

# Backpropagation Through Time: What It Does and How to Do It

PAUL J. WERBOS

*Backpropagation is now the most widely used tool in the field of artificial neural networks. At the core of backpropagation is a method for calculating derivatives exactly and efficiently in any large system made up of elementary subsystems or calculations which are represented by known, differentiable functions; thus, backpropagation has many applications which do not involve neural networks as such.*

*This paper first reviews basic backpropagation, a simple method which is now being widely used in areas like pattern recognition and fault diagnosis. Next, it presents the basic equations for backpropagation through time, and discusses applications to areas like pattern recognition involving dynamic systems, systems identification, and control. Finally, it describes further extensions of this method, to deal with systems other than neural networks, systems involving simultaneous equations or true recurrent networks, and other practical issues which arise with this method. Pseudocode is provided to clarify the algorithms. The chain rule for ordered derivatives—the theorem which underlies backpropagation—is briefly discussed.*

## I. INTRODUCTION

Backpropagation through time is a very powerful tool, with applications to pattern recognition, dynamic modeling, sensitivity analysis, and the control of systems over time, among others. It can be applied to neural networks, to econometric models, to fuzzy logic structures, to fluid dynamics models, and to almost any system built up from elementary subsystems or calculations. The one serious constraint is that the elementary subsystems must be represented by functions known to the user, functions which are both continuous and differentiable (i.e., possess derivatives). For example, the first practical application of backpropagation was for estimating a dynamic model to predict nationalism and social communications in 1974 [1].

Unfortunately, the most general formulation of backpropagation can only be used by those who are willing to work out the mathematics of their particular application. This paper will mainly describe a simpler version of backpropagation, which can be translated into computer code and applied directly by neural network users.

Section II will review the simplest and most widely used form of backpropagation, which may be called "basic back-

propagation." The concepts here will already be familiar to those who have read the paper by Rumelhart, Hinton, and Williams [2] in the seminal book *Parallel Distributed Processing*, which played a pivotal role in the development of the field. (That book also acknowledged the prior work of Parker [3] and Le Cun [4], and the pivotal role of Charles Smith of the Systems Development Foundation.) This section will use new notation which adds a bit of generality and makes it easier to go on to complex applications in a rigorous manner. (The need for new notation may seem unnecessary to some, but for those who have to *apply* backpropagation to complex systems, it is essential.)

Section III will use the same notation to describe backpropagation through time. Backpropagation through time has been applied to concrete problems by a number of authors, including, at least, Watrous and Shastri [5], Sawai and Waibel *et al.* [6], Nguyen and Widrow [7], Jordan [8], Kawato [9], Elman and Zipser, Narendra [10], and myself [1], [11], [12], [15]. Section IV will discuss what is missing in this simplified discussion, and how to do better.

At its core, backpropagation is simply an efficient and exact method for calculating all the derivatives of a single target quantity (such as pattern classification error) with respect to a large set of input quantities (such as the parameters or weights in a classification rule). Backpropagation through time extends this method so that it applies to dynamic systems. This allows one to calculate the derivatives needed when optimizing an iterative analysis procedure, a neural network with memory, or a control system which maximizes performance over time.

## II. BASIC BACKPROPAGATION

### A. The Supervised Learning Problem

Basic backpropagation is currently the most popular method for performing the supervised learning task, which is symbolized in Fig. 1.

In supervised learning, we try to adapt an artificial neural network so that its actual outputs ( $\hat{Y}$ ) come close to some target outputs ( $Y$ ) for a training set which contains  $T$  patterns. The goal is to adapt the parameters of the network so that it performs well for patterns from outside the training set.

The main use of supervised learning today lies in pattern

Manuscript received September 12, 1989; revised March 15, 1990.  
The author is with the National Science Foundation, 1800 G St.  
NW, Washington, DC 20550.  
IEEE Log Number 9039172.

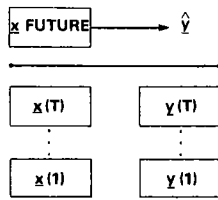


Fig. 1. Schematic of the supervised learning task.

recognition work. For example, suppose that we are trying to build a neural network which can learn to recognize handwritten ZIP codes. (AT&T has actually done this [13], although the details are beyond the scope of this paper.) We assume that we already have a camera and preprocessor which can digitize the image, locate the five digits, and provide a  $19 \times 20$  grid of ones and zeros representing the image of each digit. We want the neural network to input the  $19 \times 20$  image, and output a classification; for example, we might ask the network to output four binary digits which, taken together, identify which decimal digit is being observed.

Before adapting the parameters of the neural network, one must first obtain a training database of actual handwritten digits and correct classifications. Suppose, for example, that this database contains 2000 examples of handwritten digits. In that case,  $T = 2000$ . We may give each example a label  $t$  between 1 and 2000. For each sample  $t$ , we have a record of the input pattern and the correct classification. Each input pattern consists of 380 numbers, which may be viewed as a vector with 380 components; we may call this vector  $X(t)$ . The desired classification consists of four numbers, which may be treated as a vector  $Y(t)$ . The actual output of the network will be  $\hat{Y}(t)$ , which may differ from the desired output  $Y(t)$ , especially in the period before the network has been adapted. To solve the supervised learning problem, there are two steps:

- We must specify the "topology" (connections and equations) for a network which inputs  $X(t)$  and outputs a four-component vector  $\hat{Y}(t)$ , an approximation to  $Y(t)$ . The relation between the inputs and outputs must depend on a set of weights (parameters)  $W$  which can be adjusted.
- We must specify a "learning rule"—a procedure for adjusting the weights  $W$  so as to make the actual outputs  $\hat{Y}(t)$  approximate the desired outputs  $Y(t)$ .

Basic backpropagation is currently the most popular learning rule used in supervised learning. It is generally used with a very simple network design—to be described in the next section—but the same approach can be used with any network of differentiable functions, as will be discussed in Section IV.

Even when we use a simple network design, the vectors  $X(t)$  and  $Y(t)$  need not be made of ones and zeros. They can be made up of any values which the network is capable of inputting and outputting. Let us denote the components of  $X(t)$  as  $X_1(t) \cdots X_m(t)$  so that there are  $m$  inputs to the network. Let us denote the components of  $Y(t)$  as  $Y_1(t) \cdots Y_n(t)$  so that we have  $n$  outputs. Throughout this paper, the components of a vector will be represented by the same letter as the vector itself, in the same case; this convention turns out to be convenient because  $x(t)$  will represent a different vector, very closely related to  $X(t)$ .

Fig. 1 illustrates the supervised learning task in the gen-

eral case. Given a history of  $X(1) \cdots X(T)$  and  $Y(1) \cdots Y(T)$ , we want to find a mapping from  $X$  to  $Y$  which will perform well when we encounter new vectors  $X$  outside the training set. The index "t" may be interpreted either as a time index or as a pattern number index; however, this section will not assume that the order of patterns is meaningful.

### B. Simple Feedforward Networks

Before we specify a learning rule, we have to define exactly how the outputs of a neural net depend on its inputs and weights. In basic backpropagation, we assume the following logic:

$$x_i = X_i, \quad 1 \leq i \leq m \quad (1)$$

$$\text{net}_i = \sum_{j=1}^{i-1} W_{ij} x_j, \quad m < i \leq N + n \quad (2)$$

$$x_i = s(\text{net}_i), \quad m < i \leq N + n \quad (3)$$

$$Y_i = x_{i+N}, \quad 1 \leq i \leq n \quad (4)$$

where the function  $s$  in (3) is usually the following sigmoidal function:

$$s(z) = 1/(1 + e^{-z}) \quad (5)$$

and where  $N$  is a constant which can be any integer you choose as long as it is no less than  $m$ . The value of  $N + n$  decides how many neurons are in the network (if we include inputs as neurons). Intuitively,  $\text{net}_i$  represents the total level of voltage exciting a neuron, and  $x_i$  represents the intensity of the resulting output from the neuron. ( $x_i$  is sometimes called the "activation level" of the neuron.) It is conventional to assume that there is a threshold or constant weight  $W_{i0}$  added to the right side of (2); however, we can achieve the same effect by assuming that one of the inputs (such as  $X_m$ ) is always 1.

The significance of these equations is illustrated in Fig. 2. There are  $N + n$  circles, representing all of the neurons

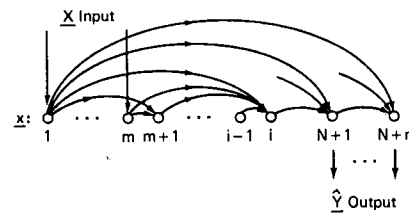


Fig. 2. Network design for basic backpropagation.

in the network, including the input neurons. The first  $m$  circles are really just copies of the inputs  $X_1 \cdots X_m$ ; they are included as part of the vector  $x$  only as a way of simplifying the notation. Every other neuron in the network—such as neuron number  $i$ , which calculates  $\text{net}_i$  and  $x_i$ —takes input from every cell which precedes it in the network. Even the last output cell, which generates  $\hat{Y}_n$ , takes input from other output cells, such as the one which outputs  $\hat{Y}_{n-1}$ .

In neural network terminology, this network is "fully connected" in the extreme. As a practical matter, it is usually desirable to limit the connections between neurons. This can be done by simply fixing some of the weights  $W_{ij}$  to zero so that they drop out of all calculations. For example, most researchers prefer to use "layered" networks, in which all

connection weights  $W_{ij}$  are zeroed out, except for those going from one "layer" (subset of neurons) to the next layer. In general, one may zero out as many or as few of the weights as one likes, based on one's understanding of individual applications. For those who first begin this work, it is conventional to define only three layers—an input layer, a "hidden" layer, and an output layer. This section will assume the full range of *allowed* connections, simply for the sake of generality.

In computer code, we could represent this network as a Fortran subroutine (assuming a Fortran which distinguishes upper case from lower case):

```

SUBROUTINE NET(X, W, x, Yhat)
  REAL X(m), W(N+n, N+n), x(N+n), Yhat(n), net
  INTEGER, i, j, m, n, N
  C First insert the inputs, as per equation (1)
  DO 1 i = 1, m
    1 x(i) = X(i)
  C Next implement (2) and (3) together for each value
  C of i
  DO 1000 i = m+1, N+n
  C calculate net, as a running sum, based on (2)
  net = 0
  DO 10 j = 1, i-1
    10 net = net + W(i, j)*x(j)
  C finally, calculate  $x_i$  based on (3) and (5)
  1000 x(i) = 1/(1+exp(-net))
  C Finally, copy over the outputs, as per (4)
  DO 2000 i = 1, n
  2000 Yhat(i) = x(i+N);

```

In the pseudocode, note that  $X$  and  $W$  are technically the inputs to the subroutine, while  $x$  and  $Yhat$  are the outputs.  $Yhat$  is usually regarded as "the" output of the network, but  $x$  may also have its uses outside of the subroutine proper, as will be seen in the next section.

### C. Adapting the Network: Approach

In basic backpropagation, we choose the weights  $W_{ij}$  so as to minimize square error over the training set:

$$E = \sum_{t=1}^T E(t) = \sum_{t=1}^T \sum_{i=1}^n (1/2)(\hat{Y}_i(t) - Y_i(t))^2. \quad (6)$$

This is simply a special case of the well-known method of least squares, used very often in statistics, econometrics, and engineering; the uniqueness of backpropagation lies in *how* this expression is minimized. The approach used here is illustrated in Fig. 3.

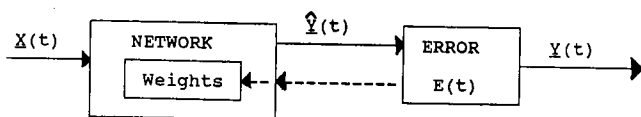


Fig. 3. Basic backpropagation (in pattern learning).

In basic backpropagation, we start with arbitrary values for the weights  $W$ . (It is usual to choose random numbers in the range from  $-0.1$  to  $0.1$ , but it may be better to guess the weights based on prior information, in cases where prior information is available.) Next, we calculate the outputs

$\hat{Y}(t)$  and the errors  $E(t)$  for that set of weights. Then we calculate the derivatives of  $E$  with respect to all of the weights; this is indicated by the dotted lines in Fig. 3. If increasing a given weight would lead to more error, we adjust that weight downwards. If increasing a weight leads to less error, we adjust it upwards. After adjusting all the weights up or down, we start all over, and keep on going through this process until the weights and the error settle down. (Some researchers iterate until the error is close to zero; however, if the number of training patterns exceeds the number of weights in the network—as recommended by studies on generalization—it may not be possible for the error to reach zero.) The uniqueness of backpropagation lies in the method used to calculate the derivatives exactly for all of the weights in only one pass through the system.

### D. Calculating Derivatives: Theoretical Background

Many papers on backpropagation suggest that we need only use the conventional chain rule for partial derivatives to calculate the derivatives of  $E$  with respect to all of the weights. Under certain conditions, this can be a rigorous approach, but its generality is limited, and it requires great care with the side conditions (which are rarely spelled out); calculations of this sort can easily become confused and erroneous when networks and applications grow complex. Even when using (7) below, it is a good idea to test one's gradient calculations using explicit perturbations in order to be sure that there is no bug in one's code.

When the idea of backpropagation was first presented to the Harvard faculty in 1972, they expressed legitimate concern about the validity of the rather complex calculations involved. To deal with this problem, I proved a new chain rule for *ordered* derivatives:

$$\frac{\partial^+ \text{TARGET}}{\partial z_i} = \frac{\partial \text{TARGET}}{\partial z_i} + \sum_{i>j} \frac{\partial^+ \text{TARGET}}{\partial z_j} * \frac{\partial z_j}{\partial z_i} \quad (7)$$

where the derivatives with the superscript represent *ordered* derivatives, and the derivatives without subscripts represent ordinary partial derivatives. This chain rule is valid only for *ordered* systems where the values to be calculated can be calculated one by one (if necessary) in the order  $z_1, z_2, \dots, z_n, \text{TARGET}$ . The simple partial derivatives represent the *direct* impact of  $z_i$  on  $z_j$  through the system equation which determines  $z_j$ . The ordered derivative represents the *total* impact of  $z_i$  on TARGET, accounting for both the *direct* and *indirect* effects. For example, suppose that we had a simple system governed by the following two equations, in order:

$$z_2 = 4 * z_1$$

$$z_3 = 3 * z_1 + 5 * z_2.$$

The "simple" partial derivative of  $z_3$  with respect to  $z_1$  (the *direct* effect) is 3; to calculate the simple effect, we *only* look at the equation which determines  $z_3$ . However, the ordered derivative of  $z_3$  with respect to  $z_1$  is 23 because of the indirect impact by way of  $z_2$ . The simple partial derivative measures what happens when we increase  $z_1$  (e.g., by 1, in this example) and assume that everything else (like  $z_2$ ) in the equation which determines  $z_3$  remains constant. The ordered derivative measures what happens when we increase  $z_1$ , and also recalculate all other quantities—like

$z_2$ —which are later than  $z_1$  in the causal ordering we impose on the system.

This chain rule provides a straightforward, plodding, "linear" recipe for how to calculate the derivatives of a given TARGET variable with respect to *all* of the inputs (and parameters) of an ordered differentiable system *in only one pass through the system*. This paper will not explain this chain rule in detail since lengthy tutorials have been published elsewhere [1], [11]. But there is one point worth noting: because we are calculating ordered derivatives of *one* target variable, we can use a simpler notation, a notation which works out to be easier to use in complex practical examples [11]. We can write the ordered derivative of the TARGET with respect to  $z_i$  as " $F_{-z_i}$ ," which may be described as "the feedback to  $z_i$ ." In basic backpropagation, the TARGET variable of interest is the error  $E$ . This changes the appearance of our chain rule in that case to

$$F_{-z_i} = \frac{\partial E}{\partial z_i} + \sum_{j>i} F_{-z_j} * \frac{\partial z_j}{\partial z_i} \quad (8)$$

For purposes of debugging, one can calculate the true value of any ordered derivative simply by perturbing  $z_i$  at the point in the program where  $z_i$  is calculated; this is particularly useful when applying backpropagation to a complex network of functions other than neural networks.

#### E. Adapting the Network: Equations

For a given set of weights  $W$ , it is easy to use (1)–(6) to calculate  $Y(t)$  and  $E(t)$  for each pattern  $t$ . The trick is in how we then calculate the derivatives.

Let us use the prefix " $F_{-}$ " to indicate the ordered derivative of  $E$  with respect to whatever variable the " $F_{-}$ " precedes. Thus, for example,

$$F_{-\hat{Y}}(t) = \frac{\partial E}{\partial \hat{Y}_i(t)} = \hat{Y}_i(t) - Y_i(t), \quad (9)$$

which follows simply by differentiating (6). By the chain rule for ordered derivatives as expressed in (8),

$$F_{-x_i}(t) = F_{-\hat{Y}_{i-n}}(t) + \sum_{j=i+1}^{N+n} W_{ji} * F_{-net_j}(t), \quad (10)$$

$$i = N + n, \dots, m + 1$$

$$F_{-net_i}(t) = s'(net_i) * F_{-x_i}(t), \quad i = N + n, \dots, m + 1 \quad (11)$$

$$F_{-W_{ij}} = \sum_{t=1}^T F_{-net_i}(t) * x_j(t) \quad (12)$$

where  $s'$  is the derivative of  $s(z)$  as defined in (5) and  $F_{-Y_k}$  is assumed to be zero for  $k \leq 0$ . Note how (10) requires us to run *backwards* through the network in order to calculate the derivatives, as illustrated in Fig. 4; this backwards prop-

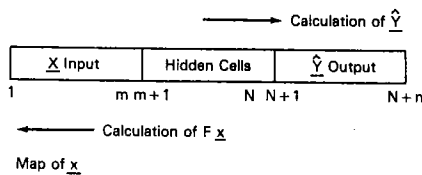


Fig. 4. Backwards flow of derivative calculation.

agation of information is what gives backpropagation its name. A little calculus and algebra, starting from (5), shows us that

$$s'(z) = s(z) * (1 - s(z)), \quad (13)$$

which we can use when we implement (11). Finally, to adapt the weights, the usual method is to set

$$\text{New } W_{ij} = W_{ij} - \text{learning\_rate} * F_{-W_{ij}} \quad (14)$$

where the *learning\_rate* is some small constant chosen on an ad hoc basis. (The usual procedure is to make it as large as possible, up to 1, until the error starts to diverge; however, there are more analytic procedures available [11].)

#### F. Adapting the Network: Code

The key part of basic backpropagation—(10)–(13)—may be coded up into a "dual" subroutine, as follows.

```

SUBROUTINE F_NET(F_Yhat, W, x, F_W)
REAL F_Yhat(n),W(N+n,N+n),x(N+n),
  F_W(N+n,N+n),F_net(N+n),F_x(N+n)
INTEGER i,j,n,m,N
C Initialize equation (10)
DO 1 i = 1,N
  1 F_x(i) = 0
DO 2 i = 1,n
  2 F_x(i+N)=F_Yhat(i)
C RUN THROUGH (10)–(12) AS A SET,
C FOR i RUNNING BACKWARDS
DO 1000 i=N+n,m+1,-1
C complete (10) for the current value of i */
DO 10 j = i+1,N+n
C modify "DO 10" if needed to be sure
C nothing is done if i=N+n
  10 F_x(i) = F_x(i)+W(j,i)*F_net(j)
C next implement (11), exploiting (13)
  F_net(i) = F_x(i)*x(i)*(1.-x(i))
C then implement (12) for the current
C value of i
DO 12 j = 1,i-1
  12 F_W(i,j)=F_net(i)*x(j)
1000 CONTINUE

```

Note that the array  $F_W$  is the only output of this subroutine.

Equation (14) represented "batch learning," in which weights are adjusted only after *all*  $T$  patterns are processed. It is more common to use pattern learning, in which the weights are continually updated after each observation. Pattern learning may be represented as follows:

```

C PATTERN LEARNING
DO 1000 pass_number=1,maximum_passes
DO 100 t = 1,T
  CALL NET(X(t), W, x, Yhat)
C Next Implement equation (9)
DO 9 i = 1,n
  9 F_Yhat(i)=Yhat(i)-Y(t,i)
C Next Implement (10)–(12)
  CALL F_NET(F_Yhat, W, x, F_W)
C Next Implement (14)
C Note how weights are updated
C within the "DO 100" loop.

```

```

DO 14 i = m+1, N+n
DO 14 j = 1, i-1
14 W(i, j) = W(i, j) -
learning_rate*
F_W(i, j)
100 CONTINUE
1000 CONTINUE

```

The key point here is that the weights  $W$  are adjusted in response to the *current* vector  $F_W$ , which only depends on the current pattern  $t$ ; the weights are adjusted after each pattern is processed. (In batch learning, by contrast, the weights are adjusted only after the "DO 100" loop is completed.)

In practice, `maximum_passes` is usually set to an enormous number; the loop is exited only when a test of convergence is passed, a test of error size or weight change which can be injected easily into the loop. True real-time learning is like pattern learning, but with only one pass through the data and no memory of earlier times  $t$ . (The equations above could be implemented easily enough as a real-time learning scheme; however, this will not be true for backpropagation through time.) The term "on-line learning" is sometimes used to represent a situation which could be pattern learning or could be real-time learning. Most people using basic backpropagation now use pattern learning rather than real-time learning because, with their data sets, many passes through the data are needed to ensure convergence of the weights.

The reader should be warned that I have not actually tested the code here. It is presented simply as a way of explaining more precisely the preceding ideas. The C implementations which I have worked with have been less transparent, and harder to debug, in part because of the absence of range checking in that language. It is often argued that people "who know what they are doing" do not need range checking and the like; however, people who think they never make mistakes should probably not be writing this kind of code. With neural network code, especially, good diagnostics and tests are very important because bugs can lead to slow convergence and oscillation—problems which are hard to track down, and are easily misattributed to the algorithm in use. If one *must* use a language without range checking, it is extremely important to maintain a version of the code which is highly transparent and safe, however inefficient it may be, for diagnostic purposes.

### III. BACKPROPAGATION THROUGH TIME

#### A. Background

Backpropagation through time—like basic backpropagation—is used most often in pattern recognition today. Therefore, this section will focus on such applications, using notation like that of the previous section. See Section IV for other applications.

In some applications—such as speech recognition or submarine detection—our classification at time  $t$  will be more accurate if we can account for what we saw at earlier times. *Even though* the training set still fits the same format as above, we want to use a more powerful class of networks to do the classification; we want the output of the network at time  $t$  to account for variables at earlier times (as in Fig. 5).

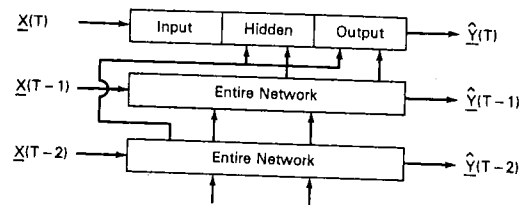


Fig. 5. Generalized network design with time lags.

The Introduction cited a number of examples where such "memory" of previous time periods is very important. For example, it is easier to recognize moving objects if our network accounts for *changes* in the scene from the time  $t - 1$  to time  $t$ , which requires memory of time  $t - 1$ . Many of the best pattern recognition algorithms involve a kind of "relaxation" approach where the representation of the world at time  $t$  is based on an *adjustment* of the representation at time  $t - 1$ ; this requires memory of the *internal* network variables for time  $t - 1$ . (Even Kalman filtering requires such a representation.)

#### B. Example of a Recurrent Network

Backpropagation can be applied to any system with a well-defined order of calculations, even if those calculations depend on past calculations within the network itself. For the sake of generality, I will show how this works for the network design shown in Fig. 5 where every neuron is potentially allowed to input values from *any* of the neurons at the two previous time periods (including, of course, the input neurons). To avoid excess clutter, Fig. 5 shows the hidden and output sections of the network (parallel to Fig. 2) only for time  $T$ , but they are present at other times as well. To translate this network into a mathematical system, we can simply replace (2) above by

$$\text{net}_i(t) = \sum_{j=1}^{i-1} W_{ij} x_j(t) + \sum_{j=1}^{N+n} W'_{ij} x_j(t-1) + \sum_{j=1}^{N+n} W''_{ij} x_j(t-2). \quad (15)$$

Again, we can simply fix some of the weights to be zero, if we so choose, in order to simplify the network. In most applications today, the  $W''$  weights are fixed to zero (i.e., erased from all formulas), and all the  $W'$  weights are fixed to zero as well, except for  $W'_{ii}$ . This is done in part for the sake of parsimony, and in part for historical reasons. (The "time-delay neural networks" of Watrous and Shastri [5] assumed that special case.) Here, I deliberately include extra terms for the sake of generality. I allow for the fact that *all* active neurons (neurons other than input neurons) can be allowed to input the outputs of *any* other neurons if there is a time lag in the connection. The weights  $W'$  and  $W''$  are the weights on those time-lagged connections between neurons. [Lags of more than two periods are also easy to manage; they are treated just as one would expect from seeing how we handle lag-two terms, as a special case of (7).]

These equations could be embodied in a subroutine:

```

SUBROUTINE NET2(X(t), W', W'', x(t-2),
x(t-1), x(t), Yhat),

```

which is programmed just like the subroutine NET, with the modifications one would expect from (15). The output arrays are  $x(t)$  and  $Yhat$ .

When we call this subroutine for the first time, at  $t = 1$ , we face a minor technical problem: there is no value for  $x(-1)$  or  $x(0)$ , both of which we need as inputs. In principle, we can use any values we wish to choose; the choice of  $x(-1)$  and  $x(0)$  is essentially part of the definition of our network. Most people simply set these vectors to zero, and argue that their network will start out with a blank slate in classifying whatever dynamic pattern is at hand, both in the training set and in later applications. (Statisticians have been known to treat these vectors as weights, in effect, to be adapted along with the other weights in the network. This works fine in the training set, but opens up questions of what to do when one applies the network to new data.)

In this section, I will assume that the data run from an initial time  $t = 1$  through to a final time  $t = T$ , which plays a crucial role in the derivative calculations. Section IV will show how this assumption can be relaxed somewhat.

### C. Adapting the Network: Equations

To calculate the derivatives of  $F_{W_{ij}}$ , we use the same equations as before, except that (10) is replaced by

$$F_{x_i}(t) = F_{\hat{Y}_{i-N}}(t) + \sum_{j=i+1}^{N+n} W_{ji} * F_{net_j}(t) + \sum_{j=m+1}^{N+n} W'_{ji} * F_{net_j}(t+1) + \sum_{j=m+1}^{N+n} W''_{ji} * F_{net_j}(t+2). \quad (16)$$

Once again, if one wants to fix the  $W''$  terms to zero, one can simply delete the rightmost term.

Notice that this equation makes it impossible for us to calculate  $F_{x_i}(t)$  and  $F_{net_i}(t)$  until after  $F_{net_j}(t+1)$  and  $F_{net_j}(t+2)$  are already known; therefore, we can only use this equation by proceeding *backwards in time*, calculating  $F_{net}$  for time  $T$ , and then working our way backwards to time 1.

To adapt this network, of course, we need to calculate  $F_{W'_{ij}}$  and  $F_{W''_{ij}}$  as well as  $F_{W_{ij}}$ :

$$F_{W'_{ij}} = \sum_{t=1}^T F_{net_i}(t+1) * x_j(t) \quad (17)$$

$$F_{W''_{ij}} = \sum_{t=1}^T F_{net_i}(t+2) * x_j(t). \quad (18)$$

In all of these calculations,  $F_{net}(T+1)$  and  $F_{net}(T+2)$  should be treated as zero. For programming convenience, I will later define quantities like  $F_{net'_i}(t) = F_{net_i}(t+1)$ , but this is purely a convenience; the subscript " $i$ " and the time argument are enough to identify which derivative is being represented. (In other words,  $net_i(t)$  represents a specific quantity  $z_j$  as in (8), and  $F_{net_i}(t)$  represents the ordered derivative of  $E$  with respect to that quantity.)

### D. Adapting the Network: Code

To fully understand the meaning and implications of these equations, it may help to run through a simple (hypothetical) implementation.

First, to calculate the derivatives, we need a new subroutine, dual to NET2.

```

SUBROUTINE F_NET2(F_Yhat, W, W', W'', x, F_net,
  F_net', F_net'', F_W, F_W', F_W'')
  REAL F_Yhat(n), W(N+n, N+n),
    W'(N+n, N+n), W''(N+n, N+n)
  REAL x(N+n), F_net(N+n), F_net'(N+n),
    F_net''(N+n)
  REAL F_W(N+n, N+n), F_W'(N+n, N+n),
    F_W''(N+n, N+n), F_x(N+n)
  INTEGER i, j, n, m, N
C Initialize equation (16)
  DO 1 i = 1, N
    1 F_x(i) = 0.
    DO 2 i = 1, n
      2 F_x(i+N) = F_Yhat(i)
C RUN THROUGH (16), (11), AND (12) AS A SET,
C RUNNING BACKWARDS
  DO 1000 i = N+n, m+1, -1
C first complete (16)
  DO 161 j = i+1, N+n
    161 F_x(i) = F_x(i) + W(j, i) * F_net(j)
  DO 162 j = m+1, N+n
    162 F_x(i) = F_x(i) + W'(j, i) * F_net'(j)
      + W''(j, i) * F_net''(j)
C next implement (11)
  F_net(i) = F_x(i) * x(i) * (1 - x(i))
C implement (12), (17), and (18)
  (as running sums)
  DO 12 j = 1, i-1
    12 F_W(i, j) = F_W(i, j)
      + F_net(i) * x(j)
  DO 1718 j = 1, N+n
    F_W'(i, j) = F_W'(i, j)
      + F_net'(i) * x(j)
    1718 F_W''(i, j) = F_W''(i, j)
      + F_net''(i) * x(j)
  1000 CONTINUE

  Notice that the last two DO loops have been set up to
  perform running sums, to simplify what follows.
  Finally, we may adapt the weights as follows, by batch
  learning, where I use the abbreviation  $x(i)$ , to represent the
  vector formed by  $x(i, j)$  across all  $j$ .

  REAL x(-1: T, N+n), Yhat(T, n)
  DATA x(0), x(-1), / (2*(N+n)) * 0.0/
  DO 1000 pass_number = 1, maximum_passes
C First calculate outputs and errors in
C a forward pass
  DO 100 t = 1, T
    100 CALL NET2(X(t), W, W', W'', x(t-2),
      x(t-1), x(t), Yhat(t))
C Initialize the running sums to 0 and
C set F_net(T), F_net(T+1) to 0
  DO 200 i = m+1, N+n
    F_net'(i) = 0.
    F_net''(i) = 0.
    DO 199 j = 1, N+n
      F_W(i, j) = 0.
      F_W'(i, j) = 0.
      199 F_W''(i, j) = 0.
  200 CONTINUE

```

