Journal of Applied Intelligence 3, 91–105 (1993)
© 1993 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Discrete Time Neural Networks

ERIC A. WAN
Department of Electrical Engineering, Stanford University, Stanford, CA 94305-4005

Abstract. Traditional feedforward neural networks are static structures that simply map input to output. To better reflect the dynamics in the biological system, time dependency is incorporated into the network by using Finite Impulse Response (FIR) linear filters to model the processes of axonal transport, synaptic modulation, and charge dissipation. While a constructive proof gives a theoretical equivalence between the class of problems solvable by the FIR model and the static structure, certain practical and computational advantages exist for the FIR model. Adaptation of the network is achieved through an efficient gradient descent algorithm, which is shown to be a temporal generalization of the popular backpropagation algorithm for static networks. Applications of the network are discussed with a detailed example of using the network for time series prediction.

Key words: Neural networks, discrete time, temporal backpropagation, finite impulse response, time series, prediction

1. Introduction

In the past few years there has been a tremendous resurgence of activity in the field of neural networks. Today, neural networks have become much more than buzz words, with applications in industry ranging from character recognition and process control, to stock market portfolio management. Much of the success of neural networks can be traced to the development of a simple gradient descent technique, called backpropagation, used for training multilayered feedforward networks [1,2]. While the field has seen many advances since the introduction of this algorithm, backpropagation still remains the most popular method to date. For completeness, we review in Section 2 the backpropagation algorithm and the basic neural network architecture.

In spite of the success of neural networks, the standard model of the neuron itself remains naively simplistic. The pragmatist reduces the entire synaptic process to that of a single multiplication. All dynamics involving action potential transmission, charge dissipation through the cell membrane, and spike generation are conveniently dismissed. In Section 3.1, we look to biological neurons to motivate a more accurate model. What results is a network of discrete time adaptive filters [3], which captures the salient features of the real neuron. The architecture of the Discrete Time Neural Network is presented in Section 3.2; various interpretations of how the network uses time to perform its computations are provided in Section 4.

While biologically motivated, our end goal is still to produce a practical network that can be trained and applied to a wide range of problems. In Section 5, we present an efficient training algorithm called *temporal backpropagation*. The algorithm is shown to be a vector generalization of the familiar static backpropagation algorithm in which a symmetry exists between the forward propagation of states in time and the backward propagation of error terms. Given the training algorithm, it is possible to use the network in a variety of applications involving inherently temporal data. Section 6 focuses on the problem of nonlinear time series prediction. An example is given for the long-term prediction of laser intensity pulsations driven in a chaotic state.

2. Background: Static Networks and Backpropagation

2.1. Network Model

The traditional model of the neuron is shown in Figure 1a. The output of the neuron is simplistically taken to be a nonlinear function of the weighted sum of its inputs. Mathematically this is expressed as

$$y = \sum_i w_i x_i \tag{1}$$

$$out = f[y], \tag{2}$$

where x_i are the inputs to the neuron and w_i are the synpatic weights (x_0 is often fixed at 1 to form a bias to the neuron). It is most common to use a *sigmoid* for the nonlinearity. This function has the property that for small input the output stays near zero, while for increasing input levels the output quickly saturates. A graph of the sigmoid is shown in Figure 1b. The sigmoid was chosen to roughly model the thresholding properties of a real neuron.

A neural network results by arranging the basic neuron model into various configurations. In a Hopfield Network, for example, the output of each neuron has a connection to the input of all neurons in the network, including a self-feedback connection [4]. The most popular configuration, however, is the multilayered feedforward network. The structure, illustrated in Figure 2, consists of layers of neurons in which the output of a neuron in a given layer feeds all neurons in the next layer. No feedback loops exist in the structure. A complex mapping results from the input of the first layer to the outputs of the last layer. Given our current model of the neuron, this mapping is static; there are no internal dynamics. Nevertheless, the feedforward network is an extremely powerful tool for computation. It has been shown [5–7] that given only two layers of weights and a sufficient number of internal neurons, any well-behaved continuous function can be approximated to an arbitrary accuracy.

2.2. Training and Backpropagation

The specific mapping associated with a layered neural network is determined by the synaptic weight values. Training a neural network entails finding a set of weights that produce the desired mapping. In most cases, the exact mapping itself is not known in advance. Only specific instances of input/output pairs may be available for training the network. In speech recognition, for example, the input to the network may be a sampled voice waveform and the desired output may be a word classification. In this case, the actual multidimensional mapping from input to output is unknown, yet many specific input/output pairs

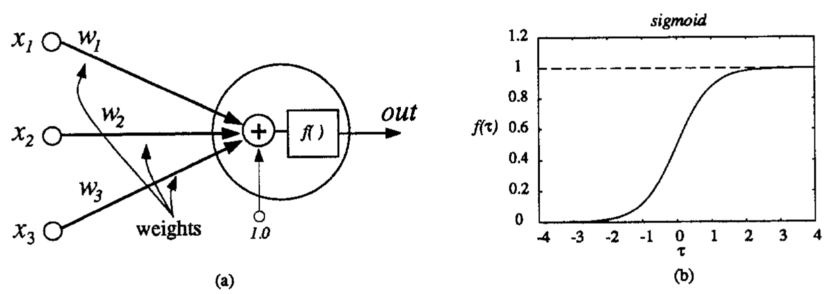


Fig. 1. (a) The static model of a neuron performs a weighted sum on its input signals and passes the result through a nonlinearity. (b) The nonlinearity is characterized by a sigmoid function, which is a continuous representation of the thresholding performed in a real neuron.

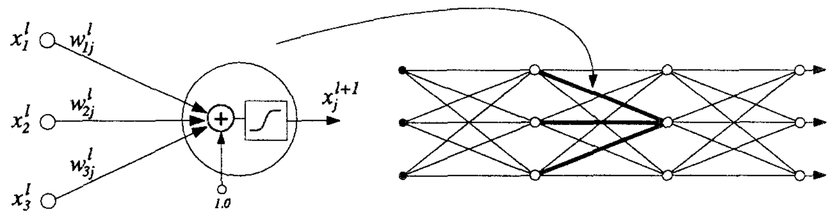


Fig. 2. Static Feedforward Network: the output of a neuron in a given layer acts as an input to neurons in the next layer. In the illustration, each line represents a synaptic connection. No feedback connections exist.

can be provided as a training set to the network.

Given a few definitions, we can express the training process mathematically. For an input $\mathbf{x}^0$ to the network, let the associated output be written as $O[W, \mathbf{x}^0]$, where W represents the set of all weights in the network. Given a training set of inputs with desired outputs, $\mathbf{d}$, the objective can then be written as minimizing over W the cost function

$$\begin{aligned} E(W) &= \sum_{k=1}^{K} \| \mathbf{d}(k) - O(W, \mathbf{x}^0(k)) \|^2 \\ &= \sum_{k=1}^{K} e^2(k), \end{aligned} \tag{3}$$

where the sum is taken over all K samples in the training set. A simple squared Euclidean metric is chosen as the error evaluation.

With this formulation, we see that training a neural network amounts to a standard optimization problem. While many optimization methods exist, the most practical approach in this case is to use a stochastic gradient descent algorithm. After each presentation of a sample from the training set, the weights are adapted according to

$$W(k+1) = W(k) - \mu\hat{\nabla}(k), \tag{4}$$

where $\hat{\nabla}(k) = \dfrac{\partial e^2(k)}{\partial W(k)}$ is the instantaneous error gradient, and μ is the learning rate. After multiple passes through the training set, the weights should converge such that the original cost function $E(W)$ is minimized.[1] This completes the training phase for the network. The capability of the network to then produce the correct output for patterns that were not present in the training set is referred to as *generalization*.

The weight update in Equation (4) is written compactly using W to represent the set of all weights in the network. In reality, this represents a set of scalar equations, one for each weight in the network:

$$w_{ij}^l(k+1) = w_{ij}^l(k) - \mu\frac{\partial e^2(k)}{\partial w_{ij}^l(k)}. \tag{5}$$

Subscripts are used to specify w_{ij}^l as the weight from neuron i in layer l which connects to neuron j in the next layer. Backpropagation is nothing more than an efficient algorithm for finding the terms $\dfrac{\partial e^2(k)}{\partial w_{ij}^l(k)}$. Without derivation, we will summarize the algorithm. Equation (5) can be written as

$$w_{ij}^l(k+1) = w_{ij}^l(k) - \mu\, \delta_j^{l+1}(k) \cdot x_i^l(k) \tag{6}$$

where x_i^l is the output of neuron i in layer l. To calculate the values of δ in a network with L layers, a recursive formula is applied in which previously calculated δ terms are *backpropagated* through connecting synapses:

$$r\delta_j^l(k) = \begin{cases} -2e_j(k)f_j'(k) & l = L \\ f_j'(k) \cdot \sum_m \delta_m^{l+1}(k) \cdot w_{jm}^{l+1} & 1 \le l \le L-1, \end{cases} \tag{7}$$

where $e_j(k)$ is the error at the output node j for the kth training sample, and $f_j'(k)$ is the derivative of the nonlinear function for that node. Thus δ for a given neuron is found by summing the next-layer δ's multiplied by the weights of the synapses fed by this neuron. This is best visualized graphically, as illustrated in Figure 3.

The success of the backpropagation is due to several desirable characteristics of the algorithm. The gradient calculations are efficient in the sense that computations only grow linearly with the size of the network. In addition, there exists a symmetry between the forward propagation of

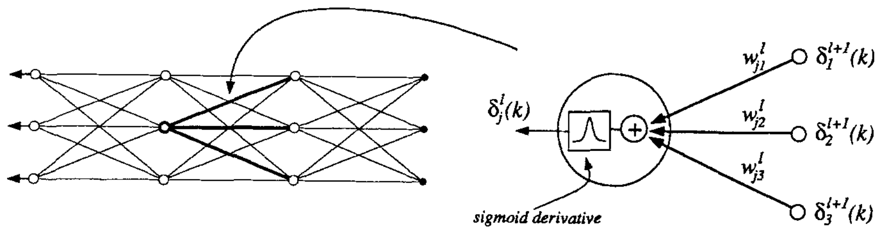


Fig. 3. Backpropagation. Delta terms are multiplied by synaptic connections to form the deltas for the previous layer. The process is applied layer by layer working backward through the network.

neuron outputs and the backward propagation of δ terms. All calculations are *locally distributed* with information flowing bidirectionally through synaptic connections. For additional coverage of the general network architecture and the backpropagation algorithm, the reader may refer to [1,8].

3. Discrete Time Model

3.1. Biological Motivation

Consider the cortical neuron illustrated in Figure 4. Starting with the axon, signals are transmitted as series of spike waveforms, called action potentials, which propagate as traveling waves. These signals terminate at a synaptic cleft located on the dendrite or soma (cell body) of the receiving neuron. The synaptic connection between neurons acts as a conductance modulator, either inhibitory or excitory, resulting in a postsynaptic potential. The potential then passively migrates according to a drift-diffusion equation along the cell membrane. All such incoming signals are spatially summed as they converge to the axon hillock (region where the axon sprouts from the cell body). At this point, provided a sufficient buildup of charge is maintained, a complicated dynamic process initiates, causing a series of outgoing spike trains to be generated.

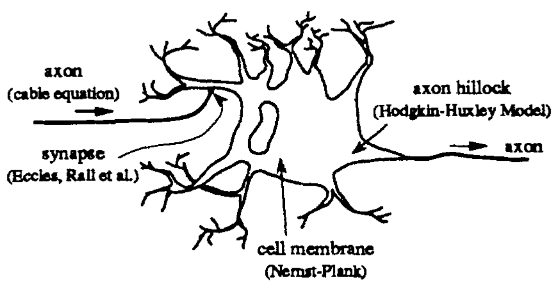


Fig. 4. The biological neuron illustrated here is divided into four main components: 1) signal transmission along the axon, 2) interneuron connectivity at the synapse, 3) passive charge dissipation in the cell membrane, and 4) spike generation in the axon hillock. Reference books on this subject include [9–11].

Our original engineering model, on the other hand, reduces axonal transport, synaptic modulation, and charge dissipation to a single multiplication, wx. Spike generation is modeled as a simple nonlinearity, f, in which the output represents the overall average firing rate rather than the true spike waveform. In spite of this, we do not wish to imply that the model is necessarily flawed. The goals of the engineer differ from those of the neurobiologist. Simplifications must be made in order to synthesize an artificial system that can be applied to practical problems. However, by reviewing the biological system, it is possible to extract an improved model that remains relatively simple, yet still retains the more salient features of the biological neuron.

Assuming time-invariance and removing all spatial dimensions, we can better approximate the processes associated with the axon, synapse, and cell membrane with the linear system

$$y(t) = \int_0^t h_c(\tau)x(t-\tau)d\tau, \tag{8}$$

where the input $x(t)$ is the action potential along the incoming axon and $y(t)$ represents the final resulting signal entering the neuron's axon hillock. The impulse response from input to output is completely characterized by $h_c(t)$. For computational purposes we sample at intervals of ΔT and discretize[2]

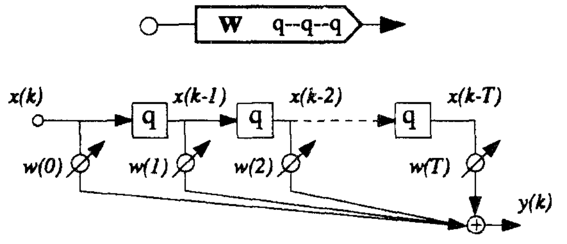


Fig. 5. FIR Filter Model: A tapped delay line shows the functional model of the Finite Impulse Response "synapse" (q represents a unit delay operator).

$$y(k\Delta T) = \sum_{n=0}^{k} h_d(n\Delta T)x(k\Delta T - n\Delta T). \qquad (9)$$

For clarification we drop the ΔT and assume that the impulse response is of finite duration. This yields

$$y(k) = \sum_{n=0}^{N} w(n)x(k - n), \qquad (10)$$

where we have substituted the truncated series $w(n)$ for $h_d(n\Delta T)$. This is a standard discrete time Finite Impulse Response (FIR) linear filter with $w(n)$ being the filter coefficients.[3] One may view this representation as a Markov model of signal transmission. A block diagram of the filter shown as a tapped delay line is drawn in Figure 5.

3.2. Model Definition

Continuing with notation, past samples of the input can be represented as the vector

$$\mathbf{x}(k) = [x(k), x(k - 1), \ldots x(k - N)]. \qquad (11)$$

Similarly, we form a weight vector for the filter coefficients

$$\mathbf{w} = [w(0), w(1), \ldots w(N)]. \qquad (12)$$

This allows us to express the operation of the filter by a vector dot product, $\mathbf{w} \cdot \mathbf{x}(k)$, where time relations are now implicit in the notation.

We can now complete the construction of the neuron by replacing the static synaptic weight in our original model by the FIR structure (Figure 6a). While the FIR filter was motivated by *several* physiological processes, for simplicity we will refer to this element as an FIR synapse. Mathematically, we define the complete FIR model of the neuron as

$$y(k) = \sum_i \mathbf{w}_i \cdot \mathbf{x}_i(k) \qquad (13)$$

$$out(k) = f[y(k)]. \qquad (14)$$

The sigmoid nonlinearity has been retained to represent the generation of the outgoing spike train. However, it is no longer necessary to view this output as simply a value for the cumulative average firing rate. The output is now a function of time and can be thought of as a sampled waveform.

Note the similarities in appearance between these equations and those of the static model in Equation (1). Notationally, scalars are replaced by vectors and multiplications by vector products. The convolution operation of the synapse is implicit in the definition. As we will show, these simple analogies will carry throughout subsequent derivations.

Finally, the temporal model of the neuron is used to construct a feedforward discrete time network as shown in Figure 6b. All interconnections are modeled with FIR filters. While similar

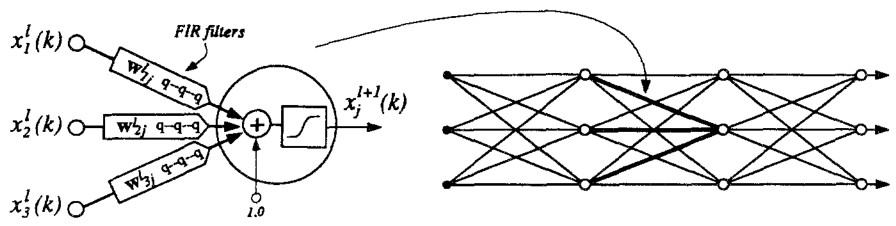


Fig. 6. (a) In the temporal model of the neuron, input signals are passed through synaptic filters. The sum of the filtered inputs is passed through a sigmoid function to form the output of the neuron. (b) In the feedforward network, all connections are modeled as FIR filters.

in appearance to our standard feedforward network, this structure can no longer be characterized as a simple static mapping. Dynamics are internal to the structure with outputs dependent on past values of the input.

4. Computational Interpretations

In the static network we saw that a single input maps to a single output:

$$x \rightarrow out. \tag{15}$$

For the discrete time network, a sampled input waveform gets mapped to a sampled output waveform:

$$x(k) \rightarrow out(k). \tag{16}$$

Since each neuron is composed of FIR filters, and the network contains no internal feedback connections, the network as a whole can be considered FIR. That is to say, for a finite input stimulus to the network, the output will be of finite duration. Equivalently, this implies that the network has a finite memory and processes only a windowed history of past inputs. Mathematically we can write the output as a function of a fixed number of past samples:

$$out(k) = g(x(k), x(k-1), \ldots, x(k-M)), \tag{17}$$

where both x and *out* can be either scalars or vectors. The output may be thought of as a path along an N-dimensional manifold in which time sampling and the order of the filters dictate the dimensionality.

A better understanding of how signals flow through the network can be achieved if we start at the neuron level. The neurons in the first layer act on the raw input data. Each passes a weighted sum from a finite window of past data through a sigmoidal nonlinearity. The output of each of these neurons acts as the preprocessed input to the second-layer neurons. Since all connections are modeled as tapped delay lines, the second-layer neurons window a history of the signals coming from the previous layer. This successive windowing results in a given input sample having a greater and greater time interval of influence from layer to layer.

Additional insight may be gained if we start from the output layer of the network and attempt to *unfold* the network in time. The general startegy is to remove all time delays by expanding the network into a larger static structure. As an example, consider the very simple network shown in Figure 7a. The network consists of three layers with a single output neuron and two neurons at each hidden layer. All connections are made by second-order (two tap) synapses. Thus while there are only 10 synapses in the network, there are actually a total of 30 variable filter coefficients. Starting at the last layer, each tap delay is interpreted as a "virtual neuron" whose input is delayed the appropriate number of time steps. A tap delay is then "removed" by replicating the previous layers of the network and delaying the input to the network accordingly. This is shown in Figure 7b. The process is then continued backward through each layer until all delays have been removed. The final unfolded network is shown in Figure 7c.

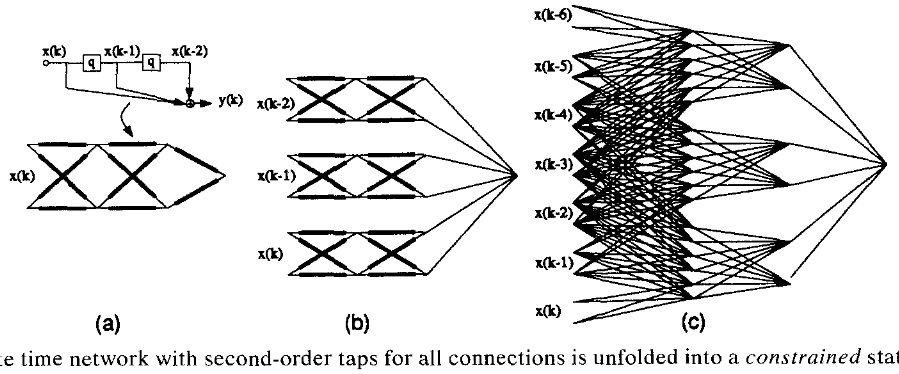


Fig. 7. A discrete time network with second-order taps for all connections is unfolded into a *constrained* static network. The original structure has 30 variable filter coefficients, while the resulting network has 150 static synapses. (a) original network; (b) expansion of the network; (c) final unfolded network.

Table 1. Discrete time network vs. static equivalent

DT Network Dimension Nodes†	Order‡	Variable Parameters	Static Equivalent
$2\times2\times2\times1$	2:2:2	30	150
$5\times5\times5\times5$	10:10:10	605	36,355
$3\times3\times3$	9:9	180	990
$3\times3\times3\times3$	9:9:9	270	9,990
$3\times3\times3\times3\times3$	9:9:9:9	360	99,990
3^n	9^{n-1}	$(n-1)90$	$10^n - 10$

†Number of Inputs × Hidden Neurons × Outputs.
‡Order of FIR synapses in each layer.
The relative size of the discrete time network to the equivalent static structure is compared. The static network grows geometrically in size with the discrete time network.

This method produces an equivalent *constrained* static structure where the time dependencies have been made external to the network itself by increasing the time window seen at the input. Notice that whereas there were initially 30 filter coefficients, the equivalent unfolded structure now has 150 static synapses. This can be seen as a result of redundancies in the static weights. In fact, the size of the equivalent static network grows *geometrically* with the number of layers and tap delays (see Table 1). In light of this, one can view the discrete time network as a compact representation of a larger static network with imposed symmetries. This is really not that surprising. All recursive algorithms, for example, can be *flattened* into equivalent, though less efficient, nonrecursive formulations. As we will see, when training the network, the more compact representation offers several inherent advantages.

5. Adaptation: Temporal Backpropagation

Let us recast the training process in terms of the discrete time network. For a given input sequence $\mathbf{x}^0(k)$, the network produces an output sequence $O[W, \mathbf{x}^0(k)]$, where W now represents the set of all synaptic filters in the network. For now, assume that at each instance in time, a desired output $\mathbf{d}(k)$ is provided. We define the instantaneous error $e^2(k)$ as the squared Euclidean distance between the network output and the desired response:

$$\begin{aligned} e^2(k) &= \| \mathbf{d}(k) - O(W, \mathbf{x}^0(k)) \|^2 \\ &= \sum_i [d_i(k) - o_i(k)]^2, \end{aligned} \tag{18}$$

where o_i is the scalar value of an individual output neuron and d_i is the corresponding desired response. The objective of training corresponds to minimizing over W the sum of the instantaneous errors over time:

$$E(W) = \sum_k e^2(k). \tag{19}$$

This is identical to the cost function in Equation (3). However, we are now minimizing an error between two sequences in time, rather than random samples in a training set. An implication of this is that a desired response need not be supplied at *all* increments of time. It is the overall sequence of outputs that is important. At any given increment the desired output may not be known, in which case the instantaneous error metric can be taken as zero.

As with backpropagation for static networks, a first-order gradient descent algorithm will be used as the optimization procedure. The basic philosophy of the approach is to iteratively adapt the weights in the direction of the negative gradient. We can expand the total gradient with respect to a vector of synaptic filter coefficients as the sum of instantaneous gradients:

$$\frac{dE}{d\mathbf{w}^l_{ij}} = \sum_k \frac{\partial e^2(k)}{\partial \mathbf{w}^l_{ij}}. \tag{20}$$

Subscripts are again included so that $\mathbf{w}^l_{ij}$ specifies the synaptic filter connecting the output of neuron i in layer l to the input of neuron j in the next layer. By taking each term in the expansion as an unbiased instantaneous estimate of the gradient, we form the stochastic gradient descent algorithm:

$$\mathbf{w}^l_{ij}(k+1) = \mathbf{w}^l_{ij}(k) - \mu \frac{\partial e^2(k)}{\partial \mathbf{w}^l_{ij}(k)}. \tag{21}$$

This is the analogous to the scalar update in Equation (5). Unfortunately, continued derivation of the gradient terms leads to an unacceptable algorithm. Without explicitly going into the math (see [12]), we can deduce the nature of the calculations by using arguments from the previous section. We start by unfolding the structure in time to produce an equivalent constrained static network. Instantaneous gradient terms can then be calculated using standard backpropagation where the input to the network has been appropriately windowed. Since the static equivalent contains "duplicated" weights, the individual gradient terms must be carefully recombined to get the *total* instantaneous gradient for each unique filter coefficient. The problem with this approach is that locally distributed processing is lost as global bookkeeping becomes necessary to keep track of all terms. The more layers in the network, the more complicated the bookkeeping becomes. A simple recursive formula cannot be found as was possible with backpropagation for static networks. The most serious drawback to this approach, however, is with regards to computational complexity of the algorithm. The number of overall calculations is proportional to the size of the equivalent static network and hence grows geometrically with the size of the FIR network (see Table 1). This computational explosion would then be reflected in the time necessary to train the structure.

5.1. Temporal Backpropagation

A more attractive algorithm can be derived if we approach the problem from a slightly different perspective. In order to proceed, it becomes necessary to introduce a bit of notation. Using additional subscripts, we define the synpatic filter connecting neuron i in layer l to the jth neuron in the next layer as

$$y_{ij}^{l+1}(k) = \mathbf{w}_{ij}^l \cdot \mathbf{x}_i^l(k), \tag{22}$$

where $\mathbf{w}_{ij}^l = [w_{i,j}^l(0), w_{i,j}^l(1), \ldots w_{ij}^l(T^l)]$ specifies the coefficients for the connecting synaptic filter and $\mathbf{x}_i^l(k) = [x_i^l(k), x_i^l(k-1), \ldots x_i^l(k-T^l)]$ is the vector of delayed states along the synapse.

The *total* input to the jth neuron in layer l at time k is specified as

$$\begin{aligned} y_j^l(k) &= \sum_{i=1}^{N_{l-1}} y_{ij}^l(k) \\ &= \sum_{i=1}^{N_{l-1}} \mathbf{w}_{ij}^{l-1} \cdot \mathbf{x}_i^{l-1}(k). \end{aligned} \tag{23}$$

The sum in the equation is taken over all N_l neurons in the layer. Finally, the output x^l for the neuron is taken to be a nonlinear sigmoidal function of its input sum:

$$x_j^l(k) = f(y_j^l(k)). \tag{24}$$

This notation completely defines the structure of the network. Note we can take $x_j^0(k)$ to specify an external input to the network while $x_j^L(k)$ specifies an output for an L layer network. Values of $x_j^l(k)$ for $1 \leq l \leq L$ specify internal states within the network. Nothing fundamentally new has been introduced here, only subscripts necessary to specify individual terms.

Proceeding with our original intention, we can now derive the improved training algorithm. In Equation (20), the total error gradient was expanded into a sum of instantaneous gradients. Consider the following alternative:

$$\frac{\partial E}{\partial \mathbf{w}_{ij}^l} = \sum_k \frac{\partial E}{\partial y_j^{l+1}(k)} \cdot \frac{\partial y_j^{l+1}(k)}{\partial \mathbf{w}_{ij}^l}. \tag{25}$$

We may interpret $\partial E/\partial y_j^{l+1}(k)$ as the change in the total squared error over all time due to a change in the input to a neuron at a single instant of time. Note that

$$\frac{\partial E}{\partial y_j^{l+1}(k)} \cdot \frac{\partial y_j^{l+1}(k)}{\partial \mathbf{w}_{ij}^l} \neq \frac{\partial e^2(k)}{\partial \mathbf{w}_{ij}^l}. \tag{26}$$

Only the sums over all k are equivalent. Using this expansion, the stochastic algorithm follows:

$$\mathbf{w}_{ij}^l(k+1) = \mathbf{w}_{ij}^l(k) - \mu \frac{\partial E}{\partial y_j^{l+1}(k)} \cdot \frac{\partial y_j^{l+1}(k)}{\partial \mathbf{w}_{ij}^l}. \tag{27}$$

From Equation (23), $\partial y_j^l(k)/\partial \mathbf{w}_{ij}^{l-1} = \mathbf{x}_i^{l-1}(k)$ for all layers in the network. Defining $\partial E/\partial y_j^l(k) \equiv \delta_j^l(k)$ allows us to rewrite Equation (27) in the more familiar notational form

$$\mathbf{w}_{ij}^l(k+1) = \mathbf{w}_{ij}^l(k) - \mu \delta_j^{l+1}(k) \cdot \mathbf{x}_i^l(k), \tag{28}$$

which will be shown to hold for any layer in the network. To complete the derivation, an explicit formula for $\delta_j^l(k)$ must be found. For the output layer we have simply

$$\begin{aligned} \delta_j^L(k) \equiv \frac{\partial E}{\partial y_j^L(k)} &= \frac{\partial e^2(k)}{\partial y_j^L(k)} \\ &= -2e_j(k)f'(y_j^L(k)), \end{aligned} \tag{29}$$

where $e_j(k)$ is the error at an output node. For a hidden layer, we again use the chain rule, expanding over all time and all N_{l+1} inputs $y^{l+1}(k)$ in the next layer:

$$\delta_j^l(k) \equiv \frac{\partial E}{\partial y_j^l(k)}$$

$$= \sum_{m=1}^{N_{l+l}} \sum_t \frac{\partial E}{\partial y_m^{l+1}(t)} \frac{\partial y_m^{l+1}(t)}{\partial y_j^l(k)}$$

$$= \sum_{m=1}^{N_{l+l}} \sum_t \delta_m^{l+1}(t) \frac{\partial y_m^{l+1}(t)}{\partial y_j^l(k)}$$

$$= f'(y_j^l(k)) \sum_{m=1}^{N_{l+l}} \sum_t \delta_m^{l+1}(t) \frac{\partial y_{jm}^{l+1}(t)}{\partial x_j^l(k)}. \tag{30}$$

But we recall

$$y_{jm}^{l+1}(t) = \sum_{k'=0}^{T^l} w_{jm}^l(k')x_j^l(t-k'). \tag{31}$$

Thus

$$\frac{\partial y_{jm}^{l+1}(t)}{\partial x_j^l(k)} = \begin{cases} w_{jm}^l(t-k) \text{ for } 0 \le t-k \le T^l \\ \quad 0 \qquad \text{otherwise,} \end{cases} \tag{32}$$

which now yields

$$\delta_j^l(k) = f'(y_j^l(k) \sum_{m=1}^{N_{l+l}} \sum_{t=k}^{T_l+k} \delta_m^{l+1}(t)w_{jm}^l(t-k)$$

$$= f'(y_j^l(k)) \sum_{m=1}^{N_{l+l}} \sum_{n=0}^{T^l} \delta_m^{l+1}(k+n)w_{jm}^l(n)$$

$$= f'(y_j^l(k)) \cdot \sum_{m=1}^{N_{l+l}} \boldsymbol{\delta}_m^{l+1}(k) \cdot \mathbf{w}_{jm}^l, \tag{33}$$

where we have defined

$$\boldsymbol{\delta}_m^l(k) = [\delta_m^l(k), \delta_m^l(k+1), \ldots \delta_m^l(k+T^{l-1})]. \tag{34}$$

Summarizing, the complete adaptation algorithm, called *temporal backpropagation,* can be expressed as follows:

$$\mathbf{w}_{ij}^l(k+1) = \mathbf{w}_{ij}^l(k) - \mu\delta_j^{l+1}(k) \cdot \mathbf{x}_i^l(k) \tag{35}$$

We immediately observe that these equations are seen as the vector generalization of the already familiar backpropagation algorithm (see Equations (6) and (7)). In fact, by replacing the vectors **x**, **w**, and **δ** by scalars, the above equations reduce to precisely the standard backpropagation algorithm for static networks. Differences in the *temporal* version are a matter of implicit time relations and filtering operations. To calculate $\delta_j^l(k)$ for a given neuron we *filter* the δ's from the next layer backwards through the FIR synapses for which the given neuron feeds (see Figure 8). Thus δ's are formed not by simply taking weighted sums, but by backward filtering. For each new input and desired response vector, the forward filters are incremented one time step and the backward filters one time step.

Thus we see that by manipulating the terms used to accumulate the error gradients, we can preserve the symmetry between the forward propagation of states and the backward propagation of error terms. Parallel distributed processing is maintained. Furthermore, the algorithm overcomes the computational complexity encountered when the instantaneous gradient was used. The number of operations only grows *linearly* with the number of layers and synapses in the network. This savings comes as a consequence of the efficient recursive formulation. Each unique coefficient enters into the calculation only once, in contrast to the redundant use of terms as in the first case. Additional implementation issues regarding a *noncausality* condition are discussed in Appendix A.

Both the instantaneous gradient method and the temporal backpropagation algorithm are based on gradient descent. They are not, however, functionally equivalent. All gradient derivations assumed that the weight parameters were fixed. During actual adaptation this is clearly not a valid assumption. Minor discrepancies in performance arise due to differences in the timing at which weights are adjusted relative to the calculation of the error gradients. It is possible to show that the weight update in Equation (27) for

$$\delta_j^l(k) = \begin{cases} -2e_j(k)f'(y_j^L(k)) & l = \quad L \\ f'(y_j^l(k)) \cdot \sum_{m=1}^{N_{l+l}} \boldsymbol{\delta}_m^{l+1}(k) \cdot \mathbf{w}_{jm}^l & 1 \le l \le L-1. \end{cases} \tag{36}$$

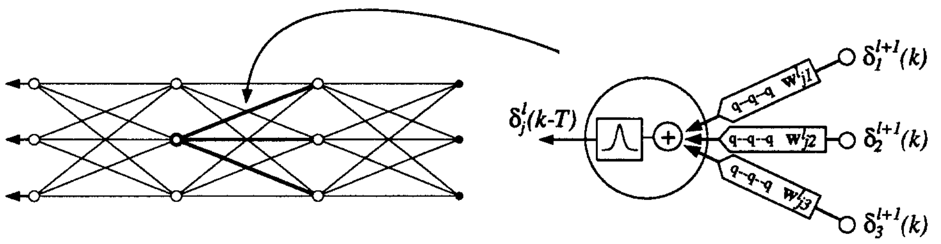


Fig. 8. In temporal backpropagation, delta terms are *filtered* through synaptic connections to form the deltas for the previous layer. The process is applied layer by layer working backward through the network.

temporal backpropagation still uses an *unbiased* gradient estimate. Figure 9 shows averaged learning curves for the two algorithms used on a two-layer network[4] modeling an unknown nonlinear system. For "small" learning rates, differences in the learning characteristics are negligible. Mathematically the algorithms become equivalent as $\mu \to 0$. Alternatively, terms may be accumulated to find the total gradient for batch mode adaptation. In this case the algorithms are functionally identical apart from specific implementation and computational differences. Note, even for this simple example where combinatorial problems were not an issue, temporal backpropagation resulted in roughly a 40% reduction in computer simulation time.

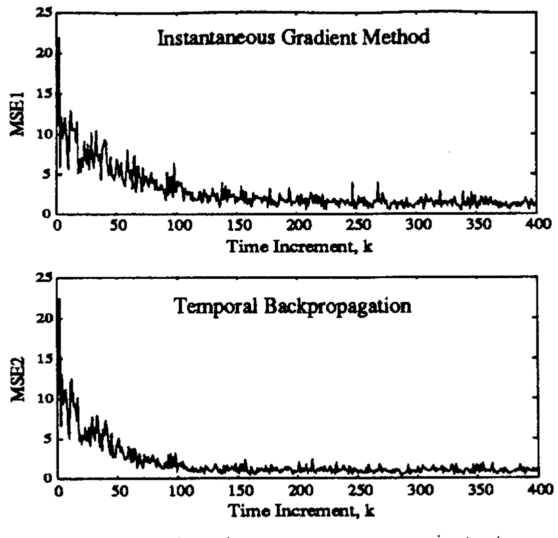


Fig. 9. Averaged learning curves compare instantaneous gradient descent to temporal backpropagation. A simple two-layer network (one input, one output, and five hidden units with 4-tap FIR filters for each synapse) was used to model the nonlinear system $y(k) = x(k) + 2\sin(x(k-1) + \sum_{j=1}^{5} x(k-j))/(x(k)+1)$. Random inputs, $x(k)$, were used to excite the plant. Differences in the learning characteristics appear negligible ($\mu = 0.05$).

6. Applications

The discrete time network is clearly most applicable to problems that are inherently temporal in nature. This covers a wide range of areas, such as adaptive nonlinear filtering, channel equalization, noise canceling, beamforming, system identification, controls, etc. Recently, Waibel et al. [13] used a similar structure called a Time-Delay Neural Network (TDNN) to perform phoneme recognition. The relationship between the structures and the computational advantages of applying temporal backpropagation to TDNNs are discussed in [14, 15]. In the next section we present some recent results on using the network for time series prediction.

6.1. Time Series Prediction

The goal of time series prediction or *forecasting* can be stated succinctly as follows. Given some sequence $y(1), y(2), \ldots, y(N)$ up to time N, find the continuation of the sequence $y(N+1)$, $y(N+2)\ldots$ The series may arise from the sampling of a continuous time system, and may be either stochastic or deterministic in origin. The standard approach involves constructing an underlying model that gives rise to the sequence. In the oldest and most studied method, which dates back to Yule [16], a linear autoregression (AR) is fit to the data:

$$\begin{aligned} y(k) &= \sum_{n=1}^{T} a(n)y(k-n) + e(k) \\ &= \hat{y}(k) + e(k). \end{aligned} \tag{37}$$

This AR model forms $y(k)$ as a weighted sum of past values of the sequence ($\hat{y}(k)$ is the single-step prediction for $y(k)$). The error term $e(k)$ is often assumed to a be a white-noise process for analysis in a stochastic framework. A weighting on past samples of $e(k)$ may also be present, in which case the model is referred to as Autoregressive Moving Average (ARMA). ARMA models are more relevant to system identification where only a single-step prediction is necessary.[5] These models can also be generalized for the multidimensional case. Several texts provide complete coverage of linear prediction and system modeling [17,18].

Neural networks may be used to extend the linear model to form a nonlinear prediction scheme. The basic form $y(k) = \hat{y}(k) + e(k)$ is retained; however, the estimate $\hat{y}(k)$ is taken as the output O of the neural network driven by past values of the sequence:

$$y(k) = \hat{y}(k) + e(k) = O(W, y(k-1)) + e(k) \tag{38}$$

and corresponds to a nonlinear autoregression, since the discrete time network has a finite memory of past input samples. Note this model is equally applicable for both scalar and vector sequences. Training ensues by adapting the network to minimize the squared single-step prediction error $e^2(k)$ over the training sequence up to time N. Using the cost function in Equation (18), this can be written as

$$\min_W E(W) = \min_W \sum_{k=0}^{N} \| y(k) - O(W, y(k-1)) \|^2, \tag{39}$$

and the network may be trained with temporal backpropagation. Figure 10a illustrates the basic training setup. Once the network is trained, it can be used to perform long-term prediction by taking the estimate $\hat{y}(k)$ and feeding it back as input to the network (i.e., $\hat{y}(k) = O(W, \hat{y}(k-1))$ for $k > N$). This closed-loop process can be iterated forward in time to get predictions as far into the future as desired. The prediction scheme is shown in Figure 10b.

Before giving a concrete example, a few words should be said about the theoretical motivation for using such a structure. In a stochastic framework, minimization of the above cost function in Equation 39 for large N is equivalent to minimizing the *expected* squared error between the desired response and the network output. Several authors [19,20] have shown that the optimal mapping for the network is given as the *conditional expectation* of the desired response given the input. Making the proper substitutions in the context of time series prediction shows that the optimal network mapping O^* is given as

$$O^* = E[y(k) | y(k-1), y(k-2) \ldots y(k-T)], \tag{40}$$

i.e., the conditional expectation of $y(k)$ given the past samples. This corresponds to the optimal single-step prediction. These arguments imply that the neural network is capable of implementing the *best* predictor. Note, however, they do not imply that adaptation will necessarily achieve the optimal for a given structure and training sequence.

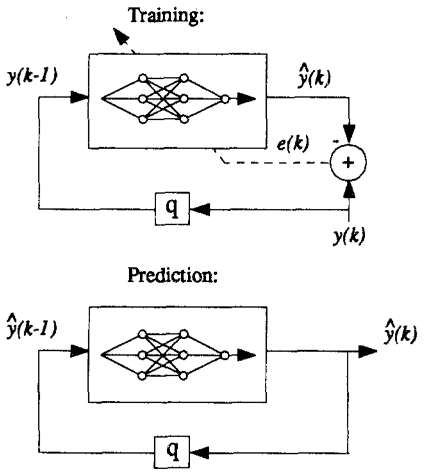


Fig. 10. Network Prediction Configuration: The single-step prediction $\hat{y}(k)$ is taken as the output of the network driven by previous samples of the sequence $y(k)$. During training, the single-step prediction error, $e(k) = y(k) - \hat{y}(k)$, is minimized using temporal backpropagation. Feeding the estimate $\hat{y}(k)$ back forms a closed-loop process used for iterated long-term prediction. (a) Basic training setup; (b) prediction scheme.

6.1.1. An Example in Chaos Prediction. The plot in Figure 11 shows the chaotic intensity pulsations of an NH_3 laser measured in a laboratory experiment.[6] These data were distributed as part of *The Santa Fe Institute Time Series Prediction and Analysis Competition* with the motivation of

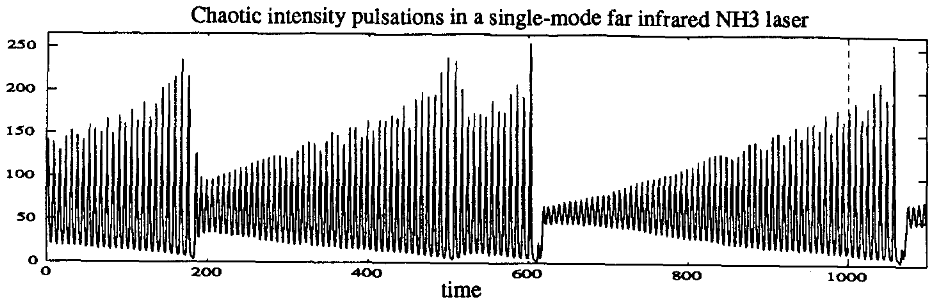


Fig. 11. 1100 time points of chaotic laser data.

making rigorous comparisons between various techniques [22,23]. For the laser data, 1000 samples of the sequence were provided. The goal was to predict the next 100 samples. During the course of the competition, the physical background of the data set, as well as the 100 point continuation, was withheld to avoid biasing the final prediction results.

A discrete time network ($1 \times 12 \times 12 \times 1$ nodes with 15:5:5 order taps) was used to learn the training set. Actual training occurred on the first 900 points of the series with the remaining 100 used for testing prior to the availability of the actual series continuation. The 100-step prediction achieved using the discrete time network is shown in Figure 12 along with the actual series continuation for comparison. It is important to emphasize that this prediction was made based on only the past 1000 samples. True values of the series for time past 1000 were not provided to the network nor were they even available when the predictions were submitted. As can be seen, the prediction is remarkably accurate, with only some slight phase degradation. A prediction based on a 25th-order linear autoregression is also shown to emphasize the differences from traditional linear methods. Other submissions to the competition included methods of k-d trees, piecewise linear interpolation, low-pass embedding, SVD, nearest neighbors, Wiener filters, as well as standard recurrent and feedforward neural networks. As reported in [22,23], the discrete time network described here outperformed all other methods on this data set.

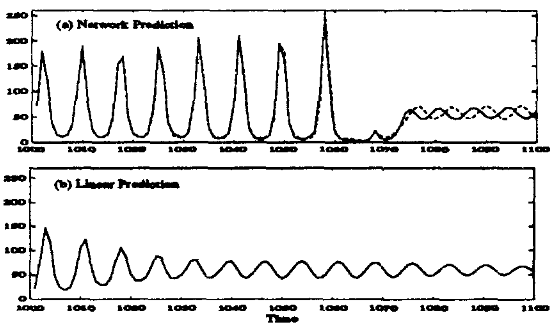


Fig. 12. Time series predictions: (a) 100-point closed-loop neural network prediction. The prediction is based only on the supplied 1000 points. The dashed line corresponds to actual series continuation. (b) 100-point iterated prediction based on a 25th-order linear autoregression. Regression coefficients were solved using a standard least squares method.

7. Summary

In this paper a temporal model of the neuron was derived from biological considerations. A discrete time FIR filter was used to model the processes of axonal transport, synaptic modulation, and charge dissipation in the cell membrane. Neurons were then arranged in a feedforward configuration to form the network. While biologically motivated, we should emphasize that we do not claim any biological plausibility. The end goal was to construct a network with certain desirable characteristics. This includes the ability to process temporal data and perform mappings on sequences. A training algorithm called *temporal backpropagation* was derived for the structure. The algorithm, which could be considered a vector generalization of standard backpropagation, allowed for efficient calculation of gradient terms and preserved local distributed processing.

An example from nonlinear time series prediction was used to demonstrate the capabilities of the network. While the results were extremely promising, it is clear that further examples and testing are necessary. In addition, more academic issues need to be addressed, such as training time, convergence, and extending the concept of *generalization* to the learning of sequences.

Appendix A: Noncausality of Temporal Backpropagation

The specification of the temporal backpropagation algorithm in Equation (36) hides an important subtlety encountered when actually implementing the algorithm. Careful inspection reveals that the calculations for the $\delta_j^{l-1}(k)$'s are in fact noncausal. The source of this noncausal filtering can be seen by considering the definition of $\delta_j^{l-1}(k) = \partial E/\partial y_j^l(k)$. Since it takes time for the output of any internal neuron to completely propagate through the network, the change in the *total* error due to a change in an internal state is a function of future values within the network. Since the network is FIR, only a finite number of future states must be considered.

The exact time reference taken for adaptation purposes is not important. Making the system causal becomes nothing more than a standard engineering task of reindexing. This can be accomplished in a number of different ways by adding a finite number of simple delay operators at various locations within the network. One possible solution follows if we require all weight vectors to be adapted based on only the current error $e^2(k)$ and past values of the error. Given this constraint, $\delta^L(k)$ for the last layer remains unaffected. For the next layer back, however, terms are only available to calculate

$$\delta_j^{L-1}(k-T) = f'(y_j^{L-1}(k-T) \cdot \sum_{m=1}^{Nl+1} \boldsymbol{\delta}_m^L(k-T) \cdot \mathbf{w}_{jm}^{L-1} \tag{41}$$

based on the requirement that $\boldsymbol{\delta}(k - T) = [\delta(k - T), \delta(k - T + 1), \ldots \delta(k)]$ be composed of only current and past terms.

Since at time k, only the time-shifted value $\delta^{L-1}(k - T)$ can be computed, the states $\mathbf{x}^{L-2}(k - T)$ must be stored so that the product of the two may be formed to adapt the synapses in the layer. Continuing one more layer back, the time shift is simply twice as long. Rewriting the algorithm in this causal form yields

$$\mathbf{w}_{ij}^{L-1-n}(k + 1) = \mathbf{w}_{ij}^{L-1-n}(k) - \mu\delta_j^{L-n}(k - nT) \cdot \mathbf{x}_i^{L-1-n}(k - nT) \tag{42}$$

$$\delta_j^{L-n}(k - nT) = \begin{cases} -2e_j(k)f'(y_j(k)) & n = 0 \\ f'(y_j^{L-n}(k - nT)) \cdot \sum\limits_{m=1}^{Nl+l} \boldsymbol{\delta}_m^{L+1-n}(k - nT) \cdot \mathbf{w}_{jm}^{L-n} & 1 \le n \le L - 1. \end{cases} \tag{43}$$

While less aesthetically pleasing than the earlier equations, they differ only in terms of a change of indices. Summarizing then, we propagate the delta terms backward continuously *without* delay. However, by definition this forces the internal values of deltas to be shifted in time. Thus one might buffer the states $\mathbf{x}(k)$ appropriately to form the proper terms for adaptation. Added storage delays are necessary only for the states $\mathbf{x}(k)$. The backward propagation of delta terms requires no added delays and is still symmetric to the forward propagation.

For simplicity in the above equations, it was assumed that the order of each synaptic filter, T, was the same in each layer. This is clearly not necessary. If the order is different for each layer in the network, we simply replace nT in Equations (42) and (43) by

$$nT \leftrightarrow \sum_{l=L-n}^{L-1} T^l. \tag{44}$$

For the general case, let T^l_{ij} be the order of the synaptic filter connecting neuron i in layer l to neuron j in the next layer. Then for arbitrary synaptic filter order we have

$$nT \leftrightarrow \sum_{l=L-n}^{L-1} \max_{ij}\{T^l_{ij}\}. \quad (45)$$

The basic rule is that the time shift for the delta associated with a given neuron is equal to the total number of tap delays along the longest path to the output of the network.

Acknowledgment

This work was sponsored in part by NASA under contract NCA2-680, EPRI under contract RP 8010-13, and Navy under contract N00014-86-K-0718.

Notes

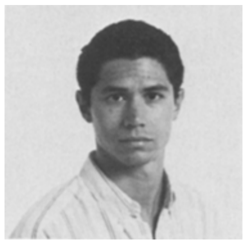
1. Only a local minimum can be guaranteed.
2. Note that h_d is not simply the sampled version of the continuous impulse response h_c. However, since we are only interested in the form of the discrete representation and not a *specific* discretization of a continuous filter, the details of this conversion are not important.
3. The Infinite Impulse Response (IIR) filter has the form $y(k) = \sum_{n=0}^{N} a(n)x(k-n) + \sum_{m=1}^{M} b(m)y(k-m)$ and will not be considered in this paper.
4. One input, one output, and five hidden units with 4-tap FIR filters for each synapse ($\mu = .05$).
5. The general Autoregressive Moving Average Exogenous Input (ARMAX) system identification model is written as $y(k) = \sum_{n=0}^{T_n} a(n)x(k-n) + \sum_{m=1}^{T_m} b(m)y(k-m) + \sum_{l=1}^{T_l} c(l)e(k-1)$. For long-term time series prediction, only the autoregression applies.
6. "Measurements were made on an 81.5-micron 14NH3 cw (FIR) laser, pumped optically by the P(13) line of an N20 laser via the vibrational aQ(8,7) NH3 transition"—U. Huebner [21].

References

1. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, MIT Press: Cambridge, MA, 1986.
2. P. Werbos, "Beyond regression: new tools for prediction and analysis in the behavioral sciences," Dissertation, Harvard University, Cambridge, MA, 1974.
3. B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Prentice Hall: Englewood Cliffs, NJ, 1985.
4. D. Tank and J. Hopfield, "Simple 'neural' optimization networks: an A/D converter, signal decision circuit, and a linear programming circuit," *IEEE Trans. Circuits Systems*, vol. 33, pp. 533–541, 1986.
5. K. Hornik, M. Stinchombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.
6. G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. of Control Signals Systems*, vol. 2, no. 4, pp. 303–314, 1989.
7. B. Irie and S. Miyake, "Capabilities of three-layered perceptrons," *Proc. IEEE Second Int. Conf. Neural Networks*, vol. I, San Diego, CA, July 1988, pp. 641–647.
8. B. Widrow and M. Lehr, "30 years of adaptive neural networks: perceptron, madaline, and backpropagation," *Proc. IEEE*, vol. 78, no. 9, pp. 1415–1442, September 1990.
9. D. Junge, *Nerve and Muscle Excitation*, Third Edition, Sinauer Associates, Inc., Sunderland, MA, 1991.
10. C. Koch and I. Segev (eds.), *Methods in Neuronal Modeling: From Synapses to Networks*, MIT Press: Cambridge, MA, 1989.
11. R. MacGregor and E. Lewis, *Neural Modeling*, Plenum Press: New York, 1977.
12. E. Wan, "Temporal backpropagation for FIR neural networks," *Int. Joint Conf. Neural Networks*, vol. 1, San Diego, CA, 1990, pp. 575–580.
13. A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Trans. Acoust. Speech Signal Processing*, vol. 37, no. 3, pp. 328–339, 1989.
14. E. Wan, "Finite Impulse Response neural networks for autoregressive time series prediction," to appear in *Proc. NATO Adv. Workshop Time Series Prediction Analysis*, edited by A. Weigend and N. Gershenfeld, Addison-Wesley: Reading, MA, 1993.
15. E. Wan, "Temporal backpropagation: an efficient algorithm for finite impulse response neural networks," *Proc. 1990 Connectionist Models Summer School*, Morgan Kaufmann: San Mateo, CA, 1990, pp. 131–140.
16. G. Yule, "On a method of investigating periodicity in disturbed series with special reference to Wolfer's sunspot numbers," *Philos. Trans. R. Soc. Lond. A*, vol. 226, pp. 267–298, 1927.
17. L. Ljung, *System Identification: Theory for the User*, Prentice-Hall: Englewood Cliffs, NJ, 1987.
18. S. Wei and W. William, *Time Series Analysis: Univariate and Multivariate Methods*, Addison-Wesley: Reading, MA, 1990.
19. H. White, "Learning in artificial neural networks: a

statistical perspective," *Neural Computations,* vol. 1, pp. 425–464, 1989.

20. E. Wan, "Neural network classification: a Bayesian interpretation," *IEEE Trans. Neural Networks,* vol. 1, no. 4, pp. 303–305, 1990.
21. U. Huebner, N. B. Abrahm, and C. O. Weiss, "Dimension and entropies of a chaotic intensity pulsations in a single-mode far-unfrared NH3 laser," *Phys. Rev.* A, vol. 40, pp. 6345, 1989.
22. *Proc. NATO Adv. Res. Workshop on Time Series Prediction and Analysis,* Santa Fe, New Mexico, May 14–17, 1992, edited by A. Weigend and N. Gershenfeld, Addison-Wesley: Reading, MA, 1993.
23. A. Weigend and N. Gershenfeld, "Results of the time series prediction competition at the Santa Fe Institute," in *Neural Information Processing Syst.,* Denver, CO, December 1992.



Eric A. Wan received his B.S. (1987) and M.S.E.E. (1988) from Stanford University. Currently, he is at Stanford University, where he is working toward the Ph.D. degree.

He is a member of Sigma XI, Tau Beta Pi, Phi Beta Kappa, and IEEE. His research interests include neural networks, control systems, and digital signal processing.