A New Model for Learning in Graph Domains

Marco Gori
Dipartimento di Ingegneria
dell'Informazione
Università di Siena, Italy
E-mail: marco@dii.unisi.it

Gabriele Monfardini
Dipartimento di Ingegneria
dell'Informazione
Università di Siena, Italy
E-mail: monfardini@dii.unisi.it

Franco Scarselli
Dipartimento di Ingegneria
dell'Informazione
Università di Siena, Italy
E-mail: franco@dii.unisi.it

***Abstract*— In several applications the information is naturally represented by graphs. Traditional approaches cope with graphical data structures using a preprocessing phase which transforms the graphs into a set of flat vectors. However, in this way, important topological information may be lost and the achieved results may heavily depend on the preprocessing stage. This paper presents a new neural model, called graph neural network (GNN), capable of directly processing graphs. GNNs extends recursive neural networks and can be applied on most of the practically useful kinds of graphs, including directed, undirected, labelled and cyclic graphs. A learning algorithm for GNNs is proposed and some experiments are discussed which assess the properties of the model.**

I. Introduction

In several machine learning applications the data of interest can be suitably represented in form of sequences, trees, and, more generally, directed or undirected graphs, f.i. in chemics [1], software engineering, image processing [2]. In those applications, the goal consists of learning from examples a function τ that maps a graph $\boldsymbol{G}$ and one of its nodes n to a vector of reals: $\tau(\boldsymbol{G}, n) \in \mathbb{R}^m$.

More precisely, we can distinguish two classes of applications according whether $\tau(\boldsymbol{G}, n)$ depends or not on the node n. Those applications will be called *node focused* and *graph focused*, respectively. Object localization is an example of node focused applications. An image can be represented by a Region Adjacency Graph (RAG), where the nodes denote the homogeneous regions of the image and the arcs represent their adjacency relationship (Fig. 1). This problem can be solved by a function τ which classifies the nodes of the RAG according to whether the corresponding region belongs to the object or not. For example, the output of τ for Fig. 1 might be 1 for the black nodes, which correspond to the house, and -1 otherwise. On the other hand, image classification is an example of graph focused applications. For instance, $\tau(\boldsymbol{G})$ may classify an image represented by $\boldsymbol{G}$ into different classes, e.g., houses, cars, people, and so on.

Traditional applications usually cope with graphs by a preprocessing procedure that transforms the graphs to simpler representations (e.g. vectors or sequences of reals) which can be successfully elaborated with common machine learning techniques. However, valuable information may be lost during the preprocessing and, as a consequence, the application may suffer from a poor performance and generalization.

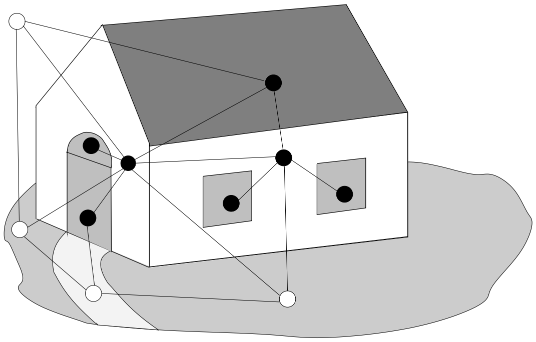


Fig. 1. An image and its graphical representation by a RAG.

Recursive neural networks (RNNs) [3], [4] are a new neural model that tries to overcome this problem. In fact, RNNs can directly process graphs. The main idea consists of encoding the graphical information into a set of states associated with the graph nodes. The states are dynamically updated following the topological relationship among the nodes. Finally, an output is computed using the encodings stored in the states. However, the RNN model suffers from a number of limitations. In fact, RNNs can process only directed and acyclic graphs and can be used only on graph focused problems, i.e. $\tau(\boldsymbol{G}, n)$ must be independent from n.

In this paper, we present a new neural network model, called graph neural network (GNN), that extends recursive neural networks. GNNs can process most of the practically useful graphs and can be applied both on graph and node focused problems. A learning algorithm for GNNs is also described along with some experimental results that assess the properties of model. Finally, it is worth to mention that, under mild conditions, any function τ on graphs can be approximated in probability by a GNN. Such a result, which, for reasons of space, is not further discussed in this paper, is proved in [5].

The structure of the paper is as follows: Section II presents the GNN model along with its main properties. Section III contains some experimental results. Finally, in Section IV conclusions are drawn.

II. Graph neural networks

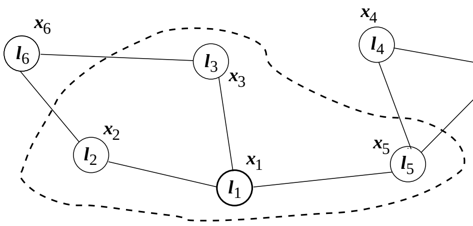
In the following, $|\cdot|$ represents the module or the cardinality operator according to whether it is applied on a real number or a set, respectively. The norm one of vector $\boldsymbol{v}$ is denoted by $\|\boldsymbol{v}\|_1$, i.e. $\|\boldsymbol{v}\|_1 = \sum_i |v_i|$. A graph $\boldsymbol{G}$ is a pair $(\boldsymbol{N}, \boldsymbol{E})$, where $\boldsymbol{N}$ is a set of nodes and $\boldsymbol{E}$ a set of edges. The nodes connected to n by an arc are represented by $\text{ne}[n]$. Each node may have a label that is denoted by $\boldsymbol{l}_n \in \mathbb{R}^q$. Usually labels include features of the object corresponding to the node. For example, in the case of a RAG (Figure 1), node labels may represent properties of the regions, e.g., area, perimeter.

The considered graphs may be either positional or non–positional. Non–positional graphs are those described so far. In positional graphs, an injective function $\mu_n : \text{ne}[n] \rightarrow \mathbb{N}$ is defined for each node n. Here, $\mathbb{N}$ is the set of natural numbers, and μ_n assigns to each neighbor $u \in \text{ne}[n]$ a different position. Actually, the position can be used to store useful information, e.g. a sorting of the neighbors according to their importance.

The intuitive idea underlying GNNs is that nodes in a graph represent objects or concepts and edges represent their relationships. Thus, we can attach to each node n a vector $\boldsymbol{x}_n \in \mathbb{R}^s$, called *state*, which collects a representation of the object denoted by n. In order to define $\boldsymbol{x}_n$, we observe that the related nodes are connected by edges. Thus, $\boldsymbol{x}_n$ is naturally specified using the information contained in the neighborhood of n (see Figure 2). More precisely, let $\boldsymbol{w}$ be a set of parameters and $f_{\boldsymbol{w}}$ be a parametric *transition function* that expresses the dependence of a node on its neighborhood. The state $\boldsymbol{x}_n$ is defined as the solution of the system of equations:

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

where $\boldsymbol{l}_n$, $\boldsymbol{x}_{\text{ne}[n]}$, $\boldsymbol{l}_{\text{ne}[n]}$ are the label of n, and the states and the labels of the nodes in the neighborhood of n, respectively.



$$x_1 = f_{\rm w}(l_1, \underbrace{x_2, x_3, x_5}_{x_{ne[1]}}, \underbrace{l_2, l_3, l_5}_{l_{ne[1]}})$$

Fig. 2. State $\boldsymbol{x}_1$ depends on the neighborhood information.

For each node n, an output vector $\boldsymbol{o}_n \in \mathbb{R}^m$ is also defined which depends on the state $\boldsymbol{x}_n$ and the label $\boldsymbol{l}_n$. The dependence is described by a parametric *output function* $g_{\boldsymbol{w}}$

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

Let $\boldsymbol{x}$ and $\boldsymbol{l}$ be respectively the vectors constructed by stacking all the states and all the labels. Then, Equations (1) and (2) can be written as:

$$\begin{array}{rcl} \boldsymbol{x} & = & F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \\ \boldsymbol{o} & = & G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \end{array} \tag{3}$$

where $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ are the composition of $|\boldsymbol{N}|$ instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, respectively.

Notice that $\boldsymbol{x}$ is correctly defined only if the solution of system (3) is unique. The key choice adopted in the proposed approach consists of designing $f_{\boldsymbol{w}}$ such that $F_{\boldsymbol{w}}$ is a contraction mapping[1] w.r.t. the state $\boldsymbol{x}$. In fact, the Banach fixed point theorem [6] guarantees that if $F_{\boldsymbol{w}}$ is a contraction mapping, then Eq. (3) has a solution and the solution is unique.

Thus, Eqs. (1) and (2) define a method to produce an output $\boldsymbol{o}_n$ for each node, i.e. they realize a parametric function $\varphi_{\boldsymbol{w}}(\boldsymbol{G}, n) = \boldsymbol{o}_n$ which operates on graphs. The corresponding machine learning problem consists of adapting the parameters $\boldsymbol{w}$ such that $\varphi_{\boldsymbol{w}}$ approximates the data in the learning set $\mathcal{L} = \{(\boldsymbol{G}_i, n_i, \boldsymbol{t}_i) |\, 1 \leq i \leq p\}$, where each triple $(\boldsymbol{G}_i, n_i, \boldsymbol{t}_i)$ denotes a graph $\boldsymbol{G}_i$, one of its nodes n_i and the desired output $\boldsymbol{t}_i$. In practice, the learning problem can be implemented by the minimization of a quadratic error function

$$e_{\boldsymbol{w}} = \sum_{i=1}^{p} (\boldsymbol{t}_i - \varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_i))^2 \,. \tag{4}$$

Notice that system (1) is particularly suited to process positional graphs, since the label and the state of each neighbor is assigned to a predefined position in the inputs of $f_{\boldsymbol{w}}$. For non–positional graphs, it may be useful to replace Eq. (1) with

$$\boldsymbol{x}_n = \sum_{u \in \text{ne}[n]} h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_u),\ \ n \in \boldsymbol{N} \tag{5}$$

The intuitive idea underlying Eq. (5) consists of computing the state $\boldsymbol{x}_n$ by the summing a set of "contributions". Each contribution is generated considering only one node in the neighborhood of n. A similar approach was already used with success in recursive neural networks [7], [8].

Moreover, it is worth to mention that GNNs can be applied also to directed graphs. For this purpose, the input of $f_{\boldsymbol{w}}$ (or $h_{\boldsymbol{w}}$) must be extended with information about the edge directions, f.i. a flag d_u for each node $u \in \text{ne}[n]$ such that $d_u = 1$, if the edge (u, v) is directed toward v, and $d_u = 0$, otherwise. Finally, in graph focused applications only one output for each graph is produced. This can be achieved in several ways. For example a special node s can be selected in each graph and the corresponding output $\boldsymbol{o}_s$ is returned. Such an approach is also used in recursive neural networks.

In order to implement the model formally defined by Equations (1) and (2), the following items must be provided:

1 A method to solve Eq. (1);

2 A learning algorithm to adapt $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ by examples from the train set;

3 An implementation of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$.

These aspects will be considered in the following subsections.

[1] A function $l : \mathbb{R}^a \rightarrow \mathbb{R}^a$ is a contraction mapping w.r.t. a vector norm $\| \cdot \|$, if it exists a real μ, $0 \leq \mu < 1$, such that for any $\boldsymbol{y}_1 \in \mathbb{R}^a$, $\boldsymbol{y}_2 \in \mathbb{R}^a$, $\|l(\boldsymbol{y}_1) - l(\boldsymbol{y}_2)\| \leq \mu \|\boldsymbol{y}_1 - \boldsymbol{y}_2\|$.

A. Computing the states

The Banach fixed point theorem suggests a simple algorithm to compute the fixed point of Eq. (3). It states that if $F_{\boldsymbol{w}}$ is a contraction mapping, then the following dynamical system

$$\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l}) \,, \tag{6}$$

where $\boldsymbol{x}(t)$ denotes the t-th iterate of $\boldsymbol{x}$, converges exponentially fast to the solution of Eq. (3) for any initial state $\boldsymbol{x}(0)$. Thus, $\boldsymbol{x}_n$ and $\boldsymbol{o}_n$ can be obtained by iterating:

$$\begin{array}{rcl} \boldsymbol{x}_n(t+1) & = & f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{x}_{\mathrm{ne}[n]}(t), \boldsymbol{l}_{\mathrm{ne}[n]}), \\ \boldsymbol{o}_n(t+1) & = & g_{\boldsymbol{w}}(\boldsymbol{x}_n(t+1), \boldsymbol{l}_n), \qquad n \in \boldsymbol{N}. \end{array} \tag{7}$$

Note that the computation described in Eq. (7) can be interpreted as the representation of a neural network, called *encoding network*, that consists of units which compute $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ (see Figure 3). In order to build the encoding network, each node of the graph can be replaced by a unit computing the function $f_{\boldsymbol{w}}$. Each unit stores the current state $\boldsymbol{x}_n(t)$ of the corresponding node n, and, when activated, it calculates the state $\boldsymbol{x}_n(t+1)$ using the labels and the states stored in its neighborhood. The simultaneous and repeated activation of the units produces the behavior described by Eq. (7). In the encoding network, the output of node n is produced by another unit which implements $g_{\boldsymbol{w}}$.

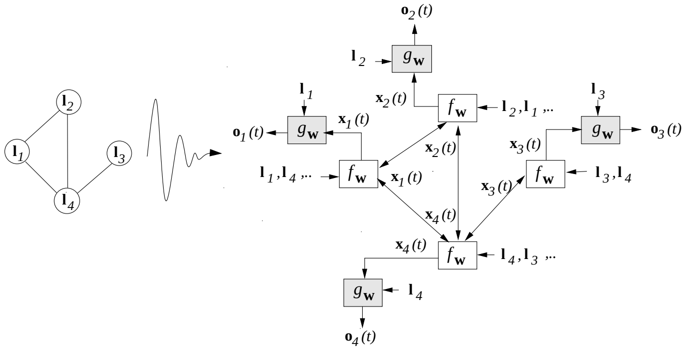


Fig. 3. A graph and its corresponding encoding network.

B. A learning algorithm

The learning algorithm consists of two phases:

(a) the states $\boldsymbol{x}_n(t)$ are iteratively updated, using Eq. (7) until they reach a stable fixed point $\boldsymbol{x}(T) = \boldsymbol{x}$ at time T;

(b) the gradient $\frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{w}}$ is computed and the weights $\boldsymbol{w}$ are updated according to a gradient descent strategy.

Thus, while phase (a) moves the system to a stable point, phase (b) adapts the weights to change the outputs towards the desired target. These two phases are repeated until a given stopping criterion is reached. It can be formally proved that if $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ in Eq. (3) are differentiable w.r.t. $\boldsymbol{w}$ and $\boldsymbol{x}$, then the above learning procedure implements a gradient descendent strategy on the error function $e_{\boldsymbol{w}}$ [5].

In fact our algorithm is obtained by combining the backpropagation through structure algorithm, which is adopted for training recursive neural networks [4], and the Almeida–Pineda algorithm [9], [10]. The latter is a particular version of the backpropagation through time algorithm which can be used to train recurrent networks. 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. It produces a set of gradients, one for each instance of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$. Those gradients are accumulated to compute $\frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{w}}$.

C. Implementing the transition and the output functions

In the following, two different GNN models, called *linear GNN* and *neural GNN*, are described. In both the cases, the output function $g_{\boldsymbol{w}}$ is implemented by a multilayer feedforward neural network and the transition function defined in (5) is used. On the other hand, linear and neural GNNs differ in the implementation of the function $h_{\boldsymbol{w}}$ of Eq. (5) and in the strategy adopted to ensure that $F_{\boldsymbol{w}}$ is a contraction mapping.

1) Linear GNN: In this model, $h_{\boldsymbol{w}}$ is

$$h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_u) = \boldsymbol{A}_{n,u}\boldsymbol{x}_u + \boldsymbol{b}_n \,.$$

where the vector $\boldsymbol{b}_n \in \mathbf{R}^s$ and the matrix $\boldsymbol{A}_{n,u} \in \mathbf{R}^{s\times s}$ are defined by the output of two feedforward neural networks, whose parameters correspond to the parameters of the GNN. More precisely, let $\phi_{\boldsymbol{w}} : \mathbf{R}^{2q} \to \mathbf{R}^{s^2}$ and $\rho_{\boldsymbol{w}} : \mathbf{R}^q \to \mathbf{R}^s$ be the functions implemented by two multilayer feedforward neural networks. Then,

$$\begin{array}{rcl} \boldsymbol{A}_{n,u} & = & \dfrac{\mu}{s|\mathrm{ne}[u]|} \cdot \mathrm{Resize}\left(\phi_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_u)\right) \\ \boldsymbol{b}_n & = & \rho_{\boldsymbol{w}}(\boldsymbol{l}_n)\,, \end{array}$$

where $\mu \in (0,1)$ and $\mathrm{Resize}(\cdot)$ denotes the operator that allocate the elements of s^2-dimensional vector into a $s \times s$ matrix. Here, it is further assumed that $\|\phi_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_u)\|_1 \le s^2$ holds, which is straightforwardly verified if the output neurons of the network implementing $\phi_{\boldsymbol{w}}$ use an appropriately bounded activation function, f.i. a hyperbolic tangent.

Notice that, in this case, $F_{\boldsymbol{w}}$ is a contraction function for any set of parameters $\boldsymbol{w}$. In fact,

$$F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}\,, \tag{8}$$

where $\boldsymbol{b}$ is the vector constructed by stacking all the $\boldsymbol{b}_n$, and $\boldsymbol{A}$ is a block matrix $\{\bar{\boldsymbol{A}}_{n,u}\}$, with $\bar{\boldsymbol{A}}_{n,u} = \boldsymbol{A}_{n,u}$ if u is a neighbor of n and $\bar{\boldsymbol{A}}_{n,u} = \boldsymbol{0}$, otherwise. By simple algebra, it is easily proved that $\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\|_1 = \|\boldsymbol{A}\|_1 \le \mu$ which implies that $F_{\boldsymbol{w}}$ is a contraction function.

2) Neural GNN: In this model, $h_{\boldsymbol{w}}$ is implemented by a feedforward neural network. Since three layer neural networks are universal approximators, this method allows to implement any function $h_{\boldsymbol{w}}$. However, not all the parameters $\boldsymbol{w}$ can be used, because it must be ensured that the corresponding global transition function $F_{\boldsymbol{w}}$ is a contraction. In practice, this goal can be achieved by adding a penalty term to the error function

$$e_{\boldsymbol{w}} = \sum_{i=1}^{p} (\boldsymbol{t}_i - \varphi_{\boldsymbol{w}}(\boldsymbol{G}, n_i))^2 + \beta L\left(\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|_1\right)\,,$$

where $L(y) = (y - \mu)^2$, if $y > \mu$, and $L(y) = 0$ otherwise. Moreover, β is a predefined parameter balancing the importance of the penalty term and the error on patterns, and the parameter $\mu \in (0, 1)$ defines a desired upper bound on contraction constant of $F_{\boldsymbol{w}}$.

III. Experimental results

The approach has been evaluated on a set of toy problems derived from graph theory and applications of practical relevance in machine learning. Each problem belongs to one of the following categories:

1) connection-based problems;
2) label-based problems;
3) general problems.

The first category contains problems where $\tau(\boldsymbol{G}, n)$ depends only on the graph connectivity and is independent from the labels. On the other hand, in label-based problems $\tau(\boldsymbol{G}, n)$ can be computed using only the label $\boldsymbol{l}_n$ of node n. Finally, the last category collects examples in which GNNs must use both topological and labeling information.

Both the linear and the neural model were tested. Three layer (one hidden layer) feedforward networks with sigmoidal activation functions were used to implement the functions involved in the two models, i.e. $g_{\boldsymbol{w}}$, $\phi_{\boldsymbol{w}}$, and $\rho_{\boldsymbol{w}}$ in linear GNNs, and $g_{\boldsymbol{w}}$, $h_{\boldsymbol{w}}$ in neural GNNs (see Section II-C).

Unless otherwise stated, state dimension s was 2. The presented results were averaged on five different trials. In each trial, the dataset was a collection of random connected graphs with a given density δ. The dataset construction procedure consisted of two steps: i) each pair of nodes is connected with a probability δ; ii) the graph is checked to verify whether it is connected or not and, if it is not, random edges are inserted until the condition is satisfied. The dataset was splitted into a train, a validation and a test set. The validation set was used to select the best GNN produced by the learning procedure. Actually, in every trial, the training procedure performed 5000 epochs and every 20 epochs it evaluated the current GNN on validation set. The best GNN was the one that achieved the lowest error on validation set.

GNN software was implemented in Matlab® 7.0.1[2]. The experiments were run on a Power Mac G5 with a 2 GHz PowerPC processor and 2 GB RAM. For all the experiments memory requirements never grew beyond 200 MB.

A. Connection-based problems

1) The Clique problem: A *Clique* of size k is a complete subgraph with k nodes[3] in a larger graph. The goal of this experiment consisted of detecting all the cliques of size 5 in the input graphs. More precisely, the function τ that should be implemented by the GNN was $\tau(\boldsymbol{G}, n) = 1$ if n belongs to a clique of size 5 and $\tau(\boldsymbol{G}, n) = -1$ otherwise. The dataset contained 1400 random graphs with 20 nodes: 200 graphs in the train set, 200 in the validation set, and 1000 in the test set. A clique of size 5 was forced into every graph of the dataset. Thus, each graph had at least one clique, but it might contain more cliques due to the random dataset construction. The desired target $t_n = \tau(\boldsymbol{G}, n)$ of each node was generated by a brute force algorithm that looked for cliques in the graphs. Table I shows the accuracies[4] achieved on this problem by a set of GNNs obtained varying the number of hidden neurons of the feedforward networks. For sake of simplicity, all the feedforward networks involved in a GNN contained the same number of hidden neurons.

TABLE I
Results on the Clique problem

Model	Hidden	Accuracy		Time	
		Test	Train	Test	Train
neural	2	83.73%	83.45%	14.2^s	36^m21^s
	5	86.95%	86.60%	$20.0s$	52^m18^s
	10	90.74%	90.33%	31.3^s	$1^h15^m52^s$
	20	90.20%	89.72%	50.3^s	$1^h34^m03^s$
	30	90.32%	89.82%	1^m10^s	$2^h11^53^s$
linear	2	81.25%	86.90%	2.6^s	53^m21^s
	5	79.87%	86.90%	2.8^s	$1^h01^m20^s$
	10	82.92%	87.54%	3.1^s	48^m54^s
	20	80.90%	88.51%	3.6^s	$1^h00^m33^s$
	30	77.79%	84.94%	4.1^s	$1^h06^m21^s$

The clique problem is a difficult test for GNNs. In fact, GNNs are based on local computation framework, where the computing activity is localized on the nodes of the graph (see Eq. (1)). On the other hand, the detection of a clique requires the knowledge of properties of all the nodes involved in the clique. Notwithstanding, the results of Tab. I confirmed that GNNs can learn to solve this problem.

Notice that Tab. I compares the accuracy achieved on test set with the accuracy of train set. The results, are very close, particularly for the neural model. This proves that the GNN model does not suffer from generalization problems on this experiment. It is also observed that, for the neural GNN, the number of hidden neurons has a clear influence on the results. In fact, a larger number of hiddens corresponds to a better averaged accuracy. On the other hand, a clear relationship between number of hidden neurons and accuracy is not evident for the linear model.

Finally, Tab. I displays the time spent by the training and the testing procedures. It is worth to mention that the computational cost of each learning epoch may depend on the particular train dataset. For example, the number of the iterations of system (7), needed to reach the fixed point, depends on the initial state $\boldsymbol{x}(0)$ (see Section II-C). For this reason, in some cases, even if the neural networks involved in the learning procedure are larger, the computation time may be smaller, e.g. the linear model with 5 and 10 hidden neurons.

In [11], it is stated that the performance of recursive neural networks may be improved if the labels of the graphs are

[2] Copyright © 1994-2004 by The MathWorks, Inc.

[3] A graph is complete if there is an edge between each pair of nodes.

[4] *Accuracy* is defined as the ratio between the correct results and the total number of patterns. A zero threshold was used to decide if the output of the GNN for a certain node is positive or negative.

extended with random vectors. Intuitively, the reason why this approach works is that the random vectors are a sort of identifiers that allow the RNNs to distinguish among the nodes. In practice, the method may or may not work, since the random vectors also inject noise on the dataset, making the learning more difficult. In order to investigate whether such a result holds also for GNNs, we added integer random labels between 0 and 8 to the graphs of the previous dataset and we ran again the experiments. Table II seems to confirm that the result holds also for GNNs, since in most cases GNNs on graph with random labels outperform GNNs on graphs with no labels.

TABLE II

RESULTS ON THE CLIQUE PROBLEM WITH RANDOM LABELS

Model	**Hidden**	**Accuracy**		**Time**	
		Test	**Train**	**Test**	**Train**
neural	2	87.30%	87.51%	16^s	39^m56^s
	5	90.48%	90.53%	23^s	52^m11^s
	10	90.37%	90.33%	35^s	$1^h08^m08^s$
	20	90.69%	91.27%	52^s	$1^h25^m43^s$
	30	89.92%	90.38%	1^m13^s	$2^h02^m04^s$
linear	2	89.05%	89.42%	2.8^s	41^m13^s
	5	86.81%	88.62%	3.0^s	45^m11^s
	10	88.59%	89.05%	3.3^s	42^m29^s
	20	81.07%	87.62%	4.3^s	$1^h17^m31^s$
	30	84.28%	89.12%	4.6^s	$1^h10^m43^s$

2) The Neighbors problem: This very simple problem consists of finding the number of neighbors of each node. Since the information needed to calculate such a number is directly available to the transition function $f_{\boldsymbol{w}}$, GNNs are expected to perform well on such a problem.

On the other hand, the peculiarity of this experiment is in the fact that the dataset involved only one single graph $\boldsymbol{G}$. Notice that this situation may really arise in practical applications. The problem of classifying the Web pages is a straightforward example. The Web is a single big graph, where the nodes represent the pages and the edges stand for the hyperlinks. The learning data set consists of a set of pages whose classification is known, whereas the other pages must be classified by generalization.

For this experiment, the graph had 500 nodes and the dataset $\{(\boldsymbol{G}, n_1, t_1) \ldots (\boldsymbol{G}, n_{500}, t_{500})\}$ contained 500 patterns, one for each node of the graph. Dataset was randomly splitted into a train (100 patterns), a validation (100 patterns), and a test set (300 patterns). The performance is measured by the percentages of the patterns where GNNs achieved an absolute relative error e_r[5] lower than 0.05 and 0.1, respectively. Table III shows the achieved results.

3) The 2-order Neighbors problem: The datasets of this experiments were the same as in the neighbors problem. Here, the problem consists of computing, for each node n, the number of distinct neighbors' neighbors. In other words, the GNN should count the number of nodes that are reachable from n by a path containing 2 edges: the nodes that are connected to n by several paths must be counted only once and n must not be counted. For this reason, such a problem is more difficult than the previous one. Table IV shows the achieved results.

TABLE III

RESULTS ON THE NEIGHBORS PROBLEM

Model	**Hidden**	**Test**		**Training time**
		$e_r < 0.05$	$e_r < 0.1$	
neural	2	73.64%	77.40%	47^m28^s
	5	89.56%	89.76%	$1^h06^m20^s$
	10	90.64%	91.44%	$1^h21^m00^s$
	20	99.04%	99.72%	$2^h23^m27^s$
	30	88.48%	89.48%	$2^h33^m03^s$
linear	2	72.48%	77.24%	58^m45^s
	5	89.60%	89.84%	46^m38^s
	10	99.44%	99.72%	42^m57^s
	20	98.92%	99.68%	42^m53^s
	30	99.16%	99.68%	49^m58^s

TABLE IV

THE 2-ORDER NEIGHBORS PROBLEM

Model	**Hidden**	**Test**		**Training time**
		$e_r < 0.05$	$e_r < 0.1$	
neural	2	65.96%	81.88%	$1^h00^m46^s$
	5	66.00%	81.64%	$1^h34^m14^s$
	10	66.04%	80.00%	$1^h26^m05^s$
	20	72.48%	88.48%	$2^h06^m03^s$
	30	70.40%	81.08%	$3^h27^m52^s$
linear	2	65.92%	81.56%	$1^h04^m14^s$
	5	69.96%	84.04%	48^m02^s
	10	68.84%	84.04%	53^m35^s
	20	73.40%	86.23%	46^m47^s
	30	64.72%	73.96%	48^m25^s

B. Label-based problems

1) The Parity problem: In the Parity problem, learn and validation sets contained 500 graphs, while the test set included 2000 graphs. Each node had a label which was a random vector with 8 boolean elements, i.e. $\boldsymbol{l}_n = [\lambda_1, \ldots, \lambda_8]$, where $\lambda_i \in \{0, 1\}$. The function $\tau(\boldsymbol{G}, n)$ to be learnt is the parity of $\boldsymbol{l}_n$, i.e. $\tau(\boldsymbol{G}, n) = 1$, if $\boldsymbol{l}_n$ contains an even number of ones, and $\tau(\boldsymbol{G}, n) = -1$, otherwise.

The purpose of this experiment is to verify whether the GNN model is able to discard the information contained in the topology of the graphs in those cases where such an information is not needed to solve the problem. Table V shows the achieved results. GNNs with a number of hidden neurons smaller than 5 have a poor performance. However, this has a simple explanation. In this experiment, the output function $g_{\boldsymbol{w}}$ must approximate the parity function: this can be done only if the network that implements $g_{\boldsymbol{w}}$ has a sufficient number of hidden neurons.

The parity problem can be worked out also by a traditional feedforward neural network applied on the node labels.

[5] The *absolute relative error* on pattern i is defined as $|t_i - \varphi_{\boldsymbol{w}}(\boldsymbol{G}, n_i)|/t_i$.

For comparison purposes, Table V shows also the accuracy achieved by a three layer feedforward neural network (FNN) with 20 hidden neurons.

TABLE V

THE PARITY PROBLEM

Model	Hidden	Accuracy		Time	
		Test	Train	Test	Train
neural	2	53.90%	55.41%	10.8^s	29^m22^s
	5	92.77%	93.41%	14.5^s	34^m20^s
	10	97.08%	97.48%	21.3^s	46^m03^s
	20	89.64%	90.05%	34.5^s	$1^h07^m36^s$
	30	92.04%	92.85%	47.8^s	$1^h30^m05^s$
linear	2	63.51%	64.48%	1.9^s	34^m39^s
	5	96.08%	96.55%	2.1^s	40^m02^s
	10	98.21%	98.50%	2.3^s	44^m25^s
	20	99.36%	99.52%	3.0^s	51^m25^s
	30	99.40%	99.64%	3.5^s	59^m47^s
FNN	20	99.46%	99.45%	0.3^s	$1^h09^m04^s$

C. General problems

1) The Subgraph Matching problem: The Subgraph Matching problem consists of identifying the presence of a subgraph $\boldsymbol{S}$ in a larger graph $\boldsymbol{G}$. Such a problem has a number of applications, including object localization and detection of active parts in chemical compounds. Machine learning techniques are useful for this problem when the subgraph is not known in advance and is available only from a set of examples or when the graphs are corrupted by noise.

In our tests, we used 600 connected random graphs, equally divided into the train, the validation and the test set. A smaller subgraph $\boldsymbol{S}$, which was randomly generated in each trial, was inserted into every graph of the dataset. The nodes had integer labels in the range $[0, 10]$, and a small normal noise, with zero mean and a standard deviation of 0.25, was added to all the labels. The goal consisted of predicting whether n is a node of a subgraph $\boldsymbol{S}$, i.e. $\tau(\boldsymbol{G}, n) = 1$, if n belongs to $\boldsymbol{S}$, and $\tau(\boldsymbol{G}, n) = -1$, otherwise.

In all the experiments, the state dimension was $s = 5$ and all the neural networks involved in the GNNs had 5 hidden neurons. Table VI shows the results with several dimensions for $\boldsymbol{S}$ and $\boldsymbol{G}$. In order to evaluate the relative importance of the labels and the connectivity in the localization of the subgraph, also a feedforward neural network (FNN) with 20 hidden neurons was exercised on this test. The FNN tries to solve the problem using only the label $\boldsymbol{l}_n$ of the node. Table VI shows that GNNs outperform the FNNs, confirming that the GNNs used also the graph topology to find $\boldsymbol{S}$.

IV. CONCLUSIONS

A new neural model, called graph neural network (GNN), was presented. GNNs extend recursive neural networks, since they can process a larger class of graphs and can be used on node focused problems. Some preliminary experimental results confirmed that the model is very promising. The experimentation of the approach on larger applications is a matter of future research. From a theoretical point of view, it is interesting to study also the case when the input graph is not predefined but it changes during the learning procedure.

TABLE VI

THE SUBGRAPH MATCHING PROBLEM

			Number of nodes in $\boldsymbol{G}$				
			6	**10**	**14**	**18**	**Avg.**
Number of nodes in $\boldsymbol{S}$	**3**	neural	92.4	90.0	90.0	84.3	89.1
		linear	93.3	84.5	86.7	84.7	87.1
		FNN	81.4	78.2	79.6	82.2	80.3
	5	neural	91.3	87.7	84.9	83.3	86.8
		linear	90.4	85.8	85.3	80.6	85.5
		FNN	85.2	73.2	65.2	75.5	74.8
	7	neural		89.8	84.6	79.9	84.8
		linear		91.3	84.4	79.2	85.0
		FNN		84.2	66.9	64.6	71.6
	9	neural		93.3	84.0	77.8	85.0
		linear		92.2	84.0	77.7	84.7
		FNN		91.6	73.7	67.0	77.4
	Avg.	neural	91.8	90.2	86.9	81.3	
		linear	91.9	85.1	80.3	84.3	
		FNN	83.3	81.8	71.1	73.3	
	Total avg	neural	86.7				
		linear	85.7				
		FNN	75.3				

REFERENCES

[1] T. Schmitt and C. Goller, "Relating chemical structure to activity: An application of the neural folding architecture," in *Workshop on Fuzzy–Neuro Systems '98 and Conference on Egineering Applications of Neural Networks, EANN '98*, 1998.

[2] E. Francesconi, P. Frasconi, M. Gori, S. Marinai, J. Sheng, G. Soda, and A. Sperduti, "Logo recognition by recursive neural networks," in *Lecture Notes in Computer Science — Graphics Recognition*, K. Tombre and A. K. Chhabra, Eds. Springer, 1997, GREC'97 Proceedings.

[3] 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, September 1998.

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

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

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

[7] M. Gori, M. Maggini, and L. Sarti, "A recursive neural network model for processing directed acyclic graphs with labeled edges," in *Proceedings of the International Joint Conference on Neural Networkss*, Portland (USA), July 2003, pp. 1351–1355.

[8] 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), Sept. 2003.

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

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

[11] M. Bianchini, M. Gori, and F. Scarselli, "Recursive processing of cyclic graphs," in *Proceedings of IEEE International Conference on Neural Networks,*, Washington, DC, USA, May 2002, pp. 154–159.