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

Abstract

One of the issues in any learning model is how it scales with problem size. 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. We show that a type of recurrent neural networks (Narendra and Parthasarathy, 1990) which has feedback but no hidden state neurons can learn a special type of FSM called Finite Memory Machines (FMMs) under certain constraints. These machines 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.

1 Introduction

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; Giles et al., 1992; Mozer and Bachrach, 1990; Pollack, 1991) and system identification in control (Barto, 1990). We discuss results concerning the learning of temporal sequences for a particular class of discrete–time recurrent neural network architectures which has tapped delays both on the input and on the feedback of the output (Narendra and Parthasarathy, 1990). Such models are similar to feedback networks described by others (Back and Tsoi, 1991; Billings et al., 1992; Frasconi et al., 1992; Jordan, 1986; Poddar and Unnikrishnan, 1991; de Vries and Principe, 1992).

We show that this model is able to learn to emulate large finite state machines (FSMs) from example strings of their associated grammars. The finite state machines that were easily learned are from a subclass of FSMs called *finite memory machines (FMMs)* (Kohavi, 1978). These FMMs, defined by the type of memory used and how fed back, have relatively low depth; and when implemented as a sequential machine require minimal memory and simple combinational logic.

[†]Published in *Neural Networks*, 8(*9*), p. 1359, 1995.

2 Finite State, Finite Memory and Sequential Machines

We briefly introduce finite state machines (FSMs) and their properties. An FSM is an abstraction of a device that can be described by a labeled directed acyclic 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 this paper are synchronous.

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. An output is defined for each corresponding input. Formally,

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*; and q_0 is the initial state. □

For this work, both the input and output alphabets will be binary, i.e. $\Sigma = \Delta = \{0, 1\}$.

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. A useful measure for characterizing an FSM is its *depth*, which is the the smallest integer d such that every state in the FSM can be reached from the starting state in no more than d steps.

Grammatical inference (Fu and Booth, 1975) is the problem of finding an FSM consistent with a set of labelled strings. These results are typically defined in terms of deterministic finite–state automata (DFA), however it is straightforward to map a DFA into an FSM. Grammatical inference is known to be NP–complete (Angluin, 1978) in the general case. However, some approaches have been suggested which seem to work well on relatively large problems (Lang, 1992).

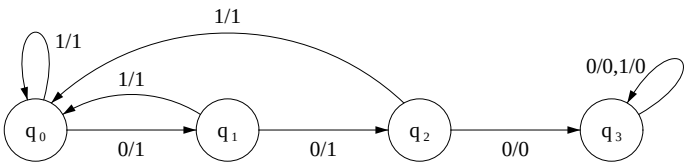
2.2 Finite Memory Machines

Consider the 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 this definition excludes the possibility of any knowledge of the initial state of the machine. For example, the FSM shown in Figure 1, 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 the table in Figure 1. Not all FSMs have finite memory, some have infinite order. For example, the FSM shown in Figure 2 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).



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

Figure 1: A finite memory machine (FMM) of input–order 2 and output–order 1. The state transition diagram of this FMM.

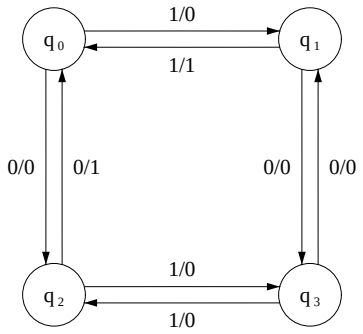


Figure 2: An FSM that has infinite order.

2.3 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 3a.

We can explicitly associate time with an FSM in the following way. The input, output and state of the machine at time k will be denoted by respectively $u(k)$, $y(k)$ and $x(k)$, and are typically encoded as binary values. Logic functions can be used to relate $x(k+1)$ and $y(k)$ to $x(k)$ and $u(k)$, to define the SM.

In an FMM, the state depends only on a finite number of previous inputs and outputs, so it can always be implemented as a SM with tapped delay lines (TDLs) on the input and output and a block of combinational logic as shown in Figure 3b.

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., 1992; Giles et al., 1992; Jordan, 1986; Poddar and Unnikrishnan, 1991; Robinson and Fallside, 1988; de Vries and Principe, 1992; Watrous and Kuhn, 1992; 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 connection is from the output of the network.*

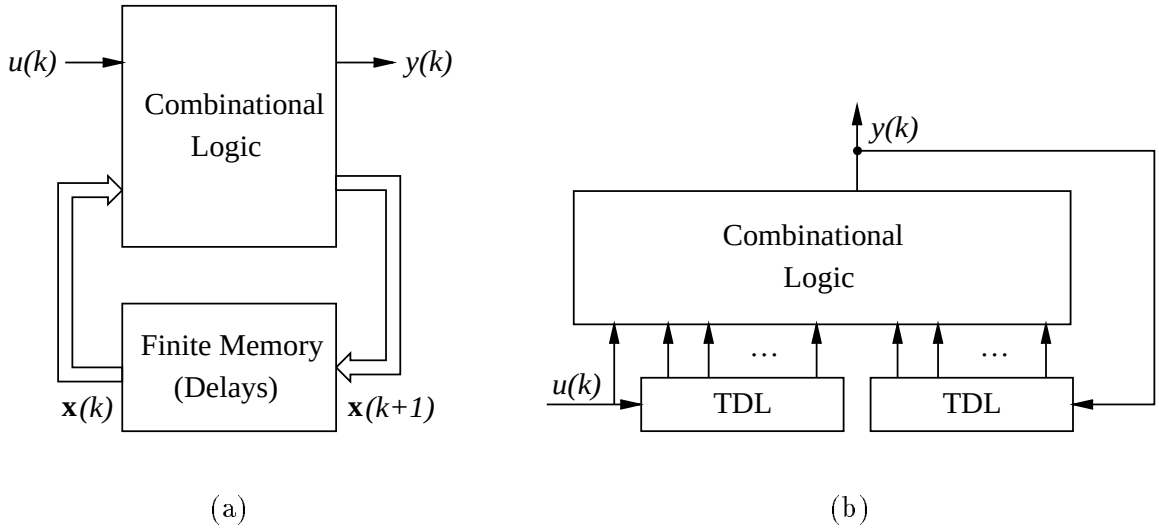


Figure 3: Sequential machines: (a) conventional implementation and (b) implementation of a finite memory machine.

Because of their similarity to infinite impulse response filters (IIRs), these networks are often referred to as neural network IIRs (NNIIRs). Many variations of this model have been proposed and used extensively for system identification and control problems (Narendra and Parthasarathy, 1990). In the most general model, the function $f(\cdot)$ is implemented as a multilayer perceptron.

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 3b, when the logic is replaced by a multilayer perceptron.

4 Learning Finite Memory Machines

4.1 Example Problems

We have successfully been able to learn various FMMs using the NNIIR model. 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. Instead, we developed theory which devises a method for constructing machines which we then use for example learning problems (Giles et al., 1994). This theory allows us to construct FMMs and have 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.

We have found that a machine can be easily learned if it can be described by a simple logic function, and is of minimal order and low depth. The problem of learning FMMs is simpler than that of learning general FSMs since there is no state assignment problem. The only problem is to learn the logic function of the sequential machine implementation. Thus, it makes sense that as that function becomes more complex, the learning problem becomes more difficult. The depth must be kept low to keep the training set small. Finally, the order is related to both the depth and complexity of logic and so must also be kept small.

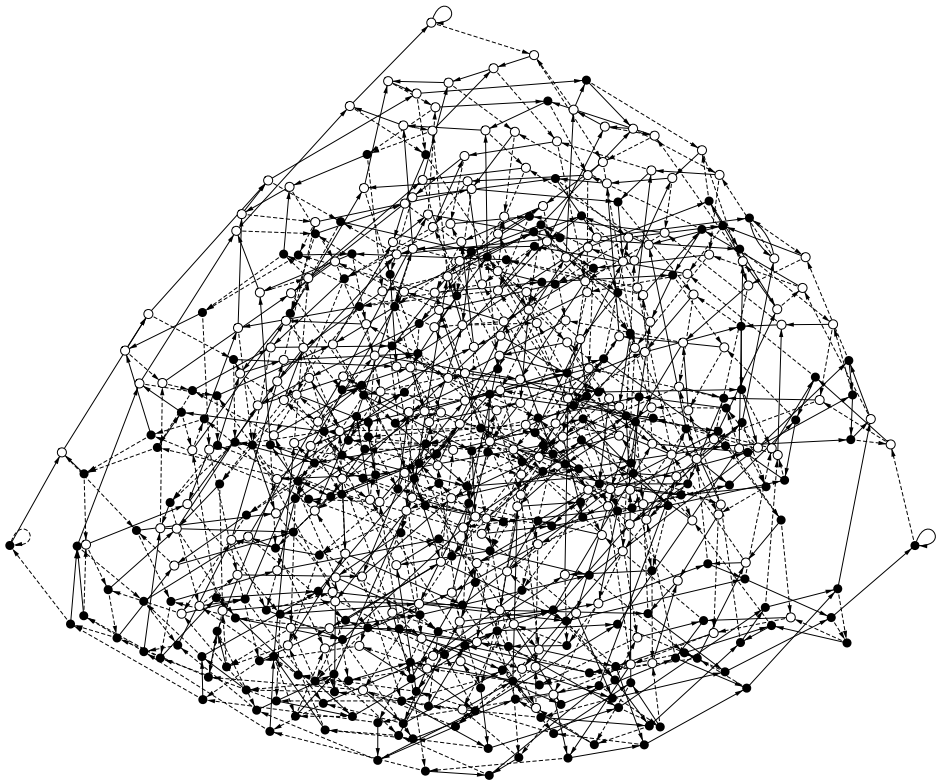


Figure 4: A 512 state finite memory machine of minimal order.

In this paper we present results for learning two FMMs, although we have successfully learned many other similar machines. The first machine has 512 states and corresponds to the relatively simple 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 4. It has an input–order of 5, an output–order of 4, and a depth of 9.

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 and output order of 4, and also has a depth of 9.

4.2 Training and Testing Set

It can be shown that the set of strings of length 1 through $d+1$ is sufficient to uniquely identify an arbitrary FMM (Giles et al., 1994). To create training sets, we began with this complete set consisting of 2,046 strings (since $d = 9$ for both problems) and randomly selected a subset of

strings for training and reserved the remaining samples for testing. A similar approach was taken by Lang (Lang, 1992), and we feel it is a reasonable technique for generating data for this kind of problem.

In principle, the neural network is capable of learning machines with a larger depth. However, in order to run a large number of experiments 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 give a set of 4,194,302 strings.

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

It is possible to generate target outputs at intermediate points in each string for a given training set. For example, if the string "0" is a negative string, then any string that begins with "0" can be assigned a target output of 0.0 on the first time step. Similarly, if the string "10" is a positive string, then any string that begins with "10" can be assigned a target value of 1.0 on the second time step. By utilizing all of this information, many intermediate target values can be constructed for each string. 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.

4.3 Network Architecture and Learning Algorithm

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. Each node used the standard sigmoid nonlinearity. The initial values of all delay elements were chosen to be zero. The networks had 49 and 181 adjustable weights respectively with the initial values randomly chosen from a uniform distribution in the range $[-0.1, 0.1]$.

The network was trained with Backpropagation Through Time Algorithm (Williams and Peng, 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. 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 5,000 epochs for the 512–state or 10,000 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 1,500 experiments. On the second experiment, the algorithm required about 2,500 epochs and did not converge on 68 of the 1,500 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.

4.4 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. 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 2,046 possible samples needed to completely specify the machine. The results are shown in Figure 5. The average error rate is plotted with an error bar of one standard deviation around the mean for the two problems in Figures 5a and 5c.

It is easy to extract the FSM that the network learns. By replacing nonlinearity of the output node 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 an FSM for that machine. This FSM can be minimized using a standard FSM minimization algorithm (Hopcroft and Ullman, 1979). The average sizes of the extracted FSMs are plotted in Figures 5b and 5d with an error bar of one standard deviation around the mean.

4.5 Discussion of Experimental Results

For learning the 512–state FMM, one notices from Figure 5a that as the percentage of training strings increases, the testing set error decreases and finally approaches zero with zero error. Similar behavior is noticed for extraction size in Figure 5b 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 complete set of 2,046 strings which uniquely identify 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.

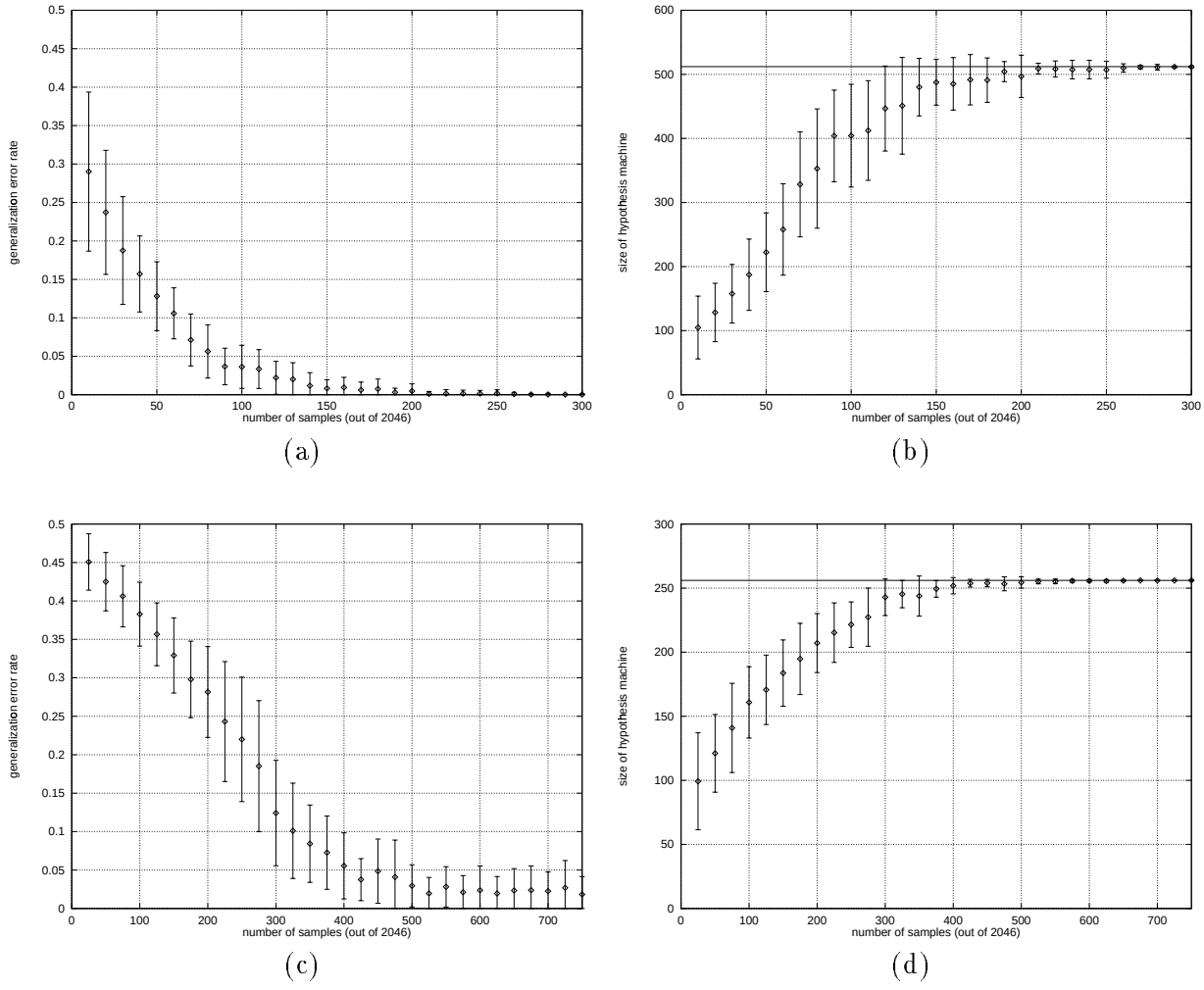


Figure 5: Generalization and extraction as a function of training set size: (a) generalization on the 512–state FMM, (b) extraction on the 512–state FMM, (c) generalization on the 256–state FMM, and (d) extraction on the 256–state FMM.

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, 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.

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

We showed an NNIIR is capable of learning large (up to 512 states) finite memory machines (FMMs) 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), is the greatest barrier for learning arbitrary FMMs. It is important to keep in mind that the restrictions on order, depth, and logic define a 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).

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.

References

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

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. A., Jamaluddin, H. B., 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. L. (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.

de Vries, B. and Principe, J. C. (1992). The gamma model — A new neural model for temporal processing. *Neural Networks*, 5:565–576.

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

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

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

Giles, C. L., Horne, B. G., and Lin, T. (1994). Learning a class of large finite state machines with a recurrent neural network. Technical Report UMIACS–TR–94–94 and CS–TR–3328, Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland.

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

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. Addison–Wesley, Reading, MA.

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

Kohavi, Z. (1978). *Switching and finite automata theory*. McGraw–Hill, New York, NY, 2nd edition.

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

Lang, K. (1992). Random DFAs can be approximately learned from sparse uniform examples. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*.

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

Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1:4–27.

Poddar, P. and Unnikrishnan, K. P. (1991). Non–linear prediction of speech signals using memory neuron networks. In Juang, B. H., Kung, S. Y., and Kamm, C. A., editors, *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE Workshop*, pages 1–10. IEEE Press.

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

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

Siegelmann, H. T. and Sontag, E. D. (1992). On the computational power of neural networks. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, pages 440–449. ACM Press.

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

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