Neural Network Training with Second Order Algorithms

H. Yu and B.M. Wilamowski

Department of Electrical and Computer Engineering,
Auburn University, Auburn, AL, USA
hzy0004@auburn.edu, wilam@ieee.org

Abstract. Second order algorithms are very efficient for neural network training because of their fast convergence. In traditional Implementations of second order algorithms [Hagan and Menhaj 1994], Jacobian matrix is calculated and stored, which may cause memory limitation problems when training large-sized patterns. In this paper, the proposed computation is introduced to solve the memory limitation problem in second order algorithms. The proposed method calculates gradient vector and Hessian matrix directly, without Jacobian matrix storage and multiplication. Memory cost for training is significantly reduced by replacing matrix operations with vector operations. At the same time, training speed is also improved due to the memory reduction. The proposed implementation of second order algorithms can be applied to train basically an unlimited number of patterns.

1 Introduction

As an efficient way of modeling the linear/nonlinear relationships between stimulus and responses, artificial neural networks are broadly used in industries, such as nonlinear control, data classification and system diagnosis.

The error back propagation (EBP) algorithm [Rumelhart et al. 1986] dispersed the dark clouds on the field of artificial neural networks and could be regarded as one of the most significant breakthroughs in neural network training. Still, EBP algorithm is widely used today; however, it is also known as an inefficient algorithm because of its slow convergence. Many improvements have been made to overcome the disadvantages of EBP algorithm and some of them, such as momentum and RPROP algorithm, work relatively well. But as long as the first order algorithms are used, improvements are not dramatic.

Second order algorithms, such as Newton algorithm and Levenberg Marquardt (LM) algorithm, use Hessian matrix to perform better estimations on both step sizes and directions, so that they can converge much faster than first order algorithms. By combining the training speed of Newton algorithm and the stability of EBP algorithm, LM algorithm is regarded as one of the most efficient algorithms for training small and medium sized patterns.

Z.S. Hippe et al. (Eds.): Human – Computer Systems Interaction, AISC 99, Part II, pp. 463–476.
springerlink.com

Table 1 shows the training statistic results of two-spiral problem using both EBP algorithm and LM algorithm. In both cases, fully connected cascade (FCC) networks were used for training and the desired sum square error was 0.01. For EBP algorithm, the learning constant was 0.005 (largest possible avoiding oscillation), momentum was 0.5 and iteration limit was 1,000,000; for LM algorithm, the maximum number of iteration was 1,000.

One may notice that EBP algorithm not only requires much more time than LM algorithm, 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 LM algorithm can solve it in only 8 neurons.

Table 1 Training results of two-spiral problem

Neurons	*Success Rate*		*Average Iteration*		*Average Time (s)*	
	EBP	LM	EBP	LM	EBP	LM
8	0%	13%	/	287.7	/	0.88
9	0%	24%	/	261.4	/	0.98
10	0%	40%	/	243.9	/	1.57
11	0%	69%	/	231.8	/	1.62
12	63%	80%	410,254	175.1	633.91	1.70
13	85%	89%	335,531	159.7	620.30	2.09
14	92%	92%	266,237	137.3	605.32	2.40
15	96%	96%	216,064	127.7	601.08	2.89
16	98%	99%	194,041	112.0	585.74	3.82

Even having such a powerful training ability, LM algorithm is not welcomed by engineers because of its complex computation and several limitations:

1. Network architecture limitation

The traditional implementation of LM algorithm by Hagan and Menhaj in their paper was developed only for multilayer perceptron (MLP) neural networks. Therefore, much more powerful neural networks [Hohil et al. 1999; Wilamowski 2009], such as fully connected cascade (FCC) or bridged multilayer perceptron (BMLP) architectures cannot be trained.

2. Network size limitation

The LM algorithm requires the inversion of Hessian matrix (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.

3. Memory limitation

LM algorithm cannot be used for the problems with many training patterns because the Jacobian matrix becomes prohibitively too large.

Fortunately, the network architecture limitation was solved by recently developed neuron-by-neuron (NBN) computation in papers [Wilamowski et al. 2008;

Wilamowski et al. 2010]. The NBN algorithm can be applied to train arbitrarily connected neural networks.

The network size limitation still remains unsolved, so that the LM algorithm can be used only for small and medium size neural networks.

In this paper, the memory limitation problem of the traditional LM algorithm is addressed and the proposed method of computation is going to solve this problem by removing Jacobian matrix storage and multiplication. In this case, second order algorithms can be applied to train very large-sized patterns [Wilamowski and Yu 2010].

The paper is organized as follows: Section 2 introduces the computational fundamentals of LM algorithm and addresses the memory limitation problem. Section 3 describes the improved computation for both quasi Hessian matrix and gradient vector in details. Section 4 implements the proposed computation on a simple parity-3 problem. Section 5 gives some experimental results on memory and training speed comparison between traditional Hagan and Menhaj LM algorithm and the improved LM algorithm.

2 Computational Fundamentals

Before the derivation, let us introduce some indices which will be used in the paper:

- p is the index of patterns, from 1 to np, where np is the number of training patterns;
- m is the index of outputs, from 1 to no, where no is the number of outputs;
- i and j are the indices of weights, from 1 to nw, where nw is the number of weights.
- k is the index of iterations and n is the index of neurons.

Other indices will be explained in related places.

The sum square error (SSE) E is defined to evaluate the training process. For all patterns and outputs, it is calculated as:

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

where: e_{pm} is the error at output m when training pattern p, defined as

$$e_{pm} = o_{pm} - d_{pm} \tag{2}$$

where: d_{pm} and o_{pm} are desired output and actual output, respectively, at output m for training pattern p.

The update rule of LM algorithm is:

$$\Delta \boldsymbol{w}_k = (\boldsymbol{H}_k + \mu \boldsymbol{I})^{-1} \boldsymbol{g}_k \tag{3}$$

where: μ is the combination coefficient, $\boldsymbol{I}$ is the identity matrix, $\boldsymbol{g}$ is the gradient vector and $\boldsymbol{H}$ is the Hessian matrix.

The gradient vector $\boldsymbol{g}$ and Hessian matrix $\boldsymbol{H}$ are de fined as:

$$\boldsymbol{g} = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \cdots \\ \frac{\partial E}{\partial w_{nw}} \end{bmatrix} \tag{4}$$

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

As one may notice, in order to perform the update rule (3), second order derivatives of E in (5) has to be calculated, which makes the computation very complex.

In the Hagan and Menhaj implementation of LM algorithm, Jacobian matrix $\boldsymbol{J}$ was introduced to avoid the calculation of second order derivatives. The Jacobian matrix has the format:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_{nw}} \\ \frac{\partial e_{12}}{\partial w_1} & \frac{\partial e_{12}}{\partial w_2} & \cdots & \frac{\partial e_{12}}{\partial w_{nw}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1no}}{\partial w_1} & \frac{\partial e_{1no}}{\partial w_2} & \cdots & \frac{\partial e_{1no}}{\partial w_{nw}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{np1}}{\partial w_1} & \frac{\partial e_{np1}}{\partial w_2} & \cdots & \frac{\partial e_{np1}}{\partial w_{nw}} \\ \frac{\partial e_{np2}}{\partial w_1} & \frac{\partial e_{np2}}{\partial w_2} & \cdots & \frac{\partial e_{np2}}{\partial w_{nw}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{npno}}{\partial w_1} & \frac{\partial e_{npno}}{\partial w_2} & \cdots & \frac{\partial e_{npno}}{\partial w_{nw}} \end{bmatrix} \tag{6}$$

By combining (1) and (4), the elements of gradient vector can be calculated as:

$$\frac{\partial E}{\partial w_i} = \sum_{p=1}^{np} \sum_{m=1}^{no} \left(\frac{\partial e_{pm}}{\partial w_i} e_{pm} \right) \tag{7}$$

So the relationship between gradient vector and Jacobian matrix can be presented by

$$\boldsymbol{g} = \boldsymbol{J}^T \boldsymbol{e} \tag{8}$$

By combining (1) and (5), the elements of Hessian matrix can be calculated as

$$\frac{\partial^2 E}{\partial w_i \partial w_j} = \sum_{p=1}^{np} \sum_{m=1}^{no} \left(\frac{\partial e_{pm}}{\partial w_i} \frac{\partial e_{pm}}{\partial w_i} + \frac{\partial^2 e_{pm}}{\partial w_i \partial w_j} e_{pm} \right) \approx \sum_{p=1}^{np} \sum_{m=1}^{no} \frac{\partial e_{pm}}{\partial w_i} \frac{\partial e_{pm}}{\partial w_i} \tag{9}$$

The relationship between Hessian matrix and Jacobian matrix can be described by

$$\boldsymbol{H} \approx \boldsymbol{J}^T \boldsymbol{J} = \boldsymbol{Q} \tag{10}$$

where: matrix $\boldsymbol{Q}$ is the approximated Hessian matrix, called quasi Hessian matrix.

By integrating equations (3) and (8), (10), the implementation of LM update rule becomes

$$\Delta \boldsymbol{w}_k = (\boldsymbol{J}_k^T \boldsymbol{J}_k + \mu \boldsymbol{I})^{-1} \boldsymbol{J}_k^T \boldsymbol{e}_k \tag{11}$$

where: $\boldsymbol{e}$ is the error vector.

Equation (11) is used as the traditional implementation of LM algorithm. Jacobian matrix $\boldsymbol{J}$ has to be calculated and stored at first; then matrix multiplications (8) and (10) are performed for further weight updating. According to the definition of Jacobian matrix $\boldsymbol{J}$ in (6), there are $np \times no \times nw$ elements needed to be stored. It may work smoothly for problems with small and medium sized training patterns; however, for large-sized patterns, the memory limitation problem could be triggered. For example, the MNIST pattern recognition problem [Cao et al. 2006] consists of 60,000 training patterns, 784 inputs and 10 outputs. Using the simplest possible neural network (one neuron per each output), the memory cost for entire Jacobian matrix storage is nearly 35 gigabytes which would be quite an expensive memory cost for real programming.

3 Improved Computation

The key issue leading to the memory limitation in traditional computation is that the entire Jacobian matrix has to be stored for further matrix multiplication. One may think that if both gradient vector and Hessian matrix could be obtained directly, without Jacobian matrix multiplication, there is no need to store all the elements of Jacobian matrix so that the problem can be solved.

3.1 Matrix Algebra for Jacobian Matrix Elimination

There are two ways of matrix multiplication. If the row of the first matrix is multiplied by the column of the second matrix, then a scalar is obtained, as shown in Fig. 1a. If the column of the first matrix is multiplied by the row of the second matrix, then a partial matrix $\boldsymbol{q}$ is obtained, as shown in Fig. 1b. The number of scalars is $nw \times nw$, while the number of partial matrices $\boldsymbol{q}$, which later have to be summed, is $np \times no$.

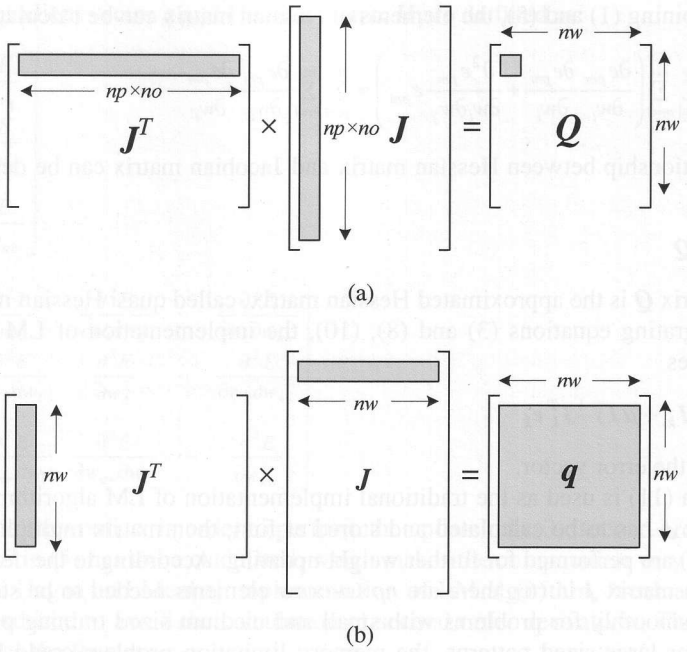

Fig. 1 Two ways of matrix multiplication: (a) row-column multiplication results in a scalar; (b) column-row multiplication results in a partial matrix ***q***

When $\boldsymbol{J}^T$ is multiplied by $\boldsymbol{J}$ using the routine shown in Fig. 1b, partial matrices $\boldsymbol{q}$ (size: $nw \times nw$) need to be calculated $np \times no$ times, then all of the $np \times no$ matrices $\boldsymbol{q}$ must be summed together. The routine of Fig. 1b seems complicated; therefore, almost all matrix multiplication processes use the routine of Fig. 1a, where only one element of the resulted matrix is calculated and stored each time.

Even the routine of Fig. 1b seems to be more complicated than the routine in Fig. 1a; after detailed analysis (Table 2), one may conclude that the computation cost for both methods of matrix multiplication are basically the same.

Table 2 Computation analysis between the two methods of matrix multiplication

Multiplication Methods	*Addition*	*Multiplication*
Row-column (Fig. 1a)	$(np \times no) \times nw \times nw$	$(np \times no) \times nw \times nw$
Column-row (Fig. 1b)	$nw \times nw \times (np \times no)$	$nw \times nw \times (np \times no)$

In the specific case of neural network training, only one row of Jacobian matrix $\boldsymbol{J}$ (column of $\boldsymbol{J}^T$) is known for each training pattern, and there is no relationship among training patterns. So if the routine in Fig. 1b is used, then the process of creation of quasi Hessian matrix can be started sooner without necessity of computing and storing the entire Jacobian matrix for all patterns and all outputs.

Table 3 roughly estimates the memory cost in two multiplication methods separately.

Table 3 Memory cost analysis between two methods of matrix multiplication

Multiplication Methods	Elements for storage
Row-column (Fig. 1a)	(np × no) × nw + nw × nw + nw
Column-row (Fig. 1b)	nw × nw + nw
Difference	(np × no) × nw

Notice that the column-row multiplication (Fig. 1b) can save a lot of memory.

3.2 Improved Gradient Vector Computation

Let us introduce gradient sub vector $\boldsymbol{\eta}_{pm}$ (size: $nw \times 1$):

$$\boldsymbol{\eta}_{pm} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} e_{pm} \\ \frac{\partial e_{pm}}{\partial w_2} e_{pm} \\ \cdots \\ \frac{\partial e_{pm}}{\partial w_N} e_{pm} \end{bmatrix} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} \\ \frac{\partial e_{pm}}{\partial w_2} \\ \cdots \\ \frac{\partial e_{pm}}{\partial w_N} \end{bmatrix} \times e_{pm} \tag{12}$$

By combining (7), (8) and (12), gradient vector g can be calculated as the sum of gradient sub vectors $\boldsymbol{\eta}_{pm}$

$$\boldsymbol{g} = \sum_{p=1}^{np} \sum_{m=1}^{no} \boldsymbol{\eta}_{pm} \tag{13}$$

By introducing vector $\boldsymbol{j}_{pm}$ (size: $1 \times nw$)

$$\boldsymbol{j}_{pm} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} & \frac{\partial e_{pm}}{\partial w_2} & \cdots & \frac{\partial e_{pm}}{\partial w_{nw}} \end{bmatrix} \tag{14}$$

sub vectors $\boldsymbol{\eta}_{pm}$ in (12) can be also written in the vector form

$$\boldsymbol{\eta}_{pm} = \boldsymbol{j}_{pm}^T e_{pm} \tag{15}$$

One may notice that for the computation of sub vector $\boldsymbol{\eta}_{pm}$, only nw elements of vector $\boldsymbol{j}_{pm}$ need to be calculated and stored. All the sub vectors can be calculated for each pattern p and output m separately, and summed together, so as to obtain the gradient vector $\boldsymbol{g}$.

Considering the independence among all training patterns and outputs, there is no need to store all the sub vector $\boldsymbol{\eta}_{pm}$. Each sub vector can be summed to a temporary vector after its computation. Therefore, during the direct computation of gradient vector $\boldsymbol{g}$ using (13), only memory for $\boldsymbol{j}_{pm}$ (nw elements) and e_{pm} (1 element) is required, instead of the whole Jacobian matrix ($np \times no \times nw$ elements) and error vector ($np \times no$ elements).

3.3 Improved Quasi Hessian Matrix Computation

Quasi Hessian sub matrix q_{pm} (size: $nw \times nw$) is introduced as

$$q_{pm} = \begin{bmatrix} \left(\frac{\partial e_{pm}}{\partial w_1}\right)^2 & \frac{\partial e_{pm}}{\partial w_1}\frac{\partial e_{pm}}{\partial w_2} & \cdots & \frac{\partial e_{pm}}{\partial w_1}\frac{\partial e_{pm}}{\partial w_{nw}} \\ \frac{\partial e_{pm}}{\partial w_2}\frac{\partial e_{pm}}{\partial w_1} & \left(\frac{\partial e_{pm}}{\partial w_2}\right)^2 & \cdots & \frac{\partial e_{pm}}{\partial w_2}\frac{\partial e_{pm}}{\partial w_{nw}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{pm}}{\partial w_{nw}}\frac{\partial e_{pm}}{\partial w_1} & \frac{\partial e_{pm}}{\partial w_{nw}}\frac{\partial e_{pm}}{\partial w_2} & \cdots & \left(\frac{\partial e_{pm}}{\partial w_{nw}}\right)^2 \end{bmatrix} \tag{16}$$

By combining (9), (10) and (16), quasi Hessian matrix $\boldsymbol{Q}$ can be calculated as the sum of quasi Hessian sub matrix q_{pm}

$$\boldsymbol{Q} = \sum_{p=1}^{np} \sum_{m=1}^{no} q_{pm} \tag{17}$$

Using the same vector j_{pm} defined in (14), quasi Hessian sub matrix can be calculated as

$$q_{pm} = j_{pm}^T j_{pm} \tag{18}$$

Similarly, quasi Hessian sub matrix q_{pm} can be calculated for each pattern and output separately, and summed to a temporary matrix. Since the same vector j_{pm} is calculated during the gradient vector computation above, no extra memory is required.

With the improved computation, both gradient vector $\boldsymbol{g}$ and quasi Hessian matrix $\boldsymbol{Q}$ can be computed directly, without Jacobian matrix storage and multiplication. During this process, only a temporary vector j_{pm} with N elements needs to be stored; in other words, the memory cost for Jacobian matrix storage is reduced by $np \times no$ times. In the MINST problem mentioned in section 2, the memory cost for the storage of Jacobian elements could be reduced from more than 35 gigabytes to nearly 30.7 kilobytes.

From (16), one may also notice that all the sub matrix q_{pm} are symmetrical. With this property, only upper or lower triangular elements of those sub matrices need to be calculated. Therefore, during the improved quasi Hessian matrix $\boldsymbol{Q}$ computation, multiplication operations in (18) and sum operations in (17) can be both reduced by half approximately.

3.4 Simplified $\partial e_{pm}/\partial w_i$ Computation

For the improved computation of gradient vector $\boldsymbol{g}$ and quasi Hessian matrix $\boldsymbol{Q}$ above, the key point is to calculate vector j_{pm} (defined in (14)) for each training pattern and each output. This vector is equivalent of one row of Jacobian matrix $\boldsymbol{J}$.

By combining (2) and (14), the element of vector $\boldsymbol{j_{pm}}$ can be computed by

$$\frac{\partial e_{pm}}{\partial w_i} = \frac{\partial (o_{pm} - d_{pm})}{\partial w_i} = \frac{\partial o_{pm}}{\partial net_{pn}} \frac{\partial net_{pn}}{\partial w_i} \tag{19}$$

where: net_{pn} is the sum of weighted inputs at neuron n, calculated by

$$net_{pn} = \sum x_{pi} w_i \tag{20}$$

where: x_{pi} and w_i are the inputs and related weights respectively at neuron n.

Inserting (19) and (20) into (14), the vector jpm can be calculated by

$$\boldsymbol{j_{pm}} = \left[\frac{\partial o_{pm}}{\partial net_{p1}} [x_{p1_1} \quad \cdots \quad x_{p1_i} \quad \cdots] \quad \cdots \quad \frac{\partial o_{pm}}{\partial net_{pn}} [x_{pn_1} \quad \cdots \quad x_{pn_i} \quad \cdots] \quad \cdots \right] \tag{21}$$

where: x_{pni} is the i-th input of neuron n, when training pattern p.

Using the neuron by neuron (NBN) computation, in (21), x_{pni} can be calculated in the forward computation, while $\partial o_{pm}/\partial net_{pn}$ is 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 above, the memory cost for all those temporary parameters can be reduced by $np \times no$ times. All matrix operations are simplified to vector operations.

4 Implementation

For a better illustration of the improved computation, let us use the parity-3 problem as an example.Parity-3 problem has 8 patterns, each of which is made up of 3 inputs and 1 output, as shown in Fig. 2.

Pattern index	Inputs	Output
1	-1 -1 -1	-1
2	-1 -1 1	1
3	-1 1 -1	1
4	-1 1 1	-1
5	1 -1 -1	1
6	1 -1 1	-1
7	1 1 -1	-1
8	1 1 1	1

Fig. 2 Parity-3 problem: 8 patterns, 2 inputs and 1 output

The structure, 2 neurons in FCC network (Fig. 3), is used to train parity-3 patterns.

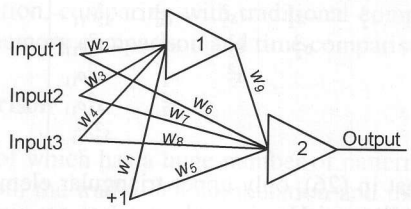

Fig. 3 Two neurons in fully connected cascade network

In Fig. 3, all weights are initialed by $\boldsymbol{w}=\{w_1,w_2,w_3,w_4,w_5,w_6,w_7,w_8,w_9\}$. Also, all elements in both gradient vector and quasi Hessian matrix are set to "0".

Applying the first training pattern (-1, -1, -1, -1), the forward computation is organized from inputs to output, as

1. $net_{11}=1\times w_1+(-1)\times w_2+(-1)\times w_3+(-1)\times w_4$
2. $o_{11}= f(net_{11})$, where f() is the activation function for neurons
3. $net_{12}=1\times w_5+(-1)\times w_6+(-1)\times w_7+(-1)\times w_8+o_{11}\times w_9$
4. $o_{12}=f(net_{12})$
5. $e_{11}=-1-o_{12}$

Then, the backward computation, from output to inputs, does the calculation of $\partial e_{11}/\partial net_{11}$ and $\partial e_{11}/\partial net_{12}$ in the following steps:

6. Using the results from steps 4) and 5), it could be obtained

$$\frac{\partial e_{11}}{\partial net_{12}}=\frac{\partial(-1-o_{12})}{\partial net_{12}}=-\frac{\partial f(net_{12})}{\partial net_{12}} \tag{22}$$

7. Using the results from steps 1), 2) and 3), and the chain-rule in differential, one can obtain that:

$$\frac{\partial e_{11}}{\partial net_{11}}=\frac{\partial(-1-o_{12})}{\partial net_{11}}=-\frac{\partial f(net_{12})}{\partial net_{12}}\frac{\partial net_{12}}{\partial o_{11}}\frac{\partial o_{11}}{\partial net_{11}}=-\frac{\partial f(net_{12})}{\partial net_{12}}\times w_9\times\frac{\partial f(net_{11})}{\partial net_{11}} \tag{23}$$

Using equation (21), the elements in $\boldsymbol{j_{11}}$ can be calculated as

$$\boldsymbol{j}_{11}=\left[\frac{\partial o_{11}}{\partial net_{11}}[1\ \ -1\ \ -1\ \ -1]\quad\frac{\partial o_{11}}{\partial net_{12}}[1\ \ -1\ \ -1\ \ -1\ \ o_{11}]\right] \tag{24}$$

By combining equations (15) and (24), the first sub vector $\boldsymbol{\eta_{11}}$ can be obtained as

$$\boldsymbol{\eta}_{11}=[s_1\ \ -s_1\ \ -s_1\ \ -s_1\ \ s_2\ \ -s_2\ \ -s_2\ \ -s_2\ \ s_2o_{11}]\times e_{11} \tag{25}$$

where: $s_1=\partial e_{11}/\partial net_{11}$ and $s_2=\partial e_{11}/\partial net_{12}$.

By combining equations (18) and (24), the first quasi Hessian sub matrix $\boldsymbol{q_{11}}$ can be calculated as

$$\boldsymbol{q}_{11}=\begin{bmatrix} s_1^2 & -s_1^2 & -s_1^2 & -s_1^2 & s_1s_2 & -s_1s_2 & -s_1s_2 & -s_1s_2 & s_1s_2o_{11} \\ & s_1^2 & s_1^2 & s_1^2 & -s_1s_2 & s_1s_2 & s_1s_2 & s_1s_2 & -s_1s_2o_{11} \\ & & s_1^2 & s_1^2 & -s_1s_2 & s_1s_2 & s_1s_2 & s_1s_2 & -s_1s_2o_{11} \\ & & & s_1^2 & -s_1s_2 & s_1s_2 & s_1s_2 & s_1s_2 & -s_1s_2o_{11} \\ & & & & s_2^2 & -s_2^2 & -s_2^2 & -s_2^2 & s_2^2o_{11} \\ & & & & & s_2^2 & s_2^2 & s_2^2 & -s_2^2o_{11} \\ & & & & & & s_2^2 & s_2^2 & -s_2^2o_{11} \\ & & & & & & & s_2^2 & -s_2^2o_{11} \\ & & & & & & & & s_2^2o_{11}^2 \end{bmatrix} \tag{26}$$

One may notice that in (26), only upper triangular elements of sub matrix q11 are calculated, since all quasi Hessian sub matrices are symmetrical (as analyzed in section 3.3). This further simplifies the computation.

So far, the first sub gradient vector η_{11} and quasi Hessian sub matrix q_{11} are calculated as equations (25) and (26), respectively. Then the last step for training the pattern (-1, -1, -1, -1) is to add the vector η_{11} and matrix q_{11} to gradient vector g and quasi Hessian matrix Q separately. After the sum operation, all memory costs in the computation, such as j_{11}, η_{11} and q_{11}, can be released.

```
% Initialization
Q=0;
 g=0
% Improved computation
for p=1:np              % Number of patterns
  % Forward computation
  ...
  for m=1:no            % Number of outputs
    % Backward computation
    ...
    calculate vector jpm;          % Eq. (21)
    calculate sub vector ηpm;      % Eq. (15)
    calculate sub matrix qpm;      % Eq. (18)
    g=g+ηpm;                       % Eq. (13)
    Q=Q+qpm;                       % Eq. (17)
  end;
end;
```

Fig. 4 Pseudo code of the improved computation

The computation above is only for training the first pattern of the parity-3 problem. For the other 7 patterns, the computation process is almost the same, except applying different input and output values. During the whole computation process, there is no Jacobian matrix storage and multiplication; only 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 *np* training patterns and *no* outputs, the improved computation can be organized as the pseudo code shown in Fig. 4.

5 Experimental Results

The experiments are designed to test the memory and training time efficiencies of the improved computation, comparing with traditional computation. They are divided into two parts, memory comparison and time comparison.

5.1 Memory Comparison

Three problems, each of which has a 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 in the tables below. The actual memory costs are measured by Windows Task Manager.

Table 4 Memory comparison for parity-14 and parity-16 problems

Problems	*Parity-14*	*Parity-16*
Patterns	16,384	65,536
Structures*	15 neurons	17 neurons
Jacobian matrix sizes	20.6Mb	106.3Mb
Weight vector sizes	1.3Kb	1.7Kb
Average iteration	99.2	166.4
Success Rate	13%	9%
Algorithms	*Actual memory cost*	
Traditional LM	87.6Mb	396.47Mb
Improved LM	11.8Mb	15.90Mb

*All neurons are in fully connected neural networks.

For the test results in Tables 4 and 5, it is clear that memory cost for training is significantly reduced in the improved computation. Notice that, in the MINST pattern recognition problem, higher memory efficiency can be obtained by the improved computation if the memory costs for training patterns storage are removed.

Table 5 Memory comparison for MINST pattern recognition problem

Problems	*MINST Problem*
Patterns	60,000
Structures	784=1 single layer network*
Jacobian matrix sizes	179.7Mb
Weight vector sizes	3.07Kb
Algorithms	*Actual memory cost*
Traditional LM	572.8Mb
Improved LM	202.8Mb

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

5.2 Time Comparison

Parity-9, parity-11 and parity-13 problems are trained to test the training time for both traditional and the improved computation, using LM algorithm. For all cases, fully connected cascade networks are used for testing. For each case, the initial weights and training parameters are exactly the same.

Table 6 Time comparison for parity-9, parity-11 and parity-13 problems

Problems	*Parity-9*	*Parity-11*	*Parity-13*
Patterns	512	2,048	8,192
Neurons	8	10	15
Weights	108	165	315
Average Iterations	35.1	58.1	88.2
Success Rate	38%	17%	21%
Algorithms	*Averaged training time (ms)*		
Traditional LM	2,226	73,563	2,868,344
Improved LM	1,078	19,990	331,531

From Table 6, one may notice that the improved computation can not only handle much larger problems, but it also computes much faster than the traditional one, especially for large-sized patterns training. The larger the pattern size is, the more time efficient the improved computation will be. As analyzed above, both the simplified quasi Hessian matrix computation and reduced memory contributes to the significantly improved training speed presented in Table 6.

From the comparisons above, 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.

6 Conclusion

In this paper, the improved computation is introduced to increase the training efficiency of Levenberg Marquardt algorithm. Instead of storage the entire Jacobian matrix for further computation, the proposed method uses only one row of Jacobian matrix each time to calculate both gradient vector and quasi Hessian matrix gradually. In this case, the corresponding memory requirement is decreased by $np \times no$ times approximately, where np is the number of training patterns and no is the number of outputs. The memory limitation problem in Levenberg Marquardt training is eliminated. Based on the proposed method, the computation process of quasi Hessian matrix is further simplified using its symmetrical property. Therefore, the training speed of the improved Levenberg Marquardt algorithm becomes much faster than the traditional one, by reducing both memory cost and multiplication operations in quasi Hessian matrix computation. With the experimental results presented in section 5, one can conclude that the improved computation is much more efficient than traditional computation, not only for memory requirement, but also training time.

The method was implemented in neural network trainer (NBN 2.10) [Yu and Wilamowski 2009; Yu et al. 2009], and the software can be downloaded from http://www.eng.auburn.edu/users/wilambm/nnt/

References

[Cao et al. 2006] Cao, L.J., Keerthi, S.S., Ong, C.J., Zhang, J.Q., Periyathamby, U., Fu, X.J., Lee, H.P.: Parallel sequential minimal optimization for the training of support vector machines. IEEE Trans. on Neural Networks 17(4), 1039–1049 (2006)

[Hagan and Menhaj 1994] Hagan, M.T., Menhaj, M.B.: Training feedforward networks with the Marquardt algorithm. IEEE Trans. on Neural Networks 5(6), 989–993 (1994)

[Hohil et al. 1999] Hohil, M.E., Liu, D., Smith, S.H.: Solving the N-bit parity problem using neural networks. Neural Networks 12, 1321–1323 (1999)

[Rumelhart et al. 1986] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature 323, 533–536 (1986)

[Wilamowski 2009] Wilamowski, B.M.: Neural network architectures and learning algorithms: How not to be frustrated with neural networks. IEEE Industrial Electronics Magazine 3(4), 56–63 (2009)

[Wilamowski et al. 2008] Wilamowski, B.M., Cotton, N.J., Kaynak, O., Dundar, G.: Computing gradient vector and jacobian matrix in arbitrarily connected neural networks. IEEE Trans. on Industrial Electronics 55(10), 3784–3790 (2008)

[Wilamowski et al. 2010] Yu, H., Wilamowski, B.M.: Neural network learning without backpropagation. IEEE Trans. on Neural Networks 21(11) (2010)

[Wilamowski and Yu 2010] Yu, H., Wilamowski, B.M.: Improved Computation for Levenberg Marquardt Training. IEEE Trans. on Neural Networks 21(6), 930–937 (2010)

[Yu and Wilamowski 2009] Yu, H., Wilamowski, B.M.: Efficient and reliable training of neural networks. In: Proc. 2nd IEEE Human System Interaction Conf. HSI 2009, Catania, Italy, pp. 109–115 (2009)

[Yu et al. 2009] Yu, H., Wilamowski, B.M.: C++ implementation of neural networks trainer. In: Proc. of 13th Int. Conf. on Intelligent Engineering Systems, INES 2009, Barbados (2009)