A Comparison between Recursive Neural Networks and Graph Neural Networks

Conference Paper · January 2006

DOI: 10.1109/IJCNN.2006.246763 · Source: DBLP

Gabriele Monfardini
Università degli Studi di Siena
9 PUBLICATIONS **671** CITATIONS

Franco Scarselli
Università degli Studi di Siena
83 PUBLICATIONS **1,646** CITATIONS

Marco Maggini
Università degli Studi di Siena
154 PUBLICATIONS **1,822** CITATIONS

Marco Gori
Università degli Studi di Siena
227 PUBLICATIONS **4,064** CITATIONS

A Comparison between Recursive Neural Networks and Graph Neural Networks

Vincenzo Di Massa, Gabriele Monfardini, Lorenzo Sarti, Franco Scarselli, Marco Maggini, Marco Gori

***Abstract*— Recursive Neural Networks (RNNs) and Graph Neural Networks (GNNs) are two connectionist models that can directly process graphs. RNNs and GNNs exploit a similar processing framework, but they can be applied to different input domains. RNNs require the input graphs to be directed and acyclic, whereas GNNs can process any kind of graphs. The aim of this paper consists in understanding whether such a difference affects the behaviour of the models on a real application. An experimental comparison on an image classification problem is presented, showing that GNNs outperforms RNNs. Moreover the main differences between the models are also discussed w.r.t. their input domains, their approximation capabilities and their learning algorithms.**

I. Introduction

In several application domains the data of interest can be naturally described by trees or graphs. In fact, nodes can be used to represent objects while edges intuitively define the relationships between them. These applications comprise problems from pattern recognition, natural language processing, Web page scoring and relational learning [1]–[6].

Classical graph theory includes several theoretical studies and ad–hoc methods that deal with graph classification (or regression), graph matching, link prediction, and similarity measures for non–flat data [7]–[9].

Standard machine learning approaches require to map graphs or trees into simpler representations, like vectors or sequences of real numbers. However, their performances vary largely with the application at hand. In fact, the preprocessing phase is rather problem dependent and its design usually results in a time–consuming trial and error procedure. Moreover, the topological information inherently contained in structural representations might be partially lost or difficult to be exploited.

More recently, new connectionist models, capable to directly elaborate graphs and trees without a preprocessing phase were proposed [10]. They extend support vector machines [11]–[14], neural networks [15]–[18] and SOMs [19] to structured data. Such techniques have been applied successfully on several problems as logical term classification [20], chemical compound classification [21], logo recognition [2], [22], Web page scoring [23], and face localization [24].

The main idea is to obtain, adaptively and automatically, a flat representation of the symbolic and subsymbolic information collected in the structures. While in kernels for graphs the internal encoding is defined by the selected kernel, in neural network models it is automatically learned by examples. Finally, SOMs–SD differ from the other methods since they implement an unsupervised learning framework, instead of a supervised one.

The authors are with the Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, Siena, Italy. Email: {dimassa,monfardini,sarti,franco,maggini,marco}@dii.unisi.it.

Graph Neural Networks have been recently proposed to process very general types of graphs and can be considered an extension of RNNs. Actually, RNNs require input graphs to be directed and acyclic, while cyclic or non–directed structures must undergo a preprocessing phase. However, GNNs have not been widely tested yet, and it is unknown whether the performance of GNNs and RNNs is different in practical applications. This paper presents an experimental comparison between GNNs and RNNs. The two models are evaluated on a real–world computer vision problem, which consists in classifying a set of images. Moreover, the theoretical differences between the two models are also investigated, paying attention to their admitted input domains, their approximation capabilities and their learning algorithms.

The paper is organized as follows. In the next section, the notation adopted in the paper is introduced. Sect. III reviews the RNN and the GNN model. Sect. IV illustrates the real–world computer vision task and shows the results obtained by RNNs and GNNs. Finally, the conclusions are drawn in Sect. V.

II. Notation

A *labeled graph* (or *graph*) $\boldsymbol{G}$ is a quadruple $(\boldsymbol{N},\boldsymbol{E},\mathcal{L},\mathcal{E})$, where $\boldsymbol{N}$ is the set of nodes (or vertices), and $\boldsymbol{E}$ is the set of edges between nodes: $\boldsymbol{E} \subseteq \{(u,v)|u,v \in \boldsymbol{N}\}$. Nodes and edges constitute the *skeleton* of the graph. The last two items, that associate vectors of real numbers of dimension respectively $\mathbb{R}^{l_N}$ and $\mathbb{R}^{l_E}$ to each node and edge, are a *node-labeling* function $\mathcal{L} : \boldsymbol{N} \to \mathbb{R}^{l_N}$ and an *edge-labeling* function $\mathcal{E} : \boldsymbol{E} \to \mathbb{R}^{l_E}$. Node labels are represented by $\boldsymbol{l}_n$ and edge labels by $\boldsymbol{l}_{(u,v)}$. In the following, the symbol $\boldsymbol{l}$ with no further specification is the vector obtained by stacking together all the labels of the graph. Finally, the symbol $|\cdot|$ denotes the cardinality or the absolute value, according to whether it is applied to a set or to a number.

If the graph is directed, an edge (u,v) is an *ordered* pair of nodes, where u is the *father* and v the *child*. If the graph is undirected, the ordering between u and v in (u,v) is not defined, i.e. $(u,v) = (v,u)$. A graph is called *acyclic* if there is no path, i.e. a sequence of connected edges, that starts and ends in the same node. Given a node n, the nodes adjacent to n (or neighbors of n) are those connected to it by an edge, and are represented by ne$[n]$. If the graph is directed, the neighbors of n either belong to the set of its children (ch$[n]$), or to the set of its parents (pa$[n]$). Moreover, a graph can be said *positional* or *non–positional* according to whether a function π_n is defined for each node n that assigns a different position $\pi_n(u)$ to each neighbor $u \in \text{ne}[n]$. Thus, combining the properties described so far it is possible to specify various graph categories: Directed Positional Acyclic Graphs (DPAGs), Directed Acyclic Graphs (DAGs) and so on.

The goal of the proposed models consists in approximating a target function $\tau(\boldsymbol{G}, n) \in \mathbb{R}^m$ that maps a graph $\boldsymbol{G}$ and one of its nodes n into a vector of real numbers[1]. In graph classification, τ does not depend on n, i.e. only one target is given for each graph. In node classification problems, each node in a given set has a target to be approximated.

III. Recursive and graph neural networks

A very general framework for graph processing must implement a function φ that computes an output $\varphi(\boldsymbol{G}, n)$ for each pair $(\boldsymbol{G}, n)$. The main idea is to derive a flat description of the information related to each node n. A node represents an object of the domain of interest: its description is a vector of real numbers called *state* and denoted by $\boldsymbol{x}_n \in R^s$, where the state dimension s is a predefined parameter. In order to obtain a distributed processing framework, the states are computed locally at each node. A reasonable choice is to design $\boldsymbol{x}_n$ as the output of a parametric function $f_{\boldsymbol{w}}$ (called *state transition function*), that depends on the node label $\boldsymbol{l}_n$ and on the relationships between n and its neighbors

$$\boldsymbol{x}_n = f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{x}_{\text{ne}[n]}, \boldsymbol{l}_{\text{ne}[n]}, \boldsymbol{l}_{(n,\text{ne}[n])}), \quad n \in \boldsymbol{N}, \quad (1)$$

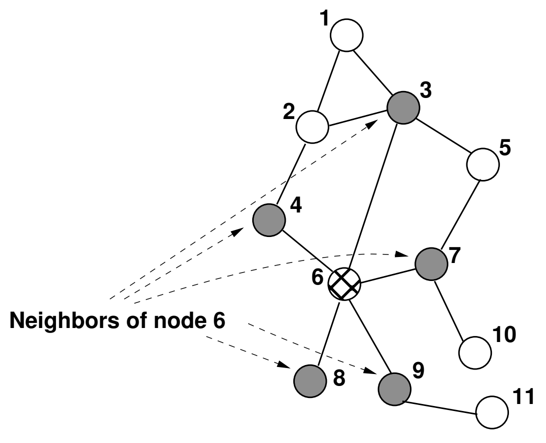
where ne$[n]$ is the set of neighbors of node n, $\boldsymbol{x}_{\text{ne}[n]}$ and $\boldsymbol{l}_{\text{ne}[n]}$ are the sets containing the states and the labels of the nodes in ne$[n]$ respectively, and $\boldsymbol{l}_{(n,\text{ne}[n])}$ is the set of the edge labels between n and its neighbors (Fig. 1). Once each node has a vectorial representation, it can also be assigned an output $\boldsymbol{o}_n$, evaluated by another parametric function $g_{\boldsymbol{w}}$, called *output function*

$$\boldsymbol{o}_n = g_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{l}_n), \quad n \in \boldsymbol{N}. \quad (2)$$

Eqs. (1) and (2) define a method to produce an output $\boldsymbol{o}_n = \varphi_{\boldsymbol{w}}(\boldsymbol{G}, n)$ for each node of the graph $\boldsymbol{G}$. Moreover, the symbolic and subsymbolic information related to the nodes is indeed automatically encoded into a vector by the state transition function.

Given a graph $\boldsymbol{G}$, the computation can be graphically visualized substituting all of the nodes with "units" that compute the function $f_{\boldsymbol{w}}$ and are connected according to graph topology. The obtained network is called the *encoding network* (Fig. 2). In this way, the encoding network has the same skeleton as the input graph. Each "f-unit" is also connected to another unit that implements the output function $g_{\boldsymbol{w}}$. Since the same parametric functions are applied to all the nodes, the units of the same type share the same set of parameters.

[1]More precisely, in classification problems, the codomain of τ is $\mathbb{N}^m$, whereas in regression problems the codomain is $\mathbb{R}^m$.



$$\boldsymbol{x}_6 = f_{\boldsymbol{w}}(\boldsymbol{l}_6, \boldsymbol{x}_3, \boldsymbol{x}_4, \boldsymbol{x}_7, \boldsymbol{x}_8, \boldsymbol{x}_9, \boldsymbol{l}_3, \boldsymbol{l}_4, \boldsymbol{l}_7, \boldsymbol{l}_8, \boldsymbol{l}_9, \boldsymbol{l}_{(6,3)}, \boldsymbol{l}_{(6,4)}, \boldsymbol{l}_{(6,7)}, \boldsymbol{l}_{(6,8)}, \boldsymbol{l}_{(6,9)})$$

Fig. 1. The state $\boldsymbol{x}_n$ depends on the information in its neighborhood.

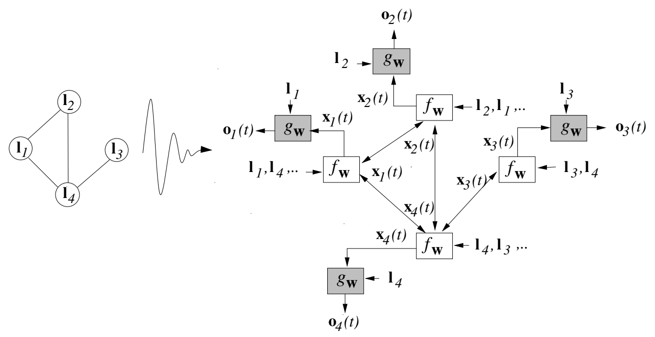


Fig. 2. Overview of the graph processing. Notice that the skeleton of the input graph defines the interconnection among the processing units.

Let $\boldsymbol{F}_{\boldsymbol{w}}$ and $\boldsymbol{G}_{\boldsymbol{w}}$ be the vectorial functions obtained stacking all the instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, respectively. Then Eqs. (1) and (2) can be rewritten as

$$\boldsymbol{x} = \boldsymbol{F}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \quad (3a)$$

$$\boldsymbol{o} = \boldsymbol{G}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \quad (3b)$$

where $\boldsymbol{l}$ represents the vector containing all the labels and $\boldsymbol{x}$ collects all the states. Equation (3a) defines the global state $\boldsymbol{x}$, while Eq. (3b) computes the output. It is worth to mention that Eq. (3a) is recursive w.r.t. the state $\boldsymbol{x}$, thus $\boldsymbol{x}$ is well defined only if Eq. (3a) has a unique solution. In conclusions, the viability of the method depends on the particular implementation of the transition function $f_{\boldsymbol{w}}$.

In our supervised framework, for a subset $\boldsymbol{S} \subseteq \boldsymbol{N}$ of nodes, called supervised nodes, a target value $\boldsymbol{t}_n$ is defined. Thus an error signal can be specified,

$$e_{\boldsymbol{w}} = \sum_{i \in \boldsymbol{S}} \big(\boldsymbol{t}_i - \varphi_{\boldsymbol{w}}(\boldsymbol{G}, i)\big)^2. \quad (4)$$

This signal drives an error backpropagation procedure that adapts the parameters of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ so that the function realized by the network approximates the targets, i.e. $\varphi_{\boldsymbol{w}}(\boldsymbol{G}, i) \approx \boldsymbol{t}_i, \ \forall i \in \boldsymbol{S}$.

In practice, Eq. (1) is well suited to process positional graphs, since each neighbor position can be associated to a specific input argument of function $f_{\boldsymbol{w}}$. In non–positional graphs, this scheme introduces an unnecessary constraint, since neighbors should be artificially ordered. A reasonable solution consists in calculating the state $\boldsymbol{x}_n$ as a sum of "contributions", one for each of its neighbors. Thus, state transition function can be rewritten as

$$\boldsymbol{x}_n = \sum_{i=1}^{|\mathrm{ne}[n]|} h_{\boldsymbol{w}}\left(\boldsymbol{l}_n, \boldsymbol{x}_{\mathrm{ne}_i[n]}, \boldsymbol{l}_{\mathrm{ne}_i[n]}, \boldsymbol{l}_{(n,\mathrm{ne}_i[n])}\right), \quad n \in \boldsymbol{N} \tag{5}$$

where $\mathrm{ne}_i[n]$ is the i-th neighbor and $|\mathrm{ne}[n]|$ is the number of neighbors of n.

Several possible implementations of the functions $f_{\boldsymbol{w}}$ (or $h_{\boldsymbol{w}}$) and $g_{\boldsymbol{w}}$ can be selected. RNNs and GNNs differ in the implementation of the state transition function $f_{\boldsymbol{w}}$ and in the class of graphs that can be processed.

A. *Recursive Neural Networks*

Recursive Neural Networks (RNNs) were originally proposed in [15], [16], [20]. In RNNs, the problem posed by the solution of Eq. (3) is faced enforcing the following two constraints:

- Input graphs are acyclic;
- The state $\boldsymbol{x}_n$ of node n depends only on its children, instead of its neighbours, i.e. Eq. (5) is replaced by

$$\boldsymbol{x}_n = \sum_{i=1}^{|\mathrm{ch}[n]|} h_{\boldsymbol{w}}\left(\boldsymbol{l}_n, \boldsymbol{x}_{\mathrm{ch}_i[n]}, \boldsymbol{l}_{\mathrm{ch}_i[n]}, \boldsymbol{l}_{(n,\mathrm{ch}_i[n])}\right) \tag{6}$$

In this way, the definition of $\boldsymbol{x}$ is not recursive and Eq. (3) has a unique solution. In fact, the state $\boldsymbol{x}$ can be computed as follows: for each leaf node n, $\boldsymbol{x}_n$ can be computed directly, assuming that $\boldsymbol{x}_{\mathrm{ch}[n]}$ is assigned a predefined value $\boldsymbol{x}_0$; for any other node, $\boldsymbol{x}_n$ is computed by Eq. (6) as soon the states of the children $\boldsymbol{x}_{\mathrm{ch}[n]}$ are available.

In graph classification problems, each graph must contains a node r, called the *supersource*, from which there is a path to all the other nodes[2]. The supersource is the only supervised node, i.e. a target $\boldsymbol{t}_r$ is assigned to it. Actually, the supersource is the best candidate to collect all the information embedded in the structure and to compute a suitable output.

The original RNN model can only process Directed Positional Acyclic Graphs (DPAGs). More recently, an extended model was proposed that can cope with non–positional acyclic graphs with labeled edges (i.e. DAGs) [25]. Such a model uses the transition function defined in Eq. (6), where function $h_{\boldsymbol{w}}$ is implemented by the composition of two different parametric functions. The former function $\phi_{\boldsymbol{w}} : \mathbb{R}^{(s+\boldsymbol{l}_E)} \rightarrow \mathbb{R}^s$ computes the sum $\overline{\boldsymbol{x}}_n$ of the neighbor contributions to the state (Eq. (7)), while the latter $\rho_{\boldsymbol{w}} : \mathbb{R}^{(s+\boldsymbol{l}_N)} \rightarrow \mathbb{R}^s$ combines this value with the node label $\boldsymbol{l}_n$ to obtain the state of the node (Eq. (8))

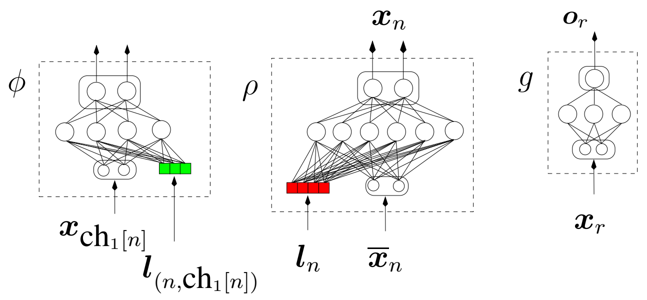
$$\overline{\boldsymbol{x}}_n = \sum_{i=1}^{|\mathrm{ch}[n]|} \phi_{\boldsymbol{w}}(\boldsymbol{x}_{\mathrm{ch}_i[n]}, \boldsymbol{l}_{(n,\mathrm{ch}_i[n])}) \tag{7}$$

$$\boldsymbol{x}_n = \rho_{\boldsymbol{w}}(\overline{\boldsymbol{x}}_n, \boldsymbol{l}_n) \tag{8}$$

As mentioned before, the output is evaluated only at the supersource r,

$$\boldsymbol{o}_r = g_{\boldsymbol{w}}(\boldsymbol{x}_r). \tag{9}$$

The parametric functions $\phi_{\boldsymbol{w}}$, $\rho_{\boldsymbol{w}}$, and $g_{\boldsymbol{w}}$ can be implemented by a variety of neural network models (Fig. 3). As shown in Fig. 4, the corresponding encoding network is a feedforward network. Thus, the output $\boldsymbol{o}_r$ can be calculated starting from the leaves and proceeding toward the supersource node r, whereas the error is backpropagated from r to the leaves of the encoding network.



ϕ
$\boldsymbol{x}_{\mathrm{ch}_1[n]}$
$\boldsymbol{l}_{(n,\mathrm{ch}_1[n])}$
ρ
$\boldsymbol{x}_n$
$\boldsymbol{l}_n$
$\overline{\boldsymbol{x}}_n$
g
$\boldsymbol{o}_r$
$\boldsymbol{x}_r$

Fig. 3. Functions $\phi_{\boldsymbol{w}}$, $\rho_{\boldsymbol{w}}$, and $g_{\boldsymbol{w}}$ may be implemented by a variety of neural network models.

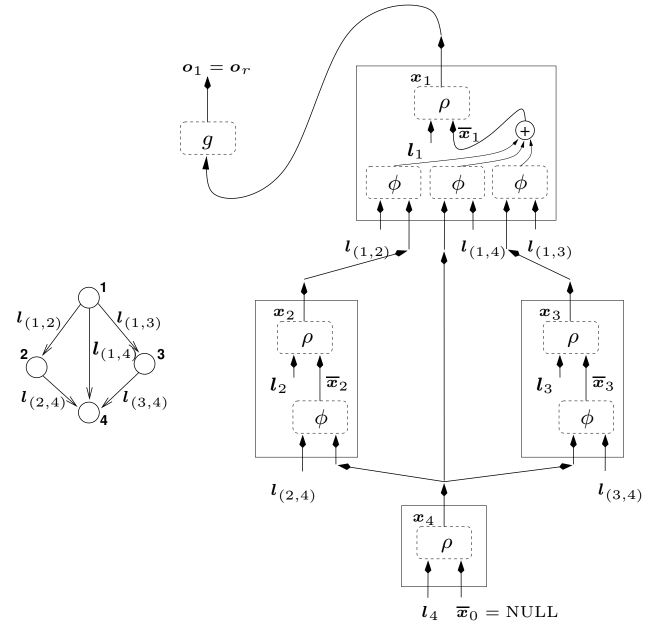


Fig. 4. An input graph (on the left) and its corresponding *encoding network*.

Finally, RNNs have been proved to be able to approximate in probability any function on positional trees [26] and positional and non–positional trees with labeled edges [25].

[2]If such a node does not exist it is always possible to add a dummy supersource node with the minimum number of edges [15].

In conclusions, the main limitations of RNNs w.r.t. the original framework are the restriction of the input domain to acyclic graphs and the dependence of a state $\boldsymbol{x}_n$ just on the children.

B. Graph Neural Networks

Graph Neural Networks (GNNs) have been recently proposed [17], [18] as an evolution of Recursive Neural Networks and have been conceived to fully implement the framework presented in Sect. III, in particular Eqs. (5) and (2). Thus, the input domain is the whole class of general graphs, including cyclic, acyclic, positional and non–positional, with labeled nodes and edges. In particular, this model can directly deal with undirected graphs.

Graph processing is performed associating to each graph $\boldsymbol{G}$ its encoding network, as shown in Fig. 2. As in RNNs, there could be many possible ways to implement the two functions $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$. In the following we assume to use three-layered feedforward neural network (i.e. with one hidden layer)[3], since they have been proved to be universal approximators [27], [28]. However, this is not a sufficient condition for GNNs to perform universal approximation on graphs.

As already remarked, in case of cyclic graphs the system described by Eqs. (3) is well defined only if state updating has a unique solution and this depends on the properties of the transition function $f_{\boldsymbol{w}}$. The proposed solution defines $f_{\boldsymbol{w}}$ such that the global function $\boldsymbol{F}_{\boldsymbol{w}}$ results to be a *contraction mapping* w.r.t. the state $\boldsymbol{x}$. A generic function $\rho : \mathbb{R}^n \to \mathbb{R}^n$ is a contraction mapping w.r.t. a vector norm $\|\cdot\|$, if it exists a real number μ, $0 \le \mu < 1$, such that for any $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^n$, $\|\rho(\boldsymbol{x}_1) - \rho(\boldsymbol{x}_2)\| \le \mu \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$. If $\boldsymbol{F}_{\boldsymbol{w}}$ is a contraction mapping, it is possible to prove that Eq. (3a) has a unique solution [29].

Since the function $f_{\boldsymbol{w}}$ is implemented using a feedforward neural network, this constraint can be satisfied limiting the range of values that weights $\boldsymbol{w}$ can assume. In fact it has been proved that if the norm of the jacobian matrix of a vectorial function is bounded by a number smaller than 1, then the function is a contraction mapping. In our implementation we used the one-norm, defined, for a generic matrix A, as $\|A\|_1 = max_j(\sum_i |A_{ij}|)$. Forcing the weights of the network to sufficiently small values surely satisfies the requirement. This approach, however, has one important drawback. If the weights are no more allowed to vary arbitrarily one can suspect that the global approximation property may be lost. For this reason it is preferable to add a penalty term to the error function,

$$e_{\boldsymbol{w}} = \sum_{i \in \boldsymbol{S}} \left(\boldsymbol{t}_i - \varphi_{\boldsymbol{w}}(\boldsymbol{G}, i)\right)^2 + \beta L\left(\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|_1\right), \qquad (10)$$

where the function L is specified as follows

$$L(y) = \begin{cases} (y-\mu)^2 & \text{if } y > \mu \\ 0 & \text{elsewhere} \end{cases}$$

[3]A different implementation of transition and output functions is presented in [18].

and μ is the desired contraction constant. The value β in Eq. (10) is an user definable parameter balancing the relative importance of the error on supervised patterns and the error w.r.t. the chosen contraction constant μ.

It should be clear that the greater is β, the harder is the constraint on the norm of the jacobian matrix of $\boldsymbol{F}_{\boldsymbol{w}}$. At the same time, no guarantee is given that the norm remains always strictly under μ. This is not a major problem, as actually the risk of instability of Eq. (3) is very low even if the norm significantly overcomes the value of 1. The reason is that the bound on the norm of the jacobian matrix gives only a sufficient condition for $\boldsymbol{F}_{\boldsymbol{w}}$ to be a contraction mapping.

1) The learning algorithm: The Banach fixed point theorem suggests a simple algorithm to compute the fixed point of Eq. (3a). It states that if ρ is a generic contraction mapping, then the dynamical system $\boldsymbol{x}(t+1) = \rho(\boldsymbol{x}(t))$, where $\boldsymbol{x}(t)$ denotes the t-th iterate of $\boldsymbol{x}$, converges exponentially fast to the solution of the equation $\boldsymbol{x} = \rho(\boldsymbol{x})$ for any initial state $\boldsymbol{x}(0)$. Thus $\boldsymbol{x}_n$ can be obtained iteratively as

$$\boldsymbol{x}_n(t+1) = f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{x}_{\mathrm{ne}[n]}(t), \boldsymbol{l}_{\mathrm{ne}[n]}, \boldsymbol{l}_{(n,\mathrm{ne}[n])}).$$

The simultaneous and repeated activation of the "f-units" in the encoding network (Fig. 2) produces the behavior described by the previous equation. Notice that there is no learning in this phase. Simply, the state vector $\boldsymbol{x}$ is evolved until it reaches its fixed point $\boldsymbol{x}^*$. Then the "g-units" evaluate all the outputs $\boldsymbol{o}_n$, that can be compared with targets $\boldsymbol{t}_n$ to derive an error signal as in Eq. (10).

The goal of the learning procedure is to adapt the parameters of the transition and the output functions for a better approximation of the targets. This can be accomplished using a gradient descent strategy, as in canonical backpropagation algorithm. Thus, for each parameter w_k, the quantity $\frac{\partial e_{\boldsymbol{w}}}{\partial w_k}$ needs to be evaluated. As $g_{\boldsymbol{w}}$ is a standard feedforward neural network, the gradient of the error w.r.t. each component of its weight vector $\boldsymbol{w}$ can be easily obtained, together with the gradient $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{x}^*}$ that expresses the sensibility of the overall error to a perturbation of the fixed point of the state vector $\boldsymbol{x}$. We can suppose that the system has reached its fixed point $\boldsymbol{x}^*$ at time T, i.e. $\boldsymbol{x}^* \simeq \boldsymbol{x}(T)$. The gradient of the error w.r.t. the parameters of function $f_{\boldsymbol{w}}$ can be obtained as

$$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{x}(T)} \frac{\partial \boldsymbol{x}(T)}{\partial \boldsymbol{w}}. \qquad (11)$$

Unfortunately, $\boldsymbol{x}(T)$ has an implicit dependence on the parameters $\boldsymbol{w}$, so the second factor of the right part of Eq. (11) cannot be evaluated directly. Instead, the error should be backpropagated from one time instant to the previous, accumulating the gradient until convergence. In fact $\boldsymbol{x}(T)$ depends on $\boldsymbol{w}$ *directly*, but also through the state $\boldsymbol{x}$ at previous time instants. Remembering that

$$\boldsymbol{x}(T) = \boldsymbol{F}_w(\boldsymbol{x}(T-1), \boldsymbol{l})$$

and using the chain rule, we have the following recursive equation

$$\frac{\partial \boldsymbol{x}(T)}{\partial \boldsymbol{w}} = \left.\frac{\partial F_{\boldsymbol{\psi}}(\boldsymbol{x}(T-1),\boldsymbol{l})}{\partial \psi}\right|_{\boldsymbol{\psi}=\boldsymbol{w}} + + \frac{\partial F_{w}(\boldsymbol{x}(T-1),\boldsymbol{l})}{\partial \boldsymbol{x}(T-1)} \frac{\partial \boldsymbol{x}(T-1)}{\partial \boldsymbol{w}} \qquad (12)$$

where term $\frac{\partial F_{\boldsymbol{w}}(\boldsymbol{x}(T-1),\boldsymbol{l})}{\partial \boldsymbol{x}(T-1)}$ is simply the jacobian matrix of $F_{\boldsymbol{w}}$ w.r.t. its second argument, evaluated at the point $\boldsymbol{x}(T-1)$. The notation in Eq. (12) is not unnecessarily burdensome. In fact the symbol $\frac{\partial F_{\boldsymbol{w}}(\boldsymbol{x}(T-1),\boldsymbol{l})}{\partial \boldsymbol{w}}$ is ambiguous in this context as it means the overall sensitivity of the steady state $\boldsymbol{x}(T)$ to a small variation of $\boldsymbol{w}$, but also the partial derivative of the function $F_{\boldsymbol{w}}$ w.r.t. to its first argument $\boldsymbol{w}$, regardless of the fact that *even the second argument* $\boldsymbol{x}(T-1)$ depends on $\boldsymbol{w}$. For this reason, it is better to resort to a more formal notation to ensure clarity. Thus the symbol $\frac{\partial F_{\boldsymbol{w}}(\boldsymbol{x}(T-1),\boldsymbol{l})}{\partial \boldsymbol{w}}$ has the former meaning while we introduce a formal parameter $\boldsymbol{\psi}$ when we need the partial derivative only w.r.t. to the first argument of $F_{\boldsymbol{w}}$.

Suppose now that in the forward phase the state evolved for a long time after convergence. Obviously

$$\boldsymbol{x}(T+\bar{t}\,) \simeq \boldsymbol{x}(T) \qquad \forall \bar{t} > 0$$

Gradient accumulation needs, in principle, to evaluate Eq. (12) backward until time $t = -\infty$. From a practical point of view, however, the series of the gradient values is convergent, as the generic term results from the multiplication of increasingly more jacobian matrices. If all these matrices have norm smaller than one, gradient accumulation can be stopped after some steps backward, as contributions become progressively smaller. The conclusion is that gradient computation is indeed feasible, given that the condition on the jacobian matrix norm is satisfied. Thus we can stop gradient accumulation at time instant t_0, that we can safely assume to be posterior to convergence time T, i.e. $t_0 > T$. This allows to consider only the terms at time instant T,

$$J_{\boldsymbol{w}} \doteq J_{\boldsymbol{w}}(T) = \left.\frac{\partial F_{\boldsymbol{\psi}}(\boldsymbol{x}(T),\boldsymbol{l})}{\partial \psi}\right|_{\boldsymbol{\psi}=\boldsymbol{w}}$$
$$J_{\boldsymbol{x}} \doteq J_{\boldsymbol{x}}(T) = \frac{\partial F_{w}(\boldsymbol{x}(T),\boldsymbol{l})}{\partial \boldsymbol{x}(T)}$$

and Eq. (12) becomes

$$\frac{\partial \boldsymbol{x}(T)}{\partial \boldsymbol{w}} = J_{\boldsymbol{w}}\big\{1 + J_{\boldsymbol{x}} + (J_{\boldsymbol{x}})^2 + (J_{\boldsymbol{x}})^3 + \ldots + (J_{\boldsymbol{x}})^{t_0}\big\}.$$

This intuition is very useful also to contain memory requirements, as there is no need to keep memory of the gradient values calculated at each time instant before state convergence. Gradient evaluation is represented schematically in Fig. 5.

The proposed algorithm is not completely new. In fact it combines the backpropagation through structure algorithm, adopted to train Recursive Neural Networks, and the Almeida–Pineda algorithm [30]–[32] (also called *Recurrent Backpropagation*). The latter is a particular version of the backpropagation through time algorithm which can be used

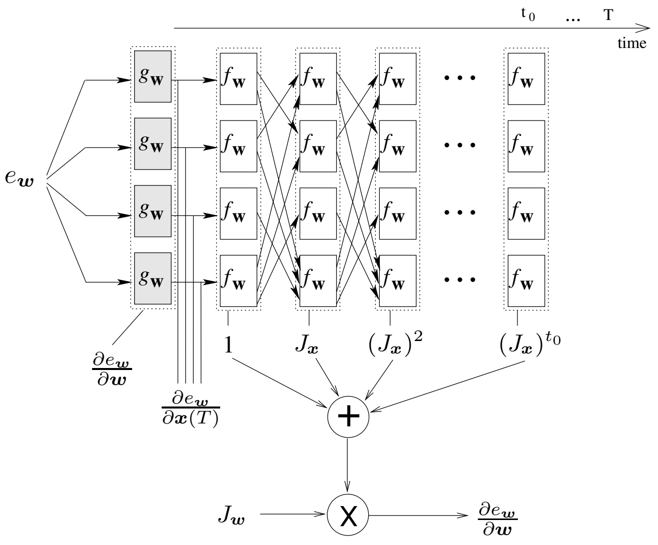

t0 ... T
time
$e_{\boldsymbol{w}}$
$g_{\mathbf{w}}$
$f_{\mathbf{w}}$
$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$
$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{x}(T)}$
1
$J_{\boldsymbol{x}}$
$(J_{\boldsymbol{x}})^2$
$(J_{\boldsymbol{x}})^{t_0}$
+
$J_{\boldsymbol{w}}$
X
$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$

Fig. 5. Overall gradient is evaluated accumulating gradient contributions backward until time instant t_0.

to train recurrent networks. In the Almeida–Pineda algorithm, input is kept constant and the network settles down to a fixed point for its dynamics. The aim is to move this fixed state toward a given target, backpropagating the error and accumulating the gradient exactly in the way seen before for GNNs. Moreover, as explained clearly in [32], Recurrent Backpropagation doesn't require the storage of any state history, assuming, as seen before, that the state equilibrium point was reached long before starting gradient backpropagation. In conclusion, our approach applies the Almeida–Pineda algorithm to the encoding network, where all instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ are considered to be independent networks, but, as usual in Recursive Neural Networks, contributions for shared weights are combined. When the gradient is available, the weights can be updated using resilient backpropagation [33], a well-known variant of standard backpropagation that speeds up the convergence.

2) Some remarks on generalization capabilities: In this section we will report some results concerning the generalization capabilities of the GNN model. More details can be found in [34].

Given a graph $\boldsymbol{G}$, the *unfolding tree of depth* d $\boldsymbol{T}_n^d$ of one of its nodes n is the graph obtained "unfolding" $\boldsymbol{G}$ up to the depth d, using n as the starting point (Fig. 6).

Two nodes n, u are said to be *unfolding equivalent* $n \sim u$, if $\boldsymbol{T}_n^d = \boldsymbol{T}_u^d$ for every $d \geq 1$. In Fig. 6 the two nodes with label $\boldsymbol{b}$ are unfolding equivalent, while the two nodes with label $\boldsymbol{a}$ are not equivalent, as their trees already differ in the very first levels.

A function $p : \mathcal{D} \rightarrow \mathbb{R}^m$ is said to preserve the unfolding equivalence on $\mathcal{D}$, if $n \sim u$ implies $p(\boldsymbol{G}, n) = p(\boldsymbol{G}, u)$, for any patterns $(\boldsymbol{G}, n)$, $(\boldsymbol{G}, u)$ that belong to $\mathcal{D}$.
The nodes of the two graphs on the left of Fig. 7 share the same unfolding tree. If a function assigns different values to them, f.i. the number of nodes of the graph to which they belong, it doesn't preserve the unfolding equivalence.

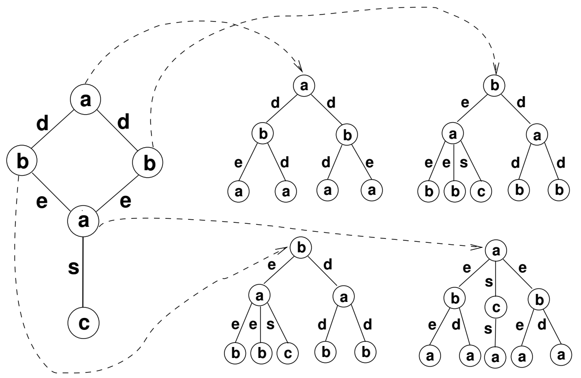


Fig. 6. A graph and the unfolding trees of depth 2 of its nodes.

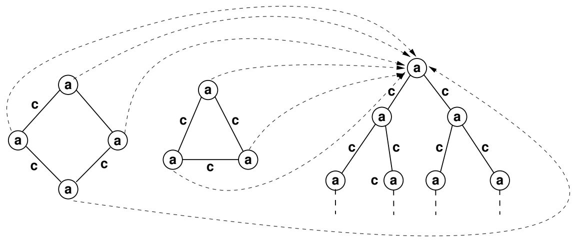


Fig. 7. The nodes of the two graphs on the left are all unfolding equivalent nodes.

Since GNNs adopt an inherently local computational scheme, they are able to approximate only functions that preserve unfolding equivalence. Noteworthy, also the converse is true. In fact it is possible to show that, given a function that preserves the unfolding equivalence, it always exists a GNN model capable of approximating it in probability to the desired degree of accuracy [34].

IV. Experimental results

RNNs and GNNs have been experimentally compared on an image classification problem which was based on images extracted from the Caltech benchmark database [35]. In the experiments, four classes have been considered, containing images of 'bottles", "camels", "guitars" and "houses". Two of the four categories, namely bottles and camels, are particularly difficult to be recognized, since they were originally collected using Google image search, without any kind of normalization.

For each class, a set of 350 random images were extracted from the original dataset. A half of those images consisted of positive examples of the category, the other half consisted of negative examples, i.e. images belonging to the other categories. Then, the set was subdivided into a training set, a validation set[4] and a test set containing 150, 50 and 150 images, respectively.

A. From images to DAGs

Each image was preprocessed in order to obtain the Region Adjacency Graph (RAG) that represents it. A RAG is a graph where nodes denote homogeneous regions of the image and edges stand for the adjacency relationships. Nodes and edges are labeled. Node labels contain geometric features of the corresponding regions (area, perimeter, orientation of the principal axes, and so on), while edge labels encode the mutual orientation of the two regions and the difference between their average colors.

RAGs were obtained by the following steps:

- The image was filtered using the Mean Shift algorithm [36][5].
- A k–means [37] color quantization procedure was performed to locate an initial big number of homogeneous regions.
- Some adjacent regions were merged to achieve the desired number of final regions. The procedure evaluated a dissimilarity function between all the pairs of regions and merged those that achieve the lowest values, updating the list at each fusion. The dissimilarity function has been chosen heuristically considering the distance between the average colors of the two regions and their dimension.

The RAGs obtained by the above procedure constituted the dataset used for the experiments with the GNNs. On the other hand, since RAGs are undirected graphs, RNNs are not able to process them. As in other applications of RNNs to image processing [24], [38], the RAGs were transformed into Directed Acyclic Graphs (DAGs). The transformation was accomplished by performing a breadth-first visit of the RAG, starting from the region which contained the central pixel of the image. Each time an edge was traversed in a certain direction, the undirected edge was replaced by an arc with that direction (Fig 8).

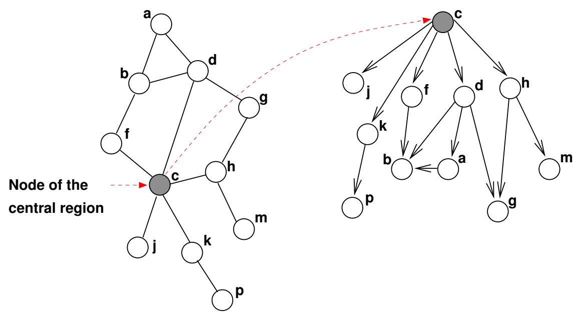


Fig. 8. The original RAG (on the left) and the corresponding DAG.

B. Network architectures and results

The experimentation consisted of a set of runs. Each experiment varies according whether a GNN or a RNN is trained, according to the category that is considered and to the network architecture that was used. In fact, every experiment concerns a category in 'bottles", "camels", "guitars"

[4]The validation set was exploited to choose the optimal network. In fact, the optimal network was the one that, during the training procedure, achieved the lowest error on the validation set.

[5]The main advantage of this technique is that the method was especially designed to preserve gradient discontinuities. Thus smallest textures, that are often over segmented by techniques that employ only color information, were heavily smoothed while significant edges remained almost crisp, easing region boundaries detection.

TABLE I

THE NUMBER OF HIDDEN UNITS OF THE RNN AND GNN FUNCTIONS, W.R.T. THE THREE CONFIGURATIONS *small*, *medium* AND *big*

Model	**Function**	**Configuration**		
		small	*medium*	*big*
RNN	ϕ	10 hidden u.	11 hidden u.	12 hidden u.
	ρ	5 hidden u.	7 hidden u.	10 hidden u.
	g	5 hidden u.	7 hidden u.	10 hidden u.
GNN	f	5 hidden u.	7 hidden u.	10 hidden u.
	g	5 hidden u.	7 hidden u.	10 hidden u.

TABLE II

COMPARISON BETWEEN THE ACCURACIES OBTAINED BY RECURSIVE NEURAL NETWORKS (RNN) AND GRAPH NEURAL NETWORKS (GNN) IN THE IMAGE CLASSIFICATION PROBLEM

Class	**Model**	**Configuration**		
		small	*medium*	*big*
Bottles	RNN	69.33%	70.66%	68.66%
	GNN	**76.67%**	**83.33%**	**84.67%**
Camels	RNN	62.67%	62.67%	65.33%
	GNN	**70.00%**	**73.33%**	**74.67%**
Guitars	RNN	61.33%	62,67%	60.00%
	GNN	**70.67%**	**70.00%**	**67.33%**
Houses	RNN	80.67%	81.33%	79.33%
	GNN	**84.00%**	**83.33%**	**84.67%**

and "houses" and the network is trained to decide whether the input graph belongs or not to the considered category.

Three different configurations, with an increasing number of weights, were experimented for each model. In order to make the comparison as fair as possible, the configurations were chosen so that GNNs and RNNs have the same number of weights (Tab. I). The three configurations will be denoted as *small*, *medium* and *big*.

Table II shows the accuracies achieved in the experiments. Here, the accuracy is defined as the percentage of correct predictions w.r.t. the total number of predictions. Table III shows the same results, taking also into account the different accuracies in the positive and negative examples. Finally, Table IV displays the *equal error rate* of the two models. The *equal error rate* is obtained plotting the accuracies on positive and negative examples of the test set varying the classification threshold (Fig. 9). The equal error rate is defined as the accuracy of the system when the number of errors in the two classes is equal.

The results show that GNNs almost always outperformed RNNs. Even if the two models are conceptually very similar, Graph Neural Networks have the advantage to process RAGs directly. In the transformation from RAGs to DAGs, the edges, which are naturally undirected, are replaced with directed ones, probably causing a loss of information. Moreover, the direction of the edges affects also the order of the computation which in RNNs must go from the leaves of the DAGs to the supersource. Those two factors have probably determined the better results obtained by GNNs.

TABLE III

COMPARISON BETWEEN THE ACCURACIES OBTAINED BY THE MODELS W.R.T. NEGATIVE AND POSITIVE EXAMPLES. THE BEST CLASS ACCURACIES ARE SHOWN IN BOLD

Class	**Model**	**Subclass**	**Configuration**		
			small	*medium*	*big*
Bottles	RNN	*positive*	69.33%	**81.33%**	68.00%
		negative	69.33%	60.00%	69.33%
	GNN	*positive*	**80.00%**	69.33%	**73.33%**
		negative	**73.33%**	**78.67%**	**80.00%**
Camels	RNN	*positive*	**82.67%**	**84.00%**	**92.00%**
		negative	42.67%	41.33%	38.66%
	GNN	*positive*	73.33%	80.00%	81.33%
		negative	**66.67%**	**66.67%**	**68.00%**
Guitars	RNN	*positive*	**82.66%**	**81.33%**	**74.67%**
		negative	44.00%	44.00%	45.33%
	GNN	*positive*	62.67%	61.33%	60.00%
		negative	**78.67%**	**78.67%**	**74.67%**
Houses	RNN	*positive*	76.67%	85.33%	**85.33%**
		negative	**84.67%**	77.33%	73.33%
	GNN	*positive*	**86.67%**	85.33%	82.67%
		negative	81.33%	**81.33%**	**86.67%**

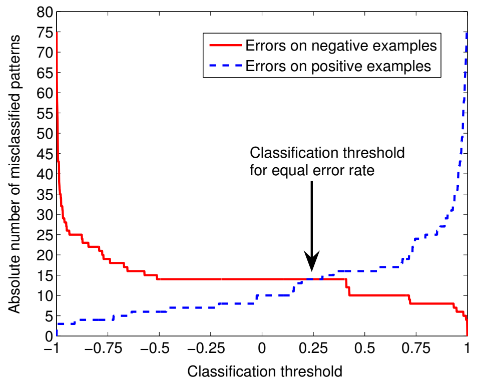


Fig. 9. Typical curves describing the absolute number of errors on the test set varying the classification threshold. Test set had 75 positive and 75 negative examples.

TABLE IV

COMPARISON BETWEEN THE EQUAL ERROR RATES OBTAINED BY RECURSIVE NEURAL NETWORKS (RNN) AND GNNS IN THE IMAGE CLASSIFICATION PROBLEM

Class	**Model**	**Configuration**		
		small	*medium*	*big*
Bottles	RNN	69.33%	68.00%	68.66%
	GNN	**77.33%**	**77.33%**	**78.67%**
Camels	RNN	58.67%	58.67%	58.67%
	GNN	**69.33%**	**76.00%**	**73.33%**
Guitars	RNN	58.67%	62.67%	60.00%
	GNN	**68.00%**	**70.67%**	**69.33%**
Houses	RNN	78.67%	78.67%	78.67%
	GNN	**81.33%**	**85.33%**	**82.67%**

On the other hand, notice that GNNs require a longer training time due to need of iterating the computation of the states and of the gradients until a stable point is achieved.

V. Conclusions

This paper presented a comparison between two neural network models, RNNs and GNNs, capable of directly processing graphs with labeled nodes and edges. Their main differences and similarities w.r.t. input domains, approximation capabilities and learning algorithms have been discussed. A real–world image classification task showed that GNNs outperforms RNNs, both in terms of accuracies and equal error rates.

References

[1] P. Baldi and G. Pollastri, "The principled design of large-scale recursive neural network architectures-dag-rnns and the protein structure prediction problem," *Journal of Machine Learning Research*, vol. 4, pp. 575–602, 2003.

[2] E. Francesconi, P. Frasconi, M. Gori, S. Marinai, J. Sheng, G. Soda, and A. Sperduti, "Logo recognition by recursive neural networks," in *GREC '97: Selected Papers from the Second International Workshop on Graphics Recognition, Algorithms and Systems*, K. Tombre and A. K. Chhabra, Eds. Springer-Verlag, 1998, pp. 104–117.

[3] E. Krahmer, S. Erk, and A. Verleg, "Graph-based generation of referring expressions," *Computational Linguistics*, vol. 29, no. 1, pp. 53–72, 2003.

[4] F. Costa, P. Frasconi, V. Lombardo, and G. Soda, "Towards incremental parsing of natural language using recursive neural networks," *Applied Intelligence*, vol. 19, no. 1–2, p. 9, July 2003.

[5] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity for indexing paths in graph-structured data," *Proc. of the 18th International Conference on Data Engineering (ICDE'02)*, pp. 129–140, 2002.

[6] M. E. J. Newman, "From the cover: The structure of scientific collaboration networks," *Proc. National Academy of Sciences*, pp. 404–409, 2001.

[7] K. Mehlhorn, *Graph algorithms and NP-completeness*. Berlin New York Springer, 1984.

[8] H. Bunke, "Graph matching: Theoretical foundations, algorithms and applications," in *Proceedings of Vision Interface 2000*, 2000, pp. 82–88.

[9] W. Christmas, J. Kittler, and M. Petrou, "Structural matching in computer vision using probabilistic relaxation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 749–764, 1995.

[10] B. Hammer and J. Jain, "Neural methods for non-standard data," in *Proceedings of the 12th European Symposium on Artificial Neural Networks*, M.Verleysen, Ed., 2004, pp. 281–292.

[11] R. Kondor and J. Lafferty, "Diffusion kernels on graphs and other discrete structures," in *Proc. 19th International Conference on Machine Learning (ICML2002)*, C. Sammut and A. e. Hoffmann, Eds. Morgan Kaufmann Publishers Inc, 2002, pp. 315–322.

[12] T. Gärtner, "Kernel-based learning in multi-relational data mining," *ACM SIGKDD Explorations*, vol. 5, no. 1, pp. 49–58, 2003.

[13] P. Mahé, N. Ueda, T. Akutsu, P. J.-L., and J.-P. Vert, "Extensions of marginalized graph kernels," in *Proc. 21th International Conference on Machine Learning (ICML2004)*. ACM Press, 2004, p. 70.

[14] J. Suzuki, H. Isozaki, and E. Maeda, "Convolution kernels with feature selection for natural language processing tasks." in *ACL*, 2004, pp. 119–126.

[15] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Transactions on Neural Networks*, vol. 8, pp. 429–459, 1997.

[16] P. Frasconi, M. Gori, and A. Sperduti, "A general framework for adaptive processing of data structures," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 768–786, 1998.

[17] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. International Joint Conference on Neural Networks (IJCNN2005)*, 2005, pp. 729–734.

[18] F. Scarselli, S. Yong, M. Gori, M. Hagenbuchner, A. Tsoi, and M. Maggini, "Graph neural networks for ranking web pages," in *Proc. of the 2005 IEEE/WIC/ACM Conference on Web Intelligence (WI2005)*, 2005, pp. 666–672.

[19] M. Hagenbuchner, A. Sperduti, and A. C. Tsoi, "A self-organizing map for adaptive processing of structured data," *IEEE Transactions on Neural Networks*, vol. 14, no. 3, pp. 491–505, May 2003.

[20] A. Küchler and C. Goller, "Inductive learning in symbolic domains using structure–driven recurrent neural networks," in *Advances in Artificial Intelligence*, G. Görz and S. Hölldobler, Eds. Berlin: Springer-Verlag, 1996, pp. 183–197.

[21] T. Schmitt and C. Goller, "Relating chemical structure to activity: An application of the neural folding architecture," in *Proc. 5th Int. Workshop Fuzzy-Neuro-Systems '98 (FNS'98)*, 1998.

[22] M. Hagenbuchner and A. Tsoi, "Recursive cascade correlation and recursive multilayer perceptron, a comparison," *IEEE Transactions on Neural Networks*, 2002 (Submitted).

[23] M. Gori, M. Maggini, E. Martinelli, and F. Scarselli, "Learning user profiles in NAUTILUS," in *AH '00: Proceedings of the International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. Springer-Verlag, 2000, pp. 323–326.

[24] M. Bianchini, P. Mazzoni, L. Sarti, and F. Scarselli, "Face localization with recursive neural networks," in *Proceedings of WIRN03*, ser. Lecture Notes in Computer Science, B. Apolloni, M. Marinaro, and R. Tagliaferri, Eds., vol. 2859. Springer-Verlag, 2003, pp. 99–105.

[25] M. Bianchini, M. Maggini, L. Sarti, and F. Scarselli, "Recursive neural networks for processing graphs with labelled edges: Theory and applications," *Neural Networks - Special Issue on Neural Networks and Kernel Methods for Structured Domains*, vol. 18, pp. 1040–1050, October 2005.

[26] B. Hammer, "Approximation capabilities of folding networks," in *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, M. Caudill and C. Butler, Eds., 1999, pp. 33–38.

[27] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, vol. 3, pp. 303–314, 1989.

[28] F. Scarselli and A. C. Tsoi, "Universal approximation using feedforward neural networks: a survey of some existing methods, and some new results," *Neural Networks*, pp. 15–37, 1998.

[29] M. A. Khamsi, *An Introduction to Metric Spaces and Fixed Point Theory*. John Wiley & Sons Inc, 2001.

[30] L. Almeida, "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment," in *Proceedings of the IEEE International Conference on Neural Networks*, M. Caudill and C. Butler, Eds., vol. 2. San Diego, 1987: IEEE, New York, 1987, pp. 609–618.

[31] F. Pineda, "Generalization of back–propagation to recurrent neural networks," *Physical Review Letters*, vol. 59, pp. 2229–2232, 1987.

[32] R. William and D. Zipser, *Back-propagation: Theory Architectures and Applications*. Lawrence Erlbaum Associates, Inc.: Lawrence Erlbaum Associates, Inc., 1995, ch. Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity, pp. 433–486.

[33] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the rprop algorithm," in *Proceedings of the IEEE International Conference on Neural Networks*, vol. 1, San Francisco, CA, USA, 1993, pp. 586–591.

[34] F. Scarselli, M. Gori, G. Monfardini, A. C. Tsoi, and M. Hagenbuchner, "A new neural network model for graph processing," Department of Information Engineering, University of Siena, Tech. Rep. DII 01/05, 2005.

[35] R. Fergus, P. Perona, and A. Zisserman, "A sparse object category model for efficient learning and exhaustive recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005, pp. 380–387.

[36] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002.

[37] J. A. Hartigan and M. A. Wong, "A k–means clustering algorithm," *Applied Statistics*, vol. 28, pp. 100–108, 1979.

[38] M. Bianchini, P. Mazzoni, L. Sarti, and F. Scarselli, "Face spotting in color images using recursive neural networks," in *Proceedings of the 1st ANNPR Workshop*, Florence (Italy), 2003.