The Graph Neural Network Model

Franco Scarselli, Marco Gori, *Fellow, IEEE*, Ah Chung Tsoi, Markus Hagenbuchner, *Member, IEEE*, and Gabriele Monfardini

***Abstract*—Many underlying relationships among data in several areas of science and engineering, e.g., computer vision, molecular chemistry, molecular biology, pattern recognition, and data mining, can be represented in terms of graphs. In this paper, we propose a new neural network model, called graph neural network (GNN) model, that extends existing neural network methods for processing the data represented in graph domains. This GNN model, which can directly process most of the practically useful types of graphs, e.g., acyclic, cyclic, directed, and undirected, implements a function $\tau(\boldsymbol{G}, n) \in \mathbb{R}^m$ that maps a graph $\boldsymbol{G}$ and one of its nodes n into an m-dimensional Euclidean space. A supervised learning algorithm is derived to estimate the parameters of the proposed GNN model. The computational cost of the proposed algorithm is also considered. Some experimental results are shown to validate the proposed learning algorithm, and to demonstrate its generalization capabilities.**

***Index Terms*—Graphical domains, graph neural networks (GNNs), graph processing, recursive neural networks.**

I. Introduction

DATA can be naturally represented by graph structures in several application areas, including proteomics [1], image analysis [2], scene description [3], [4], software engineering [5], [6], and natural language processing [7]. The simplest kinds of graph structures include single nodes and sequences. But in several applications, the information is organized in more complex graph structures such as trees, acyclic graphs, or cyclic graphs. Traditionally, data relationships exploitation has been the subject of many studies in the community of inductive logic programming and, recently, this research theme has been evolving in different directions [8], also because of the applications of relevant concepts in statistics and neural networks to such areas (see, for example, the recent workshops [9]–[12]).

In machine learning, structured data is often associated with the goal of (supervised or unsupervised) learning from examples a function τ that maps a graph G and one of its nodes n to a vector of reals[1]: $\tau(\boldsymbol{G}, n) \in \mathbb{R}^m$. Applications to a graphical domain can generally be divided into two broad classes, called *graph-focused* and *node-focused* applications, respectively, in this paper. In *graph-focused* applications, the function τ is independent of the node n and implements a classifier or a regressor on a graph structured data set. For example, a chemical compound can be modeled by a graph $\boldsymbol{G}$, the nodes of which stand for atoms (or chemical groups) and the edges of which represent chemical bonds [see Fig. 1(a)] linking together some of the atoms. The mapping $\tau(\boldsymbol{G})$ may be used to estimate the probability that the chemical compound causes a certain disease [13]. In Fig. 1(b), an image is represented by a region adjacency graph where nodes denote homogeneous regions of intensity of the image and arcs represent their adjacency relationship [14]. In this case, $\tau(\boldsymbol{G})$ may be used to classify the image into different classes according to its contents, e.g., castles, cars, people, and so on.

In *node-focused* applications, τ depends on the node n, so that the classification (or the regression) depends on the properties of each node. Object detection is an example of this class of applications. It consists of finding whether an image contains a given object, and, if so, localizing its position [15]. This problem can be solved by a function τ, which classifies the nodes of the region adjacency graph according to whether the corresponding region belongs to the object. For example, the output of τ for Fig. 1(b) might be 1 for black nodes, which correspond to the castle, and 0 otherwise. Another example comes from web page classification. The web can be represented by a graph where nodes stand for pages and edges represent the hyperlinks between them [Fig. 1(c)]. The web connectivity can be exploited, along with page contents, for several purposes, e.g., classifying the pages into a set of topics.

Traditional machine learning applications cope with graph structured data by using a preprocessing phase which maps the graph structured information to a simpler representation, e.g., vectors of reals [16]. In other words, the preprocessing step first "squashes" the graph structured data into a vector of reals and then deals with the preprocessed data using a list-based data processing technique. However, important information, e.g., the topological dependency of information on each node may be lost during the preprocessing stage and the final result may depend, in an unpredictable manner, on the details of the preprocessing algorithm. More recently, there have been various approaches [17], [18] attempting to preserve the graph structured nature of the data for as long as required before the processing

Manuscript received May 24, 2007; revised January 08, 2008 and May 02, 2008; accepted June 15, 2008. First published December 09, 2008; current version published January 05, 2009. This work was supported by the Australian Research Council in the form of an International Research Exchange scheme which facilitated the visit by F. Scarselli to University of Wollongong when the initial work on this paper was performed. This work was also supported by the ARC Linkage International Grant LX045446 and the ARC Discovery Project Grant DP0453089.

F. Scarselli, M. Gori, and G. Monfardini are with the Faculty of Information Engineering, University of Siena, Siena 53100, Italy (e-mail: franco@dii.unisi.it; marco@dii.unisi.it; monfardini@dii.unisi.it).

A. C. Tsoi is with Hong Kong Baptist University, Kowloon, Hong Kong (e-mail: act@hkbu.edu.hk).

M. Hagenbuchner is with the University of Wollongong, Wollongong, N.S.W. 2522, Australia (e-mail: markus@uow.edu.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNN.2008.2005605

[1]Note that in most classification problems, the mapping is to a vector of integers $\mathbb{N}^m$, while in regression problems, the mapping is to a vector of reals $\mathbb{R}^m$. Here, for simplicity of exposition, we will denote only the regression case. The proposed formulation can be trivially rewritten for the situation of classification.

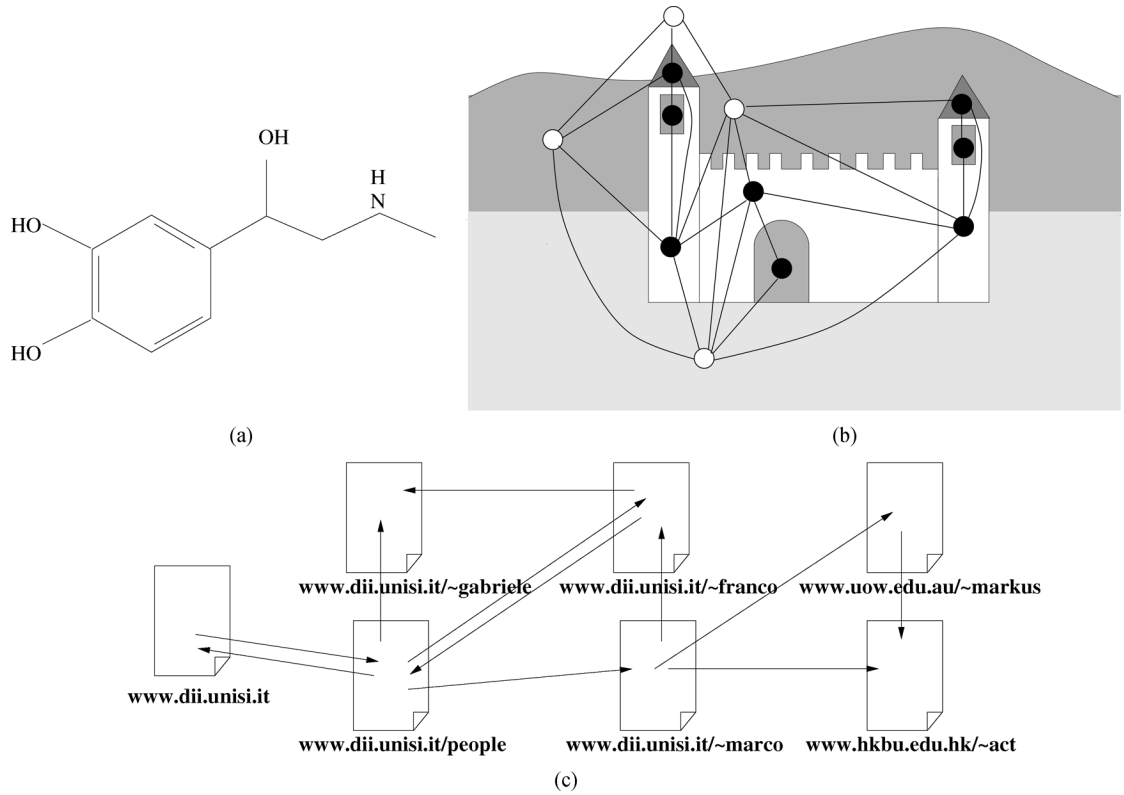

Fig. 1. Some applications where the information is represented by graphs: (a) a chemical compound (adrenaline), (b) an image, and (c) a subset of the web.

phase. The idea is to encode the underlying graph structured data using the topological relationships among the nodes of the graph, in order to incorporate graph structured information in the data processing step. *Recursive neural networks* [17], [19], [20] and *Markov chains* [18], [21], [22] belong to this set of techniques and are commonly applied both to graph and node-focused problems. The method presented in this paper extends these two approaches in that it can deal directly with graph structured information.

Existing recursive neural networks are neural network models whose input domain consists of directed acyclic graphs [17], [19], [20]. The method estimates the parameters w of a function φ_w, which maps a graph to a vector of reals. The approach can also be used for node-focused applications, but in this case, the graph must undergo a preprocessing phase [23]. Similarly, using a preprocessing phase, it is possible to handle certain types of cyclic graphs [24]. Recursive neural networks have been applied to several problems including logical term classification [25], chemical compound classification [26], logo recognition [2], [27], web page scoring [28], and face localization [29].

Recursive neural networks are also related to *support vector machines* [30]–[32], which adopt special kernels to operate on graph structured data. For example, the *diffusion kernel* [33] is based on heat diffusion equation; the kernels proposed in [34] and [35] exploit the vectors produced by a graph random walker and those designed in [36]–[38] use a method of counting the number of common substructures of two trees. In fact, recursive neural networks, similar to support vector machine methods, automatically encode the input graph into an internal representation. However, in recursive neural networks, the internal encoding is learned, while in support vector machine, it is designed by the user.

On the other hand, Markov chain models can emulate processes where the causal connections among events are represented by graphs. Recently, random walk theory, which addresses a particular class of Markov chain models, has been applied with some success to the realization of web page ranking algorithms [18], [21]. Internet search engines use ranking algorithms to measure the relative "importance" of web pages. Such measurements are generally exploited, along with other page features, by "horizontal" search engines, e.g., Google [18], or by personalized search engines ("vertical" search engines; see, e.g., [22]) to sort the universal resource locators (URLs) returned on user queries.[2] Some attempts have been made to extend these models with learning capabilities such that a parametric model representing the behavior of the system can be estimated from a set of training examples extracted from a collection [22], [40], [41]. Those models are able to generalize the results to score all the web pages in the collection. More generally, several other statistical methods have been proposed, which assume that the data set consists of patterns and relationships between patterns. Those techniques include random fields [42], Bayesian networks [43], statistical relational learning [44], transductive learning [45], and semisupervised approaches for graph processing [46].

In this paper, we present a supervised neural network model, which is suitable for both graph and node-focused applications. This model unifies these two existing models into a common

[2]The relative importance measure of a web page is also used to serve other goals, e.g., to improve the efficiency of crawlers [39].

framework. We will call this novel neural network model a graph neural network (GNN). It will be shown that the GNN is an extension of both recursive neural networks and random walk models and that it retains their characteristics. The model extends recursive neural networks since it can process a more general class of graphs including cyclic, directed, and undirected graphs, and it can deal with node-focused applications without any preprocessing steps. The approach extends random walk theory by the introduction of a learning algorithm and by enlarging the class of processes that can be modeled.

GNNs are based on an information diffusion mechanism. A graph is processed by a set of units, each one corresponding to a node of the graph, which are linked according to the graph connectivity. The units update their states and exchange information until they reach a stable equilibrium. The output of a GNN is then computed locally at each node on the base of the unit state. The diffusion mechanism is constrained in order to ensure that a unique stable equilibrium always exists. Such a realization mechanism was already used in cellular neural networks [47]–[50] and Hopfield neural networks [51]. In those neural network models, the connectivity is specified according to a predefined graph, the network connections are recurrent in nature, and the neuron states are computed by relaxation to an equilibrium point. GNNs differ from both the cellular neural networks and Hopfield neural networks in that they can be used for the processing of more general classes of graphs, e.g., graphs containing undirected links, and they adopt a more general diffusion mechanism.

In this paper, a learning algorithm will be introduced, which estimates the parameters of the GNN model on a set of given training examples. In addition, the computational cost of the parameter estimation algorithm will be considered. It is also worth mentioning that elsewhere [52] it is proved that GNNs show a sort of universal approximation property and, under mild conditions, they can approximate most of the practically useful functions φ on graphs.[3]

The structure of this paper is as follows. After a brief description of the notation used in this paper as well as some preliminary definitions, Section II presents the concept of a GNN model, together with the proposed learning algorithm for the estimation of the GNN parameters. Moreover, Section III discusses the computational cost of the learning algorithm. Some experimental results are presented in Section IV. Conclusions are drawn in Section V.

II. The Graph Neural Network Model

We begin by introducing some notations that will be used throughout the paper. A graph $\boldsymbol{G}$ is a pair $(\boldsymbol{N}, \boldsymbol{E})$, where $\boldsymbol{N}$ is the set of nodes and $\boldsymbol{E}$ is the set of edges. The set $\mathrm{ne}[n]$ stands for the neighbors of n, i.e., the nodes connected to n by an arc, while $\mathrm{co}[n]$ denotes the set of arcs having n as a vertex. Nodes and edges may have labels represented by real vectors. The labels attached to node n and edge (n_1, n_2) will be represented by $\boldsymbol{l}_n \in I\!R^{l_N}$ and $\boldsymbol{l}_{(n_1,n_2)} \in I\!R^{l_E}$, respectively. Let $\boldsymbol{l}$ denote the vector obtained by stacking together all the labels of the graph. The notation adopted for labels follows a more general scheme: if $\boldsymbol{y}$ is a vector that contains data from a graph and $\boldsymbol{S}$ is a subset of the nodes (the edges), then $\boldsymbol{y}_S$ denotes the vector obtained by selecting from $\boldsymbol{y}$ the components related to the node (the edges) in S. For example, $\boldsymbol{l}_{\mathrm{ne}[n]}$ stands for the vector containing the labels of all the neighbors of n. Labels usually include features of objects related to nodes and features of the relationships between the objects. For example, in the case of an image as in Fig. 1(b), node labels might represent properties of the regions (e.g., area, perimeter, and average color intensity), while edge labels might represent the relative position of the regions (e.g., the distance between their barycenters and the angle between their principal axes). No assumption is made on the arcs; directed and undirected edges are both permitted. However, when different kinds of edges coexist in the same data set, it is necessary to distinguish them. This can be easily achieved by attaching a proper label to each edge. In this case, different kinds of arcs turn out to be just arcs with different labels.

The considered graphs may be either positional or nonpositional. Nonpositional graphs are those described so far; positional graphs differ since a unique integer identifier is assigned to each neighbors of a node n to indicate its logical position. Formally, for each node n in a positional graph, there exists an injective function $\nu_n : \mathrm{ne}[n] \rightarrow \{1, \ldots |\mathbf{N}|\}$, which assigns to each neighbor u of n a position $\nu_n(u)$. Note that the position of the neighbor can be implicitly used for storing useful information. For instance, let us consider the example of the region adjacency graph [see Fig. 1(b)]: ν_n can be used to represent the relative spatial position of the regions, e.g., ν_n might enumerate the neighbors of a node n, which represents the adjacent regions, following a clockwise ordering convention.

The domain considered in this paper is the set $\mathcal{D}$ of pairs of a graph and a node, i.e., $\mathcal{D} = \mathcal{G} \times \mathcal{N}$ where $\mathcal{G}$ is a set of the graphs and $\mathcal{N}$ is a subset of their nodes. We assume a supervised learning framework with the learning set

$$\mathcal{L} = \{(\boldsymbol{G}_i, n_{i,j}, \boldsymbol{t}_{i,j}) | \boldsymbol{G}_i = (\boldsymbol{N}_i, \boldsymbol{E}_i) \in \mathcal{G}; \; n_{i,j} \in \boldsymbol{N}_i; \; \boldsymbol{t}_{i,j} \in I\!R^m, \; 1 \leq i \leq p, 1 \leq j \leq q_i\}$$

where $n_{i,j} \in \boldsymbol{N}_i$ denotes the jth node in the set $\boldsymbol{N}_i \in \mathcal{N}$ and $\boldsymbol{t}_{i,j}$ is the desired target associated to $n_{i,j}$. Finally, $p \leq |\mathcal{G}|$ and $q_i \leq |\boldsymbol{N}_i|$. Interestingly, all the graphs of the learning set can be combined into a unique disconnected graph, and, therefore, one might think of the learning set as the pair $\mathcal{L} = (\boldsymbol{G}, \mathcal{T})$ where $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{E})$ is a graph and $\mathcal{T}$ a is set of pairs $\{(n_i, \boldsymbol{t}_i) | \, n_i \in N, \boldsymbol{t}_i \in I\!R^m, 1 \leq i \leq q\}$. It is worth mentioning that this compact definition is not only useful for its simplicity, but that it also captures directly the very nature of some problems where the domain consists of only one graph, for instance, a large portion of the web [see Fig. 1(c)].

A. The Model

The intuitive idea underlining the proposed approach is that nodes in a graph represent objects or concepts, and edges represent their relationships. Each concept is naturally defined by its features and the related concepts. Thus, we can attach a *state*

[3]Due to the length of proofs, such results cannot be shown here and is included in [52].

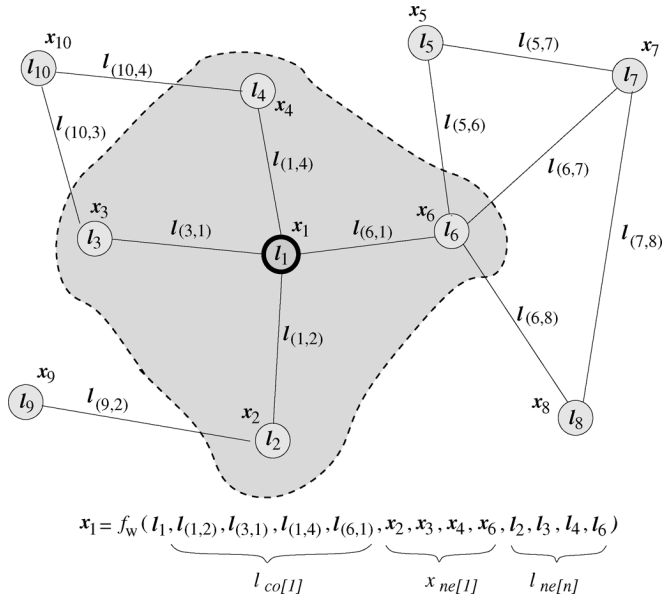

Fig. 2. Graph and the neighborhood of a node. The state $\boldsymbol{x}_1$ of the node 1 depends on the information contained in its neighborhood.

$\boldsymbol{x}_n \in \mathbb{R}^s$ to each node n that is based on the information contained in the neighborhood of n (see Fig. 2). The state $\boldsymbol{x}_n$ contains a representation of the concept denoted by n and can be used to produce an *output* $\boldsymbol{o}_n$, i.e., a decision about the concept.

Let $f_{\boldsymbol{w}}$ be a parametric function, called *local transition function*, that expresses the dependence of a node n on its neighborhood and let $g_{\boldsymbol{w}}$ be the *local output function* that describes how the output is produced. Then, $\boldsymbol{x}_n$ and $\boldsymbol{o}_n$ are defined as follows:

$$\begin{aligned} \boldsymbol{x}_n &= f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{\mathrm{co}[n]}, \boldsymbol{x}_{\mathrm{ne}[n]}, \boldsymbol{l}_{\mathrm{ne}[n]}) \\ \boldsymbol{o}_n &= g_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{l}_n) \end{aligned} \tag{1}$$

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

Remark 1: Different notions of neighborhood can be adopted. For example, one may wish to remove the labels $\boldsymbol{l}_{\mathrm{ne}[n]}$, since they include information that is implicitly contained in $\boldsymbol{x}_{\mathrm{ne}[n]}$. Moreover, the neighborhood could contain nodes that are two or more links away from n. In general, (1) could be simplified in several different ways and several minimal models[4] exist. In the following, the discussion will mainly be based on the form defined by (1), which is not minimal, but it is the one that more closely represents our intuitive notion of neighborhood. ■

Remark 2: Equation (1) is customized for undirected graphs. When dealing with directed graphs, the function $f_{\boldsymbol{w}}$ can also accept as input a representation of the direction of the arcs. For example, $f_{\boldsymbol{w}}$ may take as input a variable d_ℓ for each arc $\ell \in \mathrm{co}[n]$ such that $d_\ell = 1$, if ℓ is directed towards n and $d_\ell = 0$, if ℓ comes from n. In the following, in order to keep the notations compact, we maintain the customization of (1). However, unless explicitly stated, all the results proposed in this paper hold also for directed graphs and for graphs with mixed directed and undirected links. ■

Remark 3: In general, the transition and the output functions and their parameters may depend on the node n. In fact, it is plausible that different mechanisms (implementations) are used to represent different kinds of objects. In this case, each kind of nodes k_n has its own transition function f^{k_n}, output function g^{k_n}, and a set of parameters $\boldsymbol{w}_{k_n}$. Thus, (1) becomes $\boldsymbol{x}_n = (f^{k_n})_{\boldsymbol{w}_{k_n}}(\boldsymbol{l}_n, \boldsymbol{l}_{\mathrm{co}[n]}, \boldsymbol{x}_{\mathrm{ne}[n]}, \boldsymbol{l}_{\mathrm{ne}[n]})$ and $\boldsymbol{o}_n = (g^{k_n})_{\boldsymbol{w}_{k_n}}(\boldsymbol{x}_n, \boldsymbol{l}_n)$. However, for the sake of simplicity, our analysis will consider (1) that describes a particular model where all the nodes share the same implementation. ■

Let $\boldsymbol{x}$, $\boldsymbol{o}$, $\boldsymbol{l}$, and $\boldsymbol{l_N}$ be the vectors constructed by stacking all the states, all the outputs, all the labels, and all the node labels, respectively. Then, (1) can be rewritten in a compact form as

$$\begin{aligned} \boldsymbol{x} &= F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \\ \boldsymbol{o} &= G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N}) \end{aligned} \tag{2}$$

where $F_{\boldsymbol{w}}$, the *global transition function* and $G_{\boldsymbol{w}}$, the *global output function* are stacked versions of $|\boldsymbol{N}|$ instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, respectively.

We are interested in the case when $\boldsymbol{x}, \boldsymbol{o}$ are uniquely defined and (2) defines a map $\varphi_{\boldsymbol{w}} : \mathcal{D} \rightarrow \mathbb{R}^m$, which takes a graph as input and returns an output $\boldsymbol{o}_n$ for each node. The Banach fixed point theorem [53] provides a sufficient condition for the existence and uniqueness of the solution of a system of equations. According to Banach's theorem [53], (2) has a unique solution provided that $F_{\boldsymbol{w}}$ is a *contraction map* with respect to the state, i.e., there exists μ, $0 \leq \mu < 1$, such that $\|F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) - F_{\boldsymbol{w}}(\boldsymbol{y}, \boldsymbol{l})\| \leq \mu \|\boldsymbol{x} - \boldsymbol{y}\|$ holds for any $\boldsymbol{x}, \boldsymbol{y}$, where $\|\cdot\|$ denotes a vectorial norm. Thus, for the moment, let us assume that $F_{\boldsymbol{w}}$ is a contraction map. Later, we will show that, in GNNs, this property is enforced by an appropriate implementation of the transition function.

Note that (1) makes it possible to process both positional and nonpositional graphs. For positional graphs, $f_{\boldsymbol{w}}$ must receive the positions of the neighbors as additional inputs. In practice, this can be easily achieved provided that information contained in $\boldsymbol{x}_{\mathrm{ne}[n]}$, $\boldsymbol{l}_{\mathrm{co}[n]}$, and $\boldsymbol{l}_{\mathrm{ne}[n]}$ is sorted according to neighbors' positions and is properly padded with special null values in positions corresponding to nonexisting neighbors. For example, $\boldsymbol{x}_{\mathrm{ne}[n]} = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_M]$, where $M = \max_{n,u} \nu_n(u)$ is the maximal number of neighbors of a node; $\boldsymbol{y}_i = \boldsymbol{x}_u$ holds, if u is the ith neighbor of n ($\nu_n(u) = i$); and $\boldsymbol{y}_i = \boldsymbol{x}_0$, for some predefined null state $\boldsymbol{x}_0$, if there is no ith neighbor.

However, for nonpositional graphs, it is useful to replace function $f_{\boldsymbol{w}}$ of (1) with

$$\boldsymbol{x}_n = \sum_{u \in \mathrm{ne}[n]} h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u), \qquad n \in \boldsymbol{N} \tag{3}$$

where $h_{\boldsymbol{w}}$ is a parametric function. This transition function, which has been successfully used in recursive neural networks [54], is not affected by the positions and the number of the children. In the following, (3) is referred to as the *nonpositional form*, while (1) is called the *positional form*. In order to implement the GNN model, the following items must be provided:

1) a method to solve (1);

[4]A model is said to be minimal if it has the smallest number of variables while retaining the same computational power.

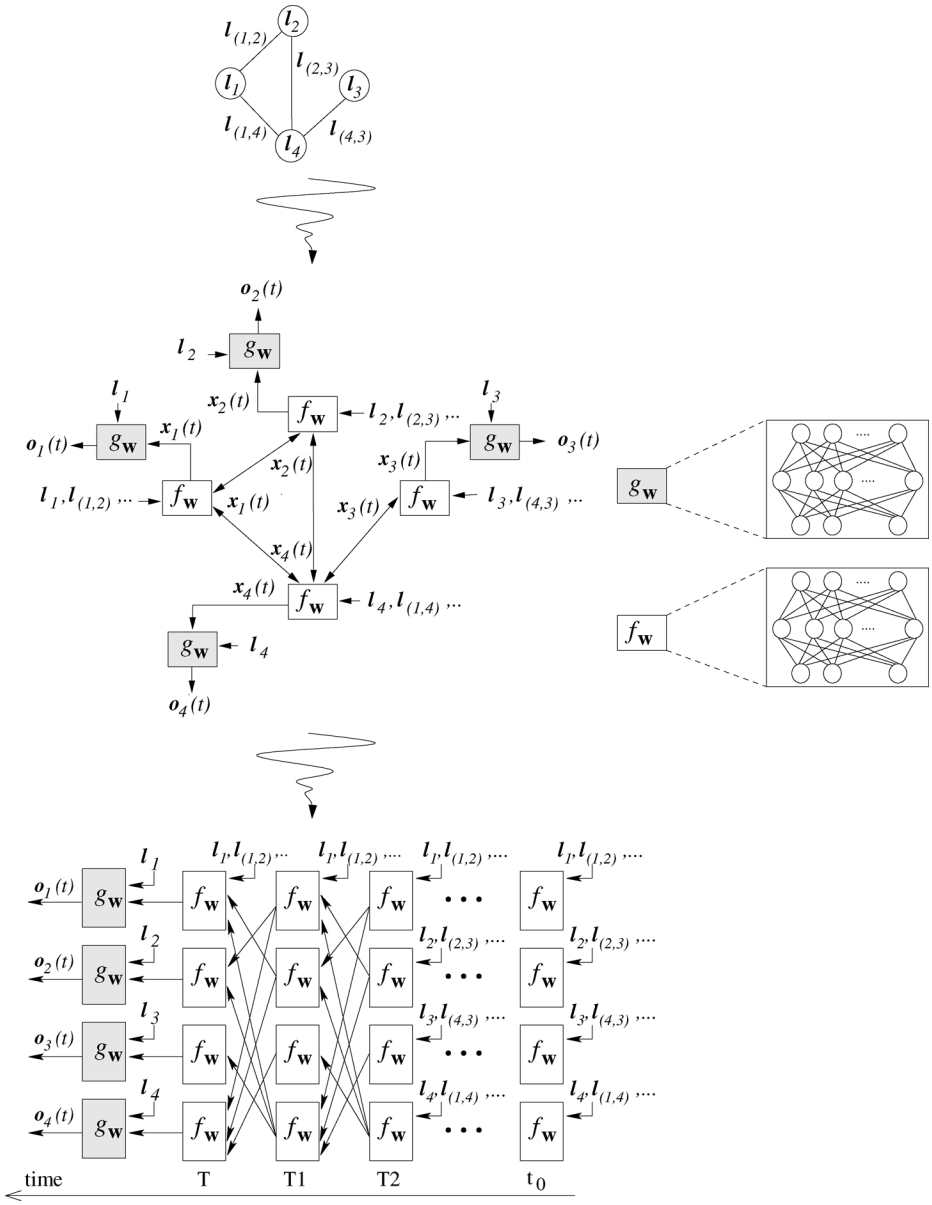

Fig. 3. Graph (on the top), the corresponding encoding network (in the middle), and the network obtained by unfolding the encoding network (at the bottom). The nodes (the circles) of the graph are replaced, in the encoding network, by units computing $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ (the squares). When $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ are implemented by feedforward neural networks, the encoding network is a recurrent neural network. In the unfolding network, each layer corresponds to a time instant and contains a copy of all the units of the encoding network. Connections between layers depend on encoding network connectivity.

2) a learning algorithm to adapt $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ using examples from the training data set[5];
3) an implementation of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$.

These aspects will be considered in turn in the following sections.

B. Computation of the State

Banach's fixed point theorem [53] does not only ensure the existence and the uniqueness of the solution of (1) but it also suggests the following classic iterative scheme for computing the state:

$$\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l}) \tag{4}$$

where $\boldsymbol{x}(t)$ denotes the tth iteration of $\boldsymbol{x}$. The dynamical system (4) converges exponentially fast to the solution of (2) for any initial value $\boldsymbol{x}(0)$. We can, therefore, think of $\boldsymbol{x}(t)$ as the state that is updated by the transition function $F_{\boldsymbol{w}}$. In fact, (4) implements the Jacobi iterative method for solving nonlinear equations [55]. Thus, the outputs and the states can be computed by iterating

$$\begin{aligned} \boldsymbol{x}_n(t+1) &= f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{\mathrm{co}[n]}, \boldsymbol{x}_{\mathrm{ne}[n]}(t), \boldsymbol{l}_{\mathrm{ne}[n]}) \\ \boldsymbol{o}_n(t) &= g_{\boldsymbol{w}}(\boldsymbol{x}_n(t), \boldsymbol{l}_n), \qquad n \in \boldsymbol{N}. \end{aligned} \tag{5}$$

Note that the computation described in (5) can be interpreted as the representation of a network consisting of units, which compute $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$. Such a network will be called an *encoding network*, following an analog terminology used for the recursive

[5]In other words, the parameters $\boldsymbol{w}$ are estimated using examples contained in the training data set.

neural network model [17]. In order to build the encoding network, each node of the graph is replaced by a unit computing the function $f_{\boldsymbol{w}}$ (see Fig. 3). Each unit stores the current state $\boldsymbol{x}_n(t)$ of node n, and, when activated, it calculates the state $\boldsymbol{x}_n(t+1)$ using the node label and the information stored in the neighborhood. The simultaneous and repeated activation of the units produce the behavior described in (5). The output of node n is produced by another unit, which implements $g_{\boldsymbol{w}}$.

When $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ are implemented by feedforward neural networks, the encoding network turns out to be a recurrent neural network where the connections between the neurons can be divided into internal and external connections. The internal connectivity is determined by the neural network architecture used to implement the unit. The external connectivity depends on the edges of the processed graph.

C. The Learning Algorithm

Learning in GNNs consists of estimating the parameter $\boldsymbol{w}$ such that $\varphi_{\boldsymbol{w}}$ approximates the data in the learning data set

$$\mathcal{L} = \{(\boldsymbol{G}_i, n_{i,j}, \boldsymbol{t}_{i,j})|, \boldsymbol{G}_i = (\boldsymbol{N}_i, \boldsymbol{E}_i) \in \mathcal{G}; \\ n_{i,j} \in \boldsymbol{N}_i; \boldsymbol{t}_{i,j} \in \mathbb{R}^m, 1 \le i \le p, 1 \le j \le q_i\}$$

where q_i is the number of supervised nodes in $\boldsymbol{G}_i$. For graph-focused tasks, one special node is used for the target ($q_i = 1$ holds), whereas for node-focused tasks, in principle, the supervision can be performed on every node. The learning task can be posed as the minimization of a quadratic cost function

$$e_{\boldsymbol{w}} = \sum_{i=1}^{p} \sum_{j=1}^{q_i} (\boldsymbol{t}_{i,j} - \varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}))^2 . \tag{6}$$

Remark 4: As common in neural network applications, the cost function may include a penalty term to control other properties of the model. For example, the cost function may contain a smoothing factor to penalize any abrupt changes of the outputs and to improve the generalization performance. ■

The learning algorithm is based on a gradient-descent strategy and is composed of the following steps.

a) The states $\boldsymbol{x}_n(t)$ are iteratively updated by (5) until at time T they approach the fixed point solution of (2): $\boldsymbol{x}(T) \approx \boldsymbol{x}$.
b) The gradient $\partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{w}$ is computed.
c) The weights $\boldsymbol{w}$ are updated according to the gradient computed in step b).

Concerning step a), note that the hypothesis that $F_{\boldsymbol{w}}$ is a contraction map ensures the convergence to the fixed point. Step c) is carried out within the traditional framework of gradient descent. As shown in the following, step b) can be carried out in a very efficient way by exploiting the diffusion process that takes place in GNNs. Interestingly, this diffusion process is very much related to the one which takes place in recurrent neural networks, for which the gradient computation is based on backpropagation-through-time algorithm [17], [56], [57]. In this case, the encoding network is unfolded from time T back to an initial time t_0. The unfolding produces the layered network shown in Fig. 3. Each layer corresponds to a time instant and contains a copy of all the units $f_{\boldsymbol{w}}$ of the encoding network. The units of two consecutive layers are connected following graph connectivity. The last layer corresponding to time T includes also the units $g_{\boldsymbol{w}}$ and computes the output of the network. Backpropagation through time consists of carrying out the traditional backpropagation step on the unfolded network to compute the gradient of the cost function at time T with respect to (w.r.t.) all the instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$. Then, $\partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{w}$ is obtained by summing the gradients of all instances. However, backpropagation through time requires to store the states of every instance of the units. When the graphs and $T - t_0$ are large, the memory required may be considerable.[6] On the other hand, in our case, a more efficient approach is possible, based on the Almeida–Pineda algorithm [58], [59]. Since (5) has reached a stable point $\boldsymbol{x}$ before the gradient computation, we can assume that $\boldsymbol{x}(t) = \boldsymbol{x}$ holds for any $t \ge t_0$. Thus, backpropagation through time can be carried out by storing only $\boldsymbol{x}$. The following two theorems show that such an intuitive approach has a formal justification. The former theorem proves that function $\varphi_{\boldsymbol{w}}$ is differentiable.

Theorem 1 (Differentiability): Let $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ be the global transition and the global output functions of a GNN, respectively. If $F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$ and $G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$ are continuously differentiable w.r.t. $\boldsymbol{x}$ and $\boldsymbol{w}$, then $\varphi_{\boldsymbol{w}}$ is continuously differentiable w.r.t. $\boldsymbol{w}$.

Proof: Let a function Θ be defined as $\Theta(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{x} - F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$. Such a function is continuously differentiable w.r.t. $\boldsymbol{x}$ and $\boldsymbol{w}$, since it is the difference of two continuously differentiable functions. Note that the Jacobian matrix $(\partial\Theta/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})$ of Θ w.r.t. $\boldsymbol{x}$ fulfills $(\partial\Theta/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{I}_a - (\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$, where $\boldsymbol{I}_a$ denotes the a-dimensional identity matrix and $a = s|\boldsymbol{N}|$, s is the dimension of the state. Since $F_{\boldsymbol{w}}$ is a contraction map, there exists $\mu, 0 \le \mu < 1$ such that $||(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})|| \le \mu$, which implies $||(\partial\Theta/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})|| \ge (1 - \mu)$. Thus, the determinant of $(\partial\Theta/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})$ is not null and we can apply the implicit function theorem (see [60]) to Θ and point $\boldsymbol{w}$. As a consequence, there exists a function Ψ, which is defined and continuously differentiable in a neighborhood of $\boldsymbol{w}$, such that $\Theta(\Psi(w), w) = \boldsymbol{0}$ and $\Psi(w) = F_{\boldsymbol{w}}(\Psi(\boldsymbol{w}), \boldsymbol{l})$. Since this result holds for any $\boldsymbol{w}$, it is demonstrated that Ψ is continuously differentiable on the whole domain. Finally, note that $\varphi_{\boldsymbol{w}}(\boldsymbol{G}, n) = [G_{\boldsymbol{w}}(\Psi(\boldsymbol{w}), \boldsymbol{l}_{\boldsymbol{N}})]_n$, where $[\cdot]_n$ denotes the operator that returns the components corresponding to node n. Thus, $\varphi_{\boldsymbol{w}}$ is the composition of differentiable functions and hence is itself differentiable. ■

It is worth mentioning that this property does not hold for general dynamical systems for which a slight change in the parameters can force the transition from one fixed point to another. The fact that $\varphi_{\boldsymbol{w}}$ is differentiable in GNNs is due to the assumption that $F_{\boldsymbol{w}}$ is a contraction map. The next theorem provides a method for an efficient computation of the gradient.

Theorem 2 (Backpropagation): Let $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ be the transition and the output functions of a GNN, respectively, and assume that $F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$ and $G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$ are continuously differentiable w.r.t. $\boldsymbol{x}$ and $\boldsymbol{w}$. Let $\boldsymbol{z}(t)$ be defined by

$$\boldsymbol{z}(t) = \boldsymbol{z}(t+1) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) . \tag{7}$$

[6]For internet applications, the graph may represent a significant portion of the web. This is an example of cases when the amount of the required memory storage may play a very important role.

Then, the sequence $\boldsymbol{z}(T), \boldsymbol{z}(T-1), \ldots$ converges to a vector $\boldsymbol{z} = \lim_{t\to-\infty} \boldsymbol{z}(t)$ and the convergence is exponential and independent of the initial state $\boldsymbol{z}(T)$. Moreover

$$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N}) + \boldsymbol{z} \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \tag{8}$$

holds, where $\boldsymbol{x}$ is the stable state of the GNN.

Proof: Since $F_{\boldsymbol{w}}$ is a contraction map, there exists $\mu, 0 \le \mu < 1$ such that $\|(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})\| \le \mu$ holds. Thus, (7) converges to a stable fixed point for each initial state. The stable fixed point $\boldsymbol{z}$ is the solution of (7) and satisfies

$$\boldsymbol{z} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l_N}) \cdot \left(\boldsymbol{I}_a - \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{-1} \tag{9}$$

where $a = s|\boldsymbol{N}|$ holds. Moreover, let us consider again the function Ψ defined in the proof of Theorem 1. By the implicit function theorem

$$\frac{\partial \Psi}{\partial \boldsymbol{w}} = \left(\boldsymbol{I}_a - \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{-1} \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \tag{10}$$

holds. On the other hand, since the error $e_{\boldsymbol{w}}$ depends on the output of the network $\boldsymbol{o} = G_{\boldsymbol{w}}(\Psi(\boldsymbol{w}), \boldsymbol{l_N}))$, the gradient $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{w}$ can be computed using the chain rule for differentiation

$$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l_N}) \cdot \frac{\partial \Psi}{\partial \boldsymbol{w}}(\boldsymbol{w}). \tag{11}$$

The theorem follows by putting together (9)–(11)

$$\begin{aligned}\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} &= \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l_N}) \\ &\quad \cdot \left(\boldsymbol{I}_a - \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{-1} \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \\ &= \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N}) + \boldsymbol{z} \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}). \end{aligned}$$

■

The relationship between the gradient defined by (8) and the gradient computed by the Almeida–Pineda algorithm can be easily recognized. The first term on the right-hand side of (8) represents the contribution to the gradient due to the output function $G_{\boldsymbol{w}}$. Backpropagation calculates the first term while it is propagating the derivatives through the layer of the functions $g_{\boldsymbol{w}}$ (see Fig. 3). The second term represents the contribution due to the transition function $F_{\boldsymbol{w}}$. In fact, from (7)

$$\begin{aligned}\boldsymbol{z}(t) &= \boldsymbol{z}(t+1) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l_N}) \\ &= \boldsymbol{z}(T) \cdot \left(\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{T-t} \\ &\quad + \sum_{i=0}^{T-t-1} \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l_N}) \cdot \left(\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{i}. \end{aligned}$$

If we assume $\boldsymbol{z}(T) = \partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{o}(T) \cdot (\partial G_{\boldsymbol{w}}/\partial \boldsymbol{x}(T))(\boldsymbol{x}(T), \boldsymbol{l_N})$ and $\boldsymbol{x}(t) = \boldsymbol{x}$, for $t_0 \le t \le T$, it follows:

$$\begin{aligned}\boldsymbol{z}(t) &= \sum_{i=0}^{T-t} \frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{o}(T)} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}(T)}(\boldsymbol{x}(T), \boldsymbol{l_N}) \\ &\quad \cdot \prod_{j=1}^{i} \left(\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}(T-j)}(\boldsymbol{x}(T-j), \boldsymbol{l})\right) \\ &= \sum_{i=0}^{T-t} \frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{x}(T-i)} = \sum_{i=T}^{t} \frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{x}(i)}. \end{aligned}$$

Thus, (7) accumulates the $\partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{x}(i)$ into the variable $\boldsymbol{z}$. This mechanism corresponds to backpropagate the gradients through the layers containing the $f_{\boldsymbol{w}}$ units.

TABLE I
LEARNING ALGORITHM. THE FUNCTION FORWARD COMPUTES THE STATES, WHILE BACKWARD CALCULATES THE GRADIENT. THE PROCEDURE MAIN MINIMIZES THE ERROR BY CALLING ITERATIVELY FORWARD AND BACKWARD

```
MAIN
    initialize w;
    x=Forward(w);
    repeat
        ∂e_w/∂w=BACKWARD(x, w);
        w=w − λ · ∂e_w/∂w;
        x=FORWARD(w);
    until (a stopping criterion);
    return w;
end

FORWARD(w)
    initialize x(0), t = 0;
    repeat
        x(t + 1) = F_w(x(t), l);
        t=t + 1;
    until ‖x(t) − x(t − 1)‖ ≤ ε_f
    return x(t);
end

BACKWARD(x,w)
    o = G_w(x, l_N);
    A = ∂F_w/∂x (x, l);
    b = ∂e_w/∂o · ∂G_w/∂x (x, l_N);
    initialize z(0), t=0;
    repeat
        z(t) = z(t + 1) · A + b;
        t=t − 1;
    until ‖z(t − 1) − z(t)‖ ≤ ε_b;
    c = ∂e_w/∂o · ∂G_w/∂w (x, l_N);
    d = z(t) · ∂F_w/∂w (x, l);
    ∂e_w/∂w = c + d;
    return ∂e_w/∂w;
end
```

The learning algorithm is detailed in Table I. It consists of a main procedure and of two functions FORWARD and BACKWARD. Function FORWARD takes as input the current set of parameters $\boldsymbol{w}$ and iterates to find the convergent point, i.e., the fixed point. The iteration is stopped when $\|\boldsymbol{x}(t) - \boldsymbol{x}(t-1)\|$ is less than a given threshold ε_f according to a given norm $\|\cdot\|$. Function BACKWARD computes the gradient: system (7) is iterated until $\|\boldsymbol{z}(t-1) - \boldsymbol{z}(t)\|$ is smaller than a threshold ε_b; then, the gradient is calculated by (8).

The main procedure updates the weights until the output reaches a desired accuracy or some other stopping criterion is achieved. In Table I, a predefined learning rate λ is adopted, but most of the common strategies based on the gradient-descent strategy can be used as well, for example, we can use a momentum term and an adaptive learning rate scheme. In our GNN simulator, the weights are updated by the resilient backpropagation [61] strategy, which, according to the literature

on feedforward neural networks, is one of the most efficient strategies for this purpose. On the other hand, the design of learning algorithms for GNNs that are not explicitly based on gradient is not obvious and it is a matter of future research. In fact, the encoding network is only apparently similar to a static feedforward network, because the number of the layers is dynamically determined and the weights are partially shared according to input graph topology. Thus, second-order learning algorithms [62], pruning [63], and growing learning algorithms [64]–[66] designed for static networks cannot be directly applied to GNNs. Other implementation details along with a computational cost analysis of the proposed algorithm are included in Section III.

D. Transition and Output Function Implementations

The implementation of the local output function $g_{\boldsymbol{w}}$ does not need to fulfill any particular constraint. In GNNs, $g_{\boldsymbol{w}}$ is a multilayered feedforward neural network. On the other hand, the local transition function $f_{\boldsymbol{w}}$ plays a crucial role in the proposed model, since its implementation determines the number and the existence of the solutions of (1). The assumption behind GNN is that the design of $f_{\boldsymbol{w}}$ is such that the global transition function $F_{\boldsymbol{w}}$ is a contraction map w.r.t. the state $\boldsymbol{x}$. In the following, we describe two neural network models that fulfill this purpose using different strategies. These models are based on the nonpositional form described by (3). It can be easily observed that there exist two corresponding models based on the positional form as well.

1) *Linear (nonpositional) GNN.* Equation (3) can naturally be implemented by

$$h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) = \boldsymbol{A}_{n,u}\boldsymbol{x}_u + \boldsymbol{b}_n \tag{12}$$

where the vector $\boldsymbol{b}_n \in \mathbb{R}^s$ and the matrix $\boldsymbol{A}_{n,u} \in \mathbb{R}^{s\times s}$ are defined by the output of two feedforward neural networks (FNNs), whose parameters correspond to the parameters of the GNN. More precisely, let us call *transition network* an FNN that has to generate $\boldsymbol{A}_{n,u}$ and *forcing network* another FNN that has to generate $\boldsymbol{b}_n$. Moreover, let $\phi_{\boldsymbol{w}} : \mathbb{R}^{2l_N+l_E} \rightarrow \mathbb{R}^{s^2}$ and $\rho_{\boldsymbol{w}} : \mathbb{R}^{l_N} \rightarrow \mathbb{R}^s$ be the functions implemented by the transition and the forcing network, respectively. Then, we define

$$\boldsymbol{A}_{n,u} = \frac{\mu}{s|\mathrm{ne}[u]|} \cdot \boldsymbol{\Xi} \tag{13}$$

$$\boldsymbol{b}_n = \rho_{\boldsymbol{w}}(\boldsymbol{l}_n) \tag{14}$$

where $\mu \in (0,1)$ and $\boldsymbol{\Xi} = \mathrm{resize}(\phi_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u))$ hold, and $\mathrm{resize}(\cdot)$ denotes the operator that allocates the elements of a s^2-dimensional vector into as $s\times s$ matrix. Thus, $\boldsymbol{A}_{n,u}$ is obtained by arranging the outputs of the transition network into the square matrix $\boldsymbol{\Xi}$ and by multiplication with the factor $\mu/s|\mathrm{ne}[u]|$. On the other hand, $\boldsymbol{b}_n$ is just a vector that contains the outputs of the forcing network. Here, it is further assumed that $\|\phi_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u)\|_1 \leq s$ holds[7]; this can be straightforwardly verified if the output neurons of the transition network use an appropriately bounded activation function, e.g., a hyperbolic tangent. Note that in this case $F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$, 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. Moreover, vectors $\boldsymbol{b}_n$ and matrices $\boldsymbol{A}_{n,u}$ do not depend on the state $\boldsymbol{x}$, but only on node and edge labels. Thus, $\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x} = \boldsymbol{A}$, and, by simple algebra

$$\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|_1 = \|\boldsymbol{A}\|_1 \leq \max_{u\in \boldsymbol{N}} \left(\sum_{n\in \mathrm{ne}[u]} \|\boldsymbol{A}_{n,u}\|_1\right)$$

$$\leq \max_{u\in \boldsymbol{N}} \left(\frac{\mu}{s|\mathrm{ne}[u]|} \cdot \sum_{n\in \mathrm{ne}[u]} \|\boldsymbol{\Xi}\|_1\right) \leq \mu$$

which implies that $F_{\boldsymbol{w}}$ is a contraction map (w.r.t. $\|\cdot\|_1$) for any set of parameters $\boldsymbol{w}$.

2) *Nonlinear (nonpositional) GNN.* In this case, $h_{\boldsymbol{w}}$ is realized by a multilayered FNN. Since three-layered neural networks are universal approximators [67], $h_{\boldsymbol{w}}$ can approximate any desired function. However, not all the parameters $\boldsymbol{w}$ can be used, because it must be ensured that the corresponding transition function $F_{\boldsymbol{w}}$ is a contraction map. This can be achieved by adding a penalty term to (6), i.e.,

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

where the penalty term $L(y)$ is $(y-\mu)^2$ if $y > \mu$ and 0 otherwise, and the parameter $\mu \in (0,1)$ defines the desired contraction constant of $F_{\boldsymbol{w}}$. More generally, the penalty term can be any expression, differentiable w.r.t. $\boldsymbol{w}$, that is monotone increasing w.r.t. the norm of the Jacobian. For example, in our experiments, we use the penalty term $p_{\boldsymbol{w}} = \sum_{i=1}^{s} L(\|\boldsymbol{A}^i\|_1)$, where $\boldsymbol{A}^i$ is the ith column of $\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x}$. In fact, such an expression is an approximation of $L(\|\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x}\|_1) = L(\max_i \|\boldsymbol{A}^i\|_1)$.

E. A Comparison With Random Walks and Recursive Neural Networks

GNNs turn out to be an extension of other models already proposed in the literature. In particular, recursive neural networks [17] are a special case of GNNs, where:

1) the input graph is a directed acyclic graph;
2) the inputs of $f_{\boldsymbol{w}}$ are limited to $\boldsymbol{l}_n$ and $\boldsymbol{x}_{\mathrm{ch}[n]}$, where $\mathrm{ch}[n]$ is the set of children of n[8];
3) there is a *supersource* node sn from which all the other nodes can be reached. This node is typically used for output $\boldsymbol{o}_{sn}$ (graph-focused tasks).

The neural architectures, which have been suggested for realizing $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, include multilayered FNNs [17], [19], cascade correlation [68], and self-organizing maps [20], [69]. Note that the above constraints on the processed graphs and on the inputs of $f_{\boldsymbol{w}}$ exclude any sort of cyclic dependence of a state on itself. Thus, in the recursive neural network model, the encoding networks are FNNs. This assumption simplifies the computation of

[7]The 1-norm of a matrix $M = \{m_{i,j}\}$ is defined as $\|M\|_1 = \max_j \sum_i |m_{i,j}|$.

[8]A node u is child of n if there exists an arc from n to u. Obviously, $\mathrm{ch}[n] \subseteq \mathrm{ne}[n]$ holds.

TABLE II
TIME COMPLEXITY OF THE MOST EXPENSIVE INSTRUCTIONS OF THE LEARNING ALGORITHM. FOR EACH INSTRUCTION AND EACH GNN MODEL, A BOUND ON THE ORDER OF FLOATING POINT OPERATIONS IS GIVEN. THE TABLE ALSO DISPLAYS THE NUMBER OF TIMES PER EPOCH THAT EACH INSTRUCTION IS EXECUTED

instruction	positional	non−linear	linear	execs.
$\boldsymbol{z}(t+1)=\boldsymbol{z}(t)\cdot\boldsymbol{A}+\boldsymbol{b}$	$s^2\lvert\boldsymbol{E}\rvert$	$s^2\lvert\boldsymbol{E}\rvert$	$s^2\lvert\boldsymbol{E}\rvert$	$\mathrm{it_b}$
$\boldsymbol{o}=G_{\boldsymbol{w}}(\boldsymbol{x}(t),\boldsymbol{l}_{\boldsymbol{w}})$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_g$	1
$\boldsymbol{x}(t+1)=F_{\boldsymbol{w}}(\boldsymbol{x}(t),\boldsymbol{l})$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_f$	$\lvert\boldsymbol{E}\rvert\overrightarrow{C}_h$	$s^2\lvert\boldsymbol{E}\rvert$ $\lvert\boldsymbol{N}\rvert\overrightarrow{C}_\rho+\lvert\boldsymbol{E}\rvert\overrightarrow{C}_\phi$	$\mathrm{it_f}$ 1
$\boldsymbol{A}=\frac{\partial F_{\boldsymbol{w}}}{\partial\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{l})$	$s\lvert\boldsymbol{N}\rvert\overleftarrow{C}_f$	$s\lvert\boldsymbol{E}\rvert\overleftarrow{C}_h$	−	1
$\frac{\partial e_{\boldsymbol{w}}}{\partial\boldsymbol{o}}$	$\lvert\boldsymbol{N}\rvert$	$\lvert\boldsymbol{N}\rvert$	$\lvert\boldsymbol{N}\rvert$	1
$\frac{\partial p_{\boldsymbol{w}}}{\partial\boldsymbol{w}}$	$\mathrm{t}_{\boldsymbol{R}}\cdot\max(s^2\cdot\mathrm{hi}_f,\overleftarrow{C}_f)$	$\mathrm{t}_{\boldsymbol{R}}\cdot\max(s^2\cdot\mathrm{hi}_h,\overleftarrow{C}_h)$	−	1
$\boldsymbol{b}=\frac{\partial e_{\boldsymbol{w}}}{\partial\boldsymbol{o}}\frac{\partial G_{\boldsymbol{w}}}{\partial\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{l}_{\boldsymbol{N}})$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_g$	1
$\boldsymbol{c}=\frac{\partial e_{\boldsymbol{w}}}{\partial\boldsymbol{o}}\frac{\partial G_{\boldsymbol{w}}}{\partial\boldsymbol{w}}(\boldsymbol{x},\boldsymbol{l}_{\boldsymbol{N}})$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_g$	1
$\boldsymbol{d}=\boldsymbol{z}(t)\frac{\partial F_{\boldsymbol{w}}}{\partial\boldsymbol{w}}(\boldsymbol{x},\boldsymbol{l})$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_f$	$\lvert\boldsymbol{E}\rvert\overleftarrow{C}_h$	$\lvert\boldsymbol{N}\rvert\overleftarrow{C}_\rho+\lvert\boldsymbol{E}\rvert\overleftarrow{C}_\phi$	1

the states. In fact, the states can be computed following a predefined ordering that is induced by the partial ordering of the input graph.

Interestingly, the GNN model captures also the random walks on graphs when choosing $f_{\boldsymbol{w}}$ as a linear function. Random walks and, more generally, Markov chain models are useful in several application areas and have been recently used to develop ranking algorithms for internet search engines [18], [21]. In random walks on graphs, the state $\boldsymbol{x}_n$ associated with a node is a real value and is described by

$$\boldsymbol{x}_n=\sum_{i\in\mathrm{pa}[n]}a_{n,i}\boldsymbol{x}_i \tag{15}$$

where $\mathrm{pa}[n]$ is the set of parents of n, and $a_{n,i}\in\mathbb{R}$, $a_{n,i}\geq 0$ holds for each n,i. The $a_{n,i}$ are normalized so that $\sum_{i\in\mathrm{pa}[n]}a_{i,n}=1$. In fact, (15) can represent a random walker who is traveling on the graph. The value $a_{n,i}$ represents the probability that the walker, when visiting node n, decides to go to node i. The state $\boldsymbol{x}_n$ stands for the probability that the walker is on node n in the steady state. When all $\boldsymbol{x}_n$ are stacked into a vector $\boldsymbol{x}$, (15) becomes $\boldsymbol{x}=\boldsymbol{A}\boldsymbol{x}$ where $\boldsymbol{A}=\{a_{n,i}\}$ and $a_{n,i}$ is defined as in (15) if $i\in\mathrm{pa}[n]$ and $a_{n,i}=0$ otherwise. It is easily verified that $\|\boldsymbol{A}\|_1=1$. Markov chain theory suggests that if there exists t such that all the elements of the matrix $\boldsymbol{A}^t$ are nonnull, then (15) is a contraction map [70]. Thus, provided that the above condition on $\boldsymbol{A}$ holds, random walks on graphs are an instance of GNNs, where $\boldsymbol{A}$ is a constant stochastic matrix instead of being generated by neural networks.

III. COMPUTATIONAL COMPLEXITY ISSUES

In this section, an accurate analysis of the computational cost will be derived. The analysis will focus on three different GNN models: *positional GNNs*, where the functions f_w and g_w of (1) are implemented by FNNs; *linear (nonpositional) GNNs*; and *nonlinear (nonpositional) GNNs*.

First, we will describe with more details the most complex instructions involved in the learning procedure (see Table II). Then, the complexity of the learning algorithm will be defined. For the sake of simplicity, the cost is derived assuming that the training set contains just one graph $\boldsymbol{G}$. Such an assumption does not cause any loss of generality, since the graphs of the training set can always be merged into a single graph. The complexity is measured by the order of floating point operations.[9]

In Table II, the notation hi is used to denote the number of hidden-layer neurons. For example, hi_f indicates the number of hidden-layer neurons in the implementation of function f.

In the following, $\mathrm{it_l}$, $\mathrm{it_f}$, and $\mathrm{it_b}$ denote the number of epochs, the mean number of forward iterations (of the repeat cycle in function FORWARD), and the mean number of backward iterations (of the repeat cycle in function BACKWARD), respectively. Moreover, we will assume that there exist two procedures FP and BP, which implement the forward phase and the backward phase of the backpropagation procedure [71], respectively. Formally, given a function $l_{\boldsymbol{w}}:\mathbb{R}^a\rightarrow\mathbb{R}^b$ implemented by an FNN, we have

$$\begin{aligned}l_{\boldsymbol{w}}(\boldsymbol{y})&=\mathrm{FP}(l_{\boldsymbol{w}},\boldsymbol{y})\\ \left[\delta\frac{\partial l_{\boldsymbol{w}}}{\partial\boldsymbol{w}}(\boldsymbol{y}),\delta\frac{\partial l_{\boldsymbol{w}}}{\partial\boldsymbol{y}}(\boldsymbol{y})\right]&=\mathrm{BP}(l_{\boldsymbol{w}},\boldsymbol{y},\delta)\,.\end{aligned}$$

Here, $\boldsymbol{y}\in\mathbb{R}^a$ is the input vector and the row vector $\delta\in\mathbb{R}^b$ is a signal that suggests how the network output must be adjusted to improve the cost function. In most applications, the cost function is $e_{\boldsymbol{w}}(\boldsymbol{y})=(\boldsymbol{t}-\boldsymbol{y})^2$ and $\delta=(\partial e_{\boldsymbol{w}}/\partial\boldsymbol{o})(\boldsymbol{y})=2(\boldsymbol{t}-\boldsymbol{o})$, where $\boldsymbol{o}=l_{\boldsymbol{w}}(\boldsymbol{y})$ and $\boldsymbol{t}$ is the vector of the desired output corresponding to input $\boldsymbol{y}$. On the other hand, $\delta(\partial l_{\boldsymbol{w}}/\partial\boldsymbol{y})(\boldsymbol{y})$ is the gradient of $e_{\boldsymbol{w}}$ w.r.t. the network input and is easily computed

[9]According to the common definition of time complexity, an algorithm requires $O(l(a))$ operations, if there exist $\alpha>0,\bar{a}\geq 0$, such that $c(a)\leq\alpha\,l(a)$ holds for each $a\geq\bar{a}$, where $c(a)$ is the maximal number of operations executed by the algorithm when the length of input is a.

as a side product of backpropagation.[10] Finally, $\overrightarrow{C}_l$ and $\overleftarrow{C}_l$ denote the computational complexity required by the application of FP and BP on $l_{\boldsymbol{w}}$, respectively. For example, if $l_{\boldsymbol{w}}$ is implemented by a multilayered FNN with a inputs, b hidden neurons, and c outputs, then $\overrightarrow{C}_l = \overleftarrow{C}_l = O(ab + ac)$ holds.

A. Complexity of Instructions

1) Instructions $\boldsymbol{z}(t+1) = \boldsymbol{z}(t) \cdot \boldsymbol{A} + \boldsymbol{b}$, $\boldsymbol{o} = G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$, and $\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l})$: Since $\boldsymbol{A}$ is a matrix having at most $s^2|\boldsymbol{E}|$ nonnull elements, the multiplication of $\boldsymbol{z}(t)$ by $\boldsymbol{A}$, and as a consequence, the instruction $\boldsymbol{z}(t+1) = \boldsymbol{z}(t) \cdot \boldsymbol{A} + \boldsymbol{b}$, costs $O(s^2|\boldsymbol{E}|)$ floating points operations. Moreover, the state $\boldsymbol{x}(t+1)$ and the output vector $\boldsymbol{o}$ are calculated by applying the local transition function and the local output function to each node n. Thus, in positional GNNs and in nonlinear GNNs, where $f_{\boldsymbol{w}}$, $h_{\boldsymbol{w}}$, and $g_{\boldsymbol{w}}$ are directly implemented by FNNs, $\boldsymbol{x}(t+1)$ and $\boldsymbol{o}$ are computed by running the forward phase of backpropagation once for each node or edge (see Table II).

On the other hand, in linear GNNs, $\boldsymbol{x}_n(t)$ is calculated in two steps: the matrices $\boldsymbol{A}_n$ of (13) and the vectors $\boldsymbol{b}_n$(14) are evaluated; then, $\boldsymbol{x}_n(t)$ is computed. The former phase, the cost of which is $O(|\boldsymbol{E}|\overrightarrow{C}_\phi + |\boldsymbol{N}|\overrightarrow{C}_\rho)$, is executed once for each epoch, whereas the latter phase, the cost of which is $O(s^2|\boldsymbol{E}|)$, is executed at every step of the cycle in the function FORWARD.

2) Instruction $\boldsymbol{A} = (\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$: This instruction requires the computation of the Jacobian of $F_{\boldsymbol{w}}$. Note that $\boldsymbol{A} = \{\boldsymbol{A}_{n,u}\}$ is a block matrix where the block $\boldsymbol{A}_{n,u}$ measures the effect of node u on node n, if there is an arc (n,u) from u to n, and is null otherwise. In the linear model, the matrices $\boldsymbol{A}_{n,u}$ correspond to those displayed in (13) and used to calculate $\boldsymbol{x}(t)$ in the forward phase. Thus, such an instruction has no cost in the backward phase in linear GNNs.

In nonlinear GNNs, $\boldsymbol{A}_{n,u} = (\partial h_{\boldsymbol{w}}/\partial \boldsymbol{x}_n)(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u)$, is computed by appropriately exploiting the backpropagation procedure. More precisely, let $\boldsymbol{q}_i \in \mathbb{R}^s$ be a vector where all the components are zero except for the ith one, which equals one, i.e., $\boldsymbol{q}_1 = [1, 0, \ldots, 0]$, $\boldsymbol{q}_2 = [0, 1, 0, \ldots, 0]$, and so on. Note that BP, when it is applied to $l_{\boldsymbol{w}}$ with $\delta = \boldsymbol{b}_i$, returns $\boldsymbol{A}^i_{n,u} = \boldsymbol{q}_i(\partial l_{\boldsymbol{w}}/\partial \boldsymbol{y})(\boldsymbol{y})$, i.e., the ith column of the Jacobian $(\partial l_{\boldsymbol{w}}/\partial \boldsymbol{y})(\boldsymbol{y})$. Thus, $\boldsymbol{A}_{n,u}$ can be computed by applying BP on all the $\boldsymbol{q}_i$, i.e.,

$$\boldsymbol{A}_{n,u} = [\boldsymbol{A}^1_{n,u}, \ldots, \boldsymbol{A}^s_{n,u}] \quad \boldsymbol{A}^i_{n,u} = \mathrm{BP}_2(h_{\boldsymbol{w}}, \boldsymbol{y}, \boldsymbol{q}_i) \tag{16}$$

where BP_2 indicates that we are considering only the first component of the output of BP. A similar reasoning can also be used with positional GNNs. The complexity of these procedures is easily derived and is displayed in the fourth row of Table II.

3) Computation of $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o}$ and $\partial p_{\boldsymbol{w}}/\partial \boldsymbol{w}$: In linear GNNs, the cost function is $e_{\boldsymbol{w}} = \sum_{i=1}^{q}(\boldsymbol{t}_i - \varphi_{\boldsymbol{w}}(\boldsymbol{G}, n_i))^2$, and, as a consequence, $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o}_k = 2(\boldsymbol{t}_k - \boldsymbol{o}_{n_k})$, if n_k is a node belonging to the training set, and 0 otherwise. Thus, $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o}$ is easily calculated by $O(|\boldsymbol{N}|)$ operations.

In positional and nonlinear GNNs, a penalty term $p_{\boldsymbol{w}}$ is added to the cost function to force the transition function to be a contraction map. In this case, it is necessary to compute $\partial p_{\boldsymbol{w}}/\partial \boldsymbol{w}$, because such a vector must be added to the gradient. Let $\boldsymbol{A}^{i,j}_{n,u}$ denote the element in position i, j of the block $\boldsymbol{A}_{n,u}$. According to the definition of $p_{\boldsymbol{w}}$, we have

$$p_{\boldsymbol{w}} = \sum_{u \in \boldsymbol{N}} \sum_{j=1}^{s} L\left(\sum_{(n,u) \in \boldsymbol{E}} \sum_{i=1}^{s} |\boldsymbol{A}^{i,j}_{n,u}| - \mu \right) = \sum_{u \in \boldsymbol{N}} \sum_{j=1}^{s} \alpha_{u,j}$$

where $\alpha_{u,j} = \sum_{(n,u) \in \boldsymbol{E}} \sum_{i=1}^{s} |\boldsymbol{A}^{i,j}_{n,u}| - \mu$, if the sum is larger than 0, and it is 0 otherwise. It follows:

$$\begin{aligned} \frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}} &= 2 \sum_{u \in \boldsymbol{N}} \sum_{j=1}^{s} \alpha_{u,j} \sum_{(n,u) \in \boldsymbol{E}} \sum_{i=1}^{s} \mathrm{sgn}(\boldsymbol{A}^{i,j}_{n,u}) \cdot \frac{\partial \boldsymbol{A}^{i,j}_{n,u}}{\partial \boldsymbol{w}} \\ &= 2 \sum_{u \in \boldsymbol{N}} \sum_{(n,u) \in \boldsymbol{E}} \sum_{j=1}^{s} \sum_{i=1}^{s} \alpha_{u,j} \cdot \mathrm{sgn}(\boldsymbol{A}^{i,j}_{n,u}) \cdot \frac{\partial \boldsymbol{A}^{i,j}_{n,u}}{\partial \boldsymbol{w}} \end{aligned}$$

where sgn is the sign function. Moreover, let $\boldsymbol{R}_{n,u}$ be a matrix whose element in position i, j is $\alpha_{u,j} \cdot \mathrm{sgn}(\boldsymbol{A}^{i,j}_{n,u})$ and let vec be the operator that takes a matrix and produce a column vector by stacking all its columns one on top of the other. Then

$$\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = 2 \sum_{u \in \boldsymbol{N}} \sum_{(n,u) \in \boldsymbol{E}} (\mathrm{vec}(\boldsymbol{R}_{n,u}))' \cdot \frac{\partial \,\mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \boldsymbol{w}} \tag{17}$$

holds. The vector $\partial\,\mathrm{vec}(\boldsymbol{A}_{n,u})/\partial \boldsymbol{w}$ depends on selected implementation of $h_{\boldsymbol{w}}$ or $f_{\boldsymbol{w}}$. For sake of simplicity, let us restrict our attention to nonlinear GNNs and assume that the transition network is a three-layered FNN. σ_j, $\boldsymbol{a}_j$, $\boldsymbol{V}_j$, and $\boldsymbol{t}_j$ are the activation function[11], the vector of the activation levels, the matrix of the weights, and the thresholds of the jth layer, respectively. The following reasoning can also be extended to positional GNNs and networks with a different number of layers. The function $h_{\boldsymbol{w}}$ is formally defined in terms of σ_j, $\boldsymbol{a}_j$, $\boldsymbol{V}_j$, and $\boldsymbol{t}_j$

$$\begin{aligned} \boldsymbol{a}_1 &= [\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u] \\ \boldsymbol{a}_2 &= \boldsymbol{V}_1\, \boldsymbol{a}_1 + \boldsymbol{t}_1 \\ \boldsymbol{a}_3 &= \boldsymbol{V}_2\, \sigma_2(\boldsymbol{a}_2) + \boldsymbol{t}_2 \\ h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) &= \sigma_3(\boldsymbol{a}_3). \end{aligned}$$

By the chain differentiation rule, it follows:

$$\begin{aligned} \mathrm{vec}(\boldsymbol{A}_{n,u}) &= \mathrm{vec}\left(\frac{\partial h_{\boldsymbol{w}}}{\partial \boldsymbol{x}_u}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) \right) \\ &= \mathrm{vec}\left(\mathrm{diag}(\sigma_3'(\boldsymbol{a}_3)) \cdot \boldsymbol{V}_2 \cdot \mathrm{diag}(\sigma_2'(\boldsymbol{a}_2)) \cdot \overline{\boldsymbol{V}}_1 \right) \end{aligned}$$

where σ_j' is the derivative of σ_j, diag is an operator that transforms a vector into a diagonal matrix having such a vector as diagonal, and $\overline{\boldsymbol{V}}_1$ is the submatrix of $\boldsymbol{V}_1$ that contains only the weights that connect the inputs corresponding to $\boldsymbol{x}_u$ to the hidden layer. The parameters $\boldsymbol{w}$ affect four components of

[10] Backpropagation computes for each neuron v the *delta* value $(\partial e_{\boldsymbol{w}}/\partial a_v)(\boldsymbol{y}) = \delta(\partial l_{\boldsymbol{w}}/\partial a_v)(\boldsymbol{y})$, where $e_{\boldsymbol{w}}$ is the cost function and a_v the activation level of neuron v. Thus, $\delta(\partial l_{\boldsymbol{w}}/\partial \boldsymbol{y})(\boldsymbol{y})$ is just a vector stacking all the delta values of the input neurons.

[11] σ_j is a vectorial function that takes as input the vector of the activation levels of neurons in a layer and returns the vector of the outputs of the neurons of the same layer.

$\mathrm{vec}(\boldsymbol{A}_{n,u})$, i.e., $\boldsymbol{a}_3, \boldsymbol{V}_2, \boldsymbol{a}_2$, and $\overline{\boldsymbol{V}}_1$. By properties of derivatives for matrix products and the chain rule

$$\begin{aligned}(\mathrm{vec}(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \boldsymbol{w}} = \\ = (\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \sigma_3'(\boldsymbol{a}_3)} \cdot \frac{\partial \sigma_3'(\boldsymbol{a}_3)}{\partial \boldsymbol{w}} \\ + (\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \mathrm{vec}(\boldsymbol{V}_2)} \cdot \frac{\partial \mathrm{vec}(\boldsymbol{V}_2)}{\partial \boldsymbol{w}} \\ + (\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \sigma_2'(\boldsymbol{a}_2)} \cdot \frac{\partial \sigma_2'(\boldsymbol{a}_2)}{\partial \boldsymbol{w}} \\ + (\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \mathrm{vec}(\overline{\boldsymbol{V}}_1)} \cdot \frac{\partial \mathrm{vec}(\overline{\boldsymbol{V}}_1)}{\partial \boldsymbol{w}} \end{aligned} \quad (18)$$

holds.

Thus, $(\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \partial \mathrm{vec}(\boldsymbol{A}_{n,u})/\partial \boldsymbol{w}$ is the sum of four contributions. In order to derive a method to compute those terms, let $\boldsymbol{I}_a$ denote the $a \times a$ identity matrix. Let $\otimes$ be the Kronecker product and suppose that $\boldsymbol{P}_a$ is a $a^2 \times a$ matrix such that $\mathrm{vec}(\mathrm{diag}(\boldsymbol{v})) = \boldsymbol{P}_a \boldsymbol{v}$ for any vector $\boldsymbol{v} \in \mathbb{R}^a$. By the Kronecker product properties, $\mathrm{vec}(\boldsymbol{ABI}) = (\boldsymbol{B}' \otimes \boldsymbol{I}_a) \cdot \mathrm{vec}(\boldsymbol{A})$ holds for matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{I}_a$ having compatible dimensions [72]. Thus, we have

$$\mathrm{vec}(\boldsymbol{A}_{n,u}) = \left((\boldsymbol{V}_2 \cdot \mathrm{diag}(\sigma_2'(\boldsymbol{a}_2)) \cdot \boldsymbol{V}_1)' \otimes \boldsymbol{I}_s\right) \cdot \boldsymbol{P}_s \cdot \sigma_3'(\boldsymbol{a}_3)$$

which implies

$$\frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \sigma_3'(\boldsymbol{a}_3)} = \left((\boldsymbol{V}_2 \cdot \mathrm{diag}(\sigma_2'(\boldsymbol{a}_2)) \cdot \boldsymbol{V}_1)' \otimes \boldsymbol{I}_s\right) \cdot \boldsymbol{P}_s\,.$$

Similarly, using the properties $\mathrm{vec}(\boldsymbol{ABC}) = (\boldsymbol{C}' \otimes \boldsymbol{A}) \cdot \mathrm{vec}(\boldsymbol{B})$ and $\mathrm{vec}(\boldsymbol{AB}) = (\boldsymbol{I}_a \otimes \boldsymbol{A}) \cdot \mathrm{vec}(\boldsymbol{B})$, it follows:

$$\begin{aligned}\frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \mathrm{vec}(\boldsymbol{V}_2)} &= (\mathrm{diag}(\sigma_2'(\boldsymbol{a}_2)) \cdot \boldsymbol{V}_1)' \otimes \mathrm{diag}(\sigma_3'(\boldsymbol{a}_3)) \\ \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial\, \sigma_2'(\boldsymbol{a}_2)} &= (\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \left(\boldsymbol{V}_1' \otimes (\mathrm{diag}(\sigma_3'(\boldsymbol{a}_3)) \cdot \boldsymbol{V}_2)\right) \cdot \boldsymbol{P}_{d_h} \\ \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \mathrm{vec}(\overline{\boldsymbol{V}}_1)} &= (\boldsymbol{I}_s \otimes (\mathrm{diag}(\sigma_3'(\boldsymbol{a}_3)) \cdot \boldsymbol{V}_2 \cdot \mathrm{diag}(\sigma_2'(\boldsymbol{a}_2))))\end{aligned}$$

where d_h is the number of hidden neurons. Then, we have

$$\begin{aligned}(\mathrm{vec}(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \sigma_3'(\boldsymbol{a}_3)} \cdot \frac{\partial \sigma_3'(\boldsymbol{a}_3)}{\partial \boldsymbol{w}} \\ = \left(\mathrm{vec}\left(\boldsymbol{R}_{u,v} \cdot \overline{\boldsymbol{V}}_1' \cdot \mathrm{diag}(\sigma_2'(\boldsymbol{a}_1)) \cdot \boldsymbol{V}_2'\right)\right)' \cdot \boldsymbol{P}_s \cdot \frac{\partial \sigma_3'(\boldsymbol{a}_3)}{\partial \boldsymbol{w}}\end{aligned} \quad (19)$$

$$\begin{aligned}(\mathrm{vec}(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \mathrm{vec}(\boldsymbol{V}_2)} \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \boldsymbol{w}} \\ = \left(\mathrm{vec}\left(\mathrm{diag}(\sigma_3'(\boldsymbol{a}_3)) \cdot \boldsymbol{R}_{u,v} \cdot \overline{\boldsymbol{V}}_1' \cdot \mathrm{diag}(\sigma_2'(\boldsymbol{a}_2))\right)\right)' \\ \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \boldsymbol{w}}\end{aligned} \quad (20)$$

$$\begin{aligned}(\mathrm{vec}(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial\, \sigma_2'(\boldsymbol{a}_2)} \cdot \frac{\partial\, \sigma_2'(\boldsymbol{a}_2)}{\partial \boldsymbol{w}} \\ = \left(\mathrm{vec}\left(\boldsymbol{V}_2' \cdot \mathrm{diag}(\sigma_3'(\boldsymbol{a}_3)) \cdot \boldsymbol{R}_{u,v} \cdot \boldsymbol{V}_1'\right)\right)' \cdot \boldsymbol{P}_{d_h} \cdot \frac{\partial\, \sigma_2'(\boldsymbol{a}_2)}{\partial \boldsymbol{w}}\end{aligned} \quad (21)$$

$$\begin{aligned}(\mathrm{vec}(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \mathrm{vec}(\overline{\boldsymbol{V}}_1)} \cdot \frac{\partial \mathrm{vec}(\overline{\boldsymbol{V}}_1)}{\partial \boldsymbol{w}} \\ = \left(\mathrm{vec}\left(\mathrm{diag}(\sigma_2'(\boldsymbol{a}_2)) \cdot \boldsymbol{V}_2' \cdot \mathrm{diag}(\sigma_3'(\boldsymbol{a}_3)) \cdot \boldsymbol{R}_{u,v}\right)\right)' \\ \cdot \frac{\partial \mathrm{vec}(\overline{\boldsymbol{V}}_1)}{\partial \boldsymbol{w}}\end{aligned} \quad (22)$$

where the mentioned Kronecker product properties have been used.

It follows that $(\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \partial \mathrm{vec}(\boldsymbol{A}_{n,u})/\partial \boldsymbol{w}$ can be written as the sum of the four contributions represented by (19)–(22). The second and the fourth term [(20) and (22)] can be computed directly using the corresponding formulas. The first one can be calculated by observing that $\sigma_3'(\boldsymbol{a}_3)$ looks like the function computed by a three-layered FNN that is the same as $h_{\boldsymbol{w}}$ except for the activation function of the last layer. In fact, if we denote by $\bar{h}_{\boldsymbol{w}}$ such a network, then

$$(\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \sigma_3'(\boldsymbol{a}_3)} \frac{\partial \sigma_3'(\boldsymbol{a}_3)}{\partial \boldsymbol{w}} = \mathrm{BP}_1(\bar{h}_{\boldsymbol{w}}, \boldsymbol{a}_1, \delta) \quad (23)$$

holds, where $\delta = (\mathrm{vec}(\boldsymbol{R}_{u,v}))' \cdot \partial \mathrm{vec}(\boldsymbol{A}_{n,u})/\partial \sigma_3'(\boldsymbol{a}_3)$. A similar reasoning can be applied also to the third contribution.

The above described method includes two tasks: the matrix multiplications of (19)–(22) and the backpropagation as defined by (23). The former task consists of several matrix multiplications. By inspection of (19)–(22), the number of floating point operations is approximately estimated as $2s^2 + 12s \cdot \mathrm{hi}_h + 10s^2 \cdot \mathrm{hi}_h$,[12] where hi_h denotes the number of hidden-layer neurons implementing the function h. The second task has approximately the same cost as a backpropagation phase through the original function $h_{\boldsymbol{w}}$.

Thus, the complexity of computing $\partial p_{\boldsymbol{w}}/\partial \boldsymbol{w}$ is $O(|\boldsymbol{E}| \max(s^2 \cdot \mathrm{hi}_h, \overleftarrow{C}_h))$. Note, however, that even if the sum in (17) ranges over all the arcs of the graph, only those arcs (n, u) such that $\boldsymbol{R}_{n,u} \neq \boldsymbol{0}$ have to be considered. In practice, $\boldsymbol{R}_{n,u} \neq \boldsymbol{0}$ is a rare event, since it happens only when the columns of the Jacobian are larger than μ and a penalty function was used to limit the occurrence of these cases. As a consequence, a better estimate of the complexity of computing $\partial p_{\boldsymbol{w}}/\partial \boldsymbol{w}$ is $O(\mathrm{t}_{\boldsymbol{R}} \cdot \max(s^2 \cdot \mathrm{hi}_h, \overleftarrow{C}_h))$, where $\mathrm{t}_{\boldsymbol{R}}$ is the average number of nodes u such that $\boldsymbol{R}_{n,u} \neq \boldsymbol{0}$ holds for some n.

4) Instructions $\boldsymbol{b} = (\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o})(\partial G_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l_N})$ and $\boldsymbol{c} = (\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o})(\partial G_{\boldsymbol{w}}/\partial \boldsymbol{w})(\boldsymbol{x}, \boldsymbol{l_N})$: The terms $\boldsymbol{b}$ and $\boldsymbol{c}$ can be calculated by the backpropagation of $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o}$ through the network that implements $g_{\boldsymbol{w}}$. Since such an operation must be repeated for each node, the time complexity of instructions $\boldsymbol{b} = (\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o})(\partial G_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l_N})$ and $\boldsymbol{c} = (\partial e_{\boldsymbol{w}}/\partial \boldsymbol{o})(\partial G_{\boldsymbol{w}}/\partial \boldsymbol{w})(\boldsymbol{x}, \boldsymbol{l_N})$ is $O(|\boldsymbol{N}|\overleftarrow{C}_g)$ for all the GNN models.

[12]Such a value is obtained by considering the following observations: for an $a \times b$ matrix $\boldsymbol{C}$ and $b \times c$ matrix $\boldsymbol{D}$, the multiplication $\boldsymbol{CD}$ requires approximately $2abc$ operations; more precisely, abc multiplications and $ac(b-1)$ sums. If $\boldsymbol{D}$ is a diagonal $b \times b$ matrix, then $\boldsymbol{CD}$ requires $2ab$ operations. Moreover, if $\boldsymbol{C}$ is an $a \times b$ matrix, $\boldsymbol{D}$ is a $b \times a$ matrix, and $\boldsymbol{P}_a$ is the $a^2 \times a$ matrix defined above and used in (19)–(22), then computing $\mathrm{vec}(\boldsymbol{CD})\boldsymbol{P}_c$ costs only $2ab$ operations provided that a sparse representation is used for P_α. Finally, $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3$ are already available, since they are computed during the forward phase of the learning algorithm.

5) Instruction $\boldsymbol{d} = \boldsymbol{z}(t)(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{w})(\boldsymbol{x}, \boldsymbol{l})$*:* By definition of $F_{\boldsymbol{w}}$, $f_{\boldsymbol{w}}$, and BP, we have

$$\boldsymbol{z}(t) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) = \sum_{n \in \boldsymbol{N}} \boldsymbol{z}_n(t) \cdot \frac{\partial f_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{\mathrm{co}[n]}, \boldsymbol{x}_u, \boldsymbol{l}_{\mathrm{ne}[n]}) = \sum_{n \in \boldsymbol{N}} \mathrm{BP}_1(f_{\boldsymbol{w}}, \boldsymbol{y}, \boldsymbol{z}_n(t)) \tag{24}$$

where $\boldsymbol{y} = [\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u]$ and BP_1 indicates that we are considering only the first part of the output of BP. Similarly

$$\boldsymbol{z}(t) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) = \sum_{n \in \boldsymbol{N}} \sum_{u \in \mathrm{ne}[n]} \boldsymbol{z}_n(t) \cdot \frac{\partial h_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) = \sum_{n \in \boldsymbol{N}} \sum_{u \in \mathrm{ne}[n]} \mathrm{BP}_1(h_{\boldsymbol{w}}, \boldsymbol{y}, \boldsymbol{z}_n(t)) \tag{25}$$

where $\boldsymbol{y} = [\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u]$. Thus, (24) and (25) provide a direct method to compute $\boldsymbol{d}$ in positional and nonlinear GNNs, respectively.

For linear GNNs, let $h^i_{\boldsymbol{w}}$ denote the ith output of $h_{\boldsymbol{w}}$ and note that

$$\begin{aligned} & h^i_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, l_u) \\ & \quad = \boldsymbol{b}^i_u + \sum_{j=1}^{s} \boldsymbol{A}^{i,j}_{n,u} \boldsymbol{x}^i_u \\ & \quad = \rho^i_u(\boldsymbol{l}_n) + \frac{\mu}{s|\mathrm{ne}[u]|} \cdot \sum_{j=1}^{s} \boldsymbol{x}^j_u \cdot \phi^{i,j}_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u) \end{aligned}$$

holds. Here, $\boldsymbol{A}^{i,j}_{n,u}$ and $\phi^{i,j}_{\boldsymbol{w}}$ are the element in position i, j of matrix $\boldsymbol{A}_{n,u}$ and the corresponding output of the transition network [see (13)], respectively, while $\boldsymbol{b}^i_u$ is the ith element of vector $\boldsymbol{b}^i_u$, ρ^i_u is the corresponding output of the forcing network [see (14)], and $\boldsymbol{x}^i_u$ is the ith element of $\boldsymbol{x}_u$. Then

$$\begin{aligned} & \boldsymbol{z}(t) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \\ & \quad = \sum_{n \in \boldsymbol{N}} \sum_{u \in \mathrm{ne}[n]} \boldsymbol{z}_n(t) \cdot \frac{\partial h_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, l_u) \\ & \quad = \sum_{n \in \boldsymbol{N}} \sum_{u \in \mathrm{ne}[n]} \sum_{i=1}^{s} \boldsymbol{z}^i_n(t) \cdot \frac{\partial h^i_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) \\ & \quad = \sum_{n \in \boldsymbol{N}} \mathrm{BP}_1(\rho_{\boldsymbol{w}}, \boldsymbol{y}, \delta) + \sum_{n \in \boldsymbol{N}} \sum_{u \in \mathrm{ne}[n]} \mathrm{BP}_1(\phi_{\boldsymbol{w}}, \bar{\boldsymbol{y}}, \bar{\delta}) \end{aligned}$$

where $\boldsymbol{y} = \boldsymbol{l}_n$, $\bar{\boldsymbol{y}} = [\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u]$, $\delta = |\mathrm{ne}[n]| \cdot \boldsymbol{z}'(t)$, and $\bar{\delta}$ is a vector that stores $\boldsymbol{z}^i_n(t) \cdot \mu/s|\mathrm{ne}[u]| \cdot \boldsymbol{x}^j_u$ in the position corresponding to i, j, that is, $\bar{\delta} = (\mu/s|\mathrm{ne}[u]|)\mathrm{vec}(\boldsymbol{z}_n(t) \cdot \boldsymbol{x}'_u)$. Thus, in linear GNNs, $\boldsymbol{d}$ is computed by calling the backpropagation procedure on each arc and node.

B. Time Complexity of the GNN Model

According to our experiments, the application of a trained GNN on a graph (test phase) is relatively fast even for large graphs. Formally, the complexity is easily derived from Table II and it is $O(|\boldsymbol{N}|\overrightarrow{C}_g + \mathrm{it_f} \cdot |\boldsymbol{N}|\overrightarrow{C}_f)$ for positional GNNs, $O(|\boldsymbol{N}|\overrightarrow{C}_g + \mathrm{it_f} \cdot |\boldsymbol{E}|\overrightarrow{C}_h)$ for nonlinear GNNs, and $O(|\boldsymbol{N}|\overrightarrow{C}_g + \mathrm{it_f} \cdot |\boldsymbol{E}|s^2 + |\boldsymbol{N}|\overrightarrow{C}_\rho + |\boldsymbol{E}|\overrightarrow{C}_\phi)$ for linear GNNs. In practice, the cost of the test phase is mainly due to the repeated computation of the state $\boldsymbol{x}(t)$. The cost of each iteration is linear both w.r.t. the dimension of the input graph (the number of edges), the dimension of the employed FNNs and the state, with the only exception of linear GNNs, whose single iteration cost is quadratic w.r.t. to the state. The number of iterations required for the convergence of the state depends on the problem at hand, but Banach's theorem ensures that the convergence is exponentially fast and experiments have shown that 5–15 iterations are generally sufficient to approximate the fixed point.

In positional and nonlinear GNNs, the transition function must be activated $\mathrm{it_f} \cdot |\boldsymbol{N}|$ and $\mathrm{it_f} \cdot |\boldsymbol{E}|$ times, respectively. Even if such a difference may appear significant, in practice, the complexity of the two models is similar, because the network that implements the $f_{\boldsymbol{w}}$ is larger than the one that implements $h_{\boldsymbol{w}}$. In fact, $f_{\boldsymbol{w}}$ has $M(s + l_E)$ input neurons, where M is the maximum number of neighbors for a node, whereas $h_{\boldsymbol{w}}$ has only $s + l_E$ input neurons. An appreciable difference can be noticed only for graphs where the number of neighbors of nodes is highly variable, since the inputs of $f_{\boldsymbol{w}}$ must be sufficient to accommodate the maximal number of neighbors and many inputs may remain unused when $f_{\boldsymbol{w}}$ is applied. On the other hand, it is observed that in the linear model the FNNs are used only once for each iteration, so that the complexity of each iteration is $O(s^2|\boldsymbol{E}|)$ instead of $O(|\boldsymbol{E}|\overrightarrow{C}_h)$. Note that $\overrightarrow{C}_h = O((s + l_E + 2l_N) \cdot \mathrm{hi}_h) = O(s \cdot \mathrm{hi}_h)$ holds, when $h_{\boldsymbol{w}}$ is implemented by a three-layered FNN with hi_h hidden neurons. In practical cases, where hi_h is often larger than s, the linear model is faster than the nonlinear model. As confirmed by the experiments, such an advantage is mitigated by the smaller accuracy that the model usually achieves.

In GNNs, the learning phase requires much more time than the test phase, mainly due to the repetition of the forward and backward phases for several epochs. The experiments have shown that the time spent in the forward and backward phases is not very different. Similarly to the forward phase, the cost of function BACKWARD is mainly due to the repetition of the instruction that computes $\boldsymbol{z}(t)$. Theorem 2 ensures that $\boldsymbol{z}(t)$ converges exponentially fast and the experiments confirmed that $\mathrm{it_b}$ is usually a small number.

Formally, the cost of each learning epoch is given by the sum of all the instructions times the iterations in Table II. An inspection of Table II shows that the cost of all instructions involved in the learning phase are linear both with respect to the dimension of the input graph and of the FNNs. The only exceptions are due to the computation of $\boldsymbol{z}(t+1) = \boldsymbol{z}(t) \cdot \boldsymbol{A} + \boldsymbol{b}$, $(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$ and $\partial p_{\boldsymbol{w}}/\boldsymbol{w}$, which depend quadratically on s.

The most expensive instruction is apparently the computation of $\partial p_{\boldsymbol{w}}/\boldsymbol{w}$ in nonlinear GNNs, which costs $O(\mathrm{t}_{\boldsymbol{R}} \cdot \max(s^2 \cdot \mathrm{hi}_h, \overleftarrow{C}_h))$. On the other hand, the experiments have shown that usually $\mathrm{t}_{\boldsymbol{R}}$ is a small number. In most epochs, $\mathrm{t}_{\boldsymbol{R}}$ is 0, since the Jacobian does not violate the imposed constraint, and in the other cases, $\mathrm{t}_{\boldsymbol{R}}$ is usually in the range 1–5. Thus, for a

small state dimension s, the computation of $\partial p_{\boldsymbol{w}}/\boldsymbol{w}$ requires few applications of backpropagation on h and has a small impact on the global complexity of the learning process. On the other hand, in theory, if s is very large, it might happen that $s^2 \cdot \mathrm{hi}_h \gg \overleftarrow{C}_h \approx (s + l_E + 2l_N) \cdot \mathrm{hi}_h$ and at the same time $\mathrm{t}_{\boldsymbol{R}} \gg 0$, causing the computation of the gradient to be very slow. However, it is worth mentioning that this case was never observed in our experiments.

IV. Experimental Results

In this section, we present the experimental results, obtained on a set of simple problems carried out to study the properties of the GNN model and to prove that the method can be applied to relevant applications in relational domains. The problems that we consider, viz., the subgraph matching, the mutagenesis, and the web page ranking, have been selected since they are particularly suited to discover the properties of the model and are correlated to important real-world applications. From a practical point of view, we will see that the results obtained on some parts of mutagenesis data sets are among the best that are currently reported in the open literature (please see detailed comparison in Section IV-B). Moreover, the subgraph matching problem is relevant to several application domains. Even if the performance of our method is not comparable in terms of best accuracy on the same problem with the most efficient algorithms in the literature, the proposed approach is a very general technique that can be applied on extension of the subgraph matching problems [73]–[75]. Finally, the web page ranking is an interesting problem, since it is important in information retrieval and very few techniques have been proposed for its solution [76]. It is worth mentioning that the GNN model has been already successfully applied on larger applications, which include image classification and object localization in images [77], [78], web page ranking [79], relational learning [80], and XML classification [81].

The following facts hold for each experiment, unless otherwise specified. The experiments have been carried out with both linear and nonlinear GNNs. According to existing results on recursive neural networks, the nonpositional transition function slightly outperforms the positional ones, hence, currently only nonpositional GNNs have been implemented and tested. Both the (nonpositional) linear and the nonlinear model were tested. All the functions involved in the two models, i.e., $g_{\boldsymbol{w}}$, $\phi_{\boldsymbol{w}}$, and $\rho_{\boldsymbol{w}}$ for linear GNNs, and $g_{\boldsymbol{w}}$ and $h_{\boldsymbol{w}}$ for nonlinear GNNs were implemented by three-layered FNNs with sigmoidal activation functions. The presented results were averaged over five different runs. In each run, the data set was a collection of random graphs constructed by the following procedure: each pair of nodes was connected with a certain probability δ; the resulting graph was checked to verify whether it was connected and if it was not, random edges were inserted until the condition was satisfied.

The data set was split into a training set, a validation set, and a test set and the validation set was used to avoid possible issues with overfitting. For the problems where the original data is only one single big graph $\boldsymbol{G}$, a training set, a validation set, and a test set include different supervised nodes of $\boldsymbol{G}$. Otherwise, when several graphs were available, all the patterns of a graph $\boldsymbol{G}_i$ were assigned to only one set. In every trial, the training procedure performed at most 5000 epochs and every 20 epochs the GNN was evaluated on the validation set. The GNN that achieved the lowest cost on the validation set was considered the best model and was applied to the test set.

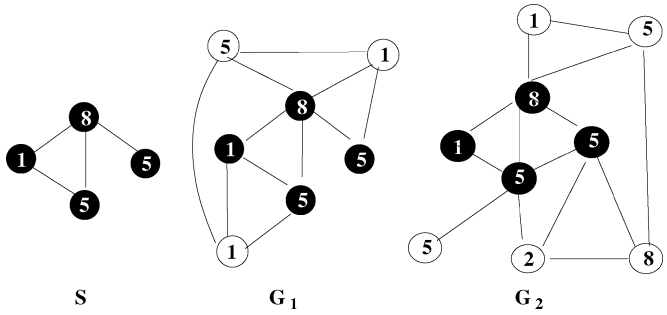

Fig. 4. Two graphs $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ that contain a subgraph $\boldsymbol{S}$. The numbers inside the nodes represent the labels. The function τ to be learned is $\tau(\boldsymbol{G}_i, n_{i,j}) = 1$, if $n_{i,j}$ is a black node, and $\tau(\boldsymbol{G}_i, n_{i,j}) = -1$, if $n_{i,j}$ is a white node.

The performance of the model is measured by the accuracy in classification problems (when $t_{i,j}$ can take only the values -1 or 1) and by the relative error in regression problems (when $t_{i,j}$ may be any real number). More precisely, in a classification problem, a pattern is considered correctly classified if $\varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}) > 0$ and $t_{i,j} = 1$ or if $\varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}) < 0$ and $t_{i,j} = -1$. Thus, accuracy is defined as the percentage of patterns correctly classified by the GNN on the test set. On the other hand, in regression problems, the relative error on a pattern is given by $|(t_{i,j} - \varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}))/t_{i,j}|$.

The algorithm was implemented in Matlab® 7[13] and the software can be freely downloaded, together with the source and some examples [82]. The experiments were carried out on a Power Mac G5 with a 2-GHz PowerPC processor.

A. The Subgraph Matching Problem

The subgraph matching problem consists of finding the nodes of a given subgraph $\boldsymbol{S}$ in a larger graph $\boldsymbol{G}$. More precisely, the function τ that has to be learned is such that $\tau(\boldsymbol{G}_i, n_{i,j}) = 1$ if $n_{i,j}$ belongs to a subgraph of $\boldsymbol{G}_i$, which is isomorphic to $\boldsymbol{S}$, and $\tau(\boldsymbol{G}_i, n_{i,j}) = -1$, otherwise (see Fig. 4). Subgraph matching has a number of practical applications, such as object localization and detection of active parts in chemical compounds [73]–[75]. This problem is a basic test to assess a method for graph processing. The experiments will demonstrate that the GNN model can cope with the given task. Of course, the presented results cannot be compared with those achievable by other specific methods for subgraph matching, which are faster and more accurate. On the other hand, the GNN model is a general approach and can be used without any modification to a variety of extensions of the subgraph matching problem, where, for example, several graphs must be detected at the same time, the graphs are corrupted by noise on the structure and the labels,

[13]Copyright © 1994–2006 by The MathWorks, Inc., Natick, MA.

TABLE III
ACCURACIES ACHIEVED BY NONLINEAR MODEL (NL), LINEAR MODEL (L), AND A FEEDFORWARD NEURAL NETWORK ON SUBGRAPH MATCHING PROBLEM

			No. of nodes in G				
			6	10	14	18	Avg.
No. of nodes in S	3	NL	92.4	90.0	90.0	84.3	89.1
		L	93.3	84.5	86.7	84.7	87.3
		FNN	81.4	78.2	79.6	82.2	80.3
	5	NL	91.3	87.7	84.9	83.3	86.8
		L	90.4	85.8	85.3	80.6	85.5
		FNN	85.2	73.2	65.2	75.5	74.8
	7	NL		89.8	84.6	79.9	84.8
		L		91.3	84.4	79.2	85.0
		FNN		84.2	66.9	64.6	71.9
	9	NL		93.3	84.0	77.8	85.0
		L		92.2	84.0	77.7	84.7
		FNN		91.6	73.7	67.0	77.4
	Avg.	NL	91.8	90.2	85.9	81.3	
		L	91.9	88.5	85.1	80.6	
		FNN	83.3	81.8	71.3	72.3	
	Total avg.	NL	87.3				
		L	86.5				
		FNN	77.2				

and the target to be detected is unknown and provided only by examples.

In our experiments, the data set $\mathcal{L}$ consisted of 600 connected random graphs (constructed using $\delta = 0.2$), equally divided into a training set, a validation set, and a test set. A smaller subgraph $\boldsymbol{S}$, which was randomly generated in each trial, was inserted into every graph of the data set. Thus, each graph $\boldsymbol{G}_i$ contained at least a copy of $\boldsymbol{S}$, even if more copies might have been included by the random construction procedure. All the nodes had integer labels in the range $[0, 10]$ and, in order to define the correct targets $t_{i,j} = \tau(\boldsymbol{G}_i, n_{i,j})$, a brute force algorithm located all the copies of $\boldsymbol{S}$ in $\boldsymbol{G}_i$. Finally, a small Gaussian noise, with zero mean and a standard deviation of 0.25, was added to all the labels. As a consequence, all the copies of $\boldsymbol{S}$ in our data set were different due to the introduced noise.

In all the experiments, the state dimension was $s = 5$ and all the neural networks involved in the GNNs had five hidden neurons. More network architectures have been tested with similar results.

In order to evaluate the relative importance of the labels and the connectivity in the subgraph localization, also a feedforward neural network was applied to this test. The FNN had one output, 20 hidden, and one input units. The FNN predicted $t_{i,j}$ using only the label $\boldsymbol{l}_{n_{i,j}}$ of node $n_{i,j}$. Thus, the FNN did not use the connectivity and exploited only the relative distribution of the labels in $\boldsymbol{S}$ w.r.t. the labels in graphs $\boldsymbol{G}$.

Table III presents the accuracies achieved by the nonlinear GNN model (nonlinear), the linear GNN model (linear), and the FNN with several dimensions for $\boldsymbol{S}$ and $\boldsymbol{G}$. The results allow to single out some of the factors that have influence on the complexity of the problem and on the performance of the models. Obviously, the proportion of positive and negative patterns affects the performance of all the methods. The results improve when $|\boldsymbol{S}|$ is close to $|\boldsymbol{G}|$, whereas when $|\boldsymbol{S}|$ is about a half of $|\boldsymbol{G}|$, the performance is lower. In fact, in the latter case, the data set is perfectly balanced and it is more difficult to guess the right response. Moreover, the dimension $|\boldsymbol{S}|$, by itself, has influence on the performance, because the labels can assume only 11 different values and when $|\boldsymbol{S}|$ is small most of the nodes of the subgraph can be identified by their labels. In fact, the performances are better for smaller $|\boldsymbol{S}|$, even if we restrict our attention to the cases when $|\boldsymbol{G}| = 2|\boldsymbol{S}|$ holds.

The results show that GNNs always outperform the FNNs, confirming that the GNNs can exploit label contents and graph topology at the same time. Moreover, the nonlinear GNN model achieved a slightly better performance than the linear one, probably because nonlinear GNNs implement a more general model that can approximate a larger class of functions. Finally, it can be observed that the total average error for FNNs is about 50% larger than the GNN error (12.7 for nonlinear GNNs, 13.5 for linear GNNs, and 22.8 for FNNs). Actually, the relative difference between the GNN and FNN errors, which measures the advantage provided by the topology, tend to become smaller for larger values of $|\boldsymbol{S}|$ (see the last column of Table III). In fact, GNNs use an information diffusion mechanism to decide whether a node belongs to the subgraph. When $\boldsymbol{S}$ is larger, more information has to be diffused and, as a consequence, the function to be learned is more complex.

The subgraph matching problem was used also to evaluate the performance of the GNN model and to experimentally verify the findings about the computational cost of the model described in Section III. For this purpose, some experiments have been carried out varying the number of nodes, the number of edges in the data set, the number of hidden units in the neural networks implementing the GNN, and the dimensionality of the state. In the base case, the training set contained ten random graphs, each one made of 20 nodes and 40 edges, the networks implementing the GNN had five hidden neurons, and the state dimension was 2. The GNN was trained for 1000 epochs and the results were averaged over ten trials. As expected, the central processing unit (CPU) time required by the gradient computation grows linearly w.r.t. the number of nodes, edges and hidden units, whereas the growth is quadratic w.r.t. the state dimension. For example, Fig. 5 depicts the CPU time spent by the gradient computation process when the nodes of each graph[14] [Fig. 5(a)] and the states of the GNN [Fig. 5(b)] are increased, respectively.

It is worth mentioning that, in nonlinear GNNs, the quadratic growth w.r.t. the states, according to the discussion of Section III, depends on the time spent to calculate the Jacobian $(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$ and its derivative $\partial p_{\boldsymbol{w}}/\partial w$. Fig. 5 shows how the total time spent by the gradient computation process is composed in this case: line $-o-$ denotes the time required by the computation of $e_{\boldsymbol{w}}$ and $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{w}$; line $-*-$ denotes that for the Jacobian $(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$; line $-x-$ denotes that for the derivative $\partial p_{\boldsymbol{w}}/\partial w$; the dotted line and the dashed line represent the rest of the time[15] required by the FORWARD and the BACKWARD procedure, respectively; the continuous line stands for the rest of the time required by the gradient computation process.

[14]More precisely, in this experiment, nodes and edges were increased keeping constant to $1/2$ their ratio.

[15]That is, the time required by those procedures except for that already considered in the previous points.

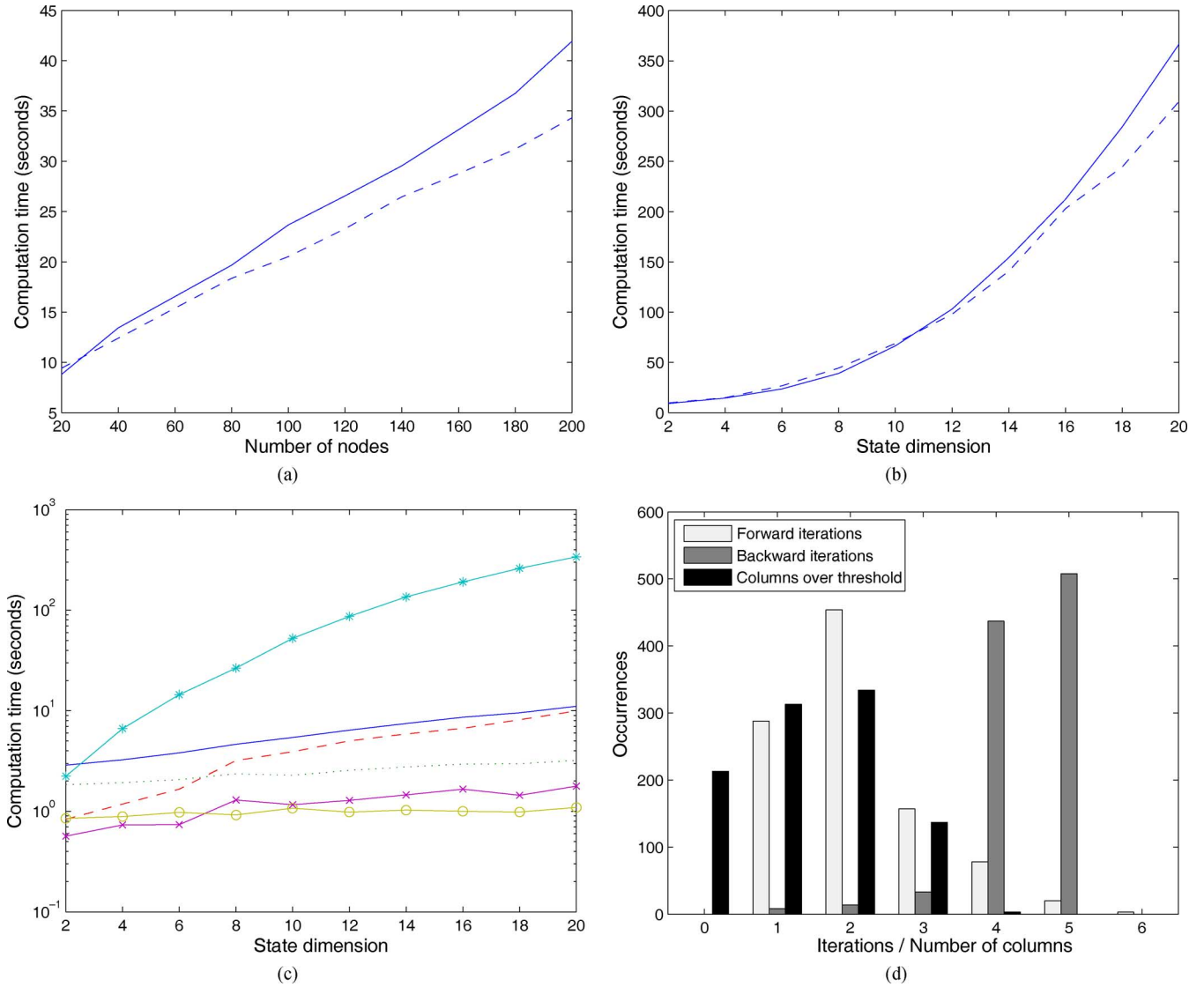

Fig. 5. Some plots about the cost of the gradient computation on GNNs. (a) and (b) CPU times required for 1000 learning epochs by nonlinear GNNs (continuous line) and linear GNN (dashed line), respectively, as a function of the number of nodes of the training set (a) and the dimension of the state (b). (c) Composition of the learning time for nonlinear GNNs: the computation of $e_{\boldsymbol{w}}$ and $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{w}$ $(-o-)$; the Jacobian $(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$ $(-*-)$; the derivative $\partial p_{\boldsymbol{w}}/\partial w$ $(-x-)$; the rest of the FORWARD procedure (dotted line); the rest of the BACKWARD procedure (dashed line); the rest of the time learning procedure (continuous line). (d) Histogram of the number of the forward iterations, the backward iterations, and the number of nodes u such that $\boldsymbol{R}_{n,u} \neq 0$ [see (17)] encountered in each epoch of a learning session.

From Fig. 5(c), we can observe that the computation of $\partial p_{\boldsymbol{w}}/\partial w$ that, in theory, is quadratic w.r.t. the states may have a small effect in practice. In fact, as already noticed in Section III, the cost of such a computation depends on the number $t_{\boldsymbol{R}}$ of columns of $(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$ whose norm is larger than the prescribed threshold, i.e., the number of nodes u and v such that $\boldsymbol{R}_{n,u} \neq 0$ [see (17)]. Such a number is usually small due to the effect of the penalty term $p_{\boldsymbol{w}}$. Fig. 5(d) shows a histogram of the number of nodes u for which $\boldsymbol{R}_{n,u} \neq 0$ in each epoch of a learning session: in practice, in this experiment, the non-null $\boldsymbol{R}_{n,u}$ are often zero and never exceed four in magnitude. Another factor that affects the learning time is the number of forward and backward iterations needed to compute the stable state and the gradient, respectively.[16] Fig. 5(d) shows also the histograms of the number of required iterations, suggesting that also those numbers are often small.

[16]The number of iterations depends also on the constant ϵ_f and ϵ_b of Table I, which were both set to $1e-3$ in the experiments. However, due to the exponential convergence of the iterative methods, these constants have a linear effect.

B. The Mutagenesis Problem

The Mutagenesis data set [13] is a small data set, which is available online and is often used as a benchmark in the relational learning and inductive logic programming literature. It contains the descriptions of 230 nitroaromatic compounds that are common intermediate subproducts of many industrial chemical reactions [83]. The goal of the benchmark consists of learning to recognize the mutagenic compounds. The log mutagenicity was thresholded at zero, so the prediction is a binary classification problem. We will demonstrate that GNNs achieved the best result compared with those reported in the literature on some parts of the data set.

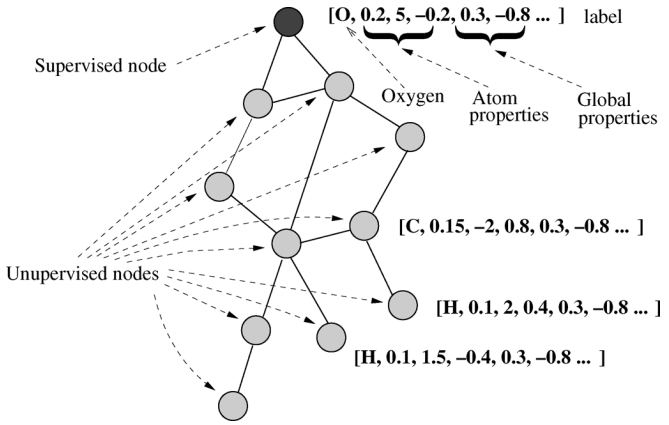

Fig. 6. Atom-bond structure of a molecule represented by a graph with labeled nodes. Nodes represent atoms and edges denote atom bonds. Only one node is supervised.

In [83], it is shown that 188 molecules out of 230 are amenable to a linear regression analysis. This subset was called "regression friendly," while the remaining 42 compounds were termed "regression unfriendly." Many different features have been used in the prediction. Apart from the atom-bond (AB) structure, each compound is provided with four global features [83]. The first two features are chemical measurements (C): the lowest unoccupied molecule orbital and the water/octanol partition coefficient, while the remaining two are precoded structural (PS) attributes. Finally, the AB description can be used to define functional groups (FG), e.g., methyl groups and many different rings that can be used as higher level features. In our experiments, the best results were achieved using AB, C, and PS, without the functional groups. Probably the reason is that GNNs can recover the substructures that are relevant to the classification, exploiting the graphical structure contained in the AB description.

In our experiments, each molecule of the data set was transformed into a graph where nodes represent atoms and edges stand for ABs. The average number of nodes in a molecule is around 26. Node labels contain atom type, its energy state, and the global properties AB, C, and PS. In each graph, there is only one supervised node, the first atom in the AB description (Fig. 6). The desired output is 1, if the molecule is mutagenic, and -1, otherwise.

In Tables IV–VI, the results obtained by nonlinear GNNs[17] are compared with those achieved by other methods. The presented results were evaluated using a tenfold cross-validation procedure, i.e., the data set was randomly split into ten parts and the experiments were repeated ten times, each time using a different part as the test set and the remaining patterns as training set. The results were averaged on five runs of the cross-validation procedure.

GNNs achieved the best accuracy on the regression-unfriendly part (Table V) and on the whole data set (Table VI), while the results are close to the state of the art techniques on the regression-friendly part (Table IV). It is worth noticing that, whereas most of the approaches showed a higher level of accuracy when applied to the whole data set with respect to the unfriendly part, the converse holds for GNNs. This suggests that GNNs can capture characteristics of the patterns that are useful to solve the problem but are not homogeneously distributed in the two parts.

TABLE IV
ACCURACIES ACHIEVED ON THE REGRESSION-FRIENDLY PART OF THE MUTAGENESIS DATA SET. THE TABLE DISPLAYS THE METHOD, THE FEATURES USED TO MAKE THE PREDICTION, AND A POSSIBLE REFERENCE TO THE PAPER WHERE THE RESULT IS DESCRIBED

Method	Features	Reference	Accuracy
non–linear GNN	AB+C+PS		94.3
Neural Networks	C+PS	[13]	89.0%
P-Progol	AB+C	[13]	82.0%
P-Progol	AB+C+FG	[13]	88.0%
MFLOG	AB+C	[84]	95.7%
FOIL	AB	[85]	76%
boosted-FOIL	not available	[86]	88.3%
$1nn(d_m)$	AB	[87]	83
$1nn(d_m)$	AB+C	[87]	91%
RDBC	AB	[88]	84%
RDBC	AB+C	[88]	83%
RSD	AB+C+FG	[89]	92.6%
SINUS	AB+C+FG	[89]	84.5%
RELAGGS	AB+C+FG	[89]	88.0%
RS	AB	[90]	88.9%
RS	AB+FG	[90]	89.9%
RS	AB+C+PS+FG	[90]	95.8%
SVM_P	not available	[91]	91.5

TABLE V
ACCURACIES ACHIEVED ON THE REGRESSION-UNFRIENDLY PART OF THE MUTAGENESIS DATA SET. THE TABLE DISPLAYS THE METHOD, THE FEATURES USED TO MAKE THE PREDICTION, AND A POSSIBLE REFERENCE TO THE PAPER WHERE THE RESULT IS DESCRIBED

Method	Knowledge	Reference	Accuracy
non–linear GNN	AB+C+PS		96.0%
$1nn(d_m)$	AB	[87]	72%
$1nn(d_m)$	AB+C	[87]	72%
TILDE	AB	[92]	85%
TILDE	AB+C	[92]	79%
RDBC	AB	[88]	79%
RDBC	AB+C	[88]	79%

TABLE VI
ACCURACIES ACHIEVED ON THE WHOLE MUTAGENESIS DATA SET. THE TABLE DISPLAYS THE METHOD, THE FEATURES USED TO MAKE THE PREDICTION, AND A POSSIBLE REFERENCE TO THE PAPER WHERE THE RESULT IS DESCRIBED

Method	Knowledge	Reference	Accuracy
non–linear GNN	AB+C+PS		90.5%
$1nn(d_m)$	AB	[87]	81%
$1nn(d_m)$	AB+C	[87]	88%
TILDE	AB	[92]	77%
TILDE	AB+C	[92]	82%
RDBC	AB	[88]	83%
RDBC	AB+C	[88]	82%

C. Web Page Ranking

In this experiment, the goal is to learn the rank of a web page, inspired by Google's PageRank [18]. According to PageRank, a page is considered authoritative if it is referred by many other pages and if the referring pages are authoritative themselves. Formally, the PageRank p_n of a page n is $p_n = d\sum_{u\in \text{pa}[n]} p_u/o_n + (1-d)$, where o_n is the outdegree of n, and $d \in [0,1]$ is the damping factor [18]. In this experiment, it is shown that a GNN can learn a modified version of PageRank, which adapts the "authority" measure according to the page content. For this purpose, a random web graph $\boldsymbol{G}$

[17]Some results were already presented in [80].

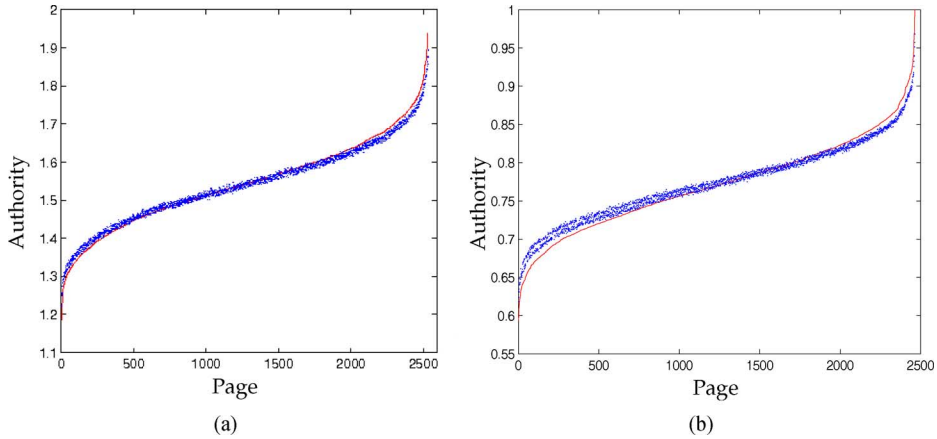

Fig. 7. Desired function τ (the continuous lines) and the output of the GNN (the dotted lines) on the pages that belong to only one topic (a) and on the other pages (b). Horizontal axis stands for pages and vertical axis stands for scores. Pages have been sorted according to the desired value $\tau(\boldsymbol{G}, n)$.

containing 5000 nodes was generated, with $\delta = 0.2$. Training, validation, and test sets consisted of different nodes of this graph. More precisely, only 50 nodes were supervised in the training set, other 50 nodes belonged to the validation set, and the remaining nodes were in the test set.

To each node n, a bidimensional boolean label $[a_n, b_n]$ is attached that represents whether the page belongs to two given topics. If the page n belongs to both topics, then $[a_n, b_n] = [1, 1]$, while if it belongs to only one topic, then $[a_n, b_n] = [1, 0]$, or $[a_n, b_n] = [0, 1]$, and if it does not belong to either topics, then $[a_n, b_n] = [0, 0]$. The GNN was trained in order to produce the following output:

$$\tau(\boldsymbol{G}, n) = \begin{cases} 2p_n/\|\boldsymbol{p}\|_1, & \text{if } (a_n \,\mathrm{XOR}\, b_n) = 1 \\ p_n/\|\boldsymbol{p}\|_1, & \text{otherwise} \end{cases}$$

where $\boldsymbol{p}$ stands for the Google's PageRank.

Web page ranking algorithms are used by search engines to sort the URLs returned in response to user's queries and more generally to evaluate the data returned by information retrieval systems. The design of ranking algorithms capable of mixing together the information provided by web connectivity and page content has been a matter of recent research [93]–[96]. In general, this is an interesting and hard problem due to the difficulty in coping with structured information and large data sets. Here, we present the results obtained by GNNs on a synthetic data set. More results achieved on a snapshot of the web are available in [79].

For this example, only the linear model has been used, because it is naturally suited to approximate the linear dynamics of the PageRank. Moreover, the transition and forcing networks (see Section I) were implemented by three-layered neural networks with five hidden neurons, and the dimension of the state was $s = 1$. For the output function, $g_{\boldsymbol{w}}$ is implemented as $g_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{l}_n) = \boldsymbol{x}'_n \cdot \pi_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{l}_n)$, where π_w is the function realized by a three-layered neural networks with five hidden neurons.

Fig. 7 shows the output of the GNN φ and the target function τ on the test set. Fig. 7(a) displays the result for the pages that belong to only one topic and Fig. 7(b) displays the result for the other pages. Pages are displayed on horizontal axes and are sorted according to the desired output $\tau(\boldsymbol{G}, n)$. The plots denote the value of function τ (continuous lines) and the value of the function implemented by the GNN (the dotted lines). The figure clearly suggests that GNN performs very well on this problem.

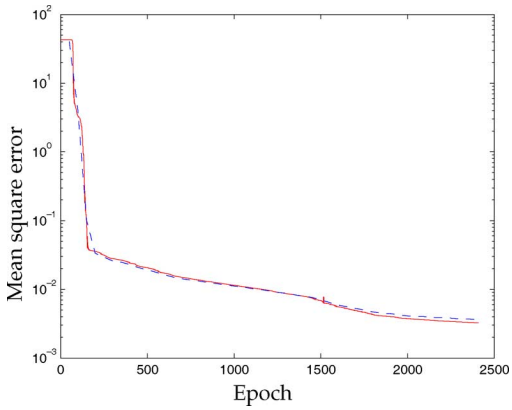

Fig. 8. Error function on the training set (continuous line) and on the validation (dashed line) set during learning phase.

Finally, Fig. 8 displays the error function during the learning process. The continuous line is the error on the training set, whereas the dotted line is the error on the validation set. It is worth noting that the two curves are always very close and that the error on the validation set is still decreasing after 2400 epochs. This suggests that the GNN does not experiment overfitting problems, despite the fact that the learning set consists of only 50 pages from a graph containing 5000 nodes.

V. Conclusion

In this paper, we introduced a novel neural network model that can handle graph inputs: the graphs can be cyclic, directed, undirected, or a mixture of these. The model is based on information diffusion and relaxation mechanisms. The approach extends into a common framework, the previous connectionist techniques for processing structured data, and the methods based on random walk models. A learning algorithm to estimate model parameters was provided and its computational complexity was studied, demonstrating that the method is suitable also for large data sets.

Some promising experimental results were provided to assess the model. In particular, the results achieved on the whole Mutagenisis data set and on the unfriendly part of such a data set are the best compared with those reported in the open literature. Moreover, the experiments on the subgraph matching and on the web page ranking show that the method can be applied to problems that are related to important practical applications.

The possibility of dealing with domains where the data consists of patterns and relationships gives rise to several new topics of research. For example, while in this paper it is assumed that the domain is static, it may happen that the input graphs change with time. In this case, at least two interesting issues can be considered: first, GNNs must be extended to cope with a dynamic domain; and second, no method exists, to the best of our knowledge, to model the evolution of the domain. The solution of the latter problem, for instance, may allow to model the evolution of the web and, more generally, of social networks. Another topic of future research is the study on how to deal with domains where the relationships, which are not known in advance, must be inferred. In this case, the input contains flat data and is automatically transformed into a set of graphs in order to shed some light on possible hidden relationships.

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," *J. Mach. Learn. Res.*, 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 *Lecture Notes in Computer Science — Graphics Recognition*, K. Tombre and A. K. Chhabra, Eds. Berlin, Germany: Springer-Verlag, 1997.

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

[4] A. Mason and E. Blake, "A graphical representation of the state spaces of hierarchical level-of-detail scene descriptions," *IEEE Trans. Vis. Comput. Graphics*, vol. 7, no. 1, pp. 70–75, Jan.-Mar. 2001.

[5] L. Baresi and R. Heckel, "Tutorial introduction to graph transformation: A software engineering perspective," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2002, vol. 2505, pp. 402–429.

[6] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proc. ACM Symp. Software Vis.*, 2003, pp. 77–86.

[7] A. Bua, M. Gori, and F. Santini, "Recursive neural networks applied to discourse representation theory," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2002, vol. 2415.

[8] L. De Raedt, *Logical and Relational Learning*. New York: Springer-Verlag, 2008, to be published.

[9] T. Dietterich, L. Getoor, and K. Murphy, Eds., *Proc. Int. Workshop Statist. Relat. Learn. Connect. Other Fields*, 2004.

[10] P. Avesani and M. Gori, Eds., *Proc. Int. Workshop Sub-Symbol. Paradigms Structured Domains*, 2005.

[11] S. Nijseen, Ed., *Proc. 3rd Int. Workshop Mining Graphs Trees Sequences*, 2005.

[12] T. Gaertner, G. Garriga, and T. Meini, Eds., *Proc. 4th Int. Workshop Mining Graphs Trees Sequences*, 2006.

[13] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg, "Mutagenesis: Ilp experiments in a non-determinate biological domain," in *Proc. 4th Int. Workshop Inductive Logic Programm.*, 1994, pp. 217–232.

[14] T. Pavlidis, *Structural Pattern Recognition*, ser. Electrophysics. New York: Springer-Verlag, 1977.

[15] M. Bianchini, M. Maggini, L. Sarti, and F. Scarselli, "Recursive neural networks learn to localize faces," *Phys. Rev. Lett.*, vol. 26, no. 12, pp. 1885–1895, Sep. 2005.

[16] S. Haykin, *Neural Networks: A Comprehensive Foundation*. New York: Prentice-Hall, 1994.

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

[18] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. 7th World Wide Web Conf.*, Apr. 1998, pp. 107–117.

[19] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Trans. Neural Netw.*, vol. 8, no. 2, pp. 429–459, Mar. 1997.

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

[21] J. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, 1999.

[22] A. C. Tsoi, G. Morini, F. Scarselli, M. Hagenbuchner, and M. Maggini, "Adaptive ranking of web pages," in *Proc. 12th World Wide Web Conf.*, Budapest, Hungary, May 2003, pp. 356–365.

[23] M. Bianchini, P. Mazzoni, L. Sarti, and F. Scarselli, "Face spotting in color images using recursive neural networks," in *Proc. 1st Int. Workshop Artif. Neural Netw. Pattern Recognit.*, Florence, Italy, Sep. 2003, pp. 76–81.

[24] M. Bianchini, M. Gori, and F. Scarselli, "Processing directed acyclic graphs with recursive neural networks," *IEEE Trans. Neural Netw.*, vol. 12, no. 6, pp. 1464–1470, Nov. 2001.

[25] A. Küchler and C. Goller, "Inductive learning in symbolic domains using structure-driven recurrent neural networks," in *Lecture Notes in Computer Science*, G. Görz and S. Hölldobler, Eds. Berlin, Germany: Springer-Verlag, Sep. 1996, vol. 1137.

[26] T. Schmitt and C. Goller, "Relating chemical structure to activity: An application of the neural folding architecture," in *Proc. Workshop Fuzzy-Neuro Syst./Conf. Eng. Appl. Neural Netw.*, 1998, pp. 170–177.

[27] M. Hagenbuchner and A. C. Tsoi, "A supervised training algorithm for self-organizing maps for structures," *Pattern Recognit. Lett.*, vol. 26, no. 12, pp. 1874–1884, 2005.

[28] M. Gori, M. Maggini, E. Martinelli, and F. Scarselli, "Learning user profiles in NAUTILUS," in *Proc. Int. Conf. Adaptive Hypermedia Adaptive Web-Based Syst.*, Trento, Italy, Aug. 2000, pp. 323–326.

[29] M. Bianchini, P. Mazzoni, L. Sarti, and F. Scarselli, "Face spotting in color images using recursive neural networks," in *Proc. Italian Workshop Neural Netw.*, Vietri sul Mare, Italy, Jul. 2003.

[30] B. Hammer and J. Jain, "Neural methods for non-standard data," in *Proc. 12th Eur. Symp. Artif. Neural Netw.*, M. Verleysen, Ed., 2004, pp. 281–292.

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

[32] T. Gärtner, J. Lloyd, and P. Flach, "Kernels and distances for structured data," *Mach. Learn.*, vol. 57, no. 3, pp. 205–232, 2004.

[33] R. Kondor and J. Lafferty, "Diffusion kernels on graphs and other discrete structures," in *Proc. 19th Int. Conf. Mach. Learn.*, C. Sammut and A. e. Hoffmann, Eds., 2002, pp. 315–322.

[34] H. Kashima, K. Tsuda, and A. Inokuchi, "Marginalized kernels between labeled graphs," in *Proc. 20th Int. Conf. Mach. Learn.*, T. Fawcett and N. e. Mishra, Eds., 2003, pp. 321–328.

[35] P. Mahé, N. Ueda, T. Akutsu, J.-L Perret, and J.-P. Vert, "Extensions of marginalized graph kernels," in *Proc. 21st Int. Conf. Mach. Learn.*, 2004, pp. 552–559.

[36] M. Collins and N. Duffy, "Convolution kernels for natural language," in *Advances in Neural Information Processing Systems*, T. G. Dietterich, S. Becker, and Z. Ghahramani, Eds. Cambridge, MA: MIT Press, 2002, vol. 14, pp. 625–632.

[37] J. Suzuki, Y. Sasaki, and E. Maeda, "Kernels for structured natural language data," in *Proc. Conf. Neural Inf. Process. Syst.*, 2003.

[38] J. Suzuki, H. Isozaki, and E. Maeda, "Convolution kernels with feature selection for natural language processing tasks," in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2004, pp. 119–126.

[39] J. Cho, H. Garcia-Molina, and L. Page, "Efficient crawling through url ordering," in *Proc. 7th World Wide Web Conf.*, Brisbane, Australia, Apr. 1998, pp. 161–172.

[40] A. C. Tsoi, M. Hagenbuchner, and F. Scarselli, "Computing customized page ranks," *ACM Trans. Internet Technol.*, vol. 6, no. 4, pp. 381–414, Nov. 2006.

[41] H. Chang, D. Cohn, and A. K. McCallum, "Learning to create customized authority lists," in *Proc. 17th Int. Conf. Mach. Learn.*, 2000, pp. 127–134.

[42] J. Lafferty, A. McCallum, and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proc. 18th Int. Conf. Mach. Learn.*, 2001, pp. 282–289.

[43] F. V. Jensen, *Introduction to Bayesian Networks*. New York: Springer-Verlag, 1996.
[44] L. Getoor and B. Taskar, *Introduction to Statistical Relational Learning*. Cambridge, MA: MIT Press, 2007.
[45] V. N. Vapnik, *Statistical Learning Theory*. New York: Wiley, 1998.
[46] O. Chapelle, B. Schölkopf, and A. Zien, Eds., *Semi-Supervised Learning*. Cambridge, MA: MIT Press, 2006.
[47] L. Chua and L. Yang, "Cellular neural networks: Theory," *IEEE Trans. Circuits Syst.*, vol. CAS-35, no. 10, pp. 1257–1272, Oct. 1988.
[48] L. Chua and L. Yang, "Cellular neural networks: Applications," *IEEE Trans. Circuits Syst.*, vol. CAS-35, no. 10, pp. 1273–1290, Oct. 1988.
[49] P. Kaluzny, "Counting stable equilibria of cellular neural networks-A graph theoretic approach," in *Proc. Int. Workshop Cellular Neural Netw. Appl.*, 1992, pp. 112–116.
[50] M. Ogorzatek, C. Merkwirth, and J. Wichard, "Pattern recognition using finite-iteration cellular systems," in *Proc. 9th Int. Workshop Cellular Neural Netw. Appl.*, 2005, pp. 57–60.
[51] J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Nat. Acad. Sci.*, vol. 79, pp. 2554–2558, 1982.
[52] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "Computation capabilities of graph neural networks," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, Jan. 2009, to be published.
[53] M. A. Khamsi, *An Introduction to Metric Spaces and Fixed Point Theory*. New York: Wiley, 2001.
[54] M. Bianchini, M. Maggini, L. Sarti, and F. Scarselli, "Recursive neural networks for processing graphs with labelled edges: Theory and applications," *Neural Netw.*, vol. 18, Special Issue on Neural Networks and Kernel Methods for Structured Domains, no. 8, pp. 1040–1050, 2005.
[55] M. J. D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *Comput. J.*, vol. 7, pp. 155–162, 1964.
[56] W. T. Miller, III, R. Sutton, and P. E. Werbos, *Neural Network for Control*. Camrbidge, MA: MIT Press, 1990.
[57] A. C. Tsoi, "Adaptive processing of sequences and data structures," in *Lecture Notes in Computer Science*, C. L. Giles and M. Gori, Eds. Berlin, Germany: Springer-Verlag, 1998, vol. 1387, pp. 27–62.
[58] L. Almeida, "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment," in *Proc. IEEE Int. Conf. Neural Netw.*, M. Caudill and C. Butler, Eds., San Diego, 1987, vol. 2, pp. 609–618.
[59] F. Pineda, "Generalization of back-propagation to recurrent neural networks," *Phys. Rev. Lett.*, vol. 59, pp. 2229–2232, 1987.
[60] W. Rudin, *Real and Complex Analysis*, 3rd ed. New York: McGraw-Hill, 1987.
[61] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The rprop algorithm," in *Proc. IEEE Int. Conf. Neural Netw.*, San Francisco, CA, 1993, pp. 586–591.
[62] M. Bishop, *Neural Networks for Pattern Recognition*. Oxford, U.K.: Oxford Univ. Press, 1995.
[63] R. Reed, "Pruning algorithms — A survey," *IEEE Trans. Neural Netw.*, vol. 4, no. 5, pp. 740–747, Sep. 1993.
[64] S. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed. Denver, San Mateo: Morgan Kaufmann, 1989, vol. 2, pp. 524–532.
[65] T.-Y. Kwok and D.-Y. Yeung, "Constructive algorithms for structure learning in feedforward neural networks for regression problems," *IEEE Trans. Neural Netw.*, vol. 8, no. 3, pp. 630–645, May 1997.
[66] G.-B. Huang, L. Chen, and C. K. Siew, "Universal approximation using incremental constructive feedforward networks with random hidden nodes," *IEEE Trans. Neural Netw.*, vol. 17, no. 1, pp. 879–892, Jan. 2006.
[67] F. Scarselli and A. C. Tsoi, "Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results," *Neural Netw.*, vol. 11, no. 1, pp. 15–37, 1998.
[68] 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.*, vol. 41, no. 1, pp. 202–218, 2001.
[69] M. Hagenbuchner, A. C. Tsoi, and A. Sperduti, "A supervised self-organising map for structured data," in *Advances in Self-Organising Maps*, N. Allinson, H. Yin, L. Allinson, and J. Slack, Eds. Berlin, Germany: Springer-Verlag, 2001, pp. 21–28.
[70] E. Seneta, *Non-Negative Matrices and Markov Chains*. New York: Springer-Verlag, 1981, ch. 4, pp. 112–158.
[71] D. E. Rumelhart and J. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: PDP Research Group, MIT Press, 1986, vol. 1.
[72] A. Graham, *Kronecker Products and Matrix Calculus: With Applications*. New York: Wiley, 1982.
[73] H. Bunke, "Graph matching: Theoretical foundations, algorithms, and applications," in *Proc. Vis. Interface*, Montreal, QC, Canada, 2000, pp. 82–88.
[74] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Graph matching applications in pattern recognition and image processing," in *Proc. Int. Conf. Image Process.*, Sep. 2003, vol. 2, pp. 21–24.
[75] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 18, no. 3, pp. 265–268, 2004.
[76] A. Agarwal, S. Chakrabarti, and S. Aggarwal, "Learning to rank networked entities," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Disc. Data Mining*, New York, 2006, pp. 14–23.
[77] V. Di Massa, G. Monfardini, L. Sarti, F. Scarselli, M. Maggini, and M. Gori, "A comparison between recursive neural networks and graph neural networks," in *Proc. Int. Joint Conf. Neural Netw.*, Jul. 2006, pp. 778–785.
[78] G. Monfardini, V. Di Massa, F. Scarselli, and M. Gori, "Graph neural networks for object localization," in *Proc. 17th Eur. Conf. Artif. Intell.*, Aug. 2006, pp. 665–670.
[79] F. Scarselli, S. Yong, M. Gori, M. Hagenbuchner, A. C. Tsoi, and M. Maggini, "Graph neural networks for ranking web pages," in *Proc. IEEE/WIC/ACM Conf. Web Intelligence*, 2005, pp. 666–672.
[80] W. Uwents, G. Monfardini, H. Blockeel, F. Scarselli, and M. Gori, "Two connectionist models for graph processing: An experimental comparison on relational data," in *Proc. Eur. Conf. Mach. Learn.*, 2006, pp. 213–220.
[81] S. Yong, M. Hagenbuchner, F. Scarselli, A. C. Tsoi, and M. Gori, "Document mining using graph neural networks," in *Proc. 5th Int. Workshop Initiative Evaluat. XML Retrieval*, N. Fuhr, M. Lalmas, and A. Trotman, Eds., 2007, pp. 458–472.
[82] F. Scarselli and G. Monfardini, The GNN Toolbox, [Online]. Available: http://airgroup.dii.unisi.it/projects/GraphNeuralNetwork/download.htm
[83] A. K. Debnath, R. Lopex de Compandre, G. Debnath, A. Schusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity," *J. Med. Chem.*, vol. 34, no. 2, pp. 786–797, 1991.
[84] S. Kramer and L. De Raedt, "Feature construction with version spaces for biochemical applications," in *Proc. 18th Int. Conf. Mach. Learn.*, 2001, pp. 258–265.
[85] J. Quinlan and R. Cameron-Jones, "FOIL: A midterm report," in *Proc. Eur. Conf. Mach. Learn.*, 1993, pp. 3–20.
[86] J. Quinlan, "Boosting first-order learning," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1996, vol. 1160, p. 143.
[87] J. Ramon, "Clustering and instance based learning in first order logic," Ph.D. dissertation, Dept. Comput. Sci., K.U. Leuven, Leuven, Belgium, 2002.
[88] M. Kirsten, "Multirelational distance-based clustering," Ph.D. dissertation, Schl. Comput. Sci., Otto-von-Guericke Univ., Magdeburg, Germany, 2002.
[89] M. Krogel, S. Rawles, F. Zelezny, P. Flach, N. Lavrac, and S. Wrobel, "Comparative evaluation of approaches to propositionalization," in *Proc. 13th Int. Conf. Inductive Logic Programm.*, 2003, pp. 197–214.
[90] S. Muggleton, "Machine learning for systems biology," in *Proc. 15th Int. Conf. Inductive Logic Programm.*, Bonn, Germany, Aug. 10 – 13, 2005, pp. 416–423.
[91] A. Woźnica, A. Kalousis, and M. Hilario, "Matching based kernels for labeled graphs," in *Proc. Int. Workshop Mining Learn. Graphs/ECML/PKDD*, T. Gärtner, G. Garriga, and T. Meinl, Eds., 2006, pp. 97–108.
[92] L. De Raedt and H. Blockeel, "Using logical decision trees for clustering," in *Lecture Notes in Artificial Intelligence*. Berlin, Germany: Springer-Verlag, 1997, vol. 1297, pp. 133–141.
[93] M. Diligenti, M. Gori, and M. Maggini, "Web page scoring systems for horizontal and vertical search," in *Proc. 11th World Wide Web Conf.*, 2002, pp. 508–516.
[94] T. H. Haveliwala, "Topic sensitive pagerank," in *Proc. 11th World Wide Web Conf.*, 2002, pp. 517–526.
[95] G. Jeh and J. Widom, "Scaling personalized web search," in *Proc. 12th World Wide Web Conf.*, May 20–24, 2003, pp. 271–279.

[96] F. Scarselli, A. C. Tsoi, and M. Hagenbuchner, "Computing personalized pageranks," in *Proc. 12th World Wide Web Conf.*, 2004, pp. 282–283.

Franco Scarselli received the Laurea degree with honors in computer science from the University of Pisa, Pisa, Italy, in 1989 and the Ph.D. degree in computer science and automation engineering from the University of Florence, Florence, Italy, in 1994.

He has been supported by foundations of private and public companies and by a postdoctoral of the University of Florence. In 1999, he moved to the University of Siena, Siena, Italy, where he was initially a Research Associate and is currently an Associate Professor at the Department of Information Engineering. He is the author of more than 50 journal and conference papers and has been has been involved in several research projects, founded by public institutions and private companies, focused on software engineering, machine learning, and information retrieval. His current theoretical research activity is mainly in the field of machine learning with a particular focus on adaptive processing of data structures, neural networks, and approximation theory. His research interests include also image understanding, information retrieval, and web applications.

Marco Gori (S'88–M'91–SM'97–F'01) received the Ph.D. degree from University di Bologna, Bologna, Italy, in 1990, while working in part as a visiting student at the School of Computer Science, McGill University, Montréal, QC, Canada.

In 1992, he became an Associate Professor of Computer Science at Università di Firenze and, in November 1995, he joint the Università di Siena, Siena, Italy, where he is currently Full Professor of Computer Science. His main interests are in machine learning with applications to pattern recognition, web mining, and game playing. He is especially interested in the formulation of relational machine learning schemes in the continuum setting. He is the leader of the WebCrow project for automatic solving of crosswords that outperformed human competitors in an official competition taken place within the 2006 European Conference on Artificial Intelligence. He is coauthor of the book *Web Dragons: Inside the Myths of Search Engines Technologies* (San Mateo: Morgan Kauffman, 2006).

Dr. Gori serves (has served) as an Associate Editor of a number of technical journals related to his areas of expertise and he has been the recipient of best paper awards and keynote speakers in a number of international conferences. He was the Chairman of the Italian Chapter of the IEEE Computational Intelligence Society and the President of the Italian Association for Artificial Intelligence. He is a Fellow of the European Coordinating Committee for Artificial Intelligence.

Ah Chung Tsoi received the Higher Diploma degree in electronic engineering from the Hong Kong Technical College, Hong Kong, in 1969 and the M.Sc. degree in electronic control engineering and the Ph.D. degree in control engineering from University of Salford, Salford, U.K., in 1970 and 1972, respectively.

He was a Postdoctoral Fellow at the Inter-University Institute of Engineering Control at University College of North Wales, Bangor, North Wales and a Lecturer at Paisley College of Technology, Paisley, Scotland. He was a Senior Lecturer in Electrical Engineering in the Department of Electrical Engineering, University of Auckland, New Zealand, and a Senior Lecturer in Electrical Engineering, University College University of New South Wales, Australia, for five years. He then served as Professor of Electrical Engineering at University of Queensland, Australia; Dean, and simultaneously, Director of Information Technology Services, and then foundation Pro-Vice Chancellor (Information Technology and Communications) at University of Wollongong, before joining the Australian Research Council as an Executive Director, Mathematics, Information and Communications Sciences. He was Director, Monash e-Research Centre, Monash University, Melbourne, Australia. In April 2007, became a Vice President (Research and Institutional Advancement), Hong Kong Baptist University, Hong Kong. In recent years, he has been working in the area of artificial intelligence in particular neural networks and fuzzy systems. He has published in neural network literature. Recently, he has been working in the application of neural networks to graph domains, with applications to the world wide web searching, and ranking problems, and subgraph matching problem.

Markus Hagenbuchner (M'02) received the B.Sc. degree (with honors) and the Ph.D. degree in computer science from University of Wollongong, Wollongong, Australia, in 1996 and 2002, respectively.

Currently, he is a Lecturer at Faculty of Informatics, University of Wollongong. His main research activities are in the area of machine learning with special focus on supervised and unsupervised methods for the graph structured domain. His contribution to the development of a self-organizing map for graphs led to winning the international competition on document mining on several occasions.

Gabriele Monfardini received the Laurea degree and the Ph.D. degree in information engineering from the University of Siena, Siena, Italy, in 2003 and 2007, respectively.

Currently, he is Contract Professor at the School of Engineering, University of Siena. His main research interests include neural networks, adaptive processing of structured data, image analysis, and pattern recognition.