TREAT: A Trust-Region-based Error-Aggregated Training Algorithm for Neural Networks

Yixin Chen and Bogdan M. Wilamowski

Abstract - A Trust-Region-based Error-Aggregated Training algorithm (TREAT) for multi-layer feedforward neural networks is proposed. In the same spirit as that of the Levenberg-Marquardt (LM) method, the TREAT algorithm uses a different Hessian matrix approximation, which is based on the Jacobian matrix derived from aggregated errors. An aggregation scheme is discussed. It can greatly reduce the size of the matrix to be inverted in each training iteration and thereby lower the iterative computational cost. Compared with the LM method, the TREAT algorithm is computationally less intensive, and requires less memory. This is especially important for large sized neural networks where the LM algorithm becomes impractical.

I. Introduction

As one of the earliest and most general methods for supervised training of multi-layer feedforward neural networks, the error backpropagation algorithm (EBP) [26] provides an efficient way to evaluate the *gradient*, and updates weights by a steepest descent step. Although EBP algorithm works fine for very simple models, a training process can take excessively long time (i.e., the *learning time* is large) when the error surface has many flat regions [11]. This is due to the facts that the *convergence rate* of the EBP algorithm is at best linear [27], and the convergence is sensitive to the choice of *learning rate* that indicates the relative size of the change in weights during each iteration. The learning rate needs to be small enough to ensure reliable convergence behaviors. But a small learning rate could significantly slow down the training process.

An enormous amount of efforts have been made to accelerate the EBP algorithm. For example, many heuristics have been proposed to adapt the learning rate automatically [12], [24] so that the descending step is large when the search point is far away from a minimum, with decreasing step size as the search approaches a minimum. Adding momentum to the weight changes can accelerate the convergence since some curvature information, which is extracted by averaging the gradients locally, is utilized to enlarge the learning rate in flat regions while maintaining a small learning rate in regions with high fluctuations [8], [19]. The learning process can also be sped up by preprocessing the training data [17], [15], using estimated optimal initial weights [20], [30], artificially enlarging errors for neurons operating in saturation regions [14], [23], [28], or optimizing modified error functions [22], [21], [1].

Yixin Chen is with the Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16801. E-mail:yixchen@cse.psu.edu

Bogdan M. Wilamowski is with the College of Engineering, University of Idaho, Boise, ID 83712. E-mail:wilam@ieee.org

Although the aforementioned approaches reduce the learning time, more efficient algorithms are needed to solve complex problems such as prediction of time series, modeling and control of robotic manipulators, and pattern classification, etc. Several layerwise training algorithms have been designed to further improve both the learning time and the convergence of the training process. Layerwise training algorithms usually treat weights of each layer separately by decomposing the original optimization problem into several low dimensional sub-problems. Each sub-problem is then solved using either linear optimization techniques [7], [29] or a hybrid of linear and nonlinear optimization methods [18], [25].

More significant improvement is made possible by employing second order (curvature) information to speed things up. For example, the second order derivatives are exploited in the conjugate gradient method [5] during line search. In [4], [3], the Newton's methods are utilized to update weights. Since the exact calculation of the full *Hessian matrix* is often prohibitively expensive for large networks, different Hessian matrix approximations are proposed. Secant method, such as the Broyden-Fletcher-Golfarb-Shanno (BFGS) algorithm [6], is also applied to network training [2]. It requires no evaluation of the Hessian matrix. Instead, an economical update for the Hessian matrix is made in each iteration.

All the above techniques fall into the category of general-purpose unconstrained optimization. However, the *error function* (objective function) of a neural network has a special structure, sum of the squares of the errors, which can be exploited to increase the efficiency of the training algorithm even further. In other words, training neural networks is essentially a nonlinear least squares (NLS) problem, and thus can be solved by a class of NLS algorithms [6]. Among them the Levenberg-Marquardt (LM) method (a *trust region method* with hyperspherical trust region) [6] works extremely well in practice, and is considered the most efficient algorithm for training medium sized neural networks [10]. Nevertheless, the LM algorithm demands large memory space to store the *Jacobian matrix* and the approximated Hessian matrix, and invert a matrix of size $N \times N$ in each iteration where N is the number of weights. As a result the convergence rate is counterbalanced by high computational cost per iteration when N is large.

This paper describes a new training algorithm, TREAT (Trust Region based Error Aggregated Training), which bears a concept similar to that of the LM method. Through using a different Hessian matrix approximation based on aggregated errors, the TREAT algorithm can cut down the memory requirement and decrease the iterative and overall computational load. The remainder of the paper is organized as follows. Section II briefly reviews the LM method. Section III derives the TREAT algorithm and describes its implementation. Extensive experiments and results are given in Section IV. We conclude in Section V.

II. Levenberg-Marquardt Method

Consider a general multilayer feedforward neural network with L inputs, K outputs, and P training patterns. Let $\vec{w} = [w_1, w_2, \cdots, w_N]^T \in \mathbb{R}^N$ denote the *weight vector* of the network, $d_{kp} \in \mathbb{R}$ be the desired value of the kth output for pth pattern, and $o_{kp}(\vec{w}) \in \mathbb{R}$ be the actual value of the kth output for pth pattern. The corresponding error function, $f : \mathbb{R}^N \to \mathbb{R}$, is then defined as

$$f(\vec{w}) = \frac{1}{2}\vec{e}(\vec{w})^T\vec{e}(\vec{w}) \tag{1}$$

where $\vec{e}(\vec{w}) = [e_{11}(\vec{w}), \cdots, e_{K1}(\vec{w}), e_{12}(\vec{w}), \cdots, e_{K2}(\vec{w}), \cdots, e_{1P}(\vec{w}), \cdots, e_{KP}(\vec{w})]^T$ is the *error vector*, $e_{kp}(\vec{w}) = d_{kp} - o_{kp}(\vec{w})$ is the error related to the kth output for pth pattern, $k = 1, \cdots, K$, $p = 1, \cdots, P$.

If we let $\vec{w}_n$ represent the network weights at the nth iteration, then the Newton's method for minimizing $f(\vec{w})$ generates the following well-know recurrence formula

$$\vec{w}_{n+1} = \vec{w}_n - [Hf(\vec{w}_n)]^{-1}\nabla f(\vec{w}_n)$$

where $\nabla f(\vec{w}_n) \in \mathbb{R}^N$ is the gradient of $f(\vec{w})$ evaluated at $\vec{w}_n$, $Hf(\vec{w}_n) \in \mathbb{R}^{N\times N}$ is the Hessian matrix of $f(\vec{w})$ evaluated at $\vec{w}_n$, $n \geq 0$. However, direct application of Newton's method to the training of medium or large sized neural networks would not be very successful for two reasons: the cost for computing the exact Hessian matrix is too expensive and the positive definite requirement for the Hessian matrix is too restrictive.

The LM method provides a fix to this problem by using a positive definite Hessian matrix approximation as

$$Hf(\vec{w}) \approx \mathbf{J}(\vec{w})^T\mathbf{J}(\vec{w}) + \mu\mathbf{I}$$

where

$$\mathbf{J}(\vec{w}) = \begin{bmatrix} \frac{\partial e_{11}(\vec{w})}{\partial w_1} & \frac{\partial e_{11}(\vec{w})}{\partial w_2} & \cdots & \frac{\partial e_{11}(\vec{w})}{\partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_{K1}(\vec{w})}{\partial w_1} & \frac{\partial e_{K1}(\vec{w})}{\partial w_2} & \cdots & \frac{\partial e_{K1}(\vec{w})}{\partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_{1P}(\vec{w})}{\partial w_1} & \frac{\partial e_{1P}(\vec{w})}{\partial w_2} & \cdots & \frac{\partial e_{1P}(\vec{w})}{\partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_{KP}(\vec{w})}{\partial w_1} & \frac{\partial e_{KP}(\vec{w})}{\partial w_2} & \cdots & \frac{\partial e_{KP}(\vec{w})}{\partial w_N} \end{bmatrix}$$

is the Jacobian matrix, $\mu > 0$, $\mathbf{I}$ is an identity matrix of size $N \times N$. The weights update rule is given by

$$\vec{w}_{n+1} = \vec{w}_n - \left[\mathbf{J}(\vec{w}_n)^T\mathbf{J}(\vec{w}_n) + \mu_n\mathbf{I}\right]^{-1}\nabla f(\vec{w}_n). \tag{2}$$

As we can see, the LM algorithm requires the inverse of $\mathbf{J}(\vec{w}_n)^T\mathbf{J}(\vec{w}_n) + \mu_n\mathbf{I}$ at each iteration. The size of the matrix is $N \times N$ where N is the number of weights. Even for medium sized networks, N is a large number. For large sized networks, the speed gained from the reduced number of iterations will be lost in the time spent on matrix inversion during each iteration. Moreover, huge memory space is needed to store the Jacobian and the approximated Hessian matrices. In the next section, a new algorithm is developed, which greatly reduces the computational complexities.

III. A New Training Algorithm

The new algorithm is conceptually similar to the LM method. But it uses a different Hessian matrix approximation based on a new Jacobian matrix derived from an *aggregated error vector*. The aggregated error vector generates exactly the same error function but a smaller sized Jacobian matrix under a proper aggregation scheme. Based on this new approximation, the size of the matrix to be inverted in each iteration is reduced by the Sherman-Morrison-Woodbury formula [9].

A. Aggregated Error Vector

Let $\mathcal{I} = \{(k,p) : 1 \leq k \leq K,\ 1 \leq p \leq P,\ k \in \mathbb{N},\ p \in \mathbb{N}\}$ be the index set for training errors in (1) where K and P are defined in Section II, $\mathbb{N}$ denotes the set of nonnegative integers. If $\mathcal{C} = \{\mathcal{I}_i : 1 \leq i \leq M,\ i \in \mathbb{N},\ \mathcal{I}_i \neq$

$\emptyset\}$ is a partition of $\mathcal{I}$, i.e., $\bigcup_{i=1}^{M} \mathcal{I}_i = \mathcal{I}$, and $\mathcal{I}_i \cap \mathcal{I}_j = \emptyset$ for all $1 \leq i, j \leq M$, $i \neq j$, then under the partition $\mathcal{C}$, we define an aggregated error vector $\vec{\tilde{e}}(\vec{w})$ for the training errors in (1) as

$$\vec{\tilde{e}}(\vec{w}) = [\tilde{e}_1(\vec{w}), \tilde{e}_2(\vec{w}), \cdots, \tilde{e}_M(\vec{w})]^T$$

where

$$\tilde{e}_i(\vec{w}) = \sqrt{\sum_{a \in \mathcal{I}_i} e_a(\vec{w})^2}$$

is named an *aggregated error*, $1 \leq i \leq M$.

Based on this aggregated error vector, a new error function is defined as

$$\tilde{f}(\vec{w}) = \frac{1}{2} \vec{\tilde{e}}(\vec{w})^T \vec{\tilde{e}}(\vec{w}). \tag{3}$$

The corresponding gradient is

$$\nabla \tilde{f}(\vec{w}) = \tilde{\mathbf{J}}(\vec{w})^T \vec{\tilde{e}}(\vec{w})$$

where

$$\tilde{\mathbf{J}}(\vec{w}) = \begin{bmatrix} \frac{\partial \tilde{e}_1(\vec{w})}{\partial w_1} & \frac{\partial \tilde{e}_1(\vec{w})}{\partial w_2} & \cdots & \frac{\partial \tilde{e}_1(\vec{w})}{\partial w_N} \\ \frac{\partial \tilde{e}_2(\vec{w})}{\partial w_1} & \frac{\partial \tilde{e}_2(\vec{w})}{\partial w_2} & \cdots & \frac{\partial \tilde{e}_2(\vec{w})}{\partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \tilde{e}_M(\vec{w})}{\partial w_1} & \frac{\partial \tilde{e}_M(\vec{w})}{\partial w_2} & \cdots & \frac{\partial \tilde{e}_M(\vec{w})}{\partial w_N} \end{bmatrix} \tag{4}$$

is called the *aggregated Jacobian matrix*.

There are several interesting relationships between $\tilde{f}(\vec{w})$ and $f(\vec{w})$, which are listed as follows:

- They are essentially identical functions of $\vec{w}$, i.e., for any $\vec{w} \in \mathbb{R}^N$, $\tilde{f}(\vec{w}) = f(\vec{w})$. Therefore, minimizing $f(\vec{w})$ is equivalent to minimizing $\tilde{f}(\vec{w})$.
- Consequently, they have same gradients and same Hessian matrices, i.e., $\nabla \tilde{f}(\vec{w}) = \nabla f(\vec{w})$ and $H\tilde{f}(\vec{w}) = Hf(\vec{w})$ for any $\vec{w} \in \mathbb{R}^N$.
- Let $\tilde{\mathbf{J}}_i$ be the ith row of $\tilde{\mathbf{J}}(\vec{w})$, then $\tilde{\mathbf{J}}_i$ can be equivalently written as

  $$\tilde{\mathbf{J}}_i = \sum_{a \in \mathcal{I}_i} \frac{e_a(\vec{w})}{\tilde{e}_i(\vec{w})} \mathbf{J}_a \tag{5}$$

  where $\mathbf{J}_a$ is the row of $\mathbf{J}(\vec{w})$ corresponding to error $e_a(\vec{w})$. In other words, the ith row of $\tilde{\mathbf{J}}(\vec{w})$ is a linear combination of rows of $\mathbf{J}(\vec{w})$ associated with the errors that are indexed by $\mathcal{I}_i$. In this sense, the aggregated Jacobian matrix can be viewed as a compressed version of the original Jacobian matrix, and the compression is lossy, though. More important, equation (5) gives an efficient way of computing $\tilde{\mathbf{J}}(\vec{w})$.
- The size of $\tilde{\mathbf{J}}(\vec{w})$ is $M \times N$, while the size of $\mathbf{J}(\vec{w})$ is $KP \times N$ where KP is the total number of errors (the number of elements in $\mathcal{I}$). M is always less than or equal to KP. If $M = KP$, then $\tilde{\mathbf{J}}(\vec{w}) = \mathbf{J}(\vec{w})$.

Next it comes to the question of how to aggregate the errors (or equivalently, how to partition $\mathcal{I}$). In this paper, we use a *random aggregation* scheme. There is no guarantee that this scheme is optimal, but it is simple and works very well in practice, which will be demonstrated in Section IV. And we believe that more complex schemes should be motivated by the failure of a simple one. The random aggregation scheme used in the experiments of Section IV can be summarized as below:

Step 1. *Specify block size $b \in \mathbb{N}$, $1 \leq b \leq N$. Set $i = 1$.*
Step 2. *If there are more than b elements in $\mathcal{I}$, then randomly take b elements out of $\mathcal{I}$ and put them into $\mathcal{I}_i$, otherwise move all elements of $\mathcal{I}$ to $\mathcal{I}_i$.*
Step 3. *If $\mathcal{I} = \emptyset$ then terminate the process, otherwise increase i by 1 and return to* **Step 2**.

Under this scheme, the errors are partitioned into M groups where M equals the smallest integer that is greater than or equal to $\frac{KP}{b}$. As we will see shortly, the computational complexities in each iteration can be reduced if b is chosen to make $\frac{KP}{b} < N$. However, an overly large b may lead to more training iterations and hence longer learning time. A discussion of selecting a proper value of b is given in Section III-C.

B. The TREAT Algorithm

From (3), the Hessian matrix of $\tilde{f}(\vec{w})$ (or equivalently $f(\vec{w})$) can be expressed as

$$H\tilde{f}(\vec{w}) = \tilde{\mathbf{J}}(\vec{w})^T \tilde{\mathbf{J}}(\vec{w}) + \tilde{\mathbf{G}}(\vec{w}) \tag{6}$$

where $\tilde{\mathbf{G}}(\vec{w}) = \sum_{i=1}^{M} \tilde{e}_i(\vec{w}) [H\tilde{e}_i(\vec{w})]$, $H\tilde{e}_i(\vec{w}) \in \mathbb{R}^{N \times N}$ is the Hessian matrix of $\tilde{e}_i(\vec{w})$. To simplify computation, (6) is approximated as

$$H\tilde{f}(\vec{w}) \approx \tilde{\mathbf{J}}(\vec{w})^T \tilde{\mathbf{J}}(\vec{w}) + \mu \mathbf{I}$$

where $\mu > 0$, $\mathbf{I}$ is an $N \times N$ identity matrix. Base upon this approximation, the weights update rule of the TREAT algorithm becomes

$$\vec{w}_{n+1} = \vec{w}_n - \lambda_n \left[\tilde{\mathbf{J}}(\vec{w}_n)^T \tilde{\mathbf{J}}(\vec{w}_n) + \mu_n \mathbf{I} \right]^{-1} \nabla f(\vec{w}_n) \tag{7}$$

where $\lambda_n > 0$, $\mu_n > 0$. In contrast to the weights update rule of the LM method given by (2), (7) introduces an extra parameter λ_n for line search and uses a different Hessian matrix approximation. Because $\tilde{\mathbf{J}}(\vec{w}_n)^T \tilde{\mathbf{J}}(\vec{w}_n) +$

$\mu_n\mathbf{I}$ is always positive definite, $(\vec{w}_{n+1}-\vec{w}_n)^T\nabla f(\vec{w}_n) < 0$. Thus at each iteration, we can decrease the value of f by choosing proper values for λ_n and μ_n [6].

Up to now, we have not seen any computational improvement compared with the LM method. Although the size of $\tilde{\mathbf{J}}(\vec{w}_n)$ is smaller that that of $\mathbf{J}(\vec{w}_n)$, the size of $\tilde{\mathbf{J}}(\vec{w}_n)^T\tilde{\mathbf{J}}(\vec{w}_n)$ is still $N \times N$. We still have to invert an $N \times N$ matrix in each iteration. However, as shown below, we convert the original $N \times N$ matrix inversion into an $M \times M$ matrix inversion using the Sherman-Morrison-Woodbury formula [9]. The computational cost for each iteration is reduced if the aggregated error vector has a dimension lower than N.

The Sherman-Morrison-Woodbury formula states that if a matrix $\mathbf{A}$ satisfies $\mathbf{A} = \mathbf{B}^{-1} + \mathbf{C}\mathbf{D}^{-1}\mathbf{C}^T$, then the inverse of $\mathbf{A}$ can be written as

$$\mathbf{A}^{-1} = \mathbf{B} - \mathbf{B}\mathbf{C}(\mathbf{D} + \mathbf{C}^T\mathbf{B}\mathbf{C})^{-1}\mathbf{C}^T\mathbf{B}.$$

If we let $\mathbf{A} = \tilde{\mathbf{J}}(\vec{w}_n)^T\tilde{\mathbf{J}}(\vec{w}_n) + \mu_n\mathbf{I}$, $\mathbf{B} = \frac{1}{\mu_n}\mathbf{I}$, $\mathbf{C} = \tilde{\mathbf{J}}(\vec{w}_n)^T$, and $\mathbf{D} = \mathbf{I}$, then

$$\left[\tilde{\mathbf{J}}(\vec{w}_n)^T\tilde{\mathbf{J}}(\vec{w}_n) + \mu_n\mathbf{I}\right]^{-1} = \frac{1}{\mu_n}\mathbf{I} - \frac{1}{\mu_n^2}\tilde{\mathbf{J}}(\vec{w}_n)^T\left[\mathbf{I} + \frac{1}{\mu_n}\tilde{\mathbf{J}}(\vec{w}_n)\tilde{\mathbf{J}}(\vec{w}_n)^T\right]^{-1}\tilde{\mathbf{J}}(\vec{w}_n). \quad (8)$$

Substituting (8) into (7) gives

$$\vec{w}_{n+1} = \vec{w}_n - \lambda_n\vec{\delta}_n \quad (9)$$

with

$$\vec{\delta}_n = \frac{1}{\mu_n}\nabla f(\vec{w}_n) - \frac{1}{\mu_n^2}\tilde{\mathbf{J}}(\vec{w}_n)^T \left[\mathbf{I} + \frac{1}{\mu_n}\tilde{\mathbf{J}}(\vec{w}_n)\tilde{\mathbf{J}}(\vec{w}_n)^T\right]^{-1}\tilde{\mathbf{J}}(\vec{w}_n)\nabla f(\vec{w}_n). \quad (10)$$

Noticing that $\tilde{\mathbf{J}}(\vec{w}_n)\tilde{\mathbf{J}}(\vec{w}_n)^T$ is an $M \times M$ matrix, instead of inverting an $N \times N$ matrix in each iteration, we are now inverting an $M \times M$ matrix. Computation intensity can be lowered when the block size b of the random aggregation scheme is set to make $M < N$.

The TREAT algorithm can be outlined as follows.

Step 1. *Set $n = 0$. Initialize network weights $\vec{w}_n$. Specify positive parameters b, ϵ, λ_n, S_λ, λ_{max}, λ_{min}, μ_n, S_μ, μ_{max}, μ_{min}, and n_{max} where*

- *b indicates the block size for random aggregation;*
- *ϵ states the stop criterion of the algorithm;*
- *S_λ, λ_{max}, and λ_{min} are the scale factor, maximum and minimum values for λ_n, respectively;*
- *S_μ, μ_{max}, and μ_{min} are the scale factor, maximum and minimum values for μ_n, respectively;*
- *n_{max} is the maximumly allowed number of iterations.*

Step 2. *Partition errors according to the random aggregation scheme in Section III-A.*

Step 3. *Compute network outputs and evaluate error function $f(\vec{w}_n)$ (or equivalently $\tilde{f}(\vec{w}_n)$).*

Step 4. *Set $\mu_{n+1} = \mu_n$. Evaluate the aggregated Jacobian matrix $\tilde{\mathbf{J}}(\vec{w}_n)$ according to (4). Calculate the gradient $\nabla f(\vec{w}_n)$ (or equivalently $\nabla\tilde{f}(\vec{w}_n)$). Compute $\vec{\delta}_n$ according to (10).*

Step 5. *Update weights according to (9).*

Step 6. *Compute network outputs and evaluate error function $f(\vec{w}_{n+1})$ (or equivalently $\tilde{f}(\vec{w}_{n+1})$. If $f(\vec{w}_{n+1}) \geq f(\vec{w}_n)$ (or equivalently $\tilde{f}(\vec{w}_{n+1}) \geq \tilde{f}(\vec{w}_n)$) then:*

- *Set $\lambda_n = \frac{\lambda_n}{S_\lambda}$, $\mu_{n+1} = \mu_{n+1}S_\mu$.*
- *If $\lambda_n < \lambda_{min}$ or $\mu_{n+1} > \mu_{max}$ then stop training else go back to* **Step 5**.

Otherwise:

- *If $f(\vec{w}_{n+1}) < \frac{\epsilon}{2}$ then stop training.*
- *Set $\lambda_{n+1} = \lambda_n S_\lambda$, $\mu_{n+1} = \frac{\mu_{n+1}}{S_\mu}$. If $\lambda_{n+1} > \lambda_{max}$ then $\lambda_{n+1} = \lambda_{max}$. If $\mu_{n+1} < \mu_{min}$ then $\mu_{n+1} = \mu_{min}$.*
- *Set $n = n + 1$. If $n > n_{max}$ then stop training, otherwise go back to* **Step 4**.

C. Computational Complexities

Computational loads in terms of *flops* [1] can be estimated based on (9) and (2), in which the Cholesky factorization [9] is used to realize the matrix inversion. The TREAT algorithm demands approximately $\frac{M^3}{3}$ flops to evaluate (9) where M is the number of rows of the aggregated Jacobian matrix $\tilde{\mathbf{J}}(\vec{w}_n)$. Similarly, the LM method needs about $\frac{N^3}{3}$ flops to calculate $\vec{w}_{n+1}$. The major memory requirement comes from the storage of the Jacobian matrix (or aggregated Jacobian matrix), the approximated Hessian matrix (only its upper triangular part is stored due to symmetry), and a triangular matrix from the Cholesky factorization. This gives an approximate memory requirement of $MN + M^2$ for the TREAT algorithm and $KPN + N^2$ for the LM method. From this rough analysis, we can safely claim that, for large N (medium sized or lager sized neural networks), the TREAT algorithm uses less memory and smaller number of flops in each iteration than the LM method does if we make $M < \min(KP, N)$ by choosing a proper b. In our experiments b is chosen from the integers that are greater than 1 and make $\frac{N}{2} \leq M < N$. We set $b = 2$ if no such integers exist. Clearly, this will always make

[1] A flop is a floating addition, subtraction, multiplication, or division.

$M < \min(KP, N)$. As demonstrated in Section IV, this simple choice of b leads to a slightly increase of the number of iterations yet a large decrease in the total training flops compared with the LM method.

IV. Experimental Results

The TREAT algorithm has been tested on a large number of problems. Due to space limitation, we only present experimental results for two problems. Comparisons with the EBP with momentum (EBPM) and the LM method in terms of success rates, the number of training iterations, the number of flops, and, in some cases, generalization errors are provided.

The EBPM weights update rule is given as

$$\vec{w}_{n+1} = \vec{w}_n - \rho \nabla f(\vec{w}_n) + \alpha(\vec{w}_n - \vec{w}_{n-1})$$

where ρ is the learning rate and α is the *momentum rate*. Parameters corresponding to the LM method are selected as $S = 10$, $\mu_0 = 0.01$, $\mu_{min} = 10^{-50}$, and $\mu_{max} = 10^{50}$. For the TREAT algorithm, parameters common for all experiments are chosen as $\lambda_0 = \lambda_{max} = 1$, $\mu_0 = 0.01$, $S_\lambda = S_\mu = 10$, $\lambda_{min} = \mu_{min} = 10^{-50}$, and $\mu_{max} = 10^{50}$. Feedforward neural networks with one hidden layer are used. We call them *I-H-O* networks where I is the number of network inputs, H is the number of hidden neurons, and O is the number of outputs. Hidden neurons have tangent sigmoid activation functions. The output layer is linear. All weights are randomly initialized to $[-1, 1]$.

A. Approximation of $h(x) = \sin(2\pi x)\cos(4\pi x)$, $x \in [0, 1]$

A 1-7-1 network (22 weights) is used here. The training set consists of 50 input-output pairs where the input values are uniformly scattered in the interval $[0, 1]$. To examine the generalization ability of the network, 50 new samples are randomly picked and tested after each successful training. The learning rate and momentum rate for the EBPM algorithm are 0.01 and 0.9, respectively. For the TREAT algorithm, b is chosen to be 3. Table I summarizes the experimental results. ϵ is the error goal. A training process is considered successful if the error function (1) is less than $\frac{\epsilon}{2}$, i.e., the summation of the squared errors for all training samples is less than ϵ. n_{max} is the maximumly allowed number of training iterations before declaring a failure of convergence. n_{trial} is the number of independent runs of training algorithms. The percentile of successful runs (success rate) is denoted by $r_{success}$. $\text{MEAN}_{\text{iterations}}$ represents the mean of the number of training iterations for all successful runs. $\text{STDV}_{\text{iterations}}$ represents the standard deviation of the number of training iterations for all successful runs. $\text{MEAN}_{\text{flops}}$ shows the mean of the number of flops for all successful runs. $\text{STDV}_{\text{flops}}$ shows the standard deviation of the number of flops for all successful runs. $\text{MEAN}_{\text{errors}}$ indicates the mean of generalization error, which is defined as the sum of squared output errors for all testing samples, for all successful runs. $\text{STDV}_{\text{errors}}$ indicates the standard deviation of generalization error for all successful runs.

TABLE I

Experimental results for approximation of $h(x) = \sin(2\pi x)\cos(4\pi x)$, $x \in [0, 1]$.

Update Rules	**TREAT**	**LM**	**EBPM**
ϵ	0.1	0.1	0.5
n_{max}	500	500	10000
n_{trial}	100	100	100
$r_{success}$	99.0%	100%	62.0%
$\text{MEAN}_{\text{iterations}}$	97.72	86.00	3173.4
$\text{STDV}_{\text{iterations}}$	32.85	29.69	1939.4
$\text{MEAN}_{\text{flops}}$	4.24×10^6	7.36×10^6	6.48×10^7
$\text{STDV}_{\text{flops}}$	1.44×10^6	2.57×10^6	3.96×10^7
$\text{MEAN}_{\text{errors}}$	0.0868	0.0776	0.4705
$\text{STDV}_{\text{errors}}$	0.0133	0.0195	0.0976

B. Parity 5 Problem

The parity problem is considered a hard classification problem, despite the fact that it does not give any information on the generalization ability of the network. A 5-6-1 network (43 weights) is trained to classify all 32 patterns, defined on $\{-1, 1\}^5$, into two classes with corresponding class labels 1 and -1. The learning and momentum rates for the EBPM algorithm are 0.01 and 0.9, respectively. The TREAT algorithm uses block size 2. As shown in Table II, the highest success rate is given by the LM method, followed by the TREAT and EBPM algorithms. The disparities are very small (2%) though. As expected, the LM method requires the least number of training iterations on the average, but the smallest number of flops is given by the TREAT algorithm (the size of the matrix to be inverted in each iteration is 16×16 for the TREAT algorithm and 43×43 for the LM method).

TABLE II

Experimental results for parity 5 problem.

Update Rules	**TREAT**	**LM**	**EBPM**
ϵ	0.1	0.1	0.1
n_{max}	200	200	1000
n_{trial}	100	100	100
$r_{success}$	91.0%	93.0%	89.0%
$\text{MEAN}_{\text{iterations}}$	41.18	23.24	156.94
$\text{STDV}_{\text{iterations}}$	25.94	28.26	162.91
$\text{MEAN}_{\text{flops}}$	2.80×10^6	5.20×10^6	5.71×10^6
$\text{STDV}_{\text{flops}}$	1.82×10^6	6.67×10^6	5.96×10^6

V. Conclusions

The TREAT algorithm developed in this article carries an idea similar to the LM method but uses a different Hessian matrix approximation based on the Jacobian matrix derived from aggregated errors. The proposed aggregation scheme can reduce the size of the matrix to be inverted in each training iteration and therefore lower the iterative computational cost. Compared with the LM method, the TREAT algorithm requires less memory and smaller number of flops if the block size of aggregated errors is chosen judiciously. The TREAT algorithm is experimentally verified on extensive examples. Simulation results demonstrate its superiority.

References

[1] S. Abid, F. Fnaiech, and M. Najim, "A Fast Feedforward Training Algorithm Using a Modified Form of the Standard Backpropagation Algorithm", *IEEE Trans. on Neural Networks*, vol 12, no. 2, pp. 424-430, 2001.

[2] R. Battiti and F. Masulli, "BFGS Optimization for Faster Automated Supervised Learning", in *Int. Neural-Network Conf.*, vol. 2, pp. 757-760, 1990.

[3] R. Battiti, "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method", *Neural Computation*, vol. 4, no. 2, pp. 141-166, 1992.

[4] S. Becker and Y. Le Cun, "Improving the Convergence of Backpropagation Learning with Second Order Methods", in *Proc. of the 1988 Connectionist Models Summer School*, pp. 29-37, 1989.

[5] C. Charalambous, "Conjugate Gradient Algorithm for Efficient Training of Artificial Neural Networks", *IEE Proceedings Part G*, vol. 139, no. 3, pp. 301-310, 1992.

[6] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[7] S. Ergezinger and E. Thomsen, "An Accelerated Learning Algorithm for Multilayer Perceptrons: Optimization Layer by Layer", *IEEE Trans. on Neural Networks*, vol 6, no. 1, pp. 31-42, 1995.

[8] S. E. Fahlman, "Fast Learning Variations on Back-Propagation: An Empirical Study", in *Proc. of the 1988 Connectionist Models Summer School*, pp. 38-51, 1989.

[9] G. H. Golub and C. F. Van Loan, *Matrix Computations,* 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.

[10] M. T. Hagan, M. B. Menhaj, "Training Feedforward Networks with the Marquardt Algorithm", *IEEE Trans. on Neural Networks*, vol. 5, no. 6, pp. 989-993, 1994.

[11] D. R. Hush, J. M. Salas, and B. Horne, "Error Surfaces for Multi-Layer Perceptrons", in *Proc. of Int. Joint Conf. on Neural Networks*, Seattle, WA USA, vol 1, pp. 759-764, 1991.

[12] R. A. Jacobs, "Increased Rates of Convergence Through Learning Rate Adaptation", *Neural Networks*, vol. 1, no.4, pp. 295-308, 1988.

[13] N. B. Karayiannis, "Accelerating the Training of Feedforward Neural Networks Using Generalized Hebbian Rules for Initializing the Internal Representations", *IEEE Trans. on Neural Networks*, vol. 7, no. 2, pp. 419-426, 1996.

[14] A. Krogh, G. I. Thorbergsson, J. A. Hertz, "A Cost Function for Internal Representations", in *Advances in Neural Information Processing Systems II*, San Mateo, CA USA, pp. 733-740, 1989.

[15] T. M. Kwon and H. Cheng, "Contrast Enhancement for Backpropagation", *IEEE Trans. on Neural Networks*, vol 7, no. 2, pp. 515-524, 1996.

[16] K. J. Lang and M. J. Witbrock, "Learning to Tell Two Spirals apart", in *Proc. of the 1988 Connectionist Models Summer School*, pp. 52-59, 1989.

[17] H. A. Malki and A. Moghaddamjoo, "Using the Karhunen-Loe'vd Transformation in the Back-Propagation Training Algorithm", *IEEE Trans. on Neural Networks*, vol 2, no. 1, pp. 162-165, 1991.

[18] S. McLoone, M. D. Brown, G. Irwin, and G. Lightbody, "A Hybrid Linear/Nonlinear Training Algorithm for Feedforward Neural Networks", *IEEE Trans. on Neural Networks*, vol 9, no. 4, pp. 669-684, 1998.

[19] A. A. Miniani, R. D. Williams, "Acceleration of Back-Propagation Through Learning Rate and Momentum Adaptation", in *Proc. of Int. Joint Conf. on Neural Networks*, San Diego, CA USA, vol 1, pp. 676-679, 1990.

[20] D. Nguyen, and B. Widrow, "Improving the Learning Speed of 2-layer Neural Networks by Choosing Initial Values of the Adaptive Weights", in *Proc. Int. Joint Conf. Neural Networks*, San Diego, CA USA, vol. 3, pp. 21-26, 1990.

[21] S. Oh, "Improving the Error Backpropagation Algorithm with a Modified Error Function", *IEEE Trans. on Neural Networks*, vol 8, no. 3, pp. 799-803, 1997.

[22] A. Van Ooyen and B. Nienhuis, "Improving the Convergence of the Backpropagation Algorithm", *Neural Networks*, vol 5, pp. 465-471, 1992.

[23] R. Parekh, K. Balakrishnan, V. Honavar, "An Empirical Comparison of Flat-Spot Elimination Techniques in Back-Propagation Networks", in *Proceedings of Third Workshop on Neural Networks - WNN92*, pp. 55-60, 1992.

[24] A. G. Parlos, B. Fernandez, A. F. Atiya, J. Muthusami, and W. K. Tsai, "An Accelerated Learning Algorithm for Multilayer Perceptron Networks", *IEEE Trans. on Neural Networks*, vol 5, no. 3, pp. 493-497, 1994.

[25] N. S. Rubanov, "The Layer-Wise Method and the Backpropagation Hybrid Approach to Learning a Feedforward Neural Network", *IEEE Trans. on Neural Networks*, vol 11, no. 2, pp. 295-305, 2000.

[26] D. E. Rumelhart, G. E. Hinton and R. J. Wiliams, "Learning Internal Representations by Back-Propagating Errors", *Nature*, vol. 323, pp. 533-536, 1986

[27] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning Internal Representations by Error Propagation", *Parallel Distributed Processing*, vol 1, pp. 318-362. MIT Press, Cambridge, MA, 1986.

[28] L. Torvik, B. M. Wilamowski, "Modification of the Backpropagation Algorithm for Faster Convergence", in *Proc. of Fourth Workshop on Neural Networks - WNN93*, pp. 191-194, 1993

[29] G. J. Wang and C. C. Chen, "A Fast Multilayer Neural-Network Training Algorithm Based on the Layer-By-Layer Optimizing Procedures", *IEEE Trans. on Neural Networks*, vol 7, no. 3, pp. 768-775, 1996.

[30] J. Y.F. Yam and T. W.S. Chow, "Feedforward Networks Training Speed Enhancement by Optimal Initialization of the Synaptic Coefficients", *IEEE Trans. on Neural Networks*, vol 12, no. 2, pp. 430-434, 2001.