Using Recurrent Neural Networks to Learn the Structure of Interconnection Networks*
UNIVERSITY OF MARYLAND TECHNICAL REPORT UMIACS-TR-94-20 AND CS-TR-3226

Mark W. Goudreau[a] and C. Lee Giles[b,c]

[a]Department of Computer Science
University of Central Florida, Orlando, FL 32816

[b]NEC Research Institute
4 Independence Way, Princeton, NJ 08540

[c]Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742

15 February 1994

Abstract

A modified Recurrent Neural Network (RNN) is used to learn a Self-Routing Interconnection Network (SRIN) from a set of routing examples. The RNN is modified so that it has *several distinct initial states*. This is equivalent to a single RNN learning multiple different synchronous sequential machines. We define such a sequential machine structure as *augmented* and show that a SRIN is essentially an Augmented Synchronous Sequential Machine (ASSM). As an example, we learn a small six-switch SRIN. After training we extract the network's internal representation of the ASSM and corresponding SRIN.

*This paper is adapted from (Goudreau, 1993, Chapter 6). A shortened version of this paper was published in (Goudreau & Giles, 1993).

1 Introduction

The use of Recurrent Neural Networks (RNNs) to learn Synchronous Sequential Machines (SSMs) from examples is a problem which has been studied extensively. A related topic that, to the authors' knowledge, has not been studied previously is the use of RNNs to learn SSMs for which *several distinct initial states* are possible.

This problem is interesting because it maps directly into the problem of learning the structure of an Interconnection Network (IN) from examples. Learning an IN from examples is an unusual approach. Traditionally, INs have been designed (and not learned) based on several criteria, including speed, complexity, ease of route calculation, and fault tolerance. Numerous different types of INs have been proposed. A detailed description of many of the INs that have been applied to parallel computing can be found in Siegel's book (Siegel, 1990).

In this paper, the learning of Self-Routing Interconnection Networks (SRINs) is discussed. SRINs are described in detail in Section 2. They can be used to describe many commonly used INs. If one considers a parallel computing system, the idea is that the processors have certain communication requirements with other processors, and certain message headers (also described in Section 2) must be used that allow the message to pass through the SRIN and reach the desired destination processor. The message headers provide routing information to the switches in the SRIN.

The method that is proposed makes use of a second-order Single-Layer Recurrent Neural Network (SLRNN) to learn the training data. The training data is a table of source processors, message headers, and destination processors. Once the training data has been learned, the structure of the SRIN can be extracted from the SLRNN.

One topic that is related to the learning of INs was presented by Hillis (Hillis, 1990). In that paper, Hillis makes use of simulated evolution to construct sorting networks. It should be also be mentioned that neural networks have been previously used for interconnection network routing: for example, see (Brown, 1989; Brown & Liu, 1990; Funabiki et al., 1991; Funabiki et al., 1993; Goudreau & Giles, 1992; Hakim & Meadows, 1990; Lee & Chang, 1993; Marrakchi & Troudet, 1989; Melsa et al., 1990a; Melsa et al., 1990b; Takefuji & Lee, 1991; Thomopoulos et al., 1991; Troudet & Walters, 1991). However, none of these methods learned the structure of the interconnection networks; the structure of the interconnection network was always directly mapped into the neural network.

2 Self-Routing Interconnection Networks

In this section we describe SRINs. The purpose of a SRIN is to allow a set of processors to communicate amongst themselves using a *store-and-forward* method-

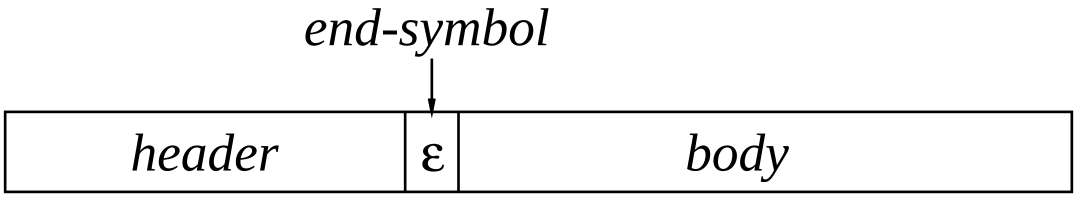

Figure 1: A message for a SRIN. The header is a string of symbols. The body is also a string of symbols. The header is separated from the body by the end-symbol, ϵ.

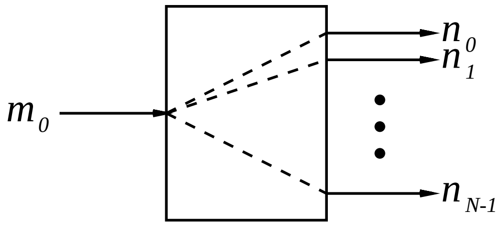

Figure 2: A $1 \times N$ self-routing switch. If the heading symbol is a i_j, the message is routed from input m_0 to output n_j. Messages are buffered until they can be routed. Messages are routed in a First-In, First-Out (FIFO) basis.

ology. For store-and-forward routing, a message travels along the path towards its destination one step at a time. Once a switch has sent a message to the next switch, it is free to be used to route a different message.

A SRIN does not use an external controller to route messages. Rather, the switches in a SRIN are *smart*; they examine the message that is being sent and decide which way to route it.

A message, as it is defined in this paper, consists of two parts: a *header* and a *body*. The header and the body are separated by an *end-symbol*, which will be denoted by ϵ. A schematic diagram of a message is shown in Figure 1.

The SRIN uses *self-routing switches* to route messages. A self-routing switch with M inputs and N outputs will be called a $M \times N$ self-routing switch. A drawing of a $1 \times N$ self-routing switch is shown in Figure 2. Figure 3 is a drawing of a $M \times N$ self-routing switch.

Intuitively, the routing works in the following manner. When a message arrives at a switch, the switch strips off the first symbol of the message header and examines it. If the symbol is not the end-symbol, ϵ, the switch sends the message (minus the header symbol that it just examined) through the appropriate output port. If the

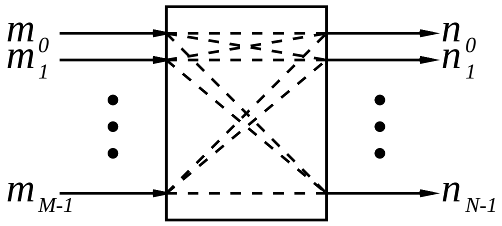

Figure 3: An $M \times N$ self-routing switch. Again, input messages are routed on a First-In, First-Out (FIFO) basis.

symbol is the end-symbol, then the message should be given to the processor that is associated with the switch. In other words, once the header has been stripped down so that only the end-symbol is left, the message does not get passed through the SRIN any longer.

Although it is not shown in Figures 2 and 3, it must be remembered that there is a connection from each switch to its associated processor. This can be thought of as another output port for the end-symbol, ϵ.

The switches work in a First-In, First-Out (FIFO) manner. If a message can not be routed immediately, it is buffered until it can be routed.

2.1 A Formal Description of a Self-Routing Interconnection Network

We will now present a more formal description of an SRIN. The SRIN will have a set of M processors, $P = \{p_0, p_1, \ldots, p_{M-1}\}$. Each processor, p_j, will be associated with a set of switches, Q_j. Each set Q_j must contain *at least* one switch. (Otherwise, the processor would have no way to communicate with the other processors.)

Note that not all of the switches in the SRIN need to be associated with a processor. Some switches in an SRIN might never be used to connect to a processor. Such switches are called *don't care* switches, or *free* switches. A message can be routed through a free switch, but a free switch should never be the first switch nor last switch in a route; to do so would imply that the free switch is associated with some processor. For the sake of convenience, we will associate some processor with each free switch, even though such an association is meaningless since it is never used. Now, the SRIN has the set of switches $Q = Q_0 \cup Q_1 \cup \cdots \cup Q_{M-1}$. The processor function, β, performs the mapping, $\beta : Q \rightarrow P$. That is, if q is a switch, then $\beta(q)$ is the processor associated with that switch.

We will let R be the finite input alphabet for the header and the body. The end-symbol, ϵ, is not a member of R; that is, $\epsilon \notin R$. The end-symbol is only used to separate the header from the body. One typical alphabet would be $R = \{r_0, r_1\}$. In general, however, the magnitude of the alphabet can be greater than two. Since most computing environments are binary, the situation becomes more complicated when the magnitude of the alphabet is greater than two. In such cases, the members of the alphabet must be encoded in some way.

There does not need to be a size limitation for the header nor the body. In a binary system, the end-symbol might consist of a string of zeros and ones that is illegal in the header. Alternatively, one might send the header and the body separately, in which case the position of the end-symbol will be understood by the receiving switch. Another approach would be to designate the first byte of the header to represent the length of the header. There are many different ways to implement the end-symbol, but for our purposes here we will assume the end-symbol is just a symbol that can be transmitted in one time step.

We now define the switch transition function, ϕ, which performs the mapping, $\phi : Q \times R \to Q$. If q is a switch and r is the input symbol that is taken from the front of the header, then $\phi(q, r)$ is the next switch that the message will be sent to.

Finally, when processor p_j sends a message, it starts the message off from one of the switches in the set Q_j. Each processor will have a switch that is designated for this purpose. We define the switch function, γ, which performs the mapping, $\gamma : P \to Q$. If p is a processor, then $\gamma(p)$ is the switch that performs the first stage of the routing for any messages that p sends. We will call the switch $\gamma(p)$ the *designated* switch for processor p.

The SRIN can now be defined formally.

Definition 1 *A self-routing interconnection network is a 7-tuple,* $(P, Q, R, \phi, \beta, \gamma, \epsilon)$*, where:*

- P *is a finite, nonempty set of processors.*
- Q *is a finite, nonempty set of switches.*
- R *is a finite, nonempty set of input symbols.*
- $\phi : Q \times R \to Q$ *is the switch transition function.*
- $\beta : Q \to P$ *is the processor function.*
- $\gamma : P \to Q$ *is the switch function.*
- ϵ *is the end-symbol.*

2.2 An Example of a Self-Routing Interconnection Network

Figure 4 shows an example of a SRIN. This SRIN has the set of switches $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$. In Figure 4, the switches are shown as white boxes with their labels in the upper left corner. The outputs, labeled r_0 and r_1, are on the right side of each switch. The inputs come to the left side of the switch. A switch with a * in its lower left corner is a designated switch.

The set of processors is $P = \{p_0, p_1, p_2, p_3\}$. In Figure 4, the processors are represented by shaded areas.

For this example, we have the input alphabet $R = \{r_0, r_1\}$. Thus, each switch has two output ports. In practice, not all of the output ports need to be connected; some can be *don't cares* if they are never used for routing.

The number of input ports for each switch can be zero or any larger integer. If a switch has zero input ports, it must be a designated switch or it will have no purpose in the SRIN.

The processor function, β, is shown here:

$$\begin{aligned} \beta(q_0) &= p_0 \\ \beta(q_1) &= p_0 \\ \beta(q_2) &= p_1 \\ \beta(q_3) &= p_2 \\ \beta(q_4) &= p_3 \\ \beta(q_5) &= p_3 \end{aligned} \tag{1}$$

The switch transition function, ϕ, is shown here:

$$\begin{array}{ll} \phi(q_0, r_0) = q_4 & \phi(q_0, r_1) = q_2 \\ \phi(q_1, r_0) = q_3 & \phi(q_1, r_1) = q_3 \\ \phi(q_2, r_0) = q_1 & \phi(q_2, r_1) = q_4 \\ \phi(q_3, r_0) = q_0 & \phi(q_3, r_1) = q_4 \\ \phi(q_4, r_0) = q_1 & \phi(q_4, r_1) = q_5 \\ \phi(q_5, r_0) = q_5 & \phi(q_5, r_1) = q_2 \end{array} \tag{2}$$

Finally, the switch function, γ, is shown here:

$$\begin{aligned} \gamma(p_0) &= q_0 \\ \gamma(p_1) &= q_2 \\ \gamma(p_2) &= q_3 \\ \gamma(p_3) &= q_4 \end{aligned} \tag{3}$$

Suppose processor p_1 has data to send to processor p_2. One possible way to send the data there is to use the header $r_1 r_0 r_1$. The message starts in switch $q_2 = \gamma(p_1)$. The

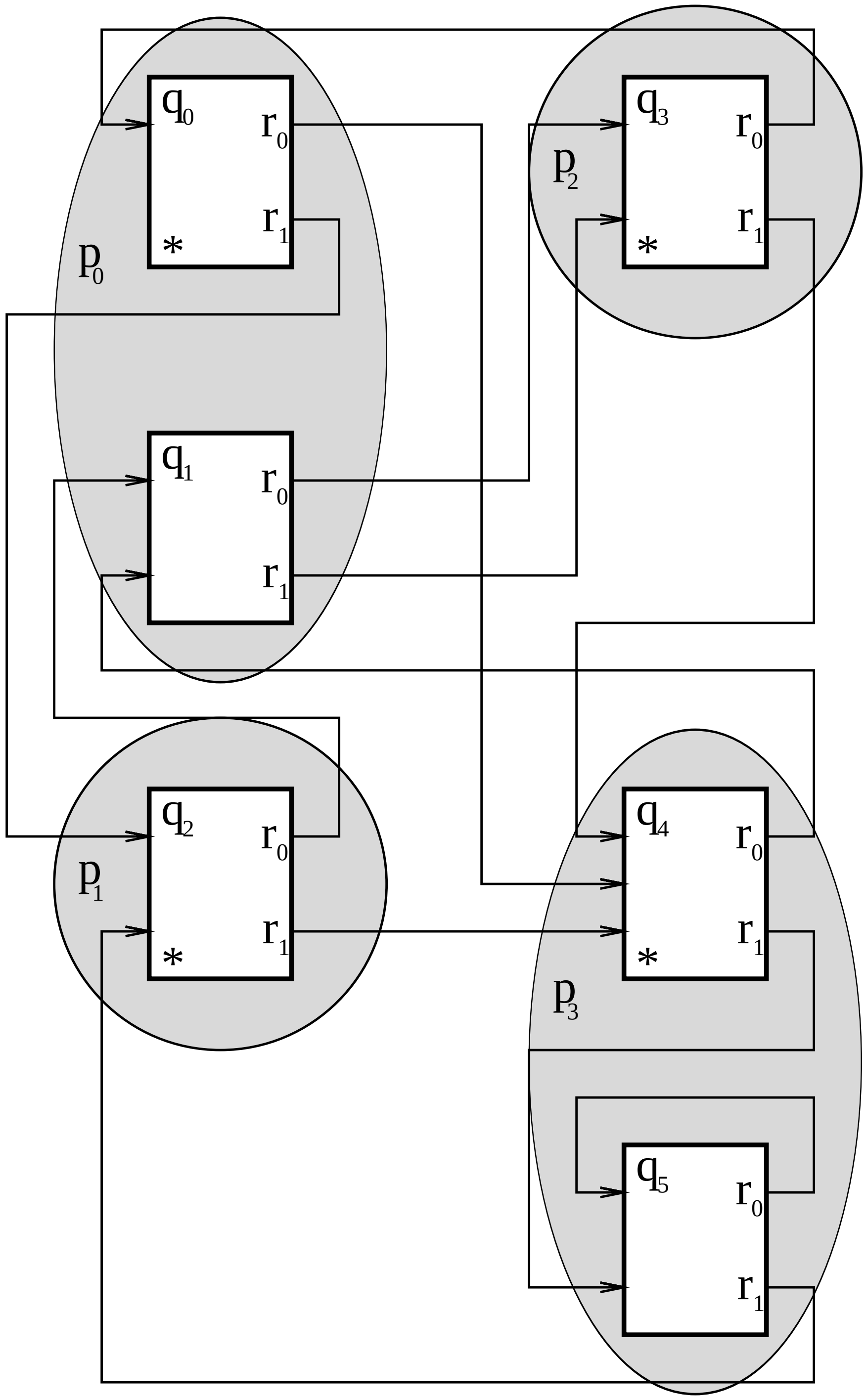

Figure 4. A sample self-routing interconnection network.

switch q_2 strips off the left-most symbol in the header, in this case r_1, and routes the message to $q_4 = \phi(q_2, r_1)$. The message then goes to switch $q_1 = \phi(q_4, r_0)$, and at last to switch $q_3 = \phi(q_1, r_1)$. At this point, the header has been spent and the message is led by the end-symbol. Switch q_3 therefore delivers the message to processor $p_2 = \beta(q_3)$.

3 Synchronous Sequential Machines

In this section we discuss the relationship between SSMs and SRINs.

FSAs and SSMs are thoroughly described in (Hopcroft & Ullman, 1979; Kohavi, 1978). We will use the definition of SSMs that is provided in (Kohavi, 1978). [1]

Definition 2 *A synchronous sequential machine is a quintuple,* $(O, S, I, \delta, \lambda)$*, where:*

- O *is a finite, nonempty set of outputs symbols.*
- S *is a finite, nonempty set of states.*
- I *is a finite, nonempty set of inputs symbols.*
- $\delta : S \times I \to S$ *is the state transition function.*
- $\lambda : S \to O$ *is the output function.*

¿From Definitions 1 and 2, it is clear that SRINs and SSMs are very similar. In fact, it only takes a slight expansion of the definition of SSMs to make them directly equivalent to SRINs. We will describe how SRINs are equivalent to Augmented SSMs (ASSMs), which will be defined below.

Let each processor in P be an output symbol in O. Similarly, let switch in Q be a state in S, and each input symbol in R be an input symbol in I. The switch transition function, ϕ, becomes the state transition function, δ. The processor function, β, becomes the output function, λ.

Now the only components of the SRIN that are not equivalent to components in the SSM are the end-symbol, ϵ, and the switch function, γ. The ASSM will have an end-symbol, κ. The meaning of the end-symbol in this context is merely that the input string has reached its conclusion, and the ASSM can now output the value corresponding to the input string. The ASSM will also have a state function, ρ. The state function ρ performs the mapping, $\rho : O \to S$. In this context, the state

[1] Specifically, our definition is for a Moore machine.

function allows for some set of initial states in the ASSM. Thus, each input string that is to be entered into the ASSM *must* have an output symbol associated with it. This output symbol allows the ASSM to choose the correct starting state.

The ASSM can now be defined formally.

Definition 3 *As augmented synchronous sequential machine is an 7-tuple,* $(O, S, I, \delta, \lambda, \rho, \kappa)$*, where:*

- O *is a finite, nonempty set of outputs.*
- S *is a finite, nonempty set of states.*
- I *is a finite, nonempty set of inputs.*
- $\delta : S \times I \rightarrow S$ *is the state transition function.*
- $\lambda : S \rightarrow O$ *is the output function.*
- $\rho : O \rightarrow S$ *is the state function.*
- κ *is the end-symbol.*

It is now clear from Definitions 1 and 3 that SRINs and ASSMs are equivalent.

4 Machine Inference

Since SRINs and ASSMs are equivalent, there are many issues that have been explored for ASSMs that can now be used for SRINs. For example, just as one can minimize the size of an ASSM by merging equivalent states (Kohavi, 1978), one can minimize the size of a SRIN by merging equivalent switches.

What we are interested in is the *inference* of a SRIN from *examples*. A great deal of work has been done on the problem of *machine inference*. It has been shown that, in the worst case, inferring a SSM from sparse data is an intractable problem (Angluin, 1978; Gold, 1978; Kearns & Valiant, 1989; Pitt & Warmuth, 1989). Approaches that can be used to infer SSMs will now be examined.

4.1 Recurrent Neural Network Approaches

The literature on the use of neural networks for grammatical inference and finite-state machine learning is now well-established (Cleeremans et al., 1989; Giles et al., 1992; Giles et al., 1992; Mozer & Bachrach, 1991; Pollack, 1991; Watrous & Kuhn, 1992; Zeng et al., 1993). These approaches use RNNs to represent SSMs. For the work done in this paper, the approach described in (Giles et al., 1992; Giles et al., 1992) will be used (see Section 5.2). We refer readers who are interested in the details to those references. In Section 5, there is a qualitative explanation of the RNN approach to learning SRINs.

Until recently, the RNN approach for SSM inference that is used in this paper had only been possible for unknown SSMs with a small number of states (approximately 30). It should be pointed out that the limited success of this approaches is due to the learning algorithms. Generally, the RNNs have rich representational capabilities. However, recent work has shown that certain types of large SSMs, with thousands of states, are learnable (Giles & Horne, 1994). Furthermore, the performance of the RNNs can sometimes be improved by using "hints" if partial information about the structure of the SSM is known (Giles & Omlin, 1993).

Other approaches that use neural networks for grammatical inference exist that will not be used in this paper. For example, the use of *update graphs* has been proposed by Rivest and Schapire (Rivest & Schapire, 1987a; Rivest & Schapire, 1987b; Schapire, 1988). An update graph is an alternate representation of a FSA that can be much smaller than the FSA for certain environments that often arise in practice. Update graphs can be mapped to a connectionist system that can learn the environment from examples (Mozer & Bachrach, 1990; Mozer & Bachrach, 1991).

4.2 Traditional Approaches

Other methods for grammatical inference, which do not use neural networks, have demonstrated some promising results. In fact, a polynomial time algorithm proposed by Trakhtenbrot and Barzdin (Trakhtenbrot & Barzdin, 1973) has been shown to be able to infer some very large finite automata. The algorithm produces a machine that is consistent with a sparsely labeled tree, but the machine that is produced is not necessarily the minimum machine that is consistent with the data. Lang (Lang, 1992) performed several experiments using this algorithm for random finite automata with 1000 states and 2000 transitions. Given enough training examples, the algorithm was almost always able to construct a machine that was similar to the correct machine.

source processor	header	destination processor
$\vdots$	$\vdots$	$\vdots$
p_0	r_1	p_1
p_2	$r_1 r_1 r_0 r_0 r_1$	p_1
p_2	$r_0 r_0 r_0$	p_0
p_3	$r_0 r_1$	p_2
p_1	$r_1 r_1 r_0$	p_3
$\vdots$	$\vdots$	$\vdots$

Table 1: An example of training data for a SRIN. The data is consistent with the SRIN in Figure 4.

5 Recurrent Neural Networks to Learn Interconnection Networks

The problem that we are trying to solve is posed in the following form. We have a training list of source processor, header, and destination processor combinations that must be implemented by a SRIN. We must infer an SRIN that can accomplish all of the routings described in the training list. Hopefully, the SRIN will also be able to *generalize*. That is, we would like the SRIN to perform correct routings even for examples that are not on the training list.

For example, Table 1 contains data for some such problem. The data in Table 1 is consistent with the SRIN in Figure 4.

5.1 Recurrent Network with Several Distinct Initial States

The structure of a second order SLRNN is shown in Figure 5. There are M inputs lines, $x_1, x_2, \ldots, x_M$. The value of input x_i $(1 \le i \le M)$ at time t is x_i^t. There is a single layer of N neurons, $y_1, y_2, \ldots, y_N$. The output value of neuron y_i $(1 \le i \le N)$ at time t is y_i^t. At each time step, these output values are stored in a bank of latches to act as the "state" of the network. The state is fed back as an input to the layer of neurons on the subsequent time step.

For a second order SLRNN, neuron j and input k to have a combined effect on neuron i that is quantified by the parameter w_{ijk}. The output of neuron i is defined by the following equation:

$$y_i^t = g\left(\sum_{j=1}^{N}\sum_{k=1}^{M} w_{ijk} y_j^{t-1} x_k^t\right) \tag{4}$$

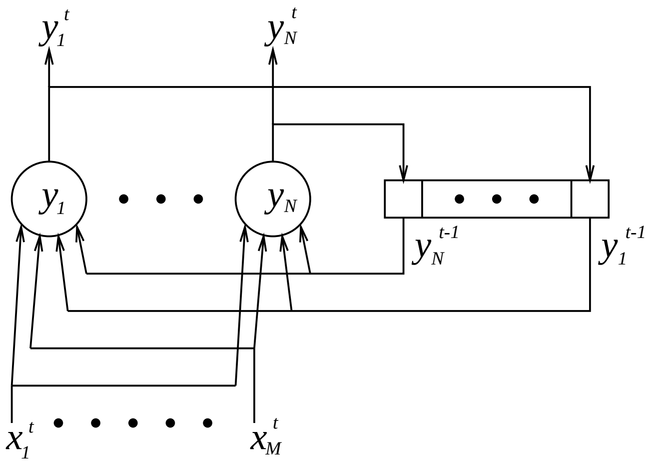

Figure 5: A Single-Layer Recurrent Neural Network (SLRNN). There are M input bits, N state bits, and (up to) N output bits. The bank of N latches is shown on the right.

The activation function, $g(x)$, is the sigmoid function shown here:

$$g(x) = \frac{1}{1 + e^{-ax}} \tag{5}$$

The second order SLRNN is used to infer the ASSM that is equivalent to the unknown SRIN. Again, the approach used in (Giles et al., 1992; Giles et al., 1992) will be used here. The SLRNN will learn the training data, and the ASSM will be extracted from the SLRNN. A gradient descent algorithm that is similar to the one proposed by Williams and Zipser (Williams & Zipser, 1989) will be used to train the SLRNN. If it were not possible to extract the ASSM from the SLRNN, then the SLRNN would not be useful for this problem. Without ASSM extraction, the SLRNN could still generalize, but when the structure of a SRIN is the desired result such generalization is not useful.

The problem of inferring an ASSM, as opposed to inferring an SSM, is that in an ASSM there will generally be several possible initial states. In an SSM, a single initial state is generally assumed. With this in mind, the reader should be able to see that the SLRNN that is trying to learn an ASSM will, in fact, be trying to learn a SSM for each possible initial state. However, all of these SSMs will have the same structure; the only difference is that the initial state varies. Since all of the SSMs have the same structure, it can reasonably be hoped that the SLRNN will try to merge these multiple SSMs into a single ASSM. In fact, this turns out to be the case. Rather than learning several different SSMs in different sections of the state space, the SLRNN will take advantage of the identical structures of the SSMs and merge them.

Intuitively, the input vectors of the SLRNN will represent the inputs and the end-symbol of the ASSM (and therefore the input symbols and the end-symbol of the SRIN). The state vectors of the SLRNN represent the states of the ASSM (and the switches of the SRIN). And the output vectors of the SLRNN represent the outputs of the the ASSM (and the processors of the SRIN).

We will use simple one-hot encodings for the input symbols and the output symbols. Recall that a one-hot code is a code for which each symbol is represented by a vector that has one element equal to one while all of the other elements are equal to zero. Additionally, the initial state, which depends on the source processor, will be a one-hot code. This structure is chosen because it is known that a solution will exist to map the SLRNN to the desired ASSM (Goudreau et al., 1994). The solution that is known to exist requires the use of one-hot codes for the states and the inputs. The representation that the SLRNN actually learns, however, can have states that are not in a one-hot code. The SLRNN might construct a solution that is different from the one-hot solution.

Clearly, there must be enough neurons to represent the processors (outputs) with a one-hot code. Therefore, the number of neurons must at least be equal to the number of processors. For the one-hot solution to exist, however, there must be one neuron for each switch as well. Unfortunately, one does not generally know the number of switches beforehand. It is necessary to estimate the number of switches, and provide at least that many neurons. This is one of the weaknesses that is common to many neural network approaches: often it is not clear what size neural network would be best. One approach is to start with as many switch neurons as reasonably possible; if training is successful, then reduce the number of neurons using a destructive heuristic (Omlin & Giles, 1993).

The SLRNN will work in the following manner. The input to the SLRNN will be an initial state vector and a series of input vectors. The initial state vector and the first input vector will be applied to the SLRNN at time step one, the second input vector will be applied at time step two, and so on. Note that the last input vector will always correspond to the end-symbol. The outputs of the neurons are not examined until the entire input has been applied. After the end-symbol is entered, gradient descent techniques can be used to train the SLRNN so that the actual output gets closer to the desired output. There are many techniques that can be used to help the SLRNN correctly handle all of the training data.

The state vectors that are created through the use of the gradient descent techniques are not examined during training, but they are examined *after* training to describe the structure of the underlying SRIN. As mentioned before, these states actually represent switches in the SRIN. Since the neurons have a continuous activation function, it is generally necessary to use some partitioning and clustering techniques to make a group of states equivalent. After this is done, an ASSM can be extracted from the SLRNN. The extracted ASSM might not be minimal, but in this case standard SSM minimization techniques can be used.

5.2 A Training Example with Analysis

Perhaps the easiest way to describe the mapping to the SLRNN would be to describe the structure of the SLRNN given data for the SRIN from Figure 4. We will choose to have $N = 8$ neurons in the SLRNN. Since one-hot codes will be used, four initial states vectors are necessary, corresponding to the four source processors that are present in the data table. In reality, however, it should be remembered that the state vectors in the SLRNN will actually correspond to switches from the SRIN, rather than processors. Thus, when we choose state vectors for the four source processors, we are really choosing state vectors for their respective designated switches (as shown in Equation 3). Let $\mathbf{h}_{n,m}$ be a vector of dimension n that has a value "1" in position m and a value "0" in all of the other positions. For example, $\mathbf{h}_{5,2} = [0, 1, 0, 0, 0]^T$. When the source processor is processor p_0 (which is equivalent to saying that the designated switch is q_0, see Equation 3) the initial state vector will be $\mathbf{h}_{8,1}$. Similarly, source processor p_1 will make the initial state vector $\mathbf{h}_{8,3}$, source processor p_2 will make the initial state vector $\mathbf{h}_{8,4}$, and source processor p_3 will make the initial state vector $\mathbf{h}_{8,5}$. This mapping of source processors to initial state vectors can be made arbitrarily. There is no reason not to use vector $\mathbf{h}_{8,2}$, for example. The only reason that this vector was not used was because of the way the training data was generated.

Now the input symbols, r_0 and r_1, as well as the end-symbol, ϵ, can be mapped to input vectors. There are three symbols that must be mapped to input vectors, so we will have three vectors of length three for this purpose. Let r_0 correspond to input vector $\mathbf{h}_{3,1}$, r_1 correspond to input vector $\mathbf{h}_{3,2}$, and ϵ correspond to input vector $\mathbf{h}_{3,3}$.

Finally, the processors can now be mapped to output vectors. There are eight neurons in the SLRNN, but only the outputs of four of them are needed for the output vectors. Therefore, the output values will only be taken from neurons 0, 1, 2, and 3. Destination processor p_0 will correspond to output vector $\mathbf{h}_{4,1}$, destination processor p_1 will correspond to output vector $\mathbf{h}_{4,2}$, destination processor p_2 will correspond to output vector $\mathbf{h}_{4,3}$, and destination processor p_3 will correspond to output vector $\mathbf{h}_{4,4}$.

Now that the specifics of the encoding have been explained, we can use the training data to train the SLRNN. The data table contained an entry for every header up to length 11 (including the end-symbol) for every processor. The data table starts with strings of length one and concludes with strings of length 11. The SLRNN does not try to learn all of the data in this table simultaneously. Rather, it first learns the first 20 lines of the data table. Then the resulting SLRNN is checked against the rest of the data table to see how it generalizes. If perfect generalization does not occur, the SLRNN adds 20 more lines to its training data. Once these 40 lines are learned, generalization is checked again. This process is repeated until all of the lines in the data table have been learned. This heuristic approach, which involves incremental expansion of the training data, has proven to be quite successful in practice.

For our experiment, by the time the SLRNN learned the first 280 lines of the table, the SLRNN was able to generalize for all of the remaining strings in the table. The full table had 8188 lines.

5.3 Extraction of the Interconnection Network from the Trained Neural Network

It was mentioned in Section 5.1 that this problem can be thought of as the problem of learning several separate SSMs, in this case four. Given this fact, one can examine the four separate SSMs that are generated from the four different initial states.

Table 2 shows the *unminimized* SSM with initial state corresponding to switch q_0 that was extracted from the SLRNN. This machine shall be called M1. Details on the method of SSM extraction that was used can be found in (Giles et al., 1992; Giles et al., 1992). The left column (S) contains the number of each state. State "1" in Tables 2 to 9 corresponds to the initial state. The next column (O) contains the output value associated with that state. The following two columns contain the next state given input zero (NS_0) and given input one (NS_1). The final column (QR) contains the *quantized representation* for the state in the SLRNN. It should be kept in mind that the SLRNN actually uses real valued state vectors, as does the clustering algorithm that was used for SSM extraction. The SSM extraction algorithm makes use of a real valued *representative* vector for each state. In the SLRNN, whenever a state vector is "close" to a representative state vector, it is assumed that the two vectors implement the same state. To simplify our analysis, the real valued representative state vectors are quantized. Any value greater than or equal to 0.5 in a representative state vector is set to 1 after quantization, while any value less than 0.5 is set to 0. Henceforth, when state vectors are mentioned, we will actually be talking about these quantized representative state vectors. Thus, in general, the state vectors in Table 2 actually represent some *volume* of the state space.

The unminimized SSMs given designated switches q_2, q_3, and q_4 are shown in Tables 3 (machine M2), 4 (machine M3), and 5 (machine M4), respectively.

Once the unminimized SSMs and their state vector representations are known, the

S	O	NS_0	NS_1	QR
1	1	2	3	10000000
2	4	4	5	01111100
3	2	4	6	00100001
4	1	7	7	11000011
5	4	8	9	11010100
6	4	4	5	01011100
7	3	10	6	00010101
8	4	11	9	11001101
9	2	4	2	00100101
10	1	6	3	10000001
11	4	11	9	11011101

Table 2: Machine M1, the unminimized SSM with initial state corresponding to switch q_0. Column S contains the state. Column O contains the output. Column NS_0 contains the next state given input 0. Column NS_1 contains the next state given input 1. Column QR contains the quantized representation for the state in the SLRNN.

S	O	NS_0	NS_1	QR
1	2	2	3	00100000
2	1	4	4	11000011
3	4	2	5	01011110
4	3	6	7	00010101
5	4	8	9	11010100
6	1	7	10	10000101
7	4	2	5	01011100
8	4	11	12	11001101
9	2	2	7	01100101
10	2	2	7	00100001
11	4	11	12	11011101
12	2	2	13	00100101
13	4	2	5	01111100

Table 3: Machine M2, the unminimized SSM with initial state corresponding to switch q_2.

S	O	NS_0	NS_1	QR
1	3	2	3	00010000
2	1	3	4	10000101
3	4	5	6	01011100
4	2	5	3	00100001
5	1	7	7	11000011
6	4	8	9	11010100
7	3	2	10	00010101
8	4	11	12	11001101
9	2	5	3	01100101
10	4	5	6	01010100
11	4	8	9	11011101
12	2	5	13	00100101
13	4	5	6	01111100

Table 4: Machine M3, the unminimized SSM with initial state corresponding to switch q_3.

S	O	NS_0	NS_1	QR
1	4	2	3	00001000
2	1	4	4	11000011
3	4	5	6	11000101
4	3	7	8	00010101
5	4	9	10	11001101
6	2	2	11	01100101
7	1	11	12	10000101
8	4	2	13	01010100
9	4	9	10	11011101
10	2	2	14	00100101
11	4	2	13	01011100
12	2	2	11	00100001
13	4	5	6	11010100
14	4	2	13	01111100

Table 5: Machine M4, the unminimized SSM with initial state corresponding to switch q_4.

S	O	NS_0	NS_1	QR
1	1	2	3	10000000 10000001
2	4	4	5	01011100 01111100
3	2	4	2	00100001 00100101
4	1	6	6	11000011
5	4	5	3	11001101 11010100 11011101
6	3	1	2	00010101

Table 6: The minimal SSM with initial state corresponding to switch q_0. This SSM is equivalent to machine M1 in Table 2.

SSMs can be minimized and merged. Table 6 contains the *minimized* SSM that is extracted from the SLRNN when the initial state vector corresponds to designated switch q_0. Each state in the minimized machine can be associated with one or more state vectors. For example, state 1 of the SSM in Table 6 is associated with two state vectors, 10000000 and 10000001.

The minimal SSMs with initial states corresponding to designated switches q_2, q_3, and q_4 are shown in Tables 7, 8, and 9, respectively. With the information in Tables 6 to 9, we can merge the SSMs into an ASSM. States in the SSMs with any overlapping vector representations are assumed to be equivalent. The results show that the SLRNN has indeed “merged” the SSMs.

In fact, if we ignore the quantized representations of the states, the four minimal SSMs are equivalent except for the fact that they have different initial states. That is, if we ignore the initial states, the four SSMs can be relabeled so that they are identical.

It should be clear how Table 6 corresponds to the SRIN in Figure 4. State 1 corresponds to switch q_0, state 2 corresponds to switch q_4, state 3 corresponds to switch q_2, state 4 corresponds to switch q_1, state 5 corresponds to switch q_5, and state 6 corresponds to switch q_3. Simple observation will show that the other three SSMs also correspond to the SRIN in Figure 4.

Table 10 merges this information to show how the SLRNN represents the switches of Figure 4. State vectors are shown and the machines that utilized them are presented.

Examination of Table 10 shows that, for the most part, machines M1, M2, M3, and M4 make use of state vectors that are approximately equal. For example, all four

S	O	NS_0	NS_1	QR
1	2	2	3	00100000 00100001 00100101 01100101
2	1	4	4	11000011
3	4	2	5	01011100 01011110 01111110
4	3	6	3	00010101
5	4	5	1	11001101 11010100 11011101
6	1	3	1	10000101

Table 7: The minimal SSM with initial state corresponding to switch q_2. This SSM is equivalent to machine M2 in Table 3.

S	O	NS_0	NS_1	QR
1	3	2	3	00010000 00010101
2	1	3	4	10000101
3	4	5	6	01010100 01011100 01111100
4	2	5	3	00100001 00100101 01100101
5	1	1	1	10000101
6	4	6	4	11001101 11010100 11011101

Table 8: The minimal SSM with initial state corresponding to switch q_3. This SSM is equivalent to machine M3 in Table 4.

S	O	NS_0	NS_1	QR
1	4	2	3	00001000
				01010100
				01011100
				01111100
2	1	4	4	11000011
3	4	3	5	11000101
				11001101
				11010100
				11011101
4	3	6	1	00010101
5	2	2	1	00100001
				00100101
				01100101
6	1	1	5	10000101

Table 9: The minimal SSM with initial state corresponding to switch q_4. This SSM is equivalent to machine M4 in Table 5.

switch	QR	machine
q_0	10000000	M1
	10000001	M1
	10000101	M2,M3,M4
q_1	11000011	M1,M2,M3,M4
q_2	00100000	M2
	00100001	M1,M2,M3,M4
	00100101	M1,M2,M3,M4
	01100101	M2,M3,M4
q_3	00010000	M3
	00010101	M1,M2,M3,M4
q_4	00001000	M4
	01010100	M3,M4
	01011100	M1,M2,M3,M4
	01011110	M2
	01111100	M1,M2,M3,M4
q_5	11000101	M4
	11001101	M1,M2,M3,M4
	11010100	M1,M2,M3,M4
	11011101	M1,M2,M3,M4

Table 10. How the SLRNN represents the switches from the SRIN in Figure 4.

machines use the state region 00010101 to represent switch q_3 from Figure 4.

Another interesting fact is that the state vectors that represent equivalent states tend to be close to each other in the state space. That is, the state vectors for equivalent states tend to have small Hamming distances from one another. For example, machine M1 uses two state regions to represent switch q_4, 01011100 and 01111100. This fact leads one to believe that the SLRNN actually uses something like a cohesive sub-space for each state. It seems likely that the unminimized SSMs that were extracted have equivalent states due to the nature of the SSM extraction algorithm that is used. If other clustering approaches were used (Das & Mozer, 1994; Watrous & Kuhn, 1992; Zeng et al., 1993), it is possible that minimal SSMs could have been extracted directly from the SLRNN. Furthermore, values such as 0.49 and 0.51 will have different values after quantization, although they are in fact quite close in the state space. Thus, two state vectors that are quite close in terms of Euclidean distance (before quantization) might be quite far from each other after quantization. This is another possible explanation for the fact that the extracted SSM was not minimal.

Another observation is that the SLRNN did not learn anything approaching the one-hot solution that was described in Section 5. In fact, the only states vectors that were one-hot after quantization were the initial state vectors that were forced upon the SLRNN. However, while these initial state vectors were not returned to, they did seem to give the SLRNN a bias on its state representation.

Finally, the fact that the correct SRIN was extracted means that good generalization for strings longer than those in the training data has obviously been achieved.

6 Conclusions

A radical approach to the construction of interconnection networks has been presented. This approach uses training data from an unknown interconnection network to teach a RNN to generate an interconnection network that is capable of routing the training data.

It was shown that this problem maps directly to the problem of learning a SSM with several distinct initial states. The proposed approach took advantage of previous work on the use of RNNs to inference SSMs. However, it should be noted that the relationship between interconnection networks and SSMs might also allow for some non-neural network approaches to be used for the same problem. It seems likely that such methods could be varied slightly to accommodate the interconnection network inference problem, just as the RNN method for SSM inference can be varied slightly to perform interconnection network inference.

It was demonstrated that given a table of training data, it is possible to use a RNN to generate the structure of an interconnection network that is capable of routing the training data. Furthermore, the interconnection network that is generated might be able to generalize for inputs that are not in the training data. A sample problem was used to illustrate the methodology.

This work clearly pointed out the need for further research into the use of RNNs to inference larger SSMs. To date, RNNs have had limited success for large problems in grammatical inference, but some recent results are promising (Giles & Horne, 1994).

The concept of learning interconnection networks is an unusual one for the interconnection network community. It remains to be seen whether such learning approaches will become a useful method for interconnection network design.

7 Acknowledgement

The authors would like to acknowledge useful discussions with Sanjeev Kulkarni and Cliff B. Miller.

References

Angluin, D. (1978). On the complexity of minimum inference of regular sets. *Information and Control*, *39*, 337–350.

Brown, T. X. (1989). Neural networks for switching. *IEEE Communications Magazine*, *27*(11), 72–81.

Brown, T. X. & Liu, K.-H. (1990). Neural network design of a Banyan network controller. *IEEE Journal on Selected Areas of Communication*, *8*(8), 1428–1438.

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, *1*, 372–381.

Das, S. & Mozer, M. (1994). A hybrid clustering/gradient descent architecture for finite state machine induction. In Cowan, J., Tesauro, G., & Alspector, J. (Eds.), *Advances in Neural Information Processing Systems 6*. San Mateo, CA: Morgan Kaufmann.

Funabiki, N., Takefuji, Y., & Lee, K. C. (1991). A neural network model for traffic controls in multistage interconnection networks. In *Proceedings of the International Joint Conference on Neural Networks 1991*, (pp. A898).

Funabiki, N., Takefuji, Y., & Lee, K. C. (1993). Comparisons of seven neural network models on traffic control problems in multistage interconnection networks. *IEEE Transactions on Computers*, *42*(4), 497–501.

Giles, C. L. & Horne, B. G. (1994). Some large DFA's are learnable with recurrent neural networks. NEC Research Institute, Inc., Princeton, NJ.

Giles, C. L., Miller, C. B., Chen, D., Chen, H. H., Sun, G. Z., & Lee, Y. C. (1992). Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, *4*(3), 393–405.

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 Moody, J., Hanson, S., & Lippmann, R. (Eds.), *Advances in Neural Information Processing Systems 4*, (pp. 317–324)., San Mateo, CA. Morgan Kaufmann Publishers.

Giles, C. L. & Omlin, C. W. (1993). Extraction, insertion and refinement of symbolic rules in dynamically-driven recurrent neural networks. *Connection Science*, *5*(3,4), 307–337. Special Issue on Architectures for Integrating Symbolic and Neural Processes.

Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control*, *37*, 302–320.

Goudreau, M. W. (1993). *Neural Network Applications for Interconnection Networks*. PhD thesis, Princeton University, Princeton, NJ.

Goudreau, M. W. & Giles, C. L. (1992). Routing in random multistage interconnection networks: Comparing exhaustive search, greedy and neural network approaches. *International Journal of Neural Systems*, *3*(2), 125–142.

Goudreau, M. W. & Giles, C. L. (1993). Discovering the structure of a self-routing interconnection network with a recurrent neural network. In Alspector, J., Goodman, R., & Brown, T. X. (Eds.), *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*, (pp. 52–59)., Hillsdale, NJ. Lawrence Erlbaum Associates, Inc.

Goudreau, M. W., Giles, C. L., Chakradhar, S. T., & Chen, D. (1994). First-order vs. second-order single layer recurrent neural networks. To be published in *IEEE Transactions on Neural Networks*.

Hakim, N. Z. & Meadows, H. E. (1990). A neural network approach to the setup of the Benes switch. In *Infocom 90*, (pp. 397–402).

Hillis, W. D. (1990). Co-evolving parasites improve simulated evolution as an optimization procedure. *Physics D*, *42*, 228–234.

Hopcroft, J. E. & Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley Publishing Company, Inc.

Kearns, M. & Valiant, L. (1989). Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*. ACM Press.

Kohavi, Z. (1978). *Switching and Finite Automata Theory* (second ed.). New York, NY: McGraw-Hill, Inc.

Lang, K. (1992). Random DFA's can be approximately learned from sparse uniform examples. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, (pp. 45–52). ACM Press.

Lee, S.-L. & Chang, S. (1993). Neural networks for routing of communication networks with unreliable components. *IEEE Transactions on Neural Networks*, *4*(5), 854–863.

Marrakchi, A. M. & Troudet, T. (1989). A neural net arbitrator for large crossbar packet-switches. *IEEE Transactions on Circuits and Systems*, *36*(7), 1039–1041.

Melsa, P. J. W., Kenney, J. B., & Rohrs, C. E. (1990a). A neural network solution for call routing with preferential call placement. In *Proceedings of the 1990 Global Telecommunications Conference*, (pp. 1377–1382).

Melsa, P. J. W., Kenney, J. B., & Rohrs, C. E. (1990b). A neural network solution for routing in three stage interconnection networks. In *Proceedings of the 1990 International Symposium on Circuits and Systems*, (pp. 483–486).

Mozer, M. C. & Bachrach, J. (1990). Discovering the structure of a reactive environment by exploration. *Neural Computation*, *2*(4), 447–457.

Mozer, M. C. & Bachrach, J. (1991). SLUG: A connectionist architecture for inferring the structure of finite-state environments. *Machine Learning*, *7*(2/3), 139–160.

Omlin, C. W. & Giles, C. L. (1993). Pruning recurrent neural networks for improved generalization performance. Technical Report TR 93-6, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY.

Pitt, L. & Warmuth, M. (1989). The minimum DFA consistency problem cannot be approximated within any polynomial. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*. ACM Press.

Pollack, J. B. (1991). The induction of dynamical recognizers. *Machine Learning*, *7*(2/3), 227–252.

Rivest, R. L. & Schapire, R. E. (1987a). Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, (pp. 78–87).

Rivest, R. L. & Schapire, R. E. (1987b). A new approach to unsupervised learning in deterministic environments. In Langley, P. (Ed.), *Proceedings of the Fourth International Workshop on Machine Learning.*

Schapire, R. E. (1988). Diversity-based inference of finite automata. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

Siegel, H. J. (1990). *Interconnection Networks for Large Scale Parallel Processing.* New York: McGraw-Hill.

Takefuji, Y. & Lee, K. C. (1991). An artificial hysteresis binary neuron: A model suppressing the oscillatory behavior of neural dynamics. *Biological Cybernetics*, *64*, 353–356.

Thomopoulos, S. C. A., Zhang, L., & Wann, C. D. (1991). Neural network implementation of the shortest path algorithm for traffic routing in communication networks. In *Proceedings of the International Joint Conference on Neural Networks 1991*, (pp. 2693–2702)., Singapore.

Trakhtenbrot, B. & Barzdin, Y. (1973). *Finite Automata: Behavior and Synthesis.* Amsterdam: North-Holland Publishing Company.

Troudet, T. P. & Walters, S. M. (1991). Neural network architecture for crossbar switch control. *IEEE Transactions on Circuits and Systems*, *38*(1), 42–56.

Watrous, R. L. & Kuhn, G. M. (1992). Induction of finite-state languages using second-order recurrent networks. *Neural Computation*, *4*(3), 406–414.

Williams, R. J. & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, *1*(2), 270–280.

Zeng, Z., Goodman, R. M., & Smyth, P. (1993). Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, *5*(6), 976–990.