EE-373 Spring 1991
Adaptive Systems B. Widrow

An Overview of Temporal Backpropagation

R. Timothy Edwards, Stanford University

Abstract

This paper presents a derivation of a training algorithm for backpropagation neural networks which operate upon time-dependent input data, called "temporal backpropagation." The derivation uses transformation rules to generate the algorithm directly from the standard backpropagation algorithm. Issues encountered when training time-delay neural networks are discussed, and a number of examples are shown.

1 Introduction to Temporal Learning

Neural networks have been used to solve problems in the field of pattern recognition and system identification with a fair amount of success. The backpropagation algorithm caused a renewal of interest in the field by providing a way of training multilayer networks with nonlinear transfer functions. The basic backpropagation algorithm assumes that the neural network is a combinational circuit, providing an output for a given input. However, many applications suitable for adaptive learning have inherently temporal structures, of which speech recognition is a prime example. Recently, a training algorithm has been derived which is the temporal equivalent of backpropagation [4], and is known as "temporal backpropagation" [2]. The first part of this paper gives an intuitive derivation of temporal backpropagation using transformational rules applied to standard backpropagation. The second part of the paper looks at a number of applications and examines various methods needed to solve temporal problems correctly.

2 A Derivation of Temporal Backpropagation

2.1 A Notation for Static Networks

In order to lay down a basic framework for a derivation of temporal backpropagation, it is crucially important to first clarify the notation which will be used throughout this paper. I will begin by describing a standard backpropagation network. Figure 1 shows a basic four-layer network in the style which will be used in all of the illustrations in the paper. Blank circles represent "placeholders"—that is to say, nodes which store an activation value (for instance, a floating-point number in the range [-1, 1] inclusive). The number of layers is defined in this notation as the number of columns containing placeholders. Each circle containing a sigma symbol (Σ) represents a summation of the values on the lines connecting it to various placeholders. Each of these connecting lines represents not only the physical connection, but also the weight or gain factor of the line. Ideally, information compression in drawing a network is best achieved by assigning a gradation of color to represent the weights and activations, such as a green line for a positively weighted connection and a red line for a negatively weighted connection, both fading to a neutral gray as weights go to zero. However, in this unenlightened day and age of black-and-white printers, it seems best to leave the weight designation out of the drawing in order to simplify the picture.

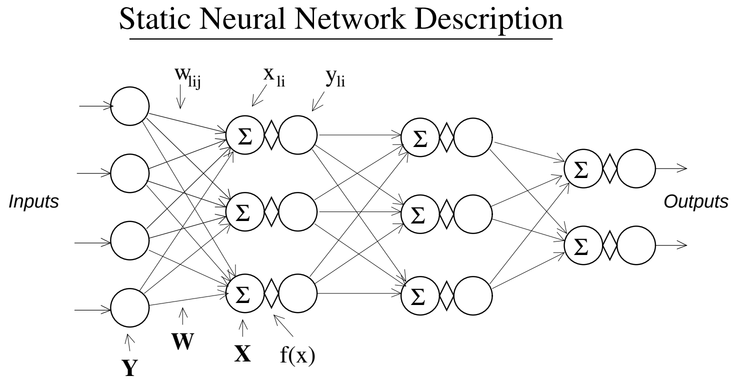

Figure 1: A standard static backpropagation network, including notation.

The diamond between the summation and placeholder represents some kind of function, usually a nonlinear compression such as a sigmoid. In the color notation, the diamond can be colored to represent the threshold value; for although mathematically the threshold functions as a weight attached to a placeholder stuck at the highest activation value, the representation is more intuitively seen and less cluttered when represented as shown here.

There are three matrices and one function represented by this figure. Placeholder activation values are denoted Y, each element of which is $y_{\ell i}$ where ℓ is a layer (in this case, $1 \leq \ell \leq 4$) and i is a node or an input (in this case $1 \leq i \leq 4$ for layer $\ell = 1$, $1 \leq i \leq 3$ for layers $\ell = 2$ and $\ell = 3$, and $1 \leq i \leq 2$ for layer $\ell = 4$). The input layer is always designated by $\ell = 1$ and since the network layout is symmetric, there is no difference between specifying $i = 1$ at the top or bottom. However, since temporal backpropagation networks have some directionality, I will simply state that $i = 1$ is at the top of the network to avoid confusion later on (a three-dimensional representation would circumvent this problem, but is rather difficult to draw on paper). The vector X is the value at the summing node after summation, and each element is $x_{\ell i}$ with the same meaning as for $y_{\ell i}$ except that there are no entries for $\ell = 1$ because that is the input. The vector W is the weighted value of a line, each element of which is $w_{\ell ij}$ where the new subscript j denotes the node in layer $(\ell - 1)$ to which the connection is made, and the subscript i denotes the node number in layer ℓ to which the connection is made. The function $f(\cdot)$ connects each summing node to a placeholder, and therefore is a function of $x_{\ell i}$. Notationally, each layer ℓ except the input contains a column of placeholders, a column of summing nodes and functions (one per placeholder), and a set of weights connecting that layer to the next layer in the direction of the input. The input layer only contains placeholders.

The relationships represented in this figure are as follows:

$$Y_\ell = \begin{cases} f(X_\ell) & 1 < \ell \leq L \\ \text{(input value)} & \ell = 1 \end{cases}$$

$$X_\ell = W_\ell^T Y_{(\ell-1)} \qquad 1 < \ell \leq L, \qquad 1 \leq i \leq I_\ell$$

where the notation I_ℓ has been introduced to represent the total number of nodes per layer, and L represents the total number of layers as defined above.

Temporal backpropagation can get very confusing in vector notation. From here on I shall use the scalar notation exclusively. By providing loop index notation it also becomes easier to extract the algorithm for use in a computer simulation, which is the basic purpose of these algorithms,

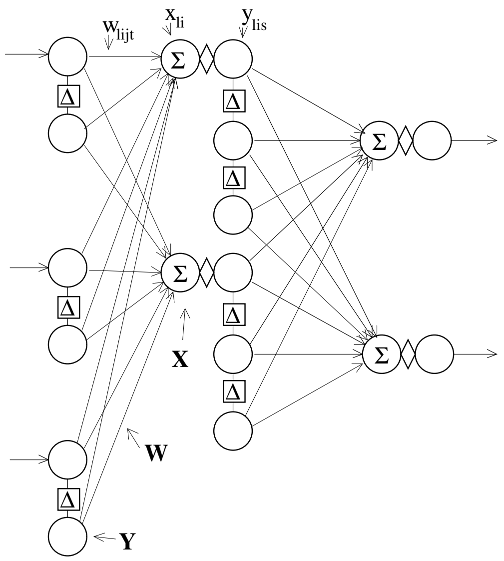

Figure 2: A standard temporal neural network, including notation.

anyway. The scalar notation of the above becomes:

$$y_{\ell i} = \begin{cases} f(x_{\ell i}) & 1 \leq i \leq I_\ell, \qquad 1 < \ell \leq L \\ \text{(input value)} & 1 \leq i \leq I_\ell, \qquad \ell = 1 \end{cases} \tag{1}$$

$$x_{\ell i} = \sum_{j=1}^{I_{(\ell-1)}} w_{\ell i j}\, y_{(\ell-1)j} \qquad 1 < \ell \leq L, \qquad 1 \leq i \leq I_\ell \tag{2}$$

The focus of this paper is to derive temporal relationships from static ones rather than deriving them from basic principles. A short derivation of standard backpropagation is provided in Appendix A for reference and as an additional source for understanding the network notation. At least a cursory look at Appendix A is recommended, as these equations will be referenced when deriving temporal backpropagation.

2.2 A Notation for Temporal Networks

Having gotten through a description of the notation for static networks, we may now proceed to define a temporal neural network. Figure 2 shows a typical example in the notation of this paper. In this illustration, a new element has been added, a square with a delta (Δ) sign representing a unit time delay (or, in z-transform notation, a multiplication by z^{-1}). Figure 3 shows a finite impulse response filter, in the same notation [1]. It can be seen that a time-delay neural network is simply a standard network in which each processing element has been replaced by a finite impulse filter, with the connectivity as shown in Figure 2. Outputs of a layer may only connect to the first tap of a node in the next layer. The term "node" is very loosely defined in the literature; for temporal networks, I define it as a chain of placeholders and delays (in other words, an FIR filter). A "tap" is defined as a given placeholder in the chain. A node with no delays is said to have one tap. Output nodes always have exactly one tap, because (as it will become apparent from the learning rule) applying the learning rule to a delayed output tap would amount to the same

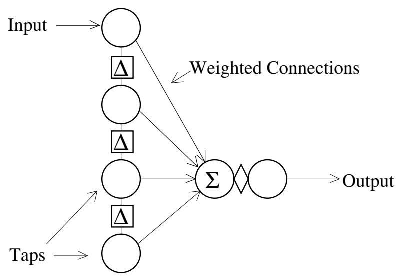

Figure 3: A finite impulse response (FIR) filter.

thing as applying the same rule as was applied at the last training step, and is therefore, in a sense, redundant. [1] In the new notation, we add the subscript t for the variable representing the tap along a filter (or "node"), and T_ℓ is the number of taps per node in a given layer. In the most generalized sense, T should be a function of layer *and* node, but it is not clear that allowing variable number of taps per node in a layer is ever necessary except in certain restrictive cases. At any rate, it is not difficult to further generalize the equations presented here for purely arbitrary architectures if the need arises. By this notation, each of the matrices X, W, and Y increases in dimensionality by one as shown in Figure 2.

2.3 Static Equivalent Networks

To understand the method of finding a temporal training algorithm for time-delay neural networks, it is vital to first understand the idea of a static equivalent network [2]. Every time-delay neural network can be represented as a standard static network simply by duplicating nodes and removing delays. The resultant net is much larger, contains a large number of weight duplications (or triplications, etc.), and is not fully interconnected. The process of creating the static equivalent can be thought of as "unfolding" the neural network, and the way that the static network matches the temporal one will become apparent as we work through an example. The idea of unfolding a network was introduced by Eric Wan of Stanford University [2]. Once the network is unfolded, the backpropagation algorithm can be applied directly to solve the static net.

In the first stage of unfolding, we make a count of all the nodes in the static net. This set of nodes is the "effective temporal reach" or "area of influence" of the network; i.e., the set of nodes whose values affect the output. Since values pass through delays, advance to other layers as part of a summation and pass through other delays, an input may have passed through the input chain of taps and delays and disappeared from the physical net while node values directly derived from that value are still in the physical net. Thus values not physically present have an effect on the output which needs to be included in the training algorithm. Therefore portions of the static equivalent network are a "virtual network" of nodes and weights representing past states of the physical network. Counting the total number of virtual nodes (a number which includes the physical nodes) is a simple procedure, starting at the last (output) layer and working backwards to the input layer. Figure 2 and its final unfolded equivalent in Figure 7 can help elucidate the following method. The output layer of the static network contains the same number of nodes as the output layer of the temporal network. Because I have set the constraint that the output layer has no delay taps, the next layer has no non-physical nodes, and the number of virtual nodes of the static equivalent of that layer is equal to the number of filters times the number of placeholders in each filter, or in

[1] however, repeatedly training previous patterns would, of course, lead to different learning behavior and may be worth investigating.

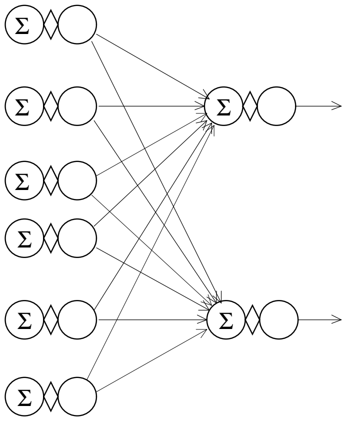

Figure 4: Partial static net in the unfolding process.

other words, the total number of placeholders in that layer of the temporal net. For each layer back to the input, the number of total virtual nodes (or total inputs, for the input layer) is a cumulative sum of the number of virtual nodes in that layer plus the number of virtual nodes calculated for the previous layer (toward the output) *minus* one. The subtraction occurs because the first placeholder in each filter accepts and propagates its input without delay to the next layer, so its effect on the output is immediate and physical, and never virtual. By this method of counting, the network of Figure 2 has a static equivalent with two nodes in the output, six nodes in the hidden layer, and twelve inputs. One can see that for each filter at the input, the value propagates to the next layer as part of the weighted sum, which takes two more time steps before it leaves the network altogether. The single delayed version of that input follows the same path one time step later. The input and its resultant values in the network have remained for four time steps. Since there are three filters in the input layer corresponding to three physical inputs, the static network has $4 \times 3 = 12$ inputs. These inputs are the three physical inputs plus the versions of those inputs delayed by 1, 2, and 3 unit delays. Incidentally, for this paper we consider that the network performs one learning cycle per unit delay, and that the input patterns are presented at the rate of one per unit delay. This seems to describe the most useful type of temporal net, but other configurations are certainly possible.

Mathematically, the notation for total virtual nodes at a layer (which I will also refer to as "accumulated taps") is:

$$T_\ell^{\mathrm{ACC}} = \begin{cases} 1 & \ell = L \\ T_{(\ell+1)}^{\mathrm{ACC}} + T_\ell - 1 & 1 \le \ell < L \end{cases} \tag{3}$$

where T_ℓ (no superscript) is the physical number of taps per filter as defined earlier.

We are now ready to proceed to the next stage, which is to actually unfold a network. Then we will see what a gigantic mess is produced, and having found the architecture for applying backpropagation, we will attempt to fold it back together. As with any road map, the task can be daunting if care has not been taken to remember how it was first unfolded! Therefore, we proceed with caution.

The first step is to copy down the output nodes, then to copy down all the placeholders of the next layer back, and make each one into a node by prepending a processing element (summation and limiter function) to it. This results in the partial network shown in Figure 4. We note that so far, the training algorithm is simply standard backpropagation without modifications, except that the hidden-layer nodes are referenced by three variables instead of two. Therefore,

$$\Delta w_{Lijt} = -\eta\, \delta_{Li1}\, y_{(L-1)jt} \qquad 1 \le j \le I_{(L-1)}, \qquad 1 \le t \le T_{(L-1)}, \qquad 1 \le i \le I_L \tag{4}$$

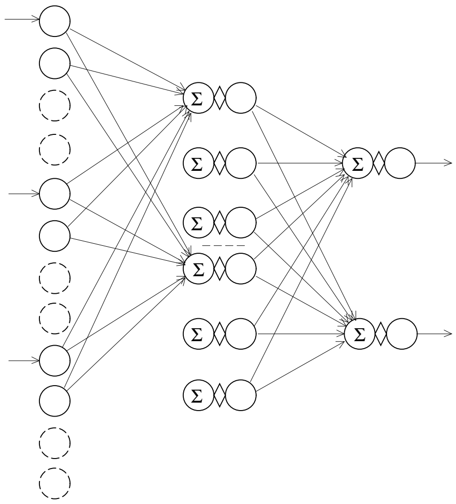

Figure 5: Second step during unfolding.

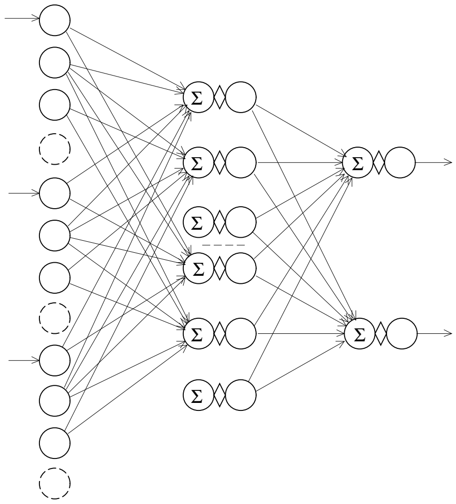

Figure 6: Third unfolding step—copying the weights.

and

$$\delta_{Li1} = -2(d_i - y_{Li1})\,\frac{\mathrm{d}}{\mathrm{d}x}(y_{Li1}) \qquad 1 \leq i \leq I_L \tag{5}$$

where I have used "1" as a subscript to denote that there is only one tap $s = 1$ associated with the output layer L. Note that in the notation of weight w, the subscript t is associated with node j of the next layer toward the input. Here it may be apparent why the notation $\frac{\mathrm{d}}{\mathrm{d}x}(y_{Li1})$ is preferred to $f'(x_{Li})$ as mentioned in Appendix A; the former expression, in the general case, is defined per tap, which is what is needed, whereas the latter has not been defined as propagating in time and therefore is not a function of tap. Alternately, we *could* propagate both x and y or propagate x and calculate a new y at each tap, but these methods are clearly computationally more expensive than the one adopted here.

Now, for the (conceptually) more difficult part: we take the next layer (in this example, the input layer) and draw it as shown in Figure 5. We still have the exact physical layout of the temporal net, but without the delay units. The next step is to *copy* the first layer and second layer weights downward, overlapping placeholders when necessary, until the number of inputs in the first

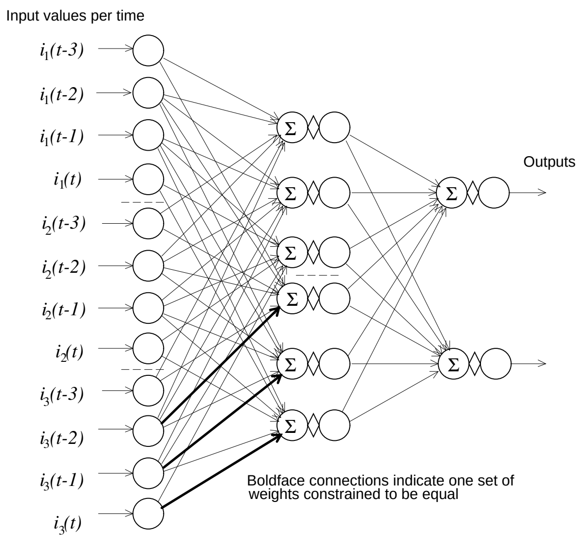

Figure 7: The final unfolded static equivalent of Figure 2.

layer equals the number of accumulated inputs calculated as shown above using equation (3). This copying is shown in Figures 6 and 7. Note that the final product does *not* have full interconnectivity among nodes! Also for the static case, we show twelve inputs when only three are physically available at a given moment; the others must be generated from time-delayed versions of the input as shown (using t for time here may be confusing but the learning algorithm will be concerned only with the input placeholder values, which are functions of filter and tap, and not with the true input, which is a function of filter and time. The difference is trivial anyway).

2.4 Transformation Equations and Temporal Backpropagation

At this messy point, we have to "refold" the network by interpreting its elements in the temporal notation. To do this, we simply write down the unfolding routine in mathematical notation.

The inputs y_{1it} previously included the set $(1 \le i \le I_1,\, 1 \le t \le T_1)$, but now includes the set $(1 \le i \le I_1,\, 1 \le t \le T_1^{\mathrm{ACC}})$, which is larger. Although we have expanded the number of connections when creating the static equivalent network, we wish to constrain certain weights to be equal so that the total number of unique weights in the static network is the same as in the temporal network. The weights which are equal are the ones which have been copied during the unfolding process. For any input and any node in the hidden layer, we are able to tell what the weight of its connection is in terms of the temporal network variables, if such a connection exists. In reverse, which we also need for the training algorithm, we are able to tell to what inputs and hidden-layer nodes a given weight connects.

For any weight w_{2ijt}, the hidden layer elements to which it connects are defined by

$$y_{2in} \qquad 1 \le n \le T_2$$

where T_2 is the number of filter taps in the hidden layer. A single weight attaches to multiple elements because the weight has been copied several times, and this formula keeps track of all of the copies. Note that the use of y is allowable here. Even though the weight connects only to the x summation element physically in the temporal network, it connects to each of the taps virtually, acting through the summation process. Each one of the hidden-layer nodes in the virtual network, then, is shown as having its own summation element, and it is just as correct to say that the weight attaches to the placeholder as it is to say that the weight attaches to the summation element.

Likewise, the inputs to which the same weight connects are defined by

$$y_{1j(t+m-1)} \qquad 1 \le m \le T_2$$

For generalized layers (any $\ell \neq 1$),

$$w_{\ell ijt} \quad \text{connects to} \quad \begin{cases} y_{\ell in} & 1 \le n \le T_\ell^{\mathrm{ACC}} \\ y_{(\ell-1)j(t+n-1)} & 1 \le n \le T_\ell^{\mathrm{ACC}} \end{cases} \tag{6}$$

In reverse, we must consider more carefully the restriction that each node is not fully interconnected to the next layer toward the output. For a given node $y_{\ell is}$, the associated weights attached to it in the direction of the output are given by

$$y_{\ell is} \quad \text{connects to} \quad w_{(\ell+1)kim} \qquad \max(1, s - T_{(\ell+1)}^{\mathrm{ACC}}) \le m \le \min(s, T_\ell) \tag{7}$$

Finally, for a given $y_{\ell is}$ *and* a given weight attached to it $w_{(\ell+1)kim}$, the associated node on the other end of the weight (there is only one) is given by

$$y_{\ell is} \quad \text{and} \quad w_{(\ell+1)kim} \quad \text{connect to} \quad y_{(\ell+1)k(s-m+1)} \tag{8}$$

These equations are of the greatest importance to this paper, because they define the relationship between the connectivity of a temporal network and its static equivalent *regardless of the training algorithm used*.

From this point, we make substitutions into the standard backpropagation algorithm to find the temporal backpropagation training algorithm. We will substitute

$$\delta_{\ell i} \mapsto \sum_{n=1}^{T_\ell^{\mathrm{ACC}}} \delta_{\ell in} \tag{9}$$

because δ transforms like y, and

$$y_{(\ell-1)j} \mapsto \sum_{n=1}^{T_\ell^{\mathrm{ACC}}} y_{(\ell-1)j(t+n-1)} \tag{10}$$

The substitutions are quite straightforward; the weight change (Δw) equation has one transformation, and the propagated error (δ) has two, and yield the following (see Appendix B for a summary of all the temporal backpropagation equations):

$$\Delta w_{\ell ijt} = -\eta \sum_{n=1}^{T_\ell^{\mathrm{ACC}}} \delta_{\ell in}\, y_{(\ell-1)j(t+n-1)} \qquad 1 \le \ell \le L, \quad 1 \le i \le I_\ell, \quad 1 \le j \le I_{(\ell-1)}, \quad 1 \le t \le T_{(\ell-1)} \tag{11}$$

$$\delta_{\ell in} = \begin{cases} -2(d_i - y_{Li1}) \dfrac{\mathrm{d}}{\mathrm{d}x}(y_{Li1}) & \ell = L, \quad 1 \le i \le I_\ell, \quad n = 1 \\ \dfrac{\mathrm{d}}{\mathrm{d}x}(y_{\ell in}) \displaystyle\sum_{k=1}^{I_{(\ell+1)}} \sum_{m=\max(1, n-T_{(\ell+1)}^{\mathrm{ACC}}+1)}^{\min(n,T_\ell)} \delta_{(\ell+1)k(n-m+1)} w_{(\ell+1)kim} & 1 < \ell < L, \quad 1 \le i \le I_\ell, \quad 1 \le n \le T_\ell^{\mathrm{ACC}} \end{cases} \tag{12}$$

2.5 Observations on the Derivation

I have made one assumption in order to get the transformation equations (9) and (10), and that is to assume that the proper transformation is a summation. It is certainly not obvious or even derivable from the intuitive approach that the proper transformation is not, say, an average such as the following:

$$\delta_{\ell i} \mapsto \frac{1}{T_\ell^{\mathrm{ACC}}} \sum_{n=1}^{T_\ell^{\mathrm{ACC}}} \delta_{\ell i n} \tag{13}$$

In fact, the averaged expression was the one used by Waibel et al. [4] in their phoneme-recognition network and does not appear to have significant effect on convergence properties of the algorithm.

The point of this discussion so far is to prove that temporal backpropagation is easily derived from normal backpropagation by a set of simple tranformational rules. This disproves the premise of the paper by Wan [2] which states, in reference to the unfolded static network,

> It is necessary, however, to keep track of which static weights are actually the same so that the gradients may be accumulated to find the total gradient for each unique parameter in the network. This need to do global bookkeeping is an immediate drawback to the instantaneous gradient approach. There is a loss of a sense of locally distributed processing .... No nice recursive formula for propagating all the error terms exists.

In the end, though, such a formula does indeed exist, and the result which it obtains is precisely the same as the result produced in Wan's non-intuitive, convoluted derivation of temporal backpropagation.

2.6 Temporal Madaline Rule

The immediate benefit of the transformation rules is their universality. This is quite good news, since it allows us to immediately write a number of new learning algorithms. The transformations are independent of the learning rule which means that they can be applied to the "Madaline" training algorithms. Of greatest worth is Temporal Madaline III Rule, a rule that would be better suited to hardware implementations of time-delay neural networks than would the temporal beckpropagation rule. For details of the Madaline III rule, see Widrow and Lehr [5]. The standard Madaline III rule is

$$\Delta w_{\ell i j} = -\eta \, y_{(\ell-1)j} \left(\frac{\Delta E^2}{\Delta x_{\ell i}}\right)$$

and the simple transformation produces the temporal version

$$\Delta w_{\ell i j t} = -\eta \sum_{n=1}^{T_\ell^{\mathrm{ACC}}} y_{(\ell-1)j(t+n-1)} \left(\frac{\Delta E^2}{\Delta x_{\ell i}}\right) \tag{14}$$

in which $\Delta x_{\ell i}$ is a disturbance produced at one of the summation nodes, and ΔE^2 is the resultant change in the squared error at the output,

$$\Delta E^2 \equiv \sum_{i=1}^{I_L} (d_i - y_{Li1})^2$$

in the notation as has already been defined.

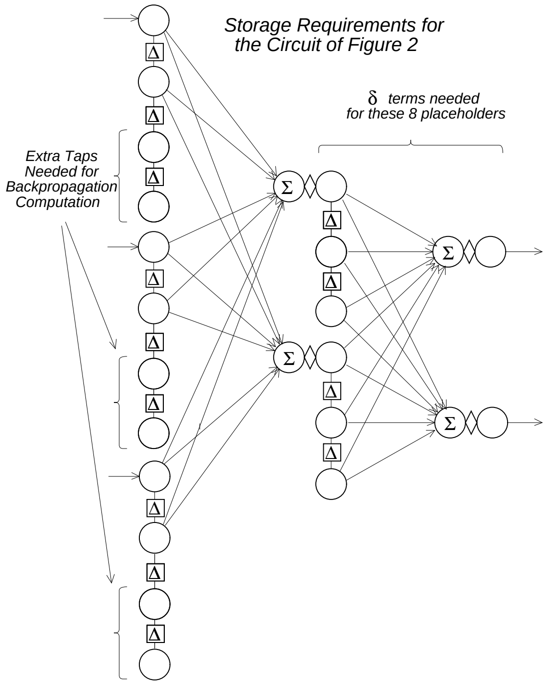

Figure 8: Computational storage requirements for a temporal network.

2.7 Computational Resources

An application-specific network which has fixed weights and does not learn, does not need extra memory or hardware in order to be implemented; the network of Figure 2 can be created as drawn. The unfortunate point about time-delay neural network training algorithms is that components of the virtual network appear in the training algorithm. Fortunately, it is possible to remove virtual weights from the algorithm, but the virtual placeholders remain and must be accounted for in any network realization, whether software or hardware, if the network must be capable of learning. Figure 8 shows the computational resources needed for implementing the network of Figure 2 as a learning system. Each of the virtual placeholders of the static network of Figure 7 must be included. Each virtual placeholder is a time-dependent value and so must have an associated delay element. For the temporal backpropagation algorithm, each δ term is associated with a node of the virtual network, and also requires memory for storage. This is a small price to pay for the benefit of compression of the entire network from its static counterpart. The total amount of computation performed is even slightly less than for a standard static backpropagation network, because the temporal network version's static equivalent is not fully interconnected.

3 Temporal Network Applications

3.1 Training Sequential Computer Elements

This section of the paper is concerned with aspects of time-delay neural network applications. The applications considered here are limited in scope and not exhaustively covered, but they serve to raise a number of issues concerned with the temporal nature of the training, and which do not occur

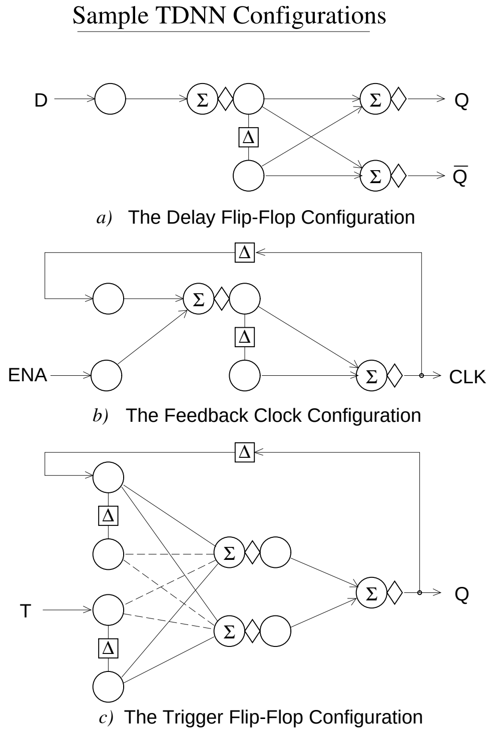

Figure 9: Three temporal network applications.

in static networks.

For this purpose, I have developed an application program written in C and running under the popular *X-Windows* workstation environment, which uses the graphical representation (color version) introduced in this paper to aid in the understanding of how the network learning process operates. For large networks, it is quite slow, but serves as a guide to selecting the proper network architecture for a given application, for selecting reasonable process parameters, and for attempting various tweaks to the learning mechanism (there exists an arsenal of them). Figure 11 shows the program running one of the applications described below. [2]

Figure 9 shows three network configurations used for an introductory exercise which serves to address some of the main issues in temporal training. The essential idea is that introductory examples for static backpropagation networks generally involve the learning of combinational logic functions: *AND*, *OR*, *XOR*, bit-shifts, and decoders. Therefore, if time delay neural networks can solve the basic sequential logic functions (latches, flip-flops, monostable multivibrators, etc.) then we have proven that, in theory, one can train a neural network to behave like any arbitrary computer (with an emphasis on "in theory").

Figure 9a shows the minimal configuration needed to represent a delay flip-flop, i.e., a net in which the input value appears at the output delayed one unit in time. We can choose to accept all output values within $\pm.1$ of the nominal digital values as representing a correct digital value. From now on we will refer to this range as the "tolerance" value (denoted by ϵ) of the training procedure. For digital applications, a tolerance of ± 2.0 for inputs in the range $[-1, 1]$ (indicating a $\tanh(\cdot)$ limiter function) accepts *all* outputs as correct regardless of their values; a tolerance of ± 1.0 is the maximum useful tolerance but gives a solution with zero noise margin; a tolerance of ± 0 will only classify as correct an output which exceeds the bit precision of the computer simulating the network. For all applications shown here the tolerance $\pm.1$ is assumed unless otherwise indicated.

[2]For those interested, this program has been compiled to run on a Sun–4 workstation, and the executeable code and the C code can be found at `leland.stanford.edu` under the directory `~rte/excess` in files `xtdnn` (executeable) and `xtdnn.c`. It is subject to revision without notice.

The delay flip-flop is simple to train, but immediately one difference with respect to static networks arises, which is that patterns may be presented to a static network in any order, but must be strictly ordered when presented to a temporal network. There are several solutions to this problem. One is to always present the (finite) training patterns in the same order and make sure the set "wraps" so that all the input taps of the network will be filled from the last pattern(s) in the set so as to give the correct answer for the first pattern on the next cycle of presentation. Alternately, if one knows the exact function to be trained, or is training by example from a system to be modeled, then an arbitrarily long pseudorandom sequence of inputs can be maintained. If neither of these methods is practicable, then we must tweak the training algorithm. The most sensible alteration seems to be one which utilizes the idea of tolerance. For a given network architecture, we know that it will take a number of steps T_0^{ACC} for a single input and its resultant effects to propagate through to the output. For any input pattern set immediately following an uncorrelated pattern set, we wish *not* to train the network for T_0^{ACC} steps. One way to accomplish this is to set the tolerance $\epsilon = f(\infty) - f(-\infty) \equiv \epsilon_{\mathrm{MAX}}$(=2 for $f(\cdot) = \tanh(\cdot)$), which is the full output range of the limiter function, during these steps. From here, we can use this value of ϵ to adjust the training by writing

$$\delta_{Li1} = -2\frac{\mathrm{d}}{\mathrm{d}x}(y_{Li1}) \min((\epsilon_{\mathrm{MAX}} - \epsilon),\ |d_i - y_{Li1}|) \operatorname{sgn}(d_i - y_{Li1}) \qquad \text{saturation method} \qquad (15)$$

or alternately,

$$\Delta w_{\ell ijt} = -\eta(1 - \frac{\epsilon}{\epsilon_{\mathrm{MAX}}}) \sum_{n=1}^{T_\ell^{\mathrm{ACC}}} \delta_{\ell in}\, y_{(\ell-1)j(t+n-1)} \qquad \text{percentage method} \qquad (16)$$

to name just a few possibilities. These equations are written in generalized form such that they are also useful for a variant method in which ϵ is graded linearly between $\epsilon = \epsilon_{\mathrm{MAX}}$ at the first time step and ϵ =(minimum desired tolerance) at time step T_0^{ACC}, or an even more sophisticated version in which ϵ is graded between the same values but proportionally to the fraction of the network through which the input has propagated. The graded-epsilon methods imply a "degree of confidence" that enough information has accumulated in the network to give a correct output. The saturation method limits the amount of weight change, preventing it from exceeding a value proportional to the confidence level. The percentage method gives a weight change directly proportional to the confidence level. Either method may be applied at the Δw equation (11) *or* at the δ equation (12). For the delay flip-flop, the degree of confidence is exactly zero at the first input and exactly 100% at the following time step. Examples in which a graded level of confidence might be desired will be considered shortly. None of these methods is firmly rooted in theory, and none has been investigated extensively.

Figure 9b shows a temporal network configured as a recurrent network, which with the correct weights can be made to toggle the output at every time step and which can be turned on and off with an enable signal, thus making a controllable clock. The fact to note about recurrent temporal networks is that T_ℓ^{ACC}, at any layer, is infinite. Any input presented at any time continues to influence the output forever. The finite impulse response filters have become infinite impulse response filters. There is no guarantee of convergence when training the network as drawn. However, there is a way to train such networks without invoking the problem of system instability (above and beyond the usual stability and convergence problems encountered in all backpropagation networks during training). That is to feed back the *desired response* to the network input after one unit delay *in place of* the actual output, and continuing to train until the network response meets the required tolerance under feedforward conditions with the actual output fed back. In this way, the input does not appear to the net to be a function of the output and the network learns as if it were not a recurrent net. [3]

The paragon of recurrent networks is the trigger flip-flop (see Figure 9c). It is the time-delay counterpart of the ubiquitous *XOR* problem in static networks. In digital computation, a trigger flip-flop is a device with one input and one output. The output toggles values after each clock cycle in which the input is logical high, and remains the same when the input is logical low (the physical device also has a reset input to force the first state of the output, but we will ignore this and assume all placeholders initialize with some predetermined value). To correctly generalize the problem, the circuit must solve an *XOR* problem between the current value of the output and the delayed value of the input; ideally, it would have strong weights connected to the corresponding placeholders (shown with solid lines in Figure 9c) and keep the other weights in the first layer close to zero (shown with dotted lines). The important lesson learned from the trigger flip-flop is that, unlike its counterpart the exclusive-or, it does *not* have four unique pattern sets. It does, in fact, have sixteen unique input pattern sets corresponding to the combinations of inputs which may exist in *any* of the four input placeholders, not just the two which we expect to have impact on the output. The training set must be carefully manufactured (or large enough if generated pseudorandomly) to cover all possibilities. This is a special case of a network which could benefit from an architecture allowing variable numbers of taps per filter in a given layer.

In actual training, the network had proper generalization of the input, but was unable to meet the required tolerance under true feedback conditions (at least during the time span in which I was willing to watch it) due to the fact that the solution using the desired output for "feedback" had converged to the point where weight changes were small, but the solution for true feedback had not reached the required tolerance. There are several solutions to this:

1. Relax the tolerance constraint.

2. After training, implement the network with a signum (hard-limiter) instead of a sigmoid at the output. This boosts the output, and therefore the feedback signal, to true digital values. This method can be used for any digital-output network, and appears to give good results.

3. Train in two separate stages, using the desired output for feedback until convergence occurs under those conditions, and then training using the actual output for feedback. This method has not been researched, and is beyond the scope of this paper.

The trigger flip-flop solution can be seen (barely) in figure 11, in which the network is running in feed-forward mode with a set of inputs which it was not trained upon. Note that the unfilled delay taps and the feedback connection cause the incorrect output for the first few inputs, but by the fourth input the network has bootstrapped into the proper state and performs as expected for the remainder of the input. It is difficult to see from the black-and-white drawing, but the weights attached to the first tap of the feedback input and the second tap of the true input do have the largest magnitude as predicted.

3.2 Four Temporal Paradigms

The computer-element exercises have dealt with two time-delay paradigms. One is the solution for a time-dependent output given a time-dependent input, such as the flip-flops, which I will call a TITO network, an acronym for Temporal-Input, Temporal-Output. The clock is an example of a

[3] A more complete discussion of recurrent networks can be found in Hecht-Nielsen [6].

static-input, time-dependent output (SITO) network. Standard backpropagation is a limiting case of time-delay networks in which all filters have exactly one tap (the time-delay training algorithm reduces to standard backpropagation when all filters are reduced to a single tap) and comprises the majority of SISO networks. In the next section we will consider the final paradigm (TISO networks) which is suited to temporal pattern classification. Any resemblance in the preceding discussion to parallel computer architectures (the Flynn classifications SIMD, MIMD, etc.) is probably intentional.

3.3 Speech Recognition Networks

A number of new issues is raised when considering a complicated application such as speech recognition. The temporal nature of speech suggests that a time-delay neural network is better suited to the task than a static backpropagation network. As an example, consider the case of recognition of a single word in the middle of a continuous stream of speech data generated in real-time. Using a static network, it is necessary to first use standard speech-recognition techniques such as monitoring average energy per unit time to locate the boundaries of the word, and then store the entire word in memory and present it to the network at one time. Recognition rates are low for a variety of reasons. The network is poor at dealing with duration variations which might occur in separate instances of the same word, although a network which has generalized well is capable of tolerating small shifts and scaling (technically, "time warping") of the input. The approach of using static networks for word recognition assumes that recognition of the entire word should be distributed over the entire network. This is a valid assumption, in some respects, but any speech data generator which samples at a rate high enough to capture the essential features of speech (usually requires $10\,\mathrm{ms}$ samples or smaller) and good frequency resolution (assuming a frequency decomposition of sound as occurs in the human ear, and separated into at least twelve frequency bands) over the course of an entire word (say, $\frac{1}{2}$ second) will create at least 600 input values for the average word [7, 8]. That implies a tremendously large neural network. To reduce the number of interconnections, it becomes necessary to discard the notion that recognition must be fully distributed over the net; i.e., that the network must be fully interconnected. This leads to the notion that word recognition is performed hierarchically by first recognizing basic features, then recognizing separate phonemes from the features, then syllables from the phonemes, then words from the syllables. We match the recognition hierarchy to the network hierarchy of layers to form a set of localized nets instead of a single global network. This is shown in Figure 10 [9]. The network is now not fully interconnected. Each hidden layer node is now only responsible for handling a set of inputs equivalent to the desired feature "size" for that layer (where the size is measured in time-delay units) plus some overlap to account for shifts in the input. This bears a striking resemblance to Figure 7 wherein each input vs. time is more sparsely connected at the endpoints than in the middle. There are two more facts to consider about the locally-distributed static network. First, if we have trained the network to recognize a word such as, for instance, "degree," then there will exist redundancy in the network where it must make separate representations of the phoneme "$\bar{e}$." Secondly, we may consider overcoming the problem of marking isolated word boundaries (which often fails because words are often concatenated in continuous speech) by ignoring boundaries altogether and passing the continuous speech past the inputs of the network in time. An exhaustively-trained network that generalizes well should suppress the output at all times except the few instances in which the word lines up properly at the input. Having made these changes to the original static network, we have essentially created an unfolded time-delay neural network, and the only remaining step is to "fold

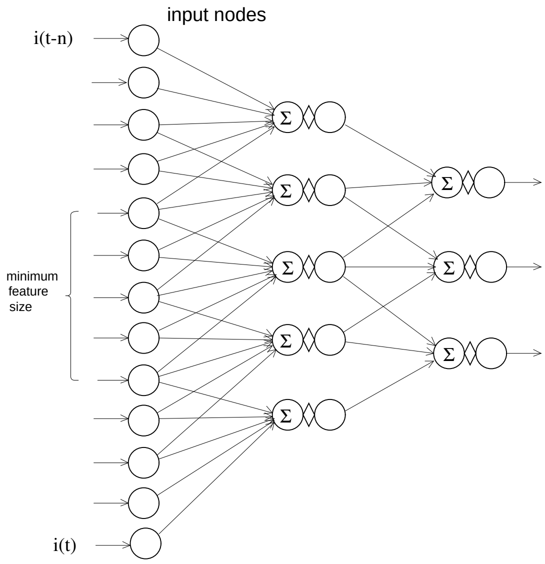

Figure 10: A locally-distributed static network.

up" the network to remove the redundancies mentioned above.

3.4 Choosing an Architecture

Given the original static network for speech recognition, it is difficult to make an educated guess as to the number of hidden layers and hidden-layer units needed to optimally solve the word-recognition problem. The considerations which we have made to derive the basic form of the time-delay neural network leads us to a simple calculation of any time-delay neural network architecture, provided that we know enough about the input data or about the learning process which we are trying to model. The number of layers is equal to the hierarchy of features we wish to detect (e.g., phoneme, syllable, word); the number of filters per layer is equal to the number of features (e.g., about 40 basic English phonemes), and the number of taps per filter is equal to the time span of the feature divided by the delay time of the tap delays. The total number of "virtual taps" at the input should equal the total time length of an input pattern group, even though the actual number of physical input taps may be quite small.

3.5 Self-Organization in Temporal Networks

The statement that a well-trained network should suppress the output at all times except the time in which the input data completely fill the virtual placeholders at the input leads to some observations on time-delay neural network training and on self-organization in general.

All neural networks are self-organizing in the sense that there are many sets of weight values representing different solutions to the same problem, and inasmuch as these weights represent "rules" which the network applies to the input, the network must organize these rules by itself. The networks which we typically call "self-organizing" are those in which the organization is reflected in the output (i.e., no desired response is given). In other words, there are no specific laws which force a network to be self-organizing or not. Temporal backpropagation networks can be self-organized also.

Consider the temporal word-recognition network. It must be trained with words, which is to say separate pattern groups. At the beginning of the presentation of a word, the data which exist in the taps are unrelated to the input word. From there, it seems appropriate to apply an epsilon rule as mentioned earlier; we cannot expect an entire word to be recognized based on the first $10\,\mathrm{ms}$ of data! But what seems more "natural"—to train *only* on the last input pattern, after all the virtual taps have been filled, or train on a gradual basis, increasing the magnitude of maximum weight change as the virtual input fills with the data? The idea of a last-input-only training rule leads to the idea of time-delay networks "learning with a critic" (also known as backpropagation-in-time), a limited form of self-organization [10]. If the input just fills the virtual network on the last input pattern of a group, then the training algorithm makes changes based upon information provided by *all* temporal states of the net, all the way back to the first. One would tend to expect a result in which the output has a number of spurious states between the first and last presentation of a word during a feed-forward run, which slowly "die out" as the training progresses and the generalization of the network improves. As an example, a word-recognition network learning the word "degree" must know what input represents the word "degree" at one moment in time, but must eventually learn what input is *not* "degree" at all points in time, which is not necessarily covered by a finite set of English words on which it is also trained during the last input pattern only. In this case, training should be accomplished by on one of the following methods:

1. Noise method: Alternatively train with a true word, then with an input set of randomly generated data to cover as many non-word events as possible.

2. Pegging method: Train on the last input pattern, and also on a small set of intermediate patterns, in which the output is said not to equal the output of the final state. Note that if we train on every step except the last, and demand that the output only have a given state during the last step, then the network is overwhelmed by a negative response; it may be attempting to force the output low for 99 steps and force it high for one, which violates the principle of minimal disturbance [5]. Pegging reduces the amount of disturbance to learned weights.

3. Scaling method: This is an idea similar to pegging, which is to train on every cycle except the last for a negative output response, but adapt the weights by a magnitude in inverse proportion to the number of input patterns in the group, and then train the final state with a full-magnitude weight change. For example, for a word which is 100 time steps (patterns) long, and for which an output of "1" is desired on the final state, the weight change Δw is calculated normally for each of the first 99 states for a desired output of "-1" but then attenuated by a factor of 100. The final weight change has a potential maximum magnitude which is equal to the weight change of the previous 99 steps put together, and so the principle of minimal disturbance is preserved.

A time-delay neural network utilizing last-input training is, then, a good choice for any application requiring a certain output on a final state without specifically training the network for states in between. This includes such standard "learning-with-a-critic" applications as the blackjack player [10] and the truck backer [5]. The time delay neural network algorithm provides a natural way to train such networks without the need to keep track of many copies of the same (static) network as is usually done.

4 Conclusions

Personal research in the area of temporal neural networks for the past year has been aimed at gaining a thorough but basic understanding of the field and of the issues involved. The training paradigms presented here are at an early stage of development and most have been only superficially researched. They are meant to provide an overview of the problems encountered when training time-delay neural networks, and to be a precursor to further investigation. At this point, it will be necessary to fully research each of these methods in order to understand in detail the time-delay learning process, and where several solutions have been proposed for a certain problem, learn the advantages and drawbacks of each. The derivation of temporal backpropagation has been the mathematically rigorous part of this paper, and I hope that it gives the reader an intuitive feel for the algorithm.

References

[1] B. Widrow and S. D. Stearns (1985) *Adaptive Signal Processing*, Englewood Cliffs, NJ, Prentice-Hall.

[2] E. A. Wan (1990) *Temporal Backpropagation: An Efficient Algorithm for Finite Impulse Response Neural Networks*, Proc. of the 1990 Connectionist Models Summer School, San Mateo, CA.

[3] Y. Pao (1989) *Adaptive Pattern Recognition and Neural Networks*, Reading, MA, Addison-Wesley, pp. 120–129.

[4] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang (1989) *Phoneme Recognition using Time-delay Neural Networks*, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 37, No. 3, March 1989, pp. 328–339.

[5] B. Widrow and M. Lehr (1990) *30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation*, IEEE Proceedings, vol. 78, No. 9, Sept. 1990, pp. 1415–1442.

[6] R. Hecht-Nielsen (1990) *Neurocomputing*, Reading, MA, Addison-Wesley, pp. 164–191.

[7] D. O'Shaughnessy (1987) *Speech Communication: Human and Machine*, Reading, MA, Addison-Wesley, p.45 and pp.420–422.

[8] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. II, The MIT Press, Cambridge, MA.

[9] T. Edwards (1990) *A Neural Network for Speech Recognition*, Duke University Senior Thesis, unpublished.

[10] B. Widrow, N. K. Gupta, S. Maitra (1973) *Punish/Reward: Learning with a Critic in Adaptive Threshold Systems*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, No. 5, Sept. 1973, pp. 455–465.

Appendices

A The Delta Rule of Backpropagation

This is a short derivation of Backpropagation which is intended for those who are already familiar with the algorithm but wish to compare it to other derivations using different notation as an aid to understanding the notation of this paper. For a more comprehensive explanation, see Pao [3].

At the output layer, the error between the output of a static neural network and the desired response for a given pattern may be written as

$$E = D - Y_L$$

where D is the desired output vector with elements d_i, for the range $0 \le i \le I_L$, and I_L is the number of nodes in the output layer. In scalar notation,

$$e_i = d_i - y_{Li} \qquad 1 < i \le I_L. \tag{17}$$

The instantaneous (per-pattern) squared error per node at the output is defined as $\mathcal{E} = e_i^2$. Weight changes are made by the least-squared-error rule

$$\Delta w_{Lij} = -\eta \frac{\partial \mathcal{E}}{\partial w_{Lij}} \qquad 1 \le i \le I_L, \qquad 1 \le j \le I_{(L-1)}$$

and an expansion by the chain rule of calculus yields

$$\Delta w_{Lij} = -\eta \frac{\partial \mathcal{E}}{\partial x_{Li}} \frac{\partial x_{Li}}{\partial w_{Lij}} = -\eta\, \delta_{Li}\, y_{(L-1)i}$$

where (and here we use the chain rule again)

$$\delta_{Li} \equiv \frac{\partial \mathcal{E}}{\partial x_{Li}} = \frac{\partial e_i^2}{\partial y_{Li}} \frac{\partial y_{Li}}{\partial x_{Li}} = -2 e_i\, f'(x_{Li})$$

because $e_i^2 = (d_i - y_{Li})^2$ and $y_{Li} = f(x_{Li})$, and $f(\cdot)$ is the sigmoid (or equivalent) function. Other notation introduced here includes η, a constant known as the training constant which is generally constrained by $0 \le \eta \le 1$; also $\Delta w_{\ell ij}$ is the change made to the specified weight between the current learning iteration and the next, and $\delta_{\ell i}$ is an effective error which will be propagated back to the next layer.

For arbitrary $\ell \neq L$ (and $\ell \neq 0$ due to the construction and notation of the network),

$$\Delta w_{\ell ij} = -\eta \frac{\partial \mathcal{E}}{\partial w_{\ell ij}} = -\eta \frac{\partial \mathcal{E}}{\partial x_{\ell i}} \frac{\partial x_{\ell i}}{\partial w_{\ell ij}}$$

$$= -\eta \frac{\partial \mathcal{E}}{\partial x_{\ell i}}\, y_{(\ell-1)j} = -\eta\, \delta_{\ell i}\, y_{(\ell-1)j}$$

where

$$\delta_{\ell i} \equiv \frac{\partial \mathcal{E}}{\partial x_{\ell i}} = \frac{\partial \mathcal{E}}{\partial y_{\ell i}} \frac{\partial y_{\ell i}}{\partial x_{\ell i}} = \frac{\partial \mathcal{E}}{\partial y_{\ell i}}\, f'(x_{\ell i})$$

Ultimately, it becomes more convenient to compute $f'(x_{\ell i})$ in terms of $y_{\ell i}$ so from this point I will replace $f'(x_{\ell i})$ with the equivalent $\frac{\mathrm{d}}{\mathrm{d}x}(y_{\ell i})$. For example, if $f(x_{\ell i})$ = tanh($x_{\ell i}$), then $f'(x_{\ell i}) = \mathrm{sech}^2(x_{\ell i})$ but also $f'(x_{\ell i}) = \frac{\mathrm{d}}{\mathrm{d}x}(y_{\ell i}) = 1 - y_{\ell i}^2$.

Now, the trick of backpropagation is to write

$$\frac{\partial \mathcal{E}}{\partial y_{\ell i}} = \sum_{k=1}^{I_{(\ell+1)}} \frac{\partial \mathcal{E}}{\partial x_{(\ell+1)k}} \frac{\partial x_{(\ell+1)k}}{\partial y_{\ell i}} = \sum_{k=1}^{I_{(\ell+1)}} \delta_{(\ell+1)k} \frac{\partial}{\partial y_{\ell i}} \sum_{j=1}^{I_\ell} w_{(\ell+1)kj}\, y_{\ell j}$$

$$= \sum_{k=1}^{I_{(\ell+1)}} \delta_{(\ell+1)k}\, w_{(\ell+1)ki}$$

and therefore we can find a general expression for the backpropagated error, and this is

$$\delta_{\ell i} = \frac{\mathrm{d}}{\mathrm{d}x}(y_{\ell i}) \sum_{k=1}^{I_{(\ell+1)}} \delta_{(\ell+1)k}\, w_{(\ell+1)ki}$$

In this way, we have managed to define a weight change in terms of feedforward values and a single feedback term, and the feedback term is defined in terms of the same term in the next highest layer, up to the special case of the output. In summary,

$$\Delta w_{\ell ij} = -\eta\, \delta_{\ell i}\, y_{(\ell-1)j} \qquad 1 \le j \le I_{(\ell-1)}, \qquad 1 \le i \le I_\ell, \qquad 1 < \ell \le L. \tag{18}$$

$$\delta_{\ell i} = \begin{cases} -2(d_i - y_{Li})\dfrac{\mathrm{d}}{\mathrm{d}x}(y_{Li}) & \ell = L, \qquad 1 \le i \le I_L \\ \dfrac{\mathrm{d}}{\mathrm{d}x}(y_{\ell i}) \displaystyle\sum_{k=1}^{I_{\ell+1}} \delta_{(\ell+1)k} w_{(\ell+1)ki} & 1 < \ell < L, \qquad 1 \le i \le I_\ell \end{cases} \tag{19}$$

B The Temporal Backpropagation Equations

The number of virtual taps per layer is a recursive equation which must be calculated for each layer starting with the output and working back to the input.

$$T_\ell^{\mathrm{ACC}} = \begin{cases} 1 & \ell = L \\ T_{(\ell+1)}^{\mathrm{ACC}} + T_\ell - 1 & 1 \le \ell < L \end{cases} \tag{20}$$

The amount of change in a weight between the current iteration and the next is given by

$$\Delta w_{\ell ijt} = -\eta \sum_{n=1}^{T_\ell^{\mathrm{ACC}}} \delta_{\ell in}\, y_{(\ell-1)j(t+n-1)} \qquad 1 \le \ell \le L, \qquad 1 \le i \le I_\ell, \qquad 1 \le j \le I_{(\ell-1)}, \qquad 1 \le t \le T_{(\ell-1)} \tag{21}$$

and should be compared with the static form (19) given above. The backpropagated error term needed to complete this equation is

$$\delta_{\ell in} = \begin{cases} -2(d_i - y_{Li1})\dfrac{\mathrm{d}}{\mathrm{d}x}(y_{Li1}) & \ell = L, \qquad 1 \le i \le I_\ell, \\ & n = 1 \\ \dfrac{\mathrm{d}}{\mathrm{d}x}(y_{\ell in}) \displaystyle\sum_{k=1}^{I_{(\ell+1)}} \sum_{m=\max(1, n-T_{(\ell+1)}^{\mathrm{ACC}}+1)}^{\min(n, T_\ell)} \delta_{(\ell+1)k(n-m+1)} w_{(\ell+1)kim} & 1 < \ell < L, \\ & 1 \le i \le I_\ell, \\ & 1 \le n \le T_\ell^{\mathrm{ACC}} \end{cases} \tag{22}$$

and should be compared to equation (18) above. Finally, the basic feedforward relationship is given by

$$y_{\ell is} = f(x_{\ell is}) \qquad 1 < \ell \leq L, \qquad 1 \leq i \leq I_\ell, \qquad 1 \leq s \leq T_\ell \tag{23}$$

In these equations, L represents the total number of network layers, I_ℓ represents the number of filters per layer, T_ℓ represents the number of taps per filter in a given layer, the value y_{Li1} is the output per output node, d_i is the desired output per output node, and $f(\cdot)$ is assumed to be a limiting function such as a sigmoid.