Backpropagation Algorithms for a Broad Class of Dynamic Networks

Orlando De Jesús and Martin T. Hagan

***Abstract*—This paper introduces a general framework for describing dynamic neural networks—the layered digital dynamic network (LDDN). This framework allows the development of two general algorithms for computing the gradients and Jacobians for these dynamic networks: backpropagation-through-time (BPTT) and real-time recurrent learning (RTRL). The structure of the LDDN framework enables an efficient implementation of both algorithms for arbitrary dynamic networks. This paper demonstrates that the BPTT algorithm is more efficient for gradient calculations, but the RTRL algorithm is more efficient for Jacobian calculations.**

***Index Terms*—Backpropagation through time (BPTT), dynamic neural networks, gradient, Jacobian, layered digital dynamic network (LDDN), real-time recurrent learning (RTRL), recurrent neural networks.**

I. Introduction

NEURAL networks can be classified into static and dynamic categories [12]. Static networks have no feedback elements and contain no delays; the output is calculated directly from the input through feedforward connections. In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs, or states of the network. These dynamic networks may be recurrent networks with feedback connections or feedforward networks with imbedded tapped delay lines (or a hybrid of the two). For static networks, the standard backpropagation algorithm [29], [37] can be used to compute the gradient of the error function with respect to the network weights, which is needed for gradient-based training algorithms. For dynamic networks, a more complex dynamic gradient calculation must be performed.

There are two general approaches (with many variations) to gradient and Jacobian calculations in dynamic networks: backpropagation-through-time (BPTT) [30] and real-time recurrent learning (RTRL) [32]. (The RTRL algorithm is also referred to as forward propagation or forward perturbation.) In the BPTT algorithm, the network response is computed for all time points, and then the gradient is computed by starting at the last time point and working backwards in time. This algorithm is computationally efficient for the gradient calculation, but it is difficult to implement online, because the algorithm works backward in time from the last time step. In the RTRL algorithm, the gradient can be computed at the same time as the network response, since it is computed by starting at the first time point, and then working forward through time. RTRL requires more calculations than BPTT for calculating the gradient, but RTRL allows a convenient framework for online implementation [19]. For Jacobian calculations, as will be explained in Section V, the RTRL algorithm is generally more efficient than the BPTT algorithm (although this will depend somewhat on the network architecture). This efficiency is due to the fact that the Jacobian calculation is a part of the gradient calculation in the RTRL algorithm.

Although the RTRL and BPTT algorithms form the two basic approaches for computing the gradients in dynamic networks, there have been a number of variations on these ideas. The fast forward propagation method [1], [26], [27] is an online version of the BPTT method. It has fewer computations than the RTRL algorithm, but more than the standard BPTT algorithm. The Green's function method [1], [24] is a variation of the RTRL algorithm, which is designed to reduce the computational complexity by efficient ordering of operations. The block update method [23], [31] updates the gradient only once every M steps. This algorithm combines characteristics of RTRL and BPTT and is well suited for very long sequences. The phased backpropagation (PB) method [10] is a combination of BPTT and temporal backpropagation (TB) [33]. TB is an efficient method for computing the gradient for a feedforward network with imbedded tapped delay lines, and PB incorporates some of these efficiencies in an algorithm that works on general ordered networks [30]. Another algorithm that is a combination of RTRL and BPTT is described in [16]. It is developed for a specific long short-term memory network architecture that is designed to store information over long time intervals.

The objective of most of these variations is to change the order in which operations are performed, in order to minimize the number of computations, or in order to allow an online implementation. Since most of the algorithms produce the same answer (since the gradient calculation is exact), the only two distinguishing features are the computational burden and the ability for online implementation.

The RTRL gradient algorithm has been discussed in a number of previous papers [20], [32], but generally in the context of specific network architectures. The BPTT gradient algorithm has been described as a basic concept in [30], and a diagrammatic method for deriving the gradient algorithm for arbitrary architectures has been provided in [34]. In this paper, however, we will describe exact RTRL and BPTT gradient and Jacobian algorithms (and provide pseudocode) for a general class of dynamic neural networks. The general formulation of these algorithms will allow us to find the most efficient implementa-

Manuscript received May 5, 2005; revised March 27, 2006.

O. De Jesús is with the Research Department, Halliburton Energy Services, Dallas, TX 75006 USA.

M. T. Hagan is with the School of Electrical and Computer Engineering, Oklahoma State University, Stillwater, OK 74078-5032 USA (e-mail: mhagan@ieee.org).

Digital Object Identifier 10.1109/TNN.2006.882371

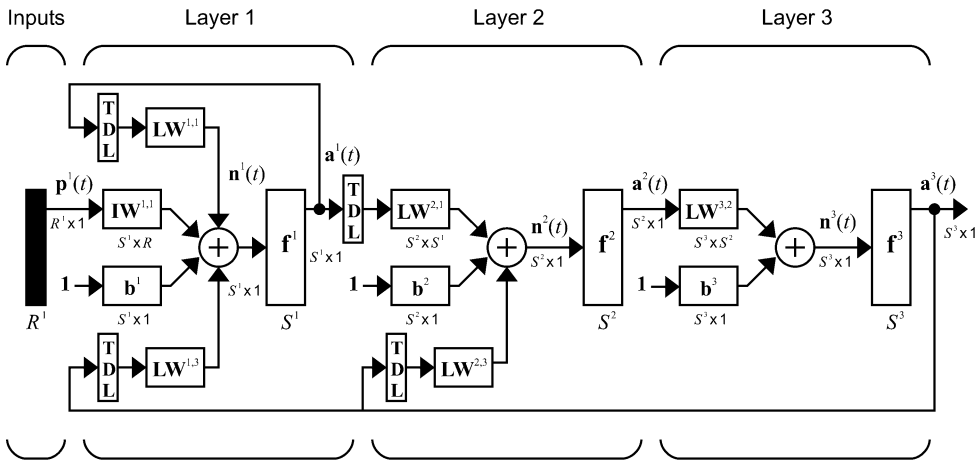


Fig. 1. Example three-layer network represented in the LDDN framework.

tions. As opposed to results described in most previous papers, we will consider arbitrary network architectures, with an arbitrary number of feedback loops and delays. Also, whereas most previous papers have concentrated on the gradient calculation alone, we will discuss the Jacobian calculation as well. Gradients (derivatives of performance index with respect to network weights) are needed for steepest descent, conjugate gradient and quasi-Newton methods, whereas Jacobians (derivatives of network outputs with respect to network weights) are used for Gauss–Newton, Levenberg–Marquardt [15], and extended Kalman filter [4], [21], [25] algorithms. As it turns out, computational complexity results for the Jacobian algorithms do not coincide with results for the gradient algorithms.

This paper begins by introducing the layered digital dynamic network (LDDN) framework and then develops general algorithms for gradient and Jacobian calculations for networks that fit within this framework. In Section II, we present the notation necessary to represent the LDDN. Section III contains a discussion of the dynamic backpropagation algorithms that are required to compute training gradients and Jacobians for dynamic networks. The concepts underlying the BPTT and RTRL algorithms are presented in a unified framework and are demonstrated for a simple, single-loop dynamic network. In Section IV, we describe general RTRL and BPTT algorithms for computing training gradients and Jacobians for networks that can be represented within the LDDN framework. The implementation of both algorithms is demonstrated. In Section V, we provide a comparison of the performance of the various algorithms on a variety of dynamic network structures.

II. Layered Digital Dynamic Networks (LDDNs)

In this section, we want to introduce the neural network framework that will be used in the remainder of this paper. We call this framework LDDN: The fundamental unit of this framework is the layer; the networks are digital as opposed to analog (or discrete-time as opposed to continuous-time); and we use the term "dynamic" rather than "recurrent" because we want to include feedforward networks that have memory.

The LDDN is not a particular network architecture, but rather a general framework for representing dynamic networks. There have been a number of general network frameworks that have been presented in the literature. The LDDN is equivalent to the class of general ordered networks discussed in [30] and [9]. It is also equivalent to the signal flow graph class of networks used in [34] and [3]. It is more general than the locally recurrent networks presented in [28], the layered recurrent networks of [2], and the internally recurrent architectures introduced in [21], which were generalized from the recurrent multilayer perceptron architectures described in [11]. Reference [20] presents four building blocks for dynamic networks, and shows how the RTRL algorithm can be implemented for each of the four blocks. If one were to consider arbitrary combinations of these four blocks, that would be equivalent to the LDDN framework. The spatiotemporal networks of [17] are more general than LDDNs, but their definition is somewhat informal: "a parallel distributed information processing structure that is capable of dealing with input data presented across time as well as space." The main purpose of their taxonomy is to classify networks according to their respective properties. The purpose of the LDDN framework is not to classify networks, but rather to allow general and efficient computations of network derivatives (gradients and Jacobians) using both RTRL and BPTT procedures. All of the specific network architectures described in [17] can be represented in the LDDN framework, except for the higher order networks. The LDDN assumes that the net input to the activation function is a linear function of the layer inputs, as in (1). (The authors have developed slightly modified procedures for networks in which there is a differentiable nonlinear relationship between layer inputs and the net input to the activation function. This would include networks with higher order connections. The notation required to describe the more general results, however, complicates the presentation.)

An example of a dynamic network in the LDDN framework is given in Fig. 1. (The architecture of this particular example is of no special importance. Its purpose is simply to allow us to introduce the notation of the LDDN framework. This notation was first introduced in [8]). The general equations for the computation of the net input $\mathbf{n}^m(t)$ for layer m of an LDDN is

$$\mathbf{n}^m(t) = \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d) + \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(t-d) + \mathbf{b}^m \quad (1)$$

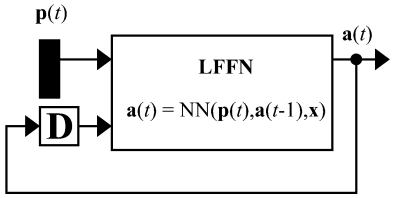


Fig. 2. Simple dynamic network.

where $\mathbf{p}^l(t)$ is the lth input to the network at time t, $\mathbf{IW}^{m,l}$ is the *input weight* between input l and layer m, $\mathbf{LW}^{m,l}$ is the *layer weight* between layer l and layer m, $\mathbf{b}^m$ is the bias vector for layer m, $DL_{m,l}$ is the set of all delays in the tapped delay line between layer l and layer m, $DI_{m,l}$ is the set of all delays in the tapped delay line between input l and layer m, I_m is the set of indices of input vectors that connect to layer m, and L_m^f is the set of indices of layers that directly connect *forward* to layer m. The output of layer m is then computed as

$$\mathbf{a}^m(t) = \mathbf{f}^m(\mathbf{n}^m(t)). \tag{2}$$

The output has S^m elements. The order in which the individual layer outputs must be computed to obtain the correct network output is called the *simulation order*. (This order may not be unique; there may be several valid simulation orders.)

The fundamental unit of the LDDN is the layer. Each layer in the LDDN is made up of five components:

1) a set of weight matrices that come into that layer (which may connect from other layers or from external inputs);
2) any tapped delay lines that appear at the input of a weight matrix (any weight matrix can be preceded by a TDL; for example, layer 1 of Fig. 1 contains the weight $\mathbf{LW}^{1,3}$ and the TDL at its input; the output of the TDL is a vector containing current and previous values of the TDL input);
3) a bias vector;
4) a summing junction;
5) a transfer function.

The output of an LDDN is generally a function not only of the weights, biases, and the current network inputs, but also of outputs of some of the network layers at previous points in time. For this reason, it is not a simple matter to calculate the gradient of the network output with respect to the weights and biases (which is needed to train the network). The weights and biases have two different effects on the network output. The first is the direct effect, which can be calculated using the standard backpropagation algorithm [29], [37], [22]. The second is an indirect effect, since some of the inputs to the network are previous outputs, which are also functions of the weights and biases. The main developments of Sections III and IV are general gradient and Jacobian calculations for arbitrary LDDNs.

III. Principles of Dynamic Learning

To illustrate dynamic backpropagation [35], [36], consider Fig. 2, which is a simple dynamic network. It consists of a layered feedforward network (LFFN) with a single feedback loop added from the output of the network, which is connected to the input of the network through a single delay. (The LFFN contains no feedback loops or delays.) In that figure, the vector $\mathbf{x}$ represents all of the network parameters (weights and biases) and the vector $\mathbf{a}(t)$ represents the output of the LFFN at time step t.

Now suppose that we want to adjust the parameters of the network so as to minimize some performance index $F(\mathbf{x})$. In order to use gradient descent, we need to find the gradient of F with respect to the network parameters. There are two different approaches to this problem. They both use the chain rule, but are implemented in different ways

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)} \tag{3}$$

or

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}}\right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)} \tag{4}$$

where the superscript e indicates an explicit derivative, not accounting for indirect effects through time. The explicit derivatives can be obtained with the standard backpropagation algorithm [29], [37], [22]. To find the complete derivatives that are required in (3) and (4), we need

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}} = \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}} + \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{a}(t-1)} \times \frac{\partial \mathbf{a}(t-1)}{\partial \mathbf{x}} \tag{5}$$

and

$$\frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e F}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}(t)} \times \frac{\partial F}{\partial \mathbf{a}(t+1)}. \tag{6}$$

Equations (3) and (5) make up the RTRL algorithm. Note that the key term is

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}} \tag{7}$$

which must be propagated forward through time. Equations (4) and (6) make up the BPTT algorithm. Here the key term is

$$\frac{\partial F}{\partial \mathbf{a}(t)} \tag{8}$$

which must be propagated backward through time.

In general, the RTRL algorithm requires somewhat more computation than the BPTT algorithm to compute the gradient. (The computational complexity will be discussed in Section V-B.) However, the BPTT algorithm cannot be conveniently implemented in real time, since the outputs must be computed for all time steps, and then the derivatives must be backpropagated back to the initial time point. The RTRL algorithm is well suited for real time implementation, since the derivatives can be calculated at each time step.

IV. Dynamic Backpropagation for the LDDN

In this section, we generalize the RTRL and BPTT gradient and Jacobian algorithms, so that they can be applied to arbitrary LDDNs. This is followed by comparisons of algorithm complexity on several different network architectures.

A. Preliminaries

To explain the general dynamic backpropagation algorithms, we must create certain definitions related to the LDDN. We do that in the following paragraphs.

First, as we stated earlier, a *layer* consists of a set of *weights*, associated *tapped delay lines*, a *summing function*, and a *transfer function*. The network has *inputs* that are connected to special weights, called *input weights*, and denoted by $\mathbf{IW}^{i,j}$, where j denotes the number of the input vector that enters the weight, and i denotes the number of the layer to which the weight is connected. The weights connecting one layer to another are called *layer weights* and are denoted by $\mathbf{LW}^{ij}$, where j denotes the number of the layer coming into the weight and i denotes the number of the layer at the output of weight. In order to calculate the network response in stages layer by layer, we need to proceed in the proper layer order, so that the necessary inputs at each layer will be available. This ordering of layers is called the *simulation order*. In order to backpropagate the derivatives for the gradient calculations, we must proceed in the opposite order, which is called the *backpropagation order*.

In order to simplify the description of the training algorithm, some layers of the LDDN will be assigned as network outputs, and some will be assigned as network inputs. A layer is an *input layer* if it has an input weight, or if it contains any delays with any of its weight matrices. A layer is an *output layer* if its output will be compared to a target during training, or if it is connected to an input layer through a matrix that has any delays associated with it.

For example, the LDDN shown in Fig. 1 has two output layers (1 and 3) and two input layers (1 and 2). For this network the simulation order is 1–2–3, and the backpropagation order is 3–2–1. As an aid in later derivations, we will define U as the set of all output layer numbers and X as the set of all input layer numbers. For the LDDN in Fig. 1, $U = \{1,3\}$ and $X = \{1,2\}$.

The general equations for simulating an arbitrary LDDN network are given in (1) and (2). At each time point, these equations are iterated forward through the layers, as m is incremented through the simulation order. Time is then incremented from $t = 1$ to $t = Q$.

B. RTRL Gradient Algorithm [7]

In this section, we will generalize the RTRL algorithm, given in (3) and (5), for LDDN networks.

1) Equation (3): The first step is to generalize (3). For the general LDDN network, we can calculate the terms of the gradient by using the chain rule, as in

$$\frac{\partial F}{\partial w} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial \mathbf{a}^u(t)}{\partial w} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right] \tag{9}$$

where w represents $lw_{i,j}^{m,l}(d)$, $iw_{i,j}^{m,l}(d)$, and b_i^m. (The equation is the same for all network parameters.)

2) Equation (5): The next step of the development of the RTRL algorithm is the generalization of (5). Again, we use the chain rule

$$\frac{\partial \mathbf{a}^u(t)}{\partial w} = \frac{\partial^e \mathbf{a}^u(t)}{\partial w} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u'}} \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial w}. \tag{10}$$

In (5), we only had one delay in the system. Now we need to account for each output and also for the number of times each output is delayed before it is input to another layer. That is the reason for the two summations in (10). These equations must be updated forward in time, as t is varied from 1 to Q. The terms

$$\frac{\partial \mathbf{a}^u(t)}{\partial w} \tag{11}$$

are generally set to zero for $t \leq 0$.

To implement (10), we need to compute the terms

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T}. \tag{12}$$

To find the second term on the right, we can use

$$n_k^x(t) = \sum_{l \in L_x^f} \sum_{d' \in DL_{x,l}} \left[\sum_{i=1}^{S_l} lw_{k,i}^{x,l}(d') a_i^l(t-d') \right] + \sum_{l \in L_x^f} \sum_{d' \in DI_{x,l}} \left[\sum_{i=1}^{S_l} iw_{k,i}^{x,l}(d') p_i^l(t-d') \right] + b_k^x. \tag{13}$$

We can now write

$$\frac{\partial^e n_k^x(t)}{\partial a_j^{u'}(t-d)} = lw_{k,j}^{x,u'}(d). \tag{14}$$

If we define the following sensitivity term:

$$s_{k,i}^{u,m}(t) \equiv \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \tag{15}$$

which can be used to make up the following matrix:

$$\begin{aligned} \mathbf{S}^{u,m}(t) &= \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^m(t)^T} \\ &= \begin{bmatrix} s_{1,1}^{u,m}(t) & s_{1,2}^{u,m}(t) & \cdots & s_{1,S^m}^{u,m}(t) \\ s_{2,1}^{u,m}(t) & s_{2,2}^{u,m}(t) & \cdots & s_{2,S^m}^{u,m}(t) \\ \vdots & \vdots & & \vdots \\ s_{S^u,1}^{u,m}(t) & s_{S^u,2}^{u,m}(t) & \cdots & s_{S^u,S^m}^{u,m}(t) \end{bmatrix} \\ &= [\mathbf{s}_1^{u,m}(t) \quad \mathbf{s}_2^{u,m}(t) \quad \cdots \quad \mathbf{s}_{S^m}^{u,m}(t)] \\ &= \begin{bmatrix} {}_1\mathbf{s}^{u,m}(t)^T \\ {}_2\mathbf{s}^{u,m}(t)^T \\ \vdots \\ {}_{S^u}\mathbf{s}^{u,m}(t)^T \end{bmatrix} \end{aligned} \tag{16}$$

then we can write (12) as

$$\left[\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T}\right]_{i,j} = \sum_{k=1}^{S_x} s_{i,k}^{u,x}(t+d) \times lw_{k,j}^{x,u'}(d) \tag{17}$$

or in matrix form

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} = \mathbf{S}^{u,x}(t) \times \mathbf{LW}^{x,u'}(d). \tag{18}$$

Therefore, (10) can be written

$$\frac{\partial \mathbf{a}^u(t)}{\partial w} = \frac{\partial^e \mathbf{a}^u(t)}{\partial w} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u'}} \mathbf{S}^{u,x}(t) \times \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial w}. \tag{19}$$

Many of the terms in the summation on the right-hand side of (19) will be zero and will not have to be computed. To take advantage of these efficiencies, we introduce the following definitions:

$$E_{LW}^U(x) = \{u \in U \ni \exists(\mathbf{LW}^{x,u} \neq 0)\} \tag{20}$$

$$E_S^X(u) = \{x \in X \ni \exists(\mathbf{S}^{u,x} \neq 0)\} \tag{21}$$

$$E_S(u) = \{x \ni \exists(\mathbf{S}^{u,x} \neq 0)\}. \tag{22}$$

Using (20) and (21), we can rearrange the order of the summations in (19) and sum only over existing terms

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{w}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \times \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{w}^T} \tag{23}$$

where $\mathbf{w}$ is a vector containing all of the weights and biases in the network.

Equation (23) makes up the generalization of (5) for the LDDN network. It remains to compute the sensitivity matrices $\mathbf{S}^{u,m}(t)$ and the explicit derivatives $\partial^e \mathbf{a}^u(t)/\partial w$, which are described in Sections IV-B3 and IV-B4.

3) Sensitivities: In order to compute the elements of the sensitivity matrix, we use a form of standard static backpropagation [14], [22], [29], [37]. The sensitivities at the outputs of the network can be computed as

$$s_{k,i}^{u,u}(t) = \frac{\partial^e a_k^u(t)}{\partial n_i^u(t)} = \begin{bmatrix} \dot{f}^u\left(n_i^u(t)\right), & \text{for } i = k \\ 0, & \text{for } i \neq k \end{bmatrix}, \qquad u \in U \tag{24}$$

or, in matrix form

$$\mathbf{S}^{u,u}(t) = \dot{\mathbf{F}}^u(\mathbf{n}^u(t)) \tag{25}$$

where $\dot{\mathbf{F}}^u(\mathbf{n}^u(t))$ is defined as

$$\dot{\mathbf{F}}^u(\mathbf{n}^u(t)) = \begin{bmatrix} \dot{f}^u\left(n_1^u(t)\right) & 0 & \cdots & 0 \\ 0 & \dot{f}^u\left(n_2^u(t)\right) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \dot{f}^u\left(n_{S^u}^u(t)\right) \end{bmatrix}. \tag{26}$$

The matrices $\mathbf{S}^{u,m}(t)$ can be computed by backpropagating through the network, from each network output, using

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in L_m^b} \mathbf{S}^{u,l}(t) \mathbf{LW}^{l,m}(0)\right] \dot{\mathbf{F}}^m(\mathbf{n}^m(t)), \qquad u \in U \tag{27}$$

where m is decremented from u through the backpropagation order, and L_m^b is the set of indices of layers that are directly connected backwards to layer m (or to which layer m connects forward) and that contain no delays in the connection.

4) Explicit Derivatives: We also need to compute the explicit derivatives

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial w}. \tag{28}$$

Using the chain rule of calculus, we can derive the following expansion of (28) for input weights:

$$\frac{\partial^e a_k^u(t)}{\partial i w_{i,j}^{m,l}(d)} = \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \times \frac{\partial^e n_i^m(t)}{\partial i w_{i,j}^{m,l}(d)} = s_{i,k}^{u,m}(t) \times p_j^l(t-d). \tag{29}$$

In vector form, we can write

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial i w_{i,j}^{m,l}(d)} = \mathbf{s}_i^{u,m}(t) \times p_j^l(t-d). \tag{30}$$

In matrix form, we have

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathrm{vec}(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t) \tag{31}$$

and in a similar way we can derive the derivatives for layer weights and biases

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathrm{vec}(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t) \tag{32}$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t) \tag{33}$$

where the vec operator transforms a matrix into a vector by stacking the columns of the matrix one underneath the other, and is the $\mathbf{A} \otimes \mathbf{B}$ Kronecker product of and $\mathbf{A}$and $\mathbf{B}$ [18].

5) Summary: The total RTRL algorithm for the LDDN network is summarized in the following pseudocode.

Real-Time Recurrent Learning Gradient

`Initialize`

$$\frac{\partial \mathbf{a}^u(t)}{\partial w} = \mathbf{0}, \qquad t \leq 0 \text{ for all } u \in U.$$

For $t = 1$ to Q

$U' = \oslash$, $E_S(u) = \oslash$, and $E_S^X(u) = \oslash$ for all $u \in U$.

For m decremented through the BP order

For all $u \in U'$

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0)\right]\dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the set $E_S(u)$

if $m \in X$, add m to the set $E_S^X(u)$

EndFor u

If $m \in U$

$$\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the sets U' and $E_S(m)$

if $m \in X$, add m to the set $E_S^X(u)$

EndIf m

EndFor m

For $u \in U$ decremented through the BP order

For all weights and biases ($\mathbf{w}$ is a vector containing all weights and biases)

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathrm{vec}(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathrm{vec}(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t)$$

EndFor weights and biases

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{w}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \times \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{w}^T}$$

EndFor u

EndFor t

Compute Gradients

$$\frac{\partial F}{\partial \mathbf{w}^T} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial^e F}{\partial \mathbf{a}^u(t)}\right]^T \times \frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{w}^T}\right]$$

C. RTRL Jacobian Algorithm

Section IV-B described the real-time recurrent learning algorithm for computing the gradient. This gradient could be used for steepest descent, conjugate gradient, or quasi-Newton training algorithms [14]. If we want to implement a Gauss–Newton or a Levenberg–Marquardt algorithm [14], we need to calculate the Jacobian instead of the gradient. In this section, we will present the RTRL implementation of the Jacobian calculation.

Consider a vector of dimension N that contains outputs for all output layers U

$$\mathbf{a} = [(\mathbf{a}^{u_1})^T (\mathbf{a}^{u_2})^T \cdots (\mathbf{a}^{u_{Nu}})^T]^T = [a_1\ a_2\ a_3\ \cdots\ a_{N-1}\ a_N]^T \tag{34}$$

where N_u is the number of output layers and $N = \sum_{u \in U} S_u$.

The k, j element of the Jacobian is

$$\frac{\partial a_k(t)}{\partial w_j}. \tag{35}$$

These terms are computed as part of the RTRL calculation of the gradient [(31)–(33)]. As also noticed by Yang [35], the Jacobian generation does not require the calculation of the derivative of the objective function, as in (9). Therefore, the RTRL implementation of the Jacobian calculation is actually a simplified version of the RTRL gradient algorithm.

D. BPTT Gradient Algorithm [6]

In this section, we will generalize the BPTT algorithm, given in (4) and (6), for LDDN networks.

1) Equation (4): The first step is to generalize (4). For the general LDDN network, we can calculate the terms of the gradient by using the chain rule, as in

$$\frac{\partial F}{\partial lw_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q} \left[\sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}\right] \frac{\partial^e n_i^m(t)}{\partial lw_{i,j}^{m,l}(d)} \tag{36}$$

(for the layer weights), where u is an output layer, U is the set of all output layers, and S_u is the number of neurons in layer u.

From (13), we can write

$$\frac{\partial^e n_i^m(t)}{\partial lw_{i,j}^{m,l}(d)} = a_j^l(t-d). \tag{37}$$

We will also define

$$d_i^m(t) = \sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}. \tag{38}$$

The terms of the gradient for the layer weights can then be written

$$\frac{\partial F}{\partial lw_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q} d_i^m(t) a_j^l(t-d). \tag{39}$$

If we use the sensitivity term defined in (15)

$$s_{k,i}^{u,m}(t) \equiv \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \tag{40}$$

then the elements $d_i^m(t)$ can be written

$$d_i^m(t) = \sum_{u\in U}\sum_{k=1}^{S_u} \frac{\partial F}{\partial a_k^u(t)} \times s_{k,i}^{u,m}(t). \tag{41}$$

In matrix form, this becomes

$$\mathbf{d}^m(t) = \sum_{u\in U} [\mathbf{S}^{u,m}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^u(t)} \tag{42}$$

where

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \left[\frac{\partial F}{\partial a_1^u(t)} \quad \frac{\partial F}{\partial a_2^u(t)} \quad \cdots \quad \frac{\partial F}{\partial a_{S_u}^u(t)}\right]^T. \tag{43}$$

Now the gradient can be written in matrix form

$$\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{a}^l(t-d)]^T \tag{44}$$

and by similar steps we can find the derivatives for the biases and input weights

$$\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{p}^l(t-d)]^T \tag{45}$$

$$\frac{\partial F}{\partial \mathbf{b}^m} = \sum_{t=1}^{Q} \mathbf{d}^m(t). \tag{46}$$

Equations (44)–(46) make up the generalization of (4) for the LDDN network.

2) Equation (6): The next step in the development of the BPTT algorithm is the generalization of (6). Again, we use the chain rule

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{u'\in U}\sum_{x\in X}\sum_{d\in DL_{x,u}} \left[\frac{\partial^e \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^x(t+d)^T} \times \frac{\partial^e \mathbf{n}^x(t+d)}{\partial \mathbf{a}^u(t)^T}\right]^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)}. \tag{47}$$

(Many of the terms in these summations will be zero. We will provide a more efficient representation later in this section.) In (6), we only had one delay in the system. Now we need to account for each network output, how that network output is connected back through a network input, and also for the number of times each network output is delayed before it is applied to a network input. That is the reason for the three summations in (47). This equation must be updated backward in time, as t is varied from Q to 1. The terms

$$\frac{\partial F}{\partial \mathbf{a}^{u'}(t)} \tag{48}$$

are generally set to zero for $t > Q$.

If we consider the matrix in the brackets on the right side of (47), from (18), we can write

$$\frac{\partial^e \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^x(t+d)^T} \times \frac{\partial^e \mathbf{n}^x(t+d)}{\partial \mathbf{a}^u(t)^T} = \mathbf{S}^{u',x}(t+d) \times \mathbf{LW}^{x,u}(d). \tag{49}$$

This allows us to write (47) as

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{u'\in U}\sum_{x\in X}\sum_{d\in DL_{x,u}} [\mathbf{S}^{u',x}(t+d) \times \mathbf{LW}^{x,u}(d)]^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)}. \tag{50}$$

Many of the terms in the summation on the right-hand side of (50) will be zero and will not have to be computed. In order to provide a more efficient implementation of (50), we define the following sets:

$$E_{LW}^X(u) = \{x \in X \ni \exists(\mathbf{LW}^{x,u} \neq 0)\} \tag{51}$$

$$E_S^U(x) = \{u \in U \ni \exists(\mathbf{S}^{u,x} \neq 0)\}. \tag{52}$$

We can now rearrange the order of the summation in (50) and sum only over the existing terms

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{x\in E_{LW}^X(u)}\sum_{d\in DL_{x,u}} \mathbf{LW}^{x,u}(d)^T \times \sum_{u'\in E_S^U(x)} \mathbf{S}^{u',x}(t+d)^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)}. \tag{53}$$

3) Summary: The total BPTT algorithm is summarized in the following pseudocode.

Backpropagation-Through-Time Gradient

```
Initialize:
        ∂F/∂a^u(t) = 0,        t > Q  for all  u ∈ U

For t = Q to 1
        U' = ∅  and  E_S(u) = ∅  for all  u ∈ U.

    For m decremented through the BP order
        For all u ∈ U'
    S^{u,m}(t) = [ Σ_{l ∈ E_S(u) ∩ L_m^b} S^{u,l}(t) LW^{l,m}(0) ] Ḟ^m(n^m(t))
            add m to the set E_S(u)
        EndFor u
        If m ∈ U
            S^{m,m}(t) = Ḟ^m(n^m(t))

            add m to the sets U' and E_S(m)
        EndIf m
    EndFor m
    For u ∈ U decremented through the BP order
     ∂F/∂a^u(t) = ∂^e F/∂a^u(t) + Σ_{x ∈ E_LW^x(u)} Σ_{d ∈ DL_{x,u}} LW^{x,u}(d)^T
                × Σ_{u' ∈ E_S^U(x)} S^{u',x}(t+d)^T
                × ∂F/∂a^{u'}(t+d)
```

```
    EndFor u
    For all layers m
```

$$\mathbf{d}^m(t) = \sum_{u \in E_S^U(m)} [\mathbf{S}^{u,m}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^u(t)}$$

```
    EndFor m
EndFor t
Compute Gradients
```

$$\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{a}^l(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{p}^l(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^m} = \sum_{t=1}^{Q} \mathbf{d}^m(t)$$

E. BPTT Jacobian Algorithm

In Section IV-D, we presented the RTRL calculation of the Jacobian [defined in (35)]. In this section, we present the BPTT calculation of the Jacobian. In the RTRL algorithm, the elements of the Jacobian are computed as part of the gradient calculation. For this reason, the RTRL Jacobian calculation was a simplified version of the RTRL gradient calculation. This is not true for the BPTT algorithm.

As shown in (8), the key term computed in the BPTT gradient algorithm is $\partial F/\partial \mathbf{a}^{u'}(t)$. In order to compute the Jacobian, we want to replace this term with $\partial a_k(t)/\partial \mathbf{a}^{u'}(t')$ [where a_k is defined in (34)]. Equation (47) can then be modified as follows:

$$\frac{\partial a_k(t)}{\partial \mathbf{a}^u(t')} = \frac{\partial^e a_k(t)}{\partial \mathbf{a}^u(t')} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u}} \left[\frac{\partial^e \mathbf{a}^{u'}(t'+d)}{\partial \mathbf{n}^x(t'+d)^T} \times \frac{\partial^e \mathbf{n}^x(t'+d)}{\partial \mathbf{a}^u(t')^T} \right]^T \times \frac{\partial a_k(t)}{\partial \mathbf{a}^{u'}(t'+d)} \quad (54)$$

where this equation must be updated backward in time, as t' is varied from t to 1. The terms

$$\frac{\partial a_k(t)}{\partial \mathbf{a}^{u'}(t')} \quad (55)$$

are set to zero for $t' > t$. The explicit derivative for each output layer will be

$$\frac{\partial^e a_k(t)}{\partial a_o^u(t')} = \frac{\partial^e a_k^{u^*}(t)}{\partial a_o^u(t')} = \begin{cases} -1, & \text{if } t = t',\ u = u^*,\ o = k \\ 0, & \text{else} \end{cases} \quad (56)$$

where $a_k^{u^*}(t)$ are the output layers being used as targets.

Using (49), and the simplification used in (53), we can modify (54) to obtain

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} = \frac{\partial^e a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} + \sum_{x \in E_{LW}^x(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^T \times \sum_{u' \in E_S^U(x)} \mathbf{S}^{u',x}(t'+d)^T \times \frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^{u'}(t'+d)} \quad (57)$$

where the explicit derivatives are computed using (56). Equation (57) must be solved backwards in time as t' is varied from t to 1.

We next want to use the terms $\partial a_k^{u^*}(t)/\partial \mathbf{a}^u(t')$ to compute the elements of the Jacobian, $\partial a_k^{u^*}(t)/\partial w_j$. This can be done by replacing the cost function F with the outputs $a_k^{u^*}(t)$ in (36)

$$\frac{\partial a_k^{u^*}(t)}{\partial lw_{i,j}^{m,l}} = \sum_{t'=1}^{t} \left[\sum_{u \in U} \sum_{k=1}^{S_u} \frac{\partial a_k^{u^*}(t)}{\partial a_k^u(t')} \times \frac{\partial^e a_k^u(t')}{\partial n_i^m(t')} \right] \times \frac{\partial^e n_i^m(t')}{\partial lw_{i,j}^{m,l}(d)}. \quad (58)$$

Applying a similar development to the one shown in (36)–(46), we have that

$$\mathbf{d}_k^m(t,t') = \sum_{u \in U} [\mathbf{S}^{u,m}(t')]^T \times \frac{\partial a_k^{u^*}(t)}{\partial \mathbf{a}^u(t')} \quad (59)$$

and

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t'=1}^{t} \mathbf{d}^m(t,t') \times [\mathbf{a}^l(t'-d)]^T \quad (60)$$

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t'=1}^{t} \mathbf{d}^m(t,t') \times [\mathbf{p}^l(t'-d)]^T \quad (61)$$

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{b}^m} = \sum_{t'=1}^{t} \mathbf{d}^m(t,t'). \quad (62)$$

The combination of (57) and (59)–(62) make up the BPTT Jacobian calculation. Unlike the RTRL algorithm, where the Jacobian calculation is simpler than the gradient calculation, the BPTT Jacobian calculation is more complex than the BPTT gradient. This is because the RTRL gradient calculation computes the elements of the Jacobian as an intermediate step, whereas the BPTT gradient calculation does not. For the BPTT Jacobian, at each point in time we must solve (57) backwards to the first time point.

V. Algorithm Comparisons

In this section, we will compare the performance of the RTRL and BPTT gradient and Jacobian algorithms in terms of speed and memory requirements. The performance is primarily dependent on the number of weights in the network N and on the length of the training sequence Q. However, it is also dependent, in a more complex way, on the architecture of the network. In recent papers, algorithm performance has been measured on one network architecture (typically a fully recurrent network). We have tested the algorithms through computer simulations on 32 different neural networks. These networks had a variety of architectures, with as many as ten different layers. A schematic diagram showing the basic architectures of the networks is shown in Fig. 3, where a rectangle represents a layer, and a filled circle represents a tapped delay line. (A complete description of all 32 networks is given in [5]). Because of limited space, we will summarize the results obtained on the 32 networks, and will analyze in more detail the results for the simple network labeled 30 in Fig. 3. The number of neurons in the first layer, and the number of delays in the tapped delay line were varied in some of the experiments.

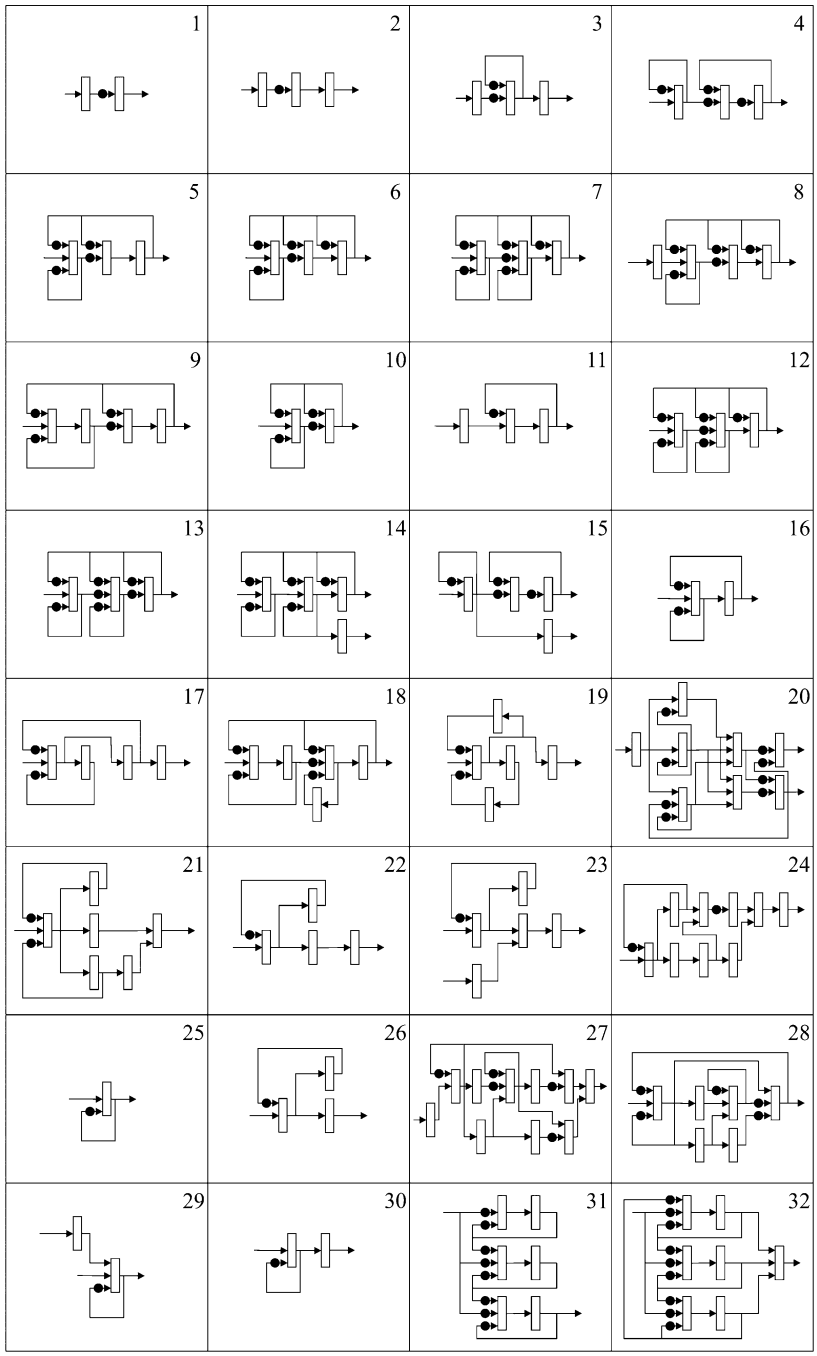


Fig. 3. Schematic representation of 32 test networks.

A. Memory

We will first compare memory requirements for the BPTT and RTRL algorithms. Except for two of the 32 networks that we tested, the BPTT gradient algorithm required about 10% more memory than the RTRL gradient algorithm. In the two networks where the RTRL algorithm required more memory, the network had more weights because there were more delays in the network structure. (We will investigate this effect later in this section.) For each test network, the BPTT Jacobian implementation required about twice as much memory as the RTRL algorithm.

To investigate the effect of the number of weights on the memory requirements, we performed two tests. First, we increased the number of neurons in layer 1 of network 30. Second, we increased the number of delays in the tapped delay line. Both the number of delays and the number of neurons affect the number of weights, but their effect on memory requirements is slightly different.

Table I summarizes the results of changing the number of neurons in the first layer S^1. Here we can see that the BPTT algorithms require more memory than the RTRL algorithms. We can also see that the memory for the BPTT gradient algorithm increases faster than the RTRL gradient algorithm as the size of the first layer increases. A different behavior is seen for the Jacobian algorithms, where the RTRL memory requirements increase faster than BPTT as the size of the first layer increases.

Table II demonstrates how changing the number of delays affects memory. This table shows a different pattern than was

TABLE I
RTRL AND BPTT MEMORY (BYTES) COMPARISON FOR NETWORK 30 AS A FUNCTION OF FIRST LAYER SIZE

	Gradient			Jacobian			
S^1	RTRL	BPTT	BPTT/ RTRL	RTRL	BPTT	BPTT/ RTRL	# Weights
1	76056	85998	1.131	83094	185442	2.232	5
2	79688	91678	1.150	87638	195730	2.233	11
3	84744	99182	1.170	93910	208146	2.216	19
4	91320	108510	1.188	102006	222690	2.183	29
5	99512	119662	1.202	112022	239362	2.137	41
6	109416	132638	1.212	124054	258162	2.081	55

TABLE II
RTRL AND BPTT MEMORY COMPARISON FOR NETWORK 30 AS A FUNCTION OF NUMBER OF DELAYS

	Gradient			Jacobian			
ND	RTRL	BPTT	BPTT/ RTRL	RTRL	BPTT	BPTT/ RTRL	# Weights
1	84744	99182	1.170	93910	208146	2.216	19
2	88472	101414	1.146	99006	221250	2.235	28
3	92776	103646	1.117	104678	234786	2.243	37
4	97656	105878	1.084	110926	248754	2.243	46
5	103112	108110	1.048	117750	263154	2.235	55
6	109144	110342	1.011	125150	277986	2.221	64

TABLE III
RTRL AND BPTT MEMORY COMPARISON FOR NETWORK 30 AS A FUNCTION OF NUMBER OF SAMPLES

	Gradient			Jacobian		
NS	RTRL	BPTT	BPTT/ RTRL	RTRL	BPTT	BPTT/ RTRL
20	94770	105638	1.115	106680	236786	2.220
40	128210	156598	1.221	153720	590786	3.843
60	161650	207558	1.284	200760	1117586	5.567
80	195090	258518	1.325	247800	1817186	7.333
100	228530	309478	1.354	294840	2689586	9.122
120	261970	360438	1.376	341880	3734786	10.924
140	295410	411398	1.393	388920	4952786	12.735
160	328850	462358	1.406	435960	6343586	14.551
180	362290	513318	1.417	483000	7907186	16.371
200	395730	564278	1.426	530040	9643586	18.194
220	429170	615238	1.434	577080	11552786	20.019
240	462610	666198	1.440	624120	13634786	21.846
260	496050	717158	1.446	671160	(*)	(*)
280	529490	768118	1.451	718200	(*)	(*)
300	562930	819078	1.455	795240	(*)	(*)
400	760130	1073878	1.413	1000440	(*)	(*)
500	897330	1328678	1.481	1235640	(*)	(*)
600	1064530	1583478	1.487	1470840	(*)	(*)
1000	1733330	2602678	1.502	2413330	(*)	(*)
5000	8421330	12794678	1.519	11819640	(*)	(*)

(*) Tests were not performed due to computer memory limitations.

seen for layer size. In this case, the memory requirements for the BPTT gradient increase slower than those for the RTRL gradient as the number of delays increases. On the other hand, the memory requirements for the BPTT Jacobian increase faster than those for the RTRL Jacobian, although the changes are small.

The storage requirements are also dependent on the number of samples in the training sequence. This is demonstrated in Table III. This table shows that the memory requirements for the BPTT gradient and Jacobian increase faster than those for the RTRL gradient and Jacobian as the number of samples increases.

B. Speed

We will next investigate algorithm speed. First, we will compare the overall performances of RTRL and BPTT on the 32 test networks. Then, we will analyze the effect of network size and sequence length on algorithm performance. The algorithms

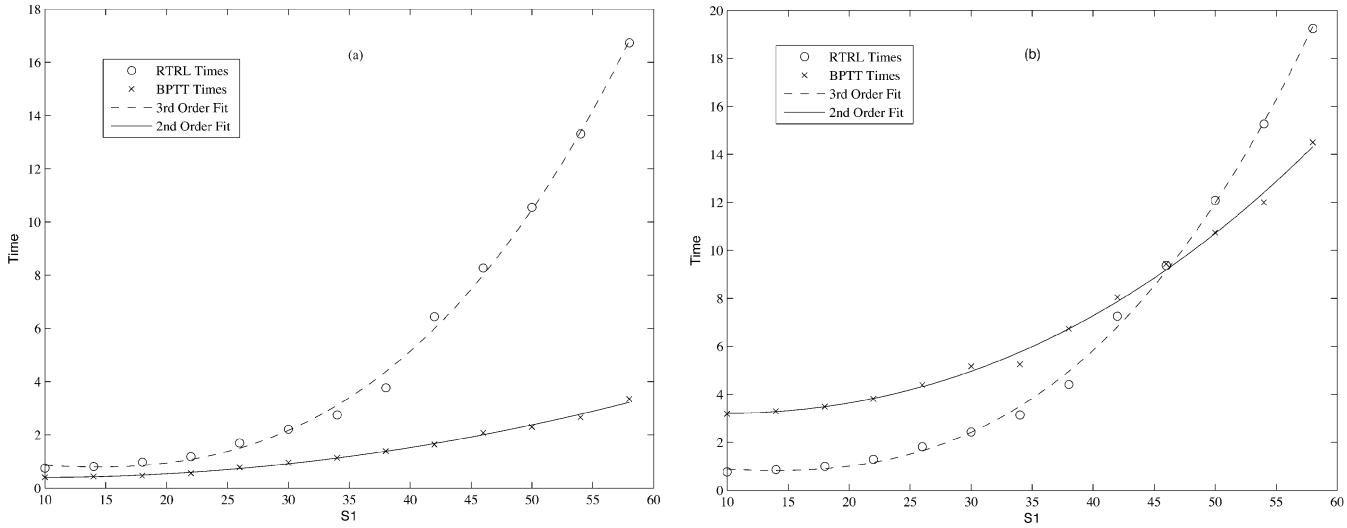


Fig. 4. (a) Gradient and (b) Jacobian calculation times for RTRL and BPTT versus S^1.

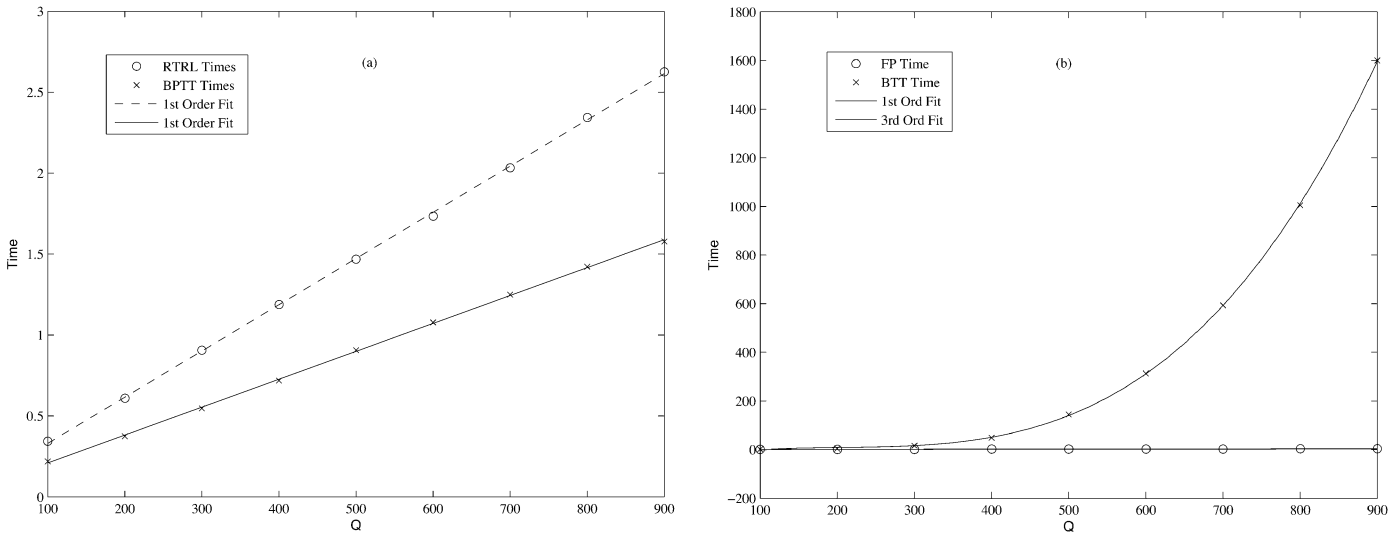


Fig. 5. (a) Gradient and (b) Jacobian calculation times for RTRL and BPTT versus Q.

were implemented in MATLAB on a Pentium III PC. Because MATLAB is optimized for vector operations, we compared computational complexity using both computation time and the number of floating point operations (flops). The basic trends were consistent using both measures.

When compared on the 32 different test networks, the BPTT gradient was between 1.7–3.8 times faster than the RTRL gradient. On the other hand, for the Jacobian calculation, the RTRL algorithm was between 1.2–4.2 times faster than the BPTT algorithm. When combined with the memory results, this suggests that BPTT is the best algorithm for the gradient calculation, but that the RTRL algorithm is the best algorithm for the Jacobian calculation. (This does not consider the fact that the RTRL algorithm is better suited for real-time implementation.)

Now let us consider the effect of network size and sequence length on performance for one simple network—network 30 of Fig. 3. Results can vary significantly with network architecture, but an analysis of one architecture can demonstrate how to approach the process and can demonstrate some general trends. We will begin with a theoretical analysis of the algorithms' complexities, and then we will verify with experimental results. We will analyze the four algorithms (BPTT and RTRL gradient and Jacobian) with respect to three variables: the number of neurons in layer 1 (S^1), the number of delays in the tapped delay line (D), and the length of the training sequence (Q).

The complexity of the BPTT gradient calculation is generally determined by (44). For network 30, the most important weights will be $\mathbf{LW}^{1,1}(d)$

$$\frac{\partial F}{\partial \mathbf{LW}^{1,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^1(t) \times [\mathbf{a}^1(t-d)]^T. \tag{63}$$

The outer product calculation involves $(S^1)^2$ operations, which must be done for Q time steps and for D delays, so the BPTT gradient calculation is $O[(S^1)^2 DQ]$.

The complexity of the BPTT Jacobian calculation is generally determined by (60). Again, the most important weights for network 30 will be $\mathbf{LW}^{1,1}(d)$

$$\frac{\partial a_k^{u^*}(t)}{\partial \mathbf{LW}^{1,1}(d)} = \sum_{t'=1}^{t} \mathbf{d}^1(t,t') \times [\mathbf{a}^1(t'-d)]^T. \tag{64}$$

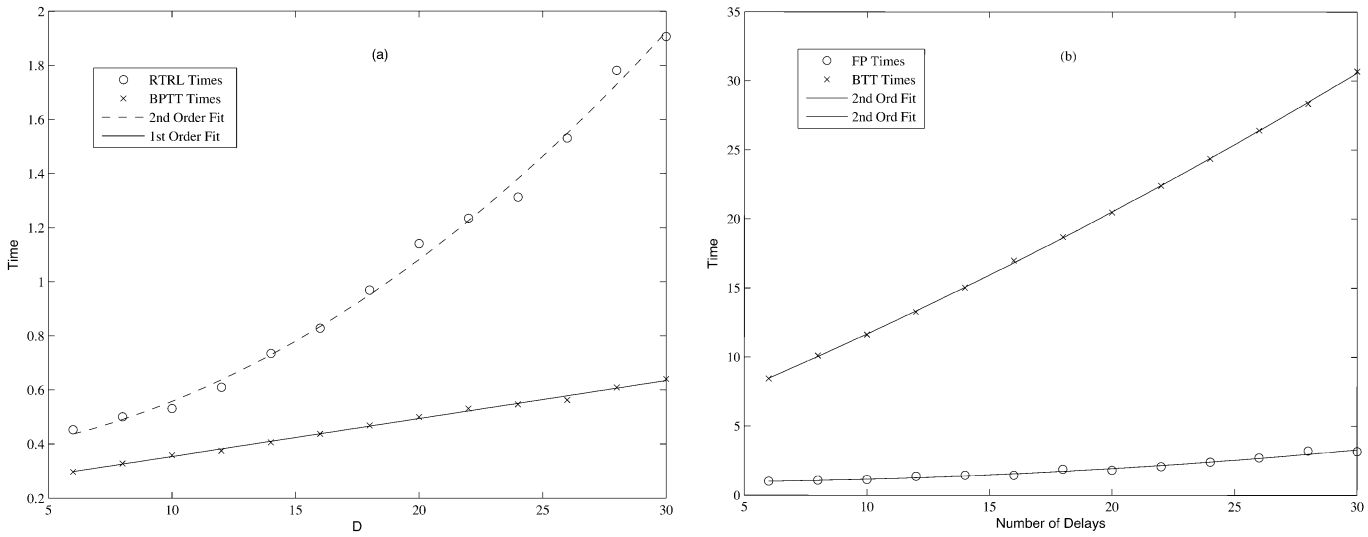


Fig. 6. (a) Gradient and (b) Jacobian calculation times for RTRL and BPTT versus D.

The outer product calculation involves $(S^1)^2$ operations, which must be done in the order of Q^2 time steps and for D delays. In addition, this operation must be performed for all k and u^*, which is of the order S^1, so the BPTT Jacobian calculation is $O[(S^1)^3DQ^2]$.

The RTRL gradient and Jacobian have roughly the same complexity, which is based generally on (23). For network 30, we can consider the equation for $u = 2$

$$\frac{\partial \mathbf{a}^2(t)}{\partial \mathbf{w}^T} = \frac{\partial^e \mathbf{a}^2(t)}{\partial \mathbf{w}^T} + \mathbf{S}^{2,1}(t)\left[\sum_{d=1}^{D} \mathbf{LW}^{1,1}(d) \times \frac{\partial \mathbf{a}^1(t-d)}{\partial \mathbf{w}^T}\right]. \tag{65}$$

Inside the summation, we have a matrix multiplication involving an $S^1 \times S^1$ matrix times an $S^1 \times \{(DS^1+3)S^1+1\}$ matrix. This multiplication will be $O[(S^1)^4D]$. It must be done for every d and every t; therefore, the RTRL gradient and Jacobian calculations are $O[(S^1)^4D^2Q]$. The multiplication by the sensitivity matrix does not change the order of the complexity.

To summarize the theoretical results, the BPTT gradient is $Q[(S^1)^2DQ]$, the BPTT Jacobian is $O[(S^1)^3DQ^2]$, and the RTRL gradient and Jacobian are $O[(S^1)^4D^2Q]$.

Fig. 4 shows experimental results for all four algorithms when varying S^1 with $D = 1$ and $Q = 200$. These are consistent with the theoretical results, in that the complexity of the RTRL algorithms increase faster than the BPTT algorithms for all cases. However, notice that the RTRL Jacobian algorithm is actually faster than the BPTT Jacobian algorithm when S^1 is less than 45 (more than 2000 weights). When S^1 is less than 30, the RTRL algorithm is three times faster than BPTT. This example demonstrates that one must be careful when only looking at the asymptotic order of the algorithm complexity. Many practical applications involve networks of moderate size, so asymptotic results may not be of primary interest. For example, in Fig. 4 the results for the RTRL algorithms show only an $O[(S^1)^3]$ characteristic; in order to see the asymptotic characteristic of $O[(S^1)^4]$, we need to increase S^1 up to 300, which corresponds to over 90 000 weights. The results for the BPTT Jacobian algorithm in Fig. 4 show only an $O[(S^1)^2]$ relationship. To see the asymptotic characteristic of $O[(S^1)^3]$, we need to increase S^1 up to 100.

Fig. 5 shows experimental results for all four algorithms when varying Q with $D = 1$ and $S^1 = 3$. These results agree with the theoretical numbers and show that all algorithms have a linear relationship to Q, except for the BPTT Jacobian. It is because the BPTT Jacobian is $O[Q^2]$ that the RTRL algorithm is generally a better choice for Jacobian calculations.

Fig. 6 shows experimental results for all four algorithms when varying D with $Q = 200$ and $S^1 = 3$. Again, the results are consistent with the theoretical numbers; the BPTT algorithms are $O[D]$, while the RTRL algorithms are $O[D^2]$. However, the limiting order of the algorithms may not be important for many practical applications. Even for delays much larger than 30, the curves are almost linear. As in the other cases, for the network sizes that we have tested, the BPTT algorithm is significantly faster than the RTRL algorithm for the gradient calculation, and the RTRL algorithm is significantly faster than the BPTT algorithm for the Jacobian calculation.

In addition to the experiments shown in Figs. 4–6, we performed a number of other experiments on various of the 32 networks shown in Fig. 3. There is not sufficient space here to detail all of these tests, but in general we can say that as the number of weights of the network increases, while the length of the sequence remains constant, the speed of BPTT will increase relative to RTRL. For small networks, with long training sequences, the RTRL algorithm will be faster. As the size of the network becomes extremely large, the BPTT algorithm will be faster, even for the Jacobian algorithm. What constitutes "large" will depend on the network architecture.

C. Summary

The results presented in this section indicate that BPTT is the best algorithm for gradient calculations and RTRL is the best algorithm for Jacobian calculations for small to medium-sized networks and large sequences. The only case where the BPTT Jacobian outperforms the RTRL Jacobian is for networks with a large number of weights and short training sequences. The

BPTT gradient requires about half of the time and fewer flops than the RTRL gradient. The BPTT gradient required about 10% more memory than RTRL, a value not critical with systems available today. If we compare the Jacobian algorithms, RTRL requires about half of the memory and a third of the time as BPTT. For very large sequences, the BPTT and RTRL gradient tend to be close in computational complexity. It may be best to use the RTRL gradient for long sequences, because of the memory requirements. In addition, theoretical asymptotic complexity may not always be the best way to select the most efficient algorithm. We found in several instances that experimental results showed that for many practical networks the RTRL algorithm provided the fastest results, even though it had higher asymptotic complexity.

Another important point to be gained from this section is that algorithm complexity is dependent on the network architecture, and not just on the number of weights. Whether or not a network has feedback connections, and the number of feedback connections it has, can affect the algorithm complexity.

VI. Summary

This paper has discussed gradient and Jacobian calculations for dynamic neural networks. In order to facilitate the development of general algorithms, we have introduced the LDDN framework, which embodies a very general class of a dynamic network. Two universal dynamic gradients and Jacobian algorithms for the LDDN were presented here. One was based on the BPTT algorithm, and the other was based on the RTRL algorithm. The BPTT gradient algorithm generally requires less computation than the RTRL algorithm, although the RTRL algorithm is better suited for online implementation. The RTRL Jacobian algorithm is a simplified version of the RTRL gradient algorithm and generally requires less computation than the BPTT Jacobian algorithm.

A number of tests of the various algorithms were performed on a variety of network architectures. These tests demonstrate that the BPTT algorithm is usually the best choice for gradient computation, whereas the RTRL algorithm is usually best for the Jacobian computation. However, the architecture of the network can have a strong effect on algorithm performance.

References

[1] A. F. Atiya and A. G. Parlos, "New results on recurrent network training: Unifying the algorithms and accelerating convergence," *IEEE Trans. Neural Netw.*, vol. 11, no. 3, pp. 697–709, May 2000.

[2] N. E. Barabanov and D. V. Prokhorov, "Stability analysis of discrete-time recurrent neural networks," *IEEE Trans. Neural Netw.*, vol. 13, no. 2, pp. 292–303, Mar. 2002.

[3] P. Campolucci, A. Marchegiani, A. Uncini, and F. Piazza, "Signal-flow-graph derivation of on-line gradient learning algorithms," in *Proc. Int. Conf. Neural Netw. (ICNN'97)*, Houston, TX, Jun. 9–12, 1997, pp. 1884–1889.

[4] J. Choi, M. Bouchard, and T. H. Yeap, "Decision feedback recurrent neural equalization with fast convergence rate," *IEEE Trans. Neural Netw.*, vol. 16, no. 3, pp. 699–708, May 2005.

[5] O. De Jesús, "Training General Dynamic Neural Networks," Ph.D. dissertation, Oklahoma State Univ., Stillwater, OK, 2002.

[6] O. De Jesús and M. Hagan, "Backpropagation through time for a general class of recurrent network," presented at the IEEE Int. Joint Conf. Neural Netw., Washington, DC, Jul. 2001.

[7] ——, "Forward perturbation for a general class of dynamic network," in *IEEE Int. Joint Conf. Neural Netw.*, Washington, DC, Jul. 2001, vol. 4, pp. 2626–2631.

[8] H. B. Demuth and M. Beale, *Users' Guide for the Neural Network Toolbox for MATLAB, ver. 4.0*. Natick, MA: The Mathworks, 2000.

[9] L. A. Feldkamp and G. V. Puskorius, "A signal processing framework based on dynamic neural networks with application to problems in adaptation, filtering, and classification," in *Proc. IEEE*, Nov. 1998, vol. 86, no. 11, pp. 2259–2277.

[10] L. A. Feldkamp and D. V. Prokhorov, "Phased backpropagation: A hybrid of BPTT and temporal BP," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 1998, vol. 3, pp. 2262–2267.

[11] B. Fernandez, A. G. Parlos, and W. K. Tsai, "Nonlinear dynamic system identification using artificial neural networks (ANNs)," in *Proc. Int. Joint Conf. Neural Netw.*, San Diego, CA, 1990, vol. 11, pp. 133–141.

[12] M. Gupta, L. Jin, and N. Homma, *Static and Dynamic Neural Networks: From Fundamentals to Advanced Theory*. New York: IEEE and Wiley, 2003.

[13] M. Hagan, O. De Jesús, and R. Schultz, "Training recurrent networks for filtering and control," in *Recurrent Neural Networks: Design and Applications*, L. R. Medsker and L. C. Jain, Eds. Boca Raton, FL: CRC Press, 2000, pp. 325–354.

[14] M. Hagan, H. B. Demuth, and M. Beale, *Neural Network Design*. Boston, MA: PWS, 1996.

[15] M. Hagan and M. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Trans. Neural Netw.*, vol. 5, no. 6, pp. 989–993, Nov. 1994.

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

[17] S. C. Kremer, "Spatio-temporal connectionist networks: A taxonomy and review," *Neural Comput.*, vol. 13, no. 2, pp. 249–306, Feb. 2001.

[18] J. R. Magnus and H. Neudecker, *Matrix Differential Calculus*. New York: Wiley, 1999.

[19] D. Mandic and J. Chambers, *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*. London, U.K.: Wiley, 2001.

[20] K. S. Narendra and A. M. Parthasrathy, "Identification and control for dynamic systems using neural networks," *IEEE Trans. Neural Netw.*, vol. 1, no. 1, p. 4, Jan. 1990.

[21] G. V. Puskorius and L. A. Feldkamp, "Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 279–297, Mar. 1994.

[22] D. Rumelhart, D. Hinton, and G. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*, D. Rumelhart and F. McClelland, Eds. Cambridge, MA: MIT Press, 1986, vol. 1.

[23] J. Schmidhuber, "A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks," *Neural Comput.*, vol. 4, pp. 243–248, 1992.

[24] G. Z. Sun, H. H. Chen, and Y. C. Lee, "Green's method for fast on-line learning algorithm of recurrent neural networks," in *In: Advances in Neural Information Processing Systems 4*, J. Moody, S. Hanson, and R. Lippmann, Eds. San Mateo, CA: Morgan Kaufmann, 1992, pp. 333–340.

[25] P. Tino, M. Cernansky, and L. Benuskova, "Markovian architectural bias of recurrent neural networks," *IEEE Trans. Neural Netw.*, vol. 15, no. 1, pp. 6–15, Jan. 2004.

[26] N. Toomarian and J. Barben, "Adjoint-functions and temporal learning algorithms in neural networks," in *In: Advances in Neural Information Processing Systems 3*, R. Lippmann, J. Moody, and D. Touretzky, Eds. San Mateo, CA: Morgan Kaufmann, 1991, pp. 113–120.

[27] N. Toomarian and J. Barben, "Learning a trajectory using adjoint functions and teacher forcing," *Neural Netw.*, vol. 5, pp. 473–484, 1992.

[28] A. C. Tsoi and A. D. Back, "Locally recurrent globally feedforward networks: A critical review of architectures," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 229–239, Mar. 1994.

[29] P. J. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. dissertation, Harvard Univ., Cambridge, MA, 1974.

[30] P. J. Werbos, "Backpropagation through time: What it is and how to do it," *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, Oct. 1990.

[31] R. J. Williams and J. Peng, "Gradient-based learning algorithms for recurrent neural networks and their computational complexity," in *Backpropagation: Theory, Architectures, and Applications*, Y. Chauvin and D. E. Rumelhart, Eds. Hillsdale, NJ: Lawrence Erlbaum, 1992, pp. 433–486.

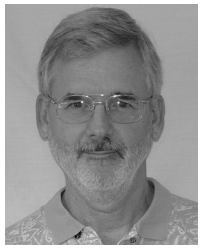
[32] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput.*, vol. 1, pp. 270–280, 1989.

[33] E. A. Wan, "Temporal backpropagation for FIR neural networks," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 1990, vol. 1, pp. 575–580.
[34] E. Wan and F. Beaufays, "Diagrammatic methods for deriving and relating temporal neural networks algorithms," in *Adaptive Processing of Sequences and Data Structures, Lecture Notes in Artificial Intelligence*, M. Gori and C. L. Giles, Eds. New York: Springer-Verlag, 1998.
[35] W. Yang, "Neurocontrol Using Dynamic Learning," Ph.D. dissertation, Oklahoma State Univ., Stillwater, OK, 1994.
[36] W. Yang and M. T. Hagan, "Training recurrent networks," in *Proc. 7th Oklahoma Symp. Artif. Intell.*, Stillwater, OK, 1993, pp. 226–233.
[37] P. J. Werbos, *The Roots of Backpropagation*. New York: Wiley, 1994.

Orlando De Jesús received the Degree of Engineer in electronics and project management specialization from Universidad Simón Bolívar, Caracas, Venezuela, in 1985 and 1992, respectively, and the M.S. and Ph.D. degrees in electrical engineering from Oklahoma State University, Stillwater, OK, in 1998 and 2002, respectively.

He held engineering and management positions at AETI C.A., Caracas, Venezuela, from 1985 to 1996, developing data acquisition and control systems for the oil industry in Venezuela. He later developed the control system blocks and the dynamic neural networks training algorithms for the Matlab Neural Network toolbox. He is currently a Principal Technical Professional in the Research Department, Halliburton Energy Services, Dallas, TX. He has coauthored 13 technical publications and holds seven U.S. patents. His research interests include control systems, signal processing, software modeling, neural networks, robotics, automation, and instrumentation applications.

Dr. De Jesús is a member of the Society of Petroleum Engineers (SPE) and the Instrument Society of America (ISA).



Martin T. Hagan received the B.S. degree in electrical engineering from the University of Notre Dame, Notre Dame, IN, in 1972, the M.S. degree in information and computer science from Georgia Institute of Technology, Atlanta, in 1973, and the Ph.D. degree in electrical engineering from the University of Kansas, Lawrence, in 1977.

He is currently a Professor of Electrical and Computer Engineering at Oklahoma State University, Stillwater, where he has taught and conducted research in the areas of statistical modeling and control systems since 1986. He was previously with the faculty of Electrical Engineering at the University of Tulsa, Tulsa, OK, from 1978 to 1986. He was also a Visiting Scholar with the Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand, during the 1994 academic year and with the Laboratoire d'Analyse et d'Architecture des Systéms, Centre National de la Recherche Scientifique, Toulouse, France, during the 2005–2006 academic year. He is the author, with H. Demuth and M. Beale, of the textbook, *Neural Network Design*. He is also a coauthor of the Neural Network Toolbox for MATLAB.