Learning Long–Term Dependencies Using Layered Graph Neural Networks

Niccolò Bandinelli, Monica Bianchini and Franco Scarselli

Abstract— **Graph Neural Networks (GNNs) are a powerful tool for processing graphs, that represent a natural way to collect information coming from several areas of science and engineering — e.g. data mining, computer vision, molecular chemistry, molecular biology, pattern recognition —, where data are intrinsically organized in entities and relationships among entities. Nevertheless, GNNs suffer, so as recurrent/recursive models, from the long–term dependency problem that makes the learning difficult in deep structures. In this paper, we present a new architecture, called Layered GNN (LGNN), realized by a cascade of GNNs: each layer is fed with the original data and with the state information calculated by the previous layer in the cascade. Intuitively, this allows each GNN to solve a subproblem, related only to those patterns that were misclassified by the previous GNNs. Some experimental results are reported, based on synthetic and real–world datasets, which assess a significant improvement in performances w.r.t. the standard GNN approach.**

I. Introduction

In several machine learning applications the data can be suitably represented in form of sequences, trees, and, more generally, graphs, f.i. in chemistry [1], software engineering [2], image processing [3], and bioinformatics [4]. In fact, in such applications, the information is naturally constituted by entities — which share some common features — and by relations among entities, describing their interactions and their mutual influence, so that the causal, hierarchical, and topological connections among parts of a given pattern yield significant contents. Therefore, the truly subsymbolic nature of these problems makes it very difficult to represent their domains by vectorial data, since feature encoding may often produce loss of information and it is problem–dependent, time–consuming and heuristic. The way in which this kind of information should be processed can be regarded neither as strictly symbolic, nor as subsymbolic, neither sequential nor parallel. The human brain is a complex graph of elementary neurons, and also the data to be processed can often be regarded as complex structures of elementary units.

Learning in graphical domains means defining a function τ that maps a graph G and, possibly, one of its nodes n into a vector of real numbers. Actually, machine learning applications on graphs can be divided into two broad classes, called *graph focused* and *node focused*. 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 G, the nodes of which stand for atoms (or chemical groups), whereas the edges represent chemical bonds. The mapping $\tau(G)$ used to estimate the probability that the chemical compound causes a certain disease [5] is graph focused, being a property of the whole compound instead of a particular atom. On the other hand, the prediction of disordered regions in protein structures [6] represents a node focused task. In the image processing context, images can be represented with a Region Adjacency Graph (RAG) [7], where the nodes denote the homogeneous regions of the image and the arcs represent their adjacency relationship. In this case, object detection is a node focused application [8], [9], whereas full image classification is graph focused [10].

In the last decade, many neural network models able to process structured information have been devised [11], both in the supervised [2], [12] and in the unsupervised framework [13], [14], [15]. In *Recursive Neural Networks* (RNNs) [2], [12], the main idea consists in encoding the graphical information into a set of states associated with the graph nodes. The states are dynamically updated following the topological arrangement of the nodes. Finally, an output is computed using the encodings stored in the states. However, the RNN model can process only directed and acyclic graphs and can be used only on graph focused problems. On the other hand, *Graph Neural Networks* (GNNs) extend the recursive paradigm and are able to compute functions on general graph spaces and can be applied both on graph and node focused problems [16].

Nevertheless, all recurrent/recursive models suffer from the long–term dependency pathology, i.e. they are unable to properly process deep structures. This is due to the very local nature of the learning procedure, that prevent both the state calculation and the weight updating to be influenced from weights, labels, states related to far nodes. In other words, practical difficulties are easily verifiable when dynamic neural network models are trained on tasks where the temporal contingiencies present in the input/output data span long intervals. These ideas have been clearly formalized in [17] for recurrent networks, where it is proven that if the system is to latch information robustly, then the fraction of the gradient due to information t time steps in the past approaches zero as t becomes large.

Interestingly, the long–term dependency problem conflicts with the idea, which has recently been explored by several researchers, that deep architectures, composed by numerous layers of neurons, are necessary in order to cope with complex applications. Actually, it has been proven that deep

Niccolò Bandinelli, Monica Bianchini and Franco Scarselli are with Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, Via Roma 56, I–53100 Siena, Italy. (Corresponding author: F. Scarselli, phone: +39 0577 234850–1052; email: franco@dii.unisi.it).

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

networks can implement, with a smaller number of neurons, functions that cannot be implemented with shallow architectures [18]. Moreover, many studies have been recently carried out on how to train a network layer by layer, mixing supervised and unsupervised algorithms, see f.i., [19], [18].

In this paper, we present a new composite architecture, called Layered GNN (LGNN), that collects a cascade of GNNs, each of which is fed with the original data (coming from the problem to be faced) and with the information calculated by the previous GNN in the cascade. LGNNs are trained layer by layer, using the original targets and forcing each network in the cascade to improve the solution of the previous one. Intuitively, this allows each GNN to solve a subproblem, related only to those patterns which were misclassified by the previous GNNs. Moreover, the solution provided by a certain layer about a particular node contains information that is not only local to that node, but it is also related to its neighbor nodes. Thus, the approach is expected to realize a sort of incremental learning that is progressively enriched by taking into account far and far information.

Some experiments have been carried out, based on synthetic and real–world datasets. Synthetic data are used to solve two fundamental problems in graph theory, i.e. subgraph matching and clique searching. Instead, real–world datasets come from the bioinformatic field and are related to the prediction of protein secondary structure, and to the classification of compounds w.r.t. their mutagenicity. Experimental results are promising and assess significant improvements in performances w.r.t. standard GNNs.

The paper is organized as follows. Section II presents the GNN model, while Section III contains the description of the proposed Layered GNN architecture. Some experimental results are then reported in Section IV. Finally, conclusions are drawn in Section V, together with future work proposals.

II. The Graph Neural Network Model

In the following, a graph G is a pair (N, E), where N is the set of nodes and E the set of edges. All the nodes directly connected to node n are indicated by ne$[n]$. Each node may be labeled, with label $\boldsymbol{l}_n \in \mathrm{R}^q$. Usually, labels describe some features of the object related to that node[1].

In fact, the intuitive idea underlying GNNs is that nodes represent objects or concepts and edges represent their relationships. Thus, we can attach to each node n a *state* vector $\boldsymbol{x}_n \in \mathrm{R}^s$, which collects a representation of the object denoted by n, also based on the information coming from its neighborhood (see Figure 1). More precisely, let $\boldsymbol{w}$ be a set of parameters and let $f_{\boldsymbol{w}}$ be a parametric *transition function*. The state $\boldsymbol{x}_n$ is defined as the solution of the system of equations:

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

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

[1]For instance, in the case of RAGs, node labels may collect geometrical and visual properties of a homogeneous region inside the image — such as perimeter, area, color, and texture.

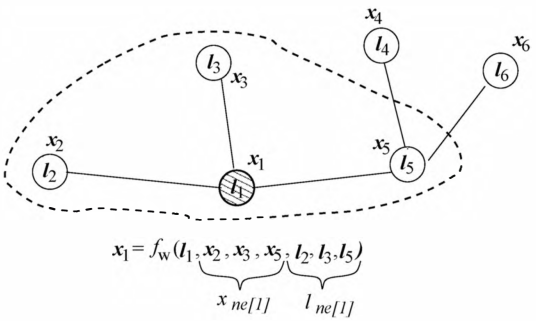

Fig. 1. The state $\boldsymbol{x}_1$ depends on the neighborhood information.

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

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

Let $\boldsymbol{x}$ and $\boldsymbol{l}$ be constructed by collecting all the states and all the labels related to the nodes of G. Then, Eqs. (1)–(2) can be rewritten as:

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

where $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ are the composition of $|N|$ instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, $\boldsymbol{l}$ stacks all the labels, and $\boldsymbol{x}$ is defined only if the solution of (3) is unique. Based on the Banach Fixed Point Theorem [20], a unique solution to (3) is guaranteed if and only if $F_{\boldsymbol{w}}$ is a contraction mapping[2] w.r.t. the state $\boldsymbol{x}$. Thus, Eqs. (1)–(2) define a method to produce an output $\boldsymbol{o}_n$ for each node n, i.e. they realize a parametric function $\varphi_{\boldsymbol{w}}(G, n) = \boldsymbol{o}_n$ on graphs.

The corresponding machine learning problem consists in adapting the parameters $\boldsymbol{w}$ such that $\varphi_{\boldsymbol{w}}$ approximates the data in the learning set $\mathcal{L} = \{(G_i, n_i, \boldsymbol{t}_i) | 1 \le i \le p\}$, where each triple $(G_i, n_i, \boldsymbol{t}_i)$ denotes a graph G_i, one of its nodes n_i, and the desired output vector $\boldsymbol{t}_i$.

In practice, the learning problem can be implemented by the minimization of a quadratic error function

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

It is worth noting that Eq. (1) is particularly suited to process positional graphs (i.e. graphs in which an order relationship is established among all the edges originating from a node), since the label and the state of each neighbor is assigned to a predefined position in the inputs of $f_{\boldsymbol{w}}$. For non–positional graphs, it may be useful to replace Eq. (1) with:

$$\boldsymbol{x}_n = \sum_{u \in \mathrm{ne}[n]} h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{x}_u, \boldsymbol{l}_u),\ n \in N, \tag{4}$$

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

that consists of computing the state $\boldsymbol{x}_n$ by summing a set of "contributions", each one generated considering only a single node in the neighborhood of n. A similar approach was already successfully used for recursive neural networks in [21], [22]. Moreover, GNNs can be applied also to directed graphs. For this purpose, the input of $f_{\boldsymbol{w}}$ (or $h_{\boldsymbol{w}}$) must be extended with information about the edge directions, f.i. a flag d_u for each node $u \in \text{ne}[n]$ such that $d_u = 1$ if the edge (u, v) is directed toward v, and $d_u = 0$ otherwise. Finally, in graph focused applications, a unique output is produced for each graph. This can be achieved in several ways. For example a special node s, called a *supersource*, can be selected in each graph and the corresponding output $\boldsymbol{o}_s$ is returned. Such an approach has also been used in recursive networks.

In order to practically implement the GNN model, a method to solve Eq. (1) must be devised together with a learning algorithm to adapt $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ by examples. Finally, $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ must be appropriately selected. These aspects will be deepened in the following.

As a matter of fact, the Banach Theorem suggests a simple algorithm to compute the fixed point $\boldsymbol{x}_T$ of Eq. (3), since it states that if $F_{\boldsymbol{w}}$ is a contraction mapping, then the dynamical system

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

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

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

Eq. (5) may be interpreted as the computation carried out by an *encoding network*, that consists of units which calculate both $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ (see Figure 2). The encoding network is ideally constructed by replacing the nodes of the input graph with $f_{\boldsymbol{w}}$ units and linking those units according to the graph connectivity. Unit n 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 labels and the states stored in its neighborhood. The asynchronous and repeated activation of the units produces the behavior described by Eq. (5). In the encoding network, the output at node n is then produced by a different network, which implements $g_{\boldsymbol{w}}$.

Learning occurs on the encoding network in two interleaved phases: first, all the states $\boldsymbol{x}_n(t)$ are iteratively updated, using Eq. (5), until they reach a stable state $\boldsymbol{x}(T) \approx \boldsymbol{x}$ at time T; then the gradient $\frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{w}}$ is computed and the weights $\boldsymbol{w}$ are updated, according to a gradient descent strategy.

In fact, the learning procedure is obtained by combining Backpropagation Through Structure, which is adopted for recursive neural networks [12], and the Almeida–Pineda algorithm [23], [24], being the latter a particular implementation of Backpropagation Through Time, usually used to train recurrent networks. Our approach applies the Almeida–Pineda algorithm to the encoding network, where all the instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ are considered to be independent networks. It produces a set of gradients, one for each instance of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, that are subsequently accumulated to compute $\frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{w}}$.

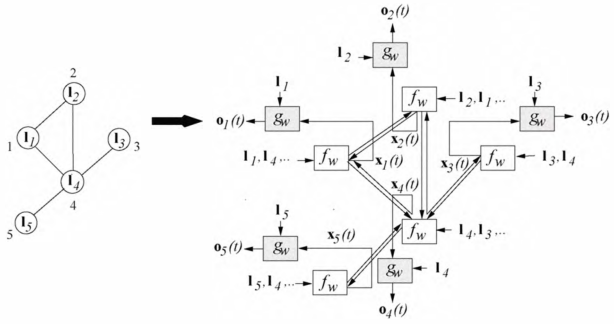

Fig. 2. A graph and its corresponding encoding network.

Finally, both the transition and the output functions may be implemented by a feedforward neural network. In particular, the output function $g_{\boldsymbol{w}}$ is always realized by a multilayer perceptron (MLP), whereas *linear* and *neural* GNNs differ in the implementation of the function $h_{\boldsymbol{w}}$ in (4) and in the strategy adopted to ensure that $F_{\boldsymbol{w}}$ is a contraction mapping. An exhaustive description of the GNN model, that includes details on its training procedure, can be found in [16].

Based on the GNN model described above, a composite architecture, called Layered GNN (LGNN), can be constructed, whose peculiarities and properties will be the object of the following section.

III. Layered GNN

In this section, we investigate the behaviour of a model that is composed by layers of GNNs. As depitected in Figure 3, the original input graph is used to feed the GNN in layer 0. Then, the graph, enriched with information provided by the computation carried out by the first GNN, constitutes the input to the second layer GNN. Such a process is iterated until the last layer is reached and the global output is computed.

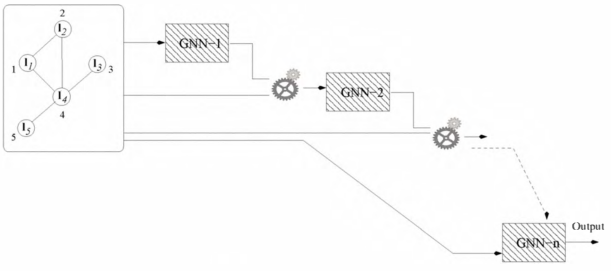

Fig. 3. The LGNN architecture.

The intuitive idea supporting the approach is that each GNN is an expert that solves the considered problem using the original data and the expertise provided by the previous

layers. Even if each GNN is apparently applied on the same problem, however it can actually rely on the computation carried out by the previous layers, so that its task becomes that of fixing the mistakes made by the previous networks, instead that of solving the whole problem.

Formally, for each graph G of the original dataset, the GNN $\mathcal{N}^k$, on layer k, is fed with a graph G^{k-1}. More precisely, the input graph of the first layer is G, i.e., $G^0 = G$. For the successive layers, G^{k-1} has the same topology of G, but its node labels are constructed from those in G and from the computation carried out by the network $\mathcal{N}^{k-1}$. In our experiments, we will consider three cases:

1) The node labels in G^{k-1} are constructed by extending the original labels with the outputs produced by the previous GNN in the cascade, i.e., $\boldsymbol{l}_n^{k-1} = [\boldsymbol{l}_n^0, \boldsymbol{o}_n^{k-1}]$ holds, where $\boldsymbol{l}_n^0$ is the label of node n in G and $\boldsymbol{o}_n^{k-1}$ is the output of $\mathcal{N}^{k-1}$ on node n;
2) The node labels in G^{k-1} are obtained by extending the original labels with the node states in the previous GNN, i.e., $\boldsymbol{l}_n^{k-1} = [\boldsymbol{l}_n^0, \boldsymbol{x}_n^{k-1}]$ holds, where $\boldsymbol{x}_n^{k-1}$ is the state for node n computed by $\mathcal{N}^{k-1}$;
3) The node labels in G^{k-1} are obtained by extending the original labels with the outputs and the states produced by the previous GNN, i.e., $\boldsymbol{l}_n^{k-1} = [\boldsymbol{l}_n^0, \boldsymbol{x}_n^{k-1}, \boldsymbol{o}_n^{k-1}]$ holds.

Thus, all the graphs $G, G^1, G^2, \ldots$ have the same structure and differ for the label content only. The computation proceeds from layer 0 to layer L, changing the labels at each step and using the output of the last layer as the output of the whole model.

During the learning phase, the networks are adapted layer by layer, starting from $\mathcal{N}^1$ up to $\mathcal{N}^L$. Each network is trained using the original targets and the graphs constructed by the previous layer. More precisely, if G belongs to the original dataset $\mathcal{L}$, then G^{k-1} belongs to the learning set $\mathcal{L}^{k-1}$ of layer k. Moreover, the target of each node n in G^{k-1} is equal to the first layer target, i.e., $\boldsymbol{t}_n^{k-1} = \boldsymbol{t}_n$.

The reason why we expect that such an architecture can alleviate the long–term depencency problem is complex and can be explained as follows. In GNNs, the long–term depencences are observed when the network, in order to produce an output $\boldsymbol{o}_n$ at node n, has to integrate the information available in nodes that are far from n. If the decision to be taken at node n is complex, then it is likely that the network can use only the data in a small neighborhood of n. On the other hand, just because the GNN tries to combine such a data, the state $\boldsymbol{x}_n$ and the output $\boldsymbol{o}_n$ finally contain a sort of summarization of the information available in the neighborhood of n, even if the neighborhood is small. Thus, each layer of an LGNN, which takes in input $\boldsymbol{x}_n$ or $\boldsymbol{o}_n$, has the theoretical possibility to expand the neighborhood, so that the learning in LGNNs can be enriched layer by layer by taking into account far and far information.

It is worth noting that all the GNNs in the layered architecture could be trained also in a single step, by imposing the supervision only on the final layer and implementing a backpropagation of the error through the layers. However, due to the complexity and the depth of the architecture, using such an approach would have reintroduced the long–term dependency in a different form. For this reason, we opted for the layer by layer learning approach.

IV. Experimental results

The proposed LGNN model was tested on two artificial datasets and on two real–life benchmarks concerning the following problems: subgraph matching, localization of cliques, classification of mutagenic molecules, prediction of the secondary structure of proteins.

A. Dataset Descriptions

The subgraph matching problem: The subgraph matching problem consists in localizing a given subgraph S in a larger graph G (see Figure 4(A)). More precisely, the function τ that has to be learned is defined by $\tau(G_i, n_{i,j}) = 1$ if $n_{i,j}$ belongs to a subgraph of G_i, which is isomorphic to S, and $\tau(G_i, n_{i,j}) = -1$, otherwise. For instance, object localization in images and detection of active parts in chemical compounds can be implemented as subgraph matching problems [25], [26].

The dataset used in the experiments contains $1,000$ random graphs with 15 nodes and integer labels [16]. After the construction of each graph G, a predefined random subgraph S, with 7 nodes, was inserted into G. The dimensions of the graphs have been fixed in order to limit the variables to be evaluated during the experiments and, as a consequence, the computation time. The dimensions have been heuristically established having in mind the goal of selecting a dataset where the performance of common GNNs is far from 100%, and a possible improvement provided by the presented architecture can be clearly statistically identified. Actually, according to a larger experimentation on subgraph matching [16], GNNs obtain the lowest performance in the cases where the subgraph S is approximately a half of the graph G.

The clique localization problem: A clique of size k is a complete subgraph with k nodes[3] contained into a larger graph (see Figure 4(B)). Despite the fact that this is a subproblem of graph matching, the localization of cliques is a computationally difficult problem[4] and is related to several application domains, including bioinformatics and chemistry (see, f.i., [27], [28]).

The dataset used for this experiment contains $1,000$ random graphs with 15 nodes, where a clique of size 5 was inserted. The goal was that of detecting such cliques. More precisely, the GNN was trained to approximate the function defined by $\tau(G_i, n_{i,j}) = 1$ if $n_{i,j}$ belongs to a clique, and $\tau(G_i, n_{i,j}) = -1$, otherwise. The graph and the clique dimensions have been chosen following the same reasoning adopted for the subgraph matching problem and according to the previous experimentation carried out in [29].

[3] A graph is complete if there is an edge between each pair of nodes.
[4] Finding the target clique in a graph is known to be an NP–hard problem.

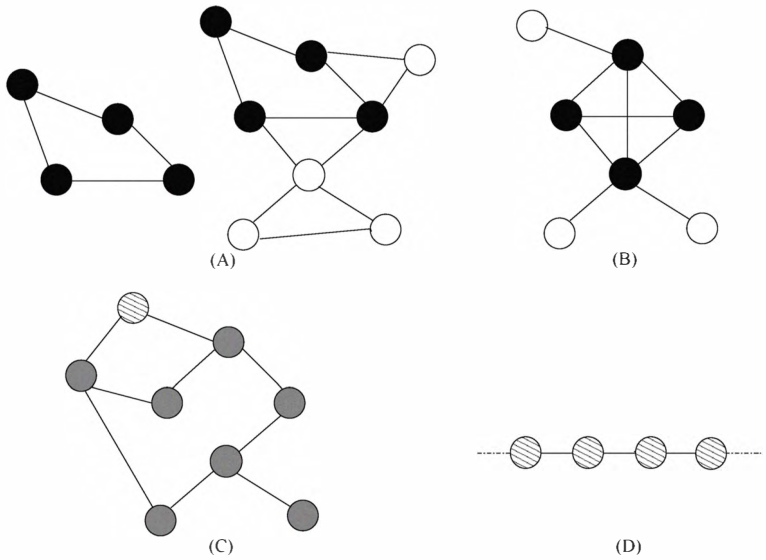

Fig. 4. Graphical information used in: (A) subgraph matching, (B) clique localization, (C) mutagenicity classification and (D) secondary protein structure prediction problems. In (A) and (B), black nodes have target 1, white nodes -1. In (C) and (D), supervision is imposed on dashed nodes, whereas grey nodes are unsupervised.

The Mutagenesis problem: The Mutagenesis dataset [5] contains the descriptions of 230 nitroaromatic compounds, which are common intermediate subproducts of many industrial chemical reactions. The goal of the benchmark consists of learning to predict the mutagenic compounds.

The dataset includes information on the atom bond structure and on the chemical properties of the molecules. In our experiments, the bond structure was used to construct a graphical representation of the compounds, where nodes stand for atoms and edges stand for bonds (see Figure 4(C)). The chemical properties are collected into the node labels, and each label is a 13–dimensional vector that contains features about the single atoms (the atom charge and a one–hot representation of the atom type) and four global features (the lowest unoccupied molecule orbitall, the water/octanol partition coefficient, and two precoded structural attributes) [5]. For each graph, there is only one supervised node, which has been chosen as the first atom in the original dataset. The desired output is 1 if the molecule is mutagenic, and -1 otherwise.

Secondary protein structure prediction: The secondary structure specifies regularly repeating local regions of proteins. The basic kinds of regions reported in literature are helixes, strands and coils. The information about the secondary structure of a protein is very important since it can be used either directly in biology and chemistry or, indirectly, in the prediction of the tertiary structure, which defines the 3D organizations of the amino acids (and accounts for the functionality of the protein).

The prediction of the secondary structure is a classification problem in a graphical domain. Actually, a protein can be simply represented by a sequence, where nodes stand for amino acids and edges denote their peptide bonds. The goal is to predict the kinds of region to which each amino acid belongs among the above mentioned three categories, i.e., to learn by examples a function τ such that $\tau(G_i, n_{i,j})$ describes the class of the region containing the amino acid n in the protein G.

For the presented results, the dataset used in [30] and public available at http://gruyere.ucd.ie/∼proteins/data/, which contains $2,171$ proteins and $344,653$ amino acids, has been used. For each amino acid, the available features are represented by a 20–dimensional real vector[5] and stored in the node labels. The target attached to each node is a 3–dimensional vector containing a one–hot coding of the three considered classes (helix, strand and coil).

B. Experimental settings

Unless explicitly stated, the following facts hold for all the experiments. The configuration of the base GNN model is: the transition function $f_{\boldsymbol{w}}$ and the output function $g_{\boldsymbol{w}}$ are implemented by three layered (one hidden layer) static networks, with linear activation function in the output layer and hyperbolic tangent activation function in the hidden layer; the networks implementing $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ have 5 hidden units; the state dimension is 2; the GNN weights are randomly initialized in the range $[-0.01, 0.01]$.

In each experiment, the dataset was splitted into a training, a validation and a test set. In the subgraph matching and in the clique localization experiments, the training set is constituted by 600 graphs, whereas both the validation and the test sets contain 200 examples. Based on the approach adopted in the papers where the datasets were introduced, the splitting of mutagenesis was carried out according to a 10–fold cross validation strategy, while a 5–fold cross validation was used for the prediction of the protein secondary structure. It is worth noting that, even if in the present work the

[5]The vector contains a one–hot representation of the amino acid type.

number of layers was selected to be large enough to show the "asymptotic" behavior of the proposed model, nevertheless, in practical applications, such a number can be appropriately chosen based on the validation set.

Each GNN was trained for a predefined number of epochs (heuristically chosen as $2,000$, $2,000$, 500 and $1,000$ for the subgraph matching, the clique localization, the mutagenesis and the secondary structure prediction problems, respectively) and tested every 10 epochs on the validation set. The network achieving the best error on the validation set was used also on the test set.

The adopted learning procedure is the Resilient BackPropagation (RPROP) algorithm, with standard parameters [31]. Finally, each experiment has been repeated five times: the average results and the standard deviation will be reported. In the artificial problems, the datasets have been re–constructed and the weights re–initialized at each repetition. On the other hand, for real–world benchmarks, the weights were reset at every fold[6].

C. Experimental results

Figure 5 shows the results achieved on the subgraph matching (first row) and on the clique localization (second row) problems, when the node labels contain the previous layer outputs (first column), the hidden states of the output network (second column) and both of them (third column). Each plot shows the accuracies, denoted by the points, and the standard deviations, denoted by the vertical bars, obtained by varying the number of layers in the architecture. Layer 0 represents the standard GNN architecture, where only a single layer is adopted.

According to Figure 5, the accuracies increase with the number of layers. Such an improvement is particularly evident in the first layers and becomes less significant (and the results may even oscillate) in the last layers. However, in all the cases, the multi–layer architecture clearly outperforms the standard GNN (layer 0).

Moreover, a comparison among the different approaches, with which new labels are constructed, points out that the architecture in which only the outputs computed in the previous GNN layer are used in the new labels achieves the best performance. Interestingly, the use of the states seems to give rise to larger variances. Actually, even if, in theory, the states should contain sufficient information to reproduce the outputs, however such an information is probably confused due to the large number of state configurations that is explored by the learning algorithm.

With respect to the performance improvements obtained by LGNNs, one may wonder if this is due to the larger number of parameters adopted by the layered architecture. In order to evaluate this aspect, different single layer GNNs with a larger dimension of states and a larger number of hidden nodes have been applied on the the same datasets. More precisely, architectures with $5, 15, 25, 35$ hiddens, both in the output and in the transition networks, and with state dimension varying in $2, 5, 10, 15$, have been evaluated. Table I reports the best performance achieved by such an experimentation along with the best configuration. Those results clearly prove that the same performance of an LGNN architecture cannot be obtained by simply increasing the parameters in a common GNN.

TABLE I

THE RESULTS ALONG WITH THE BEST CONFIGURATIONS OBTAINED BY A SINGLE LAYER GNN VARYING THE STATE DIMENSION AND THE NUMBER OF HIDDEN NODES IN BOTH THE TRANSITION AND THE OUT NETWORK.

	State dim.	Num. hiddens	Accuracy
Subgraph match.	3	35	81.43
Clique Loc.	10	5	89.07
Mutagenesis	2	35	92.26
Secondary Struct.	15	5	60.53

Moreover, it is worth mentioning that the computation resources needed by a k–layer LGNN architecture are smaller than those required by a GNN with k–time more parameters or states, since the computation time in GNNs is quadratic w.r.t. the state dimension [16].

Figure 6 shows the accuracies and the standard deviation reached by the LGNN architecture on mutagenesis (first row) and on secondary protein structure (second row) problems. Here, for the sake of simplicity, only the results obtained when the node label contains only the previously calculated output are reported, which has been proven to be the best strategy on the artificial datasets.

Interestingly, the behavior on the secondary protein structure dataset is similar to those previously observed. On the other hand, on the mutagenesis dataset, a larger variance is shown on the test set. The mean accuracy still improves, but the variance is so large that the improvement is not statistically significant. A comparison between the test and the training set suggests that the large variance is due to overfitting problems. Actually, the mutagenesis dataset is small and it can give rise to generalization issues. This points out a possible limit of the proposed approach, which is well known in machine learning: When many layers are used, many parameters are exploited and the architecture becomes capable to approximate a larger set of functions, but such capability may not be useful if a small training set is available, since it fails in generalizing to unseen patterns.

A comparison with the state–of–the–art on the considered benchmarks was not a main goal of this paper. Nevertheless, it is worth mentioning that the mean accuracy obtained by LGNNs on the mutagenesis dataset (90.78%) is larger than the best accuracy reported in literature (88%) [32], even if the large standard deviation limits the importance of such result. On the other hand, even if the performance on the secondary protein structure dataset is more than 10% points smaller than the performance achieved by Porter[7], the two results

[6]Notice that, in this case, each repetition includes a test on every fold of the cross validation splitting and the presented results will be, more precisely, the macro average and the macro standard deviation.

[7]Porter is a server for the prediction of the secondary protein structure currently available on the Internet [30]

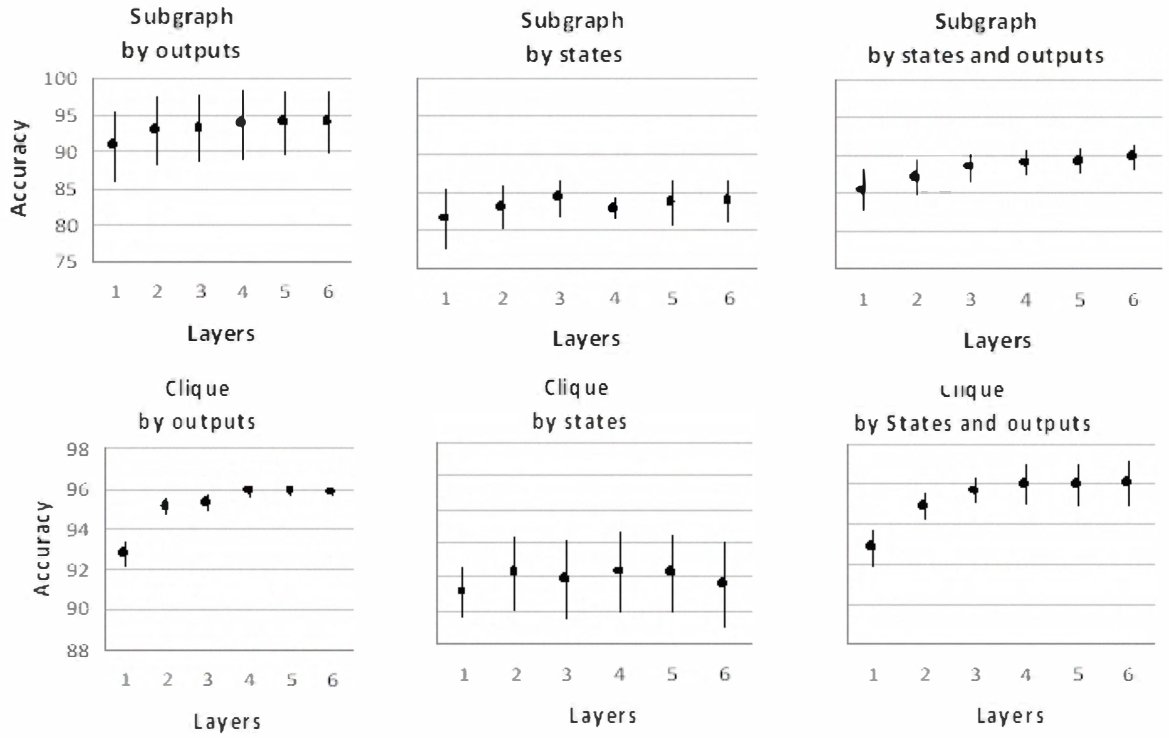

Fig. 5. The accuracies and the standard deviation (represented by the vertical bars) achieved by the LGNN architecture on the subgraph matching (first row) and on the clique localization (second row) problems. The first, the second and the third columns show the results obtained by inserting into the labels the previous layer outputs, the hidden states of the output network and both of them, respectively.

are not directly comparable. In fact, Porter adopts a complex pre and post–processing, which, due to time and resource constraints, we could not implement here. However, the core of Porter is a classifier based on recursive neural networks. Common GNNs are an extension of RNNs and can perfectly simulate their behaviour, so that we can reasonably expect that LGNNs can be useful in this domain in conjuction with an appropriate pre and post–processing.

V. Conclusions and Future Work

In this paper, we have presented a new composite architecture, called Layered GNN. In the LGNN model, each GNN is an expert that solves the considered problem using the original data and the expertise provided by the previous layers. The intuition that the progressive learning realized in LGNNs can face and control the long–term dependency problem, rising when GNNs process deep structures, was experimentally proven based on both synthetic and real–world data.

Actually, the way in which we have proposed to modify the basic GNN model is just one of the possible strategy that can be followed to allow recursive learning of deep structures. As a matter of fact, an alternative possibility is that of applying a *decomposition* policy, at least in those cases in which the original problem can be naturally partitioned into subproblems, whose partial solutions can then be used to move towards the whole solution. This is just the case,

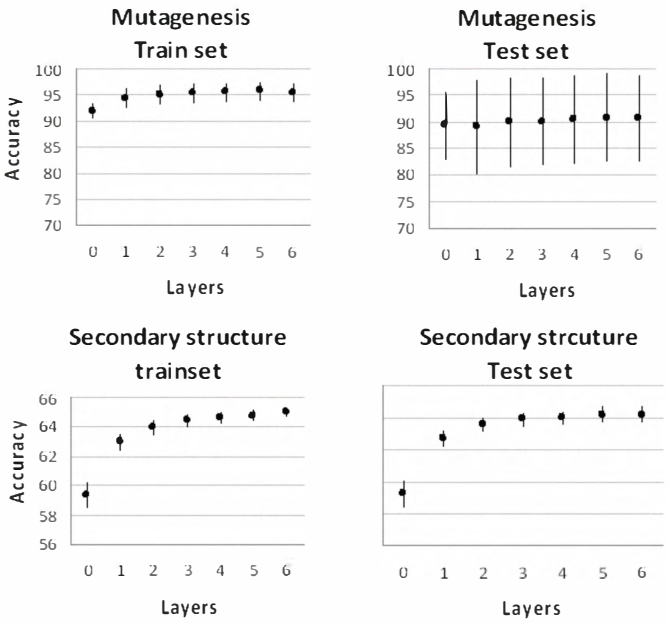

Fig. 6. The accuracies and the standard deviation (represented by the vertical bars) achieved by the LGNN architecture on mutagenesis (first row) and on secondary protein structure (second row) problems. The first and the second column show the results on training and test sets, respectively.

for example, of the clique localization problem since, if a node n does not belong to a clique of dimension N, it does not certainly belong to a clique of dimension $N-1$. Obviously the decomposition strategy just described represents a less general way to approach the GNN learning, but some preliminary experiments assess significant improvement in performance whenever it is appliable. A further possibility consists in solving, first, a classification problem on the graph labels — based, for instance, on bayesian networks, clustering or standard MLPs — and then using the information on the class label (possibly with the original labels themselves) as new labels for a graph, sharing the topology of the original one. Also in this case, preliminary results guarantee good preformances w.r.t. standard GNN. Finally, different combinations of the above mentioned methods are worth investating and they will constitute the object of future research.

REFERENCES

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

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

[3] M. Bianchini and F. Scarselli, "Artificial neural networks for processing graphs with applications to image understanding: A survey," in *Multimedia Techniques for Device and Ambient Intelligence*, E. Damiani and J. Jeong, Eds. Berlin/Heidelberg: Springer, 2009, pp. 179–199.

[4] C. Mooney and G. Pollastri, "Beyond the twilight zone: Automated prediction of structural properties of proteins by recursive neural networks and remote homology information," *Proteins*, vol. 77, no. 1, pp. 181–190, 2009.

[5] 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*, ser. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994, pp. 217–232.

[6] J. Cheng, M. J. Sweredoski, and P. Baldi, "Accurate prediction of protein disordered regions by mining protein structure data," *Data Mining and Knowledge Discovery*, vol. 11, no. 3, pp. 213–222, 2005.

[7] M. Bianchini, M. Maggini, and L. Sarti, "Recursive neural networks and their applications to image processing," in *Advances in Imaging and Electron Physics*, P. W. Hawkes, Ed. Elsevier – Academic Press, 2006, vol. 140, pp. 1–60.

[8] M. Bianchini, M. Maggini, L. Sarti, and F. Scarselli, "Recursive neural networks learn to localize faces," *Pattern Recognition Letters*, vol. 26, no. 12, pp. 1885–1895, 2005.

[9] ——, "Recursive neural networks for processing graphs with labelled edges: Theory and applications," *Neural Networks*, vol. 18, pp. 1040–1050, 2005.

[10] 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 *Proceedings of the IEEE International Joint Conference on Neural Networks*, 2006, pp. 778–785.

[11] A. C. Tsoi, M. Hagenbuchner, R. Chau, and V. Lee, "Unsupervised and supervised learning of graph domains," in *Studies in Computational Intelligence – Innovations in Neural Information Paradigms and Applications*. Berlin/Heidelberg: Springer, 2009, vol. 247, pp. 43–65.

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

[13] T. Kohonen and P. Sommervuo, "How to make large self–organizing maps for nonvectorial data," *Neural Networks*, vol. 15(8–9), pp. 945–952, 2002.

[14] B. Hammer, A. Micheli, M. Strickert, and A. Sperduti, "A general framework for unsupervised processing of structured data," *Neurocomputing*, vol. 57, pp. 3–35, 2004.

[15] M. Hagenbuchner and A. C. Tsoi, "A supervised self–organizing map for structures," in *Proceedings of the IEEE International Joint Conference on Neural Networks*, vol. 3, 2004, pp. 1923–1928.

[16] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[17] Y. Bengio, P. Simard, and P. Frasconi, "Learning long–term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[18] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends for Machine Learning*, 2010, to appear.

[19] G. E. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 2, pp. 1527–1554, 2006.

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

[21] M. Gori, M. Maggini, and L. Sarti, "A recursive neural network model for processing directed acyclic graphs with labeled edges," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, Portland (USA), July 2003, pp. 1351–1355.

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

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

[24] F. Pineda, "Recurrent back–propagation and the dynamical approach to adaptive neural computation," *Neural Computation*, vol. 1, pp. 161–172, 1989.

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

[26] 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.

[27] R. Samudrala and J. Moult, "A graph–theoretic algorithm for comparative modeling of protein structure," *Journal of Molecular Biology*, vol. 279, no. 1, pp. 287–302, 1998.

[28] N. Rhodes, P. Willett, A. Calvet, J. Dunbar, and C. Humblet, "Clip: similarity searching of 3D databases using clique detection," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 2, pp. 443–448, 2003.

[29] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "Computational capabilities of graph neural networks," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 81–102, 2009.

[30] G. Pollastri and A. Mclysaght, "Porter: a new, accurate server for protein secondary structure prediction," *Bioinformatics*, vol. 21, no. 8, pp. 1719–1720, 2005.

[31] M. Riedmiller and H. Braun, "A direct algorithm method for faster BackPropagation learning: the RPROP algorithm," in *Proceedings of the International Conference on Neural Networks*, vol. 1, Portland (USA), 1993, pp. 586–591.

[32] 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, pp. 211–220.