# 12

# Levenberg–Marquardt Training

Hao Yu
*Auburn University*

Bogdan M. Wilamowski
*Auburn University*

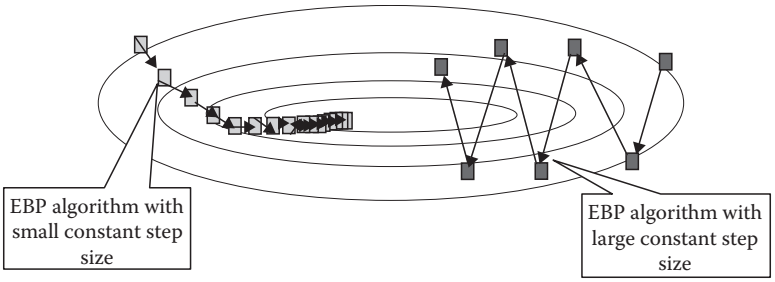


## 12.1 Introduction

The Levenberg–Marquardt algorithm [L44,M63], which was independently developed by Kenneth Levenberg and Donald Marquardt, provides a numerical solution to the problem of minimizing a nonlinear function. It is fast and has stable convergence. In the artificial neural-networks field, this algorithm is suitable for training small- and medium-sized problems.

Many other methods have already been developed for neural-networks training. The steepest descent algorithm, also known as the error backpropagation (EBP) algorithm [EHW86,J88], dispersed the dark clouds on the field of artificial neural networks and could be regarded as one of the most significant breakthroughs for training neural networks. Many improvements have been made to EBP [WT93,AW95,W96,WCM99], but these improvements are relatively minor [W02,WHM03,YW09,W09,WY10]. The EBP algorithm is still widely used today; however, it is also known as an inefficient algorithm because of its slow convergence. There are two main reasons for the slow convergence: the first reason is that its step sizes should be adequate to the gradients (Figure 12.1). Logically, small step sizes should be taken where the gradient is steep so as not to rattle out of the required minima (because of oscillation). So, if the step size is a constant, it needs to be chosen small. Then, in the place where the gradient is gentle, the training process would be very slow. The second reason is that the curvature of the error surface may not be the same in all directions, such as the Rosenbrock function, so the classic "error valley" problem [O92] may exist and may result in the slow convergence.

The slow convergence of the steepest descent method can be greatly improved by the Gauss–Newton algorithm [O92]. Using second-order derivatives of error function to "naturally" evaluate the curvature of error surface, The Gauss–Newton algorithm can find proper step sizes for each direction and converge very fast; especially, if the error function has a quadratic surface, it can converge directly in the first iteration. But this improvement only happens when the quadratic approximation of error function is reasonable. Otherwise, the Gauss–Newton algorithm would be mostly divergent.




AQ1 **FIGURE 12.1** Searching process of the steepest descent method with different learning constants: yellow trajectory is for small learning constant that leads to slow convergence; purple trajectory is for large learning constant that causes oscillation (divergence).

The Levenberg–Marquardt algorithm blends the steepest descent method and the Gauss–Newton algorithm. Fortunately, it inherits the speed advantage of the Gauss–Newton algorithm and the stability of the steepest descent method. It's more robust than the Gauss–Newton algorithm, because in many cases it can converge well even if the error surface is much more complex than the quadratic situation. Although the Levenberg–Marquardt algorithm tends to be a bit slower than Gauss–Newton algorithm (in convergent situation), it converges much faster than the steepest descent method.

The basic idea of the Levenberg–Marquardt algorithm is that it performs a combined training process: around the area with complex curvature, the Levenberg–Marquardt algorithm switches to the steepest descent algorithm, until the local curvature is proper to make a quadratic approximation; then it approximately becomes the Gauss–Newton algorithm, which can speed up the convergence significantly.

## 12.2 Algorithm Derivation

In this part, the derivation of the Levenberg–Marquardt algorithm will be presented in four parts: (1) steepest descent algorithm, (2) Newton's method, (3) Gauss–Newton's algorithm, and (4) Levenberg–Marquardt algorithm.

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

- $p$ is the index of patterns, from 1 to $P$, where $P$ is the number of patterns.
- $m$ is the index of outputs, from 1 to $M$, where $M$ is the number of outputs.
- $i$ and $j$ are the indices of weights, from 1 to $N$, where $N$ is the number of weights.
- $k$ is the index of iterations.

Other indices will be explained in related places.

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

$$E(\boldsymbol{x},\boldsymbol{w}) = \frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M}\boldsymbol{e}_{p,m}^{2} \tag{12.1}$$

where

$\boldsymbol{x}$ is the input vector

$\boldsymbol{w}$ is the weight vector

$\boldsymbol{e}_{p,m}$ is the training error at output $m$ when applying pattern $p$ and it is defined as

$$\boldsymbol{e}_{p,m} = \boldsymbol{d}_{p,m} - \boldsymbol{o}_{p,m} \tag{12.2}$$

where

$\boldsymbol{d}$ is the desired output vector

$\boldsymbol{o}$ is the actual output vector

### 12.2.1 Steepest Descent Algorithm

The steepest descent algorithm is a first-order algorithm. It uses the first-order derivative of total error function to find the minima in error space. Normally, gradient $\boldsymbol{g}$ is defined as the first-order derivative of total error function (12.1):

$$\boldsymbol{g} = \frac{\partial E(\boldsymbol{x}, \boldsymbol{w})}{\partial \boldsymbol{w}} = \left[ \frac{\partial E}{\partial w_1} \quad \frac{\partial E}{\partial w_2} \quad \cdots \quad \frac{\partial E}{\partial w_N} \right]^T \tag{12.3}$$

With the definition of gradient $\boldsymbol{g}$ in (12.3), the update rule of the steepest descent algorithm could be written as

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \mathbf{g}_k \tag{12.4}$$

where $\alpha$ is the learning constant (step size).

The training process of the steepest descent algorithm is asymptotic convergence. Around the solution, all the elements of gradient vector would be very small and there would be a very tiny weight change.

### 12.2.2 Newton's Method

Newton's method assumes that all the gradient components $g_1, g_2, \ldots, g_N$ are functions of weights and all weights are linearly independent:

$$\begin{cases} g_1 = F_1(w_1, w_2 \cdots w_N) \\ g_2 = F_2(w_1, w_2 \cdots w_N) \\ \cdots \\ g_N = F_N(w_1, w_2 \cdots w_N) \end{cases} \tag{12.5}$$

AQ2

where $F_1, F_2, \ldots, F_N$ are nonlinear relationships between weights and related gradient components.

Unfold each $g_i$ ($i = 1, 2, \ldots, N$) in Equations 12.5 by Taylor series and take the first-order approximation:

$$\begin{cases} g_1 \approx g_{1,0} + \dfrac{\partial g_1}{\partial w_1}\Delta w_1 + \dfrac{\partial g_1}{\partial w_2}\Delta w_2 + \cdots + \dfrac{\partial g_1}{\partial w_N}\Delta w_N \\ g_2 \approx g_{2,0} + \dfrac{\partial g_2}{\partial w_1}\Delta w_1 + \dfrac{\partial g_2}{\partial w_2}\Delta w_2 + \cdots + \dfrac{\partial g_2}{\partial w_N}\Delta w_N \\ \cdots \\ g_N \approx g_{N,0} + \dfrac{\partial g_N}{\partial w_1}\Delta w_1 + \dfrac{\partial g_N}{\partial w_2}\Delta w_2 + \cdots + \dfrac{\partial g_N}{\partial w_N}\Delta w_N \end{cases} \tag{12.6}$$

By combining the definition of gradient vector $\boldsymbol{g}$ in (12.3), it could be determined that

$$\frac{\partial g_i}{\partial w_j} = \frac{\partial\left(\dfrac{\partial E}{\partial w_j}\right)}{\partial w_j} = \frac{\partial^2 E}{\partial w_i \partial w_j} \tag{12.7}$$

By inserting Equation 12.7 to 12.6:

$$\begin{cases} g_1 \approx g_{1,0} + \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ g_2 \approx g_{2,0} + \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ g_N \approx g_{N,0} + \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \tag{12.8}$$

Comparing with the steepest descent method, the second-order derivatives of the total error function need to be calculated for each component of gradient vector.

In order to get the minima of total error function $E$, each element of the gradient vector should be zero. Therefore, left sides of the Equations 12.8 are all zero, then

$$\begin{cases} 0 \approx g_{1,0} + \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ 0 \approx g_{2,0} + \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ 0 \approx g_{N,0} + \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \tag{12.9}$$

By combining Equation 12.3 with 12.9

$$\begin{cases} -\dfrac{\partial E}{\partial w_1} = -g_{1,0} \approx \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ -\dfrac{\partial E}{\partial w_2} = -g_{2,0} \approx \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ -\dfrac{\partial E}{\partial w_N} = -g_{N,0} \approx \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \tag{12.10}$$

AQ3 There are $N$ equations for $N$ parameters so that all $\Delta w_i$ can be calculated. With the solutions, the weight space can be updated iteratively.

Equations 12.10 can be also written in matrix form

$$\begin{bmatrix} -g_1 \\ -g_2 \\ \cdots \\ -g_N \end{bmatrix} = \begin{bmatrix} -\dfrac{\partial E}{\partial w_1} \\ -\dfrac{\partial E}{\partial w_2} \\ \cdots \\ -\dfrac{\partial E}{\partial w_N} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial^2 E}{\partial w_1^2} & \dfrac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_1 \partial w_N} \\ \dfrac{\partial^2 E}{\partial w_2 \partial w_1} & \dfrac{\partial^2 E}{\partial w_2^2} & \cdots & \dfrac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial^2 E}{\partial w_N \partial w_1} & \dfrac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \times \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \cdots \\ \Delta w_N \end{bmatrix} \tag{12.11}$$

where the square matrix is Hessian matrix:

$$\boldsymbol{H} = \begin{bmatrix} \dfrac{\partial^2 E}{\partial w_1^2} & \dfrac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_1 \partial w_N} \\ \dfrac{\partial^2 E}{\partial w_2 \partial w_1} & \dfrac{\partial^2 E}{\partial w_2^2} & \cdots & \dfrac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial^2 E}{\partial w_N \partial w_1} & \dfrac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \dfrac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \tag{12.12}$$

By combining Equations 12.3 and 12.12 with Equation 12.11

$$-\boldsymbol{g} = \boldsymbol{H}\Delta\boldsymbol{w} \tag{12.13}$$

So

$$\Delta\boldsymbol{w} = -\boldsymbol{H}^{-1}\boldsymbol{g} \tag{12.14}$$

Therefore, the update rule for Newton's method is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k \tag{12.15}$$

As the second-order derivatives of total error function, Hessian matrix $\boldsymbol{H}$ gives the proper evaluation on the change of gradient vector. By comparing Equations 12.4 and 12.15, one may notice that well-matched step sizes are given by the inverted Hessian matrix.

### 12.2.3 Gauss–Newton Algorithm

If Newton's method is applied for weight updating, in order to get Hessian matrix $\boldsymbol{H}$, the second-order derivatives of total error function have to be calculated and it could be very complicated. In order to simplify the calculating process, Jacobian matrix $\boldsymbol{J}$ is introduced as

$$\boldsymbol{J} = \begin{bmatrix} \dfrac{\partial e_{1,1}}{\partial w_1} & \dfrac{\partial e_{1,1}}{\partial w_2} & \cdots & \dfrac{\partial e_{1,1}}{\partial w_N} \\ \dfrac{\partial e_{1,2}}{\partial w_1} & \dfrac{\partial e_{1,2}}{\partial w_2} & \cdots & \dfrac{\partial e_{1,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial e_{1,M}}{\partial w_1} & \dfrac{\partial e_{1,M}}{\partial w_2} & \cdots & \dfrac{\partial e_{1,M}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial e_{P,1}}{\partial w_1} & \dfrac{\partial e_{P,1}}{\partial w_2} & \cdots & \dfrac{\partial e_{P,1}}{\partial w_N} \\ \dfrac{\partial e_{P,2}}{\partial w_1} & \dfrac{\partial e_{P,2}}{\partial w_2} & \cdots & \dfrac{\partial e_{P,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \dfrac{\partial e_{P,M}}{\partial w_1} & \dfrac{\partial e_{P,M}}{\partial w_2} & \cdots & \dfrac{\partial e_{P,M}}{\partial w_N} \end{bmatrix} \tag{12.16}$$

By integrating Equations 12.1 and 12.3, the elements of gradient vector can be calculated as

$$g_i = \frac{\partial E}{\partial w_i} = \frac{\partial\left(\frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M} e_{p,m}^2\right)}{\partial w_i} = \sum_{p=1}^{P}\sum_{m=1}^{M}\left(\frac{\partial e_{p,m}}{\partial w_i} e_{p,m}\right) \tag{12.17}$$

Combining Equations 12.16 and 12.17, the relationship between Jacobian matrix $\boldsymbol{J}$ and gradient vector $\boldsymbol{g}$ would be

$$\boldsymbol{g} = \boldsymbol{J}\boldsymbol{e} \tag{12.18}$$

where error vector $\boldsymbol{e}$ has the form

$$\boldsymbol{e} = \begin{bmatrix} e_{1,1} \\ e_{1,2} \\ \cdots \\ e_{1,M} \\ \cdots \\ e_{P,1} \\ e_{P,2} \\ \cdots \\ e_{P,M} \end{bmatrix} \tag{12.19}$$

Inserting Equation 12.1 into 12.12, the element at $i$th row and $j$th column of Hessian matrix can be calculated as

$$h_{i,j} = \frac{\partial^2 E}{\partial w_i \partial w_j} = \frac{\partial^2\left(\frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M} e_{p,m}^2\right)}{\partial w_i \partial w_j} = \sum_{p=1}^{P}\sum_{m=1}^{M}\frac{\partial e_{p,m}}{\partial w_i}\frac{\partial e_{p,m}}{\partial w_j} + S_{i,j} \tag{12.20}$$

where $S_{i,j}$ is equal to

$$S_{i,j} = \sum_{p=1}^{P}\sum_{m=1}^{M}\frac{\partial^2 e_{p,m}}{\partial w_i \partial w_j} e_{p,m} \tag{12.21}$$

As the basic assumption of Newton's method is that $S_{i,j}$ is closed to zero [TM94], the relationship between Hessian matrix $\boldsymbol{H}$ and Jacobian matrix $\boldsymbol{J}$ can be rewritten as

$$\boldsymbol{H} \approx \boldsymbol{J}^T\boldsymbol{J} \tag{12.22}$$

By combining Equations 12.15, 12.18, and 12.22, the update rule of the Gauss–Newton algorithm is presented as

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \left(\boldsymbol{J}_k^T\boldsymbol{J}_k\right)^{-1}\boldsymbol{J}_k\boldsymbol{e}_k \tag{12.23}$$

Obviously, the advantage of the Gauss–Newton algorithm over the standard Newton's method (Equation 12.15) is that the former does not require the calculation of second-order derivatives of the total error function, by introducing Jacobian matrix $\boldsymbol{J}$ instead. However, the Gauss–Newton algorithm still faces the same convergent problem like the Newton algorithm for complex error space optimization. Mathematically, the problem can be interpreted as the matrix $\boldsymbol{J}^T\boldsymbol{J}$ may not be invertible.

### 12.2.4 Levenberg–Marquardt Algorithm

In order to make sure that the approximated Hessian matrix $\boldsymbol{J}^T\boldsymbol{J}$ is invertible, Levenberg–Marquardt algorithm introduces another approximation to Hessian matrix:

$$\boldsymbol{H} \approx \boldsymbol{J}^T\boldsymbol{J} + \mu\boldsymbol{I} \tag{12.24}$$

where

$\mu$ is always positive, called combination coefficient
$\boldsymbol{I}$ is the identity matrix

From Equation 12.24, one may notice that the elements on the main diagonal of the approximated Hessian matrix will be larger than zero. Therefore, with this approximation (Equation 12.24), it can be sure that matrix $\boldsymbol{H}$ is always invertible.

By combining Equations 12.23 and 12.24, the update rule of Levenberg–Marquardt algorithm can be presented as

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \left(\boldsymbol{J}_k^T\boldsymbol{J}_k + \mu\boldsymbol{I}\right)^{-1}\boldsymbol{J}_k\boldsymbol{e}_k \tag{12.25}$$

As the combination of the steepest descent algorithm and the Gauss–Newton algorithm, the Levenberg–Marquardt algorithm switches between the two algorithms during the training process. When the combination coefficient $\mu$ is very small (nearly zero), Equation 12.25 is approaching to Equation 12.23 and Gauss–Newton algorithm is used. When combination coefficient $\mu$ is very large, Equation 12.25 approximates to Equation 12.4 and the steepest descent method is used.

If the combination coefficient $\mu$ in Equation 12.25 is very big, it can be interpreted as the learning coefficient in the steepest descent method (12.4):

$$\alpha = \frac{1}{\mu} \tag{12.26}$$

Table 12.1 summarizes the update rules for various algorithms.

**TABLE 12.1** Specifications of Different Algorithms

| Algorithms | Update Rules | Convergence | Computation Complexity |
|---|---|---|---|
| EBP algorithm | $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha\boldsymbol{g}_k$ | Stable, slow | Gradient |
| Newton algorithm | $\mathbf{w}_{k+1} = \mathbf{w}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k$ | Unstable, fast | Gradient and Hessian |
| Gauss–Newton algorithm | $\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\boldsymbol{J}_k^T\boldsymbol{J}_k\right)^{-1}\boldsymbol{J}_k\mathbf{e}_k$ | Unstable, fast | Jacobian |
| Levenberg–Marquardt algorithm | $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \left(\boldsymbol{J}_k^T\boldsymbol{J}_k + \mu\boldsymbol{I}\right)^{-1}\boldsymbol{J}_k\boldsymbol{e}_k$ | Stable, fast | Jacobian |
| NBN algorithm [08WC][a] | $\mathbf{w}_{k+1} = \mathbf{w}_k - \boldsymbol{Q}_k^{-1}\boldsymbol{g}_k$ | Stable, fast | Quasi Hessian[a] |

[a] Reference Chapter 12.

## 12.3 Algorithm Implementation

In order to implement the Levenberg–Marquardt algorithm for neural network training, two problems have to be solved: how does one calculate the Jacobian matrix, and how does one organize the training process iteratively for weight updating.

In this section, the implementation of training with the Levenberg–Marquardt algorithm will be introduced in two parts: (1) calculation of Jacobian matrix; (2) training process design.

### 12.3.1 Calculation of Jacobian Matrix

AQ4 Different from Section 12.2, in the computation followed, $j$ and $k$ are used as the indices of neurons, from 1 to $nn$, where $nn$ is the number of neurons contained in a topology; $i$ is the index of neuron inputs, from 1 to $ni$, where $ni$ is the number of inputs and it may vary for different neurons.

As an introduction of basic concepts of neural network training, let us consider a neuron $j$ with $ni$ inputs, as shown in Figure 12.2. 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.

Node $y$ is an important and flexible concept. It can be $y_{j,i}$, meaning the $i$th input of neuron $j$. It also can be used as $y_j$ to define the output of neuron $j$. In the following derivation, 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.

The output node of neuron $j$ is calculated using

$$y_j = f_j(net_j) \tag{12.27}$$

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

$$net_j = \sum_{i=1}^{ni} w_{j,i} y_{j,i} + w_{j,0} \tag{12.28}$$

where

$y_{j,i}$ is the $i$th input node of neuron $j$, weighted by $w_{j,i}$
$w_{j,0}$ is the bias weight of neuron $j$

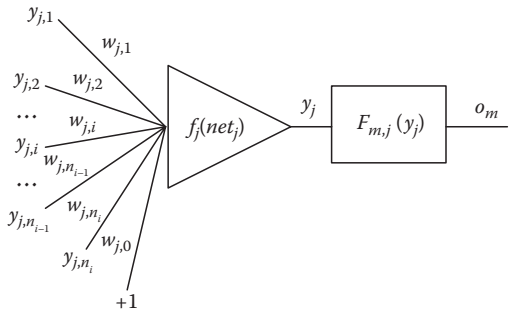



**FIGURE 12.2** Connection of a neuron $j$ with the rest of the network. Nodes $y_{j,i}$ could represent network inputs or outputs of other neurons. $F_{m,j}(y_j)$ is the nonlinear relationship between the neuron output node $y_j$ and the network output $o_m$.

Using Equation 12.28, one may notice that derivative of $net_j$ is

$$\frac{\partial net_j}{\partial w_{j,i}} = y_{j,i} \tag{12.29}$$

and slope $s_j$ of activation function $f_j$ is

$$s_j = \frac{\partial y_j}{\partial net_j} = \frac{\partial f_j(net_j)}{\partial net_j} \tag{12.30}$$

Between the output node $y_j$ of a hidden neuron $j$ and network output $o_m$, there is a complex nonlinear relationship (Figure 12.2):

$$o_m = F_{m,j}(y_j) \tag{12.31}$$

where $o_m$ is the $m$th 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 $F'_{mj}(y_j) = 1$, where $F'_{mj}$ is the derivative of nonlinear relationship between neuron $j$ and output $m$.

The elements of Jacobian matrix in Equation 12.16 can be calculated as

$$\frac{\partial e_{p,m}}{\partial w_{j,i}} = \frac{\partial \left(d_{p,m} - o_{p,m}\right)}{\partial w_{j,i}} = -\frac{\partial o_{p,m}}{\partial w_{j,i}} = -\frac{\partial o_{p,m}}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{j,i}} \tag{12.32}$$

Combining with Equations 12.28 through 12.30, 12.31 can be rewritten as

$$\frac{\partial e_{p,m}}{\partial w_{j,i}} = -F'_{mj} s_j y_{j,i} \tag{12.33}$$

where $F'_{mj}$ is the derivative of nonlinear function between neuron $j$ and output $m$.

The computation process for Jacobian matrix can be organized according to the traditional backpropagation computation in first-order algorithms (like the EBP algorithm). But there are also differences between them. First of all, for every pattern, in the EBP algorithm, only one backpropagation process is needed, while in the Levenberg–Marquardt algorithm the backpropagation process has to be repeated for every output separately in order to obtain consecutive rows of the Jacobian matrix (Equation 12.16). Another difference is that the concept of backpropagation of δ parameter [N89] has to be modified. In the EBP algorithm, output errors are parts of the δ parameter:

$$\delta_j = s_j \sum_{m=1}^{M} F'_{mj} e_m \tag{12.34}$$

In the Levenberg–Marquardt algorithm, the δ parameters are calculated for each neuron $j$ and each output $m$, separately. Also, in the backpropagation process, the error is replaced by a unit value [TM94]:

$$\delta_{m,j} = s_j F'_{mj} \tag{12.35}$$

By combining Equations 12.33 and 12.35, elements of the Jacobian matrix can be calculated by

$$\frac{\partial e_{p,m}}{\partial w_{j,i}} = -\delta_{m,j} y_{j,i} \tag{12.36}$$

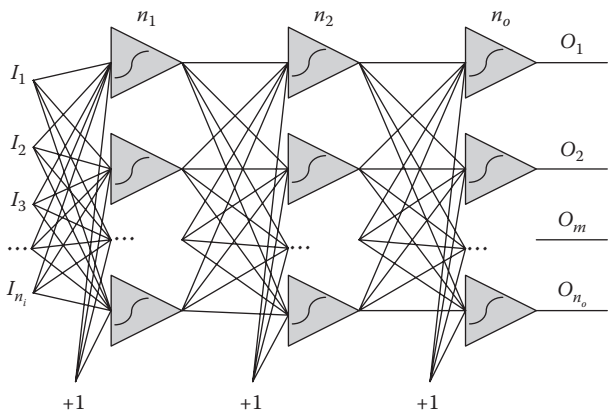



**FIGURE 12.3** Three-layer multilayer perceptron network: the number of inputs is $n_i$, the number of outputs is $n_0$, and $n_1$ and $n_2$ are the numbers of neurons in the first and second layers separately.

There are two unknowns in Equation 12.36 for the Jacobian matrix computation. The input node, $y_{j,i}$, can be calculated in the forward computation (signal propagating from inputs to outputs); while $\delta_{m,j}$ is obtained in the backward computation, which is organized as errors backpropagating from output neurons (output layer) to network inputs (input layer). At output neuron $m$ ($j = m$), $\delta_{m,j} = s_m$.

For better interpretation of forward computation and backward computation, let us consider the three-layer multilayer perceptron network (Figure 12.3) as an example.

For a given pattern, the forward computation can be organized in the following steps:

a. Calculate net values, slopes, and outputs for all neurons in the first layer:

$$net_j^1 = \sum_{i=1}^{ni} I_i w_{j,i}^1 + w_{j,0}^1 \tag{12.37}$$

$$y_j^1 = f_j^1\left(net_j^1\right) \tag{12.38}$$

$$s_j^1 = \frac{\partial f_j^1}{\partial net_j^1} \tag{12.39}$$

where
$I_i$ are the network inputs
the superscript "1" means the first layer
$j$ is the index of neurons in the first layer

b. Use the outputs of the first layer neurons as the inputs of all neurons in the second layer, do a similar calculation for net values, slopes, and outputs:

$$net_j^2 = \sum_{i=1}^{n_1} y_i^1 w_{j,i}^2 + w_{j,0}^2 \tag{12.40}$$

$$y_j^2 = f_j^2\left(net_j^2\right) \tag{12.41}$$

$$s_j^2 = \frac{\partial f_j^2}{\partial net_j^2} \tag{12.42}$$

c. Use the outputs of the second layer neurons as the inputs of all neurons in the output layer (third layer), do a similar calculation for net values, slopes, and outputs:

$$net_j^3 = \sum_{i=1}^{n_2} y_i^2 w_{j,i}^3 + w_{j,0}^3 \tag{12.43}$$

$$o_j = f_j^3\left(net_j^3\right) \tag{12.44}$$

$$s_j^3 = \frac{\partial f_j^3}{\partial net_j^3} \tag{12.45}$$

After the forward calculation, node array $\boldsymbol{y}$ and slope array $\boldsymbol{s}$ can be obtained for all neurons with the given pattern.

With the results from the forward computation, for a given output $j$, the backward computation can be organized as

d. Calculate error at the output $j$ and initial δ as the slope of output $j$:

$$e_j = d_j - o_j \tag{12.46}$$

$$\delta_{j,j}^3 = s_j^3 \tag{12.47}$$

$$\delta_{j,k}^3 = 0 \tag{12.48}$$

where

$d_j$ is the desired output at output $j$
$o_j$ is the actual output at output $j$ obtained in the forward computation
$\delta_{j,j}^3$ is the self-backpropagation
$\delta_{j,k}^3$ is the backpropagation from other neurons in the same layer (output layer)

e. Backpropagate δ from the inputs of the third layer to the outputs of the second layer

$$\delta_{j,k}^2 = w_{j,k}^3 \delta_{j,j}^3 \tag{12.49}$$

where $k$ is the index of neurons in the second layer, from 1 to $n_2$.

f. Backpropagate δ from the outputs of the second layer to the inputs of the second layer

$$\delta_{j,k}^2 = \delta_{j,k}^2 s_k^2 \tag{12.50}$$

where $k$ is the index of neurons in the second layer, from 1 to $n_2$.

g. Backpropagate δ from the inputs of the second layer to the outputs of the first layer

$$\delta_{j,k}^1 = \sum_{i=1}^{n_2} w_{j,i}^2 \delta_{j,i}^2 \tag{12.51}$$

where $k$ is the index of neurons in the first layer, from 1 to $n_1$.

h. Backpropagate δ from the outputs of the first layer to the inputs of the first layer

$$\delta^1_{j,k} = \delta^1_{j,k} s^1_k \tag{12.52}$$

where $k$ is the index of neurons in the second layer, from 1 to $n_1$.

For the backpropagation process of other outputs, the steps (d)–(h) are repeated.

By performing the forward computation and backward computation, the whole $\boldsymbol{\delta}$ array and $\boldsymbol{y}$ array can be obtained for the given pattern. Then related row elements (*no* rows) of Jacobian matrix can be calculated by using Equation 12.36.

For other patterns, by repeating the forward and backward computation, the whole Jacobian matrix can be calculated.

The pseudo code of the forward computation and backward computation for Jacobian matrix in the Levenberg–Marquardt algorithm is shown in Figure 12.4.

## 12.3.2 Training Process Design

With the update rule of the Levenberg–Marquardt algorithm (Equation 12.25) and the computation of Jacobian matrix, the next step is to organize the training process.

According to the update rule, if the error goes down, which means it is smaller than the last error, it implies that the quadratic approximation on total error function is working and the combination coefficient μ could be changed smaller to reduce the influence of gradient descent part (ready to speed up). On the other hand, if the error goes up, which means it's larger than the last error, it shows that it's necessary to follow the gradient more to look for a proper curvature for quadratic approximation and the combination coefficient μ is increased.

```
for all patterns
% Forward computation
  for all layers
    for all neurons in the layer
      calculate net;     % Equation (12.28)
      calculate output;  % Equation (12.27)
      calculate slope;   % Equation (12.30)
    end;
  end;
%Backward computation
  initial delta as slope;
  for all outputs
    calculate error;
    for all layers
         for all ne

urons in the previous layer
         for all neurons in the current layer
           multiply delta through weights
           sum the backpropagated delta at proper nodes
         end;
         multiply delta by slope;
    end;
  end;
```

AQ5

**FIGURE 12.4** Pseudo code of forward computation and backward computation implementing Levenberg–Marquardt algorithm.

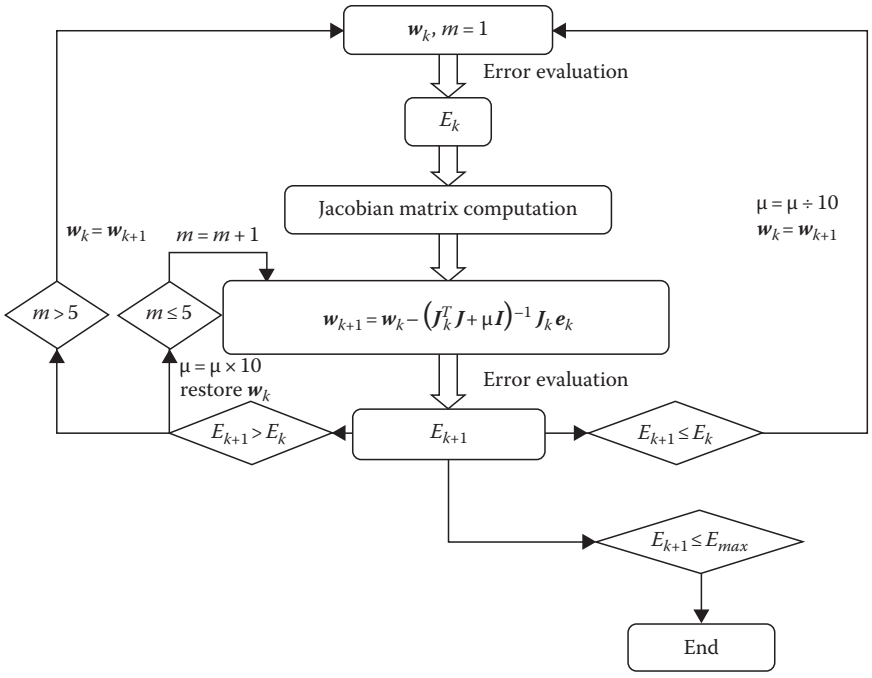



**FIGURE 12.5** Block diagram for training using Levenberg–Marquardt algorithm: $\boldsymbol{w}_k$ is the current weight, $\boldsymbol{w}_{k+1}$ is the next weight, $E_{k+1}$ is the current total error, and $E_k$ is the last total error.

Therefore, the training process using Levenberg–Marquardt algorithm could be designed as follows:

i. With the initial weights (randomly generated), evaluate the total error (SSE).
ii. Do an update as directed by Equation 12.25 to adjust weights.
iii. With the new weights, evaluate the total error.
iv. If the current total error is increased as a result of the update, then retract the step (such as reset the weight vector to the precious value) and increase combination coefficient μ by a factor of 10 or by some other factors. Then go to step ii and try an update again.
v. If the current total error is decreased as a result of the update, then accept the step (such as keep the new weight vector as the current one) and decrease the combination coefficient μ by a factor of 10 or by the same factor as step iv.
vi. Go to step ii with the new weights until the current total error is smaller than the required value.

The flowchart of the above procedure is shown in Figure 12.5.

## 12.4 Comparison of Algorithms

In order to illustrate the advantage of the Levenberg–Marquardt algorithm, let us use the parity-3 problem (see Figure 12.6) as an example and make a comparison among the EBP algorithm, the Gauss–Newton algorithm, and the Levenberg algorithm.

Three neurons in multilayer perceptron network (Figure 12.7) are used for training, and the required training error is 0.01. In order to compare the convergent rate, for each algorithm, 100 trials are tested with randomly generated weights (between –1 and 1).

| Inputs | | | Outputs |
|---|---|---|---|
| –1 | –1 | –1 | –1 |
| –1 | –1 | 1 | 1 |
| –1 | 1 | –1 | 1 |
| –1 | 1 | 1 | –1 |
| 1 | –1 | –1 | 1 |
| 1 | –1 | 1 | –1 |
| 1 | 1 | –1 | –1 |
| 1 | 1 | 1 | 1 |

**FIGURE 12.6** Training patterns of the parity-3 problem.

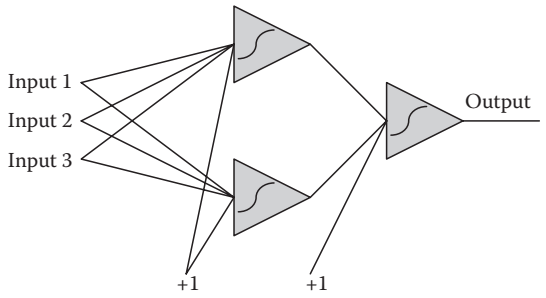



**FIGURE 12.7** Three neurons in multilayer perceptron network.

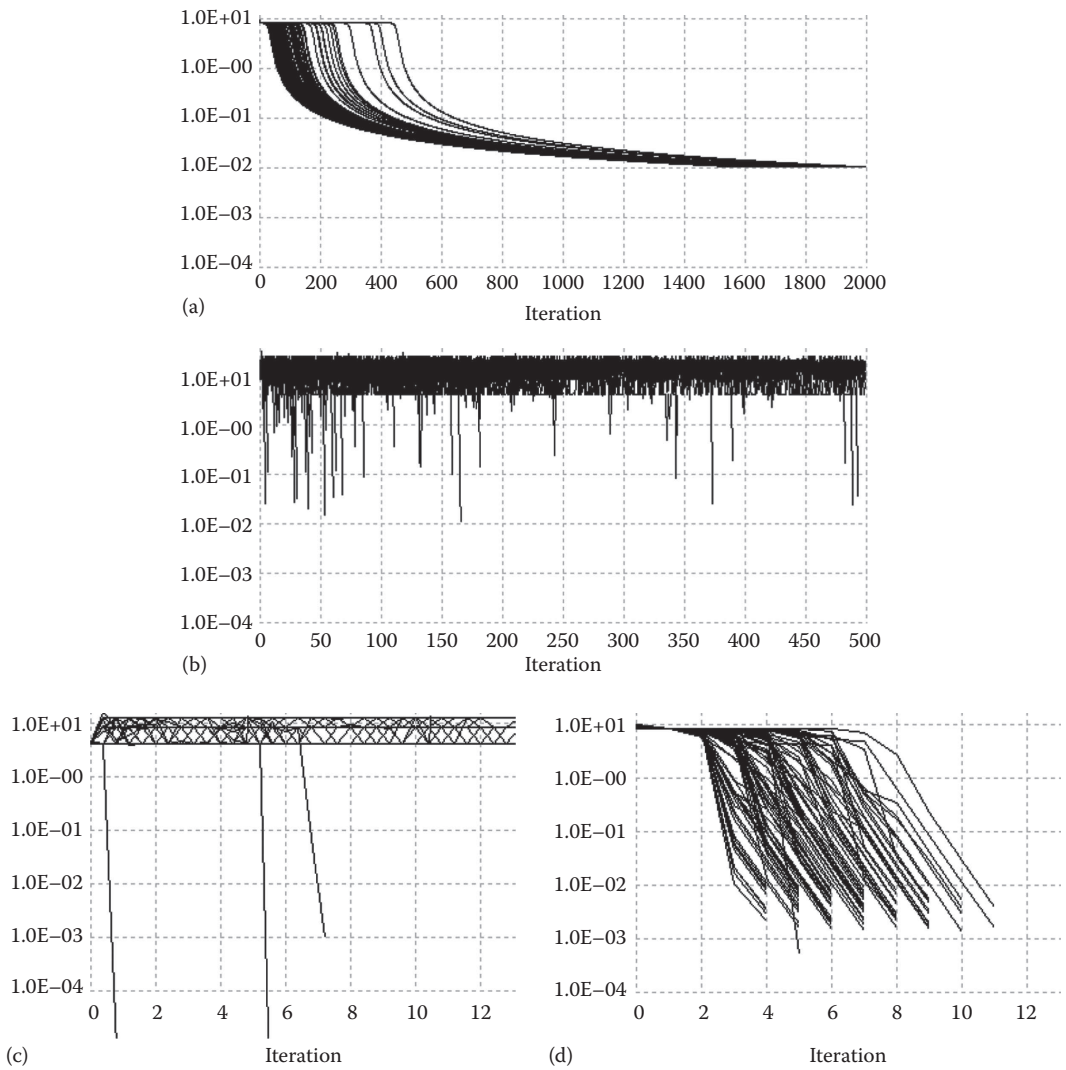



**FIGURE 12.8** Training results of parity-3 problem: (a) EBP algorithm ($\alpha = 1$), (b) EBP algorithm ($\alpha = 100$) (c) Gauss–Newton algorithm, and (d) Levenberg–Marquardt algorithm

**TABLE 12.2** Comparison among Different Algorithms for Parity-3 Problem

| Algorithms | Convergence Rate (%) | Average Iteration | Average Time (ms) |
|---|---|---|---|
| EBP algorithm ($\alpha = 1$) | 100 | 1646.52 | 320.6 |
| EBP algorithm ($\alpha = 100$) | 79 | 171.48 | 36.5 |
| Gauss–Newton algorithm | 3 | 4.33 | 1.2 |
| Levenberg–Marquardt algorithm | 100 | 6.18 | 1.6 |

The training results are shown in Figure 12.8 and the comparison is presented in Table 12.2. One may notice that: (1) for the EBP algorithm, the larger the training constant $\alpha$ is, the faster and less stable the training process will be; (2) Levenberg–Marquardt is much faster than the EBP algorithm and more stable than the Gauss–Newton algorithm.

For more complex parity-*N* problems, the Gauss–Newton method cannot converge at all, and the EBP algorithm also becomes more inefficient to find the solution, while the Levenberg–Marquardt algorithm may lead to successful solutions.

## 12.5 Summary

The Levenberg–Marquardt algorithm solves the problems existing in both gradient descent method and the Gauss–Newton method for neural-networks training, by the combination of those two algorithms. It is regarded as one of the most efficient training algorithms [TM94].

However, the Levenberg–Marquardt algorithm has its flaws. One problem is that the Hessian matrix inversion needs to be calculated each time for weight updating and there may be several updates in each iteration. For small size networks training, the computation is efficient, but for large networks, such as image recognition problems, this inversion calculation is going to be a disaster and the speed gained by second-order approximation may be totally lost. In that case, the Levenberg–Marquardt algorithm may be even slower than the steepest descent algorithm. Another problem is that the Jacobian matrix has to be stored for computation, and its size is $P \times M \times N$, where $P$ is the number of patterns, $M$ is the number of outputs, and $N$ is the number of weights. For large-sized training patterns, the memory cost for Jacobian matrix storage may be too huge to be practical. Also, the Levenberg–Marquardt algorithm was implemented only for multilayer perceptron networks.

Even though there are still some problems not solved for the Levenberg–Marquardt training, for small- and medium-sized networks and patterns, the Levenberg–Marquardt algorithm is remarkably efficient and strongly recommended for neural network training.

## References

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

[EHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, *Nature*, 323, 533–536, 1986.

[J88] P. J. Werbos, Back-propagation: Past and future, in *Proceedings of International Conference on Neural Networks*, vol. 1, pp. 343–354, San Diego, CA, 1988.

[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 Journal on Applied Mathematics*, 11(2), 431–441, June 1963.

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

[O92] M. R. Osborne, Fisher's method of scoring, *International Statistical Review*, 86, 271–286, 1992.

[PS94] V. V. Phansalkar and P. S. Sastry, Analysis of the back-propagation algorithm with momentum, *IEEE Transactions on Neural Networks*, 5(3), 505–506, March 1994.

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

[W02] B. M. Wilamowski, Neural networks and fuzzy systems, Chap. 32 in *Mechatronics Handbook*, ed. R. R. Bishop, CRC Press, Boca Raton, FL, pp. 33-1–32-26, 2002.

[W09] B. M. Wilamowski, Neural network architectures and learning algorithms, *IEEE Industrial Electronics Magazine*, 3(4), 56–63, 2009.

[W96] B. M. Wilamowski, Neural networks and fuzzy systems, Chaps. 124.1 to 124.8 in *The Electronic Handbook*, CRC Press, Boca Raton, FL, pp. 1893–1914, 1996.

[WB99] B. M. Wilamowski and J. Binfet, Do fuzzy controllers have advantages over neural controllers in microprocessor implementation in *Proceedings of 2nd International Conference on Recent Advances in Mechatronics (ICRAM'99)*, pp. 342–347, Istanbul, Turkey, May 24–26, 1999.

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

[WCKD07] B. M. Wilamowski, N. J. Cotton, O. Kaynak,, and G. Dundar, Method of computing gradient vector and Jacobian matrix in arbitrarily connected neural networks, in *IEEE International Symposium on Industrial Electronics (ISIE 2007)*, pp. 3298–3303, Vigo, Spain, June 4–7, 2007.

[WCKD08] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dundar, Computing gradient vector and Jacobian matrix in arbitrarily connected neural networks, *IEEE Transactions 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, in *1999 International Joint Conference on Neural Networks (IJCNN'99)*, pp. 1725–1728, Washington, DC, July 10–16, 1999. #295 Session: 5.1.

[WH10] B. M. Wilamowski and H. Yu, Improved computation for Levenberg Marquardt training, *IEEE Transactions on Neural Networks*, 21, 930–937, 2010.

[WHM03] B. Wilamowski, D. Hunter, and A. Malinowski, Solving parity-*n* problems with feedforward neural network, in *Proceedings of the IJCNN'03 International Joint Conference on Neural Networks*, pp. 2546–2551, Portland, OR, July 20–23, 2003.

[WJ96] B. M. Wilamowski and R. C. Jaeger, Implementation of RBF type networks by MLP networks, *IEEE International Conference on Neural Networks*, pp. 1670–1675, Washington, DC, June 3–6, 1996.

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

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