Neural Networks for Adaptive Processing of Structured Data

Alessandro Sperduti

Dip. di Informatica, Università di Pisa, Corso Italia, 40, 56125 Pisa, Italy

Abstract. 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 that use, generate or manipulate structures. In this paper I review some of the concepts underpinning Recursive Neural Networks, i.e. neural network models able to deal with data represented as directed acyclic graphs.

1 Introduction

The processing of structured data is usually confined to the domain of symbolic systems. Recently, however, there has been some effort in trying to extend the computational capabilities of neural networks to structured domains. While earlier neural approaches were able to deal with some aspects of processing of structured information, none of them established a practical and efficient way of dealing with structured information. A more powerful approach, at least for classification and prediction tasks, was proposed in [15] and further extended in [7]. In these works, Recursive Neural Networks, a generalization of recurrent neural networks for processing sequences to the case of directed graphs, were defined. These models are able to learn a mapping from a domain of ordered (or positional) directed acyclic graphs, with labels attached to each node, to the set of real numbers. The basic idea behind the models is the extension of the concept of *unfolding* from the domain of sequences to the domain of directed ordered graphs (DOGs).

In this paper, I briefly present some of the basic concepts underpinning Recursive Neural Networks. Some supervised and unsupervised models are presented, together with an outlook of the main computational, complexity, and leaning results obtained up to now.

The possibility of processing structured information using neural networks is appealing for several reasons. First of all, neural networks are universal approximators; in addition, they are able to learn from a set of examples and very often, by using the correct methodology for training, they are able to reach a quite high generalization performance. Finally, they are able to deal with noise and incomplete, or even ambiguous, data. All these capabilities are particularly useful when dealing with prediction tasks where data are usually gathered experimentally, and thus are partial, noisy, and incomplete. A typical example of such a domain is chemistry, where compounds can naturally be represented as labeled graphs.

2 Data Structures and Notation

In this paper we assume that instances in the learning domain are DOAGs (directed ordered acyclic graphs) or DPAGs (directed positional acyclic graphs). A DOAG is a DAG $\mathcal{D}$ with vertex set $\mathrm{vert}(\mathcal{D})$ and edge set $\mathrm{egd}(\mathcal{D})$, where for each vertex $v \in \mathrm{vert}(D)$ a total order on the edges leaving from v is defined. DPAGs are a superclass of DOAGs in which it is assumed that for each vertex v, a bijection $P : \mathrm{egd}(\mathcal{D}) \to I\!N$ is defined on the edges leaving from v. The *indegree* of node v is the number of incoming edges to v, whereas the *outdegree* of v is the number of outgoing edges from v.

We shall require the DAG (either DOAG or DPAG) to possess a supersource[1], i.e. a vertex $s \in \mathrm{vert}(\mathcal{D})$ such that every vertex in $\mathrm{vert}(\mathcal{D})$ can be reached by a directed path starting from s. Given a DAG $\mathcal{D}$ and $v \in \mathrm{vert}(\mathcal{D})$, we denote by $\mathrm{ch}[v]$ the set of children of v, and by $\mathrm{ch}_k[v]$ the k-th child of v.

We shall use lowercase bold letters to denote vectors, uppercase bold letters to denote matrices, and calligraphic letters for representing graphs. A data structure $\mathcal{Y}$ is a DAG whose vertices are labeled by vectors of real-valued numbers which either represent numerical or categorical variables. Subscript notation will be used when referencing the labels attached to vertices in a data structure. Hence $\boldsymbol{y}_v$ denotes the vector of variables labeling vertex $v \in \mathrm{vert}(\mathcal{Y})$.

In the following, we shall denote by $\#^{(i,c)}$ the class of DAGs with maximum indegree i and maximum outdegree c. A generic class of DAGs with bounded (but unspecified) indegree and outdegree, will simply be denoted by $\#$. The class of all data structures defined over the label universe domain $\boldsymbol{\mathcal{Y}}$ and skeleton in $\#^{(i,c)}$ will be denoted as $\boldsymbol{\mathcal{Y}}^{\#^{(i,c)}}$. The void DAG will be denoted by the special symbol ξ.

3 Recursive Neural Networks

Recursive Neural Networks are neural networks able to perform mappings from a set of labeled graphs to the set of real vectors. Specifically, the class of functions which can be realized by a recursive neural network can be characterized as the class of functional graph transductions $\mathcal{T} : \mathcal{I}^{\#} \to I\!R^k$, where $\mathcal{I} = I\!R^m$, which can be represented in the following form $\mathcal{T} = g \circ \widehat{\tau}$, where $\widehat{\tau} : \mathcal{I}^{\#} \to I\!R^n$ is the *encoding* (or *state transition*) function and $g : I\!R^n \to I\!R^k$ is the *output* function. Specifically, given a DOAG $\mathcal{D}$, $\widehat{\tau}$ is defined recursively as

$$\widehat{\tau}(\mathcal{D}) = \begin{cases} \mathbf{0} \text{ (the null vector in } I\!R^n) & \text{if } \mathcal{D} = \xi \\ \tau(\boldsymbol{y}_s, \widehat{\tau}(\mathcal{D}^{(1)}), \ldots, \widehat{\tau}(\mathcal{D}^{(c)})) & \text{otherwise} \end{cases} \tag{1}$$

[1] If no supersource is present, a new node connected with all the nodes of the graph with null *indegree* can be added.

where τ is defined as $\tau : I\!R^m \times \underbrace{I\!R^n \times \cdots \times I\!R^n}_{\text{c times}} \rightarrow I\!R^n$ where $I\!R^m$ denotes the label space, while the remaining domains represent the encoded subgraphs spaces up to the maximum out-degree of the input domain $\mathcal{I}^\#$, c is the maximum out-degree of DOAGs in $\mathcal{I}^\#$, $s = source(\mathcal{D})$, $\boldsymbol{y}_s$ is the label attached to the super-source of $\mathcal{D}$, and $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(c)}$ are the subgraphs pointed by s. A typical neural realization for τ is

$$\tau(\boldsymbol{u}_v, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(c)}) = \boldsymbol{F}(\boldsymbol{B}\boldsymbol{u}_v + \sum_{j=1}^{c} \boldsymbol{A}_j \boldsymbol{x}^{(j)} + \boldsymbol{\theta}), \tag{2}$$

where $\boldsymbol{F}_i(\boldsymbol{v}) = f(v_i)$ (sigmoidal function), $\boldsymbol{u}_v \in I\!R^m$ is a label, $\boldsymbol{\theta} \in I\!R^n$ is the bias vector, $\boldsymbol{B} \in I\!R^{m \times n}$ is the weight matrix associated with the label space, $\boldsymbol{x}^{(j)} \in I\!R^m$ are the vectorial codes obtained by the application of the encoding function $\widehat{\tau}$ to the subgraphs $\mathcal{D}^{(j)}$ (i.e., $\boldsymbol{x}^{(j)} = \widehat{\tau}(\mathcal{D}^{(j)})$), and $\boldsymbol{A}_j \in I\!R^{m \times m}$ is the weight matrix associated with the jth subgraph space. Concerning the output function g, it can be defined as a map $g : I\!R^n \rightarrow I\!R^k$, and in general, it is realized by a feed-forward network.

The encoding process of an input graph can be represented graphically by unfolding equation 1 through the input graph, and using equation 2, obtaining in this way the so called *encoding network*. An example of encoding network obtained by using two recursive neurons an a single output neuron is shown in Figure 1. The output of the encoding network depends on the values of the weights and it will used as a numerical vectorial code to represent the input graph. This code will then be further processed by the parametric function $g()$ (in the example, the single output neuron) so to obtain the desired regression (or classification) value for the specific input graph.

Given a cost function E and a training set $T = \{(\mathcal{D}_i, t_i)\}_{i=1,..,L}$, where for each data structure $\mathcal{D}_i$ a desired target value t_i is associated, using eq. 1, eq. 2, and the neural network implementing the output function g, for each $\mathcal{D}_i$ a feed-forward network can be generated and trained so to match the corresponding desired target value t_i. Since the weights are shared among all the generated feed-forward networks, the training converges to a set of weight values which reproduces the desired target value for each data structure in the training set.

For each $\mathcal{D}_l$, its vertexes are enumerated according to a chosen inverse topological order as $\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_{P_l}$. Moreover, let $\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_h$ be the set of vertexes, belonging to any DAG in the training set, for which a target value is defined. For simplicity, here we assume that only the supersources have a target defined (thus $h = L$). $\boldsymbol{U} \doteq [\boldsymbol{U}_1, \cdots, \boldsymbol{U}_L] \in R^{m+1,P^*}$ collects all the labels of the data structures (including the bias components always to 1), where $P^* \doteq \sum_{l=1}^{L} P_l$. Similarly, for each graph $\mathcal{D}_l$, and each $k \in [1, \ldots, c]$, the matrices $\boldsymbol{X}_l^{(k)} \in I\!R^{n,P_l}$ and defined as $\boldsymbol{X}_l^{(k)} \doteq [\boldsymbol{x}_{l,ch_k[\tilde{v}_1]}, \ldots, \boldsymbol{x}_{l,ch_k[\tilde{v}_{P_l}]}]$, where $\boldsymbol{x}_{l,ch_k[\tilde{v}_i]} = \boldsymbol{x}_0$ if $ch_k[\tilde{v}_i] = \emptyset$, collect the status information for each pointer k. All the information concerning a pointer k is stored into matrices $\boldsymbol{X}^{(k)} \doteq \left[\boldsymbol{X}_1^{(k)}, \ldots, \boldsymbol{X}_L^{(k)}\right] \in I\!R^{n,P^*}$. Moreover, we define the matrix collecting the information needed to compute the output

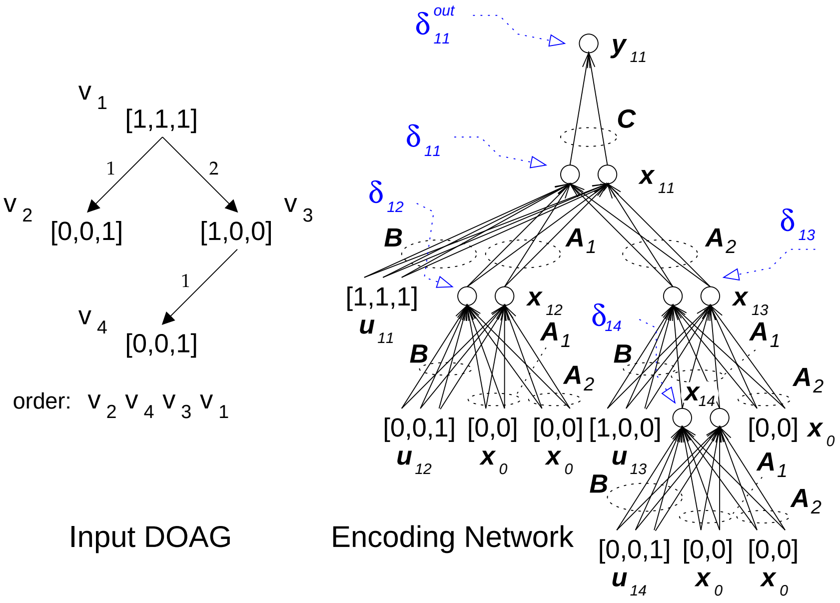

Fig. 1. Example of DOAG with corresponding encoding network.

associated to the training graphs as $\boldsymbol{X}^{target} = \begin{bmatrix} \boldsymbol{x}_{\hat{v}_1} & \boldsymbol{x}_{\hat{v}_2} & \dots & \boldsymbol{x}_{\hat{v}_L} \\ 1 & 1 & & 1 \end{bmatrix}$. Let us define $\delta_{ilv} \doteq \partial E_l / \partial net_{ilv}$. For node v of a given DOAG $\mathcal{D}_l$, the corresponding delta error δ_{ilv} for the state variables can be collected in vector $\boldsymbol{\delta}_{lv} \doteq [\delta_{1lv}, \dots, \delta_{nlv}]^t$ and the contributions from all the graph's nodes can be collected in matrix $\boldsymbol{\Delta}_l \doteq [\boldsymbol{\delta}_{l,1}, \dots, \boldsymbol{\delta}_{l,P_l}] \in I\!R^{n,P_l}$, where the order of the columns follows the inverse topological order chosen for the graph. Finally, $\boldsymbol{\Delta} \doteq [\boldsymbol{\Delta}_1, \dots, \boldsymbol{\Delta}_L] \in I\!R^{n,P^*}$ contains the delta errors for all the graphs of the learning environment, whereas the delta error corresponding to the output unit is denoted by $\delta^{out}_{ls} \doteq \frac{\partial E_l}{\partial net_{ls}}$ and collected into the matrix $\hat{\boldsymbol{\Delta}}$. The gradient of the cost can be calculated by using Backpropagation in each encoding network, that is by propagating the error through the given structure, similarly to what happens in recurrent networks when processing sequences. The gradient of the cost can be written in a compact form by using the vectorial notation $\boldsymbol{G}_B \doteq \left[\frac{\partial E}{\partial b_{ij}}\right] \in I\!R^{m+1,n}$ and $\mathbf{G}_{A_k} \doteq \left[\frac{\partial E}{\partial a_{ijk}}\right] \in I\!R^{n,n}$. Based on these definitions, $\boldsymbol{G}_B$ and $\boldsymbol{G}_{A_k}$ can be computed as follows

$$
\begin{aligned}
\boldsymbol{G}_B &= \sum_{l=1}^{L} \boldsymbol{G}_{B,l} = \sum_{l=1}^{L} \sum_{v \in \mathrm{vert}(\mathcal{D}_l)} \boldsymbol{u}_{lv} \boldsymbol{\delta}^t_{lv} = \boldsymbol{U} \boldsymbol{\Delta}^t, \\
\boldsymbol{G}_{A_k} &= \sum_{l=1}^{L} \boldsymbol{G}_{A_k,l} = \sum_{l=1}^{L} \sum_{v \in \mathrm{vert}(\mathcal{D}_l)} \boldsymbol{x}_{l,ch_k[v]} \; \boldsymbol{\delta}^t_{lv} = \boldsymbol{X}^{(k)} \boldsymbol{\Delta}^t,
\end{aligned} \tag{3}
$$

where $\mathbf{G}_{B,l}$, $\mathbf{G}_{A_k,l}$ are the gradient contributions corresponding to E_l, that is to DOAG $\mathcal{D}_l$. Specifically, let $Q_k(v) \doteq \{u \mid ch_k[u] = v\}$. The *delta-error* δ_{ilv} can be computed recursively according to the following equation

$$\delta_{ilv} = \sigma'(net_{ilv}) \sum_{k=1}^{o} \sum_{j=1}^{n} a_{kji} \left(\sum_{z \in Q_k(v)} \delta_{jlz} \right) \tag{4}$$

where if $Q_k(v) = \emptyset$ then $\sum_{z \in Q_k(v)} \delta_{jlz} = 0$.

The above equation can be rewritten in compact form as

$$\boldsymbol{\delta}_{lv} = \boldsymbol{J}_{lv} \sum_{k=1}^{o} \boldsymbol{A}_k \left(\sum_{z \in Q_k(v)} \boldsymbol{\delta}_{lz} \right), \tag{5}$$

where $\boldsymbol{J}_{lv}$ is a diagonal matrix with elements $[\boldsymbol{J}_{lv}]_{ii} = \sigma'(net_{ilv})$. Moreover, by applying recursively equation (5), we obtain

$$\boldsymbol{\delta}_{lv} = \left(\sum_{p \in Paths_l(s,v)} \prod_{(u,ch_k[u]) \in p}^{(s \rightarrow v)_{left}} \boldsymbol{J}_{l,ch_k[u]} \boldsymbol{A}_k^t \right) \boldsymbol{\delta}_{ls} \tag{6}$$

where $Paths_l(s, v)$ is the set of paths in $\mathcal{D}_l$ from the supersource s to node v, and the product is left-hand starting from the supersource s and ending to node v. This equation gives rise to the *Back-Propagation Through Structure* ($BPTS$) gradient computational scheme [9, 15].

Following the same approach, it is not difficult to generalize any supervised learning algorithm, such as RTRL, to the treatment of structured data. Constructive algorithms, such as Recurrent Cascade-Correlation, can be generalized as well [15].

A common features of all the supervised models is that, assuming stationarity, causality, and discrete labels, the training set can be optimized so to reduce the computational complexity of training. The basic idea is to represent each subgraph in the training set only once: if two graphs share the same subgraph, this subgraph only needs to be represented once, as well as only once the state associated to it must be computed. Applying this basic idea, the training set can be collapsed into a single minimal DOAG (see Figure 2 for an example) in time which is $O(P^* \log P^*)$.

3.1 Unsupervised Learning: Self-Organizing Maps for Structured Data

A Self-Organizing Map model for processing structured data has been recently proposed [11]. This model hinges on the recursive encoding idea described in eq. 1. The aim of the SOM learning algorithm is to learn a *feature map* $\mathcal{M} : \boldsymbol{\mathcal{I}} \rightarrow \boldsymbol{\mathcal{A}}$ which given a vector in the spatially continuous input space $\boldsymbol{\mathcal{I}}$ returns a point in the spatially *discrete* output display space $\boldsymbol{\mathcal{A}}$. In fact, SOM is performing

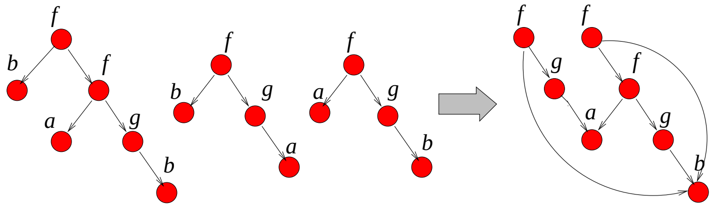

Fig. 2. Optimization of the training set.

data reduction via a vector quantization approach. When the input space is a structured domain with labels in $\boldsymbol{\mathcal{Y}}$, i.e., $\boldsymbol{\mathcal{I}} \equiv \boldsymbol{\mathcal{Y}}^{\#^{(i,c)}}$, the map $\widehat{\mathcal{M}} : \boldsymbol{\mathcal{Y}}^{\#^{(i,c)}} \to \boldsymbol{\mathcal{A}}$ can be realized by the following recursive definition:

$$\widehat{\mathcal{M}}(\mathcal{D}) = \begin{cases} nil_{\boldsymbol{\mathcal{A}}} & \text{if } \mathcal{D} = \xi \\ \overset{\circ}{\mathcal{M}}\left(\boldsymbol{y}_s, \widehat{\mathcal{M}}(\mathcal{D}^{(1)}), \ldots, \widehat{\mathcal{M}}(\mathcal{D}^{(c)})\right) & \text{otherwise} \end{cases}$$

where $nil_{\boldsymbol{\mathcal{A}}}$ is a special coordinate (the void coordinate) into the discrete output space $\boldsymbol{\mathcal{A}}$, and $\overset{\circ}{\mathcal{M}}: \boldsymbol{\mathcal{Y}} \times \boldsymbol{\mathcal{A}}^c \to \boldsymbol{\mathcal{A}}$ is a SOM, defined on a generic node, which takes in input the label of the node and the "encoding" of the subgraphs $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(c)}$ according to the $\widehat{\mathcal{M}}$ map. By "unfolding" the recursive definition, it turns out that $\widehat{\mathcal{M}}(\mathcal{D})$ can be computed by starting to apply $\overset{\circ}{\mathcal{M}}$ to leaf nodes (i.e., nodes with null outdegree), and proceeding with the application of $\overset{\circ}{\mathcal{M}}$ bottom-up from the frontier to the supersource of the graph $\mathcal{D}$.

Assuming that each label in $\boldsymbol{\mathcal{Y}}$ is encoded in $\boldsymbol{\mathcal{U}} \subset I\!R^m$, for each $v \in \mathrm{vert}(\mathcal{D})$, we have a vector $\boldsymbol{u}_v$ of dimension m. Moreover, the display output space $\boldsymbol{\mathcal{A}}$, where $\boldsymbol{\mathcal{A}} \equiv [1..n_1] \times [1..n_2] \times \cdots \times [1..n_q]$, is realized through a q dimensional lattice of neurons. So, the winning neuron is represented by the coordinates $(i_1, .., i_q)$.

With the above assumptions, we have that $\overset{\circ}{\mathcal{M}}: I\!R^m \times ([1..n_1] \times \cdots \times [1..n_q])^c \to [1..n_1] \times \cdots \times [1..n_q]$, and the $m+cq$ dimensional input vector $\boldsymbol{v}$ to $\overset{\circ}{\mathcal{M}}$, representing the information about a generic node v, is defined as $\boldsymbol{v} = \left[\boldsymbol{u}_v \; \mathcal{D}_{ch_1[v]} \; \mathcal{D}_{ch_2[v]} \; \cdots \; \mathcal{D}_{ch_c[v]}\right]$, where $\mathcal{D}_{ch_i[v]}$ is the coordinates vector[2] of the winning neuron for the subgraph pointed by the i-th pointer of v.

Of course, each neuron with coordinates vector $\boldsymbol{c}_j$ in the q dimensional lattice has an associated vector weight $\boldsymbol{w}_{\boldsymbol{c}_j} \in I\!R^{m+cq}$. The weights associated with each neuron in the q dimensional lattice $\overset{\circ}{\mathcal{M}}$ can be trained using the standard SOM's learning process, while the training algorithm for $\widehat{\mathcal{M}}$ is shown in Figure 3.

The coordinates for the (sub)graphs are stored in $\mathcal{D}_v$, once for each processing of graph $\mathcal{D}$, and then used when needed[3] for the training of $\overset{\circ}{\mathcal{M}}$. Of course, the

[2] The null pointer $nil_{\boldsymbol{\mathcal{A}}}$ can be defined, for example, by the vector with all components at -1.

[3] Notice that the use of an inverted topological order guarantees that the updating of the coordinate vectors $\mathcal{D}_v$ is done before the use of $\mathcal{D}_v$ for training.

Unsupervised Stochastic Training Algorithm for $\widehat{\mathcal{M}}$

input: $\{\mathcal{D}_i\}_{i=1,\ldots,N}$, c, $\mathring{\mathcal{M}}$;
begin
  randomly set the weights for $\mathring{\mathcal{M}}$;
  repeat
    randomly select $\mathcal{D} \in T$ with uniform distribution;
    `List(`$\mathcal{D}$`)` $\leftarrow$ an inverted topological order for vert($\mathcal{D}$);
    for $v \leftarrow$`first(List(`$\mathcal{D}$`))` **to** `last(List(`$\mathcal{D}$`))`**do**
      train($\mathring{\mathcal{M}}\left(\left[\boldsymbol{u}_v\ \mathcal{D}_{ch_1[v]}\ \mathcal{D}_{ch_2[v]}\ \cdots\ \mathcal{D}_{ch_c[v]}\right]\right)$);
      $\mathcal{D}_v \leftarrow \mathring{\mathcal{M}}\left(\left[\boldsymbol{u}_v\ \mathcal{D}_{ch_1[v]}\ \mathcal{D}_{ch_2[v]}\ \cdots\ \mathcal{D}_{ch_c[v]}\right]\right)$;
end

Fig. 3. The unsupervised SOM for structured data.

stored vector is an approximation of the true coordinate vector for the graph rooted in v, however, if the learning rate is small, this approximation may be negligible. The call *train*() refers to a single step of standard SOM training.

Preliminary experimental results showed that the model was actually able to cluster similar (sub)structures and to organize the input (sub)graphs according to both information encoded in the labels and in the structures.

Computational, Complexity, and Learnability Issues The computational power of Recursive Neural Networks has been studied in [14] by using hard threshold units and frontier-to-root tree automata. In [3], several strategies for encoding finite-state tree automata in high-order and first-order sigmoidal recursive neural networks have been proposed.

Complexity results on the amount of resources needed to implement frontier-to-root tree automata in recursive neural networks are presented in [10].

Results on function approximation and theoretical analysis of learnability and generalization of recursive neural networks (referred as folding networks) can be found in [12, 13]. Finally, an analysis of sufficient conditions that guarantee the absence of local minima when training recursive neural networks can be found in [6].

4 Conclusion

Neural networks can deal with structured data. Here I have discussed some of the basic concepts underpinning Recursive Neural Networks. Applications of Recursive Neural Networks are starting to emerge. They include learning search-control heuristics for automated deduction systems [8], logo recognition [5], chemical applications (QSPR/QSAR) [1, 2], incremental parsing of natural language [4]. More work is needed to progress in this field. For example, no accepted proposal for expanding the processing of neural networks to cyclic graphs has yet been established.

References

1. A.M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Application of cascade correlation networks for structures to chemistry. *Applied Intelligence*, 12:115–145, 2000.
2. A.M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Analysis of the internal representations developed by neural networks for structures applied to quantitative structure-activity relationship studies of benzodiazepines. *J. Chem. Inf. Comput. Sci.*, 41(1):202–218, 2001.
3. R.C. Carrasco and M.L. Forcada. Simple strategies to encode tree automata in sigmoid recursive neural networks. *IEEE TKDE*, 13(2):148–156, 2001.
4. F. Costa, P. Frasconi, V. Lombardo, and G. Soda. Towards incremental parsing of natural language using recursive neural networks. *Applied Intelligence*, To appear.
5. E. Francesconi, P. Frasconi, M. Gori, S. Marinai, J. Q. Sheng, G. Soda, and A. Sperduti. Logo recognition by recursive neural networks. In R. Kasturi and K. Tombre, editors, *Graphics Recognition – Algorithms and Systems*, pages 104–117. LNCS, Vol. 1389, Springer Verlag, 1997.
6. P. Frasconi, M. Gori, and A. Sperduti. On the efficient classification of data structures by neural networks. In *IJCAI*, pages 1066–1071, 1997.
7. P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE TNN*, 9(5):768, 1998.
8. C. Goller. *A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems.* PhD thesis, Technical University Munich, Computer Science, 1997.
9. 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.
10. M. Gori, A. Kuchler, and A. Sperduti. On the implementation of frontier-to-root tree automata in recursive neural networks. *IEEE Transactions on Neural Networks*, 10(6):1305 –1314, 1999.
11. M. Hagenbuchner, A. Sperduti, and A. C. Tsoi. A supervised self-organising map for structured data. In *WSOM'01*, 2001.
12. B. Hammer. On the learnability of recursive data. *Mathematics of Control Signals and Systems*, 12:62–79, 1999.
13. B. Hammer. Generalization ability of folding networks. *IEEE TKDE*, 13(2):196–206, 2001.
14. A. Sperduti. On the computational power of recurrent neural networks for structures. *Neural Networks*, 10(3):395–400, 1997.
15. A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.