Methods to Speed Up Error Back-Propagation Learning Algorithm

DILIP SARKAR

University of Miami

Error back propagation (EBP) is now the most used training algorithm for feedforward artificial neural networks (FFANNs). However, it is generally believed that it is very slow if it does converge, especially if the network size is not too large compared to the problem at hand. The main problem with the EBP algorithm is that it has a constant learning rate coefficient, and different regions of the error surface may have different characteristic gradients that may require a dynamic change of learning rate coefficient based on the nature of the surface. Also, the characteristic of the error surface may be unique in every dimension, which may require one learning rate coefficient for each weight. To overcome these problems several modifications have been suggested. This survey is an attempt to present them together and to compare them. The first modification was momentum strategy where a fraction of the last weight correction is added to the currently suggested weight correction. It has both an accelerating and a decelerating effect where they are necessary. However, this method can give only a relatively small dynamic range for the learning rate coefficient. To increase the dynamic range of the learning rate coefficient, such methods as the "bold driver" and SAB (self-adaptive back propagation) were proposed. A modification to the SAB that eliminates the requirement of selection of a "good" learning rate coefficient by the user gave the SuperSAB. A slight modification to the momentum strategy produced a new method that controls the oscillation of weights to speed up learning. Modification to the EBP algorithm in which the gradients are rescaled at every layer helped to improve the performance. Use of "expected output" of a neuron instead of actual output for correcting weights improved performance of the momentum strategy. The conjugate gradient method and "self-determination of adaptive learning rate" require no learning rate coefficient from the user. Use of energy functions other than the sum of the squared error has shown improved convergence rate. An effective learning rate coefficient selection needs to consider the size of the training set. All these methods to improve the performance of the EBP algorithm are presented here.

Categories and Subject Descriptors: G.1.6 [**Numerical Analysis**]: Optimization—*gradient methods*; I.2.6 [**Artificial Intelligence**]: Learning—*connectionism*, *neural nets*

General Terms: Algorithms, Simulation, Theory

Additional Key Words and Phrases: Adaptive learning rate, artificial neural networks (ANNs), conjugate gradient method, energy function, error back-propagation learning, feedforward networks, learning rate, momentum, oscillation of weights, training set size

---

Author's address: D. Sarkar, Department of Mathematics and Computer Science, University of Miami, Coral Gables, FL 33124.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1995 ACM 0360-0300/95/1200-0519 $03.50

CONTENTS

1. INTRODUCTION
2 ORIGINAL EBP ALGORITHM
3. AN OVERVIEW OF DIFFERENT MODIFICATIONS
4. MODIFICATIONS TO ORIGINAL EBP ALGORITHM
   4.1 Momentum Strategy
   4.2 Adaptive Learning Rate Coefficient
   4.3 Controlling Oscillation of Weights
   4.4 Rescaling of Variables
   4.5 EBP with Expected Source Values
   4.6 Self-Determination of Adaptive Learning Rates
   4.7 Conjugate Gradient Methods
   4.8 Different Energy Functions
   4.9 Effect of Training Set Size
5. DISCUSSION

1. INTRODUCTION

Artificial neural networks (ANNs) are mathematical models developed to mimic certain information storing and processing capabilities of the brain of higher animals. These models are developed with a quite different philosophy of information processing from that of conventional computers. It is hoped that they will overcome the conventional computers' limitation on "intelligent" information processing capability. They are assumed to be assembled from neuron-like cells that are connected by links with adjustable strengths/weights. The most attractive characteristic of ANNs is that they can be taught to perform computational tasks using some learning algorithm and few examples. When designed carefully, they can be simulated on digital computers or can be implemented using digital/analog VLSI.

Although interest of the research community in ANNs as a means for intelligent computing had existed for over 30 years (see Widrow and Lehr [1990]), there is little doubt that Rumelhart, McClelland, and the PDP research group [1986] should be credited with revitalizing wide interest in it. The different models and their applications can be found in many books and in such surveys as Hinton [1989], Lippmann [1987], and Widrow and Lehr [1990]. This article concentrates only on feedforward ANNs (FFANNs) and error-back propagation (EBP) learning algorithms for them.

The EBP learning rule for multilayer FFANNS, popularly known as the *back-propagation algorithm*, is a generalization of the *delta* learning rule for single-layer ANNs. The delta learning rule is so called because the amount of learning is proportional to the *difference* (or delta) between the actual output and the desired output provided by a teacher. As the title of Werbos' [1974] thesis suggests, the BEP learning algorithm goes "beyond regression." Werbos [1990] believes that "backpropagation has many applications which do not involve neural networks as such."

EBP is now the most popular learning algorithm for multilayer FFANNs because of its simplicity, its power to extract useful information from examples, and its capability of storing information implicitly in the connecting links in the form of weights. Thus, unlike expert systems, where knowledge as a set of rules is necessary, FFANNs need no explicit rules to perform classification tasks. The development of EBP theory is related to many disciplines, and it took a long time to reach its present form. Basic elements of the theory, as pointed out by le Cun [1988], can be traced back to the book of Bryson and Ho [1969]. It was more explicitly stated by Werbos [1974], Parker [1985], le Cun [1986], and Rumelhart et al. [1985]. However, the book by the PDP research group [1986] helped the EBP algorithm to spread widely and achieve its present popularity.

The original version of the EBP learning algorithm has been of great concern to practical users for many reasons: (1) it is extremely slow if it does converge, (2) it may get stuck in local minima before learning all the examples, (3) it is sensitive to initial conditions, (4) it may start oscillating, and so on. Several methods have been proposed to improve

the performance of the EBP algorithm. The most important suggested modifications to the original EBP algorithm are presented here. Also, the synergy among the methods is discussed so that they can be combined to possibly obtain considerable improvement over any single one.

In the next section, following Rumelhart et al. [1986], the original version of the EBP algorithm is presented. An overview of different modifications to the original EBP algorithm and their relationships is presented in Section 3. Different modifications to the original EBP algorithm for speeding up the training process are presented in Section 4. The modifications to EBP described in Section 4 are compared in Section 5.

2. ORIGINAL EBP ALGORITHM

A FFANN consists of a set of simple neurons (or processing units). A neuron is described with two entities: its activation function and its bias or threshold. It accepts inputs from output of other neurons or from external sources. The set of neurons in a FFANN is partitioned into several disjoint subsets. All neurons in one partition are assigned to the same layer. For instance, in a two-layer FFANN the neurons are partitioned into two disjoint subsets, one for each layer. Each neuron in one layer is connected to all neurons in the next layer by links with adjustable strength/weights. As shown later, a connection weight may change during the learning process. Let u_i^l be the ith neuron in lth layer. The input layer is called the 0th layer and the output layer is called the oth layer. Let n_l be the number of neurons in lth layer. The weight of the link between neuron u_j^l in layer l and neuron u_i^{l+1} in layer $l+1$ is denoted by w_{ij}^l. Let $\{x_1, x_2, \ldots, x_P\}$ be the set of input patterns that the network is supposed to learn to classify and let $\{d_1, d_2, \ldots, d_P\}$ be corresponding desired output patterns. It should be noted that x_p is an n_0-dimension vector $(x_{1p}, x_{2p}, \ldots, x_{n_0 p})$ and d_p is an n_o-dimension vector $(d_{1p}, d_{2p}, \ldots, d_{n_o p})$. The pair (x_p, d_p) is called a training pattern.

Example. A three-layer (including input layer) FFANN is shown in Figure 1. Input (0th), 1st, and output layers have two, two, and one neurons, respectively. Thus for this FFANN, $n_0 = 2$, $n_1 = 2$, and $n_o = 1$. For illustrative purposes in the rest of the article we use the following set of inputs and desired output patterns.

$$\left\{x_1 = \begin{bmatrix}0\\0\end{bmatrix}, \quad x_2 = \begin{bmatrix}0\\1\end{bmatrix}, \quad x_3 = \begin{bmatrix}1\\0\end{bmatrix}, \quad x_4 = \begin{bmatrix}1\\1\end{bmatrix}\right\},$$

$$\{d_1 = [0], \quad d_2 = [1], \quad d_3 = [1], \quad d_4 = [0]\}.$$

The output of a neuron u_i^0 is the input x_{ip} (for input pattern p). For other layers, the net input to a neuron is usually computed as the sum of all inputs to it. With the notations we have introduced, the net input net_{pi}^{l+1} to a neuron u_i^{l+1} for the input pattern x_p is given by

$$net_{pi}^{l+1} = \sum_{j=1}^{n_l} w_{ij}^l out_j^l + bias_i^{l+1}, \quad (1)$$

where out_j^l is the output of neuron u_j^l of layer l, and $bias_i^{l+1}$ is the bias of neuron u_i^{l+1} of layer $l+1$. The net input to a neuron is used by its activation function to produce an output. Any nonlinear function can be used as an activation function. The sigmoidal function is the most commonly used activation function. Using this function, the output of a neuron u_i^l with net input net_{pi}^l is given by

$$out_{pi}^l = f\left(net_{pi}^l\right) = \frac{1}{1 + e^{-\beta net_{pi}^l}}, \quad (2)$$

where β determines the steepness of the activation function. A lower value of β gives a smoother transition from a lower to a higher activation level as the net_{pi}^l changes its value; a higher value for β will cause a step-like transition in the activation level. In the rest of the article we assume that the value of $\beta = 1$.

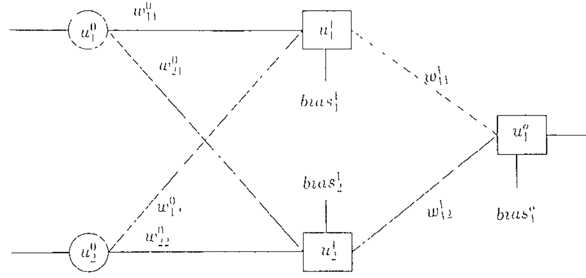

Figure 1. 3-layer neural network.

The EBP algorithm for FFANNs proceeds by representing an input pattern to the input (or the 0th) layer, after which the network produces an output. This output is compared to a desired or target output. The difference between the target output and actual network is called error. Formally, the error ε_{pi} for the ith neuron u_i^o of the output layer o for the input training pair (x_p, d_p) is computed as

$$\varepsilon_{pi} = d_{pi} - out_{pi}^o. \qquad (3)$$

An objective of a learning algorithm is to use this error to adjust the weights in such a way that the error gradually reduces. The training process stops when the error of every neuron for every training pair is reduced to an acceptable level, or when no further improvement is obtained. In the latter case, the network is again initialized with small weights and the training process starts afresh.

Example (con't.). To continue further with the example, we need to assign some values to each of the connections and biases. Let connection weights and bias for neuron 1 in layer 1 be $w_{11}^0 = 0.3$, $w_{12}^0 = -0.6$, and $bias_1^1 = 0.4$; for neuron 2 in layer 1 be $w_{21}^0 = 0.2$, $w_{22}^0 = 0.8$, and $bias_1^2 = -0.2$; and for neuron 1 in the output layer be $w_{11}^1 = 0.7$, $w_{12}^1 = 0.9$, and $bias_1^o = -0.7$.

Using these values for the connection weights and biases, one can compute net inputs and corresponding outputs of the neurons for every input pattern. For example, considering the pattern x_1 as input to the network we get net inputs for two neurons in the 1st layer as $net_{11}^1 = \sum_{j=1}^2 w_{1j}^0 x_{j1} + bias_1^1 = 0.4$, and $net_{12}^1 = \sum_{j=1}^2 w_{2j}^0 x_{j1} + bias_2^1 = -0.2$.

Now applying these net inputs to the activation function we get the output of the neurons in the 1st layer as $out_{11}^1 = f(net_{11}^1) = 1/(1+e)^{-net_{11}^1} = 0.599$, and $out_{12}^1 = f(net_{12}^1) = 1/(1+e)^{-net_{12}^1} = 0.450$. These outputs of the 1st layer act as the input to the next layer (which is the output layer in this example) and for the neuron in the output layer we get $net_{11}^o = 0.124$ and $out_{11}^o = 0.531$. Because the desired output with input pattern x_1 is $d_1 = [0]$, the error $\varepsilon_{11} = d_{11} - out_{11}^o = -0.469$.

For measuring the performance of the learning algorithm, an objective function is defined in such a way that as the error reduces so does the value of the objective. Thus a training algorithm decides the change of weights using some procedure that guarantees no increase in the objective function's value. The objective functions are known as the energy functions (from the name of similar functions in physics). Rumelhart et al. [1986] in their original EBP algorithm used the sum of the squared error as the energy function.

$$E = \frac{1}{2} \sum_{p=1}^{P} \sum_{i=1}^{n_o} (\varepsilon_{pi})^2. \qquad (4)$$

The energy function can be defined for only one training pattern pair (x_p, d_p) as

$$E_p = \frac{1}{2} \sum_{i=1}^{n_o} (\varepsilon_{pi})^2. \qquad (5)$$

There are two versions of the EBP algorithm, *online* and *batch*. In the online EBP algorithm, the weights are updated using the error corresponding to every training pattern. This method uses the energy function defined by Equation (5). However, in the batch EBP algorithm, the weights are updated after accumulating errors corresponding to all input patterns, and thus make use of the energy function defined by Equation (4). The weight updating rule for the batch mode is given by

$$w_{ij}^{l}[s+1] = w_{ij}^{l}[s] + \eta G_{ij}^{l}[s], \quad (6)$$

where

$$G_{ij}^{l}[s] = -\frac{\partial E}{\partial w_{ij}^{l}[s]}, \quad (7)$$

the index s labels the iteration or step number in the learning process, and η is the step size or learning rate coefficient. For the online EBP algorithm, the weight-updating rule is obtained if the energy E in Equation (7) is replaced by energy E_p to compute an energy-weight gradient:

$$G_{ij}^{l}[s] = -\frac{\partial E_p}{\partial w_{ij}^{l}[s]}. \quad (8)$$

Now one can use the chain rule of differentiation to write

$$\frac{\partial E_p}{\partial w_{ij}^{l}[s]} = \frac{\partial E_p}{\partial net_{pi}^{l+1}} \times \frac{\partial net_{pi}^{l+1}}{\partial w_{ij}^{l}[s]}. \quad (9)$$

Referring to expression for net_{pi}^{l+1} from Equation (1), it is easy to find that

$$\frac{\partial net_{pi}^{l+1}}{\partial w_{ij}^{l}[s]} = out_{pj}^{l}[s]. \quad (10)$$

Expressions for $\partial E_p / \partial net_{pi}^{l+1}$ are different for hidden-layer neurons and output-layer neurons. Thus it is convenient to denote $\partial E_p / \partial net_{pi}^{l+1} = -\delta_{pi}^{l+1}$ for both hidden-layer and output-layer neurons to obtain the weight update rule for all neurons as

$$w_{ij}^{l}[s+1] = w_{ij}^{l}[s] + \eta \delta_{pi}^{l+1} out_{pj}^{l}[s]. \quad (11)$$

The chain rule of differentiation is used to evaluate the expressions for δ_{pi}^{l+1}. We skip derivations, but write the expressions for them. It is assumed that activation levels of neurons are computed using the standard sigmoid function [Equation (2)] with $\beta = 1$:

$$\delta_{pi}^{o} = out_{pi}^{o}(1 - out_{pi}^{o})\varepsilon_{pi} \quad (12)$$

for neuron i in the output layer, and

$$\delta_{pi}^{l+1} = out_{pi}^{l+1}\left(1 - out_{pi}^{l+1}\right) \sum_{k=1}^{n_{l+2}} w_{ki}^{l+1} \delta_{pk}^{l+2} \quad (13)$$

for neuron i in hidden layers. It is clear from Equation (12) that the error signal δ_{pi}^{o} is computed directly from the corresponding error $\varepsilon_{pi} = d_{pi} - out_{pi}^{o}$, but the error signal for neurons in hidden layers is obtained by propagating error signals of the neurons in the layer just after it: thus the name *Error Back Propagation Learning Algorithm*. Figure 2 illustrates computation of error signals δ_{pi}^{l+1} for a neuron i in layer $l+1$.

Combining Equation (11) with Equation (12), one can compute weight corrections for output-layer neurons. Similarly, the weight correction for hidden-layer neurons can be computed by combining Equation (11) with Equation (13).

The training process of a FFANN is an iterative process. Each iteration consists of the following steps:

(1) Select a training pattern (x_p, d_p).
(2) **Forward Pass:** Present the input pattern x_p to the input of the FFANN and determine its output. Equations (1) and (2) are necessary to do the computation for this pass.
(3) **Backward Pass:** Calculate the error signal for each neuron. It should be noted that error signal calculation starts from the output layer and proceeds in the backward direction to hidden layers. For instance, if a FFANN has four layers—one input

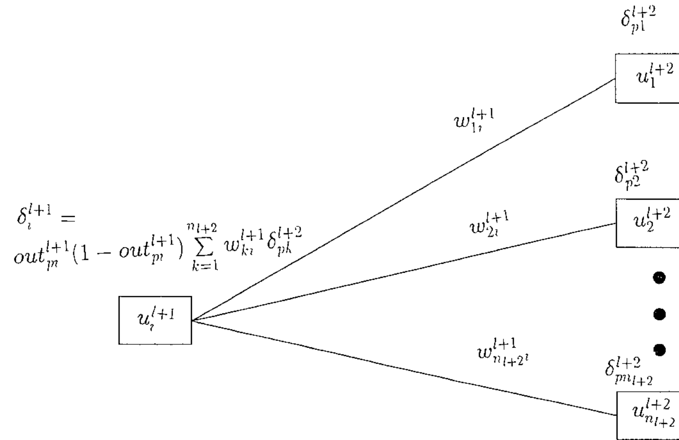

Figure 2. Computation of error signal for a hidden-layer neuron i in layer $l = 1$.

layer, two hidden layers, and one output layer—then (a) error signals for the output-layer neurons are computed first, then (b) error signals for the second hidden-layer neurons (next to the output layer) are computed, and finally (c) error signals for the first hidden-layer neurons (closest to the input layer) are computed.

(4) **Weight Adjustments:** Adjust the weights using Equation (11).

Example (con't.). Now we continue with the example started earlier. Before this, what we have completed are the first two steps of the preceding algorithm. We selected x_1 as the input pattern and completed the forward pass. We also computed ε_{11}. Next we complete the backward pass by computing error signals for the output-layer neurons [using Equation (12)] and the hidden-layer neurons [using Equation (13)]. The error signal for the output layer neuron u_1^o is $\delta_{11}^o = out_{11}^o(1 - out_{pi}^o)\varepsilon_{11} = -0.132$. Error signals for neurons u_1^1 and u_2^1 in the hidden layer are $\delta_{11}^1 = out_{11}^1(1 - out_{11}^1)\Sigma_{k=1}^1 w_{k1}^1 \delta_{1k}^o = -0.022$, and $\delta_{12}^1 = out_{11}^1(1 - out_{11}^1)\Sigma_{k=1}^1 w_{k2}^1 \delta_{1k}^o = -0.029$, respectively. This completes the third step.

In the last step, using Equation (11) the weights are adjusted to obtain $w_{11}^0 = 0.3$, $w_{12}^0 = -0.6$, and $bias_1^1 = 0.377759$ for neuron u_1^1 in the hidden layer; $w_{21}^0 = 0.2$, $w_{22}^0 = 0.8$, and $bias_2^1 = -0.229459$ for neuron u_2^1 in the hidden layer; $w_{11}^1 = 0.620827$, $w_{12}^1 = 0.840468$, and $bias_1^o = -0.832244$, for neuron u_1^o in the output layer. We used the learning rate coefficient $\eta = 1$ for weight adjustment. (For consistency, we use learning rate coefficient $\eta = 1$ in all our examples that follow.) The completes one iteration with input pattern x_1. Because the weights were updated after presenting only one pattern, this is the online version of the EBP learning algorithm.

We present input patterns x_1, x_2, x_3, and x_4 in that order to the network at each cycle for easy reproduction of the results. (However, they could be presented in any order, but possibly affecting the final outcome.) It was assumed that the network had learned all the patterns if the absolute value of the error was less than 0.3 for each pattern. The online algorithm required 420 cycles to learn all the patterns. The final connection weights of the network were as follows: $w_{11}^0 = 3.30961$, $w_{12}^0 = -3.74105$, and $bias_1^1 = -2.06024$ for neuron u_1^1 in the hidden layer; $w_{21}^0 = -3.32466$, $w_{22}^0 = 2.99838$, and $bias_2^1 = -1.81799$ for neuron u_2^1 in the hidden layer; $w_{11}^1 = 3.75645$, $w_{12}^1 = 3.73334$, and $bias_1^o =$

-1.74346, for neuron u_1^o in the output layer.

Batch Version. Now we illustrate the batch version of the EBP learning algorithm with the same sample FFANN. It may be recalled that in the batch version: each pattern is presented once and weight correction is calculated, but the weights are not actually adjusted; and calculated weight corrections for each weight are added together for all the patterns and then weights are adjusted only once using the cumulative correction. As for the online version, we present input patterns x_1, x_2, x_3, and x_4 in that order to the network at each cycle. However, it should be noted that in the batch version of the EBP algorithm the order of presentation of patterns to the network does not matter because we adjust weights only once after all the patterns have been presented and cumulative errors for the weights have been computed.

After the first cycle, the adjusted weights of the network are: $w_{11}^0 = 0.292659$, $w_{12}^0 = -0.605081$, and $bias_1^1 = 0.389771$ for neuron u_1^1 in the hidden layer; $w_{21}^0 = 0.197833$, $w_{22}^0 = 0.796052$, and $bias_2^1 = -0.208627$ for neuron u_2^1 in the hidden layer; $w_{11}^1 = 0.671218$, $w_{12}^1 = 0.871077$, and $bias_1^o = -0.750384$, for neuron u_1^o in the output layer. This version of the algorithm required 491 cycles to attain our desired error level (of less than absolute value 0.3) for all the patterns. The weights of the network after 491 cycles are given next. It should be noted that the weights obtained are quite different from the ones obtained by the online version. The final connection weights of the network were as follows: $w_{11}^0 = -2.45067$, $w_{12}^0 = -2.47898$, and $bias_1^1 = 3.51763$ for neuron u_1^1 in the hidden layer; $w_{21}^0 = 5.16276$, $w_{22}^0 = 5.33647$, and $bias_2^1 = -1.86421$ for neuron u_2^1 in the hidden layer; $w_{11}^1 = 4.12527$, $w_{12}^1 = 4.30574$, and $bias_1^o = -5.9717$, for neuron u_1^o in the output layer.

Values of several parameters are of importance in implementing the EBP algorithm. The initial value of weights should be small and randomly chosen [Rumelhart et al. 1986] to avoid the symmetry problem.[1] Sietsma and Dow [1991] have used uniformly distributed random numbers between -0.5 and 0.5 as bias weights, and between $(-0.5/n_l)$ and $(0.5/n_l)$ as initial weights for links between layers l and $l+1$. Note that the division by n_l, the number of inputs, eliminates the effect of the number of inputs to a neuron. The η value plays a very important role. A smaller value of η makes learning slow, but too large a value causes oscillation, preventing the network from learning the task [Rumelhart et al. 1986]. In practice, the most effective value of η depends on the problem. For example, Fahlmann [1988] has reported 0.9 as the learning rate coefficient for one problem, and Hinton [1987] has reported 0.002 as the learning rate coefficient for another problem. This variation in the value of the learning rate coefficient for faster training of FFANNs has drawn the attention of many researchers. Various methods have been proposed to speed up the EBP algorithm. In the next section, a brief overview of these methods is presented.

3. AN OVERVIEW OF DIFFERENT MODIFICATIONS

It has been realized that the original EBP algorithm is too slow for most practical applications, and many modifications have been suggested to speed it up. Before describing different modifications to the original EBP algorithm in Section 4, we briefly discuss them here to see how they are related to one another.

A classification tree is shown in Figure 3 to help our discussion. As can be seen from the figure, the methods proposed to speed up the original EBP can be divided into groups: (1) Dynamic learning rate adjustments, where values

[1] If all initial weights are equal and the final solution requires different weights, then EBP fails to train the net. This is known as the symmetry problem.

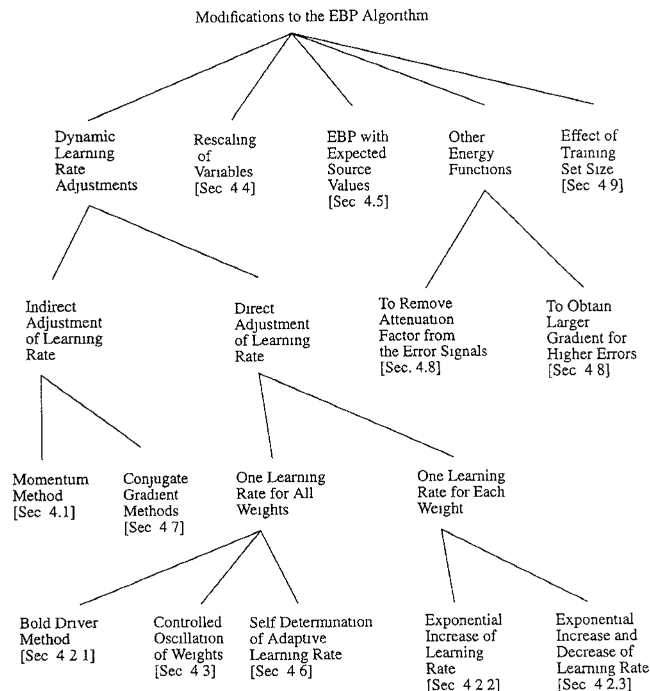

Figure 3. Classification of different methods to speed up EBP learning algorithms.

of learning rate coefficients are adjusted as the learning continues. Adjustment of learning rate coefficients could be done indirectly or directly. Examples of indirect adjustment of learning rate are the *momentum method* (see Section 4.1) and *conjugate gradient methods* (see Section 4.7). Indirect learning rate adjustment methods, especially the momentum method, have speeded up the EBP algorithm in some cases. However, they were not satisfactory enough in many cases, and hence methods for direct adjustment of the learning rate coefficient have been proposed: one learning rate coefficient for all the weights (see Sections 4.2.1, 4.3, and 4.6 for details) and one learning rate coefficient for each weight (see Sections 4.2.2 and 4.2.3 for details).

Methods other than adjustment of learning rate coefficients have shown noticeable speedup for some applications. (2) Rescaling of (error signal) variables is based on the observation that error signals for hidden-layer neurons are attenuated very quickly. The amount of attenuation increases exponentially as the distance of the hidden layer from the output layer increases linearly. Thus, to compensate for the attenuation the error signals are scaled up (see Section 4.4). (3) Another source of improvement is to use the expected output of a neuron instead of the current output (see Section 4.5). (4) The original EBP algorithm uses sum of the squared errors as its energy function. However, any function of error that decreases as error is reduced can be used as an energy function. Several other energy functions have been used to speed up EBP learning (see Section 4.8). (5) Analysis and experimentation have shown that the size of the training set is a parameter that could be used to determine a learning rate (see Section 4.9). When sizes of training sets for different classes are different, use of different learning rate coefficients for them could speed up EBP learning.

In the following sections the methods for modifying the original EBP algorithm in order to speed up the learning process are presented.

4. MODIFICATIONS TO ORIGINAL EBP ALGORITHM

This section presents a survey of the modifications to the original EBP algorithm that have been suggested to speed up the training of FFANNs. As we proceed it will become clear that some of these methods can be combined to obtain an algorithm that not only provides faster training but also helps automate the selection of certain problem-sensitive parameters.

4.1 Momentum Strategy

The learning strategy used in the original EBP algorithm is actually a *gradient descent* on the multidimensional energy surface in the weight space. A closer look on the effect of the value of the learning rate coefficient η reveals that in an area of the energy surface where the gradient is not changing sign, a larger value of η reduces the energy function faster; on the other hand, near an area where the sign of the gradient is changing quickly, a smaller value of η keeps the descent along the energy surface. This disparate need for the value of η suggests a method that adapts the value of η dynamically depending on the characteristic of the energy surface. The momentum strategy implements such a variable learning rate coefficient implicitly by adding a fraction of the last weight change to the current direction of movement in the weight space, and it is a slight change to the weight updating rule of the original EBP [Equation (7)]. The new equation for weight change is given by

$$\Delta w_{ij}^{l}[s+1] = \eta G_{ij}^{l}[s] + \alpha \Delta w_{ij}^{l}[s], \tag{14}$$

where α is a momentum coefficient that can have a value between zero and one.

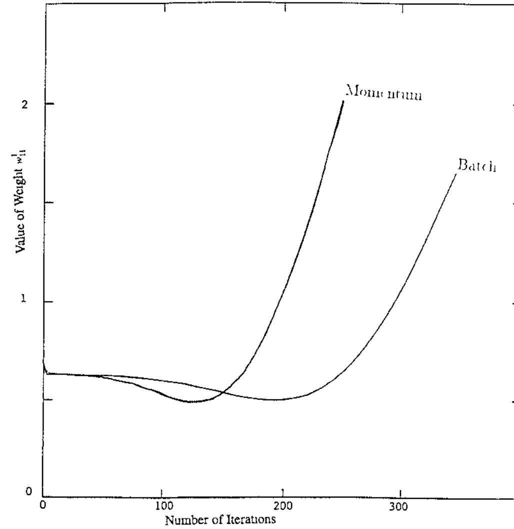

Figure 4. Comparison of two values of connection weights for batch version of EBP with and without momentum term.

Note that we get Equation (7) from the following equation:

$$w_{ij}^{l}[s+1] = w_{ij}^{l}[s] + \eta G_{ij}^{l}[s] + \alpha\left(w_{ij}^{l}[s] - w_{ij}^{l}[s-1]\right), \tag{15}$$

when $\Delta w_{ij}^{l}[k]$ is defined as

$$\Delta w_{ij}^{l}[k] = w_{ij}^{l}[k] - w_{ij}^{l}[k-1]. \tag{16}$$

Example (con't.). Figure 4 shows the *desired* effect of using a momentum term in EBP learning. We used momentum coefficient $\alpha = 0.5$, and kept all initial conditions and learning rate coefficients the same as before. We plotted the connection weight w_{11}^{1} as a function of the training step for both training with and without the momentum term. It can be seen from these curves that when the sign of the gradient does not change, the momentum method increases the amount of weight adjustments. This can be viewed as a change of *effective* learning rate coefficient to a larger value. Thus, as expected, the desired error level was reached much faster (after only 328

iterations) with the momentum term than without it (491 iterations). However, it should be noted that use of a momentum term may not always reduce the number of required iterations. We were lucky in this example! The final weights after the desired error level was reached are: $w^0_{11} = -2.44177$, $w^0_{12} = -2.4702$, and $bias^1_1 = 3.50069$ for neuron u^1_1 in the hidden layer; $w^0_{21} = 5.2046$, $w^0_{22} = 5.3867$, and $bias^1_2 = -1.9496$ for neuron u^1_2 in the hidden layer; $w^1_{11} = 4.14685$, $w^1_{12} = 4.28035$, and $bias^o_1 = -5.95402$, for neuron u^o_1 in the output layer. The final weights are very similar to those obtained from the batch version of the EBP, but slightly different. However, this may not always be the case.

A better insight into the effect of the momentum term in the finite difference equation (15) can be obtained by using a technique such as the Z transform [Eaton and Olivier 1992]. The Z transform of Equation (15) is:

$$w^l_{ij} = \left[\frac{\eta}{z-1}\right]\left[\frac{z}{z-\alpha}\right]G^l_{ij} + w^l_{ij}[0], \tag{17}$$

where $w^l_{ij}[0]$ is the initial weight. This Z transform can be viewed as a cascade of an integrator with gain η and a low-pass filter with gain $1/(1-\alpha)$. The integrator accumulates individual G^l_{ij} terms. The gain and cutoff point of the low-pass filter are controlled by the single parameter α. The maximum total gain from the system is $(\eta/(1-\alpha))$ when many consecutive inputs (values of G^l_{ij}) are approximately equal in magnitude and sign. In the filter analogy, these are low-frequency inputs and hence attain highest gain. On the other hand, the maximum attenuation from the system is $(\eta/(1+\alpha))$ when many consecutive inputs are roughly equal in magnitude, but the signs of any two consecutive inputs are opposite. Again in the filter analogy, these represent high-frequency inputs. The lower and upper bounds between which the dynamic range of the *effective* learning rate coefficient is adjusted are the minimum and maximum gains of the system. For example, if $\eta = 1$ and $\alpha = 0.2$ then the lower and upper bounds of the effective learning rate coefficient are 0.833 and 1.25, respectively. Clearly, a momentum coefficient value close to zero keeps the range for adjustment of the effective learning rate coefficient smaller, but it lets the system respond quickly in the area where the gradient of the energy surface is changing at a faster rate. However, if the momentum coefficient is closer to one, a large dynamic range for the effective learning rate coefficient is obtained and the value of the energy function decreases at a higher rate near the constant gradient area; but the system tends to be unstable near the area where the gradient changes sharply. Selection of an optimal value for the momentum coefficient is a difficult, if not impossible problem. Some experimental studies have shown that for some problems a momentum coefficient value 0.5 could be too high; yet others have used a momentum coefficient value as high as 0.99 without any difficulty [Tollenaere 1990]. Some experimental results on the effect of the change of learning rate and momentum coefficients were reported in Tollenaere [1990] and Eaton and Olivier [1992].

A variation of the momentum strategy described by Equation (14) is

$$\Delta w^l_{ij}[s+1] = (1-\alpha)\eta' G^l_{ij}[s] + \delta \Delta w^l_{ij}[s]. \tag{18}$$

It is not difficult to see that the strategies given by Equations (14) and (18) will behave in the same way, provided that

$$\eta' = \frac{\eta}{1-\alpha}.$$

For more detailed descriptions of other theoretical aspects on the effect of the momentum coefficient see Jacobs [1988] and Watrous [1988]. The momentum strategy can be considered as an approximation to the conjugate gradient method [Battiti 1992], because in both the present gradient direction is modified using

a term that takes the previous direction into account. However, the main difference between the two methods is that in the momentum strategy the momentum coefficient is fixed and guessed by the user, whereas in the conjugate gradient method it is automatically adjusted as the training proceeds. Please see Section 4.7 for details on conjugate gradient methods for training of FFANNs.

4.2 Adaptive Learning Rate Coefficient

In the previous section we found that the momentum coefficient implicitly adjusts the *effective* learning rate coefficient dynamically, depending on the nature of the energy surface. The improvement in learning time comes from this dynamic behavior of the effective learning coefficient. There are methods where the learning coefficient is explicitly adjusted to obtain an improved convergence speed. Three methods for explicitly adapting the learning coefficient are described next.

4.2.1 The "Bold Driver" Method. This method makes two trivial changes to the original EBP algorithm: it monitors the value of the energy function E given by Equation (4), and it dynamically adjusts the value of the learning rate coefficient η.

The training starts with some arbitrary value of the learning rate coefficient η. If the value of E decreases, the learning rate is increased by a factor ρ ($\rho > 1$). This helps to take a longer step in the next iteration. And also, the value of the learning rate grows exponentially in a constant gradient region of the energy function. On the other hand, if the value of the energy function E increases, it is assumed that the last step size was too large and (1) the last weight correction to every weight is canceled, (2) the value of the learning rate coefficient is decreased by a factor σ ($\sigma < 1$), and (3) a new trial is performed. If the new trial shows a reduction in the value of the energy function, the decreased learning rate coefficient is accepted as the next learning rate; otherwise the learning rate coefficient is repeatedly reduced until it gives a step size that reduces the value of the energy function. The example is continued to illustrate the basic ideas behind the method.

Example (con't.). We kept the initial conditions as before. The learning rate coefficient increasing factor ρ was set to 1.1 and the learning rate coefficient decreasing factor α was set to 0.5. Figure 5 plots the learning rate coefficient and the value of energy function against the number of training steps. For a better illustration, the value of the energy function was multiplied by 30. It is clear that the learning rate coefficient increases gradually (by a factor of 1.1) until we hit a training step where the value of the energy function increases. At this point, the learning rate coefficient is decreased (by a factor of 0.5).

Figure 6 compares energy values of momentum methods and the "bold driver" method. At the beginning both have the same value of the energy function. But soon the value of the energy function is decreased quickly by increasing the learning rate coefficient (see Figure 5). However, they gradually reach a relatively large learning rate coefficient and the energy function value increases instead of decreases. At this point the algorithm starts decreasing the value of the learning rate coefficient until it receives a value that will decrease the value of the energy function. As in the case of the momentum method, the energy function value slowly drops down without any oscillation.

The desired error level was reached after 199 iterations. The final weights of the network were very similar to those obtained by the momentum method, but had minute differences. The "bold driver" method was significantly faster for this example, as we desired. However, it should be kept in mind that the situation may not always be this good.

The "bold driver" method is nonlocal because it keeps only one learning rate coefficient for all the weights, whereas

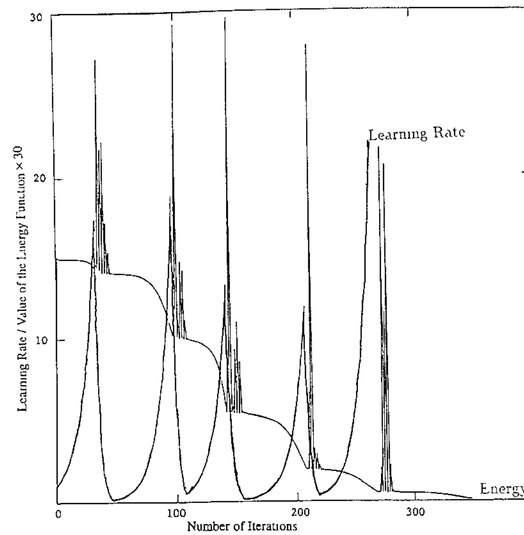

Figure 5. Relationship between learning rate and energy value in "bold driver" version of EBP learning.

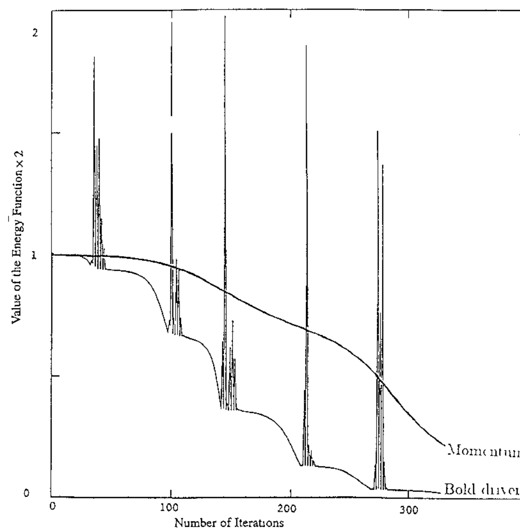

Figure 6. Comparison of energy values for "bold driver" and momentum version of EBP learning.

the momentum strategy adjusts the *effective* learning coefficient locally for each weight through the momentum term. Battiti [1989] has reported that $\rho = 1.1$ and $\sigma = 0.5$ are good choices. Because this method controls the oscillation of the value of the energy function, it could be viewed as an EBP algorithm with controlled oscillation of the energy function.

4.2.2 SAB: Self-Adaptive Back Propagation. This method was developed independently by Jacobs [1988] and Devos and Orban [1988]. It is a local acceleration

strategy, and it is supported by the following observations: (1) every weight should have its own learning rate coefficient, because the partial derivative of the energy function E with respect to each weight gives the gradient for that weight only. Also, the learning rate coefficient for one weight need not be appropriate for other weights. (2) The learning rate coefficient should be allowed to vary depending on the nature of the surface of the energy function along the dimension under consideration. (It is not unusual to have different properties of the gradient along a single dimension.) (3) The learning rate coefficient for a weight should be increased if consecutive steps have same sign. (4) The learning rate coefficient should be small when the derivative of the energy function with respect to a weight changes sign.

Let the learning rate coefficient for the weight w_{ij}^l at time step s be $\eta_{ij}^l[s]$. The algorithm is as follows:

(1) Choose an initial learning rate coefficient η.
(2) Set the learning rate coefficient $\eta_{ij}^l[0] = \eta$; {same for all weights}.
(3) Do a back propagation step *without momentum term.*
(4) If $G_{ij}^l[s]$, the negative of the partial derivative of the energy function E, has the same sign, set $\eta_{ij}^l[s+1] = \rho\eta_{ij}^l[s]$ {for weight $w_{ij}^l[s]$}; $\{\rho > 1\}$.
(5) If $G_{ij}^l[s]$ changes sign then
   - set $\eta_{ij}^l[s+1] = \eta$; {for weight $w_{ij}^l[s]$};
   - estimate a "good" weight $w_{ij}^l[s+1]$ by interpolation (based on previous Δw_{ij}^l values and assuming that the weight space is quadratic in every dimension);
   - do a number of back propagation steps *with momentum term.*
(6) Restart the algorithm from Step (3).

Example (cont.). The SAB method is fundamentally different from all other methods discussed earlier. In this method one learning rate coefficient is kept for each weight. Figure 7 shows plots of

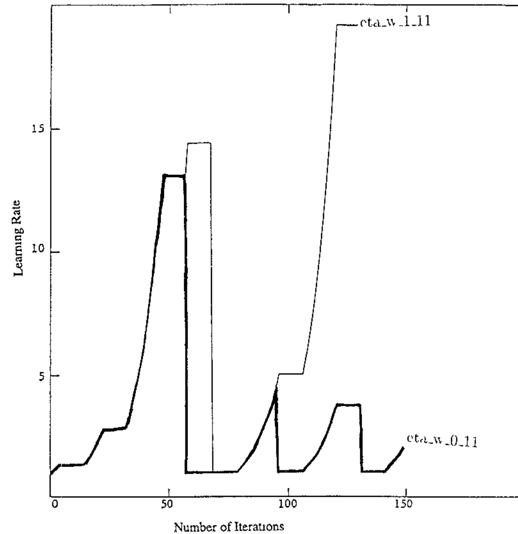

Figure 7. Separate learning rate coefficients for two weights in the SAB version of EBP learning.

learning rate coefficients for two weights of the example network. It shows that they start with the same learning rate coefficient value and keep their values, the same for a while by increasing them exponentially. We used the increment factor $\rho = 1.1$ for this example. But after a few iterations the learning rate coefficient of w_{11}^0 drops to the initial value, whereas that of w_{11}^1 does not. This is because the gradient of the energy function in the direction of w_{11}^0 changes sign but that for w_{11}^1 does not change. The desired error level was reached after 171 iterations. The final weights of the network were different from those obtained by other methods. The final weights and biases of the network are

$$w_{11}^0 = -2.00546, w_{12}^0 = -1.86626,$$

and $bias_1^1 = 2.60926$ for neuron u_1^1 in hidden layer;

$$w_{21}^0 = 12.531, w_{22}^0 = 13.3534,$$

and $bias_2^1 = -5.0778$ for neuron u_2^1 in hidden layer;

$$w_{11}^1 = 5.09768, w_{12}^1 = 4.4951,$$

and $bias_1^o = 6.42277$ for neuron u_1^o in output layer.

This algorithm performs better than the original EBP, because it can adjust the learning rate coefficient over a wide range if the initial learning rate coefficient is "small." One problem inherent in the original EBP algorithm, the selection of a "good" learning rate coefficient by the user, remains unchanged in this algorithm. In addition, this algorithm starts with the initial learning rate coefficient if a change in the sign of the gradient occurs. The next algorithm, different variations of which were proposed in Jacobs [1988], Devos and Orban [1988], and Tollenaere [1990], overcomes both problems. Following Tollenaere [1990], it is here called SuperSAB.

4.2.3 SuperSAB. Let ρ be the factor by which the learning rate coefficient is increased, if necessary, and let σ be the factor by which the learning rate is decreased, if necessary. Experimental studies suggest that the learning rate coefficient decrease should be faster than the increase [Jacobs 1988, Tollenaere 1990]. Recall that the learning rate coefficient at time step s for weight w_{ij}^l is denoted by $\eta_{ij}^l[s]$. The algorithm is as follows:

(1) Choose an initial learning rate coefficient η.
(2) Set the learning rate coefficient $\eta_{ij}^l[0] = \eta$; {for all weights}.
(3) Do a number of back propagation steps *with momentum term.*
(4) If $G_{ij}^l[s]$, the negative of the partial derivative of the energy function E has the same sign, set $\eta_{ij}^l[s+1] = \rho\eta_{ij}^l[s]$ {for weight $w_{ij}^l[s]$}; $\{\rho > 1\}$.
(5) If $G_{ij}^l[s]$ changes sign then
   - set $\eta_{ij}^l[s+1] = \sigma\eta_{ij}^l[s]$; {for weight $w_{ij}^l[s]$} $\{\sigma < 1\}$;
   - undo the previous weight update; and
   - set $\Delta w_{ij}^l[s+1] = 0$.
(6) Restart the algorithm from Step (3).

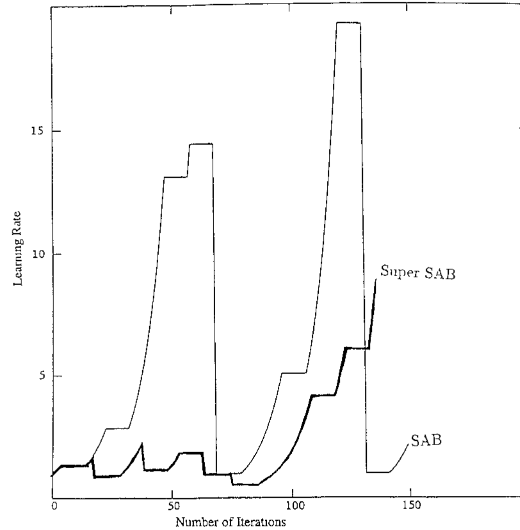

Figure 8. Comparison of learning rate coefficients for a weight in the SAB and SuperSAB versions of EBP learning.

Example (con't.). Figure 8 shows plots of learning rates of weight w_{11}^1 for SAB and SuperSAB learning algorithms. These two plots show the most important difference between SAB and SuperSAB learning methods: when there is a change in sign of the gradient of the energy function in the direction of the weight w_{11}^1, (1) in the SAB learning method the learning rate coefficient is set to the initial value, but (2) in the SuperSAB learning method the value of the learning rate coefficient is reduced by a constant factor, which is 0.5 in our example.

Although it is claimed that on an average the SuperSAB method is faster than the SAB method, in this example we were unlucky: the network did not learn even after 500 iterations. This situation makes a very important point: no matter how much *engineering* we do to speed up the EBP learning algorithm, it is going to fail with certain initial conditions.

An experimental study by Tollenaere [1990] has shown that this algorithm speeds up learning considerably. Also, because the learning rate coefficient is never set to its initial value (as in SAB),

the choice of initial learning rate coefficient would not have much influence on learning time and hence can be made arbitrarily. The algorithm of Jacobs [1988] has been extended by Minai and Williams [1990a, 1990b] and tested on a number of interesting problems to show its superior performance [Minai and Williams 1990b].

4.3 Controlling Oscillation of Weights

In this section, several methods to control the oscillation of weights are described. Before proceeding further some notation is introduced to make the presentation concise and precise. The functions SIGN($\Delta w_{ij}^l[s]$) returns the sign of the value of $\Delta w_{ij}^l[s]$. The modified algorithm saves the last weight change $\Delta w_{ij}^l[s-1]$. It is obvious that if SIGN ($\Delta w_{ij}^l[s-1]$) is different from SIGN ($\Delta w_{ij}^l[s]$) then there is an oscillation of weight. In this case, the amount of weight correction is reduced by a factor, denoted FACTOR. The value of FACTOR is between zero and one. For further reference we name each of the methods used to calculate the value of FACTOR.

4.3.1 Inverse Input-Size Correction Reduction. In this method, if at instance s an oscillation is detected for a weight connected from neuron i to neuron j then the value of FACTOR is determined by:

$$FACTOR = \frac{1}{Size_of_Input},$$

where $Size_of_Input$ is the number of neurons connected to neuron i from the previous layer l. Thus the modified weight correction $\Delta w_{ij}'^l[s]$ for the oscillating weight is

$$\Delta w_{ij}'^l[s] = \frac{1}{Size_of_Input} \times \Delta w_{ij}^l[s].$$

Recall that $\Delta w_{ij}^l[s]$ is the weight change that would have been used by the original EBP algorithm.

4.3.2 Constant Correction Reduction. This method starts with a predetermined constant c and computes FACTOR as the inverse of c, that is,

$$FACTOR = (1/c).$$

Using FACTOR, the modified weight $\Delta w_{ij}'^l[s]$ is calculated from $\Delta w_{ij}^l[s]$. In Pirez and Sarkar [1991] $c = 2$ and $c = 4$ were used as the constant factors for extensive experimental study and good performance was obtained.

4.3.3 Normalized $|\Delta w_{ij}^l[s-1]|$ Correction Reduction. In the two methods proposed earlier, modified weight correction depends only on the suggested correction $\Delta w_{ij}^l[s]$. In the present method to computer FACTOR, both the suggested correction at instance s and the actual correction at the instance $s-1$ were used. The value of FACTOR is proportional to $|\Delta w_{ij}^l[s-1]|$. Mathematically,

$$FACTOR = \left| \frac{\Delta w_{ij}^l[s-1]}{\Delta w_{ij}^l[s] - \Delta w_{ij}^l[s-1]} \right|.$$

4.3.4 Normalized $|\Delta w_{ij}^l[s]|$ Correction Reduction. This method is very similar to the previous method. Now we make FACTOR proportional to $|\Delta w_{ij}^l[s]|$ instead of $|\Delta w_{ij}^l[s-1]|$.

$$FACTOR = \left| \frac{\Delta w_{ij}^l[s]}{\Delta w_{ij}^l[s] - \Delta w_{ij}^l[s-1]} \right|.$$

It is not difficult to see that, like the original EBP algorithm, the modified EBP algorithms also reduce the sum of the squared error as the training proceeds. Extensive experimental study on complex problems has shown that these modified algorithms on the average perform better than the EBP algorithm with momentum term. For narrower networks they outperform EBP with the momentum factor [Pirez and Sarkar 1991].

4.4 Rescaling of Variables

The modifications described earlier do not consider the effect of the number of lay-

ers in a network, although it is believed that as the number of layers increases, so does the training time in the original EBP algorithm. A reason for this behavior is explained by Rigler et al. [1991], based on two simple but correlated facts. First, the sigmoidal function [see Equation (2)], when used as the activation function in the original EBP algorithm (the activation function), agrees with the logistic differential equation

$$y' = y(1 - y). \qquad (19)$$

Because the value of the sigmoidal function $f(x)$ is bounded by zero and one, the value of $y' = y(1 - y)$ is bounded by zero and 0.25. The derivative y' attains maximum value 0.25 when $y = 0.5$. Second, the term $y(1 - y)$ comes from the chain rule for computation of partial derivatives of the energy function given by Equation (4) or (5). The number of times this term occurs is exactly the number of layers the weights are away from the output layer. In other words, the term $y(1 - y)$ occurs exactly once in the partial derivatives of the energy function with respect to the weights of the output layer, exactly twice in the partial derivative of the energy function with respect to the weights in the hidden layer just behind the output layer, and so on.

Because of these two facts, the value of the partial derivative at different layers cannot exceed $1/4, 1/16, 1/64, \ldots,$ and hence the magnitude of the gradient diminishes at an exponential rate as we move away from the output layer. One splution to this problem is to *rescale* the value of the gradient in every layer. Rigler et al. [1991] suggest using powers of expected value of $y(1 - y)$, $E[y(1 - y)]^l$, assuming that the value of y is uniformly distributed between zero and one. These values are $1/6, 1/36, 1/216, \ldots,$ and the corresponding rescaling values are $6, 36, 216, \ldots,$ for layers one, two, three, and so on when the layers are counted backward starting with the output layer as layer one. The improvement in training performance using the variable rescaling procedure was reported in Rigler et al. [1991]. It was found that on the average it reduces the number of required iterations considerably.

4.5 EBP With Expected Source Values

The weight update in the EBP algorithm is proportional to the output of the neuron that is its *source*. However, once weights are updated following the EBP algorithm, the output of the source neuron will be different. Thus Samad [1991] recommended using the "expected output" of the source neuron instead of the actual output. One can see the reason for the recommendation as follows. From Equations (6) and (8), as discussed earlier, the weight change at step $s + 1$, $\Delta w_{ij}^l[s]$ can be written as

$$\Delta w_{ij}^l[s] = -\eta \frac{\partial E_p}{\partial \omega_{ij}^l[s]}. \qquad (20)$$

Using the chain rule of differentiation, $\partial E_p / \partial w_{ij}^l[s]$ can be rewritten as

$$\frac{\partial E_p}{\partial w_{ij}^l[s]} = \frac{\partial E_p}{\partial net_{pi}^{l+1}} \frac{\partial net_{pi}^{l+1}}{\partial w_{ij}^l[s]}. \qquad (21)$$

Next, referring to the expression for net_{pi}^{l+1} from Equation (1), it is easy to find that

$$\frac{\partial net_{pi}^{l+1}}{\partial w_{ij}^l[s]} = out_j^l. \qquad (22)$$

Note that to neuron u_i^{l+1} the output of neuron u_j^l, out_j^l is a *source*. Following Rumelhart et al. [1986] $\partial E_p / \partial net_{pi}^{l+1}$ can be denoted by δ_{pi}^{l+1} and expressed as

$$\delta_{pi}^{l+1} = \begin{cases} out_i'^{l+1}\left(d_{pi} - out_i^{l+1}\right) \\ \quad \text{for neuron } i \text{ in output layer} \\ out_i'^{l+1} \sum_{k=1}^{n_{l+2}} w_{kj}^{l+1} \delta_k^{l+2} \\ \quad \text{for neuron } i \text{ in hidden layers,} \end{cases} \qquad (23)$$

where $out_j'^{l+1}$ is the derivative of the activation function for neuron u_j^{l+1} with respect to the net input net_j^{l+1} (evaluated for the input pattern x_p). Using Equations (21), (22), and (23), the rule for

weight update [Equation (20)] can be rewritten as

$$\Delta w_{ij}^{l}[s] = -\eta\, out_j^l \delta_{pi}^{l+1}. \quad (24)$$

Now it can be argued that the weight correction $\Delta w_{ij}^l[s]$ is proportional to the output of the source neuron u_j^l and proportional to δ_{pi}^{l+1} of neuron u_i^{l+1}. However, the output of the source neuron u_j^l would be different after changing its input weights following the EBP algorithm. Thus instead of using the actual output of the source unit, an "estimated value" of the output of the source can be computed as the sum of the actual output and the error term at the source. This gives the new rule for weight update as

$$\Delta w_{ij}^{l}[s] = -\eta\left(out_j^l + \gamma\delta_i^l\right)\delta_{pi}^{l+1}, \quad (25)$$

where γ is a constant. Clearly, when $\gamma = 0.0$, the original EBP algorithm is obtained. The suggested modification is simple to implement and also can be used for both online and batch mode of learning. Experimental results presented by Samad [1991] show a faster learning rate with this modified weight updating rule.

4.6 Self-Determination of Adaptive Learning Rates

A problem with the EBP algorithm is that a learning rate coefficient is necessary for updating weights. The two approaches discussed thus far either use an educated guess to fix the learning rate coefficient to a constant value or adjust it dynamically using some heuristic methods. But they cannot eliminate the risk of taking "too large" steps and overshooting the goal. The method discussed in this section computes the local, most extreme, steepest gradient attainable for computing optimal step size.

Once the error (E_p) or current height and the extreme energy-weight gradient (extreme $(\partial E_p / \partial w_{ij}^l[s])$) is known, the $\Delta_{\text{opt}} w_{ij}^l[s]$ can be computed as

$$\Delta_{\text{opt}} w_{ij}^l[s] = -\frac{E_p}{\text{extreme}\left(\dfrac{\partial E_p}{\partial w_{ij}^l[s]}\right)}$$

and thus using Equation (20), the optimal learning rate coefficient can be expressed by

$$\eta\text{opt} = \frac{E_p}{\text{extreme}\left(\dfrac{\partial E_p}{\partial w_{ij}^l[s]}\right)\dfrac{\partial E_p}{\partial w_{ij}^l[s]}}. \quad (26)$$

The methods for computing the energy-weight gradient and the energy are known. Next, from Weir [1991] a method is described to compute the extreme energy-weight gradient.

4.6.1 The Extreme Gradient.

The method for computing the extreme energy-weight gradient described in Weir [1991] considers the neurons in the output layer and in the hidden layers separately. First, computation of the extreme energy-weight gradient for output layer units is described.

Using the notations introduced earlier and the chain rule of differentiation, the gradient of the energy function with respect to a weight w_{ij}^l for the input training pattern x_p can be written as

$$\frac{\partial E_p}{\partial w_{ij}^{l-1}[s]} = \frac{\partial E_p}{\partial out_i^l}\,\frac{\partial out_i^l}{\partial net_i^l}\,\frac{\partial net_i^l}{\partial w_{ij}^{l-1}[s]}. \quad (27)$$

Output Layer. For a neuron u_i^o in the output layer

$$\frac{\partial E_p}{\partial out_i^o} = out_i^o - d_{pi}, \quad (28)$$

$$\frac{\partial out_i^l}{\partial net_i^l} = out_i^o(1 - out_i^o), \quad (29)$$

$$\frac{\partial net_i^l}{\partial w_{ij}^l[s]} = out_j^{o-1}. \quad (30)$$

Setting the value of out_j^{o-1} to its possible maximum of one, the bounds of the energy-weight gradient can be computed by solving the cubic in out_i^o [obtained from Equations (28), (29), and (30)] over the interval (0, 1). Let the maximum and minimum values found be m_1 and m_2, respectively. Let us define δ_{pk}^o as

$$\delta_{pk}^o = \frac{\partial E_p}{\partial out_k^o} \frac{\partial out_k^o}{\partial net_{pk}^o}. \quad (31)$$

Now to keep the computation procedure simple, a single value of δ_{pk}^o is propagated backward via the neurons u_j^{o-1}, and the extreme value of the energy-weight gradient is obtained as

$$\text{extreme}\left(\frac{\partial E_p}{\partial w_{ij}^o[s]}\right) = \text{extreme}(\delta_{pk}^o) = m_q,$$

where $m_q = m_1$ or m_2 depending on the sign of the energy-weight gradient. The extreme energy-weight gradient for all the training patterns then can be computed as $\Sigma_p m_q$, where q is either $1 \forall p$ or $2 \forall p$.

Hidden layers. For a neuron u_j^l in a hidden layer, the value of the energy-weight gradient of the next outer layer $l + 1$ is used to compute its extreme energy-weight gradients:

$$\frac{\partial E_p}{\partial out_j^l} = \sum_{i=1}^{n_{l+1}} \frac{\partial E_p}{\partial out_i^{l+1}} \frac{\partial out_i^{l+1}}{\partial net_i^{l+1}} \frac{\partial net_i^{l+1}}{\partial out_j^l}. \quad (32)$$

Using Equation (31) and noting that

$$\frac{\partial net_i^{l+1}}{\partial out_j^l} = w_{ij}^l,$$

Equation (32) can be written as

$$\frac{\partial E_p}{\partial out_j^l} = \sum_{i=1}^{n_{l+1}} \left(\delta_{pi}^{l+1} w_{ij}^l\right).$$

Now from the last equation, δ_{pj}^l for a neuron u_j^l in a hidden layer l can be defined as

$$\delta_{pj}^l = \frac{\partial E_p}{\partial out_j^l} \frac{\partial out_j^l}{\partial net_{pj}^l} = \sum_{i=1}^{n_{l+1}} \left(\delta_{pi}^{l+1} w_{ij}^l\right) \frac{\partial out_j^l}{\partial net_{pj}^l}. \quad (33)$$

Because the value of the last term in Equation (33) cannot exceed 0.25 [see Equations (2) and (29) for explanation] and values of w_{ij}^l and δ_{pj}^l are independent of each other,

$$\text{extreme}\left(\delta_{pj}^l\right) = \frac{1}{4} \sum_{i=1}^{n_{l+1}} \text{extreme}\left(\delta_{pi}^{l+1} w_{ij}^l\right) = \frac{1}{4} \sum_{i=1}^{l+1} \left(\text{extreme}\left(\delta_{pi}^{l+1}\right) \text{extreme}\left(w_{ij}^l\right)\right) \quad (34)$$

with $\text{sgn}(\text{extreme}(\delta_{pi}^{l+1})\, \text{extreme}(w_{ij}^l)) = \text{sgn}(\text{extreme}(\delta_{pj}^l)) = \text{sgn}(\delta_{pj}^l)$ for extreme $(\delta_{pj}) \neq 0$, where

$$\text{sgn}(x) = \begin{cases} +1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0. \end{cases}$$

As for the output layer, after setting out_k^{l-1} to one we get

$$\text{extreme}\left(\frac{\partial E_p}{\partial w_{jk}^l}\right) = \text{extreme}\left(\delta_{jp}^l\right).$$

4.6.2 The Extreme Weights. Computation of the extreme energy-weight gradient for nodes in hidden layers requires values of extreme weights. Now a method is described for computing the extreme gradient. The extreme weight for the links between neuron u_j^l and neuron u_i^{l+1} is either $w_{ij}^l[s]$ or $w_{ij}^l[s+1]$. At time instant s, the value of the former is known: however, computation of the value of $w_{ij}^l[s+1]$ using Equations (6) and (8) requires the value of η_{opt} that is being computed. This circularity problem can be avoided by expressing the extreme gradient in terms of η and computing η

directly. The computation can proceed after viewing the extreme gradient as

$$\text{extreme}\left(\frac{\partial E_p}{\partial w_{ij}^l[s]}\right) = a\eta_{\text{opt}} + b, \quad (35)$$

with

$$a = \frac{1}{4}\sum_{i=1}^{n_{l+1}} \text{extreme}\left(\delta_{pi}^{l+1} h_i\right),$$

where $h_i = 0$ or $-\partial E_p / \partial w_{ij}^l[s]$ depending on whether extreme $(w_{ij}^l[s]) = w_{ij}^l[s]$ or $w_{ij}^l[l+1]$, respectively, and

$$b = \frac{1}{4}\sum_{i=1}^{n_{l+1}} \text{extreme}\left(\delta_{pi}^{l+1} w_{ij}^l[s]\right).$$

Substitution of the extreme gradient value from Equation (35) in Equation (26) results in a quadratic equation of the form

$$a\eta_{\text{opt}}^2 + b\eta_{\text{opt}} + c = 0,$$

where

$$c = \frac{-E_p}{\partial E_p / \partial w_{ij}^l[s]}.$$

The quadratic equation in η_{opt} has an unique positive real solution [Weir 1991]. Thus with the technique described in this section one can easily compute a safe step size at every iteration of the EBP algorithm. The modification increases the computation requirements. Also, because it always takes conservative step sizes, it is more likely to follow the energy surface very closely, and thus may get stuck into local minima before learning all the examples. Empirical study has shown some improved performance of this algorithm over the ones that use a momentum term [Weir 1991]. The most attractive feature of this modified algorithm is that no learning rate coefficient has to be "guessed" for use.

4.7 Conjugate Gradient Methods

In the EBP algorithm, the energy function E is minimized following steepest descent. At every step, a direction is computed that would minimize error at that instant, and this process of energy minimization is continued iteratively. One problem with this method is that a direction g_i at time step s_i and another direction g_j at time step s_j may not be mutually perpendicular, and thus the energy function minimized at time step s_i may be spoiled, at least in part, by the minimization at time step s_j. If two directions are linearly independent and *noninterfering* then the minimization will be faster. The concept of *noninterfering* direction is at the heart of the conjugate method for minimization. Two directions p_i and p_j are mutually conjugate with respect to a matrix M if

$$p_i^T M p_j = 0 \quad \text{if } i \neq j,$$

where p_i^T is the transpose of the vector p_i. The basic idea is this: if the new direction for minimization, p_{i+1} is perpendicular to the earlier direction p_i, the minimization with former direction will not be affected by the minimization in the present direction.

Before proceeding further, some notations are introduced that are used only in this section. Let the weights of the network at time step s be considered to be a vector $W[s]$. Let $G[s]$ be the energy-weight gradient vector corresponding to the elements of $W[s]$ at time step s. Let $Y[s] = G[s+1] - G[s]$; the first direction $p[1]$ is given by $p[1] = -G[1]$. Starting with initial weight matrix $W[1]$ the subsequent weight matrices are computed as follows:

$$W[s+1] = W[s] + \mu[s]p[s] \quad (36)$$

$$p[s+1] = -G[s+1] + \theta[s]p[s], \quad (37)$$

where $\mu[s]$ is chosen to minimize the energy function along the search direction $p[s]$,

$$\mu[s] = \min_{\mu} E(W[s] + \mu p[s]), \quad (38)$$

and $\theta[s]$ is defined by

$$\theta[s] = \frac{Y[s].G[s+1]}{Y[s].p[s]},$$

where $A.B$ of two vectors of identical sizes is the scalar product, defined as the sum of the product of the corresponding elements. Formally,

$$A.B = \sum_i a_i \times b_j.$$

There are different versions of the conjugate gradient method depending on the choice of computation of $\theta[s]$. For other choices of $\theta[s]$ see Shanno [1978] and references therein.

Two issues are of concern regarding the use of conjugate gradient methods for EBP. First, the computation of $\theta[s]$ using Equation (38) is expensive because every step of the linear search for an optimal value involves an evaluation of the energy function and this requires computation of error. Second, this method would guarantee a convergence after $N + 1$ iterations if the objective function were quadratic in weights. Moreover, the possibility of numerical instability exists. Nonetheless, it has been shown in Battiti [1989] and Kinsella [1992] that the conjugate gradient method performs better than the standard EBP algorithm. Others have conjectured and shown empirically that in some cases one can use a conjugate gradient method with some inexact linear search instead of the exact linear search (in Equation (38)) [Battiti 1989, 1992; Shanno 1978].

Battiti [1989] used an inexact linear search where the search direction was computed using the following equation:

$$p[s+1] = -G[s+1] + A[s]\Delta W[s] + B[s]Y[s], \tag{39}$$

where

$$\Delta W[s] = W[s] - W[s-1].$$

After every N steps, where n is the number of weights in the weight matrix $W[s]$, the search is restarted in the direction of the negative energy-weight gradient $-G[N]$ and the coefficients $A[s]$ and $B[s]$ are computed as follows:

$$A[s] = -\left(1 + \left(\frac{Y[s].Y[s]}{\Delta W[s].Y[s]}\right)\right) \times \left(\frac{\Delta W[s].G[s]}{\Delta W[s].Y[s]}\right) + \left(\frac{Y[s].G[s]}{\Delta W[s].Y[s]}\right)$$

$$B[s] = \left(\frac{\Delta W[s].G[s]}{\Delta W[s].Y[s]}\right).$$

The inexact linear search is similar to the momentum strategy described in Section 4.1, with the added feature that the former directly adjusts the coefficients as the learning proceeds. There have been reports on several simultaneous and independent efforts on the use of the modified conjugate gradient method for the EBP learning algorithm, for more information, see Battiti [1992].

4.8 Different Energy Functions

The methods described in other sections keep the sum of the squared error as the energy function [see Equations (4) and (5)] while they change the original EBP algorithm by adjusting the learning coefficient, adding a momentum term, using elegant numerical computation methods, and so on. In this section a set of new energy functions is described that is defined and used to speed up learning.

The expression for computation of the error signal when sigmoidal activation [Equation (2)] and the sum of the squared errors are used contains a factor of the form $out_i^o(1 - out_i^o)$, where the value of out_i^l is between zero and one. The value of this factor approaches zero as out_i^o approaches one of its extreme values, zero or one. Thus if the output of an output neuron is close to either extreme and far away from the target output, then the high value of its error signal is attenuated very close to zero and almost no correction to its weights is made. With this observation van Ooyen and Nienhuis [1992] proposed a new energy function

that eliminates this problem. When this new energy function

$$E_{\ln} = \sum_{p=1}^{P} \sum_{i=1}^{n_o} \Big(d_{pi} \ln out_{pi}^{o} + (1 - d_{pi}) \ln(1 + out_{pi}^{o}) \Big) \quad (40)$$

(where $\ln(x)$ is the natural logarithm of x) is used, the energy-weight gradient is given by

$$\frac{\partial E_{\ln}}{\partial w_{ij}^{o-1}} = (out_i^o - d_{pi}) out_j^{o-1},$$

which does not contain the factor that attenuates the error signal when the output of a neuron in the output layer is close to either extreme. Empirical study on several problems in van Ooyen and Nienhuis [1992] has shown improved performance. This log-likelihood function was also proposed by Solla et al. [1988] to use as an error measure in layered neural networks, and through numerical simulations they found marked reduction in learning times. This improvement was viewed as characteristic of the function. Holt and Semnani [1990] have also recommended this energy function and observed through simulations that it can speed up back propagation learning.

Ahmad and Salam [1992a, 1992c, 1992b] have used three new energy functions to improve the performance of the original EBP algorithm. Because the standard EBP algorithm descends along the gradient of the error surface, the use of any energy function that has a larger gradient than that of the sum of the squared error at higher energy values would make for faster training. This view may have motivated Ahmad and Salam to propose and use a Cauchy energy function and the polynomial energy function. In Ahmad and Salam [1992b] the Cauchy energy function E_c, defined by Equation (41), has shown faster convergence:

$$E_c = \frac{1}{2} \prod_{p=1}^{P} \prod_{i=1}^{n_o} \Big(1 + (\varepsilon_{pi})^2\Big). \quad (41)$$

The polynomial energy function given by

$$E_{pol} = a \sum_{p=1}^{P} \sum_{i=1}^{n_o} \big(|\varepsilon_{pi}| + b \big)^r, \quad (42)$$

where a and b are constants (or any other energy functions that help speed up the learning process), has been studied in Ahmad and Salam [1992c]. It was reported that faster convergence is obtained with the polynomial energy function [Ahmad and Salam 1992c]. The third energy function used by Ahmad and Salam, called the exponential energy function, appears in practical situations to help avoid local minima at higher energy levels. It is defined in terms of the sum of the squared error energy function E as

$$E_{exp} = e^{\kappa E}, \quad (43)$$

where κ is a constant. This apparently simple energy function has an excellent feature: as the error decreases so does the learning rate exponentially. Because of this effect Ahmad and Salam [1992a] appropriately call the modified learning "dynamic learning using exponential energy function." The dynamic nature presented by this new energy function can be understood by looking at the energy-weight gradient

$$\frac{\partial E_{\exp}}{\partial w_{ij}^{l}} = \kappa e^{\kappa E} \frac{\partial E}{\partial w_{ij}^{l}}. \quad (44)$$

As can be seen, the energy-weight gradient of the exponential energy function is a product of the energy-weight gradient of the original (sum of the square) energy function and an exponential function of the original energy function. Clearly, as the value of the error decreases, so will the step size, even if the learning rate coefficient is kept constant. Ahmad and Salam have shown that the exponential energy function, under easily met conditions, performs no worse than the standard EBP algorithm, and they have found through empirical study that it actually performs better than the EBP algorithm with momentum term.

4.9 Effect of Training Set Size

The original EBP and the modifications discussed here do not consider the effect of training set size. Empirical study has shown a dependence on training set size [Tesauro 1987; Tesauro and Janssens 1988; Eaton and Olivier 1992]. A study in Tesauro [1987] and [Tesauro and Janssens 1988] concentrated on a difficult problem, a parity function that requires memorization of the input patterns. If the network is not too small for memorization, then the training time grows exponentially with the number of bits for which parity is to be computed. Because studies in Tesauro [1987] and Tesauro and Janssens [1988] were limited to a difficult problem, their results may not apply to such problems as classification of patterns and regression.

Eaton and Olivier [1992] suggested that the length of the energy-weight gradient G_{ij}^{l} might be proportional to L,

$$L = \sqrt{N_1^2 + N_2^2 + \cdots + N_m^2}, \quad (45)$$

where N_i is the number of patterns of type i and m is the number of different pattern types. The intuition behind their suggestion about the length of the gradient is that "the dimensionality of the weight space is usually greater than the number of unique pattern types, so that it is possible for the gradient of each pattern type to be orthogonal to all other types" [Eaton and Olivier 1992]. To compensate for this effect they proposed a learning rate coefficient inversely proportional to L. Their experiment with several problems suggests that if η is selected using the following equation and momentum coefficient $\alpha = 0.9$ then fast training is obtained:

$$\eta = \frac{1.5}{\sqrt{N_1^2 + N_2^2 + \cdots + N_m^2}} \quad (46)$$

Anand et al. [1993] have proposed a method to accelerate learning of classification when training set sizes for each class are different. They have shown that in two class problems, if the size of one training set is much smaller than that of the other, the first iteration of the original EBP algorithm increases the error for the class with fewer examples while decreasing error for the other. This on average increases the number of iterations required to learn the classification. They have proposed a method that reduces the error for both classes simultaneously. They achieve it by selecting a direction for weight correction that favors learning of both classes. A very simple way to find such a direction is to select the direction that bisects the directions of the negative of the gradients of each class [Anand et al. 1993].

In the next section modifications to the back propagation algorithm presented in this section are compared and contrasted.

5. DISCUSSION

The modifications to the original EBP algorithm described in the previous section can be classified in many different ways. One classification could be based on whether a modification applies to the online EBP algorithm, to batch EBP algorithms, or to both. The momentum strategy, rescaling of variables, EBP with expected source values, and different energy functions can be used for both the online and batch EBP algorithms. The self-determination of adaptive learning rates can be applied only to the online EBP algorithm. All other methods described here are useful only for batch EBP algorithms.

Another classification could be based on whether there is only one (effective) learning rate coefficient for all the weights or one learning rate coefficient for each weight. The "bold driver" method, rescaling of variables, self-determination of adaptive learning rate, conjugate gradient method, and effect of training can use only one learning rate coefficient for all weights. All other methods can use one learning rate coefficient for each weight.

Some of these methods can be combined to get faster training than can be

obtained from any single one. For example, both SAB and SuperSAB use the momentum strategy. It is believed that momentum strategy, controlling oscillation of weights, rescaling of variables, EBP with expected source values, and effect of training set size can be combined together for faster convergence.

Self-determination of adaptive learning rate has the problem that, in making conservative steps to avoid overshooting of the goal, it may get the network stuck in a local minimum before learning all the examples of the training set. Every modification, except self-determination of adaptive learning rate, that applies to the online EBP algorithm requires a "good" learning rate coefficient to be supplied by the user. As discussed in Section 2, the effective learning rate coefficient for two problems may be quite different. Thus it is necessary to find a modification to the EBP algorithm for online training that does not require any effective learning rate coefficient.

ACKNOWLEDGMENTS

I thank A. van Ooyen for reading an earlier version of this report and informing me about two of the references. I also thank the anonymous reviewers for their constructive comments, and Reviewer 1 for informing me about two references.

REFERENCES

ANAND, R., MEHROTTRA, K. G., MOHAN, C. K., AND RANKA, S. 1993. An improved algorithm for neural network classification of imbalanced training sets. *IEEE Trans. Neural Netw. 4.*

AHMAD, M. AND SALAM, F. M. A. 1992a. Dynamic learning using exponential energy function. In *Proceedings of the International Joint Conference on Neural Networks.*

AHMAD, M. AND SALAM, F. M. A. 1992b. Error back-propagation learning using the polynomial energy function. In *Proceedings of the fourth International Conference on System Engineering.*

AHMAD, M. AND SALAM, F. M. A. 1992c. Supervised learning using the Cauchy energy function. In *Proceedings of the International Conference on Fuzzy Logic and Neural Networks.*

BATTITI, R. 1992. First- and second-order methods for learning: Between steepest descent and Newton's method. *Neural Comput. 4.*

BATTITI, R. 1989. Accelerated backpropagation learning: Two optimization methods. *Complex Syst. 3.*

BRYSON, A. E. AND HO, Y. C. 1969. *Applied Optimal Control.* Waltham MA.

DEVOS, M. R. AND ORBAN, G. A. 1988. Self learning backpropagation. In *Proceedings of the NeuroNimes.*

EATON, H. A. C. AND OLIVIER, T. L. 1992. Learning coefficient dependence on training set size. *Neural Netw. 5.*

FAHLMAN, S. E. 1988. An empirical study of learning speed in back-propagation networks. Tech. Rep. CMU-CS-88-162, Carnegie Mellon University, Pittsburgh, PA.

HINTON, G. E. 1989. Connectionist learning procedures. *Artif. Intell. 40.*

HINTON, G. E. 1987. Learning translation invariant recognition in massively parallel networks. In *Lecture Notes in Computer Science.* Springer-Verlag.

HOLT, M. J. J. AND SEMNANI, S. 1990. Convergence of back-propagation in neural networks using a log-likelihood cost function. *Electron. Lett. 26.*

JACOBS, R. A. 1988. Increased rate of convergence through learning rate adaptation. *Neural Netw. 1.*

KINSELLA, J. A. 1992. Comparison and evaluation of variants of the conjugate gradient method for learning in feed-forward neural networks with backward error propagation. *Network 3.*

LE CUN, Y. 1988. A theoretical framework for backpropagation. In *Proceedings of the 1988 Connectionist Models Summer School.*

LE CUN, Y. 1986. Learning process in an asymmetric threshold network. In *Disordered Systems and Biological Organization*, Springer-Verlag.

LIPPMANN, R. P. 1987. An introduction to computing with neural network. *IEEE ASSP Mag.* (April), 4–22.

MINAI, A. A. AND WILLIAMS, R. D. 1990a. Acceleration of back-propagation through learning rate and momentum adaptation. In *Proceedings of the International Joint Conference on Neural Networks* (Washington, DC).

MINAI, A. A. AND WILLIAMS, R. D. 1990b. Backpropagation heuristics: A study of the extended delta-bar-delta algorithm. In *Proceedings of the International Joint Conference on Neural Networks* (San Diego, CA).

PARKER, D. B. 1985. Learning logic. Tech. Rep. MIT, Cambridge, MA.

PIREZ, Y. M. AND SARKAR, D. 1991. Back-propagation with controlled oscillation of weights. Tech. Rep. TR-CS-01-91, University of Miami, FL.

RIGLER, A. K., IRVINE, J. M., AND VOGL, T. P. 1991. Rescaling of variables in back propagation learning. *Neural Netw. 4.*

RUMELHART, D. H., HINTON, G. E., AND WILLIAMS, R. J. 1986. Learning internal representation by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1. MIT Press, Cambridge, MA.

RUMELHART, D. E., MCCLELLAND, J. L., ET AL. 1986. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1. MIT Press, Cambridge, MA.

SAMAD, T.1991. Back propagation with expected source values. *Neural Netw. 4.*

SHANNO, D. F. 1978. Conjugate gradient methods with inexact searches. *Math. Oper. Res. 3.*

SIETSMA, J. AND DOW, R. J. 1991. Creating artificial neural networks that generalize. *Neural Netw. 4.*

SOLLA, S. A., LEVIN, E., AND FLEISHER, M. 1988. Accelerated learning in layered neural networks. *Complex Syst. 2.*

TESAURO, G. 1987. Scaling relationships in back-propagation learning: Dependence on training set size. *Complex Syst. 1.*

TESAURO, G. AND JANSSENS, B. 1988. Scaling relationships in back-propagation learning. *Complex Syst. 2.*

TOLLENAERE, T. 1990. SuperSAB: Fast adaptive back propagation with good scaling properties. *Neural Netw. 3.*

VAN OOYEN, A. AND NIENHUIS, B. 1992. Improving the convergence of the back-propagation algorithm. *Neural Netw. 5.*

WATROUS, R. 1988. Learning algorithms for connectionist networks: Applied gradient methods for nonlinear optimization. Tech. Rep. MS-SIS-88-62, Univ. of Pennsylvania.

WEIR, M. K. 1991. A method for self-determination of adaptive learning rates in back propagation. *Neural Netw. 4.*

WERBOS, P. J. 1974. *Beyond regression: New tool for prediction and analysis in the behavioral sciences.* Ph.D. thesis, Harvard Univ.

WERBOS, P. J. 1990. Backpropagation through time: What it does and how to do it. In *Proceedings of the IEEE 78.*

WIDROW, B. AND LEHR, M. A. 1990. 30 years of adaptive neural networks: Perceptron, madline, and backpropagation. In *Proceedings of the IEEE 78.*

Received June 1993, revised September 1994; accepted October 1995.