Supervised Neural Networks for the Classification of Structures

Alessandro Sperduti and Antonina Starita, *Member, IEEE*

***Abstract*—Until now neural networks have been used for classifying unstructured patterns and sequences. However, standard neural networks and statistical methods are usually believed to be inadequate when dealing with complex structures because of their feature-based approach. In fact, feature-based approaches usually fail to give satisfactory solutions because of the sensitivity of the approach to the *a priori* selection of the features, and the incapacity to represent any specific information on the relationships among the components of the structures. However, we show that neural networks can, in fact, represent and classify structured patterns. The key idea underpinning our approach is the use of the so called "generalized recursive neuron," which is essentially a generalization to structures of a recurrent neuron. By using generalized recursive neurons, all the supervised networks developed for the classification of sequences, such as backpropagation through time networks, real-time recurrent networks, simple recurrent networks, recurrent cascade correlation networks, and neural trees can, on the whole, be generalized to structures. The results obtained by some of the above networks (with generalized recursive neurons) on the classification of logic terms are presented.**

***Index Terms*— Classification of graphs, gradient methods, graph theory, learning systems, recurrent neural networks, structured domains, supervised classification.**

I. Introduction

STRUCTURED domains are characterized by complex patterns which are usually represented as lists, trees, and graphs of variable sizes and complexity. The ability to recognize and classify these patterns is fundamental for several applications, such as medical and technical diagnoses (discovery and manipulation of structured dependencies, constraints, explanations), molecular biology and chemistry [classification of chemical structures, DNA analysis, quantitative structure-property relationship (QSPR), quantitative structure-activity relationship (QSAR)], automated reasoning (robust matching, manipulation of logical terms, proof plans, search space reduction), software engineering (quality testing, modularization of software), geometrical and spatial reasoning (robotics, structured representation of objects in space, figure animation, layouting of objects), speech and text processing (robust parsing, semantic disambiguation, organizing and finding structure in texts and speech), and other applications that use, generate or manipulate structures.

Manuscript received February 5, 1996; revised August 12, 1996 and December 22, 1996.

The authors are with the University of Pisa, Dipartimento di Informatica, 56125 Pisa, Italy.

Publisher Item Identifier S 1045-9227(97)02749-5.

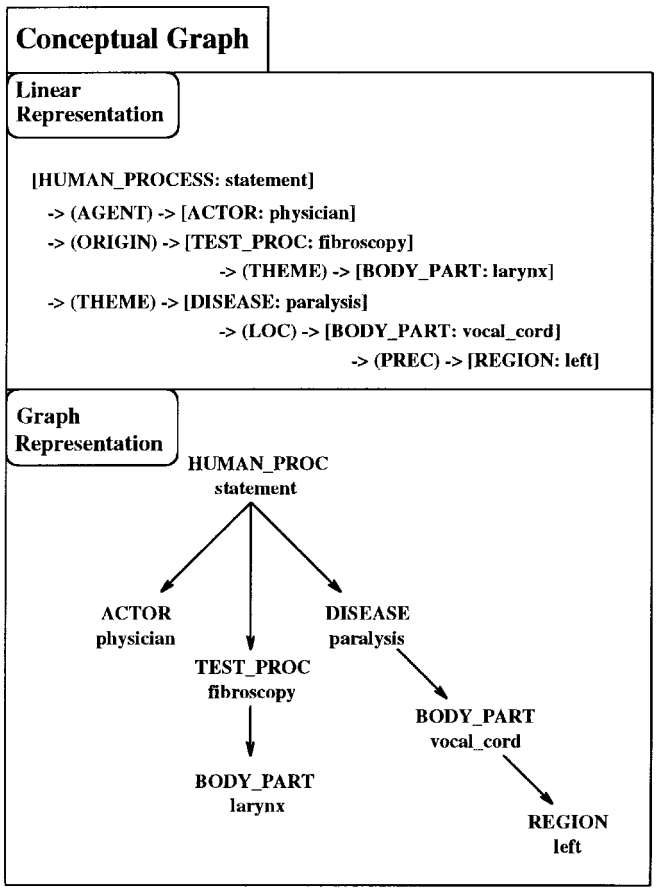


Fig. 1. Example of a conceptual graph encoding a medical concept.

While neural networks are able to classify static information or temporal sequences, the current state of the art does not allow the efficient classification of structures of different sizes. Some advances in this area have recently been presented in [33], where preliminary work on the classification of logical terms, represented as directed labeled graphs, was reported. The aim, in that case, was the realization of a hybrid (symbolic/connectionist) theorem prover, where the connectionist part had to learn a heuristic for a specific domain in order to speed-up the search for a proof. Other related techniques for dealing with structured patterns can be found, for example, in [21] and [27]–[29].

The aim of this paper is to show, at least in principle, that neural networks can deal with structured domains. Some basic concepts, which we believe are very useful for expanding the computational capabilities of neural networks to structured do-

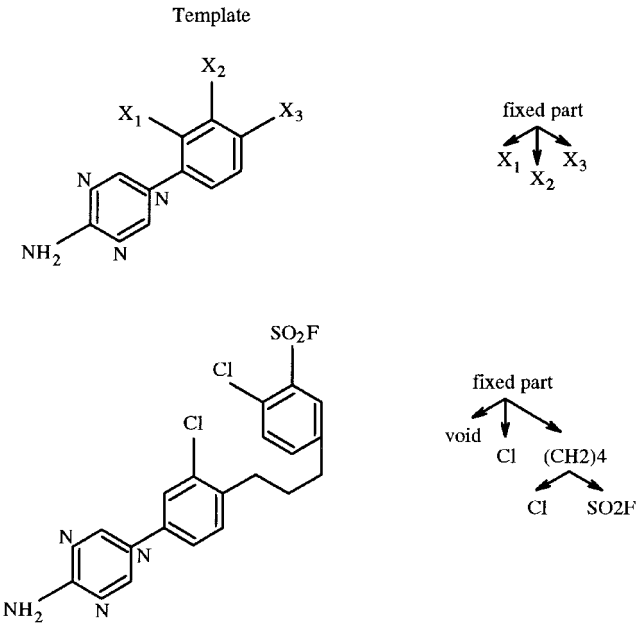


Fig. 2. Chemical structures represented as trees.

mains, are given. Specifically, we propose a generalization of a recurrent neuron—the *generalized recursive neuron*—which is able to build a map from a domain of structures to the set of reals. This newly defined neuron allows the formalization of several supervised models for structures which stem very naturally from well known models, such as backpropagation through time networks, real-time recurrent networks, simple recurrent networks, recurrent cascade correlation networks, and neural trees.

This paper is organized as follows. In Section II we introduce structured domains and some preliminary concepts on graphs and neural networks. The general problem of how to encode labeled graphs for classification is discussed in Section III. The generalized recursive neurons are defined in Section IV, where some related concepts are discussed. Several supervised learning algorithms, derived by standard and well-known learning techniques, are presented in Section V. Simulation results for a subset of the proposed algorithms are reported in Section VI, and conclusions drawn in Section VII.

II. Structured Domains

This paper deals with structured patterns which can be represented as directed labeled graphs. Some examples of these kinds of patterns are shown in Figs. 1–4, which report some typical examples from structured domains such as conceptual graphs representing medical concepts (Fig. 1), chemical structures (Fig. 2), bubble chamber events (Fig. 3), and logical terms (Fig. 4). All these structures can also be represented by using a feature-based approach. Feature-based approaches, however, are very sensitive to the features selected for the representation and are inable to represent any specific information about the relationships among components. Standard neural networks, as well as statistical methods, are usually bound to this kind of representation and are thus generally not considered as being adequate for dealing with structured domains. However, we will show that neural networks can, in fact, represent and classify structured patterns.

A. Preliminaries

Graphs: We consider finite directed vertex 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 *vertices*, 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*). Note that graphs may have loops, and labels are not restricted to being binary. Specifically, 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 vertex $x \in V_X$, the function $\mathrm{out\_degree}_X(x)$ returns the number of edges leaving from x, i.e., $\mathrm{out\_degree}_X(x) = \#\{(x,z) \mid (x,z) \in E_X \wedge z \in V_X\}$, while the function $\mathrm{in\_degree}_X(x)$ returns the number of edges entering x, i.e., $\mathrm{in\_degree}_X(x) = \#\{(z,x) \mid (z,x) \in E_X \wedge z \in V_X\}$. Given a total order on the edges leaving from x, the vertex $y = \mathrm{out}_X(x,j)$ in V_X is the vertex pointed by the jth pointer leaving from x. The *valence* of a graph X is defined as $\max_{x \in V_X}\{\mathrm{out\_degree}_X(x)\}$. A *labeled directed acyclic graph (labeled DAG)* is a graph, as defined above, without loops. A vertex $s \in V_X$ is called a *supersource* for X if every vertex in X can be reached by a path starting from s. The root of a tree (which is a special case of a directed graph) is always the (unique) *supersource* of the tree.

Structured Domain, Target Function, and Training Set: We define a *structured domain* $\mathcal{D}$ (over Σ) as any (possibly infinite) set of graphs (over Σ). The *valence of a domain* $\mathcal{D}$ is defined as the maximum among the out-degrees of the graphs belonging to $\mathcal{D}$. Since we are dealing with learning, we need to define the target function we want to learn. In approximation tasks, a *target function* $\xi()$ over $\mathcal{D}$ is defined as any function $\xi: \mathcal{D} \rightarrow \Re^k$, where k is the output dimension, while in (binary) classification tasks we have $\xi: \mathcal{D} \rightarrow \{0,1\}^k$ (or $\xi: \mathcal{D} \rightarrow \{-1,1\}^k$.) A training set T on a domain $\mathcal{D}$ is defined as a set of couples $(X, \xi(X))$, where $X \in \mathcal{U} \subseteq \mathcal{D}$ and $\xi()$ is a target function defined on $\mathcal{D}$.

Standard and Recurrent Neurons: The output $o^{(s)}$ of a standard neuron is given by

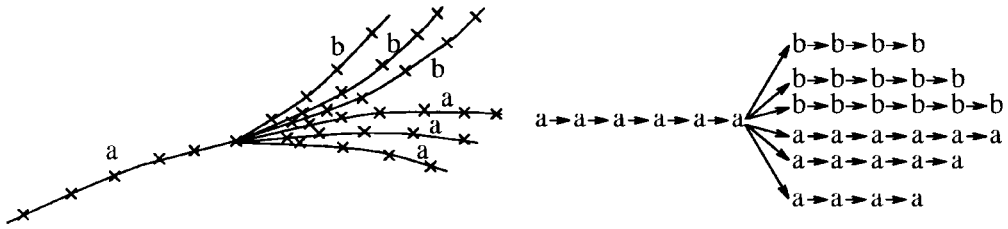
$$o^{(s)} = f\left(\sum_i w_i I_i\right) \tag{1}$$

where $f()$ is some nonlinear squashing function applied to the weighted sum of inputs I.[1] A recurrent neuron with a single self-recurrent connection, on the other hand, computes its output $o^{(r)}(t)$ as follows:

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

where $f()$ is applied to the weighted sum of inputs I plus the self-weight, w_s, times the previous output. The above formula

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



(a) (b)

Fig. 3. Bubble chamber events: (a) coded events and (b) corresponding tree representation.

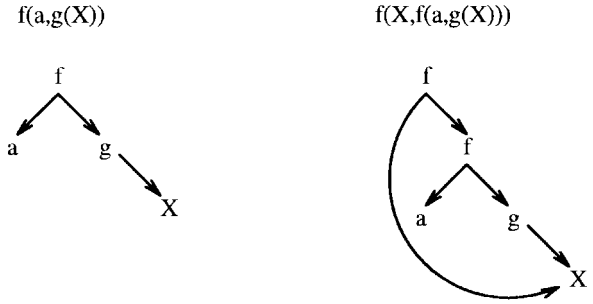


Fig. 4. Example of logic terms represented as labeled directed acyclic graphs.

can be extended both considering several interconnected recurrent neurons and delayed versions of the outputs. For the sake of presentation, we will skip these extensions.

III. The Encoding Problem

In this paper we face the problem of devising neural network architectures and learning algorithms for the classification of structured patterns, i.e., labeled graphs. We are given a structured domain $\mathcal{D}$ over Σ, and a training set $T = (X, \xi(X))$ representing a classification task.

Fig. 5 reports the standard way to approach this problem by using a standard neural network. Each graph X is encoded as a fixed-size vector which is then given as input to a feedforward neural network for classification. This approach is motivated by the fact that neural networks only have a fixed number of input units while graphs are variable in size. The encoding process is usually defined *a priori* and does not depend on the classification task. For example, in molecular biology and chemistry, where chemical compounds are represented as labeled graphs, the encoding process is performed through the definition of *topological indexes* [14], [22], which are designed by a very expensive trial and error approach.

The *a priori* definition of the encoding process has two main drawbacks.

1) The relevance of different features of the graphs may change dramatically for different learning tasks.
2) Since the encoding process has to work for any classification task, each graph must get a different representation; this may result in vectorial representations which are very difficult to classify.

To overcome the above difficulties, we propose to adapt the encoding process to the specific classification task at hand. This can be done by implementing the encoding process through an additional neural network which is trained, alongside the neural network performing the classification, to learn the best way to encode the graphs for the given classification task. Unfortunately, standard neurons as well as recurrent neurons are not powerful enough to be used in such a network. This is because the former were devised for processing unstructured patterns, while the latter can only naturally process sequences. In the next section we thus need to define a generalization of recurrent neurons which allow us to formally define how labeled directed graphs can be encoded by a neural network.

IV. The First-Order Generalized Recursive Neuron

Standard and recurrent neurons are not suited to dealing with labeled structures. In fact, neural networks using these kind of neurons can deal with approximation and classification problems in structured domains only by using a complex and very unnatural encoding scheme which maps structures onto fixed-size unstructured patterns or sequences. We propose to solve this inadequacy of the present state of the art in neural networks by introducing the *generalized recursive neuron*.

The generalized recursive neuron is an extension of the recurrent neuron where instead of just considering the output of the unit in the previous time step, we consider the outputs of the unit for all the vertices which are pointed by the current input vertex (see Fig. 6). The output $o^{(g)}(x)$ of the generalized recursive neuron for a vertex x of a graph X is defined as

$$o^{(g)}(x) = f\left(\sum_{i=1}^{N_L} w_i l_i + \sum_{j=1}^{\text{out\_degree}_X(x)} \hat{w}_j o^{(g)}(\text{out}_X(x,j))\right) \tag{3}$$

where f is a sigmoidal function, N_L is the number of units encoding the label $\mathbf{l} = \phi_X(x)$ attached to the current input x, and $\hat{w}_j$ are the weights on the recursive connections. Thus, the output of the neuron for a vertex x is computed recursively on the output computed for all the vertices pointed by it. Notice 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 vertex x with $\text{out\_degree}_X(x) < n$.

To clarify how a generalized recursive neuron works, it may be useful to show how it relates to a recurrent neuron.

1) A recurrent neuron can be considered as a generalized recursive neuron applied to lists, i.e., labeled graphs with

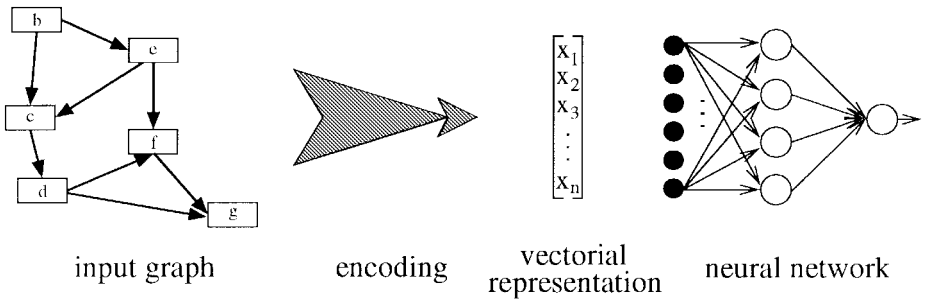


Fig. 5. Classification of graphs by neural networks. The input graph is encoded as a fixed-size vector which is given as input to a feedforward neural network for classification.

valence one; in that case, the position of a vertex within the list corresponds to the time of processing; e.g., given the list

$$x_k \rightarrow x_{k-1} \rightarrow \cdots \rightarrow x_1$$

where $\phi(x_i) = \mathbf{l}^{(i)},\ i = 1, \cdots, k$, the set of equations derived by the recursive application of (3) is

$$o(x_1) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(1)}\right)$$

$$o(x_i) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(i)} + \hat{w}_1 o(x_{i-1})\right) \quad i = 2, \cdots, k$$

where $\hat{w}_1$ is the weight on the recurrent connection; by making the time t of processing explicit, the above equations can be rewritten in the following forms:

$$o(t=1) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(1)}\right)$$

$$o(t) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(t)} + \hat{w}_1 o(t-1)\right) \quad t = 2, \cdots, k$$

which correspond to the equations obtained by using a recurrent neuron;

2) From the above equations it is clear that the output of the recurrent neuron at time t depends only on the output at time $t-1$; this is due to the fact that lists do not contain cycles; in general, however, graphs contain cycles and a generalized recursive neuron must deal with them. Specifically, the outputs of the generalized recursive neuron for the vertices belonging to a cycle are obtained as the solutions of a system of interdependent equations; e.g., the graph

$$\begin{array}{ccccccc} x_6 & \rightarrow & x_5 & \leftarrow & x_4 & & \\ & & \downarrow & & \uparrow & & \\ & & x_3 & \rightarrow & x_2 & \rightarrow & x_1 \end{array}$$

where $\mathrm{out}_X(x_2, 1) = x_1,\ \mathrm{out}_X(x_2, 2) = x_4$, and $\phi(x_i) = \mathbf{l}^{(i)}$, yields the following set of equations:

$$\left.\begin{array}{l} o(x_2) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(2)} + \hat{w}_1 o(x_1) + \hat{w}_2 o(x_4)\right) \\ o(x_3) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(3)} + \hat{w}_1 o(x_2)\right) \\ o(x_4) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(4)} + \hat{w}_1 o(x_5)\right) \\ o(x_5) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(5)} + \hat{w}_1 o(x_3)\right) \end{array}\right\} \text{cycle } (x_5, x_3, x_2, x_4, x_5)$$

$$o(x_6) = f\left(\sum_{j=1}^{N_L} w_j l_j^{(6)} + \hat{w}_1 o(x_5)\right)$$

where the output for the vertices involved in the cycle, i.e., $x_2,\ x_3,\ x_4$, and x_5, are interdependent; this means that, while $o(x_1)$ can be readily computed, $o(x_6)$ can only be computed after the subset of equations referring to the cycle has been solved.

In the next section we introduce the concept of an *encoding network* which will allow us to give a general solution to the problem of the adaptive encoding of directed graphs. Before proceeding, however, let us see how (3) is defined when considering N_g interconnected generalized recursive neurons. Equation (3) becomes

$$\mathbf{o}^{(g)}(x) = \mathbf{F}\left(\mathbf{W}\mathbf{l} + \sum_{j=1}^{\text{out\_degree}_X(x)} \hat{\mathbf{W}}_j \mathbf{o}^{(g)}(\mathrm{out}_X(x, j))\right) \tag{4}$$

where $\mathbf{F}_i(\mathbf{v}) = f(v_i),\ \mathbf{l} \in \Re^{N_L},\ \mathbf{W} \in \Re^{N_g \times N_L},\ \mathbf{o}^{(g)}(x), \mathbf{o}^{(g)}(\mathrm{out}_X(x, j)) \in \Re^{N_g},\ \hat{\mathbf{W}}_j \in \Re^{N_g \times N_g}$. In the following, when referring to the output of a generalized neuron we will drop the upper index.

A. Generation of Neural Representations for Graphs

To understand how generalized recursive neurons can generate representations for directed graphs, let us consider a single generalized recursive neuron u and a single graph X. The following two conditions must hold.

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

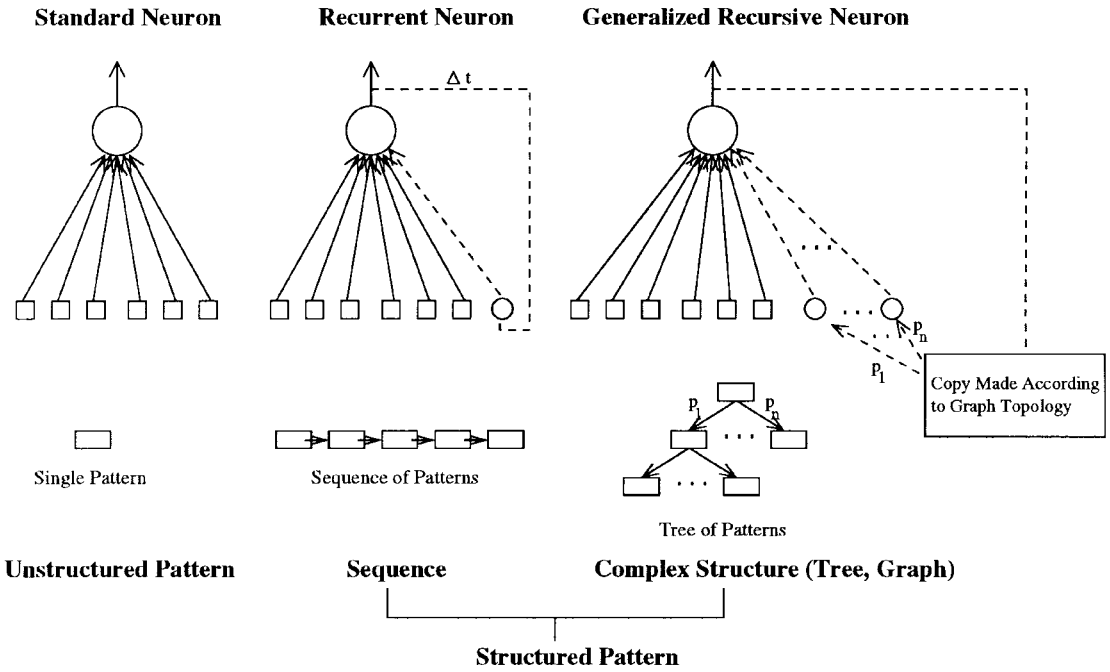


Fig. 6. Neuron models for different input domains. The standard neuron is suited for the processing of unstructured patterns, the recurrent neuron for the processing of sequences of patterns, and finally the proposed generalized recursive neuron can deal very naturally with structured patterns.

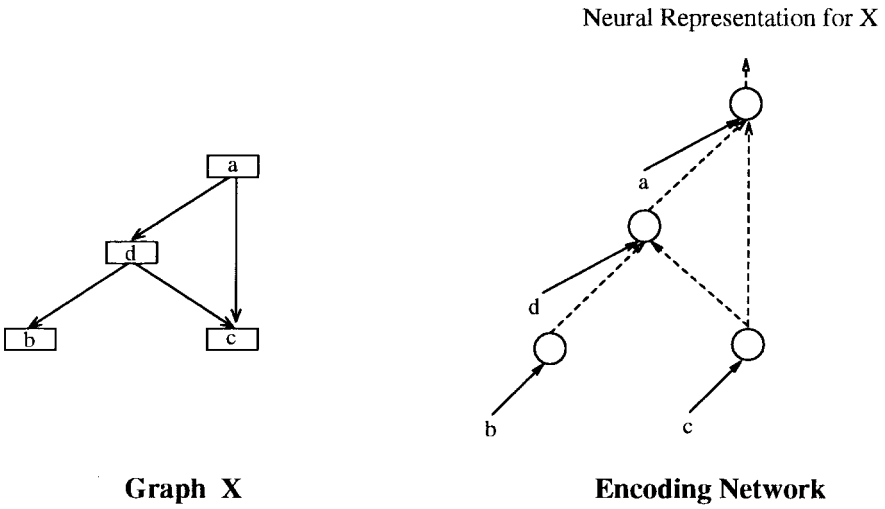


Fig. 7. The encoding network for an acyclic graph. The graph is represented by the output of the encoding network.

Supersource: graph X must have a reference *supersource*. Notice that if graph X does not have a supersource, then it is still possible to define a convention for adding to graph X an extra vertex s (with a minimal number of outgoing edges), such that s is a supersource for the new graph (see Appendix A for an algorithm).

If the above conditions are satisfied, we can adopt the convention that graph X is represented by $o(s)$, i.e., the output of u computed for the supersource s. Consequently, due to the recursive nature of (3), it follows that the neural representation for an acyclic graph 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. 7). If the structure contains cycles, then the resulting encoding network is recurrent (see Fig. 8), and the neural representation is considered to be well formed only if $o(s)$ converges to a stationary value.

The encoding network fully describes how the representation for the structure is computed and it will be used in the following to derive the learning rules for the generalized recursive connections.

When considering a structured domain, the number of recursive connections of u must be equal to the valence of the domain. The extension to a set of N_g generalized neurons is trivial: if the valence of the domain is N_V, each generalized neuron will have N_V groups of N_g recursive connections each.

B. Optimized Training Set for Directed Acyclic Graph

When considering DAG's, the training set can be organized so to improve the computational efficiency of both the reduced

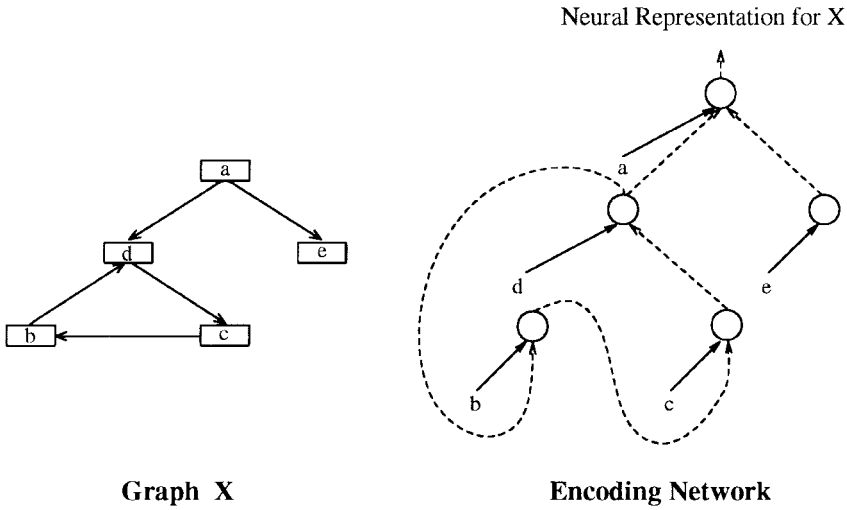
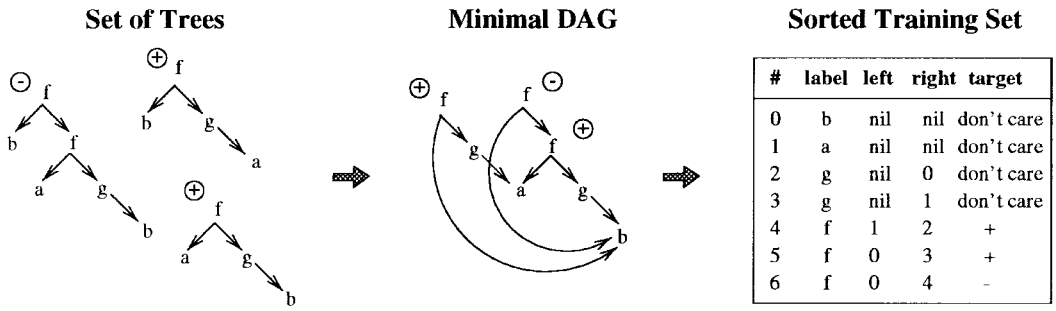


Fig. 8. The encoding network for a cyclicgraph. In this case, the encoding network is recurrent and the graph is represented by the output of the encoding network at a fixed point of the network dynamics.



#	label	left	right	target
0	b	nil	nil	don't care
1	a	nil	nil	don't care
2	g	nil	0	don't care
3	g	nil	1	don't care
4	f	1	2	+
5	f	0	3	+
6	f	0	4	-

Fig. 9. Optimization of the training set: the set ofstructures (in this case, trees) is transformed into the corresponding minimal DAG, which is then used to generate thesorted training set. The sorted training set is then transformed into a set of sorted vectors using the numeric codes for the labels and targets, and used as a training set for the network.

representations and the learning rules. In fact, given a training set T of DAG's, if there are graphs $X_1, X_2 \in T$ which share a common subgraph $\hat{X}$, then we need to explicitly represent $\hat{X}$ only once. The optimization of the training set can be performed in two stages (see Fig. 9 for an example).

1) All the DAG's in the training set are merged into a single minimal DAG, i.e., a DAG with a minimal number of vertices.
2) A topological sort on the vertices of the minimal DAG is performed to determine the updating order on the vertices for the network.

Both stages can be done in linear time with respect to the size of all DAG's and the size of the minimal DAG, respectively.[2] Specifically, stage 1) can be done by removing all the duplicate subgraphs through a special subgraph-indexing mechanism (which can be implemented in linear time).

The advantage of having a sorted training set is that all the reduced representations (and also their derivatives with respect to the weights, as we will see when considering learning) can be computed by a single ordered scan of the training set. Moreover, the use of the minimal DAG leads to a considerable reduction in space complexity. In some cases, such reduction can even be exponential.

C. Well-Formedness of Neural Representations

When considering cyclic graphs, to guarantee that each encoded graph gets a proper representation through the encoding network, we have to guarantee that for every initial state the trajectory of the encoding network converges to an equilibrium, otherwise it would be impossible to process a nonstationary representation. This is particularly important when considering cyclic graphs. In fact, acyclic graphs are guaranteed to get a convergent representation because the resulting encoding network is feedforward, while cyclic graphs are encoded by using a recurrent network. Consequently, the well-formedness of representations can be obtained by defining conditions that guarantee the convergence of the encoding network. Regarding this issue, it is well known that if the weight matrix is symmetric, an additive network with first-order connections possesses a Lyapunov function and is convergent ([5], [15]). Moreover, Almeida [1] proved a more general symmetry condition than the symmetry of the weight matrix, i.e., a system satisfying *detailed balance*

$$w_{ij} f(\mathrm{net}_j) = w_{ji} f(\mathrm{net}_i) \tag{5}$$

[2] This analysis was done by C. Goller.

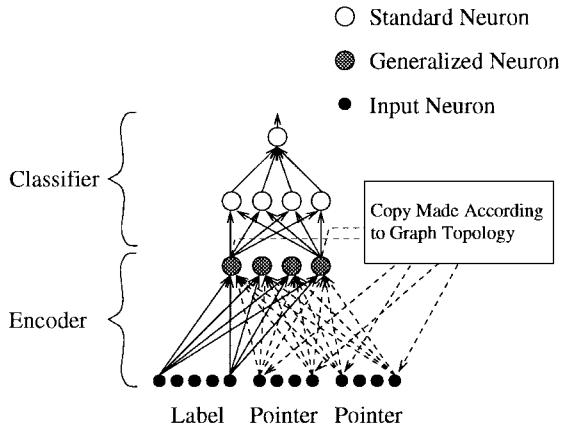


Fig. 10. A possible architecture for the classification of structures: the generalized recursive neurons generate the neural representation for the structures, which are then classified by a standard feedforward network.

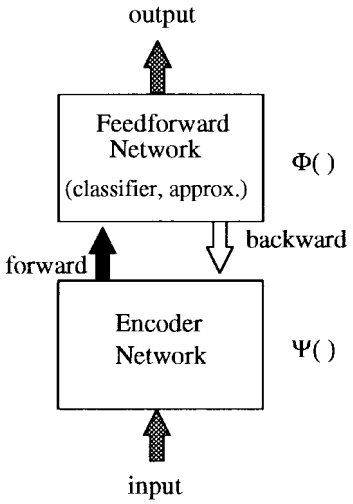


Fig. 11. The functional scheme of the proposed network. The encoder and the cClassifier are considered as two distinct entities which exchange information: the encoder forward the neural representations of the structures to the classifier; in turn the classifier returns to the encoder the deltas, which are then used by the encoder to adapt its weights.

is guaranteed to possess a Lyapunov function as well. On the other hand, if the norm of the weight matrix (not necessarily symmetric) is sufficiently small, e.g., satisfying

$$\sum_i \sum_j w_{ij}^2 < \frac{1}{\max_i \ f'(\text{net}_i)} \tag{6}$$

the network's dynamics can be shown to go to a unique equilibrium for a given input ([2]).

The above results are sufficient when the encoding network is static, however we will see that in several cases, for example, in classification tasks, the encoding network changes with learning. In these cases, these results can be exploited to define the initial weight matrix, but there is no guarantee that learning will preserve the stability of the encoding network.

V. Supervised Models

In this section, we discuss how several standard supervised algorithms for neural networks can be extended to structures.

A. Backpropagation Through Structure

The task addressed by backpropagation through time networks [23] is to produce particular output sequences in response to specific input sequences. These networks are, generally, fully recurrent, in which any unit may be connected to any other. Consequently, they cannot be trained by using plain backpropagation. A trick, however, can be used to turn an arbitrary recurrent network into an equivalent feedforward network when the input sequences have a maximum length T. In this case, all the units of the network can be duplicated T times (*unfolding of time*), so that the state of a unit in the recurrent network at time τ is held by the τth copy of the same unit in the feedforward network. By preserving the same weight values through the layers of the feedforward network, it is not difficult to see that the two networks will behave identically for T time steps. The feedforward network can be trained by backpropagation, taking care to preserve the identity constraint between weights of different layers. This can be guaranteed by adding together the individual gradient contributions of corresponding copies of the same weight and then changing all copies by the total amount.

Backpropagation through time can easily be extended to structures. The basic idea is to use generalized recursive neurons to encode the structures; the representations obtained are then classified or used to approximate an unknown function by a standard feedforward network. An example of this kind of network for classification is given in Fig. 10.

Given an input graph X, the network output $\mathbf{o}(X)$ can be expressed as the composition of the encoding function $\Psi()$ and the classification (or approximation) function $\Phi()$ (see Fig. 11):

$$\mathbf{o}(X) = \Phi(\Psi(X)). \tag{7}$$

Learning for the set of weights $\mathbf{W}_\Phi$ can be implemented by plain backpropagation on the feedforward network realizing $\Phi()$

$$\Delta \mathbf{W}_\Phi = -\eta \frac{\partial \ \text{Error}(\Phi(\mathbf{y}))}{\partial \mathbf{W}_\Phi} \tag{8}$$

where $\mathbf{y} = \Psi(X)$, i.e., the input to the feedforward network, while learning for the set of weights, $\mathbf{W}_\Psi$, realizing $\Psi()$ can be implemented by

$$\Delta \mathbf{W}_\Psi = -\eta \frac{\partial \text{Error}(\Phi(\mathbf{y}))}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{W}_\Psi} \tag{9}$$

where the first term represents the error coming from the feedforward network and the second one represents the error due to the encoding network.

How the second term is actually computed depends on the family of structures in the training set. If the training set consists of DAG's, then plain backpropagation can be used on the encoding network. Otherwise, i.e., if there are graphs with cycles, then *recurrent backpropagation* [20] must be used. Consequently, we treat these two cases separately.

Case I—DAG's: This case has been treated by Goller and Küchler in [12]. Since the training set contains only DAG's, $\frac{\partial \mathbf{y}}{\partial \mathbf{W}_\Psi}$ can be computed by backpropagating the error from the feedforward network through the encoding network of each structure. As in backpropagation through time, the gradient contributions of corresponding copies of the same weight are collected for each structure. The total amount is then used to change all the copies of the same weight. If learning is performed by *structure*, then the weights are updated after the presentation of each individual structure, otherwise, the gradient contributions are collected through the whole training set and the weights changed after all the structures in the training set have been presented to the network.

Case II—Cyclic Graphs: When considering cyclic graphs, $\frac{\partial \mathbf{y}}{\partial \mathbf{W}_\Psi}$ can only be computed by resorting to *recurrent backpropagation*. In fact, if the input structure contains cycles, then the resulting encoding network is cyclic.

In the standard formulation, a recurrent backpropagation network $\mathcal{N}$ with n units is defined as

$$\mathbf{o}^{(rbp)}(t+1) = \mathbf{F}(\mathbf{W}^{(rbp)}\mathbf{o}^{(rbp)}(t) + \mathbf{I}^{(rbp)}) \tag{10}$$

where $\mathbf{I}^{(rbp)} \in \Re^n$ is the input vector for the network, and $\mathbf{W}^{(rbp)} \in \Re^n \times \Re^n$ the weight matrix. The learning rule for a weight of the network is given by

$$\Delta w_{rs}^{(rbp)} = \eta o_s^{(rbp)} o_r'^{(rbp)} \sum_k e_k (\mathbf{L}^{-1})_{kr} \tag{11}$$

where all the quantities are taken at a fixed point of the recurrent network, e_k is the error between the current output of unit k and the desired one, $L_{ji} = \delta_{ji} - o_j'^{(rbp)} w_{ji}$ (δ_{ji} is a Kronecker delta), and the quantity

$$y_j^{(rbp)} = \sum_k e_k (\mathbf{L}^{-1})_{kj} = \sum_i o_i'^{(rbp)} w_{ij} y_i^{(rbp)} + e_i \tag{12}$$

can be computed by relaxing the *adjoint* network $\mathcal{N}'$, i.e., a network obtained from $\mathcal{N}$ by reversing the direction of the connections. The weight w_{ji} from neuron i to neuron j in $\mathcal{N}$ is replaced by $o_i^{(rbp)} w_{ij}$ from neuron j to neuron i in $\mathcal{N}'$. The activation functions of $\mathcal{N}'$ are linear and the output units in $\mathcal{N}$ become input units in $\mathcal{N}'$ with e_i as input.

Given a cyclic graph X, let $m = \#V_X$, N_g be the number of generalized neurons and $\mathbf{o}_i(t)$ the output at time t of these neurons for $x_i \in V_X$. Then we can define

$$\mathbf{o}(t) = \begin{bmatrix} \mathbf{o}_1(t) \\ \mathbf{o}_2(t) \\ \vdots \\ \mathbf{o}_m(t) \end{bmatrix} \tag{13}$$

as the global state vector for our recurrent network, where for convention $\mathbf{o}_1(t)$ is the output of the neurons representing the *supersource* of X.

To account for the labels, we have to slightly modify (10)

$$\mathbf{o}(t+1) = \mathbf{F}\big(\hat{\mathbf{W}}_\Psi^X \mathbf{o}(t) + \mathbf{W}_\Psi^X \mathbf{I}^X\big) \tag{14}$$

where

$$\mathbf{I}^X = \begin{bmatrix} \mathbf{l}_1 \\ \vdots \\ \mathbf{l}_m \end{bmatrix} \tag{15}$$

with $\mathbf{l}_i = \phi_X(x_i),\ i = 1, \cdots, m$

$$\mathbf{W}_\Psi^X = \underbrace{\begin{bmatrix} \mathbf{W} & & 0 \\ & \ddots & \\ 0 & & \mathbf{W} \end{bmatrix}}_{m \text{ repetitions of } \mathbf{W}} \tag{16}$$

and $\hat{\mathbf{W}}_\Psi^X \in \Re^{mN_g \times mN_g}$ is defined according to the topology of the graph.

In this context, the input of the adjoint network is $e_k = (\frac{\partial \text{Error}(\Phi(\mathbf{y}))}{\partial \mathbf{y}})_k$, and the learning rules become

$$\Delta \hat{w}_{rs} = \eta o_s o_r' \sum_k e_k (\mathbf{L}^{-1})_{kr} \tag{17}$$

$$\Delta w_{rs} = \eta I_s o_r' \sum_k e_k (\mathbf{L}^{-1})_{kr}. \tag{18}$$

To hold the constraint on the weights, all the changes referring to the same weight are added and then all copies of the same weight are changed by the total amount. Note that each structure gives rise to a new adjoint network and independently contributes to the variations of the weights. Moreover, the above formulation is more general than the one previously discussed and it can also be used for DAG's, in which case the adjoint network becomes a feedforward network representing the backpropagation of errors.

An important variation to the learning rules presented above is constituted by *teacher forcing*, a technique proposed by Pineda [20] to force the creation of new fixed points in standard recurrent networks. While the effects of this technique is well understood in standard recurrent networks, it is not clear to us how it can influence the dynamics of the encoding networks. In fact, while a new encoding network for each graph in the training set is generated, all the encoding networks are interdependent since the weights on the connections are shared. A more accurate study on how this sharing of resources affects the dynamics of the system when using teacher forcing is needed.

B. Extension of Real-Time Recurrent Learning

The extension of real-time recurrent learning [34] to generalized recursive neurons does not present particular problems when considering graphs without cycles. Cyclic graphs, instead, give rise to a training algorithm that can be considered only loosely in real time.

In order to be concise, we will only show how derivatives can be computed in real time, leaving to the reader the development of the learning rules, according to the chosen network architecture and error function.

Case I—DAG's: Let N_g be the number of generalized neurons, s the *supersource* of X, and

$$\mathbf{y} = \mathbf{o}(s) = \mathbf{F}\left(\mathbf{W}\mathbf{l}_s + \sum_{j=1}^{\text{out\_degree}_X(s)} \hat{\mathbf{W}}_j \mathbf{o}(\text{out}_X(s,j))\right) \tag{19}$$

be the representation of X according to the encoding network, where

$$\mathbf{W}_\Psi = [\mathbf{W}, \hat{\mathbf{W}}_1, \cdots, \hat{\mathbf{W}}_n] \tag{20}$$

and N_V is the valence of the domain. The derivatives of $\mathbf{y}$ with respect to $\mathbf{W}$ and $\hat{\mathbf{W}}_i\,(i \in [1,\cdots,N_V])$ can be computed from (19)

$$\frac{\partial y_m}{\partial W_{tk}} = o'_m(s)\left((\mathbf{l}_s)_k\delta_{mt} + \sum_{j=1}^{\text{out\_degree}_X(s)} (\hat{\mathbf{W}}_j)_m \frac{\partial \mathbf{o}(\text{out}_X(s,j))}{\partial W_{tk}}\right) \tag{21}$$

where δ_{tmt} is the Kronecker delta, $m = 1,\cdots,N_g$, $k = 0,\cdots,N_g$ and $k = 0,\cdots,N_L$, $(\hat{\mathbf{W}}_j)_t$ is the tth row of $\hat{\mathbf{W}}_j$

$$\frac{\partial y_m}{\partial (\hat{\mathbf{W}}_i)_{tq}} = o'_m(s)\left(o_q(\text{out}_X(s,i))\delta_{mt} + \sum_{j=1}^{\text{out\_degree}_X(s)} (\hat{\mathbf{W}}_j)_t \frac{\partial \mathbf{o}(\text{out}_X(s,j))}{\partial (\hat{\mathbf{W}}_i)_{tq}}\right) \tag{22}$$

where δ_{mt} is the Kronecker delta, $m = 1,\cdots,N_g$,$t = 1,\cdots,N_g$, and $q = 1,\cdots,N_g$.

These equations are recursive on the structure X, and can be computed by noting that if v is such that $\text{out\_degree}_X(v) = 0$, then

$$\frac{\partial o_m(v)}{\partial W_{tk}} = o'_m(s)(\mathbf{l}_v)_k\delta_{mt}, \quad \text{and} \quad \frac{\partial o_m(v)}{\partial (\hat{\mathbf{W}}_i)_{tq}} = 0. \tag{23}$$

This allows the computation of the derivatives in real time alongside the computation of the neural representations for the graphs.

Case II—Cyclic Graphs: When considering a cyclic graph $G = (V, E)$, the situation is very similar to the one described in Section V-A.2. However, as soon as the output $\mathbf{o}(v)$ of the encoding network for a vertex $v \in V$ of G has been computed, the corresponding derivatives $\frac{\partial \mathbf{o}(v)}{\partial \mathbf{W}_\Psi}$ must be computed as well. Of course, if there is a cycle $(v_1,\cdots,v_k,v_1)$ in G which involves all the vertices in V, then given any $v \in V$, $\mathbf{o}(v)$ can be computed only when the encoding network is relaxed, i.e., $\mathbf{o}(v)$ is computed for every $v \in V$. In this case, the learning rules are the same as in Section V-A.2.

A different situation is encountered when G does not contain global cycles, i.e., cycles involving all the vertices in V. In this case, it is useful to resort to two strictly related concepts, the strongly connected components of a graph and the related definition of a component graph.

Definition V.1—(Strongly Connected Component): A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $U \subseteq V$ such that for every pair of vertices u and v in U, there exists a path from u to v and viceversa, i.e., u and v are reachable from each other.

Definition V.2—(Component Graph): The (acyclic) component graph of $G = (V, E)$ is defined as the graph $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$, where V^{SCC} contains one vertex for each strongly connected component of G, and E^{SCC} contains the edge (u, v) if there is a directed edge from a vertex in the strongly connected component of G corresponding to u to a vertex in the strongly connected component of G corresponding to v.

Note that both the strongly connected components of a graph G and its component graph G^{SCC} can be computed by using two depth-first searches in $\Theta(\#V + \#E)$ in time [6]. Moreover, a strongly connected component of G corresponds to a maximal subset of interconnected units in the encoding network for G, while the component graph describes the functional dependences between these subsets of units. Thus, the computation of the derivatives in real time can be done by observing the following.

1) Equations (21) and (22) are still valid.
2) Each strongly connected component of the graph corresponds to a set of interdependent equations.
3) The component graph defines the dependences between sets of equations.

For example, let consider a single generalized recursive neuron and the graph

$$\begin{array}{ccccccc} x_6 & \rightarrow & x_5 & \leftarrow & x_4 & & \\ & & \downarrow & & \uparrow & & \\ & & x_3 & \rightarrow & x_2 & \rightarrow & x_1 \end{array}$$

whose component graph is

$$u_3 \rightarrow u_2 \rightarrow u_1$$

where

$$u_3 \stackrel{\text{def}}{=} \begin{array}{ccc} x_5 & \leftarrow & x_4 \\ \downarrow & & \uparrow \\ x_3 & \rightarrow & x_2 \end{array} \quad u_2 \stackrel{\text{def}}{=} x_6 \quad u_1 \stackrel{\text{def}}{=} x_1.$$

Let us consider the derivatives $y_i = \frac{\partial o(x_i)}{\partial \hat{w}_1}$. They are the solutions of the set of equations [derived by (22)]

$$\begin{array}{ll} y_1 = 0 & \text{(corresponding to } u_1\text{)} \\ \left.\begin{array}{l} y_2 = o'(x_2)(o(x_1) + \hat{w}_1 y_1 + \hat{w}_2 y_4) \\ y_3 = o'(x_3)(o(x_2) + \hat{w}_1 y_2) \\ y_4 = o'(x_4)(o(x_5) + \hat{w}_1 y_5) \\ y_5 = o'(x_5)(o(x_3) + \hat{w}_1 y_3) \end{array}\right\} & \text{(corresponding to } u_2\text{)} \\ y_6 = o'(x_6)(o(x_5) + \hat{w}_1 y_5) & \text{(corresponding to } u_3\text{)} \end{array}$$

which can be decomposed according to the component graph. Specifically, by substituting the solution for y_1 in the equation for y_2, the subset of equations corresponding to the strongly connected component u_2 can be solved through a relaxation process. Finally, once the solution value for y_5 is known, the last equation corresponding to u_3 can be solved by direct substitution. In general, by taking into account the dependences established by the component graph, both the output of the neuron and its derivatives can be computed in real time with respect to the component graph.

C. LRAAM-Based Networks and Simple Recurrent Networks

In this section, we present a class of networks which are based on the LRAAM model [33] (see Appendix B). Learning in these networks is implemented via the combination of a supervised procedure with an unsupervised one. Since it is based on truncated gradients, it is suited for both acyclic and cyclic graphs. We will see that this class of networks contains, as a special case, a network which turns out to be the extension of a simple recurrent network [7] to structures.

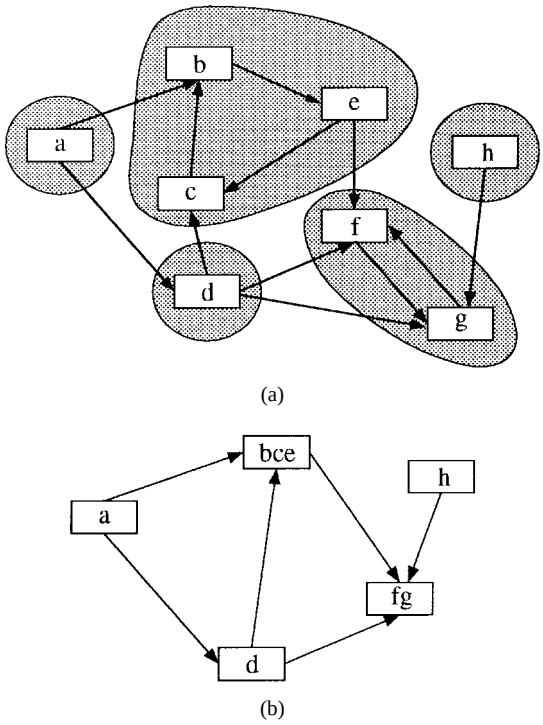


Fig. 12. (a) A labeled directed graph G. The strongly connected components of G are shown as shaded regions. (b) The acyclic component graph G^{SCC} obtained by shrinking each strongly connected component of G to a single vertex.

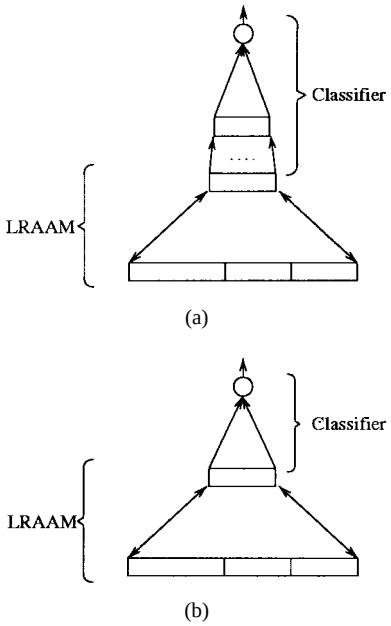


Fig. 13. LRAAM-based networks for the classification of structures.

In this type of network (see Fig. 13), the first part of the network is constituted by an LRAAM (note the double arrows on the connections) whose task is to devise a compressed representation for each structure. This compressed representation is obtained by using the standard learning algorithm for LRAAM [32]. The classification task is then performed in the second part of the network through a multilayer feedforward network with one or more hidden layers (network $\mathbf{a}$) or a simple sigmoidal neuron (network $\mathbf{b}$).

Several options for training networks $\mathbf{A}$ and $\mathbf{B}$ are available. These options can be characterized by the proportion of the two different learning rates (for the classifier and the LRAAM) and by the different degrees $\mathbf{x}$, and $\mathbf{y}$ in the presence of the following two basic features.

- The training of the classifier is not started until $\mathbf{x}$ percent of the training set has been correctly encoded and subsequently decoded by the LRAAM.
- The error from the classifier is backpropagated across $\mathbf{y}$ levels of the structures encoded by the LRAAM.[3]

Note that, even if the training of the classifier is started only when all the structures in the training set are properly encoded and decoded, the classifier's error can still change the compressed representations which, however, are maintained consistent[4] by learning in the LRAAM.

[3]The backpropagation of the error across several levels of the structures can be implemented by unfolding the encoder of the LRAAM (the set of weights from the input to the hidden layer) according to the topology of the structures.

[4]A consistent compressed representation is a representation of a structure which contains all the information sufficient for the reconstruction of the whole structure.

The reason for allowing different degrees of interaction between the classification and the representation tasks is due to the necessity of having different degrees of adaptation of the compressed representations to the requirements of the classification task. If no interaction at all is allowed, i.e., the LRAAM is trained first and then its weights frozen ($\mathbf{y} = 0$), the compressed representations will be such that similar representations will correspond to similar structures, while if full interaction is allowed, i.e., the LRAAM and the classifier are trained simultaneously, the compressed representations will be such that structures in the same class will get very similar representations.[5] On the other hand, when considering DAG's, by setting $\mathbf{y} = \max\_\mathrm{depth}$, where $\max\_\mathrm{depth}$ is the number of vertices traversed by the longest path in the structures, the classifier error will be backpropagated across the whole structure, thus implementing the backpropagation through the structure defined in Section V-A.

It is interesting to note that the SRN by Elman can be obtained as a special case of network $\mathbf{B}$. In fact, when considering network $\mathbf{B}$ (with $\mathbf{x} = 0$ and $\mathbf{y} = 1$) for the classification of lists (sequences) the same architecture is obtained, with the difference that there are connections from the hidden layer of the LRAAM back to the input layer[6], i.e., the decoding part of the LRAAM. Thus, when considering lists, the only difference between a SRN and network $\mathbf{B}$ is in the unsupervised learning performed by the LRAAM. However, when forcing the learning parameters for the LRAAM to be null, we obtain the same learning algorithm as in SRN. Consequently, we can claim that SRN is a special case of network $\mathbf{B}$. This can be better understood by looking at the right-hand side of Fig. 14, which shows network $\mathbf{B}$ in terms of

[5]Moreover, in this case, there is no guarantee that the LRAAM will be able to encode and to decode consistently all the structures in the training set, since training is stopped when the classification task has been performed correctly.

[6]The output layer of the LRAAM can be considered as being the same as the input layer.

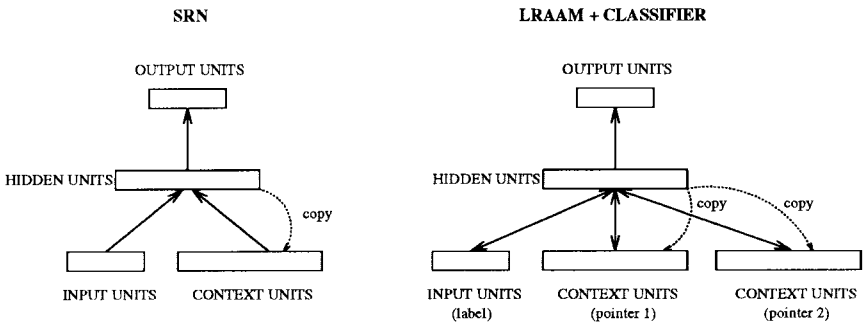


Fig. 14. (a) The network $\mathbf{B}$, with $\mathbf{x} = 0$ and $\mathbf{y} = 1$ can be considered as (b) a generalization of the simple recurrent network by Elman.

elements of an SRN. Of course, the *copy* function for network $\mathbf{B}$ is not as simple as the one used in an SRN, since the correct relationships among components of the structures to be classified must be preserved.[7]

D. Cascade-Correlation for Structures

The cascade-correlation algorithm [9] creates a standard neural network using an incremental approach for the classification of unstructured patterns. The starting network $\mathcal{N}_0$ is a network without hidden nodes trained with a least mean square algorithm. If network $\mathcal{N}_0$ 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 $\mathcal{N}_0$ is maximized.[8] The weights of u_1 are frozen and the remaining weights are retrained. If the obtained network $\mathcal{N}_1$ cannot solve the problem new hidden units are added which are connected (with frozen weights) with all the inputs and previously installed hidden units. The resulting network is a *cascade* of nodes. Fahlman extended the algorithm to the classification of sequences, obtaining good results [18]. In the following, we show that the cascade-correlation can be further extended to structures by using generalized recursive neurons. For the sake of simplicity, we will discuss the case of acyclic graphs, leaving to the reader the extension to cyclic graphs (see Section V-A, cyclic graphs).

The output of the kth hidden unit, in our framework, can be computed as

$$o^{(k)}(x) = f\left(\sum_{i=1}^{N_L} w_i^{(k)} l_i + \sum_{v=1}^{k} \sum_{j=1}^{\text{out\_degree}_X(x)\hat{w}^{(k)}(v,j)} \hat{w}_{(v,j)}^{(k)} o^{(v)} \times (\text{out}_X(x,j)) + \sum_{q=1}^{k-1} \bar{w}_q^{(k)} o^{(q)}(x)\right) \quad (24)$$

where $w_{(v,j)}^{(k)}$ is the weight of the kth hidden unit associated with 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 $u^{(\text{out})}$ is computed as

$$o^{(\text{out})}(x) = f\left(\sum_{i=1}^{k} \tilde{w}_i o^{(i)}(x)\right) \quad (25)$$

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

Learning is performed as in standard cascade-correlation, with the difference that, according to (24), the derivatives of $o^{(k)}(x)$ with respect to the weights are now

$$\frac{\partial o^{(k)}(x)}{\partial w_i^{(k)}} = \left(l_i + \sum_{j=1}^{\text{out\_degree}_X(x)} \hat{w}_{(k,j)}^{(k)} \frac{\partial o^{(k)}(\text{out}_X(x,j))}{\partial w_i^{(k)}}\right) f' \quad (26)$$

$$\frac{\partial o^{(k)}(x)}{\partial \bar{w}_q^{(k)}} = \left(o^{(q)}(x) + \sum_{j=1}^{\text{out\_degree}_X(x)} \hat{w}_{(k,j)}^{(k)} \frac{\partial o^{(k)}(\text{out}_X(x,j))}{\partial \bar{w}_q^{(k)}}\right) f' \quad (27)$$

$$\frac{\partial o^{(k)}(x)}{\partial \hat{w}_{(v,t)}^{(k)}} = \left(o^{(v)}(\text{out}_X(x,t)) + \sum_{j=1}^{\text{out\_degree}_X(x)} \hat{w}_{(k,j)}^{(k)} \times \frac{\partial o^{(k)}(\text{out}_X(x,j))}{\partial \hat{w}_{(v,t)}^{(k)}}\right) f' \quad (28)$$

where $i = 1, \cdots, N_L,\ q = 1, \cdots, (k-1),\ v = 1, \cdots, k,\ t = 1, \cdots, \text{out\_degree}_X(x),\ f'$ is the derivative of $f()$. The above equations are recurrent on the structures and can be computed by observing that for a leaf vertex y (26) reduces to $\frac{\partial o^{(k)}(y)}{\partial w_i^{(k)}} = l_i$, and all the remaining derivatives are null. Consequently, we only need to store the output values of the unit and its derivatives for each component of a structure. Fig. 15 shows the evolution of a network with two pointer fields.

Note that, if the hidden units have self-recurrent connections only, the matrix defined by the weights $\hat{w}_{(v,j)}^{(k)}$ is not triangular, but diagonal.

[7] The *copy* function needs a stack for the memorization of compressed representations. The control signals for the stack are defined by the encoding-decoding task.

[8] 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.

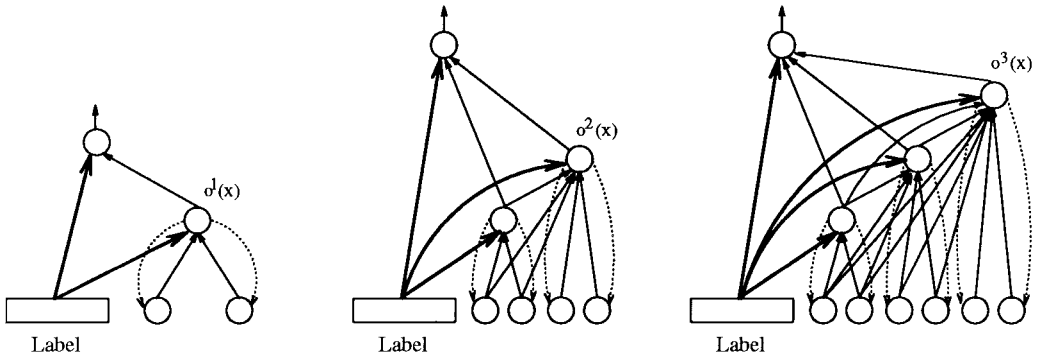


Fig. 15. The evolution of a network with two pointer fields. The units in the label are fully connected with the hidden units and the output unit.

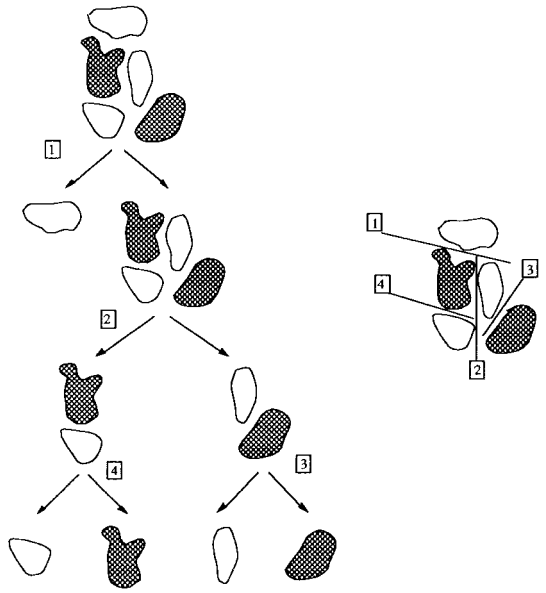


Fig. 16. An example of neural tree.

E. Extension of Neural Trees

Neural trees have recently been proposed as a fast learning method in classification tasks. They are decision trees [4] where the splitting of the data for each vertex, i.e., the classification of a pattern according to some features, is performed by a perceptron [26] or a more complex neural network [24]. After learning, each vertex 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. An example of how a binary neural tree splits the training set is shown in Fig. 16.

One advantage of the neural tree approach is that the tree structure is constructed dynamically during learning and not linked to a static structure like a standard feedforward network. Moreover, it allows incremental learning, since subtrees can be added as well as deleted to recognize new classes of patterns or to improve generalization. Both supervised [3], [24]–[26] and unsupervised [16], [18], [19] splitting of the data have been proposed. We are interested in supervised methods.

The learning and classification algorithms for binary neural trees are described in Fig. (17). Algorithms for general trees can be found in [24]. The extension of these algorithms to structures is straightforward: the standard discriminator (or network) associated to each node of the tree is replaced by a generalized recursive discriminator (or network) which can be trained with any of the learning algorithms we have presented so far.

VI. Experimental Results

We have tested some of the proposed architectures on several classification tasks involving logic terms. Note that logic terms may be represented very naturally as DAG's (see Fig. 4) and consequently, classification problems involving logic terms belong to the class of structured problems described at the beginning of the paper. In the next subsection, we present the classification problems. Then, we discuss the results obtained with LRAAM-based networks and cascade-correlation for structures.

A. Description of the Classification Problems

Table I summarizes the characteristics of each problem. The first column reports the name of the problem, the second the set of symbols (with associated arity) compounding the terms, the third shows the rule(s) used to generate the positive examples of the problem,[9] the fourth reports the number of terms in the training and test set respectively, the fifth the number of subterms in the training and test sets, and the last the maximum depth[10] of terms in the training and test sets. For example, let us consider the problem `inst1_long`. All the terms in the training and test sets are componded by three different constants, i.e., $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, and a functional $\mathbf{f}(\cdot,\cdot)$ with two arguments. Terms which have to be classified positively are required to have the first argument equal to the second one $(\mathbf{f}(\mathbf{X},\mathbf{X}))$, e.g., both $\mathbf{f}(\mathbf{f}(\mathbf{a},\mathbf{f}(\mathbf{b},\mathbf{c})),\mathbf{f}(\mathbf{a},\mathbf{f}(\mathbf{b},\mathbf{c})))$ and $\mathbf{f}(\mathbf{f}(\mathbf{f}(\mathbf{a},\mathbf{a}),\mathbf{f}(\mathbf{b},\mathbf{b})),\mathbf{f}(\mathbf{f}(\mathbf{a},\mathbf{a}),\mathbf{f}(\mathbf{b},\mathbf{b})))$ have to be classified positively, while $\mathbf{f}(\mathbf{a},\mathbf{f}(\mathbf{b},\mathbf{b}))$ has to be classified negatively. For this problem, the training set contains 202 terms, while the test set contains 98 terms; the total number of distinct subterms for the training and test sets is 403 and 204, respectively. Note

[9] Note that the terms are all ground.

[10] We define the depth of a term as the maximum number of edges between the root and leaf nodes in the term's LDAG-representation.

Learning Algorithm

Let $\{(\boldsymbol{I}_k, l_k)\}$ be the training set, where $\boldsymbol{I}_k$ is a real valued feature vector and l_k the label of the associated class, $k = 1, \cdots, P$. Let $T(t)$ be the current training set for node t, D_t the discriminator associated with t, if it is a nonterminal node, or the label of a class if it is a leaf.

1. Set $t = root$ and $T(root) = \{(\boldsymbol{I}_k, l_k)\}$;
2. If $T(t) = \emptyset$ then stop;
3. Train the discriminator D_t to split $T(t)$ into two sets: $T_l(t)$, where the output of D_t is 0, and $T_r(t)$, where the output of D_t is 1;
4. (a) Add to t a left node t_l; if $T_l(t)$ contains only patterns with label l_j for any j, then t_l is a leaf with training set $T(t_l) = \emptyset$, otherwise it is an internal node with training set $T(t_l) = T_l(t)$;
   (b) Add to t a right node t_r; if $T_r(t)$ contains only patterns with label l_j for any j, then t_r is a leaf with training set $T(t_r) = \emptyset$, otherwise it is an internal node with training set $T(t_r) = T_r(t)$;
5. repeat the same algorithm for t_l and t_r, starting from step 2.

The discriminators can be implemented by a simple perceptron or by a feed-forward network.

Classification Algorithm

The class of a pattern $\boldsymbol{I}_k$ can be established by the following algorithm:

1. Set $t = root$;
2. If the node t is a leaf, classify $\boldsymbol{I}_k$ by the associated label l_k and stop, otherwise:
   (a) If $D_t(\boldsymbol{I}_k) = 0$ then set $t = t_l$;
   (b) otherwise set $t = t_r$;
3. go to step 2.

Fig. 17. The learning and classification algorithms for binary neural trees.

TABLE I
DESCRIPTION OF A SET OF CLASSIFICATION PROBLEMS INVOLVING LOGIC TERMS

Problem	Symbols	Positive Examples.	#terms (tr.,test)	#subterms (tr.,test)	depth (pos.,neg.)
lblocc1 long	f/2 i/1 a/0 b/0 c/0	no occurrence of label c	(259,141)	(444,301)	(5,5)
termocc1	f/2 i/1 a/0 b/0 c/0	the (sub)terms i(a) or f(b,c) occur somewhere	(173,70)	(179,79)	(2,2)
termocc1 very long	f/2 i/1 a/0 b/0 c/0	the (sub)terms i(a) or f(b,c) occur somewhere	(280,120)	(559,291)	(6,6)
inst1	f/2 a/0 b/0 c/0	instances of f(X,X)	(200,83)	(235,118)	(3,2)
inst1 long	f/2 a/0 b/0 c/0	instances of f(X,X)	(202,98)	(403,204)	(6,6)
inst4	f/2 a/0 b/0 c/0	instances of f(X,f(a,Y))	(175,80)	(179,97)	(3,2)
inst4 long	f/2 a/0 b/0 c/0	instances of f(X,f(a,Y))	(290,110)	(499,245)	(7,6)
inst7	t/3 f/2 g/2 i/1 j/1 a/0 b/0 c/0 d/0	instances of t(i(X),g(X,b),b)	(191,109)	(1001,555)	(6,6)

that the total number of subterms for the training set will be equal to the number of vertices of the minimal DAG obtained after the optimization of the training set. Finally, the maximal depth of the terms, both in the training and test sets, is six.

For each problem about the same number of positive and negative examples is given. Both positive and negative examples were generated randomly. Training and test sets are disjoint and were generated by the same algorithm. Some examples of logic terms represented as trees are shown in Figs. 18–20.

Note that the set of proposed problems ranges from the detection of a particular atom (label) in a term to the satisfaction of a specific unification pattern. Specifically, in the unification patterns for the problems `inst1` and `inst1_long`, the variable X occurs twice making these problems much more difficult than `inst4_long`, because any classifier for these problems would have to compare arbitrary subterms corresponding to X.

As a final remark, it must be pointed out that the number of training examples is a very small fraction of the total number of terms characterizing each problem. In fact, the total number of distinct terms belonging to a problem is defined both by the number and arity of the symbols, and by the maximal depth allowed for the terms. Making some calculations, it

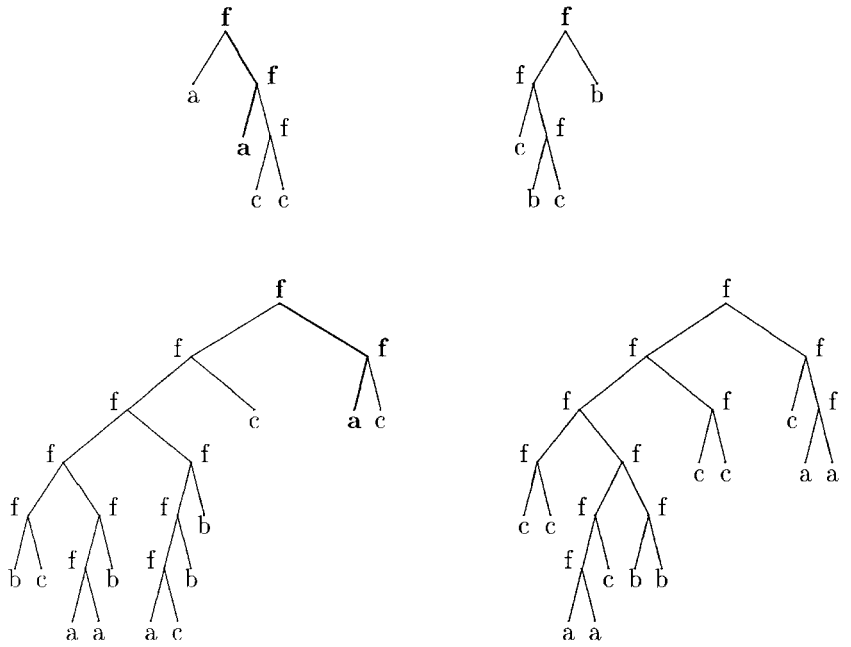


Fig. 18. Samples of trees representing terms for problem `inst4_long`. The positive terms are shown on the left, while negative terms are shown on the right. The vertices and edges characterizing the positive terms are drawn in bold.

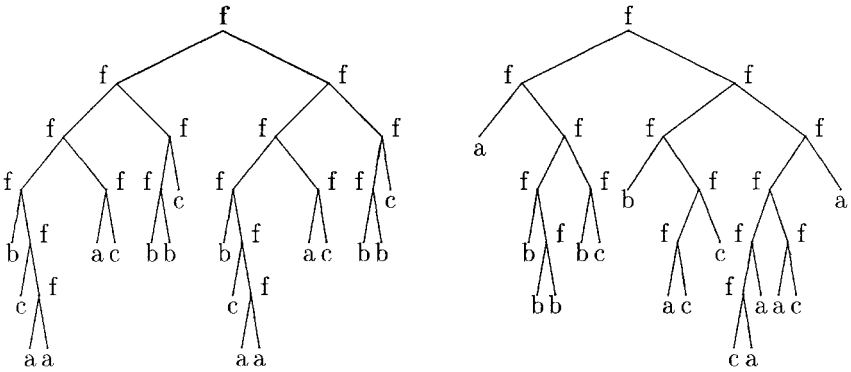


Fig. 19. Samples of trees representing terms for problem `inst1_long`. The positive term is shown on the left, while the negative term is shown on the right.

is not difficult to compute that the total number of terms allowed for problem `inst1` is equal to 21 465. This number, however, grows exponentially with the maximal depth, e.g., if the maximal depth is set to four, the total number of distinct terms is larger than 10^8. This gives an idea of how small is the training set for problem `inst1_long`, where the maximal depth is six.

B. LRAAM-Based Networks

Table III reports the best results we obtained for the majority of the problems described in Table I, over four different network settings (both in number of hidden units for the LRAAM and learning parameters) for the LRAAM-network with a single unit as classifier. The simulations were stopped after 30 000 epochs, apart from problem `inst1_long` for which we used a bound of 80 000 epochs, or when the classification problem over the training set has been completely solved. We made no extended effort to optimize the size of the network and the learning parameters, thus it should be possible to improve on the reported results. The first column of the table shows the name of the problem, the second the number of units used to represent the labels, the third the number of hidden units, the fourth the learning parameters (η is the learning parameter for the LRAAM, ϵ the learning parameter for the classifier, μ the momentum for the LRAAM), the fifth the percentage of terms in the training set which the LRAAM was able to properly encode and decode, the sixth the percentage of terms in the training set correctly classified, the seventh the percentage of terms in the test set correctly classified, and the eighth the number of epochs the network employed to reach the reported performances.

The results highlight that some problems get a very satisfactory solution even if the LRAAM performs poorly. Moreover, this behavior does not seem to be related to the complexity of the classification problem, since both problems involving the simple detection of an atom (label) in the terms

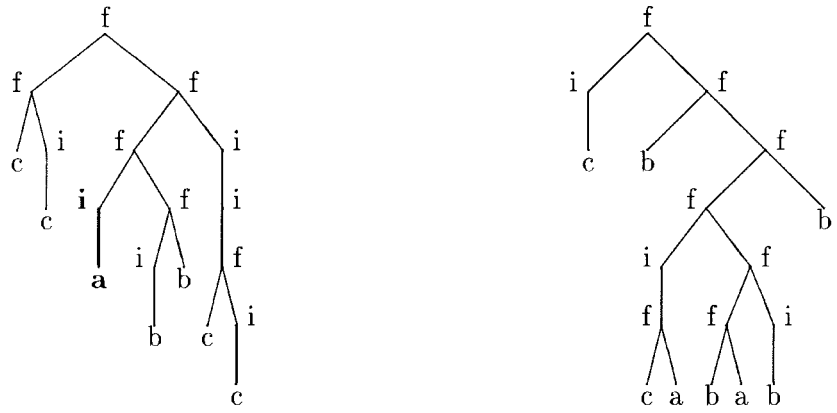


Fig. 20. Samples of trees representing terms for problem `termocc1_very-long`. The positive term is shown on the left, while the negative term is shown on the right.

TABLE II
RESULTS OBTAINED ON THE TEST SETS FOR SOME OF THE CLASSIFICATION PROBLEMS USING LRAAM-BASED NETWORKS WITH DIFFERENT NUMBERS OF HIDDEN UNITS AND DIFFERENT SETTINGS FOR THE LEARNING PARAMETERS

Problem	# Trials	% Training Mean (Min - Max)	% Test Mean (Min - Max)
lblocc1 long	6	99.42 (98.07 - 100)	96.45 (94.33 - 98.58)
termocc1	7	98.51 (96.53 - 99.42)	87.79 (78.57 - 94.29)
inst1	6	96.58 (96 - 97.5)	89.16 (83.13 - 93.98)
inst1 long	5	92.77 (89.11 - 96.04)	86.12 (80.61 - 90.82)
inst4	3	100 (100 - 100)	97.5 (96.25 - 100)
inst4 long	3	100 (100 - 100)	98.48 (97.27 - 100)
inst7	7	100 (100 - 100)	98.16 (96.33 - 100)

(`lblocc1_long`) and problems involving the satisfaction of a specific unification rule (`inst4_long`, `inst7`) can be solved without a fully developed LRAAM. Thus, it is clear that the classification of the terms is exclusively based on the encoding power of the LRAAM's encoder, which is influenced both by the LRAAM error and the classification error. However, even if the LRAAM's decoder is not directly involved in the classification task, it helps the classification process since it forces the network to generate different representations for terms in different classes.[11]

In order to give a feeling of how learning proceeds, the performance of the networks during training is shown for the problems `termocc1`, `inst4` and `inst4_long` in Figs. 21–23, where the encoding–decoding performance curve of the LRAAM on the training set is reported, together with the classification curves on the training and test set.

Reduced Representations for Classification: In this section we briefly discuss the representational differences between a basic LRAAM (without a classifier) and the architecture we used. The basic LRAAM organizes the representational space in such a way that similar structures get similar reduced representations (see [28] for more details). This happens because, although the LRAAM is trained in supervised mode both on the output of the network and on the relationships among components (i.e., the information about the pointers), the network is auto-associative and thus it decides by itself the representation for the pointers. Consequently, the learning mode for the LRAAM is on the whole unsupervised. When a classifier is introduced (as in our system), the training of the LRAAM is no longer unsupervised, since the error of the classifier constrains the learning. The resulting learning regime is somewhere between an unsupervised and a supervised mode.

In order to understand the differences between representations devised by a basic LRAAM and the ones devised in the present paper, we trained a basic LRAAM (of the same size as the LRAAM used by our network) on the training set of `inst1`, then we computed the first, second, and third principal components of the reduced representations obtained both for the training and test sets.[12] These principal components are plotted in Fig. 24. Note that the representations obtained mainly cluster themselves in specific points of the space. Terms of the same depth constitute a single cluster, and terms of different depths are in different clusters. The same plot for the reduced representations devised by our network (as from Table III, row 3) is presented in Fig. 25. The overall differences with respect to the basic LRAAM plot consist in a concentration of more than half (57%) of the positive examples of the training and test sets in a well defined cluster, while the remaining representations are spread over two main subspaces. The well-defined cluster can be understood as the set of representations for which there was no large-scale interference between the decoder of the LRAAM and the classifier (this allowed the formation of the cluster), while the remaining representations do not preserve the cluster structure since they have to satisfy competitive constraints deriving from the classifier and the decoder. Specifically, the classifier tends to cluster the representations into two well-defined clusters (one for each class), while the LRAAM decoder tends to develop

[11] The decoder error forces the LRAAM network to develop a different representation for each term, however, when the error coming from the classifier is very strong, terms in the same class may get almost identical representations.

[12] We considered only the reduced representations for terms. No reduced representation for subterms was included in the analysis.

TABLE III
THE BEST RESULTS OBTAINED FOR SOME OF THE CLASSIFICATION PROBLEMS BY AN LRAAM-BASED NETWORK

Problem	#L-unit	#H-unit	Learning Par.	% Enc.-Dec.	% Tr.	% Ts.	#epochs
lblocc1 long	8	35	$\eta = 0.2,\ \epsilon = 0.001,\ \mu = 0.5$	4.25	100	98.58	11951
termocc1	8	25	$\eta = 0.1,\ \epsilon = 0.1,\ \mu = 0.2$	100	98.84	94.29	27796
inst1	6	35	$\eta = 0.2,\ \epsilon = 0.06,\ \mu = 0.5$	100	97	93.98	10452
inst1 long	6	45	$\eta = 0.2,\ \epsilon = 0.005,\ \mu = 0.5$	36.14	94.55	90.82	80000
inst4	6	35	$\eta = 0.2,\ \epsilon = 0.005,\ \mu = 0.5$	98.86	100	100	1759
inst4 long	6	35	$\eta = 0.2,\ \epsilon = 0.005,\ \mu = 0.5$	8.97	100	100	6993
inst7	13	40	$\eta = 0.1,\ \epsilon = 0.01,\ \mu = 0.2$	1.05	100	100	6158

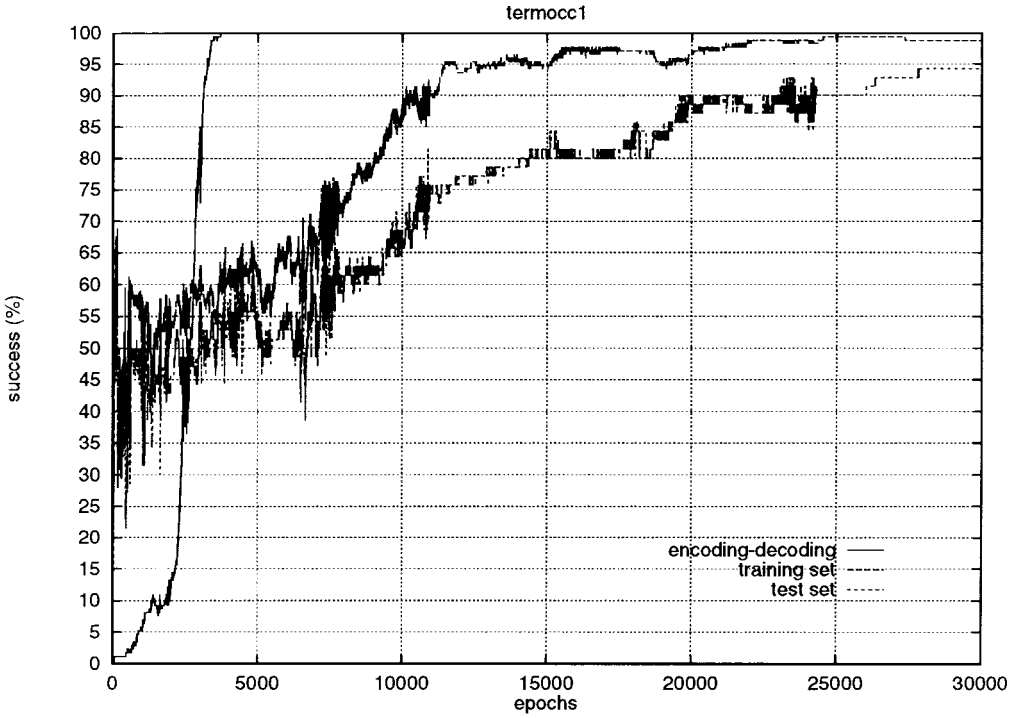


Fig. 21. Performance curves for `termocc1`.

distinct reduced representations since they must be decoded into different terms.

The above considerations on the final representations for the terms are valid only if the LRAAM reaches a good encoding-decoding performance on the training set. However, as reported in Table III, some classification problems can be solved even if the LRAAM performs poorly. In this case, the reduced representations contain almost exclusively information about the classification task. Figs. 26 and 27 report the results of a principal components analysis on the representations developed for the problems `inst4_long` and `inst7`, respectively. In the former, the first and second principal components suffice for a correct solution of the classification problem. In the latter, the second principal component alone gives enough information for the solution of the problem. Moreover, notice how the representations developed for `inst7` clustered with smaller variance than the representations developed for `inst4_long`, and how this is in accordance with the better performance in encoding-decoding of the latter than the former. Of course, this does not constitute enough evidence to conclude that the relationship between the variance of the clusters and the performance of the LRAAM is demonstrated. However, it does seem to be enough to call for a more accurate study on this issue.

C. Cascade-Correlation for Structures

The results obtained by cascade-correlation for structures, shown in Table IV, are obtained for a subset of the problems using a pool of eight units. The networks used have both triangular and diagonal recursive connections matrices and *no connections between hidden units*. We decided to remove the connections between hidden units to reduce the probability of overfitting.

We made no extended effort to optimize the learning parameters and the number of units in the pool, thus it should be possible to significantly improve on the reported results.

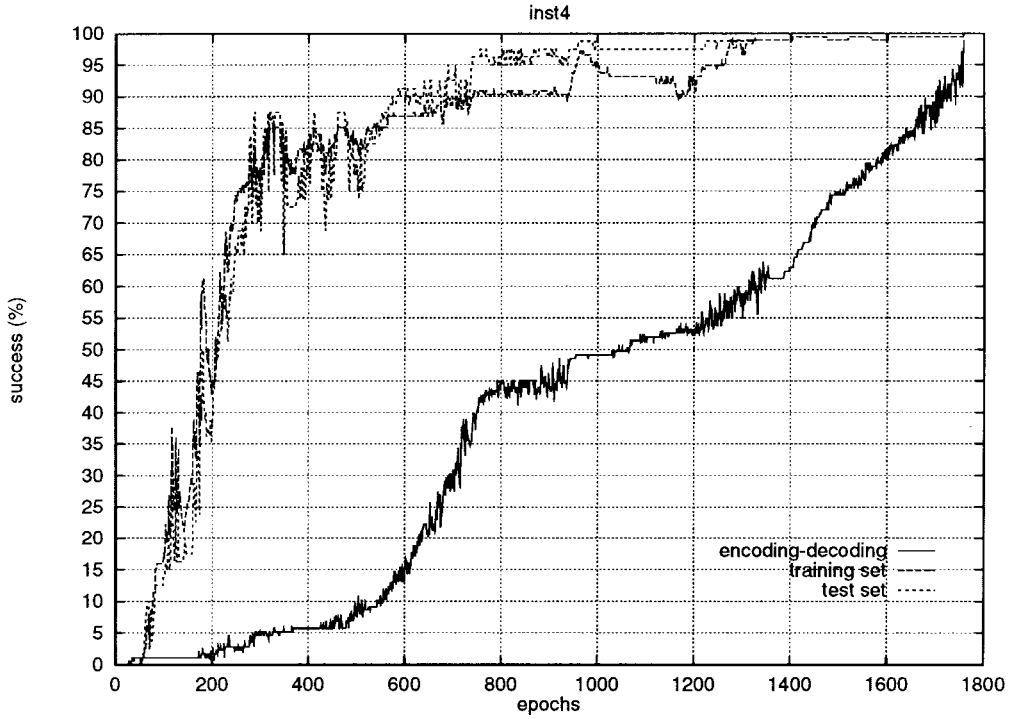


Fig. 22. Performance curves for `inst4`.

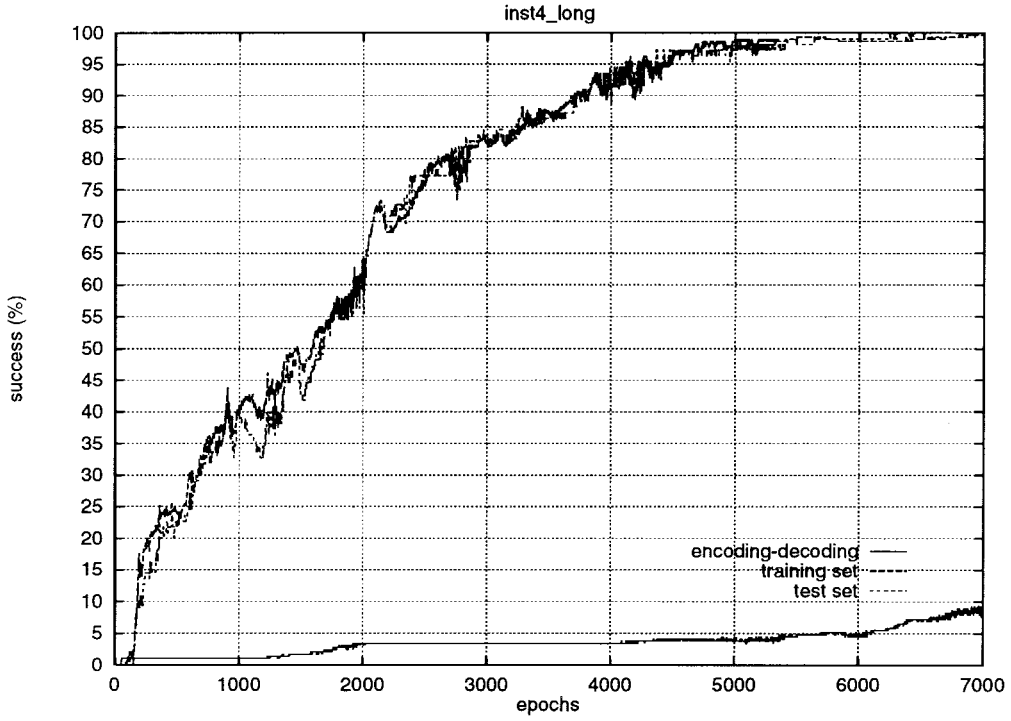


Fig. 23. Performance curves for `inst4_long`.

VII. Conclusion

We have proposed a generalization of the standard neuron, namely the generalized recursive neuron, for extending the computational capability of neural networks to the processing of structures. On the basis of the generalized recursive neuron, we have shown how several of the learning algorithms defined for standard neurons, can be adapted to deal with structures. We believe that other learning procedures, besides those covered in this paper, can be adapted as well.

The proposed approach to learning in structured domains can be adopted for automatic inference in syntactic and structural pattern recognition [10], [13]. Specifically, in this paper, we have demonstrated the possibility to perform classification

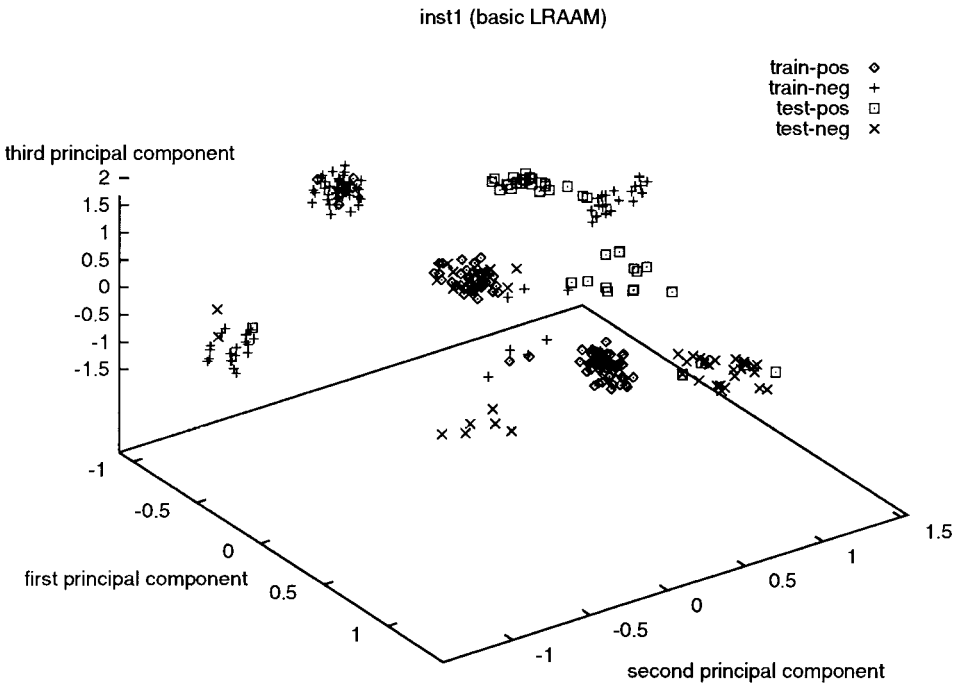


Fig. 24. The first, second, and third principal components of the reduced representations, devised by a basic LRAAM on the training and test sets of the `inst1` problem, yield a nice three-dimensional view of the term's representations.

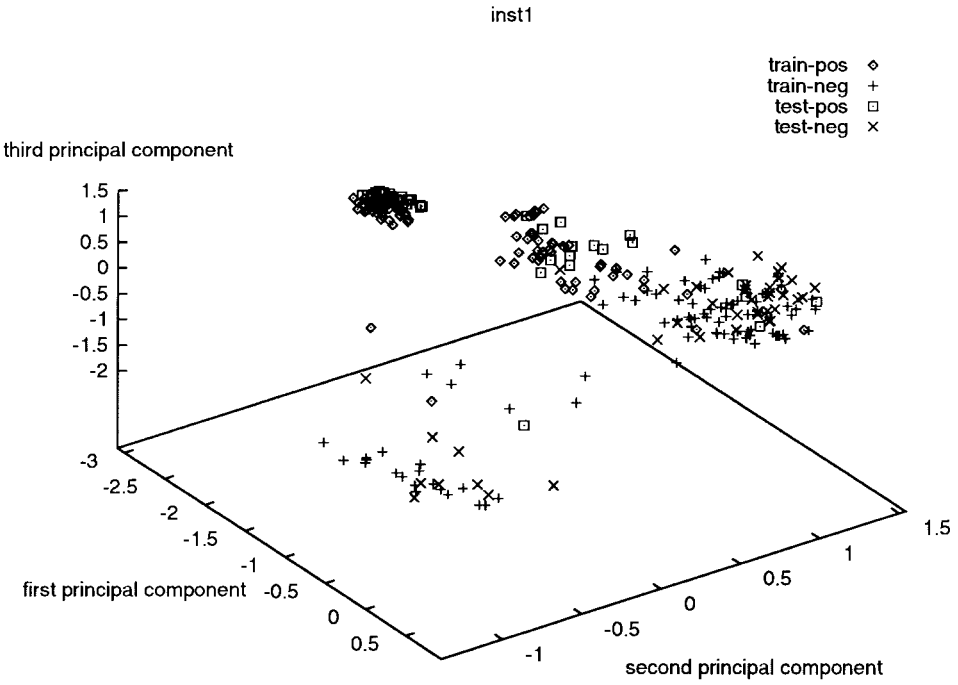


Fig. 25. Results of the principal components analysis (first, second, and third principal components) of the reduced representations developed by the proposed network (LRAAM + Classifier) for the `inst1` problem.

tasks involving logic terms. Note that automatic inference can also be obtained by using *inductive logic programming* [17]. The proposed approach, however, has its own specific peculiarity, since it can approximate functions from a structured domain (which may have real-valued vectors as labels) to the reals. Specifically, we believe that the proposed approach can fruitfully be applied to molecular biology and chemistry (QSPR and QSAR), where it can be used for the automatic determination of *topological indexes* [14], [22], which are usually designed through a very expensive trial and error approach.

In conclusion, the proposed architectures extend the processing capabilities of neural networks, allowing the processing of structured patterns which can be of variable size and

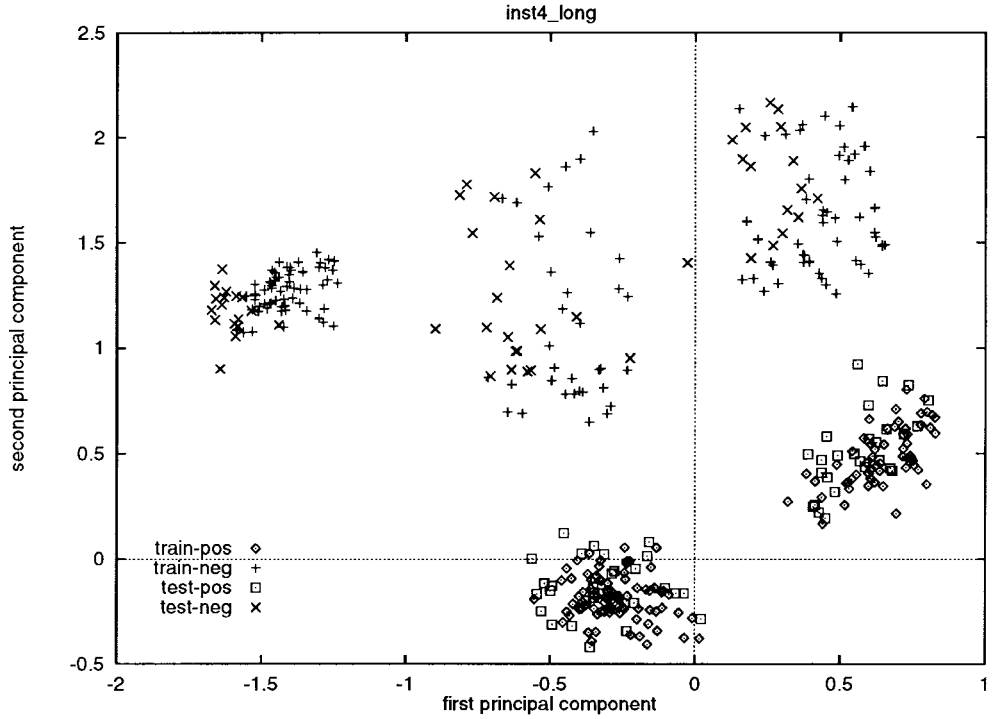


Fig. 26. Results of the principal components analysis (first and second principal components) of the reduced representations developed by the proposed network (LRAAM + Classifier) for the `inst4-long` problem. The resulting representations are clearly linearly separable.

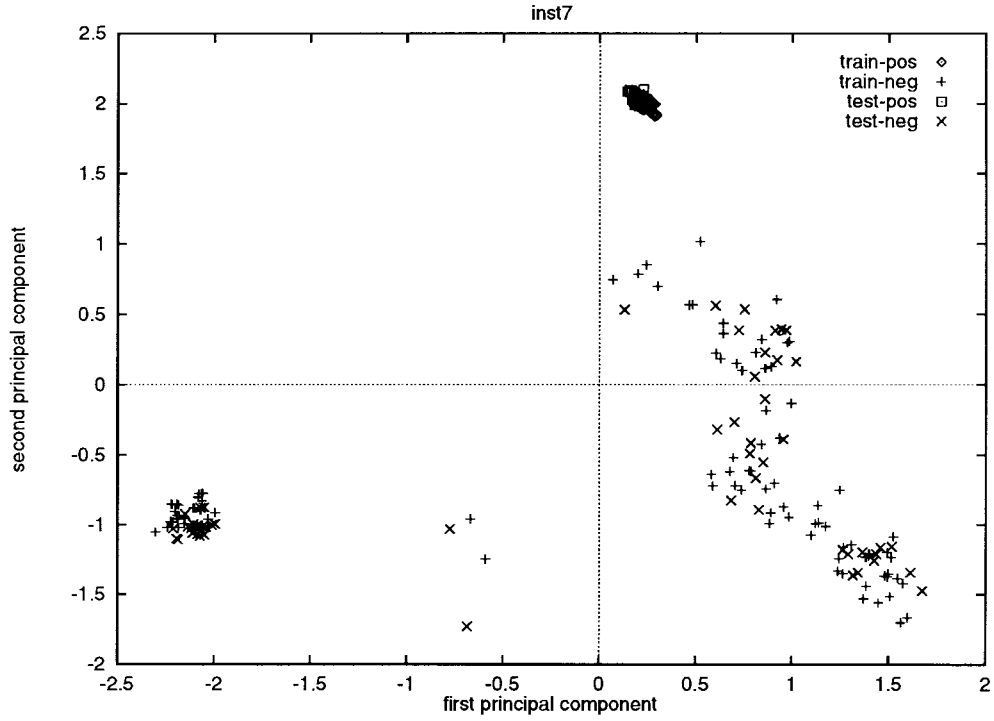


Fig. 27. Results of the principal components analysis (first, and second principal component) of the reduced representations developed by the proposed network (LRAAM + Classifier) for the `inst7` problem. The resulting representations can be separated using only the second principal component.

complexity. However, some of the proposed architectures do nevertheless have computational limitations. For example, cascade-correlation for structures has computational limitations due to the fact that frozen hidden units cannot receive input from hidden units introduced after their insertion into the network. These limitations, in the context of standard recurrent cascade-correlation (RCC), have been discussed in [11], where it is demonstrated that certain finite state automata cannot be implemented by networks built by RCC using monotone activation functions. Since our algorithm reduces

TABLE IV
RESULTS OBTAINED ON THE TEST SETS FOR SOME OF THE CLASSIFICATION PROBLEMS USING BOTH NETWORKS WITH TRIANGULAR AND DIAGONAL RECURSIVE CONNECTION MATRICES. THE SAME NUMBER OF UNITS (8) IN THE POOL WAS USED FOR ALL NETWORKS

	Problem	# Label Units	# Trials	# Hidden Units Mean (Min - Max)	% Test Mean (Min - Max)
Triangular	termocc1 very-long	8	4	13.25 (8 - 19)	97.70 (95 - 100)
	inst4 long	6	5	7.2 (4 - 12)	99.64 (99.09 - 100)
	inst1	6	9	11.33 (7 - 19)	90.09 (86.75 - 92.77)
	inst1 long	6	16	7.81 (6 - 11)	91.65 (88.87 - 94.89)
Diagonal	termocc1 very-long	8	3	11 (8 - 16)	96.94 (95 - 99.17)
	inst4 long	6	3	12.66 (9 - 15)	98.48 (97.27 - 100)
	inst1	6	5	12.4 (7 - 21)	90.6 (87.95 - 92.77)
	inst1 long	6	3	18.66 (17 - 21)	82.99 (75.51 - 91.84)

to standard RCC when considering sequences, it follows that it has limitations as well.

APPENDIX
ALGORITHM FOR THE ADDITION OF A SUPERSOURCE

In the following we define the algorithm for the addition of a supersource to a graph. The algorithm uses the concepts of strongly connected components of a graph and the related definition of a component graph (see Section V-B.2 for formal definitions).

A. Algorithm Supersource

Input: A graph $G = (V, E)$
Output: A graph $G' = (V', E')$ with supersource s
Begin

- Compute the component graph G^{SCC} of G;
- Define the set $\hat{V}^{\mathrm{SCC}} = \{u \in V^{\mathrm{SCC}} \mid \mathrm{in\_degree}(u) = 0\}$;
- Define for each vertex $u \in \hat{V}^{\mathrm{SCC}}$ the set S_u containing all the vertices (in V) within the strongly connected component corresponding to u;
- For each S_u define $r_u = \mathrm{select\_representative}(S_u)$, i.e., $r_u \in S_u$ is a vertex representing the strongly connected component corresponding to u;
- **if** ($\hat{V}^{\mathrm{SCC}} = \{u\}$ (i.e., $\#\hat{V}^{\mathrm{SCC}} = 1$))
  then $s = r_u$ is the supersource, $V' = V$, and $E' = E$;
  else
  - Create a new vertex s (the supersource) and set $V' = V \cup \{s\}$;
  - Set $E' = E \cup E^+$, where $E^+ = \{(s, r_u) \mid u \in \hat{V}^{\mathrm{SCC}}\}$.

End.

In the algorithm above we left the procedure $\mathrm{select\_representative}(S_u)$ undefined since it can be defined in different ways according to the complexity of the structured domain. For example, if the labels which appear in a graph are all distinct, then we can define a total order $\succeq$ on Σ (the set of labels) and note that the set

$$S_u^1 = \{x \in S_u \mid \phi_G(x) \succeq \phi_G(x') \quad \text{for each } x' \in S_u\}$$

contains a single element which can be used as representative (i.e., r_u). If the same label can have multiple occurrences, we can use information on the out-degree of the vertices

$$S_u^2 = \{x \in S_u^1 | \mathrm{out\_degree}(x) \geq \mathrm{out\_degree}(x') \text{ for each } x' \in S_u^1\}$$

and further on the in-degree of the vertices

$$S_u^3 = \{x \in S_u^2 | \mathrm{in\_degree}(x) \geq \mathrm{in\_degree}(x') \text{ for each } x' \in S_u^2\}.$$

If the set S_u^3 still contains more than one vertex, then it can be refined by resorting to more sophisticated criteria (see [31]).

Theorem 1: The algorithm Supersource applied to a graph $G = (V, E)$ adds to E the minimum number of edges to create a supersource.

Proof: If $\#\hat{V}^{\mathrm{SCC}} = 1$ no edge is added by the algorithm Supersource. Otherwise both a new vertex s, the supersource, and a new edge for each strongly connected component u of G with $\mathrm{in\_degree}(u) = 0$ are created. The vertex s is a supersource, since, by definition, all the verices in V can be reached by a path starting from s. Moreover, if one edge leaving from s is removed, say (s, r_u), the vertices belonging to the strongly connected component u cannot be reached by any other vertex since $\mathrm{in\_degree}(u) = 0$. This demonstrates that all the added edges are needed. □

LABELING RAAM

The *labeling RAAM (LRAAM)* [27], [28], [30], is an extension of the RAAM model [21] which allows one to encode

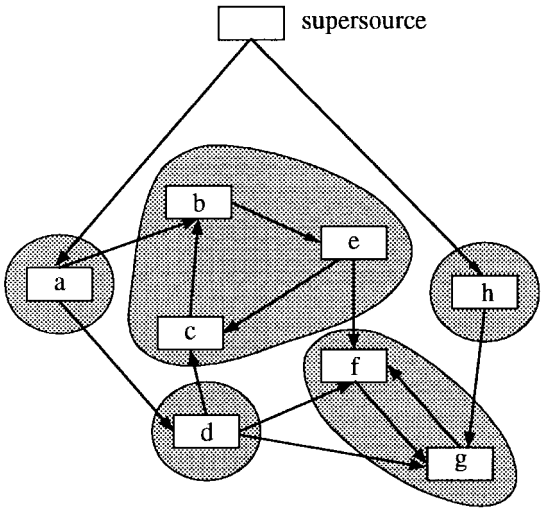


Fig. 28. The result of algorithm Supersource applied to the graph shown in Fig. 12(a).

labeled structures. The general structure of the network for an LRAAM is shown in Fig. 29.

The network is trained by backpropagation to learn the identity function. The idea is to obtain a compressed representation (hidden layer activation) of a node of a labeled directed graph by allocating a part of the input (output) of the network to represent the label (N_l units), and the rest to represent one or more pointers. This representation is then used as a pointer to the node. To allow the recursive use of these compressed representations, the part of the input (output) layer which represents a pointer must be the same size as the hidden layer (N_h units). Thus, a general LRAAM is implemented by an $N_I - N_h - N_I$ feedforward network, where $N_I = N_l + nN_h$, and n is the number of pointer fields.

Labeled directed graphs can easily be encoded using an LRAAM. Each node in the graph only needs to be represented as a record, with one field for the label and one field for each pointer to a connected node. The pointers only need to be logical pointers, since their actual values will be the patterns of hidden activation in the network. At the beginning of learning, their values are set at random. A graph is represented by a list of these records, and this list is the initial training set for the LRAAM. During training the representations of the pointers are continuously updated depending on the hidden activations. Consequently, the training set is dynamic.

For example, the network for the graph shown in Fig. 30 can be trained as follows:

input		hidden		output
$(L_1\ P_{n_2}(t) P_{n_3}(t))$	$\rightarrow$	$P'_{n_1}(t)$	$\rightarrow$	$(L''_1(t)\ P''_{n_2}(t) P''_{n_3}(t))$
$(L_2\ P_{n_3}(t)\ P_{n_4}(t))$	$\rightarrow$	$P'_{n_2}(t)$	$\rightarrow$	$(L''_2(t)\ P''_{n_3}(t)\ P''_{n_4}(t))$
$(L_3\ P_{n_1}(t)\ P_{n_5}(t))$	$\rightarrow$	$P'_{n_3}(t)$	$\rightarrow$	$(L''_3(t)\ P''_{n_1}(t) P''_{n_5}(t))$
$(L_4\ \mathrm{nil}_1(t)\mathrm{nil}_2(t))$	$\rightarrow$	$P'_{n_4}(t)$	$\rightarrow$	$(L''_4(t)\ \mathrm{nil}''_1(t)\ \mathrm{nil}''_2(t))$
$(L_5\ P_{n_4}(t)\mathrm{nil}_3(t))$	$\rightarrow$	$P'_{n_5}(t)$	$\rightarrow$	$(L''_5(t)\ P''_{n_4}(t)\ \mathrm{nil}''_3(t))$

where L_i and P_{n_i} are the label of and the pointer to the ith node, respectively; and t represents the time, or epoch, of training. At the beginning of training $(t = 1)$ the representations for the nonvoid pointers $(P_{n_i}(1))$ and void pointers $(\mathrm{nil}_i(1))$ in the training set are set at random. After each epoch, the representations for the nonvoid pointers in the training set are updated depending on the hidden activation obtained in the previous epoch for each pattern: $\forall i\ P_{n_i}(t+1) = P'_{n_i}(t)$. The void representations are, on the other hand, copied from the output: $\mathrm{nil}_i(t+1) = \mathrm{nil}''_i(t)$.

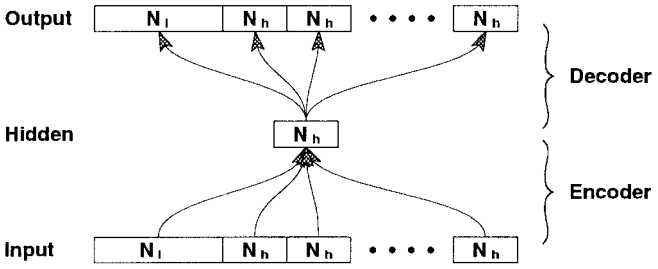


Fig. 29. The network for a general LRAAM. The first layer in the network implements an encoder; the second layer, the corresponding decoder.

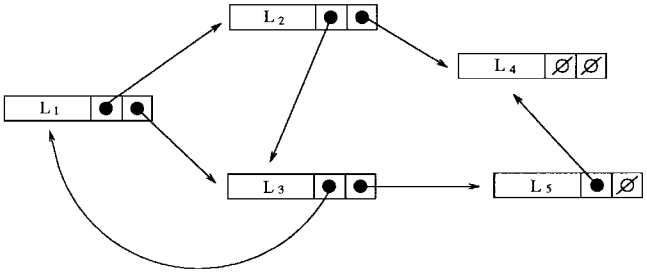


Fig. 30. An example of a labeled directed graph.

If the backpropagation algorithm converges to perfect learning, i.e., the total error goes to zero, it can be stated that

$$\begin{array}{cccc} L_1 = L''_1 & L_2 = L''_2 & L_3 = L''_3 & L_4 = L''_4 \\ L_5 = L''_5 & P_{n_2} = P''_{n_2} & P_{n_3} = P''_{n_3} & P_{n_4} = P''_{n_4} \\ P_{n_5} = P''_{n_5} & \mathrm{nil}_1 = \mathrm{nil}''_1 & \mathrm{nil}_2 = \mathrm{nil}''_2 & \mathrm{nil}_3 = \mathrm{nil}''_3. \end{array}$$

Once training is complete, the patterns of activation representing pointers can be used to retrieve information. Thus, for example, if the activity of the hidden units of the network is clamped to P_{n_1}, the output of the network becomes (L_1, P_{n_2}, P_{n_3}), thus enabling further retrieval of information by decoding P_{n_2} or P_{n_3}, and so on. Note that multiple labeled directed graphs can be encoded in the same LRAAM.

When the LRAAM is used inside an LRAAM-based network for the classification of terms, it is more convenient to force a single representation for the void pointer, i.e., a null vector. This allows the network to encode graphs which do not belong to the training set without looking for a valid representation for the void pointer.

ACKNOWLEDGMENT

The authors would like to thank C. Goller for the generation of the training and test sets used in this paper and D. Majidi for the implementation of the cascade-correlation algorithm.

REFERENCES

[1] L. B. Almeida, "A learning rule for asynchronous perceptrions with feedback in a combinatorial environment," in *Proc. IEEE 1st Annu. Int. Conf. Neural Networks*, M. Caudil and C. Butler, Eds. New York: IEEE Press, 1987, pp. 609–618.

[2] A. Atiya, "Learning on a general network," in *Neural Information Processing Systems*, D. Z. Anderson, Ed. New York: AIP, 1988, pp. 22–30.
[3] L. Atlas *et al.*, "A performance comparison of trained multilayer perceptrons and trained classification trees," *Proc. IEEE*, vol. 78, pp. 1614–1619, 1992.
[4] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Belmont, CA: Wadsworth, 1984.
[5] M. A. Cohen and S. Grossberg, "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-13, pp. 815–826, 1983.
[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
[7] J. L. Elman, "Finding structure in time," *Cognitive Sci.*, vol. 14, pp. 179–211, 1990.
[8] S. E. Fahlman, "The recurrent cascade-correlation architecture," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-91-100, 1991.
[9] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, vol. 2, 1990, pp. 524–532.
[10] K. S. Fu, *Syntactical Pattern Recognition and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
[11] C. L. Giles, D. Chen, G. Z. Sun, H. H. Chen, Y. C. Lee, and M. W. Goudreau, "Constructive learning of recurrent neural networks: Limitations of recurrent casade correlation and a simple solution," *IEEE Trans. Neural Networks*, vol. 6, pp. 829–836, 1995.
[12] C. Goller and A. Küchler, *Learning Task-Dependent Distributed Structure-Representations by Backpropagation Through Structure*, Institut für Informatik, Technische Universität München, Germany, AR Rep. AR-95-02, 1995.
[13] R. C. Gonzalez and M. G. Thomason, *Syntactical Pattern Recognition*. Reading, MA: Addison-Wesley, 1978.
[14] L. H. Hall and L. B. Kier, "Reviews in computational chemistry," *The Molecular Connectivity Chi Indexes and Kappa Shape Indexes in Structure-Property Modeling*. New York: VCH, 1991, ch. 9, pp. 367–422.
[15] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," in *Proc. Nat. Academy Sci.*, 1984, pp. 3088–3092.
[16] T. Li, L. Fang, and A. Jennings, "Structurally adaptive self-organizing neural trees," in *Proc. Int. Joint Conf. Neural Networks*, 1992, pp. 329–334.
[17] S. Muggleton and L. De Raedt, "Inductive logic programming: Theory and methods," *J. Logic Programming*, vol. 19, no. 20, pp. 629–679, 1994.
[18] M. P. Perrone, "A soft-competitive splitting rule for adaptive tree-structured neural networks," in *Proc. Int. Joint Conf. Neural Networks*, 1992, pp. 689–693.
[19] M. P. Perrone and N. Intrator, "Unsupervised splitting rules for neural tree classifiers," in *Proc. Int. Joint Conf. Neural Networks*, 1992, pp. 820–825.
[20] F. J. Pineda, "Dynamics and architecture for neural computation," *J. Complexity*, vol. 4, pp. 216–245, 1988.
[21] J. B. Pollack, "Recursive distributed representations," *Artificial Intell.*, vol. 46, nos. 1–2, pp. 77–106, 1990.
[22] D. H. Rouvray, *Computational Chemical Graph Theory*. New York: Nova Sci., 1990, p. 9.
[23] D. E. Rumelhart and J. L. McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Cambridge, MA: MIT Press, 1986.
[24] A. Sankar and R. Mammone, "Neural tree networks," *Neural Networks: Theory and Applications*. New York: Academic, 1991, pp. 281–302.
[25] I. K. Sethi, "Entropy nets: From decision trees to neural networks," *Proc. IEEE*, vol. 78, pp. 1605–1613, 1990.
[26] J. A. Sirat and J.-P. Nadal, "Neural trees: A new tool for classification," *Network*, vol. 1, pp. 423–438, 1990.
[27] A. Sperduti, "Encoding of labeled graphs by labeling RAAM," in *Advances in Neural Information Processing Systems*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds. San Mateo, CA: Morgan Kaufmann, vol. 6, pp. 1125–1132, 1994.
[28] ———, "Labeling RAAM," *Connection Sci.*, vol. 6, no. 4, pp. 429–459, 1994.
[29] ———, "Stability properties of labeling recursive autoassociative memory," *IEEE Trans. Neural Networks*, vol. 6, pp. 1452–1460, 1995.
[30] A. Sperduti and A. Starita, "An example of neural code: Neural trees implemented by LRAAM's," in *Proc. Int. Conf. Neural Networks Genetic Algorithms*, Innsbruck, Austria, 1993, pp. 33–39.
[31] ———, "Encoding of graphs for neural network processing," Dipartimento di Informatica, Università di Pisa, Italy, Tech. Rep., 1996.
[32] A. Sperduti, A. Starita, and C. Goller, "Fixed length representation of terms in hybrid reasoning systems, report i: Classification of ground terms," Dipartimento di Informatica, Università di Pisa, Italy, Tech. Rep. TR-19/94, 1994.
[33] A. Sperduti, A. Starita, and C. Goller, "Learning distributed representations for the classification of terms," in *Proc. Int. Joint Conf. Artificial Intell.*, 1995, pp. 509–515.
[34] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computa.*, vol. 1, pp. 270–280, 1989.

Alessandro Sperduti received the "laurea" and Doctoral degrees in 1988 and 1993, respectively, all in computer science from the University of Pisa, Italy.

In 1993 he spent a period at the International Computer Science Institute, Berkeley, supported by a postdoctoral fellowship. In 1994 he returned to the Computer Science Department, University of Pisa, where he is presently Assistant Professor. His research interests include data sensory fusion, image processing, neural networks, hybrid systems. In the field of hybrid systems his work has focused on the integration of symbolic and connectionist systems.

Dr. Sperduti has contributed to the organization of several workshops on this subject and also served on the program committee of conferences on Neural Networks.

Antonina Starita (A'91–M'96) received the "laurea" degree in physics from the University of Naples, Italy, and the Doctoral degree in computer science from the University of Pisa, Italy.

She was Research Fellow of the Italian National Council of Research at the Information Processing Institute of Pisa and then she became a Researcher for the same institution. Since 1980, she has become Associate Professor at the Computer Science Department of the University of Pisa, where she is in charge of the two regular courses, "Knowledge Acquisition and Expert Systems" and "Neural Networks," at the University of Pisa. She has been active in the areas of biomedical signal processing, rehabilitation engineering, and biomedical applications of signal processing, responsible for many research projects at the national and international level. Her current scientific interests are now shifted to AI methodologies, robotics, neural networks, hybrid systems, and sensory integration in artificial systems.

Dr. Starita is member of the Italian Medical and Biological Engineering Society, the International Medical and Biological Engineering Society, and of the INNS (International Neural Network Society).