The Graph Neural Network Model

Franco Scarselli, Marco Gori, Ah Chung Tsoi, 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, 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 the graph domain. This GNN model, which can directly process most of the practically useful types of graphs, e.g. acyclic, cyclic, directed, un-directed, implements a transduction function $\tau(\boldsymbol{G}, n) \in I\!R^m$ 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. Computational cost of the proposed algorithm is also considered. Some experimental results are shown to validate the proposed learning algorithm, and demonstrate its generalization capability.

Index Terms

Graph Neural Networks, Graph Processing, Recursive Neural Networks, Graphical Domains.

I. Introduction

Data can be naturally represented by graph structures in several application areas including proteomics [1], pattern recognition and 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 application domains, the information is organized in more complex graph structures such as trees, acyclic graphs, or cyclic graphs. Traditionally, the exploitation of data relationships 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 marriage with statistics and neural networks (see e.g. the recent workshops [9], [10], [11], [12]).

In machine learning, the 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]: $\tau(\boldsymbol{G}, n) \in I\!R^m$.

Applications to a graphical domain can generally be divided into two classes: *graph focused* and *node focused* applications respectively in this paper.

Scarselli, Gori, Monfardini are with the University of Siena, Siena, Italy. Email: {franco,marco,monfardini}@dii.unisi.it.

Tsoi is with Hong Kong Baptist University, Kowloon, Hong Kong. Email: act@hkbu.edu.hk

Hagenbuchner is with University of Wollongong, Wollongong, Australia. Email: markus@uow.edu.au

[1]Note that in most classification problems, the mapping is to a set of integers $I\!N^m$, while in regression problems, the mapping is to a set of reals $I\!R^m$. Here for simplicity of exposition, we will denote only the regression case. The proposed formulation can be trivially re-written for the situation of classification.

In *graph focused* applications, the function τ is independent of the node n and implements a classifier or a regressor on a graph structured dataset. For example, a chemical compound can be modeled by a graph $\boldsymbol{G}$, the nodes of which stand for atoms and the edges of which represent chemical bonds (see Figure 1-**A**) linking some of the atoms together. The mapping $\tau(\boldsymbol{G})$ may be used to estimate the probability that the chemical compound is active against a certain disease. In Figure 1-**B**, an image can be represented by a region adjacency graph (RAG) where nodes denote homogeneous regions of intensity of the image and arcs represent their adjacency relationship [13]. 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 or not, and, if so, localizing its position. This problem can be solved by a function τ which classifies the nodes of the RAG according to whether the corresponding region belongs to the object or not. For example, the output of τ for Figure 1-**B** might be 1 for the black nodes, which corresponds to the castle, and 0 otherwise. Another example comes from web page classifications. The web can be represented by a graph where nodes stand for pages and edges represent the hyperlinks between the web pages (Figure 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.

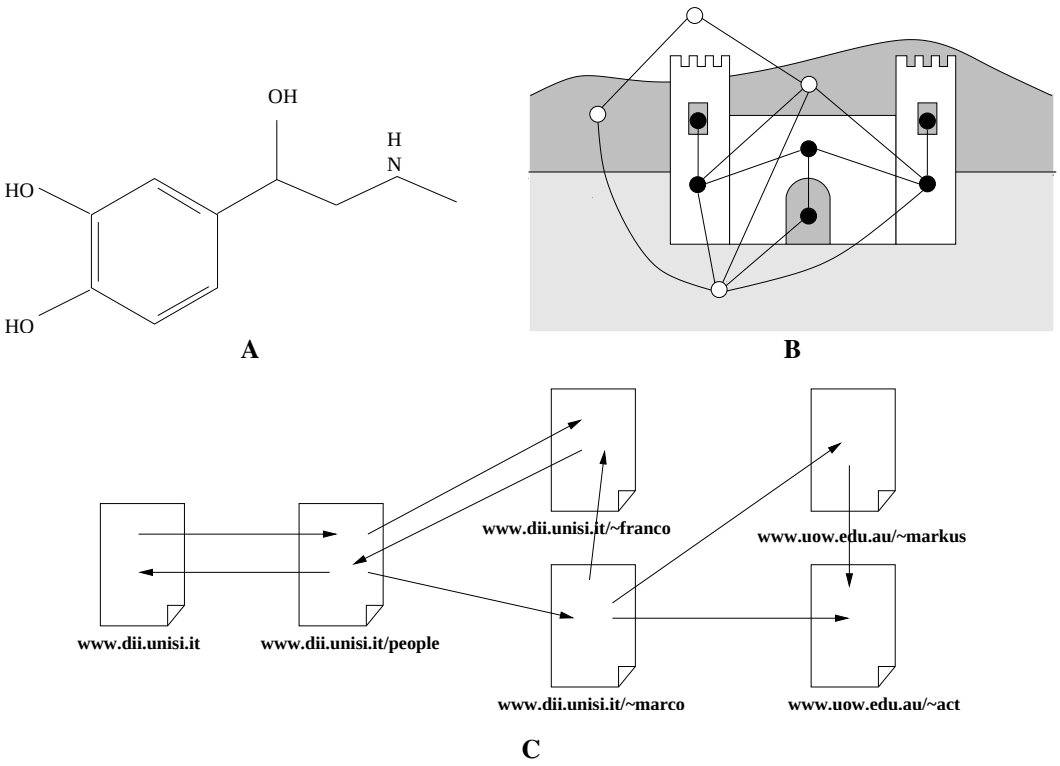

Fig. 1. Some applications where the information is represented by graphs: **A** is a chemical compound (adrenaline); **B** an image; and **C** a subset of the web.

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. In other words, the preprocessing step first "squashes" the graph structured data into a vector of reals and then deal with the preprocessed data using a list based data processing technique. However, important information, e.g., the topological dependency of information on node n 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 are various approaches [14], [15] attempting to preserve the graph structured nature of the data for as long as required before processing the data. In these recent approaches the idea is to encode the underlying graph structured data using the topological relationships among the nodes of the graph to incorporate the graph structured information in the data processing step. *Recursive neural networks* [14], [16], [17] and *Markov chains* [15], [18], [19] belong to this set of techniques and are commonly applied both to graph and node focused problems. Our method 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 [14], [16], [17]. The method estimates the parameters φ_w of a function 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 [20]. Similarly, using a preprocessing phase, it is possible to handle certain types of cyclic graphs [21]. Recursive neural networks have been applied to several problems including logical term classification [22], chemical compound classification [23], logo recognition [2], [24], web Page scoring [25], and face localization [26].

Recursive neural networks are also related to *support vector machines* [27], [28], [29], which adopt special kernels to operate on graph structured data. For example, the *diffusion kernel* [30] is based on a heat equation; the kernels proposed in [31], [32] exploit the vectors produced by a graph random walker and those designed in [33], [34], [35] use a method of counting the number of common substructures of two trees. In fact, recursive neural networks, as 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 [15], [18]. Internet search engines use ranking algorithms to measure the relative "importance" of web pages. Such measurements are generally exploited, along other page features, by "horizontal" search engines, e.g., Google [15], or by personalized search engines ("vertical" search engines, see, e.g., [19]) 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 training examples [19], [37], [38]. Such models are able to generalize the results to score all the web pages in the collection.

[2]The relative importance measure of a web page is also used to serve other goals, e.g. to improve the efficiency of crawlers [36].

In this paper, we present a new neural network model which is suitable for both graph and node focused applications. This new model unifies these two existing models into a common framework. We will call this new 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 to 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.

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 to mention that elsewhere 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] [39].

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 graph neural network model, together with a learning algorithm for the parameter estimation. 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 neighbours 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 S is 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 neighbours 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 Figure 1-**b**, in an image, node labels might represent properties of the regions (e.g., area, perimeter, 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 the momentums). No assumption is made on the arcs, directed and undirected edges are both permitted. However, when different kinds of edges co-exist in the same dataset, 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 non–positional. Non–positional graphs are those described so far, positional graphs differs since for each node n, there exists an injective function $\nu_n : \mathrm{ne}[n] \rightarrow \{1, \ldots, |\boldsymbol{N}|\}$

[3]Due to the length of proof, such a result cannot be proved here and is included [39].

which assigns to each neighbour of n a different position. Notice that, the position of the neighbour can be implicitly used for storing useful information. For instance, let us consider the example of the Region Adjacency Graph (RAG), (see Figure 1-**b**). ν_n can be used to represent the relative spatial position of the regions; e.g., ν_n might enumerate the neighbours of a node n, which represents the adjacent regions, following a clockwise ordering.

The domain considered in this paper is the set $\mathcal{D}$ of pairs graph–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 j-the node in the set $\boldsymbol{N}_i \in \mathcal{N}$; $(\boldsymbol{G}_i, n_{i,j}, \boldsymbol{t}_{i,j})$ indicates that $\boldsymbol{t}_{i,j}$ is the target for node $n_{i,j}$ (of graph $\boldsymbol{G}_i$), $p \leq |\mathcal{G}|$ 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 Internet (see Figure 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* $\boldsymbol{x}_n \in I\!R^s$ to each node n, that is based on the information contained in the neighborhood of n (see Figure 2). The variable $\boldsymbol{x}_n$ 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.

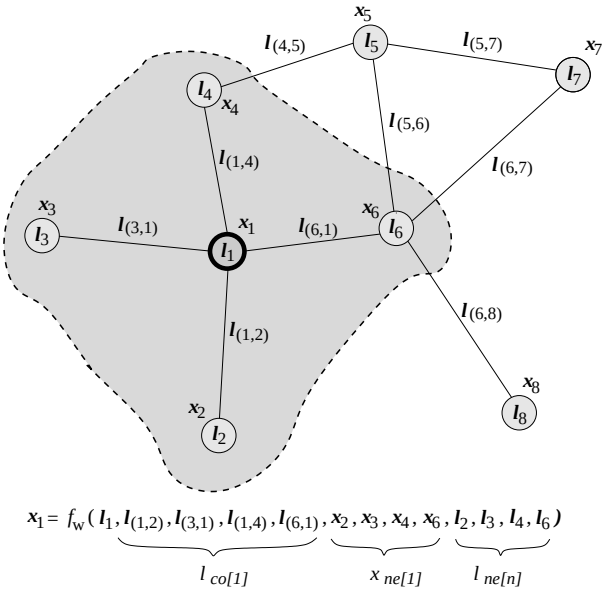

Fig. 2. The variable $\boldsymbol{x}_1$ depends on the information in the neighborhood of node 1.

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

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 also in $\boldsymbol{x}_{\mathrm{ne}[n]}$. Moreover, the neighborhood could contain nodes that are two or more links from n. In general, Eq. (1) could be simplified in several different ways and we could obtain a set of minimal models[4]. In the following, the discussion will mainly be based on the form defined by Eq. (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 Eq. (1). However, unless explicitly stated, all the results stated in this paper hold also for directed graphs and for 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 set of parameters $\boldsymbol{w}_{k_n}$. Thus, Eqs. (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 Eq. (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 variable $\boldsymbol{x}_n$, all the outputs, all the labels, and all the node labels, respectively. Eq. (1) can be re-written 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 Eq. (2) defines a map $\varphi_{\boldsymbol{w}} : \mathcal{D} \rightarrow I\!R^m$ which takes a graph input and returns an output $\boldsymbol{o}_n$ for each node. The Banach Fixed Point theorem [40] provides a

[4]By minimal, we address a model that has the smallest number of variables while retaining the computational power.

sufficient condition for the existence and uniqueness of the solution of a system of equations. According to the Banach theorem, Eq. (2) has a unique solution provided that $F_{\boldsymbol{w}}$ is a *contraction map* w.r.t. the state, i.e. there exists μ, $0 < \mu < 1$, 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 mapping. Later, we will show that, in GNNs, this property is enforced by an appropriate implementation of the transition function.

Note that Eq. (1) makes it possible to process both positional and non–positional 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 $\boldsymbol{x}_{\mathrm{ne}[n]}$, $\boldsymbol{l}_{\mathrm{co}[n]}$, $\boldsymbol{l}_{\mathrm{ne}[n]}$ are sorted according to neighbor positions and is properly padded with special null values in positions corresponding to non–existing 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 the node n; $\boldsymbol{y}_i = \boldsymbol{x}_u$ holds, if u is the i–the 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 i–the neighbor.

However, for non-positional graphs it is useful to replace function $f_{\boldsymbol{w}}$ of Eq. (1) with

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

where $h_{\boldsymbol{w}}$ is a parametric function. This transition function, which has been successfully used in recursive neural networks [41], is not affected by the positions and the number of the children. In the following, Eq. (3) is referred to as the *Non–positional Form*, while Eq. (1) is called the *Positional Form*.

In order to implement the GNN model, the following items must be provided:

(1) A method to solve Eq. (1);

(2) A learning algorithm to adapt $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ 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 subsections.

B. Computation of the state

Banach's Fixed Point theorem 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 t-the iteration of $\boldsymbol{x}$. The dynamic system (4) converges exponentially fast to the solution of Eq. (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, Eq. (4) implements the Jacobi iterative method for solving non–linear equations [42]. Thus, the outputs and the states can be computed by iterating

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

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

Note that the computation described in Eq. (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 neural network model [14]. In order to build the encoding network, each node of the graph can be replaced by a unit computing the function $f_{\boldsymbol{w}}$ (see Figure 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 by Eq. (5). The output for node n is produced by another unit which implements $g_{\boldsymbol{w}}$.

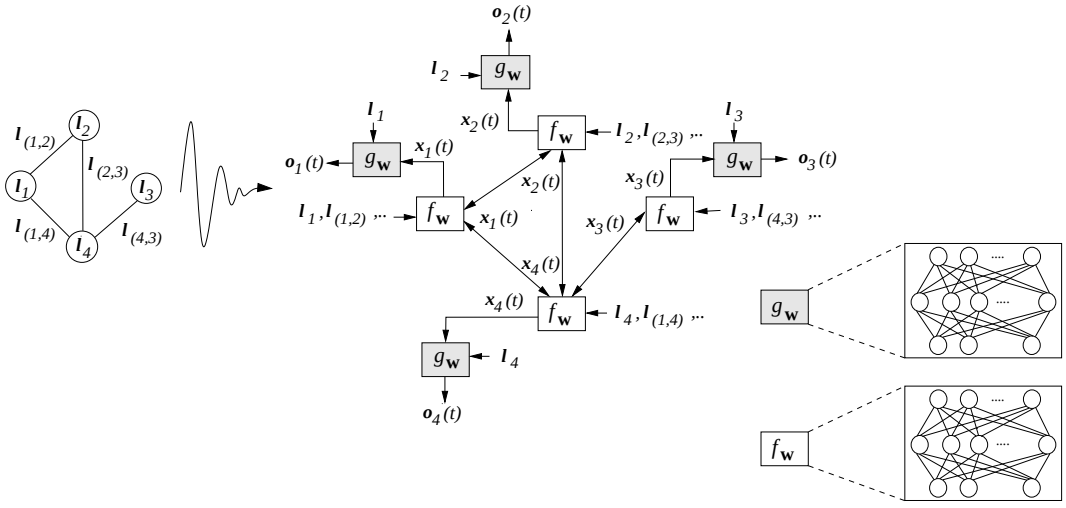

Fig. 3. A graph and its corresponding encoding network. When $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ are implemented by static neural networks, the encoding network is a recurrent neural network.

When $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ are implemented by static 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 is dependent 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 $\mathcal{L} = \{(\boldsymbol{G}_i, n_{i,j}, \boldsymbol{t}_{i,j})|, \boldsymbol{G}_i = (\boldsymbol{N}_i, \boldsymbol{E}_i) \in \mathcal{G}; n_{i,j} \in \boldsymbol{N}_i; \boldsymbol{t}_{i,j} \in I\!R^m, 1 \leq i \leq p, 1 \leq j \leq q_i\}$, 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 descendant strategy and is composed of the following steps:

(a) The states $\boldsymbol{x}_n(t)$ are iteratively updated by Eq. (5) until at time T they approach the fixed point solution of Eq. (2): $\boldsymbol{x}(T) \approx \boldsymbol{x}$;

(b) The gradient $\frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{w}}$ is computed;

(c) The weights $\boldsymbol{w}$ are updated according to the gradient.

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 [43], [14], [44]. In this case, the encoding network is unfolded from time T back to an initial time t_0. The unfolding produces the layered network as shown in Figure 4. Each layer corresponds to a time instance and contains a copy of all the units $f_{\boldsymbol{w}}$ of the encoding network. The units of two consecutive layers are connected following the graph connectivity. The last layer corresponding to time T includes also the unit $g_{\boldsymbol{w}}$ 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 all the instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$. Then, $\frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{w}}$ is obtained by summing the gradients of all instances.

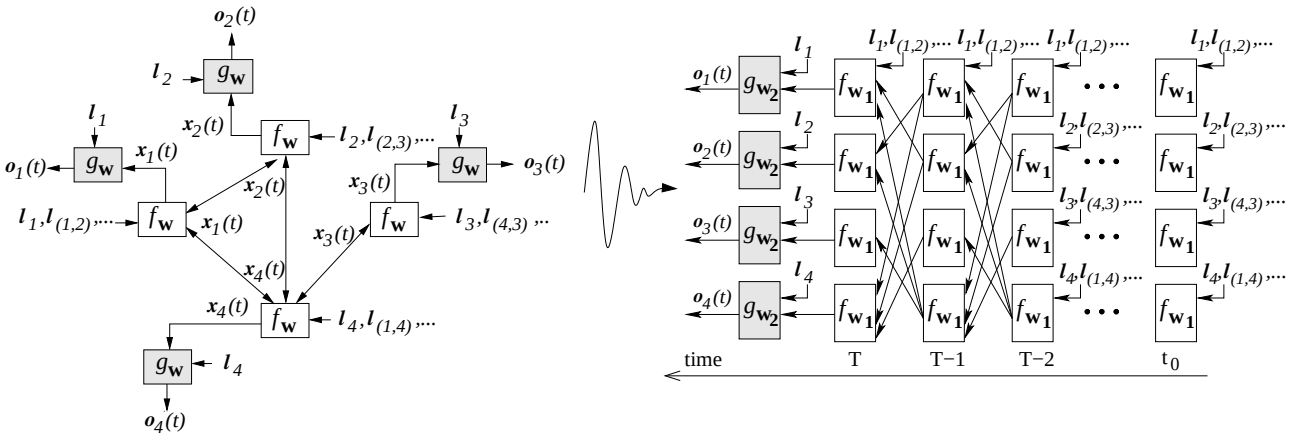

Fig. 4. The encoding network and the unfolding through time concept.

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 [45], [46]. Since Eq. (5) has reached a stable point $\boldsymbol{x}$ before the gradient computation, we can assume that $\boldsymbol{x}(t) = \boldsymbol{x}$ holds for any $t \geq t_0$. Thus, backpropagation through time can be carried out by storing only $\boldsymbol{x}$.

The folllowing 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 respectively the global transition and the global output functions of a GNN. 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 $\frac{\partial \Theta}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{I}_a - \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})$, where $\boldsymbol{I}_a$ denotes the a-dimension identity matrix and $a = s|\boldsymbol{N}|$, s is the dimension of the state. Since $F_{\boldsymbol{w}}$ is a contraction function, there exists $\mu, 0 \leq \mu < 1$ such that $\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\| \leq \mu$, which implies $\|\frac{\partial \Theta}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{w})\| > (1 - \mu)$. Thus, the determinant of $\frac{\partial \Theta}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{w})$ is not null and we can apply the implicit function theorem (see [47]) 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.

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 respectively the transition and the output functions of a GNN. 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

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

holds, where $\boldsymbol{x}$ is the stable state of the GNN and $\boldsymbol{z} = \lim_{t \to -\infty} \boldsymbol{z}(t)$ is the limit of the sequence $\boldsymbol{z}(T), \boldsymbol{z}(T-1), \ldots$ defined by

$$\boldsymbol{z}(t) = \boldsymbol{z}(t+1) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}). \tag{8}$$

[6]Internet applications where the graph may represent a portion of the web, are a straightforward example of cases when the amount of required storage may have a very important role.

Moreover, the convergence to $\boldsymbol{z}$ is exponentially fast and is independent of the initial state $\boldsymbol{z}(T)$.

Proof: Since $F_{\boldsymbol{w}}$ is a contraction mapping, there exists $\mu, 0 \leq \mu < 1$ such that $\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{w})\| \leq \mu$ holds. Thus, Eq. (8) converges to a stable fixed point for each initial state. The stable fixed point $\boldsymbol{z}$ is the solution of Eq. (8) 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 $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ can be computed using the chain rule for differentiation,

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

The theorem follows by putting together Eqs. (9), (10), and (11),

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

■

The relationship between the gradient defined by Eq. (7) and the gradient computed by the Almeida–Pineda algorithm can be easily recognized. The first term on the left hand side of Eq. (7) 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. 4). The second term represents the contribution due to the transition function $F_{\boldsymbol{w}}$. In fact, from Eq. (8),

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

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

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

Thus, Eq (8) accumulates the $\frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{x}(i)}$ into the variable $\boldsymbol{z}$. This mechanism corresponds to back propagate the gradients through the layers containing the $f_{\boldsymbol{w}}$ units.

The learning algorithm is detailed in Table I. It consists of a main procedure and of the two functions FORWARD and BACKWARD. Function FORWARD takes as input the current set of parameters $\boldsymbol{w}$ and iterates to find 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 (8) is iterated until $\|\boldsymbol{z}(t-1) - \boldsymbol{z}(t)\|$ is smaller than a threshold ε_b; then the gradient is calculated by Eq. (7). The main procedure updates the weights until the output reaches a desired accuracy or some other stopping criterion is achieved. More implementation details along with a computational cost analysis are included in Section III.

MAIN
  initialize $\boldsymbol{w}$;
  $\boldsymbol{x}$=FORWARD($\boldsymbol{w}$);
  repeat
    $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$=BACKWARD($\boldsymbol{x}, \boldsymbol{w}$);
    $\boldsymbol{w}$=$\boldsymbol{w} - \lambda \cdot \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$;
    $\boldsymbol{x}$=FORWARD($\boldsymbol{w}$);
  until (a stopping criterion);
  return $\boldsymbol{w}$;
end

FORWARD($\boldsymbol{w}$)
  initialize $\boldsymbol{x}(0)$, $t = 0$;
  repeat
    $\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l})$;
    t=$t + 1$;
  until $\|\boldsymbol{x}(t) - \boldsymbol{x}(t-1)\| \leq \varepsilon_f$
  return $\boldsymbol{x}(t)$;
end

BACKWARD($\boldsymbol{x}$,$\boldsymbol{w}$)
  $\boldsymbol{o} = G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N})$;
  $\boldsymbol{A} = \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})$;
  $\boldsymbol{b} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l_N})$;
  initialize $\boldsymbol{z}(0)$, t=0;
  repeat
    $\boldsymbol{z}(t) = \boldsymbol{z}(t+1) \cdot \boldsymbol{A} + \boldsymbol{b}$;
    t=$t - 1$;
  until $\|\boldsymbol{z}(t-1) - \boldsymbol{z}(t)\| \leq \varepsilon_b$;
  $\boldsymbol{c} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N})$;
  $\boldsymbol{d} = \boldsymbol{z}(t) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$;
  $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \boldsymbol{c} + \boldsymbol{d}$;
  return $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$;
end

TABLE I

THE LEARNING ALGORITHM. THE FUNCTION FORWARD COMPUTES THE STATES, WHILE BACKWARD CALCULATES THE GRADIENT. THE PROCEDURE FORWARD MINIMIZE THE ERROR THE BY CALLING ITERATIVELY FORWARD AND BACKWARD/.

D. GNN 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 multi-layered 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 Eqs. (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 w.r.t. the state $\boldsymbol{x}$. In the followings, we describe two neural network models that fulfill this purpose using different strategies. These models are based on the non–positional form described by Eq. (3). It can be easily observed that there exist two corresponding models based on the positional form.

1) LINEAR (NON–POSITIONAL) GNN

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

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

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

where $\mu \in (0,1)$ and $\boldsymbol{\Xi} \stackrel{\Delta}{=} \mathrm{resize}\left(\phi_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u)\right)$, where $\mathrm{resize}(\cdot)$ denotes the operator that allocates the elements of a s^2-dimensional vector into a $s\times s$ matrix. 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 uses 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 the node and edge labels. Thus, $\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}} = \boldsymbol{A}$, and, by simple algebra,

$$\begin{aligned}\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|_1 &= \|\boldsymbol{A}\|_1 \leq \max_{u\in \boldsymbol{N}}\left(\sum_{n\in \mathrm{ne}[u]} \|\boldsymbol{A}_{n,u}\|_1\right) \\ &\leq \max_{u\in \boldsymbol{N}}\left(\frac{\mu}{s|\mathrm{ne}[u]|}\cdot \sum_{n\in \mathrm{ne}[u]} \|\boldsymbol{\Xi}\|_1\right) \leq \mu\,,\end{aligned}$$

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

2) NON–LINEAR (NON–POSITIONAL) GNN

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

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

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

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. $\boldsymbol{w}$ which has a small Jacobian function. For example, in our experiments, we use the penalty term: $p_{\boldsymbol{w}} = \sum_{i=1}^{s} L\left(\|\boldsymbol{A}^i\|_1\right)$, where $\boldsymbol{A}^i$ is the i–the column of $\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}$. In fact, such an expression is an approximation of $L\left(\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|_1\right) = L\left(\max_i \|\boldsymbol{A}^i\|_1\right)$.

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 [14] are a special case of GNNs, where

1) The input graph is a Directed Acyclic Graph (DAG);
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* s from which all the other nodes can be reached. This node s is typically used for output $\boldsymbol{o}_s$ (graph-focused tasks).

The neural architectures, which have been suggested for realizing $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, include multilayered FNNs [14], [16], cascade correlation [48], and self organizing maps [17], [49]. 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 the states. In fact, the states can be computed following a predefined ordering that is induced by the partial ordering of the DAG.

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 Web search engines [15], [18]. In random walks on graphs, the state $\boldsymbol{x}_n$ associated with a node is a real value and is described by:

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

where $\mathrm{pa}[n]$ is the set of parents of n, and $a_{n,i} \in I\!R$, $a_{n,i} \geq 0$ 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, Eq. (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 $\boldsymbol{x}$, Eq. (15) becomes $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{x}$ where $\boldsymbol{A} = \{a_{n,i}\}$ and $a_{n,i}$ is defined as in Eq. (15) if $i \in \mathrm{pa}[n]$ and $a_{n,i} = 0$ otherwise. It is easily verified that $\|\boldsymbol{A}\|_1 = 1$, where the 1–norm $\|\cdot\|_1$ is defined as $\|\boldsymbol{A}\|_1 = \max_i \sum_n |a_{n,i}|$. Markov chain theory

[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

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

instruction	**positional**	**non−linear**	**linear**	**execs.**
$\boldsymbol{z}(t+1)=\boldsymbol{z}(t)\cdot\boldsymbol{A}+\boldsymbol{b}$	$s\lvert\boldsymbol{E}\rvert$	$s\lvert\boldsymbol{E}\rvert$	$s\lvert\boldsymbol{E}\rvert$	$\mathrm{it_b}$
$\boldsymbol{o}=G_{\boldsymbol{w}}(\boldsymbol{x}(t),\boldsymbol{l}_{\boldsymbol{w}})$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_g$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_g$	1
$\boldsymbol{x}(t+1)=F_{\boldsymbol{w}}(\boldsymbol{x}(t),\boldsymbol{l})$	$\lvert\boldsymbol{N}\rvert\overrightarrow{C}_f$	$\lvert\boldsymbol{E}\rvert\overrightarrow{C}_h$	$s\lvert\boldsymbol{E}\rvert$ $\lvert\boldsymbol{N}\rvert\overrightarrow{C}_\rho+\lvert\boldsymbol{E}\rvert\overrightarrow{C}_\phi$	$\mathrm{it_f}$ 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

suggests that if there exists t such that all the elements of the matrix $\boldsymbol{A}^t$ are non–null, then Eq. (15) is a contraction map [50]. 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 COST 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 Eq. (1) are implemented by FNNs; *linear (non–positional) GNNs*; *non–linear (non–positional) 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 the floating point operations[9]

In the following, $\mathrm{it_l}$, $\mathrm{it_f}$, $\mathrm{it_b}$ 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 [51], respectively. Formally, given a

[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.

function $l_{\boldsymbol{w}} : I\!R^a \rightarrow I\!R^b$ implemented by a 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, $\boldsymbol{y} \in I\!R^a$ is the input vector and the row vector $\delta \in I\!R^b$ is a signal which 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 $\delta = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}}(\boldsymbol{y}) = 2(\boldsymbol{t} - \boldsymbol{o})$, where $\boldsymbol{o} = l_{\boldsymbol{w}}(\boldsymbol{y})$ and $\boldsymbol{t}$ is the vector of the desired output corresponding to input $\boldsymbol{x}$. On the other hand, $\delta\frac{\partial l_{\boldsymbol{w}}}{\partial \boldsymbol{y}}(\boldsymbol{y})$ is the gradient of $e_{\boldsymbol{w}}$ w.r.t. the network input and is easily computed 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 $l_{\boldsymbol{w}}$. 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.

Complexity of instructions

Instructions $\boldsymbol{z}(t+1) = \boldsymbol{z}(t) \cdot \boldsymbol{A} + \boldsymbol{b}$, $\boldsymbol{o} = G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N})$, *and* $\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l})$*:* Since $\boldsymbol{A}$ is a matrix having at most $2s|\boldsymbol{E}|$ non–null 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 $O(s|\boldsymbol{E}|)$ 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 non–linear GNNs, where $f_{\boldsymbol{w}}$, $h_{\boldsymbol{w}}$, $g_{\boldsymbol{w}}$ 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 Eq. (13) and the vectors $\boldsymbol{b}_n$ Eq. (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 $O(s|\boldsymbol{E}|)$, is executed at every step of the cycle in the function *Forward*.

Instruction $\boldsymbol{A} = \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})$*:* 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 zero otherwise. In the linear model, the matrices $\boldsymbol{A}_{n,u}$ correspond to those displayed Eq. (13) and used to calculate $\boldsymbol{x}(t)$. Thus, such an instruction has no cost in the linear GNNs.

In non–linear GNNs, $\boldsymbol{A}_{n,u} = \frac{\partial h_{\boldsymbol{w}}}{\partial \boldsymbol{x}_n}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u)$, is computed by appropriately exploiting the backpropagation procedure. More precisely, let $\boldsymbol{q}_i \in I\!R^s$ be a vector where all the components are zero except for the i–the 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 $\boldsymbol{A}^i_{n,u} = \boldsymbol{q}_i\frac{\partial l_{\boldsymbol{w}}}{\partial \boldsymbol{y}}(\boldsymbol{y})$, i.e. the i-the column of the Jacobian $\frac{\partial l_{\boldsymbol{w}}}{\partial \boldsymbol{y}}(\boldsymbol{y})$. Thus, $\boldsymbol{A}_{n,u}$ can be computed by applying *BP* on all the $\boldsymbol{q}_i$, i.e.

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

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

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 third row of Table II.

Computation of $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}}$ and $\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$: 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, $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}_k} = 2(\boldsymbol{t}_k - \boldsymbol{o}_{n_k})$, if n_k is a node belonging to the training set, and 0 otherwise. Thus, $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}}$ is easily calculated by $O(|\boldsymbol{N}|)$ operations.

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

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

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

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

where sgn is the sign function. Moreover, let $\boldsymbol{R}_{n,u}$ be a matrix whose element in position i, j is $\alpha_{u,j} \cdot \mathrm{sgn}(\boldsymbol{A}_{n,u}^{i,j})$ and let vec be the operator that takes a matrix and produces 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 $\frac{\partial \,\mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \boldsymbol{w}}$ depends on selected implementation of $h_{\boldsymbol{w}}$ or $f_{\boldsymbol{w}}$. For sake of simplicity, let us restrict our attention to non–linear 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 j-the 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, $\boldsymbol{a}_j$, $\boldsymbol{V}_j$, $\boldsymbol{t}_j$: $\boldsymbol{a}_1 = [\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u]$, $\boldsymbol{a}_2 = \boldsymbol{V}_1 \, \boldsymbol{a}_1 + \boldsymbol{t}_1$, $\boldsymbol{a}_3 = \boldsymbol{V}_2 \, \sigma_2(\boldsymbol{a}_2) + \boldsymbol{t}_2$, and $h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) = \sigma_3(\boldsymbol{a}_3)$. By the chain 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 derivate 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

[11] σ_j is a vectorial function that takes in input the vector of the activations of a layer and returns the vector of the outputs for the same layer.

to $\boldsymbol{x}_u$ to the hidden layer. The parameters $\boldsymbol{w}$ affects four components of $\mathrm{vec}(\boldsymbol{A}_{n,u})$, i.e. $\boldsymbol{a}_3$, $\boldsymbol{V}_2$, $\boldsymbol{a}_2$, $\overline{\boldsymbol{V}}_1$. Thus, by properties of derivatives for matrix products and the chain rule,

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

holds.

Moreover, 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 $\boldsymbol{v} \in I\!R^a$. By the Kronecker product property $\mathrm{vec}(\boldsymbol{AB}) = (\boldsymbol{B}' \otimes \boldsymbol{I}_a) \cdot \mathrm{vec}(\boldsymbol{A})$ holds for matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{I}_a$ having compatible dimensions [], 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. When the above values are multiplied by $(\mathrm{vec}\,(\boldsymbol{R}_{u,v}))'$, 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}} & \quad (18) \\ (\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}} & \quad (19) \\ (\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}} & \quad (20) \\ (\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}} \,, & \quad (21)\end{aligned}$$

where the mentioned Kronecker product properties have been used.

Thus, $(\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial \,\mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \boldsymbol{w}}$ is the sum of four contributions represented by (18), (19), (20) and (21). The second and the fourth term (Eqs. (19) and (21)) can be computed directly using the corresponding formulas. The

first one can be calculated by observing that $\sigma_3'(\boldsymbol{a}_3)$ is in a three layered FNN form which 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

$$(\mathrm{vec}\,(\boldsymbol{R}_{u,v}))' \cdot \frac{\partial\,\mathrm{vec}(\boldsymbol{A}_{n,u})}{\partial \sigma_3'(\boldsymbol{a}_3)} \frac{\partial \sigma_3'(\boldsymbol{a}_3)}{\partial \boldsymbol{w}} = BP_1(\bar{h}_{\boldsymbol{w}}, \boldsymbol{a}_1, \delta)\,, \tag{22}$$

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

The above described method includes two tasks: the matrix multiplications of Eqs. (18)-(21) and the back-propagation as defined by Eq. (22). The former task consists of several matrix multiplications. By inspection of Eqs. (18)-(21), the number of floating point operations is approximately estimated as $2s^2 + 12s \cdot \mathrm{hi}_h + 10s^2 \cdot \mathrm{hi}_h$ [12]. The second task has approximately the same cost as a backpropagation phase through the original function $h_{\boldsymbol{w}}$.

Thus, the complexity of computing $\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ is $O(|\boldsymbol{E}| \max(s^2 \cdot \mathrm{hi}_h, \overleftarrow{C}_h))$. Note, however, even if the sums in Eq. (17) ranges over all the arcs of the graph, only those arcs (n, u) such that $\boldsymbol{R}_{n,u} \neq \boldsymbol{0}$ need 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 $\frac{\partial p_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$ 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.

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

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

$$\begin{aligned} \boldsymbol{z}(t) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) &= \sum_{n \in \boldsymbol{N}} \boldsymbol{z}_n(t) \cdot \frac{\partial f_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{\mathrm{co}[n]}, \boldsymbol{x}_u, \boldsymbol{l}_{\mathrm{ne}[n]}) \\ &= \sum_{n \in \boldsymbol{N}} \mathrm{BP}_1(f_{\boldsymbol{w}}, \boldsymbol{y}, \boldsymbol{z}_n(t))\,, \end{aligned} \tag{23}$$

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,

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

where $\boldsymbol{y} = [\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u]$. Thus, Eqs. (23) and (24) provide a direct method to compute $\boldsymbol{d}$ in positional and non–linear GNNs, respectively.

[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 above defined and used in Eqs. (18)-(21), then computing $\mathrm{vec}(\boldsymbol{CD})\boldsymbol{P}_c$ costs only $2ab$ operations. 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.

For linear GNNs, let $h^i_{\boldsymbol{w}}$ denote the i-the output of $h^i_{\boldsymbol{w}}$ and note that $h^i_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, l_u) = \boldsymbol{b}^i_u + \sum_{j=1}^{s} \boldsymbol{A}^{i,j}_{n,u} \boldsymbol{x}^i_u = \rho^i_u(\boldsymbol{l}_n) + \frac{\mu}{s|\mathrm{ne}[u]|} \cdot \sum_{j=1}^{s} \boldsymbol{x}^j_u \cdot \phi^{i,j}_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u)$ holds, where $\boldsymbol{A}^{i,j}_{n,u}$ 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 Eq. (13)), respectively, while $\boldsymbol{b}^i_u$ is the i–the element of vector $\boldsymbol{b}^i_u$, ρ^i_u is the corresponding output of the forcing network (see Eq. (14)), and $\boldsymbol{x}^i_u$ is the i–the element of $\boldsymbol{x}_u$. Then,

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

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

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 non–linear GNNs, and $O(|\boldsymbol{N}|\overrightarrow{C}_g + \mathrm{it_f} \cdot |\boldsymbol{E}|s + |\boldsymbol{N}|\overrightarrow{C}_\rho + |\boldsymbol{E}|\overrightarrow{C}_\phi)$ 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) and the dimension of the employed FNNs. 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 sufficient to approximate the fixed point.

In positional and non–linear GNNs, the transition functions 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 neighbours 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 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 $O(s|\boldsymbol{E}|)$ instead of $O(|\boldsymbol{E}|\overrightarrow{C}_h)$. Note that $\overrightarrow{C}_h = O((s + l_E + 2l_N) \cdot \mathrm{hi}_h)$ holds, when $h_{\boldsymbol{w}}$ is implemented by a three layered FNN with hi_h hidden neurons. Thus, the linear model is always faster than the non–linear model. As confirmed by the experiments, such an advantage is mitigated by the smaller accuracy which 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 the backward phases for several epochs. The experiments have shown that the time spent in the forward and the backward phases are not very different [13]. Similar 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 exception is due to the possible computation of $\frac{\partial p_{\boldsymbol{w}}}{\boldsymbol{w}}$ which depends quadratically on s.

In fact, the non–linear GNNs needs to calculate the term $\frac{\partial p_{\boldsymbol{w}}}{\boldsymbol{w}}$, which costs $O(\mathrm{t}_{\boldsymbol{R}} \cdot \max(s^2 \cdot \mathrm{hi}_h, \overleftarrow{C}_h))$. 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 $\frac{\partial p_{\boldsymbol{w}}}{\boldsymbol{w}}$ requires few applications of backpropagation on h and has a small impact on the global complexity of the learning. On the other hand, in theory, if s is very large, it might happen that $s^2 \cdot \mathrm{hi}_h >> \overleftarrow{C}_h \approx (s + l_E + 2l_N) \cdot \mathrm{hi}_h$ and at the same time $\mathrm{t}_{\boldsymbol{R}} >> 0$ rendering the computation of the gradient very slow. However, it is worth mentioning that this case was never observed in our experiments.

IV. Experimental results

GNNs have been experimentally evaluated to assess their performance. In this paper, we present the results obtained on a set of small problems. Other examples of applications based on GNNs can be found in [52], [53], [54], [55], [56].

In particular the experiments have been carried out with both linear and non–linear GNNs. Since according to existing results on recursive neural networks, the non–positional transition function slightly outperforms positional one, currently only non–positional GNNs have been implemented and tested.

The following facts hold for each experiment, unless otherwise specified. Both the (non–positional) linear and the non–linear 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 $g_{\boldsymbol{w}}$, $h_{\boldsymbol{w}}$ for GNNs were implemented by three layered FNNs with sigmoidal activation functions.

The presented results were averaged over five different runs. In each run, the dataset 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 or not and if it was not, random edges were inserted until the condition was satisfied.

The dataset was split into a training set, a validation set and a test set and the validation set is used to avoid possible issues with overfitting. For the problems where the original data is only one single big graph $\boldsymbol{G}$, the training set, the validation set and the test set include different supervised nodes of $\boldsymbol{G}$. Otherwise, when several graphs

[13] More precisely, the time spent in function *Backward* is usually 1.5-2 times larger than the time spent in function *Forward*.

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 $5,000$ 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.

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 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[14] and the software can be freely downloaded, together with the source and some examples [57]. 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. 5). Subgraph matching has a number of practical applications, such as object localization and detection of active parts in chemical compounds [58], [59], [60]. Very often the subgraph is not exactly known in advance and is available only from a set of examples corrupted by noise.

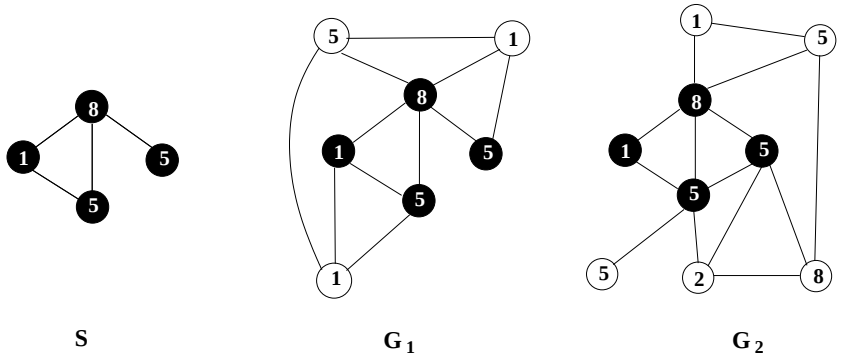

Fig. 5. Two graphs $\boldsymbol{G}_1$, $\boldsymbol{G}_2$ that contain a subgraph $\boldsymbol{S}$. The numbers inside the nodes represent the labels. The function τ to be learned is $\tau(\boldsymbol{G}_i, n_{i,j}) = 1$, if $n_{i,j}$ is a dark gray node, and $\tau(\boldsymbol{G}_i, n_{i,j}) = -1$, if $n_{i,j}$ is a light gray node.

In our experiments, the dataset $\mathcal{L}$ consisted of 600 connected random graphs (constructed using $\delta = 0.2$), equally divided into a training, a validation and a test set. A smaller subgraph $\boldsymbol{S}$, which was randomly generated in each trial, was inserted into every graph of the dataset. Thus, each graph $\boldsymbol{G}_i$ contained at least a copy of $\boldsymbol{S}$, even if

[14]Copyright © 1994-2006 by The MathWorks, Inc.

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 dataset were different due to the noise introduced.

In all the experiments, the state dimension was $s = 5$ and all the neural networks involved in the GNNs had 5 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 (FNN) 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 non–linear GNN model (non–linear), 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 dataset 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 the label contents and the graph topology at the same time. Moreover, the non–linear GNN model achieved a slightly better performance than the linear one, probably because non–linear 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 fifty per cent larger than the GNN error (13.7 for non–linear GNNs, 14.6 for linear GNNs and 22.8 for FNNs). Actually, the relative difference between the GNN and the FNN errors, which measures the advantage provided by the topology, tend to become smaller for larger values of $|\boldsymbol{S}|$ (see last column of Table III). In fact, GNNs use an information diffusion mechanism to decide whether a node belongs or not 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.

B. The Mutagenesis problem

The Mutagenesis dataset [61] is a small dataset, 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 [62]. 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.

In [62] it is shown that 188 molecules out of 230 are amenable to a linear regression analysis. This subset was

			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				

TABLE III

THE ACCURACIES ACHIEVED BY NON–LINEAR MODEL (NL), THE LINEAR MODEL (L) AND A FEEDFORWARD NEURAL NETWORK (FNN) ON SUBGRAPH MATCHING PROBLEM.

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 structure (AB), each compound is provided with four global features [62]. The first two features are chemical measurements (C): the lowest unoccupied molecule orbital and the water/octanol partition coefficient, while the remaining two are pre-coded structural attributes (PS). Finally the atom-bond 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 atom–bond description.

In our experiments, each molecule of the dataset was transformed into a graph where nodes represent atoms and edges stand for atom–bonds. The average number of nodes in a molecule is around 26. Node labels contain the type of the atom, its energy state and the global properties AB, C and PS[15] . In each graph there is only one supervised node, the first atom in the atom-bond description (Fig. 6). The desired output is positive, if the molecule

[15]Our best results were achieved without the functional groups. Probably, the reason is that GNNs can recover the substructures that are relevant to the classification, exploiting the graphical information contained in the atom–bond description.

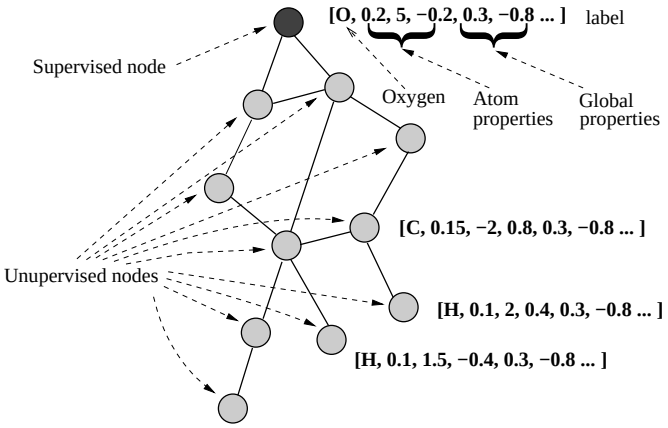

Fig. 6. The atom-bond structure of a molecule represented by a graph with labelled nodes.

is mutagenic, and negative, otherwise.

In Tables IV,V and VI, the results obtained by non–linear GNNs[16] are compared with those achieved by other methods. The presented results were evaluated using a 10-fold cross-validation procedure, i.e. the dataset was randomly split into 10 parts and the experiments were repeated 10 times, each time using a different part as test set and the remaining patterns as training set. The results were averaged on 5 runs of the cross-validation procedure.

GNNs achieved the best accuracy on the regression–unfriendly part (Table V) and on the whole dataset (Table VI), whereas the results are close to the state of the art on the regression–friendly part (Table IV). Surprisingly, the accuracy is better on the unfriendly part, instead of the friendly subset. This fact may be due to small number of patterns contained in the friendly part (42 patterns), which does not allow a good generalization. GNNs suffers from this problem more than the other techniques, because other methods are mainly based on inductive logic programming instead of neural networks.

Moreover, whereas most of the approaches showed a higher level of accuracy when applied to the whole dataset with respect to the unfriendly part, the converse holds for GNNs. This suggests GNNs that can capture characteristics of the patterns that are useful to solve the problem but are not homogeneously distributed in the two parts.

C. Web page ranking

In this experiment, the goal is to learn a web page ranking algorithm, inspired by Google's PageRank [15]. According to PageRank a page is considered authoritative if it is referred by many other pages and if the referring pages are authoritative. Formally, the PageRank p_n of a page n is $p_n = d\sum_{u\in \mathrm{pa}[n]} \frac{p_u}{o_n} + (1-d)$, where o_n is the outdegree of n, and $d \in [0,1]$ is the damping factor [15]. In this experiments, 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 $\boldsymbol{G}$ containing 5000 nodes was generated, with $\delta = 0.2$. Training, validation and

[16]Some results were already presented in [63].

TABLE IV

THE RESULTS ON THE REGRESSION-FRIENDLY PART OF THE MUTAGENESIS DATASET

Method	Knowledge	Reference	Accuracy
non–linear GNN	AB+C+PS		94.3
Neural Networks	C+PS	[61]	89.0%
P-Progol	AB+C	[61]	82.0%
P-Progol	AB+C+FG	[61]	88.0%
MFLOG	AB+C	[64]	95.7%
FOIL	AB	[65]	76%
boosted-FOIL	not available	[66]	88.3%
$1nn(d_m)$	AB	[67]	83
$1nn(d_m)$	AB+C	[67]	91%
RDBC	AB	[68]	84%
RDBC	AB+C	[68]	83%
RSD	AB+C+FG	[69]	92.6%
SINUS	AB+C+FG	[69]	84.5%
RELAGGS	AB+C+FG	[69]	88.0%
RS	AB	[70]	88.9%
RS	AB+FG	[70]	89.9%
RS	AB+C+PS+FG	[70]	95.8%
SVM_P	not available	[71]	91.5

TABLE V

THE RESULTS ON THE REGRESSION-UNFRIENDLY PART OF THE MUTAGENESIS DATASET

Method	Knowledge	Reference	Accuracy
non–linear GNN	AB+C+PS		96.0%
$1nn(d_m)$	AB	[67]	72%
$1nn(d_m)$	AB+C	[67]	72%
TILDE	AB	[72]	85%
TILDE	AB+C	[72]	79%
RDBC	AB	[68]	79%
RDBC	AB+C	[68]	79%

test set 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 was 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

TABLE VI
THE RESULTS ON THE WHOLE MUTAGENESIS DATASET

Method	Knowledge	Reference	Accuracy
non–linear GNN	AB+C+PS		90.5%
$1nn(d_m)$	AB	[67]	81%
$1nn(d_m)$	AB+C	[67]	88%
TILDE	AB	[72]	77%
TILDE	AB+C	[72]	82%
RDBC	AB	[68]	83%
RDBC	AB+C	[68]	82%

was trained in order to produce the following output: $\tau(\boldsymbol{G}, n) = \begin{cases} 2\frac{p_n}{\|\boldsymbol{p}\|_1} & \text{if } (a_n \,\mathrm{XOR}\, b_n) = 1 \\ \frac{p_n}{\|\boldsymbol{p}\|_1} & \text{otherwise} \end{cases}$ where $\boldsymbol{p}$ stands for the Google's PageRank.

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 [73], [74], [75], [76].

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 the forcing networks (see Section 1) were implemented by three layered neural networks with 5 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 5 hidden neurons.

Figure 7 shows the output of the GNN φ and the target function τ on test set. Plot **A** displays the result for the pages that belong to only one topic and Plot **B** 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 vertical axes denote the value of function τ (continuous lines) and the value of the function implemented by the GNN (the dotted lines). The two plots show clearly that GNN performs very well.

Finally, Fig. 8 displays the error functions 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.

D. The parity problem

Whereas in the web page ranking experiments, the network output mainly depended on the graph connectivity, in this problem the converse holds and the output depends only on the node labels. In fact, the purpose of this experiment is to verify whether the GNN model is capable of discarding information contained in the topology of

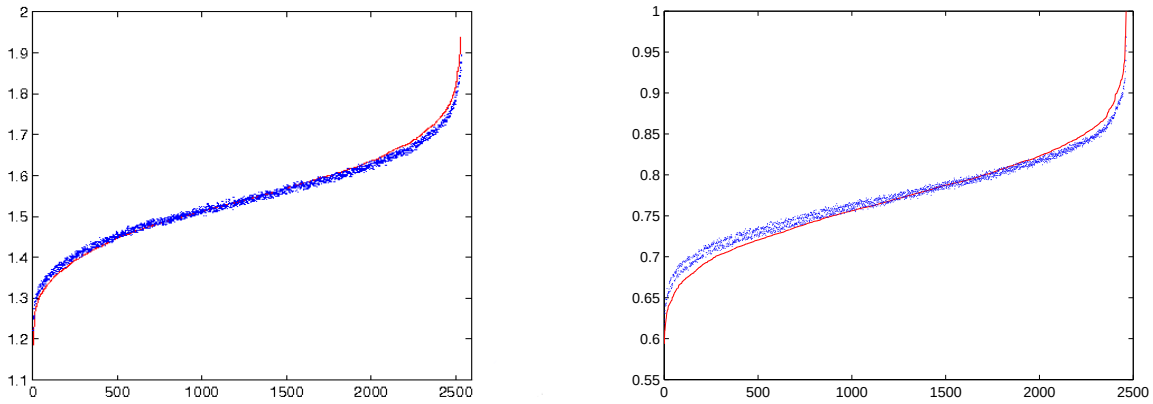

Fig. 7. The desired function τ (the continuous lines) and the output of the GNN (the dotted lines) on the pages that belong to only one topic (Plot **A**) and on the other pages (Plot **B**). Horizontal axis stands for pages, vertical axis for scores. Pages have been sorted according to the desired value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

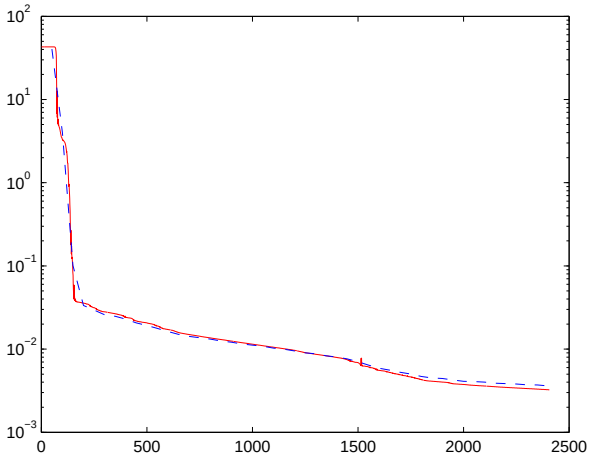

Fig. 8. The plots of the error function on the training set (continuous line) and on the validation (dashed line) set during learning phase.

the graphs, when such information is not needed to solve the problem.

Train and validation sets contained 500 graphs, while the test set consisted of $2,000$ graphs. Each node was labelled with a random vector of eight binary integers, i.e. $\boldsymbol{l}_{n_{i,j}} = [\lambda_1, \ldots, \lambda_8]$, $\lambda_k \in \{0, 1\}$. The function to be learned is: $\tau(\boldsymbol{G}, n_{i,j}) = 1$, if $\boldsymbol{l}_{n_{i,j}}$ contains an even number of ones, and $\tau(\boldsymbol{G}, n_{i,j}) = -1$, otherwise.

The results are shown in Table VII. Note that the output network $g_{\boldsymbol{w}}$, which has to approximate the parity function, must contain a sufficient number of hidden neurons. It is observed that in this case, it requires 10 hidden neurons in each output network to achieve an accuracy of larger than 97%.

For comparison purposes, a three layer FNN with 20 hidden neurons was applied on this task (see Table VII). The difference between the accuracy achieved by the FNN and the GNNs provides an insight of the impact due to the noise introduced in GNNs by the topology of the graphs. This impact is larger for the non–linear model that is more general and can realize a larger set of functions. However, the results confirms that GNNs are able to learn a function that depends only on node labels.

TABLE VII
THE PARITY PROBLEM

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

V. CONCLUSIONS

In this paper, we have introduced a new neural network model which can handle graph inputs: the graphs can be acyclic, cyclic, directed, un-directed. A learning algorithm is furnished to estimate the parameters of the neural networks based on the back propagation techniques. The computational complexity of the learning algorithm is considered. Moreover, some promising experimental results have been provided to asses the model.

VI. ACKNOWLEDGEMENT

The authors acknowledge financial support from the Australian Research Council in the form of an International Research Exchange scheme which facilitated the visit by the first author to University of Wollongong when the initial work on this paper was performed.

REFERENCES

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

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

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

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

[5] L. Baresi and R. Heckel, "Tutorial introduction to graph transformation: A software engineering perspective," in *Lecture Notes in Computer Science (ICGT 2002)*, 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. of the 2003 ACM symposium on Software visualization SoftVis 2003*. 2003, ACM Press.

[7] A. Bua, M. Gori, and F. Santini, "Recursive neural networks applied to discourse representation theory," in *Lecture Notes in Computer Science (ICANN'02)*, 2002, vol. 2415.

[8] L. De Raedt, *Logical and Relational Learning: From Inductive Logic Programming to Multi-Relational Data Mining*, Springer, 2006, in press.

[9] "International workshop on statistical relational learning and its connections to other fields (srl2004)," in *ICML-2004*, T Dietterich, L. Getoor, and K. Murphy, Eds.

[10] "International workshop on sub-symbolic paradigms in structured domains (rml2005)," in *ECML-PKDD 2005*, P. Avesani and M. Gori, Eds.

[11] "Third international workshop on mining graphs, trees, and sequences (mgts2005)," in *ECML-PKDD 2005*, S. Nijseen, Ed.

[12] "Fourth international workshop on mining graphs, trees, and sequences (mgts2006)," in *ECML-PKDD 2006*, T. Gaertner, G. Garriga, and T. Meini, Eds.

[13] T. Pavlidis, *Structural pattern recognition*, Springer, Series in Electrophysics, 1977.

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

[15] S. Brin and L. Page, "The anatomy of a large–scale hypertextual Web search engine," in *Proceedings of the 7th World Wide Web Conference*, Apr. 1998.

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

[17] M. Hagenbuchner, A. Sperduti, and A. C. Tsoi, "A self-organizing map for adaptive processing of structured data," *IEEE Transactions on Neural Networks*, 2003.

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

[19] A. C. Tsoi, G. Morini, F. Scarselli, M. Hagenbuchner, and M. Maggini, "Adaptive ranking of web pages," in *Proceedings of the 12th WWW Conference*, Budapest, Hungary, May 2003.

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

[21] M. Bianchini, M. Gori, and F. Scarselli, "Processing directed acyclic graphs with recursive neural networks," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1464–1470, 2001.

[22] A. Küchler and C. Goller, "Inductive learning in symbolic domains using structure-driven recurrent neural networks," in *20th Annual German Conference on Artificial Intelligence*, G. Görz and S. Hölldobler, Eds., Dresden, Germany, Sept. 1996, vol. 1137 of *Lecture Notes in Computer Science*, Springer.

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

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

[25] M. Gori, M. Maggini, E. Martinelli, and F. Scarselli, "Learning user profiles in NAUTILUS," in *International Conference on Adaptive Hypermedia and Adaptive Web–based Systems*, Trento (Italy), August 2000.

[26] M. Bianchini, P. Mazzoni, L. Sarti, and F. Scarselli, "Face spotting in color images using recursive neural networks," in *Proceedings of WIRN03*, Vietri sul Mare (Italy), July 2003.

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

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

[29] T. Gärtner, J.W. Lloyd, and P.A. Flach, "Kernels and distances for structured data," *Machine Learning*, vol. 57, no. 3, pp. 205–232, 2004.

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

[31] H. Kashima, K. Tsuda, and A. Inokuchi, "Marginalized kernels between labeled graphs," in *Proc. 20th International Conference on Machine Learning (*, T. Fawcett and N. (eds) Mishra, Eds. 2003, pp. 321–328, AAAI Press.

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

[33] M. Collins and N. Duffy, "Convolution kernels for natural language," in *Advances in Neural Information Processing Systems 14, Proc. of NIPS 2001*, T. G. Dietterich, S. Becker, and Z. Ghahramani, Eds. 2002, pp. 625–632, MIT Press.

[34] J. Suzuki, Y. Sasaki, and E. Maeda, "Kernels for structured natural language data.," in *NIPS*, 2003.

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

[36] J. Cho, H. Garcia-Molina, and L. Page, "Efficient crawling through url ordering," in *Proceedings of the 7th World Wide Web Conference*, Brisbane, Australia, Apr. 1998.

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

[38] H. Chang, D. Cohn, and McCallum A. K., "Learning to create customized authority lists," in *Proceedings of the 17th International Conference on Machine Learning*. 2000, pp. 127–134, Morgan Kaufmann.

[39] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "Computation capabilities of graph neural networks," *IEEE Transactions on Neural Networks*, 2007, Submitted.

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

[41] M. Bianchini, M. Maggini, L. Sarti, and F. Scarselli, "Recursive neural networks for processing graphs with labelled edges: Theory and applications," *Neural Networks - Special Issue on Neural Networks and Kernel Methods for Structured Domains*, 2005, to appear.

[42] 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.

[43] W. T. Miller III, R. Sutton, and P. Ed. Werbos, *Neural Network for Control*, MIT Press, Camrbidge, Mass., 1990.

[44] A. C. Tsoi, "Adaptive processing of sequences and data structures, international summer school on neural networks, "e.r. caianiello", vietri sul mare, salerno, italy, september 6-13, 1997, tutorial lectures," in *Summer School on Neural Networks*, C. Lee Giles and Marco Gori, Eds. 1998, vol. 1387 of *Lecture Notes in Computer Science*, pp. 27–62, Springer.

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

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

[47] W. Rudin, *Real and Complex Analysis, Third Edition*, McGraw Hill, New York, 1987.

[48] 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," *Journal of Chemical Information and Computer Sciences*, vol. 41, no. 1, pp. 202–218, 2001.

[49] M. Hagenbuchner, A. C. Tsoi, and A. Sperduti, "A supervised self-organising map for structured data," in *WSOM 2001 - Advances in Self-Organising Maps*, N.Allinson, H.Yin, L.Allinson, and J.Slack, Eds. June 2001, pp. 21–28, Springer.

[50] E. Seneta, *Non–negative matrices and Markov chains*, Springer Verlag, 1981, Chapter 4, pages 112–158.

[51] D. E. Rumelhart, J.L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, MIT Press, Cambridge, 1986.

[52] F. Scarselli, S.L. Yong, M. Gori, M. Hagenbuchner, A. C. Tsoi, and M. Maggini, "Graph neural networks for ranking web pages," in *Proc. of the 2005 IEEE/WIC/ACM Conference on Web Intellligence*, 2005.

[53] M. Gori, M. Hagenbuchner, F. Scarselli, and A. C. Tsoi, "Graphical-based learning environment for pattern recognition," in *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshops, SSPR 2004 and SPR 2004. Lecture Notes in Computer Science*, 2004, vol. 3138, pp. 42–56.

[54] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings of the International Joint Conference on Neural Networks*, 2005.

[55] 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 *International Joint Conference on Neural Networks*, July 2006.

[56] G. Monfardini, V. Di Massa, F. Scarselli, and M. Gori, "Graph neural networks for object localization," in *17-th European Conference on Artificial Intelligence*, August 2006.

[57] The GNN toolbox, ," Available at *http://airgroup.dii.unisi.it/projects/GraphNeuralNetwork/download.htm*.

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

[59] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Graph matching applications in pattern recognition and image processing," in *International Conference on Image Processing*, September 2003, vol. 2, pp. 21–24.

[60] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 3, pp. 265–268, 2004.

[61] A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg, "Mutagenesis: Ilp experiments in a non-determinate biological domain," in *Proceedings of the 4th International Workshop on Inductive Logic Programming*, 1994, Gesellschaft für Mathematik und Datenverarbeitung MBH, pp. 217–232.

[62] A. K. Debnath, R.L. Lopex de Compandre, G. Debnath, A.J. Schusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity," *Journal of Medicinal Chemistry*, vol. 34, no. 2, pp. 786–797, 1991.

[63] W. Uwents, G. Monfardini, H. Blockeel, F. Scarselli, and M Gori, "Two connectionist models for graph processing: an experimental comparison on relational data," in *European Conference on Machine Learning*, 2006.

[64] S. Kramer and L. De Raedt, "Feature Construction with Version Spaces for Biochemical Applications," *Proceedings of the Eighteenth International Conference on Machine Learning table of contents*, pp. 258–265, 2001.

[65] J.R. Quinlan and R.M. Cameron-Jones, "FOIL: A midterm report," *Proceedings of the European Conference on Machine Learning*, pp. 3–20, 1993.

[66] J.R. Quinlan, "Boosting first-order learning," *LNCS*, vol. 1160, pp. 143, 1996.

[67] J. Ramon, *Clustering and instance based learning in first order logic*, Ph.D. thesis, K.U. Leuven, Belgium, 2002.

[68] M. Kirsten, *Multirelational Distance-Based Clustering*, Ph.D. thesis, School of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2002.

[69] M.A. Krogel, S. Rawles, F. Zelezny, P. Flach, N. Lavrac, and S. Wrobel, "Comparative evaluation of approaches to propositionalization," *Proc. 13th Int. Conf. on Inductive Logic Programming*, pp. 197–214, 2003.

[70] S. Muggleton, "Machine learning for systems biology," in *15th International Conference on Inductive Logic Programming (ILP 2005) Bonn, Germany, August 10 - 13, 2005*, 2005.

[71] A. Woźnica, A. Kalousis, and M. Hilario, "Matching based kernels for labeled graphs," in *Proceedings of the International Workshop on Mining and Learning with Graphs (MLG 2006) in conjunction with ECML/PKDD 2006*, T. Gärtner, G.C. Garriga, and T. Meinl, Eds., 2006, pp. 97–108.

[72] L. De Raedt and H. Blockeel, "Using logical decision trees for clustering," in *Proceedings of the 7th International Workshop on Inductive Logic Programming ILP 1997*. 1997, vol. 1297 of *Lecture Notes in Artificial Intelligence*, pp. 133–141, Springer-Verlag.

[73] M. Diligenti, M. Gori, and M. Maggini, "Web page scoring systems for horizontal and vertical search," in *Proceedings of the 11th World Wide Web Conference*, 2002.

[74] T. H. Haveliwala, "Topic sensitive pagerank," in *Proceedings of the 11th World Wide Web Conference (WWW11)*, 2002, Available on the Internet at http://dbpubs.stanford.edu:8090/pub/2002-6.

[75] G. Jeh and J. Widom, "Scaling personalized web search," in *Proceedings of the 12th World Wide Web Conference*, 20–24May 2003.

[76] F. Scarselli, A. C. Tsoi, and M. Hagenbuchner, "Computing personalized pagerankss," in *Proceedings of the 12th WWW Conference*, 2003.