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 backpropagation." 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 current 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.

U.S. Government work not protected by U.S. copyright

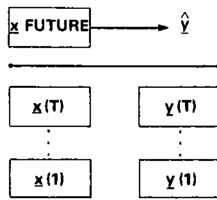

Fig. 1. Schematic of the supervised learning task.

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 19×20 grid of ones and zeros representing the image of each digit. We want the neural network to input the 19×20 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 $\boldsymbol{X}(t)$. The desired classification consists of four numbers, which may be treated as a vector $\boldsymbol{Y}(t)$. The actual output of the network will be $\hat{\boldsymbol{Y}}(t)$, which may differ from the desired output $\boldsymbol{Y}(t)$, 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 $\boldsymbol{X}(t)$ and outputs a four-component vector $\hat{\boldsymbol{Y}}(t)$, an approximation to $\boldsymbol{Y}(t)$. The relation between the inputs and outputs must depend on a set of weights (parameters) $\boldsymbol{W}$ which can be adjusted.
- We must specify a "learning rule"—a procedure for adjusting the weights $\boldsymbol{W}$ so as to make the actual outputs $\hat{\boldsymbol{Y}}(t)$ approximate the desired outputs $\boldsymbol{Y}(t)$.

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 $\boldsymbol{X}(t)$ and $\boldsymbol{Y}(t)$ 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 $\boldsymbol{X}(t)$ as $X_1(t) \cdots X_m(t)$ so that there are m inputs to the network. Let us denote the components of $\boldsymbol{Y}(t)$ 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 $\boldsymbol{x}(t)$ will represent a different vector, very closely related to $\boldsymbol{X}(t)$.

Fig. 1 illustrates the supervised learning task in the general case. Given a history of $\boldsymbol{X}(1) \cdots \boldsymbol{X}(T)$ and $\boldsymbol{Y}(1) \cdots \boldsymbol{Y}(T)$, we want to find a mapping from $\boldsymbol{X}$ to $\boldsymbol{Y}$ which will perform well when we encounter *new* vectors $\boldsymbol{X}$ 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:

$$x_i = X_i, \qquad 1 \le i \le m \tag{1}$$

$$\text{net}_i = \sum_{j=1}^{i-1} W_{ij} x_j, \qquad m < i \le N + n \tag{2}$$

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

$$\hat{Y}_i = x_{i+N}, \qquad 1 \le i \le n \tag{4}$$

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

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

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, 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_{io} 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

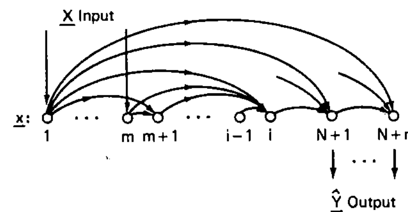

Fig. 2. Network design for basic backpropagation.

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 $\boldsymbol{x}$ only as a way of simplifying the notation. *Every* other neuron in the network—such as neuron number i, which calculates 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_i 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:

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

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.

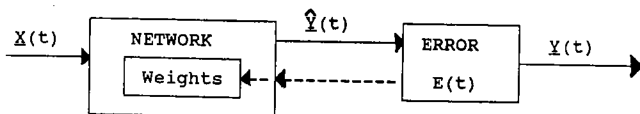

Fig. 3. Basic backpropagation (in pattern learning).

In basic backpropagation, we start with arbitrary values for the weights $\boldsymbol{W}$. (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:

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

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, $z_2, \cdots, z_n$, 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

$$F_z_i = \frac{\partial E}{\partial z_i} + \sum_{j>i} F_z_j * \frac{\partial z_j}{\partial z_i}. \tag{8}$$

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 $\boldsymbol{W}$, it is easy to use (1)-(6) to calculate $\boldsymbol{Y}(t)$ 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,

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

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

$$F_x_i(t) = F_\hat{Y}_{i-N}(t) + \sum_{j=i+1}^{N+n} W_{ji} * F_net_j(t), \quad i = N+n, \cdots, m+1 \tag{10}$$

$$F_net_i(t) = s'(net_i) * F_x_i(t), \quad i = N+n, \cdots, m+1 \tag{11}$$

$$F_W_{ij} = \sum_{t=1}^{T} F_net_i(t) * x_j(t) \tag{12}$$

where s' is the derivative of $s(z)$ as defined in (5) and F_Y_k is assumed to be zero for $k \le 0$. Note how (10) requires us to run *backwards* through the network in order to calculate the derivatives, as illustrated in Fig. 4; this backwards propagation of information is what gives backpropagation its name. A little calculus and algebra, starting from (5), shows us that

$$s'(z) = s(z) * (1 - s(z)), \tag{13}$$

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

$$\text{New } W_{ij} = W_{ij} - \text{learning_rate} * F_W_{ij} \tag{14}$$

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].)

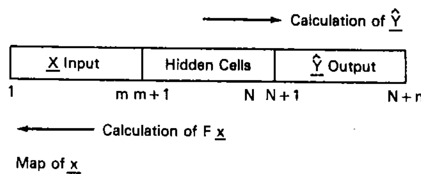

Fig. 4. Backwards flow of derivative calculation.

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).

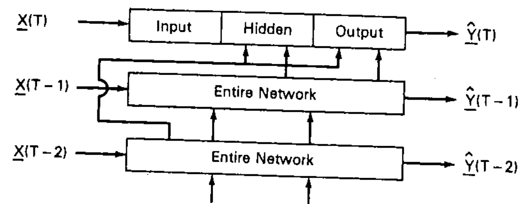

Fig. 5. Generalized network design with time lags.

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

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

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
```

```
C  NEXT CALCULATE THE DERIVATIVES IN A SWEEP
   BACKWARDS THROUGH TIME
            DO  500 t = T,1,-1
C                   First, calculate the errors at the
C                     current time t
                    DO  410 i = 1,n
  410                   F_Yhat(i)=Yhat(t,i)-Y(t,i)
C               Next, calculate F_net(t) for time t and
C                 update the F_W running sums
                  call F_NET2(F_Yhat,W,W',W",x(t,),
                  F_net,F_net',F_net",F_W,
                  F_W',F_W")
C               Move F_net(t+1) to F_net(t), in effect,
C                 to prepare for a new t value
                    DO  420 i = m+1,N+n
                        F_net"(i)=F_net'(i)
  420                   F_net'(i)=F_net(i)
  500               CONTINUE
C  FINALLY, UPDATE THE WEIGHTS BY
   STEEPEST DESCENT
            DO  999 i = m+1,N+n
                DO  998 j = 1,N+n
                    W(i,j)=W(i,j)-
                      learning_rate*F_W(i,j)
                    W'(i,j)=W'(i,j)-
                      learning_rate*F_W'(i,j)
  998               W"(i,j)-learning_rate*
                      F_W"(i,j);
  999           CONTINUE
 1000       CONTINUE
```

Once again, note that we have to go backwards in time in order to get the required derivatives. (There are ways to do these calculations in forward time, but exact results require the calculation of an entire *Jacobian* matrix, which is far more expensive with large networks.) For backpropagation through time, the natural way to adapt the network is in one big batch. Also note that we need to store a lot of intermediate information (which is inconsistent with real-time adaptation). This storage can be reduced by clever programming if W' and W'' are sparse, but it cannot be eliminated altogether.

In using backpropagation through time, we usually need to use much smaller learning rates than we do in basic backpropagation if we use steepest descent at all. In my experience [20], it may also help to start out by fixing the W' weights to zero (or to 1 when we want to force memory) in an initial phase of adaptation, and slowly free them up.

In some applications, we may not really care about errors in classification at all times t. In speech recognition, for example, we may only care about errors at the *end* of a word or phoneme; we usually do output a preliminary classification before the phoneme has been finished, but we usually do not care about the accuracy of that preliminary classification. In such cases, we may simply set F_Yhat to zero in the times we do not care about. To be more sophisticated, we may replace (6) by a more precise model of what we *do* care about; whatever we choose, it should be simple to replace (9) and the F_Yhat loop accordingly.

IV. Extensions of the Method

Backpropagation through time is a very general method, with many extensions. This section will try to describe the most important of these extensions.

A. Use of Other Networks

The network shown in (1)–(5) is a very simple, basic network. Backpropagation can be used to adapt a wide variety of other networks, including networks representing econometric models, systems of simultaneous equations, etc. Naturally, when one writes computer programs to implement a different kind of network, one must either describe *which* alternative network one chooses or *else* put options into the program to give the user this choice.

In the neural network field, users are often given a choice of network "topology." This simply means that they are asked to declare which *subset* of the possible weights/connections will actually be used. Every weight removed from (15) should be removed from (16) as well, along with (12) and (14) (or whichever apply to that weight); therefore, simplifying the network by removing weights simplifies all the other calculations as well. (Mathematically, this is the same as fixing these weights to zero.) Typically, people will remove an entire block of weights, such that the limits of the sums in our equations are all shrunk.

In a truly brain-like network, each neuron [in (15)] will only receive input from a small number of other cells. Neuroscientists do not agree on how many inputs are typical; some cite numbers on the order of 100 inputs per cell, while others quote 10 000. In any case, all of these estimates are small compared to the billions of cells present. To implement this kind of network efficiently on a conventional computer, one would use a linked list or a list of offsets to represent the connections actually implemented for each cell; the same strategy can be used to implement the backwards calculations and keep the connection costs low. Similar tricks are possible in parallel computers of all types. Many researchers are interested in devising ways to *automatically* make and break connections so that users will not have to specify all this information in advance [20]. The research on topology is hard to summarize since it is a mixture of normal science, sophisticated epistemology, and extensive ad hoc experimentation; however, the paper by Guyon *et al.* [13] is an excellent example of what works in practice.

Even in the neural network field, many programmers try to avoid the calculation of the exponential in (5). Depending on what kind of processor one has available, this calculation can multiply run times by a significant factor.

In the first published paper which discussed backpropagation at length as a way to adapt neural networks [14], I proposed the use of an artificial neuron ("continuous logic unit," CLU) based on

$$s(z) = 0, \quad z < 0$$
$$s(z) = z, \quad 0 < z < 1$$
$$s(z) = 1, \quad z > 1.$$

This leads to a very simple derivative as well. Unfortunately, the *second* derivatives of this function are not well behaved, which can affect the efficiency of some applications. Still, many programmers are now using piecewise linear approximations to (5), along with lookup tables, which can work relatively well in some applications. In earlier experiments, I have also found good uses for a Taylor series approximation:

$$s(z) = 1/(1 - z + 0.5 * z^2), \qquad z < 0$$

$$s(z) = 1 - 1/(1 + z + 0.5 * z^2), \quad z > 0.$$

In a similar spirit, it is common to speed up learning by "stretching out" $s(z)$ so that it goes from -1 to 1 instead of 0 to 1.

Backpropagation can also be used without using neural networks at all. For example, it can be used to adapt a network consisting entirely of user-specified functions, representing something like an econometric model. In that case, the way one proceeds depends on who one is programming for and what kind of model one has.

If one is programming for oneself and the model consists of a sequence of equations which can be invoked one after the other, then one should consider the tutorial paper [11], which also contains a more rigorous definition of what these "F_x_i" derivatives really mean and a proof of the chain rule for ordered derivatives. If one is developing a tool for others, then one might set it up to look like a standard econometric package (like SAS or Troll) where the user of the system types in the equations of his or her model; the backpropagation would go *inside* the package as a way to speed up these calculations, and would mostly be transparent to the user. If one's model consists of a set of simultaneous equations which need to be solved at each time, then one must use more complicated procedures [15]; in neural network terms, one would call this a "doubly recurrent network." (The methods of Pineda [16] and Almeida [17] are special cases of this situation.)

Pearlmutter [18] and Williams [19] have described alternative methods, designed to achieve results similar to those of backpropagation through time, using a different computational strategy. For example, the Williams–Zipser method is a special case of the "conventional perturbation" equation cited in [14], which rejected this as a neural network method on the grounds that its computational costs scale as the square of the network size; however, the method does yield exact derivatives with a time-forward calculation.

Supervised learning problems or forecasting problems which involve memory can also be translated into control problems [15, p. 352], [20], which allows the use of adaptive critic methods, to be discussed in the next section. Normally, this would yield only an approximate solution (or approximate derivatives), but it would also allow time-forward real-time learning. If the network itself contains calculation noise (due to hardware limitations), the adaptive critic approach might even be more robust than backpropagation through time because it is based on mathematics which allow for the presence of noise.

B. Applications Other Than Supervised Learning

Backpropagation through time can also be used in two other major applications: neuroidentification and neurocontrol. (For applications to sensitivity analysis, see [14] and [15].)

In neuroidentification, we try to do with neural nets what econometricians do with forecasting models. (Engineers would call this the identification problem or the problem of identifying dynamic systems. Statisticians refer to it as the problem of estimating stochastic time-series models.) Our training set consists of vectors $\boldsymbol{X}(t)$ and $\boldsymbol{u}(t)$, not $\boldsymbol{X}(t)$ and $\boldsymbol{Y}(t)$. Usually, $\boldsymbol{X}(t)$ represents a set of observations of the eXternal (sic) world, and $\boldsymbol{u}(t)$ represents a set of actions that we had control over (such as the settings of motors or actuators). The *combination* of $\boldsymbol{X}(t)$ and $\boldsymbol{u}(t)$ is input to the network at each time t. Our target, at time t, is the vector $\boldsymbol{X}(t + 1)$.

We could easily build a network to input these inputs, and aim at these targets. We could simply collect the inputs and targets into the format of Section II, and then use basic backpropagation. But basic backpropagation contains no "memory." The forecast of $\boldsymbol{X}(t + 1)$ would depend on $\boldsymbol{X}(t)$, but *not* on previous time periods. If human beings worked like this, then they would be unable to predict that a ball might roll out the far side of a table after rolling down under the near side; as soon as the ball disappeared from sight [from the current vector $\boldsymbol{X}(t)$], they would have no way of accounting for its existence. (Harold Szu has presented a more interesting example of this same effect: if a tiger chased after such a memoryless person, the person would forget about the tiger after first turning to run away. Natural selection has eliminated such people.) Backpropagation through time permits more powerful networks, which do have a "memory," for use in the same setup.

Even this approach to the neuroidentification problem has its limitations. Like the usual methods of econometrics [15], it may lead to forecasts which hold up poorly over multiple time periods. It does not properly identify where the noise comes from. It does not permit real-time adaptation. In an earlier paper [20], I have described some ideas for overcoming these limitations, but more research is needed. The first phase of Kawato's cascade method [9] for controlling a robot arm is an identification phase, which is more robust over time, and which uses backpropagation through time in a different way; it is a special case of the "pure robust method," which also worked well in the earliest applications which I studied [1], [20].

After we have solved the problem of identifying a dynamic system, we are then ready to move on to controlling that system.

In neurocontrol, we often *start out* with a model or network which describes the system or plant we are trying to control. Our problem is to adapt a *second* network, the *action* network, which inputs $\boldsymbol{X}(t)$ and outputs the control $\boldsymbol{u}(t)$. (In actuality, we can allow the action network to "see" or input the *entire* vector $\boldsymbol{x}(t)$ calculated by the model network; this allows it to *account* for memories such as the recent appearance of a tiger.) Usually, we want to adapt the action network so as to maximize some measure of performance or utility $U(\boldsymbol{X}, t)$ summed over time. Performance measures used in past applications have included everything from the energy used to move a robot arm [8], [9] through to net profits received by the gas industry [11]. Typically, we are given a set of possible initial states $\boldsymbol{x}(1)$, and asked to train the action network so as to maximize the sum of utility from time 1 to a final time T.

To solve this problem using backpropagation through time, we simply calculate the derivatives of our performance measure with respect to all of the weights in the action network. "Backpropagation" refers to how we calculate the derivatives, not to anything involving pattern recognition or error. We then adapt the weights according to these derivatives, as in (12), except that the sign of the adjustment term is now positive (because we are *maximizing* rather than *minimizing*).

The easiest way to implement this approach is to merge

the utility function, the model network, and the action network into one big network. We can then construct the dual to this entire network, as described in 1974 [1] and illustrated in my recent tutorial [11]. However, if we wish to keep the three component networks distinct, then the bookkeeping becomes more complicated. The basic idea is illustrated in Fig. 6, which maps exactly into the approach used by Nguyen and Widrow [7] and by Jordan [8].

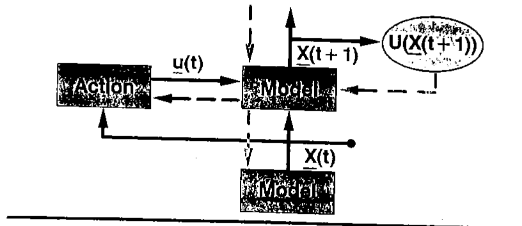

Fig. 6. Backpropagating utility through time. (Dashed lines represent derivative calculations.)

Instead of working with a *single* subroutine, NET, we now need *three* subroutines:

$$\mathrm{UTILITY}(\boldsymbol{X};\ t;\ \boldsymbol{x}'';\ U)$$

$$\mathrm{MODEL}(\boldsymbol{X}(t),\ \boldsymbol{u}(t);\ \boldsymbol{x}(t);\ \boldsymbol{X}(t+1))$$

$$\mathrm{ACTION}(\boldsymbol{x}(t);\ \boldsymbol{W};\ \boldsymbol{x}'(t);\ \boldsymbol{u}(t)).$$

In each of these subroutines, the two arguments on the right are technically outputs, and the argument on the far right is what we usually think of as the output of the network. We need to know the full vector $\boldsymbol{x}$ produced inside the model network so that the action network can "see" important memories. The action network does not need to have its own internal memory, but we need to save its internal state ($\boldsymbol{x}'$) so that we can later calculate derivatives. For simplicity, I will assume that MODEL does not contain any lag-two memory terms (i.e., W'' weights). The primes after the x's indicate that we are looking at the internal states of different networks; they are unrelated to the primes representing lagged values, discussed in Section III, which we will also need in what follows.

To use backpropagation through time, we need to construct dual subroutines for *all* three of these subroutines:

$$F_\mathrm{UTILITY}(\boldsymbol{x}'';\ t;\ F_X)$$

$$F_\mathrm{MODEL}(F_\mathrm{net}',\ F_X(t+1);\ \boldsymbol{x}(t);\ F_\mathrm{net},\ F_u)$$

$$F_\mathrm{ACTION}(F_u;\ \boldsymbol{x}'(t);\ F_W).$$

The outputs of these subroutines are the arguments on the far right (including *F_*net), which are represented by the broken lines in Fig. 4. The subroutine *F_*UTILITY simply reports out the derivatives of $U(\boldsymbol{x}, t)$ with respect to the variables X_i. The subroutine *F_*MODEL is like the earlier subroutine *F_*NET2, except that we need to output *F_u* instead of derivatives to weights. (Again, we are adapting only the action network here.) The subroutine *F_*ACTION is virtually identical to the old subroutine *F_*NET, except that we need to calculate *F_W* as a running sum (as we did in *F_*NET2).

Of these three subroutines, *F_*MODEL is by far the most complex. Therefore, it may help to consider some possible code.

```
      SUBROUTINE F_MODEL(F_net', F_X', x, F_net, F_u)
C     The weights inside this subroutine are those
C     used in MODEL, analogous to those in NET2, and are
C     unrelated to the weights in ACTION
      REAL F_net'(N+n),F_X'(n),x(N+n),
     F_net(N+n),F_u(p),F_x(N+n)
      INTEGER i,j,n,m,N,p
      DO 1 i=1,N
    1   F_x(i)=0.
      DO 2 i = 1,n
    2   F_x(i+N)=F_X(i)
      DO 1000 i = N+n,1,-1
        DO 910 j = i+1,N+n
  910     F_x(i)=F_x(i)+W(j,i)*F_net(j)
        DO 920 j = m+1,N+n
  920     F_x(i)=F_x(i)+W'(j,i)*F_net'(j)
 1000   F_net(i)=F_x(i)*x(i)*(1-x(i));
      DO 2000 i = 1,p
 2000   F_u(i)=F_x(n+i)
```

The last small DO loop here assumes that $\boldsymbol{u}(t)$ was part of the input vector to the original subroutine MODEL, inserted into the slots between $\boldsymbol{x}(n+1)$ and $\boldsymbol{x}(m)$. Again, a good programmer could easily compress all this; my goal here is only to illustrate the mathematics.

Finally, in order to adapt the action network, we go through multiple passes, each starting from one of the starting values of $\boldsymbol{x}(1)$. In each pass, we call ACTION and then MODEL, one after the other, until we have built up a stream of forecasts from time 1 up to time T. Then, for each time t going backwards from T to 1, we call the UTILITY subroutine, then *F_*UTILITY, then *F_*MODEL, and then *F_*ACTION. At the end of the pass, we have the correct array of derivatives *F_W*, which we can then use to adjust the weights of the action network.

In general, backpropagation through time has the advantage of being relatively quick and exact. That is why I chose it for my natural gas application [11]. However, it cannot account for noise in the process to the controlled. To account for noise in maximizing an arbitrary utility function, we must rely on adaptive critic methods [21]. Adaptive critic methods do *not* require backpropagation through time in any form, and are therefore suitable for true real-time learning. There are other forms of neurocontrol as well [21] which are not based on maximizing a utility function.

C. Handling Strings of Data

In most of the examples above, I assumed that the training data form one lone time series, from t equals 1 to t equals T. Thus, in adapting the weights, I always assumed batch learning (except in the code in Section II); the weights were always adapted *after* a complete set of derivatives was calculated, based on a complete pass through all the data. Mechanically, one could use pattern learning in the backwards pass through time; however, this would lead to a host of problems, and it is difficult to see what it would gain.

Data in the real world are often somewhere between the two extremes represented by Sections II and III. Instead of having a set of unrelated patterns or one continuous time series, we often have a *set* of time series or strings. For example, in speech recognition, our training set may consist of a set of strings, each consisting of one word or one sen-

tence. In robotics, our training set may consist of a set of strings, where each string represents one experiment with a robot.

In these situations, we can apply backpropagation through time to a *single string* of data at a time. For each string, we can calculate complete derivatives and update the weights. Then we can go on to the next string. This is like pattern learning, in that the weights are updated incrementally before the entire data set is studied. It requires intermediate storage for only one string at a time. To speed things up even further, we might adapt the net in stages, initially fixing certain weights (like W_{ii}) to zero or one.

Nevertheless, string learning is not the same thing as real-time learning. To solve problems in neuroidentification and supervised learning, the only consistent way to have internal memory terms *and* to avoid backpropagation through time is to use adaptive critics in a supporting role [15]. That alternative is complex, inexact, and relatively expensive for these applications; it may be unavoidable for true real-time systems like the human brain, but it would probably be better to live with string learning and focus on *other* challenges in neuroidentification for the time being.

D. Speeding Up Convergence

For those who are familiar with numerical analysis and optimization, it goes without saying that steepest descent—as in (12)—is a very inefficient method.

There is a huge literature in the neural network field on how to speed up backpropagation. For example, Fahlman and Touretzky of Carnegie-Mellon have compiled and tested a variety of intuitive insights which can speed up convergence a hundredfold. Their benchmark problems may be very useful in evaluating other methods which claim to do the same. A few authors have copied simple methods from the field of numerical analysis, such as quasi-Newton methods (BFGS) and Polak-Ribiere conjugate gradients; however, the former works only on small problems (a hundred or so weights) [22], while the latter works well only with batch learning and very careful line searches. The need for careful line searches is discussed in the literature [23], but I have found it to be unusually important when working with large problems, including simulated linear mappings.

In my own work, I have used Shanno's more recent conjugate gradient method with batch learning; for a *dense* training set—made up of distinctly different patterns—this method worked better than anything else I tried, including pattern learning methods [12]. Many researchers have used *approximate* Newton's methods, without saying that they are using an approximation; however an *exact* Newton's method can also be implemented in $O(N)$ storage, and has worked reasonably well in early tests [12]. Shanno has reported new breakthroughs in function minimization which may perform still better [24]. Still, there is clearly a lot of room for improvement through further research.

Needless to say, it can be much easier to converge to a set of weights which do not minimize error or which assume a simpler network; methods of that sort are also popular, but are useful only when they clearly fit the application at hand for identifiable reasons.

E. Miscellaneous Issues

Minimizing square error and maximizing likelihood are often taken for granted as fundamental principles in large parts of engineering; however, there is a large literature on alternative approaches [12], both in neural network theory and in robust statistics.

These literatures are beyond the scope of this paper, but a few related points may be worth noting. For example, instead of minimizing square error, we could minimize the 1.5 power of error; all of the operations above still go through. We can minimize E of (5) *plus* some constant k times the sum of squares of the weights; as k goes to infinity and the network is made linear, this converges to Kohonen's pseudoinverse method, a common form of associative memory. Statisticians like Dempster and Efron have argued that the linear form of this approach can be better than the usual least squares methods; their arguments capture the essential insight that people can forecast by analogy to historical precedent, instead of forecasting by a comprehensive model or network. Presumably, an ideal network would bring together both kinds of forecasting [12], [20].

Many authors worry a lot about local minima. In using backpropagation through time in robust estimation, I found it important to keep the "memory" weights near zero at first, and free them up gradually in order to minimize problems. When T is much larger than m—as statisticians recommend for good generalization—local minima are probably a lot less serious than rumor has it. Still, with T larger than m, it is very easy to construct local minima. Consider the example with $m = 2$ shown in Table I.

Table 1 Training Set for Local Minima

t	$X(t)$	$Y(t)$
1	0 1	.1
2	1 0	.1
3	1 1	.9

The error for each of the patterns can be plotted as a contour map as a function of the two weights w_1 and w_2. (For this simple example, no threshold term is assumed.) Each map is made up of straight contours, defining a fairly sharp trough about a central line. The three central lines for the three patterns form a triangle, the vertices of which correspond roughly to the local minima. Even when T is much larger than m, conflicts like this can exist within the training set. Again, however, this may not be an overwhelming problem in practical applications [19].

V. Summary

Backpropagation through time can be applied to many different categories of dynamical systems—neural networks, feedforward systems of equations, systems with time lags, systems with instantaneous feedback between variables (as in ordinary differential equations or simultaneous equation models), and so on. The derivatives which it calculates can be used in pattern recognition, in systems identification, and in stochastic and deterministic control. This paper has presented the key equations of backpropagation, as applied to neural networks of varying degrees of complexity. It has also discussed other papers which elaborate on the extensions of this method to more general applications and some of the tradeoffs involved.

References

[1] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Committee on Appl. Math., Harvard Univ., Cambridge, MA, Nov. 1974.

[2] D. Rumelhart, D. Hinton, and G. Williams, "Learning internal representations by error propagation," in D. Rumelhart and F. McClelland, eds., *Parallel Distributed Processing, Vol. 1.* Cambridge, MA: M.I.T. Press, 1986.

[3] D. B. Parker, "Learning-logic," M.I.T. Cen. Computational Res. Economics Management Sci., Cambridge, MA, TR-47, 1985.

[4] Y. Le Cun, "Une procedure d'apprentissage pour reseau a seuil assymetrique," in *Proc. Cognitiva '85*, Paris, France, June 1985, pp. 599–604.

[5] R. Watrous and L. Shastri, "Learning phonetic features using connectionist networks: an experiment in speech recognition," in *Proc. 1st IEEE Int. Conf. Neural Networks*, June 1987.

[6] H. Sawai, A. Waibel, P. Haffner, M. Miyatake, and K. Shikano, "Parallelism, hierarchy, scaling in time-delay neural networks for spotting Japanese phonemes/CV-syllables," in *Proc. IEEE Int. Joint Conf. Neural Networks*, June 1989.

[7] D. Nguyen and B. Widrow, "The truck backer-upper: An example of self-learning in neural networks," in W. T. Miller, R. Sutton, and P. Werbos, Eds., *Neural Networks for Robotics and Control.* Cambridge, MA: M.I.T. Press, 1990.

[8] M. Jordan, "Generic constraints on underspecified target trajectories," in *Proc. IEEE Int. Joint Conf. Neural Networks*, June 1989.

[9] M. Kawato, "Computational schemes and neural network models for formation and control of multijoint arm trajectory," in W. T. Miller, R. Sutton, and P. Werbos, Eds., *Neural Networks for Robotics and Control.* Cambridge, MA: M.I.T. Press, 1990.

[10] R. Narendra, "Adaptive control using neural networks," in W. T. Miller, R. Sutton, and P. Werbos, Eds., *Neural Networks for Robotics and Control.* Cambridge, MA: M.I.T. Press, 1990.

[11] P. Werbos, "Maximizing long-term gas industry profits in two minutes in Lotus using neural network methods," *IEEE Trans. Syst., Man, Cybern.*, Mar./Apr. 1989.

[12] —, "Backpropagation: Past and future," in *Proc. 2nd IEEE Int. Conf. Neural Networks*, June 1988. The transcript of the talk and slides, available from the author, are more introductory in nature and more comprehensive in some respects.

[13] I. Guyon, I. Poujaud, L. Personnaz, G. Dreyfus, J. Denker, and Y. Le Cun, "Comparing different neural network architectures for classifying handwritten digits," in *Proc. IEEE Int. Joint Conf. Neural Networks*, June 1989.

[14] P. Werbos, "Applications of advances in nonlinear sensitivity analysis," in R. Drenick and F. Kozin, Eds., *Systems Modeling and Optimization: Proc. 10th IFIP Conf. (1981).* New York: Springer-Verlag, 1982.

[15] —, "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, Oct. 1988.

[16] F. J. Pineda, "Generalization of backpropagation to recurrent and higher order networks," in *Proc. IEEE Conf. Neural Inform. Processing Syst.*, 1987.

[17] L. B. Almeida, "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment," in *Proc. 1st IEEE Int. Conf. Neural Networks*, 1987.

[18] B. A. Pearlmutter, "Learning state space trajectories in recurrent neural networks," in *Proc. Int. Joint Conf. Neural Networks*, June 1989.

[19] R. Williams, "Adaptive state representation and estimation using recurrent connectionist networks," in W. T. Miller, R. Sutton, and P. Werbos, Eds., *Neural Networks for Robotics and Control.* Cambridge, MA: M.I.T. Press, 1990.

[20] P. Werbos, "Learning how the world works: Specifications for predictive networks in robots and brains," in *Proc. 1987 IEEE Int. Conf. Syst., Man, Cybern.*, 1987.

[21] —, "Consistency of HDP applied to a simple reinforcement learning problem," *Neural Networks*, Mar. 1990.

[22] J. Dennis and R. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Englewood Cliffs, NJ: Prentice-Hall, 1983.

[23] D. Shanno, "Conjugate-gradient methods with inexact searches," *Math. Oper. Res.*, vol. 3, Aug. 1978.

[24] —, "Recent advances in numerical techniques for large-scale optimization," in W. T. Miller, R. Sutton, and P. Werbos, Eds., *Neural Networks for Robotics and Control.* Cambridge, MA: M.I.T. Press, 1990.

Paul J. Werbos received degrees from Harvard University, Cambridge, MA, and the London School of Economics, London, England, that emphasized mathematical physics, international political economy, and economics. He developed backpropagation for the Ph.D. degree in applied mathematics at Harvard.

He is currently Program Director for Neuroengineering and Emerging Technology Initiation at the National Science Foundation (NSF) and Secretary of the International Neural Network Society. While an Assistant Professor at the University of Maryland, he developed advanced adaptive critic designs for neurocontrol. Before joining the NSF in 1989, he worked nine years at the Energy Information Administration (EIA) of DOE, where he variously held lead responsibility for evaluating long-range forecasts (under Charles Smith), and for building models of industrial, transportation, and commercial demand, and natural gas supply using backpropagation as one among several methods. In previous years, he was Regional Director and Washington representative of the L-5 Society, a predecessor to the National Space Society, and an organizer of the Global Futures Roundtable. He has worked on occasion with the National Space Society, the Global Tomorrow Coalition, the Stanford Energy Modeling Forum, and Adelphi Friends Meeting. He also retains an active interest in fuel cells for transportation and in the foundations of physics.