Neural Networks for Processing Data Structures

Alessandro Sperduti

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, I-56125 Pisa, Italy

Abstract. In recent years neural networks for the representation and processing of structures have been developed. Besides to be useful in application domains where data is structured, such as Chemistry, Artificial Intelligence Systems, Structural Pattern Recognition, these kind of networks may turn out to be relevant for the development of Hybrid Systems combining symbolic and sub-symbolic approaches, since they allow the treatment of structured information very naturally and, in several cases, very efficiently. We present the basic concepts underpinning this class of networks and discuss computational and complexity issues.

1 Introduction

In recent years, neural networks for the representation and processing of structures have been developed (see, for example, [Pol90, SSG95, GK96, SS97]). These kind of networks are of paramount importance both for Structural Pattern Recognition and for the development of Hybrid Systems, since they allow the treatment of structured information very naturally and, in several cases, very efficiently. The purpose of this tutorial is to examine the current state of the art for these connectionist networks.

1.1 Motivation

The main motivations for using neural networks (NN) for the processing of structures are the following:

- NN are universal approximators;
- NN can perform automatic inference (learning);
- NN hold very good classification capabilities;
- NN can deal with noise and incomplete data.

Specifically, the above properties are particularly useful when facing classification problems in a structured domain.

Let consider, for example, the classification of labeled graphs which represent chemical compounds. The standard approach consists in encoding each graph X as a fixed-size vector which is then given as input to a feed-forward 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, e.g., in molecular biology and chemistry, the encoding process is performed through the definition of *topological indexes* [HK91], which are designed by a very expensive trial and error approach.

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

(i) the relevance of different features of the graphs may change dramatically for different learning tasks;
(ii) 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, the encoding process should be adapted 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.

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. We will show how it is possible to formalize several supervised models for classification of structures which stem very naturally from well known models, such as Back-Propagation Through Time networks, Real-Time Recurrent networks, Recurrent Cascade Correlation networks, and Neural Trees.

On the computational side, it is well known that recurrent neural networks can simulate any finite state automata [ADO91, OG96] as well as any multi-stack Turing machine in real time [SS95]. When constraining the network architecture, however, this computational power may no longer hold. For example, Elman-style Recurrent Networks can simulate any finite state automata [GGCC94, Kre95], while Recurrent Cascade-Correlation cannot [GCS+95, Kre96]. For this reason, it is of paramount importance to assess the computational power of a given network architecture, since this characterizes the class of functions which, in principle, can be computed by such network.

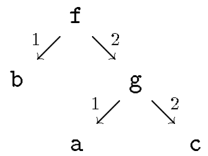
We will discuss the computational capabilities of Elman-style Recurrent Networks, Recurrent Cascade Correlation networks, and Neural Trees with respect to the classification of structures. We will relate them to Frontier-to-Root Tree Automata (FRA) [Tha73, GT78] and show that Elman-style Recurrent Networks can simulate any FRA, while neither Cascade-Correlation networks, nor Neural Trees can. Results on function approximation and complexity of FRA implementation are given as well.

2 Definitions and background topics

2.1 Structured domains

Here we consider structured domains which are sets of annotated directed ordered acyclic graphs (DOAGs). For a DOAG we mean a DAG $\boldsymbol{Y}$ with vertex set vert($\boldsymbol{Y}$) and edge set egd($\boldsymbol{Y}$), where for each vertex $v \in$ vert($\boldsymbol{Y}$) a total order on the edges leaving from v is defined. The void DOAG will be denoted by the special symbol ξ.

For example, in the case of graphs representing logical terms, the order on outgoing edges is immediately induced by the order of the arguments to a function; e.g., the logical term `f(b,g(a,c))` can be represented as:



In problems of structure classification, we shall require the DOAG either to be empty or to possess a supersource, i.e. a vertex $s \in$ vert($\boldsymbol{Y}$) such that every vertex in vert($\boldsymbol{Y}$) can be reached by a directed path starting from s. Note that if a DOAG does not possess a supersource, it is still possible to define a convention for adding an extra vertex s (with a minimal number of outgoing edges), such that s is a supersource for the expanded DOAG [SS97]. The function $source(\boldsymbol{Y})$ returns the (unique) supersource of $\boldsymbol{Y}$.

Given a DOAG $\boldsymbol{Y}$ and $v \in$ vert($\boldsymbol{Y}$), we denote by pa$[v]$ the set of parents of v, by ch$[v]$ the set of children of v, by de$[v]$ the set of descendants of v, and by an$[v]$ the set of ancestors of v. The *indegree* of v is the cardinality of the set pa$[v]$, the *outdegree* of v is the cardinality of the set ch$[v]$. In the following, we shall denote by $\#^{(i,o)}$ the class of DOAGs with maximum indegree i and maximum outdegree o. In our logical terms example, the maximum outdegree corresponds to the maximum arity of the functions being considered, e.g., the maximum outdegree of `f(b,g(a,c))` is 2. A generic class of DOAGs with bounded (but unspecified) indegree and outdegree, will simply be denoted by $\#$.

The data structures we are considering are labeled DOAGs, where labels are tuples of variables and are attached to vertices.

A data structure $\boldsymbol{Y}$ is a DOAG whose vertices are labeled by subsets of variables in the local universe of discourse. Subscript notation will be used when referencing the labels attached to vertices in a data structure. Hence $\boldsymbol{Y}_v$ denotes the set of variables labeling vertex $v \in$ vert($\boldsymbol{Y}$).

Given a data structure $\boldsymbol{Y}$, the DOAG obtained by ignoring all node labels will be referred to as the *skeleton* of $\boldsymbol{Y}$, denoted skel($\boldsymbol{Y}$). Clearly, any two data structures can be distinguished because they have different skeletons, or, if they have the same skeleton, because they have different node labels. The class of all data structures defined over the local universe domain $\mathcal{Y}$ and skeleton in $\#^{(i,o)}$ will be denoted as $\mathcal{Y}^{\#^{(i,o)}}$.

2.2 Generalized shift operators

A discrete time operator applied to a temporal variable Y_t is essentially a symbolic transformation that maps Y_t into an expression involving the variable Y

measured at different time steps. In particular, the shift operator q^{-1} applied to Y_t returns the variable Y at time $t-1$: $q^{-1}Y_t = Y_{t-1}$. In our graphical framework, a finite length temporal sequence corresponds to a linear chain. The supersource (the head of the chain) corresponds to the last time step in the sequence, and thus time indices decrease following the direction of the arrows. Hence, q^{-1} can be thought of as an operator that gives access to the (unique) child of a given node, when a sequence is graphically represented as a linear chain. Shift operators can be composed so that $q^{-t} = q^{-1}q^{1-t}$, where $1 = q^0$ is the *neutral* operator defined as $q^0 Y_t = Y_t$.

When general DOAGs are considered, an ordered set of generalized shift operators can be defined and associated with the ordered set of children. For $k = 1, \ldots, o$, we denote by q_k^{-1} the shift operator associated with the k-th child of a given node. For the class $\#^{(1,1)}$ (i.e. the class of linear chains) the subscript will be omitted. In some special classes of graphs, we sometimes use more descriptive subscript notations. For example, when considering the class of binary trees $\#^{(1,2)}$, q_L^{-1} and q_R^{-1} will be used to denote the operators associated with the left child and the right child, respectively.

When considering a labeled graph $\boldsymbol{Y}$ with local universe domain $\mathcal{Y}$, the expression $q_k^{-1}\boldsymbol{Y}_v$ denotes the label found in the k-th child of node v. For example, in the graphical representation $\boldsymbol{Y}$ of `f(b,g(a,c))` (see above), let $s = source(\boldsymbol{Y})$. Then, $q_1^{-1}\boldsymbol{Y}_s = \mathtt{b}$ and $q_2^{-1}\boldsymbol{Y}_s = \mathtt{g}$. Shift operators can be composed and expressions involving composition of shift operators specify directed paths in the DOAG. Note that the composition of shift operators is not commutative. For example, $q_2^{-1}q_1^{-1}\boldsymbol{Y}_s = \mathtt{a}$, however, the composition $q_1^{-1}q_2^{-1}\boldsymbol{Y}_s$ is not defined since the first child of s has no children. When considering the class $\#^{(1,m)}$ of m-ary trees, the *inverse* shift operators q_k can also be defined as $q_k q_k^{-1}\boldsymbol{Y}_v = \boldsymbol{Y}_v$, $k = 1, \ldots, m$.

2.3 Graph Transductions

A functional graph transduction $\mathcal{T} : \mathcal{I}^\# \rightarrow \mathcal{O}^\#$ is defined as a mapping from an input structured domain $\mathcal{I}^\#$ to an output structured domain $\mathcal{O}^\#$. We consider the special class of transductions where $\mathcal{O}^\#$ is the trivial set of structures compounded by a single labeled node, i.e., $\mathcal{O}^{\#^{(0,0)}}$. A particular instance of transductions which belong to this class is obtained when the local universe of discourse $\mathcal{O}$ is equal to $\{0, 1\}$. In this case, the transduction can be interpreted as a classification function of the structures in $\mathcal{I}^\#$.

In addition, we further restrict to graph transductions which can be represented in the following form

$$\mathcal{T} = g \circ \hat{\tau}, \tag{1}$$

where $\hat{\tau} : \mathcal{I}^\# \rightarrow \mathcal{X}$ is the *encoding* (or *state transition*) function and $g : \mathcal{X} \rightarrow \mathcal{O}^{\#^{(0,0)}}$ is the *output* function. Specifically, $\hat{\tau}$ is defined recursively as

$$\hat{\tau}(\boldsymbol{Y}) = \begin{cases} nil & \text{if } \boldsymbol{Y} = \xi \\ \tau(s, \boldsymbol{Y}_s, \hat{\tau}(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}(\boldsymbol{Y}^{(o)})) & \text{otherwise} \end{cases} \tag{2}$$

where $\tau : vert(\mathcal{I}^{\#}) \times \mathcal{I} \times \mathcal{X}^{o} \rightarrow \mathcal{X}$ is the *c-model* function, $s = source(\boldsymbol{Y})$, $nil \in \mathcal{X}$ is used to represent the void graph, and $\boldsymbol{Y}^{(1)}, \ldots, \boldsymbol{Y}^{(o)}$ are the subgraphs pointed by s, i.e., $source(\boldsymbol{Y}^{(i)}) = q_i^{-1} s$. The function τ is called c-model function since it defines a computational model for the encoding function.

Note that, because of eq. (2), $\mathcal{T}$ is *causal* since τ only depends on the current node and nodes descending by it. Moreover, when τ does not depend on any specific vertex, i.e., $\tau(\boldsymbol{Y}_s, \hat{\tau}(\boldsymbol{Y}^{(1)}), \ldots, \hat{\tau}(\boldsymbol{Y}^{(o)}))$, then $\mathcal{T}$ is also *stationary.* We will mainly focus on stationary transductions.

Example 1 Encoding of logical terms. Given a stationary encoding function $\hat{\tau}$, the encoding of the logical term `f(b,g(a,c))` is defined by the following set of equations

$$\hat{\tau}\left(\begin{array}{ccccc} & & \mathtt{f} & & \\ & {}^1\swarrow & & \searrow^2 & \\ \mathtt{b} & & & & \mathtt{g} \\ & & & {}^1\swarrow & & \searrow^2 \\ & & \mathtt{a} & & & & \mathtt{c} \end{array}\right) = \tau\left(\mathtt{f}, \hat{\tau}(\mathrm{b}), \hat{\tau}\left(\begin{array}{ccc} & \mathtt{g} & \\ {}^1\swarrow & & \searrow^2 \\ \mathtt{a} & & \mathtt{c} \end{array}\right)\right),$$

$$\hat{\tau}\left(\begin{array}{ccc} & \mathtt{g} & \\ {}^1\swarrow & & \searrow^2 \\ \mathtt{a} & & \mathtt{c} \end{array}\right) = \tau(\mathtt{g}, \hat{\tau}(\mathrm{a}), \hat{\tau}(\mathrm{c})),$$

$$\hat{\tau}(\mathrm{b}) = \tau(\mathtt{b}, nil, nil),$$
$$\hat{\tau}(\mathrm{a}) = \tau(\mathtt{a}, nil, nil),$$
$$\hat{\tau}(\mathrm{c}) = \tau(\mathtt{c}, nil, nil),$$

where a, b, and c denote the graphs with a single node labeled `a`, `b`, and `c`, respectively. By proper substitutions, the encoding can finally be written as

$$\hat{\tau}\left(\begin{array}{ccccc} & & \mathtt{f} & & \\ & {}^1\swarrow & & \searrow^2 & \\ \mathtt{b} & & & & \mathtt{g} \\ & & & {}^1\swarrow & & \searrow^2 \\ & & \mathtt{a} & & & & \mathtt{c} \end{array}\right) = \tau(\mathtt{f}, \tau(\mathtt{b}, nil, nil), \tau(\mathtt{g}, \tau(\mathtt{a}, nil, nil), \tau(\mathtt{c}, nil, nil))).$$

It is interesting to notice that causal stationary transductions cannot compute any function of the input DOAG. For example, consider the following two DOAGs

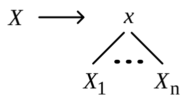
$$\begin{array}{ccc} & \mathtt{f} & \\ {}^1\swarrow & & \searrow^2 \\ \mathtt{h} & & \mathtt{q} \\ {}_1\downarrow & & {}_1\downarrow \\ \mathtt{a} & & \mathtt{a} \end{array} \qquad \begin{array}{ccc} & \mathtt{f} & \\ {}^1\swarrow & & \searrow^2 \\ \mathtt{h} & & \mathtt{q} \\ {}_1\searrow & & \swarrow_1 \\ & \mathtt{a} & \end{array}$$

Any given transduction described by equations (1-2) will necessarily map these two graphs into the same output, regardless of the form of functions $g(\cdot)$ and $\tau(\cdot)$. This is because the graph's processing is bottom-up, so it is not possible to recognize that in the right-side graph $pa[a]$ is a set of two nodes (labeled `h` and `q`, respectively).

2.4 Tree Grammars and Tree Automata

A *tree grammar* is defined as a four-tuple $G_t = (V, r, P, S)$ where $V = N \cup \Sigma$ is the grammar alphabet (nonterminals and terminals); (V, r) a ranked alphabet; productions in P are of the form $T_i \rightarrow T_j$, where T_i and T_j are trees; and S in T_V is a finite set of "starting trees," where T_V denotes the set of trees with nodes labeled by elements in V.

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



(3)

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

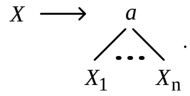
1. The frontier of the state tree is labeled q_0 (the status associated to nodes with $m = 0$, i.e, the leaves).
2. For any node in the input labeled l with rank m, the corresponding node of the state tree is labeled $f_l(q_1, \ldots, q_m)$, where $q_1, \ldots, q_m$ are the states labeling the offsprings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

An input tree is accepted by $\mathcal{A}_t$ if the automaton can enter a final state upon encountering the root.

According to the above definition, the *language recognized* by $\mathcal{A}_t$ is the set

$T(\mathcal{A}_t) = \{T | T \text{ in } T_\Sigma,\ \mathcal{A}_t$ can halt in a state in F when the root of T is reached $\}$.

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



(4)

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

2.5 Target Function and Training Set

Since we will deal with learning, we need to define the target function we want to learn. In approximation tasks, a *target function* $d()$ over $\mathcal{D}^\#$ is defined as any function $d : \mathcal{D}^\# \rightarrow I\!R^k$, where k is the output dimension, while in (binary) classification tasks we have $d : \mathcal{D}^\# \rightarrow \{0,1\}^k$ (or $d : \mathcal{D}^\# \rightarrow \{-1,1\}^k$.) A training set T on a domain $\mathcal{D}^\#$ is defined as a set of couples $(\boldsymbol{X}, d(\boldsymbol{X}))$, where $\boldsymbol{X} \in \mathcal{U}^\# \subseteq \mathcal{D}^\#$ and $d()$ is a target function defined on $\mathcal{D}^\#$.

2.6 Graphical models for structural processors

We now give a graphical notation for the recursive representation of causal transductions. This notation is a tool for clearifying the functional form of the transduction.

Let $\mathcal{T}(\cdot)$ be a causal stationary transduction from $\mathcal{I}^\#$ to $\mathcal{O}^\#$. The canonical *recursive network* of $\mathcal{T}(\cdot)$ is a directed (possibly cyclic) labeled graph $N(\mathcal{T}, \#)$ defined as follows:

- $N(\mathcal{T}, \#)$ has $|Var(\mathcal{I})| + |Var(\mathcal{X})| + |Var(\mathcal{O})|$ nodes and each node is labeled by (i.e., associated with) one distinct variable in the sets $Var(\mathcal{I})$, $Var(\mathcal{X})$, and $Var(\mathcal{O})$.
- Edges in $N(\mathcal{T}, \#)$ describe causal dependencies amongst the input, state, and output variables in the recursive representation of $\mathcal{T}(\cdot)$. Edges are labeled by shift operators. Let A,B be generic variables. An edge $(A, B, 1)$ labeled by the neutral shift operator indicates that B is causally dependent on A. Similarly, an edge (A, B, q_k^{-1}) indicates that B is causally dependent on $q_k^{-1}A$. The indegree of nodes in $Var(\mathcal{I})$ is 0. Edges from $Var(\mathcal{O})$ to $Var(\mathcal{X})$ are forbidden. Edges incident on output nodes must always be labeled by the neutral shift operator 1. Cycles in which all edges are labeled by the neutral shift operator are not allowed.

Non canonical recursive networks can be defined by labeling the edges of $N(\mathcal{T}, \#)$ with a composition of shift operators $q_{k_1}^{-1} q_{k_2}^{-1}, \cdots, q_{k_n}^{-1}$ that will be abbreviated by $L(\boldsymbol{q}^{-1})$.

The whole set of variables involved during the processing of a given data structure $\boldsymbol{I}$ can be graphically represented using the recursive network associated with the transduction. Essentially, the recursive network is used as a template which is unfolded (i.e., expanded) according to the skeleton of the input data structure. The resulting labeled graph (function of both the recursive network $N(\mathcal{T}, \#)$ and the input-output data structures $\boldsymbol{I} \in \mathcal{I}^\#$,$\boldsymbol{O} \in \mathcal{O}^\#$) is called *encoding network* of the transduction, denoted $\mathcal{E}(N, \boldsymbol{I}, \boldsymbol{O})$. Nodes and edges of $\mathcal{E}(N, \boldsymbol{I}, \boldsymbol{O})$ are constructed as follows:

- The vertex set of $\mathcal{E}$ is the Cartesian product of the vertex sets of N and $\mathrm{skel}(\boldsymbol{I})$. For each $v \in \mathrm{vert}(\mathrm{skel}(\boldsymbol{I}))$, $I_{i,v}$ denotes the i-th input variable at node v, $X_{i,v}$ denotes the i-th state variable at node v, and $O_{i,v}$ denotes the i-th output variable at node v.

– The edge set of $\mathcal{E}$ is obtained as follows. Let $A_{i,v}$ $B_{j,w}$ denote two vertices of $\mathcal{E}$. The directed edge $(A_{i,v}, B_{j,w})$ is present in $\mathcal{E}$ if and only if the edge (A_i, B_j) is present in N and is labeled by an expression $L(\boldsymbol{q}^{-1})$ such that $L(\boldsymbol{q}^{-1})\boldsymbol{I}_w = \boldsymbol{I}_v$.

Algorithm 1 builds the skeleton of the encoding network associated with the recursive network N and the skeleton $S = \mathrm{skel}(\boldsymbol{I}) = \mathrm{skel}(\boldsymbol{O})$ of the input-output DOAGs.

Build-Encoding-Network (N, S)
 $\mathrm{vert}(\mathcal{E}) \leftarrow \emptyset$
 $\mathrm{egd}(\mathcal{E}) \leftarrow \emptyset$
 Foreach $v \in \mathrm{vert}(S)$ **Do**
  Foreach $A_i \in \mathrm{vert}(N)$ **Do**
   $\mathrm{vert}(\mathcal{E}) \leftarrow \mathrm{vert}(\mathcal{E}) \cup \{A_{i,v}\}$
 Foreach $(A_i, A_j, L(\boldsymbol{q}^{-1})) \in \mathrm{egd}(N)$ **Do**
  Foreach $v \in \mathrm{vert}(S)$ **Do**
   If $L(\boldsymbol{q}^{-1})v \neq nil$ **Then**
    $\mathrm{egd}(\mathcal{E}) \leftarrow \mathrm{egd}(\mathcal{E}) \cup \{(L(\boldsymbol{q}^{-1})A_{i,v}, A_{j,v})\}$
   Else
    $\mathrm{egd}(\mathcal{E}) \leftarrow \mathrm{egd}(\mathcal{E}) \cup \{(\boldsymbol{X}_i^F, A_{j,v})\}$
 Return $\mathcal{E}$

An example of encoding network construction for our logical term example is shown in Figure 1. Note that arrows in the encoding networks are reversed with respect to the direction of arrows in the input data structure. This is because computation in recursive causal transductions proceeds from the frontier to the supersource. Shift operators can be applied also to nodes of the encoding network. In particular, $q_k^{-1}\boldsymbol{X}_v$ returns the state label attached to the k-th child of v in the skeleton of $\boldsymbol{I}$ (which is the k-th parent of $\boldsymbol{X}_v$ in the encoding network).

3 Neural Graph Transductions

Graph transductions can be implemented by standard and recurrent neural networks only by resorting to a complex and very unnatural encoding scheme which maps structures onto fixed-size unstructured patterns or sequences. As early realized by researchers in speech recognition, the encoding of a complex object by an object with an inferior degree of complexity can preserve all the information needed for the computational task at hand only at the expenses of a remarkable increase in the size of the representation. For example, since graphs present two main sources of variability, namely the size of the graph and the relative arrangement of single nodes, an encoding of a graph by a static array will have two major drawbacks: *(i)* the number of components of the array should be equal to the maximum size allowed for the graphs; *(ii)* the information on the edges should be preserved, for example by using an incidence matrix, which means a quadratic increase in the dimension of the representation. The consequence

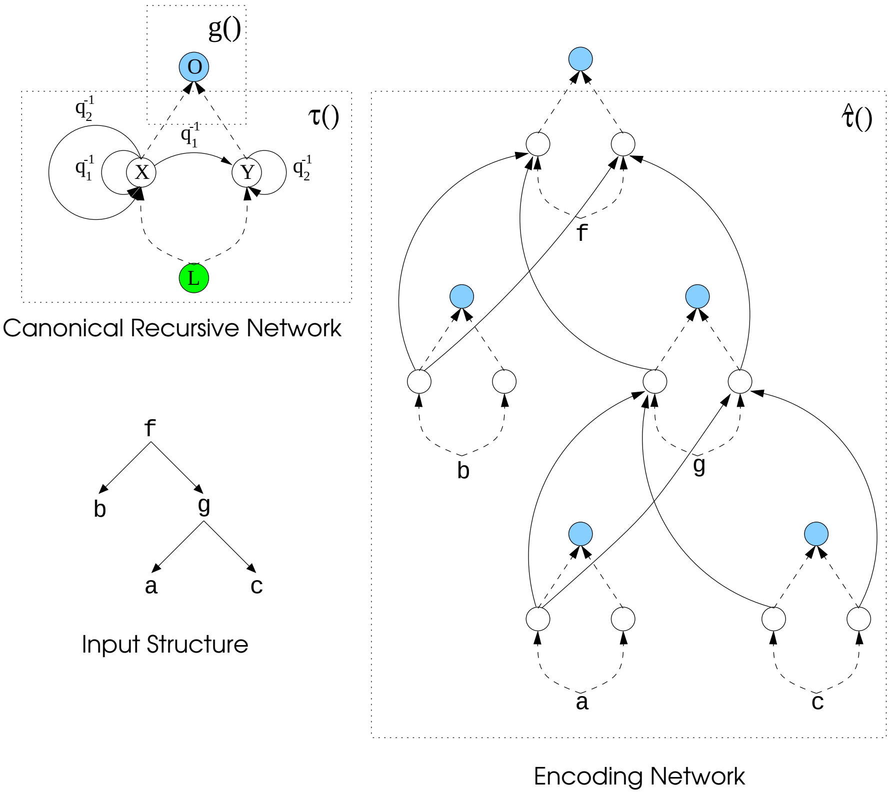


Fig. 1. Example of Canonical Recursive Network and Encoding Network. The relationships with $g(\cdot)$, $\tau(\cdot)$, and $\hat{\tau}(\cdot)$ are shown by dotted boxes.

of such increase in size of the representations is that graphs of very different size are represented by objects of the same size, rendering the representational scheme wasteful. Moreover, it is well known that both training algorithms for neural networks and generalization do not scale well with the input dimension. Especially generalization can suffer from the introduction of several free parameters associated with the input connections. In the context of classification of structures, a partial solution to the above problem is to use some a priori defined features of the graphs as input to a standard feedforward network, i.e., in chemistry topological indexes are used. Unfortunately, this approach works only if knowledge about the domain is available for driving the extraction of the set of features used for classification. In fact, it is very unlikely that a blind extraction of graph's features will lead to good results.

No better solution, in general, can be expected by encoding graphs through sequences, e.g., binary trees can uniquely be represented by combining inorder and preorder trasversal, since both the length of the sequences will explode exponentially with the diameter of the graph and nice regularities inherently associated with the structure of the graph can be lost in the sequential coding. The problem related with the length of the sequences is also amplificated in the context of neural network by the well known problem of learning long-term dependencies [BFS94].

In conclusion, standard and recurrent neural networks seem to be inadequate for implementing neural graph transductions. This, however, does not mean that neural networks are inadequate for this task. In fact, recursive graph transductions can be implemented by neural networks by following the schema below.

Let $\mathcal{I}^{\#}$ be the input structured domain with $\mathcal{I} = I\!R^n$, i.e., the labels are real valued vectors, and $\mathcal{O} = I\!R^k$. The encoding function $\hat{\tau}$ is completely defined by choosing a representation for the void graph and by defining the c-model function τ. By choosing $\mathcal{X} = I\!R^m$, any real valued vector would be fine for representing the void graph, however, for computational reasons which will be clear in the following, we choose the null vector for representing the void graph, i.e., $nil = \mathbf{o} \in I\!R^m$. Consequently the c-model function τ will be defined as

$$\tau : I\!R^n \times \underbrace{I\!R^m \times \cdots \times I\!R^m}_{\text{o times}} \rightarrow I\!R^m \tag{5}$$

where the first domain of the cartesian product denotes the label space, while the remaining domains represent the encoded subgraphs spaces up to the maximum outdegree of the input domain $\mathcal{I}^{\#}$. Finally, we can define the output function g as

$$g : I\!R^m \rightarrow I\!R^k. \tag{6}$$

Note that eqs. (5) and (6) only describe the general form for τ and g. Different realizations can be given which satisfy the above equations. For example, both τ and g can be implemented by feedforward neural networks. Before to reach such level of complexity, however, it is worth to explore simplier realizations. Specifically, in the following section we study recursive neurons.

4 Recursive Neurons

Let first consider the case in which $m = 1$. The simpliest non-linear neural realization for $\tau(\cdot)$ is given by

$$\tau(\boldsymbol{l}, x_1, \ldots, x_o) = f(\sum_{i=1}^{n} w_i l_i + \sum_{j=1}^{o} \hat{w}_j x_j + \theta), \tag{7}$$

where f is a sigmoidal function, w_i are the weights associated to the label space, $\hat{w}_j$ are the weights associated to the subgraphs spaces, θ is the bias, $\boldsymbol{l}$ is the current input label, and $x_1, \ldots, x_o$ are the encoded representations of subgraphs. The grafical representation of the recursive neuron is given in Figure 2.

Example 2 Logical term `f(b,g(a,c))`. Let define a coding function

$$\phi : \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{f}, \mathtt{g}\} \rightarrow \{0, 1\}^5$$

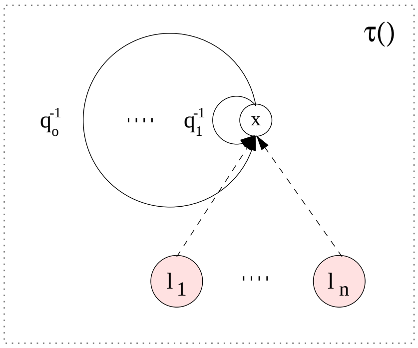


Fig. 2. Graphical representation for a recursive neuron. Dashed lines represent edges with neutral shift operator.

which encodes the simbolic labels into binary vectors, i.e., $\phi(\mathtt{a}) = [1, 0, 0, 0, 0]$, $\phi(\mathtt{b}) = [0, 1, 0, 0, 0]$, $\phi(\mathtt{c}) = [0, 0, 1, 0, 0]$, $\phi(\mathtt{f}) = [0, 0, 0, 1, 0]$, $\phi(\mathtt{g}) = [0, 0, 0, 0, 1]$, and $nil = 0$. Thus, $\tau(\cdot)$ will be defined by five parameters for the label, i.e., $w_1, \ldots, w_5$, two parameters for the subgraphs (maximum outdegree), i.e., $\hat{w}_1$ and $\hat{w}_2$, and the bias θ.

Then,

$$\hat{\tau}\left(\begin{array}{ccccccc} & & \mathtt{f} & & & & \\ & {}^{1}\swarrow & & \searrow^{2} & & & \\ \mathtt{b} & & & & \mathtt{g} & & \\ & & & {}^{1}\swarrow & & \searrow^{2} & \\ & & \mathtt{a} & & & & \mathtt{c} \end{array}\right) = f(\sum_{i=1}^{5} w_i \phi_i(\mathtt{f}) + \hat{w}_1 f(\sum_{i=1}^{5} w_i \phi_i(\mathtt{b}) + \theta) +$$

$$\hat{w}_2 f(\sum_{i=1}^{5} w_i \phi_i(\mathtt{g}) + \hat{w}_1 f(\sum_{i=1}^{5} w_i \phi_i(\mathtt{a}) + \theta) +$$

$$\hat{w}_2 f(\sum_{i=1}^{5} w_i \phi_i(\mathtt{b}) + \theta) + \theta) + \theta).$$

A graphical representation of the above equation is given in Figure 3.

When $m > 1$, $\tau(\cdot) \in I\!R^m$ can be written as

$$\tau(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) = \boldsymbol{F}(\boldsymbol{W}\boldsymbol{l} + \sum_{j=1}^{o} \widehat{\boldsymbol{W}}_j \boldsymbol{x}^{(j)} + \boldsymbol{\theta}), \tag{8}$$

where $\boldsymbol{F}_i(\boldsymbol{v}) = f(v_i)$, $\boldsymbol{l} \in I\!R^n$, $\boldsymbol{\theta} \in I\!R^m$, $\boldsymbol{W} \in I\!R^{m \times n}$, $\boldsymbol{x}^{(j)} \in I\!R^m$, $\widehat{\boldsymbol{W}}_j \in I\!R^{m \times m}$. The graphical representation of $\tau(\cdot)$ for $m = 2$ is given in Figure 4.

Concerning the output function $g(\cdot)$, it can be defined as a standard neuron taking in input the encoded representation of the graph $\boldsymbol{x} = \hat{\tau}(\boldsymbol{X})$, i.e.,

$$g(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{M}\boldsymbol{x} + \boldsymbol{\beta}), \tag{9}$$

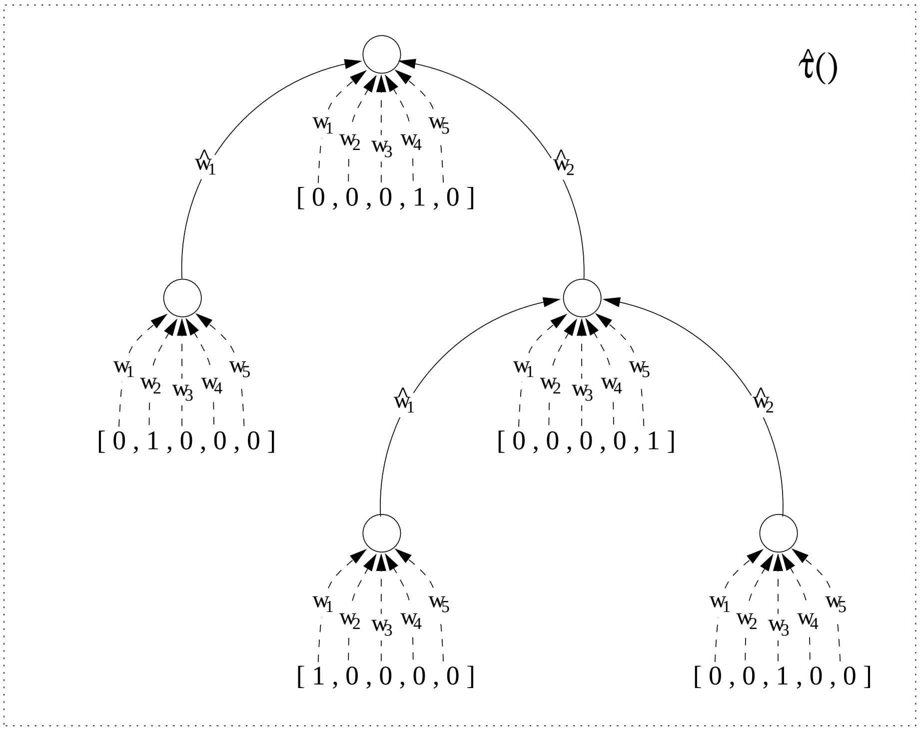


Fig. 3. Graphical representation for the encoding function of Example 2.

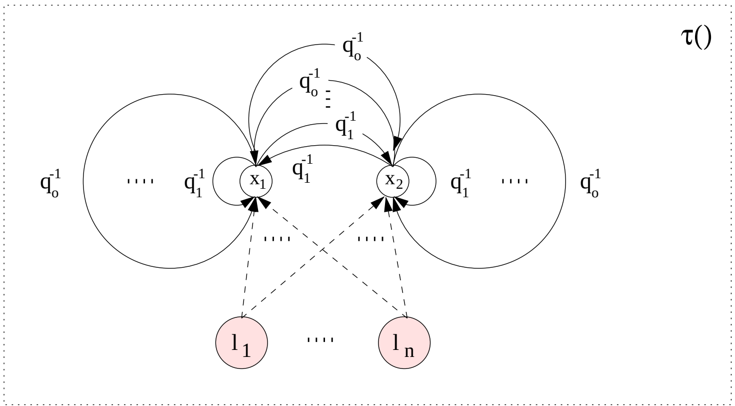


Fig. 4. Graphical representation of $\tau(\cdot)$ for $m = 2$.

where $\boldsymbol{M} \in I\!R^{k \times m}$ and $\boldsymbol{\beta} \in I\!R^{k}$ are the weight matrix and bias terms defining $g(\cdot)$, respectively.

As a special case, $g(\cdot)$ can be defined as a function which selects a subset of the state vector in $\mathcal{X}$. For example,

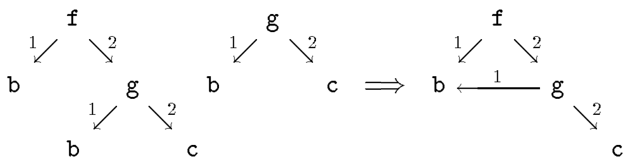
$$g(\boldsymbol{x}) = \boldsymbol{M}\boldsymbol{x}, \tag{10}$$

where $\boldsymbol{M} \in I\!R^{k \times m}$ is a 0-1 matrix with a single 1 for each row and at most a single 1 for each column. This corresponds to have the selected components of the state vector to contribute to both the output and the coding of the state.

4.1 Optimized Training Set

In Section 2.3 we have observed that causality imposes some limitations on the kind of transductions which can be computed. Specifically, we showed that the bottom-up flow of computation actually disregards information about parents. This limitation, however, allows to improve the efficiency of computation. In fact, given a training set T of DOAGs, if there are graphs $\boldsymbol{X}_1, \boldsymbol{X}_2 \in T$ which share a common subgraph $\hat{\boldsymbol{X}}$, then we need to explicitly represent $\hat{\boldsymbol{X}}$ only once. The optimization of the training set can be performed by merging all the DOAGs in the training set into a single *minimal* DOAG, i.e., a DOAG with a minimal number of vertices which contains all the DOAGs in the training set. Note that the minimal DOAG may also be non-connected, i.e., it may be constituted by multiple disjoint subgraphs.

Example 3 Optimization of two DOAGs.



Training Set　　　　Minimal DOAG

Moreover, after the generation of the minimal DOAG, which can be performed efficiently [Gol97], a topological sort on the vertices of the minimal DOAG can be performed to determine the updating order on the vertices for the encoding network. The advantage of having a sorted set of vertices is that all the computation over partial structures (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 vertices set. Moreover, the use of the minimal DOAG leads to a considerable reduction in space complexity. In some cases, such reduction can even be exponential.

5 Supervised Models

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

5.1 Back-propagation through Structure

The task addressed by back-propagation through time networks [RM86] 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 back-propagation. A trick, however, can be used to turn an arbitrary recurrent network into an equivalent feed-forward 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 t is held by the tth copy of the same unit in the feed-forward network. By preserving the same weight values through the layers of the feed-forward network, it is not difficult to see that the two networks will behave identically for T time steps. The feed-forward network can be trained by back-propagation, 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.

Back-propagation through Time can easily be extended to structures. The basic idea is to use recursive neurons to encode the structures, i.e., $\hat{\tau}(\cdot)$; the representations obtained are then classified or used to approximate an unknown function by a standard feed-forward network, i.e., $g(\cdot)$.

Following our framework, given an input graph $\boldsymbol{X}$, the network output $\mathcal{T}(\boldsymbol{X})$ can be expressed as the composition of the encoding function $\hat{\tau}(\cdot)$ and the classification (or approximation) function $g(\cdot)$:

$$\mathcal{T}(\boldsymbol{X}) = g(\hat{\tau}(\boldsymbol{X})). \tag{11}$$

Learning for the set of weights $\boldsymbol{W}_g$ can be implemented by plain back-propagation on the feed-forward network realizing $g(\cdot)$

$$\Delta \boldsymbol{W}_g = -\eta \frac{\partial Error(g(\boldsymbol{y}))}{\partial \boldsymbol{W}_g}, \tag{12}$$

where $\boldsymbol{y} = \hat{\tau}(\boldsymbol{X})$, i.e., the input to the feed-forward network, while learning for the set of weights $\boldsymbol{W}_{\hat{\tau}}$, realizing $\hat{\tau}(\cdot)$, can be implemented by

$$\Delta \boldsymbol{W}_{\hat{\tau}} = -\eta \frac{\partial Error(g(\boldsymbol{y}))}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}_{\hat{\tau}}}, \tag{13}$$

where the first term represents the error coming from the feed-forward network and the second one represents the error due to the encoding function $\hat{\tau}(\cdot)$. Note that $\boldsymbol{W}_{\hat{\tau}}$ is actually referring to the weights associated with the c-model function $\tau(\cdot)$. Thus, in order to clarify this point we will denote this weight matrix as $\boldsymbol{W}_\tau$

Goller and Küchler [GK96, KG96, SKG97] devised a learning algorithm, namely *Backpropagation Through Structure*, on the basis of the above equations. The basic idea it to observe that $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}_\tau}$ can be computed by backpropagating the error from the feed-forward network implementing $g(\cdot)$ through the network implementing the encoding function $\hat{\tau}(\boldsymbol{X})$. As in back-propagation 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. Recalling that the encoding network of a graph $\boldsymbol{X}$ is defined as the composition of the network implementing $\hat{\tau}(\boldsymbol{X})$ with the network implementing $g(\cdot)$, the learning algorithm for a (optimized) training set of L structures can be summarized in the following way:

Backpropagation Through Structure

Repeat
for l=1 **to** L
    Compute $\mathcal{T}(X)$;
    Compute the gradient using the Backpropagation Algorithm over the encoding network of X;
    Collect (add) the gradient for the different instances of weights in W_τ;
Update W_τ and W_g;
Until Convergence

5.2 Extension of Real-Time Recurrent Learning

The extension of Real-Time Recurrent Learning [WZ89] to recursive neurons does not present particular problems. Here we consider a neural model where $g(\cdot)$ is implemented as in equation (10), while the encoding is performed by m recursive neurons, i.e., (by including the bias into the label weight matrix)

$$\hat{\tau}(\boldsymbol{X}) = \tau(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) = \boldsymbol{F}(\boldsymbol{W}\boldsymbol{l} + \sum_{j=1}^{o} \widehat{\boldsymbol{W}}_j \boldsymbol{x}^{(j)}), \tag{14}$$

where $\boldsymbol{x}^{(j)} = \hat{\tau}(\boldsymbol{X}^{(j)})$ with $source(\boldsymbol{X}^{(j)}) = q_j^{-1} source(\boldsymbol{X})$, and

$$\boldsymbol{W}_\tau = [\boldsymbol{W}, \widehat{\boldsymbol{W}}_1, \ldots, \widehat{\boldsymbol{W}}_o]. \tag{15}$$

Here we show how to compute the derivatives of $\hat{\tau}(\boldsymbol{X})$ leaving to the reader the derivation of the final learning rule according to the chosen error function.

The derivatives of $\hat{\tau}(\boldsymbol{X})$ with respect to $\boldsymbol{W}$ and $\widehat{\boldsymbol{W}}_i$ ($i \in [1, \ldots, o]$) can be computed as follows:

$$\frac{\partial \hat{\tau}_p(\boldsymbol{X})}{\partial W_{tk}} = \tau'_p(\boldsymbol{l}_k \delta_{pt} + \sum_{j=1}^{o} (\widehat{\boldsymbol{W}}_j)_p \frac{\partial \boldsymbol{x}^{(j)}}{\partial W_{tk}}), \tag{16}$$

where τ'_p is the first derivative of the pth component of $\tau(\cdot)$ computed on the supersource of $\boldsymbol{X}$, δ_{mt} is the Kronecker delta, $p = 1, \ldots, m$, $t = 1, \ldots, m$, $k = 0, \ldots, n$ and $(\widehat{\boldsymbol{W}}_j)_p$ is the pth row of $\widehat{\boldsymbol{W}}_j$;

$$\frac{\partial \hat{\tau}_p(\boldsymbol{X})}{\partial (\widehat{\boldsymbol{W}}_i)_{tq}} = \tau'_p(\boldsymbol{x}_p^{(j)}\delta_{pt} + \sum_{j=1}^{o} (\widehat{\boldsymbol{W}}_j)_t \frac{\partial \boldsymbol{x}^{(j)}}{\partial (\widehat{\boldsymbol{W}}_i)_{tq}}), \tag{17}$$

where $t = 1, \ldots, m$, and $q = 1, \ldots, m$.

These equations are recursive on the input graph $\boldsymbol{X}$, and can be computed by noting that if $\boldsymbol{V}$ is a graph composed by a single vertex, then

$$\frac{\partial \hat{\tau}_p(\boldsymbol{V})}{\partial W_{tk}} = \tau'_p(\boldsymbol{l}_{source(\boldsymbol{V})})_k \delta_{pt}, \text{ and } \frac{\partial \hat{\tau}_t(\boldsymbol{V})}{\partial (\widehat{\boldsymbol{W}}_i)_{tq}} = 0. \tag{18}$$

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

5.3 Cascade-Correlation for Structures

The Cascade-Correlation algorithm [FL90] 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 maximised[1]. 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 [Fah91]. In the following, we show that the Cascade-Correlation can be further extended to structures by using recursive neurons.

The graphical representation of the c-model function $\tau(\cdot)$ for the Cascade-Correlation for structure is given in Figure 5. Specifically, the general shape of the c-moldel function can be expressed by the following set of equations:

$$\begin{aligned}
\tau_1 &= h_1(\boldsymbol{l}, \hat{\tau}_1(s_1), \ldots, \hat{\tau}_1(s_t)) \\
\tau_2 &= h_2(\boldsymbol{l}, \hat{\tau}_1(s_1), \ldots, \hat{\tau}_1(s_t), \hat{\tau}_2(s_1), \ldots, \hat{\tau}_2(s_t), \hat{\tau}_1(X)) \\
&\vdots \\
\tau_m &= h_m(\boldsymbol{l}, \hat{\tau}_1(s_1), \ldots, \hat{\tau}_1(s_t), \hat{\tau}_2(s_1), \ldots, \hat{\tau}_2(s_t), \\
&\qquad \ldots, \hat{\tau}_m(s_1), \ldots, \hat{\tau}_m(s_t), \hat{\tau}_1(X), \ldots, \hat{\tau}_{m-1}(X))
\end{aligned} \tag{19}$$

[1] 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.

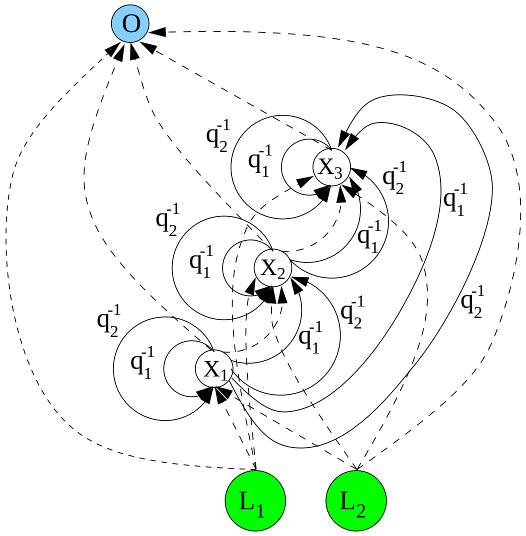


Fig. 5. Graphical representation of the c-model function $\tau(\cdot)$ of a Cascade-Correlation network for structure with three hidden units and one output.

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

$$\tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) = f(\sum_{i=0}^{n} w_i^{(k)} l_i + \sum_{v=1}^{k} \sum_{j=1}^{o} \hat{w}_{(v,j)}^{(k)} \boldsymbol{x}_v^{(j)} + \sum_{q=1}^{k-1} \bar{w}_q^{(k)} \tau_q(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})), \tag{20}$$

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 subgraph code $\boldsymbol{x}^{(j)}$, 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 network is then computed according to equation (9).

Learning is performed as in standard Cascade-Correlation, with the difference that, according to equation (20), the derivatives of $\tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})$ with respect to the weights are now:

$$\frac{\partial \tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})}{\partial w_i^{(k)}} = (l_i + \sum_{j=1}^{o} \hat{w}_{(k,j)}^{(k)} \frac{\partial \boldsymbol{x}_k^{(j)}}{\partial w_i^{(k)}}) f' \tag{21}$$

$$\frac{\partial \tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})}{\partial \bar{w}_q^{(k)}} = (\tau_q(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)}) + \sum_{j=1}^{o} \hat{w}_{(k,j)}^{(k)} \frac{\partial \boldsymbol{x}_k^{(j)}}{\partial \bar{w}_q^{(k)}}) f' \tag{22}$$

$$\frac{\partial \tau_k(\boldsymbol{l}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(o)})}{\partial \hat{w}_{(v,t)}^{(k)}} = (\boldsymbol{x}_v^{(t)} + \sum_{j=1}^{o} \hat{w}_{(v,j)}^{(k)} \frac{\partial \boldsymbol{x}_k^{(j)}}{\partial \hat{w}_{(v,t)}^{(k)}}) f' \tag{23}$$

where $i = o, \ldots, n$, $q = 1, \ldots, (k-1)$, $v = 1, \ldots, k$, $t = 1, \ldots, o$, f' is the derivative of $f()$. The above equations are recurrent on the structures and can be computed by observing that, for graphs composed by a single vertex, equation (21) reduces to $\frac{\partial \boldsymbol{x}_k^{(j)}}{\partial w_i^{(k)}} = l_i f'$, 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.

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.

5.4 Extension of Neural Trees

Neural Trees (NT) have recently been proposed as a fast learning method in classification tasks. They are decision trees [BFOS84] 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 [SN90] or a more complex neural network [SM91]. 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.

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 feed-forward 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 [SN90, Set90, Aa92, SM91] and unsupervised [PI92, LFJ92, Per92] splitting of the data have been proposed.

The learning and classification algorithms for binary and general trees can be found in [SM91]. 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 recursive discriminator (or network) which can be trained with any of the learning algorithms we have presented so far.

6 Computational Power

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

In this section we try to characterize the computational power of some neural network models for processing of structures. Here we use the term Elman-style network to denote a computational model where the c-model function is computed as in equation (8) and the output function as in equation (9).

6.1 Results

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

Theorem 1. *An Elman-style network can simulate any FRA.*

The proof is based on the observation that a sigmoid function can approximate a step function to an arbitrary degree of precision by augmenting the modulus of the associated weight vector. Thus, if it is demonstrated that an Elman-style network with step functions can implement any FRA, the result holds for Elman-style networks with sigmoids as well. So, using step function units, it is not difficult to show how to implement a FRA: there will be an hidden unit for each transition function of the FRA and a state is represented by the set of units associated to transition functions which introduce the same state in the computation.

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

Theorem 2. *A Cascade-Correlation network for structures cannot simulate any FRA.*

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

Theorem 3. *A Neural Tree for structures cannot simulate any FRA.*

This is demonstrated by showing the following result

Theorem 4. *Any Neural Tree for structures can be implemented by a Cascade-Correlation network for structures.*

The above theorem holds since it is observed that a Neural Tree can be cast in the form of a three layered feedforward network. The trick is to retain all the units associated with internal nodes of the Neural Tree as units of the first layer. The second layer of units is then used to represent all the possible paths from the root to one leaf. Finally, the output layer will implement the **or** of the paths associated to leaves of the corresponding class. The proof is then concluded by observing that the feedforward network belongs to the class of architectures implemented by the Cascade-Correlation for structures.

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

Corollary 5. *A Neural Tree for sequences cannot simulate any Finite State Machine.*

Corollary 6. *Any Neural Tree for sequences can be simulated by a Recurrent Cascade-Correlation network.*

6.2 Function Approximation

It is well known that feedforward neural networks can approximate arbitrarly well any continuous function given a suitable set of hidden units. A similar result holds for neural networks for structures. Specifically, given

- S set of Σ-labeled trees with Σ finite set;
- $\bar{f} : S \rightarrow \Re^q$, the desired function;
- $\mathcal{F}$ set of functions implementable by a feed-forward neural network for structures,

then the following theorem holds [HS97]

Theorem 7. *For any $\epsilon > 0$ there exists a function $f^\epsilon \in \mathcal{F}$ such that $\forall s \in S,\ P(|f^\epsilon(s) - \bar{f}(s)| > \epsilon) < \epsilon$.*

7 Some Complexity issues

A typical problem in neural networks is the evaluation of how many resources a network needs to implement the desired function. In this section, we explore this problem from a node complexity perspective when considering the implementation of *Frontier-to-Root Tree Automata* (FRA) into recursive neural networks. We give upper bounds to the number of units needed to implement a given FRA in four and three layers recursive networks. While the bound for four layers networks constitutes a generalization of the bound discussed in [HH96] for the implementation of FSM in recurrent networks, the bound for three layers networks produces, as a special case, a new bound to the implementation of FSA in three layers networks. Moreover, the bound for three layers networks is constructive and, thus, it may turn useful for practical applications. Its constructive proof is based on the encoding of each possible input configuration to the FRA by a different integer in a predefined interval. This integer is then processed by the repeated application of the *telescopic* technique [SRK95], which is called in this way because it consists in the progressive activation of a set of units according to the magnitude of the processed integer. On the basis of this progressive encoding, it is possible to implement both the transition and output function of the desired FRA.

7.1 Encoding of FRA by Boolean Functions

In [HH96] the m states of a FSA are encoded through $\log m$ bits. Hence, the state and output transition functions can be implemented by a boolean function

$$\{0,1\}^{\log m+\log l} \rightarrow \{0,1\}^{\log m+\log l} \tag{24}$$

where $\log m$ bits in input are used to encode the current status, while the remaining $\log l$ bit are used to encode the current input label. Similarly, $\log m$ bits in output are used to encode the new status and one bit to encode the output bit.

In a similar way an FRA can be implemented by a boolean function

$$\{0,1\}^{\log l+N_v \log m} \rightarrow \{0,1\}^{\log l+\log m} \tag{25}$$

where $\log l$ is the number of bits encoding the input-output symbols, and N_v is the maximum rank for the symbols.

7.2 Upper Bounds

The question we address is how many units we need to implement a tree automaton in a recursive neural network. To answer this question we consider hard threshold units. Since a sigmoid function can approximate a step function to an arbitrary degree of precision by augmenting the modulus of the associated weight vector, if we demonstrate that a recursive network with threshold gates can implement any FRA, the result holds for recursive networks with sigmoids as well. In the following we will give upper bounds for four and three layers recursive networks.

Four Layers Networks It is known that Finite State Automata with m states and binary alphabet can be inserted in four layers recurrent neural networks with $O(\sqrt{m})$ units [HH96]. This result was obtained by exploiting the following lemma by Lupanov:

Lemma 8. *Arbitrary boolean logic functions of the form* $f : \{0,1\}^x \rightarrow \{0,1\}^y$ *can be implemented in a four layer network of perceptrons with a node complexity of* $O(\sqrt{\frac{y2^x}{x-\log y}})$.

Using the same lemma we can prove the following result:

Theorem 9 Four Layers Insertion. *A four layers recurrent neural network can implement FRAs with* m *states,* l *input-output labels, and maximum rank* N_v*, with a node complexity of* $O(\sqrt{\frac{(\log l+\log m)lm^{N_v}}{\log l+N_v \log m}})$*. If* $l < m$ *the node complexity can be simplified to* $O(\sqrt{\frac{lm^{N_v}}{N_v}})$*.*

Three Layers Networks If the network is constrained to only have three layers, then by using the telescopic technique [SRK95] it can be proven that

Theorem 10 Three Layers Insertion. *A three layers recurrent neural network can implement FRAs with m states, l input-output labels, and maximum rank N_v, with a node complexity of $O((\log l + \log m)\sqrt{lm^{N_v}})$ and a number of connections of $O((\log l + \log m)lm^{N_v})$.*

The proof of this theorem is constructive and is based on the encoding of each possible input configuration to the FRA by a different integer in a predefined interval. This integer is then processed by the repeated application of the *telescopic* technique [SRK95], which is called in this way because it consists in the progressive activation of a set of units according to the magnitude of the processed integer. On the basis of this progressive encoding, it is possible to implement both the transition and output function of the desired FRA.

Note that the result stated in the theorem gives a direct upper bound to the insertion of FSA in three layers recurrent networks as well:

Corollary 11. *A three layers recurrent neural network can implement FSM having m states, and l input-output labels, with a node complexity of $O((\log l + \log m)\sqrt{lm})$ and a number of connections of $O((\log l + \log m)lm)$.*

8 Conclusion

The possibility to represent and to process structures in a neural network greatly increases the computational capability of neural networks. This new capability, besides to provide a new tool for the classification of structures, can also be exploited to integrate neural networks and symbolic systems in a hybrid system. In fact, structures generated by a symbolic module can be evaluated by this type of networks and their evaluation can be used to modify the behavior of the symbolic module. An instance of this integration scheme is given, for example, by learning heuristics for automated deduction systems. Goller reported very successful results in using a Back-propagation Through Structure network within the SETHEO theorem prover [Gol97]. On the other side, it is not difficult to figure out, in analogy with finite state automata extraction from recurrent networks, how to extract tree automata from a neural network for structures. This would allow the above scheme to work on the other side around, with a neural module which is driven by a symbolic subsystem.

The computational results presented in this tutorial, however, need to be extended to DOAGs in order to fully characterize the kind of symbolic computations which can be performed by recursive neural network. Nevertheless it must be pointed out that, due to its numerical nature, recursive neural networks perform a kind of computation which is not possible to be easily reproduced by a symbolic system. The impact of this new style of computation should find very useful application in domains where structure and numerical values are both important aspects of the computational problem, such as in the prediction of the biological activity of drugs in Chemistry.

References

[Aa92] L. Atlas and al. A performance comparison of trained multilayer perceptrons and trained classification trees. *Proceedings of the IEEE*, 78:1614–1619, 1992.

[ADO91] N. Alon, A. K. Dewdney, and T. J. Ott. Efficient simulation of finite automata by neural nets. *Journal of the Association for Computing Machinery*, 38(2):495–514, 1991.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

[BFS94] Y. Bengio, P. Frasconi, and P. Simard. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(5):157–166, March 1994. Special Issue on Recurrent Neural Networks.

[Fah91] S. E. Fahlman. The recurrent cascade-correlation architecture. Technical Report CMU-CS-91-100, Carnegie Mellon, 1991.

[FL90] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 524–532. San Mateo, CA: Morgan Kaufmann, 1990.

[GCS+95] 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 Transactions on Neural Networks*, 6(4):829–836, 1995.

[GGCC94] M.W. Goudreau, C.L. Giles, S.T. Chakradhar, and D. Chen. First-order vs. second-order single layer recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(3):511–513, 1994.

[GK96] C. Goller and A. Küchler. Learning task-dependent distributed structure-representations by backpropagation through structure. In *IEEE International Conference on Neural Networks*, pages 347–352, 1996.

[Gol97] C. Goller. *A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems.* PhD thesis, Technical University Munich, Computer Science, 1997. .

[GT78] R. C. Gonzalez and M. G. Thomason. *Syntactical Pattern Recognition.* Addison-Wesley, 1978.

[HH96] B. G. Horne and D. R. Hush. Bounds on the complexity of recurrent neural network implementations of finite state machines. *Neural Networks*, 9(2):243–252, 1996.

[HK91] L. H. Hall and L. B. Kier. *Reviews in Computational Chemistry*, chapter 9, The Molecular Connectivity Chi Indexes and Kappa Shape Indexes in Structure-Property Modeling, pages 367–422. VCH Publishers, Inc.: New York, 1991.

[HS97] B. Hammer and V. Sperschneider. Neural networks can approximate mappings on structured objects. In *Proceedings of the 2nd International Conference on Computational Intelligence and Neuroscience*, 1997. Research Triangle Park, USA.

[KG96] A. Küchler and C. Goller. Inductive Learning in Symbolic Domains Using Structure-Driven Recurrent Neural Networks. In Günther Görz and Steffen Hölldobler, editors, *KI-96: Advances in Artificial Intelligence*, Lecture Notes in Computer Science (LNCS 1137), pages 183–197, Berlin, 1996. Springer.

[Kre95] S. C. Kremer. Comments on "constructive learning of recurrent neural networks: ...", cascading the proof describing limitations of recurrent cascade correlation. *IEEE Transactions on Neural Networks*, 1995. In press.

[Kre96] S.C. Kremer. Finite state automata that recurrent cascade-correlation cannot represent. In D. Touretzky, M. Mozer, and M. Hasselno, editors, *Advances in Neural Information Processing Systems 8*. MIT Press, 1996. 612-618.

[LFJ92] T. Li, L. Fang, and A. Jennings. Structurally adaptive self-organizing neural trees. In *International Joint Conference on Neural Networks*, pages 329–334, 1992.

[OG96] C.W. Omlin and C.L. Giles. Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM*, 43(6):937–972, 1996.

[Per92] M. P. Perrone. A soft-competitive splitting rule for adaptive tree- structured neural networks. In *International Joint Conference on Neural Networks*, pages 689–693, 1992.

[PI92] M. P. Perrone and N. Intrator. Unsupervised splitting rules for neural tree classifiers. In *International Joint Conference on Neural Networks*, pages 820–825, 1992.

[Pol90] J. B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2):77–106, 1990.

[RM86] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, 1986.

[Set90] I. K. Sethi. Entropy nets: From decision trees to neural networks. *Proceeding of the IEEE*, 78:1605–1613, 1990.

[SKG97] S. Schulz, A. Küchler, and C. Goller. Some Experiments on the Applicability of Folding Architecture Networks to Guide Theorem Proving. In *Proceedings of the 10th International FLAIRS Conference*, 1997.

[SM91] A. Sankar and R. Mammone. *Neural Tree Networks*, pages 281–302. Neural Networks: Theory and Applications. Academic Press, 1991.

[SN90] J. A. Sirat and J-P. Nadal. Neural trees: a new tool for classification. *Network*, 1:423–438, 1990.

[SRK95] K.-Y. Siu, V. Roychowdhury, and T. Kailath. *Discrete Neural Computation.* Englewood Cliffs, New Jersey: Prentice Hall, 1995.

[SS95] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

[SS97] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[SSG95] A. Sperduti, A. Starita, and C. Goller. Learning distributed representations for the classification of terms. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 509–515, 1995.

[Tha73] J. W. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[WZ89] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.