

## Spatiotemporal Connectionist Networks: A Taxonomy and Review

**Stefan C. Kremer**

*Guelph Natural Computation Group, Department of Computing and Information Science, University of Guelph, Guelph, Ontario, N1G 2W1 Canada*  
skremer@uoguelph.ca

This article reviews connectionist network architectures and training algorithms that are capable of dealing with patterns distributed across both space and time—spatiotemporal patterns. It provides common mathematical, algorithmic, and illustrative frameworks for describing spatiotemporal networks, making it easier to compare and contrast their representational and operational characteristics. Computational power, representational issues, and learning are discussed. In additional references to the relevant source publications are provided. This article can serve as a guide to prospective users of spatiotemporal networks by providing an overview of the operational and representational alternatives available.

### 1 Introduction ---

In this article, we propose a new taxonomy for characterizing connectionist approaches to learning input-output relationships in which data are distributed across both space and time—spatiotemporal patterns. The ability to learn these types of relationships is directly applicable to dynamical system identification and control (Alippi & Piuri, 1996), syntactic pattern recognition (Fu, 1982), and grammatical induction (Angluin & Smith, 1983).

Throughout the article, we maintain a problem-driven philosophy rather than an architecture-driven approach, though we do restrict ourselves broadly to the connectionist paradigm. In this regard, the taxonomy presented is not a taxonomy of recurrent networks but rather of spatiotemporal networks. Since there are certainly recurrent networks that are also spatiotemporal, these are discussed, yet other recurrent networks, such as those that are designed to settle to a stable fixed point, are omitted. Conversely, we describe a number of nonrecurrent networks that are nonetheless used to classify, identify, or generate spatiotemporal sequences. The problem-driven philosophy differentiates the taxonomy presented here from previously described architectural approaches (Mozer, 1994; Horne & Giles, 1995; Tsoi & Back, 1994), which we examine in section 3.2.

The new taxonomy is based along the four fundamental design decisions that the designer of any spatiotemporal sequence processing system must make: (1) selecting a representation for state transition functions, (2) selecting possible output functions, (3) defining the initial state of the system, and (4) choosing an algorithm for parameter update. This design decision approach ensures that the scope of the taxonomy is broad enough to accommodate any current or future spatiotemporal connectionist networks, something that could not be accomplished with an architecture-based taxonomy. We apply our taxonomy to the leading connectionist networks and describe their relation to each other.

In section 2, we define the notation and terminology used. In section 3, we identify features of good taxonomies, survey several existing categorization schemes, and identify some goals for the development, in section 4, of a new taxonomy possessing the ideal features identified. This new taxonomy is structured along four design decisions: (1) how the state of the system is computed (discussed in section 5), (2) how the output of the system is computed (section 6), (3) how the system is initialized (section 7), and (4) how the system is trained (section 8). In section 10, we present some conclusions.

## 2 Terminology and Mathematical Preliminaries ---

**2.1 Spatiotemporal Connectionist Networks.** We define a *spatiotemporal connectionist network* (STCN) as a parallel distributed information processing structure that is capable of dealing with input data presented across time as well as space. One might argue that this definition is too informal and describes too diverse a set of computational systems to be of practical use; however, it is difficult to assign a more precise mathematical definition to STCNs that does not contradict the common use of the term to describe a variety of very diverse architectures. In this article, we provide precise mathematical descriptions of the most popular classes of STCNs and their constituent components, as opposed to dwelling on the formulation of one definition for all STCNs. This will allow us to develop formal descriptions of STCNs with specific properties, without unnaturally restricting the definition of STCN in general. The following formalisms serve as a basis for the subsequent description of specific STCN components.

**2.2 A Note About Notation.** The field of STCNs has attracted researchers from a wide range of specializations, including computer science, engineering, linguistics, and psychology. Perhaps this partially accounts for the inconsistent notations employed throughout the existing STCN literature. In this article, we have attempted to use the more common mathematical representations, while maintaining consistency across all architectures described. We begin by introducing these mathematical formalisms.

It is convention in algebra to interpret a vector  $\vec{x} = (x_1, x_2, x_3, \dots)$ , as being equivalent to the column matrix

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix}.$$

(Note that we index our vector components beginning at 1 and not at 0.) This translation of vectors to matrices will prove practical, given the frequency with which we will be computing a vector (e.g.,  $\vec{z}$ ) based on a matrix product of a two-dimensional matrix (e.g.,  $\mathbf{W}$ ) and another vector (e.g.,  $\vec{x}$ ), as in the equation

$$z_j = \sum_i \mathbf{W}_{ji} \cdot x_i. \quad (2.1)$$

Note the order of the indices of  $\mathbf{W}$ :  $\mathbf{W}_{ji}$  measures the effect of component  $i$  of vector  $\vec{x}$  on component  $j$  of vector  $\vec{z}$ , and not vice versa. This allows us to write the equation for the vector  $\vec{z}$  as simply

$$\vec{z} = \mathbf{W} \cdot \vec{x}. \quad (2.2)$$

More frequently, we will apply a nonlinearity to each component of the product  $\vec{z}$ . We denote this by a vector function,  $\vec{f}(\cdot)$ , which maps each component of one vector to another according to the scalar function,  $f(\cdot)$ . Typically, the nonlinearity employed is the sigmoid,

$$\sigma(z_i) = \frac{1}{1 + e^{-z_i}}, \quad (2.3)$$

in which case, we write  $\vec{\sigma}(\vec{z})$  for the vector function. The results of forward-propagating one layer of activations in a network, through a layer of weighted connections and a nonlinearity to compute the next layer's activations, can now be written

$$\vec{y} = \vec{\sigma}(\mathbf{W} \cdot \vec{x}). \quad (2.4)$$

In addition to frequent matrix vector multiplications, we will commonly use vectors whose elements consist of the concatenation of the elements of two smaller vectors. In order to simplify the description of these vectors, we will use  $\oplus$  to represent a function mapping two vectors, such as  $\vec{b}$  and  $\vec{c}$  in vectors spaces  $B$  and  $C$ , to a new vector  $\vec{a}$  in the vector space defined by

the Cartesian product of the spaces,  $B \times C$ . The operation is defined such that the representation of vector  $\vec{a}$  is formed by concatenating the elements of vector  $\vec{c}$  onto elements of vector  $\vec{b}$ . Specifically, if  $\vec{a} = \vec{b} \oplus \vec{c}$ , then the first elements of  $\vec{a}$  are all the elements of  $\vec{b}$  and the remaining elements of  $\vec{a}$  are all the elements of  $\vec{c}$ . Mathematically,

$$a_i = \begin{cases} b_i & \text{if } i \leq \dim(\vec{b}) \\ c_{i-\dim(\vec{b})} & \text{if } i > \dim(\vec{b}) \end{cases} \quad (2.5)$$

and

$$\dim(\vec{a}) = \dim(\vec{b}) + \dim(\vec{c}), \quad (2.6)$$

where  $\dim(\vec{a})$  represents the dimensionality of  $\vec{a}$ .

We will further require an inverse of this operator, capable of extracting a smaller vector from a larger vector. To do this, we define the use of square parentheses in the expression  $\vec{b} = \vec{a}[i..j]$  such that vector  $\vec{b}$  is equal to a vector whose components are equal to components  $i$  through  $j$  of vector  $\vec{a}$ ,  $i, j \in \{1, 2, 3, \dots, \dim(\vec{a})\}$  and  $i < j$ . More precisely we require that the  $k$ th component of vector  $\vec{b}$  is equal to the  $(i + k - 1)$ th component of  $\vec{a}$ ,

$$b_k = a_{i+k-1}, \quad (2.7)$$

and that the dimensionality of vector  $\vec{b}$  is equal to the number of terms in the inclusive sequence from  $i$  to  $j$ :

$$\dim(\vec{b}) = j - i + 1. \quad (2.8)$$

It is now possible to invert the  $\oplus$  operator,

$$\vec{a} = \vec{b} \oplus \vec{c}, \quad (2.9)$$

such that

$$\vec{b} = \vec{a}[1.. \dim(\vec{a}) - \dim(\vec{c})], \quad (2.10)$$

and

$$\vec{c} = \vec{a}[(\dim(\vec{b}) + 1).. \dim(\vec{a})]. \quad (2.11)$$

The vector notation presented here allows us to represent patterns distributed across the processing units of an STCN. Each vector component denotes the activation value of a particular processing element in an STCN.

**2.3 Input Dimensions.** In STCNs, input and output patterns can vary across time as well as space. For the purposes of simulation on digital computers, it is useful to discretize the temporal dimension by sampling at regular intervals and consider a system in which time proceeds by intervals of  $\Delta t$ . We shall use the symbol  $t$  to represent a particular point in time, where  $t \in \{0, \Delta t, 2\Delta t, 3\Delta t, \dots\}$ . In this formulation,  $\Delta t$  can be considered to be the unit of measure for the quantity  $t$ , and thus it is simplest to omit the units and express  $t$  simply as a member of the set of whole numbers,  $t \in \{0, 1, 2, 3, \dots\}$ . It should be noted that even a continuous-time system, simulated on a digital computer, is usually converted into a set of simple first-order difference equations, making it formally identical to a discrete-time network. Furthermore, barring high-frequency components in the network's input (relative to  $\Delta t$ ), a continuous-time STCN can be precisely duplicated by a discrete-time system (Pearlmutter, 1995).

The dimension of time in STCNs differs from the spatial dimensions in conventional connectionist networks. Components of an input pattern distributed across space (i.e., across various input nodes) can all be accessed at the same time. However, only the current component of patterns distributed across time is accessible at any given instant. We assume that the observable world for an STCN at the instant  $t$  is described by an input vector,  $\vec{x}(t)$ . Systems that use older input vectors (e.g., time-delay networks; see below) can still be considered, but such a system must include a mechanism for storing historical input vectors (typically a queue of previous values). This vector is supplied to the STCN at time  $t$  by setting the activation values of the input units of the STCN to the components of the vector. Thus, the input vector can be considered a stimulus.

It is important to note the difference between information encoded across the input vector  $\vec{x}(t)$  and information encoded in different input vectors  $\vec{x}(0), \vec{x}(1), \dots$ , presented to the network at different times. Values of the former can be accessed in parallel, while the latter must be accessed sequentially (in a forward direction). This makes it the responsibility of the network to store any information contained in  $\vec{x}(t)$  until it is required at a later time.

**2.4 Long- and Short-Term Memory in Spatiotemporal Connectionist Networks.** Both conventional and spatiotemporal connectionist networks are equipped with memory in the form of connection weights, denoted by one or more matrices, depending on the number of layers of connections in the network. These are typically updated after each training step and constitute a memory of all previous training. In the sense that this memory extends back past the current input pattern, all the way to the first training step, we shall refer to the weights as long-term memory. Once a connectionist network has been successfully trained, this long-term memory remains fixed during the operation of the network.

In addition to these weight matrices, some networks also use other trainable parameters. These may represent the connectivity scheme of the net-

work (e.g., recurrent cascade correlation; see section 8.9), the types of transmission delays associated with the connections (e.g., gamma networks; see section 5.5), or the initial activation values of the internal processing elements (see section 8.7). These parameters are also part of the long-term memory. We defer a more detailed discussion of the details of these types of parameters until later in the article. For now, we shall only define  $\mathfrak{W}$  to denote an  $n$ -tuple representing all the adaptable parameters of a network.  $\mathfrak{W}$  includes one or more weight matrices and may include connectivity scheme parameters, transmission delay parameters, and initial activation values depending on the type of network.

The distinguishing characteristic of STCNs is that they also include a form of short-term memory. It is this memory that allows these networks to deal with input and output patterns that vary across time and thus defines them as STCNs. Conventional connectionist networks compute the activation values of all nodes at time  $t$  based only on the input at time  $t$ . By contrast, in STCNs the activations of some nodes at time  $t$  are computed based on activations at time  $t - 1$ , or earlier. These activations serve as a short-term memory. We use the state vector,  $\vec{s}(t - 1)$ , to represent the activations at time  $t - 1$  of those nodes that are used to compute the activations of other nodes at time  $t$ , that is, state nodes. Unlike the long-term memory, which remains static once training is completed, the short-term memory is continually recomputed with each new input vector during both training and operation. This is due to the fact that long-term memory is stored in connection weights (which are updated only during training) while short-term memory is represented by node activations (which are computed with each time-step, even after training).

**2.5 Output, Teaching, and Error.** Finally, it is necessary to specify a representation for the response of the network to its stimuli. Like the traditional models, STCNs encode their response in the activations of a special set of units called output units. Thus, we represent the output of an STCN by a vector,  $\vec{y}(t)$ . Most connectionist architectures learn by computing the difference between their response and a teacher-supplied desired (or ideal) response and adjusting their long-term memory accordingly. We denote the desired response by another vector,  $\vec{y}^*(t)$ . The difference between the desired output vector and the actual output vector is the error vector  $\vec{E}(t) = \vec{y}^*(t) - \vec{y}(t)$ , and the total network error,  $\varepsilon$ , is defined as one-half of the square of the magnitude of this vector:

$$\varepsilon = \sum_{t \geq 0} \left\{ \frac{1}{2} \|\vec{E}(t)\|^2 \right\}. \quad (2.12)$$

The total error,  $\varepsilon$ , is a measure of the overall performance (or lack thereof). It is this quantity that is minimized by gradient descent during training (see section 8).

Table 1: Notation.

$\vec{f}(\cdot)$	Generic vector nonlinearity
$\vec{\sigma}(\cdot)$	Sigmoid vector nonlinearity
$\vec{b} \oplus \vec{c}$	Concatenation of components of vectors $\vec{b}$ and $\vec{c}$
$\text{dim}(\vec{a})$	Dimensionality of vector $\vec{a}$
$\vec{a}[i..j]$	Vector consisting of components $i$ through $j$ of vector $\vec{a}$
$\Delta t$	Units of time
$\vec{x}(t)$	Input to the network at time $t$
$\mathbf{W}$	Weight matrix
$\mathcal{W}$	Adaptable parameters of network
$\vec{s}(t)$	State vector at time $t$
$\vec{y}(t)$	Output of network at time $t$
$\vec{y}^*(t)$	Desired response at time $t$
$\vec{E}(t)$	Error vector at time $t$
$\varepsilon$	Total error scalar

**2.6 Summary.** In this section, we have introduced a number of symbols to represent various properties and components of STCNs. This notation is summarized in Table 1 for easy reference.

### 3 Evaluation of Taxonomies

**3.1 Criteria of a Good Taxonomy.** Before presenting our taxonomy of STCNs for spatiotemporal processing, we briefly discuss three general properties of good taxonomies: (1) descriptive adequacy (it must classify all objects in its domain), (2) simplicity (it should be relatively simple to classify and name any of the objects in the domain), and, most important, (3) predictive power (objects in the taxonomy with similar classifications should possess similar properties, and objects in the taxonomy with differing classifications should have differing properties).

A good taxonomy balances simplicity with predictive power while maintaining descriptive adequacy. In other words, it must have a level of discrimination that is not so low that it groups dissimilar objects together or so high that the number of groups required to encompass all objects becomes unwieldy. This can best be accomplished if the taxonomy describes features along multiple orthogonal dimensions rather than along one single dimension. If multiple dimensions are available, then different yet similar objects can be classified as having the same feature along one or more dimensions,

Table 2: Horne and Giles' Taxonomy.

<p><b>Observable states</b></p> <p>Narendra and Parthasarathy, (1990)</p> <p>TDNN (Lang, Waibel, &amp; Hinton, 1990)</p> <p>Gamma network (de Vries &amp; Principe, 1992b)</p> <p><b>Hidden dynamics</b></p> <p>Single-Layer</p> <p>First order</p> <p>High order (Giles et al., 1990)</p> <p>Bilinear</p> <p>Quadratic (Watrous &amp; Kuhn, 1992b)</p> <p>Multilayer</p> <p>Robinson and Fallside (1988)</p> <p>Simple recurrent networks (Elman, 1988)</p> <p>Local feedback</p> <p>Back and Tsoi (1991)</p> <p>Frasconi, Gori, and Soda (1992)</p> <p>Poddar and Unnikrishnan (1991b)</p>
--

while having different features along other dimensions. By contrast, in a single-dimension taxonomy, the two objects must be classified as either the same or different.

**3.2 Previous Work.** Horne and Giles (1995), in an article comparing the performance of different architectures, have developed a simple taxonomy for STCNs (see Table 2). Their approach focuses on partitioning the space of existing STCN architectures. The first partition separates those networks whose state representations are encoded in input and output units only from those networks whose state representations are encoded in hidden units. Horne and Giles refer to the former class as networks with observable states and the latter as networks with hidden dynamics. Networks with

observable states include Narendra and Parthasarathy's (1990) networks; Lang, Waibel, & Hinton's time delay neural networks (TDNN) (1990); and de Vries and Principe's (1992b) gamma networks. The class of networks with hidden dynamics is further partitioned into single-layer, multilayer, and local feedback networks.

Horne and Giles's taxonomy considers only the issue of connectivity and not those concerning adaptation. Nor does it describe a number of popular classes of spatiotemporal networks including recurrent cascade correlation (Fahlman, 1991), recursive autoassociative memory (Pollack, 1989, 1990), autoassociative recurrent network (Maskara & Noetzel, 1992), connectionist pushdown automaton (Giles, Sun, Chen, Lee, & Chen, 1990), connectionist Turing machine (Williams & Zipser, 1989b), and second-order constructive learning (Giles et al., 1995). This limits the taxonomy's descriptive adequacy and predictive power.

Tsoi and Back (1994) have developed a taxonomy designed specifically for locally recurrent, globally feedforward networks, a subset of STCNs. These are networks in which all connections are feedforward with the exception of one temporal self-looping connection for each node. Tsoi and Back base their taxonomy on the processing performed by the feedback connections used and which value is delayed. They refer to these two considerations as synapse type and feedback location, respectively. The feedback location is subdivided along three dimensions depending on which combination of the nodes' previous activation values—the nodes' previous net inputs or the previous output value—is transmitted by the local feedback synapse. Thus, Tsoi and Back's taxonomy is composed of four dimensions: synapse type (which can assume a value of simple or dynamic), synapse feedback (which can be yes or no), activation feedback (which can be yes or no), and output feedback (which can be yes or no). A simple synapse corresponds to a feedback connection with a single scalar weight, while a dynamic synapse has a linear transfer function. These dimensions are illustrated in Table 3. Tsoi and Back's taxonomy is capable of dealing only with locally recurrent, globally feedforward networks, a small proportion of STCNs.

Mozer (1994) has developed a taxonomy for STCNs, which is illustrated in Table 4. It is based on the assumption that every STCN consists of two mechanisms: a short-term memory and a predictor. The short-term memory computes the state of the STCN, and the second mechanism uses the state to compute output. Mozer characterizes the operation of the short-term memory of STCNs along the dimensions of content and form (which define how the short-term memory is computed), and adaptability (which defines how changes in the computation of the short-term memory are made during the adaptation process). Mozer notes that the predictor component of an STCN is always a feedforward network. The table illustrates the dimensions Mozer used to describe STCNs.

Despite being the most sophisticated and comprehensive taxonomy of STCNs developed to date, it has two limitations. The first concerns sacri-

Table 3: Tsoi and Back's Taxonomy.

<b>Synapse type</b>
Simple
Dynamic
<b>Feedback at synapse</b>
Yes
No
<b>Feedback at activation</b>
Yes
No
<b>Feedback at output</b>
Yes
No

ficing predictive power for simplicity. Mozer's taxonomy applies only to the short-term memory component of the network. He does not describe the predictor component beyond stating that it is a network. This means that networks with different predictor components (e.g., multilayer versus single-layer versus no-layer, or first-order connections versus second-order connections) are grouped in the same category. Since it is well-known that the number of layers and order of connections greatly affect what a connectionist network can compute (see Lippmann, 1987), predictive power is limited by not categorizing STCNs based on their predictor components. The same argument applies to the short-term memory component. Mozer uses the term *transformed input and state* memories to encompass a very broad range of connectionist networks with varying layer numbers and connectivity schemes, including Fahlman's recurrent cascade correlation networks and Giles's second-order networks, two approaches that are vastly different in approach and computational power (see Giles et al., 1995; Kremer, 1996a, 1996b).

The second problem with Mozer's taxonomy is that it also sacrifices too much descriptive adequacy in favor of simplicity. Mozer himself identifies one interesting and very popular architecture type of STCN, which he calls the *standard* architecture (e.g., Elman, 1990a; Mozer, 1989) that does not

Table 4: Mozer's Taxonomy.

<b>Content</b>
Input (I)
Transformed input (TI)
Transformed input and state (TIS)
Output (O)
Transformed output (TO)
Transformed output and state (TOS)
<b>Form</b>
Delay line
Exponential trace
Gamma
<b>Adaptability</b>
Static
Adaptive

fit within his classification scheme. Although the standard architecture is very similar to TIS, there are differences between the two approaches (see Figure 1). Note that the standard architecture is like Mozer's transformed input and state, except that the first two hidden layers have been collapsed into a single layer. The standard architecture does not fit anywhere else in the different possibilities for memory contents discussed by Mozer either.

Another example of sacrificing descriptive adequacy concerns the networks of Narendra and Parthasarathy (1990). These networks compute their outputs based on a history of both input and output patterns. In this sense, these networks could be considered as having input and output, or IO, memories. Yet Mozer's taxonomy does not include this type of memory content. Nor does it adequately describe different systems of

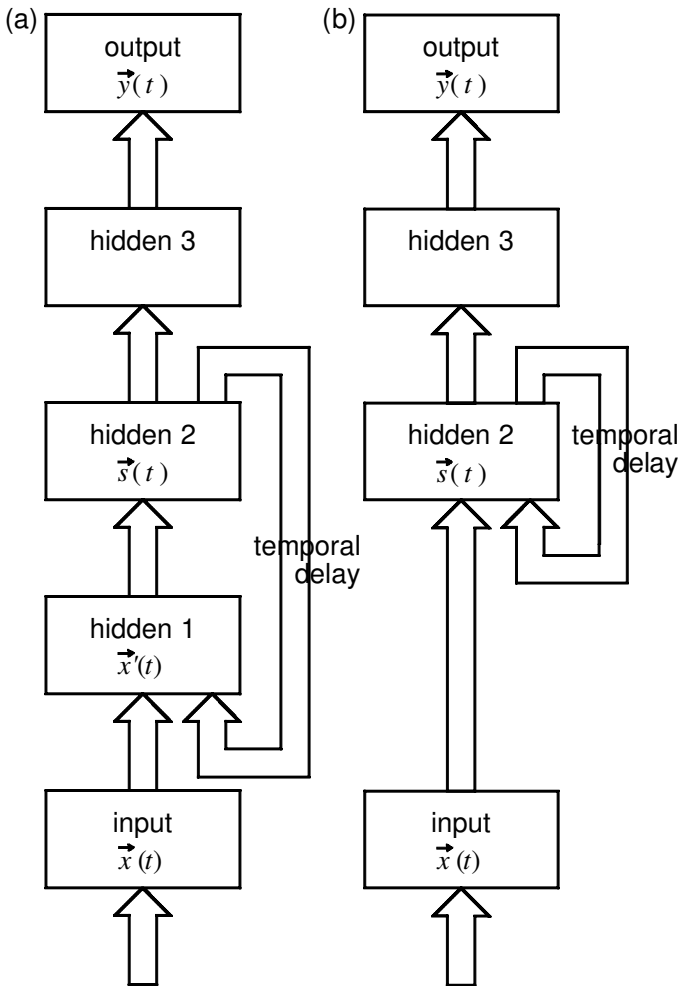


Figure 1: TIS and the standard architecture. (a) TIS memory architecture. (b) Standard architecture. (Adapted from Mozer, 1989).

memory adaptability. Elman's SRN algorithm, for example is described as lying somewhere between static and adaptive. This seems strange, since Elman's system clearly satisfies Mozer's criterion for adaptiveness ("the neural net can adjust memory parameters"; Mozer, 1994, p. 252). Yet Elman's algorithm is clearly different—and perhaps less effective at adapting (Servan-Schreiber, Cleeremans, & McClelland, 1988) than the conventional recurrent adaptation algorithms—because it uses a truncated version of gra-

dient descent, which we will describe in greater detail later. Fahlman has developed yet another adaption algorithm that does not fit into Mozer's taxonomy. Fahlman's networks adapt not only parameters like weights but also the number of nodes in the network. This is an important distinction, since it implies that Fahlman's networks are capable of increasing their own memory capacity, which other STCNs are not. Clearly this type of adaption will have a profound impact on how these networks perform on temporal problems.

Two other classification systems have been developed in articles on spatiotemporal pattern recognition systems: that of Ghosh and Deuser (1995) focuses on classifying different approaches in the context of recognizing sonar sequences, while that of Maren (1990) looks at a broader range of applications to dynamical systems classification. Both approaches use a fairly coarse-grained approach similar to that of Mozer and share a number of Mozer's limitations.

Table 5 summarizes these previous approaches. When Mozer and Horne and Giles first developed their classification schemes, they provided simple, descriptively adequate, and predictively powerful frameworks for describing some of the leading STCN approaches of the time. Tsoi and Back also developed a simple taxonomy addressing a popular subset of STCNs. The recent flurry of activity in the field of STCN research, however, has dated these ways of classifying networks. The problems that these three taxonomies exhibit in the light of new STCN designs can all be traced to a common cause: all are attempts to classify existing STCNs—they use an architectural approach. Since it is impossible to predict future STCN developments, new designs are difficult to fit into the categories defined earlier. One solution to this problem is to omit the new STCN designs from the taxonomy; however, this results in a lack of descriptive adequacy. An alternative solution is to place new designs into the closest possible categories (even if they do not fit well). This situation tends to degrade the predictive of the taxonomy since it will be more difficult to make predictions based on tenuous categorizations. Another option is to add a new category for the new network design, but this tends to decrease simplicity due to a proliferation of categories. A final option is to develop a new taxonomy. Although a new taxonomy will never be able to withstand all future developments, it can be designed to anticipate future developments (especially hybrid approaches that mix components of existing systems). This article attempts to define such a taxonomy.

#### **4 A New Taxonomy for Spatiotemporal Connectionist Networks** \_\_\_\_\_

An alternative to attempting to classify existing STCNs based on their respective properties is to examine what all STCNs have in common. We develop our taxonomy around the computational components of spatiotemporal systems (examining the goals of computation) rather than around

Table 5: Summary of Taxonomies.

Taxonomy	Dimensions Covered	Architectures Covered	Categories
Horne and Giles (1995)	Connectivity only, not adaptation	Many popular classes of networks missed	Good
Tsoi and Back (1994)	Good	Covers only locally recurrent, globally feedforward networks	Good
Mozer (1994)	Short-term memory only, not predictor	Some networks do not fit into any categories	Very broad categories
Ghosh and Deuser (1995)	Good	Focuses on networks for sonar sequences only	Very broad categories
Maren (1990)	Good	Architectures for dynamical systems classification	Very broad categories

the representations and algorithms of particular STCNs. This approach is related to Marr's (1982) trilevel hypothesis which describes information processing systems at three levels: (1) the computational theory level, which describes the goal of the computation; (2) the representation and algorithm level, which describes how the computation can be implemented; and (3) the hardware implementation level, which describes the physical realization of the computation. While other taxonomies are structured around the representation and algorithm level, focusing on how computations are implemented, we base our taxonomy around the computational theory level.

Our taxonomy is based on the perspective of anyone wishing to develop a spatiotemporal processing system. Those in this position are forced to answer two basic questions. The first is, How does one select, limit, and represent the set of systems that can be induced? This set of systems is defined as the hypothesis space of the processing system. The second ques-

tion is, What algorithm is used to identify, from the hypothesis space, the particular system that best suits the training data? This is the search algorithm. By basing our taxonomy on these two fundamental questions that need to be answered for any search algorithm, we can group STCNs into categories without placing any restrictions on the descriptive adequacy of the taxonomy.

We now break down the first question—How does one represent the hypothesis space?—into two components. For STCNs, a hypothesis represents a particular network with a specific set of trainable parameters  $\mathfrak{W}$ , or, equivalently, the spatiotemporal input-output relationship defined by these parameters. Every spatiotemporal system can be defined by two components: one that computes the new internal state of the system based on the input and previous state, and one that computes the output of the system based on the new state. Thus, it is natural to break the process of representing the hypothesis space along these lines and identify two new questions that must be answered. The first is, How does one represent the possible computations that could be performed in order to compute the system's state? The second is, How does one represent the possible computations that could be performed in order to compute the system's output? These two questions together are equivalent to the original question (What is the hypothesis space?). This is because the representation of any spatiotemporal mechanism can be broken into state and output components and because these two components completely define the spatiotemporal processing.

The second question, What is the search algorithm? can also be broken down into two components. The search algorithm effectively defines operators for moving the induction system from one point in the (parameter) hypothesis space to another. This too can be broken into two subquestions: What is the initial state of the network's parameters? and, How are the parameters updated in response to the training data?

This leaves us with four fundamental questions that must be answered by any STCN design. These questions, illustrated in Table 6, form the basis dimensions of our taxonomy. Any spatiotemporal system can be defined by its position in this four-dimensional space. Specific coordinates along each dimension represent specific answers to the four questions. A set of four such coordinates completely defines an STCN.

Each dimension of the taxonomy can be described as a generic function, and the specific instantiation of each function will determine the behavior of the STCN. The first function defines how the state vector,  $\vec{s}(t)$ , is computed, based on the previous state vector,  $\vec{s}(t-1)$ , the current input vector,  $\vec{x}(t)$ , and the network's trainable parameters,  $\mathfrak{W}$ . This function will be denoted  $f_s(\vec{s}(t-1), \vec{x}(t), \mathfrak{W})$ . The second function computes the output vector,  $\vec{y}(t)$ , based on the current state vector,  $\vec{s}(t)$ , and the parameters of the STCN,  $\mathfrak{W}$ . It will be denoted  $f_y(\vec{s}(t), \mathfrak{W})$ . The third function computes the initial state of the STCN's parameters based on available a priori information about

Table 6: Four basic questions to Be answered by Any STCN and the Mathematical Functions Describing the Dimensions.

Question	Function
How does one represent the possible computations that could be performed in order to compute the system's state?	$f_{\bar{s}}(\bar{s}(t-1), \bar{x}(t), \mathfrak{W})$
How does one represent the possible computations that could be performed in order to compute the system's output?	$f_{\bar{y}}(\bar{s}(t), \mathfrak{W})$
What is the initial state of the system?	$f_{\mathfrak{W}_0}$ (a priori info)
How does one adapt the trainable parameters of the system?	$f_{\Delta \mathfrak{W}}(\bar{E}(t), \mathfrak{W}, \bar{x}(\cdot))$

the problem to be solved. It is denoted  $f_{\mathfrak{W}_0}$ . The fourth and final function computes the new value of the parameters of the STCN, in response to the error,  $\bar{E}(t)$ , the current parameters,  $\mathfrak{W}$ , and the input history,  $\bar{x}(\cdot)$ , of the STCN. This function is denoted  $f_{\Delta \mathfrak{W}}(\bar{E}(t), \mathfrak{W}, \bar{x}(\cdot))$ . These functions are illustrated opposite their corresponding dimensions in Table 6. Note that we have as yet made absolutely no assumptions about the representation or algorithms employed by each of the four functions. We have focused only on the goal of the computation, thus working at Marr's first level rather than the second. In the following sections, we provide example instantiations of the functions that correspond to the representations and algorithms employed in a variety of popular STCNs.

Using these four functions, it is now possible to describe the operation of any STCN. This operation is described in pseudocode in Table 7. In a sense, the pseudocode can be thought of as a mathematical definition of STCN paralleling the informal definition provided in section 2.1. Note that at this point, we have developed a general framework that is applicable to any STCN. By specifying the instantiations of the four functions described above, it is possible to define points along the four dimensions of the taxonomy and hence specific STCN induction systems.

The algorithm presented assumes that weight updates are performed every time a target is available. Sometimes networks are trained in a batch mode, in which case error values are accumulated in step 11, and step 12 is executed only at the end of the presentation of a fixed number of sequences, called an epoch. The same sequence may then be presented again, or additional strings may be added before the next epoch begins.

Table 7: Generic Algorithm for STCNs.

---

00	<b>algorithm</b> STCN	
01	$\mathbb{W} \leftarrow f_{\mathbb{W}_0}$ (a priori info)	/* initialize */
02	<b>do</b>	/* begin training */
03	$t \leftarrow 0$	/* reset time */
04	<b>do</b>	/* begin sequence */
05	$t \leftarrow t + 1$	/* increment time */
06	$\bar{x}(t) \leftarrow$ environment	/* determine input */
07	$\bar{s}(t) \leftarrow f_{\bar{s}}(\bar{s}(t-1), \bar{x}(t), \mathbb{W})$	/* update state */
08	$\bar{y}(t) \leftarrow f_{\bar{y}}(\bar{s}(t), \mathbb{W})$	/* determine output */
09	<b>if</b> target $\bar{y}^*(t)$ available	
10	$\bar{E} \leftarrow \bar{y}^*(t) - \bar{y}(t)$	/* compute error */
11	$\varepsilon \leftarrow \left\{ \frac{1}{2} \ \bar{E}\ ^2 \right\}$	/* compute scalar */
12	$\mathbb{W} \leftarrow \mathbb{W} + f_{\Delta \mathbb{W}}(\bar{E}, \mathbb{W}, \bar{x}(\cdot))$	/* update parameters */
13	<b>while not</b> end of sequence	
14	<b>while</b> $t < t_{max}$ <b>and</b> $\varepsilon > threshold$	/* no solution found and not out of time */

---

Note: Particular network designs differ only in their instantiations of the four fundamental functions.

## 5 Computing the State Vector

---

We begin by examining instantiations of the state vector function,  $f_{\bar{s}}(\bar{s}(t-1), \bar{x}(t), \mathbb{W})$  used in existing STCNs. In particular, we discuss thirteen popular alternatives: window in time memories, NARX memories, time-delay neural network memories, feedforward exponential decay memories, gamma memories, Jordan memories, local feedback memories, connectionist infinite impulse response filter memories, first-order context memories, second-order context memories, long short-term memories, connectionist push-down automata memories, and connectionist Turing machine memories.

**5.1 Window in Time (WIT) Memory.** The window in time (WIT) memory approach to computing the state vector is one of the first types of short-term memory in spatiotemporal networks. Its best-known implementation is NETalk (Sejnowski & Rosenberg, 1987, 1988; Rosenberg & Sejnowski, 1986), but it has also been used by a variety of other authors including Lang et al. (1990) and Waibel, Hanazawa, Hinton, Shikano, & Lang, (1989). The short-term memory vector,  $\bar{s}(t)$ , is always computed in the same manner throughout the training process and represents a finite, so-called window in time on the input vectors. This window contains a finite history of input vectors. If we consider a window with width  $n$  input vectors, then the state

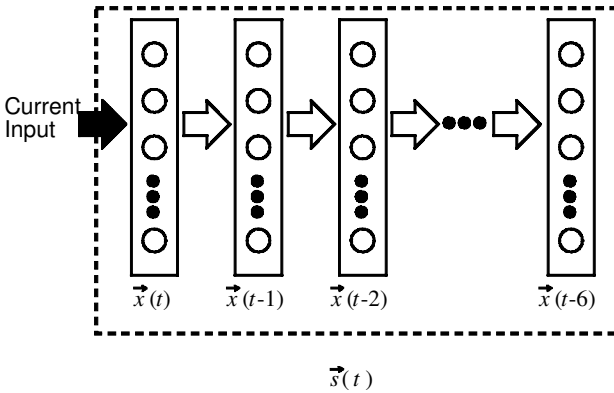


Figure 2: Short-term memory of WIT networks.

of the STCN is given by

$$\vec{s}(t) = \vec{x}(t) \oplus \vec{x}(t - 1) \oplus \dots \oplus \vec{x}(t - n + 1), \tag{5.1}$$

and the function to compute state recursively is defined as:

$$f_{\vec{s}}(\vec{s}(t - 1), \vec{x}(t), \mathbb{W}) \equiv \vec{x}(t) \oplus \vec{s}(t - 1)[1..(n - 1) \cdot \dim(\vec{x}(t))]. \tag{5.2}$$

Figure 2 illustrates a WIT memory. The illustrative convention used for this and subsequent images is described in Table 8.

At time  $t$ , the input vector,  $\vec{x}(t)$ , which corresponds to the current input, is added to the left side of the state vector,  $\vec{s}(t)$  (indicated by the dashed box). All other vectors are shifted right. It is critical to note that the equation to compute the next state of the STCN does not make use of the trainable parameters,  $\mathbb{W}$ . This means that the function remains completely static; the state vector is computed the same way every time.

The WIT architecture has been widely used in a number of applications but is best known for the NETalk application. A WIT architecture with  $n = 6$  is also used to predict the dynamics of a plant in Back and Tsoi (1991), although the authors present the network as an example of a more general architecture called a FIR MLP (see section 5.9 below). The computational power of these networks has been identified as being equivalent to definite machines (Kohavi, 1978, in Giles & Horne, 1994).

**5.2 Nonlinear Autoregressive with Exogenous Inputs Memory.** A variation on the WIT memory is to use two temporal windows: one window on the input (as in WIT memories) and a second on the output produced by the network. Because of their similarities to nonlinear autoregressive with exogenous inputs (NARX) models, STCNs with this type of memory have

Table 8: Figure Key.

- 
- Nodes are represented by hollow circles.
  - Ellipses (drawn using three solid circles) indicate the possibility of arbitrary numbers of nodes forming an activation vector or arbitrary numbers of activation vectors contributing to a state vector.
  - Units forming input and output activation vectors are grouped in solid-lined rectangles.
  - Units forming state activation vectors are grouped in dotted rectangles.
  - Connections between nodes are represented by lines with information flow always pointing in an upward direction; wherever practical and unless otherwise indicated, all connections are shown.
  - Flow through time is illustrated by thick, hollow arrows pointing from recent activation values to older activation arrows.
  - Activation values can be thought of as being copied once per time step through the thick, hollow arrows.
  - Activation values that are set external to the nodes illustrated (i.e., from output nodes, Turing machine controller or continuous stack) are indicated by solid arrows
- 

been called NARX networks in (Lin, Horne, Tiño, and Giles (1996a and 1996b)). The state in this type of network is equivalent to:

$$\begin{aligned} \bar{s}(t) = & \{\bar{x}(t) \oplus \bar{x}(t-1) \oplus \dots \oplus \bar{x}(t-n+1)\} \\ & \oplus \{\bar{y}(t-1) \oplus \bar{y}(t-2) \oplus \dots \oplus \bar{y}(t-m)\}. \end{aligned} \quad (5.3)$$

Clearly, the WIT memory described above is a special case of the NARX memory where  $m = 0$  (i.e., the size of the output window is zero). NARX memory and WIT memories are classified separately in this taxonomy because the more restrictive WIT memory has important representational limitations that are not shared by NARX memory. The state function for the NARX memory can be defined as the recurrent function:

$$\begin{aligned} f_{\bar{s}}(\bar{s}(t-1), \bar{x}(t), \mathfrak{W}) \\ \equiv & \bar{x}(t) \oplus \bar{s}(t-1)[1..(n-1) \cdot \dim(\bar{x}(t))] \oplus \bar{y}(t-1) \oplus \bar{s}(t-1) \\ & \cdot [n \cdot \dim(\bar{x}(t)) + 1..n \cdot \dim(\bar{x}(t)) + (m-1) \cdot \dim(\bar{y}(t))]. \end{aligned} \quad (5.4)$$

In this equation, there is no direct dependence of state on the weights of the network  $\mathfrak{W}$ . An indirect dependence is present, however, in that the output  $\bar{y}$  depends on the weights according to the output function used (see section 6). A NARX memory with input window of length,  $n = 7$ , and output window of length,  $m = 7$ , is illustrated in Figure 3. Unlike the WIT memory, which is not recurrent, output activations in this network are used to compute activation values of internal nodes, which in turn are used to

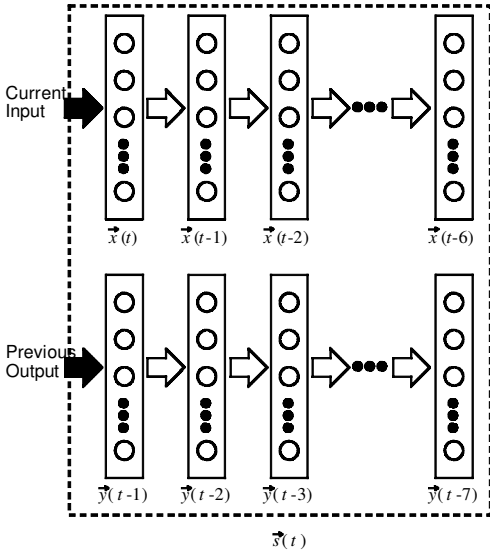


Figure 3: NARX memory.

recompute output activations. This recurrence gives the NARX memory the ability to encode a history of activations extending back arbitrarily in time.

NARX networks have been used in Lapedes and Farber (1987), Chen, Billings, and Grant (1991), Narendra and Parthasarathy (1990), Lin, Giles, Horne, and Kung (1996), Lin et al. (1996b), Lin, Horne, Tiño, and Giles (1996a), and Siegelmann et al. (1997). The last of these articles contains a proof that NARX networks can have the computational power of Turing machines.

**5.3 Time-Delay Neural Network (TDNN) Memories.** Thus far, we have seen two approaches to using previous node activations to represent state. The WIT memory stores previous inputs, and the NARX memory stores previous inputs and output. Another alternative is to store the activations of input and hidden units. This approach has been used extensively by Waibel (1988; Waibel et al., 1989). Here, the state is defined as:

$$\begin{aligned}
 \bar{s}(t) = & \bar{x}(t) \oplus \bar{x}(t-1) \oplus \bar{x}(t-2) \oplus \dots \oplus \bar{x}(t-n) \\
 & \oplus \bar{h}^{(I)}(t) \oplus \bar{h}^{(I)}(t-1) \oplus \bar{h}^{(I)}(t-2) \oplus \dots \oplus \bar{h}^{(I)}(t-m) \\
 & \oplus \bar{h}^{(II)}(t) \oplus \bar{h}^{(II)}(t-1) \oplus \bar{h}^{(II)}(t-2) \oplus \dots \oplus \bar{h}^{(II)}(t-l). \quad (5.5)
 \end{aligned}$$

$\vec{h}^{(l)}(t)$  and  $\vec{h}^{(ll)}(t)$  represents the activations of the first and second hidden layer, respectively. They are computed as:

$$\vec{h}^{(l)}(t) = \sigma(\mathbf{W}^{(l)} \cdot \{\vec{x}(t-1) \oplus \vec{x}(t-2) \oplus \dots \oplus \vec{x}(t-n)\})$$

and

$$\vec{h}^{(ll)}(t) = \sigma(\mathbf{W}^{(ll)} \cdot \{\vec{h}^{(l)}(t-1) \oplus \vec{h}^{(l)}(t-2) \oplus \dots \oplus \vec{h}^{(l)}(t-m)\}),$$

respectively, where  $\mathbf{W}^{(l)}$  and  $\mathbf{W}^{(ll)}$  represent the weight matrices leading to the first and second hidden layers, respectively. These matrices are a component of the network's trainable parameters  $\mathfrak{W}$ . This formulation implies a state function, which can be formulated as:

$$\begin{aligned} f_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathfrak{W}) \\ \equiv \vec{x}(t) \oplus \vec{s}(t-1)[1 \dots (n-1) \cdot \dim(\vec{x}(t))] \\ \oplus \vec{h}^{(l)}(t) \\ \oplus \vec{s}(t-1)[n \cdot \dim(\vec{x}(t)) + 1 \\ \dots n \cdot \dim(\vec{x}(t)) + (m-1) \cdot \dim(\vec{h}^{(l)}(t))] \\ \oplus \vec{h}^{(ll)}(t) \\ \oplus \vec{s}(t-1)[n \cdot \dim(\vec{x}(t)) + m \cdot \dim(\vec{x}(t)) + 1 \\ \dots n \cdot \dim(\vec{x}(t)) + m \cdot \dim(\vec{h}^{(l)}(t)) \\ + (l-1) \cdot \dim(\vec{h}^{(ll)}(t))] \end{aligned} \quad (5.6)$$

Variations on this network might include more or fewer hidden layers, but this two-hidden-layer approach is the one implemented for various applications in the area of phoneme recognition (Waibel, 1988; Waibel et al., 1989).

**5.4 Feedforward Exponential Decay (FED) Memories.** An alternative to storing explicitly a finite number of previous activation values from nodes in the network is to store an infinite history of previous activations implicitly. This can be accomplished by basing the network's state on the entire history of various node activations. In order to realize such a system in a finite number of state nodes, it is necessary to represent the infinite history in a manner that it can be computed incrementally based on the previous history.

Poddar and Unnikrishnan (1991a, 1991b) have used the following state function:

$$\vec{s}(t) = \alpha \cdot \vec{s}(t-1) + (\mathbf{I} - \alpha) \cdot \{\vec{x}(t-1) \oplus \vec{h}(t-1)\}. \quad (5.7)$$

Here,  $\alpha$  represents a diagonal matrix of decay constants that remains fixed during training,  $\mathbf{I}$  a diagonal matrix of ones, and  $\vec{h}(t-1)$  is a vector of hidden

unit activations computed,

$$\vec{h}(t-1) = \vec{\sigma}(\mathbf{W} \cdot \{\vec{x}(t-1) \oplus \vec{s}(t-1)[1.. \dim(\vec{x})]\}). \quad (5.8)$$

Unwrapping the recursive equation 5.7 reveals that the state is indeed based on the infinite, exponentially decaying history of input and hidden unit activations of the network:

$$\vec{s}(t) = \sum_{\tau=1}^t (\mathbf{I} - \alpha)^\tau \cdot \{\vec{x}(t-\tau) \oplus \vec{h}(t-\tau)\}. \quad (5.9)$$

Thus, this architecture avoids the difficulty of having to decide on a particular window length as in the WIT memory. It should be noted that despite the fact that equation 5.7 is defined recursively, this network is not a recurrent network. Every node's activation can be computed based on only a finite input history for the network. FED memories have been used for time-series prediction (Poddar & Unnikrishnan, 1991a) and prediction of speech signals (Poddar & Unnikrishnan, 1991b).

**5.5 Gamma Memories.** The FED memory uses an exponentially decaying history. Other history functions could also be used. To be of practical use, however, the selected history function must be incrementally computable. That is, the state of the system should be computable based on a finite number of parameters rather than requiring storage of the arbitrarily long activation history of the nodes in the network. A history function that satisfies this criterion is based on the (normalized) gamma density function ( $\Gamma(x) \equiv \int_0^\infty t^{x-1} e^{-t} dt$ ). The memory uses two parameters per node represented by two diagonal matrices: depth ( $\mu$ ) and resolution ( $\omega$ ). The state vector of a gamma memory is computed according to the equation

$$\begin{aligned} \vec{f}_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathfrak{B})[i \cdot \dim(\vec{x})..(i+1) \cdot \dim(\vec{x}) - 1] \\ \equiv \mathbf{B}((1-\mu) \cdot \vec{s}(t-1)[(i-1) \cdot \dim(\vec{x})..(i) \cdot \dim(\vec{x}) - 1] \\ + \mu \cdot \vec{s}(t-1)[i \cdot \dim(\vec{x})..(i+1) \cdot \dim(\vec{x}) - 1]), \end{aligned} \quad (5.10)$$

where

$$\vec{s}(t)[0.. \dim(\vec{x}) - 1] \equiv \vec{x}(t), \quad (5.11)$$

and  $\mathbf{B}$  is a matrix of binomials computed as

$$\mathbf{B}_{i,j} = \binom{t}{\omega_{i,j}}. \quad (5.12)$$

This network is nonrecurrent since no part of the state is required to compute a subsequent state. In this network, the parameter  $\mu$  is one of the trainable parameters contained in  $\mathfrak{B}$ . The gamma memory approach has been used by de Vries and Principe (1991, 1992a) and applied to signal prediction, including electroencephalograms.

**5.6 Habituation-Based Memories.** Stiles and Ghosh (1996, 1997) have developed another interesting alternative to computing state. In their networks, state is based on a mathematical model of habituation suggested by Arbib. This discrete-time model describes the effect that repeated input signals have on synaptic weight strength. Stiles and Ghosh use a set of weight values computed according to Arbib's model as inputs to a feedforward network. Since the habituation model causes weights to change in response to input patterns, the weights can serve to represent a history of the input or state of the system. Specifically,

$$s_i(t) = s_i(t-1) + \tau_i(\alpha_i(1 - s_i(t-1)) - s_i(t-1)x_{f(i)}), \quad (5.13)$$

where  $\tau_i$  and  $\alpha_i$  are constants defining the rate of habituation, and  $f(i)$  is a function mapping a state index to an input index. The authors found that having multiple state values associated with each input improved performance and was a biologically realistic assumption.

Stiles and Ghosh proved mathematically that memories of this type are able to approximate arbitrarily well any continuous, causal, time-invariant mapping from input sequences to output sequences. These networks have been shown to outperform time-delay neural networks on a variety of problems, including classification of Banzhaf sonograms.

**5.7 Jordan Memories.** Michael Jordan (1986a, 1986b) has developed a variation on the FED approach that uses a history of the output units. Specifically, Jordan defines the state of his networks as

$$\vec{s}(t) = \frac{1}{2}\vec{s}(t-1) + \vec{y}(t-1). \quad (5.14)$$

Jordan networks can be viewed as storing an infinite history of past outputs since the state of the network can be rewritten as

$$\vec{f}_s(\vec{s}(t-1), \vec{x}(t), \mathfrak{W}) \equiv \vec{y}(t-1) + \left(\frac{1}{2}\right)\vec{y}(t-2) + \left(\frac{1}{2}\right)^2\vec{y}(t-3) + \dots + \left(\frac{1}{2}\right)^{t-1}\vec{y}(0). \quad (5.15)$$

Here, it can be seen that  $\vec{s}(t)$  is computed based on the entire history of  $\vec{y}(t)$ , though with exponentially decaying influence as time increases. These networks are recurrent since each state vector depends on the previous state vector. Jordan applied his networks to the problem of producing articulatory sequences. This network is also interesting for historical reasons since it is one of the earliest reported recurrent networks and provided the foundation for subsequent approaches, including Elman's networks (see below).

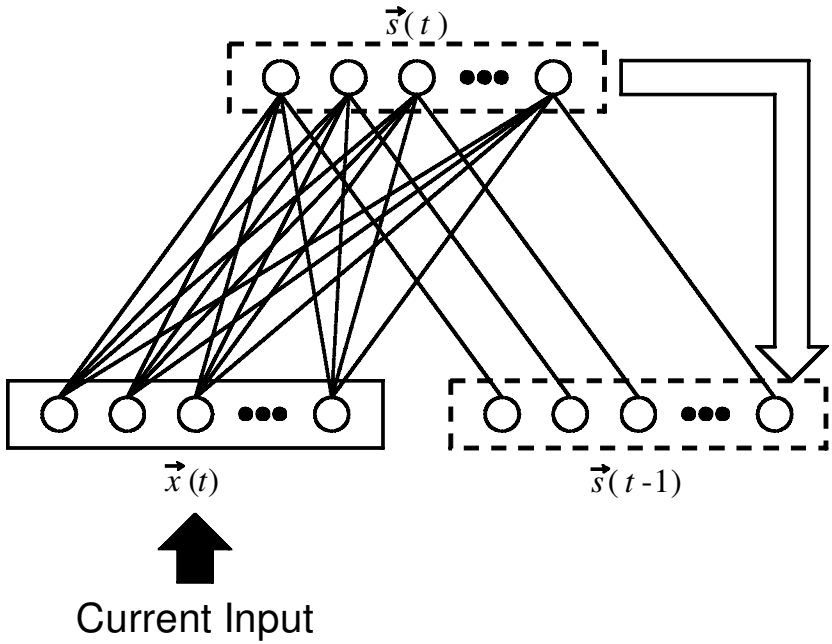


Figure 4: LF memory.

**5.8 Local Feedback Memories.** Another simple memory is the local feedback (LF) memory. These memories use a set of hidden units to represent the network's state. Each state unit's activation is computed based not only on the inputs, but also on that same unit's previous activation:

$$\vec{f}_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathfrak{W}) \equiv \vec{\sigma}(\mathbf{W} \cdot \vec{x}(t) + \mathbf{X} \cdot \vec{s}(t-1)). \quad (5.16)$$

Here,  $\mathbf{W}$  (which is a trainable parameter and part of  $\mathfrak{W}$ ) represents a matrix of connection weights between the input units and the hidden units that make up  $\vec{s}(t)$ , while  $\mathbf{X}$  (also a trainable parameter and part of  $\mathfrak{W}$ ) is a diagonal matrix. The matrix  $\mathbf{X}$  represents a set of recurrent (self) connections among the state units.

This type of memory is graphically represented in Figure 4. Rather than showing the locally recurrent connections in this type of memory, we illustrate the operation of the network using a fictitious set of processing units called *context units*. These units can be considered to contain a copy of the activations of the state units at the previous time step. The context units allow us to separate the temporal delay from the modulation component of a connection's operation and greatly simplify illustrations of STCNs. The use of context units was pioneered by Elman (1990a, 1991b).

The LF approach was developed by Gori, Bengio, and De Mori (1989) and used in Frasconi et al. (1992). A variation of this technique, presented in the same articles, uses the same approach without the nonlinearity. In this case,

$$\vec{f}_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathfrak{W}) \equiv \mathbf{W} \cdot \vec{x}(t) + \mathbf{X} \cdot \vec{s}(t-1). \quad (5.17)$$

An extension of this approach has been used by Fahlman (1991) by the name, *recurrent cascade correlation* (RCC). While the LF memories used by Gori et al. (1989) and Frasconi et al. (1992) use only one layer, Fahlman's networks use a cascaded approach in which unit  $i$  receives connections from all lowered-numbered nodes. Fahlman's LF memories are computed incrementally. That is, each component of the state vector depends on all lower-numbered components for the same time step. The first components of the state vector are equivalent to the input vector. Additionally, each node receives a recurrent connection from itself with time delay of one unit. Since no node receives delayed connections from nodes other than itself, this represents a strictly local recurrence. It is the local recurrence that introduces temporal dependency. This type of memory has been studied by Fahlman (1991), Giles et al. (1995), and Kremer (1996a, 1996b).

Once again, we shall not show recurrent connections in our diagram, but rather an additional set of context units whose activations are equal to the state units at the previous time step. The computation of state vector in an RCC memory is defined as

$$f_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathfrak{W})[i] \equiv \begin{cases} \vec{x}_i(t) & \forall i \leq \dim(\vec{x}) \\ \vec{\sigma}(\mathbf{W}_{i \cdot} \cdot \{1 \oplus \vec{s}(t)[1..i-1] \oplus \vec{s}(t-1)[i]\}) & \text{otherwise,} \end{cases} \quad (5.18)$$

where  $\mathbf{W}$  represents a submatrix of  $\mathfrak{W}$  containing the connections to the state nodes. The architecture that computes this state vector is depicted in Figure 5 (adapted from an illustration in Giles et al., 1995). Each component of the state vector is computed based on the values of all lower-numbered components at the same time step and the value of the component itself at the previous time step.

Fahlman's RCC memory is incapable of representing certain classes of finite state automata. Details of these limitations are found in Giles et al. (1995) and Kremer (1996a, 1996b). LF memories have similar limitations which are detailed in Frasconi and Gori (1996) and Kremer (1999).

**5.9 Infinite Impulse Response Filter Memories.** The infinite impulse response (IIR) memories use connections between nodes that are modeled after IIR filters. While a standard connection performs a simple multiplication—multiplying a presented signal by the connection weight,  $z_i(t) =$

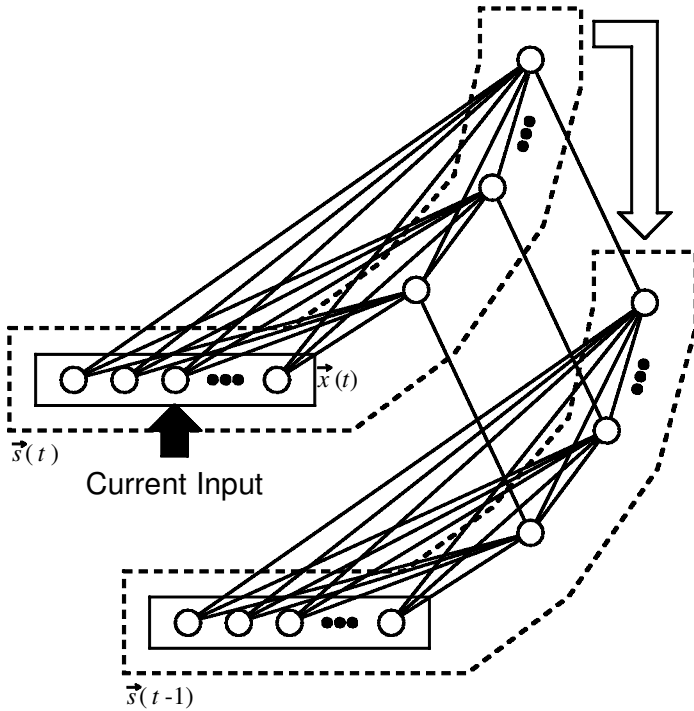


Figure 5: RCC memory.

$w_{ij} \cdot x_j(t)$ — an IIR connection performs a more complex temporal computation:

$$z_i(t) = \sum_{\tau=0}^{p-1} w_{ij}^{(x)}(\tau) \cdot x_j(t - \tau) + \sum_{\tau=1}^q w_{ij}^{(z)}(\tau) \cdot z_j(t - \tau). \tag{5.19}$$

In both equations,  $x_j(t)$  represents the value transmitted into the connection at time  $t$ , and  $z_i(t)$  represents the value emitted from the other side of the connection at time  $t$  (before it passes through the destination node’s non-linearity).  $w^{(x)}(\tau)$  is the weight multiplier for the  $\tau$ th previous input and  $w^{(z)}(\tau)$  is the weight multiplier for the  $\tau$ th previous output.

It is possible to create an entire layer of such connections, connecting the STCN’s input,  $\vec{x}(t)$ , to a layer of hidden processing units,  $\vec{h}(t)$ . In such a network, the transmitted values can be computed by

$$\vec{z}(t) = \sum_{\tau=0}^{p-1} \mathbf{W}^{(x)}(\tau) \cdot \vec{x}(t - \tau) + \sum_{\tau=1}^q \mathbf{W}^{(z)}(\tau) \cdot \vec{z}(t - \tau), \tag{5.20}$$

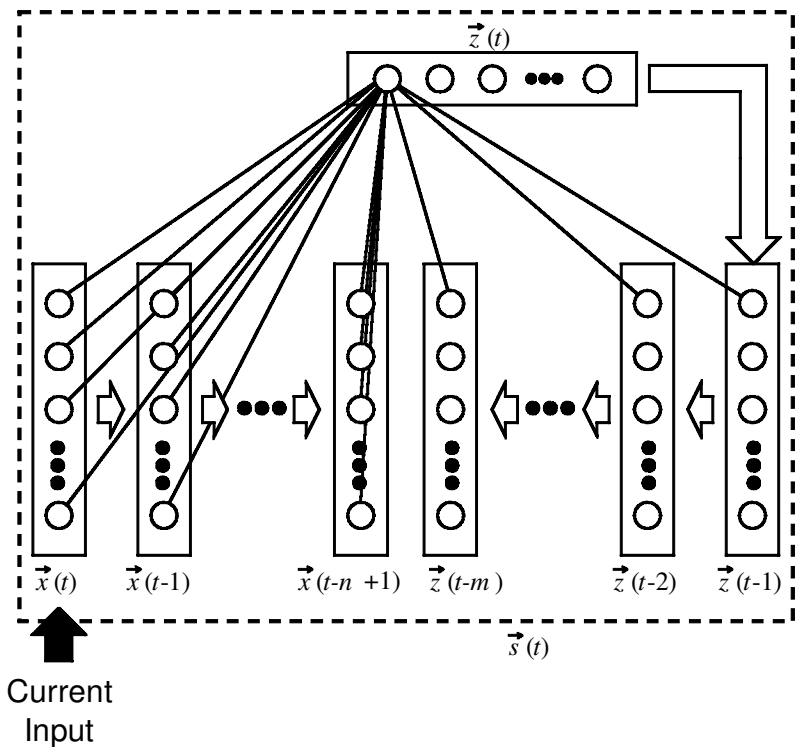


Figure 6: IIR memory.

and, the hidden unit activations by

$$\vec{h}(t) = \vec{\sigma}(\vec{z}(t)). \tag{5.21}$$

This approach is illustrated in Figure 6.

Of course, in practice, an STCN like this would require storage of not only the history of input patterns,  $\vec{x}(t), \vec{x}(t-1), \vec{x}(t-2), \dots$ , but also the history of transmitted values,  $\vec{z}(t-1), \vec{z}(t-2), \vec{z}(t-3), \dots$ . Thus, the state vector of such a network is defined as:

$$\vec{s}(t) = \{\vec{x}(t) \oplus \vec{x}(t-1) \oplus \dots \oplus \vec{x}(t-n+1)\} \oplus \{\vec{z}(t-1) \oplus \vec{z}(t-2) \oplus \dots \oplus \vec{z}(t-m)\}, \tag{5.22}$$

which can be computed recurrently as:

$$\vec{f}_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathcal{W})$$

$$\begin{aligned}
&\equiv \bar{x}(t) \oplus \bar{s}(t-1)[1 \dots (n-1) \cdot \dim(\bar{x}(t))] \\
&\quad \oplus \bar{z}(t-1) \\
&\quad \oplus \bar{s}(t-1)[(n+1) \cdot \dim(\bar{x}(t)) \dots \\
&\quad (n) \cdot \dim(\bar{x}(t)) + (m-1) \cdot \dim(\bar{z}(t))]. \quad (5.23)
\end{aligned}$$

This architecture differs from the LF memory described above in that the LF architecture sums the signals transmitted through numerous connections and optionally also passes the result through a nonlinearity before looping it back. By contrast, the IIR memory sends back the signal transmitted through each connection separately (prior to summation). The IIR memory has been used in Back and Tsoi (1990, 1991, 1993), Back, Wan, Lawrence, and Tsoi (1995), and Lawrence, Tsoi, and Back (1996). While some of these articles assume a general model in which the hidden unit activations can be computed in multiple layers, in practice only one layer of connections is typically used. The equations for the multilayer version can be derived in a fashion analogous to that used above.

**5.10 First-Order Context Memory.** Another approach to short-term memory that has been widely used is based on computing the STCN's state using a single-layer first-order feedforward network (Rumelhart, Hinton, & Williams, 1986). The network uses the previous state (also called context) and the current input as input and produces the new state as output. This approach, abbreviated FOC memory, is used in Elman's (1990a, 1991b) simple recurrent networks (SRN), Pollack's (1989, 1990, 1994) recurrent autoassociative memory (RAAM), Maskara and Noetzel's (1992) autoassociative recurrent network, and Williams and Zipser's (1989b) real time recurrent learning (RTRL) networks.

The FOC short-term memory is stored in a set of hidden units whose activations are computed based on the activations in a layer of input units and a layer of context units, and on the weights from these two layers to the state units. By convention (see Elman, 1990a), the activations of all the context units are initially set to 0 or 0.5 (though we will see a different approach in section 8.7) and subsequently set to the activation values of the hidden units at the previous time step (the number of context and hidden units must be equal). Specifically, the hidden unit activation values are copied to the context units at each time step. Thus, at any given time, the hidden unit activations represent the current state, and the context unit activations represent the previous state. Thus, the function used to compute the state vector is:

$$f_{\bar{s}}(\bar{s}(t-1), \bar{x}(t), \mathfrak{W}) \equiv \bar{\sigma}(\mathbf{W} \cdot \{1 \oplus \bar{x}(t) \oplus \bar{s}(t-1)\}), \quad (5.24)$$

where  $\mathbf{W}$  is a matrix and a component of  $\mathfrak{W}$  representing the adaptable weights of the connections in the memory. As in section 2.2,  $\bar{\sigma}(\bar{x})$  represents

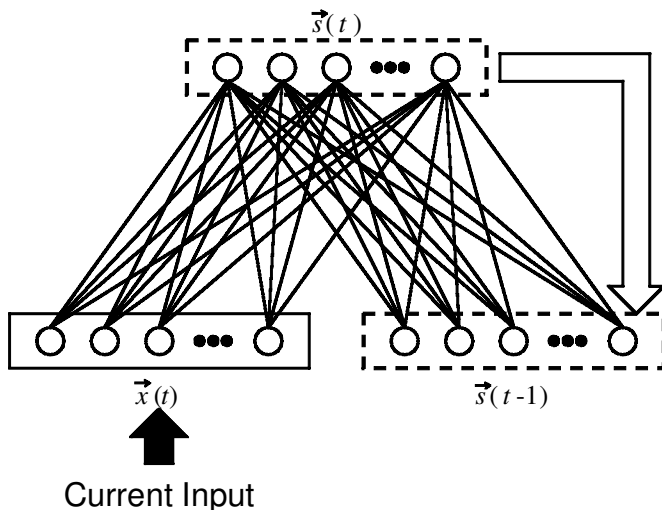


Figure 7: FOC memory.

the result of applying the function  $\sigma(x) = \frac{1}{1+e^{-x}}$  to each component of vector  $\vec{x}$ . The FOC short-term memory is depicted in Figure 7.

Unlike WIT and NARX memories, whose state vector is equivalent to a finite number of previous input and output vectors, FOC memories use states based on the previous state vector and input vector. This implies that the state vector of this type of memory can contain information not found in recent input and output vectors. The FOC memory approach has also been employed in Elman (1988, 1989, 1990b, 1991a), Cleeremans, Servan-Schreiber, and McClelland (1989), Cleeremans and McClelland (1991), Servan-Schreiber et al. (1988), Servan-Schreiber, Cleeremans, and McClelland (1989, 1991, 1994), and Williams and Zipser (1989a). These networks have the representational power of finite state automata as was shown in Kremer (1995) and Goudreau, Giles, Chakradhar, and Chen (1994).

**5.11 Second-Order Context Memory.** A variation on FOC memory involves using a single-layer second-order network to compute state based on previous state and current input. Second-order connections are connections that connect three nodes (rather than just two) and are a restricted type of the Sigma-Pi units studied by Rumelhart et al. (1986). In these connections, the activation of one of the nodes is modulated by that of another and transmitted as a signal to the third, which computes a weighted sum of all of the products and passes the value through a squashing function. The processing performed by first-order and second-order connections is illustrated in Figure 8.

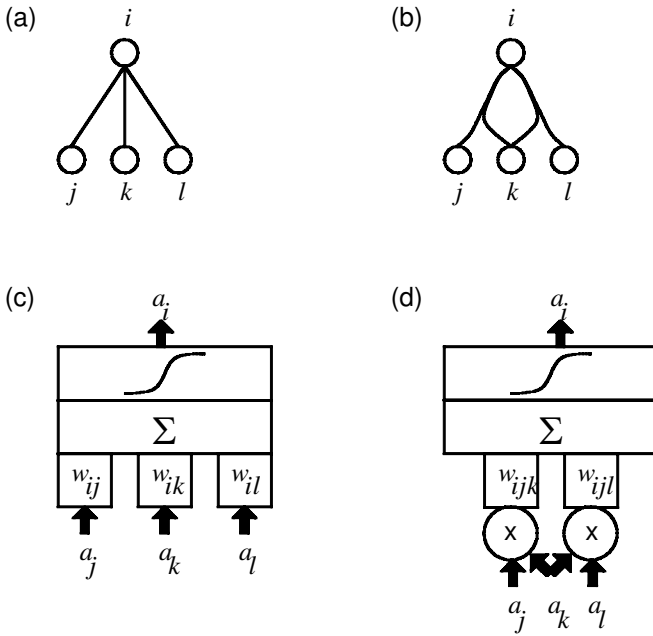


Figure 8: First-order versus second-order connections. (a) Three first-order connections from nodes  $j, k, l$  to node  $i$ . (b) Two second-order connections from  $j$  and  $k, k$  and  $l$ , to node  $i$ . (c) Diagram of computation performed by first-order connections. (d) Diagram of computation performed by second-order connections.

In the SOC memory, every input unit is connected with every state unit to every other state unit via a second-order connection with a temporal delay of one time step. The context units in this network operate in a manner analogous to the FOC memory. That is, their activation values are equal to the activation values of the corresponding state units at the previous time step. In this network, second-order connections then connect every possible pair of one input and one context unit to every state unit. Thus,

$$f_{\mathfrak{S}}(\bar{s}(t-1), \bar{x}(t), \mathfrak{W}) \equiv \bar{\sigma}((\mathbf{W} \cdot \bar{x}(t)) \cdot \bar{s}(t-1)), \tag{5.25}$$

where  $\mathbf{W}$  is a three-dimensional weight array component of  $\mathfrak{W}$ , and  $\mathbf{W} \cdot \bar{x}(t)$  is a two-dimensional matrix,  $\mathbf{M}$ , whose components  $\mathbf{M}_{ij}$  are computed:

$$\mathbf{M}_{ij} = \sum_k \mathbf{W}_{ijk} \cdot \bar{x}_k(t). \tag{5.26}$$

Figure 9 depicts the SOC short-term memory. Note that only the connections leading to one of the state nodes are depicted here. The remaining 48

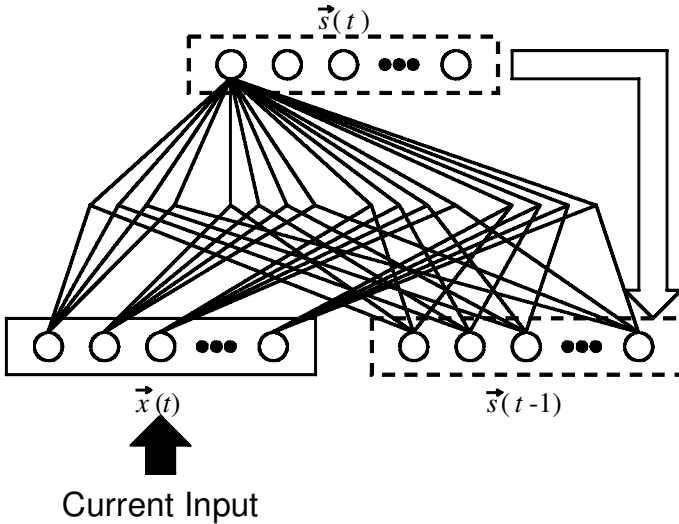


Figure 9: SOC memory.

connections leading into the other three state nodes in the diagram are not illustrated in the interest of clarity. SOC memories have been used extensively by Giles et al. (1991, 1992a, 1992b), Giles and Omlin (1992), Goudreau et al. (1994), Liu, Sun, Chen, Lee, and Giles (1990), Miller and Giles (1993), Omlin and Giles (1994a, 1996b), Shaw and Mitchell (1990), Sun, Chen, Giles, Lee, and Chen (1990a), Sun, Chen, Lee, and Giles (1991), and Watrous and Kuhn (1992a, 1992b).

The principle of this approach is based on using the input units to gate signals transmitted from context to state units. SOC networks are particularly adept at encoding finite state automata (the details of such an encoding are found in Goudreau et al., 1994).

**5.12 Long Short-Term Memory.** Another second-order approach has been pioneered by Hochreiter and Schmidhuber (1996, 1997a, 1997b). This architecture is specifically designed to avoid some of the limitations associated with the training of recurrent architectures. We will discuss these limitations in greater detail in section 8 but describe the architecture here. This approach is called long short-term memory (LSTM). The basic idea is to use both node activations and the net inputs to the nodes' transfer functions as state,

$$\begin{aligned}
 f_s(\vec{s}(t-1), \vec{x}(t), \mathcal{W}) \\
 \equiv \vec{s}(t-1)[1.. \dim(\vec{s})/2] \\
 + (\text{in}(t) \cdot \bar{\sigma}(\mathbf{W}_c \cdot \{\vec{s}(t-1)[\dim(\vec{s})/2 + 1.. \dim(\vec{s})] \oplus \vec{x}(t)\})) \\
 \oplus \text{out}(t) \cdot \bar{\sigma}(\vec{s}(t-1)[1.. \dim(\vec{s})/2]).
 \end{aligned} \tag{5.27}$$

where  $in(t)$  and  $out(t)$  are two specialized gating units that control the flow of information into and out of the memory. Their activations are computed:

$$in(t) = \sigma(\bar{W}_{in} \cdot \{\bar{s}(t-1)[\dim(\bar{s})/2 + 1 \dots \dim(\bar{s})] \oplus \bar{x}(t) \oplus in(t-1) \oplus out(t-1)\}), \quad (5.28)$$

and

$$out(t) = \sigma(\bar{W}_{out} \cdot \{\bar{s}(t-1)[\dim(\bar{s})/2 + 1 \dots \dim(\bar{s})] \oplus \bar{x}(t) \oplus in(t-1) \oplus out(t-1)\}), \quad (5.29)$$

respectively.

It is also possible to use multiple memory blocks of the types described by the equations above, each with its own input and output gating units. The LSTM memory approach has been shown by its developers to be very effective for learning a number of spatiotemporal input-output associations spanning long time periods.

**5.13 Connectionist Pushdown Automaton Memory with Continuous Stack.** Connectionist pushdown automaton (CPA) memories are radically different from the STCN memories that have previously been discussed. This is because in addition to storing state in the activations of nodes, these memories also store state on an external continuous stack in a hybrid connectionist-classical system. In this sense, they are like classical pushdown automata (Hopcroft & Ullman, 1979), whose state can be considered to be equivalent to the state of the finite controller and the contents of an unbounded stack. Hopcroft and Ullman refer to this type of extended state as an *instantaneous description* (ID).

While a conventional finite state controller can send three discrete signals to the stack (pop, push, no action), the stack of a CPA memory must respond to continuous (numerical) signals received from the nodes in the connectionist controller. Similarly, while a conventional finite state controller retrieves discrete symbols from its stack, the connectionist controller of a CPA memory must be able to accept continuous values from its stack. These are requirements of the gradient-descent learning algorithms used in STCNs, which require a differentiable error function. The stack controller in a CPA memory is an extension of the SOC memories described above. While SOC memories are described by three classes of nodes (input, context, state), the stack controller in a CPA memory has two additional node classes (action nodes and read nodes), which are used to manipulate the external stack. There is only one action node used to send and remove symbols (pop a symbol, push a symbol, no change) to or from the stack. The read nodes are used to encode the symbol at the top of the stack. CPA memories also use third-order connections instead of second-order connections, combining one input, one context, and one read node in a connection's transmission.

Since a pushdown automaton accepts input strings only if, after presentation of the entire string, the stack is empty, the error function for a CPA memory must take into account the length of the stack for legal strings. Since the error function for any gradient-descent algorithm must be continuous, the stack length of a CPA memory must also be represented by a continuous value. This continuity is achieved by giving each symbol in the stack a certain real-valued length and defining the stack length to be equal to the sum of the lengths of the symbols on the stack. Symbols are pushed onto or real-valued popped off the stack based on the activation value of the action node. This activation value can range from  $-1$  to  $+1$ . Activation values greater than a small constant  $\epsilon$  are interpreted as a signal to push the current input symbol onto the stack, while activation values less than  $-\epsilon$  are interpreted as a pop signal, and intermediate values are interpreted as no change in stack content.

When a symbol (encoded as a vector) is pushed onto the stack, its length in the stack is defined to be equal to the activation value of the action node. When a pop operation occurs, the action node's activation value determines the length of stack, which is deleted. Since symbol lengths in the stack can vary from  $\epsilon$  to 1 and deletions can vary from  $\epsilon$  to 1, there are three possible pop operations: (1) a pop that will only remove part of a symbol from the stack (reducing the symbol's length), (2) a pop that removes an entire symbol from the stack, or (3) a pop that removes multiple symbols from the stack. After CPA memories have been trained, push and pop operations typically become discretized.

The second additional node class consists of read nodes whose activation values are equal to the symbol encoded at the top of the stack. If the symbol at the top of the stack has a length of less than one, its activation vector is multiplied by its length and added to subsequent symbol vectors to form the read node activation values, until a total length of 1 is read. In the CPA memory, the read nodes are treated as input nodes, and the action node is implemented as a special output node. Third-order connections lead from every triple, consisting of an input, state, and read node, to every output unit as well as the action node.

For our purposes, the state vector  $\vec{s}(t)$  will represent only the state of the stack controller and not the stack itself, since it is external to the network and cannot be easily represented in the finite length vector  $\vec{s}(t)$ . Thus,

$$f_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathfrak{W}) \equiv \vec{\sigma}(((\mathbf{W} \cdot \vec{x}(t)) \cdot \vec{s}(t-1)) \cdot \vec{r}(t)), \quad (5.30)$$

where  $\vec{r}(t)$  is the vector representing the symbol read from the stack at time  $t$ ,  $\mathbf{W}$  is a four-dimensional array corresponding to the connection weights to the state units, and  $\mathfrak{W}$  is a component of  $\mathfrak{W}$ . Here, the multiplications are performed such that

$$((\mathbf{W} \cdot \vec{x}(t)) \cdot \vec{s}(t-1)) \cdot \vec{r}(t) = \sum_l \sum_k \sum_j (\mathbf{W}_{ijkl} \cdot x_l(t) \cdot s_k(t-1) \cdot r_j(t)). \quad (5.31)$$

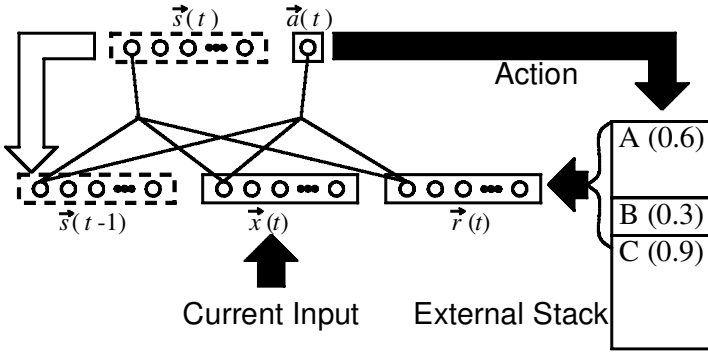


Figure 10: The CPA memory.

The CPA memory is illustrated in Figure 10. Note that only two of the third-order connections are shown in order to simplify the illustration. CPA memories have been studied by Das, Giles and Sun (1993), Giles et al. (1990), Sun, Chen, Giles, Lee, and Chen (1990b), Sun et al. (1990a), and Zeng, Goodman, and Smyth (1994).

**5.14 Connectionist Turing Machine Memory.** Williams and Zipser (1989b) developed a Turing machine connectionist memory (CTM) by replacing the finite state controller of a classical Turing machine by an FOC memory in order to design an STCN capable of universal computation. The connectionist network controls the operations performed on a standard Turing machine tape (move left, no change, write 1, write 0, move right) based on the activations of five action nodes and reads the current symbol at the tape head into its read nodes.

This network was never trained to learn the tape operations. Instead, the tape operations were supplied to the network during training, so only the control mechanism needed to be learned. For this reason, a continuous version of a tape was not required. The state vector for a CTM memory (which does not represent the external tape) is defined as:

$$f_{\vec{s}}(\vec{s}(t - 1), \vec{x}(t), \mathbb{X}) \equiv \vec{\sigma}(\mathbf{W} \cdot \mathbf{1} \oplus \vec{s}(t - 1) \oplus \vec{r}(t)). \tag{5.32}$$

A CTM memory is illustrated in Figure 11. Only the connections to one of the state nodes are shown.

## 6 Computing the Output Vector

We now examine the instantiations of the output vector function,  $f_{\vec{y}}(\vec{s}(t), \mathbb{X})$ . In particular, we discuss the three leading alternatives: zero-layer, one-layer, and two-layer output functions.

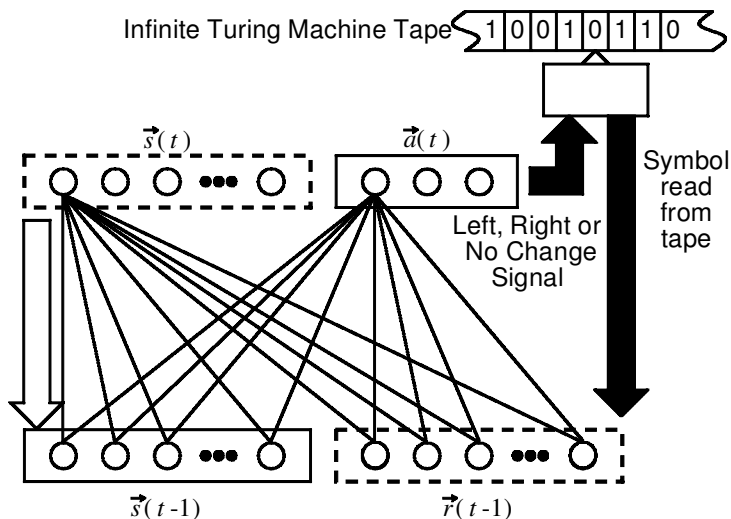


Figure 11: CTM memory.

**6.1 Zero Layer.** The simplest possible manner in which to compute the output of an STCN given the state vector is to use the state vector (or a portion thereof) as the output. Without loss of generality, we assume that the first few components of the state vector represent the output. This simple way of computing STCN output can then be easily represented:

$$f_{\vec{y}}(\vec{s}(t), \mathcal{S}) = \vec{s}(t)[1.. \dim(\vec{y})]. \quad (6.1)$$

This zero-layer approach to output computation has been used by Giles et al. (1992b), Giles and Omlin (1992, 1993b), Goudreau et al. (1994), Liu et al. (1990), Miller and Giles (1993), Omlin and Giles (1994a), Shaw and Mitchell (1990), Sun et al. (1990a, 1990b, 1991), Watrous and Kuhn (1992a, 1992b), Das et al. (1993), Giles et al. (1990), Sun, Giles, Chen, and Lee (1993), Zeng et al. (1994) and Williams and Zipser (1989b). It is illustrated in Figure 12a. Zero-layer output functions are typically used with computationally powerful memories, which can represent the state of network in an arbitrary form, including one in which the low-numbered nodes also contain the appropriate output vector. If used with a memory in which encodings are limited (as with FOC memories), a zero-layer output function will limit the types of outputs that can be computed.

**6.2 One Layer.** The next output function, increasing in complexity, computes the output vector using an additional layer of nodes. These nodes are massively parallelly connected to all the nodes that form the state vector,

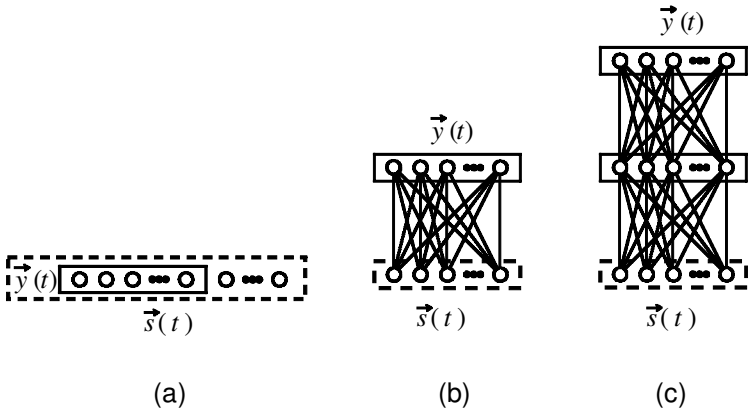


Figure 12: Output functions of STCNs. (a) Zero layer. (b) One layer. (c) Two layer.

giving an output function defined by:

$$f_{\vec{y}}(\vec{s}(t), \mathfrak{W}) = \vec{\sigma} \left( \mathbf{W}^{(\text{out})} \cdot \{1 \oplus \vec{s}(t)\} \right), \tag{6.2}$$

where  $\mathbf{W}^{(\text{out})}$  is a matrix of connection weights between the state and output nodes ( $\mathbf{W}^{(\text{out})}$  is a component of  $\mathfrak{W}$ ). This one-layer approach to output computation has been used in Elman’s SRNs (Elman, 1990a, 1991b), Pollack’s RAAM (Pollack, 1989, 1990), Maskara and Noetzel’s AARNs (Maskara & Noetzel, 1992), and Fahlman’s RCC (Fahlman, 1991), and is illustrated in Figure 12b. The one-layer approach is typically used with memories that can encode their state in such a manner that the desired output can be extracted from the state using a linear mapping (for details, see Kremer, 1995). For networks that can adopt arbitrary state representations, one-layer networks are not required, though they can provide a more parsimonious representation of the desired mappings.

**6.3 Two Layer.** The logical next step is to use a two-layer system where:

$$f_{\vec{y}}(\vec{s}(t), \mathfrak{W}) = \vec{\sigma} \left( \mathbf{W}^{(\text{out}2)} \cdot \{1 \oplus \vec{\sigma}(\mathbf{W}^{(\text{out}1)} \cdot \{1 \oplus \vec{s}(t)\})\} \right), \tag{6.3}$$

and  $\mathbf{W}^{(\text{out}1)}$  and  $\mathbf{W}^{(\text{out}2)}$  are components of  $\mathfrak{W}$  representing the weights to the first and second output layers of connections, respectively. This approach to computing output is employed by NETtalk (Sejnowski & Rosenberg, 1988) and by Narendra and Parthasarathy (1990) and is illustrated in Figure 12c. As noted earlier, for the latter case, the output of the NARX network could also be interpreted as the network’s state, in which case a zero-layer output

function would be used. Two-layer outputs are commonly used with memories that are static and thus cannot be manipulated to form a convenient representation for the desired output. They can also be used to provide more parsimonious representations.

## 7 Initializing the Parameters

---

We now turn our attention to the initialization of the trainable parameters,  $\mathfrak{B}$ . Not surprisingly, initialization can have a significant effect on learning performance.

**7.1 Randomization of Initial Parameters.** The simplest and most common approach is to randomize the trainable parameters according to an arbitrary probability distribution. The greatest advantage of this approach is that no a priori knowledge about the problem to be solved is required. But more sophisticated approaches have been suggested when additional knowledge is available to set initial weight values.

**7.2 Split-State Encoding.** For many applications, the researcher may already have some prior information about potential types of solutions that are to be learned. The most natural way to incorporate this type of a priori information about a problem to be solved by an STCN is to encode it in the weights of the network, since it is the weights that define the computation a network performs. By initializing the weights of an STCN based on the a priori information, network training can begin at a point in weight space that lies close to a final solution. Training can then enhance the initial weights to drive the network to an even better solution. This process has been called refinement (Giles & Omlin, 1993a, 1993b).

The key to initializing an STCN lies in the encoding of the a priori knowledge in the connection weights of the STCN. Kremer (1995) has described a method for translating finite state machines into FOC memories. The approach is based on a state-splitting technique (also described in Goudreau et al., 1994). Each state in the finite state machine corresponds to one or more state nodes in the FOC. If one of these state nodes has a high activation value, then the STCN is “in” the corresponding state. Once the encoding of states has been defined, a straightforward method can be used to set weights such that the desired state transitions are implemented in the system. (For more details, see Kremer, 1995.) The advantage of this approach lies in its ability to encode finite state machine rules into network. The disadvantages are that finite state machine rules representing a priori information about real-world problems are often not readily available and the number of hidden units required is equal to the number of input symbols in the automaton times the number of states.

**7.3 Direct Encoding.** Das, Giles, Omlin, and others (Giles & Omlin, 1992, 1993a, 1993b; Omlin & Giles, 1992, 1994a, 1996a, 1996b; Das et al., 1993) have developed a more straightforward method for incorporating finite state transitions into STCNs using SOC memories. In this architecture, it is possible to map each state in the finite state machine directly to one state node in the STCN. The result of this more direct mapping is that it is possible also to map each line in the transition table of the finite state machine to one second-order connection in the STCN. This affords the implementor the possibility of incorporating partial information about a finite state automaton (i.e., a subset of the state transitions) in the SOC memory. The direct encoding is advantageous in that it offers a trivial mapping from finite state machines to network weights (each state transition corresponds to exactly one weight) and that each state requires only one hidden unit. Once again, however, real-world rules are not readily available.

**7.4 Chain Graph Encoding.** A third approach to encoding an automaton in STCN has been developed by Frasconi, Gori, Maggini, and Soda (1991, 1995). They encode a chain graph automaton in a network using a linear programming algorithm. This approach is particularly appealing for speech applications because phonetic and acoustic sequences can be readily translated into chains.

## 8 Updating the Parameters

---

Most of the parameter change functions described here and in the STCN literature are based on applying the gradient-descent algorithm to the weights of the network. In particular, changes are made in proportion to the negative of the error gradient in weight space according to the equation

$$\Delta \mathbf{W} \equiv -c \cdot \nabla \varepsilon, \quad (8.1)$$

where  $c$  is a small, real-valued constant, and  $\varepsilon$  is the total network error defined in section 2.5. Expanding  $\varepsilon$  gives

$$\Delta \mathbf{W} \equiv -c \cdot \nabla \left( \sum_t \left( \frac{1}{2} \|\bar{E}(t)\|^2 \right) \right). \quad (8.2)$$

In conventional connectionist learning tasks, it is computationally convenient to evaluate this function piecewise over time by making weight changes after presenting each input pattern, as illustrated in the algorithm of Table 7. For many spatiotemporal problems, this type of evaluation becomes a necessity since there are a potentially infinite number of sample patterns, and hence it would be impossible to wait until all had been presented before adjusting the weights. Thus, weight changes are made according to

the formula

$$\Delta \mathbf{W}(t) \equiv -c \cdot \nabla \left\{ \frac{1}{2} \|\bar{E}(t)\|^2 \right\}. \quad (8.3)$$

Note that in the limit as  $c$  approaches 0, the effect of repeated applications of equation 8.3 approaches that of equation 8.2. However, for larger values of  $c$ , successive evaluations of the gradient in equation 8.3 will occur at different points in the weight space. Equation 8.3 is an approximation to the gradient, which improves as  $c \rightarrow 0$ . We now turn our attention to evaluating the gradient,

$$\nabla \left\{ \frac{1}{2} \|\bar{E}(t)\|^2 \right\} = \left( \sum_{i,j} \mathbf{I}_{i,j} \frac{\partial}{\partial \mathbf{W}_{i,j}} \right) \frac{1}{2} \sum_k \bar{E}_k(t)^2, \quad (8.4)$$

where  $\mathbf{I}_{i,j}$  represents a matrix whose components are all 0 except for the  $(i, j)$ th component whose value is 1, and where  $\mathbf{W}_{i,j}$  and  $\bar{E}_k(t)$  represent the  $(i, j)$ th and  $k$ th components of  $\mathbf{W}$  and  $\bar{E}(t)$ , respectively. This equation can be simplified to

$$\nabla \left\{ \frac{1}{2} \|\bar{E}(t)\|^2 \right\} = \sum_{i,j} \mathbf{I}_{i,j} \sum_k \bar{E}_k(t) \frac{\partial \bar{E}_k(t)}{\partial \mathbf{W}_{i,j}}, \quad (8.5)$$

and further to

$$\nabla \left\{ \frac{1}{2} \|\bar{E}(t)\|^2 \right\} = \sum_{i,j} \mathbf{I}_{i,j} \sum_k (\bar{y}_k^*(t) - \bar{y}_k(t)) \frac{\partial \bar{y}_k(t)}{\partial \mathbf{W}_{i,j}}. \quad (8.6)$$

The weight change functions discussed below differ in how they compute the final gradient in this equation.

**8.1 Full Gradient Descent (FGD).** The calculation of the final gradient can be computed by expanding the numerator to its maximum extent (using the function that computes state) and then computing the gradient of the resulting expression:

$$\frac{\partial \bar{y}_k(t)}{\partial \mathbf{W}_{i,j}} = \frac{\partial}{\partial \mathbf{W}_{i,j}} f_{\bar{y}} \left( f_{\bar{s}} \left( f_{\bar{s}} \left( f_{\bar{s}} (\dots \right. \right. \right. \\ \left. \left. \left. f_{\bar{s}}(\bar{s}(0), \bar{x}(0), \mathbf{W}) \dots, \right. \right. \right. \\ \left. \left. \left. \bar{x}(t-2), \mathbf{W}), \bar{x}(t-1), \mathbf{W}), \right. \right. \right. \\ \left. \left. \left. \bar{x}(t), \mathbf{W}), \mathbf{W}) \right) \right) \right). \quad (8.7)$$

If the state function does not use the value of  $\mathbf{W}$  (e.g., WIT memory), the computation of this derivative is quite simple, since it reduces to the computation of the weight gradient in error space in a nonrecurrent network.

If the state function does use the value  $\mathbf{W}$ , then the computation of this derivative is much more complicated due to its recurrent nature. A number of different algorithms have been suggested to perform the full gradient-descent computation in an efficient manner. Typically, the approaches trade space complexity for time complexity. Since all of the different (FGD) algorithms compute the same result and we will not be doing a space or time complexity analysis of the learning algorithms used by STCNs, we will not discuss differences in implementations of full gradient descent and refer interested readers to Rumelhart et al. (1986) for a description of back-propagation through time (BPTT), Williams and Zipser (1988, 1989b) for a description of real-time recurrent learning (RTRL), Schmidhuber (1992) for a BPTT/RTRL hybrid approach, and Sun et al. (1992) for a Green's function method approach to gradient descent. It is worth noting, however, that BPTT is not considered a real-time algorithm because the length of time and amount of memory required to compute a weight update are linear in the length of the input sequence being considered. By contrast RTRL, Schmidhuber's and Sun et al.'s approaches require constant time and memory to compute a weight update (regardless of input pattern length). Full gradient descent has been used by Sejnowski and Rosenberg (1987, 1988), Giles et al. (1991, 1992a, 1992b), Giles and Omlin (1992), Goudreau et al. (1994), Liu et al. (1990), Miller and Giles (1993), Omlin and Giles (1994a, 1994b, in press), Shaw and Mitchell (1990), Sun et al. (1990a, 1991, 1993); Watrous and Kuhn (1992a, 1992b), Das et al. (1993), Giles et al. (1990), Zeng et al. (1994), and Williams and Zipser (1989b). The equation for weight change for full gradient descent is:

$$\begin{aligned}
 & f_{\Delta \mathfrak{W}}(\bar{\mathbf{E}}(t), \mathfrak{W}, \bar{\mathbf{x}}(\cdot)) \\
 &= -c \cdot \sum_{i,j} \mathbf{I}_{i,j} \sum_k (\bar{y}_k^*(t) - \bar{y}_k(t)) \frac{\partial}{\partial \mathbf{W}_{i,j}} \\
 & \cdot f_{\bar{y}} \left( f_{\bar{s}} \left( f_{\bar{s}} \left( f_{\bar{s}} \left( \cdots f_{\bar{s}}(\bar{s}(0), \bar{\mathbf{x}}(0), \mathbf{W}) \cdots, \right. \right. \right. \right. \\
 & \quad \left. \left. \left. \bar{\mathbf{x}}(t-2), \mathbf{W} \right), \bar{\mathbf{x}}(t-1), \mathbf{W} \right), \bar{\mathbf{x}}(t), \mathbf{W} \right), \mathbf{W} \right). \quad (8.8)
 \end{aligned}$$

An important limitation to the full gradient-descent approach has been independently identified by Hochreiter (1991) and Bengio, Frasconi and Simard (1993, 1994). This limitation is based on the fact that as temporal sequences increase in length, the influence of early components of the sequence has less and less impact on the systems' output. This causes the

partial gradients, which define the weight changes, to shrink to zero as sequence lengths increase (a solution to this problem can be found in the LSTM memory described in section 5.12).

A special case of full gradient descent occurs for locally recurrent networks like the LF memories described above. In locally recurrent networks, the complexity of the computation of the gradient is significantly reduced to the point where the computation can be performed in a manner that is local in both space and time. This approach has been used in LF memories (Gori et al. 1989, Frasconi et al. 1992).

Another variation on this approach updates not only connection weights by means of gradient descent, but also other parameters of the network such as the values  $\bar{\mu}$  and  $\bar{\omega}$  in equation 5.10. This approach has been used in de Vries and Principe (1991, 1992a).

**8.2 Teacher Forcing.** If an STCN's state (or part thereof) is equivalent to the STCN's previous output, then a simplification of full gradient descent, called teacher forcing (TF), can be used. NARX memories use a state based entirely on input and output. Also, any STCN that uses a zero-layer output function uses part of the output vector as its state vector. In either of these cases, it is possible to substitute the value of the desired output,  $\bar{y}^*$ , for the value of the actual output,  $\bar{y}$ , whenever the latter occurs in the state vector's computation. This can eliminate a great deal of the recurrence in the equation and thus simplify computation. The equation for teacher-forced weight update is:

$$\begin{aligned}
 f_{\Delta \mathfrak{W}}(\bar{E}(t), \mathfrak{W}, \bar{x}(\cdot)) &= \sum_{i,j} \mathbf{I}_{i,j} \sum_k (\bar{y}_k^*(t) - \bar{y}_k(t)) \frac{\partial}{\partial \mathbf{W}_{i,j}} \\
 & f_{\bar{y}} \left( f_{\bar{s}} \left( f_{\bar{s}} \left( f_{\bar{s}} \left( \dots \right. \right. \right. \right. \\
 & \quad \left. \left. \left. f_{\bar{s}}(\bar{s}(0), \bar{x}(0), \mathbf{W}) \Big|_{\bar{y}(0) \equiv \bar{y}^*(0)} \dots, \right. \right. \\
 & \quad \quad \left. \left. \bar{x}(t-2), \mathbf{W} \Big|_{\bar{y}(t-2) \equiv \bar{y}^*(t-2)}, \right. \right. \\
 & \quad \quad \quad \left. \left. \bar{x}(t-1), \mathbf{W} \Big|_{\bar{y}(t-1) \equiv \bar{y}^*(t-1)}, \right. \right. \\
 & \quad \quad \quad \quad \left. \left. \bar{x}(t), \mathbf{W} \Big|_{\bar{y}(t) \equiv \bar{y}^*(t)}, \mathbf{W} \right) \right) \right) \right). \quad (8.9)
 \end{aligned}$$

Intuitively, this is equivalent to assuming that the network has already learned to produce the desired output for all previous time steps when computing the weight change for the current time step. This approach has been used by Jordan (1986a, 1986b), Williams and Zipser (1989a), and Narendra and Parthasarathy (1990).

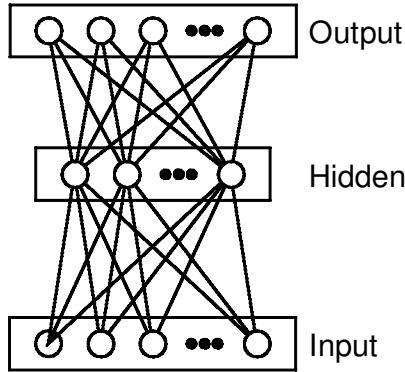


Figure 13: An encoder network.

**8.3 Truncated Gradient Descent.** Another approach to simplifying the gradient computation is to not completely expand the equation to compute output, prior to computing the derivative. This is referred to as truncated gradient descent (TGD). In this method the current state is evaluated, but the previous state is not. This results in the following weight update equation:

$$f_{\Delta \mathcal{W}}(\vec{E}(t), \mathfrak{B}, \vec{x}(\cdot)) \equiv \sum_{i,j} \mathbf{I}_{i,j} \sum_k (\vec{y}_k^*(t) - \vec{y}_k(t)) \frac{\partial}{\partial \mathbf{W}_{i,j}} f_{\vec{y}}(f_{\vec{s}}(\vec{s}(t-1), \vec{x}(t), \mathbf{W}), \mathbf{W}). \quad (8.10)$$

Since this equation is only an approximation to the true gradient, an STCN using this technique is not guaranteed to follow the true gradient in its search for a local minimum. Further, a local minimum in the error space defined by this function may not correspond to a local minimum in the original error space. This form of weight adjustment has been used by Elman (1990a, 1991b), Cleeremans et al. (1989), and Servan-Schreiber et al. (1988, 1989, 1991).

**8.4 Autoassociative Gradient Descent.** By definition, the memory vector of an STCN must contain some information about previous inputs. This implies that the weights in the network should be adapted to preserve input information in the memory vector. One way of accomplishing this task is to force the state explicitly to encode the input and the previous state by using an encoder network. An encoder network is a nonrecurrent network consisting of an input layer, a hidden layer, and an output layer. The input and output layers are of equal size, and the hidden layer is smaller (see Figure 13). The target vector for the output units is always equivalent to the current input vector. By training the output layer to reproduce the

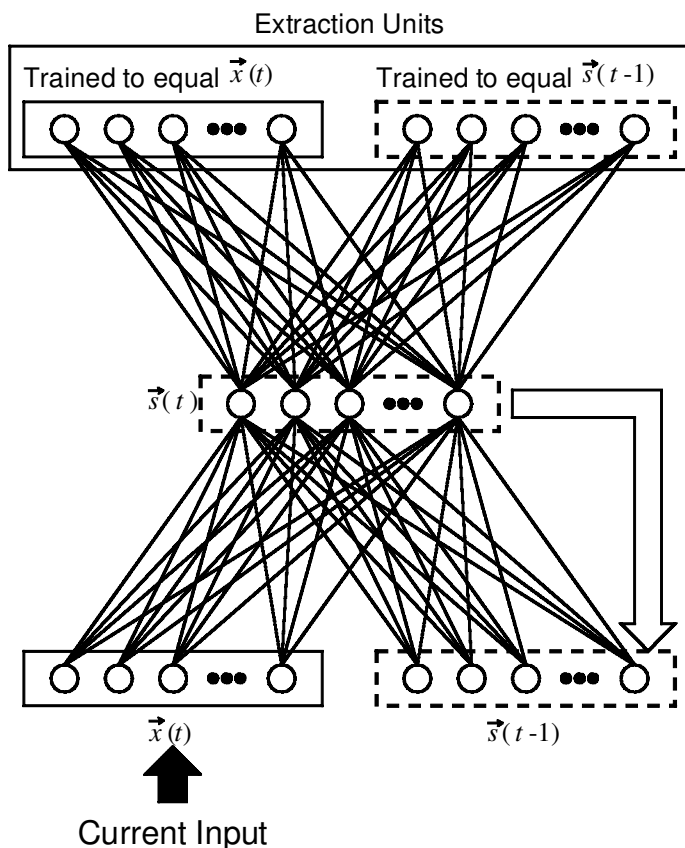


Figure 14: Extraction units are trained to reproduce input and previous state.

input, the smaller hidden layer adopts a compressed representation of the input.

Autoassociative gradient descent (AAGD) weight update works by temporarily creating an additional set of “extraction” units (see Figure 14). These units receive their inputs from the state units and are trained to reproduce the current input and previous state. It is the difference between the activations of the “extraction” units and the activations of the input units that defines the error that is minimized by gradient descent. Thus, the memory is trained not to produce a desired output pattern, but rather to preserve information about inputs and state. Of course, this information can then be used to generate a desired output pattern using any of the systems described in section 6. By training in this fashion, the state computation becomes the first step of an encoder network and is thus trained to encode the state as

a compressed representation of input and previous state. Once training is complete, the extraction units can be removed, leaving a network that encodes state. Furthermore, training can be accomplished using the classical feedforward algorithm on the architecture shown in Figure 14. In this approach, a complete coding of the sequence must be developed in the state representations (a requirement not found in other approaches where only relevant input information need be encoded).

Mathematically, if the vector  $\bar{u}(t)$  represents the activations of the “extraction units,” then the weight update equation is:

$$f_{\Delta \mathfrak{W}}(\bar{E}(t), \mathfrak{R}, \bar{x}(\cdot)) \equiv \sum_{i,j} \mathbf{I}_{i,j} \sum_k ((\bar{x}(t) \oplus \bar{s}(t-1))_k - \bar{u}_k(t)) \frac{\partial}{\partial \mathbf{W}_{i,j}} f(\mathbf{W}^{(u)} \cdot f_{\bar{s}}(\bar{s}(t-1), \bar{x}(t), \mathbf{W})), \quad (8.11)$$

where  $\mathbf{W}^{(u)}$  is an additional matrix of weights from state to extraction units, which is simultaneously trained according to the equation:

$$\Delta \mathbf{W}^{(u)} \equiv \sum_{i,j} \mathbf{I}_{i,j} \sum_k ((\bar{x}(t) \oplus \bar{s}(t-1))_k - \bar{u}_k(t)) \frac{\partial}{\partial \mathbf{W}_{i,j}^{(u)}} \bar{\sigma}(\mathbf{W}^{(u)} \cdot (\bar{x}(t) \oplus \bar{s}(t-1))). \quad (8.12)$$

AAGD weight update is used by Maskara and Noetzel (1992), and Pollack (1989, 1990).

**8.5 Stack Learning.** Adjustment of weights in an STCN with a continuous stack (see section 5.13) can be accomplished by means of a full gradient-descent procedure. In order to take advantage of the stack, however, it is also necessary to retrieve information from the stack and perform push and pop operations on it. In conventional pushdown automata (PDA), such operations are discrete in terms of their operation.

Das et al. (1993) developed an ingenious way to incorporate the use of a stack by relying on an error term added to the conventional Euclidean error measure. The stack used, unlike in a conventional PDA, is a continuous device capable of pushing partial symbols onto itself as well as popping multiple symbols at a time. The continuous nature of the stack extends to the number of symbols on the stack at any given time (i.e., the stack’s length).

By adding an error term equal to the square of the stack length at the end of a string,  $l$ , for legal strings, the network can be trained to produce an empty stack after processing any legal string. The length of the stack is equal to the sum of the activation values of the action node throughout the string, so it is possible to minimize error by backpropagating the error from the action node. Then,

$$f_{\Delta \mathfrak{W}}(\bar{E}(t), \mathfrak{R}, \bar{x}(\cdot)) = \frac{1}{2} c \cdot \nabla(\|\bar{E}(t)\|^2 + l^2) \quad (8.13)$$

for  $t = T$  and zero for other values of  $t$ . Stack learning (SL) has been studied by Das et al. (1993), Giles et al. (1990), Sun et al. (1990a, 1993), and Zeng et al. (1994).

**8.6 Alopex.** Unnikrishnan and Venugopal (1994) and Forcada and Carrasco (1995) have experimented with an alternative to gradient descent called the Alopex algorithm. This algorithm employs a simulated annealing type of approach to adapting parameters. Specifically weight changes are made probabilistically according to:

$$f_{\Delta\mathfrak{W}}(\bar{E}(t), \mathfrak{W}, \bar{x}(\cdot)) = \Delta \mathbf{W} \quad (8.14)$$

$$\Delta W_{ji}(t) = \begin{cases} W_{ji}(t-1) - \delta & \text{with probability } p_{ji}(t) \\ W_{ji}(t-1) + \delta & \text{with probability } 1 - p_{ji}(t) \end{cases} \quad (8.15)$$

where  $\delta$  is a small, positive constant. The probability of making a negative weight change is computed:

$$p_{ji}(t) = \frac{1}{1 + e^{-\frac{\delta W_{ji}(t) - \Delta E(t)}{T(t)}}}, \quad (8.16)$$

where  $T(t)$  represents a temperature and

$$\delta W_{ji}(t) = W_{ji}(t-1) - W_{ji}(t-2) \quad (8.17)$$

$$\delta E(t) = E(t-1) - E(t-2). \quad (8.18)$$

Temperature is controlled by an annealing schedule, which gradually reduces the number of changes that are not correlated to decreasing error:

$$T(t) = \frac{\delta}{N} \sum_{t'=t-N}^{t-1} |\Delta E(t')|. \quad (8.19)$$

Using this probabilistic approach, it is possible to avoid getting stuck in local minima.

**8.7 Training the Initial State.** While section 5 described how to compute new values of the STCN's state vector based on previous input, output, or state vectors, we usually assumed that the initial state vector,  $\bar{s}^{(init)} = \bar{s}(0)$ , was a vector consisting entirely of components with value 0 or one half. A better solution has been proposed by Forcada and Carrasco (1995) and Bulsari and Saxén (1995), who recommend training the initial state vector,  $\bar{s}^{(init)}$ , just like any other adaptable parameter in the network. That is,  $\bar{s}^{(init)}$  becomes part of  $\mathfrak{W}$ :

$$\mathfrak{W} = \langle \mathbf{W}, \bar{s}^{(init)} \rangle. \quad (8.20)$$

Then the component  $\mathbf{W}$  can be trained using any one of the weight adaptation techniques described above.  $\bar{s}$  is updated according to the Alopex algorithm, where each component  $i$  is updated as:

$$\delta \bar{s}_i^{(init)}(t) = \begin{cases} \bar{s}_i^{(init)}(t-1) - \delta & \text{with probability } p_i(t) \\ \bar{s}_i^{(init)}(t-1) + \delta & \text{with probability } 1 - p_i(t) \end{cases} \quad (8.21)$$

$$f_{\Delta \mathcal{W}}(\bar{E}(t), \mathfrak{W}, \bar{x}(\cdot)) = \langle \Delta \mathbf{W}, \delta \bar{s}^{(init)} \rangle. \quad (8.22)$$

This approach can improve training time and success rate (see Forcada and Carrasco, 1995).

**8.8 Manual Architecture Changes.** Finally, we turn our attention to the process that adjusts the size of the state vector. Most networks assume a fixed-size state vector, but this is not the only option. Instead, the state vector size can be considered part of the trainable parameters:

$$\mathfrak{W} = \langle \mathbf{W}, \dim(\bar{s}) \rangle. \quad (8.23)$$

If state vector size is adaptable, then the size of the weight matrix of connections leading into or out of the state vector must also be adaptable.

By far the most common approach to adjusting state vector size to a given problem is a manual trial-and-error procedure. That is, some heuristic is used to guess a suitable number of state nodes, and only if those nodes are unable to learn the given problem are more added. If overgeneralization occurs, nodes may also be removed. Note that the new architecture does not reuse any of the weight parameters of the previous architecture; all weight values are reinitialized when the state vector size is changed:

$$f_{\Delta \mathcal{W}}(\bar{E}(t), \mathfrak{W}, \bar{x}(\cdot)) = -\mathfrak{W} + \langle \mathbf{W}^{(new)}, \dim(\bar{s}^{(new)}) \rangle. \quad (8.24)$$

**8.9 Automatically Incrementing Nodes.** The inadequacy of the trial-and-error method described led Fahlman (1991) and Giles et al. (1995) to devise automatic incremental architecture adjustment techniques. These techniques create new processing units in the network to represent memory as required. If the network is unable to learn by weight adjustment, then new state nodes are added individually. Each state node is connected to compute the desired state function used by the STCN. Unlike the trial-and-error approach, existing weights are maintained, while new nodes to and from the new node are initialized to small values. In this sense, the short-term memory is increased until it is large enough to solve the given problem. Figure 15 illustrates the growth of the memory in an RCC memory; growth in other types of memory can be done similarly. The parameter update equation for

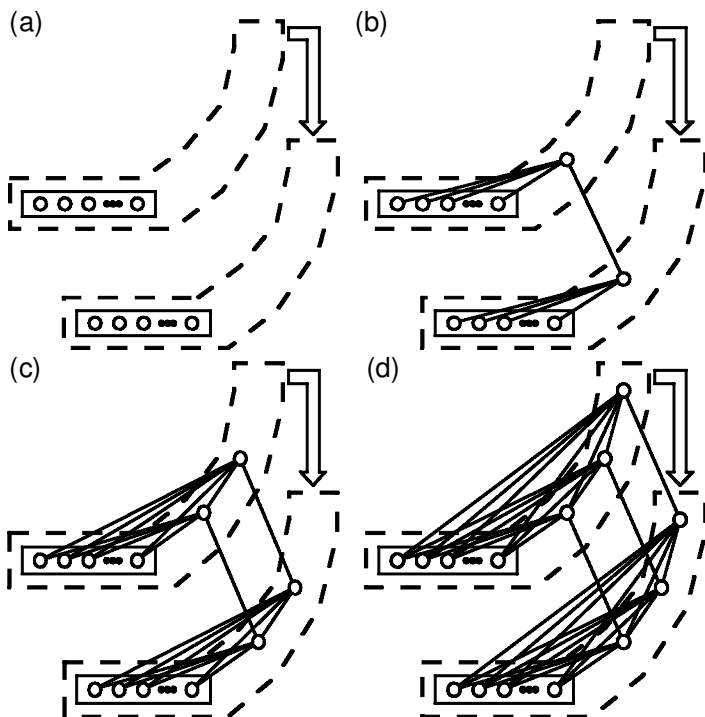


Figure 15: Adding nodes to a LRSI memory. (a) Before any nodes are added. (b) After one node is added. (c) After two nodes are added. (d) After the addition of a third node.

node incrementation is

$$f_{\Delta \mathfrak{M}}(\vec{E}(t), \mathfrak{M}, \vec{x}(\cdot)) = \mathfrak{M} \oplus \mathfrak{M}^{(random)}, \quad (8.25)$$

where  $\oplus \mathfrak{M}^{(random)}$  represents the addition of any necessary rows and columns with randomized values to the original matrix.

**8.10 Other Work Describing Weight Change Functions.** For an excellent discussion of weight update functions in the context of nonlinear adaptive filtering, see Nerrand, Roussel-Ragot, Personnaz, Dreyfus, and Marcos (1993). Pearlmutter (1995) has also written a comprehensive review of gradient-based weight change function for STCNs.

Table 9: Specific STCN Designs.

<i>Network</i>	<i>State Function</i>	<i>Output Function</i>	<i>Initial State</i>	<i>Training Algorithm</i>
Window in time: Sejnowski and Rosenberg (1987, 1988)	WIT	Two-layer	Randomization	FGD & MAC
Neural network infinite impulse response filter: Narendra and Parthasarathy (1990)	NARX	Two-layer	Randomization	TF
Simple recurrent network: Elman (1990a, 1991b)	FOC	One-layer	Randomization	TGD
Simple recurrent network: Kremer, 1995	FOC	One-layer	Split state	—
Recursive autoassociative memory: Pollack (1989, 1990, 1994)	FOC	One-layer	Randomization	AAGD
Autoassociative recurrent network: Maskara and Noetzel (1992)	FOC	One-layer	Randomization	AAGD
Real time recurrent learning: Williams and Zipser (1988, 1989a, 1989b)	FOC	Zero-layer	Randomization	FGD
Second-order network: Giles et al. (1991, 1992a, 1992b), Giles and Omlin (1992)	SOC	Zero-layer	Randomization	FGD
Second-order automaton encoding: Giles and Omlin (1993a, 1993b)	SOC	Zero-layer	Direct	FGD
Local feedback: Frasconi et al. (1991, 1995)	LF	One-layer	Chain graph	FGD
Recurrent cascade correlation: Fahlman (1991)	LF	One-layer	Randomization	FGD & AIN
Connectionist pushdown automaton: Giles et al. (1990)	CPA	Zero-layer	Randomization	SL
Connectionist Turing machine: Williams and Zipser (1998, 1989b)	CTM	Zero-layer	Randomization	FGD
Second-order constructive learning: Giles et al. (1995)	SLCOCC	Zero-layer	Randomization	FGD & AIN

## 9 Open Questions

---

Finally, we will mention some of the open problems in the field. Our goal is not to provide an in-depth discussion of each problem (since each could easily occupy its own article). Instead we provide some key references for readers who feel challenged to investigate further.

The temporal credit assignment problem that must be solved in order to find weights for a network to accomplish a given purpose is very difficult. Most algorithms fail to learn long dependencies reliably even in the simplest toy problems. The LSTM architecture described here represents one approach to overcoming this challenge (some others are identified in (Bengio et al., 1994, and Hochreiter & Schmidhuber, 1997a). Despite these successes, this problem remains one of the greatest challenges facing the field.

The representation capabilities of many networks have been studied in the context of formal grammars, but there remain many unanswered questions about the capabilities of specific architectures in relation to specific computational models. Most results in this area have focused on classical symbolic metaphors of computation. Much less has been done in relating STCNs to dynamical systems, fuzzy automata or noisy sequence identification, though Kolen (1994), Omlin et al. (1998), and Maass and Orponen (1997, 1998) represent notable developments in these three areas, respectively.

The issue of knowledge encoding also warrants additional future study. Although there have been a number of results showing that randomly generated rules can be readily incorporated into recurrent networks, there have been very few examples of networks in which real-world a priori knowledge was used to improve performance. Frasconi et al. (1991, 1995) represent an effective application of knowledge from a real-world problem domain.

## 10 Conclusions

---

In this article, we described the architectures and operation of a wide variety of spatiotemporal connectionist networks. Rather than adopting an architecture-by-architecture approach, we described the networks by developing a taxonomy describing four fundamental design decisions. This permitted us to present many architectures more efficiently since there is considerable overlap in design decisions across different networks. Table 9 provides a comparative view of the works discussed in this review to give a clear picture of its representational structure and predictive power.

Prior to developing the taxonomy, we identified three properties that make taxonomies effective: descriptive adequacy, simplicity, and predictive power. We examined three taxonomies developed by Mozer (1994), Horne and Giles (1995), and Tsoi and Back (1994), and found that the recent flurry of research into STCNs has made them obsolete in terms of both predictive power and descriptive adequacy.

In response, we developed a new taxonomy based on four dimensions: how the state vector is computed, how the output vector is computed, how the changes in initial trainable parameters of the network are selected, and how these parameters are trained. These four dimensions were chosen because they represent the fundamental design decisions that any STCN system must address.

Next, we identified specific points along these four dimensions. Along the state-vector dimension we identified window in time (WIT) memories, nonlinear autoregressive with exogeneous inputs (NARX) memories, time-delay neural network (TDNN) memories, feedforward exponential decay (FED) memories, gamma memories, Jordan memories, local feedback (LF) memories, infinite impulse response (IIR) filter memories, first-order context (FOC) memories, second-order context (SOC) memories, long short-term memories (LSTM), connectionist pushdown automaton (CPA) memories, and connectionist Turing machine (CTM) memories. Along the output dimension we identified zero-layer, one-layer, and two-layer functions. We described randomization of initial parameters, split-state encoding, direct encoding, and chain graph encoding as points on the initializing parameters dimension. Finally, on the parameter change dimension, we defined full gradient descent (FGD), teacher forcing (TF), truncated gradient descent (TGD), autoassociative gradient descent (AAGD), stack learning (SL), training initial state (TIS), manual architecture changes (MAC), and automatically incrementing nodes (AIN).

Because the new taxonomy is the only one developed around the fundamental design decisions that must be addressed by any STCN system, it has superior predictive power when used to compare and analyze how different STCNs might perform on various problems. Furthermore, the fact that the taxonomy is based around the principles of spatiotemporal processing, rather than specific existing STCN designs, implies that it will easily accommodate future STCN designs as well.

## Acknowledgments

---

I acknowledge the helpful comments of Ramon Neco and three anonymous reviewers on an earlier draft of the manuscript for this article. I was supported by a research grant from the Natural Sciences and Engineering Council of Canada.

## References

---

- Alippi, C., & Piuri, V. (1996). Neural methodology for prediction and identification of non-linear dynamic system. In *Proceedings: International Workshop on Neural Networks for Identification, Control, Robotics, and Signal/Image Processings, Venice Italy, August 21–23, 1996* (pp. 305–313). Los Alamitos, CA: IEEE Computer Society Press.

- Angluin, D., & Smith, C. (1983). Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3), 237–269.
- Back, A., & Tsoi, A. (1990). A time series modelling methodology using FIR and IIR synapses. In F. Murtagh (Ed.), *Proc Workshop on Neural Networks for Statistical and Economic Data* (pp. 187–194). Dublin: DOSES, Statistical Office of European Communities.
- Back, A. D., & Tsoi, A. C. (1991). FIR and IIR synapses, a new neural network architecture for time series modelling. *Neural Computation*, 3(3), 375–385.
- Back, A. D., & Tsoi, A. C. (1993). A simplified gradient algorithm for IIR synapse multilayer perceptrons. *Neural Computation*, 5(3), 456–462.
- Back, A., Wan, E., Lawrence, S., & Tsoi, A. (1995). A unifying view of some training algorithms for multilayer perceptrons with FIR filter synapses. In J. Vlontzos, J. Hwang, & E. Wilson (Eds.), *Neural networks for signal processing*, 4 (pp. 146–154). New York: IEEE Press.
- Bengio, Y., Frasconi, P., & Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In *Proceedings of the 1993 IEEE International Conference on Neural Networks* (vol. 3), (pp. 1183–1188). San Francisco: IEEE Press.
- Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 157–166.
- Bulsari, A. B., & Saxén, H. (1995). A recurrent network for modeling noisy temporal sequences. *Neurocomputing*, 7, 29–40.
- Chen, S., Billings, S., & Grant, P. (1991). Non-linear system identification using neural networks. *International Journal of Control*, 51(6), 1191–1214.
- Cleeremans, A., & McClelland, J. L. (1991). Learning the structure of event sequences. *Journal of Experimental Psychology: General*, 120(3), 235–253.
- Cleeremans, A., Servan-Schreiber, D., & McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1(3), 372–381.
- Das, S., Giles, C. L., & Sun, G.-Z. (1993). Using prior knowledge in a NNPD to learn context-free languages. In S. J. Hanson, J. D. Cowan, & C. L. Giles (Eds.), *Advances in neural information processing systems*, 5 (pp. 65–72). San Mateo, CA: Morgan Kaufmann.
- de Vries, B., & Principe, J. M. (1991). A theory for neural networks with time delays. In R. P. Lippmann, J. E. Moody, & D. S. Touretzky (Eds.), *Advances in neural information processing systems/advances in neural information processing*, 3 (pp. 162–168). San Mateo, CA: Morgan Kaufmann.
- de Vries, B., & Principe, J. (1992a). The gamma model—A new neural network for temporal processing. *Neural Networks*, 5(4), 565–576.
- de Vries, B., & Principe, J. C. (1992b). The gamma model—A new neural model for temporal processing. *Neural Networks*, 5(4), 565–576.
- Elman, J. L. (1988). *Finding structure in time* (Tech. Rep. No. 8801). La Jolla, CA: Center for Research in Language, University of California at San Diego, University of California, San Diego.
- Elman, J. L. (1989). Structured representations and connectionist models. In *Proceedings of the 11th Annual Conference of the Cognitive Science Society* (pp. 17–25). Ann Arbor: University of Michigan.

- Elman, J. (1990a). Finding structure in time. *Cognitive Science*, 14, 179–211.
- Elman, J. (1990b). Representation and structure in connectionist models. In G. T. Altmann (Ed.), *Cognitive models of speech processing: Psycholinguistica and computational perspectives*. Cambridge, MA: MIT Press.
- Elman, J. (1991a). *Incremental learning, or the importance of starting small* (Tech. Rep. No. 9101). La Jolla, CA: Center for Research in Language, University of California at San Diego.
- Elman, J. L. (1991b). Distributed representations, simple recurrent networks and grammatical structure. *Machine Learning*, 7(2/3), 195–226.
- Fahlman, S. E. (1991). *The recurrent cascade-correlation architecture* (Tech. Rep. No. CMU-CS-91-100). Pittsburgh, PA: School of Computer Science, Carnegie-Mellon University.
- Forcada, M. L., & Carrasco, R. C. (1995). Learning the initial state of a second-order recurrent neural network during regular-language inference. *Neural Computation*, 7(5), 923–930.
- Frasconi, P., Gori, M., Maggini, M., & Soda, G. (1991). A unified approach for integrating explicit knowledge and learning by example in recurrent networks. In 1991 IEEE INNS International Joint Conference on Neural Networks—Seattle (Vol. 1, pp. 811–816). Piscataway, NJ: IEEE Press.
- Frasconi, P., Gori, M., & Soda, G. (1992). Local feedback in multilayered networks. *Neural Computation*, 4, 120–130.
- Frasconi, P., Gori, M., Maggini, M., & Soda, G. (1995). Unified integration of explicit rules and learning by example in recurrent networks. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), 340–346.
- Frasconi, P., & Gori, M. (1996). Computational capabilities of local-feedback recurrent networks acting as finite-state machines. *IEEE Transactions on Neural Networks*, 7(6), 1521–1525.
- Fu, K.-S. (1982). *Syntactic pattern recognition and applications*. Englewood Cliffs, NJ: Prentice-Hall.
- Ghosh, J., & Deuser, L. (1995). Classification of spatiotemporal patterns with applications to recognition of sonar sequences. In E. Covey (Ed.), *Neural representations of temporal patterns*. New York: Plenum.
- Giles, C., Sun, G., Chen, H., Lee, Y., & Chen, D. (1990). Higher order recurrent networks and grammatical inference. In D. S. Touretzky (Ed.), *Advances in neural information processing systems*, 2 (pp. 380–387). San Mateo, CA: Morgan Kaufmann.
- Giles, C. L., Chen, D., Miller, C., Chen, H., Sun, G., & Lee, Y. (1991). Second-order recurrent neural networks for grammatical inference. In 1991 IEEE INNS International Joint Conference on Neural Networks—Seattle (Vol. 2, pp. 273–281). Piscataway, NJ: IEEE Press.
- Giles, C. L., Miller, C. B., Chen, D., Sun, G. Z., Chen, H. H., & Lee, Y. C. (1992). Extracting and learning an unknown grammar with recurrent neural networks. In J. E. Moody, S. J. Hanson, & R. P. Lippmann (Eds.), *Advances in neural information processing systems*, 4 (pp. 317–324). San Mateo, CA: Morgan Kaufmann.
- Giles, C., Miller, C., Chen, D., Chen, H., Sun, G., & Lee, Y. (1992). Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3), 393–405.

- Giles, C. L., & Omlin, C. (1992). Inserting rules into recurrent neural networks. In S. Kung, F. Fallside, J. A. Sorenson, & C. Kamm (Eds.), *Neural Networks for Signal Processing II, Proceedings of the 1992 IEEE Workshop* (pp. 13–22). Piscataway, NJ: IEEE Press.
- Giles, C., & Omlin, C. (1993a). Rule refinement with recurrent neural networks. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN'93)* (Vol. 2, pp. 801–806).
- Giles, C. L., & Omlin, C. (1993b). Extraction, insertion and refinement of symbolic rules in dynamically-driven recurrent neural networks. *Connection Science*, 5(3,4), 307–337.
- Giles, C. L., & Horne, B. G. (1994). Representation and learning in recurrent neural network architectures. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems* (pp. 128–134). New Haven, CT: Center for Systems Science, Dunham Laboratory, Yale University.
- Giles, C. L., Chen, D., Sun, G.-Z., Chen, H.-H., Lee, Y.-C., & Goudreau, M. W. (1995). Constructive learning of recurrent neural networks: Limitations of recurrent cascade correlation and a simple solution. *IEEE Transactions on Neural Networks*, 6(4), 829–836.
- Gori, M., Bengio, Y., & De Mori, R. (1989). BPS: A learning algorithm for capturing the dynamic nature of speech. In *Proceedings of the First International Joint Conference on Neural Networks, Washington, DC* (Vol. 2, pp. 417–423). San Diego: IEEE TAB Neural Network Committee.
- Goudreau, M., Giles, C., Chakradhar, S., & Chen, D. (1994). First-order vs. second-order single-layer recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(3), 511–513.
- Hochreiter, J. (1991). *Untersuchungen zu dynamischen neuronalen Netzen*. Master's thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.
- Hochreiter, S., & Schmidhuber, J. (1996). *Long short term memory* (Tech. Rep. No. FKI-207-95). München: Fakultät fuer Informatik, Technische Universität München.
- Hochreiter, S., & Schmidhuber, J. (1997a). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
- Hochreiter, S., & Schmidhuber, J. (1997b). LSTM can solve hard long time lag problems. In M. C. Mozer, M. I. Jordan, & T. Petsche (Eds.), *Advances in neural information processing systems*, 9 (pp. 473–479). Cambridge, MA: MIT Press.
- Hopcroft, J., & Ullman, J. (1979). *Introduction to automata theory, languages, and computation*. Reading, MA: Addison-Wesley.
- Horne, B. G., & Giles, C. L. (1995). An experimental comparison of recurrent neural networks. In G. Tesauro, D. Touretzky, & T. Leen (Eds.), *Advances in neural information processing systems*, 7 (pp. 697–704). Cambridge, MA: MIT Press.
- Jordan, M. I. (1986a). *Serial order: A parallel distributed processing approach* (Tech. Rep. No. ICS Report 8604). La Jolla, CA: Institute for Cognitive Science, University of California at San Diego.
- Jordan, M. I. (1986b). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (pp. 531–546). Hillsdale, NJ: Erlbaum.

- Kohavi, Z. (1978). *Switching and finite automata theory* (2nd ed). New York: McGraw-Hill.
- Kolen, J. (1994). Recurrent networks: State machines or iterated function systems? In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, & A. Weigend (Eds.), *Proceedings of the 1993 Connectionist Models Summer School*. Hillsdale, NJ: Erlbaum.
- Kremer, S. C. (1995). On the computational power of Elman-style recurrent networks. *IEEE Transactions on Neural Networks*, 6(4), 1000–1004.
- Kremer, S. C. (1996a). Comments on “constructive learning of recurrent neural networks: Limitations of recurrent cascade correlation and a simple solution.” *IEEE Transactions on Neural Networks*, 7(4). [References in this article were incorrectly numbered in the published version: “[2]” should read “[1],” “[3]” should read “[2],” and “[4]” should read “[3]”.]
- Kremer, S. C. (1996b). Finite state automata that recurrent cascade-correlation cannot represent. In D. S. Touretzky, M. C. Mozer, & M. E. Hasselmo (Eds.), *Advances in neural information processing systems*, 8 (pp. 612–618). MIT Press.
- Kremer, S. C. (1999). Identification of a specific limitation on local-feedback recurrent networks acting as Mealy Moore machines. *IEEE Transactions Neural Networks*, 10, 433–438.
- Lang, K. J., Waibel, A. H., & Hinton, G. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3(1), 23–43.
- Lapedes, A., & Farber, R. (1987). *Nonlinear signal processing using neural networks prediction and system modelling*. (Tech. Rep. No. LA-UR87-2662). Los Alamos: Los Alamos National Laboratory.
- Lawrence, S., Tsoi, A. C., & Back, A. D. (1996). The gamma MLP for speech phoneme recognition. In D. S. Touretzky, M. C. Mozer, & M. E. Hasselmo, (Eds.), *Advances in neural information processing systems*, 8 (pp. 785–791). Cambridge, MA: MIT Press.
- Lin, T., Giles, C. L., Horne, B. G., & Kung, S.-Y. (1996). A delay damage model selection algorithm for NARX neural networks (Tech. Rep. Nos. CS-TR-3707 and UMIACS-TR-96-77). College Park, MD: Institute for Advanced Computer Studies, University of Maryland.
- Lin, T., Horne, B., Tiño, P., & Giles, C. (1996a). Learning long-term dependencies is not as difficult with NARX recurrent neural networks. In D. Touretzky, M. Mozer, & M. Hasselmo (Eds.), *Advances in neural information processing systems*, 8 (p. 577). Cambridge, MA: MIT Press.
- Lin, T., Horne, B. G., Tiño, P., & Giles, C. L. (1996b). Learning long-term dependencies in NARX recurrent neural networks. *IEEE Transactions on Neural Networks*, 7(6), 1329–1338.
- Lin, T., Giles, C., & Horne, B. G. (1998). What to remember: How memory orders affect the performance of NARX networks. In *IEEE World Conference on Computational Intelligence* (p. 1051).
- Lippmann, R. P. (1987). An introduction to computing with neural nets. *IEEE ASSP Magazine*, pp. 4–22.
- Liu, Y., Sun, G., Chen, H., Lee, Y., & Giles, C. (1990). Grammatical inference and neural network state machines. In *Proceedings of the International Joint Conference on Neural Networks 1990* (Vol. 1, pp. 285–288).

- Maass, W., & Orponen, P. (1997). On the effect of analog noise in discrete-time analog computations. In *Proceedings of the Annual Conference on Neural Information Processing Systems 1996, in Denver, USA (Advances in Neural Information Processing Systems, vol. 9)*, Cambridge, MA: MIT Press.
- Maass, W., & Orponen, P. (1998). On the effect of analog noise in discrete-time analog computations. *Neural Computation*, 10(5), 1071–1095.
- Maren, A. (1990). Neural networks for spatiotemporal recognition. In A. Maren (Ed.), *Handbook of neural computing applications*. Orlando, FL: Academic Press.
- Marr, D. (1982). *Chapter 1: The philosophy and the approach*. San Francisco: W. H. Freeman.
- Maskara, A., & Noetzel, A. (1992). Forced simple recurrent neural networks and grammatical inference. In *Proceedings of the 14th Conference of the Cognitive Science Society* (pp. 420–425).
- Miller, C., & Giles, C. L. (1993). Experimental comparison of the effect of order in recurrent neural networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(4), 849–872.
- Mozer, M. (1989). A focused backpropagation algorithm for temporal pattern processing/recognition. *Complex Systems*, 3(4), 349–381.
- Mozer, M. C. (1994). Neural net architectures for temporal sequence processing. In A. Weigend & N. Gershenfeld (Eds.), *Time series prediction* (pp. 243–264). Reading, MA: Addison–Wesley.
- Narendra, K. S., & Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1(1), 4–27.
- Nerrand, O., Rousel-Ragot, P., Personnaz, L., Dreyfus, G., & Marcos, S. (1993). Neural networks and nonlinear adaptive filtering: Unifying concepts and new algorithms. *Neural Computation*, 5, 165–199.
- Omlin, C., & Giles, C. L. (1992). Training second-order recurrent neural networks using hints. In D. Sleeman & P. Edwards (Eds.), *Proceedings of the Ninth International Conference on Machine Learning* (pp. 363–368). San Mateo, CA: Morgan Kaufmann.
- Omlin, C., & Giles, C. (1994a). Constructing deterministic finite-state automata in sparse recurrent neural networks. In *IEEE International Conference on Neural Networks (ICNN'94)* (pp. 1732–1737). Piscataway, NJ: IEEE Press.
- Omlin, C., & Giles, C. L. (1994b). Extraction and insertion of symbolic information in recurrent neural networks. In V. Honavar & L. Uhr (Eds.), *Artificial intelligence and neural networks: Steps toward principled integration* (pp. 271–299). San Diego, CA: Academic Press.
- Omlin, C., & Giles, C. (1996a). Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM*, 43(6), 937–972.
- Omlin, C. W., & Giles, C. L. (1996b). Stable encoding of large finite-state automata in recurrent neural networks with sigmoid discriminants. *Neural Computation*, 8(7), 675–696.
- Omlin, C., Thornber, K., & Giles, C. (1998). Fuzzy finite-state automata can be deterministically encoded into recurrent neural networks. *IEEE Transactions on Fuzzy Systems*, 6(1), 76–89.

- Pearlmutter, B. A. (1995). Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5), 1212–1228.
- Poddar, P., & Unnikrishnan, K. (1991a). *Memory neuron networks: A prolegomenon* (Tech. Rep. No. GMR-7493). Warren, MI: Computer Science Department, General Motors Research Laboratories.
- Poddar, P., & Unnikrishnan, K. (1991b). Nonlinear prediction of speech signals using memory neuron networks. In B. Juang, S. Kung, and C. Kamm (Eds.), *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE Workshop* (pp. 395–404). New York: IEEE Press.
- Pollack, J. (1989). Implications of recursive distributed representations. In D. Touretzky (Ed.), *Advances in neural information processing systems*, 1 (pp. 527–536). San Mateo, CA: Morgan Kaufmann.
- Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence*, 46, 77–105.
- Pollack, J. (1994). *Chapter 4: Limits of connectionism* (Tech. Rep.). Columbus, OH: Department of Computer Science, Ohio State University.
- Robinson, A. J., & Fallside, F. (1988). Static and dynamic error propagation networks with application to speech coding. In D. Z. Anderson (Ed.), *Neural information processing systems* (pp. 632–641). New York; American Institute of Physics.
- Rosenberg, C. R., & Sejnowski, T. J. (1986). The spacing effect on “nettalk,” a massively-parallel network. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (pp. 72–89). Hillsdale, NJ: Erlbaum.
- Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representation by error propagation. In J. L. McClelland, D. Rumelhart, & the PDP Group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1). Cambridge, MA: MIT Press.
- Schmidhuber, J. H. (1992). A fixed size storage  $o(n^3)$  time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2), 243–248.
- Sejnowski, T., & Rosenberg, C. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, 1, 145–168.
- Sejnowski, T. J., & Rosenberg, C. R. (1988). NETtalk: A parallel network that learns to read aloud. In J. A. Anderson & E. Rosenfeld (Eds.), *Neurocomputing: Foundations of research* (pp. 663–672). Cambridge, MA: MIT Press.
- Servan-Schreiber, D., Cleeremans, A., & McClelland, J. L. (1988). *Encoding sequential structure in simple recurrent networks* (Tech. Rep. CMU-CS-88-183). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University.
- Servan-Schreiber, D., Cleeremans, A., & McClelland, J. (1989). Learning sequential structure in simple recurrent networks. In D. Touretzky (Ed.), *Advances in neural information processing systems*, 1. San Mateo, CA: Morgan Kaufmann.
- Servan-Schreiber, D., Cleeremans, A., & McClelland, J. (1991). Graded state machines: The representation of temporal contingencies in simple recurrent networks. *Machine Learning*, 7, 161–193.
- Servan-Schreiber, D., Cleeremans, A., & McClelland, J. L. (1994). *Graded state machines: The representation of temporal contingencies in simple recurrent networks* (pp. 241–269). San Diego, CA: Academic Press.

- Shaw, A., & Mitchell, R. (1990). Phoneme recognition with a time-delay neural network. In *International Joint Conference on Neural Networks, San Diego* (Vol. 2, pp. 191–195). New York: IEEE.
- Siegelmann, H., Horne, B., & Giles, C. (1997). Computational capabilities of recurrent NARX neural networks. *IEEE Trans. on Systems, Man & Cybernetics*, 27, 208.
- Stiles, B. W., & Ghosh, J. (1996). Two stage habituation based neural networks for dynamic signal classification. In C. T. Leondes (Ed.), *Computer techniques and algorithms in digital signal processing techniques and applications* (pp. 301–337). San Diego, CA: Academic Press.
- Stiles, B. W., & Ghosh, J. (1997). A habituation based neural network for spatiotemporal classification. *Neurocomputing*, 15(3,4), 273–307.
- Sun, G., Chen, H., Giles, C., Lee, Y., & Chen, D. (1990a). Connectionist pushdown automata that learn context-free grammars. In M. Caudill (Ed.), *Proceedings of the International Joint Conference on Neural Networks 1990* (Vol. 1, pp. 577–580). Hillsdale, NJ: Erlbaum.
- Sun, G., Chen, H., Giles, C., Lee, Y., & Chen, D. (1990b). Neural networks with external memory stack that learn context-free grammars from examples. In *Proceedings of the 1990 Conference on Information Sciences and Systems* (Vol. 2, pp. 649–653). Princeton, NJ: Department of Electrical Engineering, Princeton University.
- Sun, G., Chen, H., Lee, Y., & Giles, C. L. (1991). Turing equivalence of neural networks with second order connection weights. In *1991 IEEE INNS International Joint Conference on Neural Networks—Seattle* (Vol. 2, pp. 357–362). Piscataway, NJ: IEEE Press.
- Sun, G.-Z., Chen, H.-H., & Lee, Y.-C. (1992). Green's function method for fast on-line learning algorithm of recurrent neural networks. In J. E. Moody, S. J. Hanson, & R. P. Lippmann (Eds.), *Advances in neural information processing systems*, 4 (pp. 333–340). San Mateo, CA: Morgan Kaufmann.
- Sun, G., Giles, C., Chen, H., & Lee, Y. (1993). *The neural network pushdown automaton: Model, stack and learning simulations* (Tech. Rep. No. UMIACS-TR-93-77/CS-TR-3118). College Park, MD: University of Maryland.
- Tsoi, A. C., & Back, A. D. (1994). Locally recurrent globally feedforward networks: A critical review of architectures. *IEEE Transactions on Neural Networks*, 5(2), 229–239.
- Unnikrishnan, K. P., & Venugopal, K. P. (1994). Alopex: A correlation-based learning algorithm for feedforward and recurrent neural networks. *Neural Computation*, 6(3), 469–490.
- Waibel, A. (1988). Consonant recognition by modular construction of large phonemic time-delay neural networks. In D. Anderson (Ed.), *Neural information processing systems* (pp. 215–223). New York: American Institute of Physics.
- Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(3), 328–339.
- Watrous, R. L., & Kuhn, G. M. (1992a). Induction of finite-state automata using second-order recurrent networks. In J. E. Moody, S. J. Hanson, & R. P. Lipp-

- mann (Eds.), *Advances in neural information processing systems*, 4 (pp. 309–316). San Mateo, CA: Morgan Kaufmann.
- Watrous, R. L., & Kuhn, G. M. (1992b). Induction of finite-state languages using second-order recurrent networks. *Neural Computation*, 4(3), 406–414.
- Williams, R. J., & Zipser, D. (1988). *A learning algorithm for continually running fully recurrent neural networks* (Tech. Rep. No. ICS 8805). La Jolla, CA: Institute for Cognitive Science, University of California at San Diego.
- Williams, R. J., & Zipser, D. (1989a). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1), 87–111.
- Williams, R. J., & Zipser, D. (1989b). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2), 270–280.
- Zeng, Z., Goodman, R. M., & Smyth, P. (1994). Discrete recurrent neural networks for grammatical inference. *IEEE Transactions on Neural Networks*, 5(2), 320–330.

---

Received December 3, 1998; accepted March 20, 2000.