Pergamon

Neural Networks, Vol. 10, No. 3, pp. 395–400, 1997
© 1997 Elsevier Science Ltd. All rights reserved
Printed in Great Britain
0893-6080/97 $17.00+.00

PII: S0893-6080(96)00105-0

NEURAL NETWORKS LETTER

On the Computational Power of Recurrent Neural Networks for Structures

ALESSANDRO SPERDUTI

University of Pisa

(*Received* 28 *May* 1996; *accepted* 20 *September* 1996)

Abstract—*Recurrent neural networks can simulate any finite state automata as well as any multi-stack Turing machine. When constraining the network architecture, however, this computational power may no longer hold. For example, recurrent cascade-correlation cannot simulate any finite state automata. Thus, it is important to assess the computational power of a given network architecture, since this characterizes the class of functions which, in principle, can be computed by it. We discuss the computational power of neural networks for structures. Elman-style networks, cascade-correlation networks and neural trees for structures are introduced. We show that Elman-style networks can simulate any frontier-to-root tree automation, while neither cascade-correlation networks nor neural trees can. As a special case of the latter result, we obtain that neural trees for sequences cannot simulate any finite state machine. © 1997 Elsevier Science Ltd. All Rights Reserved.*

Keywords—Neural networks for structures, Recurrent networks, Labelled trees, Tree automata, Computational theory.

1. INTRODUCTION

It is well known that recurrent neural networks can simulate any finite state automata (Alon et al., 1991; Omlin & Giles, 1996) as well as any multi-stack Turing machine in real time (Siegelmann & Sontag, 1995). When constraining the network architecture, however, this computational power may no longer hold. For example, Elman-style recurrent networks can simulate any finite state automata (Goudreau et al., 1994; Kremer, 1995), while recurrent cascade-correlation cannot (Giles et al., 1995; Kremer, 1996a, b). For this reason, it is of paramount importance to assess the computational power of a given network architecture, since this characterizes the class of functions which, in principle, can be computed by such a network. Given an application domain, and based on the observation that the difficulty of training a network is directly proportional to the computational power exhibited by the network, computational results can be used to select the least complex architecture able to deal with the application.

In this paper we study the computational capabilities of recurrent neural networks for structures (i.e., lists, trees, and graphs of variable sizes and complexity, Sperduti, 1994, 1995; Sperduti et al., 1995, 1996). This class of networks are relevant for applications like medical and technical diagnosis, molecular biology and chemistry, automated reasoning. The key idea underpinning these networks is the use of the so called "generalized recursive neuron". A generalized recursive neuron can be understood as a generalization to structures of a recurrent neuron. By using generalized recursive neurons, basically all the supervised networks developed for the classification of sequences, such as back-propagation through time networks, real-time recurrent networks, Elman-style recurrent networks, recurrent cascade correlation networks, and neural trees (NTs) can be generalized to structures. In this paper, we study the computational capabilities of the last three models, relating them to frontier-to-root tree automata (FRA) (Thatcher, 1973; Gonzalez & Thomason, 1978). We show that Elman-style recurrent networks can simulate any FRA, while neither cascade-correlation networks nor NTs can.

In Section 2 we introduce some preliminary concepts on graphs and neural networks. The generalized recursive neuron is defined in Section 3, where some related concepts are discussed. Some neural networks models for processing of structures are presented in Section 4, while tree grammars and automata are introduced in Section 5. Computational results for the models are obtained in Sections 6 and 7. Conclusions are drawn in Section 8.

Requests for reprints should be sent to Alessandro Sperduti, Computer Science Department, University of Pisa, Corso Italia 40, 56125 Pisa, Italy.

2. PRELIMINARIES

We consider finite directed node labeled graphs without multiple edges. For a set of labels Σ, a *graph* X (over Σ) is specified by a finite set V_X of *nodes*, a set E_X of ordered couples of $V_X \times V_X$ (called the set of *edges*) and a function ϕ_X from V_X to Σ (called the *labeling function*). In this paper, the labels are restricted to be binary (0,1), even if for the studied models the labels may also be real-valued vectors. A graph X' (over Σ) is a *subgraph* of X if $\phi_{X'} = \phi_X$, $V_{X'} \subseteq V_X$, and $E_{X'} \subseteq E_X$. For a finite set V, $\#V$ denotes its cardinality. Given a graph X and any node $x \in V_X$, the function $out_degree_X(x)$ returns the number of edges leaving from x, i.e., $out_degree_X(x) = \#\{(x,z)|(x,z) \in E_X \wedge z \in V_X\}$. Given a total order on the edges leaving from x, the node $y = out_X(x,j)$ in V_X is the node pointed by the jth pointer leaving from x. The *valence* of a graph X is defined as $\max_{x \in V_X} \{out_degree_X(x)\}$. A *labeled directed acyclic graph (labeled DAG)* is a graph, as defined above, without loops. A node $s \in V_X$ is called a *supersource* for X if every node in X can be reached by a path starting from s. The root of a tree (which is a special case of directed graph) is always the (unique) *supersource* of the tree. The *frontier* of a tree is the set of nodes (leaves) at the bottom of the tree.

We define a *structured domain* D (over Σ) as any (possibly infinite) set of graphs (over Σ). The *valence of a domain* D is defined as the maximum among the valences of the graphs belonging to D.

The output $o^{(s)}$ of a standard neuron is given by

$$o^{(s)} = f(\sum_i w_i I_i), \quad (1)$$

where $f()$ is some non-linear squashing function applied to the weighted sum of inputs I[1]. When the input vector is binary, it is well known that both the **or** and the **and** operator can be implemented by using the step function and an opportune setting for the weights.

A recurrent neuron with a single self-recurrent connection, instead, computes its output $o^{(r)}(t)$ as follows

$$o^{(r)}(t) = f(\sum_i w_i I_i(t) + w_s o^{(r)}(t-1)), \quad (2)$$

where $f()$ is applied to the weighted sum of inputs (I), plus the self-weight (w_s) times the previous output. The above formula can be extended both considering several interconnected recurrent neurons and delayed versions of the outputs. For the sake of presentation, we skip these extensions.

3. THE FIRST-ORDER GENERALIZED RECURSIVE NEURON

The neural networks recently proposed for the processing of structures are based on *generalized recursive neurons*. A generalized recursive neuron is an extension of the recurrent neuron where instead of just re-using the output of the unit on the previous time step, the outputs of the unit for all the nodes which are pointed by the current input node are considered. Then the output $o^{(c)}(x)$ of the generalized recursive neuron to a node x of a graph X is defined as

$$o^{(c)}(x) = f(\sum_{i=1}^{N_L} w_i l_i + \sum_{j=1}^{out_degree_X(x)} \hat{w}_j o^{(c)}(out_X(x,j))), \quad (3)$$

where N_L is the number of units encoding the label $l = \phi_X(x)$ attached to the current input x, and $\hat{w}_j$ are the weights on the recursive connections. Note that, if the valence of the considered domain is n, then the generalized recursive neuron will have n recursive connections, even if not all of them will be used for computing the output of a node x with $out_degree_X(x) < n$.

When considering k interconnected generalized recursive neurons, eqn (3) becomes

$$o^{(c)}(x) = F(Wl + \sum_{j=1}^{out_degree_X(x)} \hat{W}_j o^{(c)}(out_X(x,j))), \quad (4)$$

where $F_i(v) = f(v_i)$, $l \in \Re^{N_L}$, $W \in \Re^{k \times N_L}$, $o^{(c)}(x) \in \Re^k$, $o^{(c)}(out_X(x,j)) \in \Re^k$, $\hat{W}_j \in \Re^{k \times k}$.

In the following, we will refer to the output of a generalized neuron dropping the upper index.

3.1. Generation of Neural Representations for DAGs

To understand how generalized recursive neurons can generate representations for DAGs, let us consider a single generalized recursive neuron u and a single DAG X. The following conditions must hold:

Number of connections: the generalized recursive neuron u must have as many recursive connections as the valence of the graph X;

Supersource: the graph X must have a reference *supersource*.

Note that, if the graph X does not have a supersource, then it is always possible to define a convention for adding to the graph X an extra node s (with a minimal number of outgoing edges) such that s is a supersource for the new graph.

If the above conditions are satisfied, we can adopt the convention that the graph X is represented by $o(s)$, i.e., the output of u to s. Consequently, due to the recursive nature of eqn (3), it follows that the neural representation for a DAG is computed by a feedforward network (*encoding network*) obtained by replicating the same generalized recursive neuron u and connecting these copies according to the topology of the structure (see Fig. 1). The encoding network fully describes how the representation for the structure is computed.

[1] The threshold of the neuron is included in the weight vector by expanding the input vector with a component always to 1.

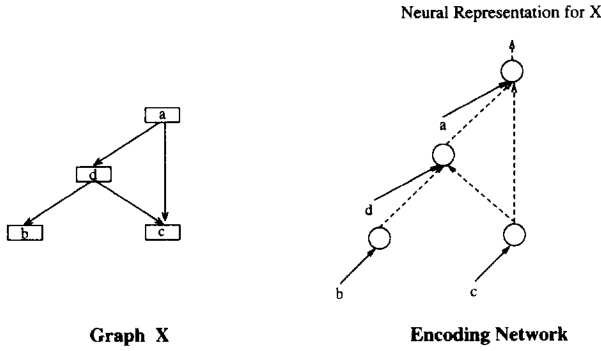

FIGURE 1. A labelled graph *X* and the associated encoding network.

4. NEURAL NETWORKS FOR STRUCTURES

Recently, several neural networks for the processing of structures have been proposed. In this paper, we study the computational capabilities of Elman-style networks (Sperduti et al., 1995), Cascade-Correlation networks for structures (Sperduti et al., 1996), and NTs for structures. Specifically, we restrict to labeled trees and we relate the computational capabilities of the models described below to FRAs.

4.1. Elman-style Networks

An Elman-style network for processing of structures can be obtained by generalizing the Simple Recurrent Network (SRN) proposed by Elman (1990). Essentially, the network has an hidden layer of generalized units and an output layer of standard units. In Fig. 2, the pointer fields are represented explicitly. The label field is subdivided in two parts: one part is used to represent the label, while the second one encodes the *pointer condition bits*, i.e., there is a bit for each pointer field, and whenever the pointer field is void (all pointer units are set to 0), the corresponding bit is set to 1. Otherwise the corresponding bit is set to 0. For more details on the relationship between this network and a related model see (Sperduti et al., 1995).

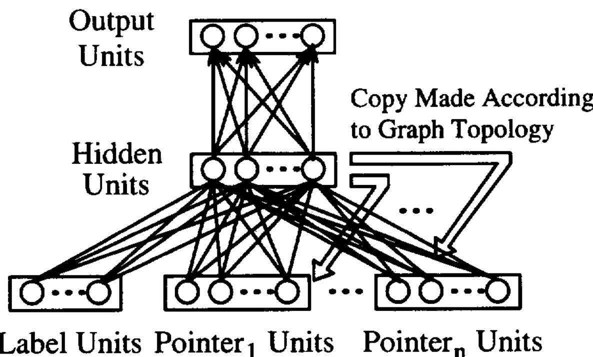

FIGURE 2. The Elman-style network for classification of structures.

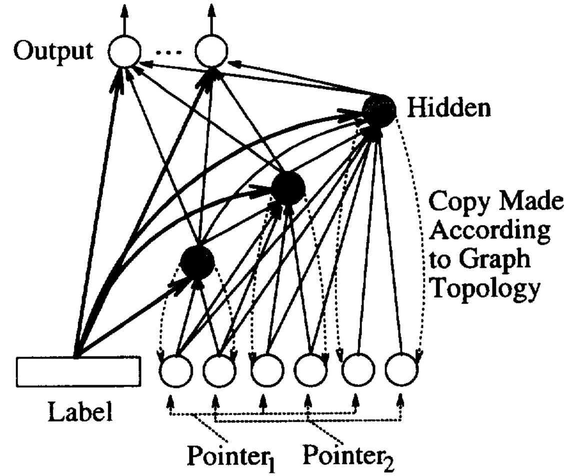

FIGURE 3. The Cascade-Correlation network for classification of structures.

4.2. Cascade-Correlation Networks

The Cascade-Correlation algorithm (Fahlman & Lebiere, 1990) grows a standard neural network using an incremental approach for classification of unstructured patterns. The starting network N_0 is a network without hidden nodes trained with a least mean square algorithm; if network N_0 is not able to solve the problem, a hidden unit u_1 is added such that the *correlation* between the output of the unit and the residual error of network N_0 is maximized[2]. The weights of u_1 are frozen and the remaining weights are re-trained. If the obtained network N_1 cannot solve the problem, the network is further grown, adding new hidden units which are connected (with frozen weights) with all the inputs and previously installed hidden units. The resulting network is a *cascade* of nodes. Fahlman (1991) extended the algorithm to classification of sequences, obtaining good results. Cascade-Correlation can further be extended to structures by using generalized recursive neurons.

The output of the kth hidden unit can be computed as

$$o^{(k)}(x) = f\Big(\sum_{i=1}^{N_L} w_i^{(k)} l_i + \sum_{v=1}^{k} \sum_{j=1}^{out_degree_X(x)} \hat{w}_{(v,j)}^{(k)} o^{(v)}(out_X(x,j)) + \sum_{q=1}^{k-1} \bar{w}_q^{(k)} o^{(q)}(x)\Big), \tag{5}$$

where $\hat{w}_{(v,j)}^{(k)}$ is the weight of the kth hidden unit associated to the output of the vth hidden unit computed on the jth component pointed by x, and $\bar{w}_q^{(k)}$ is the weight of the connection from the qth (frozen) hidden unit, $q < k$, and the kth hidden unit. The output of the output neuron $u^{(out)}$ is computed as

$$o^{(out)}(x) = f\Big(\sum_{i=1}^{k} \tilde{w}_i o^{(i)}(x)\Big), \tag{6}$$

[2] Since the maximization of the correlation is obtained using a gradient ascent technique on a surface with several maxima, a pool of hidden units is trained and the best one selected.

where $\tilde{w}_i$ is the weight on the connection from the ith (frozen) hidden unit to the output unit (see Fig. 3).

4.3. Neural Trees for Structures

NTs have been recently proposed as a fast learning method in classification tasks. They are decision trees (Breiman et al., 1984) where the splitting of the data for each node, i.e., the classification of a pattern according to some features, is performed by a perceptron (Sirat & Nadal, 1990) or a more complex neural network (Sankar & Mammone, 1991). After learning, each node at every level of the tree corresponds to an exclusive subset of the training data and the leaf nodes of the tree completely partition the training set. In the operative mode, the internal nodes route the input pattern to the appropriate leaf node which represents the class of it. The extension of these algorithms to structures is straightforward: the standard discriminator associated to each node of the tree is replaced by a generalized recursive discriminator (with step function). Thus, instead of evaluating an unstructured pattern, each node of the tree evaluates a structure (see Fig. 4).

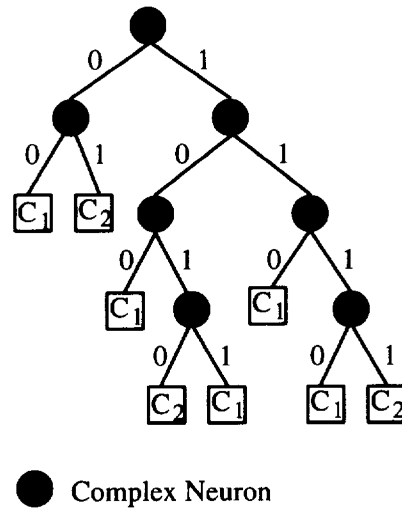

FIGURE 4. Neural tree for classification of structures.

5. TREE GRAMMARS AND TREE AUTOMATA

In this section, we introduce *tree grammars* and *FRA*. FRA will then be used to understand the computational power of the neural models introduced above. A *tree grammar* is defined as a four-tuple $G_t = (V,r,P,S)$ where $V = N \cup \sum$ is the grammar alphabet (nonterminals and terminals); (V,r) a ranked alphabet; productions in P are of the form $T_i \rightarrow T_j$, where T_i and T_j are trees; and S in T_V is a finite set of "starting trees," where T_V denotes the set of trees with nodes labeled by elements in V.

A tree grammar is in *expansive form* if all its productions are of the form

$$X \longrightarrow \overset{x}{\underset{X_1 \quad\cdots\quad X_n}{\diagup \; \diagdown}} \tag{7}$$

A (deterministic) FRA is a system $A_t = (Q,F,\{f_a|a \text{ in } \Sigma\})$ where Σ is a ranked alphabet; Q is a finite set of states; F is a set of final states, a subset of Q; $\{f_a|a$ in $\Sigma\}$ is a set of transition functions $f_a{:}Q^m \rightarrow Q$ such that m is a rank of the symbol $a \in \Sigma$. The recognition process computed by the automation can be described inductively by:

1. The frontier of the state tree is labeled q_0 (the status associated to nodes with $m = 0$, i.e., the leaves).
2. For any node in the input labeled l with rank m, the corresponding node of the state tree is labeled $f_l(q_1,\ldots,q_m)$, where $q_1,\ldots,q_m$ are the states labeling the offsprings of that state tree node.

An input tree is accepted by A_t if the automaton can enter a final state upon encountering the root.

According to the above definition, the *language recognized* by A_t is the set

$T(A_t) = \{T|T$ in T_Σ, A_t can halt in a state in F when the root of T is reached$\}$.

It is not difficult to realize that, given an expansive tree grammar $G_t(V, r, S, P)$ generating the set $L(G_t)$ of trees with nodes labeled with elements in Σ, it is always possible to construct a FRA A_t that recognizes $L(G_t)$. In fact, let $Q = N$ with $F = \{S\}$ and, for each symbol a in Σ, define a transition function f_a such that $f_a(X_1,\ldots,X_n) = X$ *iff* there is in G_t a production

$$X \longrightarrow \overset{a}{\underset{X_1 \quad\cdots\quad X_n}{\diagup \; \diagdown}} \tag{8}$$

When implementing a FRA in one of the neural model described previously, we will represent relations of non maximum rank as relations with maximum rank by introducing a fake state s_0 where needed, i.e., in correspondence to void pointers. Thus, for example, if the maximum rank is 3, the transition function $f_a(s_j)$ becomes $f_a(s_j,s_0,s_0)$.

6. COMPUTATIONAL POWER OF ELMAN-STYLE NETWORKS

Elman-style networks turns out to be powerful enough to simulate any FRA:

THEOREM 6.1. *An Elman-style network can simulate any FRA.*

Proof. It is well known that a sigmoid function can approximate a step function to an arbitrary degree of precision by augmenting the modulus of the associated weight vector. Thus, if we demonstrate that an Elman-style network with step functions can implement any

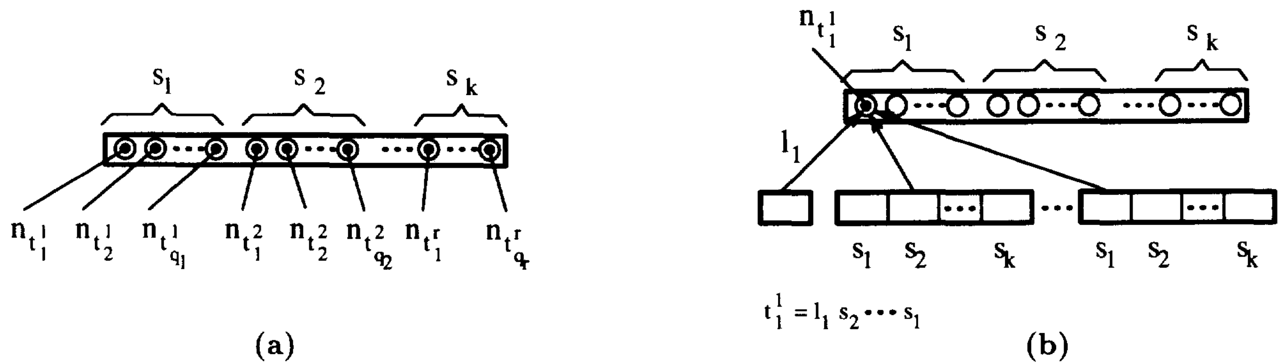

FIGURE 5. (a) The hidden layer of an Elman-style network can be organized in sets of units. Each set represents a state of the FRA, while units in a specific set represent tuples which introduce in the computation the state corresponding to the considered set. (b) Connections for the first hidden units. An arrow indicates a set of connections from a set of context units representing the same state s_j to the hidden unit. The weights of the connections from the label field are set according to the label involved in the augmented tuple represented by the hidden unit.

FRA, the result holds for Elman-style networks with sigmoids as well. In the following, we consider units computing the step function.

The basic idea is to have an hidden unit for each transition function of the FRA. A state is thus represented by the set of units associated to transition functions which introduce the same state in the computation.

Let $V_{s_j} = \{l, s_1', ..., s_n', s_j) | f_l(s_1', ..., s_n') = s_j, l \in \sum\}$ be the set of augmented[3] tuples which yield to the introduction of state s_j. For *each* augmented tuple $t_i^j \in V_{s_j}, i \in [1, ..., \#V_{s_j}]$, and for *each* state s_j, we introduce a unit $n_{t_i^j}$ at the hidden level. The units $n_{t_i^j}$ will represent the state s_j at the hidden level of the network (see Fig. 5a). In order for the construction to work, we require that only a single hidden unit fires at each time step. This can be obtained if each unit $n_{t_i^j}$, such that $t_i^j = (l^{(i)}, s_1^{(i)}, ..., s_n^{(i)}, s_j)$, receives a connection with weight 1 from *every* context unit representing $s_1^{(i)}$ in the first pointer field, from *every* context unit representing $s_2^{(i)}$ in the second pointer field, and so on, up to the units representing $s_n^{(i)}$ in the nth pointer field (see Fig. 5b). If any of the $s_q^{(i)}$ is equal to s_0 (void pointer), then there will only be a *single* connection with weight 1 from the qth pointer condition bit of the label. Moreover, $n_{t_i^j}$ will receive connections from *every* unit in the label field, where the rth connection will have weight 1 if $l_r^{(i)} = 1$, and -1 otherwise. Finally, the threshold of each hidden unit is set to be equal to $(-n_{max} - \#L^{(i)} + 0.5)$, where $L^{(i)} = \{r | l_r^{(i)} = 1\}$ and n_{max} is the rank of the associated label.

Under the hypothesis that at the beginning of the computation, the pointer fields have *at most* a single unit active[4], it is easy to show that only a single hidden unit will be activated. In fact, it is not difficult to verify that only the unit corresponding to the tuple for which the preconditions are satisfied will fire. This is especially true for the leaves of the tree. Thus, at each time step, only a single hidden unit will be active.

The desired output for the Elman-style network can be obtained by observing that a single output unit can compute the **or** on the hidden units representing final states. Moreover, if a specific output $v_t \in \{0, 1\}$ is required for each state transition t, this can be implemented by setting accordingly (i.e., to the value v_t) the weight on the connection from n_t to the output unit.□

7. COMPUTATIONAL POWER OF CASCADE-CORRELATION NETWORKS AND NEURAL TREES

Unfortunately, not all the neural networks for the processing of structures are as powerful as Elman-style networks. In fact, the following theorems states that both cascade-correlation networks and NTs cannot simulate any FRA.

THEOREM 7.1. *A Cascade-Correlation network for structures cannot simulate any FRA.*

Proof. A Cascade-Correlation network for structures is a generalization of a Recurrent Cascade-Correlation network which has been proved unable to simulate any finite state machine (Giles et al., 1995; Kremer, 1996a, b). Since a finite state machine is equivalent to a FRA with relations having rank 1, it follows that a Cascade-Correlation network for structures, which in the finite state machine case reduces to a Recurrent Cascade-Correlation network, cannot simulate any FRA.□

THEOREM 7.2. *A NT for structures cannot simulate any FRA.*

We demonstrate this theorem by showing the following result.

THEOREM 7.3. *Any NT for structures can be implemented by a Cascade-Correlation network for structures.*

Proof. We observe that a NT can be restructured as a layered network by following the rules below:

[3] We include the label component in order to discriminate between transition functions which are identical apart for the involved label.

[4] And *exactly* a single unit active if the corresponding pointer is not void.

- the units in the first layer of the layered network are the same as the internal nodes of the NT;
- all leaf nodes have a corresponding unit in the second hidden layer. Each unit in the second layer implements the **and** of nodes which belong to the path from the root to the considered leaf;
- the output layer will have as many units as the number of different classes in NT. Each output unit will implement the **or** of the paths associated to leaves of the corresponding class.

The corresponding layered network can easily be implemented by a Cascade-Correlation network. The units at the first level have the same set of input weights as the units in the NT. Moreover, the connections between these (hidden) units are set to zero. All the remaining units will have input connections set to zero, while the connections between (hidden) units will be such as to implement the **and** functions at the second hidden layer and the **or** functions at the output layer. Given the set of nodes belonging to a path P, the **and** function at the second layer of the network can be implemented by setting to 1 the weights of the connections referring to nodes with output equal to 1 and to -1 the weights of the connections referring to nodes with output equal to zero. The threshold of the unit is set to $(-N_P + 0.5)$, where N_P is the number of nodes with output equal to 1 in the path P. The **or** function at the output layer is, instead, implemented by connecting (with weight 1) to the output unit only the units of the second layer involved in the **or** operation. The threshold of the unit is set to -0.5.□

Note that from the previous two theorems, and from the fact that Cascade-Correlation networks and NTs for structures are generalizations of Recurrent Cascade-Correlation networks and NTs for sequences, respectively, we have

COROLLARY 7.4. *A NT for sequences cannot simulate any finite state machine.*

COROLLARY 7.5. *Any NT for sequences can be simulated by a Recurrent Cascade-Correlation network.*

8. CONCLUSIONS

We have shown that not all the neural networks proposed for processing of structure have the same computational power. Specifically, we demonstrated that Elman-style networks can implement any FRA, while neither cascade-correlation networks nor NTs can. As a special case of the latter result, we obtained that NTs for sequences cannot implement any finite state machine. These limitations of cascade-correlation networks, and NTs for structures should be considered when designing solutions for specific applications. We note that the present study is restricted to domains of trees, while all the discussed models can deal with DAGs and some (Elman-style networks) with cyclic graphs. Thus, it is necessary to further study the computational capabilities of these models with respect to classes of more general labeled graphs.

REFERENCES

Alon, N., Dewdney, A.K., & Ott, T.J. (1991). Efficient simulation of finite automata by neural nets. *Journal of the Association for Computing Machinery*, **38 (2),** 495–514.

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and regression trees.* Belmont, CA: Wadsworth International Group.

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

Fahlman, S.E. (1991). The recurrent cascade-correlation architecture. In R.P. Lippmann, J.E. Moody, & D.S. Touretzky (Eds.), *Advances in neural information processing systems, 3* (pp. 190–196). San Mateo, CA: Morgan Kaufmann.

Fahlman, S.E., & Lebiere, C. (1990), The cascade-correlation learning architecture. In D.S. Touretzky (Ed.), *Advances in neural information processing systems, 2* (pp. 524–532). San Mateo, CA: Morgan Kaufmann.

Giles, C.L., Chen, D., Sun, G.Z., Chen, H.H., Lee, Y.C., & Goudreau, M.W. (1995). Constructive learning of recurrent neural networks: Limitations of recurrent cascade correlation and a simple solution. *IEEE Transactions on Neural Networks*, **6 (4),** 829–836.

Goudreau, M.W., Giles, C.L., Chakradhar, S.T., & Chen, D. (1994). First-order vs. second-order single layer recurrent neural networks. *IEEE Transactions on Neural Networks*, **5 (3),** 511–513.

Gonzalez, R.C., & Thomason, M.G. (1978). *Syntactical pattern recognition.* Menlo Park, CA: Addison-Wesley.

Kremer, S.C. (1996a). Comments on "constructive learning of recurrent neural networks:...", cascading the proof describing limitations of recurrent cascade correlation. *IEEE Transactions on Neural Networks*, **6**(4), 1047–1049.

Kremer, S.C. (1996). On the computational power of Elman-style recurrent networks. *IEEE Transactions on Neural Networks*, **6 (4),** 1000–1004.

Kremer, S.C. (1996). Finite state automata that recurrent cascade-correlation cannot represent. In D. Touretzky, M. Mozer, & M. Hasselno (Eds.), *Advances in neural information processing systems, 7* (pp. 612–618). Cambridge, MA: MIT Press.

Omlin, C.W., & Giles, C.L. (1996). Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM*, **43**(6), 937–972.

Sankar, A., & Mammone, R. (1991). *Neural tree networks* (pp. 281–302). Neural Networks: Theory and Applications. London: Academic Press.

Siegelmann, H.T., & Sontag, E.D. (1995). On the computational power of neural nets. *Journal of Computer and System Sciences*, **50 (1),** 132–150.

Sirat, J.A., & Nadal, J.-P. (1990). Neural trees: a new tool for classification. *Network*, **1,** 423–438.

Sperduti, A. (1994). Labeling RAAM. *Connection Science*, **6 (4),** 429–459.

Sperduti, A. (1995). Stability properties of labeling recursive auto-associative memory. *IEEE Transactions on Neural Networks*, **6 (6),** 1452–1460.

Sperduti, A., Starita, A., & Goller, C. (1995). Learning distributed representations for the classification of terms. In *Proceedings of the International Joint Conference on Artificial Intelligence*, C.S. Mellish (Ed.), Montreal, Canada, 20–25 August (pp. 509–515). San Mateo, CA: Morgan Kaufmann.

Sperduti, A., Majidi, D., & Starita, A. (1996). Extended cascade-correlation for syntactic and structural pattern recognition. In P. Perner, P. Wang, & A. Rosenfeld (Eds.), *Advances in structural and syntactical pattern recognition*, Lecture Notes in Computer Science, 1121 (pp. 90–99). Berlin: Springer.

Thatcher, J.W. (1973). Tree automata: An informal survey. In A.V. Aho (Ed.), *Currents in the theory of computing*. Englewood Cliffs, NJ: Prentice-Hall.