A General Framework for Dynamic Networks

Martin Hagan[1] and Orlando De Jesús[2]

[1] School of Electrical and Computer Engineering, Oklahoma State University,
Stillwater, Oklahoma, 74078
mhagan@ieee.org

[2] Halliburton - Carrollton Technology Center - Research,
Halliburton Energy Services
Carollton, Texas, 75006
Orlando.DeJesus@Halliburton.com

(*Invited Paper*)

Abstract. The field of dynamic neural networks is expansive. It has applications in such disparate areas as control systems, prediction in financial markets, channel equalization in communication systems, phase detection in power systems, sorting, fault detection, speech recognition, and even the prediction of protein structure in genetics. Within these various application areas, a great number of dynamic neural network architectures have been proposed. These dynamic networks are often trained using gradient-based optimization algorithms. In this paper we will attempt to present a unified view of the training of dynamic networks. We will begin with a very general framework for representing dynamic networks and will demonstrate how gradient-based algorithms can be efficiently developed using this framework.

1 Introduction

Dynamic networks are networks that contain delays (or integrators, for continuous-time networks). These dynamic networks can have purely feedforward connections, or they can also have some feedback (recurrent) connections. Dynamic networks have memory. Their response at any given time will depend not only on the current input, but on the history of the input sequence.

Because dynamic networks have memory, they can be trained to learn sequential or time-varying patterns. This has applications in such diverse areas as control of dynamic systems [1], prediction in financial markets[2], channel equalization in communication systems [3], phase detection in power systems [4], sorting [5], fault detection [6], speech recognition [7], learning of grammars in natural languages [8], and even the prediction of protein structure in genetics [9].

Dynamic networks can be trained using standard gradient-based or Jacobian-based optimization methods. However, the gradients and Jacobians that are required for these methods cannot be computed using the standard backpropagation algorithm. In this paper we will discuss a general dynamic network framework, in which dynamic backpropagation algorithms can be efficiently developed.

There are two general approaches (with many variations) to gradient and Jacobian calculations in dynamic networks: backpropagation-through-time (BPTT) [10] and real-time recurrent learning (RTRL) [11]. 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 on-line, 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 on-line implementation. For Jacobian calculations, the RTRL algorithm is generally more efficient than the BPTT algorithm [12,13].

In order to more easily present general BPTT and RTRL algorithms, it will be helpful to introduce modified notation for networks that can have recurrent connections. In the next section we will introduce this notation and will develop a general dynamic network framework. The following section will present procedures for computing gradients for the general framework.

The RTRL and BPTT methods can be thought of as general concepts, rather than as specific algorithms that can be implemented as general computer codes applicable to arbitrary network architectures. The RTRL gradient algorithm has been discussed in a number of previous papers [11, 14], but generally in the context of specific network architectures. The BPTT gradient algorithm has been described as a basic concept in [10], and a diagrammatic method for deriving the gradient algorithm for a certain class of architectures has been provided in [15].

As a general rule, there have been two major approaches to using dynamic training. The first approach has been to use the general RTRL or BPTT concepts to derive algorithms for particular network architectures. The second approach has been to put a given network architecture into a particular canonical form (e.g., [16-18]), and then to use the dynamic training algorithm which has been previously designed for the canonical form.

In this paper our approach will be to develop a very general framework in which to conveniently represent a large class of dynamic networks, and then to derive the RTRL and BPTT algorithms for the general framework. In this way, one computer code can be used to train arbitrarily constructed network architectures, without requiring that each architecture be first converted to a particular canonical form.

2 Development of a General Class of Dynamic Network

We will build up to our general notation by starting with a simple multilayer perceptron network, as shown in Fig. 1. The equations of operation for this network are

$$\mathbf{n}^{m+1} = \mathbf{W}^{m+1}\mathbf{a}^{m} + \mathbf{b}^{m+1} \text{ for } m = 0,1,\ldots,M-1 \quad \mathbf{(1)}$$

$$\mathbf{a}^m = \mathbf{f}^m\left(\mathbf{n}^m\right) \tag{2}$$

where $\mathbf{n}^m$ is the net input at layer m, $\mathbf{a}^m$ is the output of layer m, and $\mathbf{a}^0 = \mathbf{p}$ is the input to the network. The overall network output is the output of the last layer, $\mathbf{a}^M$.

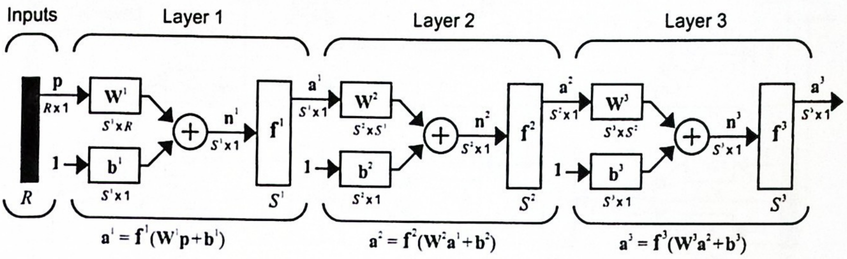

Fig. 1. Multilayer Perceptron

In the Multilayer Perceptron, each layer connects forward to the following layer, and each layer consists of four components:

1. A weight matrix $\mathbf{W}^m$,
2. A bias vector $\mathbf{b}^m$,
3. A summing junction, and
4. A transfer function $\mathbf{f}^m(\ )$.

We can generalize the Multilayer Perceptron by allowing layers to connect forward to an arbitrary number of other layers. In addition, we can have multiple input vectors, each one of which can be connected to any layer. We call the resulting class of networks Layered Feedforward Neural Networks (LFNN). An example is shown in Fig. 2. The equations of operation of an LFFN can be written as

$$\mathbf{n}^m(t) = \sum_{l \in I_m} \mathbf{IW}^{m,l}\mathbf{p}^l(t) + \sum_{l \in L_f^m} \mathbf{LW}^{m,l}\mathbf{a}^l(t) + \mathbf{b}^m \tag{3}$$

$$\mathbf{a}^m = \mathbf{f}^m\left(\mathbf{n}^m\right) \tag{4}$$

where I_m is the set of indices of all inputs that connect to layer *m*, L_f^m is the set of indices of all layers that connect forward to layer *m*., $\mathbf{p}^l$ is the lth input to the network, $\mathbf{IW}^{m,l}$ is the *input* weight between input *l* and layer *m*, $\mathbf{LW}^{m,l}$ is the *layer* weight between layer *l* and layer *m*, and $\mathbf{b}^m$ is the bias vector for layer *m*.

For the LFFN class of networks, we can have multiple weight matrices associated with each layer - some coming from external inputs, and others coming from other layers.

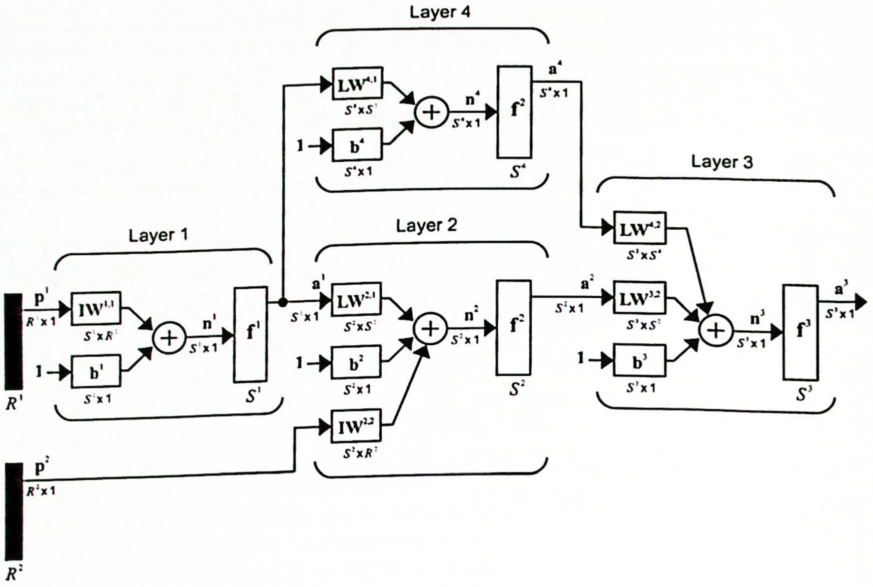

Fig. 2. Example Layered Feedforward Network

So far, we have considered only static networks. The LFFN class of networks can be further generalized to allow delays to be associated with each weight matrix. This leads to Layered Digital Dynamic Networks (LDDN) [12]: the fundamental unit of this framework is the layer (as with the multilayer perceptron); 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.

An example of a dynamic network in the LDDN framework is shown in Fig. 3. Note that the key component that has been added to certain layers is the tapped delay line (indicated by TDL in the figure). The equations of operation of an LDDN network are

$$\mathbf{n}^m(t) = \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(t-d) + \sum_{l \in L_f^m} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d) + \mathbf{b}^m \tag{5}$$

$$\mathbf{a}^m(t) = \mathbf{f}^m\left(\mathbf{n}^m(t)\right) \tag{6}$$

where $DL_{m,l}$ 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

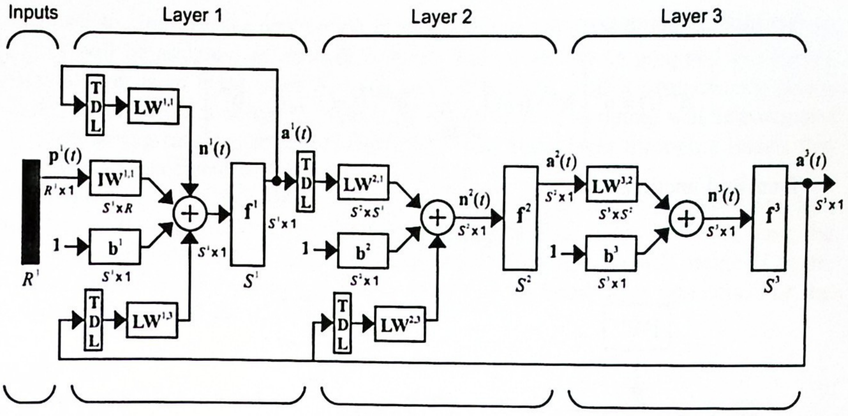

Fig. 3. Example Dynamic Network in the LDDN Framework

The LDDN framework is quite general. It is equivalent to the class of general ordered networks discussed in [10] and [19]. It is also equivalent to the signal flow graph class of networks used in [15] and [20]. However, we can increase the generality of the LDDN further. In all of the classes of networks that we have presented so far, the weight matrix multiplies the corresponding vector coming into the layer (from an external input in the case of **IW**, and from another layer in the case of **LW**). This means that a dot product is formed between each row of the weight matrix and the input vector.

We can consider more general *weight functions* than simply the dot product. For example, radial basis layers compute the distances between the input vector and the rows of the weight matrix. We can allow weight functions with arbitrary (but differentiable) operations between the weight matrix and the input vector. This enables us to include higher-order networks as part of our framework.

Another generality we can introduce is for the *net input function*. This is the function that combines the results of the weight function operations with the bias vector. In all of the networks that we have considered to this point, the net input function has been a simple summation. We can allow arbitrary, differentiable net input functions to be used.

The resulting network framework is the Generalized LDDN (GLDDN). A block diagram for a simple GLDDN (without delays) is shown in Fig. 4. The equations of operation for a GLDDN are

Weight Functions:

$$\mathbf{iz}^{m,l}(t,d) = \mathbf{ih}^{m,l}\left(\mathbf{IW}^{m,l}(d), \mathbf{p}^l(t-d)\right) \tag{7}$$

$$\mathbf{lz}^{m,l}(t,d) = \mathbf{lh}^{m,l}\left(\mathbf{LW}^{m,l}(d), \mathbf{a}^l(t-d)\right) \tag{8}$$

Net Input Functions:

$$\mathbf{n}^m(t) = \mathbf{o}^m\left(\mathbf{iz}^{m,l}(t,d)\Big|_{\substack{l \in I_m \\ d \in DI_{m,l}}}, \mathbf{lz}^{m,l}(t,d)\Big|_{\substack{l \in L_m^f \\ d \in DL_{m,l}}}, \mathbf{b}^m\right) \tag{9}$$

Transfer Functions:

$$\mathbf{a}^m(t) = \mathbf{f}^m\left(\mathbf{n}^m(t)\right) \tag{10}$$

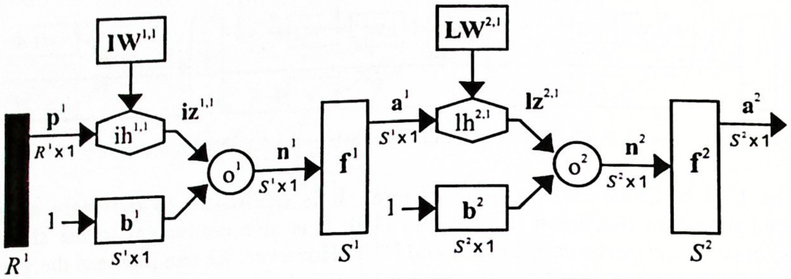

Fig. 4. Example Network with General Weight Functions and Net Input Functions

3 Gradient Calculation for the GLDDN

The next step is to develop an algorithm for computing the gradient for the GLDDN. This can be done using the BPTT or the RTRL approaches. Because of limited space, we will describe only the RTRL algorithm in this paper. (Both approaches are described for the LDDN framework in [12].)

To explain the gradient calculation for the GLDDN, we must create certain definitions. We do that in the following paragraphs.

3.1 Preliminary Definitions

First, as we stated earlier, a *layer* consists of a set of *weights*, associated *weight functions*, associated *tapped delay lines*, a *net input function*, and a *transfer function*. The network has *inputs* that are connected to special weights, called *input weights*. The weights connecting one layer to another are called *layer weights*. 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 gradient calculation, some layers of the GLDDN 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. 3 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. 3, *U*={1,3} and *X*={1,2}.

3.2 Gradient Calculation

The objective of training is to optimize the network performance, quantified in the performance index *F*(**x**), where **x** is a vector containing all of the weights and biases in the network. In this paper we will consider gradient-based algorithms for optimizing the performance (e.g., steepest descent, conjugate gradient, quasi-Newton, etc.). For the RTRL approach, the gradient is computed using

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} \right]^T \frac{\partial^e F(\mathbf{x})}{\partial \mathbf{a}^u(t)}, \tag{11}$$

where

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u'}} \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}. \tag{12}$$

The superscript *e* in these expressions indicates an explicit derivative, not accounting for indirect effects through time.

Many of the terms in Eq. 12 will be zero and need not be included. To take advantage of these efficiencies, we introduce the following definitions

$$E_{LW}^{U}(x) = \{ u \in U \ni \exists (\mathbf{LW}^{x,u} \neq 0) \}, \tag{13}$$

$$E_{S}^{X}(u) = \{ x \in X \ni \exists (\mathbf{S}^{x,u} \neq 0) \}, \tag{14}$$

where

$$\mathbf{S}^{x,u}(t) \equiv \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \tag{15}$$

is the sensitivity matrix.

Using these definitions, we can rewrite Eq. 12 as

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{x,u}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}. \quad \textbf{(16)}$$

The sensitivity matrix can be computed using static backpropagation, since it describes derivatives through a static portion of the network. The static backpropagation equation is

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in L_b^m \cap E_S(u)} \mathbf{S}^{u,l}(t) \frac{\partial^e \mathbf{n}^l(t)}{\partial \mathbf{lz}^{l,m}(t,0)^T} \frac{\partial^e \mathbf{lz}^{l,m}(t)}{\partial \mathbf{a}^m(t)^T} \right] \dot{\mathbf{F}}^m(\mathbf{n}^m(t)),\ u \in U, \quad \textbf{(17)}$$

where m is decremented from u through the backpropagation order, L_b^m 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, and

$$\dot{\mathbf{F}}(\mathbf{n}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{n})}{\partial n_1} & \frac{\partial f_1(\mathbf{n})}{\partial n_2} & \cdots & \frac{\partial f_1(\mathbf{n})}{\partial n_S} \\ \frac{\partial f_2(\mathbf{n})}{\partial n_1} & \frac{\partial f_2(\mathbf{n})}{\partial n_2} & \cdots & \frac{\partial f_2(\mathbf{n})}{\partial n_S} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_S(\mathbf{n})}{\partial n_1} & \frac{\partial f_S(\mathbf{n})}{\partial n_2} & \cdots & \frac{\partial f_S(\mathbf{n})}{\partial n_S} \end{bmatrix}. \quad \textbf{(18)}$$

There are four terms in Eqs. 16 and 17 that need to be computed:

$$\frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T},\ \frac{\partial^e \mathbf{n}^l(t)}{\partial \mathbf{lz}^{l,m}(t,0)^T},\ \frac{\partial^e \mathbf{lz}^{l,m}(t,0)}{\partial \mathbf{a}^m(t)^T},\ \text{and}\ \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T}. \quad \textbf{(19)}$$

The first term can be expanded as follows:

$$\frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} = \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{lz}^{l,m}(t,d)^T} \frac{\partial^e \mathbf{lz}^{l,m}(t,d)}{\partial \mathbf{a}^{u'}(t-d)^T} \quad \textbf{(20)}$$

The first term on the right of Eq. 20 is the derivative of the net input function, which is the identity matrix if the net input is the standard summation. The second term is the derivative of the weight function, which is the corresponding weight matrix if the weight function is the standard dot product. Therefore, the right side of Eq. 20 becomes simply a weight matrix for LDDN networks.

The second term in Eq. 19 is the same as the first term on the right of Eq. 20. It is the derivative of the net input function. The third term in Eq. 19 is the same as the second term on the right of Eq. 20. It is the derivative of the weight function.

The final term that we need to compute is the last term in Eq. 19, which is the explicit derivative of the network outputs with respect to the weights and biases in the network. One element of that matrix can be written

$$\frac{\partial^e a_k^u(t)}{\partial iw_{i,j}^{m,l}(d)} = \sum_{q=1}^{S} \frac{\partial^e a_k^u(t)}{\partial n_q^m(t)} \frac{\partial^e n_q^m(t)}{\partial iz_q^{m,l}(t,d)} \frac{\partial^e iz_q^{m,l}(t,d)}{\partial iw_{i,j}^{m,l}(d)} \tag{21}$$

The first term in this summation is an element of the sensitivity matrix, which is computed using Eq. 17. The second term is the derivative of the net input, and the third term is the derivative of the weight function. (We have made the assumption here that the net input function operates on each element individually.) Eq. 21 is the equation for an input weight. Layer weights and biases would have similar equations.

This completes the RTRL algorithm for networks that can be represented in the GLDDN framework. The main steps of the algorithm are Eqs. 11 and 16, where the components of Eq. 16 are computed using Eqs. 20 and 21. Computer code can be written from these equations, with modules for weight functions, net input functions and transfer functions added as needed. Each module should define the function response, as well as its derivative. The overall framework is independent of the particular form of these modules.

4 Conclusions

There are generally two different approaches for writing software to train dynamic networks. In the first approach, you use the general concepts of BPTT or RTRL to derive the dynamic backpropagation equations for a specific network architecture; each architecture has its own code. The second standard approach is to use some particular canonical form for dynamic networks. Such canonical forms can represent arbitrary dynamic networks. You can implement a dynamic backpropagation algorithm for the canonical form, and then you can transform the specific architecture in question into the canonical form in order to use the software.

In this paper we have taken a different approach. We have developed a framework that encompasses a very general class of network architectures, and then we have presented a set of equations that can be used to compute the gradients for any network that fits within that framework. This makes it possible to write software that is applicable to arbitrarily connected networks, without having to convert complex architectures into a standard canonical form.

Dynamic networks have applications in a wide variety of areas, and such disparate applications have lead to diverse network architectures. The ability to test new architectures quickly, without having to write problem-specific code or to convert each architecture into a canonical form, will enable the more rapid spread of dynamic neural networks into new fields.

References

1. M. Hagan, H. Demuth, O. De Jesús, "An Introduction to the Use of Neural Networks in Control Systems," invited paper, International Journal of Robust and Nonlinear Control, Vol. 12, No. 11 (2002) pp. 959-985.

2. J. Roman, and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, vol. 2, (1996) pp. 454-460.
3. J. Feng, C. K. Tse, F. C. M. Lau, "A neural-network-based channel-equalization strategy for chaos-based communication systems," IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, vol. 50, no. 7 (2003) pp. 954-957.
4. I. Kamwa, R. Grondin, V. K. Sood, C. Gagnon, V. T. Nguyen, J. Mereb, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," IEEE Transactions on Instrumentation and Measurement, vol. 45, no. 2, (1996) pp. 657-664.
5. Jayadeva and S. A. Rahman, "A neural network with O(N) neurons for ranking N numbers in O(1/N) time," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 51, no. 10, (2004) pp. 2044-2051.
6. G. Chengyu, and K. Danai, "Fault diagnosis of the IFAC Benchmark Problem with a model-based recurrent neural network," in Proceedings of the 1999 IEEE International Conference on Control Applications, vol. 2, (1999) pp. 1755-1760.
7. A. J. Robinson, "An application of recurrent nets to phone probability estimation," in IEEE Transactions on Neural Networks, vol. 5, no. 2 (1994).
8. L. R. Medsker, and L. C. Jain, Recurrent neural networks: design and applications, Boca Raton, FL: CRC Press (2000).
9. P. Gianluca, D. Przybylski, B. Rost, and P. Baldi, "Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles," in Proteins: Structure, Function, and Genetics, vol. 47, no. 2 , (2002) pp. 228-235.
10. P. J. Werbos, "Backpropagation through time: What it is and how to do it," Proceedings of the IEEE, vol. 78, (1990) pp. 1550–1560.
11. R. J. Williams, and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," Neural Computation, vol. 1, (1989) pp. 270–280.
12. O. De Jesús, and M. Hagan, "Backpropagation Algorithms for a Broad Class of Dynamic Networks," IEEE Transactions on Neural Networks, Vol. 18, No. 1 (2007) pp. 14 -27.
13. O. De Jesús, Training General Dynamic Neural Networks, Doctoral Dissertation, Oklahoma State University, Stillwater OK, (2002).
14. K. S. Narendra, and A. M. Parthasrathy, Identification and control for dynamic systems using neural networks, IEEE Transactions on Neural Networks, Vol. 1, No. 1 (1990) pp. 4-27.
15. 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, Gori, M., and Giles, C.L., eds., Springer Verlag (1998).
16. G. Dreyfus, and Y. Idan, "The Canonical Form of Nonlinear Discrete-Time Models," Neural Computation 10, 133–164 (1998).
17. A. C. Tsoi, and A. Back, "Discrete time recurrent neural network architectures: A unifying review," Neurocomputing 15 (1997) 183-223.
18. L. Personnaz, and G. Dreyfus, "Comment on 'Discrete-time recurrent neural network architectures: A unifying review'," Neurocomputing 20 (1998) 325-331.
19. 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," Proceedings of the IEEE, vol. 86, no. 11 (1998) pp. 2259 - 2277.
20. P. Campolucci, A. Marchegiani, A. Uncini, and F. Piazza, "Signal-Flow-Graph Derivation of On-line Gradient Learning Algorithms," Proceedings of International Conference on Neural Networks ICNN'97 (1997) pp.1884-1889.

Neural Networks and Associative Memories