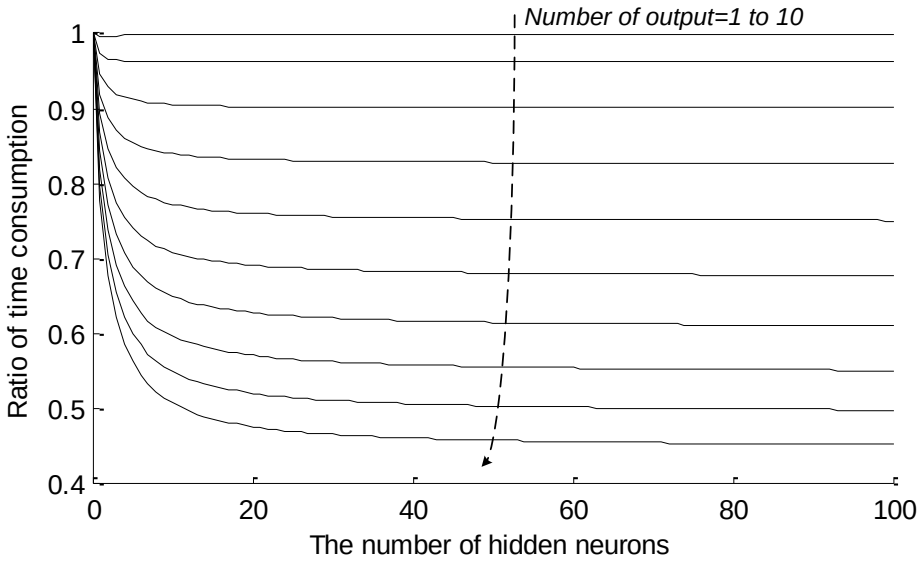


Fig. 5-10 Comparison of computation cost for MLP networks with one hidden layer; *x*-axis is the number of neurons in hidden layer; *y*-axis is the time consumption radio between the forward-only computation and the forward-backward computation

5.4 Experiments

The experiments were organized in three parts: (1) ability of handling various network topologies; (2) training neural networks with generalization abilities; (3) computational efficiency.

5.4.1 *Ability of Handling Various Network Topologies*

The ability of training arbitrarily connected networks of the proposed forward-only computation is illustrated by the two-spiral problem.

The two-spiral problem is considered as a good evaluation of training algorithms [77]. Depending on neural network architecture, different numbers of neurons are required for successful training. For example, using standard MLP networks with one hidden layer, 34 neurons are required for the two-spiral problem [85]. Using the proposed computation in LM algorithm, two types of topologies, MLP networks with two hidden layers and fully connected

cascade (FCC) networks, are tested for training the two-spiral patterns, and the results are presented in the Tables below. In MLP networks with two hidden layers, the number of neurons is assumed to be equal in both hidden layers.

Results for MLP architectures shown in the Table 5-4 are identical no matter if the Hagan and Menhaj LM algorithm or the proposed LM algorithm is used (assuming the same initial weights). In other words, the proposed algorithm has the same success rate and the same number of iterations as those obtained by Hagan and Menhaj LM algorithm. The difference is that the proposed algorithm can handle also other than MLP architectures and in many cases (especially with multiple outputs) computation time is shorter.

Table 5-4 Training results of the two-spiral problem with the proposed forward-only implementation of LM algorithm, using MLP networks with two hidden layers; maximum iteration is 1,000; desired error=0.01; there are 100 trials for each case

Hidden neurons	Success rate	Average number of iterations	Average time (s)
12	Failing	/	/
14	13%	474.7	5.17
16	33%	530.6	8.05
18	50%	531.0	12.19
20	63%	567.9	19.14
22	65%	549.1	26.09
24	71%	514.4	34.85
26	81%	544.3	52.74

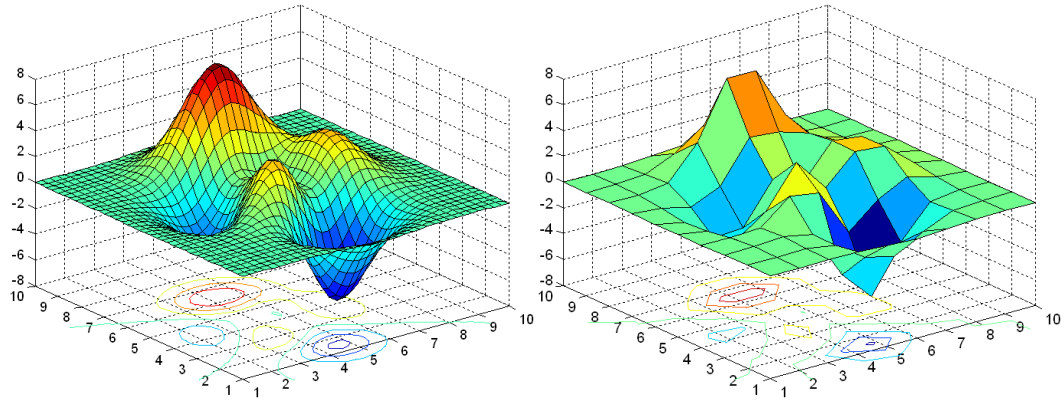
Table 5-5 Training results of the two-spiral problem with the proposed forward-only implementation of LM algorithm, using FCC networks; maximum iteration is 1,000; desired error=0.01; there are 100 trials for each case

Hidden Neurons	Success Rate	Average Number of Iterations	Average Time (s)
7	13%	287.7	0.88
8	24%	261.4	0.98
9	40%	243.9	1.57
10	69%	231.8	1.62
11	80%	175.1	1.70
12	89%	159.7	2.09
13	92%	137.3	2.40
14	96%	127.7	2.89
15	99%	112.0	3.82

From the testing results presented in Table 5-5, one may notice that the fully connected cascade (FCC) networks are much more efficient than other networks to solve the two-spiral problem, with as little number of neurons as 8. The proposed LM algorithm is also more efficient than the well-known cascade correlation algorithm, which requires 12-19 hidden neurons in FCC architectures to converge [86].

5.4.2 *Train Neural Networks with Generalization Abilities*

To compare generalization abilities, FCC networks, being proved to be the most efficient in section 2.3, are applied for training. These architectures can be trained by both EBP algorithm and the forward-only implementation of LM algorithm. The slow convergence of EBP algorithm is not the issue in this experiment. Generalization abilities of networks trained with both algorithms are compared. The Hagan and Menhaj LM algorithm was not used for comparison here because it cannot handle FCC networks.



(a) Testing surface with 37×37=1,369 points (b) Training surface with 10×10=100 points

Fig. 5-11 Peak surface approximation problem

Let us consider the peak surface [85] as the required surface (Fig. 5-11a) and let us use equally spaced 10×10=100 patterns (Fig. 5-11b) to train neural networks. The quality of trained networks is evaluated using errors computed for equally spaced 37×37=1,369 patterns. In order

to make a valid comparison between training and verification errors, the sum squared error (SSE), as defined in (5-1), is divided by 100 and 1,369 respectively.

Table 5-6 Training Results of peak surface problem using FCC architectures

Neurons	Success Rate		Average Iteration		Average Time (s)	
	EBP	LM	EBP	LM	EBP	LM
8	0%	5%	Failing	222.5	Failing	0.33
9	0%	25%	Failing	214.6	Failing	0.58
10	0%	61%	Failing	183.5	Failing	0.70
11	0%	76%	Failing	177.2	Failing	0.93
12	0%	90%	Failing	149.5	Failing	1.08
13	35%	96%	573,226	142.5	624.88	1.35
14	42%	99%	544,734	134.5	651.66	1.76
15	56%	100%	627,224	119.3	891.90	1.85

For EBP algorithm, learning constant is 0.0005 and momentum is 0.5; maximum iteration is 1,000,000 for EBP algorithm and 1,000 for LM algorithm; desired error=0.5; there are 100 trials for each case. The proposed version of LM algorithm is used in this experiment

The training results are shown in Table 5-6. One may notice that it was possible to find the acceptable solution (Fig. 5-12) with 8 neurons (52 weights). Unfortunately, with EBP algorithm, it was not possible to find acceptable solutions in 100 trials within 1,000,000 iterations each. Fig. 5-13 shows the best result out of the 100 trials with EBP algorithm. When the network size was significantly increased from 8 to 13 neurons (117 weights), EBP algorithm was able to reach the similar training error as with LM algorithm, but the network lost its generalization ability to respond correctly for new patterns (between training points). Please notice that with enlarged number of neurons (13 neurons), EBP algorithm was able to train network to small error SSE_{Train}=0.0018, but as one can see from Fig. 5-14, the result is unacceptable with verification error SSE_{Verify}=0.4909.

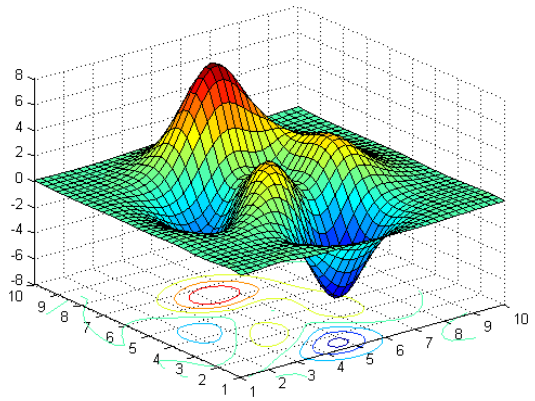


Fig. 5-12 The best training result in 100 trials, using LM algorithm, 8 neurons in FCC network (52 weights); maximum training iteration is 1,000; SSE_{Train}=0.0044, SSE_{Verify}=0.0080 and training time=0.37 s

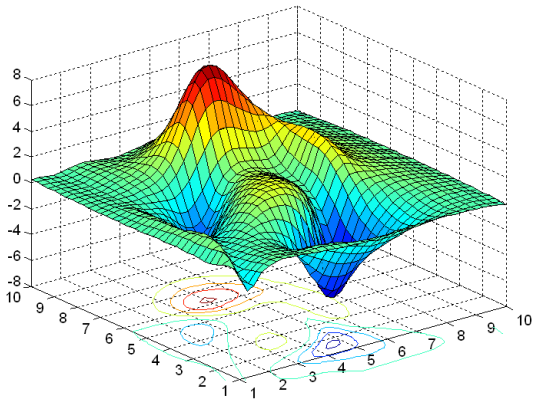


Fig. 5-13 The best training result in 100 trials, using EBP algorithm, 8 neurons in FCC network (52 weights); maximum training iteration is 1,000,000; SSE_{Train}=0.0764, SSE_{Verify}=0.1271 and training time=579.98 s

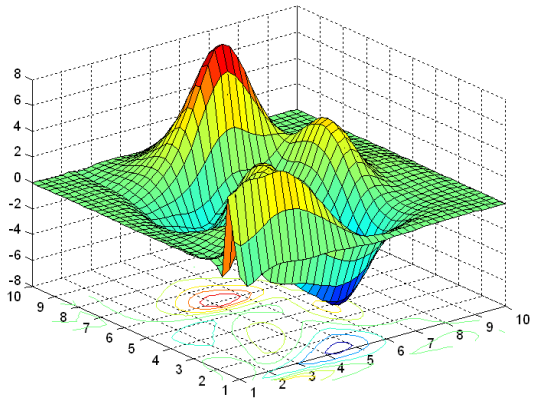


Fig. 5-14 The best training result in 100 trials, using EBP algorithm, 13 neurons in FCC network (117 weights); maximum training iteration is 1,000,000; SSE_{Train}=0.0018, SSE_{Verify}=0.4909 and training time=635.72 s

From the presented examples, one may notice that often in simple (close to optimal) networks, EBP algorithm can't converge to required training error (Fig. 5-13). When the size of networks increases, EBP algorithm can reach the required training error, but trained networks lose their generalization ability and can't process new patterns well (Fig. 5-14). On the other hand, the proposed implementation of LM algorithm in this chapter, works not only significantly faster but it can find good solutions with close to optimal networks (Fig. 5-12).

5.4.3 *Computational Speed*

Several problems are presented to test the computation speed of both the Hagan and Menhaj LM algorithm, and the proposed LM algorithm. The testing of time costs is divided into forward part and backward part separately. In order to compare with the analytical results in section 5.3, the MLP networks with one hidden layer are used for training.

5.4.3.1 ASCII Codes to Images Conversion

This problem is to associate 256 ASCII codes with 256 character images, each of which is made up of 7×8 pixels (Fig. 5-15). So there are 8-bit inputs (inputs of parity-8 problem), 256 patterns and 56 outputs. In order to solve the problem, the structure, 112 neurons in 8-56-56 MLP network, is used to train those patterns using LM algorithm. The computation time is presented in Table 5-7. The analytical result is 5.83% as shown in Table 5-3.

Table 5-7 Comparison for ASCII characters recognition problem

Computation methods	Time cost (ms/iteration)		Relative time
	Forward	Backward	
Traditional	8.24	1,028.74	100.0%
Forward-only	61.13	0.00	5.9%

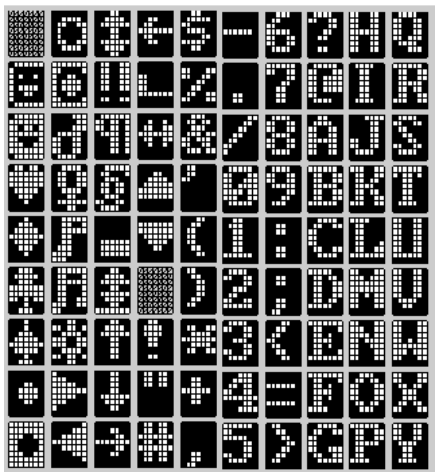


Fig. 5-15 The first 90 images of ASCII characters

Testing results in Table 5-7 show that, for this multiple outputs problem, the forward-only computation is much more efficient than traditional computation, in LM training.

5.4.3.2 Error Correction

Error correction is an extension of parity-N problems [74] for multiple parity bits. In Fig. 5-16, the left side is the input data, made up of signal bits and their parity bits, while the right side is the related corrected signal bits and parity bits as outputs. The number of inputs is equal to the number of outputs.

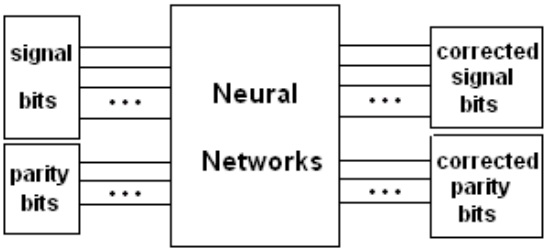


Fig. 5-16 Using neural networks to solve an error correction problem; errors in input data can be corrected by well trained neural networks

The error correction problem in the experiment has 8-bit signal and 4-bit parity bits as inputs, 12 outputs and 3,328 patterns (256 correct patterns and 3,072 patterns with errors), using

42 neurons in 12-30-12 MLP network (762 weights). Error patterns with one incorrect bit must be corrected. Both traditional computation and the forward-only computation were performed with the LM algorithm. The testing results are presented in Table 5-8. The analytical result is 36.96% as shown in Table 5-3.

Table 5-8 Comparison for error correction problem

Problems	Computation Methods	Time Cost (ms/iteration)		Relative Time
		Forward	Backward	
8-bit signal	Traditional	40.59	468.14	100.0%
	Forward-only	175.72	0.00	34.5%

Compared with the traditional forward-backward computation in LM algorithm, again, the forward-only computation has a considerably improved efficiency. With the trained neural network, all the patterns with one bit error are corrected successfully.

5.4.3.3 Forward Kinematics

Neural networks are successfully used to solve many practical problems in the industry, such as control problems, compensation nonlinearities in objects and sensors, issues of identification of parameters which cannot be directly measured, and sensorless control [87-89].

Forward kinematics is an example of these types of practical applications [43][90-92]. The purpose is to calculate the position and orientation of robot's end effector as a function of its joint angles. Fig. 5-17 shows the two-link planar manipulator.

As shown in Fig. 5-17, the end effector coordinates of the manipulator is calculated by:

$$x = L_1 \cos\alpha + L_2 \cos(\alpha + \beta) \tag{5-30}$$

$$y = L_1 \sin\alpha + L_2 \sin(\alpha + \beta) \tag{5-31}$$

Where: *(x, y)* is the coordinate of the end effector which is determined by angles α and β; L_1 and L_2 are the arm lengths. In order to avoid scanning "blind area", let us assume $L_1=L_2=1$.

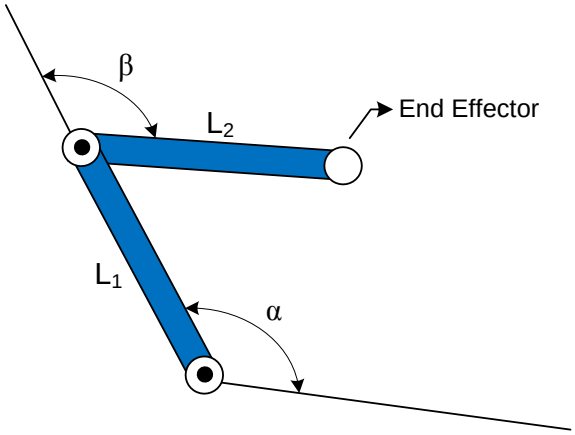


Fig. 5-17 Tow-link planar manipulator

In this experiment, 224 patterns are applied for training the MLP network 3-7-3 (59 weights), using LM algorithm. The comparison of computation cost between the forward-only computation and traditional computation is shown in Table 5-9. In 100 trials with different starting points, the experiment got 22.2% success rate and the average iteration cost for converge was 123.4. The analytical result is 88.16% as shown in Table 5-3.

Table 5-9 Comparison for forward kinematics problem

Computation methods	Time cost (ms/iteration)		Relative time
	Forward	Backward	
Traditional	0.307	0.771	100.0%
Forward-only	0.727	0.00	67.4%

The presented experimental results match the analysis in section 5.3 well: for networks with multiple outputs, the forward-only computation is more efficient than the traditional backpropagation computation.

5.5 Conclusion

One of the major features of the proposed forward-only algorithm is that it can be easily adapted to train arbitrarily connected neural networks and not just MLP topologies. This is very important because neural networks with connections across layers are much more powerful than commonly used MLP architectures. For example, if the number of neurons in the network is limited to 8 then popular MLP topology with one hidden layer is capable to solve only parity-7 problem. If the same 8 neurons are connected in fully connected cascade then with this network parity-255 problem can be solved [93].

It was shown (Figs. 5-13 and 5-14) that in order to secure training convergence with first order algorithms the excessive number of neurons much be used, and this results with a failure of neural network generalization abilities. This was the major reason for frustration in industrial practice when neural networks were trained to small errors but they would respond very poorly for patterns not used for training. The presented forward-only computation for second order algorithms can be applied to train arbitrarily connected neural networks, so it is capable to train neural networks with reduced number of neurons and as consequence a good generalization abilities were secured (Fig. 5-12).

The proposed forward-only computation gives identical number of training iterations and success rates, as the Hagan and Menhaj implementation of the LM algorithm does, since the same Jacobian matrix are obtained from both methods. By removing backpropagation process, the proposed method is much simpler than traditional forward and backward procedure to calculate elements of Jacobian matrix. The whole computation can be described by a regular table (Fig. 5-7) and a general formula (equation 5-22). Additionally, for networks with multiple outputs, the proposed method is less computationally intensive and faster than traditional forward and backward computations [27][80].

CHAPTER 6

C++ IMPLEMENTATION OF NEURAL NETWORK TRAINER

Currently, there are some excellent tools for neural networks training, such as the MATLAB Neural Network Toolbox (MNNT) and Stuttgart Neural Network Simulator (SNNS). The MNNT can do both EBP and LM training, but only for standard MLP networks which are not as efficient as other networks with connections across layers. Furthermore, it's also well-known that MATLAB is very inefficient in executing "for" loop, which may slow down the training process. SNNS can handle FCN networks well, but the training methods it contains are all developed based on EBP algorithm, such as QuickPROP algorithm [94] and Resilient EBP [68], which makes the training still somewhat slow.

In this chapter, the neural network trainer, named NBN 2.0 [44-46], is introduced as a powerful training tool. It contains EBP algorithm with momentum [93], LM algorithm [80], NBN algorithm [26] and a newly developed second order algorithm. Based on neuron-by-neuron computation [27] and forward-only computation [78-79], all those algorithms can handle arbitrarily connected neuron (ACN) networks. Comparing with the former MATLAB version [96], the revised one is supposed to perform more efficient and stable training.

The NBN 2.0 is developed based on Visual Studio 6.0 using C++ language. Its main graphic user interface (GUI) is shown in Fig. 8-6. In the following part of this chapter, detailed instructions about the software are presented. Then several examples are applied to illustrate the functionalities of NBN 2.0.

6.1 File Instruction

The software is made up of 6 types of files, including executing files, parameter file (unique), topology files, training pattern files, training result files and training verification files.

6.1.1 *Executing Files*

Executing files contain three files: files "FauxS-TOON.ssk" and "skinppwtl.dll" for the GUI design; file "NBN 2.0 exe" is used for running the software. Also, other files, such as user instruction, correction log and accessory tools (Matlab code "PlotFor2D.m" for 2-D plotting), are included.

6.1.2 *Parameter File*

This file is named "Parameters.dat" and it is necessary for running the software. It contains initial data of important parameters shown in Table 6-1.

Table 6-1 Parameters for training

Parameters	***Descriptions***
algorithm	Index of algorithms in the combo box
alpha	Learning constant for EBP
scale	Parameter for LM/NBN
mu	Parameter for LM/NBN
max mu	Parameter for LM/NBN (fixed)
min mu	Parameter for LM/NBN (fixed)
max error	Maximum error
ITE_FOR_EBP	Maximum iteration for EBP
ITE_FOR_LM	Maximum iteration for LM/NBN
ITE_FOR_PO	Maximum iteration for the new Alg.
momentum	Momentum for EBP
po alpha	Parameter for the improved NBN
po beta	Parameter for the improved NBN
po gama	Parameter for the improved NBN
training times	Training times for automatic running

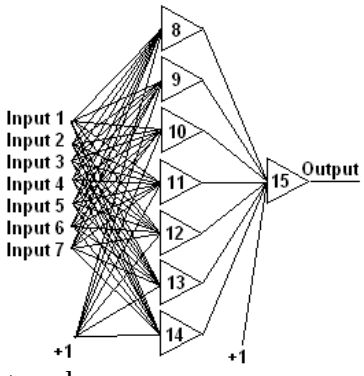
There are two ways to set those parameters: (1) Edit the parameter file manually, according to the descriptions of parameters in Table 6-1; (2) All those parameters can be edited in the user interface, and they will be saved in the parameter file automatically once training is executed, as the initial values for next time of running the software.

6.1.3 *Topology Files*

Topology files are named "*.in", and they are mainly used to construct the neural network topologies for training. Topology files consist of four parts: (1) topology design; (2) weight initialization (optional); (3) neuron type instruction and (4) training data specification.

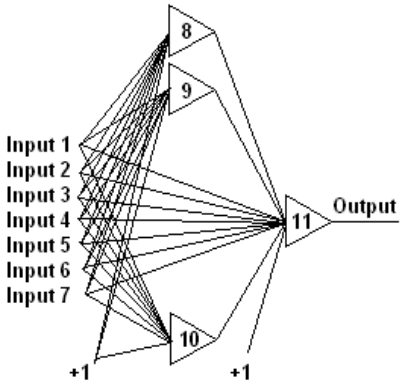
(1) **Topology design**: the topology design is aimed to create neural structures. The general format of the command is "n [*b*] [*type*] [a_1 a_2 ... a_n]", which means inputs/neurons indexed with a_1, a_2...a_n are connected to neuron *b* with a specified neural type (bipolar, unipolar or linear). Fig. 6-1 presents the commands and the related neural network topologies. Notice: the neuron type *mbip* stands for bipolar neurons; *mu* is for unipolar neurons and *mlin* is for linear neurons. They types are defined in neuron type instructions.

```
//8 neurons in 7–7–1 MLP network
n 8   mbip 1 2 3 4 5 6 7
n 9   mbip 1 2 3 4 5 6 7
n 10  mbip 1 2 3 4 5 6 7
n 11  mbip 1 2 3 4 5 6 7
n 12  mbip 1 2 3 4 5 6 7
n 13  mbip 1 2 3 4 5 6 7
n 14  mbip 1 2 3 4 5 6 7
n 15  mbip 8 9 10 11 12 13 14
```



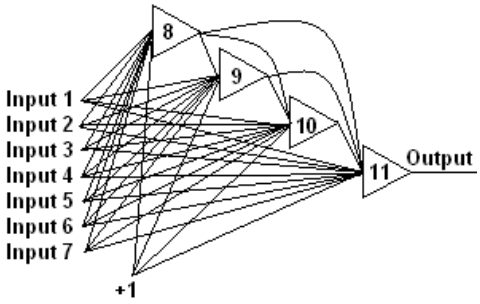
(a) 7-7-7 MLP network

```
//4 neurons in 7=3=1 BMLP network
n 8   mbip 1 2 3 4 5 6 7
n 9   mbip 1 2 3 4 5 6 7
n 10  mbip 1 2 3 4 5 6 7
n 11  mbip 1 2 3 4 5 6 7 8 9 10
```



(b) 7=3=1 BMLP network

```
//4 neurons in FCC network
n 8   mbip 1 2 3 4 5 6 7
n 9   mbip 1 2 3 4 5 6 7 8
n 10  mbip 1 2 3 4 5 6 7 8 9
n 11  mbip 1 2 3 4 5 6 7 8 9 10
```



(c) 7=1=1=1=1 FCC network

Fig. 6-1 Commands and related neural network topologies

(2) **Weight initialization**: the weight initialization part is used to specify initial weights for training and this part is optional. If there is no weight initialization in the topology file, the software will generate all the initial weights randomly (from -1 to 1) before training. The general command is "w [w_{bias}] [w_1 w_2 ... w_n]", corresponding to the topology design. Fig. 6-2 shows the example of weight initialization for parity-3 problem with 2 neurons in FCC network.

```
//Topology design
n 4 mbip 1 2 3
n 5 mbip 1 2 3 4

//Weight initialization
w 0.11 0.12 0.13 0.14
w 0.15 0.16 0.17 0.18 0.19
```

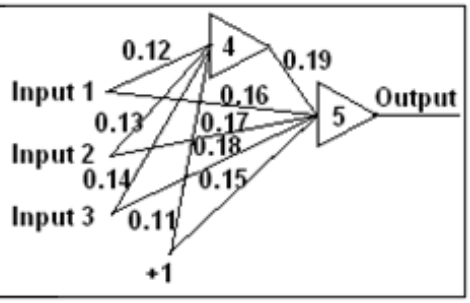


Fig. 6-2 Weight initialization for parity-3 problem with 2 neurons in FCC network

(3) **Neuron type**: in the neuron type instruction part, three different types of neurons are defined. They are bipolar ("*mbip*"), unipolar (*mu*) and linear ("mlin"). Bipolar neurons have both positive and negative outputs, while unipolar neurons only have positive outputs. The outputs of both bipolar and unipolar neurons are no more than 1. If the desired outputs are larger than 1, linear neurons are considered to be the output neurons. The general command is ".model [*mbip/mu/mlin*] fun=[*bip/uni/lin*], gain=[*value*], der=[*value*]". Table 6-2 presents the three types of neurons used in the software.

Table 6-2 Three types of neurons in the software

Neuron Types/Commands	***Activation Functions***
bipolar *.model mbip fun=bip, gain=0.5, der=0.001*	$f_b(net) = \frac{2}{1+e^{-gain \times net}} - 1 + der \times net$
unipolar *.model mu fun=uni, gain=0.5, der=0.001*	$f_u(net) = \frac{1}{1+e^{-gain \times net}} + der \times net$
linear *.model mlin fun=lin, gain=1, der=0.005*	$f_l(net) = gain \times net$

From Table 6-2, it can be seen that "gain" and "der" are parameters of activation functions. Parameter "der" is introduced to adjust the slope of activation function (for unipolar and bipolar), which is a trick we used in the software to avoid training process entering the saturation region where slope is approching to zero.

(4) **Training data**: the training data specification part is used to set the name of training pattern file, in order to get correct training data. The general command is "datafile=[*file name*]". The *file name* needs to be specified by users.

With at least three part settings (weight initialization is optional), the topology file can be correctly defined.

6.1.4 *Training Pattern Files*

The training pattern files include input patterns and related desired outputs. In a training pattern file, the number of rows is equal to the number of patterns, while the number of columns is equal to the sum of the number of inputs and the number of outputs. However, only with the data in training pattern file, one can't tell the number of inputs and the number of outputs, so the neural topology file should be considered together in order to decide those two parameters (Fig. 6-3). The training pattern files are specified in the topology files as mentioned above, and they should be in the same folder as related topology files.

training data	*topology*	*explanation*
-1 -1 -1 -1 -1 -1 1 1 -1 1 -1 1 -1 1 1 -1 1 -1 -1 1 1 -1 1 -1 1 1 -1 -1 1 1 1 1	//2 inputs and 2 outputs n 3 mbip 1 2 n 4 mbip 1 2	In command "n 3 mbip 1 2", the index of the first neuron is 3, so the number of inputs is 2, and the number of outputs is 2
	//3 inputs and 1 output n 4 mbip 1 2 3 n 5 mbip 1 2 3 4	In command "n 4 mbip 1 2 3", the index of the first neuron is 4, so the number of inputs is 3, and the number of outputs is 1

Fig. 6-3 Extract the number of inputs and the number of outputs from the data file and topology

As described in Fig. 6-3, the number of inputs is obtained from the first command line of topology design and it is equal to the index of the first neuron minus 1. After that, the number of outputs is calculated by the number of columns in training pattern files minus the number of inputs.

6.1.5 *Training Result Files*

Training result files are used to store the training information and results. Once the "save data" function is enabled in the software (by users), important information for current training, such as training algorithm, training pattern file, topology, parameters, initial weights, result weights and

training results, will be saved after the training is finished. The name of the training result file is generated automatically depending on the starting time and the format is "*date_time*_result.txt". Fig. 6-4 shows a sample of training result file.

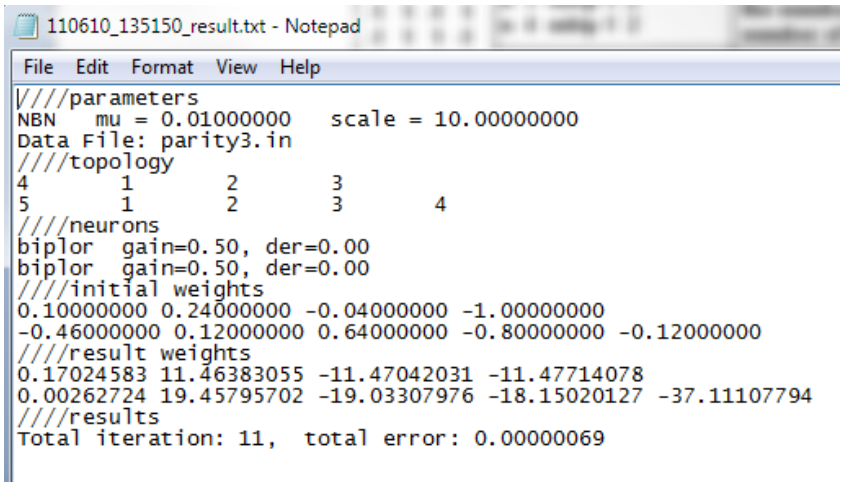


Fig. 6-4 A sample of training result file

6.1.6 *Training Verification Files*

Training verification files are generated by the software when the verification function is performed (by users). The result weights from the current training will be verified, by computing the actual outputs of related patterns. The name of training verification file is also created by the system time when the verification starts and it is "*date_time*_verification.txt". Fig. 6-5 gives a sample of training verification file for parity-3 problem.

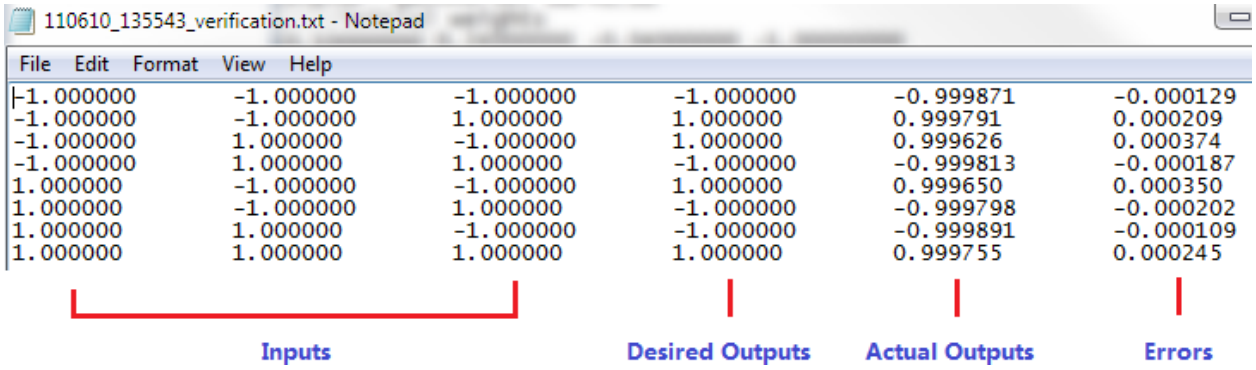


Fig. 6-5 A sample of training verification file for parity-3 problem

6.2 Graphic User Interface Instruction

As shown in Fig. 6-6, the user interface consists of 6 areas: (1) Plotting area; (2) Training information area; (3) Plot modes setting area; (4) Execute modes setting area; (5) Control area; (6) Parameter setting area; (7) Verification area; (8) Command consoler area.

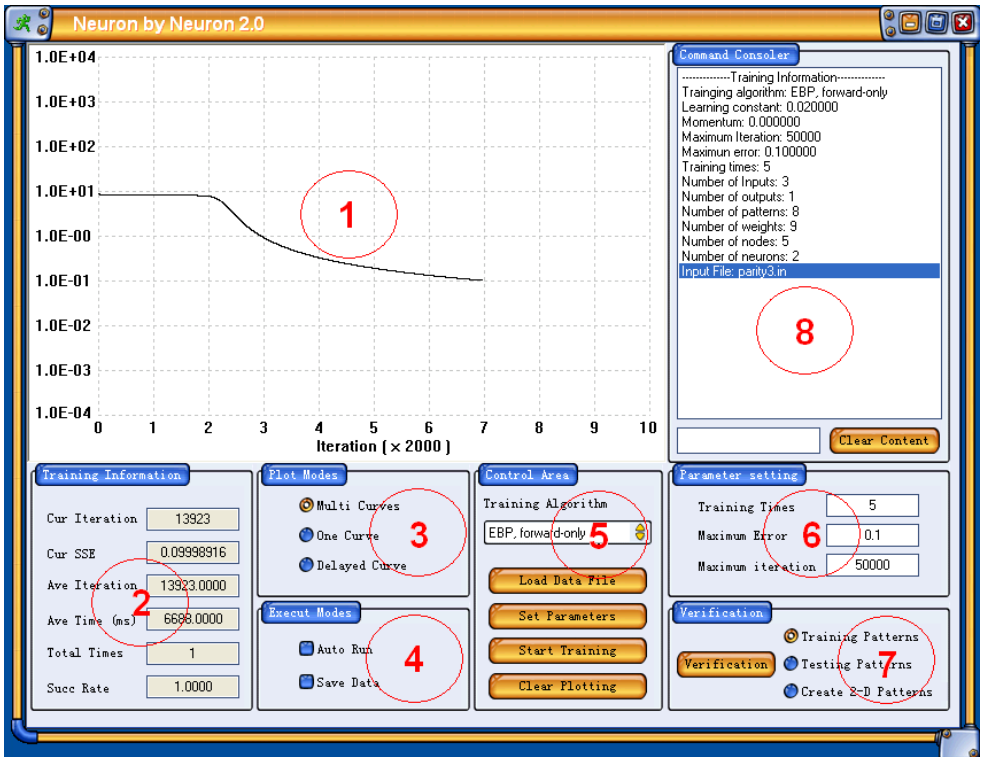


Fig. 6-6 The user interface of NBN 2.0

6.2.1 *Plotting Area*

This area is marked as ① in Fig. 6-6 and it is used to depict the sum squared error (SSE) during training. The log scaled vertical axis presents SSE values from 0.0001 to 10000, while the horizontal axis, which is linearly scaled automatically with the coefficient at the bottom of plotting area ("×[*value*]" in Fig. 6-6), shows the number of iterations cost for training. Current training process or previous training process can be recorded in the same plotting, depending on the users' settings.

6.2.2 *Training Information Area*

This area is marked as ② in Fig. 6-6 and instantaneous training data are presented in this area, including sum square error (SSE) and cost iterations for current training, average iteration and time spent in solving the same problems, and the success rate for multiple trials.

6.2.3 *Plot Modes Setting Area*

This area is marked as ③ in Fig. 6-6. Three plot modes are available in this software, multi curves, one curve and delayed curve. In multi curves mode, all the training curves will be plotted together and updated instantaneously. In one curve mode, only current training is plotted, while other curves will be erased. The delayed curve mode is only used in automatic training for multiple trials: during the training process, there is no plotting; while all the curves will be shown together after the whole training process is finished. The delayed curve mode is designed for training process which needs huge iterations and costs time for plotting.

6.2.4 *Execute Modes Setting Area*

This area is marked as ④ in Fig. 6-6 and used to control training mode, either training one time or automatic training for several trials, either saving the training results or not.

If it is set to run automatically, the train will not stop unless it reaches the required training times or the "Stop to Train" button is clicked. The default value of automatically training trials is 100 and it can be changed through command consoler (area ⑧) or parameter settings (area ⑥).

If it is set to save training data, all the important information, such as algorithm type, topology, training parameters, initial weights, result weights and training results (SSE and cost

iteration) will be saved in training result files (discussed in section 6.1.5).

6.2.5 *Control Area*

This area is marked as ⑤ in Fig. 6-6. The combo box is used to select training algorithms. There are 6 choices: (1) "EBP"; (2) "LM"; (3) "NBN"; (4) "EBP, forward-only"; (5) "NBN, forward-only"; (6) "NBN, improved". All algorithms will be introduced in section 6.3.

Button "Load Data File" is used to choose a topology file for training; button "Set Parameters" is used to set training parameters for the selected training algorithm; button "Start To Train/Stop To Train" helps control the training process; button "Clear Plotting" is used to erase the current plotting in the plotting area.

6.2.6 *Parameter Setting Area*

This area is marked as ⑥ in Fig. 6-6 and is used to set training related parameters, such as training times, maximum error and maximum iteration. The *training times* specifies the number of trials if automatic training is enabled. The *maximum error* is used to judge whether the training process reaches convergence. The *maximum iteration* limits the number of iterations for each training case: training process stops if it reaches the value of *maximum iteration*. All the settings will be saved in the parameter file once training is executed, and they will be loaded as the initial values for the next time using the software.

6.2.7 *Verification Area*

This area is marked as ⑦ in Fig. 6-6 and it is used for training results verification, by calculating the actual outputs for each pattern with the result weights. The verifying patterns can be training

patterns, testing patterns or user created patterns for 2-D situation. The verification results will be stored in training verification files (discussed in section 6.1.6) automatically and shown as popup windows.

The verification results can be easily uploaded by Matlab, MS Excel, Origin, or other software for analysis and plotting. For 2-D input patterns, the verification data can be plotted in Matlab by the accessory tool included in the software, named as "PlotFor2D.m".

6.2.8 *Command Consoler Area*

The list box is used to show the important information or hints for users' operations. It is also a command consoler. Most of the operations can be achieved by related commands. Table 6-3 presents the available commands and their functions.

Table 6-3 Available commands and related functionalities

Commands	***Functions***
help	list all the available commands and instructions
clr	clear the content of the list box
cc	clear all the curves in plotting area
sus	suspend training and save current status
res	resume training with the status at suspending point
tra	start/stop training
sav	data saving control
aut	automatic training control
pm=	select plotting mode, e.g. pm=2 (one curve mode)
load	load input file
para	set training parameters for selected algorithm, e.g. para
info	show current training setting
iw	show current iw value (used for debug)
topo	show current topology
alpha? / alpha=	get/set learning constant, e.g. alpha=1
mom? / mom=	get/set momentum, e.g. mom=0.001
tt? / tt=	get/set training times for automatic training, e.g. tt?
me? / me=	get/set maximum error, e.g. me=0.01
mi? / mi=	get/set maximum iteration, e.g. mi=100
th? / th=	get/set parameter for the "NBN, improved" algorithm

6.3 Implemented algorithms

As introduced above, there are 6 available algorithms in the software for training. The following part is going to introduce the characteristics and limitations for each algorithm.

EBP: This is EBP algorithm with traditional forward-backward computation. For EBP algorithm, the forward-back computation may work a little bit faster than forward-only computation. Now it is only used for standard MLP networks. EBP algorithm can be used for huge patterns training because of its simplification, and the tradeoff is the slow convergence.

LM: This is LM algorithm with traditional forward-backward computation. For LM (and NBN) algorithm, the improved forward-only computation [78] (discussed in chapter 5) performs faster training than forward-backward computation for networks with multiple outputs. Now it is also only used for standard MLP networks. LM (and NBN) algorithm converges much faster than EBP algorithm for small and media sized patterns training. For huge patterns (huge Jacobian matrix, we have solved this problem) and huge networks (huge Hessian matrix), it may work slower than EBP algorithm.

NBN: This is NBN algorithm with forward-backward computation. NBN algorithm is developed based on LM algorithm, but it can handle arbitrarily connected neuron (ACN) networks, also, the convergence is improved.

EBP, forward-only: This is EBP algorithm with forward-only computation [78]. It can work on arbitrarily connected neuron networks.

NBN, forward-only: This is NBN algorithm with forward-only computation [78]. It can handle arbitrarily connected neuron networks and, as mentioned above, it works faster than "NBN" algorithm, especially for networks with multiple outputs.

NBN, improved: This is a newly developed second order algorithm, implemented with forward-only computation, so it can handle arbitrarily connected neuron networks. In this algorithm, Hessian matrix is inverted only one time per iteration, so this algorithm is supposed to compute faster than LM (and NBN) algorithm which may have several times Hessian matrix inversion per iteration. The train ability (convergence) is also improved in this algorithm. Furthermore, a local minima detector is implemented in this algorithm. When the detector diagnoses that the training is trapped in local minima, all the weights will be regenerated randomly for further training. The detailed description of this algorithm will be introduced in section 6.4.3.

For all second order algorithms, the improved second order computation [79] discussed in chapter 4 is applied in the implementation.

6.4 Strategies for Improving Training Performance

Besides the neuron-by-neuron algorithm [27], forward-only algorithm [78] and the improved second order computation [79], there are several other strategies and algorithms implemented in the software in order to improve the computation efficiency and convergent ability: (1) In first order algorithms, momentum [95] is incorporated to stabilize the training process, so that big learning constants can be applied to speed up to convergence; (2) In first order algorithms, the slope of activation function is modified [97] to enhance the convergent ability; (3) In second order algorithms, Levenberg Marquardt update rule is modified routing to reduce the matrix inversion operations, so as to accelerate the training process; (4) Sigmoidal activation function is modified to enhance computation, so as to improve the convergence.

6.4.1 *Momentum in First Order Gradient Search*

It was shown in section 2.4 that, for EBP algorithm, applying big learning constants may speed up the convergence, or lead to oscillation (Fig. 2-14). The momentum concept proposed in [95] was used to stabilize the training process, so that big learning constants can be applied to accelerate the training process without oscillation.

As discussed in section 2.4, the original update rule of EBP algorithm is

$$\Delta \mathbf{w}_k = -\alpha \mathbf{g}_k \tag{6-1}$$

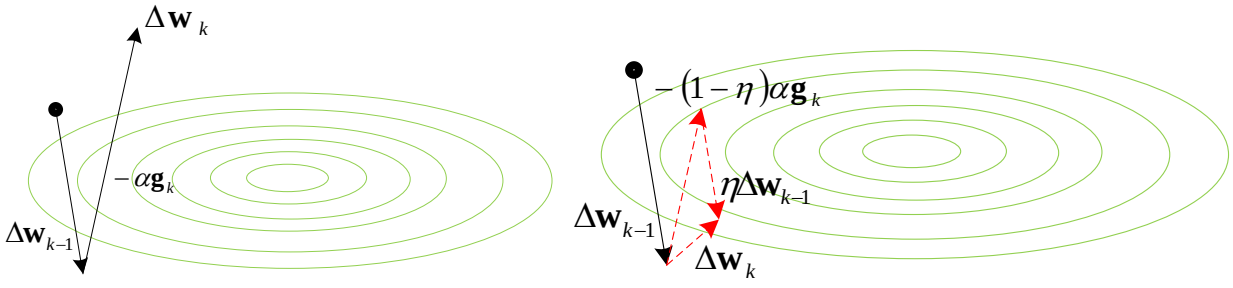
Where: k is the index of iteration and α is the learning constant.

By incorporating the momentum, the update rule can be modified to

$$\Delta \mathbf{w}_k = -(1-\eta)\alpha\, \mathbf{g}_k + \Delta \mathbf{w}_{k-1}\eta \tag{6-2}$$

Where: η is the momentum ranged between [0, 1].

Fig. 6-7 interprets how the momentum stabilizes the training process. As shown in Fig. 6-7a, the training process oscillates because the direction of current weight updating $\Delta \boldsymbol{w}_k$ is completely far from minima (center point). By adding momentum η, the current weight updating $\Delta \boldsymbol{w}_k$ is recalculated as the combination of part of the previous information $\Delta \boldsymbol{w}_{k-1}$ and part of the current gradient $\boldsymbol{g}_k$, as shown in Fig. 6-7b. One may notice that the $\Delta \boldsymbol{w}_k$ in Fig. 6-7b is more likely to stably approaching to the center point than the $\Delta \boldsymbol{w}_k$ in Fig. 6-7a.



(a) Oscillation (b) Stabilization

Fig. 6-7 Training process with and without momentum

Let us use the XOR problem as an example to illustrate the advantage of the modified update rule (6-2) by comparing with update rule (6-1). Fig. 6-8 is the neural network used in the experiment. It consists of 2 inputs, 2 neurons in FCC network and 1 output.

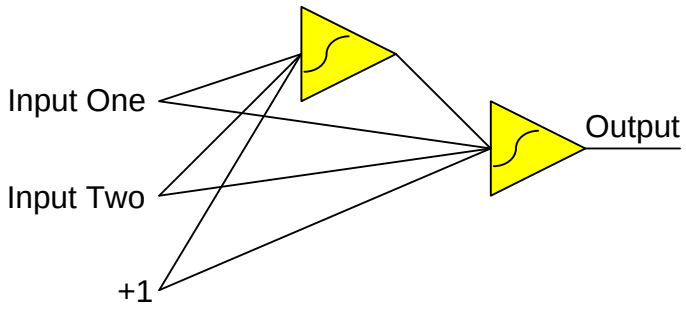
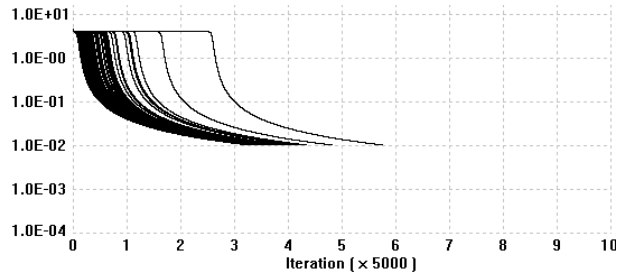
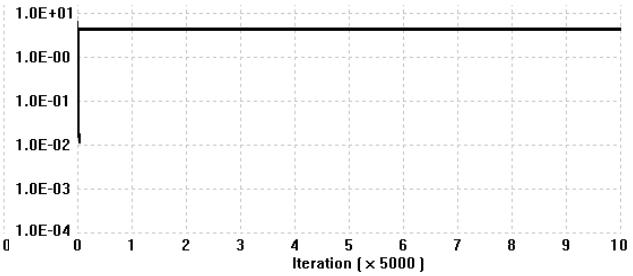


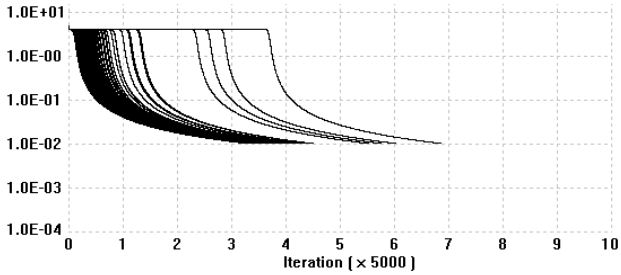
Fig. 6-8 Network architecture used for XOR problem



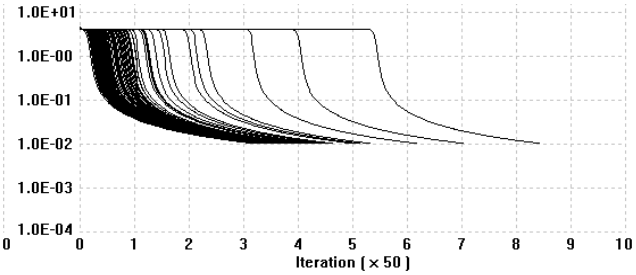
(a) Update rule (8-1), α=0.1



(b) Update rule (8-1), α=10



(c) Update rule (8-2), α=0.1 and η=0.5



(d) Update rule (8-2), α=10 and η=0.5

Fig. 6-9 Training results of XOR problem

Table 6-4 Comparison of different EBP algorithms for solving XOR problem

XOR problem	α=0.1	α=10	α=0.1 η=0.5	α=10 η=0.5
success rate	100%	18%	100%	100%
average iteration	17845.44	179.00	18415.84	187.76
average time (ms)	3413.26	46.83	4687.79	49.27

For each test case, the training process is repeated for 100 trials with randomly generated initial weights in range [-1, 1]. Fig. 6-9a and Fig. 6-9b present the training results using update rule (6-1). It can be seen that, the bigger the learning constant is, the less iteration it costs for convergence. However, too bigger learning constants also cause the divergence and lower the rate of successful training. Fig. 6-9c and Fig. 6-9d show the training results using update rule (6-2). One may notice that, when using the same learning constant, the update rule (6-2) with momentum converges a little bit slower than the update rule (6-1), but it definitely stabilizes the training process with big learning constants (α=10 in the example).

6.4.2 *Modified Slope*

The famous "flap spot" problem is that if the slope of the neuron is very small, while the error is huge, then the training will be pushed into the saturate region and get stuck. In order to avoid being trapped in saturate region, an equivalent slope is introduced instead of the derivative of activation function (see Fig. 6-10), and it is calculated by the followed algorithm [97]:

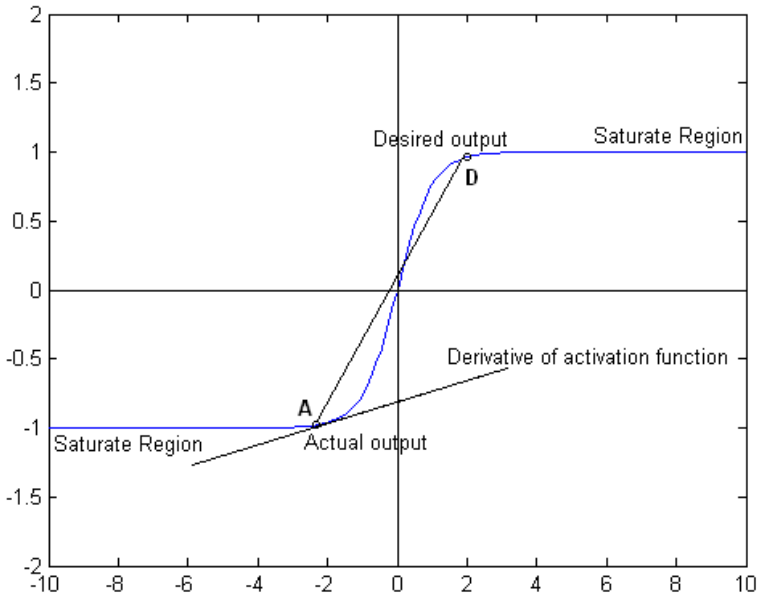
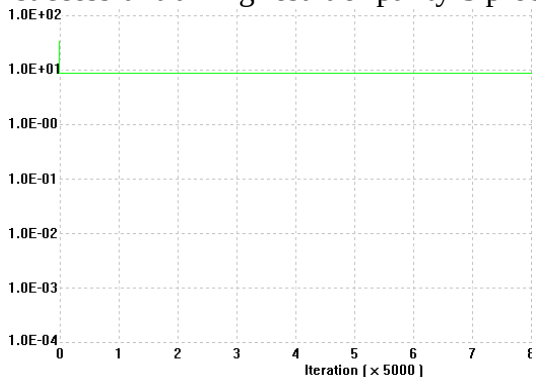


Fig. 6-10 The "flat spot" problem in sigmoidal activation function

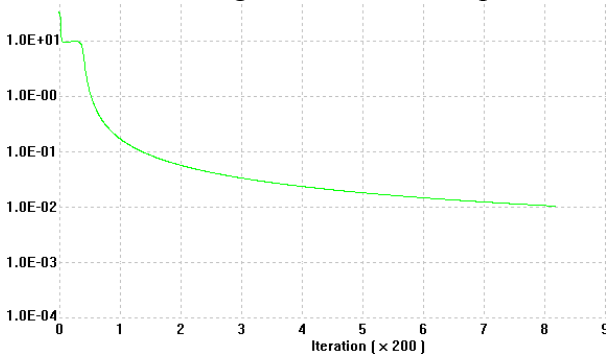
Compute derivative of activation function;
Compute the slope of line AD S_{AD};
If derivate > S_{AD} improved slope = derivative;
Else improved slope = S_{AD};

In order to test the modification, the "worst case" training is performed. The "worst case" training is to use a successful training result as the initial status to train the same patterns with all outputs reversed. For parity-3 problem, the training result of "worst case" is showed in Fig. 6-11.

(a) A successful training result of parity-3 problem



(b) "Worst case" training result without slope modification



(c) "Worst case" training with slope modification

Fig. 6-11 Test the modified slope by "worst case" training

From the results showed in Fig. 6-11, it is clear that the modified slope works well for "worst case" training, which means the convergent ability of first order algorithms is enhanced.

6.4.3 *Modified Second Order Update Rule*

The update rule of Levenberg Marquardt algorithm is

$$\Delta \boldsymbol{w}_k = -\left(\boldsymbol{J}_k^T \boldsymbol{J} + \mu \boldsymbol{I}\right)^{-1} \boldsymbol{J}_k^T \boldsymbol{e}_k \tag{6-3}$$

During the training process using Levenberg Marquardt update rule, parameter μ may be adjusted (multiplied or divided by a constant) for several times in each iteration. From (6-3), it can be known that matrix $(\boldsymbol{J}^T\boldsymbol{J}+\mu\boldsymbol{I})$ needs to be inverted for each adjustment of parameter μ. In order to avoid the multiple time matrix inversion in single iteration, the update rule can be modified as

$$\Delta \boldsymbol{w}_k = -\left((1-\tanh\beta)\alpha \boldsymbol{J}_k^T \boldsymbol{e}_k + \tanh\beta\left(\boldsymbol{J}_k^T \boldsymbol{J}_k + \mu \boldsymbol{I}\right)^{-1} \boldsymbol{J}_k^T \boldsymbol{e}_k\right) \tag{6-4}$$

In the update rule (6-4), there are three parameters, learning constant α, combination coefficient μ and conjugate coefficient β. For each iteration, learning constant α is fixed as a small constant; while μ and β are adjusted only once according to the algorithm in Fig. 6-12.

If $SSE_k < SSE_{k-1}$
Then $\beta = \beta \times P$ and $\mu = \mu / P$;
Else
$\beta = \beta / P$ and $\mu = \mu \times P$;

Fig. 6-12 Parameter adjustment in update rule (6-4)

Several parity problems are applied to test the performance of the modified update rule (6-4), by comparing with (6-3). In the experiment, each testing case is repeated for 100 trials with randomly generated initial conditions between -1 and 1. The desired training error is 0.01

and the maximum iteration is 500. The gain parameter in activation function is set as 0.5. All neurons are connected in FCC networks and NBN computation is applied for training.

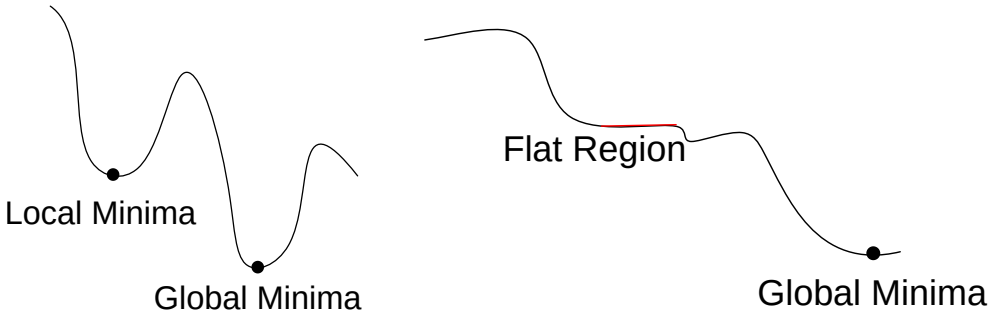
Table 6-5 Testing results of parity problems using update rules (6-3) and (6-4)

Pairty-N	Neurons	Ave. # of Iterations		Ave. # of Training Time (ms)		Success Rates	
		(6-3)	(6-4)	(6-3)	(6-4)	(6-3)	(6-4)
5	3	21.1176	24.9403	63.7843	14.3881	51%	67%
7	4	21.1667	39.6176	103.3333	90.9118	22%	36%
9	5	32.1250	41.3333	527.5000	459.3667	8%	30%
11	6	51.3333	56.1250	8052.0000	4548.7500	3%	16%

As the testing results presented in Table 6-5, it can be noticed that, even though it takes more iterations for the update rule (6-4) than (6-3), the computation time in (6-4) is reduced because there is only one time matrix inversion per iteration. Moreover, a little bit improved success rate is obtained for each testing case, which means the update rule (6-4) gets more powerful search ability than (6-3).

6.4.4 *Modified Activation Function*

The convergent ability of gradient descent methods is limited by local minima and "flat region". For one dimension case, the local minima and flat region are visualized in Fig. 6-13.



(a) Local minima (b) Flat Region

Fig. 6-13 Failures of gradient based optimization

From the standpoint of computation, when the training process is trapped in local minima or flat region, elements of gradient are approaching to zero and weight updating gets stuck. In real neural network training cases, the local minima problem can be overcome by properly increasing the number of network size.

Considering the sigmoidal shape of activation function, there is no absolutely flat region on the error surface. Normally, the training stuck in the "flat region" means the elements of gradient vector are approximated to zero by compilers because of the limitation of computer precision. For 32-bit CPU, the "double" type variable can only have 15-16 bits precision at the right side of the point. Therefore, when the elements of gradient are not exact 0, but smaller than the precision limit, they will be approximated as 0 by compiler. This is a hardware limitation on software computation and cannot be avoided. However, the computation can be furthered by programming strategies, such as enlarging the output range of activation functions.

The common used sigmoidal activation function for neural network training is:

$$f(x) = \frac{2}{1+\exp(-\beta \times x)} - 1 \tag{6-5}$$

Equation (6-5) presents the behaviors of bipolar neurons and it is ranged from [-1, 1]. In order to enlarge the output range, a factor can be applied and (6-6) is rewritten as

$$f(x) = \left(\frac{2}{1+\exp(-\beta \times x)} - 1 \right) \times \alpha \tag{6-6}$$

Where: the parameter α is larger than "1". So the output of the enlarged bipolar neuron will be [-α, α].

Several parity-N problems are applied to test the performance of modified activation function (6-6), by comparing with the commonly used (6-5). In the experiments, all neurons are in fully connected cascade (FCC) networks and NBN algorithm is used for training. The desired

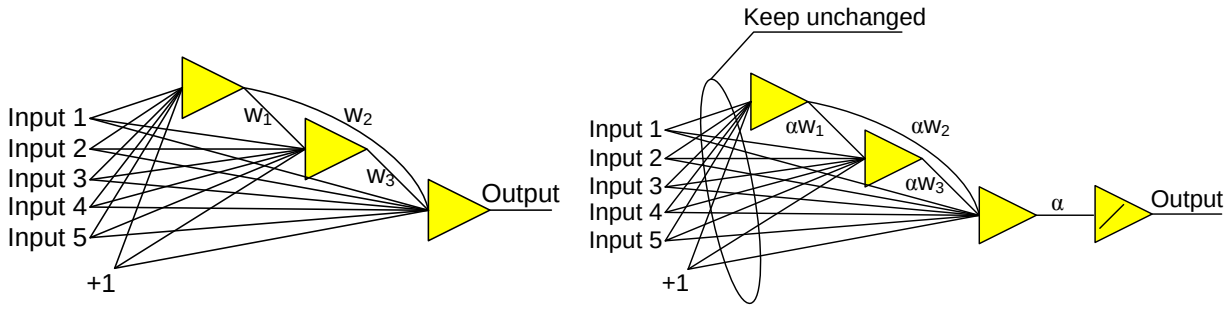
training accuracy is 0.01 and the maximum iteration is 1000. For each case, the testing is repeated for 100 trials with randomly generated initial weights between [-1, 1].

Table 6-6 Testing results of parity-N problems using different activation functions with the minimal network architecture analyzed in section 2.3

Parity-N	Neurons	Parameters	Iteration	Times (ms)	Success Rate
2	2	α=20, β=0.1	13.10 (8.83)	17.60 (6.94)	100% (100%)
3	2	α=20, β=0.1	12.22(8.08)	14.94 (7.13)	100% (99%)
4	3	α=20, β=0.1	54.24 (22.65)	76.55 (38.69)	100% (45%)
5	3	α=20, β=0.1	50.32 (21.29)	75.84 (91.91)	100% (37%)
6	3	α=25, β=0.25	502.39 (/)	998.60 (/)	98% (0%)
7	3	α=25, β=0.25	117.06 (/)	459.87 (/)	98% (0%)
8	4	α=25, β=0.25	292.79 (/)	1869.47 (/)	93% (0%)
9	4	α=25, β=0.3	383.48 (/)	4367.27 (/)	77% (0%)
10	4	α=25, β=0.3	1260.12 (/)	30180.31 (/)	21% (0%)

Table 6-6 presents the testing results. Testing results for activation function (6-5) are in the parenthesis. One may notice that, with the modified activation function (6-6), the training success rates are significantly improved and the tradeoff is extra training time. Notice that, in chapter 2, the analytical results show that using FCC networks, parity-7 problem can be solved with 3 neurons. Networks with activation function (6-5) cannot solve the parity-7 problem at all; while with the modified activation function (6-6), the training success rate can reach 98% for parity-7 problem with 3 neurons in FCC network.

It is worth to mention that the trained network using the modified activation function (6-6) can be scaled back as a trained network using the activation function (6-5). As shown in Fig. 6-14, the two networks are equivalent: network in Fig. 6-14a is trained using the modified activation function (6-6); while network in Fig. 6-14b consists of the neurons with activation function (6-7).



(a) Neurons with activation functions in (6-6) (b) Neurons with activation functions in (6-7)

Fig. 6-14 Two equivalent networks

6.5 Case Studies Using NBN 2.0

In industrial application, dataset is often obtained by sampling and testing process. In order to get the responses of inputs which are not sampled for testing, an approximated function has to be build to represent the relationship between stimulus and response, based on the known sampled testing results. Neural networks can always be considered as a candidate of approximator and the software NBN 2.0 is recommended for neural network design.

In this section, two examples are presented to test the abilities of NBN 2.0, from the standpoints of data classification and function approximation. Two-dimensional examples are considered so that the verification results can be visualized.

6.5.1 *Data Classification*

Neural networks could be considered as a supervised data classification method. In this example, we got *21×21=441* two-dimensional dataset uniformly sampled from $x \in [-1, 1]$ and $y \in [-1, 1]$. As shown in Fig. 6-16a, there are 5 groups, A, B, C, D and E, in the sampling range. The purpose is to build a classification system which can classify the non-sample inputs into correct groups.

The first step is to define the expression of each group. Considering the network with outputs, without losing generalization, let us define output of group A as "*0 0 0 0 1*", group B as

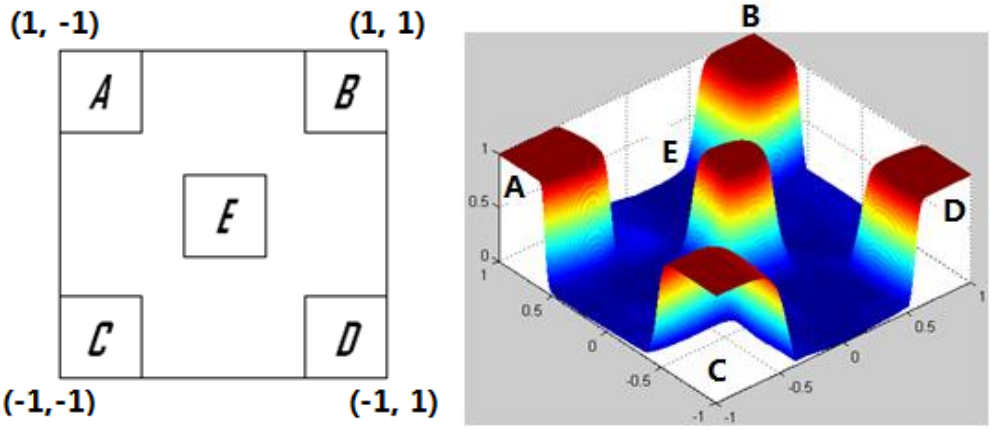
"*0 0 0 1 0*", group C as "*0 0 1 0 0*", group D as "*0 1 0 0 0*" and group E as "*1 0 0 0 0*". For other cases, the output is defined as "*0 0 0 0 0*".

15 neurons in FCC network are used to build the system and NBN algorithm is applied for system parameter adjustment. Fig. 6-15 shows the network construction commands in topology file.

```
n 3  mbip 1 2
n 4  mbip 1 2 3
n 5  mbip 1 2 3 4
n 6  mbip 1 2 3 4 5
n 7  mbip 1 2 3 4 5 6
n 8  mbip 1 2 3 4 5 6 7
n 9  mbip 1 2 3 4 5 6 7 8
n 10 mbip 1 2 3 4 5 6 7 8 9
n 11 mbip 1 2 3 4 5 6 7 8 9 10
n 12 mbip 1 2 3 4 5 6 7 8 9 10 11
n 13 mbip 1 2 3 4 5 6 7 8 9 10 11 12
n 14 mbip 1 2 3 4 5 6 7 8 9 10 11 12
n 15 mbip 1 2 3 4 5 6 7 8 9 10 11 12
n 16 mbip 1 2 3 4 5 6 7 8 9 10 11 12
n 17 mbip 1 2 3 4 5 6 7 8 9 10 11 12
```

Fig. 6-15 Network construction commands: 15 neurons in FCC network with 2 inputs and 5 outputs

After successful training (training sum square error < 0.01), *50×50=2,500* patterns in the same range as sample patterns are applied to test the trained networks. Testing patterns are generalized by click the button "Generalization" as introduced in section 6.2.7. Fig. 6-16b shows the visualized testing results and it can be noticed that the five groups are classified correctly.



(a) Location of the five groups (b) Generalization result with *50×50=2,500* points

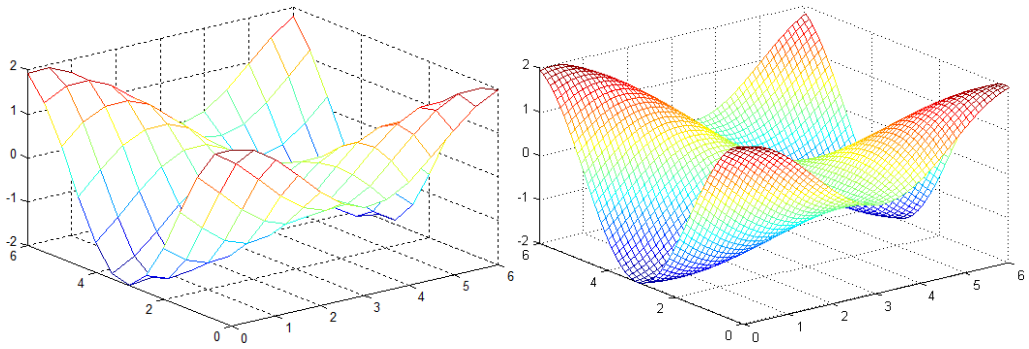
Fig. 6-16 Data classification

6.5.2 *Function Approximation*

In the experiment, the purpose is to simulate the behaviors of forward kinematics described in Fig. 5-17. The location (*x*, *y*) of the end effector is given by the two equations (5-30) and (5-31). In order to avoid scanning "blind area", let us assume the arm lengths *L1=L2*=1. The sample range of angles α and β are: $\alpha \in [0, 6]$ and $\beta \in [0, 6]$. For each dimension, 13×13=169 points are uniformly sampled as training dataset; after training, 61×61=3,721 points in the same range as sampling are uniformly generated as testing dataset. All the training/testing points are visualized in Fig. 6-17 and Fig. 6-18 for X and Y dimension, respectively. The approximation results are evaluated using the averaged sum square error *E* defined as:

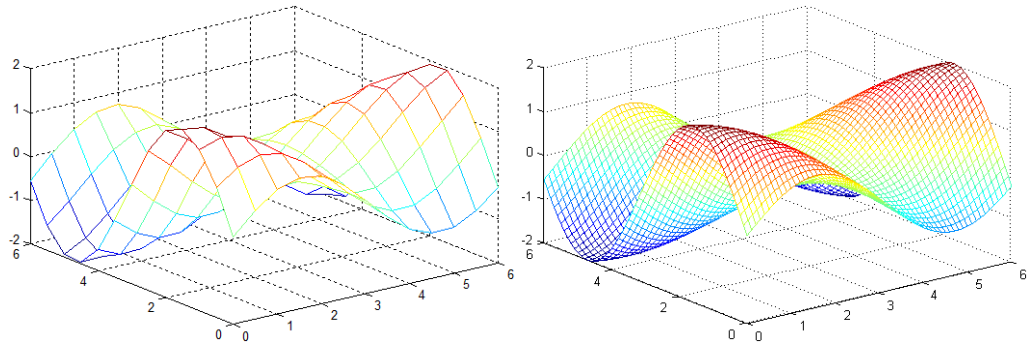
$$E = \frac{1}{P}\sum_{p=1}^{P} e_p^2 \tag{6-7}$$

Where: e_p is the difference between desired output (from equations 6-30 or 6-31) and actual output (from designed neural networks) for a given pattern *p*. *P* is the number of training/testing patterns.



(a) Training patterns, 13×13=169 points (b) Testing patterns, 61×61=3,721 points

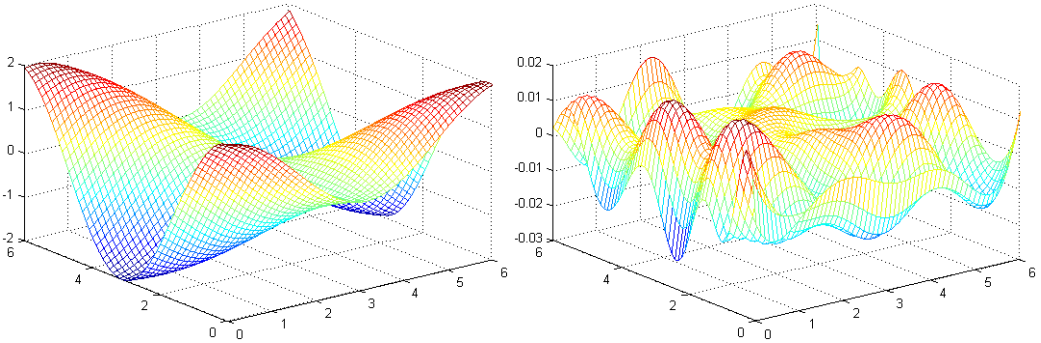
Fig. 6-17 X-dimension surface of forward kinematics



(a) Training patterns, 13×13=169 points (b) Testing patterns, 61×61=3,721 points

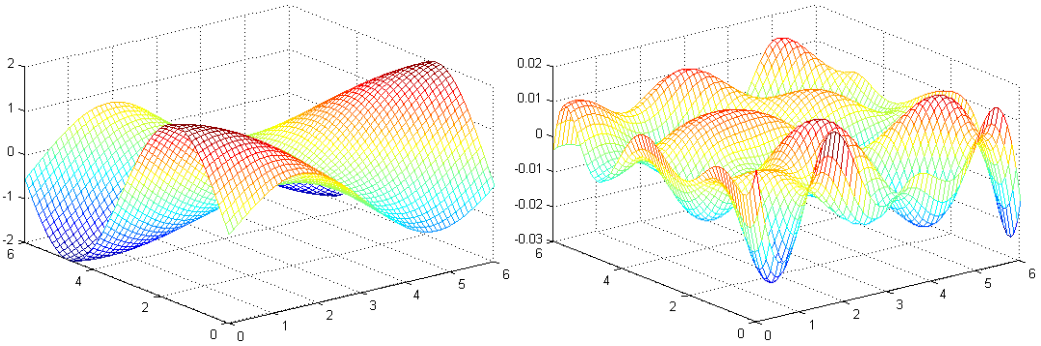
Fig. 6-18 Y-dimension surface of forward kinematics

For x-dimension approximation, using NBN algorithm, FCC network with 8 neurons can reach the training accuracy E_{Train}=5.7279e-005. The related test result is shown in Fig. 6-19. For y-dimension approximation, using NBN algorithm, FCC network with 6 neurons can obtain the training accuracy E_{Train}=5.7281e-005. The related test result is shown in Fig. 6-20.



(a) Testing surface, SSE_{Test}=5.3567e-005 (b) Error surface

Fig. 6-19 X-dimension testing results



(a) Testing surface, SSE_{Test}=5.1324e-005 (b) Error surface

Fig. 6-20 Y-dimension testing results

6.6 Conclusion

In this chapter, the software NBN 2.0 is introduced for neural network training. This software contains both first order and second order training algorithms, which are implemented by traditional forward-backward computation and a newly developed forward-only computation respectively. It can handle not only MLP networks, but also more powerful networks with connections across layers, such as BMLP networks and FCC networks.

Detailed instructions about the files and GUI operations of NBN 2.0 are presented. Also, several strategies used for improving the training performance are considered in the software. The momentum proposed in is incorporated in the first order algorithms in order to stabilize the training process with big learning constant, so as to speed up the convergence. The modified slope of activation function enhances the convergent ability of first order algorithms and the improvement is proven by "Worst cast" test. The other two strategies, modified second order update rule in section 6.4.3 and modified activation 6.4.4, are used to accelerate the computation and enhance the convergence, respectively. The latter two strategies are our recently unpublished work.

The software NBN 2.0 can be used for data analysis, such as data classification, pattern recognition and function approximation, as shown by the three examples in section 6.5. The NBN 2.0 is available at: http://www.eng.auburn.edu/users/wilambm/nnt/. And also, all the data of the examples presented in this chapter are included in the software package.

CHAPTER 7

CONCLUSION

The dissertation started with two interesting experiments to show potentially good behaviors of neural networks for pattern recognition and function approximation. Then, the efficiency of different neural network architectures were evaluated and compared based on parity problems. Analytically, we proved that neural networks with connections across layers, such as bridged multilayer perceptron (BMLP) networks and fully connected cascade (FCC) networks, are much more powerful and efficient than the popular multilayer perceptron (MLP) networks. Then, both first and second order gradient descent based learning algorithms were studied and derived from scratch. The comparison between first order algorithms, such as error backpropagation (EBP) algorithm [81], and second order algorithms, such as Gaussian-Newton method and Levenberg Marquardt (LM) algorithm [80], drew the conclusion that Levenberg Marquardt algorithm was indeed one of the most efficient and stable algorithms for neural network learning. Experimental results demonstrated the existence of over-fitting problem in neural network training which is mainly due to improperly choosing the size of neural networks. It was proposed that by utilizing efficient neural network architectures and second order learning algorithms, very compact neural networks could be designed for solving problems [28], so as to reduce the probability of occurrence of the over-fitting problem. The recently developed second order neuron-by-neuron (NBN) algorithm [27] is recommended for neural network training because it was designed to training efficient neural network architectures, such as BMLP networks and FCC networks.

The purpose of the dissertation was addressed by analyzing the disadvantages and computation inefficiencies in second order algorithms, including both the famous Hagan and Menhaj Levenberg Marquardt algorithm [80] and the powerful NBN algorithm [27]. There are three main disadvantages in Hagan and Menhaj Levenberg Marquardt algorithm and they are: (1) network limitation; (2) inefficient repetition in backpropagation computation; (3) memory limitation. The original NBN algorithm was highlighted because it solved the first problem. Being inherited from the NBN algorithm, the proposed forward-only computation [78] can handle not only traditional MLP networks, but also very efficient network architectures with connections across layers (solved problem one). By replacing the backpropagation process with extra computation in forward process of Jacobian matrix computation, the forward-only computation is more efficient than forward-backward (adopted by both Hagan and Menhaj LM algorithm and NBN algorithm) computation to handle networks with multiple outputs. The forward-only computation is designed based on a regular table (Fig. 5-7) and a general formula (5-22). The complex neural network training process with second order update rule is significantly simplified to a puzzle game: to obtain the unknown elements in a table based on the given rule (formula 5-22), which is very easy to be solved by computer programming. Another improvement in second order computation was proposed to solve the memory limitation (problem three) successfully [79]. Jacobian matrix storage was avoided and quasi Hessian matrix could be calculated directly. Experimental results showed that the training speed was also improved significantly because of the memory reduction. The improved second order computation made it possible to design neural networks using second order algorithms to handle problems with basically unlimited number of patterns. All the algorithms are implemented in the software NBN 2.0, developed based on Visual C++ 6.0. The NBN 2.0 is designed for neural

network training and can be also used to test the generalization ability of the trained neural networks.

So far, very efficient second order algorithms can be applied to train the powerful networks with connections across layers. Therefore, compact (close to optimal) neural networks can be obtained in practical applications to enhance the generalization ability. However, there are still several unsolved problems. For example, we know that compact neural networks should be applied to solve problems; however, we do not know how compact they should be? Trial-by-trial is often used as the possible way to find compact neural architecture but it is very time-consuming for complex problems. Some pruning/growing algorithms are introduced to find the optimized size of neural networks [11-12]. Recently, we have moved our research to a special type of neural networks: radial basis function (RBF) networks [99-100]. Comparing with traditional feedforward neural networks, RBF networks has the advantages of easy design, stable and good generalization ability, good tolerance to input noise and online learning ability. RBF networks are strongly recommended as an efficient and reliable way of designing dynamic systems [98]. Our recently developed error correction (ErrCor) algorithm [101] and improved second order computation [102] provide an efficient and robust way for design compact RBF networks.

REFERENCES

[1] P. J. Werbos, "Back-propagation: Past and Future," *Proceeding of International Conference on Neural Networks*, San Diego, CA, 1, 343-354, 1988.

[2] J. Moody and C. J. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, vol. 1, no. 2, pp. 281-294, 1989.

[3] R. Hecht-Nielsen, "Counterpropagation Networks," *Appl. Opt.* 26(23):4979-4984, 1987.

[4] A. G. Ivakhnenko and J. A. Mueller, "Self-Organizing of Nets of Active Neurons," *System Analysis Modeling Simulation*, vol. 20, pp. 93-106, 1995.

[5] Y. L. Chow, L. L. Frank and etc., "Disturbance and Friction Compensations in Hard Disk Drives Using Neural Networks," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 2, pp. 784-792, Feb. 2010.

[6] M. A. M. Radzi and N. A. Rahim, "Neural Network and Bandless Hysteresis Approach to Control Switched Capacitor Active Power Filter for Reduction of Harmonics," *IEEE Trans. on Industrial Electronics*, vol. 56, no. 5, pp. 1477-1484, May 2009.

[7] M. Cirrincione, M. Pucci, G. Vitale and A. Miraoui, "Current Harmonic Compensation by a Single-Phase Shunt Active Power Filter Controlled by Adaptive Neural Filtering," *IEEE Trans. on Industrial Electronics*, vol. 56, no. 8, pp. 3128-3143, Aug. 2009.

[8] Q. N. Le and J. W. Jeon, "Neural-Network-Based Low-Speed-Damping Controller for Stepper Motor With an FPGA," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 9, pp. 3167-3180, Sep. 2010.

[9] C. Xia, C. Guo and T. Shi, "A Neural-Network Identifier and Fuzzy-Controller-Based Algorithm for Dynamic Decoupling Control of Permanent-Magnet Spherical Motor," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 8, pp. 2868-2878, Aug. 2010.

[10] F. F. M. E. Sousy, "Hybrid H^{∞}-Based Wavelet-Neural-Network Tracking Control for Permanent-Magnet Synchronous Motor Servo Drives," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 9, pp. 3157-3166, Sep. 2010.

[11] Z. Li, "Robust Control of PM Spherical Stepper Motor Based on Neural Networks," *IEEE Trans. on Industrial Electronics*, vol. 56, no. 8, pp. 2945-2954, Aug. 2009.

[12] S. M. Gadoue, D. Giaouris and J. W. Finch, "Sensorless Control of Induction Motor Drives at Very Low and Zero Speeds Using Neural Network Flux Observers," *IEEE Trans. on Industrial Electronics*, vol. 56, no. 8, pp. 3029-3039, Aug. 2009.

[13] V. Machado, A. D. D. Neto and J. D. D. Melo, "A Neural Network Multiagent Architecture Applied to Industrial Networks for Dynamic Allocation of Control Strategies Using Standard Function Blocks," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 5, pp. 1823-1834, May 2010.

[14] H. P. Huang, J. L. Yan and T. H. Cheng, "Development and Fuzzy Control of a Pipe Inspection Robot," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 3, pp. 1088-1095, March 2010.

[15] H. Chaoui, P. Sicard and W. Gueaieb, "ANN-Based Adaptive Control of Robotic Manipulators With Friction and Joint Elasticity," *IEEE Trans. on Industrial Electronics*, vol. 56, no. 8, pp. 3174-3187, Aug. 2009.

[16] L. Y. Wang, T. Y. Chai and L. F. Zhai, "Neural-Network-Based Terminal Sliding-Mode Control of Robotic Manipulators Including Actuator Dynamics," *IEEE Trans. on Industrial Electronics*, vol. 56, no. 9, pp. 3296-3304, Sep. 2009.

[17] F. Moreno, J. Alarcón, R. Salvador and T. Riesgo, "Reconfigurable Hardware Architecture of a Shape Recognition System Based on Specialized Tiny Neural Networks With Online Training," *IEEE Trans. on Industrial Electronics*, vol. 56, no. 8, pp. 3253-3263, Aug. 2009.

[18] Y. J. Lee and J. Yoon, "Nonlinear Image Upsampling Method Based on Radial Basis Function Interpolation," *IEEE Trans. on Image Processing*, vol. 19, issue 10, pp. 2682-2692, 2010.

[19] S. Ferrari, F. Bellocchio, V. Piuri and N. A. Borghese, "A Hierarchical RBF Online Learning Algorithm for Real-Time 3-D Scanner," *IEEE Trans. on Neural Networks*, vol. 21, issue 2, pp. 275-285, 2010.

[20] K. Meng, Z. Y. Dong, D. H. Wang and K. P. Wong, "A Self-Adaptive RBF Neural Network Classifier for Transformer Fault Analysis," *IEEE Trans. on Power Systems*, vol. 25, issue 3, pp. 1350-1360, Feb. 2010.

[21] S. Huang and K. K. Tan, "Fault Detection and Diagnosis Based on Modeling and Estimation Methods," *IEEE Trans. on Neural Networks*, vol. 20, issue 5, pp. 872-881, Apr. 2009.

[22] K. Hornik, M. Stinchcombe and H. White, "Multilayer Feedforward Networks Are Universal Approximators," *Neural Networks*, vol. 2, issue 5, pp. 359-366, 1989.

[23] Werbos P. J., "Back-propagation: Past and Future," *Proceeding of International*

Conference on Neural Networks, San Diego, CA, 1, 343-354, 1988.

[24] K. Levenberg, "A method for the solution of certain problems in least squares," *Quarterly of Applied Machematics*, 5, pp. 164-168, 1944.

[25] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM J. Appl. Math.*, vol. 11, no. 2, pp. 431-441, Jun. 1963.

[26] B. M. Wilamowski, H. Yu and N. Cotton, "Neuron by Neuron Algorithm," *Industrial Electronics Handbook*, vol. 5 – INTELLIGENT SYSTEMS, 2nd Edition, 2010, chapter 13, pp. 13-1 to 13-24, CRC Press.

[27] B. M. Wilamowski, N. J. Cotton, O. Kaynak, G. Dundar, "Computing Gradient Vector and Jacobian Matrix in Arbitrarily Connected Neural Networks," *IEEE Trans. on Industrial Electronics*, vol. 55, no. 10, pp. 3784-3790, Oct. 2008.

[28] B. M. Wilamowski, "Neural Network Architectures and Learning Algorithms: How Not to Be Frustrated with Neural Networks," *IEEE Ind. Electron. Mag.*, vol. 3, no. 4, pp. 56-63, 2009.

[29] T. Takagi and M. Sugeno, "Fuzzy Identification of Systems and Its Application to Modeling and Control," *IEEE Transactions on System, Man, Cybernetics*, Vol. 15, No. 1, pp. 116-132, 1985.

[30] T. T. Xie, H. Yu and B. M. Wilamowski, "Neuro-fuzzy System," *Industrial Electronics Handbook*, vol. 5 – INTELLIGENT SYSTEMS, 2nd Edition, 2010, chapter 20, pp. 20-1 to 20-9, CRC Press.

[31] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag: New York, 1995.

[32] C. Saunders, A. Gammerman and V. Vovk, "Ridge Regression Learning Algorithm in Dual Variables," *Proceedings of the 15th International Conference on Machine Learning*, ICML-98, Madison-Wisconsin, 1998.

[33] T. Kohonen, "Self-Organized Formation of Topologically Correct Feature Maps," *Biological Cybernetics*, vol. 43, pp. 59- 69, 1982.

[34] I. T. Jolliffe, *Principal Component Analysis*. Series in Statistics. Springer; 2nd edition, Oct. 1, 2002.

[35] Q. K. Pan, M. F. Tasgetiren and Y. C. Liang, "A Discrete Particle Swarm Optimization Algorithm for the No-Wait Flowshop Scheduling Problem," *Computer & Operations Research*, vol. 35, issue 9, pp. 2807-2839, Sep. 2008.

[36] M. Dorigo, V. Maniezzo and A. Colorni, "Ant System: Optimization by a Colony of Cooperating Agents," *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, vol. 26,

issue 1, pp. 29-41, Feb. 1996.

[37] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, Reading, MA, 1989.

[38] S. Wu and T. W. S. Chow, "Induction Machine Fault Detection Using SOM-Based RBF Neural Networks," *IEEE Trans. on Industrial Electronics*, vol. 51, no. 1, pp. 183-194, Feb. 2004.

[39] C. K. Goh, E. J. Teoh and K. C. Tan, "Hybrid Multiobjective Evolutionary Design for Artificial Neural Networks," *IEEE Trans. on Neural Networks*, vol. 19, no. 9, pp. 1531-1548, Sept 2008.

[40] Y. Song, Z. Q. Chen and Z. Z. Yuan, "New Chaotic PSO-Based Neural Network Predictive Control for Nonlinear Process," *IEEE Trans. on Neural Networks*, vol. 18, no. 2, pp. 595-601, Feb 2007.

[41] T. T. Xie, H. Yu and B. M. Wilamowski, "Replacing Fuzzy Systems with Neural Networks," in *Proc. 3nd IEEE Human System Interaction Conf. HSI 2010*, Rzeszow, Poland, May 13-15, 2010, pp. 189-193.

[42] M. Sugeno and G. T. Kang, "Structure Identification of Fuzzy Model," *Fuzzy Sets and Systems*, Vol. 28, No. 1, pp. 15-33, 1988.

[43] A. Malinowski and H. Yu, "Comparison of Embedded System Design for Industrial Applications," *IEEE Trans. on Industrial Informactics*, vol. 7, issue 2, pp. 244-254, May 2011.

[44] R. J. Wai, J. D. Lee and K. L. Chuang, "Real-Time PID Control Strategy for Maglev Transportation System via Particle Swarm Optimization," *IEEE Trans. on Industrial Electronics*, vol. 58, no. 2, pp. 629-646, Jan. 2011.

[45] M. A. S. K. Khan and M. A. Rahman, "Implementation of a Wavelet-Based MRPID Controller for Benchmark Thermal System," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 12, pp. 4160-4169, Nov. 2010.

[46] Y. Z. Li and K. M. Lee, "Thermohydraulic Dynamics and Fuzzy Coordination Control of A Microchannel Cooling Network for Space Electronics," *IEEE Trans. on Industrial Electronics*, vol. 58, no. 2, pp. 700-708, Feb. 2011.

[47] R. Masuoka, N. Watanabe, A. Kawamura, Y. Owada and K. Asakawa, "Neuraofuzzy system-Fuzzy inference using a structured neural network," *Proceedings of the International Conference on FuzzyLogic&Neural Networks*, Hzuka, Japan, pp.173-177, July20-24,1990.

[48] LIBSVM link: http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[49] H. Yu and B. M. Wilamowski, "Efficient and Reliable Training of Neural Networks," *IEEE Human System Interaction Conference, HSI 2009,* Catania. Italy, May 21-23, 2009, pp. 109-115.

[50] H. Yu and B. M. Wilamowski, "C++ Implementation of Neural Networks Trainer," 13th *IEEE Intelligent Engineering Systems Conference, INES 2009*, Barbados, April 16-18, 2009, pp. 237-242.

[51] N. Pham, H. Yu and B. M. Wilamowski, "Neural Network Trainer through Computer Networks," 24th *IEEE International Conference on Advanced Information Networking and Applications, AINA2010*, Perth, Australia, April 20-23, 2010, pp. 1203-1209.

[52] H. Yu and B. M. Wilamowski, "Fast and efficient and training of neural networks," in *Proc. 3nd IEEE Human System Interaction Conf. HSI 2010*, Rzeszow, Poland, May 13-15, 2010, pp. 175-181.

[53] H. Yu and B. M. Wilamowski, "Neural Network Training with Second Order Algorithms," monograph by Springer on Human-Computer Systems Interaction. Background and Applications, 31st October, 2010. (Accepted)

[54] B. M. Wilamowski, "Human factor and computational intelligence limitations in resilient control systems" *ISRCS-10, 3-rd International Symposium on Resilient Control Systems,* Idaho Falls, Idaho, August 10-12, 2010, pp. 5-11.

[55] W. S. McCulloch and W. Pitts, "A Logical Calculus of the Idea Immanent in Nerveous Activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.

[56] D. O. Hebb, *The Organization of Behavior*. John Wiley & Sons, New York, 1949.

[57] A. M. Uttley, "A Theory of the Mechanism of Learning Based on the Computation of Conditional Probabilities," *Proceedings of the First International Conference on Cybernetics, Namur, Gauthier-Villars*, Paris, France, 1956.

[58] F. Rosenblatt, "The perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, vol. 65, pp. 386-408, 1959.

[59] B. Widrow and M. E. Hoff, Jr., "Adaptive Switching Circuits," *IRE WESCON Convention Record*, pp. 96-104, 1960.

[60] M. Minsky and S. Papert, *Perceptrons*. Oxford, England: M. I. T. Press, 1969.

[61] D. E. Rumelhart, G. E. Hinton and R. J. Wiliams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533-536, 1986 MA.

[62] S. I. Gallant, "Perceptron-based learning algorithms," *IEEE Trans. on Neural Networks*, vol. 1, no. 2, pp. 179-191, Feb 1990.

[63] K. S. Narendra, S. Mukhopadhyay, "Associative learning in random environments using neural networks ," *IEEE Trans. on Neural Networks*, vol. 2, no. 1, pp. 20-31, Jan 1991.

[64] R. C. Frye, E.A. Rietman, C.C. Wong, "Back-propagation learning and nonidealities in analog neural network hardware," *IEEE Trans. on Neural Networks*, vol. 2, no. 1, pp. 110-117, Jan 1991.

[65] J. N. Hwang, J.J. Choi, S. Oh, R.J. Marks, "Query-based learning applied to partially trained multilayer perceptrons," *IEEE Trans. on Neural Networks*, vol. 2, no. 1, pp. 131-136, Jan 1991.

[66] P. A. Shoemaker, "A note on least-squares learning procedures and classification by neural network models," *IEEE Trans. on Neural Networks*, vol. 2, no. 1, pp. 158-160, Jan 1991.

[67] R. Batruni, "A multilayer neural network with piecewise-linear structure and back-propagation learning," *IEEE Trans. on Neural Networks*, vol. 2, no. 3, pp. 395-403, March 1991.

[68] M. Riedmiller, H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proc. International Conference on Neural Networks*, San Francisco, CA, 1993, pp. 586-591.

[69] R. B. Allen, J. Alspector, "Learning of stable states in stochastic asymmetric networks," *IEEE Trans. on Neural Networks*, vol. 1, no. 2, pp. 233-238, Feb 1990.

[70] T. M. Martinetz, H. J. Ritter and K. J. Schulten, "Three-dimensional neural net for learning visuomotor coordination of a robot arm," *IEEE Trans. on Neural Networks*, vol. 1, no. 1, pp. 131-136, Jan 1990.

[71] Image link: http://koaboi.wordpress.com/category/uncategorized/

[72] M. E. Hohil, D. Liu, and S. H. Smith, "Solving the N-bit parity problem using neural networks," *Neural Networks*, vol. 12, pp1321-1323, 1999.

[73] B. M. Wilamowski, D. Hunter, A. Malinowski, "Solving parity-N problems with feedforward neural networks," *Proc. 2003 IEEE IJCNN*, 2546-2551, IEEE Press, 2003.

[74] B. M. Wilamowski, H. Yu and K. T. Chung, "Parity-N problems as a vehicle to compare efficiency of neural network architectures," *Industrial Electronics Handbook*, vol. 5 – INTELLIGENT SYSTEMS, 2nd Edition, 2010, chapter 10, pp. 10-1 to 10-8, CRC Press.

[75] M. R. Osborne, "Fisher's method of scoring," Internat. Statist. Rev., 86 (1992), pp. 271-286.

[76] H. Yu and B. M. Wilamowski, "Levenberg Marquardt Training," *Industrial Electronics Handbook*, vol. 5 – INTELLIGENT SYSTEMS, 2nd Edition, 2010, chapter 12, pp. 12-1 to 12-16, CRC Press.

[77] J. X. Peng, Kang Li, G.W. Irwin, "A New Jacobian Matrix for Optimal Learning of Single-Layer Neural Networks," *IEEE Trans. on Neural Networks*, vol. 19, no. 1, pp. 119-129, Jan 2008.

[78] B. M. Wilamowski and H. Yu, "Neural Network Learning without Backpropagation," *IEEE Trans. on Neural Networks*, vol. 21, no.11, Nov. 2010.

[79] B. M. Wilamowski and H. Yu, "Improved Computation for Levenberg Marquardt Training," *IEEE Trans. on Neural Networks*, vol. 21, no. 6, pp. 930-937, June 2010.

[80] M. T. Hagan, M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Trans. on Neural Networks*, vol. 5, no. 6, pp. 989-993, Nov. 1994.

[81] H. N. Robert, "Theory of the Back Propagation Neural Network," *Proc. 1989 IEEE IJCNN*, 1593-1605, IEEE Press, New York, 1989.

[82] H. B. Demuth, M. Beale, "Neural Network Toolbox: for use with MATLAB," *Mathworks Natick*, MA, USA, 2000.

[83] L. J. Cao, S. S. Keerthi, Chong-Jin Ong, J. Q. Zhang, U. Periyathamby, Xiu Ju Fu, H. P. Lee, "Parallel sequential minimal optimization for the training of support vector machines," *IEEE Trans. on Neural Networks*, vol. 17, no. 4, pp. 1039- 1049, April 2006.

[84] D. C. Lay, *Linear Algebra and its Applications*. Addison-Wesley Publishing Company, 3rd version, pp. 124, July, 2005.

[85] S. Wan, L.E. Banta, "Parameter Incremental Learning Algorithm for Neural Networks," *IEEE Trans. on Neural Networks*, vol. 17, no. 6, pp. 1424-1438, June 2006.

[86] S. E. Fahlman, C. Lebiere, "The cascade-correction learning architecture," in *D. S. Touretzky (ed.), Advances in Neural Information Processing Systems 2*, Morgan Kaufmann, 1990.

[87] B. K. Bose, "Neural Network Applications in Power Electronics and Motor Drives – An Introduction and Perspective," *IEEE Trans. on Industrial Electronics*, vol. 54, no. 1, pp. 14-33, Feb. 2007.

[88] J. A. Farrell, M. M. Polycarpou, "Adaptive Approximation Based Control: Unifying Neural, Fuzzy and Traditional Adaptive Approximation Approaches [Book review]," *IEEE Trans. on Neural Networks*, vol. 19, no. 4, pp. 731-732, April 2008.

[89] W. Qiao, R. G. Harley, G. K. Venayagamoorthy, "Fault-Tolerant Indirect Adaptive Neurocontrol for a Static Synchronous Series Compensator in a Power Network With Missing Sensor Measurements," *IEEE Trans. on Neural Networks*, vol. 19, no. 7, pp. 1179-1195, July 2008.

[90] J. Y. Goulermas, A. H. Findlow, C. J. Nester, P. Liatsis, X.-J. Zeng, L. P. J. Kenney, P. Tresadern, S. B. Thies, D. Howard, "An Instance-Based Algorithm With Auxiliary Similarity Information for the Estimation of Gait Kinematics From Wearable Sensors," *IEEE Trans. on Neural Networks*, vol. 19, no. 9, pp. 1574-1582, Sept 2008.

[91] N. J. Cotton, and Bogdan M. Wilamowski, "Compensation of Nonlinearities Using Neural Networks Implemented on Inexpensive Microcontrollers" *IEEE Trans. on Industrial Electronics*, vol. 58, No 3, pp. 733-740, March 2011.

[92] N. J. Cotton and Bogdan M. Wilamowski "Compensation of Sensors Nonlinearity with Neural Networks", *24th IEEE International Conference on Advanced Information Networking and Applications 2010*, pp. 1210-1217, April 2010.

[93] B. M. Wilamowski, "Challenges in Applications of Computational Intelligence in Industrials Electronics," (keynote) *ISIE' 10 IEEE International Symposium on Industrial Electronics*, Bari, Italy, July 5-7, 2010, pp. 15-22.

[94] F. H. C. Tivive and A. Bouzerdoum, "Efficient training algorithms for a class of shunting inhibitory convolutional neural networks," *IEEE Trans. on Neural Networks*, vol. 16, no. 3, pp. 541-556, March 2005.

[95] V. V. Phansalkar, P. S. Sastry, "Analysis of the back-propagation algorithm with momentum," *IEEE Trans. on Neural Networks*, vol. 5, no. 3, pp. 505-506, March 1994.

[96] B. M. Wilamowski, N. Cotton, J. Hewlett, O. Kaynak, "Neural network trainer with second order learning algorithms," *Proc. International Conference on Intelligent Engineering Systems*, June 29 2007-July 1 2007, pp. 127-132.

[97] B. M. Wilamowski, "Modified EBP algorithm with instant training of the hidden layer," *Proc. in 23rd International Conference on Industrial Electronics, Control, and Instrumentation*, vol. 3, pp.1098-1101, 1997.

[98] H. Yu, T. T. Xie, Stanislaw Paszczynski and B. M. Wilamowski, "Advantages of Radial Basis Function Networks for Dynamic System Design," *IEEE Trans. on Industrial Electronics* (Accepted)

[99] H. Yu, T. T. Xie, M. Hamilton and B. M. Wilamowski, "Comparison of Different Neural Network Architectures for Digit Image Recognition," in *Proc. 3nd IEEE Human System Interaction Conf. HSI 2011*, Yokohama, Japan, pp. 98-103, May 19-21, 2011.

[100] T. T. Xie, H. Yu and B. M. Wilamowski, "Comparison of Traditional Neural Networks and Radial Basis Function Networks," in *Proc. 20th IEEE International Symposium on Industrial Electronics*, ISIE2011, Gdansk, Poland, 27-30 June 2011 (Accepted)

[101] H. Yu, T. T. Xie and B. M. Wilamowski, "Error Correction – A Robust Learning Algorithm for Designing Compact Radial Basis Function Networks," *IEEE Trans. on Neural Networks* (Major Revision)

[102] T. T. Xie, H. Yu, B. M. Wilamowski and J. Hewllet, "Fast and Efficient Second Order Method for Training Radial Basis Function Networks," *IEEE Trans. on Neural Networks* (Major Revision)