13

NBN Algorithm

Bogdan M. Wilamoswki
Auburn University

Hao Yu
Auburn University

Nicholas Cotton
Auburn University

13.1 Introduction

Since the development of EBP—error backpropagation—algorithm for training neural networks, many attempts were made to improve the learning process. There are some well-known methods like momentum or variable learning rate and there are less known methods which significantly accelerate learning rate [WT93,AW95,WCM99,W09,WH10]. The recently developed NBN (neuron-by-neuron) algorithm [WCHK07,WCKD08,YW09] is very efficient for neural network training. Comparing with the well-known Levenberg–Marquardt algorithm (introduced in Chapter 12) [L44,M63], the NBN algorithm has several advantages: (1) the ability to handle arbitrarily connected neural networks; (2) forward-only computation (without backpropagation process); and (3) direct computation of quasi-Hessian matrix (no need to compute and store Jacobian matrix). This chapter is organized around the three advantages of the NBN algorithm.

13.2 Computational Fundamentals

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

- p is the index of patterns, from 1 to np, where np is the number of patterns.
- m is the index of outputs, from 1 to no, where no is the number of outputs.

- j and k are the indices of neurons, from 1 to nn, where nn is the number of neurons.
- i is the index of neuron inputs, from 1 to ni, where ni is the number of inputs and it may vary for different neurons.

Other indices will be explained in related places.

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

$$E = \frac{1}{2}\sum_{p=1}^{np}\sum_{m=1}^{no} e_{p,m}^2 \tag{13.1}$$

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

$$e_{p,m} = o_{p,m} - d_{p,m} \tag{13.2}$$

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

In all algorithms, besides the NBN algorithm, 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 following derivations, unless it is essential.

13.2.1 Definition of Basic Concepts in Neural Network Training

Let us consider neuron j with ni inputs, as shown in Figure 13.1. 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 network inputs if connections across layers are allowed.

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 chapter, if node y has one index (neuron), then it is used as a neuron output node; while if it has two indices (neuron and input), it is a neuron input node.

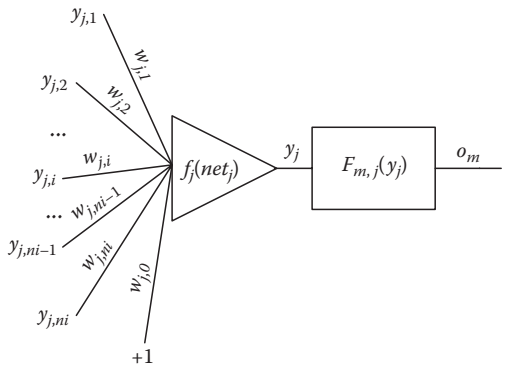


FIGURE 13.1 Connection of a neuron j with the rest of the network. Nodes $y_{j,i}$ could represent network inputs or outputs of other neurons. $F_{m,j}(y_j)$ is the nonlinear relationship between the neuron output node y_j and the network output o_m.

Output node of neuron j is calculated using

$$y_j = f_j(net_j) \tag{13.3}$$

where

f_j is the activation function of neuron j

net value net_j is the sum of weighted input nodes of neuron j:

$$net_j = \sum_{i=1}^{ni} w_{j,i} y_{j,i} + w_{j,0} \tag{13.4}$$

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 (13.4) one may notice that derivative of net_j is

$$\frac{\partial net_j}{\partial w_{j,i}} = y_{j,i} \tag{13.5}$$

and slope s_j of activation function f_j is

$$s_j = \frac{\partial y_j}{\partial net_j} = \frac{\partial f_j(net_j)}{\partial net_j} \tag{13.6}$$

Between the output node y_j of a hidden neuron j and network output o_m, there is a complex nonlinear relationship (Figure 13.1):

$$o_m = F_{m,j}(y_j) \tag{13.7}$$

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.

13.2.2 Jacobian Matrix Computation

The update rule of Levenberg–Marquardt algorithm is [TM94]

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \left(\mathbf{J}_n^T \mathbf{J}_n + \mu \mathbf{I}\right)^{-1} \mathbf{J}_n \mathbf{e}_n \tag{13.8}$$

where

n is the index of iterations

μ is the combination coefficient

$\mathbf{I}$ is the identity matrix

$\mathbf{J}$ is the Jacobian matrix (Figure 13.2)

From Figure 13.2, 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 $np \times no$.

$$
J = \begin{array}{c}
\begin{array}{cccccc} \text{neuron 1} & & \cdots & \text{neuron } j & & \cdots \end{array} \\
\begin{bmatrix}
\frac{\partial e_{1,1}}{\partial w_{1,1}} & \frac{\partial e_{1,1}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{1,1}}{\partial w_{j,1}} & \frac{\partial e_{1,1}}{\partial w_{j,2}} & \cdots \\
\frac{\partial e_{1,2}}{\partial w_{1,1}} & \frac{\partial e_{1,2}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{1,2}}{\partial w_{j,1}} & \frac{\partial e_{1,2}}{\partial w_{j,2}} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\frac{\partial e_{1,no}}{\partial w_{1,1}} & \frac{\partial e_{1,no}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{1,no}}{\partial w_{j,1}} & \frac{\partial e_{1,no}}{\partial w_{j,2}} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\frac{\partial e_{p,1}}{\partial w_{1,1}} & \frac{\partial e_{p,1}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{p,1}}{\partial w_{j,1}} & \frac{\partial e_{p,1}}{\partial w_{j,2}} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\frac{\partial e_{p,m}}{\partial w_{1,1}} & \frac{\partial e_{p,m}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{p,m}}{\partial w_{j,1}} & \frac{\partial e_{p,m}}{\partial w_{j,2}} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\frac{\partial e_{np,1}}{\partial w_{1,1}} & \frac{\partial e_{np,1}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{np,1}}{\partial w_{j,1}} & \frac{\partial e_{np,1}}{\partial w_{j,2}} & \cdots \\
\frac{\partial e_{np,1}}{\partial w_{1,1}} & \frac{\partial e_{np,2}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{np,2}}{\partial w_{j,1}} & \frac{\partial e_{np,2}}{\partial w_{j,2}} & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\frac{\partial e_{np,no}}{\partial w_{1,1}} & \frac{\partial e_{np,no}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{np,no}}{\partial w_{j,1}} & \frac{\partial e_{np,no}}{\partial w_{j,2}} & \cdots
\end{bmatrix}
\end{array}
\begin{array}{ll}
m = 1 & \\
m = 2 & p = 1 \\
\cdots & \\
m = no & \\
\cdots & \cdots \\
m = 1 & \\
\cdots & p = p \\
m = m & \\
\cdots & \cdots \\
m = 1 & \\
m = 2 & p = np \\
\cdots & \\
m = no &
\end{array}
$$

FIGURE 13.2 Structure of the Jacobian matrix: (1) the number of columns is equal to the number of weights and (2) each row corresponds to a specified training pattern p and output m.

The elements of Jacobian matrix can be calculated by

$$\frac{\partial e_m}{\partial w_{j,i}} = \frac{\partial e_m}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{j,i}} \tag{13.9}$$

By combining with (13.2), (13.5), (13.6), and (13.7), (13.9) can be written as

$$\frac{\partial e_m}{\partial w_{j,i}} = y_{j,i} s_j F'_{m,j} \tag{13.10}$$

In second-order algorithms, the parameter δ [N89,TM94] is defined to measure the EBP process, as

$$\delta_{m,j} = s_j F'_{m,j} \tag{13.11}$$

By combining (13.10) and (13.11), elements of Jacobian matrix can be calculated by

$$\frac{\partial e_m}{\partial w_{j,i}} = y_{j,i}\delta_{m,j} \tag{13.12}$$

Using (13.12), in backpropagation process, the error can be replaced by a unit value "1."

13.3 Training Arbitrarily Connected Neural Networks

The NBN algorithm introduced in this chapter is developed for training arbitrarily connected neural networks using Levenberg–Marquardt update rule. Instead of layer-by-layer computation (introduced in Chapter 12), the NBN algorithm does the forward and backward computation based on NBN routings [WCHK07], which makes it suitable for arbitrarily connected neural networks.

13.3.1 Importance of Training Arbitrarily Connected Neural Networks

The traditional implementation of Levenberg–Marquardt algorithm [TM94], like MATLAB® neural network toolbox (MNNT), was adopted only for standard MLP (multilayer perceptron) networks, it turns out that the MLP networks are not efficient. AQ1

Figure 13.3 shows the smallest structures to solve parity-7 problem. The standard MLP network with one hidden layer (Figure 13.3a) needs at least eight neurons to find the solution. The BMLP (bridged multiplayer perceptron) network (Figure 13.3b) can solve the problem with four neurons. The FCC (fully connected cascade) network (Figure 13.3c) is the most powerful one, and it only requires three neurons to get the solutions. One may notice that the last two types of networks are better choices for efficient training, but they also require more challenging computation.

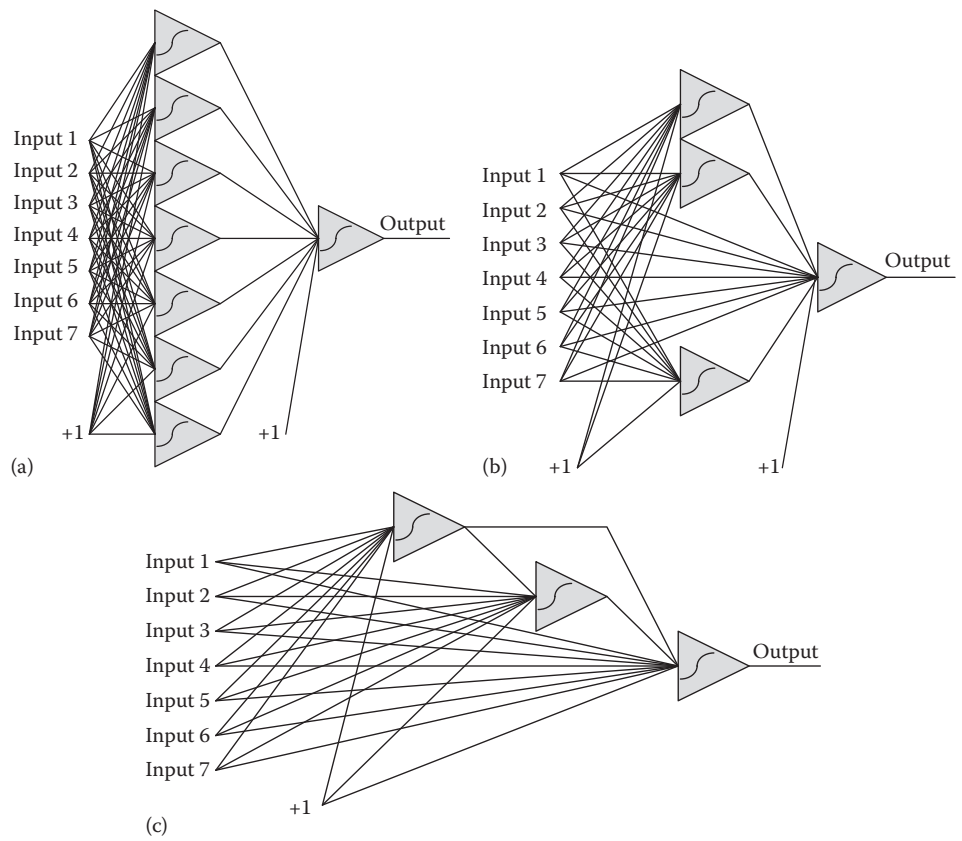


FIGURE 13.3 Smallest structures for solving parity-7 problem: (a) standard MLP network (64 weights), (b) BMLP network (35 weights), and (c) FCC network (27 weights).

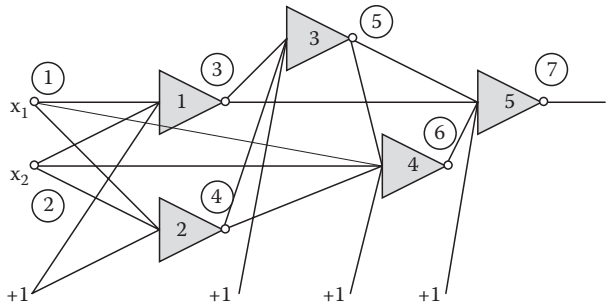


FIGURE 13.4 Five neurons in arbitrarily connected network.

13.3.2 Creation of Jacobian Matrix for Arbitrarily Connected Neural Networks

In this section, the NBN algorithm for calculating the Jacobian matrix for arbitrarily connected feedforward neural networks is presented. The rest of the computations for weight updating follow the LM algorithm, as shown in Equation 13.9.

In the forward computation, neurons are organized according to the direction of signal propagation, while in the backward computation, the analysis will follow the backpropagation procedures.

Let us consider the arbitrarily connected network with one output, as shown in Figure 13.4.

For the network in Figure 13.4, using the NBN algorithm, the network topology can be described similarly in SPICE program:

```
n1  [model]  3 1 2
n2  [model]  4 1 2
n3  [model]  5 3 4
n4  [model]  6 1 2 4 5
n5  [model]  7 3 5 6
```

Notice that each line corresponds to one neuron. The first part (n_1–n_5) is the neuron name (Figure 13.4). The second part "*[model]*" is the neuron models, such as bipolar, unipolar, and linear. Models are declared in separate lines where the types of activation functions and the neuron gains are specified. The first digit in each line after the neuron model indicates the network nodes starting with the output node of the neuron, followed with its input nodes.

Please notice that neurons must be ordered from inputs to neuron outputs. It is important that, for each given neuron, the neuron inputs must have smaller indices than its output.

The row elements of the Jacobian matrix for a given pattern are being computed in the following three steps [WCKD08]:

1. Forward computation
2. Backward computation
3. Jacobian element computation

13.3.2.1 Forward Computation

In the forward computation, the neurons connected to the network inputs are first processed so that their outputs can be used as inputs to the subsequent neurons. The following neurons are then processed as their input values become available. In other words, the selected computing sequence has to follow the concept of feedforward signal propagation. If a signal reaches the inputs of several neurons at the same time, then these neurons can be processed in any sequence. In the example in Figure 13.4, there are

two possible ways in which neurons can be processed in the forward direction: $n_1n_2n_3n_4n_5$ or $n_2n_1n_3n_4n_5$. The two procedures will lead to different computing processes but with exactly the same results. When the forward pass is concluded, the following two temporary vectors are stored: the first vector **y** with the values of the signals on the neuron output nodes and the second vector **s** with the values of the slopes of the neuron activation functions, which are signal dependent.

13.3.2.2 Backward Computation

The sequence of the backward computation is opposite to the forward computation sequence. The process starts with the last neuron and continues toward the input. In the case of the network in Figure 13.4, the following are two possible sequences (backpropagation paths): $n_5n_4n_3n_2n_1$ or $n_5n_4n_3n_1n_2$, and also they will have the same results. To demonstrate the case, let us use the $n_5n_4n_3n_2n_1$ sequence. The vector δ represents signal propagation from a network output to the inputs of all other neurons. The size of this vector is equal to the number of neurons.

For the output neuron n_5, its sensitivity is initialed using its slope $\delta_{1,5} = s_5$. For neuron n_4, the delta at n_5 will be propagated by w_{45}—the weight between n_4 and n_5, then by the slope of neuron n_4. So the delta parameter of n_4 is presented as $\delta_{1,4} = \delta_{1,5}w_{45}s_4$. For neuron n_3, the delta parameters of n_4 and n_5 will be propagated to the output of neuron n_3 and summed, then multiplied by the slope of neuron n_3, as $\delta_{1,3} = (\delta_{1,5}w_{35} + \delta_{1,4}w_{34})s_3$. For the same procedure, it could be obtained that $\delta_{1,2} = (\delta_{1,3}w_{23} + \delta_{1,4}w_{24})s_2$ and $\delta_{1,1} = (\delta_{1,3}w_{13} + \delta_{1,5}w_{15})s_1$. After the backpropagation process is done at neuron $N1$, all the elements of array δ are obtained.

13.3.2.3 Jacobian Element Computation

After the forward and backward computation, all the neuron outputs **y** and vector δ are calculated. Then using Equation 13.12, the Jacobian row for a given pattern can be obtained.

By applying all training patterns, the whole Jacobian matrix can be calculated and stored.

For arbitrarily connected neural networks, the NBN algorithm for Jacobian matrix computation can be organized as shown in Figure 13.5.

```
for all patterns (np)
% Forward computation
 for all neurons (nn)
  for all weights of the neuron (nx)
   calculate net;         % Eq. (4)
  end;
  calculate neuron output;  % Eq. (3)
  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;
  related Jacobian row computation;  %Eq. (12)
 end;
end;
```

FIGURE 13.5 Pseudo code using NBN algorithm for Jacobian matrix computation

13.3.3 Solve Problems with Arbitrarily Connected Neural Networks

13.3.3.1 Function Approximation Problem

Function approximation is usually used in nonlinear control realm of neural networks, for control surface prediction. In order to approximate the function shown below, 25 points are selected from 0 to 4 as the training patterns. With only four neurons in FCC networks (as shown in Figure 13.6), the training result is presented in Figure 13.7.

$$z = 4exp\left(-0.15(x-4)^2 - 0.5(y-3)^2\right) + 10^{-9} \tag{13.13}$$

13.3.3.2 Two-Spiral Problem

Two-spiral problem is considered as a good evaluation of both training algorithms and training architectures [AS99]. Depending on the 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 two-spiral problem [PLI08]; while with the FCC architecture, it can be solved with only eight neurons using the NBN algorithm. NBN algorithms are not only much faster but also can train reduced size networks which cannot be handled by the traditional EBP algorithm (see Table 13.1).

For EBP algorithm, learning constant is 0.005 (largest possible to avoid oscillation) and momentum
AQ2 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.

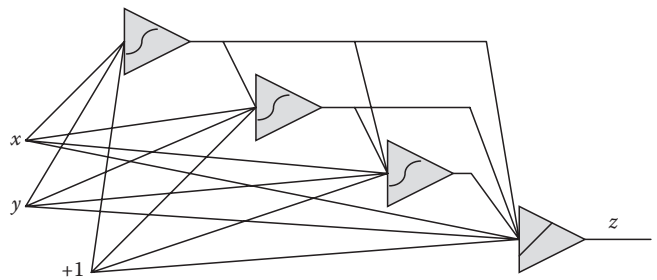


FIGURE 13.6 Network used for training the function approximation problem; notice the output neuron is a linear neuron with gain =1.

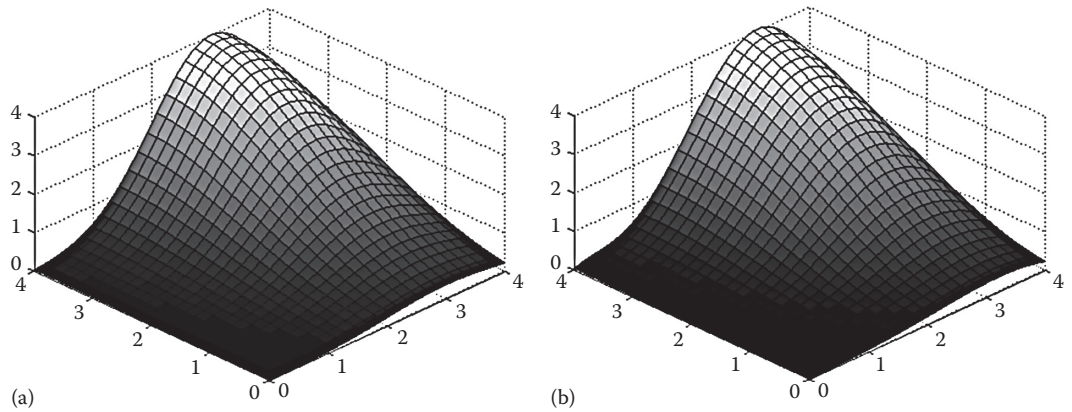


FIGURE 13.7 Averaged SSE between desired surface (a) and neural prediction (b) is 0.0025.

TABLE 13.1 Training Results of Two-Spiral Problem

	Success Rate (%)		Average Number of Iterations		Average Time (s)	
Neurons	EBP	NBN	EBP	NBN	EBP	NBN
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	633.91	1.70
13	85	89	335,531	159.7	620.30	2.09
14	92	92	266,237	137.3	605.32	2.40

13.4 Forward-Only Computation

The NBN procedure introduced in Section 3 requires both forward and backward computation. Especially, as shown in Figure 13.5, one may notice that for networks with multiple outputs, the backpropagation process has to be repeated for each output.

In this section, an improved NBN computation is introduced to overcome the problem, by removing backpropagation process in the computation of Jacobian matrix.

13.4.1 Derivation

The concept of $\delta_{m,j}$ was described in Section 13.2. 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 (Figures 13.8 and 13.10). The notation of $\delta_{k,j}$ is an extension of Equation 13.11 and can be interpreted as signal gain between neurons j and k, and it is given by

$$\delta_{k,j} = \frac{\partial F_{k,j}(y_j)}{\partial net_j} = \frac{\partial F_{k,j}(y_j)}{\partial y_j}\frac{\partial y_j}{\partial net_j} = F'_{k,j}s_j \tag{13.14}$$

where

k and j are indices of neurons

$F_{k,j}(y_j)$ is the nonlinear relationship between the output node of neuron k and the output node of neuron j

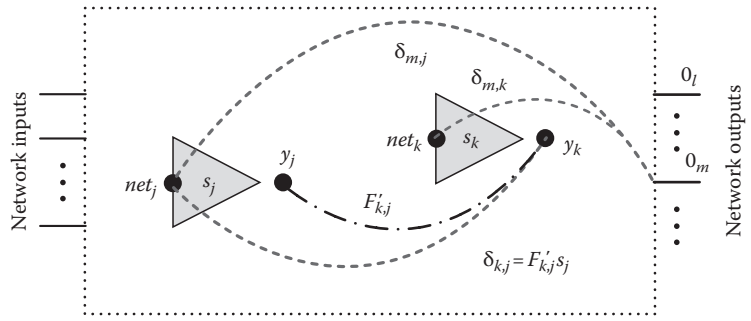


FIGURE 13.8 Interpretation of $\delta_{k,j}$ as a signal gain, where in feedforward network neuron j must be located before neuron k.

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 calculated by Equation 13.6. Figure 13.8 illustrates this extended concept of $\delta_{k,j}$ parameter as a signal gain.

The matrix $\boldsymbol{\delta}$ has a triangular shape and its elements can be calculated in the forward-only process. Later, elements of Jacobian can be obtained using Equation 13.12, where only last rows of matrix $\boldsymbol{\delta}$ associated with network outputs are used. The key issue of the proposed algorithm is the method of calculating of $\delta_{k,j}$ parameters in the forward calculation process, and it will be described in the next part of this section.

13.4.2 Calculation of δ Matrix for FCC Architectures

Let us start our analysis with fully connected neural networks (Figure 13.9). Any other architecture could be considered as a simplification of fully connected neural networks by eliminating connections (setting weights to zero). If 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 the form of δ parameter as $\delta_{j,j} = s_j$. By inspecting Figure 13.10, δ parameters can be written as

For the first neuron, there is only one δ parameter

$$\delta_{1,1} = s_1 \tag{13.15}$$

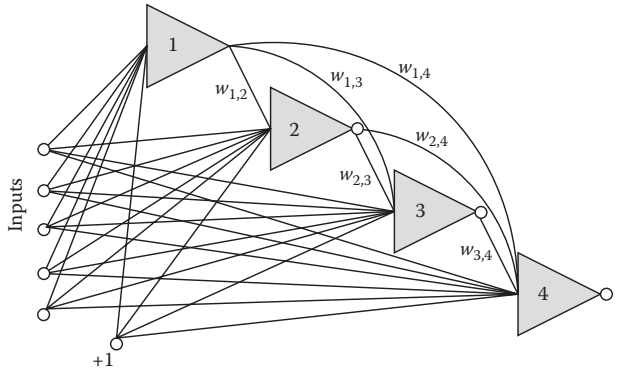


FIGURE 13.9 Four neurons in fully connected neural network, with five inputs and three outputs.

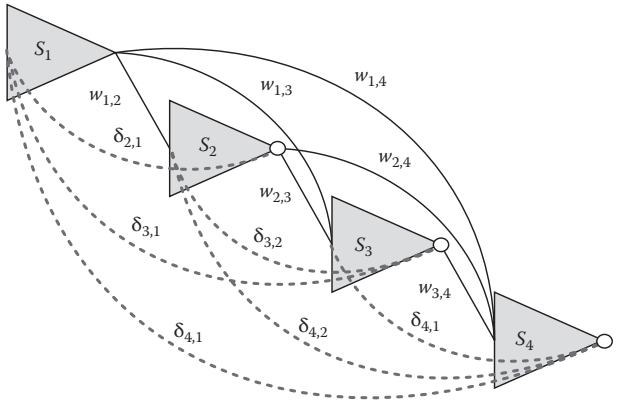


FIGURE 13.10 The $\delta_{k,j}$ parameters for the neural network of Figure 13.9. Input and bias weights are not used in the calculation of gain parameters.

For the second neuron, there are two δ parameters

$$\begin{aligned}\delta_{2,2} &= s_2\\ \delta_{2,1} &= s_2 w_{1,2} s_1\end{aligned} \tag{13.16}$$

For the third neuron, there are three δ parameters

$$\begin{aligned}\delta_{3,3} &= s_3\\ \delta_{3,2} &= s_3 w_{2,3} s_2\\ \delta_{3,1} &= s_3 w_{1,3} s_1 + s_3 w_{2,3} s_2 w_{1,2} s_1\end{aligned} \tag{13.17}$$

One may notice that all δ parameters for the third neuron can be also expressed as a function of δ parameters calculated for previous neurons. Equations 13.17 can be rewritten as

$$\begin{aligned}\delta_{3,3} &= s_3\\ \delta_{3,2} &= \delta_{3,3} w_{2,3} \delta_{2,2}\\ \delta_{3,1} &= \delta_{3,3} w_{1,3} \delta_{1,1} + \delta_{3,3} w_{2,3} \delta_{2,1}\end{aligned} \tag{13.18}$$

For the fourth neuron, there are four δ parameters

$$\begin{aligned}\delta_{4,4} &= s_4\\ \delta_{4,3} &= \delta_{4,4} w_{3,4} \delta_{3,3}\\ \delta_{4,2} &= \delta_{4,4} w_{2,4} \delta_{2,2} + \delta_{4,4} w_{3,4} \delta_{3,2}\\ \delta_{4,1} &= \delta_{4,4} w_{1,4} \delta_{1,1} + \delta_{4,4} w_{2,4} \delta_{2,1} + \delta_{4,4} w_{3,4} \delta_{3,1}\end{aligned} \tag{13.19}$$

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

$$\delta_{4,1} = \delta_{4,4} \sum_{i=1}^{3} w_{i,4} \delta_{i,1} \tag{13.20}$$

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

$$\delta_{k,j} = \delta_{k,k} \sum_{i=j}^{k-1} w_{i,k} \delta_{i,j} \tag{13.21}$$

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

In order to organize the process, the $nn \times nn$ computation table is for calculating signal gains between neurons, where nn is the number of neurons (Figure 13.11). Natural indices (from 1 to nn) are given for each neuron according to the direction of signal propagation. For signal gain 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 Figure 13.11. One may notice that the indices of rows and columns are the same as the indices of neurons. In the following derivation, let us use k and j used as

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	$\delta_{nn,1}$	$\delta_{nn,2}$	...	$\delta_{nn,j}$	...	$\delta_{nn,k}$	...	$\delta_{nn,nn}$

FIGURE 13.11 The $nn \times nn$ computation table; gain matrix $\boldsymbol{\delta}$ contains all the signal gains between neurons; weight array $\boldsymbol{w}$ presents only the connections between neurons, while network input weights and biasing weights are not included.

neuron indices to specify the rows and columns in the computation table. In feedforward network, $k \geq j$ and matrix $\boldsymbol{\delta}$ 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 $\boldsymbol{\delta}$ in lower triangle. Only main diagonal and lower triangular elements are computed for each pattern. Initially, elements on main diagonal $\delta_{k,k} = s_k$ are known as slopes of activation functions and values of signal gains $\delta_{k,j}$ are being computed subsequently using Equation 13.21.

The computation is being processed NBN starting with the neuron closest to network inputs. At first, the row number one is calculated and then elements of subsequent rows. Calculation on row below is done using elements from above rows using Equation 13.21. After completion of forward computation process, all elements of $\boldsymbol{\delta}$ matrix in the form of the lower triangle are obtained.

In the next step, elements of Jacobian matrix are calculated using Equation 13.12. In the case of neural networks with one output, only the last row of $\boldsymbol{\delta}$ matrix is needed for gradient vector and Jacobian matrix computation. If networks have more outputs *no*, then last *no* rows of $\boldsymbol{\delta}$ matrix are used. For example, if the network shown in Figure 13.9 has three outputs, the following elements of $\boldsymbol{\delta}$ matrix are used

$$\begin{bmatrix} \delta_{2,1} & \delta_{2,2} = s_2 & \delta_{2,3} = 0 & \delta_{2,4} = 0 \\ \delta_{3,1} & \delta_{3,2} & \delta_{3,3} = s_3 & \delta_{3,4} = 0 \\ \delta_{4,1} & \delta_{4,2} & \delta_{4,3} & \delta_{4,4} = s_4 \end{bmatrix} \tag{13.22}$$

and then for each pattern, the three rows of Jacobian matrix, corresponding to three outputs, are calculated in one step using Equation 13.12 without additional propagation of δ

$$\begin{bmatrix} \delta_{2,1} \times \{\boldsymbol{y}_1\} & s_2 \times \{\boldsymbol{y}_2\} & 0 \times \{\boldsymbol{y}_3\} & 0 \times \{\boldsymbol{y}_4\} \\ \delta_{3,1} \times \{\boldsymbol{y}_1\} & \delta_{3,2} \times \{\boldsymbol{y}_2\} & s_3 \times \{\boldsymbol{y}_3\} & 0 \times \{\boldsymbol{y}_4\} \\ \underbrace{\delta_{4,1} \times \{\boldsymbol{y}_1\}}_{\text{neuron 1}} & \underbrace{\delta_{4,2} \times \{\boldsymbol{y}_2\}}_{\text{neuron 2}} & \underbrace{\delta_{4,3} \times \{\boldsymbol{y}_3\}}_{\text{neuron 3}} & \underbrace{s_4 \times \{\boldsymbol{y}_4\}}_{\text{neuron 4}} \end{bmatrix} \tag{13.23}$$

where neurons' input vectors $\boldsymbol{y}_1$ through $\boldsymbol{y}_4$ have 6, 7, 8, and 9 elements respectively (Figure 13.9), corresponding to number of weights connected. Therefore, each row of Jacobian matrix has 6 + 7 + 8 + 9 = 30 elements. If the network has three outputs, then from six elements of $\boldsymbol{\delta}$ matrix and 3 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 with other matrices used in calculation.

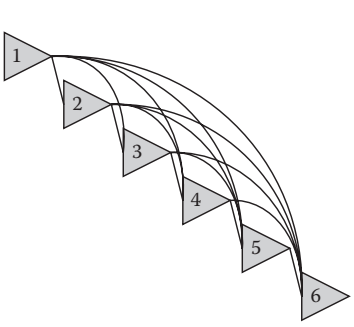
The improved NBN procedure gives all the information needed to calculate Jacobian matrix (13.12), without backpropagation process; instead, δ parameters are obtained in relatively simple forward computation (see Equation 13.21).

13.4.3 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. Figure 13.12 shows computation

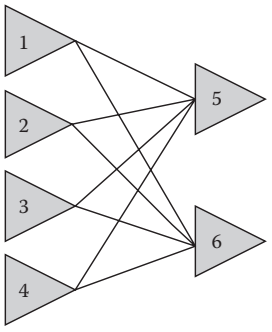
Index	1	2	3	4	5	6
1	s_1	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	$w_{1,6}$
2	$\delta_{2,1}$	s_2	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	$w_{2,6}$
3	$\delta_{3,1}$	$\delta_{3,2}$	s_3	$w_{3,4}$	$w_{3,5}$	$w_{3,6}$
4	$\delta_{4,1}$	$\delta_{4,2}$	$\delta_{4,3}$	s_4	$w_{4,5}$	$w_{4,6}$
5	$\delta_{5,1}$	$\delta_{5,2}$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

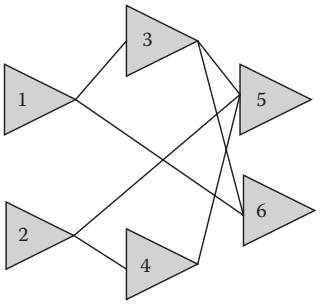


FIGURE 13.12 Three different architectures with six neurons: (a) FCC network, (b) MLP network, and (c) arbitrarily connected neural network.

tables for different neural network topologies with six neurons each. Please notice zero elements are for not connected neurons (in the same layers). This can further simplify the computation process for popular MLP topologies (Figure 13.12b).

Most of used neural networks have many zero elements in the computation table (Figure 13.12). In order to reduce the storage requirements (do not store weights with zero values) and to reduce computation process (do not perform operations on zero elements), a part of the NBN algorithm in Section 13.3 was adopted for forward computation.

In order to further simplify the computation process, Equation 13.21 is completed in two steps

$$x_{k,j} = \sum_{i=j}^{k-1} w_{i,k}\delta_{i,j} \tag{13.24}$$

and

$$\delta_{k,j} = \delta_{k,k}x_{k,j} = s_k x_{k,j} \tag{13.25}$$

The complete algorithm with forward-only computation is shown in Figure 13.13. By adding two additional steps using Equations 13.24 and 13.25 (highlighted in bold in Figure 13.13), all computation can be completed in the forward-only computing process.

13.4.4 Experimental Results

Several problems are presented to test the computing speed of two different NBN algorithms—with and without backpropagation process.

The testing of time costs for both the backpropagation computation and the forward-only computation are divided into forward part and backward part separately.

13.4.4.1 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 (Figure 13.14). So there are 8 bit inputs (inputs of parity-8 problem), 256 patterns, and

```
for all patterns (np)
% Forward computation
 for all neurons (nn)
  for all weights of the neuron (nx)
   calculate net;              % Eq. (4)
  end;
  calculate neuron output;       % Eq. (3)
  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. (24)
    end;
    multiply the sum by the slope;          % Eq. (25)
  end;
  related Jacobian elements computation;    % Eq. (12)
 end;
 for all outputs (no)
  calculate error;     % Eq. (2)
 end;
end;
```

FIGURE 13.13 Pseudo code of the forward-only computation, in second-order algorithms.

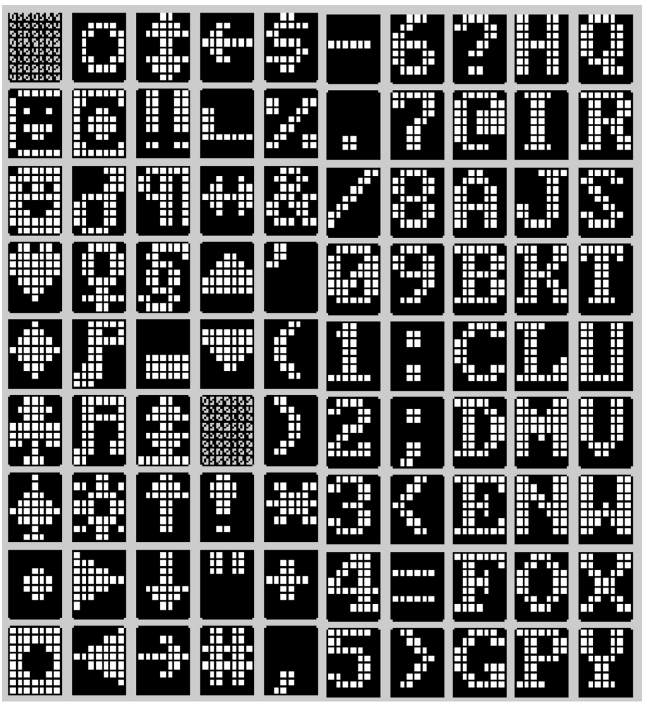


FIGURE 13.14 The first 90 images of ASCII characters.

TABLE 13.2 Comparison for ASCII Character Recognition Problem

Computation Methods	Time Cost (ms/Iteration)		Relative Time (%)
	Forward	Backward	
Backpropagation	8.24	1,028.74	100
Forward-only	61.13	0.00	5.9

56 outputs. In order to solve the problem, the structure, 112 neurons in 8-56-56 MLP network, is used to train those patterns using NBN algorithms. The computation time is presented in Table 13.2.

13.4.4.2 Parity-7 Problem

Parity-*N* problems are aimed to associate *n*-bit binary input data with their parity bits. It is also considered to be one of the most difficult problems in neural network training, although it has been solved analytically [03BDA].

Parity-7 problem is trained with NBN algorithms, using both the forward-only computation and traditional computation separately. Two different network structures are used for training: eight neurons in 7-7-1 MLP network (64 weights) and three neurons in FCC network (27 weights). Time cost comparison is shown in Table 13.3.

13.4.4.3 Error Correction Problems

Error correction is an extension of parity-*N* problems for multiple parity bits. In Figure 13.15, 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, so number of inputs is equal to the number of outputs.

TABLE 13.3 Comparison for Parity-7 Problem

Networks	Computation Methods	Time Cost (μs/Iteration)		Relative Time (%)
		Forward	Backward	
MLP	Backpropagation	158.57	67.82	100
	Forward-only	229.13	0.00	101.2
FCC	Backpropagation	54.14	31.94	100
	Forward-only	86.30	0.00	100.3

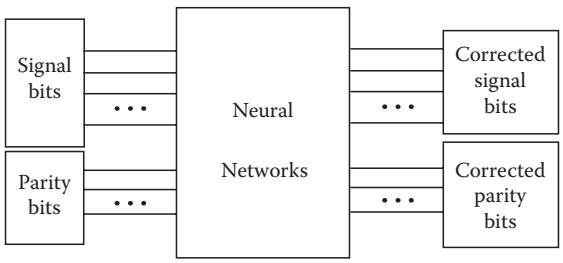


FIGURE 13.15 Using neural networks to solve error correction problem; errors in input data can be corrected by well-trained neural networks.

Two error correction experiments are presented, one has 4 bit signal with its 3 bit parity bits as inputs, 7 outputs, and 128 patterns (16 correct patterns and 112 patterns with errors), using 23 neurons in 7-16-7 MLP network (247 weights); the other has 8 bit signal with its 4 bit parity bits as inputs, 12 outputs, and 3328 patterns (256 correct patterns and 3072 patterns with errors), using 42 neurons in 12-30-12 MLP network (762 weights). Error patterns with one incorrect bit must be corrected. Both backpropagation computation and the forward-only computation were performed with the NBN algorithms. The testing results are presented in Table 13.4.

13.4.4.4 Encoders and Decoders

Experiment results on 3-to-8 decoder, 8-to-3 encoder, 4-to-16 decoder, and 16-to-4 encoder, using NBN algorithms, are presented in Table 13.5. For 3-to-8 decoder and 8-to-3 encoder, 11 neurons are used in 3-3-8 MLP network (44 weights) and 8-83 MLP network (99 weights) respectively; while for 4-to-16 decoder and 16-to-4 encoder, 20 neurons are used in 4-4-16 MLP network (100 weights) and 16-16-4 MLP network (340 weights) separately.

In the encoder and decoder problems, one may notice that for the same number of neurons, the more outputs the networks have, the more efficiently the forward-only computation works.

From the presented experimental results, one may notice that, for networks with multiple outputs, the forward-only computation is more efficient than the backpropagation computation; while for single output situation, the forward-only computation is slightly worse.

TABLE 13.4 Comparison for Error Correction Problem

Problems	Computation Methods	Time Cost (ms/Iteration)		Relative Time (%)
		Forward	Backward	
4 bit signal	Backpropagation	0.43	2.82	100
	Forward-only	1.82	0.00	56
8 bit signal	Backpropagation	40.59	468.14	100
	Forward-only	175.72	0.00	34.5

TABLE 13.5 Comparison for Encoders and Decoders

Problems	Computation Methods	Time Cost (μs/Iteration)		Relative Time (%)
		Forward	Backward	
3-to-8	Traditional	10.14	55.37	100
decoder	Forward-only	27.86	0.00	42.5
8-to-3	Traditional	7.19	26.97	100
encoder	Forward-only	29.76	0.00	87.1
4-to-16	Traditional	40.03	557.51	100
decoder	Forward-only	177.65	0.00	29.7
16-to-4	Traditional	83.24	244.20	100
encoder	Forward-only	211.28	0.00	62.5

13.5 Direct Computation of Quasi-Hessian Matrix and Gradient Vector

Using Equation 13.8 for weight updating, one may notice that the matrix multiplication $\mathbf{J}^T\mathbf{J}$ and $\mathbf{J}^T\mathbf{e}$ have to be calculated

$$\mathbf{H} \approx \mathbf{Q} = \mathbf{J}^T\mathbf{J} \tag{13.26}$$

$$\mathbf{g} = \mathbf{J}^T\mathbf{e} \tag{13.27}$$

where
matrix **Q** is the quasi-Hessian matrix
g is the gradient vector [09YW]

Traditionally, the whole Jacobian matrix **J** is calculated and stored [TM94] for further multiplication operation using Equations 13.26 and 13.27. The memory limitation may be caused by Jacobian matrix storage, as described below.

In the NBN algorithm, quasi-Hessian matrix **Q** and gradient vector **g** are calculated directly, without Jacobian matrix computation and storage. Therefore, the NBN algorithm can be used in training the problems with unlimited number of training patterns.

13.5.1 Memory Limitation in Levenberg–Marquardt Algorithm

In the Levenberg–Marquardt algorithm, Jacobian matrix **J** has to be calculated and stored for Hessian matrix computation [TM94]. In this procedure, as shown in Figure 13.2, at least $np \times no \times nn$ elements (Jacobian matrix) have to be stored. For small and median size pattern training, this method may work smoothly. However, it would be a huge memory cost for training large-sized patterns, since the number of elements of Jacobian matrix **J** is proportional to the number of patterns.

For example, the pattern recognition problem in MNIST handwritten digit database [CKOZ06] 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 Gb. This huge memory requirement cannot be satisfied by any Windows compliers, where there is a 3 Gb limitation for single-array storage. Therefore, Levenberg–Marquardt algorithm cannot be used for problems with large number of patterns.

13.5.2 Review of Matrix Algebra

There are two ways to multiply rows and columns of two matrices. If the row of the first matrix is multiplied by the column of the second matrix, then we obtain a scalar, as shown in Figure 13.16a. When the column of the first matrix is multiplied by the row of the second matrix then the result is a partial matrix **q** (Figure 13.16b) [L05]. The number of scalars is $nn \times nn$, while the number of partial matrices **q**, which later have to be summed, is $np \times no$.

When $\mathbf{J}^T$ is multiplied by **J** using routine shown in Figure 13.16b, at first, partial matrices **q** (size: $nn \times nn$) need to be calculated $np \times no$ times, then all of $np \times no$ matrices **q** must be summed together. The routine of Figure 13.16b seems complicated; therefore almost all matrix multiplication processes use the routine of Figure 13.16a, where only one element of resulted matrix is calculated and stored each time.

Even the routine of Figure 13.16b seems to be more complicated; after detailed analysis (see Table 13.6), one may conclude that the computation time for matrix multiplication of the two ways is basically the same.

In a specific case of neural network training, only one row of Jacobian matrix **J** (column of $\mathbf{J}^T$) is known for each training pattern, so if routine from Figure 13.16b is used then the process of creation of quasi-Hessian matrix can start sooner without the necessity of computing and storing the entire Jacobian matrix for all patterns and all outputs.

Table 13.7 roughly estimates the memory cost in two multiplication methods separately.

The analytical results in Table 13.7 show that the column-row multiplication (Figure 13.16b) can save a lot of memory.

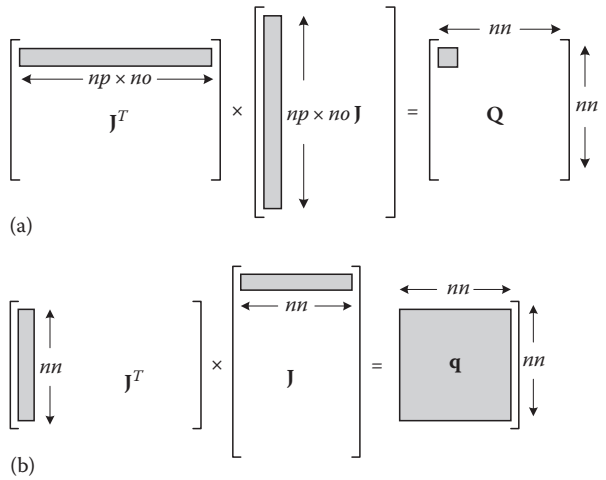


FIGURE 13.16 Two ways of multiplying matrices: (a) row–column multiplication results in a scalar and (b) column–row multiplication results in a partial matrix **q**.

TABLE 13.6 Computation Analysis

Multiplication Methods	Addition	Multiplication
Row–column	$(np \times no) \times nn \times nn$	$(np \times no) \times nn \times nn$
Column–row	$nn \times nn \times (np \times no)$	$nn \times nn \times (np \times no)$

np is the number of training patterns, *no* is the number of outputs, and *nn* is the number of weights.

TABLE 13.7 Memory Cost Analysis

Multiplication Methods	Elements for Storage
Row–column	$(np \times no) \times nn + nn \times nn + nn$
Column–row	$nn \times nn + nn$
Difference	$(np \times no) \times nn$

13.5.3 Quasi-Hessian Matrix Computation

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

$$\mathbf{q}_{p,m} = \begin{bmatrix} \left(\frac{\partial e_{p,m}}{\partial w_1}\right)^2 & \frac{\partial e_{p,m}}{\partial w_1}\frac{\partial e_{p,m}}{\partial w_2} & \cdots & \frac{\partial e_{p,m}}{\partial w_1}\frac{\partial e_{p,m}}{\partial w_{nn}} \\ \frac{\partial e_{p,m}}{\partial w_2}\frac{\partial e_{p,m}}{\partial w_1} & \left(\frac{\partial e_{p,m}}{\partial w_2}\right)^2 & \cdots & \frac{\partial e_{p,m}}{\partial w_2}\frac{\partial e_{p,m}}{\partial w_{nn}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{p,m}}{\partial w_{nn}}\frac{\partial e_{p,m}}{\partial w_1} & \frac{\partial e_{p,m}}{\partial w_{nn}}\frac{\partial e_{p,m}}{\partial w_2} & \cdots & \left(\frac{\partial e_{p,m}}{\partial w_{nn}}\right)^2 \end{bmatrix} \tag{13.28}$$

Using the procedure in Figure 13.5b, the $nn \times nn$ quasi-Hessian matrix $\mathbf{Q}$ can be calculated as the sum of submatrices $\mathbf{q}_{p,m}$

$$\mathbf{Q} = \sum_{p=1}^{np}\sum_{m=1}^{no}\mathbf{q}_{p,m} \tag{13.29}$$

By introducing $1 \times nn$ vector $\mathbf{j}_{p,m}$

$$\mathbf{j}_{p,m} = \begin{bmatrix} \frac{\partial e_{p,m}}{\partial w_1} & \frac{\partial e_{p,m}}{\partial w_2} & \cdots & \frac{\partial e_{p,m}}{\partial w_{nn}} \end{bmatrix} \tag{13.30}$$

submatrices $\mathbf{q}_{p,m}$ in Equation 13.13 can be also written in the vector form (Figure 13.5b)

$$\mathbf{q}_{p,m} = \mathbf{j}_{p,m}^T\mathbf{j}_{p,m} \tag{13.31}$$

One may notice that for the computation of submatrices $\mathbf{q}_{p,m}$, only N elements of vector $\mathbf{j}_{p,m}$ need to be calculated and stored. All the submatrices can be calculated for each pattern p and output m separately, and summed together, so as to obtain quasi-Hessian matrix $\mathbf{Q}$.

Considering the independence among all patterns and outputs, there is no need to store all the quasi-Hessian submatrices $\mathbf{q}_{p,m}$. Each submatrix can be summed to a temporary matrix after its computation. Therefore, during the direct computation of quasi-Hessian matrix $\mathbf{Q}$ using (13.29), only memory for nn elements is required, instead of that for the whole Jacobian matrix with $(np \times no) \times nn$ elements (Table 13.7).

From (13.13), one may notice that all the submatrices $\mathbf{q}_{p,m}$ are symmetrical. With this property, only upper (or lower) triangular elements of those submatrices need to be calculated. Therefore, during the improved quasi-Hessian matrix $\mathbf{Q}$ computation, multiplication operations in (13.31) and sum operations in (13.29) can be both reduced by half approximately.

13.5.4 Gradient Vector Computation

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

$$\boldsymbol{\eta}_{p,m} = \begin{bmatrix} \frac{\partial e_{p,m}}{\partial w_1} e_{p,m} \\ \frac{\partial e_{p,m}}{\partial w_2} e_{p,m} \\ \cdots \\ \frac{\partial e_{p,m}}{\partial w_{nn}} e_{p,m} \end{bmatrix} = \begin{bmatrix} \frac{\partial e_{p,m}}{\partial w_1} \\ \frac{\partial e_{p,m}}{\partial w_2} \\ \cdots \\ \frac{\partial e_{p,m}}{\partial w_{nn}} \end{bmatrix} \times \boldsymbol{e}_{p,m} \tag{13.32}$$

With the procedure in Figure 13.16b, gradient vector $\mathbf{g}$ can be calculated as the sum of gradient subvector $\boldsymbol{\eta}_{p,m}$

$$\mathbf{g} = \sum_{p=1}^{np} \sum_{m=1}^{no} \boldsymbol{\eta}_{p,m} \tag{13.33}$$

Using the same vector $\mathbf{j}_{p,m}$ defined in (13.30), gradient subvector can be calculated using

$$\boldsymbol{\eta}_{p,m} = \mathbf{j}_{p,m} e_{p,m} \tag{13.34}$$

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

With the improved computation, both quasi-Hessian matrix $\mathbf{Q}$ and gradient vector $\mathbf{g}$ can be computed directly, without Jacobian matrix storage and multiplication. During the process, only a temporary vector $\mathbf{j}_{p,m}$ with nn elements needs to be stored; in other words, the memory cost for Jacobian matrix storage is reduced by ($np \times no$) times. In the MINST problem mentioned in part one of this section, the memory cost for the storage of Jacobian elements could be reduced from more than 35 GB to nearly 30.7 kB.

AQ3

13.5.5 Jacobian Row Computation

The key point of the improved computation above for quasi-Hessian matrix $\mathbf{Q}$ and gradient vector $\mathbf{g}$ is to calculate vector $\mathbf{j}_{p,m}$ defined in (13.30) for each pattern and output. This vector is equivalent of one row of Jacobian matrix $\mathbf{J}$.

By combining Equations 13.12 and 13.30, the elements of vector $\mathbf{j}_{p,m}$ can be calculated by

$$\mathbf{j}_{p,m} = \left[\delta_{p,m,1} \left[y_{p,1,1} \quad \cdots \quad y_{p,1,i} \quad \cdots \right] \cdots \delta_{p,m,j} \left[y_{p,j,1} \quad \cdots \quad y_{p,j,i} \quad \cdots \right] \cdots \right] \tag{13.35}$$

where $y_{p,j,i}$ is the ith input of neuron j, when training pattern p.

Using the NBN procedure introduced in Section 13.3, all elements $y_{p,j,i}$ in Equation 13.35 can be calculated in the forward computation, while vector $\boldsymbol{\delta}$ is obtained in the backward computation; or, using the improved NBN procedure in Section 13.4, both vectors $\mathbf{y}$ and δ can be obtained in the improved forward computation. Again, since only one vector $\mathbf{j}_{p,m}$ needs to be stored for each pattern

```
% Initialization
Q=0;
g=0
% Improved computation
for p=1:np      % Number of patterns
 % Forward computation
 …
 for m=1:no     % Number of outputs
  % Backward computation
  …
  calculate vector j_{p,m};        % Eq. (35)
  calculate sub matrix q_{p,m};    % Eq. (31)
  calculate sub vector η_{p,m};    % Eq. (34)
  Q=Q+q_{p,m};                     % Eq. (29)
  g=g+η_{p,m};                     % Eq. (33)
 end;
end;
```

FIGURE 13.17 Pseudo code of the improved computation for quasi-Hessian matrix and gradient vector in NBN algorithm.

and output in the improved computation, the memory cost for all those temporary parameters can be reduced by ($np \times no$) times. All matrix operations are simplified to vector operations.

Generally, for the problem with *np* patterns and *no* outputs, the NBN algorithm without Jacobian matrix storage can be organized as the pseudo code shown in Figure 13.17.

13.5.6 Comparison on Memory and Time Consumption

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

13.5.6.1 Memory Comparison

Three problems, each of which has huge number of patterns, are selected to test the memory cost of both the traditional computation and the improved computation. LM algorithm and NBN algorithm are used for training, and the test results are shown in Tables 13.8 and 13.9. In order to make more precise comparison, memory cost for program code and input files were not used in the comparison.

TABLE 13.8 Memory Comparison for Parity Problems

Parity-*N* problems	*N* = 14	*N* = 16
Patterns	16,384	65,536
Structures[a]	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 (Mb)	
LM algorithm	79.21	385.22
NBN algorithm	3.41	4.3

[a] All neurons are in FCC networks.

TABLE 13.9 Memory Comparison for MINST Problem

Problem	MINST
Patterns	60,000
Structures	784 = 1 single layer network[a]
Jacobian matrix sizes	47,100,000
Weight vector sizes	785
Algorithms	**Actual Memory Cost (Mb)**
LM algorithm	385.68
NBN algorithm	15.67

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

TABLE 13.10 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 the test results in Tables 13.8 and 13.9, it is clear that memory cost for training is significantly reduced in the improved computation.

13.5.6.2 Time Comparison

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

From Table 13.10, one may notice that the NBN computation cannot only handle much larger problems, but also computes much faster than LM algorithm, especially for large-sized pattern training. The larger the pattern size is, the more time efficient the improved computation will be.

13.6 Conclusion

In this chapter, the NBN algorithm is introduced to solve the structure and memory limitation in Levenberg–Marquardt algorithm. Based on the specially designed NBN routings, the NBN algorithm can be used not only for traditional MLP networks, but also other arbitrarily connected neural networks.

The NBN algorithm can be organized in two procedures—with backpropagation process and without backpropagation process. Experimental results show that the former one is suitable for networks with single output, while the latter one is more efficient for networks with multiple outputs.

The NBN algorithm 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. Therefore, the training speed of the NBN algorithm becomes much faster than the traditional Levenberg–Marquardt algorithm.

In the NBN algorithm, quasi-Hessian matrix can be computed on fly when training patterns are applied. Moreover, it has the special advantage for applications which require dynamically changing the number of training patterns. There is no need to repeat the entire multiplication of $\mathbf{J}^T\mathbf{J}$, but only add to or subtract from quasi-Hessian matrix. The quasi-Hessian matrix can be modified as patterns are applied or removed.

There are two implementations of the NBN algorithm on the website: http://www.eng.auburn.edu/~wilambm/nnt/index.htm. MATLAB version can handle arbitrarily connected networks, but Jacobian matrix is computed and stored [07WCHK]. In the C++ version [09YW], all new features of the NBN algorithm mentioned in this chapter are implemented.

References

[AS99] J. R Alvarez-Sanchez, Injecting knowledge into the solution of the two-spiral problem. *Neural Compute and Applications*, 8, 265–272, 1999.

[AW95] T. J. Andersen and B.M. Wilamowski, A modified regression algorithm for fast one layer neural network training, *World Congress of Neural Networks*, Washington DC, July 17–21, 1995, Vol. 1, 687–690.

[BDA03] B. M. Wilamowski, D. Hunter, and A. Malinowski, Solving parity-N problems with feedforward neural networks. *Proceedings of the 2003 IEEE IJCNN*, pp. 2546–2551, IEEE Press, 2003.

[CKOZ06] L. J. Cao, S. S. Keerthi, Chong-Jin Ong, J. Q. Zhang, U. Periyathamby, Xiu Ju Fu, and H. P. Lee, Parallel sequential minimal optimization for the training of support vector machines, *IEEE Transactions on Neural Networks*, 17(4), 1039–1049, April 2006.

[L05] David C. Lay, *Linear Algebra and its Applications*, Addison-Wesley Publishing Company, 3rd version, pp. 124, July, 2005.

[L44] K. Levenberg, A method for the solution of certain problems in least squares. *Quarterly of Applied Mathematics*, 5, 164–168, 1944.

[M63] D. Marquardt, An algorithm for least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math.*, 11(2), 431–441, June 1963.

[N89] Robert Hecht Nielsen, Theory of the back propagation neural network. *Proceedings of the 1989 IEEE IJCNN*, pp. 1593–1605, IEEE Press, New York, 1989.

[PLI08] Jian-Xun Peng, Kang Li, and G. W. Irwin, A new Jacobian matrix for optimal learning of single-layer neural networks, *IEEE Transactions on Neural Networks*, 19(1), 119–129, January 2008.

[TM94] Hagan M. T., M. Menhaj, Training feedforward networks with the Marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6), 989–993, 1994.

[W09] B. M. Wilamowski, Neural network architectures and learning algorithms, *IEEE Industrial Electronics Magazine*, 3(4), 56–63.

[WB01] B. M. Wilamowski and J. Binfet, Microprocessor implementation of fuzzy systems and neural networks, *International Joint Conference on Neural Networks (IJCNN'01)*, Washington DC, July 15–19, 2001, pp. 234–239. AQ4

[WCHK07] B. M. Wilamowski, N. Cotton, J. Hewlett, and O. Kaynak, Neural network trainer with second order learning algorithms, *Proceedings of the International Conference on Intelligent Engineering Systems*, June 29, 2007–July 1, 2007, pp. 127–132.

[WCKD08] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dundar, Computing gradient vector and Jacobian matrix in arbitrarily connected neural networks, *IEEE Trans. on Industrial Electronics*, 55(10), 3784–3790, October 2008.

[WCM99] B. M. Wilamowski, Y. Chen, and A. Malinowski, Efficient algorithm for training neural networks with one hidden layer, presented at *1999 International Joint Conference on Neural Networks (IJCNN'99)*, Washington, DC, July 10–16, 1999, pp. 1725–1728.

[WH10] B. M. Wilamowski and H. Yu, Improved computation for Levenberg Marquardt training, *IEEE Transactions on Neural Networks*, 21, 2010.

[WT93] B. M. Wilamowski and L. Torvik, Modification of gradient computation in the back-propagation algorithm, presented at *Artificial Neural Networks in Engineering (ANNIE'93)*, St. Louis, MI, November 14–17, 1993.

[YW09] Hao Yu and B. M. Wilamowski, C++ implementation of neural networks trainer, *13th International Conference on Intelligent Engineering Systems (INES-09)*, Barbados, April 16–18, 2009.