Graphical-Based Learning Environments for Pattern Recognition

Franco Scarselli[1], Ah Chung Tsoi[3], Marco Gori[1], and Markus Hagenbuchner[2]

[1] Dipartimento di Ingegneria dell'Informazione
University of Siena, Siena, Italy
[2] Faculty of Informatics
University of Wollonong, Wollongong, Australia
[3] Executive Director, Mathematics, Informatics & Comunication–Science
Australian Research Council, Camberra, Australia

Abstract. In this paper, we present a new neural network model, called graph neural network model, which is a generalization of two existing approaches, viz., the graph focused approach, and the node focused approach. The graph focused approach considers the mapping from a graph structure to a real vector, in which the mapping is independent of the particular node involved; while the node focused approach considers the mapping from a graph structure to a real vector, in which the mapping depends on the properties of the node involved. It is shown that the graph neural network model maintains some of the characteristics of the graph focused models and the node focused models respectively. A supervised learning algorithm is derived to estimate the parameters of the graph neural network model. Some experimental results are shown to validate the proposed learning algorithm, and demonstrate the generalization capability of the proposed model.

1 Introduction

In several applications, the data can be naturally represented by graph structures. The simplest kind of graph structures is a sequence, but, in many application domains, the information is organized in more complex graph structures such as trees, acyclic graphs, or cyclic graphs. In machine learning, the structured data is often associated with the goal of either supervised or unsupervised learning from examples, a function $\boldsymbol{h}$ which maps a graph G and one of its nodes n to a vector of reals[1]: $\boldsymbol{h}(G,n) \in \mathcal{R}^m$.

In general, applications to a graphical domain can be divided into two classes: called *graph focused* and *node focused* applications, respectively.

In graph focused applications, $\boldsymbol{h}$ is independent of the node n and implements a classifier or a regressor on a graph structured dataset. For example, an image can be represented by a Region Adjacency Graph (RAG) where the nodes

[1] Note that in classification problems, the mapping is to a set of integers $\mathcal{I}^m$, while in regression problems, the mapping is to a set of reals $\mathcal{R}^m$. Here for simplicity of exposition, we will denote only the regression case.

A. Fred et al. (Eds.): SSPR&SPR 2004, LNCS 3138, pp. 42–56, 2004.
© Springer-Verlag Berlin Heidelberg 2004

denote homogeneous regions of the image and the arcs represent their adjacency relationship (see Fig. 1(a)). In this case, $\boldsymbol{h}(G)$ may be used to classify the image into different classes, e.g., castles, cars, people, and so on.

In node focused applications, $\boldsymbol{h}$ depends on n, so that the classification (or the regression) relates to each node. Object localization is an example of this class of applications. It consists of finding whether an image contains a given object or not, and, if so, detect its position. This problem can be solved by a function $\boldsymbol{h}$ which classifies the nodes of the RAG according to whether the corresponding region belongs to the object or not. For example, in Fig. 1(a), the output of $\boldsymbol{h}$ might be 1 for the black nodes, which correspond to the castle, and 0 otherwise. Another example comes from web page classification. The web can be represented by a graph where nodes stand for pages and edges represent the hyperlinks (see Fig. 1(b)). The web connectivity can be exploited, along with page contents, for several purposes, e.g. classifying the pages into a set of topics.

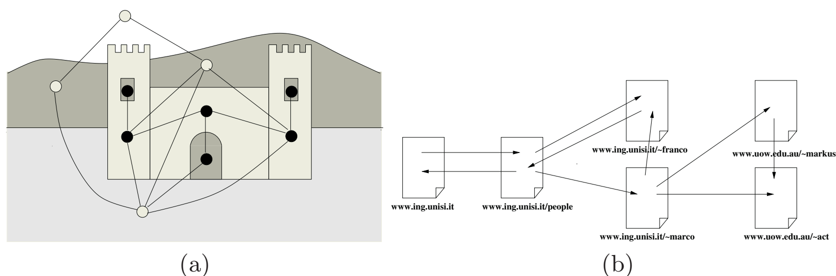

Fig. 1. Some applications where the information is represented by graphs: (a) an image; and (b) a subset of the web.

Most applications cope with graph structured data using a preprocessing phase which maps the graph structured information to a simpler representation, e.g. vectors of reals. 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 [3, 1] attempting to preserve the graph structured nature of the data, for as long as required, before processing the data. In other words, these approaches attempt to avoid the preprocessing step of "squashing" the graph structured data into a vector of reals first, and then deal with the preprocessed data using a list based data processing technique, rather than paying special attention to the underlying graph structured nature of the data. In these recent approaches, the idea is to encode the underlying graph structured data using the topological relationship among the nodes of the graph. In other words, these recent approaches attempt to incorporate the graph structured information in the data processing step. In the graph focused approaches [3, 11, 5] this is done using *recursive neural networks* and in the node focused approaches [1, 8, 12], this is done commonly by using *random walk* techniques.

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 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.

The structure of this paper is as follows: The notation used in this paper as well as preliminary materials are described in Section 2. Then, the concept of a graph neural network model, together with a learning algorithm for the parameter estimation of the model are presented in Section 3. Furthermore, some experimental results are presented in Section 4, and some conclusions are drawn in Section 5.

2 Notation and Preliminaries

In the following, $\|\cdot\|_1$ denotes norm 1, i.e. for a vector $\boldsymbol{V} = [v_1, .., v_k]$, $\|\boldsymbol{V}\|_1 = \sum_{i=1}^{k} |v_i|$, for a matrix $\boldsymbol{M} = \{m_{i,j}\}$, $\|\boldsymbol{M}\|_1 = \max_j \sum_i |m_{i,j}|$. A graph G is a pair $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N} = \{n_1, \ldots, n_r\}$ is a set of nodes and $\mathcal{E} = \{e_1, \ldots, e_p\}$ a set of edges. The set of children and parents of n are denoted by $\mathrm{ch}[n]$ and $\mathrm{pa}[n]$, respectively. The set $\mathrm{ne}[n]$ stands for the nodes connected to n by an arc: for directed graphs, we have $\mathrm{ne}[n] = \mathrm{pa}[n] \cup \mathrm{ch}[n]$, and for undirected graphs, $\mathrm{ne}[n] = \mathrm{pa}[n] = \mathrm{ch}[n]$ holds. Similarly, the set of arcs entering and emerging from node n are represented by $\mathrm{to}[n]$ and $\mathrm{from}[n]$, respectively, while $\mathrm{co}[n]$ represents their union. Nodes and edges may have labels, which we assume to be represented by real vectors. The labels attached to node n and edge (n_1, n_2) will be represented by $L_n \in \mathcal{R}^{l_N}$ and $L_{(n_1,n_2)} \in \mathcal{R}^{l_E}$ respectively. Given a set of integers $\mathcal{S}$ and a set of vectors $\boldsymbol{y}_i, i \in \mathcal{S}$, $\boldsymbol{y}_{\mathcal{S}}$ denotes the vector obtained by stacking together the $\boldsymbol{y}_i$. Thus, for example, $L_{\mathrm{ch}[n]}$ stands for the vector containing the labels of all the children of n.

Remark 1. Labels usually include features of objects related to nodes and features of the relationships between the objects. For example, in the case of a RAG (Fig. 1(a)), node labels may represent properties of the regions (e.g., area, perimeter, average color intensity), and edge labels the relative position of the regions (e.g., distance between baricenters and the angle between the momentums). Similarly, in the example shown in Fig. 1(b), node and edge labels can include a representation of the text contained in the documents and in the anchor texts, respectively. ■

No assumption is made on the nature of the arcs, directed and undirected edges are both permitted. However, when different kinds of edges coexist in the

same dataset, it is necessary to distinguish among them. Such a goal can be easily reached by attaching a proper label to each edge. Thus, in this case, different kinds of arcs turn out to be just arcs with different labels.

The purpose of the proposed method is to learn by examples a function $\boldsymbol{h} : \mathcal{G} \times \mathcal{N} \rightarrow \mathcal{R}^m$, where $\mathcal{G}$ is any set of graphs and $\mathcal{N}$ is the set of their nodes. Thus, a learning data set is a set of three tuples $\mathcal{L} = \{(G_i, n_i, \boldsymbol{t}_i) | \, G_i = (\mathcal{N}_i, \mathcal{E}_i) \in \mathcal{G}, n_i \in \mathcal{N}, \boldsymbol{t}_i \in \mathcal{R}^m, 1 \leq i \leq m\}$. A three tuple $(G_i, n_i, \boldsymbol{t}_i)$ denotes the fact that the desired target for node n_i (of graph G_i) is $\boldsymbol{t}_i$.

Remark 2. The learning data set may contain any number of graphs. In the limit it is possible both from a theoretical and a practical point of view that the whole dataset comprises of a single graph. The dataset consists of nodes with their associated data and the learning problem is well defined provided that there are reasonable numbers of nodes both in the learning data set and in the testing data set respectively. The problem of classifying web pages is a straightforward example of the limiting case. The web is represented by one single graph, the learning data set consists of some pages whose desired classification is known, whereas the classification of other pages on the web should be obtained by generalization. ■

Finally, our approach is based on fixed point theory and contraction mappings [7]. Here, we use the following simple fixed point theorem.

Theorem 1. *If* $g : \mathrm{R}^d \rightarrow \mathrm{R}^d$ *is differentiable and there exists* $0 \leq e < 1$ *such that* $\left\| \frac{\partial g}{\partial x}(x) \right\|_1 \leq e$ *where* $\frac{\partial g}{\partial x}$ *is the Jacobian matrix of* g*, then* g *is a contraction function. Thus, the following system*

$$x = g(x)$$

has one and only one solution x^**. Moreover, the sequence*

$$x(t) = g(x(t-1))$$

converges exponentially to x^* *for any* $x(0)$*.* ■

3 A New Neural Network Model

The intuitive idea underlining the proposed approach is that graph nodes represent objects or concepts and edges represent their relationships. Thus, we can attach to each node n a vector $\boldsymbol{x}_n \in \mathcal{R}^s$, called *state*, which collects a representation of the object denoted by n [2]. In order to define $\boldsymbol{x}_n$, we observe that the related nodes are connected by edges. Thus, $\boldsymbol{x}_n$ is naturally specified using the information contained in the neighborhood of n, which includes the label of n, the labels of the edges which are connected to n, and the states and the labels of the nodes on the neighborhood of n, respectively (see Figure 2).

[2] More precisely, $\boldsymbol{x}_n$ should collect all the information which is relevant for deciding the output $\boldsymbol{h}(G, n)$ in correspondence of n.

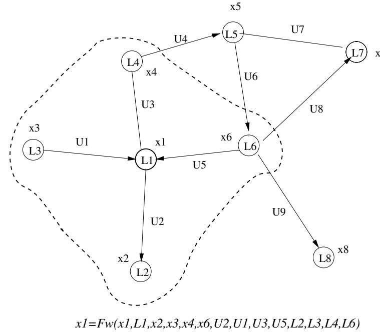

Fig. 2. The state $\boldsymbol{x}_1$ depends on the neighborhood information.

More precisely, let F_w be a parametric function that expresses the dependence of a node on its neighborhood. The states $\boldsymbol{x}_n$ are defined as the solution of the following system of equations

$$\boldsymbol{x}_n = \boldsymbol{F}_w(\boldsymbol{x}_n, L_n, \boldsymbol{x}_{\mathrm{ne}[n]}, L_{\mathrm{co}[n]}, L_{\mathrm{ne}[n]}),\ 1 \le n \le r \tag{1}$$

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

Remark 3. Definition (1) is customized for undirected graphs. When dealing with directed graphs, $L_{\mathrm{co}[n]}$ should be replaced by $L_{\mathrm{from}[n]}$, $L_{\mathrm{to}[n]}$ and similarly, $\boldsymbol{x}_{\mathrm{ne}[n]}$, $L_{\mathrm{ne}[n]}$ by $\boldsymbol{x}_{\mathrm{ch}[n]}$, $\boldsymbol{x}_{\mathrm{pa}[n]}$, and $L_{\mathrm{ch}[n]}$ $L_{\mathrm{pa}[n]}$, respectively. In the following sections, in order to keep the notations simple, we maintain this customization. However, unless explicitly stated, all the results hold also for directed graphs and mixed undirected and directed graphs. ■

Remark 4. Equation (1) should be considered only an example of the possible dependences of a node on its neighborhood. More generally, x_n could be computed from a subset of the parameters in (1) or, on the other hand, the neighborhood could include nodes which are k edges far from n. ■

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

$$\boldsymbol{o}_n = \boldsymbol{O}_w(\boldsymbol{x}_n, L_n),\ 1 \le n \le r. \tag{2}$$

Notice that, in order to ensure that $\boldsymbol{x}_n$ is correctly defined, system (1) must have a unique solution. In general, the number and the existence of solutions depend on $\boldsymbol{F}_w$. Here, we assume that $\boldsymbol{F}_w$ is appropriately designed so that the solution is unique. More precisely, let $\boldsymbol{X}$ and $\boldsymbol{L}$ respectively be the vectors constructed by stacking all the states and all the labels. Then, Equations (1) can be written as:

$$\boldsymbol{X} = \boldsymbol{\Phi}_{\boldsymbol{w},\boldsymbol{L}}(\boldsymbol{X}) \tag{3}$$

where $\boldsymbol{\Phi}_{\boldsymbol{w},\boldsymbol{L}}$ consists of r instances of $\boldsymbol{F}_w$, and $\boldsymbol{w}$ is the set of parameters. The key choice adopted in the proposed approach consists of designing $\boldsymbol{F}_w$ such that it will be a contraction mapping and $\boldsymbol{\Phi}_{\boldsymbol{w},\boldsymbol{L}}$ will satisfy the hypothesis of Theorem 1, i.e. there exists $0 \leq e < 1$ such that $\| \frac{\partial \boldsymbol{\Phi}_{\boldsymbol{w},\boldsymbol{L}}}{\partial \boldsymbol{X}}(\boldsymbol{X}) \| \leq e$ for any $\boldsymbol{w}, \boldsymbol{L}, \boldsymbol{X}$.

In fact, function $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$ will be implemented by particular models of static neural networks. Thus, Equations (1) and (2) specify a new theoretical model suitable for node focused applications. In fact, (1) and (2) define a method to attach an output o_n to each node of a graph, i.e. a parametric function $\boldsymbol{f}_w(G, n) = \boldsymbol{o}_n$ which operates on graphs.

The corresponding learning problem consists of adapting the parameters $\boldsymbol{w}$ of $\boldsymbol{O}_w$ and $\boldsymbol{F}_w$ so that f_w approximates the data in the learning data set. In practice, the learning problem may be implemented by the minimization of a quadratic error function

$$e_w = \sum_{i=1}^{r} (\boldsymbol{t}_i - \boldsymbol{f}_w(G_i, n_i))^2 . \tag{4}$$

Finally, since the number of inputs of $\boldsymbol{F}_w$ is not fixed, but depends on the number of neighbors of each node, the implementation of $\boldsymbol{F}_w$ may be difficult, particularly when the degree of node connectivity undergoes large changes. For this reason, it may be useful to replace Equations (1) with

$$\boldsymbol{x}_n = \sum_{\ell \in \mathrm{ne}[n]} \boldsymbol{H}_w(\boldsymbol{x}_n, L_n, L_{(n,\ell)}, L_\ell) \tag{5}$$

The intuitive idea underlining eq. (5) consists of computing the state $\boldsymbol{x}_n$ by the summing a set of “contributions”. Each contribution is generated considering only one node in the neighborhood of n. Definition eq. (5) is less general than (1), but the implementation of $\boldsymbol{H}_w$ is easier since $\boldsymbol{H}_w$ has a fixed number of parameters.

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

- A method to solve (1);
- A learning algorithm to adapt $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$ by examples from the training data set[3];
- An implementation of $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$ for which $\boldsymbol{\Phi}_{\boldsymbol{w},\boldsymbol{L}}$ is a contraction mapping.

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

3.1 Computing the States

Theorem 1 does not only provide a sufficient condition for the existence of the solution of equation (1), but it also suggest how to compute its fixed point. In fact, for any initial set of states the following dynamical system

[3] In other words, the parameters $\boldsymbol{w}$ are estimated from the training data set.

$$\boldsymbol{x}_n(t) = \boldsymbol{F}_w(\boldsymbol{x}_n(t-1), L_n, x_{\mathrm{ne}[n]}(t-1), L_{\mathrm{co}[n]}, L_{\mathrm{ne}[n]}), \qquad (6)$$

where $\boldsymbol{x}(t)$ denotes the t-th iterate of $\boldsymbol{x}$, converges exponentially fast to the solution of system (1).

Notice that system (6) can be interpreted as the representation of a network consists of units which compute $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$. Such a network will be called an *encoding network*, following a similar terminology used for the recursive neural network model [3]. In order to build the encoding network, each node of the graph can be replaced by a unit computing the function $\boldsymbol{F}_w$ (see Figure 3). Each unit stores the current state $\boldsymbol{x}_n(t)$ of the corresponding node n, and, when activated, it calculates the state $\boldsymbol{x}_n(t+1)$ using the labels and the states stored in its neighborhood. The simultaneous and repeated activation of the units produces the behavior described by system (6). In the encoding network, the output for node n is produced by another unit which implements $\boldsymbol{O}_w$.

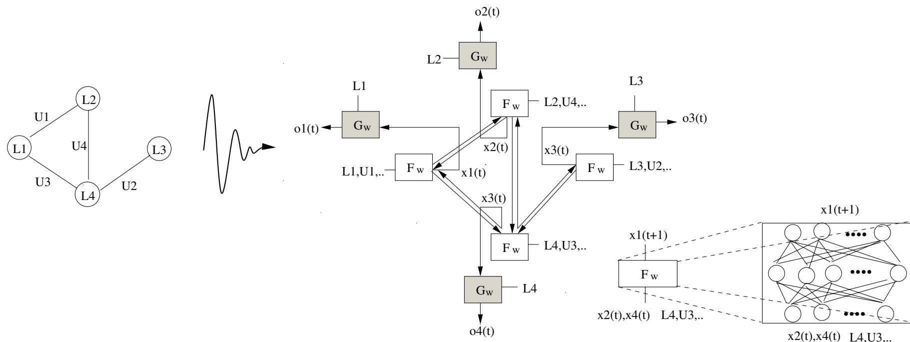

Fig. 3. A graph and its corresponding encoding network.

When $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$ are implemented by static neural networks, the encoding network is a large recurrent neural network where the connections between the neurons can be divided into internal and external connections respectively. The internal connectivity is determined by the neural network architecture used to implement the unit. The neural architecture which have been suggested for realizing this type of problems in the literature for solving a graph focused problem include multilayer perceptrons [3, 11], cascade correlation, and self organizing maps [5, 6]. For node focused problems, e.g., in web page classifications, as far as we are aware, there is only one application of such a concept using a linear model [12]. The external connectivity mimics the graph connections. Moreover, the weights of such a recurrent neural network are shared, since the same parameters w are common to all the units.

3.2 A Learning Algorithm

Without loss of generality let us assume that the learning data set contains one single graph. This is a general case, as when we have many graphs, it is

possible to transform this into a single graph by grouping them into one single non–connected graph. The learning algorithm we propose consists of two phases:

(a) the states $\boldsymbol{x}(t)$ are repeatedly updated, using Eq. (6) until they reach a stable point at time T;
(b) the gradient $\frac{\partial e_w(T)}{\partial \boldsymbol{w}}$ is computed and the weights $\boldsymbol{w}$ are updated according to a gradient descent strategy.

These two phases are repeated until a given stopping criterion is reached A similar approach, based on a stabilizing and a learning phase, was already proposed for training a random walk process in [2]. Thus, while phase (a) moves the system to the stable point, phase (b) adapts the weights to change the outputs towards the desired target. It is worth noting that the gradient $\frac{\partial e_w(T)}{\partial \boldsymbol{w}}$ depends only on the error at time T, when the system is supposed to be stable. In fact, the output of our model depends on function $\boldsymbol{O}_w$ and on the stable point which is determined by $\boldsymbol{F}_w$. In order to obtain the desired outputs, it is necessary to change the fixed point along with $\boldsymbol{O}_w$. The proposed algorithm can be interpreted as a gradient descend whose goal consists of moving the fixed point to a new position where the function $\boldsymbol{O}_w$ can produce the desired output more readily. For this reason, only the error at time T is to be considered.

The Gradient Computation. The gradient could be computed using a backpropagation through time algorithm [4]. In this case, the encoding network is unfolded from time T back to an initial time t_0. The unfolding produces a layered network (see Figure 4). Each layer corresponds to a time instance and contains a copy of all the units $\boldsymbol{F}_w$ of the encoding network. The units of two consecutive layers are connected following the encoding network connectivity (i.e. the graph connectivity). The last layer corresponding to time T includes also the unit $\boldsymbol{O}_w$ and allows to compute the output of the network. Backpropagation through time consists of carrying out a common backpropagation on the unfolded network in order to compute the gradient of the error at time T with respect to all the instances of $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$. Then, $\frac{\partial e_w(T)}{\partial \boldsymbol{w}}$ is obtained summing the gradients of all instances.

However, backpropagation through time requires to store the states of every instance of the units. When the graphs and $T - t_0$ are large, the memory required may be considerable[4]. On the other hand, in our case, a more efficient approach is possible. Since the system has reached a stable point, we can assume that $\boldsymbol{x}(t) = \boldsymbol{x}(t_0)$ for any $t \geq t_0$. Thus, the states of units remain constant for each instant, and backpropagation through time can be carried out storing only $\boldsymbol{x}(t_0)$. More details on gradient computation are available in [9].

[4] 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.

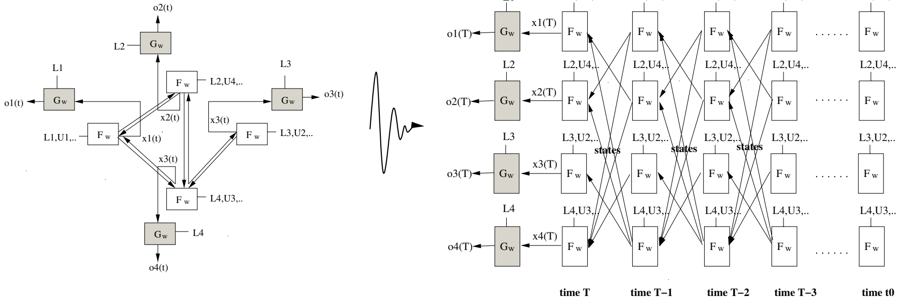

Fig. 4. A graph and its encoding network to illustrate the backpropagation through time concept.

The Learning Algorithm. The learning algorithm is summarized in Table 1. It consists of a main procedure and two functions called *Forward* and *Backward* respectively. The function *Forward* takes in input the current set of parameters $\boldsymbol{w}$ and the current state $\boldsymbol{X}$ and iterates the system equations. The iteration is stopped when $\|\boldsymbol{X}(t+1)-\boldsymbol{X}(t)\|$ is less than a given threshold ϵ_f. The function *Backward* computes the gradient using a time window $[T, t_0]$ such that $\|\frac{\partial e_w(T)}{\partial \boldsymbol{X}(t_0)} - \frac{\partial e_w(T)}{\partial \boldsymbol{X}(t_0-1)}\| \leq \epsilon_b$.

Table 1. The learning algorithm.

/* *Main procedure* */
Learn($\boldsymbol{F}_w, \boldsymbol{O}_w, \mathcal{L}$)
initialize $\boldsymbol{w}$, $\boldsymbol{X}$;
$\boldsymbol{X}$:=Forward($\boldsymbol{X}, \boldsymbol{w}$);
repeat
  $\frac{\partial e_w}{\partial \boldsymbol{w}}$:=Backward($\boldsymbol{X}, \boldsymbol{w}$);
  $\boldsymbol{w}$:=$\boldsymbol{w} - \lambda \cdot \frac{\partial e_w}{\partial \boldsymbol{w}}$;
  $\boldsymbol{X}$:=Forward($\boldsymbol{X}, \boldsymbol{w}$);
until the stopping criterion is achieved;
return $\boldsymbol{w}$;
end

/* *Move to a stable point* */
Forward($\boldsymbol{X}, \boldsymbol{w}$)
$\boldsymbol{X}(0)$:=$\boldsymbol{X}$, $t = 0$;
repeat
  Compute $\boldsymbol{X}(t+1)$; from $\boldsymbol{X}(t)$;
  t:=$t+1$;
until $\|\boldsymbol{X}(t+1) - \boldsymbol{X}(t)\| \leq \epsilon_f$;
return $\boldsymbol{X}(t+1)$;
end

/* *Compute the gradient* */
Backward($\boldsymbol{X}$,$\boldsymbol{w}$)
Assume $\boldsymbol{X}(t) = \boldsymbol{X}$ for each t;
Find a window $[T, t_0]$ s.t.
  $\|\frac{\partial e_w(T)}{\partial \boldsymbol{X}(t_0)} - \frac{\partial e_w(T)}{\partial \boldsymbol{X}(t_0-1)}\| \leq \epsilon_b$;
Compute $\frac{\partial e_w(T)}{\partial \boldsymbol{w}}$ by backpropagation through time on the window $[T, t_0]$;
end

The main procedure calls the functions Forward and Backward and updates the weights until the output reaches a desired accuracy or some other stopping criterion is achieved. In Table 1, the weights are updated according to a simple gradient descent strategy with a fixed learning rate λ. However, other strategies are also possible, e.g. based on an adaptive learning rate, as long as the adaptive learning rate decreases faster than a constant rate. Moreover, while the initialization of parameters w depends on the particular implementation of F_w and O_w, in theory, X in the main procedure can be initialized to any value. In practice, it is simplest to set this to be $X = 0$.

Finally, the values ϵ_f and ϵ_b are design parameters. It can be proved that if $\boldsymbol{\Phi}_{\boldsymbol{w},\boldsymbol{L}}$ is a contraction mapping, then $\|\boldsymbol{X}(t+1)-\boldsymbol{X}(t)\|$ and $\|\frac{\partial e_w(T)}{\partial \boldsymbol{X}(t_0)} - \frac{\partial e_w(T)}{\partial \boldsymbol{X}(t_0-1)}\|$ converge exponentially to 0, when t and $t - t_0$ increase, respectively. Thus, it is possible to set ϵ_f and ϵ_b to very small values without effecting heavily the performance of the algorithm.

3.3 Comparing Our Approach with Recursive Neural Networks and Random Walks

Recursive neural networks are a special case of the model described in (6), where

- the input graph is directed and acyclic;
- the inputs of $\boldsymbol{F}_w$ are limited to $\boldsymbol{L}_n$ and $\boldsymbol{x}_{\mathrm{ch}[n]}(t-1)$;
- the graph should contain a node s called supersource from which all the other nodes can be reached;
- the recursive network is the output $\boldsymbol{o}_s$ computed the supersource.

Note that the above constraints on the processed graphs and on the inputs of $\boldsymbol{F}_w$ exclude any sort of cyclic dependence of a state on itself. Thus, in the recursive model, the encoding networks are feedforward networks.

This assumption simplifies the computation of the node states. In fact, the states can be computed following a predefined direction, i.e. from the leaf nodes to the supersource node of the graph. First, the states of the leaf nodes are calculated, then the states of their parents are computed and so on up to the supersource node. For the supersource node, the recursive neural network computes also an output, which is returned as the result of the graph computation.

Moreover, the above assumptions allow to train recursive neural networks by applying a common backpropagation procedure on the encoding network [3, 11]. This solution is not viable for GNNs, since the presence of cyclic dependencies among the states transforms the encoding network into a dynamic system. For this reason, it has been necessary to assume that the function $\boldsymbol{\Phi}_{\boldsymbol{w},\boldsymbol{L}}$ is a contraction map and to propose a new learning algorithm. However, it must be pointed out that the learning algorithm adopted for GNNs is an extension of the one used for recursive neural networks and that the two algorithms behave in the same way on acylic graphs.

On the other hand, in a random walk model, $\boldsymbol{F}_w$ is a linear function. In a simple case, the states x_n associated with nodes are real values and satisfy

$$x_n(t+1) = \sum_{i \in \mathrm{pa}[n]} w_{n,i} x_i(t) \tag{7}$$

where $w_{n,i} \in \mathcal{R}$ and $w_{n,i} \geq 0$ hold for each n, i. The $w_{n,i}$ are normalized so that $\sum_{i \in ch[n]} w_{i,n} = 1$. In fact, Eq. (7) can represent a random walker who is traveling on the graph. The value $w_{n,i}$ represents the probability that the walker, when he/she is on node n, decides to go to node i. The state x_n stands for the

probability that the walker is on node n in the steady state. When all the x_n are stacked into a vector $\boldsymbol{X}$, Eq. (7) becomes

$$\boldsymbol{X}(t+1) = \boldsymbol{W}\boldsymbol{X}(t) \tag{8}$$

where $\boldsymbol{W} = \{w_{n,i}\}$ and $w_{n,i}$ is as defined in Eq. (7) if $i \in \mathrm{pa}[n]$ and $w_{n,i} = 0$ otherwise. It is easily verified that $\|\boldsymbol{W}\|_1 = 1$. Markov chain theory suggests that if there exists t such that all the elements of the matrix $\boldsymbol{W}^t$ are non–null, then Eq. (7) is a contraction mapping [10].

Thus random walks on graphs are an instance of our model, since they implement a linear version of it. The set of processed graphs include cyclic graphs, but these graphs are usually unlabeled. Moreover, random walk theory does not provide a learning algorithm. In our development described in this paper, we have proposed a learning algorithm which allows the estimation of the set of parameters from training samples. Thus, our model extends the work on random walk models by providing the possibility of learning the parameters from training samples.

3.4 Implementing $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$

The implementation of $\boldsymbol{O}_w$ does not need to fulfill any particular constraints. In our experiments, $\boldsymbol{O}_w$ will be simply implemented by a feedforward neural network (a multilayer perceptron). On the other hand, $\boldsymbol{F}_w$ plays a crucial role in the proposed model, since its implementation determines the number and the existence of the solution of Equation (1).

The key choice adopted in our approach consists of designing $\boldsymbol{F}_w$ such that $\boldsymbol{\Phi}_{w,L}$ is a contraction mapping. Let $\delta_{n,i,u,j}$ denote the element of the Jacobian $\frac{\partial \boldsymbol{\Phi}_{w,L}(X)}{\partial \boldsymbol{X}}$ of $\boldsymbol{\Phi}_{w,L}$ whose row corresponds to j–th component of node u and whose column corresponds to i–th component of node n. By Theorem 1, $\boldsymbol{\Phi}_{w,L}$ is a contraction mapping provided that it is differentiable and

$$\left| \sum_{n,i} \delta_{n,i,u,j} \right| \leq e \tag{9}$$

holds for some real number $0 \leq e < 1$. Inequality (9) can be used to design the $\boldsymbol{F}_w$ in (1) or the $\boldsymbol{H}_w$ in (5) such that $\boldsymbol{\Phi}_{w,L}$ is a contraction mapping.

In this paper, two implementations of $\boldsymbol{\Phi}_{,L}$ are suggested:

(a) $\boldsymbol{H}_w$ is realized by a linear system whose parameters are determined by a neural network; the model is such that Eq. (9) holds for any set of parameters w.

(b) $\boldsymbol{F}_w$ is realized by a common feedforward neural network: the cost function adopted in the learning procedure includes a penalty term that keeps the parameters w in the region where Eq. (9) is fulfilled.

Implementation of $\boldsymbol{H_w}$ by a Linear Function. In the first implementation, $\boldsymbol{H}_w$ is a linear function

$$\boldsymbol{x}_n = \boldsymbol{E}_n + \sum_{r \in \mathrm{ne}[n]} \boldsymbol{W}_{n,r} \boldsymbol{x}_r$$

similar to that used in random walks. But, the state attached to the nodes are vector of reals instead of simple reals and the parameters are not statically defined, but they are computed by two feedforward neural networks $\mathcal{N}_E$ and $\mathcal{N}_W$. The neural network decides the parameters $\boldsymbol{E}_n \in \mathcal{R}^s$ and $\boldsymbol{W}_{n,r} \in \mathcal{R}^{s \times s}$ on the basis of the labels attached to nodes n, r and the arc (r,n). More formally, let $\boldsymbol{f}_E : \mathcal{R}^{l_N} \to \mathcal{R}^{nr}$ and $\boldsymbol{f}_W : \mathcal{R}^{l_E} \to \mathcal{R}^{n^2 r^2}$ be the functions implemented by $\mathcal{N}_E$ and $\mathcal{N}_W$, respectively. Then, we can define

$$\begin{aligned} \boldsymbol{E}_n &= \boldsymbol{f}_E(L_n) \\ \boldsymbol{W}_{n,r} &= \mathrm{Resize}_{s \times s}\left(\frac{e}{s|\mathrm{ne}[r]|} \boldsymbol{f}_W(L_{(n,r)})\right) \end{aligned}$$

where $\mathrm{Resize}_{s \times s}(\cdot)$ denotes the operator that rearranges the components of a $s^2 \times 1$ vector into a $s \times s$ matrix and $|\mathrm{ne}[r]|$ represents the cardinality of $\mathrm{ne}[r]$.

In this case, inequality (9) holds provided that the output of $\boldsymbol{f}_W$ is in the range $[-1, 1]$, which can be achieved by using a sigmoidal activation function in the output layer of $\mathcal{N}_W$. In fact, $\delta_{n,i,u,j} = (\boldsymbol{W}_{n,u})_{i,j}$ and, as a consequence,

$$\left| \sum_{n,i} \delta_{n,i,u,j} \right| \le \sum_{n,i} |\boldsymbol{W}_{n,u}|_{i,j} \le \sum_{n \in \mathrm{ne}[u], i} \frac{e}{s|\mathrm{ne}[u]|} = 1 \, .$$

Implementation of $\boldsymbol{F_w}$ by a Neural Network. Let us suppose that $\boldsymbol{F}_w$ is realized by a layered feedforward neural network with logistic sigmoid activation functions. In this case, (9) holds only for some values of the parameters $\boldsymbol{w}$. In fact, $\delta_{r,j,n,i}$ is small for small values of the network parameters, but it may become large for large values, e.g. when the hidden–to–output weights are large. In order to ensure (9) is fulfilled, a penalty term can be added to the error function which becomes

$$e_w = \sum_{k=1}^{m} (t_k - f_w(G_k, n_k))^2 + \beta \sum_{n,i} L\left(\sum_{r,j} \delta_{r,j,n,i} \right)$$

where β is a predefined parameter balancing the importance of the error on patterns and the penalty term, and $L(y)$ is $(y-e)^2$ if $|y| > e$ and 0, otherwise. Note that the same reasoning can be applied also to the case when $\boldsymbol{H}_w$ instead of $\boldsymbol{F}_w$ is implemented by a layered neural network.

4 Experimental Results

In the paper, we present some preliminary results obtained using the linear implementation of $\boldsymbol{H}_w$. More experiments, including some obtained by directly

implementing $\boldsymbol{H}_w$ with a neural network, are in [9]. The linear implementation of GNN has been verified on the subgraph recognition problem and the web page ranking problem.

The subgraph recognition problem consists of identifying the presence of a subgraph in a larger graph. In our experiments, we used random graphs. The graphs have integer labels in the range $[0, 10]$ and these labels have been added a normal distribute noise having mean of 0 and a variance 0.25. Different dimensions for the graphs and the subgraphs have been evaluated in different experiments.

Tables 2 shows the results of the experiments. Each column is related to a different set of experiments. The notation $s - g$ in the header of the column defines the number of nodes s of the subgraph to be identified and the total number of nodes g in the graphs that contain the subgraph. For any pair $s - g$ the experiment has been carried out 5 times with different subgraphs. In each experiment the dataset contained 450 graphs, equally distributed between the training dataset, the testing dataset, and a validation set.

The results are interesting. In fact, it must be pointed out that that the classic subgraph recognition algorithms cannot evaluate situations when there are corruptions in the graph labels. On the other hand, as may be observed in our results, our proposed method can handle such situation quite easily. Moreover, to verify the capability of the method in learning the graph connectivity, the results have been compared with those achieved by a common three layered (one hidden) neural network (FNN) which takes in as its input only the label of a node n. It is observed that GNNs clearly outperforms this latter approach.

Table 2. The results of the subgraph recognition problem.

	$3-6$	$5-6$	$3-10$	$5-10$	$7-10$	$3-14$	$5-14$	$7-14$	*Average*
GNN on testset with noise	91.62	93.05	86.41	78.70	86.94	86.71	78.56	79.81	85.22
FNN on testset with noise	71.67	87.22	69.39	58.17	74.16	72.86	67.34	55.93	69.59
GNN on trainset with noise	92.28	93.85	86.96	79.64	87.97	86.87	80.56	80.99	86.14
FNN on traintset with noise	70.85	87.08	69.83	57.71	74.23	73.09	67.43	55.85	69.51
GNN on testset no noise	94.22	93.03	89.95	84.88	90.24	89.75	83.49	80.11	88.21
GNN on trainset no noise	94.77	93.58	91.06	85.81	90.86	90.41	83.97	80.17	88.83
FNN on trainset no noise	73.48	88.23	69.96	66.45	78.74	71.68	65.49	58.89	71.62

In a second experiment, the goal was to simulate a web page ranking algorithm. For this experiment a random graph containing 1000 nodes has been generated. To each node, a label $[a, b]$ has been attached where a, b are binary values. The label represents whether the page belongs to two given topics. If the page belongs to both topics, then, $[a, b] = [1, 1]$; if it belongs to only one topic, then $[a, b] = [1, 0]$, or $[a, b] = [0, 1]$ and if it does not belong to either topics then $[a, b] = [0, 0]$. The GNN was trained in order to produce the following output o_i

$$o_i = \begin{cases} 2 * PR_i \text{ if } a = 0, b = 1 \; or \; a = 1, b = 0 \\ PR_i \quad\quad \text{otherwise} \end{cases}$$

where PR_i stands for the Google's PageRank [1] of page n. Thus this experiment simulates the situation when a user wishes to see pages which belong to one topic

and not the other and have their page ranks raised to twice that of the PageRank as given by method described in [1]. Such wishes are encountered often in the construction of web portals.

The two function $\boldsymbol{F}_w$ and $\boldsymbol{f}_E$ were implemented by three layer neural networks (one hidden layer) with linear output function. For the output function $\boldsymbol{O}_w$, two implementations have been evaluated: a three layer neural network and a two layer neural network. Figure 5 (a) and (b) show the output of the two layer network and the output of the three layer network, respectively. The plots display the desired rank (the continuous line) w.r.t. the rank computed by GNN (the dots). The pages, sorted by the desired rank, are displayed on horizontal axes, the ranks on the vertical ones. The plots show that the three layered network achieves better results and approximate well the desired function.

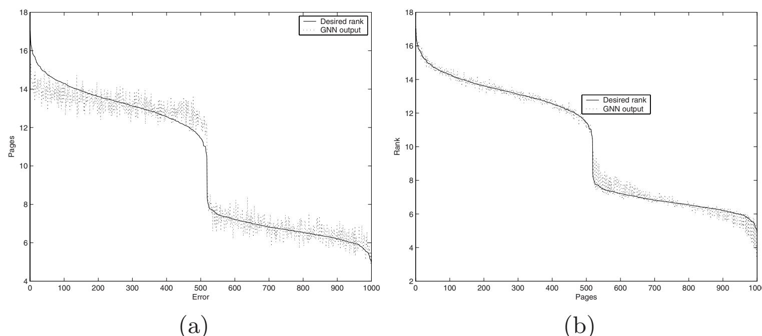

Fig. 5. The output of the model using a two layer network (a) and a three layer network (b) to implement $\boldsymbol{O}_w$.

5 Conclusions

In this paper, we have presented a unified approach to considering both a graph focused approach and a node focused approach to graph structured data. We have discussed the properties of the new neural model (GNN) and we have further provided a learning algorithm which can estimate the parameters. The preliminary experimental results confirms the viability of the approach.

Future research directions include a wide experimentation of GNNs, both to validate them on real life applications and to test different implementations of the functions $\boldsymbol{F}_w$ and $\boldsymbol{O}_w$. At the same time, a number of theoretical questions are still open, including an analysis of the approximation capability of GNNs and more general sufficient conditions to guarantee the existence and the uniqueness of the solution of Eq. (1).

Acknowledgements

The first, third and fourth authors acknowledge the financial support from the Australian Research Council through a Linkage International Exchange scheme

which makes the research reported in this paper possible. In addition, the first and fourth authors acknowledge the financial support from the Australian Research Council through a discovery project grant.

References

1. Sergey Brin and Lawrence Page. The anatomy of a large–scale hypertextual Web search engine. In *Proceedings of the 7th World Wide Web Conference*, April 1998.
2. M. Diligenti, M. Gori, and M. Maggini. A learning algorithm for web page scoring systems. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2003.
3. P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, September 1998.
4. C. Lee Giles and Marco Gori, editors. *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*, volume 1387 of *Lecture Notes in Computer Science*. Springer, 1998.
5. M. Hagenbuchner, A. Sperduti, and A.C. Tsoi. A self-organizing map for adaptive processing of structured data. *IEEE Transactions on Neural Networks*, 2003.
6. M. Hagenbuchner, A.C. Tsoi, and A. Sperduti. A supervised self-organising map for structured data. In N.Allinson, H.Yin, L.Allinson, and J.Slack, editors, *WSOM 2001 - Advances in Self-Organising Maps*, pages 21–28. Springer, June 2001.
7. Mohamed A. Khamsi. *An Introduction to Metric Spaces and Fixed Point Theory.* John Wiley & Sons Inc, 2001.
8. J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.
9. F. Scarselli, A. C. Tsoi, M. Gori, and M. Hegenbuchner. A new neural network approach to graph processing. Technical Report DII /04, Dipartimento di Ingeneria dell'Inforazione, University of Siena, Siena, Italy, 2004.
10. E. Seneta. *Non–negative matrices and Markov chains.* Springer Verlag, 1981. Chapter 4, pages 112–158.
11. A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8:429–459, 1997.
12. A. C. Tsoi, G. Morini, F. Scarselli, Hagenbuchner, and M. M., Maggini. Adaptive ranking of web pages. In *Proceedings of the 12th WWW Conference*, Budapest, Hungary, May 2003.