New Results on Recurrent Network Training: Unifying the Algorithms and Accelerating Convergence

Amir F. Atiya, *Senior Member, IEEE,* and Alexander G. Parlos, *Senior Member, IEEE*

***Abstract*—How to efficiently train recurrent networks remains a challenging and active research topic. Most of the proposed training approaches are based on computational ways to efficiently obtain the gradient of the error function, and can be generally grouped into five major groups. In this study we present a derivation that unifies these approaches. We demonstrate that the approaches are only five different ways of solving a particular matrix equation. The second goal of this paper is develop a new algorithm based on the insights gained from the novel formulation. The new algorithm, which is based on approximating the error gradient, has lower computational complexity in computing the weight update than the competing techniques for most typical problems. In addition, it reaches the error minimum in a much smaller number of iterations. A desirable characteristic of recurrent network training algorithms is to be able to update the weights in an on-line fashion. We have also developed an on-line version of the proposed algorithm, that is based on updating the error gradient approximation in a recursive manner.**

***Index Terms*—Backpropagation through time, constrained optimization, gradient approximation, optimal control, real time recurrent learning, recurrent networks.**

I. Introduction

RECURRENT networks are basically dynamic systems where the states evolve according to certain nonlinear state equations. Because of their dynamic nature, they have been successfully applied to many problems involving modeling and processing of temporal signals, such as adaptive control, forecasting, system identification, and speech recognition [1]–[9]. Despite the potential and capability of recurrent networks, the main problem is the difficulty of training them, and the complexity and slow convergence of the existing training algorithms.

The training problem consists of adjusting the parameters (weights) of the network, so that the trajectories have certain specified properties. Because of the fact that a weight adjustment affects the states at all times during the course of network state evolution, obtaining the error gradient is a rather complicated procedure. However, methods from the well-developed optimal control theory and dynamic programming were of help in giving some insights into the problem. Several training methods have been developed, that are basically different computational methods to obtain the gradient [10]–[20]. Some of these methods compute the gradient very efficiently. However, the main shortcoming of these methods is the excessive number of iterations needed to reach the minimum. Only few methods in the literature attempt to overcome this shortcoming. To name a few, some are based on approximating the gradient (the truncated backpropagation through time (BTT) approach and the BTT(h, h') approach [21]), others are based on novel formulations of the training problem (e.g., the extended Kalman filter approach [6], [22], and the EM-based algorithm [23], [24]), and some are based on novel architectures (e.g., the focused backpropagation [25], the gating-unit-based method of [26] and [27]), and most recently the approximated Levenberg–Marquardt algorithm [53].

The first goal of this paper is to present a derivation that unifies the existing recurrent network *gradient-based* training algorithms (we are considering here only the problem of learning trajectories, not learning fixed points such as [28], [29] and [30]). We demonstrate that the algorithms are only several different ways of solving a particular matrix equation (see a preliminary study in [31]). The second goal is to develop a new recurrent network algorithm based on the insights gained from the novel formulation. The algorithm has two versions: an off-line version and an on-line version. Both versions are faster than their respective competing algorithms.

This paper is organized as follows. In the next section we describe briefly the five main recurrent network training algorithms. Section III presents preliminaries and definitions. In Section IV we present the novel formulation, and show how this unifies the main existing algorithms. In Section V we develop the new training algorithm. The algorithm is tested through some numerical simulations in Section VI. Section VII presents the conclusions of this work.

II. An Overview of the Algorithms

The recurrent network training algorithms can be grouped into five major categories, which are described below (for details of the algorithms refer to the cited references and to the reviews [32]–[34]).

1) The forward fropagation approach (FP) [10], [11], also called real time recurrent learning (RTRL): The weights

Manuscript received October 11, 1997; revised January 5, 1999 and December 21, 1999. The work of A. Atiya was supported by NSF's Engineering Research Center at Caltech. The work of A. G. Parlos was supported by the Texas Advanced Technology and Research Program and the US Department of Energy to Texas A&M University.

A. F. Atiya is with Learning Systems Group, Department of Electrical Engineering, California Institute of Technology, Pasadena, CA 91125 USA (e-mail: amir@work.caltech.edu) .

A. G. Parlos is with the Department of Mechanical Engineering, Texas A&M University, College Station, TX 77843 USA.

Publisher Item Identifier S 1045-9227(00)03004-6.

are updated in an on-line fashion. The gradients of the network states with regard to the weights at time instant k are obtained in terms of those at time instant $k-1$. Once these are evaluated, the error gradients can be obtained in a straightforward way. This method is suitable for applications which need an on-line adaptation property. Its drawback is the large computational complexity. $O(N^4)$ computations are performed for each data point, where N denotes the number of nodes in the network.

2) The BTT [12]–[14]: This is a computationally very efficient off-line training technique. The basic idea is to turn the recurrent network into an equivalent feedforward network by unfolding time iterations into layers, each layer with identical weights. The equivalent feedforward network is trained using the backpropagation algorithm. For off-line applications the method needs only $O(N^2)$ computations per data point. On the other hand, for on-line applications the method is not very practical, since for each data point the error gradients have to be propagated all the way to time $k=1$.
3) The fast forward propagation approach (FFP) [15], [16]: The drawback of the BTT method is its inefficiency in the case of on-line update. The FFP approach is a procedure for obtaining the gradients recursively, and is thus an on-line technique. The basic idea of the FFP technique is that the boundary conditions of the backpropagated gradients at time $k=1$ are obtained recursively. This allows solving the gradient recursion forward in time, rather than backwards. The computational complexity of this method is $O(N^3)$.
4) The Green's function (GF) approach [17]: In this approach the computational complexity of the FP approach is improved, by considering the recursive equations for the output gradients. A Green's function solution is computed, from which the sought error gradient is obtained by a simple dot product. This method is an on-line technique, and the complexity is $O(N^3)$.
5) The block update approach (BU) [18]–[20]: This is an on-line approach that updates the weights every $O(N)$ data points using some aspects of the FP and BTT methods. Each update has complexity $O(N^4)$. Since the update is performed every $O(N)$ data points, the complexity per data point is $O(N^3)$.

In this paper we present a novel formulation to unify these five algorithms. This also serves as a review of these algorithms. We also wish to refer the reader to an alternate study [35], [36], which connects the FP method with the BTT method using the concept of flow graph interreciprocity and Tellegen's theorem. Another study [32] uses adjoint systems for the differential equation formulation. We wish to note that the formulation presented is based on the standard fully recurrent network model. It can be easily generalized for the many other models which use recurrent network training principles [1], [37]–[46]. In fact, networks which possess a feedforward component in addition to a recurrent component [1], [37], [38] have been shown to be easier to train (less prone to local minimum) than fully connected recurrent networks, and have therefore found many applications. Other models, such as spatiotemporal or finite impulse response (FIR) networks [39]–[41], networks with trainable delays [42], NL_q networks [43], memory neuron networks [44], generalized recursive neuron networks [45], [46], etc, have shown much success, and can be trained with some modifications of existing recurrent network algorithms. Also, for the problem of training a feedforward neural-network controller with an outer control loop (see for example [7] and [47]), it is preferable to train it using recurrent network training principles, because of the composite recurrent structure. Practice has shown in such a case that training the network using the standard (feedforward based) backpropagation algorithm—which amounts to looking one-step-ahead in time—is not as powerful as training the whole structure using a recurrent network-type algorithm. This also applies for feedforward networks used for recursive multi-step-ahead forecast. Recurrent training algorithms perform better than the standard backpropagation algorithm trained to predict simply the next-step-ahead [48]. For all these models, the proposed formulation will still apply with some modification. Also, the algorithm to be developed in the later part of the paper can be easily adapted for these models.

III. Preliminaries

Let $x(k) = (x_1(k), \cdots, x_N(k))^T$ be the network state vector (vector of node outputs) at time k, and let W be the weight matrix. Furthermore, let f denote the node function (e.g., sigmoid, tanh, etc.). As an extension, f applied to a vector $y = (y_1, \cdots, y_N)^T$, is defined as $f(y) = (f(y_1), \cdots, f(y_N))^T$. The network dynamics are given by

$$x(k+1) = f[Wx(k)], \qquad k = 0, \cdots, K-1 \tag{1}$$

where k denotes the time index. Note that $x(0)$ represents the vector of initial conditions for the network. It is noted that in the above definition, all external input nodes and network nodes are lumped into one vector $x(k)$, for simplicity. Thus, the input nodes are always clamped at the particular input values: $x_i(k) = u_i(k)$, $i \in I$, where $u(k)$ represents the input vector at time k, and I represents the set of input nodes.

Training the network is accomplished by updating the weights in the negative direction of the error gradient, i.e.,

$$W(\text{new}) = W(\text{old}) - \eta \frac{dE}{dW} \tag{2}$$

where η is the learning rate, and E represents the sum of square error

$$E = \tfrac{1}{2} \sum_{k=1}^{K} \sum_{i \in O} (x_i(k) - d_i(k))^2 \tag{3}$$

where $d_i(k)$ is the desired output for unit i at time step k, and O is the set of output nodes. Most of the recurrent network training algorithms revolve around ways to efficiently evaluate the error gradient. The five major algorithms can be unified as shown next section.

IV. Unification of the Algorithms

The training problem can be formulated in the form of the following constrained minimization problem.

Minimize E, given by (3)

subject to

$$g(k+1) \equiv f[Wx(k)] - x(k+1) = 0 \qquad k = 0, \cdots, K-1. \tag{4}$$

The weights are considered the "control" or "decision variables," according to optimal control terminology [49]. The $x(k)$'s are the "state variables," whose values are determined by the control parameters through the constraint equations (4). Let $g = (g^T(1), \cdots, g^T(K))^T$. Let us arrange the rows w_i^T of W in one long column vector w, i.e., $w = (w_1^T, \cdots, w_N^T)^T$. Similarly, let $x = (x^T(1), \cdots, x^T(K))^T$. To clarify the variable dependencies in the following derivation, let us write x as $x(w)$, E as $E(x(w))$, and g as $g(w, x(w))$. We can write

$$\frac{dE(x(w))}{dw} = \frac{\partial E(x(w))}{\partial w} + \frac{\partial E(x(w))}{\partial x}\,\frac{\partial x(w)}{\partial w}, \tag{5}$$

where $(\partial E(x(w))/\partial w)$ equals zero, because there is no explicit dependence on w. By taking the derivative of the equation $g(w, x(w)) = 0$, we get

$$\frac{\partial g(w, x(w))}{\partial w} + \frac{\partial g(w, x(w))}{\partial x}\,\frac{\partial x(w)}{\partial w} = 0. \tag{6}$$

Solving (5) and (6), we get

$$\frac{dE(x(w))}{dw} = -\frac{\partial E(x(w))}{\partial x}\left(\frac{\partial g(w, x(w))}{\partial x}\right)^{-1}\frac{\partial g(w, x(w))}{\partial w}. \tag{7}$$

A similar derivation of (7), but with more detailed analysis can be found in [49, pp 2–20]. For simplicity, let us from now on drop the notations expressing variable dependencies in (7), to obtain

$$\frac{dE}{dw} = -\frac{\partial E}{\partial x}\left(\frac{\partial g}{\partial x}\right)^{-1}\frac{\partial g}{\partial w}. \tag{8}$$

This equation will be the basis for the unification formulas to be obtained this section. Note the convention here is that $(\partial u/\partial v)$ for two vectors u and v is the matrix whose (i, j)th element is $(\partial u_i/\partial v_j)$.

The matrices in (8) can be evaluated as follows:

$$\frac{\partial E}{\partial x} = (e(1), \cdots, e(K)) \equiv e \tag{9}$$

$$\frac{\partial g}{\partial x} = \begin{pmatrix} -I & 0 & 0 & \cdots & 0 & 0 \\ D(1)W & -I & 0 & \cdots & 0 & 0 \\ 0 & D(2)W & -I & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & D(K-1)W & -I \end{pmatrix} \tag{10}$$

and

$$\frac{\partial g}{\partial w} = \begin{pmatrix} D(0)X(0) \\ D(1)X(1) \\ \vdots \\ D(K-1)X(K-1) \end{pmatrix} \tag{11}$$

where $e(k)$ is a row vector, whose elements are given by

$$[e(k)]_i = \begin{cases} x_i(k) - d_i(k), & i \in O \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

$$D(k) = \operatorname{diag}\left(f'\left(\sum_{j=1}^{N} w_{ij}x_j(k)\right)\right) \tag{13}$$

$$X(k) = \begin{pmatrix} x^T(k) & 0 & 0 & \cdots & 0 \\ 0 & x^T(k) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & x^T(k) \end{pmatrix} \tag{14}$$

I is the identity matrix, and 0 in (10) and (14) is a matrix (or vector) of zeros. In other instances later we will also use the error gradient in a matrix form, rather than row vector form. We will denote this $\partial E/\partial W$ (note the upper case W), and this will mean the matrix with components $\partial E/\partial w_{ij}$.

The recurrent network training algorithms are basically different approaches in solving the matrix equation (8), as will be shown in the following five sections.

A. The Forward Propagation Method

In this approach the product $(\partial g/\partial x)^{-1}\partial g/\partial w$ (say $= Y$) is computed first, and then Y is premultiplied by $\partial E/\partial x$ to obtain dE/dw. This is demonstrated as follows. By the definition of Y, one can write

$$\frac{\partial g}{\partial x}Y = \frac{\partial g}{\partial w}. \tag{15}$$

Let Y be written as

$$Y = \begin{pmatrix} Y(0) \\ Y(1) \\ \vdots \\ Y(K-1) \end{pmatrix} \tag{16}$$

where $Y(k)$ is an $N \times N^2$ partition matrix. Substituting the values of the gradients (10) and (11), into (15), the following equations are obtained:

$$Y(0) = -D(0)X(0) \tag{17}$$

$$Y(k) = D(k)WY(k-1) - D(k)X(k) \qquad k = 1, \cdots, K-1. \tag{18}$$

Thus, in the forward propagation approach, the recursion (18) is solved with initial conditions (17). The final gradient is obtained as

$$\frac{dE}{dw} = -\frac{\partial E}{\partial x}Y = -\sum_{k=1}^{K} e(k)Y(k-1) \tag{19}$$

which follows from (8) and the definition of Y.

B. The Backpropagation Through Time Method

In the BTT approach, on the other hand, we evaluate first $\partial E/\partial x(\partial g/\partial x)^{-1}$, (say $= \delta$), and then postmultiply the result by $\partial g/\partial w$. By the definition above of δ, we can write

$$\frac{\partial E}{\partial x} = \delta \frac{\partial g}{\partial x}. \tag{20}$$

Let

$$\delta = (\delta(1), \cdots, \delta(K)) \tag{21}$$

where $\delta(k)$ is a $1 \times N$ row vector associated with time step k. Note that the $\delta(k)$ here is the same as the δ in the delta rule of backpropagation. The gradient values (9) and (10) are substituted into (20), resulting in

$$e(k) = -\delta(k) + \delta(k+1)D(k)W, \qquad k = 1, \cdots, K-1 \tag{22}$$

$$e(K) = -\delta(K). \tag{23}$$

Solving (22) backwards for the $\delta(k)$'s, starting from the boundary condition (23) at time K leads to the famous backpropagation through time. Substituting the definition of δ, $\delta = \partial E/\partial x(\partial g/\partial x)^{-1}$ into (8), we get the error gradient

$$\frac{dE}{dw} = -\delta \frac{\partial g}{\partial w}. \tag{24}$$

It is easier to express it in the form of a matrix dE/dW. After little manipulation, we get

$$\frac{dE}{dW} = -\sum_{k=1}^{K} D(k-1)\delta^T(k)x^T(k-1). \tag{25}$$

C. The Fast Forward Propagation Method

Assume a solution is obtained for the system (22), (23) of K equations. When a new data point arrives, there is no point in solving the new system of $K+1$ equations from scratch by propagating the gradients back according to (22) till $K = 1$, but rather, the old solution of the K equations is utilized. The update (22), we observe, represents iteration of a linear system [for $\delta(k)$] with boundary conditions at $k = K$. If the boundary conditions at $k = 1$ are known, the system can be solved equally well forward in time. The FFP determines these initial conditions recursively, and based on these it evaluates the gradients.

Let $\delta^{(K)}(k)$ be the solution of the system (22), (23) of K equations. The following linear equation is obtained upon solving (22) forwards:

$$\delta^{(K)}(k) = \delta^{(K)}(1)A(k) + b(k) \tag{26}$$

where

$$A(k) = W^{-1}D^{-1}(1)\cdots W^{-1}D^{-1}(k-1) \tag{27}$$

$$\begin{aligned} b(k) = {} & e(1)W^{-1}D^{-1}(1)W^{-1}D^{-1}(2)\cdots W^{-1}D^{-1}(k-1) \\ & + e(2)W^{-1}D^{-1}(2)\cdots W^{-1}D^{-1}(k-1) + \cdots \\ & + e(k-1)W^{-1}D^{-1}(k-1). \end{aligned} \tag{28}$$

As can be seen $A(k)$ and $b(k)$ are independent of K. In fact they can be recursively obtained, as will be shown in the description of the algorithm later. The initial condition $\delta^{(K)}(1)$, on the other hand, is changed every time step in order to ensure that (26) satisfies the boundary condition at $k = K$. To obtain $\delta^{(K)}(1)$, the inverse of $\partial g/\partial x$ is first evaluated, to obtain the equation shown in (29) at the bottom of the page. Since $\delta^{(K)} = (\partial E/\partial x)(\partial g/\partial x)^{-1}$, the following recursion can be obtained:

$$\delta^{(K)}(1) = \delta^{(K-1)}(1) - e(K)D(K-1)W\cdots D(1)W. \tag{30}$$

Using (24) and (11), we can write

$$\frac{dE}{dw} = -\sum_{k=1}^{K} \delta(k)D(k-1)X(k-1). \tag{31}$$

Substituting from (26), we get

$$\begin{aligned} \frac{dE}{dw} = {} & -\delta^{(K)}(1)\sum_{k=1}^{K} A(k)D(k-1)X(k-1) \\ & -\sum_{k=1}^{K} b(k)D(k-1)X(k-1). \end{aligned} \tag{32}$$

The basic idea of the algorithm is to obtain the initial condition $\delta^{(K)}(1)$ recursively according to (30), obtain $A(K)$ and $b(K)$ recursively, and then update (32) to obtain the updated gradient. The algorithm is summarized below. First, let $H(K, j)$ and $R(K, j)$ represent an $N \times N$ matrix and a $1 \times N$ row vector, respectively. They are to hold the first sum and the second sum in (32), respectively. Also let $C(K)$ be an $N \times N$ matrix.

1) Initialize: $K = 1$, $\delta^{(1)}(1) = -e(1)$, $A(1) = I$, $b(1) = 0$, $C(1) = I$, $H(0, j) = 0$, $R(0, j) = 0$, $j = 1, \cdots, N$.
2) $H(K, j) = H(K-1, j) + A(K)D(K-1)x_j(K-1)$, $j = 1, \cdots, N$,
   $R(K, j) = R(K-1, j) + b(K)D(K-1)x_j(K-1)$, $j = 1, \cdots, N$,
   $(\partial E/\partial w_{ij})(K) = -\sum_{l=1}^{N} H_{li}(K, j)\delta_l^{(K)}(1) - R_i(K, j)$.
3) $K = K + 1$.
4) $C(K) = D(K-1)WC(K-1)$,
   $\delta^{(K)}(1) = \delta^{(K-1)}(1) - e(K)C(K)$,
   $A(K) = A(K-1)W^{-1}D^{-1}(K-1)$,

$$\left(\frac{\partial g}{\partial x}\right)^{-1} = -\begin{pmatrix} I & 0 & 0 & \cdots & 0 \\ D(1)W & I & 0 & \cdots & 0 \\ D(2)WD(1)W & D(2)W & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ D(K-1)W\cdots D(1)W & D(K-1)W\cdots D(2)W & D(K-1)W\cdots D(3)W & \cdots & I \end{pmatrix}. \tag{29}$$

$b(K) = (b(K-1) + e(K-1))W^{-1}D^{-1}(K-1)$,
Go to 2) until end of data.

D. The Green Function Approach

The GF approach exploits the fact that the matrix $X(k)$ in update (18) of the FP approach has many zeros and has repeated rows, in an attempt to reduce the complexity. Assume the gradient at the previous time step $K-1$, $(dE/dw)(K-1)$, is already evaluated, and we would like to obtain $(dE/dw)(K)$ in terms of it. We evaluate $(dE/dw)(K-1)$ and $(dE/dw)(K)$ by substituting in (8) the values of the constituent matrices from (9), (11), and (29), then subtracting both gradients. This results in

$$\begin{aligned}\frac{dE}{dw}(K) - \frac{dE}{dw}(K-1) &= e(K)\Big(D(K-1)W\cdots D(1),\ D(K-1)W\cdots \\ &\quad D(2)W, \cdots, I\Big) \\ &\quad \cdot \begin{pmatrix} D(0)X(0) \\ D(1)X(1) \\ \vdots \\ D(K-1)X(K-1) \end{pmatrix}. \end{aligned} \tag{33}$$

The matrix in the middle of the right-hand side of the previous equation represents the last block of rows (last N rows) of the inverse matrix in (29). Multiplying the two last matrices in the right-hand side, and taking a common factor $D(K-1)WD(K-2)\cdots D(1)WD(0)$, we get

$$\begin{aligned}\frac{dE}{dw}(K) = \frac{dE}{dw}(K-1) &+ e(K)D(K-1)(WD(K-2) \\ &\cdots D(1)WD(0))[X(0) + D^{-1}(0)W^{-1}X(1) \\ &+ D^{-1}(0)W^{-1}D^{-1}(1)W^{-1}X(2) + \cdots \\ &+ D^{-1}(0)W^{-1}D^{-1}(1)\cdots D^{-1}(K-2) \\ &\cdot W^{-1}X(K-1)]. \end{aligned} \tag{34}$$

Let

$$U(K) = WD(K-2)\cdots D(1)WD(0). \tag{35}$$

It can be seen that $U(K)$ can be obtained recursively from $U(K-1)$. As of the term in the brackets [] in (34), it can be interpreted as a sum $S(K)$ of tensor products, that can be obtained recursively as

$$S(K) = S(K-1) + U^{-1}(K) \otimes x(K-1). \tag{36}$$

The algorithm is as follows (let $S(K, j)$ be an $N \times N$ matrix):

1) $K = 1$, $(dE/dw_{ij})(0) = 0$, $U(1) = I$, $S(1, j) = Ix_j(0)$;
2) $(dE/dw_{ij})(K) = (dE/dw_{ij})(K-1) + e(K)D(K-1)U(K)S_i(K, j)$, $S_i(K, j)$ is the ith column of S;
3) $K = K+1$,
   $U(K) = WD(K-2)U(K-1)$,
   $S(K, j) = S(K-1, j) + U^{-1}(K)x_j(K-1)$,
   Go to 2) till end of data.

E. The Block Update Approach

Assume the gradient is computed at time step $K-N$. We wait until data point K, and then update the gradient. Denote the gradient update $(dE/dW)(K) - (dE/dW)(K-N)$ as $G(K)$. The update $G(K)$, is obtained by solving (8), with (dE/dx) replaced by the vector

$$e' = (0\ 0\ \cdots\ 0\ e(K-N+1)\ e(K-N+2)\ e(K)) \tag{37}$$

because the effect of the errors $e(k)$ earlier than data point $K-N+1$ should not be taken into account (they will get subtracted out). Thus $G(K)$ is the result of solving the following equations similar to (22) but with e' used instead of e

$$0 = -\delta(k) + \delta(k+1)D(k)W, \quad k = 1, \cdots, K-N \tag{38}$$

$$e(k) = -\delta(k) + \delta(k+1)D(k)W, \quad k = K-N+1, \cdots, K-1 \tag{39}$$

$$e(K) = -\delta(K) \tag{40}$$

with $G(K)$ given similar to (25)

$$G(K) = -Z(1, K-N) - Z(K-N+1, K) \tag{41}$$

where

$$Z(k_1, k_2) = \sum_{k=k_1}^{k_2} D(k-1)\delta^T(k)x^T(k-1) \tag{42}$$

(note that we have partitioned the summation in (25) into two parts).

At every update the term $Z(K-N+1, K)$ is easily computed using (42) and the δ's obtained from (39) and (40). To obtain the term $Z(1, K-N)$, on the other hand, we have first to solve for $\delta(k)$, $k = 1, \cdots, K-N$. By solving the backward recursion (38), we get

$$\delta(k) = \delta(K-N+1)D(K-N)W\cdots D(k)W \quad k = 1, \cdots, K-N. \tag{43}$$

Substituting (43) into the expression for $Z(1, K-N)$, we get

$$\begin{aligned} Z(1, K-N) &= \sum_{k=1}^{K-N} D(k-1)W^TD(k)\cdots W^TD(K-N) \\ &\quad \cdot \delta^T(K-N+1)x^T(k-1). \end{aligned} \tag{44}$$

Define

$$Q_i(K-N) = \sum_{k=1}^{K-N} D(k-1)W^TD(k)\cdots W^TD(K-N)x_i(k-1). \tag{45}$$

Then

$$Z(1, K-N)|_{i\text{th column}} = Q_i(K-N)\delta^T(K-N+1). \tag{46}$$

The matrices $Q_i(K)$ can be obtained recursively in terms of $Q_i(K-N)$. This is because $Q_i(K)$ can be expressed as

$$\begin{aligned} Q_i(K) &= Q_i(K-N)W^T\Gamma(K-N+1, K) \\ &\quad + \sum_{k=K-N+1}^{K} \Gamma(k-1, K)x_i(k-1) \end{aligned} \tag{47}$$

where

$$\Gamma(k, K) = D(k)W^TD(k+1)\cdots W^TD(K). \tag{48}$$

The quantity $\Gamma(k, K)$ can be obtained recursively as

$$\Gamma(k, K) = D(k)W^T\Gamma(k+1, K). \tag{49}$$

Once we obtain $G(K)$ as described, the gradient $(dE/dW)(K)$ can be obtained as $(dE/dW)(K-N) + G(K)$.

V. The New Method

A. Derivation of the Algorithm

Rather than insisting on computing the exact gradient for the recurrent network training problem and paying the price in high computational complexity, the approach proposed here is to obtain an approximation for the gradient that can be efficiently computed. As we will see later, although the search direction proposed is different from the actual gradient direction, it will result in faster convergence to the minimum than the gradient-based approach.

The idea of the proposed algorithm is to interchange the roles of the network states $x(k)$ and the weight matrix W. The states are considered as the control variables, and the change in the weights is determined according to the changes in $x(k)$. In other words, we calculate the gradient of E with respect to the states $x(k)$, and assume a small change in the states $x(k)$ in the negative direction of that gradient $[\Delta x_i(k) = -\eta(\partial E/\partial x_i(k))]$. Next, we determine the change in the weights that will result in changes in $x(k)$ as close as possible to the targeted changes $\Delta x(k)$. The details of the algorithm are as follows:

Take

$$\Delta x = -\eta\left(\frac{\partial E}{\partial x}\right)^T. \tag{50}$$

Note that the transpose in (50) is because $\partial E/\partial x$ is by definition a row vector. From (9),

$$\Delta x = -\eta e^T = -\eta(e(1), \cdots, e(K))^T. \tag{51}$$

Before we proceed, we make a simple adjustment in the formulation presented in Section IV, since this will simplify the derived formulas. The constraints in (4) can be rewritten as

$$g'(k+1) \equiv Wx(k) - f^{-1}(x(k+1)) = 0, \quad k = 0, \cdots, K-1, \tag{52}$$

and let $g' = (g'^T(1), \cdots, g'^T(K))^T$. The derivatives of g' with respect to w and x will be modified as shown in (53) at the bottom of the page, and as

$$\frac{\partial g'}{\partial w} = \begin{pmatrix} X(0) \\ X(1) \\ \vdots \\ X(K-1) \end{pmatrix}. \tag{54}$$

Since g', given by (52), equals zero, we get

$$\frac{\partial g'}{\partial w}\Delta w + \frac{\partial g'}{\partial x}\Delta x = 0. \tag{55}$$

It is required to find the direction of weight change Δw that will result in (55) being approximately satisfied, i.e., such that

$$\frac{\partial g'}{\partial w}\Delta w \approx -\frac{\partial g'}{\partial x}\Delta x. \tag{56}$$

Since the number of weights is typically less than the dimension of x, an exact solution is usually not possible. We therefore opt for the minimum sum of square error solution for the overdetermined system (56). That leads to the following solution involving the pseudoinverse of $\partial g/\partial w$:

$$\Delta w = -\left[\left(\frac{\partial g'}{\partial w}\right)^T\left(\frac{\partial g'}{\partial w}\right)\right]^{-1}\left(\frac{\partial g'}{\partial w}\right)^T\frac{\partial g'}{\partial x}\Delta x. \tag{57}$$

Let

$$\gamma = \frac{\partial g'}{\partial x}e^T = \frac{-1}{\eta}\frac{\partial g'}{\partial x}\Delta x \tag{58}$$

and partition the vector γ into the K vectors

$$\gamma = \begin{pmatrix} \gamma(1) \\ \gamma(2) \\ \vdots \\ \gamma(K) \end{pmatrix} \tag{59}$$

where $\gamma(k)$ is the $N \times 1$ vector associated with time step k. Using (53) and (58), we can evaluate $\gamma(k)$. We get the following recursion:

$$\gamma(1) = -D^{-1}(0)e^T(1) \tag{60a}$$

$$\gamma(2) = -D^{-1}(1)e^T(2) + We^T(1) \tag{60b}$$

$$\vdots$$

$$\gamma(K) = -D^{-1}(K-1)e^T(K) + We^T(K-1). \tag{60c}$$

$$\frac{\partial g'}{\partial x} = \begin{pmatrix} -D^{-1}(0) & 0 & 0 & \cdots & 0 & 0 \\ W & -D^{-1}(1) & 0 & \cdots & 0 & 0 \\ 0 & W & -D^{-1}(2) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & W & -D^{-1}(K-1) \end{pmatrix}. \tag{53}$$

In addition, the matrix inverse $[(\partial g'/\partial w)^T(\partial g'/\partial w)]^{-1}$ can be evaluated using the definitions of $(\partial g'/\partial w)$ and $X(k)$ [(14) and (54)]. We get (61), shown at the bottom of the page. Let

$$V'(K) = \sum_{k=0}^{K-1} x(k)x^T(k). \tag{62}$$

Rather than representing the weights in the form of one long vector w, we return to representing them in the more convenient form of matrix W. Substituting (61), (62), (54), and (58) into (57), we get after some manipulation

$$\Delta W = \eta \left[\sum_{k=1}^{K} \gamma(k)x^T(k-1)\right] V'^{-1}(K). \tag{63}$$

The problem with the previous equation is that there is a matrix inversion, causing in most likelihood ill-conditioning problems. We have alleviated this problem by adding to the outer product matrix $V'(K)$ a small matrix ϵI, where ϵ is a small positive constant, to obtain

$$V(K) = \epsilon I + \sum_{k=0}^{K-1} x(k)x^T(k). \tag{64}$$

$$\Delta W = \eta \left[\sum_{k=1}^{K} \gamma(k)x^T(k-1)\right] V^{-1}(K). \tag{65}$$

This will make the new matrix strictly positive definite. Even a very small ϵ was effective in practically preventing ill-conditioning of the matrix. The algorithm is summarized as follows:

Off-Line Training Algorithm:

1) $\gamma(1) = -D^{-1}(0)e^T(1)$,
   $\gamma(k) = -D^{-1}(k-1)e^T(k) + We^T(k-1).k = 2, \cdots, K.$
2) $V(K) = \epsilon I + \sum_{k=0}^{K-1} x(k)x^T(k).$
3) $\Delta W = \eta[\sum_{k=1}^{K} \gamma(k)x^T(k-1)]V^{-1}(K).$

This algorithm represents the off-line version of the proposed method. Although it is a computationally efficient update, it is advantageous to have also an on-line version, whereby the update at data point K can be efficiently computed on the basis of the update at data point $K-1$. A method to implement that is discussed in the Section V-C.

B. Computational Complexity

The complexity of the algorithm is $3N^2K + 2NN_OK + 14N^3/3$ operations (multiplications and additions) for a whole cycle, where N_O is the number of output nodes. The first term $3N^2K$ represents the number of operations in computing the summation in the brackets [] in (65) ($2N^2$ operations) and in computing $V(K)$ (N^2 operations, by exploiting the matrix symmetry). The second term $2NN_OK$ represents the complexity in computing the γ's. The last term represents the multiplication of the term in the brackets [] in (65) with $V^{-1}(K)$, which amounts to a solution of a linear system, and can be computed with $14N^3/3$ operations (see [50]). Since typically $N \ll K$ (the number of examples is typically several hundreds or thousands, whereas the number of nodes is typically ten or 20), the complexity is approximately $3N^2K + 2NN_OK$ operations, or $3N^2 + 2NN_O$ operations per data point. Also, typically $N_O << N$, and in a majority of applications there is only one output node. In such a case we will have about $3N^2$ operations per data point. As a comparison, the complexity of the fastest of the existing off-line techniques is about $4N^2$ operations per data point. Thus the new techniques is about 25% faster in terms of update direction calculation in case of a small number of outputs, and is always faster if $N_O < N/2$, which is usually true in most applications.

C. On-Line Update

Assume that the update based on the data $(x(1), d(1)), \cdots, (x(K-1), d(K-1))$ is already available

$$\left[\left(\frac{\partial g}{\partial w}\right)^T\left(\frac{\partial g}{\partial w}\right)\right]^{-1} = \begin{pmatrix} \sum_{k=0}^{K-1} x(k)x^T(k) & 0 & \cdots & 0 \\ 0 & \sum_{k=0}^{K-1} x(k)x^T(k) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sum_{k=0}^{K-1} x(k)x^T(k) \end{pmatrix}^{-1}$$

$$= \begin{pmatrix} \left[\sum_{k=0}^{K-1} x(k)x^T(k)\right]^{-1} & 0 & \cdots & 0 \\ 0 & \left[\sum_{k=0}^{K-1} x(k)x^T(k)\right]^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \left[\sum_{k=0}^{K-1} x(k)x^T(k)\right]^{-1} \end{pmatrix}. \tag{61}$$

[say $= \Delta W(K-1)$]. Now, a new data point $(x(K), d(K))$ arrives. The update $\Delta W(K)$ becomes

$$\Delta W(K) = \eta \left[\sum_{k=1}^{K-1} \gamma(k) x^T(k-1) + \gamma(K) x^T(K-1) \right] \cdot [V(K-1) + x(K-1)x^T(K-1)]^{-1}. \quad (66)$$

Using the small rank adjustment matrix inversion lemma (see [51]), we can obtain the inverse of $V(K)$ recursively in terms of the inverse of $V(K-1)$, as follows:

$$\begin{aligned} V^{-1}(K) &= (V(K-1) + x(K-1)x^T(K-1))^{-1} \\ &= V^{-1}(K-1) \\ &\quad - \frac{[V^{-1}(K-1)x(K-1)][V^{-1}(K-1)x(K-1)]^T}{1 + x^T(K-1)V^{-1}(K-1)x(K-1)}. \end{aligned} \quad (67)$$

Let

$$B(K) = \sum_{k=1}^{K} \gamma(k) x^T(k-1). \quad (68)$$

Substituting (67) into (66), and after some simplification, we get the final on-line update formula shown in (69) at the bottom of the page.

The algorithm can be summarized as follows.

On-Line Training Algorithm::

1) $K = 1$, $\gamma(1) = -D^{-1}(0)e^T(1)$, $B(1) = \gamma(1)x^T(0)$, $V^{-1}(1) = I/\epsilon - x(0)x^T(0)/[\epsilon^2 + \epsilon x^T(0)x(0)]$, $\Delta W = \eta B(1)V^{-1}(1)$.
2) $K = K + 1$,
   $x(K) = f[Wx(K-1)]$,
   $e_i(K) = x_i(k) - d_i(k)$, if $i \in O$, $e_i(K) = 0$, otherwise.
   $\gamma(K) = -D^{-1}(K-1)e^T(K) + We^T(K-1)$,
   Update $\Delta W(K)$ according to (69),
   $B(K) = B(K-1) + \gamma(K)x^T(K-1)$
   $V^{-1}(K) = V^{-1}(K-1) - ([V^{-1}(K-1)x(K-1)][V^{-1}(K-1)x(K-1)]^T/(1 + x^T(K-1)V^{-1}(K-1)x(K-1))$.
   Go to 2) until end of data.

We note here that the on-line update can also be advantageous in problems, which need only off-line training. The reason is that the on-line algorithm permits implementing the so-called sequential update (also called pattern update). By sequential update we mean that the weights are updated after the presentation of each example. For the other type of update, the batch update, the weights are updated only after cycling through all examples. It is well known that for the standard backpropagation algorithm the sequential update is generally faster than the batch update, so the on-line algorithm presented in this section could provide an advantage in this respect.

D. Complexity of the On-Line Update

It can be shown that the complexity of one iteration is on the order of $11N^2 + 2N_O N$ operations per data point (including multiplications and additions). This can be easily shown by counting the number of matrix–vector or vector–vector multiplications, in the algorithm discussed previous section. One can see that the complexity for the on-line update is higher than that of the off-line update (about 11 to three). This apparently is the price one has to pay for the flexibility of being able to update the weights in an on-line fashion.

E. Treating Input Nodes Separately

For applications such as time-series prediction or system identification there are external inputs to the recurrent network. As mentioned in Section III, we assumed for such a case that we have input nodes that are clamped at all times at the input values. To obtain more direct formulas in this case, let us here consider the input nodes as a separate entity.

Let the model be

$$x(k+1) = f[W_1 x(k) + W_2 u(k)] \quad (70)$$

where $u(k)$ is the input vector at time k, W_1, and W_2 are weight matrices. Define the augmented matrices

$$W' = [W_1 | W_2] \quad (71)$$

$$x'(k) = \begin{pmatrix} x(k) \\ u(k) \end{pmatrix}. \quad (72)$$

The algorithm will be the same, except that (60a)–(60c) for the update of the γ's will be modified to

$$\gamma(1) = -D^{-1}(0)e^T(1) \quad (73a)$$

$$\gamma(2) = -D^{-1}(1)e^T(2) + W_1 e^T(1) \quad (73b)$$

$$\vdots$$

$$\gamma(K) = -D^{-1}(K-1)e^T(K) + W_1 e^T(K-1) \quad (73c)$$

and any $x(k)$ in (61)–(69) will be replaced by $x'(k)$. Also, ΔW in (65) and (69) will be replaced by $\Delta W'$.

F. Extension to Continuous-Time Networks

The new method can be generalized to the case of continuous-time networks. Consider the following model:

$$\frac{du}{dt} = -u + Wf(u) + \theta \quad (74)$$

$$y = f(u) \quad (75)$$

$$\Delta W(K) = \Delta W(K-1) + \eta \frac{\gamma(K)x^T(K-1)V^{-1}(K-1) - B(K-1)V^{-1}(K-1)x(K-1)[V^{-1}(K-1)x(K-1)]^T}{1 + x^T(K-1)V^{-1}(K-1)x(K-1)}. \quad (69)$$

where

u represents an intermediate state vector;
y is the output vector;
θ is the vector of biases.

By discretizing the equations, we obtain

$$u(t_0+(i+1)\Delta t) = u(t_0+i\Delta t)(1-\Delta t)+\Delta t W f(u(t_0+i\Delta t))+\theta \quad i=0,\cdots,K-1 \tag{76}$$

where $t = t_0 + i\Delta t$. Expressing the error as

$$E=\sum_{i=1}^{K}\sum_{j\in O}[y_j(t_0+i\Delta t)-d_j(t_0+i\Delta t)]^2 \tag{77}$$

we obtain the constraint equations "$g = 0$" [similar to (4)], and the discretized objective function. The analysis follows similar steps as described for the discrete-time case, though we will not describe such detail in this paper.

VI. Simulation Results

Both versions of the new algorithm have been tested on a number of problems. To obtain a comparative idea about the speed of the algorithm, we have also implemented on these same problems the following two methods: 1) the BTTapproach, which is the fastest of the existing gradient-descent-based algorithms and 2) the BTT(h, h') approach [21], which is a very efficient accelerated technique for recurrent networks. The BTT(h, h') is summarized as follows.

1) Run the network for h steps.
2) Propagate backwards for h' steps ($h' > h$), and update the weights.
3) Run the network for the next h steps, then propagate backwards h' steps, and update the weights. Continue in a similar manner till the end of the data, and then repeat another cycle.

The BTT(h, h') method basically approximates the gradient, but with less computations. An attractive feature about this method is the ability to perform sequential (or on-line) update, which typically is superior to batch update. Typically, h' is chosen as twice the value of h. In the comparison simulations, we have implemented two versions: BTT(5,10) and BTT(10,20). These fall in the typical range of h and h' used by the inventors of the method.

We wish to note that when comparing the speed of two algorithms, two different components of the speed have to be taken into account: the speed of computing the update direction, and the speed of converging to the minimum error solution. We have seen in previous sections that for the off-line version and the on-line version of the developed algorithm the update direction is computed using $3N^2$ and $11N^2$ operations, respectively (assuming small number of output nodes). The BTT approach, on the other hand, needed $4N^2$ operations. As for the BTT(h, h') method, the update direction is computed using about $4h'N^2/h$ operations. Since typically $h' = 2h$, the number of operations is about $8N^2$. As for the number of iterations till convergence, this has to be tested through numerical simulations, by comparing the number of iterations till reaching an acceptable error. For all of the gradient-descent-based methods (BTT, FP, FFP, GF, and BU), they will have identical number of iterations until convergence, because they update the weights in exactly the same direction (the negative direction of the gradient). They differ in only the algorithm to obtain the update direction. We have therefore decided to make the comparison study with only one of the gradient-descent-based methods, the BTT method.

In the test comparisons we used the abbreviation OFFL and ONL to denote respectively the off-line and the on-line version of the proposed algorithm. We have implemented the on-line algorithm in a sequential update mode, whereas the off-line algorithm is implemented as a batch update. We used as our error function the normalized mean of square error, defined as

$$\text{NMSE}=\frac{\sum_{k=1}^{K}\sum_{i\in O}(x_i(k)-d_i(k))^2}{\sum_{k=1}^{K}\sum_{i\in O}d_i^2(k)}. \tag{78}$$

In the first trial we tune the parameter values η and ϵ for each of the methods (all runs start from the same initial weights). The way we have done the comparison is to perform the following for each method. Several runs each with different parameter values (learning rate, etc.) are performed. We then choose the parameter values that lead to fastest convergence. We then fix the parameters on these values, and run ten more trials each with different initial weights. For a particular trial we fix the initial weight configuration across the five methods to make the comparison fair. We record the number of iterations needed to reach particular error levels, and obtain the average for each method. We note that for recurrent networks it is always better to start with small weights, because if we have long sequences for large initial weights the states tend to wander off into the saturation region. We have generated the initial weights always in the range from -0.2 to 0.2. We have trained all methods for a maximum of 10 000 iterations. If the method did not reached the prespecified error levels by then, then we declare that it failed to converge on this particular trial.

The following is a description of the comparison results.

Example 1—A Second-Order Dynamical System: We considered the dynamical system.

$$y(k+1)=0.4y(k)+0.4y(k)y(k-1)+0.6u^3(k)+0.1. \tag{79}$$

The problem is to design a network that identifies the given system. The recurrent network receives one input $u(k)$, and attempts to emulate the dynamics of the given system and to produce an output as close as possible to $y(k+1)$. We generated a sequence of 200 points by considering $u(k)$ to be independent uniform noise (uniform in [0, 0.5]), and generating $y(k)$ according to (79). The output is scaled by multiplying by two, to get it into a range suitable for the network. We have used a fully connected recurrent network with ten hidden nodes. In the tuning trial we observed that the minimum is approximately at $\text{NMSE} = 0.002$. We have set up two targeted levels: $\text{NMSE} = 0.005$, and $\text{NMSE} = 0.003$, and recorded the average number of iterations to reach these two levels. Table I shows the results of this simulation.

TABLE I
THE AVERAGE AND THE STANDARD DEVIATION OF THE NUMBER OF ITERATIONS UNTIL REACHING ERROR LEVELS 0.005 AND 0.003 FOR EXAMPLE 1 FOR THE PROPOSED METHODS OFFL AND ONL, AND FOR THE COMPETING METHODS BTT, BTT(5, 10), AND BTT(10, 20). ALSO SHOWN ARE THE NUMBER OF CONVERGED TRIALS (AMONG 10 TRIALS), AND THE VALUES OF THE PARAMETERS η AND ϵ

	Error Level=0.005			Error Level=0.003				
Method	Avg No. Iter.	StD No. Iter	No. Converged	Avg No. Iter.	StD No. Iter	No. Converged	eta	epsilon
BTT	357.6	146.7	10	1119.3	318.2	10	0.005	
OFFL	4.6	0.7	10	7.2	3.3	10	0.5	0.02
ONL	7.9	0.9	9	12	6.1	9	0.5	0.02
BTT(5,10)	36.5	11.7	10	108.6	60.6	10	0.1	
BTT(10,20)	27.8	6.8	10	54	8.9	10	0.03	

TABLE II
THE AVERAGE AND THE STANDARD DEVIATION OF THE NUMBER OF ITERATIONS UNTIL REACHING ERROR LEVELS 0.005 AND 0.003 FOR EXAMPLE 2 FOR THE PROPOSED METHODS OFFL AND ONL, AND FOR THE COMPETING METHODS BTT, BTT(5, 10), AND BTT(10, 20). ALSO SHOWN ARE THE NUMBER OF CONVERGED TRIALS (AMONG 10 TRIALS), AND THE VALUES OF THE PARAMETERS η AND ϵ

	Error Level=0.005			Error Level=0.003				
Method	Avg No. Iter.	StD No. Iter	No. Converged	Avg No. Iter.	StD No. Iter	No. Converged	eta	epsilon
BTT	410.9	203.1	10	815.9	307.1	10	0.004	
OFFL	2.5	0.7	10	2.5	0.7	10	1	0.02
ONL	5.1	0.7	10	6.2	0.8	10	1	0.02
BTT(5,10)	33.8	10.6	10	102.6	60.9	10	0.02	
BTT(10,20)	22.5	5.6	10	63.1	35.1	10	0.03	

TABLE III
THE AVERAGE AND THE STANDARD DEVIATION OF THE NUMBER OF ITERATIONS UNTIL REACHING ERROR LEVELS 0.01 AND 0.006 FOR EXAMPLE 3 FOR THE PROPOSED METHODS OFFL AND ONL, AND FOR THE COMPETING METHODS BTT, BTT(5, 10), AND BBT(10, 20). ALSO SHOWN ARE THE NUMBER OF CONVERGED TRIALS (AMONG 10 TRIALS), AND THE VALUES OF THE PARAMETERS η AND ϵ

	Error Level=0.01			Error Level=0.006				
Method	Avg No. Iter.	StD No. Iter	No. Converged	Avg No. Iter.	StD No. Iter	No. Converged	eta	epsilon
BTT	201	65.8	10	927.6	195.4	10	0.002	
OFFL	4.6	0.52	10	11.9	4.1	9	1	10
ONL	2	0	10	3.1	0.32	10	4	10
BTT(5,10)	3.5	0.53	10	11.8	1.03	10	0.1	
BTT(10,20)	3.7	2.2	10	11.4	2.6	10	0.1	

Example 2—A Complex Second-Order Problem: We consider the second-order problem given by

$$y(k+1) = \frac{y(k)y(k-1)(y(k)+0.25)}{1+y^2(k)+y^2(k-1)} + u(k) \tag{80}$$

(see [7], but parameters are somewhat different). We generated 200 points from a uniform density in [0, 0.5], and drove the system using this uniform noise, to generate the input–output examples. We have used a network with ten hidden nodes. The minimum was found in the tuning stage to be approximately 0.002. So we took 0.005 and 0.003 as our target error levels. Table II shows the comparison results.

Example 3—Time Series Competition Data: We consider data set D from the Santa Fe Institute Time Series Competition [52]. Data set D represents a chaotic oscillator. It is a time-series, and the task here is to predict one-step-ahead. We considered the first 1000 data points as our training set. We have also used a network with ten hidden nodes. In the tuning trial we found that the minimum is approximately at $\mathrm{NMSE} = 0.005$. We have therefore measured the speed till reaching the two levels: $\mathrm{NMSE} = 0.01$ and $\mathrm{NMSE} = 0.006$. Table III shows the average number of iterations to reach these levels.

Example 4—A Two-Input/Two-ouput Nonlinear System: We consider a two-input/two-output nonlinear system described by the following nonlinear state equations:

$$x_1(k+1) = 0.5x_1^{2/3}(k) + 0.3x_2(k)x_3(k) + 0.2u_1(t) \tag{81}$$

$$x_2(k+1) = 0.5x_2^{2/3}(k) + 0.3x_3(k)x_1(k) + 0.5u_1(t) \tag{82}$$

$$x_3(k+1) = 0.5x_3^{2/3}(k) + 0.3x_1(k)x_2(k) + 0.5u_2(t) \tag{83}$$

$$y_1(k+1) = 0.7(x_1(k+1) + x_2(k+1)) \tag{84}$$

$$y_2(k+1) = 1.5x_1^2(k+1) \tag{85}$$

where

$x_1(k), x_2(k), x_3(k)$ are the three states of the system;
$u_1(k), u_2(k)$ are the two inputs;
$y_1(k), y_2(k)$ are the two outputs.

The problem is to identify the given system. We generated the inputs as independent identically distributed (iid) from a uniform density in [0, 0.5]. We considered 500 data points as our training examples. Being a multiinput–multioutput system, and

TABLE IV
THE AVERAGE AND THE STANDARD DEVIATION OF THE NUMBER OF ITERATIONS UNTIL REACHING ERROR LEVELS 0.001 AND 0.0006 FOR EXAMPLE 4 FOR THE PROPOSED METHODS OFFL AND ONL, AND FOR THE COMPETING METHODS BTT, BTT(5, 10), AND BTT(10, 20). ALSO SHOWN ARE THE NUMBER OF CONVERGED TRIALS (AMONG 10 TRIALS), AND THE VALUES OF THE PARAMETERS η AND ϵ

	Error Level=0.001			Error Level=0.0006				
Method	Avg No. Iter.	StD No. Iter	No. Converged	Avg No. Iter.	StD No. Iter	No. Converged	eta	epsilon
BTT	3429	418.6	10	9763		1	0.001	
OFFL	7.4	1	10	8.9	1.1	9	0.4	0.005
ONL	12.3	0.7	10	16.8	1.8	8	0.4	0.005
BTT(5,10)	130.5	37.6	10	206.6	159.3	10	0.04	
BTT(10,20)	826.6	499.9	10	1122.7	736.6	10	0.04	

TABLE V
THE AVERAGE AND THE STANDARD DEVIATION OF THE NUMBER OF ITERATIONS UNTIL REACHING ERROR LEVELS 0.03 AND 0.02 FOR EXAMPLE 5 FOR THE PROPOSED METHODS OFFL AND ONL, AND FOR THE COMPETING METHODS BTT, BTT(5, 10), AND, BTT(10, 20). ALSO SHOWN ARE THE NUMBER OF CONVERGED TRIALS (AMONG 10 TRIALS), AND THE VALUES OF THE PARAMETERS η AND ϵ

	Error Level=0.03			Error Level=0.02				
Method	Avg No. Iter.	StD No. Iter	No. Converged	Avg No. Iter.	StD No. Iter	No. Converged	eta	epsilon
BTT	767.8	328.6	4	910.8	302.8	4	0.004	
OFFL	9.4	2.8	10	20	18.7	10	0.2	0.002
ONL	13.8	1.6	10	27.2	28.3	10	0.2	0.002
BTT(5,10)	126.9	38.9	10	211.6	90.9	10	0.02	
BTT(10,20)	119	35.2	10	165.5	69.3	10	0.02	

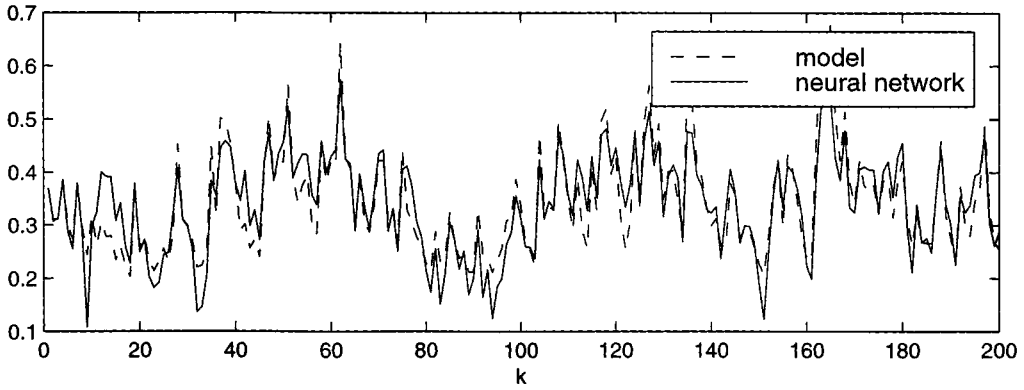

Fig. 1. The network output versus the desired output for the proposed off-line algorithm for Example 5.

being expressed in a state-space formulation, this problem is more difficult than the preceding problems. We have considered a network with 20 hidden nodes. The minimum error was found in a few exploratory runs to be approximately 0.0005. So we set 0.001 and 0.0006 as our target error levels. Table IV shows the comparison results.

Example 5—A Tenth-Order Problem: We consider the following tenth-order system:

$$y(k+1) = 0.3y(k) + 0.05y(k)\left[\sum_{i=0}^{9} y(k-i)\right] + 1.5u(k-9)u(k) + 0.1. \quad (86)$$

It is well known that for problems with long-term time dependencies recurrent networks are very hard to train [27]. This problem has a time-lag of ten time steps, and hence it is a fairly difficult problem compared to Examples 1–3. We used 200 data points as a training set. Again, the inputs are generated from a uniform density in [0, 0.5]. We have considered a network with 40 hidden nodes. The minimum error was found in a few exploratory runs to be approximately 0.015. So we set 0.03 and 0.02 as our target error levels. Table V shows the comparison results. Also, Fig. 1 shows a plot of the network output as compared to the desired output for the proposed off-line algorithm at the achieved minimum.

Comments on the Results: One can see from the tables that both versions of the developed algorithm significantly outperform the BTT method and the BTT(h, h') versions. The acceleration as achieved over BTT is typically several hundreds of times. Adding to that the fact that the off-line version of the developed algorithm is about 25% faster than the BTT and about 8/3 times faster than BTT(h, h') in calculating the update direction, we realize that some extra speed-up is gained. As for the on-line technique, the results in terms of convergence speed are about the same as those of the off-line version. One powerful aspect of the developed method is being able to use large learning rates, making it possible to take big steps to reach the minimum faster. But this also causes the side effect of having an overshoot in the error a few iterations after reaching the minimum. This, however, does not cause any problems, since one can simply stop further iterations when error increases beyond a

certain percentage. We wish to note that we have implemented the new algorithm to many other problems, and the results are very consistent in terms of speed-up over the methods compared. The five examples presented are a representative of the sample problems considered. It would be interesting to compare the algorithms for other recurrent-type architectures [37]–[46], since these architectures are often more powerful than fully connected recurrent networks, but this will be performed in a future study. Another aspect that could be performed in future work is to modify the algorithm so that it can handle very large time lags, say of the order of several hundreds or thousands of time steps. So far, there is only one method [27] that can handle such large time lags. So far the approximation quality of the proposed algorithm has been shown through simulations. One possible future study is to derive that analytically, in order to gain more understanding of the method.

VII. Conclusions

In this paper we have developed a novel formulation of recurrent network training algorithms. This formulation allows the unification of existing algorithms. In addition it gives insights into the problem, and can potentially help in deriving new algorithms. Based on this formulation we have derived a new algorithm for training recurrent networks. The off-line version of the new algorithm has complexity of $3N^2$ operations per iteration for the case of small number of output nodes, about 25% faster than that of the backpropagation through time algorithm, which is the fastest of the existing gradient-based algorithms. In addition, the number of iterations needed to converge to the minimum is significantly less than that of the backpropagation through time and the BTT(h, h') acceleration technique. In addition, an on-line version of the proposed algorithm has been developed, that also exhibits considerable speed approaching that of the off-line version.

References

[1] A. Parlos, K. Chong, and A. Atiya, "Application of the recurrent multilayer perceptron in modeling complex process dynamics," *IEEE Trans. Neural Networks*, vol. 5, pp. 255–266, Mar. 1994.

[2] J. Moody and L. Wu, "Optimization of trading systems and portfolios," in *Proc. Neural Networks Capital Markets Conf.* Pasadena, CA, Nov. 1996.

[3] J. Connors, D. Martin, and L. Atlas, "Recurrent neural networks and robust time series prediction," *IEEE Trans. Neural Networks*, vol. 5, pp. 240–254, Mar. 1994.

[4] D. Bassi, "Stock price predictions by recurrent multilayer neural network architectures," in *Proc. Neural Networks in the Capital Markets Conf.*, A. Refenes, Ed.. London, U.K., October 1995, London Business School, pp. 331–340.

[5] C. Omlin and L. Giles, "Exatraction of rules from discrete-time recurrent neural networks," *Neural Networks*, vol. 9, no. 1, pp. 41–52, 1996.

[6] G. Puskorius and L. Feldman, "Neurocontrol of dynamical systems with Kalman filter trained recurrent networks," *IEEE Trans. Neural Networks*, vol. 5, pp. 279–297, Mar. 1994.

[7] K. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Trans. Neural Networks*, vol. 1, pp. 4–27, 1990.

[8] J. Suykens, B. De Moor, and J. Vandewalle, "Nonlinear system identification using neural state-space models, applicable to robust control design," *Int. J. Contr.*, vol. 62, no. 1, pp. 129–152, 1995.

[9] S. Haykin, *Neural Networks: A Comprehensive Foundation*: IEEE Press, 1994.

[10] R. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput.*, vol. 1, pp. 270–280, 1989.

[11] A. Robinson and F. Fallside, "The utility driven dynamic error propagation network," Cambridge Univ. Eng. Dept., Tech. Rep. CUED/F-INFENG/TR.1, 1987.

[12] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representation by error propagation," *Parallel Distibuted Processing*, 1986.

[13] P. Werbos, "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, Oct. 1990.

[14] B. Pearlmutter, "Learning state-space trajectories in recurrent neural networks," *Neural Comput.*, vol. 1, pp. 263–269, 1989.

[15] N. Toomarian and J. Barhen, "Adjoint-functions and temporal learning algorithms in neural networks," *Advances in Neural Information Processing Systems, 3*, pp. 113–120, 1991.

[16] ——, "Learning a trajectory using adjoint functions and teacher forcing," *Neural Networks*, vol. 5, no. 3, pp. 473–484, 1992.

[17] G.-Z. Sun, H.-H. Chen, and Y.-C. Lee, "Green's function method for fast on-line learning algorithm of recurrent neural networks," *Advances Neural Inform. Processing Syst. 4*, pp. 333–340, 1992.

[18] J. Schmidhuber, "A fixed storage $O(N^3)$ time complexity learning algorithm for fully recurrent continually running networks," *Neural Comput.*, vol. 4, no. 2, pp. 243–248, 1992.

[19] R. Williams, "Complexity of exact gradient computation algorithms for recurrent neural networks," Northeastern Univ., College Comp. Sci., Boston, MA, Tech. Rep. NU-CCS-89-27, 1989.

[20] R. Williams and J. Peng, "Gradient-based learning algorithms for recurrent networks and their computational complexity," in *Backpropagation: Theory, Architectures, and Applications*. Hillsdale, NJ: Lawrence Erlbaum, 1992.

[21] ——, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Comput.*, vol. 2, pp. 490–501, 1990.

[22] J. Sum, L. Chan, C. Leung, and G. Young, "Extended Kalman filter-based pruning method for recurrent neural networks," *Neural Comput.*, vol. 10, no. 6, pp. 1481–1506, 1998.

[23] S. Ma and C. Ji, "Fast training of recurrent networks based on EM-algorithm," *IEEE Trans. Neural Networks*, vol. 9, pp. 11–26, Jan. 1998.

[24] S. Ma and C. Ji, "A unified approach on fast training of feedforward and recurrent networks using EM algorithm," *Trans. Signal Processing*, vol. 46, pp. 2270–2274, Aug. 1998.

[25] M. Mozer, "A focused back-propagation algorithm for temporal sequence recognition," *Complex Syst.*, vol. 3, pp. 349–381, 1989.

[26] J. Hochreiter, "Untersuchungen zu dynamischen neuronalen Netzen," Diploma Thesis, Institut für Informatik, Technische Universität München, 1991.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[28] F. Pineda, "Dynamics and architecture for neural computation," *J. Complexity*, vol. 4, pp. 216–245, 1988.

[29] L. Almeida, "A learning rule for asynchronous perceptrons with feedback in a combinatotrial environment," in *Proc. IEEE 1st Int. Conf. Neural Networks*, vol. 2, San Diego, CA, 1987, pp. 609–618.

[30] A. Atiya, "Learning on a general network," in *Neural Inform. Processing Syst.*, D. Anderson, Ed. New York: Amer. Inst. Physics, 1988.

[31] A. Atiya and A. Parlos, "Unifying recurrent network training algorithms," in *Proc. World Congr. Neural Networks*, Portland, OR, Jul. 1993.

[32] P. Baldi, "Gradient descent learning algorithms: Ageneral overview," *IEEE Trans. Neural Networks*, vol. 6, pp. 182–195, Jan. 1995.

[33] B. Pearlmutter, "Gradient calculation for dynamic recurrent neural networks—A survey," *IEEE Trans. Neural Networks*, vol. 6, pp. 1212–1228, Sept. 1995.

[34] F. Pineda, "Time dependent adaptive neural networks," in *Advances in Neural Information Processing Systems, 2*, D. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990, pp. 710–718.

[35] F. Beaufays and E. Wan, "Relating real-time backpropagation and backpropagation-through-time: An application of flow graph interreciprocity," *Neural Comput.*, vol. 6, pp. 296–306, 1994.

[36] E. Wan and F. Beaufays, "Diagrammatic derivation of gradient algorithms for neural networks," *Neural Comput.*, vol. 8, pp. 182–201, 1996.

[37] A. Parlos, A. Atiya, K. Chong, W. Tsai, and B. Fernandez, "Recurrent multilayer perceptron for nonlinear system identification," in *Proc. Int. Joint Neural Network Conf.*, Seattle, WA, Jul. 1991.

[38] A. Tsoi and A. Back, "Locally recurrent globally feedforward networks—A critical review of architectures," *IEEE Trans. Neural Networks*, vol. 5, pp. 229–239, Mar. 1994.

[39] A. Atiya and A. Parlos, "Identification of nonlinear dynamics using a general spatio-temporal network," *Math. Comput. Modeling J.*, vol. 21, no. 1, pp. 53–71, Jan. 1995.
[40] ——, "Nonlinear system identification using spatio-temporal neural networks," in *Proc. Int. Joint Neural Network Conf.*, Baltimore, MD, June 1992.
[41] E. Wan, "Temporal backprogation for FIR neural networks," in *Proc. Int. Joint Conf. Neural Networks*, San Diego, CA, 1990, pp. 575–580.
[42] P. Baldi and A. Atiya, "How delays affect neural dynamics and learning," *IEEE Trans. Neural Networks*, vol. 5, pp. 612–621, Jul. 1994.
[43] J. Suykens, J. Vandewalle, and B. De Moor, "NL_q theory: Checking and imposing stability of recurrent neural networks for nonlinear modeling," *IEEE Trans. Signal Processing*, vol. 45, pp. 2682–2691, Nov. 1997.
[44] P. Sastry, G. Santharam, and K. Unnikrishnan, "Memory neuron networks for identification and control of dynamical systems," *IEEE Trans. Neural Networks*, vol. 5, pp. 305–319, Mar. 1994.
[45] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Trans. Neural Networks*, vol. 8, pp. 714–735, Mar. 1997.
[46] C. Goller, "A connectionist control component for the theorem prover SETHEO," in *Proc. ECAI'94 Workshop W14: Combining Symbolic and Connectionist Processing*, 1994, pp. 88–93.
[47] C. Nguyen and C. Widrow, "The truck backer-upper: An example of self learning in neural networks," in *IEEE/INNS Int. Joint Conf. Neural Networks*, vol. 1, Washington, D.C., 1989, pp. 357–364.
[48] A. Atiya, S. El-Shoura, S. Shaheen, and M. El-Sherif, "A comparison between neural-network forecasting techniques—Case study: River flow forecasting," *IEEE Trans. Neural Networks*, vol. 10, pp. 402–409, Mar. 1999.
[49] A. Bryson and Y.-C. Ho, *Applied Optimal Control*. Washington, D.C.: Hemisphere, 1975.
[50] G. Golub and C. Van Loan, *Matrix Computations*. Baltimore, MD: Johns Hopkins Univ. Press, 1989.
[51] A. Horn and C. Johnson, *Matrix Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1985.
[52] A. Weigend and N. Gerschenfeld, Eds., *Time Series Prediction: Forecasting the Future and Understanding the Past*. Reading, MA: Addison-Wesley, 1994.
[53] L. Chan, "Training recurrent network with block-diagonal approximated Levenberg–Marquardt algorithm," in *Proc. IJCNN'99*, Washington, D.C., July 1999.

Amir F. Atiya (S'86–M'90–SM'97) was born in Cairo, Egypt, in 1960. He received the B.S. degree in 1982 from Cairo University, and the M.S. and Ph.D. degrees in 1986 and 1991, respectively, from the California Institute of Technology, (Caltech), Pasadena, CA, all in electrical engineering.

From 1985 to 1990, he was a Teaching and Reseach Assistant at Caltech. From September 1990 to July 1991, he held a Research Associate position at Texas A&M University. From July 1991 to February 1993, he was a Senior Research Scientist at QANTXX, in Houston, TX, a financial modeling firm. In March 1993, he joined the Computer Engineering Department at Cairo University as an Assistant Professor. He spent the summer of 1993 with Tradelink Inc., in Chicago, IL, and the summers of 1994, 1995, and 1996 with Caltech. In March 1997, he joined Caltech as a Visiting Associate in the Department of Electrical Engineering. In January 1998, he joined Simplex Risk Management Inc., Hong Kong as a Research Scientist, and he is currently holding this position jointly with his position at Caltech. His research interests are in the areas of neural networks, learning theory, theory of forecasting, pattern recognition, data compression, and optimization theory. His most recent interests are the application of learning theory and computational methods to finance.

Dr. Atiya received the Egyptian State Prize for Best Research in Science and Engineering, in 1994. He also received the Young Investigator Award from the International Neural Network Society, in 1996. Currently, he is an Associate Editor for IEEE TRANSACTIONS ON NEURAL NETWORKS. He served on the organizing and program committees of several conferences, most recent of which is Computational Finance CP'99, New York, and IDEAL'98, Hong Kong.

Alexander G. Parlos (S'81–M'86–SM'92) received the B.S. degree in nuclear engineering from Texas A&M University, College Station, in 1983, and the S.M. degree in nuclear engineering and the S.M. degree in mechanical engineering, both from the Massachusetts Institute of Technology, Cambridge, in 1985, 1985, and 1986, respectively.

He has been on the faculty at Texas A&M University since 1987, where he is currently an Associate Professor of Mechanical Engineering, with joint appointments in the Department of Nuclear Engineering and Department of Electrical Engineering. His current research interests are in the development and application of recurrent neural networks algorithms for nonlinear identification, prediction, filtering, and control, with special emphasis to system condition assessments (of diagnosis), end-of-life prediction (or prognosis), and reconfigurable control. He has been involved with research and teaching in neural networks, multivariate control, and system identification since 1988, and he has conducted extensive funded research in these areas. His research has resulted in one U.S. patent, one pending U.S. patent, and 16 invention disclosures. He has co-founded and was involved in the management of a high-tech start-up company commercializing software developed at Texas A&M. He has more than 110 publications in journals and conferences.

Dr. Parlos has served as an Associate Editor of the IEEE TRANSACTIONS ON NEURAL NETWORKS since 1994. He has served as a technical reviewer to numerous professional journal and government organizations, and he has participated in technical, organizing, and program committees of various conferences. Dr. Parlos is a Senior Member of AIAA, a member of ASME, ANS, INNS, and a registered professional Engineer in the state of Texas.