Learning a Class of Large Finite State Machines with a Recurrent Neural Network

C. Lee Giles* B. G. Horne T. Lin†

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
{giles,horne,lin}@research.nj.nec.com

* Also with
UMIACS
University of Maryland
College Park, MD 20742

† Also with
EE Department
Princeton University
Princeton, NJ 08540

August, 1994

Technical Report
UMIACS-TR-94-94 and CS-TR-3328
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Learning a Class of Large Finite State Machines with a Recurrent Neural Network

C. Lee Giles* B. G. Horne T. Lin†

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
{giles,horne,lin}@research.nj.nec.com

* Also with
UMIACS
University of Maryland
College Park, MD 20742

† Also with
EE Department
Princeton University
Princeton, NJ 08540

August, 1994

Abstract

One of the issues in any learning model is how it scales with problem size. Neural networks have not been immune to scaling issues. We show that a dynamically-driven discrete-time recurrent network (DRNN) can learn rather large grammatical inference problems when the strings of a finite memory machine (FMM) are encoded as temporal sequences. FMMs are a subclass of finite state machines which have a finite memory or a finite order of inputs and outputs. The DRNN that learns the FMM is a neural network that maps directly from the sequential machine implementation of the FMM. It has feedback only from the output and not from any hidden units; an example is the recurrent network of Narendra and Parthasarathy. (FMMs that have zero order in the feedback of outputs are called definite memory machines and are analogous to Time-delay or Finite Impulse Response neural networks.) Due to their topology these DRNNs are as least as powerful as any sequential machine implementation of a FMM and should be capable of representing any FMM. We choose to learn "particular FMMs." Specifically, these FMMs have a large number of states (simulations are for 256 and 512 state FMMs) but have minimal order, relatively small depth and little logic when the FMM is implemented as a sequential machine. Simulations for the number of training examples versus generalization performance and FMM extraction size show that the number of training samples necessary for perfect generalization is less than that sufficient to completely characterize the FMM to be learned. This is in a sense a best case learning problem since any arbitrarily chosen FMM with a minimal number of states would have much more order and string depth and most likely require more logic in its sequential machine implementation.

1 Introduction

1.1 Background

Dynamically–driven recurrent neural networks (DRNNs) have empirically shown the ability to perform inference in problems as diverse as grammar induction (Cleeremans et al., 1989; Das and Das, 1991; Elman, 1991; Frasconi et al., 1992a; Giles et al., 1992; Mozer and Bachrach, 1990; Pollack, 1991; Zeng et al., 1994) and system identification in control (Barto, 1990; Billings et al., 1992; Narendra and Parthasarathy, 1990). We discuss results concerning the learning of temporal sequences for a particular class of discrete–time recurrent neural network architectures (Narendra and Parthasarathy, 1990). This DRNN has tapped delays both on the input and on the feedback of the output. Because of this model's similarity to an IIR filter, we will refer to it as a neural network IIR (NNIIR). Such models are very similar to feedback networks described by others (Back and Tsoi, 1991; Billings et al., 1992; Frasconi et al., 1992b; Jordan, 1986; Sastry et al., 1994; Vries and Principe, 1992).

We show that this DRNN when trained on strings encoded as temporal sequences is able to learn and emulate a large finite state machine (FSM) and its associated grammar. The finite state machines we easily learned have the following distinct properties: they are from a subclass of FSMs called *finite memory machines* (Kohavi, 1978) that are defined by the type of memory used and how fed back, they have relatively low depth, and when implemented as a sequential machine they require minimal memory and simple combinational logic.

1.2 Benchmark Problems for Recurrent Neural Networks

Though there are many benchmark databases for feedforward networks, few exist for dynamic networks. We propose a specific problem of system identification as one good benchmark for the computational capabilities of dynamically–driven recurrent networks. As a benchmark for dynamic networks, the training data must have dynamical characteristics. Temporal signals can have many characteristics: discrete or continuous; real, complex or binary valued; dimensionality; stochastic or deterministic; one or many samples; labeled or unlabeled. If grammatical strings from deterministic regular grammars are interpreted as temporal sequences, then these temporal sequences have the simple set of characteristics described above. However, the problem of learning (or inferring these sequences) can be NP–complete in the worst case. As such we propose grammatical inference with temporal sequences as a good benchmark problem for the computational capabilities of recurrent neural networks, irrespective of possible applications. However, potential applications in natural language processing (Fu, 1994; Sun, 1994) and more recently in intelligent control (Nerode and Kohn, 1993a; Nerode and Kohn, 1993b) have been proposed.

2 Properties of Finite State and Memory Machines

Since we are learning finite state machines from temporal sequences, we briefly introduce finite state machines (FSMs) and their properties. A FSM is an abstraction of a device that can be described by a labeled directed cyclic graph that consists of inputs, states and outputs. In this paper all FSMs are deterministic. A sequential machine (SM) refers specifically to the logical implementation of that machine, consisting of logic and fed back memory functions, for example delay lines, latches, flip–flops, etc. All SMs described in the paper are synchronous. Another important difference between sequential machines and FSM is that because of feedback, time is an explicit parameter for sequential machines. For machine, time is just one of many possible parameterizations of the

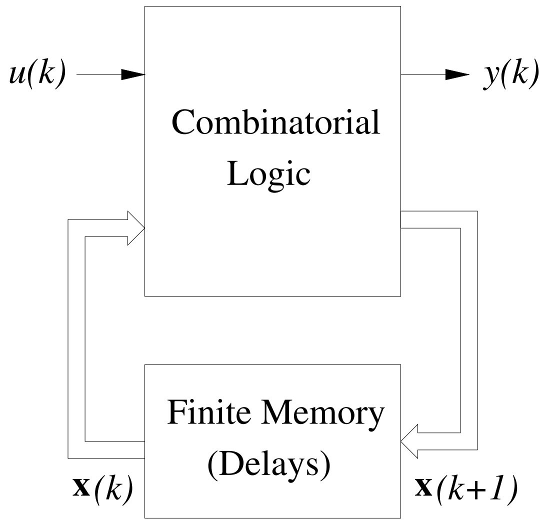

Figure 1: A sequential machine.

state transitions and of input and output sequences. We will see that there a topological similarities between various types of sequential machines and recurrent neural networks.

2.1 Finite State Machines

Finite state machines operate with a finite number of input and output symbols and have a finite number of internal states and an output for each corresponding input. An FSM is defined as:

Definition 1 A *finite state machine* (FSM) is a sextuple $\mathcal{M} = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where Q is a finite set of *states*; Σ is a finite set of symbols called the *input alphabet*; Δ is a finite set of symbols called the *output alphabet*; $\delta : Q \times \Sigma \longrightarrow Q$ is a *transition function*; $\lambda : Q \times \Sigma \longrightarrow \Delta$ is an *output function*; q_0 is the initial state. □

For this work, the output alphabet, like the input alphabet, will always be binary, i.e. $\Delta = \{0, 1\}$. We shall assume that the reader is familiar with the conventional extensions of δ and λ to the free monoid of Σ, if not see citehopcroft79b.

2.2 Sequential Machines

Sequential Machines (SMs) are implementations of an FSM which consist of logic and memory elements. An example of a SM is shown in Figure 1. In high–level VLSI synthesis generating the SM from the high–level FSM design is one of the first steps in logic synthesis (Ashar et al., 1992). A great deal of effort has gone into facilitating and automating this process.

We can explicitly associate time with an FSM in the following way. The input, output and state to the machine at time k will be denoted by respectively $u(k)$, $y(k)$ and $x(k)$. The encoding of the input and output alphabets into $u(k)$ and $y(k)$ must be defined. When these variables are related by logic and time delays or memory, this implementation is called a sequential machine (see Figure 1). Note that if the combinational logic in Figure 1 is replaced by a feedforward neural network, the sequential machine becomes a general-purpose recurrent neural network.

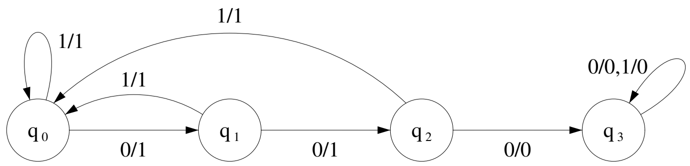

Figure 2: A finite memory machine of input–order 2 and output–order 1.

2.3 Useful properties of FSMs

A finite state machine is minimal if it is the machine with the fewest number of states for a given input/output behavior. The FSMs described here are all minimal. Two useful measures of characterizing a FSM are its *depth* and its *degree of distinguishability.* States q_i and q_j of a FSM are said to be *distinguishable* if there exists a finite length sequence which when the machine is started in state q_i produces a different output sequence than the output sequence produced when the machine is started in state q_j. The degree of distinguishability is the smallest integer ρ such that for every pair of non–equivalent states in the FSM there exists an input sequence not longer than ρ that induces a different output sequence when the machine is started in each of the two states. (Two states, p and q, of a FSM are nonequivalent if there exists a string wa, called a distinguishing string, such that $\lambda(\delta(p,w),a) \neq \lambda(\delta(q,w),a)$.) The depth is the smallest integer d such that every state in the FSM can be reached from the starting state in no more than d steps.

2.4 Finite Memory Machines

We will be interested in a subclass of FSMs known as finite memory machines (FMMs).

Definition 2 A finite state machine $\mathcal{M}$ is said to be a *finite memory machine of input–order n and output–order m* if n and m are the least integers, such that the present state of $\mathcal{M}$ can always be determined uniquely from the knowledge of the last n inputs and the last m outputs, for all possible sequences of length $\max(n,m)$. □

Note that the definition excludes the possibility of any knowledge of the initial state of the machine. For example, the FSM shown in Figure 2, has input–order two and output–order one, since for any input sequence of length two, the state of the FSMs can always be determined from knowledge of the past two inputs and the last output as illustrated in Table 1. Not all FSMs have finite memory, some have infinite order. For example, the Dual Parity FSM, shown in Figure 3, has infinite order since one can observe an infinite sequence of ones at the input and an infinite sequence of zeros at the output without being able to determine whether the FSM is in state q_2 or q_3 (unless one has knowledge of the initial state of the machine).

Given an arbitrary FSM there exist efficient algorithms to determine if the machine has finite memory and, if so, its corresponding order (Kohavi, 1978). For more properties of FMMs, please see the Appendix.

Since the state of an FMM depends only on a finite number of previous inputs and outputs, the sequential machine implementation of an FMM can always be implemented by tapped delay lines (TDLs) on the input and output and a block of combinational logic as shown in Figure 4. Again, if the combinational logic is replaced by a feedforward neural network, the sequential machine

y(k-1)	u(k-2)	u(k-1)	state
0	0	0	q_3
0	0	1	q_3
0	1	0	q_3
0	1	1	q_3
1	0	0	q_2
1	0	1	q_0
1	1	0	q_1
1	1	1	q_0

Table 1: The state of the machine as a function of the previous two inputs and previous output.

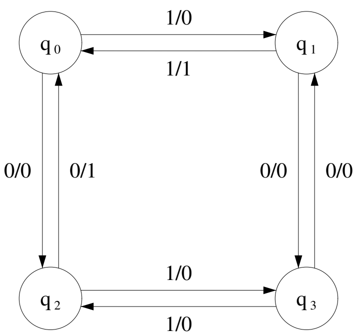

Figure 3: The Dual Parity FSM.

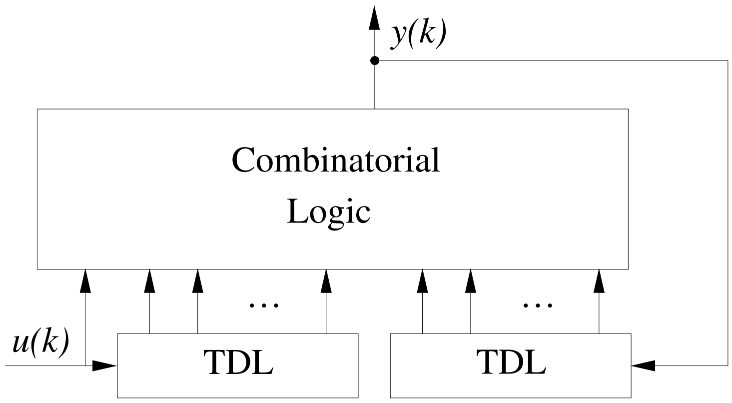

Figure 4: Sequential machine implementation of a finite memory machine. TDL refers to a tapped delay line.

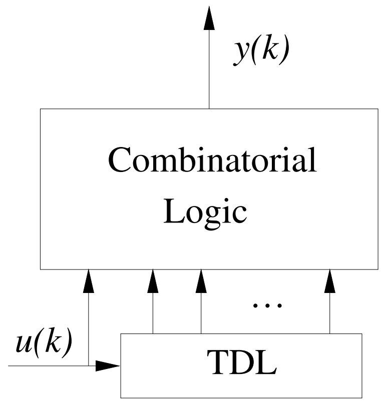

Figure 5: Sequential machine implementation of a definite memory machine.

implementation of a FMM becomes a recurrent network similar to those used in control (Narendra and Parthasarathy, 1990) and time series (Connor et al., 1994).

FMMs of input–order n and output–order 0 are said to be *definite memory machines*. Implementations of such machines do not require feedback from the combinational logic as shown in Figure 5. Definite memory machines are analogous to recurrent neural networks that have no feedback from the output states but still time delays on the inputs. These neural networks have been called TDNNs (Lang, 1992) and finite impulse response neural nets (Wan, 1994).

2.5 Constructing FMMs of Minimal Order and Small Logic

Finding example FMMs with a large number of states is nontrivial. One could potentially pick the tap size and logic function of a SM implementation randomly. However, the resulting FMM more often than not has an smaller order than the choice of taps and an unpredictable number of states. Indeed, using this approach the resulting FMM is often a trivial machine with only a few states. Instead, we developed theory to devise a method for constructing machines to use for example learning problems. This theory allows us to construct FMMs through a method we call the *Group Linking Method* (GLM), a method which permits a certain amount of control over a number of properties of the FMM including the order, number of states, and the complexity of the logic function which defines the mapping from previous inputs and outputs to the current output. See the Appendix for a complete discussion of the GLM.

2.6 Grammatical Inference and FMMs

Grammatical inference (Fu and Booth, 1975) is the problem of finding a FSM consistent with a set of positive and negative strings. (These results are often given for deterministic finite-state automata (DFA). However, it is straightforward to map a DFA into a FSM.) Grammatical inference is known to be NP–complete (Angluin, 1978) in the worst case. However, some approaches have been suggested which seem to work well on relatively large problems.

First, if there is a sufficient amount of data it is always possible to construct the smallest corresponding FSM in polynomial time (Trakhenbrot and Barzdin, 1973). Specifically, given the complete set of strings not longer than $d + \rho + 1$, where d is the depth of the machine and ρ its degree of distinguishability, it is always possible to find the minimum consistent FSM. The input to the algorithm is a tree–structured FSM that directly embodies the training set. The tree can then be collapsed into a smaller graph by merging all pairs of states that represent compatible mappings from string suffices to labels.

If the FSM is known to have finite memory, then it is possible to construct the corresponding FMM from a much smaller set of strings. We prove in the appendix that it is possible to identify a minimal order FMM of depth d from the complete set of strings not longer than $d + 1$.

3 Recurrent Neural Networks

In the past few years several recurrent neural network (RNN) models have been proposed (Back and Tsoi, 1991; Billings et al., 1992; Elman, 1990; Frasconi et al., 1992b; Giles et al., 1990; Hopfield, 1982; Jordan, 1986; Leighton and Conrath, 1991; Narendra and Parthasarathy, 1990; Nerrand et al., 1993; Poddar and Unnikrishnan, 1991; Robinson and Fallside, 1988; Vries and Principe, 1992; Watrous and Kuhn, 1992a; Williams and Zipser, 1989) Here we use a class of networks in which output is computed as a nonlinear function of a window of past inputs and outputs (Narendra and Parthasarathy, 1990), i.e.

$$y(t) = f\left(u(t), u(t-1), \ldots, u(t-n), y(t-1), y(t-2), \ldots, y(t-m)\right)$$

where n and m are the size of the input and output windows respectively. Note that *the activations of hidden neurons are not fed back, the only recurrent connections are from the output(s) of the network.* Because of the similarity to infinite impulse response filters (IIRs), we (as well as others) will refer to these recurrent network models as neural network IIRs (NNIIRs). Many variations of this model have been proposed by Narendra and Parthasarathy (Narendra and Parthasarathy, 1990), and have been used extensively for system identification and control problems. In the most general model, the function $f(\cdot)$ is implemented as a multilayer perceptron. One can also interpret the NNIIR model as a special case of the recurrent net proposed by (Jordan, 1986).

This class of networks also includes the Time Delay Neural Networks (TDNN), which are simply a tapped delay line followed by some kind of multilayer perceptron (Lang et al., 1990; Lapedes and Farber, 1987; Waibel et al., 1989). Strictly speaking, this network is not a RNN, since no nodes are fed back, i.e. the network implements a functions of the form

$$y(t) = f\left(u(t), u(t-1), \ldots, u(t-n)\right).$$

However, the tapped delay line does provide a simple form of dynamics that gives the network the ability model a limited class of nonlinear dynamic systems.

Since multilayer networks are capable of implementing arbitrary logic functions, it follows that these models are capable of implementing arbitrary FMMs using the implementation shown in Figure 4. Similarly, networks like the TDNN are capable of implementing arbitrary definite machines when the combinatorial logic in Figure 5 is replaced with a multilayer feedforward network. It should be obvious that neural networks that feedback hidden neurons have full FSM representational capabilities and are also capable of representing FMMs, a subclass of FSMs in general.

4 Learning Finite Memory Machines

4.1 A Large FMM with Little Logic

We have successfully been able to learn various FMMs with minimal order using the NNIIR models. Because of the minimal order, it is possible to learn very large machines. We also make the further

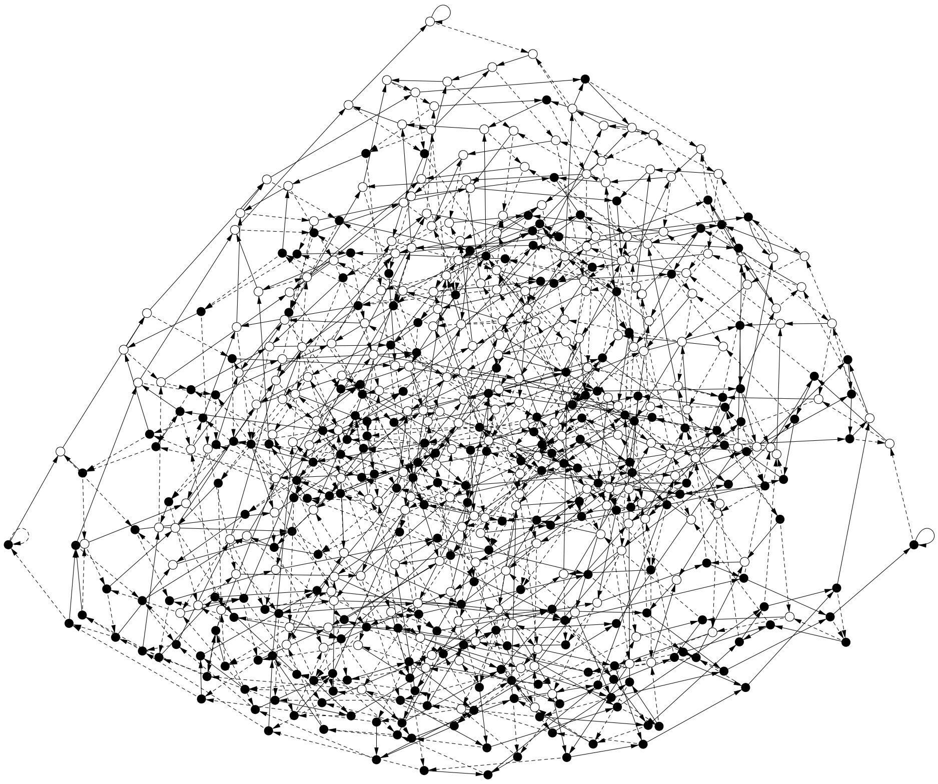

Figure 6: A 512 state finite memory machine of minimal order.

restriction that the specific FSM to be learned has a simple logic function.

In this paper we present results for learning two FMMs. The first machine has 512 states and corresponds to the following logic function,

$$y(k) = \bar{u}(k-5)\bar{u}(k) + \bar{u}(k-5)y(k-4) + u(k)u(k-5)\bar{y}(k-4) \tag{1}$$

where $\bar{x}$ represents the complement of x. The FSM is shown in Figure 6. It has an input–order of 5, an output–order of 4, a depth of 9 and a degree of distinguishability of 6.

The second machine has 256 states and has the more complex, though still learnable, logic function

$$\begin{aligned} y(k) &= \bar{u}(k-1)\left[\bar{u}(k-4)y(k-4)\bar{u}(k) + u(k-4)u(k) + u(k-4)\bar{y}(k-4)\right] \\ &+ u(k-1)\bar{y}(k-1)\left[u(k-4)y(k-4)u(k) + \bar{u}(k-4)\bar{y}(k-4) + \bar{u}(k-4)\bar{u}(k)\right] \\ &+ u(k-1)y(k-1)\left[\bar{u}(k-4)y(k-4)u(k) + u(k-4)\bar{y}(k-4) + u(k-4)\bar{u}(k)\right]. \end{aligned} \tag{2}$$

This machine has an input–order of 4, an output–order of 4, a depth of 9 and a degree of distinguishability of 6.

Positive Strings	Target Values	Negative Strings	Target Values
10	?1	0	0
110	?01	11	?0
0010	0??1	000	0?0
0101	0??1	011	0?0
0110	0?01	0000	0?00
1010	?1?1	0011	0??0
1110	?0?1	1100	?010
		1111	?0?0

Table 2: Example of how to construct intermediate target information from a data set.

4.2 Training and Testing Set

To create a training set, we generated all strings of length 1 to L and labeled them with a 0 or 1 depending on whether the FSM rejected or accepted them. All strings of length $L = d + 1$ are sufficient to identify an FMM according to Theorem 2 (see the Appendix). For both the 256- and 512-state FMMs $d = 9$ and $L = 10$ giving a total of 2046 strings. Thus, this value should be sufficient for any algorithm (neural network or otherwise) which has a representational bias towards FMMs. However, if the algorithm is not biased toward an FMM, then according to (Trakhenbrot and Barzdin, 1973) all strings of length $L = d + \rho + 1 = 16$ may be required for a total of 131,071 strings. From such a data set we randomly selected subsets of strings for training and reserved the remaining samples for testing.

In principle, the neural network is capable of learning machines with a larger depth. However, in order to run the large number of experiments we did in a reasonable amount of time, we have limited ourselves to machines with relatively low depth, and thus to small training and testing sets. It should be noted that the size of these sets would become unmanageably large as the depth of the target machine increases. For example, a machine of depth 20 would have a set of 4,194,302 strings.

It is possible to generate target outputs at intermediate points in each string for a given training set. For example, consider the set of strings shown in Table 2. Since the string "0" is a negative string, then for any string that begins with "0" can be assigned a target output of 0 on the first time step. Similarly, any string that begins with "10" can be assigned a target value of 1 on the second time step. By utilizing all of this information, many intermediate target values can be constructed for each string, although typically not nearly as many as illustrated in the table above. One benefit of intermediate labeling is to give an improved error measure for each string. In addition, teacher forcing (Williams and Zipser, 1989) can be used to force the target value into the feedback loop to improve the speed of convergence, and indeed to enhance the ability of the network to converge at all.

The strings were encoded such that input and output (target) values of 0s and 1s corresponded to floating point values of 0.0 and 1.0. However, many experiments in which we tried different encodings such as -1.0 and 1.0 did not give significantly different results.

4.3 Specific Network Architecture

The NNIIR architecture for both problems had five input taps and four output taps. On the first problem, we used a two layer network with 4 nodes in the hidden layer and one output node, on the second problem we used 15 hidden layer nodes. In both networks each node used the standard sigmoid nonlinearity. The initial values of all delay elements was chosen to be zero. The networks had 49 and 181 adjustable weights respectively with the initial values of the weights randomly chosen from a uniform distribution in the range $[-0.1, 0.1]$.

4.4 Training Algorithm

The network was trained with Backpropagation Through Time Algorithm (Williams and Peng, 1990; Williams and Zipser, 1990), augmented with a number of heuristics found useful for grammatical inference problems. No batching was done on the training set, i.e. the weights were updated after processing each string (although see comment below on selective updating). Weight decay (Krogh and Hertz, 1992) was used with a weight decay parameter of 0.0001.

For sample presentation we used teacher forcing. When target values are available at intermediate points during the processing of a string, these target values are used in the feedback loop instead of the actual node output values. However, this presents several complications. First, teacher forcing effectively replaces feedback with an external input, and therefore gradients can not propagate back through that pathway. Second, when the network is run during the testing phase, it can only feedback the actual node outputs. This can lead to poor performance if the fed back values are not sufficiently close to the teacher forced values. In order to compensate for this effect, we replaced the output node's nonlinearity with a hard limiter during testing. This assures that the network feeds back values that are either 0 or 1. In addition, this effectively converts the feedforward part of the network to a logic function, which can be immediately used to extract an FSM from the final network.

We used a selective updating scheme in which the weights were only updated if the absolute error on the training sample currently being processed was greater than 0.2. This effectively speeds up the learning algorithm by avoiding gradient calculations for weight updates that only add a marginal improvement to the overall performance.

We have also found it useful to encourage the network to learn the shortest strings first by using an incremental training algorithm. In this algorithm the training set is ordered lexicographically, and an epoch is terminated if there are more than thirty samples that have an absolute error greater than 0.2. Thus, the network must learn the shortest strings first in order to train on longer strings. Additionally, we imposed the condition that an initial set of 50 samples must be learned to within an absolute error of 0.2 before the remaining samples are used for training. Once this initial set is learned, an additional fifty samples are added and then these must be learned to within the same error, then another 50 samples are added, and so on.

The learning algorithm was stopped when all examples in the training set yield a absolute error less than 0.2, or if the network exceeded 5000 epochs for the 512–state or 10000 epochs for the 256–state FMM respectively. On the first experiment, the algorithm typically required about 500 epochs to converge. It did not converge in only 9 of the 1500 experiments. On the second experiment, the algorithm required about 2500 epochs and did not converge on 68 of the 1500 experiments.

All of the parameters discussed above were selected by trial and error and our experiences with learning similar problems. For every simulation we used a learning and momentum rate of 0.25. No effort was made to try to optimize any of the parameters described.

4.5 Experimental Results

We ran many experiments to determine the generalization ability and the size of the extracted FSM implemented by the learned network as a function of the size of the training set. Because a NNIIR model is representationally biased towards FMMs, data is randomly selected from a complete data set of length $L = 10$. For learning the 512-state FMM we chose 30 different training set sizes ranging from 10 to 300 samples in increments of 10, while for the 256-state FMM the set sizes ranged from 25 to 750 in increments of 25. For each training set size we ran 50 experiments. In each case a different random sample of strings was chosen, and the weights of the network were initialized differently each time.

The generalization was determined by computing the performance on the samples which were not chosen for training from the 2046 possible samples needed to completely specify the machine. The results are shown in Figures 7 and 9. The average error rate is plotted with an error bar of one standard deviation around the mean.

It is easy to extract the size of the FSM that the network actually learns. By replacing the output node's nonlinearity with a hard limiter, the network effectively implements a logic function since all input and output values are zeros and ones. This logic function defines a FSM for that machine. This FSM can be minimized using a standard FSM minimization algorithm (Hopcroft and Ullman, 1979). The average size of the extracted FSMs are plotted in Figures 8 and 10 with an error bar of one standard deviation around the mean.

4.6 Discussion of Experimental Results

For learning the 512-state FMM, one notices from Figure 7 that as the percentage of training strings increases, the variance in generalization performance decreases and finally approaches zero. Similar behavior is noticed for extraction size in Figure 8 as the extracted FMM approaches the correct size. Note that the number of strings needed for perfect generalization was about 250. This is approximately an order of magnitude less than the 2046 strings necessary to characterize the FMM.

For learning the 256-state FMM we see a similar behavior, although the network is not usually able to achieve perfect generalization. In fact, when the sigmoid is replaced by a hard–limiting threshold function, the network does not even correctly classify the training set most of the time. This implies that the network may actually be utilizing the transition region of the sigmoid in order to solve the problem, and so a more complex extraction algorithm may be needed (see for example (Watrous and Kuhn, 1992b)), although we have not investigated this. Nevertheless, the extracted FMM does get the majority of test samples correct and infers an FMM with size comparable to the target machine.

In the first experiment (the 512-state FMM), the order or number of taps in the recurrent net was exactly equal to the order of the target machine, while in the second experiment (the 256-state FMM) there was a single unnecessary tap in the recurrent net. It would be interesting to explore how the NNIIR's performance changes as the number of input and output taps (or order) is varied.

5 Conclusions

The problem of learning finite state machines (FSMs) from examples with recurrent neural networks has been extensively explored. However, these results are somewhat disappointing in the sense that the machines that can be learned are too small to be competitive with existing grammatical inference algorithms. In this paper we show that large finite state machines can be learned if we

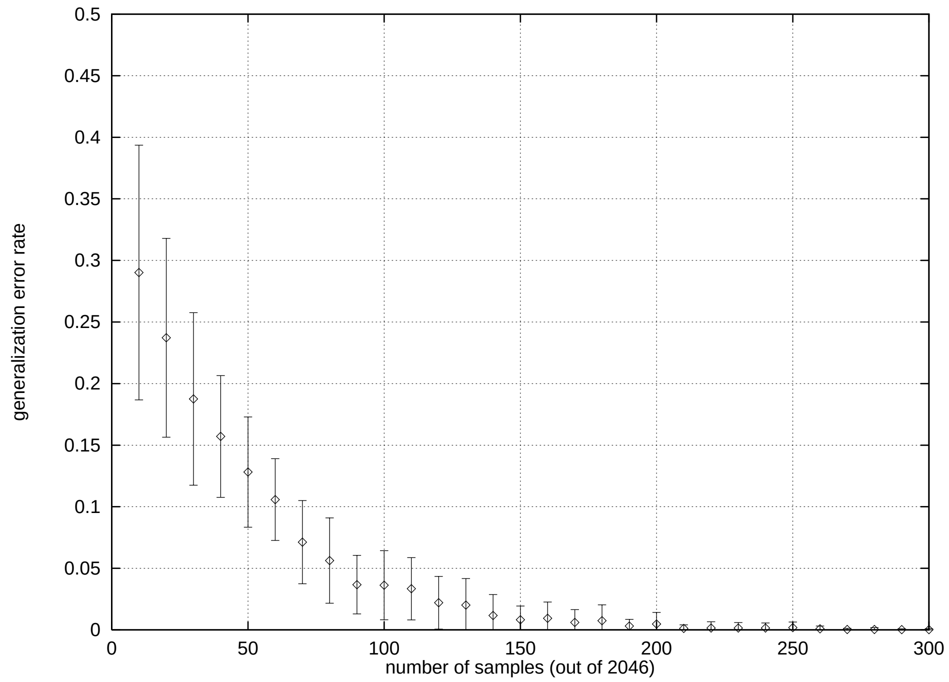

Figure 7: Generalization as a function of training set size on the 512-state FMM.

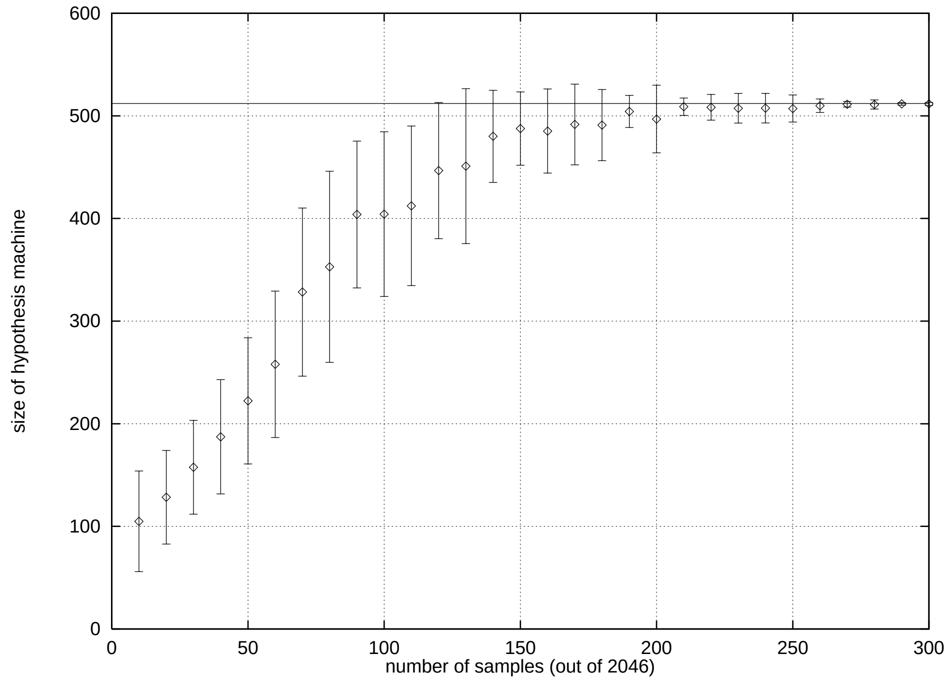

Figure 8: Size of extracted FSM as a function of training set size on 512-state FMM.

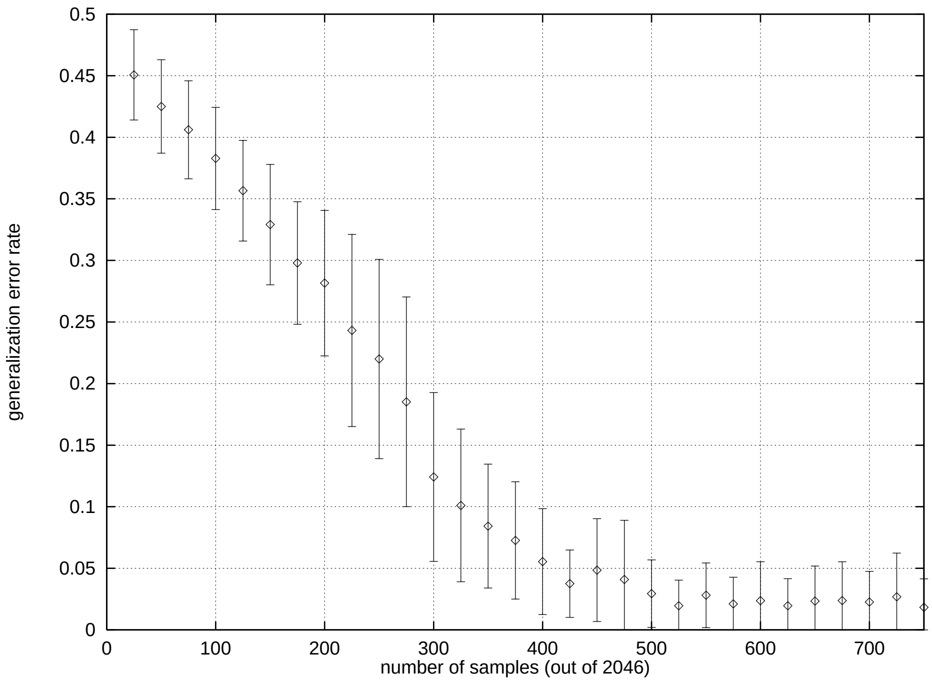

Figure 9: Generalization as a function of training set size on 256-state FMM.

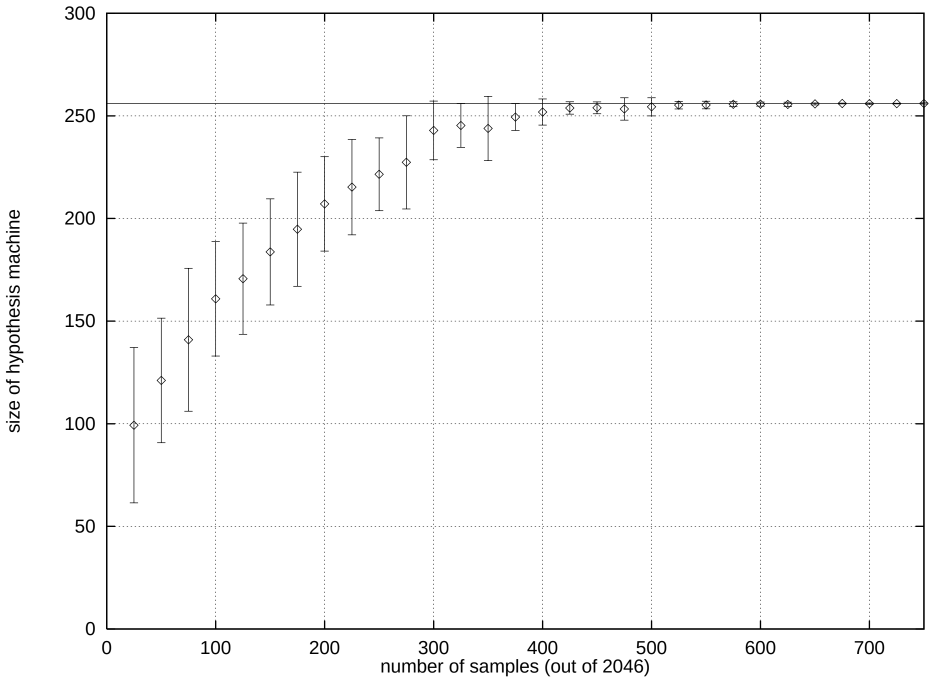

Figure 10: Size of extracted FSM as a function of training set size on 256-state FMM.

limit the class of machines and choose a neural network whose structure is representationally biased towards the problem class to be learned.

The particular DRNN we investigate has tapped delay lines on both the inputs and outputs but no feedback from any hidden states. These values are fed through a Multilayer Perceptron to compute the next output value. For convenience and because of the network's similarity to IIR filters, we refer to this recurrent network as a NNIIR. This network can be interpreted as a sequential machine if the nodes of the neural network are interpreted as threshold logic functions and the delays as memory elements. In fact, this structure corresponds to a specific class of FSMs called finite memory machines (FMMs).

We showed an NNIIR is capable of learning large (up to 512 states) finite memory machines when trained on grammatical strings encoded as temporal sequences. After training on a sufficient sized training set, the correct FMM, or at least one with a very low error rate, could be consistently extracted from the NNIIR. However, certain restrictions were required in order to make the problem practical. These restrictions include limiting the order (which is related to the required tap delay length) and depth (which impacts the size of the training set) of the FSM. Furthermore the sequential machine implementation of the FMM could only have relatively simple logic. As the logic becomes more complex, the task of finding an appropriate set of weights becomes more difficult. We speculate that the task of learning arbitrary logic functions, i.e. the *loading problem* (Blum and Rivest, 1988; Judd, 1990), is the greatest barrier for learning arbitrary FMMs (and FSMs in general). It is important to keep in mind that the restrictions discussed above on what can be learned with a recurrent net define a very small class of all possible FMMs.

It might be possible to identify other types of DRNNs which have a representational bias towards other classes of FSMs. For example, it would be interesting to establish if networks with local recurrence correspond to some other subclass of FSMs, or if they are capable of implementing arbitrary FSMs. The reader should keep in mind that this analogy is somewhat limited since it has been shown that the nonlinearity in simple DRNNs enables them to be computationally very powerful (Siegelmann and Sontag, 1992). It is an open question how the nonlinearity of DRNNs with restricted and local topological connections (Tsoi and Back, 1994) limits their representational power.

Acknowledgements

We would like to acknowledge K. Lang for insightful suggestions. We also acknowledge useful discussions with P. Ashar, S. Chakradhar, L. Leerink and C. Omlin.

Appendix: The Group Linking Method

In this section, we describe a method, called the *Group Linking Method* (GLM), for constructing minimal FMMs with 2^{n+m} states that have the lowest possible order. In addition, we prove that the GLM yields FMMs that are minimal, derive the number of machines in this class, and prove how the set of strings up to length $d+1$, where d is the depth of the FMM, is sufficient to uniquely identify an FMM.

Properties of FMMs

Perhaps the most important consideration is the order of the machine. The maximum input or output order of an FMM is $\frac{|Q|(|Q|-1)}{2}$ (Kohavi, 1978). However, this implies that even if these machines have a small number of states $|Q|$, a large number of tapped delay lines may be required for their implementation. For example, for $|Q| = 20$, the maximum order is 190, which implies that as many as 380 taps may be required to implement the machine! In contrast the minimum order of the machine is $\frac{1}{2}\log|Q|$, since a system with $n+m$ taps and binary inputs and outputs can implement a machine of size at most 2^{n+m}. In such cases, very large machines can be implemented with a very small number of taps. For example, a machine with 1024 states may be implementable with only 10 taps. In order to learn large FMMs, we only consider those machines that have minimal order. In general, such machines will have a relatively low depth compared to the number of states, which in turn implies that a relatively small training set will be sufficient to infer the machine (see the discussion at the end of this appendix). The GLM provides a way to create a wide variety of such machines.

An interesting property of FMMs is that they have a very limited next state function. In unconstrained FSMs, a state can potentially make a transition to any one of the n states within the machine. But in FMMs, each state can only go to two possible states on a given input. For example, assume that the states are labeled as $n+m$ bit numbers from $00\ldots0$ to $11\ldots1$, where the first n bits correspond to the values $u(k-n), u(k-n+1), \ldots, u(k-1)$ and the following m bits correspond to $y(k-m), y(k-m+1), \ldots, y(k-1)$. Since the next state is completely defined by the content of the taps and the current input and output, there are only two possible states that any state can transition to on an input of zero, and another two on an input of one. For example, consider the case when $n = m = 2$. On an input of zero, the state $q_7 = 0111$, can only transition to either $q_{10} = 1010$ if the output is defined to be zero or $q_{11} = 1011$ if the output is defined to be one. Table 3 shows the possible sets of next states for each of the sixteen states in an FMM with $n = m = 2$.

It turns out that there are always exactly four states that can go to the same two possible next states. We define these four states as a *group*. The four states within each group correspond to the possible values of $u(k-n)$ and $y(k-m)$, since these are the values that are discarded on a subsequent time step. Formally,

Definition 3 A set of states is said to define a *group* if the encoding of these states are identical except for the values of $u(k-n)$ and $y(k-m)$. Denote a group by the vector of the common $n-1$ values of the input and $m-1$ values of the output, i.e. by

$$G = \Big[u(k-n+1)\ \ldots\ u(k-1)\ \ y(k-m+1)\ \ldots\ y(k-1)\Big].$$

□

Since there are only four possible assignments to $u(k-n)$ and $y(k-m)$, every group consists of exactly four states. So, if the machine has $|Q|$ states, it will have $\frac{|Q|}{4}$ groups. Furthermore, every state in the same group has the same set of possible next states as illustrated in Table 3.

Property 1 Two states in different groups cannot have a common next state. □

The above property must be true since if the groups are different, then by definition they differ in at least one bit that will define the encoding of the state on the next time step, thus corresponding to different states.

Constructing FMMs of Minimal Order: The Group Linking Method

The fundamental rule of the Group Linking Method is to ensure that there are no two states in the same group which produce the same outputs for both $u(k) = 0$ and $u(k) = 1$. There are exactly

Present state	*Encoding*	*Group*	*Next state*	
			$u(k) = 0$	$u(k) = 1$
q_0	0000	[00]	q_0 or q_1	q_4 or q_5
q_2	0010	[00]	q_0 or q_1	q_4 or q_5
q_8	1000	[00]	q_0 or q_1	q_4 or q_5
q_{10}	1010	[00]	q_0 or q_1	q_4 or q_5
q_1	0001	[01]	q_2 or q_3	q_6 or q_7
q_3	0011	[01]	q_2 or q_3	q_6 or q_7
q_9	1001	[01]	q_2 or q_3	q_6 or q_7
q_{11}	1011	[01]	q_2 or q_3	q_6 or q_7
q_4	0100	[10]	q_8 or q_9	q_{12} or q_{13}
q_6	0110	[10]	q_8 or q_9	q_{12} or q_{13}
q_{12}	1100	[10]	q_8 or q_9	q_{12} or q_{13}
q_{14}	1110	[10]	q_8 or q_9	q_{12} or q_{13}
q_5	0101	[11]	q_{10} or q_{11}	q_{14} or q_{15}
q_7	0111	[11]	q_{10} or q_{11}	q_{14} or q_{15}
q_{13}	1101	[11]	q_{10} or q_{11}	q_{14} or q_{15}
q_{15}	1111	[11]	q_{10} or q_{11}	q_{14} or q_{15}

Table 3: Possible state transitions for a FMM of input order n and output order m. The encoding of the states corresponds to the values $u(k-2)$, $u(k-1)$, $y(k-2)$, $y(k-1)$. Each entry labeled "q_i or q_j" corresponds to an output of either 0 or 1 respectively.

Present state	*encoding*	*group*	$y(k)$		*Next state*	
			$u(k)=0$	$u(k)=1$	$u(k)=0$	$u(k)=1$
q_0	0000	[00]	0	0	q_0	q_4
q_2	0010	[00]	0	1	q_0	q_5
q_8	1000	[00]	1	0	q_1	q_4
q_{10}	1010	[00]	1	1	q_1	q_5
q_1	0001	[01]	0	1	q_2	q_7
q_3	0011	[01]	1	1	q_3	q_7
q_9	1001	[01]	0	0	q_2	q_6
q_{11}	1011	[01]	1	0	q_3	q_6
q_4	0100	[10]	0	0	q_8	q_{12}
q_6	0110	[10]	1	0	q_9	q_{12}
q_{12}	1100	[10]	1	1	q_9	q_{13}
q_{14}	1110	[10]	0	1	q_8	q_{13}
q_5	0101	[11]	1	1	q_{11}	q_{15}
q_7	0111	[11]	1	0	q_{11}	q_{14}
q_{13}	1101	[11]	0	0	q_{10}	q_{14}
q_{15}	1111	[11]	0	1	q_{10}	q_{15}

Table 4: An example FMM constructed to have minimal order. The encoding of the states corresponds to the values $u(k-2)$, $u(k-1)$, $y(k-2)$, $y(k-1)$.

four choices for output assignments for each state. Specifically, the choices are

$$\left\{0/0, 1/0\right\}, \quad \left\{0/0, 1/1\right\}, \quad \left\{0/1, 1/0\right\}, \quad \text{and} \quad \left\{0/1, 1/1\right\} \tag{3}$$

where u/y denotes an input/output pair. Since there are exactly four states in every group, then the choices of possible outputs must be a permutation of the values in equation (3). For each group there are exactly $4! = 24$ possible ways, called *group mappings*, to specify the next state mapping for each of the four states within a group in such a way that no two states have the same next state mapping for both inputs. A consequence of the GLM is that any pair of states within the same group is distinguishable in one time step.

For example, one possible FMM constructed by the GLM with $n = m = 2$ is illustrated in Table 4.

Finally, we will shall always assume that the initial state of the machine q_0 corresponds to the zero vector, i.e. $u(k-i) = 0$ for $i = 1, \ldots, n$ and $y(k-i) = 0$ for $i = 1, \ldots, m$.

Controlling the logic complexity with the GLM

If the group mappings are chosen to have the same "pattern" for each group, then the resulting logic function is simple and only depends on the current input and the last tap in each delay line, i.e. $u(k)$, $u(k-n)$, and $y(k-m)$. For example, our the FMM defined by equation (1) was obtained by having one group mapping for all groups. The resulting logic function was easily be obtained by simply forming a logic table which defined $y(k)$ in terms of $u(k)$, $u(k-5)$ and $y(k-4)$, and then deriving the function through a standard Karnaugh map (Kohavi, 1978).

One the other hand, if the group mappings are chosen randomly, then the resulting logic function may be quite complex, and will in all likelihood depend on all of the previous input and output values.

Logic functions of intermediate levels of complexity can be constructed by having the group mapping depend on a small collection of other variables. For example, the FMM defined by equation (2) was obtained by having three different group mappings: one when $u(k-1) = 0$, one when $u(k-1) = 1$ and $y(k-1) = 0$, and a different group mapping when $u(k-1) = 1$ and $y(k-1) = 1$. In fact, the terms inside the brackets are logic functions corresponding to each group mapping, and the terms outside of each bracket effectively multiplex the appropriate group mapping depending on the values of $u(k-1)$ and $y(k-1)$.

A Proof that the GLM yields minimal FMMs

In order to prove that the GLM yields minimal FMMs of size 2^{n+m}, then we must prove that every pair of states is distinguishable and every state is reachable from the initial state. Proving distinguishability turns out to be relatively simple, but the proof of reachability is rather complex. We begin with the proof of distinguishability.

Lemma 1 Every pair of states in an FMM constructed using the Group Linking Method is distinguishable. □

Proof: Because the machine has finite memory, then by definition all pairs of states either have a distinguishing string of length at most $\max(n, m) + 1$, or the pair of states transitions to a single state on some input. According to Property 1, if the states are in different groups, they cannot have the same next state. Thus, they must have a distinguishing string. If the pair of states is in the same group, then by definition of the GLM, all such pairs are 1–distinguishable.

Q.E.D.

In order to prove that every state is reachable from the initial state, we shall prove that the graph corresponding to the FMM is strongly connected. This is a sufficient, but much stronger condition than necessary to prove reachability. We begin with the following property.

Property 2 If the FMM with $n, m > 2$ is constructed using the GLM, then each state has two different successors in different groups and has two different predecessors from the same group. □

This property must hold for the following two reasons. First, the successors must be different since on an input of zero, every state goes to a state in some group that has a zero in the $(n-1)$–st component of the group vector, i.e. to

$$G_0 = \left[u(k-n+1) \;\ldots\; 0 \;\; y(k-m+1) \;\ldots\; y(k-1)\right],$$

while on an input of one, the group must have a one in that component, i.e. to a state in group

$$G_1 = \left[u(k-n+1) \;\ldots\; 1 \;\; y(k-m+1) \;\ldots\; y(k-1)\right].$$

Second, according to Property 1, the successors of a state must be in the same group.

Lemma 2 If the graph corresponding to a FMM is connected and has Property 2, then the graph is strongly connected. □

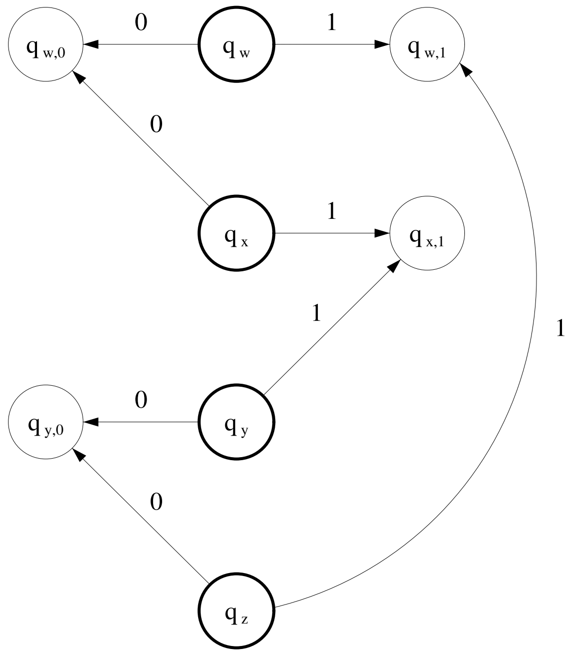

Figure 11: States q_w, q_x, q_y, and q_z are all in the same group. According to the GLM, it follows that they must be in a connected subgraph.

Proof: Assume this connected graph is not strongly connected, then we can always divide this graph into two subgraphs, G_i and G_j, such that there are edges from nodes in G_i to nodes in G_j, but not in the opposite direction. According to Property 2, there are exactly two incoming and two outgoing edges for each node in the graph, so all incoming edges in graph G_j are accounted for by the outgoing edges from the nodes within G_j. Thus, it is impossible to have additional incoming edges from graph G_i without violating Property 2, and thus by contradiction, if the graph is connected, it must also be strongly connected.

Q.E.D.

Lemma 2 is not sufficient to prove that the graph is strongly connected, since it leaves open the possibility that the graph is disconnected with multiple strongly connected components. We shall now prove that the graph is connected, by first showing that the states within a group must be in a common connected graph, and then showing that all of the groups are connected.

Lemma 3 If the graph corresponding to a FMM has a subgraph which contains at least one element of a group, then this subgraph must contain all elements of this group. □

Proof: Suppose some state q_w from group G is in some subgraph. By definition q_w is connected to at least two other nodes $q_{w,0}$ and $q_{w,1}$, corresponding to the transitions on an input of 0 and 1 respectively. (Note that $q_{w,0}$ or $q_{w,1}$ may be the same state as q_w, but this will not affect the result.) According to the way the FMM is constructed, then there is exactly one additional state, $q_x \in G$, that transitions to $q_{w,0}$ on an input of 0. By definition of the GLM this state must transition to a different state, $q_{x,1}$, on an input of 1, as illustrated in Figure 11. By the same argument, there must be a third state, $q_y \in G$, that connects to $q_{x,1}$, but not to $q_{w,0}$. Instead, on an input of 0, q_y must transition to yet another state $q_{y,0}$. Applying this argument a third and final time, it follows

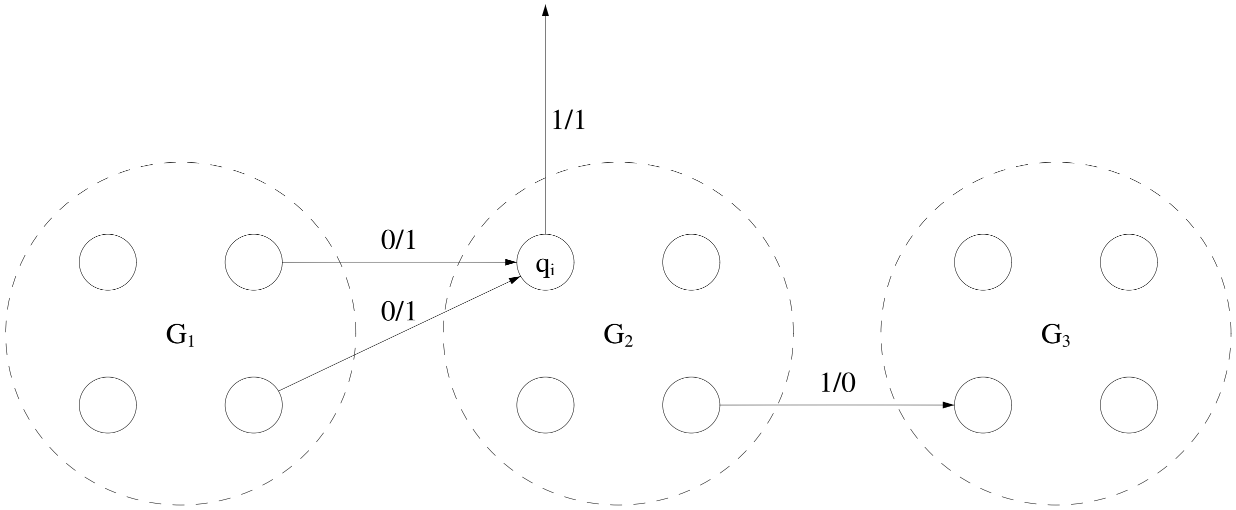

Figure 12: A path in a group graph need not correspond to a valid path in the FMM. Here the path corresponding the the input/output pair 0/1 followed by 1/0 will traverse a path from group G_1 to G_2 to G_3. In the corresponding FMM, the input/output pair 0/1 can take the machine to a state $q_i \in G_2$, but from here it is impossible to get to a state in group G_3 since an input of 1 produces and output of 1 from state q_i.

that there must be a fourth state, $q_z \in G$, that connects to $q_{y,0}$. On an input of 1, this state will transition to $q_{w,1}$.

Q.E.D.

Lemma 3 shows that all of the states within a group must be in a common connected subgraph. The following lemma, proves that all of these connected subgraphs are themselves connected together, thus showing that the entire graph is connected. First, we need the following definition.

Definition 4 The *group graph* for $\mathcal{M}$ is a labeled digraph $\mathcal{G} = (V, E)$ where each vertex v_i corresponds to a group G_i, and an arc exists between two vertices v_i and v_j if there is a transition from some state in G_i to some state in G_j. This arc is labeled by u/y where u is the input which causes the transition between the states and y is the corresponding output. □

Lemma 4 The group graph is strongly connected. □

Proof: Assume the FMM has input order n and output order m. According to the GLM, for any value of u and y there exists exactly two states in every group that on an input of u produces an output of y. Thus, every node in the group graph will have four outgoing arcs labeled 0/0, 0/1, 1/0, and 1/1. Because every group corresponds to the values of

$$\left[u(k-n+1)\ \ldots\ u(k-1)\ \ y(k-m+1)\ \ldots\ y(k-1)\right]$$

then there exists a path of length $\max(n-1, m-1)$ from any vertex in $\mathcal{G}$ to any other vertex. For example, in order to get to vertex corresponding to the group

$$\left[0\ \ldots\ 0\ \ 0\ \ldots\ 0\right]$$

from any other vertex, we simply follow the path corresponding to all arcs labeled 0/0. (In general, a path in the group graph will not correspond to a path in the corresponding FMM, as illustrated in Figure 12.) Since any state can be reached by any other state, then by definition the graph is strongly connected.

Q.E.D.

Lemma 5 Every state in an FMM constructed according to the Group Linking Method is reachable from the initial state q_0. □

Proof: Lemmas 3 and 4 show that the graph corresponding to an FMM constructed by the GLM is connected. According to Lemma 2, the graph must be strongly connected. Therefore, every state is reachable from the initial state.

Q.E.D.

Theorem 1 An FMM constructed according to the Group Linking Method is minimal. □

Proof: From Lemmas 1 and 5, it follows that an FMM constructed according to the GLM is minimal.

Q.E.D.

The number of FMMs of minimal order

The number of FMMs that can be constructed using the GLM is extremely large. First, we note that any two FMMs constructed according to the method are different. Minimal FSMs have the property that they are unique up to a relabeling of their states. Because of the nature of FMMs, the encoding is predetermined by the fact that the states of the machine are delayed versions of the input and output. In addition, the initial state of an FMM constructed according to the GLM is always the zero vector, i.e. $u(k-i)=0$ for $i=1,\ldots,n$ and $y(k-i)=0$ for $i=1,\ldots,m$. Thus, it is impossible for two different FMMs constructed according to the GLM to be equivalent.

In each group, there are $4! = 24$ different next state assignments collectively for the four states in each group corresponding to the possible permutations of the four values given in equation (3). In a FMM of input order n and output order m, there are 2^{n+m-2} groups. So the number of different FMMs constructible according to the GLM is

$$(24)^{2^{n+m-2}}.$$

For comparison, if there are no restrictions placed on the transitions of states within a group, there are $4^4 = 256$ possible different next state assignments for the four states in each group, since each of the four states can be given one of the four output assignments in equation (3). Many of these assignments may yield equivalent FMMs, or even FMMs with non–reachable states. In any case, there are at most

$$(256)^{2^{n+m-2}}.$$

different FMMs. Thus the FMMs constructed according to the GLM is a considerable portion of all possible FMMs.

Minimal Order FMM Identification

Theorem 2 It is possible to identify a minimal order FMM of depth d from the complete set of strings not longer than $d+1$. □

Proof: We assume that the input and output alphabets are binary. We also assume that the initial state of the machine is known. Specifically, for a hardware implementation like the one illustrated in Figure 4, the appropriate initial values of the taps must be known. Thus, all we need to know is how many states there are and what the transition and output functions are.

First, in a complete data set every prefix of every string is also in the data set. Thus, it is possible to use the labeling of these prefixes to determine the output at every time step for every string in the training set. Since the state of an FMM is completely specified by it previous n inputs and m outputs (where knowledge of the initial state defines previous values for strings of length less than $\max(m, n)$), then the state of the system is known for every time step for every string. Furthermore, since the data set consists of every string up to length d, every state is visited.

Second, if, for each state q and each input symbol a, the data set contains the string $w = w_q a$ such that $\delta(q_0, w) = q$, then $\lambda(q, a)$ is defined by the label of w. Since $|w_q| \leq d$ and, by the arguments above, the state is known at each time step, having the complete data set up to length $d + 1$ will be sufficient to define every value of $\lambda(q, a)$.

Finally, once the output function is known it is trivial to determine the transition function, since the states of the FMM can always be implemented as tapped delay lines of the inputs and outputs, as shown in Figure 4. The values of the taps at the next time step are easily computed from the current values, and the current input and output.

Q.E.D.

References

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

Ashar, P., Devadas, S., and Newton, A. (1992). *Sequential Logic Synthesis*. Kluwer Academic Publishers, Norwell, MA.

Back, A. and Tsoi, A. (1991). FIR and IIR synapses, a new neural network architecture for time series modeling. *Neural Computation*, 3(3):375–385.

Barto, A. G. (1990). Connectionist learning for control. In Miller, W., Sutton, R., and Werbos, P., editors, *Neural Networks for Control*. MIT Press, Cambridge, MA.

Billings, S., Jamaluddin, H., and Chen, S. (1992). Properties of neural networks with applications to modelling non-linear dynamical systems. *International Journal of Control*, 55(1):193–224.

Blum, A. and Rivest, R. (1988). Training a 3–node neural network is NP–complete. In *Proceedings of the Computational Learning Theory (COLT) Conference*, pages 9–18. Morgan Kaufmann.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. (1989). Finite state automata and simple recurrent recurrent networks. *Neural Computation*, 1(3):372–381.

Connor, J., Martin, R., and Atlas, L. (1994). Recurrent neural networks and robust time series prediction. *IEEE Transactions on Neural Networks*, 5(2):240–254.

Das, S. and Das, R. (1991). Induction of discrete state-machine by stabilizing a continuous recurrent network using clustering. *Computer Science and Informatics*, 21(2):35–40. Special Issue on Neural Computing.

Elman, J. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Elman, J. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7(2/3):195–226.

Frasconi, P., Gori, M., and Soda, G. (1992a). Injecting nondeterministic finite state automata into recurrent neural networks. Technical Report DSI-RT15/92, Universitá di Firenze, Dipartimento di Sistemi e Informatica, Firenze, Italy.

Frasconi, P., Gori, M., and Soda, G. (1992b). Local feedback multilayered networks. *Neural Computation*, 4:120–130.

Fu, K. and Booth, T. (1975). Grammatical inference: Introduction and survey – Part I. *IEEE Transactions on Systems, Man and Cybernetics*, 5:95–111.

Fu, L. (1994). *Neural Networks in Computer Intelligence.* McGraw-Hill, Inc., New York, NY.

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

Giles, C., Sun, G., Chen, H., Lee, Y., and Chen, D. (1990). Higher order recurrent networks & grammatical inference. In Touretzky, D., editor, *Advances in Neural Information Processing Systems 2*, pages 380–387, San Mateo, CA. Morgan Kaufmann Publishers.

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

Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences - USA*, volume 79, page 2554.

Jordan, M. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Ninth Annual conference of the Cognitive Science Society*, pages 531–546. Lawrence Erlbaum.

Judd, J. (1990). *Neural Network Design and the Complexity of Learning.* MIT Press, Cambridge, MA.

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

Krogh, A. and Hertz, J. (1992). A simple weight decay can improve generalization. In Moody, J., Hanson, S., and Lippmann, R., editors, *Advances in Neural Information Processing Systems 4*, pages 950–957.

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*, pages 45–52, New York, N.Y. ACM.

Lang, K., Waibel, A., and Hinton, G. (1990). A time–delay neural network architecture for isolated word recognition. *Neural Networks*, 3(1):23–44.

Lapedes, A. and Farber, R. (1987). Nonlinear signal processing using neural networks: Prediction and signal modeling. Technical Report LA–UR–87–2662, Los Alamos National Laboratories, Los Alamos, New Mexico.

Leighton, R. and Conrath, B. (1991). The autoregressive backpropagation algorithm. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 369–377.

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

Narendra, K. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Trans. on Neural Networks*, 1(1):4.

Nerode, A. and Kohn, W. (1993a). Models for hybrid systems: Automata, topologies, controllability, observability. Technical Report TR 93-28, Mathematical Sciences Institute, Cornell University, Ithaca, New York 14850.

Nerode, A. and Kohn, W. (1993b). Multiple agent hybrid control architecture. Technical Report TR 93-12, Mathematical Sciences Institute, Cornell University, Ithaca, New York 14850.

Nerrand, O., Roussel-Ragot, P., L. Personnaz, G. D., and Marcos, S. (1993). Neural networks and non-linear adaptive filtering: Unifying concepts and new algorithms. *Neural Computation*, 5:165–197.

Poddar, P. and Unnikrishnan, K. (1991). Nonlinear prediction of speech signals using memory neuron networks. In Juang, B., Kung, S., and Camm, C. A., editors, *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE Workshop*, pages 395–404, Piscataway, NJ. IEEE Press.

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

Robinson, A. and Fallside, F. (1988). Static and dynamic error propagation networks with application to speech coding. In Anderson, D., editor, *Neural Information Processing Systems*, pages 632–641, New York, NY. American Institute of Physics.

Sastry, P., Santharam, G., and Unnikrishnan, K. (1994). Memory neuron networks for identification and control of dynamical systems. *IEEE Transactions on Neural Networks*, 5(2):306–319.

Siegelmann, H. and Sontag, E. (1992). On the computational power of neural nets. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, pages 440–449, New York, N.Y. ACM.

Sun, R. (1994). *Integrating Rules and Connectionism for Robust Commonsense Reasoning.* John Wiley, New York, NY.

Trakhenbrot, B. and Barzdin, Y. (1973). *Finite automata: Behavior and synthesis.* North–Holland, Amsterdam.

Tsoi, A. and Back, A. (1994). Locally recurrent globally feedforward networks, a critical review of architectures. *IEEE Transactions on Neural Networks*, 5(2):229–239.

Vries, B. d. and Principe, J. (1992). The gamma model - a new neural network for temporal processing. *Neural Networks*, 5(4):565–576.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. (1989). Phoneme recognition using time–delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(3):328–339.

Wan, E. (1994). Time series prediction by using a connectionist network with internal delay lines. In Weigend, A. and Gershenfeld, N., editors, *Time Series Prediction*, pages 195–217. Addison–Wesley.

Watrous, R. and Kuhn, G. (1992a). Induction of finite-state languages using second-order recurrent networks. *Neural Computation*, 4(3):406.

Watrous, R. and Kuhn, G. (1992b). Induction of finite state languages using second-order recurrent networks. In Moody, J., Hanson, S., and Lippmann, R., editors, *Advances in Neural Information Processing Systems 4*, pages 309–316, San Mateo, CA. Morgan Kaufmann Publishers.

Williams, R. and Peng, J. (1990). An efficient gradient–based algorithm for on–line training of recurrent network trajectories. *Neural Computation*, 2(4):490–501.

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

Williams, R. and Zipser, D. (1990). Gradient–based learning algorithms for recurrent connectionist networks. Technical Report NU–CCS–90–9, College of Computer Science, Northeastern University.

Zeng, Z., Goodman, R., and Smyth, P. (1994). Discrete recurrent neural networks for grammatical inference. *IEEE Transactions on Neural Networks*, 5(2):320–330.