Fast and Efficient and Training of Neural Networks

Hao Yu and Wilamowski
Auburn University, Auburn, Alabama, US
hzy0004@auburn.edu, wilam@ieee.org

Abstract. **In this paper, second order algorithms, such as Levenberg Marquardt algorithm [1][2], are recommended for neural network training. Being different from traditional computation in second order algorithms, the proposed method simplifies Hessian matrix computation, by removing Jacobian matrix computation and storage. Matrix multiplications are replaced by vector operations. The proposed computation not only makes the training process faster, but also reduces the memory cost significantly. Based upon the improvement, second order algorithms can be applied for application with unlimited number of patterns.**

Keywords**: Neural network training, Levenberg Marquardt algorithm**

I. Introduction

NEURAL networks can be very powerful for nonlinear signal processing [3][4], if they are trained well. The well-trained neural networks should not only correctly match the training patterns, but also properly handle the patterns which are not applied for training. Dissatisfaction in either condition makes the trained neural networks useless.

In order to secure the generalization ability, the networks should be as simple as possible [5]. With first order algorithms, such as the error back propagation (EBP) algorithm [6], it is often not possible to train those simple (close to optimal) networks. Instead, an excessive number of neurons are required to match the training patterns with small errors. This "success" is misleading because such networks will be over-fitted and perform poorly in processing new patterns which were not used for training.

It is well-known that second order algorithms, such as Levenberg Marquardt (LM) algorithm, are much faster than EBP algorithm. For the same problem, LM algorithm can also find solutions with much smaller number of neurons than that required for EBP algorithm. Therefore, it is reasonable to consider LM algorithm as a potentially better choice than EBP algorithm to find well-trained networks.

This issue will be illustrated with the example below. Let us consider the peak surface [7] as the required surface (Fig. 1a) and let us use equally spaced *10×10=100* patterns (Fig. 1b) 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 error, the sum squared error (SSE), as defined in (4), is divided by 100 and 1,369 respectively.

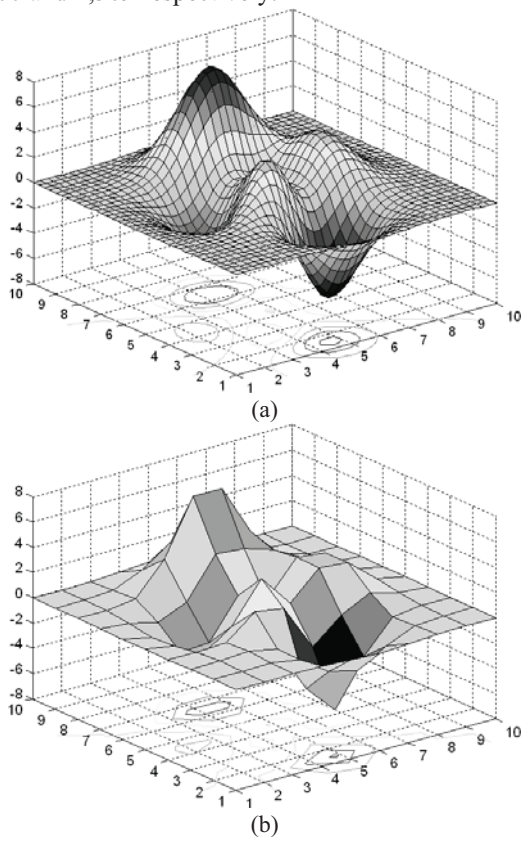

Fig. 1 Surface matching problem: (a) Required 2-D surface with *37×37=1,369* points, used for verification; (b) *10×10=100* training patterns extracted in equal space from (a), used for training

As the training results shown in Table I, using the neuron by neuron (NBN) algorithm [8][9], which is an improved LM algorithm for training arbitrarily connected neural networks, it was possible to find the acceptable

Hao Yu: 468 Broun Hall, Auburn University, Auburn, AL. 36849-5201 USA. Phone: +1-334-498-0953. Email: hzy0004@auburn.edu.

Bogdan M. Wilamowski: 420 Broun Hall, Auburn University, Auburn, AL. 36849-5201 USA. Phone: +1-334-844-1629. Email: wilam@ieee.org.

978-1-4244-7562-9/10/$26.00 ©2010 IEEE

solution (Fig. 2) 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. 3 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 ability to response correctly for new patterns (between training points). Please notice that indeed with enlarged number of neurons (13 neurons), EBP algorithm was able to train network to a small error SSE_{Train}=0.0018, but as one can see from Fig. 4, the result is unacceptable with verification error SSE_{Verify}=0.4909.

TABLE I
TRAINING RESULTS OF PEAK SURFACE PROBLEM

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; all neurons are in FCC networks; there are 100 trials for each case.

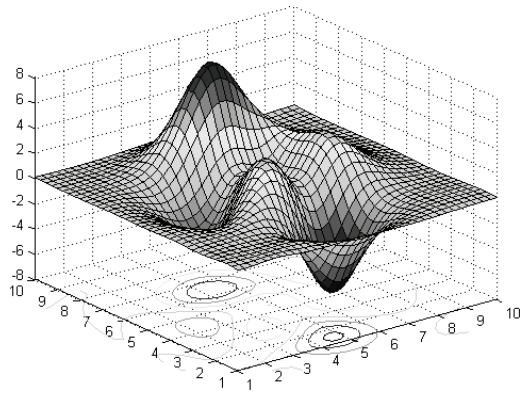

Fig. 2 The best training result in 100 trials, using LM algorithm, 8 neurons in FCC network (52 weights); maximum training iteration is 1,000; SSE_{Train}=0.0044, SSE_{Verify}=0.0080 and training time=0.37 s

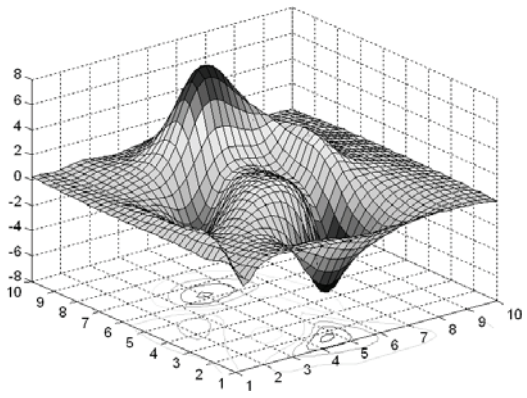

Fig. 3 The best training result in 100 trials, using EBP algorithm, 8 neurons in FCC network (52 weights); maximum training iteration is 1,000,000; SSE_{Train}=0.0764, SSE_{Verify}=0.1271 and training time=579.98 s

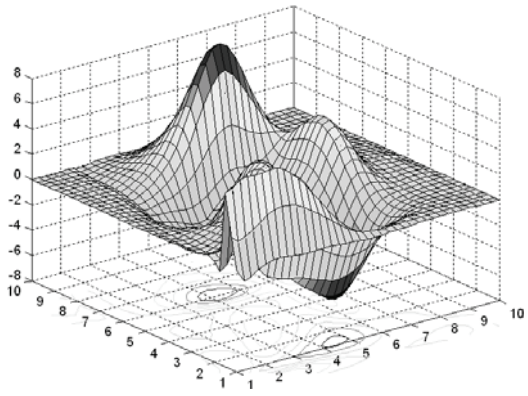

Fig. 4 The best training result in 100 trials, using EBP algorithm, 13 neurons in FCC network (117 weights); maximum training iteration is 1,000,000; SSE_{Train}=0.0018, SSE_{Verify}=0.4909 and training time=635.72 s

From the presented example, one may notice that often in simple (close to optimal) networks, EBP algorithm can't converge to required training error (Fig. 3). When the size of networks increase, EBP algorithm can reach the required training error, but trained networks lose their generalization ability and can't process new patterns well (Fig. 4). On the other hand, second order algorithms, such as LM algorithm, works not only significantly faster but it can find good solutions with close to optimal networks (Fig. 2).

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 (65536 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 of this paper, 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 gives some experimental results on memory and training time comparison between the traditional computation and the improved computation.

II. COMPUTATIONAL FUNDAMENTALS

Derived from EBP algorithm and Newton method, the update rule of Levenberg Marquardt algorithm is [10]

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

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

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

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 [12] 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.

III. Improved computation

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

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

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

The Hessian matrix can be found by

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

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

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

where i and j are weight indexes.

Equation (6) can be approximated as [10][11]

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

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

Combining (3) and (7), quasi Hessian matrix $\boldsymbol{Q}$ can be calculated as an approximation of Hessian matrix

$$\boldsymbol{H}\approx\boldsymbol{Q}=2\boldsymbol{J}^T\boldsymbol{J} \tag{8}$$

Gradient vector $\boldsymbol{g}$ is defined as

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

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

$$g_i=\frac{\partial E}{\partial w_i}=2\sum_{p=1}^{P}\sum_{m=1}^{M}\left(\frac{\partial e_{pm}}{\partial w_i}e_{pm}\right) \tag{10}$$

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

$$\boldsymbol{g}=2\boldsymbol{J}^T\boldsymbol{e} \tag{11}$$

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

$$\Delta\boldsymbol{w}=\left(\frac{1}{2}\boldsymbol{Q}+\mu\boldsymbol{I}\right)^{-1}\frac{1}{2}\boldsymbol{g} \tag{12}$$

or

$$\Delta\boldsymbol{w}=\left(\boldsymbol{Q}+\mu\boldsymbol{I}\right)^{-1}\boldsymbol{g} \tag{13}$$

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

In order to avoid computation and storage of large Jacobian matrix $\boldsymbol{J}$ in (1), the quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$ must be computed directly.

A. Improved quasi Hessian matrix computation

Let us introduce quasi Hessian sub matrix $\boldsymbol{q_{pm}}$

$$\boldsymbol{q}_{pm} = \begin{bmatrix} \left(\frac{\partial e_{pm}}{\partial w_1}\right)^2 & \frac{\partial e_{pm}}{\partial w_1}\frac{\partial e_{pm}}{\partial w_2} & \cdots & \frac{\partial e_{pm}}{\partial w_1}\frac{\partial e_{pm}}{\partial w_N} \\ \frac{\partial e_{pm}}{\partial w_2}\frac{\partial e_{pm}}{\partial w_1} & \left(\frac{\partial e_{pm}}{\partial w_2}\right)^2 & \cdots & \frac{\partial e_{pm}}{\partial w_2}\frac{\partial e_{pm}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{pm}}{\partial w_N}\frac{\partial e_{pm}}{\partial w_1} & \frac{\partial e_{pm}}{\partial w_N}\frac{\partial e_{pm}}{\partial w_2} & \cdots & \left(\frac{\partial e_{pm}}{\partial w_N}\right)^2 \end{bmatrix} \quad (14)$$

Using (7) and (14), quasi Hessian matrix $\boldsymbol{Q}$ can be calculated as the sum of sub matrix $\boldsymbol{q}_{pm}$

$$\boldsymbol{Q} = 2\sum_{p=1}^{P}\sum_{m=1}^{M}\boldsymbol{q}_{pm} \quad (15)$$

By introducing vector $\boldsymbol{j}_{pm}$

$$\boldsymbol{j}_{pm} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} & \frac{\partial e_{pm}}{\partial w_2} & \cdots & \frac{\partial e_{pm}}{\partial w_N} \end{bmatrix} \quad (16)$$

sub matrix $\boldsymbol{q}_{pm}$ in (14) can be also written in the vector form

$$\boldsymbol{q}_{pm} = \boldsymbol{j}_{pm}^T \boldsymbol{j}_{pm} \quad (17)$$

One may notice that for the computation of sub matrix $\boldsymbol{q}_{pm}$, 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 matrix $\boldsymbol{q}_{pm}$. 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 (15), only memory for N elements is required, instead of that for the whole Jacobian matrix with $P \times M \times N$ elements (2).

From (14), 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 (17) and sum operations in (15) can be both reduced to half approximately. As is known, binary multiplication often costs several times executing periods more than data copying. Therefore, this simplification is supposed to perform a much more efficient computation.

B. Improved gradient vector computation

Gradient sub vector $\boldsymbol{\eta}_{pm}$ is introduced as

$$\boldsymbol{\eta}_{pm} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} e_{pm} \\ \frac{\partial e_{pm}}{\partial w_2} e_{pm} \\ \cdots \\ \frac{\partial e_{pm}}{\partial w_N} e_{pm} \end{bmatrix} = \begin{bmatrix} \frac{\partial e_{pm}}{\partial w_1} \\ \frac{\partial e_{pm}}{\partial w_2} \\ \cdots \\ \frac{\partial e_{pm}}{\partial w_N} \end{bmatrix} \times e_{pm} \quad (18)$$

Combining (10) and (18), gradient vector $\boldsymbol{g}$ can be calculated as the sum of gradient sub vector $\boldsymbol{\eta}_{pm}$

$$\boldsymbol{g} = 2\sum_{p=1}^{P}\sum_{m=1}^{M}\boldsymbol{\eta}_{pm} \quad (19)$$

Using the same vector $\boldsymbol{j}_{pm}$ defined in (16), gradient sub vector can be calculated using

$$\boldsymbol{\eta}_{pm} = \boldsymbol{j}_{pm} e_{pm} \quad (20)$$

Similarly, gradient sub vector $\boldsymbol{\eta}_{pm}$ can be calculated for each pattern and output separately, and summed to a temporary vector. After training all patterns and outputs, the temporary vector is the required gradient. 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.

C. Simplified $\partial e_{pm}/\partial w_i$ computation

One may notice that the key point of the proposed computation above for quasi Hessian matrix $\boldsymbol{Q}$ and gradient vector $\boldsymbol{g}$ is to calculate vector $\boldsymbol{j}_{pm}$ defined in (16) 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

$$\frac{\partial e_{pm}}{\partial w_i} = \frac{\partial (o_{pm} - d_{pm})}{\partial w_i} = \frac{\partial o_{pm}}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_i} \quad (21)$$

where $\boldsymbol{d}$ is the desired output and $\boldsymbol{o}$ is the actual output. net_{pj} is the sum of weighted inputs at neuron j described as

$$net_{pj} = \sum x_{pi} w_i \quad (22)$$

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

Inserting (21) and (22) into (16), vector $\boldsymbol{j}_{pm}$ can be calculated by

$$\boldsymbol{j}_{pm} = \begin{bmatrix} \frac{\partial o_{pm}}{\partial net_{p1}} \begin{bmatrix} x_{p1_1} & \cdots & x_{p1_i} & \cdots \end{bmatrix} & \cdots \\ \frac{\partial o_{pm}}{\partial net_{pj}} \begin{bmatrix} x_{pj_1} & \cdots & x_{pj_i} & \cdots \end{bmatrix} & \cdots \end{bmatrix} \quad (23)$$

where x_{pji} is the ith input for neuron j, when training pattern p.

Generally, for the problem with P patterns and M outputs, the improved computation can be organized as the pseudo code shown in Fig. 5.

```
% 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 jpm;          % Eq. (23)
    calculate sub matrix qpm;      % Eq. (17)
    calculate sub vector ηpm;      % Eq. (20)
    Q=Q+qpm;                       % Eq. (15)
    g=g+ηpm;                       % Eq. (19)
  end;
end;
```

Fig. 5 Pseudo code of the improved computation for quasi Hessian matrix and gradient vector

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 (23); 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 gigabytes to nearly 30.7 kilobytes. Therefore, it is possible to train those huge patterns with LM algorithm.

The same quasi Hessian matrices and gradient vectors are obtained in both traditional computation (8 and 11) and the proposed computation (15 and 19). Therefore, the proposed computation does not affect the success rate.

IV. Examples

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: (A) Memory comparison and (B) Time comparison.

A. 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 Tables II and III. In order to make a more precise comparison, memory cost for program code and input files were not used in the comparison.

TABLE II
MEMORY COMPARISON FOR PARITY PROBLEMS

Parity-N Problems	*Parity 14*	*Parity 16*
Patterns	16,384	65,536
Structures*	15 neurons	17 neurons
Jacobian matrix	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.

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

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.

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	38.5	59.0	68.1	126.1
Success Rate	58%	37%	24%	12%
Algorithms	*Averaged training time (s)*			
Traditional LM	0.8	68.0	1508.5	43,417.1
Improved LM	0.3	22.1	173.8	2,797.9

From Table IV, one may notice that the improved computation can not only handle much larger problems, but 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.

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.

Another experiment, two-spiral problem, was used to test the efficiency of LM algorithm implemented with the proposed computation, comparing with EBP algorithm.

The two-spiral problem is considered as a good evaluation of training algorithms [7]. Depending on training algorithms, different numbers of neurons are required for successful training (See Tables V).

Table V presents the training results of the two-spiral problem using different number of neurons in fully connected cascade (FCC) networks. LM algorithm with the proposed implementation can solve the two-spiral

problem, using 8 neurons (52 weights) in about 290 iterations (Fig .6a). EBP algorithm can solve it in about 400,000 iterations, but only if when the number of neurons is increased to 12 (102 weights). The result (the best one in 100 trials), shown in Fig. 6b, is not as good as the result (Fig. 6a) from LM algorithm with a much simpler architecture. One can conclude that EBP algorithm is only successful if an excessive number of neurons is used.

TABLE V
TRAINING RESULTS OF TWO-SPIRAL PROBLEM

Neurons	Success rate		Average number of iterations		Average time (s)	
	EBP	LM	EBP	LM	EBP	LM
8	0%	13%	Failing	287.7	Failing	0.88
9	0%	24%	Failing	261.4	Failing	0.98
10	0%	40%	Failing	243.9	Failing	1.57
11	0%	69%	Failing	231.8	Failing	1.62
12	63%	80%	410,254	175.1	633.9	1.70
13	85%	89%	335,531	159.7	620.3	2.09
14	92%	92%	266,237	137.3	605.3	2.40
15	96%	96%	216,064	127.7	601.0	2.89
16	98%	99%	194,041	112.0	585.7	3.82

For EBP algorithm, learning constant is 0.005 and momentum is 0.5; maximum iteration is 1,000,000 for EBP algorithm and 1,000 for LM algorithm; desired error=0.01; all neurons are in FCC networks; there are 100 trials for each case.

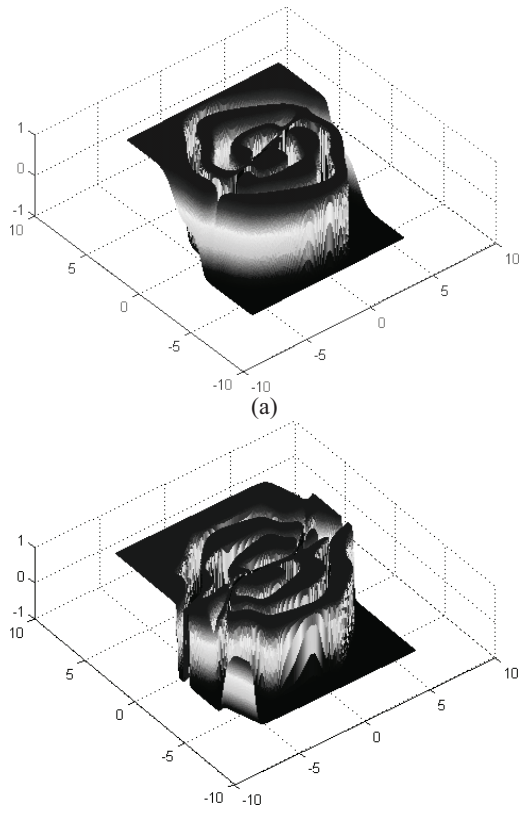

Fig. 6 Best results of two-spiral problem in 100 trails: (a) 8 neurons in FCC network (52 weights), using LM algorithm and training time=0.82 s; (b) 12 neurons in FCC network (102 weights), using EBP algorithm and training time=694.32 s

The experimental results of the two-spiral problem show that second order algorithms are not only much faster but they can train reduced size networks which can't be handled by EBP algorithm.

V. Conclusion

In this paper, an improved Hessian matrix computation was presented to solve the memory limitation in second order algorithms. 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. An 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.

Second order algorithms are much more efficient than EBP algorithm. EBP algorithm can solve problems only when excessive number of neurons is used, where the networks lose their generalization ability (example in section I).

The method presented in this paper solved the problem of training neural networks using second order algorithms with basically unlimited number of training patterns.

The method was implemented in neural networks trainer (NBN 2.09), and the software can be downloaded from website [13]:
http://www.eng.auburn.edu/~wilambm/nnt/index.htm

References

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

[2] Wu, J.-M., "Multilayer Potts Perceptrons with Levenberg–Marquardt Learning". *IEEE Trans. on Neural Networks*, vol. 19, no. 12, pp. 2032-2043, Feb 2008.

[3] C. Alzate, J. A. K. Suykens, "Kernel Component Analysis Using an Epsilon-Insensitive Robust Loss Function," *IEEE Trans. on Neural Networks,* vol. 19, no. 9, pp. 1583-1598, Sept 2008.

[4] H. Deng, H.-X. Li, Y.-H. Wu, "Feedback-Linearization-Based Neural Adaptive Control for Unknown Nonaffine Nonlinear Discrete-Time Systems," *IEEE Trans. on Neural Networks,* vol. 19, no. 9, pp. 1615-1625, Sept 2008.

[5] B. M. Wilamowski, "Neural Network Architectures and Learning Algorithms: *How Not to Be Frustrated with Neural Networks*," *IEEE Industrial Electronics Magazine,* vol. 3, no. 4, pp. 56-63, Dec. 2009.

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

[7] Sheng 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.

[8] 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*, vol. 55, no. 10, pp. 3784-3790, Oct. 2008.

[9] Hao Yu and B. M. Wilamowski, "Efficient and reliable training of neural networks", in *Proc. 2nd IEEE Human System Interaction Conf. HSI 2009*, Catania, Italy, May 21-23, 2009, pp. 109-115.

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

[11] Jian-Xun 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.

[12] 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.

[13] Hao Yu and B. M. Wilamowski, “C++ Implementation of Neural Networks Trainer”, *13-th International Conference on Intelligent Engineering Systems*, INES-09, Barbados, April 16-18, 2009.