Rule Revision with Recurrent Neural Networks*

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

[a]NEC Research Institute, 4 Independence Way, Princeton, New Jersey

[b]Computer Science Department, Rensselaer Polytechnic Institute, Troy, New York

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

Abstract

Recurrent neural networks readily process, recognize and generate temporal sequences. By encoding grammatical strings as temporal sequences, recurrent neural networks can be trained to behave like deterministic sequential finite-state automata. Algorithms have been developed for extracting grammatical rules from trained networks. Using a simple method for inserting prior knowledge (or rules) into recurrent neural networks, we show that recurrent neural networks are able to perform *rule revision*. Rule revision is performed by comparing the inserted rules with the rules in the finite-state automata extracted from trained networks. The results from training a recurrent neural network to recognize a known non-trivial, randomly generated regular grammar show that not only do the networks preserve correct rules but that they are able to correct through training inserted rules which were *initially incorrect*. (By incorrect, we mean that the rules were not the ones in the randomly generated grammar.)

Index Terms: Deterministic Finite-State Automata, Genuine and Incorrect Rules, Knowledge Insertion and Extraction, Recurrent Neural Networks, Regular Languages, Rule Revision.

*Published in *IEEE Trans. on Knowledge and Data Engineering*, vol. 8, no. 1, p. 183, 1996.

1 MOTIVATION

It is important to be able to revise rules in a rule-based system. A natural question is: what if the input data disagrees with the rules; how can the rules be changed? The problem of changing incorrect rules has been addressed for traditional rule-based systems [13, 23, 24]. The context of this work is the use of recurrent networks as tools for acquisition of knowledge which requires hidden-state information. In particular, we are interested in applications where a partial domain theory is available. In cases where this knowledge is incomplete, recurrent networks perform knowledge refinement; in cases where the prior knowledge is inconsistent with some given data, recurrent networks perform rule revision. This work demonstrates that recurrent networks can successfully revise rules, i.e. once rules have been inserted into a network, they can be verified and even be corrected!

Inserting *a priori* knowledge has been shown useful in training feed-forward neural networks; for example see [1, 2, 3, 10, 25, 27, 30, 32]. The resulting networks usually performed better than networks that were trained without a priori knowledge. Recurrent networks are inherently more powerful than feed-forward networks because they are able to dynamically store and use state information indefinitely due to the built-in feedback [29]. In particular, they can be encoded [20, 18] and trained [8, 11, 16, 26, 31, 33] to behave like deterministic, sequential finite-state automata. Methods for inserting prior knowledge into recurrent neural networks have been previously discussed [4, 6, 7, 12, 15, 21]. It has been demonstrated [12, 21] that prior knowledge can significantly reduce the amount of training necessary for a network to correctly classify a training set of temporal sequences.

Our interpretation of rule revision consists of three stages: 1) insert all the available prior knowledge by programming some of the weights of a network (recently this has been shown to be experimentally [20] and theoretically [18] stable); 2) train a network on a data set; 3) compare the rules extracted in the form of a deterministic finite-state automaton (DFA) with the previously inserted rules. The grammatical rules (the grammar) can be easily extracted from the trained neural network [5, 9, 22, 19, 31, 33]. We say a network is preserving a known rule if it appears in the extracted DFA. The network is permitted to change the programmed weights. In order for a network to be a good tool for rule revision, we expect a network to preserve previously inserted *correct* initial rules and to correct *incorrect* initial knowledge. For a testbed, we trained networks to recognize a regular language generated by a random, non-trivial 10-state DFA. We show that, as might be expected, networks are able to correct incorrect prior information and to preserve genuine prior knowledge. Thus, they meet our criteria of good rule revisors.

2 FINITE STATE AUTOMATA & REGULAR GRAMMARS

Regular languages represent the smallest class of formal languages in the Chomsky hierarchy [14]. Regular languages are generated by regular grammars. A regular grammar G is a quadruple $G =< S, N, T, P >$ where S is the start symbol, N and T are non-terminal and terminal symbols, respectively, and P are productions of the form $A \rightarrow a$ or $A \rightarrow aB$ where $A, B \epsilon N$ and $a \epsilon T$. The regular language generated by G is denoted $L(G)$.

Associated with each regular language L is a deterministic finite-state automaton (DFA) M which is an acceptor for the language $L(G)$, i.e. $L(G) = L(M)$. DFA M accepts only strings which are a member of the regular language $L(G)$. Formally, a DFA M is a 5-tuple $M =< \Sigma, Q, R, F, \delta >$ where $\Sigma = \{a_1, \ldots, a_{N_\Sigma}\}$ is the alphabet of the language L, $Q = \{s_1, \ldots, s_{N_Q}\}$ 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 string x has been read by M. Alternatively, a DFA M can also be considered a generator which generates the regular language $L(M)$.

3 RECURRENT NEURAL NETWORK

The recurrent network used to learn regular languages consists of N recurrent neurons S_j with bias b_j, K nonrecurrent input neurons labeled I_k, and $N^2 \times K$ second-order weights labeled W_{ijk} [9, 11, 26, 33]. The complexity of the network only grows as $O(N^2)$ as long as the number of input neurons is small compared to the number of hidden neurons. We refer to the values of the hidden neurons collectively as a state *vector* **S** in the finite N-dimensional space $[0, 1]^N$. A network accepts a time-ordered sequence of inputs and evolves with dynamics defined by the following equations (figure 1):

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

where $g(.)$ is a sigmoid discriminant function. Note that the weights W_{ijk} modify a product of the hidden S_j and input I_k neurons. This form directly represents the state transition diagrams of a state process — $\{input, state\} \Rightarrow \{nextstate\}$ and can be considered as a canonical form of a DFA neural network [17]. The input neurons accept an encoding of one character of a string per discrete time step t. The above equation is then evaluated for each hidden neuron S_i to compute the next state vector **S** of the hidden neurons at the next time step $t + 1$. With unary encoding a separate input neuron is assigned to each character in the alphabet of the relevant language. (A denser input encoding may be used for languages with large alphabets.) The weights W_{ijk} are updated according to a second-order form of the RTRL learning algorithm

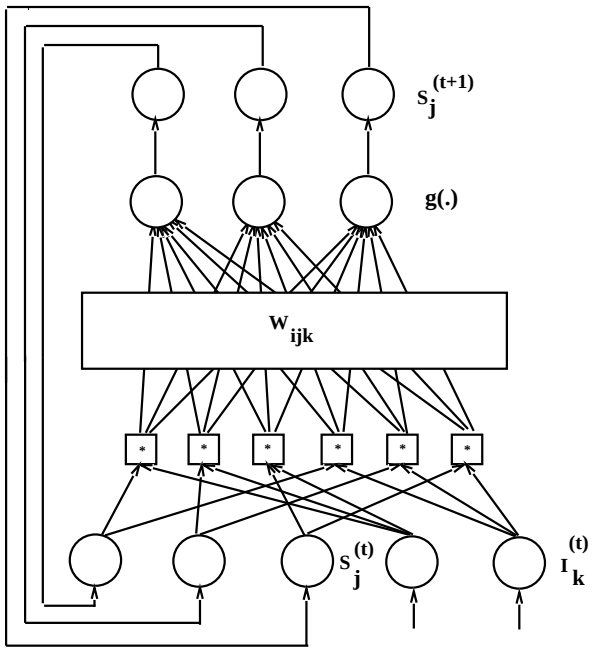

Figure 1: A second-order, single layer recurrent neural network. $S_i^{(t)}$ and $I_k^{(t)}$ represent the values of the i^{th} state and k^{th} input neuron, respectively, at time t. The block marked '*' represent the operation $W_{ijk} \times S_j^{(t)} \times I_k^{(t)}$. $g()$ is the sigmoidal discriminant function.

for recurrent neural networks [34]. A special neuron S_0 is selected as the output neuron of the network; this output neuron is used to define a quadratic error function. Weight changes ΔW_{ijk} occur after the end of each input string; they are proportional to the gradient of the error function with respect to W_{ijk}. For more detail see [11].

4 RULE INSERTION

Recall that a DFA M is a quintuple $M =< \Sigma, Q, R, F, \delta >$ with alphabet $\Sigma = \{a_1, \ldots, a_{N_\Sigma}\}$, states $Q = \{s_1, \ldots, s_{N_Q}\}$, 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 state neurons. Although the number of states N in a DFA is not known a priori, we assume throughout this paper that $N > N_Q$.

Our *rule-insertion method* follows directly from the similarity of state transitions in a DFA and the dynamics of a recurrent neural network. Recall that the network changes it state $\mathbf{S}$ at time $t+1$ according to Eq. (1). Consider a known transition $\delta(s_j, a_k) = s_i$. 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 all other state neurons (including 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 of S_j^{t+1} will be low. Weights which are not *programmed* are initialized with small random values. We set the value of the biases b_i of state neurons that have been assigned to known DFA states to $-H/2$. This ensures that all state neurons which do not correspond to the the previous or the current DFA state have a low output. Thus, the rule insertion algorithm defines a nearly *orthonormal internal representation* of all known DFA states.

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 strings. 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 it is unknown whether or not state s_i is an accepting state, then we do not modify the weight W_{0ie}. (The end symbol is not a crucial component of the rule insertion algorithm, it just improves the convergence time.) We define the values for the programmed weights as a rational number H, and let large *programmed* weight values be $+H$ and small values $-H$. We will refer to H as the *strength* of a rule.

We assume that the DFA generated the example strings starting in its initial state. Therefore, we can arbitrarily select the output of one of the state neurons to be 1 and set the output of all other state neurons initially to zero. When all known transitions have been inserted into the network by programming the weights according to the above scheme, we train the network on a given training set. Notice that all weights including the ones that have been programmed are *still adaptable* - they are not fixed.

5 RULE EXTRACTION

We extract symbolic rules about the learned grammar in the form of DFA's. The extraction algorithm is based on the hypothesis that the outputs of the recurrent state neurons tend to cluster when the network is well-trained and that these clusters correspond to the states of the learned DFA [9, 11]. Thus, rule extraction is reduced to finding clusters in the output space of recurrent state neurons; transitions between clusters correspond to DFA state transitions. We employ a simple state space partitioning algorithm along with pruning heuristics to make the extraction computationally feasible. The DFA extracted from a network depends on a partitioning parameter. A small value for the parameter yields a coarse description of the learned DFA which may not be consistent with the training data; large values of the parameter yield DFA's

with a large number of states that tend to overfit the training data. Thus, we determine the value for the partitioning parameter experimentally as follows: We extract DFA's for increasing values of the partitioning parameter and choose the first DFA that is consistent with the training data as the model of the finite-state source that generated the training set. We have empirical evidence that this model selection policy chooses among all possible DFA's the simplest model which also best describes the unknown finite-state source [22, 19]. (All extracted DFA are minimized in number of DFA states [14].)

6 RULE REVISION

6.1 Random Grammar

In order to explore the rule revising capability of recurrent neural networks, we used a non-trivial, randomly generated DFA with alphabet {0,1}(figure 2a). The networks we trained had 11 states neurons, one neuron for every state of the automaton and an additional output neuron. The training set consisted of 1,000 strings, alternating between positive and negative example strings in alphabetical order. The weights and biases were initialized to random values in the interval [-0.1, 0.1] and some weights were programmed to $+H$ or $-H$ as required by the rule insertion algorithm along with the biases.

We distinguished three different kinds of rules: 1) Rules that have (partial) correct prior knowledge; the networks supplement the missing states and transitions of the DFA through training on the given data set. 2) Rules that have partial incorrect information 3) Rules which have no resemblance with the actual rules of the DFA to be learned. The prior knowledge we used to initialize networks prior to training do not necessarily reflect knowledge found in a particular real-world problem. Different prior knowledge may be available for different applications. Some applications may require larger DFA models or the kind of prior knowledge available may be different altogether. Instead, we view this work as a conceptual study which highlights the advantages and limitations of using recurrent networks as knowledge acquisition tools with emphasis on their capability to revise rules.

6.2 Correct Rules

We first investigated the ability of the recurrent neural networks to supplement incomplete rules by learning from a training data set. To demonstrate the effectiveness of our rule insertion technique, we inserted rules for the entire DFA (figure 2a), i.e. we programmed all the transitions and the accepting DFA states into the network. The convergence times in table 1 (row a) show that the network did not need any training at all

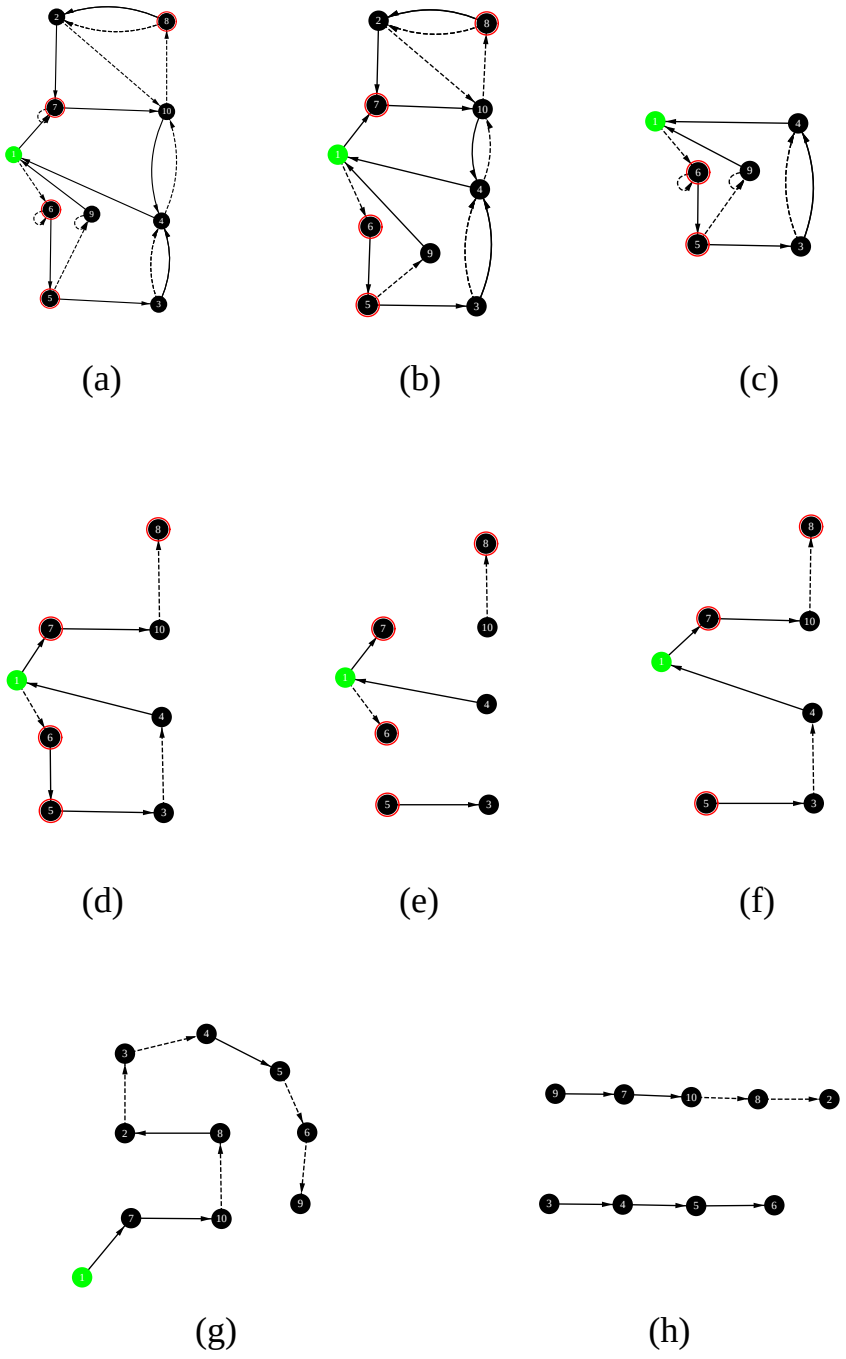

Figure 2: **Partial Rules Inserted into Networks** State 1 is the start state of the (partial) DFA. Accepting states are drawn with double circles. State transitions on input symbols '0' and '1' are shown as solid and dashed arcs, respectively. Insertion of (a) all rules (entire DFA) (b) all rules except self-loops (c) partial DFA (d) rules for string '10010001' (e) rules for string '1(0)0(1)00(0)1' (f) rules for string '(10)010001' (g) rules for string '001011011' without programming loop (h) rules for strings '000' and '0011' but no transitions shared (i) rules for 10-parity (Transitions for symbols in parentheses are not known.)

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

Table 1: **Training with Rules:** The training times and the size of extracted minimal DFA's shown for different rules (figures 2 and 3) inserted into recurrent networks with 11 state neurons. For each rule, there exists a rule strength H such that the ideal 10-state DFA can be extracted from a trained network. For some values of H, networks failed to converge within 5,000 epochs; since the extracted DFA's are not consistent with the training set, no data on DFA size is shown.

for large enough rule strength H. The perfect 10-state DFA was extracted for all values of H. Self-loops, i.e. transitions from a state to itself (figure 2b), are easily learned in recurrent networks (row b of table 1).

In order to demonstrate a network's capability to supplement correct, but incomplete prior knowledge other than the easy case of self-loops, we inserted the rules of a subset of all states and transitions (figure 2c). The network learned the training set and preserved the inserted rules (row c of table 1). For some values of H, the extracted DFA was identical with the original DFA. In general, this was not the case; a network could developed an internal representation of the DFA with more states, but the inserted correct rules were preserved.

Suppose we knew the state transitions for a single string in the training data set. How helpful is that information? We inserted the rules for a single string which visited almost all states of the DFA, but used only a small subset of all transitions (figure 2d). Information about the transitions of a single string can improve the convergence time significantly (row d of table 1).

Even partial information about the transitions occurring for a single string can be of help and the network is able to learn the remainder of the rules from the training data set. We inserted the rules '1(0)0(1)00(0)1' (figure 2e) and '(10)010001' (figure 2f) where parentheses mean that we do not know the transition for that particular symbol. The training times (rows e and f of table 1) support our hypothesis that knowing the transitions from the start state is more helpful than knowing state transitions deeper in the DFA.

6.3 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 the 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 2g). 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 in row g of table 1. The extracted DFA demonstrates that the network recognized that the inserted rules were incorrect and it corrected the error by forming the loop, although a DFA different from the original automaton was extracted for some values of H.

Many strings of a given training data set share some of the transitions in the corresponding DFA. Two

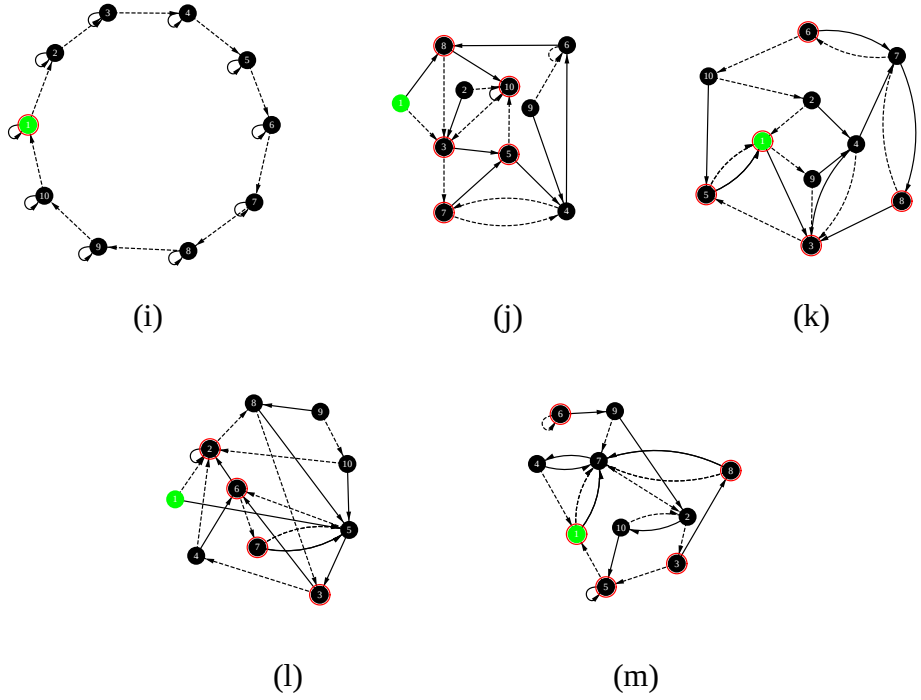

Figure 3: **Malicious Rules:** The following malicious rules were inserted into networks prior to training (a) modulo-10 DFA, (b)-(e) randomly generated minimal DFA's with 10 states.

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 that transitions were shared (figure 2h). 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 (row h of table 1).

6.4 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 incorrect information such as the wrong number of states, wrong accepting states, and wrong transitions. For the purpose of our investigation, we used the language 10-parity (figure 3a) and randomly generated minimal DFA's with 10 states (figure 3b-e). The language of the DFA 3a accepts all strings in which the number of occurrences of the symbol '0' is a multiple of 10. We would have expected rule revision to be difficult for a recurrent neural network in this case as 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 defined more rigidly which potentially makes the unlearning of the 10-parity DFA even more difficult. The simulation results show, however, that the network learns the unknown grammar rather easily for small values of H (row 3a of table 1); networks initialized with larger values for H failed to converge within 5,000 epochs. Similar results can be observed for networks initialized

with randomly generated 10-state DFA's (figures 3b-e); for small values of H they learn the perfect DFA (figure 2a), but they fail to converge to any solution for larger values of H.

7 CONCLUSIONS

We have demonstrated that recurrent neural networks can be applied to rule revision. Given a set of rules about the unknown deterministic finite-state automaton (DFA) and a training data set, networks can be trained on the data set after the partial knowledge has been inserted into the networks. By comparing the rules extracted from the trained networks in the form of a DFA with the prior knowledge, the validity of this knowledge can be established. We tested the networks' ability to revise rules by training them to behave like a non-trivial, random DFA with 10 states. Our simulation results show that recurrent networks meet our criterion for good tools for rule revision, i.e. they preserve genuine *correct* knowledge and they correct *incorrect* prior information. In some simulations, the extracted DFA was identical with the original, randomly generated automaton. In general, however, it is not required that the extracted DFA be identical with the unknown DFA and we consider a recurrent neural network to be good at revising rules as long as genuine rules are preserved and incorrect rules are corrected. Rule revision becomes more and more difficult with increasing rule strength H when incorrect rules are inserted into networks. The results we obtained for rule revision using second-order recurrent neural networks are promising. It is an open question as how this approach will handle larger grammars and more complex rules and how other rule-extraction approaches [31, 33] will compare to this one.

Another open question is the combination of rule insertion and extraction *during* training [28]. By continuously inserting and extracting rules from a network, starting with little or no prior knowledge, the size of the network would change after each rule insertion/network training/rule extraction cycle and could be determined by the current partial knowledge of the DFA, i.e. the extracted symbolic knowledge would control the growth and decay of the network architecture. Furthermore, the symbolic knowledge may substitute for training samples, i.e. the network may select example strings for further training based on the extracted knowledge rather than using all strings for training. Hopefully this symbolically-guided training procedure could lead to faster training and better generalization performance.

8 ACKNOWLEDGMENTS

We would like to acknowledge useful discussions with P.J. Hayes, C. Ji and G.Z. Sun.

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] 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.

[4] 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.

[5] S. Das and M. Mozer, "A unified gradient-descent/clustering architecture for finite state machine induction," in *Advances in Neural Information Processing Systems 6* (J. Cowan, G. Tesauro, and J. Alspector, eds.), pp. 19–26, San Mateo, CA: Morgan Kaufmann, 1994.

[6] 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.

[7] 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*, vol. 7, no. 2, pp. 340–346, 1995.

[8] C. Giles, D. Chen, C. Miller, H. Chen, G. Sun, and Y. Lee, "Second-order recurrent neural networks for grammatical inference," in *1991 IEEE INNS International Joint Conference on Neural Networks*, vol. II, (Piscataway, NJ), pp. 273–281, IEEE Press, 1991.

[9] C. Giles, D. Chen, C. Miller, H. Chen, G. Sun, and Y. Lee, "Second-order recurrent neural networks for grammatical inference," in *1991 IEEE INNS International Joint Conference on Neural Networks*, vol. II, (Piscataway, NJ), pp. 273–281, IEEE Press, 1991. Reprinted in *Artificial Neural Networks*, eds. E. Sanchez-Sinencia, C. Lau, IEEE Press, 1992.

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

[11] 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.

[12] 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.

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

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

[15] R. Maclin and J. Shavlik, "Refining algorithms with knowledge-based neural networks: Improving the chou-fasman algorithm for protein folding," in *Computational Learning Theory and Natural Learning Systems* (S. Hanson, G. Drastal, and R. Rivest, eds.), MIT Press, 1992.

[16] P. Manolios and R. Fanelli, "First order recurrent neural networks and deterministic finite state automata," *Neural Computation*, vol. 6, no. 6, pp. 1154–1172, 1994.

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

[18] C. Omlin and C. Giles, "Constructing deterministic finite-state automata in recurrent neural networks," *Journal of the ACM.* accepted.

[19] C. Omlin and C. Giles, "Extraction of rules from discrete-time recurrent neural networks," *Neural Networks*, vol. 9, no. 1, pp. 41–52, 1996.

[20] C. Omlin and C. Giles, "Stable encoding of large finite-state automata in recurrent neural networks with sigmoid discriminants," *Neural Computation*, vol. 8, no. 4, 1996.

[21] 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.

[22] 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.

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

[24] 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.

[25] S. Perantonis and P. Lisboa, "Translation, rotation, and scale invariant pattern recognition by higher-order neural networks and moment classifiers," *IEEE Transactions on Neural Networks*, vol. 3, no. 2, p. 241, 1992.

[26] J. Pollack, "The induction of dynamical recognizers," *Machine Learning*, vol. 7, no. 2/3, pp. 227–252, 1991.

[27] 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.

[28] J. W. Shavlik, "Combining symbolic and neural learning," *Machine Learning*, vol. 14, no. 3, pp. 321–331, 1994.

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

[30] 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.

[31] P. Tino and J. Sajda, "Learning and extracting initial mealy machines with a modular neural network model," *Neural Computation*, vol. 7, no. 4, pp. 822–844, 1995.

[32] 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.

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

[34] R. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.