Neural Network Learning without Backpropagation

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

***Abstract*— The method introduced in this paper allows for training arbitrarily connected neural networks, therefore, more powerful neural network architectures with connections across layers can be efficiently trained. The proposed method also simplifies neural network training, by using the forward-only computation instead of the traditionally used forward and backward computation.**

Information needed for the gradient vector (for first-order algorithms) and Jacobian or Hessian matrix (for second-order algorithms) is obtained during forward computation. With the proposed algorithm, it is now possible to solve the same problems using a much smaller number of neurons because the proposed algorithm is able to train more complex neural network architectures that require a smaller number of neurons. Comparison results of computation cost show that the proposed forward-only computation can be faster than the traditional implementation of the Levenberg–Marquardt algorithm.

***Index Terms*— Forward-only computation, Levenberg–Marquardt algorithm, neural network training.**

I. Introduction

THE popular EBP algorithm [1] is relatively simple and it can handle problems with basically an unlimited number of patterns. Also, because of its simplicity, it was relatively easy to adopt the EBP algorithm for more efficient neural network architectures where connections across layers are allowed [2]. However, the EBP algorithm can be up to 1000 times slower than more advanced second-order algorithms [3]– [5]. Many improvements [6], [7] have been made to speed up the EBP algorithm and some of them, such as momentum [8], adaptive learning constant, and RPROP algorithm [9], work relatively well. But as long as first-order algorithms are used, improvements are not dramatic.

The very efficient second-order Levenberg–Marquardt (LM) algorithm [10], [11] was adopted for neural network training by Hagan and Menhaj [12], and later was implemented in the MATLAB neural network toolbox [13]. The LM algorithm uses significantly more number of parameters describing the error surface than just gradient elements as in the EBP algorithm. As a consequence, the LM algorithm is not only fast but it can also train neural networks for which the EBP algorithm has difficulty in converging [5]. Many researchers now are using the Hagan and Menhaj LM algorithm for neural network training, but this algorithm has also several disadvantages.

1) It cannot be used for problems with many training patterns because the Jacobian matrix becomes prohibitively too large.
2) The LM algorithm requires the inversion of a quasi-Hessian matrix of size $nw \times nw$ in every iteration, where nw is the number of weights. Because of the necessity of matrix inversion in every iteration, the speed advantage of the LM algorithm over the EBP algorithm is less evident as the network size increases.
3) The Hagan and Menhaj LM algorithm was developed only for multilayer perceptron (MLP) neural networks. Therefore, much more powerful neural networks [14], [15], such as fully connected cascade (FCC) or bridged multilayer preceptron architectures cannot be trained.
4) In implementing the LM algorithm, Hagan and Menhaj calculated elements of the Jacobian matrix using basically the same routines as in the EBP algorithm. The difference is that the error backpropagation process (for Jacobian matrix computation) must be carried on not only for every pattern but also for every output separately.

Problems 1) and 2) inherited the property of the original LM algorithm. The disadvantage 1) of the LM algorithm was addressed in the recently proposed modification of the LM algorithm [16]. The problem 2) is still unsolved, so the LM algorithm can be used only for small and medium size neural networks.

Also, an attempt was made to adopt the Hagan and Menhaj forward and backward computation routine for arbitrarily connected neural networks [2], but this method is relatively complicated. It is easier to handle these networks with arbitrarily connected neurons when there is no need for backward computation process.

In this paper, the limitations 3) and 4) of the Hagan and Menhaj LM algorithm [12] are addressed and the proposed method of computation allows it to train networks with arbitrarily connected neurons. This way, more complex feedforward neural network architectures than MLP can be efficiently trained. A further advantage of the proposed algorithm is that the learning process requires only forward computation without the necessity of the backward computations. This way, the proposed method, in many cases, may also lead to the reduction of the computation time.

The ability to solve problems with smaller networks is very important. The common mistake made by many researchers is the use of an excessive number of neurons. This way it is

Manuscript received April 1, 2010; revised July 29, 2010; accepted August 25, 2010. Date of current version November 3, 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).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNN.2010.2073482

much easier to train such networks, but when new patterns (not used in the training) are applied, the networks respond very poorly [5]. Since the EBP algorithm usually requires a long training time, over-eager researchers often end up with larger networks than required and this leads to frustration when such networks are used for new patterns.

In order to preserve the generalization abilities of neural networks, the size of the networks should be as small as possible. The proposed algorithm partially addresses this problem because it allows training smaller networks with arbitrarily connected neurons.

This paper is organized as follows. In Section II, a short review of the computational methodology for the first- and second-order algorithms is given. In Section III, the description of the proposed forward-only computation is presented. Section IV gives a comparison of computational efficiency between the Hagan and Menhaj LM algorithm and the proposed implementation of the LM algorithm. Section V is an experimental section where the advantages of the more powerful network architectures are shown and actual computation time is compared for both algorithms on several examples.

II. Computational Fundamentals

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

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

Other indices will be explained at appropriate places.

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

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

where $e_{p,m}$ is the error at output m defined as

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

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

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

A. Definition of Basic Concepts in Neural Network Training

Let us consider neuron j with ni inputs, as shown in Fig. 1. If neuron j is in the first layer, all its inputs would be connected to the inputs of the network, otherwise, its inputs can be connected to outputs of other neurons or to networks' inputs if connections across layers are allowed.

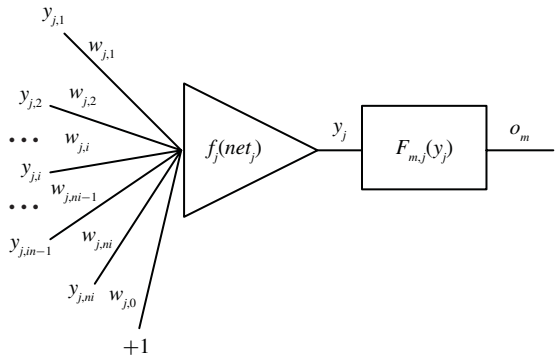


Fig. 1. Connection of a neuron j with the rest of the network. Nodes $y_{j,i}$ could represents network inputs or outputs of other neurons. $F_{m,j}(y_j)$ is the nonlinear relationship between the neuron output node y_j and the network output o_m.

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

Output node of neuron j is calculated using

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

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

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

where $y_{j,i}$ is the ith input node of neuron j, weighted by $w_{j,i}$, and $w_{j,0}$ is the bias weight of neuron j.

Using (4), one may notice that derivative of net_j is

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

and slope s_j of activation function f_j is

$$s_j = \frac{\partial y_j}{\partial net_j} = \frac{\partial f_j\left(net_j\right)}{\partial net_j}. \tag{6}$$

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

$$o_m = F_{m,j}\left(y_j\right) \tag{7}$$

where o_m is the mth output of the network.

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

B. Gradient Vector and Jacobian Matrix Computation

For every pattern, in EBP algorithm only one backpropagation process is needed, while in second-order algorithms the backpropagation process has to be repeated for every output separately in order to obtain consecutive rows of the Jacobian

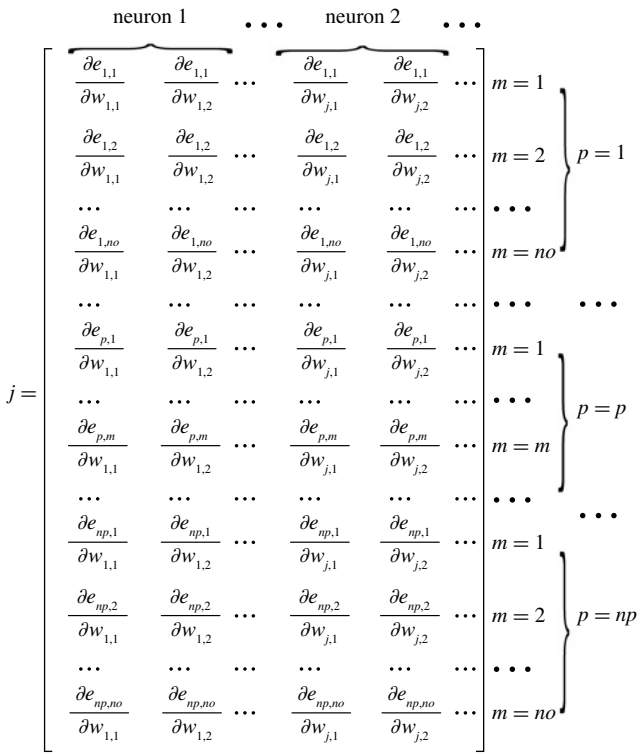


Fig. 2. Structure of Jacobian matrix. 1) The number of columns is equal to the number of weights. 2) Each row corresponds to a specified training pattern p and output m.

matrix (Fig. 2). Another difference in second-order algorithms is that the concept of backpropagation of the δ parameter [17] has to be modified. In EBP algorithm, output errors are parts of the δ parameter

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

In second-order algorithms, the δ parameters are calculated for each neuron j and each output m separately. Also, in the backpropagation process [12], the error is replaced by a unit value

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

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

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

In the EBP algorithm, elements of gradient vector are computed as

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

where δ_j is obtained with errorbackpropagation process. In second-order algorithms, gradient can be obtained from partial results of Jacobian calculations

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

where m indicates a network output and δ_{mj} is given by (9).

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

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

The update rule of the EBP algorithm is

$$w_{n+1} = w_n - \alpha\, g_n \tag{13}$$

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

Derived from the Newton algorithm and the steepest descent method, the update rule of the LM algorithm is [12], [18]

$$w_{n+1} = w_n - \left(J_n^T J_n + \mu\, I\right)^{-1} g_n \tag{14}$$

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

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

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

III. Forward-Only Computation

A. Derivation

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

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

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

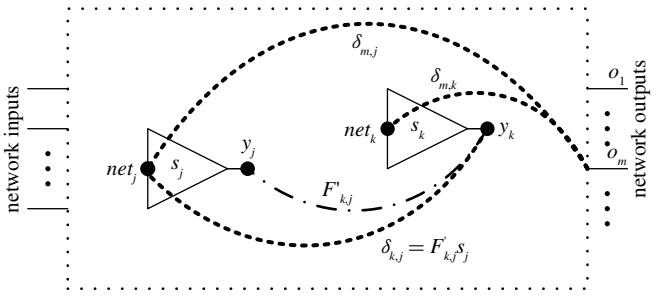


Fig. 4. Interpretation of $\delta_{k,j}$ as a signal gain, where in feedforward network neuron j must be located before neuron k.

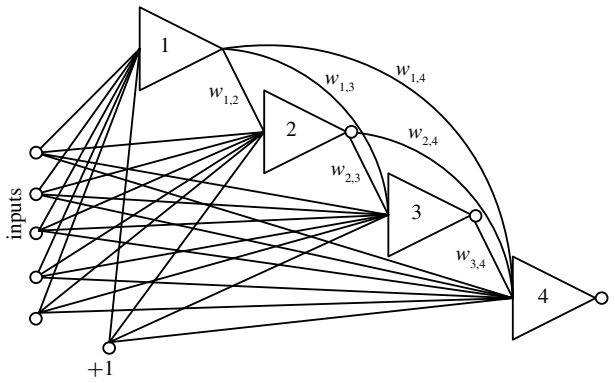


Fig. 5. Four neurons in fully connected neural network, with five inputs and three outputs.

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

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

B. Calculation of δ Matrix for FCC Architectures

Let us start our analysis with fully connected neural networks (Fig. 5). Any other architecture could be considered as a simplification of fully connected neural networks by eliminating connections (setting weights to zero). If the feedforward principle is enforced (no feedback), fully connected neural networks must have cascade architectures.

Slopes of neuron activation functions s_j can be also written in form of the δ parameter as $\delta_{j,j} = s_j$. By inspecting Fig. 6, the δ parameters can be written as follows.

For the first neuron, there is only one δ parameter

$$\delta_{1,1} = s_1. \tag{16}$$

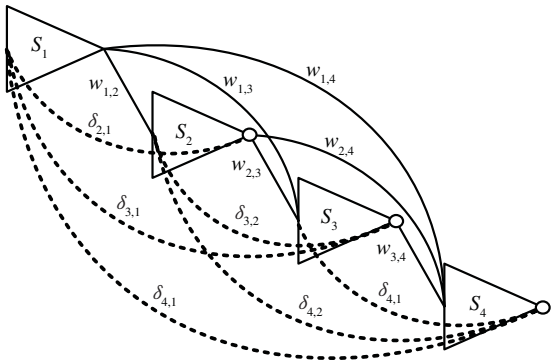


Fig. 6. $\delta_{k,j}$ parameters for the neural network of Fig. 5. Input and bias weights are not used in the calculation of gain parameters.

For the second neuron, there are two δ parameters

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

For the third neuron, there are three δ parameters

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

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

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

For the fourth neuron, there are four δ parameters

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

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

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

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

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

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

In order to organize the process, the $nn \times nn$ computation table is used for calculating signal gains between neurons,

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

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

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

The computation table consists of three parts: weights between neurons in upper triangle; vector of slopes of activation functions in main diagonal; and signal gain matrix δ in lower triangle. Only main diagonal and lower triangular elements are computed for each pattern. Initially, elements on main diagonal $\delta_{k,k} = s_k$ are known, as slopes of the activation functions and values of signal gains $\delta_{k,j}$ are being computed subsequently using (22).

The computation is processed neuron by neuron starting with the neuron closest to network inputs. At first, the row no. 1 is calculated and then elements of the subsequent rows. Calculation on the row below is done using elements from the rows above using (22). After completion of the forward computation process, all elements of the δ matrix in the form of the lower triangle are obtained.

In the next step, elements of gradient vector and Jacobian matrix are calculated using (10) and (12). In the case of neural networks with one output only, the last row of the δ matrix is needed for gradient vector and Jacobian matrix computation. If networks have more outputs no, then last no rows of the δ matrix are used. For example, if the network shown in Fig. 5 has three outputs, the following elements of the δ matrix are used:

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

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

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

where neurons' input vectors $\boldsymbol{y}_1$ through $\boldsymbol{y}$ have six, seven, eight, and nine elements, respectively (Fig. 5), corresponding to number of weights connected. Therefore, each row of the Jacobian matrix has $6 + 7 + 8 + 9 = 30$ elements. If the network has three outputs, then from six elements of the δ matrix and three slopes, 90 elements of Jacobian matrix are calculated. One may notice that the size of newly introduced δ matrix is relatively small, and it is negligible in comparison to other matrixes used in calculation.

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

C. Training Arbitrarily Connected Neural Networks

The proposed computation above was derived for fully connected neural networks. If network is not fully connected, then some elements of the computation table are zero. Fig. 8 shows computation tables for different neural network topologies with six neurons each. Note that the zero elements are for unconnected neurons (in the same layers). This can further simplify the computation process for popular MLP topologies [Fig. 8(b)].

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

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

and

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

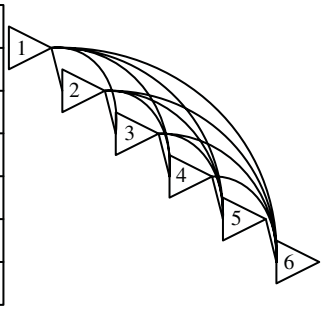
The complete algorithm with forward-only computation is shown in Fig. 9. By adding two additional steps using (25) and (26) (highlighted in bold in Fig. 9), all computation can be completed in the forward-only computing process.

IV. Comparison of the Traditional and the Proposed Algorithm

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

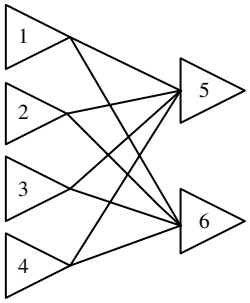
As is shown in Figs. 3 and 9, the cost of traditional computation and the forward-only computation depends on

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



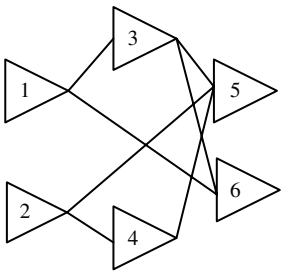
(a)

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



(b)

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



(c)

Fig. 8. Three different architectures with six neurons. (a) FCC network. (b) MLP network. (c) Arbitrarily connected neural network.

```
for all patterns
% Forward computation
    for all neurons (nn)
        for all weights of the neuron (nx)
            calculate net;              % Eq. (4)
        end;
        calculate neuron output;    % Eq. (7)
        calculate neuron slope;     % Eq. (6)
        set current slope as delta;
        for weights connected to previous neurons (ny)
            for previous neurons (nz)
                multiply delta through weights then sum; % Eq. (25)
            end;
            multiply the sum by the slope;              % Eq. (26)
        end;
    end;
    for all outputs (no)
        calculate error;            % Eq. (2)
    end;
end;
```

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

TABLE I

ANALYSIS OF COMPUTATION COST IN LM ALGORITHM

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

*Exponential operation.

the neural network topology. In order to do the analytical comparison, for each neuron let us consider:

1) nx as the average number of weights

$$nx = \frac{nw}{nn}; \tag{27}$$

2) ny as the average number of weights between neurons

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

3) nz as the average number of previous neurons

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

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

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

Table II shows the computation cost for the neural network that will be used for the ASCII problem in Section V, using the equations of Table I.

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

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

Based on the analytical results, it can be seen that, in the LM algorithm for single output networks, the forward-only computation is similar with the traditional computation. while for networks with multiple outputs, the proposed forward-only computation is faster.

V. EXPERIMENTAL RESULTS

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

TABLE II
COMPARISON FOR ASCII PROBLEM

	Hagan and Menhaj computation		Proposed forward-only computation	
	Forward	Backward	Forward	Backward
$+/-$	4088	175 616	7224	0
$\times/\div$	4144	178 752	8848	0
exp	7280	0	7280	0
Total	552 776		32 200	
Relative time	100%		5.83%	

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

TABLE III
ANALYTICAL RELATIVE TIME OF THE FORWARD-ONLY COMPUTATION OF PROBLEMS

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

A. Ability of Handling Various Network Topologies

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

The two-spiral problem is considered as a good evaluation of training algorithms [19]. Depending on neural network architecture, different numbers of neurons are required for successful training. For example, using standard MLP networks with one hidden layer, 34 neurons are required for the two-spiral problem [20]. Using the proposed computation in LM algorithm, two types of topologies, MLP networks with two hidden layers and FCC networks, are tested for training the two-spiral patterns, and the results are presented in the tables below. In MLP networks with two hidden layers, the number of neurons is assumed to be equal in both hidden layers.

Results for MLP architectures shown in the Table IV are identical, whether or not the Hagan and Menhaj LM algorithm or the proposed LM algorithm is used (assuming the same initial weights). In other words, the proposed algorithm has the same success rate and the same number of iterations as those obtained by the Hagan and Menhaj LM algorithm. The difference is that the proposed algorithm can handle also other than MLP architectures and in many cases (especially with multiple outputs) computation time is shorter.

One may notice that the FCC networks are much more efficient than other networks to solve the two-spiral problem, with as few as eight neurons. The proposed LM algorithm is also more efficient than the well-known cascade correlation algorithm, which requires 12–19 hidden neurons in FCC architectures to converge [21].

B. Train Neural Networks with Generalization Abilities

To compare generalization abilities, FCC networks, which proved to be the most efficient in Example 1, are applied for training. These architectures can be trained by both the EBP algorithm and the proposed LM algorithm. The slow convergence of EBP algorithm is not the issue in this experiment. Generalization abilities of networks trained with both algorithms are compared. The Hagan and Menhaj LM algorithm was not used for comparison here because it cannot handle FCC networks.

TABLE IV
TRAINING RESULTS OF THE TWO-SPIRAL PROBLEM WITH THE PROPOSED IMPLEMENTATION OF THE LM ALGORITHM, USING MLP NETWORKS WITH TWO HIDDEN LAYERS; MAXIMUM ITERATION IS 1000; DESIRED ERROR = 0.01; THERE ARE 100 TRIALS FOR EACH CASE

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

TABLE V
TRAINING RESULTS OF THE TWO-SPIRAL PROBLEM WITH THE PROPOSED IMPLEMENTATION OF LM ALGORITHM, USING FCC NETWORKS; MAXIMUM ITERATION IS 1000; DESIRED ERROR = 0.01; THERE ARE 100 TRIALS FOR EACH CASE

Hidden neurons	Success rate (%)	Average number iterations	Average time (s)
7	13	287.7	0.88
8	24	261.4	0.98
9	40	243.9	1.57
10	69	231.8	1.62
11	80	175.1	1.70
12	89	159.7	2.09
13	92	137.3	2.40
14	96	127.7	2.89
15	99	112.0	3.82

Let us consider the peak surface [20] as the required surface [Fig. 10(a)] and let us use equally spaced $10 \times 10 = 100$ patterns [Fig. 10(b)] to train neural networks. The quality of trained networks is evaluated using errors computed for equally spaced $37 \times 37 = 1369$ patterns. In order to make a valid comparison between training and verification errors, the SSE, as defined in (1), is divided by 100 and 1369, respectively.

For EBP algorithm, the learning constant is 0.0005 and momentum is 0.5, maximum iteration is 1 000 000 for EBP algorithm and 1000 for LM algorithm, desired error = 0.5, there are 100 trials for each case. The proposed version of LM algorithm is used in this experiment.

The training results are shown in Table VI. One may notice that it is possible to find the acceptable solution (Fig. 11) with eight 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. 12 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

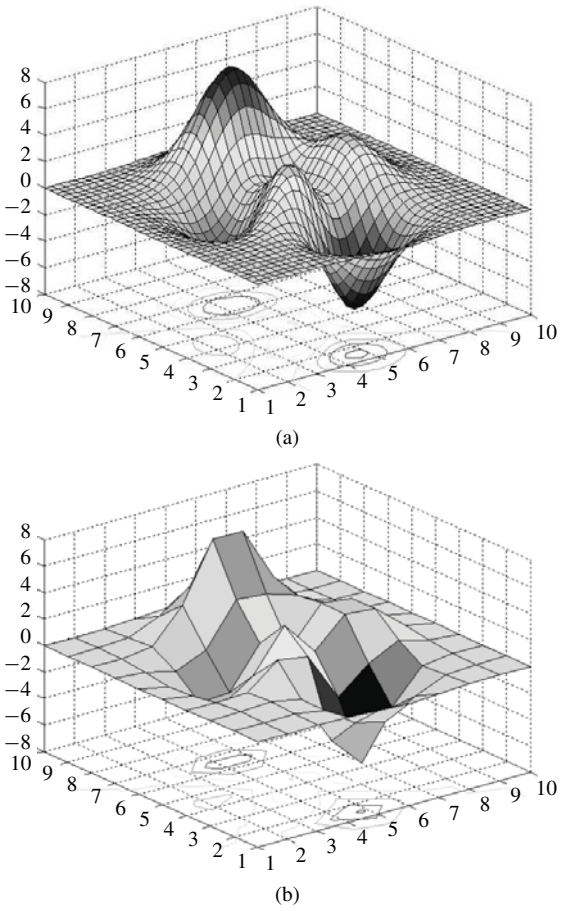


Fig. 10. Surface matching problem. (a) Required 2-D surface with $37 \times 37 = 1369$ points, used for verification. (b) $10 \times 10 = 100$ training patterns extracted in equal space from (a), used for training.

TABLE VI
TRAINING RESULTS OF PEAK SURFACE PROBLEM USING FCC ARCHITECTURES

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

generalization ability to respond correctly for new patterns (between training points). Note that with a larger number of neurons (13 neurons), the EBP algorithm was able to train the network to a small error $\text{SSE}_{Train} = 0.0018$, but as one can see from Fig. 13, the result is unacceptable with verification error $\text{SSE}_{Verify} = 0.4909$.

From the presented examples, one can see that often in simple (close to optimal) networks, the EBP algorithm cannot converge to the required training error (Fig. 12). When the size of networks increases, the EBP algorithm can reach the required training error, but trained networks lose their generalization ability and cannot process new patterns well (Fig. 13). On the other hand, the proposed version of LM algorithm in this paper works not only significantly faster but can also find good solutions with close to optimal networks (Fig. 11).

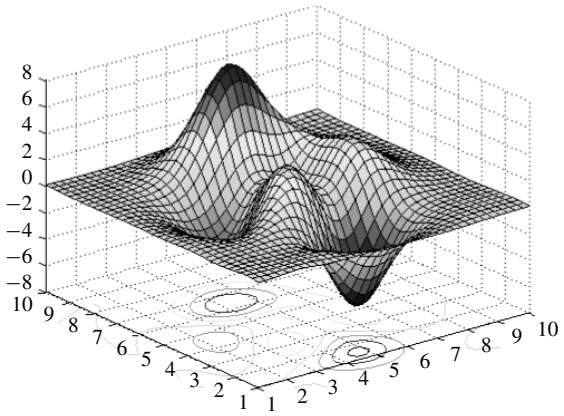


Fig. 11. Best training result in 100 trials, using LM algorithm, eight neurons in FCC network (52 weights); maximum training iteration is 1000; $\text{SSE}_{Train} = 0.0044$, $\text{SSE}_{Verify} = 0.0080$, and training time = 0.37 s.

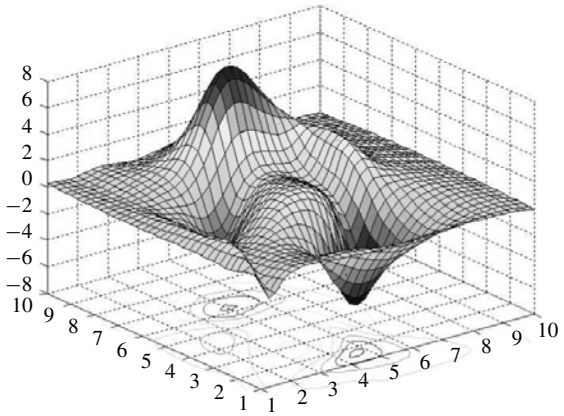


Fig. 12. Best training result in 100 trials, using EBP algorithm, eight neurons in FCC network (52 weights); maximum training iteration is 1 000 000; $\text{SSE}_{Train} = 0.0764$, $\text{SSE}_{Verify} = 0.1271$, and training time = 579.98 s.

C. Computational Speed

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

1) ASCII Codes to Image Conversion: This problem is to associate 256 ASCII codes with 256 character images, each of which is made up of 7×8 pixels (Fig. 14). So there are 8-bit inputs (inputs of parity-8 problem), 256 patterns, and 56 outputs. In order to solve the problem, the structure, i.e., 112 neurons in 8-56-56 MLP network, is used to train those patterns using LM algorithm. The computation time is

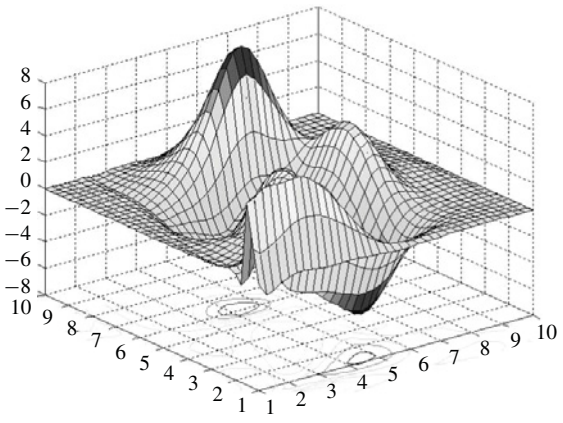


Fig. 13. Best training result in 100 trials, using EBP algorithm, 13 neurons in FCC network (117 weights); maximum training iteration is 1 000 000; $SSE_{Train} = 0.0018$, $SSE_{Verify} = 0.4909$, and training time = 635.72 s.

TABLE VII
COMPARISON FOR ASCII CHARACTERS RECOGNITION PROBLEM

Computation methods	Time cost (ms/iteration)		Relative time (%)
	Forward	Backward	
Traditional	8.24	1028.74	100.0
Forward-only	61.13	0.00	5.9

TABLE VIII
COMPARISON FOR ERROR CORRECTION PROBLEM

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

presented in Table VII. The analytical result is 5.83%, as shown in Table III.

Testing data in Table VII shows that, for this multiple outputs problem, the forward-only computation is much more efficient than traditional computation, in LM training.

2) Error Correction: Error correction is an extension of parity-N problems [14], [15] for multiple parity bits. In Fig. 15, the left side is the input data, made up of signal bits and their parity bits, while the right side is the related corrected signal bits and parity bits as outputs. The number of inputs is equal to the number of outputs.

The error correction problem in the experiment has 8-bit signal and 4-bit parity bits as inputs, 12 outputs, and 3328 patterns (256 correct patterns and 3072 patterns with errors), using 42 neurons in a 12-30-12 MLP network (762 weights). Error patterns with one incorrect bit must be corrected. Both traditional computation and the forward-only computation were performed with the LM algorithm. The testing results are presented in Table VIII. The analytical result is 36.96% as shown in Table III.

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

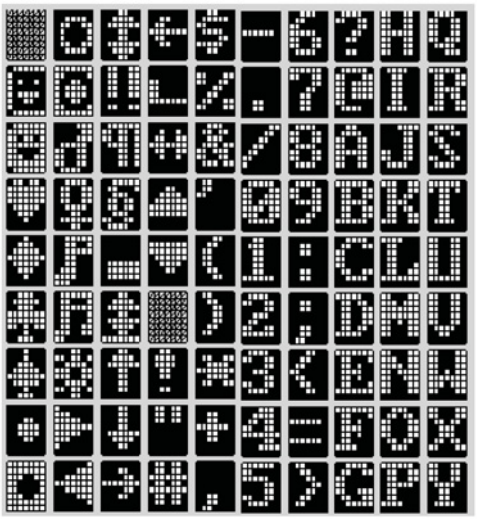


Fig. 14. First 90 images of ASCII characters.

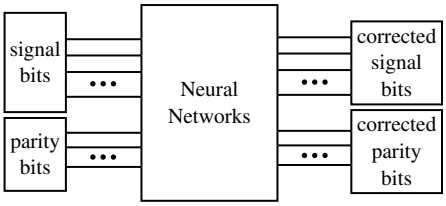


Fig. 15. Using neural networks to solve an error correction problem. Errors in input data can be corrected by well-trained neural networks.

TABLE IX
COMPARISON FOR FORWARD KINEMATICS PROBLEM

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

3) Forward Kinematics: Neural networks are successfully used to solve many practical problems in the industry, such as control problems, compensation nonlinearities in objects and sensors, which are issues of identification of parameters that cannot be directly measured, and sensorless control [22]–[24].

Forward kinematics is an example of these types of practical applications [25], [26]. The purpose is to calculate the position and orientation of robot's end effector as a function of its joint angles.

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

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

VI. Conclusion

One of the major features of the proposed algorithm is that it can be easily adapted to train arbitrarily connected neural networks and not just MLP topologies. This is very important because neural networks with connections across layers are much more powerful than commonly used MLP architectures [14], [27], [15]. For example, if the number of neurons in the network is limited to eight, then popular MLP topology with one hidden layer is capable of solving only parity-7 problem. If the same eight neurons are connected in FCC, then with this network parity-255 problem can be solved [27].

It was shown (Figs. 12 and 13) that, in order to secure training convergence with first-order algorithms, an excessive number of neurons much be used, and this results in a failure of the generalization abilities of the neural network. This was the major reason for frustration in industrial practice when neural networks were trained to small errors but would respond very poorly for patterns not used for training. The presented computation for second-order algorithms can be applied to train arbitrarily connected neural networks, so it is capable of training neural networks with reduced number of neurons and as consequence has good generalization abilities (Fig. 11).

The proposed method of computation gives identical number of training iterations and success rates as the Hagan and Menhaj implementation of the LM algorithm, since the same Jacobian matrices are obtained from both methods. By removing backpropagation process, the proposed method is much simpler than traditional forward and backward procedure to calculate the elements of the Jacobian matrix. The whole computation can be described by a regular table (Fig. 7) and a general formula (22). Additionally, for networks with multiple outputs, the proposed method is less computationally intensive and faster than traditional forward and backward computations [12], [2].

The algorithm was implemented in the neural networks trainer (NBN 2.10), and the software can be downloaded from http://www.eng.auburn.edu/users/wilambm/nnt/.

References

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

[2] 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.

[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] 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.

[5] 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.

[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] Q. Song, J. C. Spall, Y. C. Soh, and J. Ni, "Robust neural network tracking controller using simultaneous perturbation stochastic approximation," *IEEE Trans. Neural Netw.*, vol. 19, no. 5, pp. 817–835, May 2008.

[8] 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.

[9] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in *Proc. Int. Conf. Neural Netw.*, San Francisco, CA, 1993, pp. 586–591.

[10] A. Toledo, M. Pinzolas, J. J. Ibarrola, and G. Lera, "Improvement of the neighborhood based Levenberg–Marquardt algorithm by local adaptation of the learning coefficient," *IEEE Trans. Neural Netw.*, vol. 16, no. 4, pp. 988–992, Jul. 2005.

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

[12] 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.

[13] H. B. Demuth and M. Beale, *Neural Network Toolbox: For Use with MATLAB*. Natick, MA: Mathworks, 2000.

[14] M. E. Hohil, D. Liu, and S. H. Smith, "Solving the *N*-bit parity problem using neural networks," *Neural Netw.*, vol. 12, no. 9, pp. 1321–1323, Nov. 1999.

[15] B. M. Wilamowski, D. Hunter, and A. Malinowski, "Solving parity-N problems with feedforward neural networks," in *Proc. IEEE IJCNN*, Piscataway, NJ: IEEE Press, 2003, pp. 2546–2551.

[16] B. M. Wilamowski and H. Yu, "Improved computation for Levenberg–Marquardt training," *IEEE Trans. Neural Netw.*, vol. 21, no. 6, pp. 930–937, Jun. 2010.

[17] R. Hecht-Nielsen, "Theory of the back propagation neural network," in *Proc. IEEE IJCNN*, Washington D.C., Jun. 1989, pp. 593–605.

[18] K. Levenberg, "A method for the solution of certain problems in least squares," *Quart. Appl. Math.*, vol. 2, no. 2, pp. 164–168, 1944.

[19] 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.

[20] S. Wan and L. E. Banta, "Parameter incremental learning algorithm for neural networks," *IEEE Trans. Neural Netw.*, vol. 17, no. 6, pp. 1424–1438, Nov. 2006.

[21] S. E. Fahlman and C. Lebiere, "The cascade-correction learning architecture," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990.

[22] B. K. Bose, "Neural network applications in power electronics and motor drives—An introduction and perspective," *IEEE Trans. Ind. Electron.*, vol. 54, no. 1, pp. 14–33, Feb. 2007.

[23] 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.

[24] W. Qiao, R. G. Harley, and G. K. Venayagamoorthy, "Fault-tolerant indirect adaptive neurocontrol for a static synchronous series compensator in a power network with missing sensor measurements," *IEEE Trans. Neural Netw.*, vol. 19, no. 7, pp. 1179–1195, Jul. 2008.

[25] J. Y. Goulermas, A. H. Findlow, C. J. Nester, P. Liatsis, X.-J. Zeng, L. Kenney, P. Tresadern, S. B. Thies, and D. Howard, "An instance-based algorithm with auxiliary similarity information for the estimation of gait kinematics from wearable sensors," *IEEE Trans. Neural Netw.*, vol. 19, no. 9, pp. 1574–1582, Sep. 2008.

[26] Y. Zhang and S. S. Ge, "Design and analysis of a general recurrent neural network model for time-varying matrix inversion," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1477–1490, Nov. 2005.

[27] B. M. Wilamowski, "Challenges in applications of computational intelligence in industrials electronics," in *Proc. IEEE Int. Symp. Ind. Electron.*, Bari, Italy, Jul. 2010, pp. 15–22.

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, in 1966, 1970, and 1977, respectively.

He is currently the Director of the Alabama Micro/Nano Science and Technology Center, Auburn University, Auburn, AL, and a Professor with the Department of Electrical and Computer Engineering. He is the author of four textbooks and about 300 refereed publications and holds 28 patents.

He was the Major Professor for over 150 graduate students. His current research interests include computational intelligence and soft computing, computer-aided design development, solid-state electronics, mixed- and analog-signal processing, and network programming.

Dr. Wilamowski was one of the founders of the IEEE Computational Intelligence Society and the President of the IEEE Industrial Electronics Society. He served as an Associate Editor for six journals including the IEEE TRANSACTIONS ON NEURAL NETWORKS. He also was the Editor-in-Chief of the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS. Recently, he was appointed Editor-in-Chief of the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS.

Hao Yu received the M.S. degree in electrical engineering from Huazhong University of Science and Technology, Hubei, China, in 2006. He is currently working toward the Ph.D. degree in electrical engineering in Auburn University, Auburn, AL.

He is a Research Assistant with the Department of Electrical and Computer Engineering, Auburn University. His current research interests include computational intelligence, neural networks, and computer aided design.

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