Extraction, Insertion and Refinement of Symbolic Rules in Dynamically-Driven Recurrent Neural Networks *

C.L. Giles[a,c] and C. W. Omlin[a,b]

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

[b]Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180

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

Abstract

Recurrent neural networks readily process, learn and generate temporal sequences. In addition, they have been shown to have impressive computational power. Recurrent neural networks can be trained with symbolic string examples encoded as temporal sequences to behave like sequential finite state recognizers. We discuss methods for extracting, inserting and refining symbolic grammatical rules for recurrent networks. This paper discusses various issues: how rules are inserted into recurrent networks, how they affect training and generalization, and how those rules can be checked and corrected. The capability of exchanging information between a symbolic representation (grammatical rules) and a connectionist representation (trained weights) has interesting implications. After partially known rules are inserted, recurrent networks can be trained to preserve inserted rules that were correct and to correct through training inserted rules that were "incorrect" - rules inconsistent with the training data.

1 INTRODUCTION

1.1 Motivation

Artificial neural networks have become an accepted tool in the machine learning toolbox. They are well known for their ability to be trained from data but usually not so for their use of symbolic knowledge. Neural networks do not readily store knowledge in symbolic form; rather, their knowledge is stored in distributed, internal weights. Certainly some a priori knowledge is used in neural networks, for example in the features selected for neural network inputs, the input order and, of course, the choice of learning algorithms. These learning or training algorithms are based on numerical optimization techniques which adapt the weights of a neural network for a particular task such as pattern classification or regression. Though neural networks are known and well-used for their generalization capabilities, other applications may require that the knowledge be utilized in symbolic form. In addition, the problem of understanding the rules that the network has learned is an interesting problem in itself - a "reading of entrails" [12]. We (and others - see some of the references in this paper) contend that in many instances it is useful to interpret and understand the knowledge learned by connectionist networks. From a different perspective, if some prior symbolic knowledge about some learning task is available, then it can be effectively utilized to simplify the learning task, yield better generalization performance and validate and correct that prior knowledge. The role of learning can then become that of *knowledge refinement* [67]. Thus, it is of interest to study how the *exchange of information between symbolic and connectionist representations* can be effectively utilized.

*Appeared in *Connection Science*, vol. 5, no. 3-4, p. 307, 1993.

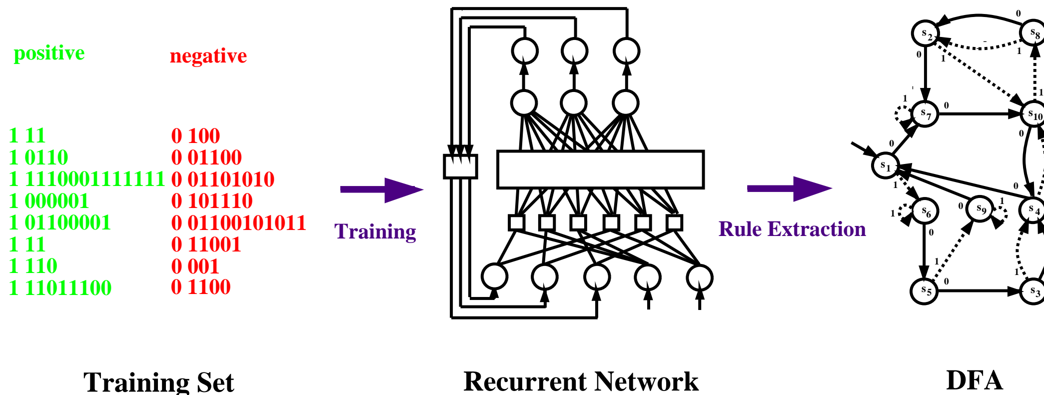

Figure 1: **Overview of the Training and Rule Extraction Process.** This figure gives a descriptive overview of the training and rule extraction process which is discussed in detail in the text. The training set consists of sample strings of an unknown regular language which is associated with a deterministic finite automata (DFA). Partially known rules (transitions) of this regular grammar are inserted into a recurrent neural network by a rule insertion procedure that pre-programs a small subset of the network's weights. The strings are dynamically fed into the network one string at a time and the network is trained in real-time at the end of each string. If the training is successful, the trained network now behaves as a neural network's DFA. It correctly classifies strings of the training set and relies on its generalization capability to classify unknown strings. In the trained network, clusters of hidden neuron activations represent states of the DFA to be learned. A dynamic space exploration (clustering heuristic) extracts a symbolic representation of the DFA that the trained network has learned.

1.2 Background

Our perspective will focus on a restricted class of knowledge utilization - knowledge that is represented by discrete finite state processes [30, 35]. We focus exclusively on knowledge utilization in dynamically-driven recurrent neural networks which are trained with grammatical strings encoded as *temporal sequences*. Thus, the knowledge utilization is for a *dynamic not a static* learning structure and for temporally-encoded problems.

The use of symbolic knowledge utilization in neural networks is quite old. [42] discusses building computational devices with hard-threshold logic neural network sequential machines. Representing implicit knowledge and semantic networks in connectionist networks was discussed by respectively [14] and [32]. Implementing expert systems in neural networks is discussed by [19]. For the direct use of symbol-based knowledge in feed-forward networks, see the survey paper by [61] and other papers in this issue. For a discussion of the training of recurrent neural networks to recognize finite state languages, see for example [8, 10, 22, 34, 44, 45, 55, 68, 70]. (For a discussion of the training of neural networks to recognize context-free grammars and beyond, see for example [3, 9, 25, 38, 56, 70].)

1.3 Overview of Paper

To determine the effectiveness of knowledge utilization, we train a recurrent network with temporal symbolic sequences and prior knowledge (correct or incorrect) which is programmed into the neural network before training. More explicitly, given a training set of positive and negative example strings of a regular language, a recurrent network is trained until it correctly classifies these example strings. We then say the trained network emulates a deterministic sequential finite-state recognizer. If we have some prior knowledge in the form of known grammatical rules (or state transitions), we discuss methods for inserting this information into the network before training. We view this as a method for translating symbolic information into a neural network or connectionist representation. Since these rules are linked and depend on inputs, this a

priori information is dynamic.Training is accomplished using a gradient-descent, real-time on-line algorithm similar to RTRL [70]. The trained network relies on its generalization capability to classify unknown strings. After training we use state partitioning and clustering methods to extract deterministic finite state automata (DFA) (figure 1). The extracted DFA is readily transformed into grammatical rules. Often, these extracted DFA's outperform the trained networks in classifying unseen strings (particularly long strings) because of the error accumulation in the continuous nonlinear dynamics of the recurrent network [55]. [The work of [71] proposes a preliminary solution to this problem using discrete-space recurrent networks training methods.] After extraction and insertion of these production rules, we demonstrate that recurrent neural networks can be used as rule checkers, i.e. once rules have been inserted into a network, they can be verified and even corrected by the training data.

We are not proposing that recurrent neural networks are good methods for learning grammars or representing sequential finite state machines. However, the problem of grammatical inference is in the worst case NP [4] and is thus a good test-bed for exploring the computational capabilities and, quantitatively, the temporal symbolic knowledge utilization of recurrent networks. Because of their general purpose computational power [63], recurrent networks might not be competitive with algorithms that are specifically tailored to the problem of grammatical inference. Using the training algorithms discussed in this paper, we have learned DFA's which have on the order of 10-20 states whereas other tailored approaches have been able to learn DFA's with many more states (e.g. [36] learns randomly generated DFA's with 800 or more states). To repeat, our goal has been to explore the computational capabilities of recurrent neural networks and to develop effective methods for encoding and extracting symbolic information in these nets.

2 REGULAR LANGUAGES AND GRAMMATICAL INFERENCE

Since we will be learning strings of regular languages, we give a brief description of regular grammars; see [33] for more details. Regular languages represent the smallest and simplest class of formal languages in the Chomsky hierarchy and are generated by regular grammars. A regular grammar G is a quadruple $G =< S, V, T, P >$ where S is the start symbol, V and T are respectively non-terminal and terminal symbols and P are productions of the form $A \rightarrow a$ or $A \rightarrow aB$ where $A, B \,\epsilon\, V$ and $a \,\epsilon\, T$. The regular language generated by G is denoted $L(G)$.

A deterministic finite-state automaton (DFA) M is the recognizer of each regular language L; $L(G) = L(M)$. Formally, a DFA M is a 5-tuple $M =< \Sigma, Q, R, F, \delta >$ where $\Sigma = \{a_1, \ldots, a_k\}$ is the alphabet of the language L, $Q = \{s_1, \ldots, s_{N_s}\}$ is a set of states, $R \,\epsilon\, Q$ is the start state, $F \subseteq Q$ is a set of accepting states and $\delta : Q \times \Sigma \rightarrow Q$ define state transitions in M. A string x is accepted by the DFA M and hence is a member of the regular language $L(M)$ if an accepting state is reached after the entire string x has been read by M. Alternatively, a DFA M can be interpreted as a grammar which generates the regular language $L(M)$. A sequential finite state machine [35] is the actual implementation in some logical form consisting of logic (or neurons) and delay elements that will recognize L when the strings are encoded as temporal sequences. It is this type of representation that the recurrent neural network will learn.

The process of learning grammars from example strings is also known as *grammatical inference* [18, 4, 40]. The inference of regular grammars from positive and negative example strings has been shown to be in the worst case a NP-complete problem [27]. However, good heuristic methods have recently been developed for randomly generated DFA's [36].

3 DYNAMICALLY-DRIVEN RECURRENT NEURAL NETWORK

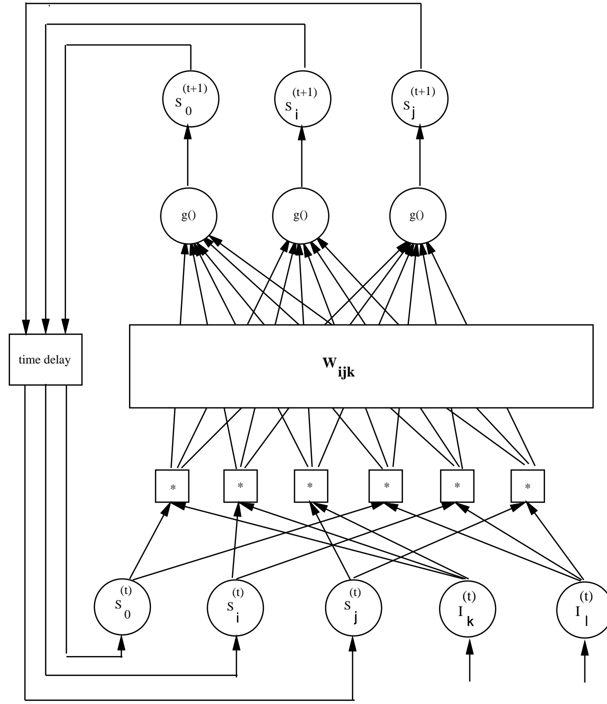

Figure 2: **Network Architecture.** A second-order, single-layer recurrent neural network consisting of hidden recurrent state neurons $S_i^{(t)}$, nonrecurrent input neurons $I_k^{(t)}$ and second-order weights W_{ijk}. The blocks marked '*' represent the operation $W_{ijk} \times S_j^{(t)} \times I_k^{(t)}$. $g()$ is the sigmoid transfer function.

3.1 Architecture Dynamics

A discussion of various recurrent neural network models and their associated references can be found in [31]. Recently, recurrent nets have been shown to have powerful capabilities for modeling many computational structures [62]. For all the learning described in this paper, we use a higher order recurrent neural network [5, 25, 37, 51, 53, 54, 55, 59]. In particular, recent theoretical and experimental work in machine representation has shown that second-order recurrent networks have some significant training and computational advantages over first-order nets [9, 28, 41, 69]. The network architecture illustrated in figure 2 is dynamically driven by temporal sequences. It has N recurrent hidden neurons labeled S_j; K special, nonrecurrent input neurons labeled I_k; and $N^2 \times K$ real-valued weights labeled W_{ijk}. The complexity of the network only grows as $O(N^2)$ as long as $K \ll N$, the same as a linear network. The values of the hidden neurons are referred to collectively as state *vectors* $\mathbf{S}$ in the finite N-dimensional space $[0,1]^N$. Second order in this case means that the weights W_{ijk} modify a product of the hidden S_j and input I_k neurons. Thus there is a direct mapping of the ordered triple W_{ijk} of the state process — $\{input, state\} \Rightarrow \{nextstate\}$. This neural network has the representational potential of at least finite-state automata. The network accepts a time-ordered sequence of inputs and evolves with dynamics defined by the following equations:

$$S_i^{(t+1)} = g(\Xi_i), \qquad \Xi_i \equiv \sum_{j,k} W_{ijk} S_j^{(t)} I_k^{(t)},$$

where g is a sigmoid discriminant function.

3.2 Input Dynamics

Each input string is encoded into the input neurons one character per discrete time step t. Each hidden neuron S_i in the above equation is updated to compute the next state vector $\mathbf{S}$ of the same hidden neurons at the next time step $t+1$. This is why we call the recurrent network "dynamically-driven." Using a unary

or one-hot encoding [35], there is one input neuron for each character in the string alphabet. After this recurrent network is trained on strings generated by a regular grammar, it can be considered as a neural network finite state recognizer or DFA.

3.3 Training Dynamics

For training, the error function and error update must be defined. In addition, the presentation of the training samples must be considered. The error function E_0 is defined by selecting a special "output" neuron S_0 of the hidden state neurons which is either on ($S_0 > 1-\epsilon$) if an input string is accepted, or off ($S_0 < \epsilon$) if rejected, where ϵ is the tolerance of the response neuron. Two error cases result from this definition: (1) the network fails to reject a negative string (*i.e.* $S_0 > \epsilon$); (2) the network fails to accept a positive string (*i.e.* $S_0 < 1-\epsilon$).

The error function is defined as:

$$E_0 = \frac{1}{2}(\tau_0 - S_0^{(f)})^2,$$

where τ_0 is the desired or *target* response value for the response neuron S_0. The target response is defined as $\tau_0 = 0.8$ for positive examples and $\tau_0 = 0.2$ for negative examples. The notation $S_0^{(f)}$ indicates the *final* value of S_0 after the final input symbol.

The training is an on-line real-time algorithm that updates the weights at the end of each sample string presentation with a gradient-descent weight update rule:

$$\Delta W_{lmn} = -\alpha \frac{\partial E_0}{\partial W_{lmn}} = \alpha(\tau_0 - S_0^{(f)}) \cdot \frac{\partial S_0^{(f)}}{\partial W_{lmn}},$$

where α is the learning rate. We also add a momentum term η as an additive update to ΔW_{lmn}. To determine ΔW_{lmn}, the $\partial S_i^{(f)} / \partial W_{lmn}$ must be evaluated.

3.4 Real-time On-Line Training Algorithm

This training algorithm updates the weights at the *end* of the input string and should be contrasted to methods that train by predicting the next string [8]. From the recursive network state equation, we see that

$$\frac{\partial S_i^{(f)}}{\partial W_{lmn}} = g'(\Xi_i) \cdot \left[\delta_{il} S_m^{(f-1)} I_n^{(f-1)} + \sum_{j,k} W_{ijk} I_k^{(f-1)} \frac{\partial S_j^{(f-1)}}{\partial W_{lmn}} \right],$$

where g' is the derivative of the discriminant function. For the last time step f, replace t and $t-1$ by f and $f-1$. (Note that this is a second-order form of the RTRL training method of Williams and Zipser [70].) Since these partial derivative terms are calculated one iteration per input symbol, the training rule can be implemented *on-line* and in *real-time*. The initial values are $\partial S_i^{(0)} / \partial W_{lmn}$ set to zero. Thus the error term is forward-propagated and accumulated at each time step t. Note that for this training algorithm each update of $\partial S_i^{(t)} / \partial W_{lmn}$ is computationally expensive and requires $O(N^4 \times K^2)$ terms. For $N >> K$, this update is $O(N^4)$ which is the same as a forward-propagated linear network. For scaling, it would be most useful to use a training algorithm that was not so computationally expensive such as gradient-descent back-propagation through time.

4 TRAINING PROCEDURE

For all experiments second-order networks with 11 recurrent hidden state neurons and three nonrecurrent input neurons (including the end symbol) are used. We start the neuron activations with a *known* initial value. All strings used in training were accepted by the DFA in figure 10a (and figure 1). This randomly

generated automaton is minimal in size and has 4 accepting states with the initial state also a rejecting state. The training set consisted of the first 500 positive and 500 negative example strings. The strings presentation was in alphabetical order, alternating between positive and negative examples [11, 25, 57]. The weights, unless initially programmed, were initialized to small random values in the interval $[-0.1, 0.1]$.

5 EXTRACTING RULES FROM RECURRENT NETWORKS

Once the network is trained (or even during training), we want to extract meaningful internal representations of the network, such as rules. For previous work on rule extraction from recurrent neural networks see [8, 15, 22, 69, 71]. For an overview that includes discussions of rule extraction in feedforward and recurrent networks see [29, 61]. [8] and [10] trained recurrent networks to recognize the Reber grammar by predicting the next symbol using a truncation of the backward recurrence. The conclusion of [8] was that the hidden unit activations represented past histories and that clusters of these activations can represent the states of the generating automaton. [22] showed that a *complete deterministic finite-state automata and their equivalence classes* can be extracted from recurrent networks both *during and after training*. This was extended in [23] to include a method for extracting bounded "unknown" grammars from a trained recurrent network. (It is important to note that nearly all of the work on grammatical inference, including what is discussed here, has been concerned with learning "known" grammars.) An alternative approach to state machine extraction was implemented by [68].

5.1 A Rule Extraction Algorithm

Since our interest is in "simple" production rules, we describe a heuristic for extracting rules from recurrent networks in the form of DFA's. Different extraction methods are described in [8, 68, 71]. The algorithm we use is based on the observation that the outputs of the recurrent state neurons of a trained network tend to cluster in the neuron activation space (see figure 3). Figure 3 shows the outputs of two-dimensional projections of hidden neuron activations in the (S_i, S_j)-plane for all possible pairs (S_i, S_j) (6 projections) for a well-trained 4-neuron recurrent network. This net was trained on strings from a 4-state DFA and tested on a small test set. The different colors correspond to the different states of the 4-state DFA. If the recurrent network has learned a good representation of the DFA of the training set, then the same colors should cluster (for a hard-threshold logic neuron or gate, the clusters would represent points in the N dimensional neuron space [71].) DFA extraction becomes identifying clusters in the output space $[0, 1]^N$ of all state neurons. We use a dynamical state space exploration which identifies the DFA states and at the same time avoids the computationally infeasible exploration of the entire space.

The extraction algorithm divides the output of each of the N state neurons into q intervals or *quantization levels* of equal size, producing q^N partitions in the space of the hidden state neurons. Starting in a defined initial network state, a string of inputs will cause the trained weights of the network to follow a discrete state trajectory connecting continuous state neuron values. The algorithm presents all strings up to a certain length in alphabetical order starting with length 1. This procedure generates a search tree with the initial state as its root and the number of successors of each node equal to the number of symbols in the input alphabet. Links between nodes correspond to transitions between DFA states. The search is performed in breadth-first order. Paths are made from one partition to another depending on the following: (1) When a previously visited partition is reached, then only the new transition is defined between the previous and the current partition: i.e. no new DFA state is created and the search tree is pruned at that node. (2) When an input causes a transition immediately to the same partition, then a loop is created and the search tree is pruned at that node. The algorithm terminates when no new DFA states are created from the string set initially chosen and all possible transitions from all DFA states have been extracted.

Obviously, the extracted DFA depends on the quantization level q chosen, i.e., in general, different DFA will be extracted for different values of q. Furthermore, different DFA may be extracted depending on the

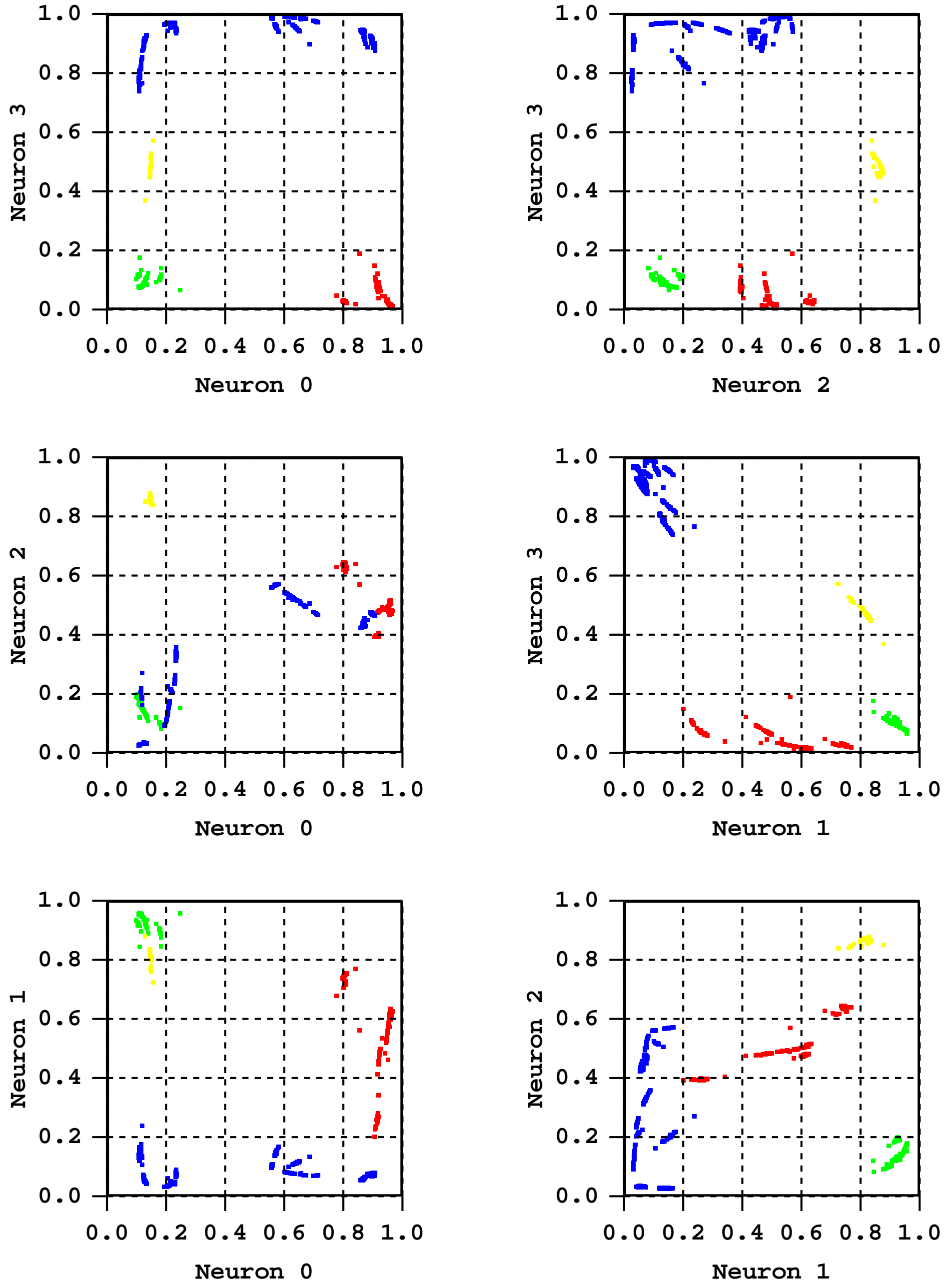

Figure 3: **Clustering of the States of a Known DFA in Hidden Neuron State Space.** A recurrent network with 4 state neurons was trained to accept only strings which do not contain three consecutive 0's. The 4-state DFA which accepts this language is shown in figure 5(b). Each of these states is represented by a different color. The network builds an internal representation of the learned DFA in the space of its hidden neurons. Two-dimensional projections of the hidden neuron state space $[0,1]^4$ into the (S_i, S_j)-plane for all possible pairs (S_i, S_j) are shown as the 1st 1024 strings are fed into the trained network. The colors denote each of the states shown in the DFA in figure 5 (b) and tend to form clusters. These clusters are the trained network's internal representation of the DFA's states. Transitions between clusters correspond to state transitions in the DFA.

order of strings presented which leads to different successors of a node visited by the search tree. Usually these distinctions are not significant because the minimization algorithm [33] guarantees a unique, minimal representation for any extracted DFA. Thus, many different DFA's extracted for different initial conditions, different numbers of neurons, etc. collapse into *equivalence classes* [22]. Finally we must distinguish between accepting and nonaccepting states. If at the end of a string the output of the response neuron S_0 is larger than 0.5, the DFA state is accepting; otherwise, rejecting.

5.2 Example of DFA Extraction

An example of the extraction algorithm is illustrated in figure 4. Assume a recurrent network with 2 state and 2 input neurons is trained on a data set. The range of possible values of S_0 and S_1 can be represented as a unit square in the (S_0, S_1)-plane. For illustration, choose a quantization level $q = 3$, i.e. the activation of each of the two state neurons is divided into 3 equal length intervals, defining $3^2 = 9$ discrete partitions. Each of these partitions corresponds to a hypothetical state in an unknown DFA. Assign labels *1, 2, 3, ...* to the partitions in the order in which they are visited for the first time.

The start state of the to-be-extracted DFA is the initial network state vector used in training - partition *1* in figure 4(a) which is also an accepting state (denoted by a shaded circle) since the output of the response neuron (S_0) is larger than 0.5. On input '0' and '1', the network makes a transition into partitions *1* and *2*, respectively. This causes the creation of a transition to a new accepting DFA state 2 and a transition from state 1 to itself. In the next step, transitions occur from partition *2* into partitions *3* and *4* on input '0' and '1', respectively. The resulting partial DFA is shown in figure 4(b). The DFA in figure 4(c) shows the current knowledge about the DFA after all state transitions from states 3 and 4 have been extracted from the network. In the last step, only one more new state is created (figure 4(d)). When the final string of this string set is seen, the extraction algorithm terminates. Notice that not all partitions have been assigned to DFA states. The algorithm usually only visits a subset of all available partitions for the DFA extraction. Many more partitions are reached when large test sets (especially when they contain many long strings) are used (e.g. when measuring the generalization performance on a large test set). The extracted DFA and its unique, minimized representation are shown in figure 5. The DFA's accept all strings which do not contain three consecutive 0's (notice that both DFA's accept exactly the same regular language).

5.3 Selection of DFA Models

If several DFA's are extracted with different quantization levels q_i, then one or more of the extracted DFA M_{q_i} may be consistent with the given training set, i.e correctly classify the training set. To make a choice between different consistent DFA, we devise a heuristic algorithm [49].

Let M denote the unknown DFA and $L(M)$ the language accepted by M. By choosing a particular quantization level q_i, we extract a minimized finite-state automaton, the ***hypothesis*** M_{q_i} for the grammar to be inferred. A DFA M is defined as *consistent* if it correctly classifies all strings of the training set; otherwise, it is an *inconsistent* model of the unknown source grammar. Given a set of consistent hypotheses M_{q_1}, M_{q_2}, ..., M_{q_Q} we need a criterion for model selection that permits the choice of a hypothesis that best represents the unknown language $L(M)$. A possible heuristic for model selection would be to split a given data set into two disjoint sets (training and testing set), to train the network on the training set and to test the network's generalization performance on the test set. However, by disregarding a subset of the original data set for training, we may be eliminating valuable data from the training set which would improve the network's generalization performance if the entire data set were used for training. However, we wish to make a model selection based solely on simple properties of the extracted DFA's and not resort to a test set. [Keep in mind that all DFA's discussed here will accept strings of arbitrary length.] The model selection algorithm will be motivated by the simulation results and discussed next.

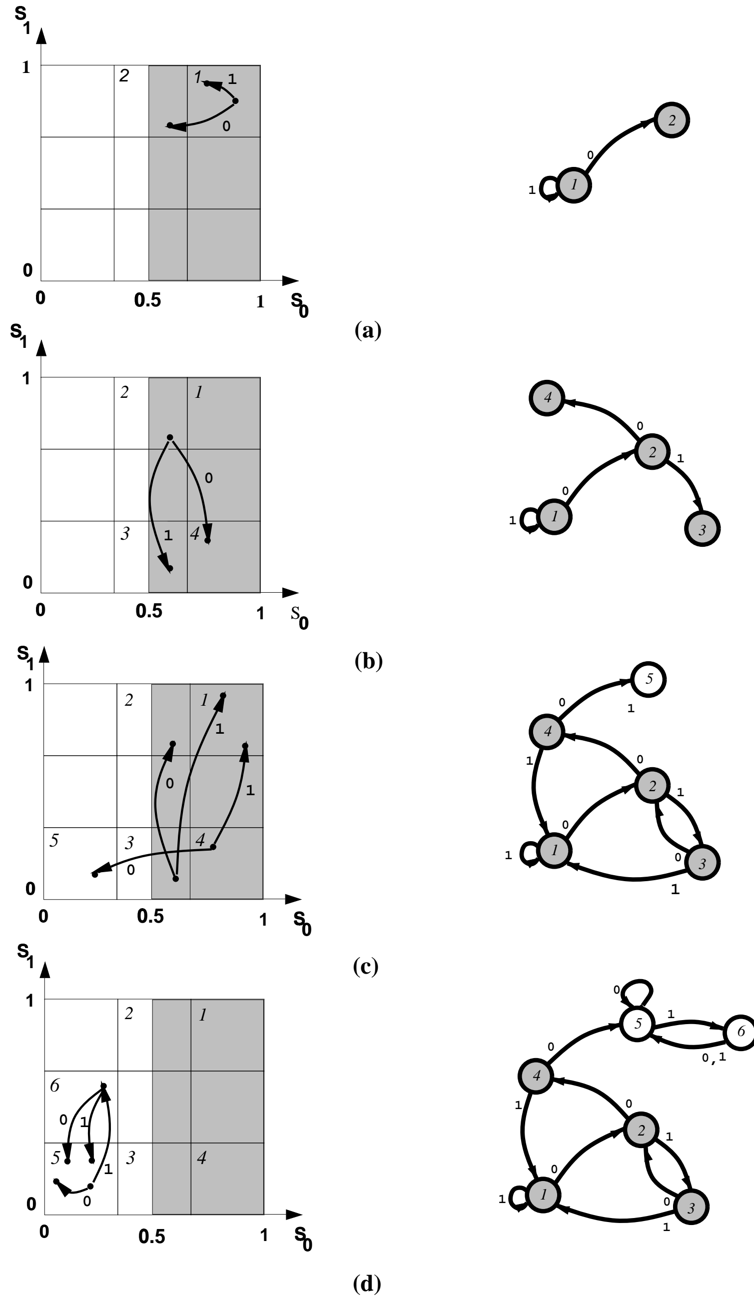

Figure 4: **Example of DFA Extraction Algorithm.** Example of extraction of a DFA from a recurrent network with 2 state neurons. The state space is represented as a unit square in the (S_0, S_1)-plane. The output range of each state neuron has been divided into 3 intervals of equal length resulting in 9 partitions in the networks state space. Shaded states are accepting states. The figures show the transitions performed between partitions and the (partial) extracted DFA at different stages of the extraction algorithm: (a) the initial state 1 and all possible transitions, (b) all transitions from state 2, (c) all transitions from states 3 and 4, and (d) all possible transitions from states 5 and 6.

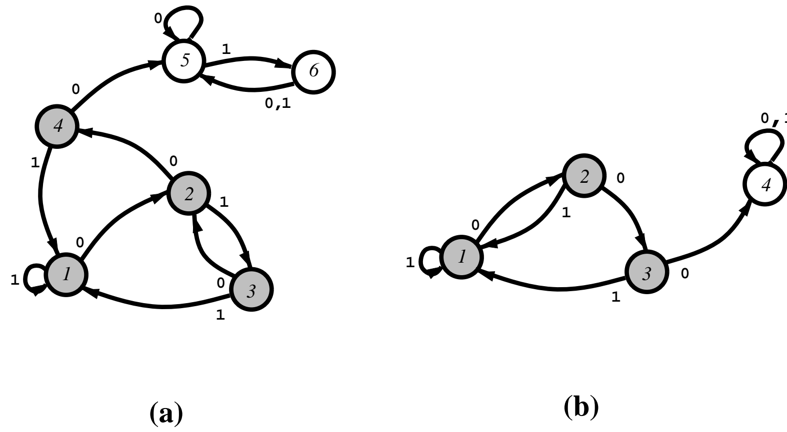

Figure 5: **Minimization of DFA.** (a) extracted DFA and (b) its unique representation obtained through a standard minimization algorithm. Both DFA's accept the same language, consisting of all strings which do not contain the substring '000'.

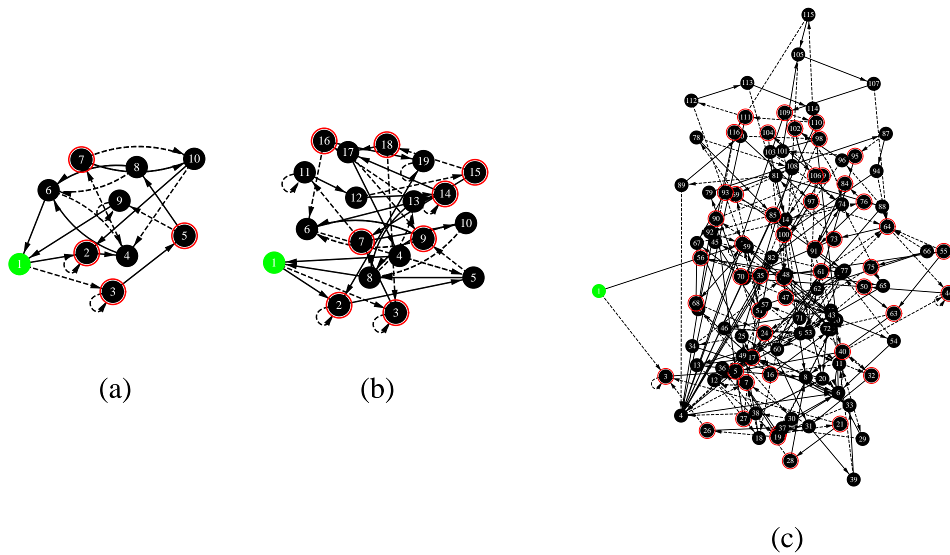

Figure 6: **Examples of Extracted DFA's.** Minimized DFA's extracted from a trained network with quantization levels (a) q=3 (b) q=6 and (c) q=8. All three DFA's are consistent with the training set, i.e. they correctly classify each string in the set. Which DFA best models the unknown regular grammar? We contend DFA(a) because it has the least number of states.

5.4 Simulation Results

An example of model selection and performance is shown in Figure 6. Minimized DFA's are extracted from a trained network for quantization levels $q = 3$, $q = 6$ and $q = 8$. All three DFA's are consistent with the training set, i.e. they correctly classified all strings in the training set. But which DFA best models the unknown source M?

The generalization performance of the network and DFA extracted for $q = 2 \ldots 8$ are shown in figure 7. The generalization performance in percentage of misclassified strings is plotted as a function of the total string length which ranged from 6 to 25. The DFA M_2 is not consistent with the training set and clearly performs worse than the network itself. The poor generalization performance is due to the small number of quantization regions which do not adequately represent the DFA states. The DFA's M_3, M_4, and M_5 are all the same and identical with the DFA shown in figure 6a (the DFA to be learned). These DFA's should make no classification errors on all strings of length up to and including 25 or on any strings at all. The generalization performance of the consistent DFA's M_6, M_7, and M_8 becomes poorer with increasing

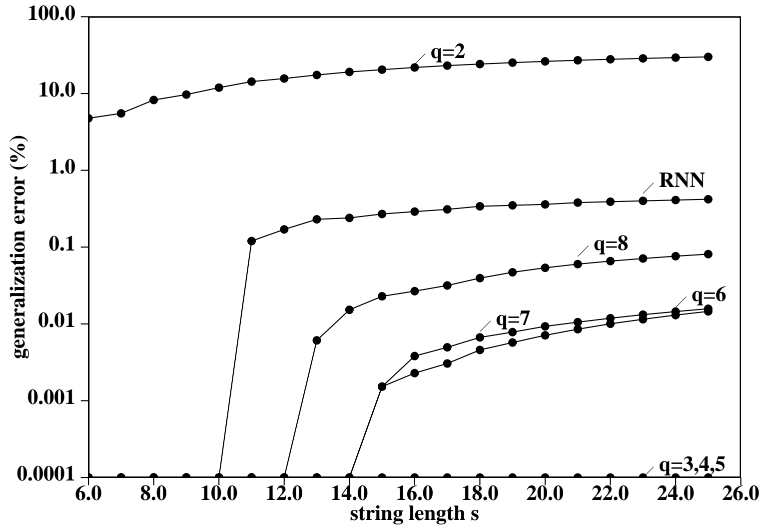

Figure 7: **Good Generalization Performance.** Generalization performance of a well-trained network (trained with 1,000 strings) and of all extracted DFA's with quanitization paramenter $q \leq 8$. The plot shows (on a logarithmic scale) the cumulative generalization performances in percentage of misclassified strings as a function of the string length. The largest test set contained all strings of length up to and including 25. Consistent DFA's $M_3, \ldots, M_8$ outperform the recurrent neural network. DFA M_2 which is inconsistent with the training set performs worse than both the network and all other DFA's shown here. DFA's M_3, M_4, M_5 are the DFA of the grammar to be learned and thus have no generalization error.

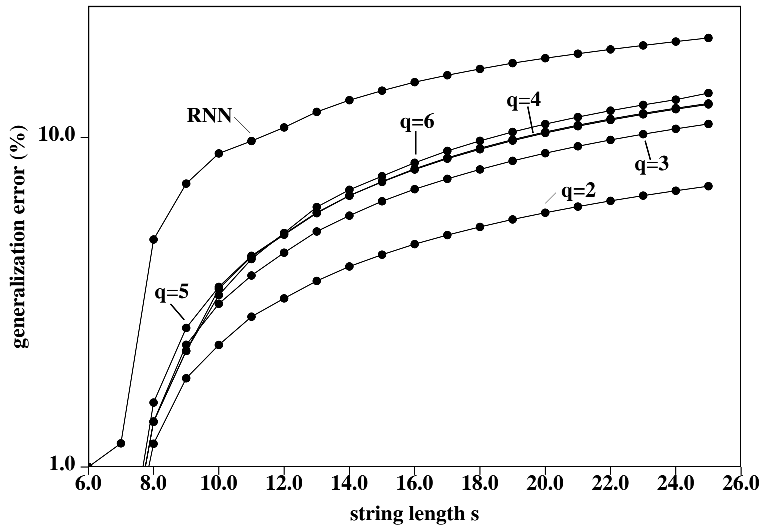

Figure 8: **Poor Generalization Performance.** Generalization performance of a poorly-trained network (trained with 128 strings) and of all extracted DFA's with quanitization paramenter $q \leq 6$. The plot shows (on a logarithmic scale) the cumulative generalization performances in percentage of misclassified strings as a function of the string length. The largest test set contained all strings of length up to and including 25. All of the DFA's were consistent with the small training set but showed poor performance when tested on large data sets. The trained recurrent neural network misclassifies more strings than any of the extracted DFA's.

quantization level q. Note that the generalization error becomes less sensitive as the length of the test strings increases. This suggests that the generalization error reaches a limit as the string length approaches infinity. (We have also observed that the generalization performance can show an asymptotic, oscillatory behavior.)

By the above definition the recurrent network is equivalent to the DFA M_∞. In general consistent DFA's always outperform the trained network and the network in turn outperforms inconsistent DFA's. Figure 7 further suggests that the extracted DFA's asymptotically approach the generalization performance of the trained network. We speculate that the generalization performance of consistent and inconsistent DFA's asymptotically approach the network's performance from below and above, respectively, with increasing quantization level q. The graphs in figure 8 show the generalization performance of the network and extracted DFA's for a *poorly trained* network. We trained the same network on only the first 128 of the original training set of 1,000 strings. The generalization performance shows the same general behavior as in figure 7 with the exception that no inconsistent DFA's were extracted and that both the network and the extracted DFA's make classification errors on more strings than before. This is not surprising since we used a much smaller training set.

From the simulations a policy for selecting a "good" model of the unknown source grammar can be formulated. We choose the DFA M_s which is the smallest consistent DFA. M_s can be found by extracting DFA's $M_2, M_3, \ldots, M_{s-1}, M_s, M_{s+1}, \ldots, M_Q$ in that order and the best model is the first consistent DFA M_s. We hypothesize that there always exists an s such that M_s is the smallest consistent DFA. While we have no proof for this claim, we believe that large DFA's tend to overfit the given training data set and thus yield poorer generalization performance. This is an example of *Occam's Razor - complex models should not be preferred over simple models that explain the phenomena equally well.* As might be expected the network generalization performance improved as the size of the training set increased.

6 RULE INSERTION

6.1 Introduction

The importance of using prior knowledge in a learning problem has been noted by many. [43] state that *"... significant learning at significant success rate presupposes some significant prior structure. Simple learning schemes based on adjusting coefficients can indeed be practical and valuable when the partial functions are reasonably matched on the task ..."*. More recently, [20] have investigated the strengths and weaknesses of neural learning from a statistical viewpoint. In formulating the bias/variance dilemma, they conclude that *"... important properties must be built-in or hard-wired, perhaps to be tuned later by experience, but not learned in any statistical meaningful way".* Recently, the use of prior knowledge about a learning task to be solved with a neural network has been studied by several authors.

Inserting *a priori* knowledge has been shown useful in training feed-forward neural networks (e.g. see [1, 6, 21, 58, 64, 67]). The resulting networks usually performed better than networks that were trained without a priori knowledge. In the context of training feed-forward networks, it has been pointed out by [1] that using partial information about the implementation of a function f which uses input-output examples may be valuable to the learning process in two ways: it may reduce the number of functions that are candidates for f and it may reduce the number of steps needed to find the implementation. In related work, [2] has trained feed-forward networks using hints, thus improving the learning time and the generalization performance; and [66] has shown how approximate rules about a domain are translated into a feed-forward network.

Our focus has been on methods for inserting prior knowledge into dynamically-driven *recurrent* neural networks [9, 15, 17, 24, 39, 48, 60]. The work by [15] inserts rules into first-order recurrent networks by solving

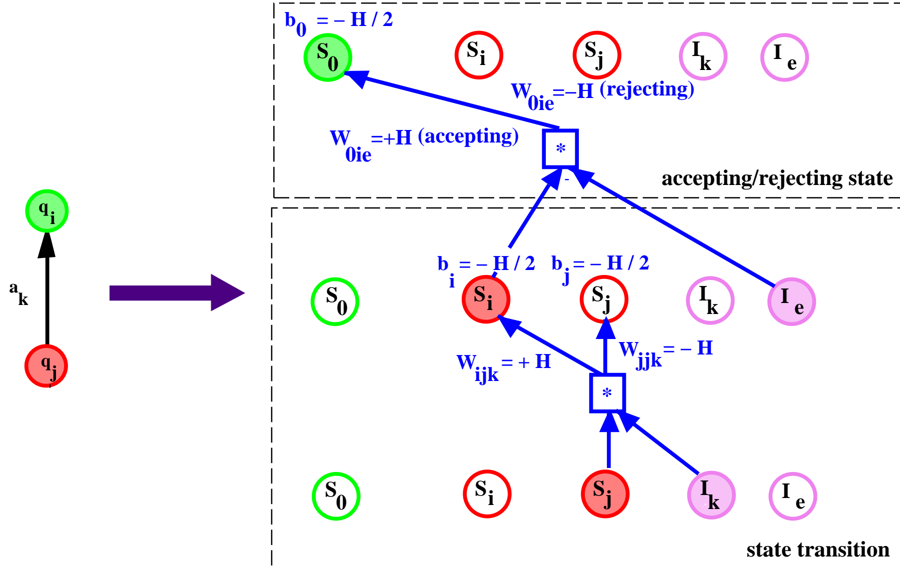

Figure 9: **Rule Insertion Algorithm.** A known DFA transition $\delta(q_j, a_k) = q_i$ is programmed into a network. A special end symbol is used to mark the end of each training string. The rule insertion algorithm consists of two parts: pre-programming the network's state transition and the output values of the output neuron. Neurons S_j and S_i correspond to DFA states q_j and q_i. I_k denotes the input neuron for symbol a_k and the input neuron I_e is assigned to the end symbol. Programming the weights W_{jjk}, W_{ijk} and biases b_i and b_j as shown in the figure ensures a nearly orthonormal internal representation of DFA states q_j and q_i (illustrated by shaded state neurons). The value of the weight W_{0ie} connected to the response neuron S_0 depends on whether DFA state q_i is an accepting or a rejecting state. H denotes the rule (or hint) strength.

a linear programming problem for the weights. The theoretical foundation for this approach is discussed in [17]. However, [28] has shown that there exist simple deterministic finite-state automata which cannot be represented with a first-order, single-layer fully recurrent network architecture unless additional layers of weights (or an end symbol) are added. Giving the recurrent network helpful hints about the strings, such as too long, etc., has also been shown to help learning [9]. It is also useful to put rules directly into the sample strings themselves [39].

6.2 Rule Insertion Algorithm

We present a method [24, 48] that incorporates a priori knowledge in the training of recurrent neural networks. This a priori knowledge can be interpreted as hints about the problem to be learned. These hints are encoded as rules which are then inserted *directly* before training into the *recurrence* of the neural network. We demonstrate the approach by training recurrent neural networks with *inserted rules* to learn to recognize regular languages from grammatical string examples. Because the recurrent networks have second-order connections, rule-insertion is a straightforward mapping of rules into weights and neurons. Our simulations show that training recurrent networks with different amounts of partial knowledge to recognize simple grammars usually improves the training time up to orders of magnitude, even when only a small fraction of all transitions are inserted as rules. In addition there appears to be no loss in generalization performance.

The *rule-insertion method* follows directly from the similarity of state transitions in a DFA to the dynamics of a recurrent neural network. Recall the previous definition of a DFA M: a quintuple $M = < \Sigma, Q, R, F, \delta >$ with alphabet $\Sigma = \{a_1, \ldots, a_k\}$, states $Q = \{s_1, \ldots, s_{N_s}\}$, a start state R, a set $F \epsilon Q$ of accepting states and state transitions $\delta : Q \times \Sigma \rightarrow Q$. We insert rules for *known* transitions by programming some of the initial weights of a second-order recurrent network with N state neurons. The rule insertion algorithm assumes $N > N_s$; the number of neurons is greater than the number of states in the DFA. Obviously, this does not use the full representational range of the neural network; the encoding could be more dense. Thus, one

should at least start with more neurons than known DFA state transitions. (Our initial investigations into the case where $N < N_s$ were not successful.)

Recall that the network changes its state $\mathbf{S}$ at time $t+1$ according to the equation

$$S_i^{(t+1)} = g(\Xi_i), \qquad \Xi_i \equiv \sum_{j,k} W_{ijk} S_j^{(t)} I_k^{(t)}.$$

Consider a known transition $\delta(s_j, a_k) = s_i$ (figure 9). We arbitrarily identify DFA states s_j and s_i with state neurons S_j and S_i, respectively. One method of representing this transition is to have state neuron S_i have a high output ≈ 1 and state neuron S_j have a low output ≈ 0 after the input symbol a_k has entered the network via input neuron I_k. One implementation is to adjust the weights W_{jjk} and W_{ijk} accordingly: setting W_{ijk} to a large positive value will ensure that S_i^{t+1} will be high and setting W_{jjk} to a large negative value will guarantee that the output S_j^{t+1} will be low. All other weights are set to small random values. In addition to the encoding of the known DFA states, we also need to program the response neuron, indicating whether or not a DFA state is an accepting state. A special end symbol e marks the end of each string. We program the weight W_{0ie} as follows: if state s_i is an accepting state, then we set the weight W_{0ie} to a large positive value; otherwise, we will initialize the weight W_{0ie} to a large negative value. If we do not know whether a state s_i is an accepting state, then the weight W_{0ie} is not modified. (The end symbol is not a crucial component of the rule insertion algorithm, it just improves the convergence time by giving the network an extra layer with which to make a classification.) We define the values for the programmed weights as a rational number H, and set the *programmed* weight values to be $+H$ and $-H$ accordingly. We will refer to H as the *strength* of a rule. The above procedure is descriptively shown in figure 9.

The value of the biases b_i of state neurons assigned to known DFA states is set to $-H/2$ ensuring that all state neurons which do not correspond to the previous or current DFA state have a low output. Thus, the rule insertion algorithm defines an approximately *orthonormal internal representation* of all known DFA states. For the initial state of the network, we arbitrarily select the initial state neuron values. When all known rule transitions have been inserted into the network by encoding the weights according to the above scheme, the network is trained. Notice that all weights including the ones that have been programmed are *still adaptable* - they are not fixed.

This method also permits second-order recurrent networks to constructively implement entirely known DFA's (i.e. all states and all transitions are specified). Our experimental results indicate that such a constructed network behaves exactly like the DFA and classifies correctly very long input strings.

6.3 Variety of Inserted Rules

Rules (or DFA's) to be inserted prior to training are shown in figures 10 and 13. The shaded state is the start state, accepting states have an additional circle and transitions for input symbols '0' and '1' are shown respectively as solid and dashed arcs between states. (For figures 10(b-f) the numbers on the states indicate the states present in the DFA of figure 10(a). For figure 10 (g) only states 1,7,10,8,2 are the same as in (a). For the others the numbers just indicate the number of states.) The weights were initialized to random values in the interval [-0.1,0.1] except for the *rule weights* which were programmed to $+H$ or $-H$. Assuming that the state neuron S_1 corresponds to DFA state 1, the output of state neuron S_1^0 was initialized to 1 and the output of all other state neurons to 0.

The inserted rules shown in figures 10 and 13 represent a range of prior knowledge about the problem to be solved. For a baseline comparison, figure 10(a) represents the entire rule set. Rules shown in figures 10(b) and (c-f) represent respectively no knowledge of self-loops and partial knowledge of complete segments of the DFA. Figures 10(g-h) represent strings that the DFA accepts but without knowledge of start or accepting states. In this sense these rules can be considered "incorrect." Figures 13(i-m) are very incorrect rules which we term "malicious."

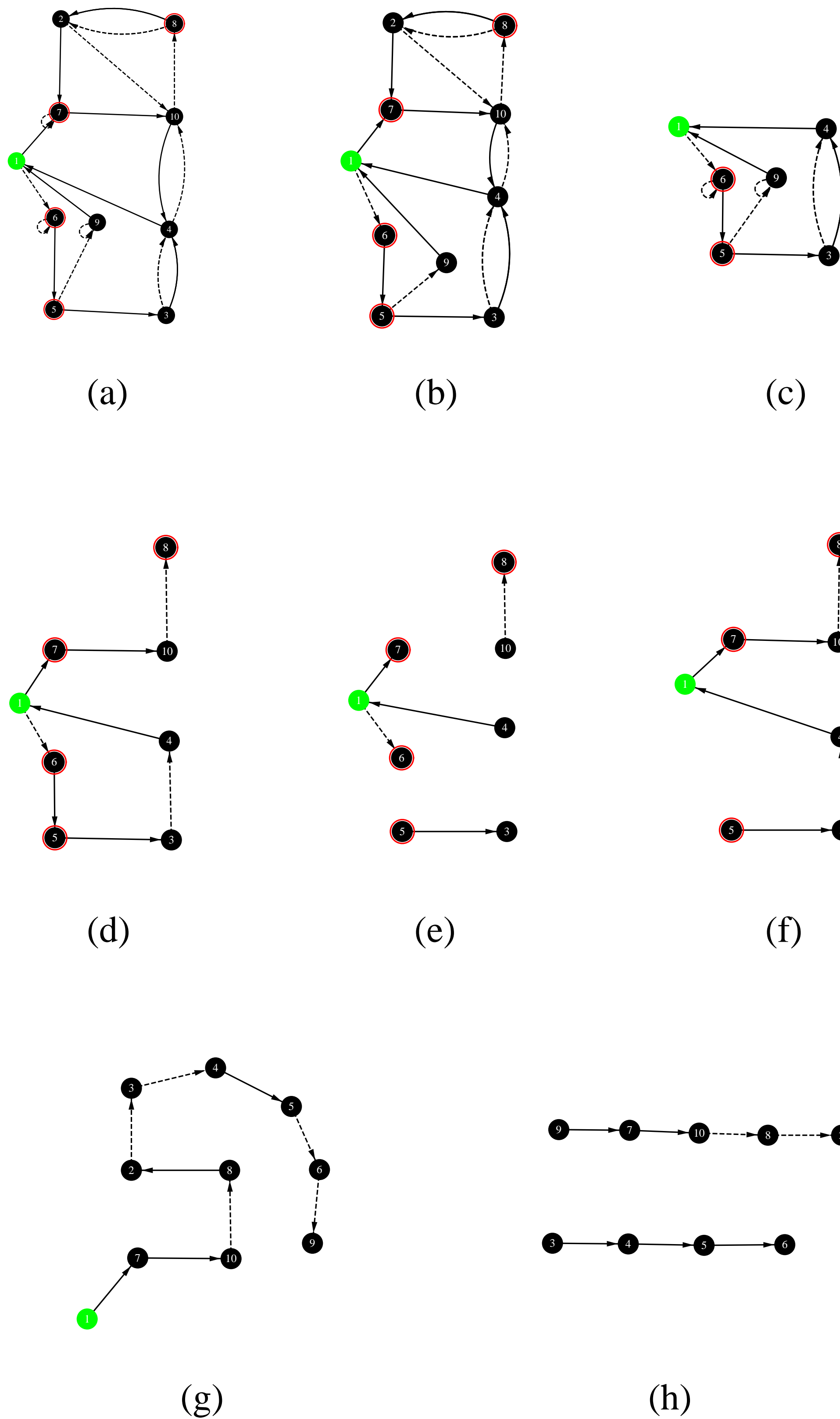

Figure 10: **Partial Rules Inserted into Networks.** Shown are the rules inserted into the recurrent neural network before training. State 1 is the start state. Accepting states are drawn with double circles. State transitions on input symbols '0' and '1' are shown respectively as solid and dashed arcs. The figures show: (a) all rules (entire DFA), (b) all rules except self-loops, (c) partial DFA, (d) rules for string '(10010)*001', (e) rules for disjointed transitions, (f) rules that do not start with a start state, (g) rules for string '001011011' without programming loop, (h) rules for separate strings '000' and '0011'

6.4 Simulations: Training Times and Generalization

For the inserted rules shown in figure 10, training times for runs that converged are plotted on a log scale as a function of integer values of rule strength H in figure 11. The training time without any rules, $H = 0$, is 659 epochs. (The connected lines are connected points for integer values of H; H was not continuously varied.) The training times are also listed in table 1. The absence of connected lines indicates when the network failed to converge after 5,000 epochs. Generalization error for the trained networks (when measured) usually varied between 1 and 10 percent.

Table 1: **Training with Rules:** The training times and the size of extracted DFA's shown for different rules

Rule #	time/DFA size	H=1	H=2	H=3	H=4	H=5	H=6	H=7
a	time	149	86	56	33	10	5	1
	DFA size	10	10	10	10	10	10	10
b	time	166	112	70	40	29	25	60
b	DFA size	10	10	10	10	10	10	10
c	time	361	312	608	513	677	2447	412
	DFA size	10	10	**139**	**207**	10	**50**	10
d	time	186	131	102	102	99	106	144
	DFA size	10	10	10	10	10	**167**	10
e	time	226	383	222	216	434	484	1803
	DFA size	10	10	10	10	**174**	**125**	10
f	time	285	208	434	551	557	>5000	>5000
f	DFA size	10	10	10	10	**122**	-	-
g	time	404	228	235	264	534	>5000	1580
g	DFA size	10	10	10	10	**208**	-	**103**
h	time	454	221	339	4032	>5000	1434	>5000
h	DFA size	10	10	**76**	10	-	**64**	-
i	time	284	526	792	>5000	>5000	>5000	>5000
i	DFA size	10	**16**	10	-	-	-	-
j	time	327	321	>5000	>5000	>5000	>5000	>5000
j	DFA size	10	**48**	-	-	-	-	-
k	time	345	>5000	604	>5000	>5000	>5000	>5000
k	DFA size	10	-	**110**	-	-	-	-
l	time	245	340	>5000	>5000	>5000	>5000	>5000
l	DFA size	10	**30**	-	-	-	-	-
m	time	173	176	305	>5000	>5000	>5000	>5000
m	DFA size	**235**	10	10	-	-	-	-

inserted into recurrent networks with 11 state neurons. For $H = 0$, the training time for convergence was 659 epochs. For each rule (cf. figures 10 and 13), there exists a rule strength H such that the ideal 10-state DFA can be extracted from a trained network. DFA's were extracted with the smallest quantization level such that the DFA was consistent with the training set. In some cases, the consistent DFA (indicated by bold numbers) was not identical with the ideal DFA. For some values of H, networks failed to converge within 5,000 epochs.

The learning curve 11(a) is for all rules inserted and decreases monotonically with increasing rule strength H. Though not shown, the network did not need any training for rule strengths $H \geq 7$. For smaller values of H, the training occurred because of the small error tolerance ($\epsilon = 0.2$) and the sigmoid discriminant function. When all the rules are inserted, it is often not necessary to train at all. For $H \geq 7$, without training the network classifies correctly all strings up to length 16 (65K strings). For $H \leq 6$ training significantly

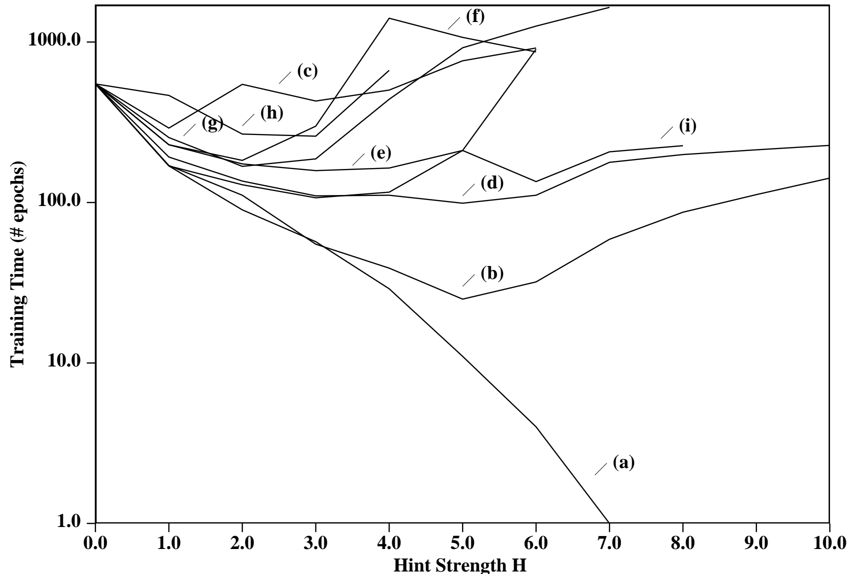

Figure 11: **Training with Known Rules**: Training times (number of epochs) for inserted rules shown in figure 10 plotted on a log scale as a function of the rule strength H. The training time without rules, $H = 0$, is 659 epochs. The plots are only for "integer" values of H and are connected only for clarity. The actual H values are given in table 1. If a network failed to converge within 5,000 epochs, no training times are shown. The letters (a)-(i) are for inserted rules shown in figure 10.

improved the classification of the 65K strings, especially as H decreases.

The effect of leaving out self-loops, plot (b), causes the training time to decrease with increasing rule strength up to $H = 6$ and then start to increase. The fastest training time is more than an order of magnitude faster than that for training without rules. The learning plot 11(d) illustrates that partial rules can improve the training time for proper choice of the rule strength but, for larger rule strength values, the training time starts to increase. This contradicts the intuition that learning time might always decrease with increasing rule strength. For partial knowledge the rule from figure 10(d) performs quite well. The training plots (e,f,g) exhibit improved training times over learning without rules. For most H values, these inserted rules surprisingly outperform the partial rule set of figure 10 (c). One explanation is that these rules have direct access to state number 8, the state the greatest distance from the start state. Figures 10 (g-h) represent encodings of strings that are not entirely correct, but they can still improve training times over no rules at all. The training times in table 1 for the malicious rules shown in figure 13 indicates that even malicious rules can significantly reduce training time when they converge, but not as well as correct rules. Were it not for plots (a), (b) and (d), one might conclude that just inserting strong rules was the only common denominator in reducing training times.

7 RULE CHECKING

Checking previously encoded a priori knowledge in a training system is an important task. What if the incorrect rules are inserted; how can they be checked and then corrected? The problem of changing incorrect rules has been addressed for rule-based systems [26, 52, 50]. We demonstrate that recurrent networks can be used successfully as *rule checkers*; that is, once rules have been inserted into the network, they can be verified and corrected.

We distinguish three different kinds of rule-based training: I). Training with (partial) correct rules inserted before training. (Here, we are concerned with whether or not these correct rules are preserved in the grammatical rules extracted from trained networks.) II). Training with partial information, which is incomplete,

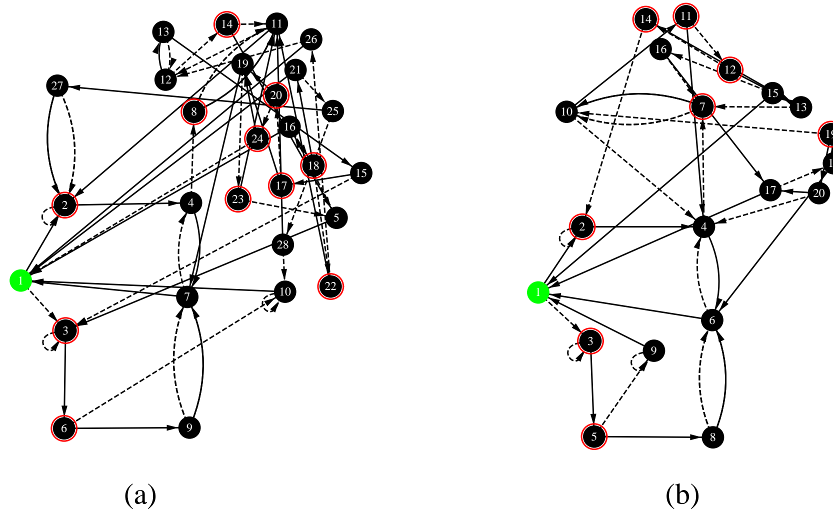

Figure 12: **Rule Preservation and Correction.** Extracted DFA's are not identical to the DFA that generated the training set but are consistent with all the training strings. Figure (a) is the consistent extracted DFA when the rule in figure 10(c) (partial DFA) was inserted into the network before training. The correct rules are preserved after training. Figure (b) is the consistent extracted DFA when the rule in figure 10(g) (no loops) was inserted before training. The network recognized that the rules needed to be part of the loop in the DFA and corrected the wrong prior rules.

but which contains some useful information. III). Training with rules which have no resemblance to the actual rules of the grammar to be learned.

Rule checking consists of these stages: 1) inserting all the available prior knowledge by pre-programming that knowledge into the weights of a network; 2) training the network on the data set; 3) checking the inserted rules by extracting rules in the form of DFA's from the trained network; 4) comparing the rules in the extracted DFA with the initial prior knowledge. We previously discussed a method for extracting rules (DFA's) from a trained or training recurrent network. We say a network preserves a known rule if it appears in the inferred grammar (DFA). The network is permitted to change the programmed weights. In order for a network to be a good rule checker, it must preserve previously inserted *genuine or correct* initial rules and correct *incorrect* initial rules. Our simulation results show that recurrent networks are able to correct "incorrect" prior information and to preserve genuine prior knowledge [47]. Thus, they meet the criteria of good rule checkers.

In table 1 we see the minimal DFA size for all inserted rules in figures 10 and 13. The DFA's were extracted with the smallest quantization level such that the DFA was consistent with the training data. In some cases, the consistent DFA was not identical with the ideal DFA.

7.1 Genuine Rules

We first performed rule checking on correct but incomplete prior knowledge. We inserted rules according to figures 10(b-f) and compared the inserted rules with the rules extracted from the trained network. In all cases where the training converged, the correct inserted rules were also extracted from the trained network. In most cases (see table 1) the ideal 10-state DFA (figure 10(a)) was extracted from the trained network. However, the networks did not necessarily develop an internal representation of the ideal DFA (see for example the bold number of states in table 1). The DFA shown in figure 12(a) is an example (not shown in the table) where the extracted DFA has many more states, but still correctly classifies all strings of the training set. However, the *inserted rules* were preserved.

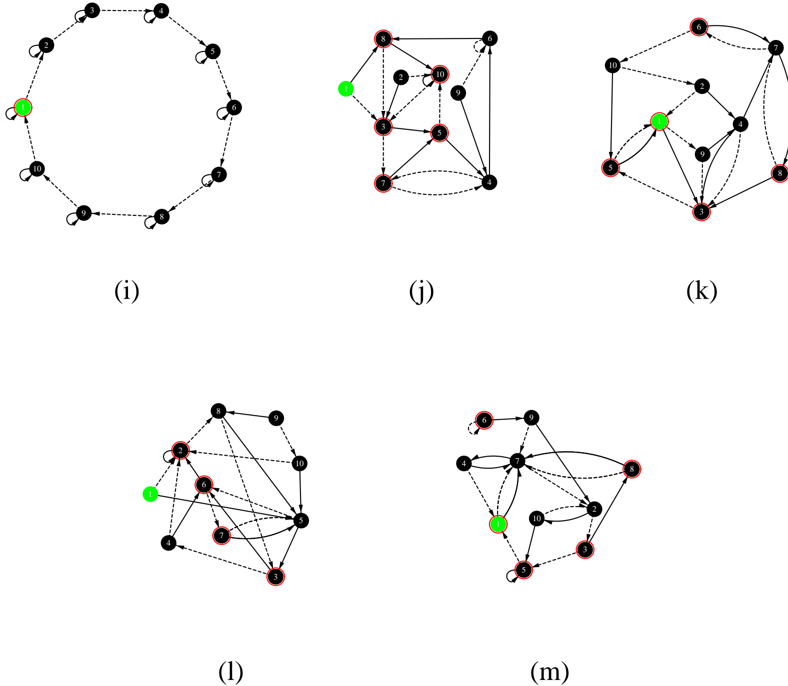

Figure 13: **Malicious Hints**: Rule (i): DFA accepting all strings where the number of 1's is a multiple of 10. Rules (j-m): Randomly generated DFA's with 10 states.

7.2 Incorrect Rules

We define incorrect rules as rules which correctly represent some aspects of the rules of the DFA, but which contain some error in the way the rules are represented. Often, strings visit states several times when a DFA has loops. Suppose we are given a string but we do not know that there is a loop. Is the network able to detect and correct that error? We inserted rules for the string '001011011' where the transitions on substring '101' form a loop in the DFA, figure 10(g). We programmed the weights of the network as if there were no such loop, i.e. a new state is reached on every symbol of the string. The training times are shown by plot 11(g). The extracted DFA's of table 1 demonstrate that the network recognized that the inserted rules were incomplete and corrected the error by forming the loop, although a DFA different from the original automaton was extracted for some values of H, figure 12(b) (also not shown in the table).

Many strings of a given training data set share some of the transitions in the corresponding DFA. Two strings obviously share transitions if they have a common prefix and, if the rules for two such strings were known, then the inserted rules would reflect this transition sharing. However, we wanted to test a network's ability to recognize if transitions were shared, figure 10(h). Two separate paths with distinct states for the strings '0011' and '000' were programmed into a network. The network was able to merge the common parts of the paths through the DFA taken by the two strings (see the 10-state extracted DFA's shown for (h) in table 1).

7.3 Malicious Rules

It is difficult to give a precise definition of a malicious rule, because there are many ways in which a rule can convey wrong information - such as wrong number of states, wrong accepting states, and wrong transitions. For this study, we used the language 10-parity and 4 different randomly generated 10-state DFA's as malicious rules, figures 13(i-m).

We would expect rule checking to be difficult for a recurrent neural network in this case because the rules define a *complete internal state representation*, i.e. transitions for every possible input are accounted for. As the rule strength increases, the dynamics of the recurrent network become more rigidly defined which potentially makes the unlearning of the 10-parity DFA even more difficult. The simulation results show,

however, that the network learns the unknown grammars rather easily (plot 11(i) and table 1), and the perfect DFA's can be extracted (table 1) for at least one of the converged runs. However, the runs only converged for a few small values of the rule strength H.

8 CONCLUSIONS

We have demonstrated techniques for extracting and encoding symbolic information in recurrent neural networks trained to recognize strings of a regular language. Knowledge can be extracted from trained networks in the form of deterministic finite-state automata (DFA's). The extracted DFA's show better generalization performance than the networks themselves, especially when long strings are presented as input. This phenomenon raises questions about the ability of long-time dynamics of recurrent networks to preserve state memory, irrespective of the training methodology. The problem of knowledge (i.e. grammatical rules) extraction is portrayed as a dynamical state exploration and cluster analysis in the output space of recurrent state neurons followed by a model selection heuristic algorithm. This heuristic chooses among the possible candidates the extracted DFA that best models the unknown source grammar. The smallest extracted DFA (number of states) always outperforms larger DFA's extracted from the same network. Our experience tells us that in general better DFA's can be extracted from small networks; i.e. the DFA's extracted from larger networks tend to show a poorer generalization performance. This suggests that we should attempt to train networks of minimal size in order to achieve good grammatical inference. Constructive and destructive training methods which might obtain these minimal networks have recently been proposed by [7, 13, 46].

Prior knowledge about some of the grammatical rules of an otherwise unknown source grammar can be initially encoded into a recurrent network by programming a small subset of the network's weights. These inserted rules can significantly reduce the time needed to train networks. We observed an initial trend for small values of rule strength H, that the larger the number of inserted rules, the faster the convergence time. Of course, this needs to be verified by more experiments. In addition, known rules that give some information about the full depth of the unknown rules also seem to significantly help reduce training time. The generalization performances of networks trained with and without prior knowledge are comparable. Our rule insertion method makes use of second-order weights (in contrast to the first-order methods of [15, 60]). Our method requires that the size of the network be larger than the number of DFA states. As yet, we have no explanation for the increase in training time for large values of the rule strength. Intuitively, we expect the training time to decrease with increasing rule strength. Apparently, the network does not easily recognize when partial correct rules are inserted if the rule strength is too large. Further investigation is needed.

Another aspect of symbolic knowledge extraction and insertion is rule checking. By comparing rules extracted from trained networks with prior knowledge, the validity of this knowledge can be established. Our preliminary results show that recurrent networks meet our criterion for good rule checkers, i.e. they preserve genuine knowledge and they correct wrong prior information. As one might expect, rule checking becomes more difficult with increasing rule strength when wrong rules are inserted into a network.

An important issue in training recurrent networks is the choice of the initial architecture (e.g. its size). Training algorithms have been developed which adapt the network architecture during training. Usually, these techniques change the network during training based on some numerical criterion (e.g. the average error does not decrease by further training on the current architecture). Given the capability of symbolic knowledge extraction and insertion, we suggest that the network architecture can be adapted during training with symbolic guidance. In addition symbolic information extracted from an undertrained network might be used to determine the current network architecture.

The insertion and extraction methods previously discussed can be applied to other recurrent network models. For example, recurrent neural networks combined with an external stack can learn context-free languages, and pushdown automata can be "extracted" using a similar dynamic space exploration algorithm [25, 65, 9].

Future work could focus on incorporating different kinds of rules (e.g. knowledge about modularity) into networks and on how well larger network structures could be learned. There are potentially many other applications for our knowledge extraction and insertion in recurrent neural networks [16]. Such heuristics could be used to integrate traditional symbolic AI and connectionist methods, in order to take advantage of the strengths of both paradigms. Finally, there is no restriction that these methods have to be used with only symbolic data. It would be interesting to see if some form of symbolic state-space rules could be inserted, extracted and refined for recurrent networks trained with real-valued data.

9 ACKNOWLEDGEMENTS

We would like to acknowledge useful discussions with E. Baum, D. Chen, H.H. Chen, R. Das, M. Goudreau, P. Hayes, C. Ji, K. Lang, Y.C. Lee, C. Miller, G.Z. Sun, and L. Valiant and useful suggestions made by the reviewers.

References

[1] Y. Abu-Mostafa, "Learning from hints in neural networks," *Journal of Complexity*, vol. 6, p. 192, 1990.

[2] K. Al-Mashouq and I. Reed, "Including hints in training neural nets," *Neural Computation*, vol. 3, no. 3, pp. 418–427, 1991.

[3] R. Allen, "Connectionist language users," *Connection Science*, vol. 2, no. 4, pp. 279–311, 1990.

[4] D. Angluin and C. Smith, "Inductive inference: Theory and methods," *ACM Computing Surveys*, vol. 15, no. 3, pp. 237–269, 1983.

[5] P. Baldi and S. Venkatesh, "Number of stable points for spin-glasses and neural networks of higher orders," *Physical Review Letters*, vol. 58, no. 9, pp. 913–915, 1987.

[6] H. Berenji, "Refinement of approximate reasoning-based controllers by reinforcement learning," in *Machine Learning, Proceedings of the Eighth International International Workshop* (L. Birnbaum and G. Collins, eds.), (San Mateo, CA), p. 475, Morgan Kaufmann Publishers, 1991.

[7] D. Chen, C. Giles, G. Sun, H. Chen, Y. Lee, and M. Goudreau, "Constructive learning of recurrent neural networks," in *1993 IEEE International Conference on Neural Networks*, vol. III, (Piscataway, NJ), IEEE Press, 1993.

[8] A. Cleeremans, D. Servan-Schreiber, and J. McClelland, "Finite state automata and simple recurrent recurrent networks," *Neural Computation*, vol. 1, no. 3, pp. 372–381, 1989.

[9] S. Das, C. Giles, and G. Sun, "Learning context-free grammars: Limitations of a recurrent neural network with an external stack memory," in *Proceedings of The Fourteenth Annual Conference of the Cognitive Science Society*, (San Mateo, CA), pp. 791–795, Morgan Kaufmann Publishers, 1992.

[10] J. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.

[11] J. Elman, "Incremental learning, or the importance of starting small," Tech. Rep. CRL Tech Report 9101, Center for Research in Language, University of California at San Diego, La Jolla, CA, 1991.

[12] S. Fahlman, "Reading the entrails: Understanding what's going on inside a neural net," 1992. Post-conference Workshop - Vail, Colorado.

[13] S. Fahlman, "The recurrent cascade-correlation architecture," in *Advances in Neural Information Processing Systems 3* (R. Lippmann, J. Moody, and D. Touretzky, eds.), (San Mateo, CA), pp. 190–196, Morgan Kaufmann Publishers, 1991.

[14] S. Fahlman, "Representing implicit knowledge," in *Parallel Models of Associative Memory* (G. Hinton and J. Anderson, eds.), ch. 5, pp. 145–159, Hillsdale, N.J.: Lawrence Erlbaum Publishers, 1981.

[15] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "A unified approach for integrating explicit knowledge and learning by example in recurrent networks," in *1991 IEEE INNS International Joint Conference on Neural Networks - Seattle*, vol. 1, (Piscataway, NJ), pp. 811–816, IEEE Press, 1991.

[16] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "Unified integration of explicit rules and learning by example in recurrent networks," *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[17] P. Frasconi, M. Gori, and G. Soda, "Injecting nondeterministic finite state automata into recurrent neural networks," tech. rep., Dipartimento di Sistemi e Informatica, Università di Firenze, Italy, 1992. Submitted.

[18] K. Fu, *Syntactic Pattern Recognition and Applications.* Englewood Cliffs, N.J: Prentice-Hall, 1982.

[19] S. Gallant, *Neural Network Learning and Expert Systems.* Cambridge, MA: MIT Press, 1993.

[20] S. Geman, E. Bienenstock, and R. Dourstat, "Neural networks and the bias/variance dilemma," *Neural Computation*, vol. 4, no. 1, pp. 1–58, 1992.

[21] C. Giles and T. Maxwell, "Learning, invariance, and generalization in high-order neural networks," *Applied Optics*, vol. 26, no. 23, pp. 4972–4978, 1987.

[22] C. Giles, C. Miller, D. Chen, H. Chen, G. Sun, and Y. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Computation*, vol. 4, no. 3, pp. 393–405, 1992.

[23] C. Giles, C. Miller, D. Chen, G. Sun, H. Chen, and Y. Lee, "Extracting and learning an unknown grammar with recurrent neural networks," in *Advances in Neural Information Processing Systems 4* (J. Moody, S. Hanson, and R. Lippmann, eds.), (San Mateo, CA), pp. 317–324, Morgan Kaufmann Publishers, 1992.

[24] C. Giles and C. Omlin, "Inserting rules into recurrent neural networks," in *Neural Networks for Signal Processing II, Proceedings of The 1992 IEEE Workshop* (S. Kung, F. Fallside, J. A. Sorenson, and C. Kamm, eds.), (Piscataway, NJ), pp. 13–22, IEEE Press, 1992.

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

[26] A. Ginsberg, "Theory revision via prior operationalization," in *Proceedings of the Sixth National Conference on Artificial Intelligence*, p. 590, 1988.

[27] E. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, pp. 302–320, 1978.

[28] M. Goudreau, C. Giles, S. Chakradhar, and D. Chen, "First-order vs. second-order single layer recurrent neural networks," *IEEE Transactions on Neural Networks*, 1993. accepted for publication.

[29] S. Hanson and D. Burr, "What connectionist models learn: Learning and representation in connectionist networks," in *Neural Networks: Theory and Applications* (R. Mammone and Y. Zeevi, eds.), pp. 169–208, Boston, Ma.: Academic Press, 1991.

[30] M. Harrison, *Introduction to Formal Language Theory.* Reading, MA: Addison-Wesley Publishing Company, Inc., 1978.

[31] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation.* Redwood City, CA: Addison-Wesley Publishing Company, Inc., 1991.

[32] G. Hinton, "Implementing semantic networks in parallel hardware," in *Parallel Models of Associative Memory* (G. Hinton and J. Anderson, eds.), ch. 6, pp. 161–187, Hillsdale, N.J.: Lawrence Erlbaum Publishers, 1981.

[33] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1979.

[34] B. Horne, D. Hush, and C. Abdallah, "The state space recurrent neural network with application to regular grammatical inference," Tech. Rep. UNM Technical Report No. EECE 92-002, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM, 87131, 1992.

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

[36] K. Lang, "Random dfa's can be approximately learned from sparse uniform examples," in *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, (New York, N.Y.), pp. 45–52, ACM, 1992.

[37] Y. Lee, G. Doolen, H. Chen, G. Sun, T. Maxwell, H. Lee, and C. Giles, "Machine learning using a higher order correlational network," *Physica D*, vol. 22-D, no. 1-3, pp. 276–306, 1986.

[38] S. Lucas and R. Damper, "Syntactic neural networks," *Connection Science*, vol. 2, pp. 199–225, 1990.

[39] R. Maclin and J. Shavlik, "Using knowledge-based neural networks to improve algorithms: Refining the Chou-Fasman algorithm for protein folding," *Machine Learning*, vol. 11, pp. 195–215, 1993.

[40] L. Miclet, "Grammatical inference," in *Syntactic and Structural Pattern Recognition; Theory and Applications* (H. Bunke and A. Sanfeliu, eds.), ch. 9, Singapore: World Scientific, 1990.

[41] C. Miller and C. Giles, "Experimental comparison of the effect of order in recurrent neural networks," *International Journal of Pattern Recognition and Artifical Intelligence*, accepted for publication - 1993.

[42] M. Minsky, *Computation: Finite and Infinite Machines*, ch. 3, pp. 32–66. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1967. Ch: Neural Networks. Automata Made up of Parts.

[43] M. Minsky and S. Papert, *Perceptrons*. Cambridge, MA: MIT Press, 1969.

[44] M. Mozer and J. Bachrach, "Discovering the structure of a reactive environment by exploration," *Neural Computation*, vol. 2, no. 4, pp. 447–457, 1990.

[45] I. Noda and M. Nagao, "A learning method for recurrent networks based on minimization of finite automata," in *Proceedings International Joint Conference on Neural Networks 1992*, vol. I, pp. 27–32, June 1992.

[46] C. Omlin and C. Giles, "Pruning recurrent neural networks for improved generalization performance," Tech. Rep. TR 93-6, Rensselaer Polytechnic Institute, Computer Science, Troy, N.Y., April 1993.

[47] C. Omlin and C. Giles, "Rule checking with recurrent neural networks," *IEEE Transactions on Knowledge and Data Engineering*, 1993. accepted for publication.

[48] C. Omlin and C. Giles, "Training second-order recurrent neural networks using hints," in *Proceedings of the Ninth International Conference on Machine Learning* (D. Sleeman and P. Edwards, eds.), (San Mateo, CA), pp. 363–368, Morgan Kaufmann Publishers, 1992.

[49] C. Omlin, C. Giles, and C. Miller, "Heuristics for the extraction of rules from discrete-time recurrent neural networks," in *Proceedings International Joint Conference on Neural Networks 1992*, vol. I, pp. 33–38, June 1992.

[50] D. Oursten and R. Mooney, "Changing the rules: A comprehensive approach to theory refinement," in *Proceedings of the Eighth National Conference on Artificial Intelligence*, p. 815, 1990.

[51] Y. Pao, *Adaptive Pattern Recognition and Neural Networks.* Reading, MA: Addison-Wesley Publishing Co., Inc., 1989.

[52] M. Pazzani, "Detecting and correcting errors of omission after explanation-based learning," in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, p. 713, 1989.

[53] P. Peretto and J. Niez, "Long term memory storage capacity of multiconnected neural networks," *Biological Cybernetics*, vol. 54, p. 53, 1986.

[54] L. Personnaz, I. Guyon, and G. Dreyfus, "High-order neural networks: Information storage without errors," *Europhysics Letters*, vol. 4, pp. 863–867, 1987.

[55] J. Pollack, "The induction of dynamical recognizers," *Machine Learning*, vol. 7, pp. 227–252, 1991.

[56] J. Pollack, "Recursive distributed representations," *Artificial Intelligence*, vol. 46, pp. 77–105, 1990.

[57] S. Porat and J. Feldman, "Learning automata from ordered examples," *Machine Learning*, vol. 7, no. 2-3, pp. 109–138, 1991.

[58] L. Pratt, "Non-literal transfer of information among inductive learners," in *Neural Networks: Theory and Applications II* (R. Mammone and Y. Zeevi, eds.), Academic Press, 1992.

[59] D. Psaltis, C. Park, and J. Hong, "Higher order associative memories and their optical implementations," *Neural Networks*, vol. 1, pp. 149–163, 1988.

[60] A. Sanfeliu and R. Alquezar, "Understanding neural networks for grammatical inference and recognition," in *Advances in Structural and Syntactic Pattern Recognition* (H. Bunke, ed.), World Scientific, 1992. To appear.

[61] J. W. Shavlik, "A framework of combining symbolic and neural learning," Tech. Rep. TR 1123, Computer Sciences Dept., Computer Sciences Dept, U of Wisconson - Madison, 1992.

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

[63] H. Siegelmann and E. Sontag, "Turing computability with neural nets," *Applied Mathematics Letters*, vol. 4, no. 6, pp. 77–80, 1991.

[64] S. Suddarth and A. Holden, "Symbolic neural systems and the use of hints for developing complex systems," *International Journal of Man-Machine Studies*, vol. 34, pp. 291–311, 1991.

[65] G. Sun, H. Chen, C. Giles, Y. Lee, and D. Chen, "Connectionist pushdown automata that learn context-free grammars," in *Proceedings of the International Joint Conference on Neural Networks 1990* (M. Caudill, ed.), vol. I, (Hillsdale, N.J.), pp. 577–580, Lawerence Erlbaum, 1990.

[66] G. Towell, M. Craven, and J. Shavlik, "Constructive induction using knowledge-based neural networks," in *Eighth International Machine Learning Workshop* (L. Birnbaum and G. Collins, eds.), (San Mateo, CA), p. 213, Morgan Kaufmann Publishers, 1990.

[67] G. Towell, J. Shavlik, and M. Noordewier, "Refinement of approximately correct domain theories by knowledge-based neural networks," in *Proceedings of the Eighth National Conference on Artificial Intelligence*, (San Mateo, CA), p. 861, Morgan Kaufmann Publishers, 1990.

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

[69] R. Watrous and G. Kuhn, "Induction of finite-state languages using second-order recurrent networks," *Neural Computation*, vol. 4, no. 3, p. 406, 1992.

[70] R. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, pp. 270–280, 1989.

[71] Z. Zeng, R. Goodman, and P. Smyth, "Learning finite state machines with self-clustering recurrent networks," *Neural Computation*, 1993. accepted for publication.