11

Neural Networks Learning

Bogdan M. Wilamowski
Auburn University

11.1 Introduction

The concept of systems that can be learned was well described over half a century ago by Nilsson [N65] in his book *Learning Machines* where he summarized many developments of that time. The publication of the Mynsky and Paper [MP69] book slowed down artificial neural network research, and the mathematical foundation of the back-propagation algorithm by Werbos [W74] went unnoticed. A decade later, Rumelhart et al. [RHW86] showed that the error back-propagation (EBP) algorithm effectively trained neural networks [W96,W02]. Since that time many learning algorithms have been developed and only a few of them can efficiently train multilayer neuron networks. But even the best learning algorithms currently known have difficulty training neural networks with a reduced number of neurons. AQ1

Similar to biological neurons, the weights in artificial neurons are adjusted during a training procedure. Some use only local signals in the neurons, others require information from outputs; some require a supervisor who knows what outputs should be for the given patterns, and other unsupervised algorithms need no such information. Common learning rules are described in the following sections.

11.2 Foundations of Neural Network Learning

Neural networks can be trained efficiently only if networks are transparent so small changes in weights' values produce changes on neural outputs. This is not possible if neurons have hard-activation functions. Therefore, it is essential that all neurons have soft activation functions (Figure 11.1).

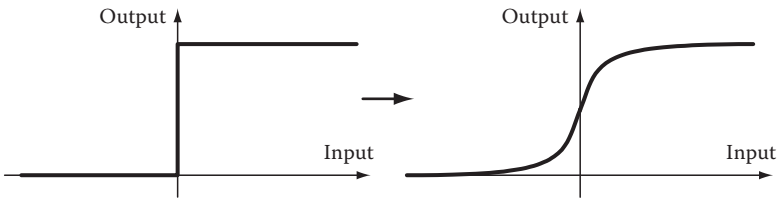

FIGURE 11.1 Neurons in the trainable network must have soft activation functions.

Neurons strongly respond to input patterns if weights' values are similar to incoming signals. Let us analyze the neuron shown in Figure 11.2 with five inputs, and let us assume that input signal is binary and bipolar (–1 or +1). For example, inputs **X** = [1, –1, 1, –1, –1] and also weights **W** = [1, –1, 1, –1, –1] then the *net* value

$$net = \sum_{i=1}^{5} x_i w_i = \mathbf{X}\mathbf{W}^T = 5 \tag{11.1}$$

This is maximal *net* value, because for any other input signals the net value will be smaller. For example, if input vector differs from the weight vector by one bit (it means the Hamming distance HD = 1), then the *net* = 3. Therefore,

$$net = \sum_{i=1}^{n} x_i w_i = \mathbf{X}\mathbf{W}^T = n - 2HD \tag{11.2}$$

where

n is the size of the input

HD is the Hamming distance between input pattern **X** and the weight vector **W**.

This is true for binary bipolar values, but this concept can be extended to weights and patterns with analog values, as long as both lengths of the weight vector and input pattern vectors are the same. Therefore, the weights' changes should be proportional to the input pattern

$$\Delta\mathbf{W} \sim \mathbf{X} \tag{11.3}$$

In other words, the neuron receives maximum excitation if input pattern and weight vector are equal. The learning process should continue as long as the network produces wrong answers. Learning may stop if there are no errors on the network outputs. This implies the rule that weight change should be

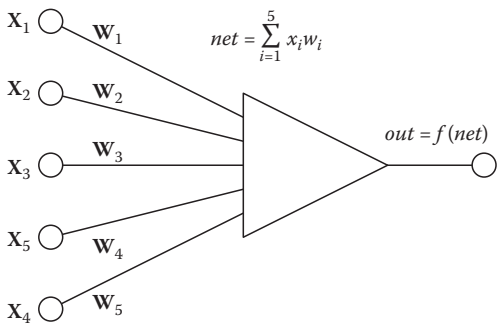

FIGURE 11.2 Neuron as the Hamming distance classifier.

proportional to the error. This rule is used in the popular EBP algorithm. Unfortunately, when errors become smaller, then weight corrections become smaller and the training process has very slow asymptotic character. Therefore, this rule is not used in advanced fast-learning algorithms such as LM [HM94] or NBN [WCKD08,W09,WY10].

11.3 Learning Rules for Single Neuron

11.3.1 Hebbian Learning Rule

The Hebb [49H] learning rule is based on the assumption that if two neighbor neurons must be activated and deactivated at the same time, then the weight connecting these neurons should increase. For neurons operating in the opposite phase, the weight between them should decrease. If there is no signal correlation, the weight should remain unchanged. This assumption can be described by the formula

$$\Delta w_{ij} = c x_i o_j \tag{11.4}$$

where

w_{ij} is the weight from ith to jth neuron
c is the learning constant
x_i is the signal on the ith input
o_j is the output signal

The training process usually starts with values of all weights set to zero. This learning rule can be used for both soft- and hard-activation functions. Since desired responses of neurons are not used in the learning procedure, this is the unsupervised learning rule. The absolute values of the weights are usually proportional to the learning time, which is undesired.

11.3.2 Correlation Learning Rule

The correlation learning rule is based on a similar principle as the Hebbian learning rule. It assumes that weights between simultaneously responding neurons should be largely positive, and weights between neurons with opposite reaction should be largely negative.

Contrary to the Hebbian rule, the correlation rule is the supervised learning. Instead of actual response, o_j, the desired response, d_j, is used for the weight-change calculation

$$\Delta w_{ij} = c x_i d_j \tag{11.5}$$

where d_j is the desired value of output signal. This training algorithm usually starts with initialization of weights to zero.

11.3.3 Instar Learning Rule

If input vectors and weights are normalized, or if they have only binary bipolar values (–1 or +1), then the *net* value will have the largest positive value when the weights and the input signals are the same. Therefore, weights should be changed only if they are different from the signals

$$\Delta w_i = c(x_i - w_i) \tag{11.6}$$

Note that the information required for the weight is taken only from the input signals. This is a very local and unsupervised learning algorithm [G69].

11.3.4 Winner Takes All

The winner takes all (WTA) is a modification of the instar algorithm, where weights are modified only for the neuron with the highest *net* value. Weights of remaining neurons are left unchanged. Sometimes this algorithm is modified in such a way that a few neurons with the highest *net* values are modified at the same time. Although this is an unsupervised algorithm because we do not know what desired outputs are, there is a need for a "judge" or "supervisor" to find a winner with a largest *net* value. The WTA algorithm, developed by Kohonen [K88], is often used for automatic clustering and for extracting statistical properties of input data.

11.3.5 Outstar Learning Rule

In the outstar learning rule, it is required that weights connected to a certain node should be equal to the desired outputs for the neurons connected through those weights

$$\Delta w_{ij} = c(d_j - w_{ij}) \tag{11.7}$$

where

d_j is the desired neuron output

c is the small learning constant, which further decreases during the learning procedure

This is the supervised training procedure, because desired outputs must be known. Both instar and outstar learning rules were proposed by Grossberg [G69].

11.3.6 Widrow–Hoff LMS Learning Rule

Widrow and Hoff [WH60] developed a supervised training algorithm that allows training a neuron for the desired response. This rule was derived so the square of the difference between the *net* and output value is minimized. The $Error_j$ for jth neuron is

$$Error_j = \sum_{p=1}^{P} \left(net_{jp} - d_{jp} \right)^2 \tag{11.8}$$

where

P is the number of applied patterns

d_{jp} is the desired output for jth neuron when pth pattern is applied

net is given by

$$net = \sum_{i=1}^{n} w_i x_i \tag{11.9}$$

This rule is also known as the least mean square (LMS) rule. By calculating a derivative of Equation 11.8 with respect to w_{ij} to find the gradient, the formula for the weight change can be found:

$$\frac{\partial Error_j}{\partial w_{ij}} = 2x_{ij} \sum_{p=1}^{P} \left(d_{jp} - net_{jp} \right) \tag{11.10}$$

so

$$\Delta w_{ij} = c x_{ij} \sum_{p=1}^{P} \left(d_{jp} - net_{jp} \right) \tag{11.11}$$

Note that weight change, Δw_{ij}, is a sum of the changes from each of the individual applied patterns. Therefore, it is possible to correct the weight after each individual pattern is applied. This process is known as *incremental updating*. The *cumulative updating* is when weights are changed after all patterns have been applied once. Incremental updating usually leads to a solution faster, but it is sensitive to the order in which patterns are applied. If the learning constant c is chosen to be small, then both methods give the same result. The LMS rule works well for all types of activation functions. This rule tries to enforce the *net* value to be equal to desired value. Sometimes, this is not what the observer is looking for. It is usually not important what the *net* value is, but it is important if the *net* value is positive or negative. For example, a very large *net* value with a proper sign will result in a correct output and in a large error as defined by Equation 11.8, and this may be the preferred solution.

11.3.7 Linear Regression

The LMS learning rule requires hundreds of iterations, using formula (11.11), before it converges to the proper solution. If the linear regression is used, the same result can be obtained in only one step [W02,AW95]. Considering one neuron and using vector notation for a set of the input patterns **X** applied through weight vector **w**, the vector of *net* values **net** is calculated using

$$\mathbf{X}\mathbf{w}^{\mathbf{T}} = \mathbf{net} \tag{11.12}$$

where

X is the rectangular array $(n + 1) \times p$ of input patterns
n is the number of inputs
p is the number of patterns

Note that the size of the input patterns is always augmented by one, and this additional weight is responsible for the threshold (see Figure 11.3).

This method, similar to the LMS rule, assumes a linear activation function, and so the *net* values should be equal to desired output values **d**

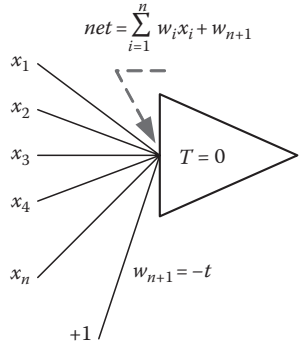

FIGURE 11.3 Single neuron with the threshold adjusted by additional weight w_{n+1}.

$$\mathbf{X}\mathbf{w}^{\mathbf{T}} = \mathbf{d} \tag{11.13}$$

Usually, $p > n + 1$, and the preceding equation can be solved only in the least mean square error sense. Using the vector arithmetic, the solution is given by

$$\mathbf{W} = \left(\mathbf{X}^{\mathbf{T}} \mathbf{X} \right)^{-1} \mathbf{X}^{\mathbf{T}} \mathbf{d} \tag{11.14}$$

The linear regression that is an equivalent of the LMS algorithm works correctly only for linear activation functions. For typical sigmoidal activation functions, this learning rule usually produces a wrong answer. However, when it is used iteratively by computing $\Delta\mathbf{W}$ instead of **W**, correct results can be obtained (see Figure 11.4).

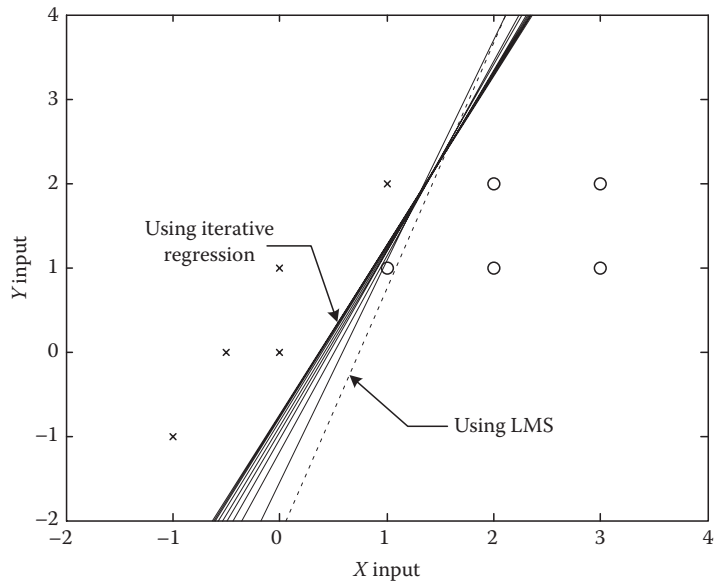

FIGURE 11.4 Single neuron training to separate patterns using LMS rule and using iterative regression with sigmoidal activation function.

11.3.8 Delta Learning Rule

The LMS method assumes linear activation function ***net*** = ***o***, and the obtained solution is sometimes far from optimum as it is shown in Figure 11.4 for a simple two dimensional case, with four patterns belonging to two categories. In the solution obtained using the LMS algorithm, one pattern is misclassified. The most common activation functions and its derivatives are for bipolar neurons:

$$o = f(net) = \tanh\left(\frac{k\,net}{2}\right) \quad f'(o) = 0.5k\left(1 - o^2\right) \tag{11.15}$$

and for unipolar neurons

$$o = f(net) = \frac{1}{1 + \exp\left(-k\,net\right)} \quad f'(o) = ko\left(1 - o\right) \tag{11.16}$$

where k is the slope of the activation function at $net = 0$. If error is defined as

$$Error_j = \sum_{p=1}^{P} \left(o_{jp} - d_{jp}\right)^2 \tag{11.17}$$

then the derivative of the error with respect to the weight w_{ij} is

$$\frac{d\,Error_j}{dw_{ij}} = 2\sum_{p=1}^{P} \left(o_{jp} - d_{jp}\right)\frac{df(net_{jp})}{dnet_{jp}}x_i \tag{11.18}$$

where $o = f(net)$ are given by (11.15) or (11.16) and the *net* is given by (11.10). Note that this derivative is proportional to the derivative of the activation function $f'(net)$.

In case of the incremental training for each applied pattern

AQ2 $$\Delta w_{ij} = cx_i f_j'(d_j - o_j) = cx_i \delta_j \tag{11.15'}$$

Using the cumulative approach, the neuron weight, w_{ij}, should be changed after all patterns are applied:

$$\Delta w_{ij} = cx_i \sum_{p=1}^{P} (d_{jp} - o_{jp})\, f_{jp}' = cx_i \sum_{p=1}^{P} \delta_{pj} \tag{11.16'}$$

The weight change is proportional to input signal x_i, to the difference between desired and actual outputs $d_{jp} - o_{jp}$, and to the derivative of the activation function f_{jp}'. Similar to the LMS rule, weights can be updated using both ways: incremental and cumulative methods. One-layer neural networks are relatively easy to train [AW95,WCM99]. In comparison to the LMS rule, the delta rule always leads to a solution close to the optimum. When the delta rule is used, then all patterns on Figure 11.4 are classified correctly.

11.4 Training of Multilayer Networks

The multilayer neural networks are more difficult to train. The most commonly used feed-forward neural network is multilayer perceptron (MLP) shown in Figure 11.5. Training is difficult because signals propagate by several nonlinear elements (neurons) and there are many signal paths. The first algorithm for multilayer training was error back-propagation algorithm [W74,RHW86], and it is still often used because of its simplicity, even though the training process is very slow and training of close-to-optimal networks seldom produces satisfying results.

11.4.1 Error Back-Propagation Learning

The delta learning rule can be generalized for multilayer networks [W74,RHW86]. Using a similar approach, the gradient of the global error can be computed with respect to each weight in the network, as was described for the delta rule. The difference is that on top of a nonlinear activation function of a neuron, there is another nonlinear term $F\{z\}$ as shown in Figure 11.6. The learning rule for EBP can be derived in a similar way as for the delta learning rule:

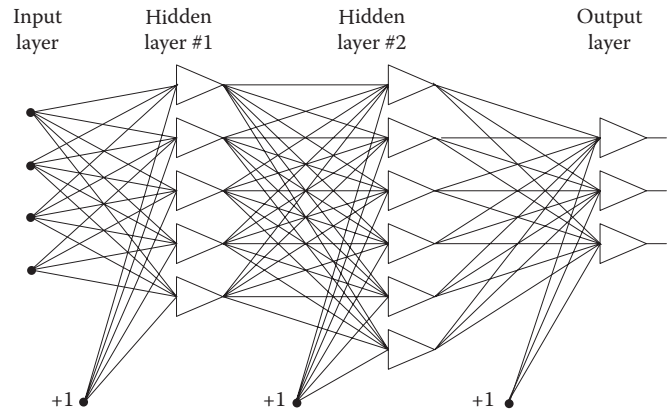

FIGURE 11.5 An example of the four layer (4-5-6-3) feed-forward neural network, which is sometimes known also as multilayer perceptron (MLP) network.

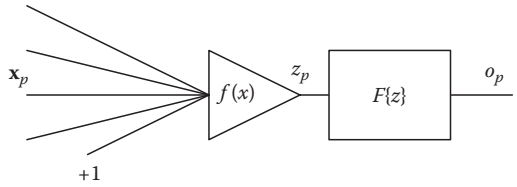

FIGURE 11.6 Error back propagation for neural networks with one output.

$$o_p = F\left\{f\left(w_1 x_{p1} + w_2 x_{p2} + \cdots + w_n x_n\right)\right\} \tag{11.17′}$$

$$TE = \sum_{p=1}^{np}\left[d_p - o_p\right]^2 \tag{11.18′}$$

$$\frac{d(TE)}{dw_i} = -2\sum_{p=1}^{np}\left[\left(d_p - o_p\right)F'\left\{z_p\right\}f'\left(net_p\right)x_{pi}\right] \tag{11.19}$$

The weight update for a single pattern p is

$$\Delta\mathbf{w}_p = \alpha\left(d_p - o_p\right)F_o'\left\{z_p\right\}f'\left(net_p\right)\mathbf{x}_p \tag{11.20}$$

In the case of batch training (weights are changed once all patterns are applied),

$$\Delta\mathbf{w} \;= \alpha\sum_{p=1}^{np}\left(\Delta\mathbf{w}_p\right) = \alpha\sum_{p=1}^{np}\left[\left(d_p - o_p\right)F_o'\left\{z_p\right\}f''\left(net_p\right)\mathbf{x}_p\right] \tag{11.21}$$

The main difference is that instead of using just derivative of activation function f' as in the delta learning rule, the product of $f'F'$ must be used. For multiple outputs as shown in Figure 11.7, the resulted weight change would be the sum of all the weight changes from all outputs calculated separately for each output using Equation 11.20. In the EBP algorithm, the calculation process is organized in such a way that error signals, Δ_p, are being propagated through layers from outputs to inputs as it is shown in Figure 11.8. Once the delta values on neurons inputs are found, then weights for this neuron are updated using a simple formula:

$$\Delta\mathbf{w}_p = \alpha\mathbf{x}_p\Delta_p \tag{11.22}$$

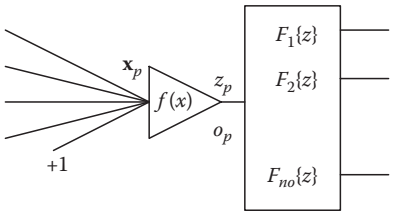

FIGURE 11.7 Error back propagation for neural networks with multiple outputs.

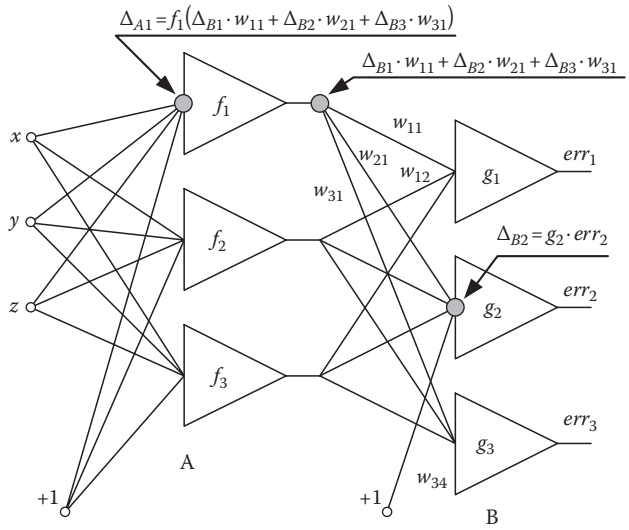

FIGURE 11.8 Calculation errors in neural network using error back-propagation algorithm. The symbols f_i and g_i represent slopes of activation functions.

where Δ was calculated using the error back-propagation process for all outputs K:

$$\Delta_p = \sum_{k=1}^{K} \left[\left(d_{pk} - o_{pk} \right) F_k' \left\{ z_{pk} \right\} f' \left(net_{pk} \right) \right] \tag{11.23}$$

(11.24)

AQ3

The calculation of the back-propagating error is kind of artificial to the real nervous system. Also, the error back-propagation method is not practical from the point of view of hardware realization. Instead, it is simpler to find signal gains $A_{j,k}$ from the input of the *jth* neuron to each of the network output k (Figure 11.9). For each pattern, the Δ value for a given neuron, j, can be now obtained for each output k:

$$\Delta_{j,k} = A_{j,k} \left(o_k - d_k \right) \tag{11.25}$$

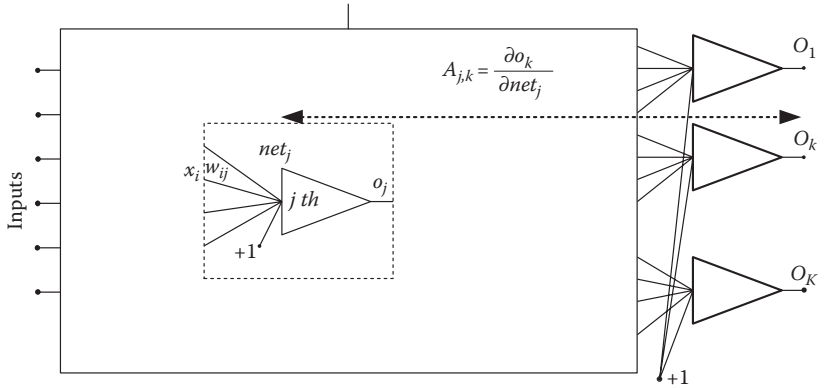

FIGURE 11.9 Finding gradients using evaluation of signal gains, $A_{j,k}$.

and for all outputs for neuron j

$$\Delta_j = \sum_{k=1}^{K} \Delta_{j,k} = \sum_{k=1}^{K} \left[A_{jk} \left(o_k - d_k \right) \right] \tag{11.26}$$

Note that the above formula is general, no matter whether neurons are arranged in layers or not. One way to find gains $A_{j,k}$ is to introduce an incremental change on the input of the *jth* neuron and observe the change in the *kth* network output. This procedure requires only forward signal propagation, and it is easy to implement in a hardware realization. Another possible way is to calculate gains through each layer and then find the total gains as products of layer gains. This procedure is equally or less computation intensive than a calculation of cumulative errors in the error back-propagation algorithm.

11.4.2 Improvements of EBP

11.4.2.1 Momentum

The back-propagation algorithm has a tendency for oscillation. In order to smooth up the process, the
AQ4 weights increment, Δw_{ij}, can be modified according to Rumelhart et al. [RHW86] (Figure 11.10):

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n) + \eta \Delta w_{ij}(n-1) \tag{11.27}$$

or according to Sejnowski and Rosenberg [SR87]

$$w_{ij}(n+1) = w_{ij}(n) + \left(1 - \alpha\right) \Delta w_{ij}(n) + \eta \Delta w_{ij}(n-1) \tag{11.28}$$

11.4.2.2 Gradient Direction Search

The back-propagation algorithm can be significantly accelerated, when after finding components of the gradient, weights are modified along the gradient direction until a minimum is reached. This process can be carried on without the necessity of computational intensive gradient calculation at each step. The new gradient components are calculated once a minimum on the direction of the previous gradient is obtained. This process is only possible for cumulative weight adjustment. One method to find a minimum along the gradient direction is the three step process of finding error for three points along gradient direction and then, using a parabola approximation, jump directly to the minimum (Figure 11.11).

11.4.2.3 Elimination of Flat Spots

The back-propagation algorithm has many disadvantages that lead to very slow convergence. One of the most painful is that in the back-propagation algorithm, it has difficulty to train neurons with the

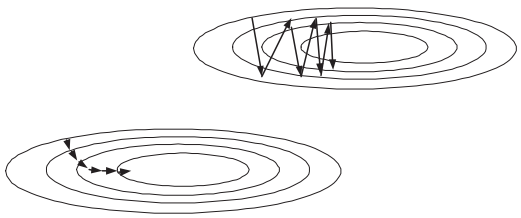

FIGURE 11.10 Solution process without and with momentum term.

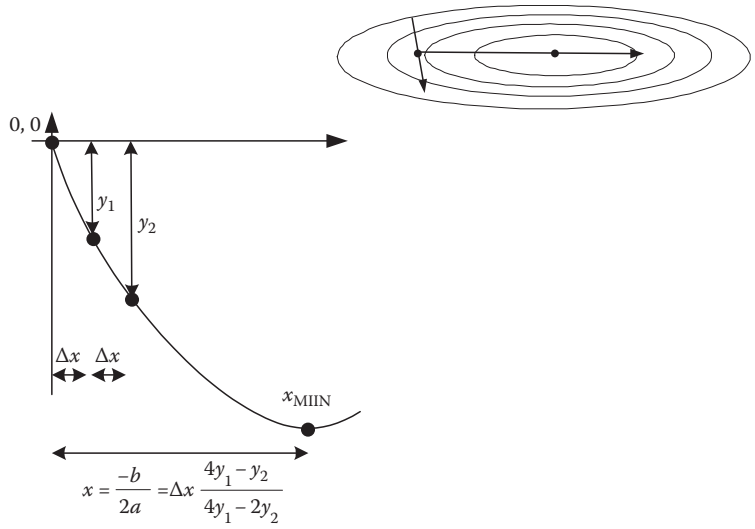

FIGURE 11.11 Search on the gradient direction before a new calculation of gradient components.

maximally wrong answer. In order to understand this problem, let us analyze a bipolar activation function shown in Figure 11.12. Maximum error equal 2 exists if the desired output is –1 and actual output is +1. At this condition, the derivative of the activation function is close to zero so the neuron is not transparent for error propagation. In other words, this neuron with large output error will not be trained or it will be trained very slowly. In the mean time, other neurons will be trained and weights of this neuron would remain unchanged.

To overcome this difficulty, a modified method for derivative calculation was introduced by Wilamowski and Torvik [WT93]. The derivative is calculated as the slope of a line connecting the point of the output value with the point of the desired value as shown in Figure 11.11:

$$f_{modif} = \frac{o_{desired} - o_{actual}}{net_{desired} - net_{actual}} \tag{11.29}$$

If the computation of the activation derivative as given by

$$f'(net) = k\left[1 - o^2\right] \tag{11.30}$$

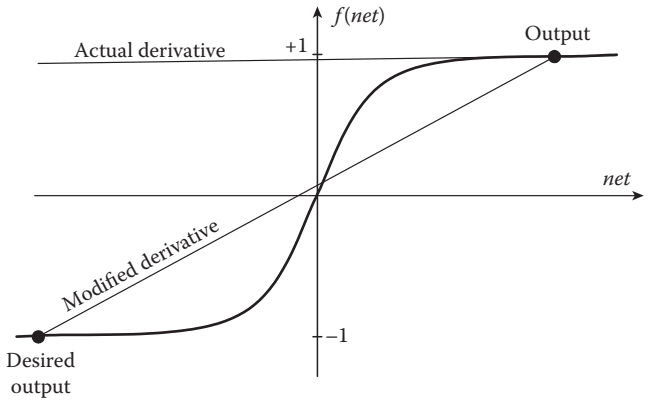

FIGURE 11.12 Lack of error back propagation for very large errors.

is replaced by

$$f'(net) = k\left[1 - o^2\left(1 - \left(\frac{err}{2}\right)^2\right)\right] \tag{11.31}$$

then for small errors

$$f'(net) = k\left[1 - o^2\right] \tag{11.32}$$

and for large errors ($err = 2$)

$$f'(net) = k \tag{11.33}$$

Note that for small errors, the modified derivative would be equal to the derivative of activation function at the point of the output value. Therefore, modified derivative is used only for large errors, which cannot be propagated otherwise.

11.4.3 Quickprop Algorithm

The fast-learning algorithm using the approach below was proposed by Fahlman [88F], and it is known as the *quickprop*:

$$\Delta w_{ij}(t) = -\alpha S_{ij}(t) + \gamma_{ij}\Delta w_{ij}(t-1) \tag{11.34}$$

$$S_{ij}(t) = \frac{\partial E\left(\mathbf{w}(t)\right)}{\partial w_{ij}} + \eta w_{ij}(t) \tag{11.35}$$

where

α is the learning constant

γ is the memory constant (small 0.0001 range) leads to reduction of weights and limits growth of weights

η is the momentum term selected individually for each weight

$$\begin{array}{lll} 0.01 < \alpha < 0.6 & \text{when} & \Delta w_{ij} = 0 \text{ or sign of } \Delta w_{ij} \\ \alpha = 0 & \text{otherwise} & \end{array} \tag{11.36}$$

$$S_{ij}(t)\Delta w_{ij}(t) > 0 \tag{11.37}$$

$$\Delta w_{ij}(t) = -\alpha S_{ij}(t) + \gamma_{ij}\Delta w_{ij}(t-1) \tag{11.38}$$

The momentum term selected individually for each weight is a very important part of this algorithm. Quickprop algorithm sometimes reduces computation time hundreds of times.

11.4.4 RPROP-Resilient Error Back Propagation

Very similar to EBP, but weights are adjusted without using values of the propagated errors but only its sign. Learning constants are selected individually to each weight based on the history:

$$\Delta w_{ij}(t) = -\alpha_{ij}\,\mathrm{sgn}\left(\frac{\partial E(\mathbf{w}(t))}{\partial w_{ij}(t)}\right) \tag{11.39}$$

$$S_{ij}(t) = \frac{\partial E(\mathbf{w}(t))}{\partial w_{ij}} + \eta w_{ij}(t) \tag{11.40}$$

$$\alpha_{ij}(t) = \begin{cases} \min\left(a \cdot \alpha_{ij}(t-1), \alpha_{\max}\right) & \text{for} \quad S_{ij}(t) \cdot S_{ij}(t-1) > 0 \\ \max\left(b \cdot \alpha_{ij}(t-1), \alpha_{\min}\right) & \text{for} \quad S_{ij}(t) \cdot S_{ij}(t-1) < 0 \\ \alpha_{ij}(t-1) & \text{otherwise} \end{cases}$$

11.4.5 Back Percolation

Error is propagated as in EBP and then each neuron is "trained" using the algorithm to train one neuron such as pseudo inversion. Unfortunately, pseudo inversion may lead to errors, which are sometimes larger than 2 for bipolar or larger than 1 for unipolar.

11.4.6 Delta-Bar-Delta

For each weight, the learning coefficient is selected individually. It was developed for quadratic error functions

$$\Delta\alpha_{ij}(t) = \begin{cases} a & \text{for} \quad S_{ij}(t-1) D_{ij}(t) > 0 \\ -b \cdot \alpha_{ij}(t-1) & \text{for} \quad S_{ij}(t-1) D_{ij}(t) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.41}$$

$$D_{ij}(t) = \frac{\partial E(t)}{\partial w_{ij}(t)} \tag{11.42}$$

$$S_{ij}(t) = (1-\xi) D_{ij}(t) + \xi S_{ij}(t-1) \tag{11.43}$$

11.5 Advanced Learning Algorithms

Let us introduce basic definitions for the terms used in advanced second-order algorithms. In the first-order methods such as EBP, which is the steepest descent method, the weight changes are proportional to the gradient of the error function:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \mathbf{g}_k \tag{11.44}$$

where **g** is gradient vector.

$$\text{gradient} \quad \mathbf{g} = \begin{matrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{matrix} \tag{11.45}$$

For the second-order Newton method, the Equation 11.42 is replaced by

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{A}_k^{-1}\mathbf{g}_k \tag{11.46}$$

where $\mathbf{A}_k$ is Hessian.

$$\mathbf{A} = \begin{matrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_2 \partial w_1} & \cdots & \frac{\partial^2 E}{\partial w_n \partial w_1} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_n \partial w_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_n} & \frac{\partial^2 E}{\partial w_2 \partial w_n} & \cdots & \frac{\partial^2 E}{\partial w_n^2} \end{matrix} \tag{11.47}$$

Unfortunately, it is very difficult to find Hessians so in the Gauss–Newton method the Hessian is replaced by product of Jacobians:

$$\mathbf{A} = 2\mathbf{J}^T\mathbf{J} \tag{11.48}$$

where

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_n} \\ \frac{\partial e_{21}}{\partial w_1} & \frac{\partial e_{21}}{\partial w_2} & \cdots & \frac{\partial e_{21}}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{M1}}{\partial w_1} & \frac{\partial e_{M1}}{\partial w_2} & \cdots & \frac{\partial e_{M1}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{1P}}{\partial w_1} & \frac{\partial e_{1P}}{\partial w_2} & \cdots & \frac{\partial e_{1P}}{\partial w_N} \\ \frac{\partial e_{2P}}{\partial w_1} & \frac{\partial e_{2P}}{\partial w_2} & \cdots & \frac{\partial e_{2P}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{MP}}{\partial w_1} & \frac{\partial e_{MP}}{\partial w_2} & \cdots & \frac{\partial e_{MP}}{\partial w_N} \end{bmatrix} \tag{11.49}$$

Knowing Jacobian, **J**, the gradient can be calculated as

$$\mathbf{g} = 2\mathbf{J}^T\mathbf{e} \tag{11.50}$$

Therefore, in Gauss–Newton algorithm, the weight update is calculated as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k\right)^{-1}\mathbf{J}_k^T\mathbf{e} \tag{11.51}$$

This method is very fast, but it works well only for systems that are almost linear and this is not true for neural networks.

11.5.1 Levenberg–Marquardt Algorithm

Levenberg and Marquardt, in order to secure convergence, modified Equation (11.46) to the form

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k + \mu\mathbf{I}\right)^{-1}\mathbf{J}_k^T\mathbf{e} \tag{11.52}$$

where the μ parameter changed during the training process. If μ = 0 algorithm works as Gauss–Newton method and for large values of μ algorithm works as steepest decent method.

The Levenberg–Marquardt algorithm was adopted for neural network training by Hagan and Menhaj [HM94], and then Demuth and Beale [DB04] adopted the LM algorithm in MATLAB® Neural Network Toolbox.

The LM algorithm is very fast, but there are several problems:

1. It was written only for MLP networks, which are not the best architectures for neural networks.
2. It can handle only problems with relatively small patterns because the size of Jacobian is proportional to the number of patterns.

11.5.2 Neuron by Neuron

The recently developed neuron by neuron (NBN) algorithm [WCKD08,W09,YW09] is very fast. Figures 11.13 and 11.14 show speed comparison of EBP and NBN algorithms to solve the parity-4 problem.

The NBN algorithm eliminates most deficiencies of the LM algorithm. It can be used to train neural networks with arbitrarily connected neurons (not just MLP architecture). It does not require to compute and to store large Jacobians, so it can train problems with basically unlimited number of patterns [WY10]. Error derivatives are computed only in forward pass, so back-propagation process is not needed. It is equally fast, but in the case of networks with multiple outputs faster than LM algorithm. It can train

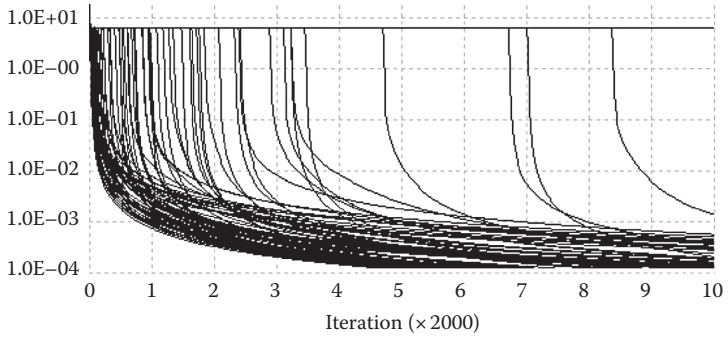

FIGURE 11.13 Sum of squared errors as a function of number of iterations for the parity-4 problem using EBP algorithm, and 100 runs.

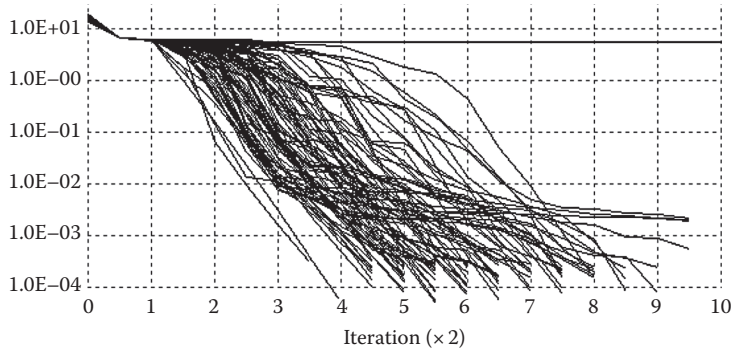

FIGURE 11.14 Sum of squared errors as a function of number of iterations for the parity-4 problem using NBN algorithm and 100 runs.

networks that are impossible to train with other algorithms. A more detailed description of the NBN
AQ5 algorithm is given in another chapter.

11.6 Warnings about Neural Network Training

It is much easier to train neural networks where the number of neurons is larger than required. But, with a smaller number of neurons the neural network has much better generalization abilities. It means it will respond correctly for patterns not used for training. If too many neurons are used, then the network can be overtrained on the training patterns, but it will fail on patterns never used in training. With a smaller number of neurons, the network cannot be trained to very small errors, but it may produce much better approximations for new patterns. The most common mistake made by many researchers is that in order to speed up the training process and to reduce the training errors, they use neural networks with a larger number of neurons than required. Such networks would perform very poorly for new patterns not used for training [W09].

11.7 Conclusion

There are several reasons for the frustration of people trying to adapt neural networks for their research:

1. In most cases, the relatively inefficient MLP architecture is used instead of more powerful topologies [WHM03] where connections across layers are allowed.
2. When a popular learning software is used, such as EBP, the training process is not only very time consuming, but frequently the wrong solution is obtained. In other words, EBP is often not able to find solutions for neural network with the smallest possible number of neurons.
3. It is easy to train neural networks with an excessive number of neurons. Such complex architectures for a given pattern can be trained to very small errors, but such networks do not have generalization abilities. Such networks are not able to deliver a correct response to new patterns, which were not used for training [W09,HW09]. In other words, the main purpose of using neural networks is missed. In order to properly utilize neural networks, its architecture should be as simple as possible to perform the required function.
4. In order of find solutions for close-to-optimal architectures, second-order algorithms such as NBN or LM should be used [WCKD07,WCKD08]. Unfortunately, the LM algorithm adopted in the popular MATLAB NN Toolbox can handle only MLP topology without connections across layers and these topologies are far from optimal.

The importance of the proper learning algorithm was emphasized, since with an advanced learning algorithm we can train those networks, which cannot be trained with simple algorithms. The software used in this work, which implements the NBN algorithm, can be downloaded from [WY09].

References

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

[CWD08] N. J. Cotton, B. M. Wilamowski, and G. Dundar, A neural network implementation on an inexpensive eight bit microcontroller, in *12th International Conference on Intelligent Engineering Systems (INES 2008)*, pp. 109–114, Miami, FL, February 25–29, 2008.

[DB04] H. Demuth and M. Beale, *Neural Network Toolbox, User's Guide, Version 4*, The MathWorks, Inc., Natick, MA, revised for version 4.0.4 edition, October 2004, http://www.mathworks.com

[F88] S. E. Fahlman, Faster-learning variations on back-propagation: An empirical study, in *Connectionist Models Summer School*, eds. T. J. Sejnowski G. E. Hinton, and D. S. Touretzky, Morgan Kaufmann, San Mateo, CA, 1988.

[G69] S. Grossberg, Embedding fields: A theory of learning with physiological implications, *Journal of Mathematical Psychology*, 6, 209–239, 1969.

[H49] D. O. Hebb, *The Organization of Behavior, a Neuropsychological Theory*, John Wiley, New York, 1949.

[HM94] M. T. Hagan and M. Menhaj, Training feedforward networks with the Marquardt algorithm, *IEEE Transactions on Neural Networks*, 5(6), 989–993, 1994.

[HW09] 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.

[K88] T. Kohonen, The neural phonetic typewriter, *IEEE Computer*, 27(3), 11–22, 1988.

[MP69] M. Minsky and S. Papert, *Perceptrons*, MIT Press, Cambridge, MA, 1969.

[RHW86] D. E., Rumelhart, G. E. Hinton, and R. J. Wiliams, Learning representations by back-propagating errors, *Nature*, 323, 533–536, 1986.

[N65] N. J. Nilson, *Learning Machines: Foundations of Trainable Pattern Classifiers*, McGraw Hill, New York, 1965.

[SR87] T. J. Sejnowski and C. R. Rosenberg, Parallel networks that learn to pronounce English text. *Complex Systems*, 1, 145–168, 1987.

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

[W07] B. M. Wilamowski, Neural networks and fuzzy systems for nonlinear applications, in *11th International Conference on Intelligent Engineering Systems (INES 2007)*, pp. 13–19, Budapest, Hungary, June 29, 2007–July 1, 2007.

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

[W74] P. Werbos, Beyond regression: New tools for prediction and analysis in behavioral sciences, PhD diss., Harvard University, Cambridge, MA, 1974.

[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, *IEEE International Symposium on Industrial Electronics (ISIE 2007)*, pp. 3298–3303, Vigo, Spain, 4–7 June 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 *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 International Joint Conference on Neural Networks (IJCNN'03)*, 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, in *IEEE International Conference on Neural Networks*, pp. 1670–1675, Washington, DC, June 3–6, 1996.

[WJK99] B. M. Wilamowski, R. C. Jaeger, and M. O. Kaynak, Neuro-fuzzy architecture for CMOS implementation, *IEEE Transaction on Industrial Electronics*, 46(6), 1132–1136, December 1999.

[WK00] B. M. Wilamowski and O. Kaynak, Oil well diagnosis by sensing terminal characteristics of the induction motor, *IEEE Transactions on Industrial Electronics*, 47(5), 1100–1107, October 2000.

[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.

[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; also in C. H. Dagli, ed., *Intelligent Engineering Systems through Artificial Neural Networks*, vol. 3, pp. 175–180, ASME Press, New York, 1993.

[WY09] NNT—Neural Network Trainer, http://www.eng.auburn.edu/~wilambm/nnt/

[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.