Improved Computation for Levenberg–Marquardt Training

Bogdan M. Wilamowski, *Fellow, IEEE*, and Hao Yu

***Abstract*—The improved computation presented in this paper 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.**

***Index Terms*—Levenberg–Marquardt (LM) algorithm, neural network training.**

I. Background

ARTIFICIAL NEURAL NETWORKS (ANNs) are widely used in industrial fields, such as nonlinear control [1], [5], system analysis and diagnosis [8], [14], and data classification [10], [18].

As a first-order algorithm, error backpropagation (EBP) algorithm [17], [19] is the most welcomed method in training neural networks, since it was invented in 1986. However, EBP algorithm is also well known for its low training efficiency. Using dynamic learning rates is a common way to increase the training speed of EBP algorithm [6], [13]. Also, momentum [16] can be introduced to speed up its convergence. But, even with those good suggestions, EBP algorithms (including improved versions) are still network and iteration expensive in practical applications.

Training speed is significantly increased by using second-order algorithms [3], [9], [22], such as Newton algorithm and Levenberg–Marquardt (LM) algorithm [12]. Comparing with the constant (or manually adjusted) learning rates in EBP algorithms, second-order algorithms can "naturally" estimate the learning rate in each direction of gradient using Hessian matrix. By combining EBP algorithm and Newton algorithm, LM algorithm is ranked as one of the most efficient training algorithms for small and median size patterns.

LM algorithm was implemented for neural network training in [7], but only for multilayer perceptron (MLP) architectures.

Manuscript received June 16, 2009; revised January 21, 2010; accepted February 18, 2010. Date of publication April 19, 2010; date of current version June 03, 2010.

The authors are with the Department of Electrical and Computer Engineering, Auburn University, Auburn, AL 36849-5201 USA (e-mail: wilam@ieee.org; hzy0004@auburn.edu).

Digital Object Identifier 10.1109/TNN.2010.2045657

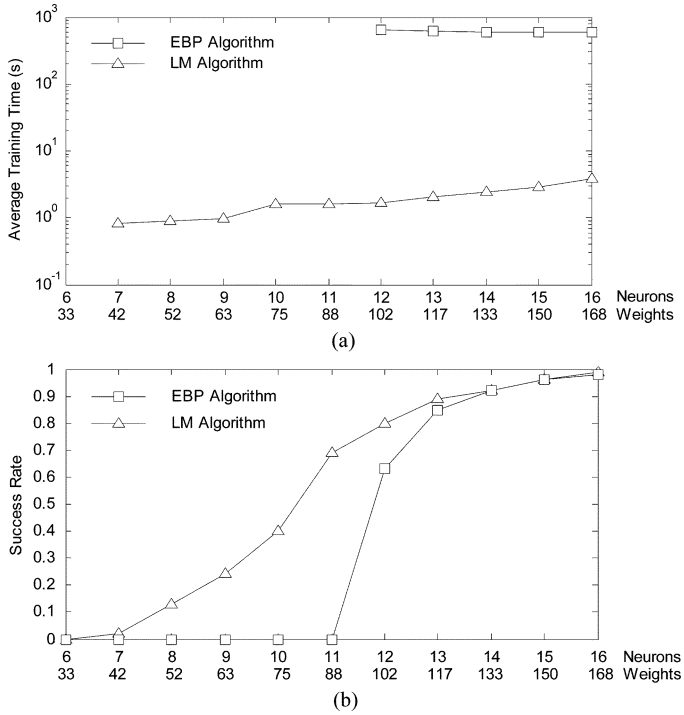


Fig. 1. Comparison between EBP algorithm and LM algorithm, for different number of neurons in fully connected cascade networks: (a) average training time; (b) success rate.

The best known example is popular neural network toolbox of MATLAB. Recently, LM algorithm was adapted for arbitrarily connected neural (ACN) networks [20], [21]. Those ACN networks can handle more complex problems with less number of neurons. LM algorithm implementations require calculation of Jacobian matrix, whose size is proportional to the number of training patterns.

Fig. 1 presents the training results the two-spiral problem [2], 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 1000 for LM algorithm. EBP algorithm not only requires much more time than LM algorithm [Fig. 1(a)], 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 seven neurons [Fig. 1(b)].

Fig. 2 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

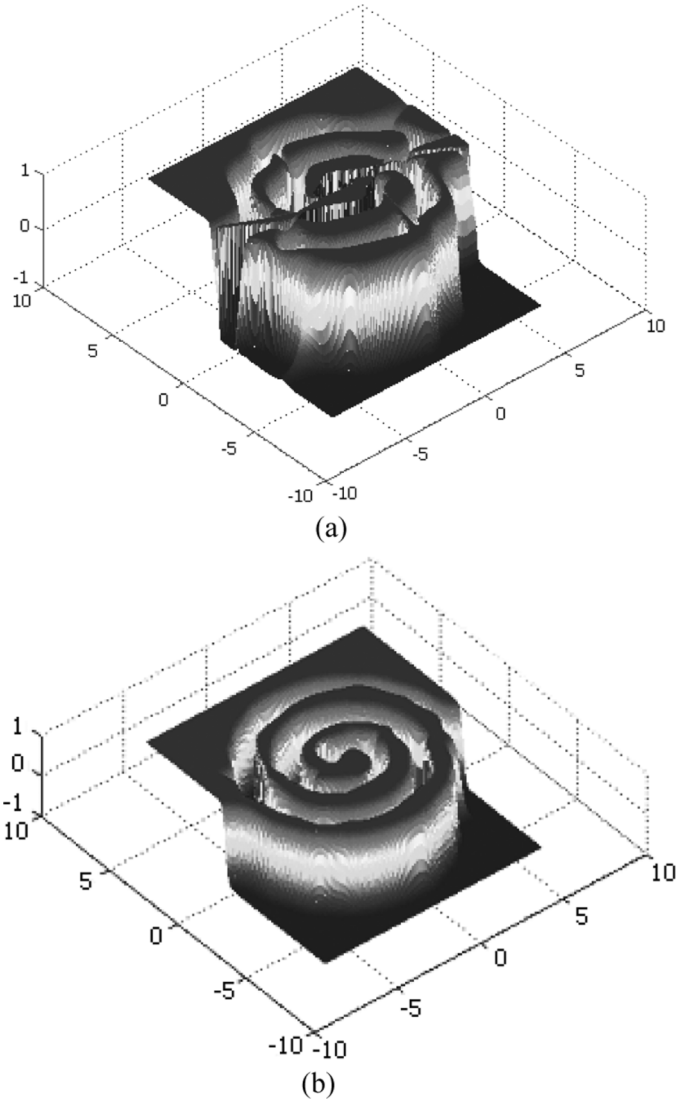


Fig. 2. Training results of the two-spiral problem with 16 neurons in fully connected cascade network. (a) EBP algorithm. (b) LM algorithm.

the same topology, LM algorithm is able to find better solutions than those found using EBP algorithm.

The comparison results above illustrate a significant advantage of LM algorithm training small and medium size patterns. However, for problems with large number of patterns, such as parity-16 problem (65 536 patterns), LM algorithm will have to use excessive memory for Jacobian matrix storage and multiplication.

In the proposed modification of LM algorithm, Jacobian matrix needs not to be stored and Jacobian matrix multiplication was replaced by vector operations. Therefore, the proposed algorithm can be used for problems with basically unlimited number of training patterns. Also, the proposed improvement accelerates training process.

In Section II, computational fundamentals of LM algorithm are introduced to address the memory problem. Section III describes the improved computation for both quasi-Hessian matrix and gradient vector in details. Section IV illustrates the improved computation on a simple problem. Section V gives some experimental results on memory and training time comparison between the traditional computation and the improved computation.

II. Computational Fundamentals

Derived from steepest descent method and Newton algorithm, the update rule of LM algorithm is [7]

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

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

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

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

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

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 (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 [4] consists of 60 000 training patterns, 784 inputs, and ten outputs. Using only the simplest possible neural network with ten neurons (one neuron per each output), the memory cost for the entire Jacobian matrix storage is nearly 35 GB. This huge memory requirement cannot be satisfied by any 32-bit Windows compliers, where there is a 3-GB limitation for single array storage. At this point, with traditional computation, one may conclude that LM algorithm cannot be used for problems with large number of patterns.

III. Improved Computation

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

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

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

The $N \times N$ Hessian matrix $\boldsymbol{H}$ is [15]

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_N} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 E}{\partial w_N \partial w_1} & \frac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \quad (5)$$

where N is the number of weights.

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

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

where i and j are weight indexes.

For LM algorithm, (6) is approximated as [7], [15]

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

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

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

$$\boldsymbol{H} \approx \boldsymbol{Q} = \boldsymbol{J}^T \boldsymbol{J}. \quad (8)$$

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

$$\boldsymbol{g} = \begin{bmatrix} \frac{\partial E}{\partial w_1} & \frac{\partial E}{\partial w_2} & \cdots & \frac{\partial E}{\partial w_N} \end{bmatrix}^T. \quad (9)$$

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

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

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

$$\boldsymbol{g} = \boldsymbol{J}^T \boldsymbol{e}. \quad (11)$$

Combining (8), (11), and (1), the update rule of LM algorithm can be rewritten as

$$\Delta \boldsymbol{w} = (\boldsymbol{Q} + \mu \boldsymbol{I})^{-1} \boldsymbol{g}. \quad (12)$$

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.

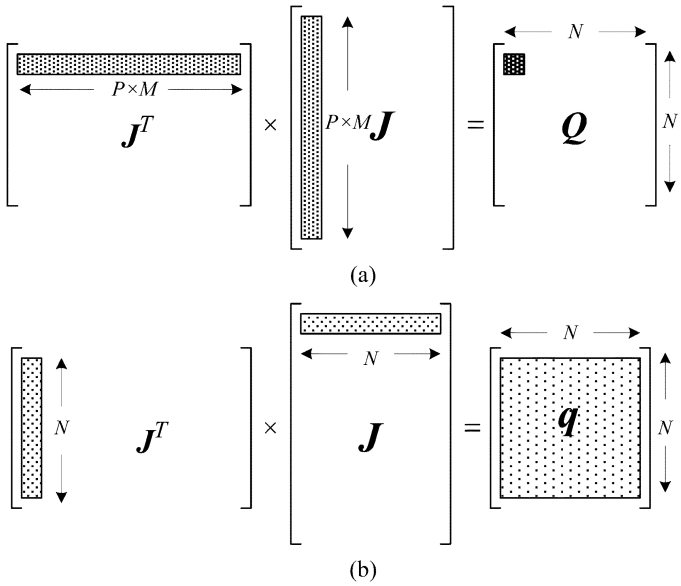


Fig. 3. Two ways of multiplying matrixes: (a) row–column multiplication results in a scalar; (b) column–row multiplication results in a partial matrix $\boldsymbol{q}$.

Equations (1) and (12) are producing identical results for weight updating. The major difference is that in (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 is done in (1).

A. 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. 3(a). 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. 3(b)] [11]. The number of scalars is $N \times N$, while number of partial matrices $\boldsymbol{q}$ which later have to be summed is $P \times M$.

When $\boldsymbol{J}^{\mathrm{T}}$ is multiplied by $\boldsymbol{J}$ using routine shown in Fig. 3(b), 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. 3(b) seems complicated, therefore almost all matrix multiplication processes use the routine of Fig. 3(a), where only one element of resulted matrix is calculated and stored every time.

Even the routine of Fig. 3(b) 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. 3(a), but they are performed in different order.

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. 3(b) 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 I
MEMORY COST ANALYSIS

Multiplication Methods	*Elements for storage*
Row-column (Fig. 3a)	$(P \times M) \times N \quad + \quad N \times N \quad + \quad N$
Column-row (Fig. 3b)	$N \times N \quad + \quad N$
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 I show that the column–row multiplication [Fig. 3(b)] can save a lot of memory.

B. Improved Quasi-Hessian Matrix Computation

Let us introduce quasi-Hessian submatrix $\boldsymbol{q}_{\boldsymbol{p}m}$ (size: $N \times N$)

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

Using (7) and (13), the $N \times N$ quasi-Hessian matrix $\boldsymbol{Q}$ can be calculated as the sum of submatrices $\boldsymbol{q}_{\boldsymbol{p}m}$

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

By introducing $1 \times N$ vector $\boldsymbol{j}_{\boldsymbol{p}m}$

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

submatrices $\boldsymbol{q}_{\boldsymbol{p}m}$ in (13) can be also written in the vector form [Fig. 3(b)]

$$\boldsymbol{q}_{\boldsymbol{p}m} = \boldsymbol{j}_{pm}^T \boldsymbol{j}_{pm}. \tag{16}$$

One may notice that for the computation of submatrices $\boldsymbol{q}_{\boldsymbol{p}m}$, only N elements of vector $\boldsymbol{j}_{\boldsymbol{p}m}$ need to be calculated and stored. All the submatrixes 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 submatrices $\boldsymbol{q}_{\boldsymbol{p}m}$. Each submatrix can be summed to a temporary matrix after its computation. Therefore, during the direct computation of quasi-Hessian matrix $\boldsymbol{Q}$ using (14), only memory for N elements is required, instead of that for the whole Jacobian matrix with $(P \times M) \times N$ elements (Table I).

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

C. Improved Gradient Vector Computation

Gradient subvector $\boldsymbol{\eta}_{\boldsymbol{p}m}$ (size: $N \times 1$) is

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

Combining (10) and (17), gradient vector $\boldsymbol{g}$ can be calculated as the sum of gradient subvector $\boldsymbol{\eta}_{\boldsymbol{p}m}$

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

Using the same vector $\boldsymbol{j}_{\boldsymbol{p}m}$ defined in (15), gradient subvector can be calculated using

$$\boldsymbol{\eta}_{pm} = \boldsymbol{j}_{pm} e_{pm}. \tag{19}$$

Similarly, gradient subvector $\boldsymbol{\eta}_{\boldsymbol{p}m}$ can be calculated for each pattern and output separately, and summed to a temporary vector. Since the same vector $\boldsymbol{j}_{\boldsymbol{p}m}$ is calculated during quasi-Hessian matrix computation above, there is only an extra scalar $e_{\boldsymbol{p}m}$ that 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 $\boldsymbol{j}_{\boldsymbol{p}m}$ 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 II, the memory cost for the storage of Jacobian elements could be reduced from more than 35 GB to nearly 30.7 kB.

D. Simplified $\partial e_{pm}/\partial w_i$ Computation

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

The elements of vector $\boldsymbol{j}_{\boldsymbol{p}m}$ can be calculated by

$$\frac{\partial e_{pm}}{\partial w_i} = \frac{\partial (o_{pm} - d_{pm})}{\partial w_i} = \frac{\partial o_{pm}}{\partial \mathrm{net}_{pn}} \frac{\partial \mathrm{net}_{pn}}{\partial w_i} \tag{20}$$

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

$$\mathrm{net}_{pn} = \sum_{i=1}^{I} x_{pi} w_i \tag{21}$$

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 (20) and (21) into (15), vector $\boldsymbol{j}_{\boldsymbol{p}m}$ can be calculated by

$$\boldsymbol{j}_{pm} = \left[\frac{\partial o_{pm}}{\partial \mathrm{net}_{p1}} [\, x_{p1,1} \quad \cdots \quad x_{p1,i} \quad \cdots] \quad \cdots \right. \\ \left. \frac{\partial o_{pm}}{\partial \mathrm{net}_{pn}} [\, x_{pm,1} \quad \cdots \quad x_{pn,i} \quad \cdots] \quad \cdots \right] \tag{22}$$

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

Fig. 4. Parity-2 problem: four patterns, two inputs, and one output.

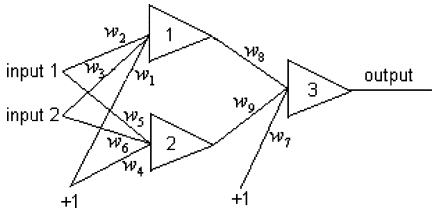


Fig. 5. Three neurons in MLP network used for training parity-2 problem; weight and neuron indexes are marked in the figure.

```
% 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. (22)
        calculate sub matrix q_pm;      % Eq. (16)
        calculate sub vector η_pm;      % Eq. (19)
        Q=Q+q_pm;                       % Eq. (14)
        g=g+η_pm;                       % Eq. (18)
    end;
end;
```

Fig. 6. Pseudocode of the improved computation for quasi-Hessian matrix and gradient vector.

where $x_{pn,i}$ is the ith input of neuron n, when training pattern p.

Using the neuron by neuron computation [20], [21], elements $x_{pn,i}$ in (22) can be calculated in the forward computation, while $\partial o_{pm}/\partial \mathrm{net}_{pn}$ 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.

IV. 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 four training patterns, two inputs, and one output. See Fig. 4.

The structure, three neurons in MLP topology (see Fig. 5), is used.

As shown in Fig. 5, weight values are initialed as 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 as follows:

1) $\mathrm{net}_{11} = 1 \times w_1 + (-1) \times w_2 + (-1) \times w_3$;
2) $o_{11} = f(\mathrm{net}_{11})$;
3) $\mathrm{net}_{12} = 1 \times w_4 + (-1) \times w_5 + (-1) \times w_6$;
4) $o_{12} = f(\mathrm{net}_{12})$;
5) $\mathrm{net}_{13} = 1 \times w_7 + o_{11} \times w_8 + o_{12} \times w_9$;
6) $o_{13} = f(\mathrm{net}_{13})$;
7) $e_{11} = 1 - o_{13}$.

   Then the backward computation is performed to calculate $\partial e_{11}/\partial \mathrm{net}_{11}, \partial e_{11}/\partial \mathrm{net}_{12}$ and $\partial e_{11}/\partial \mathrm{net}_{13}$ in the following steps.
8) With results of steps (f) and (g), it can be calculated

$$s_3 = \frac{\partial e_{11}}{\partial \mathrm{net}_{13}} = \frac{\partial (1 - f(\mathrm{net}_{13}))}{\partial \mathrm{net}_{13}} = -\frac{\partial f(\mathrm{net}_{13})}{\partial \mathrm{net}_{13}}. \quad (23)$$

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

$$s_2 = \frac{\partial e_{11}}{\partial \mathrm{net}_{12}} = -\frac{\partial f(\mathrm{net}_{12})}{\partial \mathrm{net}_{12}} \times w_9 \times \frac{\partial f(\mathrm{net}_{13})}{\partial \mathrm{net}_{13}} \quad (24)$$

$$s_1 = \frac{\partial e_{11}}{\partial \mathrm{net}_{11}} = -\frac{\partial f(\mathrm{net}_{11})}{\partial \mathrm{net}_{11}} \times w_8 \times \frac{\partial f(\mathrm{net}_{13})}{\partial \mathrm{net}_{13}}. \quad (25)$$

In this example, using (22), the vector $\boldsymbol{j}_{11}$ is calculated as

$$j_{11} = \left[\frac{\partial e_{11}}{\partial \mathrm{net}_{11}} \times [1 \quad -1 \quad -1] \frac{\partial e_{11}}{\partial \mathrm{net}_{12}} \times [1 \quad -1 \quad -1] \times \frac{\partial e_{11}}{\partial \mathrm{net}_{13}} \times [1 \quad o_{11} \quad o_{12}] \right]. \quad (26)$$

With (16) and (19), submatrix $\boldsymbol{q}_{11}$ and subvector $\boldsymbol{\eta}_{11}$ can be calculated separately

$$\boldsymbol{q}_{11} = \begin{bmatrix} s_1^2 & -s_1^2 & -s_1^2 & \cdots & s_1 s_3 o_{11} & s_1 s_3 o_{12} \\ 0 & s_1^2 & s_1^2 & \cdots & -s_1 s_3 o_{11} & -s_1 s_3 o_{12} \\ 0 & 0 & s_1^2 & \cdots & -s_1 s_3 o_{11} & -s_1 s_3 o_{12} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \cdots & s_3^2 o_{11} o_{12} \\ 0 & 0 & 0 & \cdots & 0 & s_3^2 o_{12}^2 \end{bmatrix} \quad (27)$$

$$\boldsymbol{\eta}_{11} = [s_1 \quad -s_1 \quad -s_1 \quad \cdots \quad s_3 o_{11} \quad s_3 o_{12}] \times e_{11}. \quad (28)$$

One may notice that only upper triangular elements of submatrix $\boldsymbol{q}_{11}$ are calculated, since all submatrixes are symmetrical. This can save nearly half of computation.

The last step is to add submatrix $\boldsymbol{q}_{11}$ and subvector $\boldsymbol{\eta}_{11}$ 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 pseudocode shown in Fig. 6.

TABLE II
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 III
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.

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

V. EXPERIMENTAL RESULTS

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: memory comparison and time comparison.

A. Memory Comparison

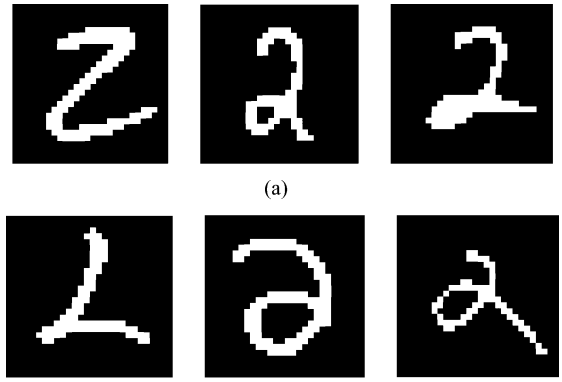
Three problems, each of which has huge number of patterns, are selected to test the memory cost of both the traditional computation and the improved computation. LM algorithm is used for training and the test results are shown Tables II and III. In order to make more precise comparison, memory cost for program code and input files were not used in the comparison.

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

In the MNIST problem [4], there are 60 000 training patterns, each of which is a digit (from 0 to 9) image made up of grayed 28 $\times$ 28 pixels. 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. 7(a)]; for seriously rotated or distorted images, it is hard to recognize them [see Fig. 7(b)].

B. 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.



(a)

(b)

Fig. 7. Some testing results for digit "2" recognition. (a) Recognized patterns. (b) Unrecognized patterns.

TABLE IV
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 IV, 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 LM algorithm, not only on memory requirements, but also training time.

VI. CONCLUSION

In this paper, the improved computation is introduced to increase the training efficiency of LM 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 one can see from the two-spiral example in Section I, first-order algorithm (EBP algorithm) is not able to solve some problems unless excessive number of neurons is used (Fig. 1). 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 paper at least solved the problem of training neural networks using second-order algorithm with basically unlimited number of training patterns.[1]

REFERENCES

[1] A. Y. Alanis, E. N. Sanchez, and A. G. Loukianov, "Discrete-time adaptive backstepping nonlinear control via high-order neural networks," *IEEE Trans. Neural Netw.*, vol. 18, no. 4, pp. 1185–1195, Jul. 2007.

[2] J. R. Alvarez-Sanchez, "Injecting knowledge into the solution of the two-spiral problem," *Neural Comput. Appl.*, vol. 8, pp. 265–272, Aug. 1999.

[3] N. Ampazis and S. J. Perantonis, "Two highly efficient second-order algorithms for training feedforward networks," *IEEE Trans. Neural Netw.*, vol. 13, no. 5, pp. 1064–1074, Sep. 2002.

[4] L. J. Cao, S. S. Keerthi, C.-J. Ong, J. Q. Zhang, U. Periyathamby, X. J. Fu, and H. P. Lee, "Parallel sequential minimal optimization for the training of support vector machines," *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 1039–1049, Jul. 2006.

[5] J. A. Farrell and M. M. Polycarpou, "Adaptive approximation based control: Unifying neural, fuzzy and traditional adaptive approximation approaches," *IEEE Trans. Neural Netw.*, vol. 19, no. 4, pp. 731–732, Apr. 2008.

[6] S. Ferrari and M. Jensenius, "A constrained optimization approach to preserving prior knowledge during incremental training," *IEEE Trans. Neural Netw.*, vol. 19, no. 6, pp. 996–1009, Jun. 2008.

[7] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Trans. Neural Netw.*, vol. 5, no. 6, pp. 989–993, Nov. 1994.

[8] S. Khomfoi and L. M. Tolbert, "Fault diagnostic system for a multilevel inverter using a neural network," *IEEE Trans. Power Electron.*, vol. 22, no. 3, pp. 1062–1069, May 2007.

[9] C.-T. Kim and J.-J. Lee, "Training two-layered feedforward networks with variable projection method," *IEEE Trans. Neural Netw.*, vol. 19, no. 2, pp. 371–375, Feb. 2008.

[1]The method was implemented in neural networks trainer (NBN 2.08) [23], and the software can be downloaded from website: http://www.eng.auburn.edu/~wilambm/nnt/index.htm.

[10] M. Kyperountas, A. Tefas, and I. Pitas, "Weighted piecewise LDA for solving the small sample size problem in face verification," *IEEE Trans. Neural Netw.*, vol. 18, no. 2, pp. 506–519, Mar. 2007.

[11] D. C. Lay, *Linear Algebra and Its Applications*, 3rd ed. , MA: Addison-Wesley, 2005.

[12] K. Levenberg, "A method for the solution of certain problems in least squares," *Quart. Appl. Mach.*, vol. 2, pp. 164–168, 1944.

[13] Y. Liu, J. A. Starzyk, and Z. Zhu, "Optimized approximation algorithm in neural networks without overfitting," *IEEE Trans. Neural Netw.*, vol. 19, no. 6, pp. 983–995, Jun. 2008.

[14] J. F. Martins, V. F. Pires, and A. J. Pires, "Unsupervised neural-network-based algorithm for an online diagnosis of three-phase induction motor stator fault," *IEEE Trans. Ind. Electron.*, vol. 54, no. 1, pp. 259–264, Feb. 2007.

[15] J.-X. Peng, K. Li, and G. W. Irwin, "A new Jacobian matrix for optimal learning of single-layer neural networks," *IEEE Trans. Neural Netw.*, vol. 19, no. 1, pp. 119–129, Jan. 2008.

[16] V. V. Phansalkar and P. S. Sastry, "Analysis of the back-propagation algorithm with momentum," *IEEE Trans. Neural Netw.*, vol. 5, no. 3, pp. 505–506, May. 1994.

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

[18] B. Vigdor and B. Lerner, "Accurate and fast off and online fuzzy ARTMAP-based image classification with application to genetic abnormality diagnosis," *IEEE Trans. Neural Netw.*, vol. 17, no. 5, pp. 1288–1300, Sep. 2006.

[19] P. J. Werbos, "Back-propagation: Past and future," in *Proc. Int. Conf. Neural Netw.*, San Diego, CA, Jul. 1988, vol. 1, pp. 343–353.

[20] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dundar, "Computing gradient vector and Jacobian matrix in arbitrarily connected neural networks," *IEEE Trans. Ind. Electron.*, vol. 55, no. 10, pp. 3784–3790, Oct. 2008.

[21] 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, Dec. 2009.

[22] J. M. Wu, "Multilayer Potts perceptrons with Levenberg-Marquardt learning," *IEEE Trans. Neural Netw.*, vol. 19, no. 12, pp. 2032–2043, Feb. 2008.

[23] H. Yu and B. M. Wilamowski, "Efficient and reliable training of neural networks," in *Proc. IEEE Human Syst. Interaction Conf.*, Catania, Italy, May 21–23, 2009, pp. 109–115.

Bogdan M. Wilamowski (M'82–SM'83–F'00) received the M.S. degree in computer engineering, the Ph.D. degree in neural computing, and the Dr. Habil. degree in integrated circuit design from Gdansk University of Technology, Gdansk, Poland, in 1966, 1970, and 1977, respectively.

He received the title of Full Professor from the President of Poland in 1987. He was the Director of the Institute of Electronics (1979–1981) and the Chair of the Solid State Electronics Department (1987–1989), Technical University of Gdansk, Gdansk, Poland. He was/has been a Professor with the Gdansk University of Technology, Gdansk (1987–1989), the University of Wyoming, Laramie (1989–2000), and the University of Idaho, Moscow (2000–2003). Since 2003, he has been with the Auburn University, Auburn, AL, where he is currently the Director of the Alabama Micro/Nano Science and Technology Center and a Professor with the Department of Electrical and Computer Engineering. He was also with the Research Institute of Electronic Communication, Tohoku University, Sendai, Japan (1968–1970), and the Semiconductor Research Institute, Sendai (1975–1976), Auburn University (1981–1982 and 1995–1996), and the University of Arizona, Tucson (1982–1984). He is the author of four textbooks and about 300 refereed publications and is the holder of 28 patents. He was the Major Professor for over 130 graduate students. His main areas of interest include computational intelligence and soft computing, computer-aided design development, solid-state electronics, mixed- and analog-signal processing, and network programming.

Dr. Wilamowski was the President of the IEEE Industrial Electronics Society (2004–2005). He was an Associate Editor for the IEEE TRANSACTIONS ON NEURAL NETWORKS, the IEEE TRANSACTIONS ON EDUCATION, the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, the *Journal of Intelligent and Fuzzy Systems*, the *Journal of Computing*, the *International Journal of Circuit Systems*, and the *IES Newsletter*. He also was the Editor-in-Chief of the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS.

Hao Yu received the M. S. degree in electrical engineering from Huazhong University of Science and Technology, Wuhan, Hubei, China. He is currently working towards the Ph.D. degree in electrical engineering at Auburn University, Auburn, AL.

He is a Research Assistant with the Department of Electrical and Computer Engineering, Auburn University. His main interests include computational intelligence, neural networks and CAD.

Mr. Yu is a Reviewer for the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS.