Pyramidal Reservoir Graph Neural Network

Filippo Maria Bianchi

Department of Mathematics and Statistics, UiT the Arctic University of Norway.
Hansine Hansens veg 18 – 9019 Tromsø (Norway),
NORCE the Norwegian Research Centre

Claudio Gallicchio*

Department of Computer Science, University of Pisa
Largo B. Pontecorvo, 3 – 57127 Pisa (Italy)

Alessio Micheli

Department of Computer Science, University of Pisa
Largo B. Pontecorvo, 3 – 57127 Pisa (Italy)

Abstract

We propose a deep Graph Neural Network (GNN) model that alternates two types of layers. The first type is inspired by Reservoir Computing (RC) and generates new vertex features by iterating a non-linear map until it converges to a fixed point. The second type of layer implements graph pooling operations, that gradually reduce the support graph and the vertex features, and further improve the computational efficiency of the RC-based GNN. The architecture is, therefore, pyramidal. In the last layer, the features of the remaining vertices are combined into a single vector, which represents the graph embedding. Through a mathematical derivation introduced in this paper, we show formally how graph pooling can reduce the computational complexity of the model and speed-up the convergence of the dynamical updates of the vertex features. Our proposed approach to the design of RC-based GNNs offers an advantageous and principled trade-off between accuracy and complexity, which we extensively demonstrate in experiments on a large set of graph datasets.

Keywords: Reservoir Computing, Graph Echo State Networks, Graph Neural Networks, Graph Pooling

*Corresponding author
Email addresses: filippo.m.bianchi@uit.no (Filippo Maria Bianchi), gallicch@di.unipi.it (Claudio Gallicchio), micheli@di.unipi.it (Alessio Micheli)

1. Introduction

1.1. Representation learning on graphs with neural networks

Graphs naturally express entities and their relationships found in many kinds of real data, ranging from molecules to biological and social networks, just to mention a few noteworthy cases. There is an established tradition in processing structured data with Neural Networks, which goes back to the '90s and recently found a renewed interest in the field of deep learning for graphs [1, 2]. Recursive Neural Networks for processing trees [3, 4, 5] and directed acyclic graphs [6], provided a neural implementation of a state transition system traversing the input structures to generate the embedding of the input data. To process general graphs it was necessary to handle directed cycles and undirected edges, which translate into mutual dependencies among the state variables represented by the neural units.

The Graph Neural Network (GNN)[7] and the Neural Network for Graphs (NN4G) [8] started the field following two different lines. The NN4G pioneered the class of deep spatial graph convolution approaches. In such approaches, the neural units traverse the graphs with shared weights and the mutual dependencies among state variables are modeled by stacking layers, which progressively capture the vertices' context. Recent advancements of the convolutional and related approaches (including kernel based approaches) can be found for instances in [9, 10, 11, 12, 13, 14, 15].

The GNN exploits a Recursive Neural Network to implement a state transition that allows cycles, whereas the contractivity of the state dynamics ensures the stability of the recursive encoding process. In this approach, the context of each vertex is formed through graph propagation, which is iterated until the dynamics converge to a steady-state. The fixed point of the dynamical system is used to represent (or *embed*) the vertices of input graphs. Theoretical approximation capability and VC dimension of the GNN were recently studied [16].

In a context characterized by high complexity and computational cost related to the processing of structured data and the use of deep models, the need for efficient approaches is becoming more and more important. For sequences and trees, the Reservoir Computing (RC) [17] paradigm provides efficient recurrent/recursive models based on fixed (randomized) values of the recurrent weights under stability conditions of the dynamical system (Echo State Property - ESP) [18, 19]. A GNN based on RC was introduced in [20], whereas the contractivity of the reservoir dynamics (ESP) assures the stability condition of the GNN, without any need to alternate training and convergence as for GNN. The RC model for graphs has been recently extended to deep architectures in [21], which combines the advantages of a hierarchical abstraction of the input graphs representation (embedding) through the layers and the efficiency of randomized neural networks.

Another relevant field of graph representation learning is graph pooling, which reduces the dimension of the graph by clustering or by dropping the vertices. Global graph representations can be derived by combining at once the

embedding of vertex features and use them for inference on downstream tasks at graph level. However, some networks may exhibit scale-dependent behaviors, which have to be accounted for when solving the task. In these cases, pooling operations are exploited to build deep architectures of Neural Networks for graphs, which compute local summaries on the graph to gradually distill the global properties necessary for graph-level inference [9, 22, 23, 24].

So far, pooling operations have been only used in conjunction with Neural Networks for graphs that learn vertex representations through functions endowed with parameters optimized with gradient descent [25]. This leaves open the challenge to investigate the contribution of pooling in randomized Neural Networks. To fill this gap, in this paper, we propose a further advancement in the design of efficient GNN architecture combining the RC and the graph pooling strategies, exploring their synergy in terms of classification accuracy and computational time.

1.2. Contributions

We extend our previous work in [26] and propose a deep architecture composed of RC layers, which generate vertex representations, interleaved with graph pooling operations. Since the adopted GNN architecture based on RC generates untrained embeddings, we rely on pooling methods that pre-compute the coarsened graphs without supervision. The potential of hierarchical RC architectures in designing fast and deep models for graph classification has been recently shown in [21]. In this context, we investigate the benefit of pooling to further reduce the computational complexity of such randomized neural models.

The contributions of our work are summarized as follows:

- We propose a deep pyramidal architecture based on RC for generating graph embeddings. Thanks to suitable graph pooling operations, the proposed model further improves the computational efficiency of RC architectures for graphs.

- We derive a mathematical proof of the convergence time in the recursive update necessary to compute the vertex features. This formal analysis, beside explaining the speed-up introduced by the pooling operations, links the computational complexity of generic GNNs based on RC with graph theoretical properties of the input data.

- We provide an extensive experimental evaluation of the Pyramidal RC architecture in graph classification tasks, by testing different configurations and considering a large variety of datasets. Notably, we introduce two new datasets for graph classification, with the aim to provide a solid benchmark for testing the proposed method and also other types of GNNs and graph kernels. We also perform dimensionality reduction on the graph embeddings generated by the proposed architecture, which allows us to perform a qualitative evaluation of the embeddings.

1.3. Paper organization

The remainder of this paper is structured as follows. We introduce the basic concepts of RC for graphs in Section 2. In Section 3 we introduce graph pooling and discuss three pooling operators that pre-compute coarsened versions of the graph based on the topological structure. Building upon the RC for graphs and graph pooling, we introduce the Pyramidal Reservoir Graph Neural Network architecture in Section 4. In Section 5 we perform a theoretical analysis and derive an upper-bound of the computational complexity of an RC layer. The experimental assessment of the approach is given in Section 6. Finally, we draw our conclusions in Section 7.

2. Reservoir Computing for Graphs

In this section we present the elementary concepts of RC for graph structures. We start by introducing the necessary terminology for graph processing used in the rest of the paper in Section 2.1. Then, in Section 2.2, we describe the RC approach to develop graph embeddings.

2.1. Preliminaries on Graphs.

We denote a graph $\mathcal{G}$ as a tuple $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$, where $V_\mathcal{G}$ is the set of vertices and $E_\mathcal{G} \subset V_\mathcal{G} \times V_\mathcal{G}$ is the set of edges. The number of vertices of the graph $\mathcal{G}$ is denoted by $N_\mathcal{G}$, the number of edges is denoted by $M_\mathcal{G}$. The pattern of connectivity among the vertices in $\mathcal{G}$ is represented by the $N_\mathcal{G} \times N_\mathcal{G}$ adjacency matrix $\mathbf{A}_\mathcal{G}$, where $\mathbf{A}_\mathcal{G}(i,j) = 1$ whenever there is an edge between vertex i and vertex j, and $\mathbf{A}_\mathcal{G}(i,j) = 0$ otherwise.

While the computational framework that we describe in this paper is able to deal with both directed and undirected graphs, we here restrict our analysis to the undirected case, which leads to symmetric adjacency matrices $A_\mathcal{G}$. Given a vertex $v \in V_\mathcal{G}$, we define its neighborhood as the set of vertices that are adjacent to v, denoted as $\mathcal{N}_\mathcal{G}(v) = \{v' \in V_\mathcal{G} : (v, v') \in E_\mathcal{G}\}$. In this context, the degree of a vertex $v \in V_\mathcal{G}$ is defined as the size of its neighborhood $\mathcal{N}_\mathcal{G}(v)$. Also notice that in this case the sum of the degrees of all the vertices equals $2M_\mathcal{G}$.

We also introduce the symmetrically normalized adjacency matrix $\tilde{\mathbf{A}}_\mathcal{G} = \mathbf{D}_\mathcal{G}^{-1/2}\mathbf{A}_\mathcal{G}\mathbf{D}_\mathcal{G}^{-1/2}$, where $\mathbf{D}_\mathcal{G}$ is a diagonal degree matrix, i.e. a diagonal matrix containing the degrees of the vertices $v \in V_\mathcal{G}$. For our purposes, to each vertex $v \in V_\mathcal{G}$ it is associated an F-dimensional feature vector $\mathbf{x}(v) \in \mathbb{R}^F$. Let $\mathbf{X}_\mathcal{G} \in \mathbb{R}^{N_\mathcal{G} \times F}$ be the matrix whose i-th row contains the feature vector associated to the i-th vertex. We can then represent the graph $\mathcal{G}$ by the couple $(\mathbf{A}_\mathcal{G}, \mathbf{X}_\mathcal{G})$. In what follows, to ease the notation, we drop the subscript $\mathcal{G}$ whenever the reference to the graph in question is unambiguous or evident from the context.

2.2. Reservoir Graph Neural Networks

Reservoir Computing (RC) is a popular paradigm for modeling Recurrent Neural Networks based on untrained (but asymptotically stable) hidden dynamics [17, 27]. The RC approach to process graph structures is based on computing

representations, or *graph embeddings*, as fixed points of a dynamical system[1]. The approach has been introduced in [20] under the name of Graph Echo State Network, and subsequently extended towards deep architectures in [21], under the name of Fast and Deep Graph Neural Network (FDGNN). Crucially, as long as the embeddings are obtained by a stable system, the weights on the internal connections of the neural architecture can be left untrained, and training algorithms can be limited to a simple readout layer. Here we summarize the salient aspects of graph processing by untrained recursive layers in a deep RC-based neural network for graphs, hereafter referred to as *Reservoir Graph Neural Network* (RGNN).

We consider a neural architecture for graphs composed of a number of L hidden recursive reservoir layers, where each layer l computes an embedding for a vertex v by means of the following function:

$$\mathbf{h}^{(l)}(v) = \tanh\left(\sum_{v' \in \mathcal{N}(v)} \mathbf{h}^{(l)}(v')\mathbf{W}^{(l)} + \mathbf{x}^{(l)}(v)\mathbf{V}^{(l)}\right), \tag{1}$$

where $\mathbf{h}^{(l)}(v) \in \mathbb{R}^H$ is the embedding, or *state* computed for vertex v by layer l. Matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{H \times H}$ is the recurrent weight matrix and modulates the contribution of the neighbors information in determining the embedding for the vertex v. Moreover, $\mathbf{x}^{(l)}(v)$ is the feature vector that plays the role of input information for the layer l under consideration. Following the deep architectural construction, $\mathbf{x}^{(1)}(v)$ is the feature vector associated to v in the input graph, i.e. $\mathbf{x}^{(1)}(v) = \mathbf{x}(v)$. For $l > 1$, the feature vector $\mathbf{x}^{(l)}(v)$ is instead obtained as the embedding computed by the previous layer for the same vertex, i.e. $\mathbf{x}^{(l)}(v) = \mathbf{h}^{(l-1)}(v)$. Accordingly, matrix $\mathbf{V}^{(l)}$ modulates the influence of the input information in determining the embedding, with $\mathbf{V}^{(1)} \in \mathbb{R}^{H \times F}$ for the first layer, and $\mathbf{V}^{(l)} \in \mathbb{R}^{H \times H}$ for layers $l > 1$. Notice that (1) is introduced here to set the ground for the mathematical description of the deep RC model, and it is the basic building block of the iterated system on graphs described later in this section (see (4)). Essentially, the states for all vertices (i.e., the $\mathbf{h}^{(l)}(v)$ embeddings) are started with (usually $\mathbf{0}$) initial conditions and then (1) is iterated for all vertices until a fixed point is reached. The computation then continues with the successive layer until the last one in the architecture is reached. To ease the architectural setup and description, we are making an assumption[2] here that every reservoir layer has the same number of neurons H. The computation of graph embeddings by reservoir layers in RGNNs is illustrated in Figure 1.

[1] The approach of iterating the operation of a neural architecture until reaching the fixed point of the internal representation has recently been revived under the umbrella of deep equilibrium neural models [28, 29].

[2] Using the same number of neurons in every reservoir layer is a simple and commonly adopted choice of architectural design in Deep RC, although the models are generally more flexible and allow a different choice of H for every layer.

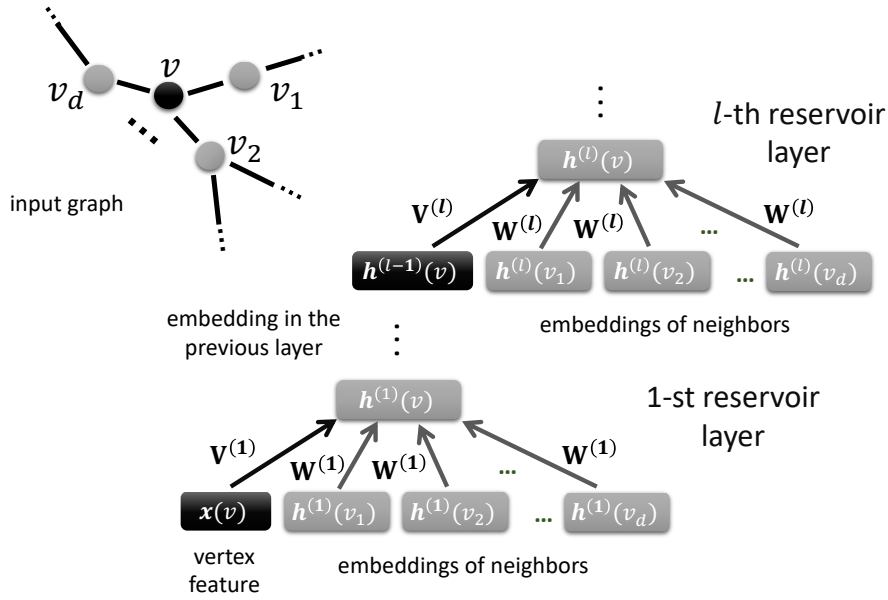

Figure 1: Stack of RGNN layers for graph embedding, applied to a vertex v in the input graph (top left).

For any given layer in the deep architecture, $l = 1, \ldots, L$, the state transition equation in (1) is applied to every vertex v of an input graph. Instead of considering the operations on the vertices in isolation, it is convenient to represent the embedding operation performed by a RGNN layer on the entire graph. This can be done by grouping in a row-wise fashion the states for all the N vertices of the graph computed at layer l in a matrix $\mathbf{H}^{(l)} \in \mathbb{R}^{N \times H}$, and analogously grouping the feature vectors in a matrix $\mathbf{X}^{(l)}$ (with $\mathbf{X}^{(1)} \in \mathbb{R}^{N \times F}$ for the first layer, and $\mathbf{X}^{(l)} \in \mathbb{R}^{N \times H}$ for $l > 1$). The embedding of the entire graph computed by the l-th RGNN layer can then by described as follows:

$$\mathbf{H}^{(l)} = \tanh\left(\mathbf{A}\mathbf{H}^{(l)}\mathbf{W} + \mathbf{X}^{(l)}\mathbf{V}^{(l)}\right), \tag{2}$$

which expresses a recursive relation in $\mathbf{H}^{(l)}$. In general, existence and uniqueness of solutions of (2) is not ensured (for instance, when the input graph contains cyclic substructures or undirected connections). Fortunately, taking the RC approach, we can study (2) as the function governing a dynamical system and impose a global asymptotic stability property to its dynamics, called Graph Embedding Stability (GES) [21], to guarantee existence and uniqueness of solutions. Under the GES property, the system in (2) will converge to a fixed point (that depends on both $\mathbf{A}$ and $\mathbf{X}$) upon repeated iteration. In [21] two conditions have been introduced, one sufficient and one necessary for initializing the $\mathbf{W}^{(l)}$ matrices to satisfy the GES stability property.

Essentially, to control the stability of the reservoir we need to carefully control the eigenvalues of the system in (2). Since the spectral radius of the adjacency matrix $\mathbf{A}$ is unbounded, it is necessary to normalize $\mathbf{W}$ by the maximum vertex degree (see [21] for details). If we are dealing with multiple graphs, we can normalize $\mathbf{W}$ by using the maximum vertex degree among all the graphs in the training set. However, in this way graphs with low vertex degree would be subject to a normalization that is too strong and stability issues might arise when processing out-of-sample graphs with a vertex degree larger than the maximum one found in the training set.

A more elegant and effective solution is to replace $\mathbf{A}$ with the symmetrically normalized adjacency matrix $\tilde{\mathbf{A}}$ and obtain the following formulation:

$$\mathbf{H}^{(l)} = \tanh\left(\tilde{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W} + \mathbf{X}^{(l)}\mathbf{V}^{(l)}\right). \tag{3}$$

Since the eigenvalues of $\tilde{\mathbf{A}}$ are always contained in $[-1, 1]$, it is possible to factor out from $\mathbf{W}^{(l)}$ the dependency on the vertex degree. In particular, the formulation in (3) allows us to control the stability of the reservoir system for graphs by simply controlling the eigenvalues of $\mathbf{W}^{(l)}$, as in conventional RC approaches. Accordingly, $\mathbf{W}^{(l)}$ is initialized randomly with values from a uniform distribution in $(-1, 1)$, and then re-scaled to have a desired *spectral radius*[3], denoted as $\rho^{(l)}$, exploring values in the range $(0, 1)$ for stability [21]. Similarly, the elements in $\mathbf{V}^{(l)}$ are randomly initialized from a uniform distribution on $(-1, 1)$, and then re-scaled. For the first layer, we use ω_{in} as *input scaling* coefficient, while for successive layers $l > 1$ we use $\omega_{hid}^{(l)}$ as *hidden scaling* coefficient (between two consecutive reservoir layers). Usually, the same values of ρ and ω_{hid} are shared among all the layers. As in common RC settings, ρ, ω_{in} and ω_{hid} are treated as hyper-parameters of the network (to be fitted on a validation set for each task at hand).

To compute the embedding for a given graph, we use (3) as the iterated map of a discrete-time dynamical system, i.e.:

$$\mathbf{H}^{(l)}[t+1] = \tanh\left(\tilde{\mathbf{A}}\ \mathbf{H}^{(l)}[t]\ \mathbf{W}^{(l)} + \mathbf{X}^{(l)}\ \mathbf{V}^{(l)}\right), \tag{4}$$

which is applied until convergence to its fixed point $\mathbf{H}^{(l)*}$. As initial condition, it is common to set $\mathbf{H}^{(l)}[0] = \mathbf{0}$ for all layers. In practice, for each RGNN layer (4) is iterated until the distance between two consecutive iterations becomes smaller than a convergence threshold ϵ, i.e. $\|\mathbf{H}^{(l)}[t+1] - \mathbf{H}^{(l)}[t]\|_2 < \epsilon$, or until a maximum number of iteration max_{iter} is reached. The encoding operation proceeds from the first reservoir layer up to the last one in the architecture, bringing to convergence the dynamical systems one layer after the other. In this regard, notice that for $l > 1$, the $\mathbf{X}^{(l)}$ term in (4) corresponds to the fixed point reached by the previous layer in the deep architecture, i.e., $\mathbf{X}^{(l)} = \mathbf{H}^{(l-1)*}$.

[3] I.e., the largest among its eigenvalues in modulus.

3. Graph pooling

Pooling operators in GNN architectures can be partitioned in two main classes: *feature-based* and *topological* pooling operators.

Feature-based pooling methods compute a coarsened version of the graph through differentiable functions, which can be used to learn a cluster assignment matrix that aggregates graph vertexes into super-nodes [24, 30] or to implement a selection function that determines if the vertices should be kept or dropped [31, 32]. The differentiable functions usually take as input the vertex representations, which change across the different layers of the GNN during training. The functions are parametrized by weights that are learned end-to-end, along with the other GNN parameters, by minimizing the loss of a downstream task (e.g., classification or regression loss) and additional (optional) regularization losses. Feature-based pooling methods are very flexible, as they adapt the coarsening on the data and the task at hand. However, GNNs with feature-based pooling layers have more parameters and, thus, their training is slower and more difficult. Most importantly, there is a weak synergy between feature-based pooling and RGNN layers, since the former require to be trained with gradient descent, while the latter generate unsupervised vertex representation using randomized and untrained weights.

Topological pooling methods account only for the graph topology described by the adjacency matrix $\mathbf{A}$ and they are typically unsupervised. These methods pre-compute the coarsened graphs by optimizing some unsupervised loss function, such as a graph cut or a vertex clustering objective. The result of the coarsening procedure is, therefore, not influenced by the vertex features or by a downstream task, like in the case of feature-based methods. Pre-computing graph coarsening not only relieves the need of adapting additional weights with supervised training, but also provides a strong inductive bias that helps to avoid degenerate solutions, such as entire graphs collapsing into a single node or entire graph sections being discarded. This is important when dealing with small datasets in tasks such as graph signal classification. Topological pooling methods can be effectively combined with reservoir computing layers, which fit their vertex representations to the pre-computed coarsened graphs. By alternating reservoir computing and topological pooling layers, it is possible to gradually extract hierarchical graph features in a completely unsupervised fashion.

In the following, we provide the details of the three different topological pooling methods that are considered in this paper. First, in Section 3.1 we introduce *Graclus* pooling. Then, in Section 3.2 we illustrate the *Non-negative Matrix Factorization* pooling. Finally, in Section 3.3 we describe the recently introduced *Node Decimation Pooling*.

3.1. Graclus pooling

Graclus is a hierarchical spectral clustering algorithm that has been originally proposed in [33] and it can be used to compute coarsened graphs. *Gr-*

aclus has been implemented in several GNN architectures to perform pooling [34, 9, 35, 36]. The l-th pooling operation first clusters together two vertices v_i and v_j into a new vertex v_k. Then, a standard pooling operation (average or max pool) is applied to compute the vertex feature $\mathbf{x}^{(l+1)}(v_k)$ of the new vertex v_k from $\mathbf{x}^{(l)}(v_i)$ and $\mathbf{x}^{(l)}(v_j)$. To make the pooling output consistent with the clusters assignment, the rows of graph signal $\mathbf{X}$ are rearranged so that vertices v_i and v_j end up in consecutive positions.

Graclus pooling has two main drawbacks. First, the connectivity of the original graph is not preserved in the coarsened graphs and the spectrum of their associated Laplacians might not be contained in the spectrum of the original Laplacian. Second, *Graclus* pooling adds "fake" vertices by constructing a balanced binary tree, so that the number of vertices can be exactly halved at each pooling step. Unfortunately, this not only injects noisy information in the graph, but also increases the computational complexity in the GNN, both in the training and in the evaluation step.

3.2. Non-negative Matrix Factorization (NMF) pooling

Originally proposed in [22], this approach exploits the non-negative matrix factorization to cluster the rows (or the columns) of the adjacency matrix of the graph. Such an approach is slightly different from *Graclus*, where the vertices of the graph are partitioned with spectral clustering. In particular, the pooling algorithm uses the *NMF* factorization such that $\mathbf{A} \approx \mathbf{QS}$, where $\mathbf{Q} \in \mathbb{R}^{N \times K}$ has the role of cluster representatives matrix, and $\mathbf{S} \in \mathbb{R}^{K \times N}$ gives a soft-clustering of the columns in $\mathbf{A}$. Given the vertices features before the pooling $\mathbf{X}$, the new vertices features are computed as $\mathbf{X}_{\text{pool}} = \mathbf{S}^T\mathbf{X}$. The new adjacency matrix is computed as $\mathbf{A}_{\text{pool}} = \mathbf{S}^T\mathbf{AS}$.

The main drawback of this pooling method is that *NMF* does not scale well to large graphs. Indeed, both the space and memory complexity are cubic in the number of vertices, meaning that *NMF* cannot be computed only for relatively small graphs [37]. A second drawback is that the coarsened graphs obtained with *NMF* pooling are extremely dense. This poses another scalability issue, since dense graphs are expensive to store and the sparse operations (commonly used to implement multiplications with the adjacency matrix) become extremely inefficient if the graph density is too high.

3.3. Node Decimation Pooling (NDP)

This pooling scheme for GNNs, originally proposed in [23], computes coarsened graphs in three steps.

1. The vertices of the original graph are divided in two sets according to the MAXCUT partition and then one of the two sides of the partition is dropped, reducing the number of vertices approximately by half. If two vertices are strongly connected on the graph, i.e., there are many paths that connects them, they will be assigned to opposite sides of the MAXCUT partition. The idea is that such strongly connected vertices will exchange a lot of information during the message-passing operations

implemented by the GNN and, therefore, their content will be very similar and redundant. In this sense, one side of the MAXCUT partition will summarize well the whole graph.

2. After dropping the vertices and all their incident edges, the resulting graph is likely to be disconnected. The remaining vertices are connected together by a link construction procedure based on Kron reduction [38]. Kron reduction enjoys the connectivity preservation property, i.e., it generates a new graph where two vertices are connected if and only if there was a path between them in the original graph. Additionally, Kron reduction does not introduce self-loops, guarantees the preservation of resistance distance, and preserves the spectrum of the original graph [39]. As we will discuss later, these properties prevent the spectrum of the coarsened graph to grow and allows a faster convergence of the RGNN layers.
3. A sparsification procedure allows to prune the *weak* edges from the coarsened graph, i.e., those edges associated with small weights in the weighted adjacency matrix. The sparsification procedure is controlled by an hyperparameter δ that determines the amount of edges to be dropped. The graph sparsification allows to drop a large amount of edges at the cost of compromising the graph structure. In this sense, graph sparsification trades the computational cost (which scales linearly with the number of edges) with the quality of the graph representations

Each method gives in output a coarsened graph, $\mathbf{A}_{\text{pool}}$ and a pooling matrix $\mathbf{S}$ used to compute the new vertex features, $\mathbf{X}_{\text{pool}} = \mathbf{S}^T\mathbf{X}$. In the case of *Graclus* and *NMF*, $\mathbf{S}$ is a matrix that defines how the vertices should be clustered together. Instead, in *NDP* $\mathbf{S}$ is a selection matrix that identifies the vertices that should be kept or dropped.

4. Pyramidal Reservoir GNN

In this section we describe the proposed modular GNN architecture composed of L blocks, where RGNN layers are interleaved by pooling layers.

A pooling operator can be applied recursively to compute increasingly coarsened versions of the adjacency matrix. We introduce the notation $p^l(\mathbf{A})$ to indicate that the pooling operator has been applied l times to $\mathbf{A}$. For example:

- $\mathbf{A}^{(1)} = p^0(\mathbf{A}) = \mathbf{A}$,
- $\mathbf{A}^{(2)} = p(\mathbf{A}) = \mathbf{A}_{\text{pool}}$,
- $\mathbf{A}^{(3)} = p^2(\mathbf{A}) = p(p(\mathbf{A})) = p(\mathbf{A}_{\text{pool}})$.

The l-th block consists of an RGNN layer and a pooling layer. The RGNN layer computes the new vertex features $\mathbf{H}^{(l)}$, using the vertex features $\mathbf{X}^{(l)}$ and the adjacency matrix $\mathbf{A}^{(l)}$. The pooling layer generates a pooled version of the vertex embeddings $\mathbf{X}^{(l+1)} = \left(\mathbf{S}^{(l)}\right)^T \mathbf{H}^{(l)}$, which become the input vertex features for the RGNN layer in the next block. The last block in the architecture

consists only of a RGNN layer. The pooling matrices $\mathbf{S}^{(l)}, l = 1, \ldots, L-1$ and the coarsened adjacency matrices $\mathbf{A}^{(l)} = p^{l-1}(\mathbf{A}), l = 1, \ldots, L$, are obtained according to one of the three graph pooling scheme, *Graclus*, *NMF* or *NDP*, described in Section 3.

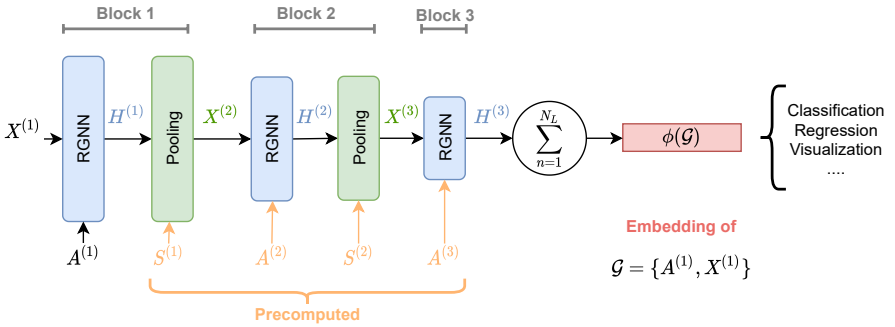

Figure 2: Schematic depiction of a Pyramidal RGNN with $L = 3$ blocks. In every block l, the RGNN layer computes the new vertex features $\mathbf{H}^{(l)}$ from the vertex features in the original input graph, $\mathbf{X}^{(1)}$ (or from those generated by the previous block $\mathbf{X}^{(l)}$, for $l > 1$), and from the l-th version of the adjacency matrix, $\mathbf{A}^{(l)}$. The pooling layer reduces the vertex features, according to a pooling matrix $\mathbf{S}^{(l)}$. The corasened versions of the adjacency matrix $\mathbf{A}^{(l)}, l = 1, \ldots, L$, and the pooling matrices $\mathbf{S}^{(l)}, l = 1, \ldots, L-1$ are pre-computed by a graph pooling algorithm. The vertex features of the last layer, $\mathbf{H}^{(L)}$, are aggregated all together to form the graph embedding $\phi(\mathcal{G})$, which can be used to perform different types of supervised or unsupervised tasks.

After the last pooling operation, the number of remaining vertices N_L is usually less than the number of vertices in the original graph, but greater than 1, i.e., $1 \leq N_L \leq N$. Therefore, to obtain a vectorial graph representation, we combine the N_L vertex features with an aggregation operator. Common aggregation operators are the average and the sum; we used the latter in this work and obtained a graph embedding vector $\phi(\mathcal{G}) \in \mathbb{R}^H$, defined as

$$\phi(\mathcal{G}) = \sum_{n=1}^{N_L} \mathbf{H}^{(L)}[n, :], \tag{5}$$

where $\mathbf{H}^{(L)}[n, :]$ is the n-th row of the matrix of vertex features in the last block L of the architecture. The sum aggregation can be considered as a global pooling operation, which replaces the local pooling layer in the last block. Figure 2 reports a schematic depiction of a Pyramidal RGNN with $L = 3$ blocks: [`RGNN`, `Pool`]-[`RGNN`, `Pool`]-[`RGNN`].

Both the RGNN and the topological pooling layers do not require any form of supervised training: the first exploits randomized weights to compute the vertex representations in a stable fashion, while the latter pre-computes coarsened graphs by optimizing some unsupervised loss, such as a clustering objective. Therefore, each graph embedding $\phi(\mathcal{G})$ is an *unsupervised* vectorial representation of the original graph $\mathcal{G} = (\mathbf{A}_\mathcal{G}, \mathbf{X}_\mathcal{G})$.

The vectorial representations can be used to solve supervised downstream tasks, such as graph classification or graph regression, by means of common classification or regression models for vectorial data that act as readout layer. In line with typical RC approaches [17], in this paper we use a linear readout layer trained by ridge regression when we use the graph embeddings $\phi(\mathcal{G})$ to solve classification tasks. Using a linear readout is also aligned with common approaches followed by other types of randomized architectures [40]. We notice that since the graph embeddings $\phi(\mathcal{G})$ are computed without supervision, they can, in principle, also be used in tasks such as clustering and unsupervised dimensionality reduction.

5. Convergence Analysis and Computational Cost

In this section, we analyze the cost of running RGNNs on graph structures. The aim is to show the connection between this cost and some major structural properties of the input graph in question (number of vertices, number of edges, and spectrum of the graph). First, in Section 5.1 we study the convergence behavior of RGNN layers on graphs. Then, in Section 5.2 we present the computational cost analysis. Notice that Section 5.1 is mainly devoted to building the mathematical tools necessary for characterizing the asymptotic behavior of contractive RGNN systems. The readers interested in the final results on the computational cost analysis can directly jump to Section 5.2.

5.1. Convergence Analysis

In this section we analyze the convergence behavior of reservoir systems on graph structures. We recall from Section 2 that an RGNN layer with H recursive neurons implements a hidden state dynamics given by the equation[4]:

$$\mathbf{H}[t+1] = \tanh(\tilde{\mathbf{A}}\,\mathbf{H}[t]\,\mathbf{W} + \mathbf{X}\,\mathbf{V}), \tag{6}$$

where $\mathbf{H}[t]$ is the matrix of graph embeddings, $\tilde{\mathbf{A}}$ is the normalized adjacency matrix of the graph, $\mathbf{X}$ is the input feature matrix at the current layer, while $\mathbf{W}$ and $\mathbf{V}$ are respectively the recurrent and input weight matrices. Finally, tanh denotes the element-wise applied non-linearity of the state transition. In what follows, we use $\|\cdot\|$ to refer the 2-norm of a matrix (i.e., its largest singular value). Equation (6) is iterated until convergence to its fixed point, with a convergence threshold denoted as ϵ. Our main result in this section is to link the number of iterations required for convergence, denoted by T, to the spectrum of the input graph. We restrict our analysis to the cases in which the state transition function in (6) is contractive. In Lemma 1 we show that the contraction coefficient for reservoirs on graphs can be expressed as $K = \rho(\tilde{\mathbf{A}})\|\mathbf{W}\|$. This result is then used in Theorem 1 to compute the value of T. Interestingly, we finally observe in Corollary 1 that the derived expression for T is an increasing function with $\rho(\tilde{\mathbf{A}})$, i.e. the spectral radius of $\tilde{\mathbf{A}}$.

[4] In this section we remove the layer superscript to ease the notation.

Lemma 1 (Lipschitz continuity of reservoir systems on graphs)**.** *Consider a reservoir system on graphs governed by the state transition equation in* (6)*. The system is Lipschitz continuous with constant* $K = \rho(\tilde{\mathbf{A}})\|\mathbf{W}\|$*.*

Proof. Let us consider one iteration step of the reservoir system in (6), applied to the graph $(\tilde{\mathbf{A}}, \mathbf{X})$, and starting from states $\mathbf{H}$ and $\mathbf{Z}$. Under the action of the reservoir system, states $\mathbf{H}$ and $\mathbf{Z}$ are mapped respectively to $\tanh(\tilde{\mathbf{A}}\mathbf{H}\mathbf{W}+\mathbf{X}\mathbf{V})$ and $\tanh(\tilde{\mathbf{A}}\mathbf{Z}\mathbf{W} + \mathbf{X}\mathbf{V})$. The distance between the mapped states can then be bounded as follows:

$$\begin{aligned}
&\|\tanh(\tilde{\mathbf{A}}\mathbf{H}\mathbf{W} + \mathbf{X}\mathbf{V}) - \tanh(\tilde{\mathbf{A}}\mathbf{Z}\mathbf{W} + \mathbf{X}\mathbf{V})\| \leq \\
&\|\tilde{\mathbf{A}}\mathbf{H}\mathbf{W} + \mathbf{X}\mathbf{V} - \tilde{\mathbf{A}}\mathbf{Z}\mathbf{W} - \mathbf{X}\mathbf{V}\| = \\
&\|\tilde{\mathbf{A}}\mathbf{H}\mathbf{W} - \tilde{\mathbf{A}}\mathbf{Z}\mathbf{W}\| \leq \\
&\|\tilde{\mathbf{A}}\|\|\mathbf{W}\|\,\|\mathbf{H} - \mathbf{Z}\| = \\
&\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|\,\|\mathbf{H} - \mathbf{Z}\|,
\end{aligned} \tag{7}$$

where in the last step we have used the symmetry of the adjacency matrix ($\|\tilde{\mathbf{A}}\| = \rho(\tilde{\mathbf{A}})$). From the last line in the above (7) it is straightforward to observe that $K = \rho(\tilde{\mathbf{A}})\|\mathbf{W}\|$ is a Lipschitz constant of the reservoir system. □

In light of the result of Lemma 1, we observe that whenever the reservoir system satisfies the condition $K = \rho(\tilde{\mathbf{A}})\|\mathbf{W}\| < 1$ the resulting iterated map is a contraction and the system globally converges to its unique fixed point independently on the initial conditions. This observation is in line with the sufficient condition for the GES property in [21]. In this case, we can derive an expression for T, the number of iterations required for convergence.

Theorem 1 (Convergence of reservoir systems on graphs)**.** *Consider a contractive reservoir system on graphs governed by the state transition equation in* (6)*. Given a convergence threshold* ϵ*, the system approaches the fixed point within a number of iterations* T *given by:*

$$T = \left\lceil \frac{\ln \epsilon + \ln(1 - \rho(\tilde{\mathbf{A}})\|\mathbf{W}\|) - \ln H_1}{\ln \rho(\tilde{\mathbf{A}})\|\mathbf{W}\|} \right\rceil, \tag{8}$$

where $H_1 = \|\tanh(\mathbf{X}\mathbf{V})\|$ *is the norm of the reservoir state after the first iteration.*

Proof. Let us denote by $\mathbf{H}_0$, $\mathbf{H}_1$, $\mathbf{H}_t$, and $\mathbf{H}^*$ respectively the initial conditions, the first, the t-th iterate and the fixed point of equation (6). Let K denote a contraction coefficient of the reservoir system. As $\mathbf{H}^*$ is not modified by (6), it is straightforward to see that

$$\|\mathbf{H}^* - \mathbf{H}_t\| \leq K^t\|\mathbf{H}^* - \mathbf{H}_0\|. \tag{9}$$

We also observe that $\|\mathbf{H}^* - \mathbf{H}_0\| \leq \|\mathbf{H}^* - \mathbf{H}_1\| + \|\mathbf{H}_1 - \mathbf{H}_0\|$, from which it follows that $\|\mathbf{H}^* - \mathbf{H}_0\| \leq K\|\mathbf{H}^* - \mathbf{H}_0\| + \|\mathbf{H}_1 - \mathbf{H}_0\|$, hence:

$$\|\mathbf{H}^* - \mathbf{H}_0\| \leq \frac{1}{1 - K}\|\mathbf{H}_1 - \mathbf{H}_0\|. \tag{10}$$

Now, using (10) into (9) we have that:

$$\|\mathbf{H}^* - \mathbf{H}_t\| \leq \frac{K^t}{1-K}\|\mathbf{H}_1 - \mathbf{H}_0\|, \tag{11}$$

which is interesting as it allows us to estimate the distance between the fixed point and any t-th iterate by only knowing the contraction coefficient and the distance between the first iterate and the initial conditions. Successive iterations will then bring the system closer and closer to the fixed point, until the distance becomes smaller than the convergence threshold ϵ. This certainly occurs for $\frac{K^t}{1-K}\|\mathbf{H}_1 - \mathbf{H}_0\| \leq \epsilon$, which leads to $K^t \leq \frac{\epsilon(1-K)}{\|\mathbf{H}_1 - \mathbf{H}_0\|}$, and (observing that $K < 1$) to $t \geq \log_K \frac{\epsilon(1-K)}{\|\mathbf{H}_1 - \mathbf{H}_0\|}$. A number of iterations after which we are sure to be not farther than ϵ from the fixed point is then $T = \left\lceil \log_K \frac{\epsilon(1-K)}{\|\mathbf{H}_1 - \mathbf{H}_0\|} \right\rceil$. Using a zero initial condition $\mathbf{H}_0 = \mathbf{0}$, the Lipschitz constant resulting from Lemma 1, and expanding the logarithm, we finally derive that:

$$T = \left\lceil \frac{\ln \epsilon + \ln(1 - \rho(\tilde{\mathbf{A}})\|\mathbf{W}\|) - \ln H_1}{\ln(\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|)} \right\rceil. \tag{12}$$

□

Theorem 1 tells us that the number of iterations required for convergence depends on several factors. Besides the dependence on the desired convergence tolerance ϵ, some of these factors are related to the RGNN hyper-parametrization ($\mathbf{W}$ and $\mathbf{V}$), others depend on the input graph features ($\mathbf{X}$) and structure ($\tilde{\mathbf{A}}$). Interestingly, keeping fixed the reservoir weight matrices and the input features, the number of iterations T is smaller for smaller values of $\rho(\tilde{\mathbf{A}})$, as expressed by the following Corollary 1.

Corollary 1. *Consider a contractive RGNN layer on graphs governed by the state transition equation* (6)*. Keeping fixed the RGNN weight matrices and the graph input features, the number of iterations required for convergence is an increasing function of* $\rho(\tilde{\mathbf{A}})$*.*

Proof. This result follows straightforwardly from Theorem 1. When taking the derivative of the argument of the ceiling operator on the right hand side of (8) with respect to $\rho(\tilde{\mathbf{A}})$, we get:

$$\begin{aligned} &\tfrac{d}{d\rho(\tilde{\mathbf{A}})}\left(\tfrac{\ln \epsilon + \ln(1-\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|) - \ln H_1}{\ln(\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|)}\right) = \\ &\left(\tfrac{-\ln(\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|)\,\|\mathbf{W}\|}{(1-\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|)} + \tfrac{\ln(H_1/\epsilon) - \ln(1-\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|)}{\rho(\tilde{\mathbf{A}})}\right)\tfrac{1}{(\ln(\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|))^2}. \end{aligned} \tag{13}$$

We recall that $\rho(\tilde{\mathbf{A}})\|\mathbf{W}\| < 1$ is the contraction coefficient of the reservoir system on graphs (see Lemma 1). Therefore, $\ln(\rho(\tilde{\mathbf{A}})\|\mathbf{W}\|) < 0$ and $0 < (1 - \rho(\tilde{\mathbf{A}})\|\mathbf{W}\|) < 1$. Also observe that $\ln(H_1/\epsilon)$ is positive when $H_1 > \epsilon$, which is

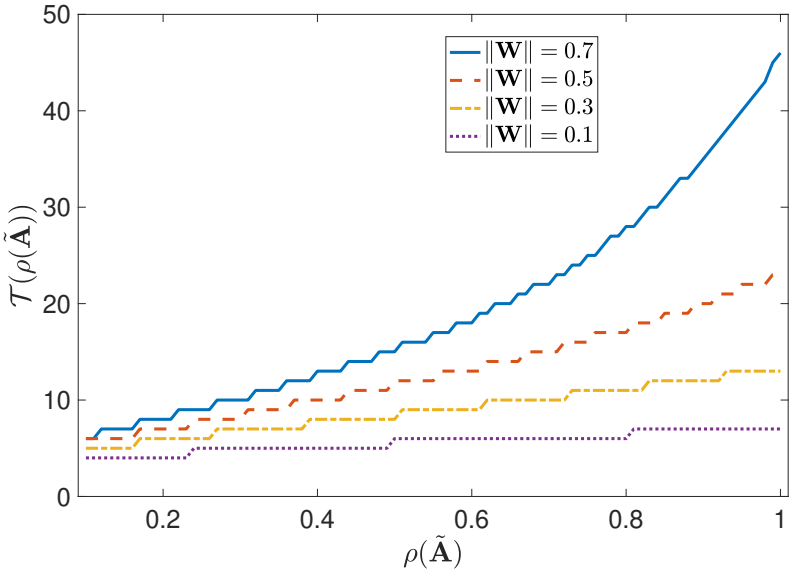

Figure 3: Number of iterations needed to reach the fixed point of contractive reservoir systems on graphs, $\mathcal{T}(\rho(\tilde{\mathbf{A}}))$, for increasing values of the spectral radius of the normalized adjacency matrix, $\rho(\tilde{\mathbf{A}})$. Different lines correspond to values of $\|\mathbf{W}\|$ from 0.1 to 0.7. The shown values are computed from (8), using $\epsilon = 10^{-5}$. The specific value of H_1 does not have a sensible impact on the plot (to emulate a realistic setting, here we used $\ln H_1 = 3.5$).

easily satisfied since ϵ can be made arbitrary small. Thereby, the derivative in (13) is positive. We can then conclude that T is increasing with $\rho(\tilde{\mathbf{A}})$.

□

Hence, we can express T as $\mathcal{T}(\rho(\tilde{\mathbf{A}}))$, an increasing function of $\rho(\tilde{\mathbf{A}})$. A graphical illustration is provided in Figure 3, which shows the number of iterations needed to reach the fixed point (i.e., $\mathcal{T}(\rho(\tilde{\mathbf{A}}))$, vertical axis) for increasing values of the spectral radius of the normalized adjacency matrix (i.e., $\rho(\tilde{\mathbf{A}})$, horizontal axis).

5.2. Computational Complexity of RGNN

We analyze here the computational cost of running an RGNN layer on graphs, recalling that for each input graph the embedding computation requires to iterate (6) until convergence to its fixed point. Noticing that the matrix product $\mathbf{X}\,\mathbf{V}$ in (6) can be performed only once, the cost per each iteration scales as $\mathcal{O}(M\,H + N\,H^2)$, where N and M are respectively the number of vertices and the number of edges in the graph, and H is the number of reservoir neurons in

the layer[5]. The overall cost per layer is $\mathcal{O}(T\,(M\,H + N\,H^2)\,)$, with T the number of iterations required for convergence. As shown in the previous Section 5.1 (see Corollary 1), under the assumption of contractive reservoir dynamics[6], T can be easily expressed as an increasing function of $\rho(\tilde{\mathbf{A}})$, i.e. $\mathcal{T}(\rho(\tilde{\mathbf{A}}))$. Thereby, the computational complexity for graph embedding computation by RGNN can be finally expressed as:

$$\mathcal{O}\left(\mathcal{T}\big(\rho(\tilde{\mathbf{A}})\big)\big(M\,H + N\,H^2\big)\right). \tag{14}$$

From the above (14) we can derive a number of interesting remarks.

Remark 1. *Relevantly for the scopes of this paper, we see that, keeping fixed the reservoir weight matrices, the cost of the embedding computation increases with the number of vertices (N), with the number of edges (M), and with the largest absolute eigenvalue of the normalized adjacency matrix ($\rho(\tilde{\mathbf{A}})$). This allows us to motivate in a mathematically grounded way the use of pooling operators interleaved between successive layers of RGNN. In fact, while trying to preserve as much as possible the information content of the original graph, pooling can act on all 3 indicated factors (N, M, and $\rho(\tilde{\mathbf{A}})$), with a substantial impact on the resulting computation time.*

Remark 2. *The cost of the embedding is the same for both training and test, as the reservoir parameters do not undergo any training process. This also implies that* (14) *gives a lower bound to the cost of the embedding process for the entire class of recursive GNNs (possibly with training).*

Remark 3. *Whenever the set of graphs of interest has a bounded degree - a condition that is typical in many application domains such as chemistry - the cost scales* linearly *with the number of vertices.*

Remark 4. *Deep RC models for time-series [41] significantly reduce the computational complexity compared to shallow architectures with the same total number of recursive hidden units. Such an advantage can also appear in deep RC models for trees [42] or for graphs. On one hand, in a deep architecture the overall cost depends on the number of layers (i.e.,* (14) *applies to each layer). On the other hand, when the number H of recursive hidden units in a shallow architecture is distributed across L layers, the quadratic term NH^2 in* (14) *becomes $LN(H/L)^2 = NH^2/L$. In addition, if L is proportional to H, i.e. $L = aH$, the quadratic term is replaced by $NH^2/(aH) = NH/a$, hence implying a linear instead of quadratic complexity in H.*

[5]The cost can be made linear in H by using sparse recurrent weight matrices $\mathbf{W}$. However, in this paper we focus the analysis on the input graph structure, rather than trying to optimize the reservoir matrix operations.

[6]Although this assumption could be too strong to hold always in practice (like in the case of convetional RC neural networks [17]), the derived insights can be of interest in general to guide the network design.

6. Experiments

In our recent work, we showed the competitive performance of RGNN-based architectures in terms of classification accuracy [21, 43]. By using a sufficiently deep RGNN it is possible to achieve a classification accuracy that is on par or even superior to state-of-the-art neural networks and kernel for graphs [21].

Here, instead, we focus on studying the trade-off between classification accuracy and computing time when a deep RGNN is equipped with graph pooling operations. We perform the experimental evaluation by using an architecture that is simpler compared to that in [21], as we use fewer layers and hidden units, as well as a linear readout. A simpler model allows to better understand the effect of the pooling operations and to relate our results to the theoretical analysis presented in Section 5.

In our experiments, we consider two groups of datasets for graph classification. The first group comprises two new datasets proposed for the first time in this paper and is described in Section 6.1. Then, as a second group, we consider some popular benchmark datasets for graph classification, summarized in Section 6.2. We perform two types of experiments. First, in Section 6.3 we evaluate the performance, in terms of classification accuracy and computing time, achieved by different pooling approaches. Then, in Section 6.4 we perform a qualitative analysis of the graph embeddings by visualizing them with a supervised dimensionality reduction technique.

All the experiments were performed on a server with an Intel i7-10700K processor, two Nvidia RTX-2080Ti, and 64GB of RAM. A library that allows to implement the proposed Pyramidal RGNN is publicly available online[7].

6.1. New benchmark datasets for graph classification

An important shortcoming in several benchmark dataset for graph classification is that in some cases the vertex features contain enough information to achieve high classification accuracy, even without considering the graph structure. Similarly, in other datasets the vertex features are trivial or they are not present at all. This problem has been highlighted by recent works, which showed that very simple baselines that disregard either the graph connectivity or the vertex features are able to achieve similar performance to elaborated state-of-the-art models [44, 45, 30].

Motivated by these reasons, we propose a new class of datasets where both the vertex features and the adjacency matrix are completely uninformative if considered alone. Therefore, an algorithm that relies only on the vertex features or on the graph structure will fail to perform better than a random classifier. The proposed datasets consist of graphs belonging to 3 different classes. Each graph is constructed by first generating 5 random clusters of points, each one with a particular shape (Gaussian blob, two moons, and concentric circles). The points in each cluster are assigned with a different color, encoded by a one-hot

[7] `https://github.com/FilippoMB/Pyramidal-Reservoir-Graph-Nerual-Networks`

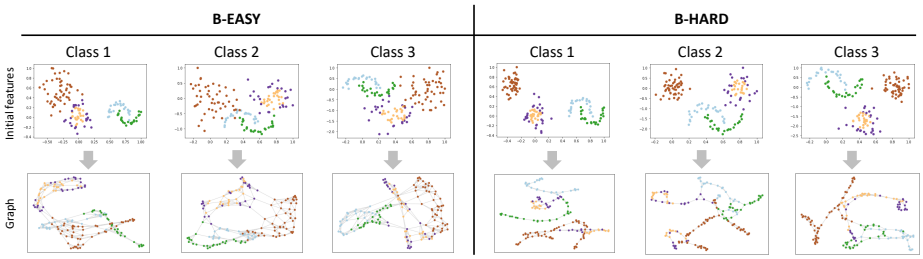

Figure 4: The figure depicts an example of how the graphs of the 3 different classes are generated in the two benchmark datasets for graph classification. Each class is determined by the relative positions of the clusters of 2-D points. For example, in class 1 there is a Gaussian blob on the left, concentric circles in the middle, and two-moons on the right. In the *easy* version the clusters are more scattered and overlapping, while in the *hard* version are more compact. A k-NN graph with $k = 5$ for the version *easy* and $k = 4$ for *hard* is used to generate a graph from the original data points.

vector in $\mathbb{R}^5$. Such vectors are the input features associated with each vertex and represent the rows of the vertex feature matrix $\mathbf{X} \in \mathbb{R}^{N \times 5}$. The graph is built by connecting the points with a k-nn graph and the vertex features are the $\mathbb{R}^5$ one-hot color vectors. The three classes of graph are determined by the relative position of the original clusters. An example is depicted in Figure 4, where the first row shows the original clusters and the second row the resulting graphs, obtained by connecting each point with its closest k neighbors.

Based on this approach we introduce two dataset versions: *easy* and *hard*. The graphs in the first version are easier to classify than those in the second one. The difficulty is controlled by the number of edges in the k-nn graph ($k = 5$ in *easy* and $k = 4$ in *hard*) and the compactness of the original clusters. In particular, the clusters in *easy* have higher variance, while the clusters in *hard* are more compact and isolated. In the latter case, it is more difficult to understand the relative position of the clusters in the graph. Indeed, due to fewer inter-cluster connections, it is necessary to explore larger and more heterogeneous neighborhoods to determine the correct class of the graph. Additionally, in the *hard* variant some graphs are disconnected.

The two versions of the dataset are available online[8] and they are already split in train, validation, and test set. The online repository contains also a small version of the two datasets (*easy-small* and *hard-small*), which can be useful during debug and preliminary testing stages.

6.2. TUD datasets

Next, we consider the popular TUD datasets for graph classification [46]. We selected a rich variety of datasets, where graphs are either representing molecules (Mutagenicity, Proteins, Enzymes, NCI1, D&D) or social networks (IMDB-BINARY, IMDB-MULTI, COLLAB, Reddit-2K, Reddit-5K).

[8] `https://github.com/FilippoMB/Benchmark_dataset_for_graph_classification`

The datasets are very different from each other, in terms of number of graph classes, number of vertices and edges in each graph, and type of vertex attributes and vertex labels. Therefore, the selected datasets allow to test the performance of the graph classification algorithms in a large variety of use-cases.

Table 1 reports the statistics of each dataset used in our study. In each dataset, the vertices can be associated both with a "vertex attribute" (a real-valued vector) and a "vertex label" (a discrete value represented by a one-hot vector). Vertex labels should not be confused with the graph label, which defines the class of the entire graph. The node attribute and the node label are concatenated to obtain the vertex feature vector $\mathbf{x}$. We note that in some datasets there are neither vertex attributes nor vertex labels: in those cases we use the node degree as surrogate vertex feature.

Table 1: Summary of statistics of the graph classification datasets.

Dataset	**samples**	**classes**	**avg. vertices**	**avg. edges**	**vertex attr.**	**vertex labels**
B-easy	1,800	3	147.82	922.66	–	yes
B-hard	1,800	3	148.32	572.32	–	yes
Mutagenicity	4,337	2	30.32	91.87	–	yes
Proteins	1,113	2	39.06	72.82	1	no
Enzymes	600	6	32.63	62.14	18	yes
NCI1	4,110	2	29.87	32.30	–	yes
D&D	1,178	2	284.32	715.66	–	yes
IMDB-BIN	1,000	2	19.77	96.53	–	no
IMDB-MUL	1,500	3	13.00	65.94	–	no
COLLAB	5,000	3	74.49	2,457.78	–	no
Reddit-2K	2,000	2	429.63	497.75	–	no
Reddit-5K	4,999	5	508.52	594.87	–	no

6.3. Graph classification

We evaluate the performance in terms of classification accuracy and computational time of the 12 datasets described in Section 6.1 and 6.2. To generate the graph embeddings, we consider two types of architectures. The first is the Pyramidal RGNN presented in Section 4, which is equipped with one of the three pooling operators described in Section 3: *Graclus*, *NMF* and *NDP*. The second architecture is identical to the previous one but without pooling layers and is referred to as "*No-pool*". In our experimental analysis, we use the *No-pool* model as a baseline reference for comparing the results obtained by the Pyramidal architectures, as this allows us to assess directly the effect of introducing the pooling operations. For both variants, we considered an architecture both with $L = 2$ and with $L = 3$ blocks, for a total of 8 architectures. We also notice that the *No-pool* architecture is similar to the one presented in our previous work on FDGNN [21], to which we refer the interested reader for a comparison, in terms of classification accuracy, with other state-of-the-art models.

To perform the classification, once the graph embeddings are computed they are processed by a linear ridge regression classifier that is controlled through a

Tikhonov regularization hyper-parameter α. The model selection is performed by following the same cross-validation scheme described in [45], with the following setting:

1. 5 external folds (test split);
2. internal hold out (90% train, 10% validation);
3. 100 different random hyper-parameters configurations obtained by uniformly and randomly drawing $\rho \in [.1, .9]$, $\omega_{in} \in [.1, .8]$, $\omega_{hid} \in [.1, .8]$;
4. For each hyper-parameter configuration, we perform a grid search on the regularization hyper-parameter $\alpha \in \{1e2, 1e1, 1e0, 1e-1, 1e-2\}$;
5. For each configuration $\{\rho, \omega_{in}, \omega_{hid}, \alpha\}$ we generate 3 random initialization of the reservoir weights and compute the average accuracy on the validation set;
6. The model defined by the optimal configuration $\{\rho^*, \omega^*_{in}, \omega^*_{hid}, \alpha^*\}$, i.e., the one that achieves the highest average accuracy on the validation set, is tested on the external fold (test set);
7. The average accuracy obtained on all the 5 (test) folds is reported as the model performance.

The number of latent features, i.e. the reservoir dimension, is kept fixed to $H = 50$ across all layers. The weight matrices in each reservoir layer are initialized as described in Section 2. We set the convergence threshold $\epsilon = 1e-5$ and we set the maximum number of iterations to $max_{iter} = 50$, meaning that (4) is iterated until the norm of the difference between two consecutive updates is less than $1e-5$, or 50 iterations are done. The value $max_{iter} = 50$ is in line with the theoretical analysis in Section 5.1 and with the simulations reported in Figure 3. We also notice that, in practice, the convergence is almost always reached before surpassing the maximum number of allowed iterations.

The pooling methods do not depend on hyper-parameters, with two exceptions:

- In *NMF* pooling, it is possible to specify the number K of super-nodes in the coarsened graph. For each graph $\mathcal{G}$, we let $K_{\mathcal{G}} = N_{\mathcal{G}}/2$, so that the number of vertices in a graph are halved each time. This is consistent with the two other pooling strategies, which also reduce the number of nodes by a factor of 2.

- In Section 3.3 we explained that a hyper-parameter δ controls the sparsification procedure in the *NDP* pooling strategy. We set $\delta = 0.1$, meaning that all the edges with weight larger than 0.1 are dropped. While this is a quite aggressive pruning, it allows to trade some accuracy in the graph representation with the computational efficiency of working with a graph that is sparse.

In Table 2 we report the mean and the standard deviation of the classification accuracy obtained on the 5 external folds, obtained by the Pyramidal RGNN architecture with 2 blocks: [`RGNN`, `Pool`]-[`RGNN`], where `Pool` denotes one of the

Table 2: Results obtained for the Pyramidal architectures [RGNN, Pool]-[RGNN] and for the *No-pool* architecture [RGNN]-[RGNN]. For each model, we report: i) the mean and the standard deviation of the classification accuracy obtained on the 5 external folds; ii) average training time (t_{tr}) and test time (t_{tr}), in seconds. We denote with "Mem Err" the experiments that failed due to memory errors.

	No-pool			**Graclus**			**NMF**			**NDP**		
	Acc.	t_{tr}	t_{ts}	Acc.	t_{tr}	t_{ts}	Acc.	t_{tr}	t_{ts}	Acc.	t_{tr}	t_{ts}
B-easy	96.9±0.6	8.9	2.3	93.4±1.4	6.5	1.6	87.6±1.2	6.1	1.5	91.5±0.5	5.3	1.3
B-hard	75.3±2.0	8.5	2.1	68.7±2.9	5.6	1.4	62.1±1.9	4.7	0.6	67.9±2.7	4.4	0.3
Mutagenicity	76.1±0.9	5.1	1.3	74.4±1.8	5.3	1.3	69.2±0.7	3.8	0.9	69.1±0.5	5.7	1.4
Proteins	70.9±1.9	1.3	0.3	70.7±0.2	1.1	0.3	70.1±1.8	1.2	0.3	71.8±1.7	1.3	0.3
Enzymes	45.5±2.9	0.6	0.1	37.0±0.3	0.5	0.1	33.5±2.4	0.4	0.1	36.7±4.5	0.4	0.1
NCI1	70.4±0.8	5.0	1.3	68.1±0.8	5.3	1.3	66.9±2.1	4.2	1.0	67.4±1.2	4.5	1.1
D&D	76.4±1.7	8.4	2.2	72.5±0.9	8.2	2.0	71.4±1.7	9.4	2.3	71.2±3.0	8.2	2.0
IMDB-BIN	71.3±1.9	0.7	0.2	70.7±5.1	0.9	0.2	63.2±2.3	0.8	0.1	69.7±1.9	0.9	0.2
IMDB-MUL	49.5±2.3	0.9	0.2	48.5±1.9	1.3	0.3	46.1±1.7	0.9	0.2	47.3±2.7	1.1	0.3
COLLAB	74.3±1.2	13.1	3.3	72.7±1.5	15.2	3.8	54.8±0.9	12.2	3.1	71.2±0.6	12.0	3.0
Reddit-2K	84.7±0.9	31.5	8.0	81.6±0.9	41.7	10.6	52.2±4.1	37.5	9.5	80.5±2.0	13.0	3.2
Reddit-5K	53.6±1.1	98.3	24.6	Mem Err			35.3±1.4	98.5	24.7	49.3±1.3	42.5	10.6

three pooling strategies: *Graclus*, *NMF*, or *NDP*. For comparison, the same table also contains the results achieved by an architecture without pooling layers, *No-pool*: [RGNN]-[RGNN]. For each model, we also report the mean training and test time (in seconds) obtained on the 5 folds. We indicate with "Mem Err" the experiments that failed due to a memory error. Those errors occurred when the adjacency matrices of a certain dataset were very large or contained too many edges and it was not possible to store them in the 64GB of memory of the server used to perform the experiments.

Despite some variability, there are some patterns that emerge from the results in Table 2. As expected, *No-pool* generally achieves the highest classification accuracy, with the only exception of Proteins on which the best accuracy is achieved in correspondence of *NDP*. Among the Pyramidal RGNNN variants, the one using *Graclus* generally leads to higher accuracy, and *NDP* is runner up. Regarding the computation times, on the other hand, Pyramidal RGNNs variants show a significant advantage, allowing calculation times to be reduced by up to almost 60% compared to the No-pool model. The achievable speedup is particularly evident, e.g., in the case of the large Reddit datasets using *NDP* pooling. Comparing the 3 Pyramidal RGNN variants, we notice that *NDP* generally leads to the smallest times, and *NMF* is runner up. *Graclus*, on the other hand is almost always giving the worst results in terms of computational times, in some cases being even slower than *No-pool*.

To summarize, *No-pool* achieves a better classification accuracy, but the Pyramidal approach offers significant advantages in terms of computational complexity. To evaluate with a single metric the trade-off between accuracy

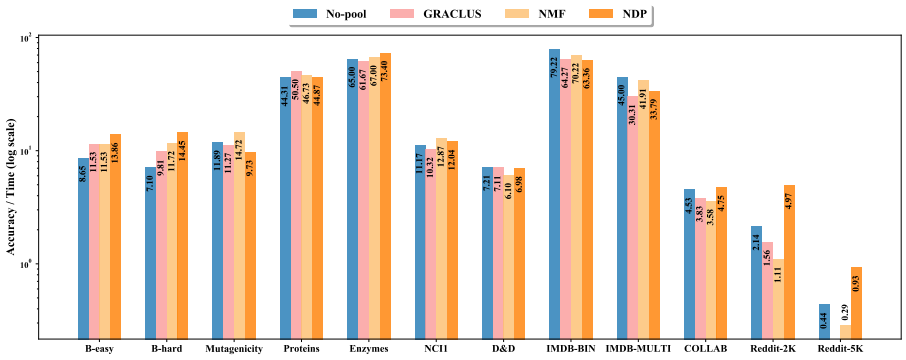

Figure 5: The performance measure $\frac{\text{classification accuracy}}{\text{train time + test time}}$ achieved by the Pyramidal architectures [RGNN, Pool]-[RGNN], compared to the results of the *No-pool* architecture [RGNN]-[RGNN] on the 12 classification datasets.

and complexity, we consider the ratio $\frac{\text{classification accuracy}}{\text{train time + test time}}$, which should be as high as possible. Intuitively, this ratio tells us how many points of accuracy it is possible to achieve per second of computation. In Figure 5 we visualize in a bar plot the values of the ratio for the 4 architectures on the 12 classification datasets. It is evident from the figure that the Pyramidal approach is preferable to *No-pool* (with just a couple of exceptions for the IMDB datasets). In particular, Pyramidal RGNN with *NDP* pooling shows the best overall accuracy/complexity trade-off, with outstanding results in both the Reddit datasets.

We continue the analysis of the classification performance by repeating the same classification experiment with an architecture composed by $L = 3$ blocks, rather than $L = 2$ blocks as in the previous case. The results obtained by the Pyramidal architectures [RGNN, Pool]-[RGNN, Pool]-[RGNN] and by the *No-pool* model [RGNN]-[RGNN]-[RGNN] are reported in Table 3.

By comparing these new results with the previous ones for $L = 2$ (in Table 2), we notice that the *No-pool* baseline consistently achieves a slightly higher classification accuracy at the expense of a higher training and test time. On the other hand, the classification accuracy of the Pyramidal architectures achieve a slightly worse classification accuracy when $L = 3$, probably because too much information is discarded by two graph pooling operations. In the architectures with *Graclus* and *NMF* pooling this drop in performance is not compensated by a speed-up in computing time. For example, the Pyramidal architecture with $L = 3$ and *Graclus* achieves on Reddit-2K a worse classification accuracy and the train/test time is higher, compared to the architecture with $L = 2$. The situation is, however, different for the Pyramidal architecture with *NDP* pooling, which achieves a significant improvement in computing time on Reddit-2K and Reddit-5K when $L = 3$. Once again, we highlight that on these two datasets the improvements in computing speed compared to the baseline are remarkable: *NDP* is more than three time faster than *No-pool*, as it can be trained in 37.3

Table 3: Results obtained for the Pyramidal architecture [RGNN, Pool]-[RGNN, Pool]-[RGNN] and for the *No-pool* architecture [RGNN]-[RGNN]-[RGNN]. For each model, we report: i) the mean and the standard deviation of the classification accuracy obtained on the 5 external folds; ii) average training time (t_{tr}) and test time (t_{tr}), in seconds. We denote with "Mem Err" the experiments that failed due to memory errors.

	No-pool			**Graclus**			**NMF**			**NDP**		
	Acc.	t_{tr}	t_{ts}	Acc.	t_{tr}	t_{ts}	Acc.	t_{tr}	t_{ts}	Acc.	t_{tr}	t_{ts}
B-easy	97.6±0.8	10.1	2.7	85.0±1.8	7.6	1.9	76.0±1.2	6.3	1.6	83.6±0.8	5.5	1.4
B-hard	76.4±1.5	9.8	2.6	54.4±1.6	5.9	1.5	49.6±1.5	6.3	1.6	57.4±1.8	4.8	1.2
Mutagenicity	76.6±1.1	7.7	1.9	70.5±0.8	8.4	2.1	61.5±0.6	5.5	1.4	66.5±1.0	5.9	1.5
Proteins	71.1±2.6	2.4	0.6	69.6±2.3	1.7	0.4	62.2±1.6	1.8	0.5	68.7±1.9	1.6	0.4
Enzymes	47.2±4.5	0.8	0.2	30.3±2.7	0.7	0.2	27.8±2.7	0.6	0.2	29.6±2.5	0.6	0.1
NCI1	70.5±1.9	6.4	1.6	65.0±1.3	6.9	1.7	61.2±1.2	4.1	1.0	63.4±1.4	4.3	1.1
D&D	73.7±1.2	13.5	3.4	71.2±3.8	9.8	2.4	63.0±3.3	11.1	2.8	69.1±2.7	9.6	2.3
IMDB-BIN	72.3±1.4	1.3	0.3	70.8±2.3	1.3	0.3	57.2±2.0	0.8	0.2	68.3±1.4	1.2	0.3
IMDB-MUL	49.7±0.8	1.4	0.4	47.5±2.9	1.2	0.3	42.0±1.6	1.1	0.3	47.0±2.5	1.2	0.3
COLLAB	74.4±0.1	20.6	5.2	72.2±1.4	19.0	4.7	Mem Err			69.2±0.5	13.9	3.4
Reddit-2K	85.1±0.8	34.2	8.6	76.7±1.8	65.0	16.3	Mem Err			74.7±1.0	10.7	2.7
Reddit-5K	53.5±0.8	115.6	28.8	Mem Err			Mem Err			48.9±0.8	37.3	9.3

seconds (on average) on Reddit-5K, rather than 115.6 seconds (on average). Remarkably, those are the largest dataset and the improvement in the computing time here is crucial.

To effectively compare the differences in performance between the architectures with $L = 2$ and $L = 3$ blocks, we consider again the ratio between classification accuracy and the combination of train and test time. In Figure 6 we report in colors the values of the ratio for $L = 3$ and in gray the values of the ratio for $L = 2$ (the same that were previously reported in Figure 6). Since the higher the ratio the better, it is immediately possible to notice that most architectures achieve better performance when $L = 2$, confirming the observations made regarding Table 3. From Figure 6 we can also notice that the Pyramidal architecture with *NDP* and $L = 3$ achieves a higher ratio on Reddit-2K and Reddit-5K.

Two important considerations should be made about the classification performance.

- In the *No-pool* variant, the final vectorial representation of the graph can preserve more information as it accounts for all the vertices in the original graph. It has been shown that by summing all the vertex features generated by message-passing operations, such as those implemented by the RGNN layers, it is possible to obtain powerful representations that discriminate well between different graphs [47]. Indeed, the final readout layer manages to separate such informative representations well. On the other hand, local pooling operations drastically reduce the amount of information each time.

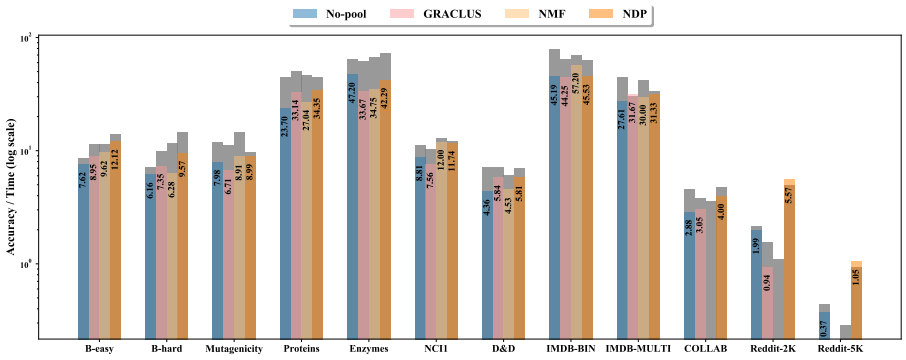

Figure 6: The performance measure $\frac{\text{classification accuracy}}{\text{train time} + \text{test time}}$ achieved by the Pyramidal and by the *No-pool* architectures with $L = 3$ blocks. For comparison, we depict in gray the results reported in Figure 5, i.e., those achieved by the architectures with $L = 2$ blocks.

- The components of our pyramidal architecture (generation of new vertex representations with RGNN layers and local pooling) are optimized without any form of supervision and in a greedy fashion. Indeed, there is not a training procedure that allows learning vertex features in function of the endowing vertex pooling. Similarly, the vertex pooling is based on graph coarsening procedures that rely on predefined graph-theoretical principles. This would not be the case in fully trainable GNNs, where the message-passing and pooling layers jointly adapt their behavior by accounting for their interactions.

Despite these challenges, the proposed pyramidal architecture maintains a good classification performance, while significantly lowering the computation time. This indicates that the unsupervised pooling schemes are effective even if they optimize generic graph theoretical properties rather than task-specific objectives. In addition, our results highlights the representational power of the RGNN, which manages to generate rich vertex representations that are also robust to perturbations on the local structure induced by pooling.

To complete our analysis, we link the achieved numerical results to the outcomes of the theoretical investigation reported in Section 5. In particular, in Remark 1 we have highlighted that the cost of graph embeddings computation in RGNN is an increasing function of the number of vertices (N), of the number of edges (M) and of the largest absolute eigenvalue of the normalized adjacency matrix ($\rho(\tilde{\mathbf{A}})$). In Table 4, we report these statistics for all the used datasets.

We start by observing that all the pooling algorithms effectively reduce the number of vertices N compared to the original graphs. Since the embedding cost in (14) depends on N, this is a first explanation of the consistent reduction in the required computation time observed for Pyramidal RGNN variants in comparison to *No-pool*. Among the pooling alternatives, we notice that *NMF* and *NDP* generally lead to a similar reduction in N, with the substantial difference that *NDP* is easier to compute even on larger datasets (where memory errors

Table 4: Statistics of the pooled graphs. For each dataset, we report the average number of vertices (N), the average number of edges (M), and the average largest absolute eigenvalue of the normalized adjacency matrices ($\rho(\tilde{\mathbf{A}})$). We detail the results corresponding to the original dataset, and after one (l1) ore two (l2) levels of the considered pooling algorithms. Missing values correspond to the memory errors in Table 2 and 3.

	B-easy			**B-hard**			**Mutagenicity**		
	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$
Original	147.8	922.6	1.00	148.3	572.3	1.00	30.3	91.8	1.00
Graclus l1	91.9	476.8	1.00	100.5	316.7	1.00	25.6	56.9	1.00
Graclus l2	45.5	209.3	1.00	50.1	154.3	1.00	12.8	33.7	1.00
NMF l1	73.0	3482.1	1.00	73.2	3110.7	0.99	14.9	223.4	1.00
NMF l2	36.3	1388.9	1.00	37.0	1390.4	1.00	7.2	77.5	1.00
NDP l1	73.0	7014.0	0.98	73.2	354.8	0.76	14.6	49.6	0.98
NDP l2	36.1	359.0	0.97	37.5	170.1	0.66	6.7	26.7	0.92
	Proteins			**Enzymes**			**NCI1**		
	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$
Original	39.1	72.8	1.00	32.6	62.1	1.00	29.9	62.1	1.00
Graclus l1	25.9	95.0	1.00	22.4	79.2	1.00	21.0	56.1	1.00
Graclus l2	12.9	47.2	1.00	11.2	37.1	1.00	11.3	30.1	1.00
NMF l1	19.3	580.5	1.00	16.1	238.4	0.99	15.1	193.2	1.00
NMF l2	9.5	220.1	1.00	7.8	75.8	1.00	7.1	62.9	1.00
NDP l1	19.2	106.0	0.98	16.2	83.3	0.98	15.2	44.8	0.98
NDP l2	9.5	56.4	0.89	8.1	38.9	0.97	7.4	23.1	0.96
	D&D			**IMDB-BINARY**			**IMDB-MULTI**		
	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$
Original	284.3	715.6	1.00	19.7	96.5	1.00	13.0	65.9	1.00
Graclus l1	175.1	864.4	1.00	12.4	68.3	1.00	8.3	44.2	1.00
Graclus l2	87.0	424.5	1.00	6.2	23.2	1.00	4.1	13.8	1.00
NMF l1	141.3	24996.2	1.00	9.7	118.4	1.00	6.2	57.2	1.00
NMF l2	70.7	9625.0	1.00	4.6	27.3	1.00	3.0	13.4	1.00
NDP l1	141.2	1364.7	0.98	9.7	86.7	0.99	6.3	46.8	0.96
NDP l2	70.2	902.2	0.99	4.5	23.8	0.85	3.2	12.0	0.73
	COLLAB			**Reddit-2K**			**Reddit-5K**		
	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$	N	M	$\rho(\tilde{\mathbf{A}})$
Original	74.5	2457.8	1.00	429.6	497.8	1.00	508.5	594.9	1.00
Graclus l1	40.5	1589.3	1.00	1050.5	1157.6	0.99	-	-	-
Graclus l2	20.3	457.4	1.00	525.0	998.9	0.99	-	-	-
NMF l1	37.0	2338.2	1.00	214.5	90540.9	0.99	254.0	73675.5	1.00
NMF l2	-	-	-	-	-	-	-	-	-
NDP l1	37.1	2284.1	0.89	142.1	1560.1	0.55	211.4	2268.4	0.28
NDP l2	18.3	620.7	0.87	67.4	1091.4	0.69	103.2	1769.8	0.61

are obtained in the other cases), and is then preferable. Moreover, commenting the numerical results we noticed that in some cases *Graclus* resulted to be slower than *No-pool*. In Tables 2 and 3 this is especially apparent for Reddit-2K. Interestingly, we can see from Table 4 that running *Graclus* pooling on Reddit-2K tends to actually increase the number of vertices rather than decreasing it. As discussed in Section 3.1, the reason is that *Graclus* needs to create a balanced

binary tree by adding fake vertices before performing the pooling operation. In this way, the total number of vertices in the pooled graphs might increase in the intermediate layers and, in some cases, such an increment is significant, as observed in Table 4. Even if the fake vertices are disconnected and do not affect the representations of the other vertices generated by the RGNN, their presence impact the computation time, as observed in Tables 2 and 3.

The models with *Graclus* pooling generally achieve the highest classification accuracy among the Pyramidal RGNN models, but they are usually slower than the models equipped with *NMF* and *NDP* pooling, for the reason discussed above. *NDP* outperforms *NMF* in terms of classification accuracy and usually achieves the lowest computational time among all methods. This is particularly evident (Table 2) in the largest datasets such as COLLAB, Reddit-2K and Reddit-5K, where *NDP* is up to three times faster than the second fastest method. By looking at Table 4, we notice that the number of vertices (N) in the pooled graphs generated by *NDP* and *NMF* is similar but *NDP* generates sparser graphs, i.e., with less edges and hence a smaller M. Since the embedding cost in (14) depends on M, this represents another computational advantage of Pyramidal RGNN with *NDP* pooling.

By looking the computational times in Tables 2 and 3, it seems that the average number of vertices N and the average number of edges M do not completely explain the difference in the computational performance between the three pooling methods. For example, on the COLLAB dataset, the *Graclus* and *NDP* methods result in a comparable reduction on N, while *NDP* leads to larger values of M. Still, as shown in Table 2 and especially Table 3, *NDP* has lower computation times. In Section 5, we discussed that the computational time of the embedding process is heavily affected by the convergence speed of RGNN layers, which, in turn, depends on the largest absolute eigenvalue $\rho(\tilde{\mathbf{A}})$ (see Theorem 1 and Corollary 1).

The columns of Table 4 with header $\rho(\tilde{\mathbf{A}})$ report the average maximum eigenvalue (in absolute value) of the normalized adjacency matrices obtained with the different pooling methods. We recall that the eigenvalues of a symmetrically normalized adjacency matrix $\tilde{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ are always contained in $[-1, 1]$. We can see in Table 4 that the coarsened graphs generated by *NDP* have an average $\rho(\tilde{\mathbf{A}})$ that is significantly smaller than what obtained in the other cases, especially for the Reddit-2K and Reddit-5K datasets. A smaller value of $\rho(\tilde{\mathbf{A}})$ allows the iterative update in (4) to converge faster, thus reducing the overall computational time, as shown in Section 5.1 (see Figure 3). The spectral properties of the coarsened graphs generated by *NDP* derives from the Kron reduction procedure used to generate the new edges, which is described in detail in [23].

Overall, we see that *NDP* has intriguing features that generally allow us to impact all three factors that we identified as crucial to the computational cost of the graph embedding process in our theoretical analysis in Section 5. Eventually, this leads to the advantageous performance trade-off highlighted in the experiments.

6.4. Embedding visualization

In this section, we apply a dimensionality reduction technique for visualizing the structure of the graph embeddings computed by the architectures under consideration. The purpose of this analysis is to investigate if there are qualitative differences between the embeddings. To perform dimensionality reduction, we adopt Linear Discriminant Analysis (LDA) [48], which computes low-dimensional projections by searching for linear combinations of the variables that best explain the data and the class labels. LDA is a linear method and, therefore, the more the low-dimensional projections generated by LDA appear to be well separated, the better a linear classifier (like the readout used in the Pyramidal RGNN) can correctly predict the class of the graph embeddings.

The number of dimensions where LDA can project the data must be lower than the number of classes. Therefore, a 2-dimensional visualization is possible only for datasets with 3 or more classes. We selected B-Easy, B-Hard, Enzymes, and IMDB-MULTI datasets since are datasets with more than 2 classes.

Since we are performing a qualitative analysis, we do not perform model selection here, but we use the same hyper-parameters for every architecture ($\rho = 0.9$, $\omega_{in} = 0.5$, $\omega_{hid} = 0.8$, $H = 100$, $L = 3$). In addition, we do not generate training, validation, and test splits but we rather fit the LDA model to the whole dataset and depict the two-dimensional representations of the fitted data. Indeed, since the datasets are relatively small, depicting the embeddings only for a test set would make it difficult to appreciate the qualitative differences between the methods under consideration.

Figure 7 shows the two-dimensional representations of the graph embeddings generated with the different architectures. The first row shows the embeddings of the graphs in the B-Easy dataset. As it can be seen in the figure, *No-pool* manages to perfectly separate the three classes, while in the pyramidal architectures we observe some overlaps between the classes depicted in brown and in blue. The graph embeddings in the second rows, representing the graphs in the B-hard dataset, are more overlapping, suggesting that it is more difficult to classify correctly some of the graphs associated with the embeddings depicted in the middle. The embeddings generated by *No-pool* appear slightly more separated than those generated by the pyramidal architectures. In the third row, we observe that some of the embeddings of the Enzymes dataset overlap considerably with each other. The green class is the one that can be classified more accurately in every case. Interestingly, some classes are separated better by different methods. For example, *No-pool* separates well the pink class, *Graclus* separates well the purple, and *NDP* the orange class. The last row shows the embeddings associated to the IMDB-MULTI dataset. In this case, there is a larger difference between the methods under consideration. The embeddings of *No-pool* form some tight clusters, while the pyramidal methods form clusters that are less compact. The embeddings generated by *Graclus* appear more separated, while those generated by *NMF* have a strong overlap.

The visualization results are well aligned with the numerical results reported in Table 3: *No-pool* yields graph embedding that can be separated slightly better

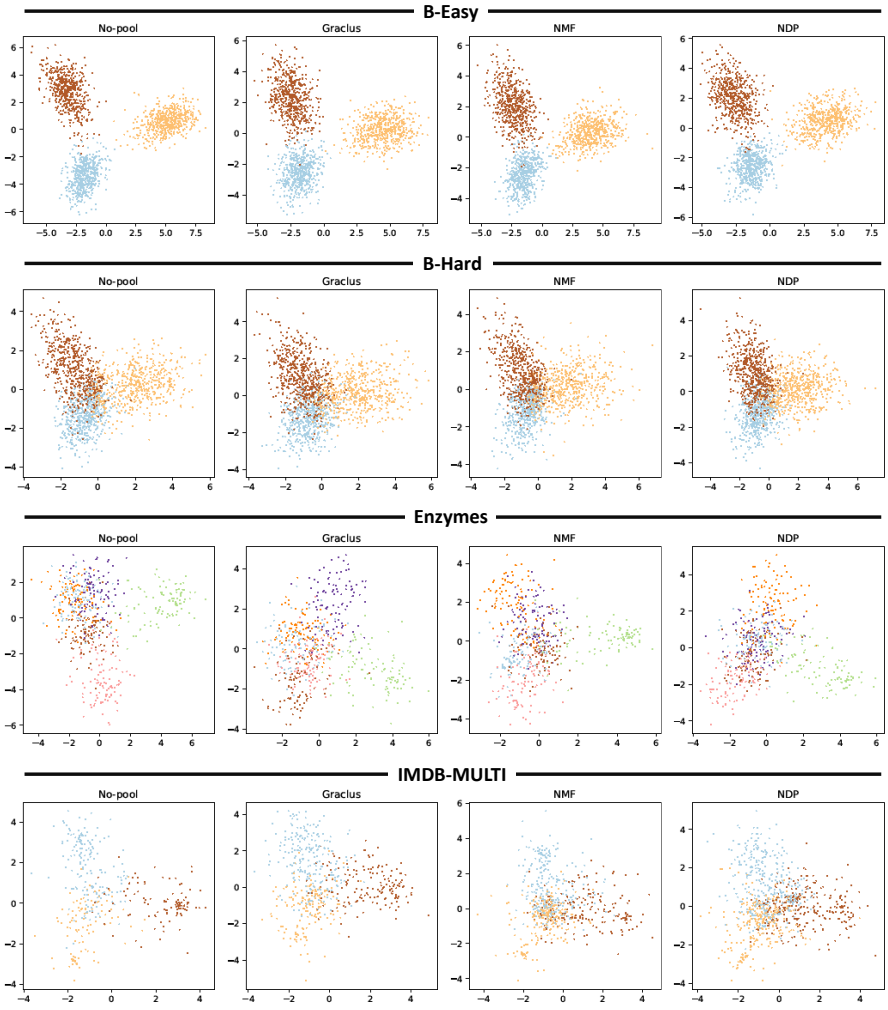

Figure 7: Visualization of the graph embeddings generated by the different architectures. The dimensionality reduction is performed with LDA.

by a linear classifier than those generated by the pyramidal variants. Remarkably, apart from the occasional distinction mentioned above, results in Figure 7 indicate that there are no strong qualitative differences between the graph embeddings generated by the different pyramidal architectures. Hence, among the graph pooling methods, *NDP* is preferable due to its lowest computational times, as analyzed in Section 6.3.

7. Conclusions

In this paper, we introduced a novel Reservoir Computing model for graphs endowed with graph pooling operators. In particular, we built a deep architecture with a pyramidal structure, where Reservoir GNN layers that compute embeddings for the graph vertices are interleaved by graph pooling layers that generate coarsened representations of the underlying graph. The vertex features in the last layer of the architecture are combined to generate a graph embedding vector, which can be used to perform tasks such as classification or dimensionality reduction.

The main advantage of the proposed pyramidal architecture is the reduction of the computing time required to generate the graph embeddings. Therefore, our analysis focused on investigating the trade-off between performance in classification accuracy and computational complexity. To this end, we introduced a formal mathematical analysis of the convergence of stable recursive graph embeddings and derived an upper-bound to its computational cost. Remarkably, we found that this bound depends on graph-theoretical properties of the input data, and in particular on the number of vertices, on the number of edges (hence on the density), and on the spectrum of the graph. By acting on these properties, the graph pooling algorithms can have a decisive impact on the computation times. Moreover, the derived bound allowed us to understand the differences in the computing time required by the different architectures analyzed. The theoretical analysis presented in this paper is general and can be extended to other Reservoir Computing and recursive architectures for graphs.

Our experimental evaluation focused on two different types of tasks. First, we performed graph classification and compared the classification accuracy and computing time of the proposed pyramidal architecture, equipped with different types of graph pooling operators, and of the same architecture without pooling layers. Then, we considered a dimensionality reduction task, which allowed us to perform a qualitative analysis by visualizing the embeddings generated by the different models.

The experiments were performed on a large collection of datasets for graph classification. Two of the datasets used in the experiments are introduced in this paper, to provide a solid benchmark to test the robustness of generic graph kernels or graph neural networks for classification.

Our results show that graph pooling allows to significantly accelerate the computing time at the price of partially reducing the quality of the graph embeddings. We showed such a trade-off both from a quantitative and a qualitative perspective, i.e., in terms of classification accuracy and visualization of

the graph embeddings. Among the pooling methods, we found that Node Decimation Pooling offers the best trade-off between computational complexity and quality of the graph embeddings.

In the perspective of studying the synergy between Graclus and Node Decimation Pooling, which offer different performance in terms of classification accuracy and computational complexity, it would be interesting to combine the two operations within a hybrid model. However, since the coarsening strategy of the two procedures are orthogonal to each other, this could introduce a form of perturbation that is detrimental in Reservoir Computing models. On the other hand, mixing different pooling procedures could provide a regularization that is useful in fully trainable GNN models and is similar to a strategy proposed in traditional CNN architectures [49].

Acknowledgements

This work was partially funded by the Visiting Fellows Programme of University of Pisa.

References

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P. S. Yu, A comprehensive survey on graph neural networks, IEEE Transactions on Neural Networks and Learning Systems (2020) 1–21.

[2] D. Bacciu, F. Errica, A. Micheli, M. Podda, A gentle introduction to deep learning for graphs, Neural Networks 129 (2020) 203 – 221.

[3] A. Sperduti, A. Starita, Supervised neural networks for the classification of structures, IEEE Transactions on Neural Networks 8 (3) (1997) 714–735.

[4] P. Frasconi, M. Gori, A. Sperduti, A general framework for adaptive processing of data structures, IEEE Transactions on Neural Networks 9 (5) (1998) 768–786.

[5] B. Hammer, A. Micheli, A. Sperduti, M. Strickert, Recursive self-organizing network models, Neural Networks 17 (8-9) (2004) 1061–1085, publisher: Elsevier.

[6] A. Micheli, D. Sona, A. Sperduti, Contextual processing of structured data by recursive cascade correlation, IEEE Transactions on Neural Networks 15 (6) (2004) 1396–1410.

[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Transactions on Neural Networks 20 (1) (2009) 61–80.

[8] A. Micheli, Neural network for graphs: A contextual constructive approach, IEEE Transactions on Neural Networks 20 (3) (2009) 498–511.

[9] M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering, in: Advances in neural information processing systems, 2016, pp. 3844–3852.

[10] F. M. Bianchi, D. Grattarola, L. Livi, C. Alippi, Graph neural networks with convolutional arma filters, IEEE Transactions on Pattern Analysis and Machine Intelligence (2021) 1–1doi:10.1109/TPAMI.2021.3054830.

[11] M. Zhang, Z. Cui, M. Neumann, Y. Chen, An end-to-end deep learning architecture for graph classification, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[12] D. V. Tran, N. Navarin, A. Sperduti, On filter size in graph convolutional networks, in: 2018 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE, 2018, pp. 1534–1541.

[13] J. Atwood, D. Towsley, Diffusion-convolutional neural networks, in: Advances in Neural Information Processing Systems, 2016, pp. 1993–2001.

[14] M. Niepert, M. Ahmed, K. Kutzkov, Learning convolutional neural networks for graphs, in: International conference on machine learning, 2016, pp. 2014–2023.

[15] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks?, in: Proceedings of ICLR 2019, 2018.

[16] F. Scarselli, A. C. Tsoi, M. Hagenbuchner, The vapnik–chervonenkis dimension of graph and recursive neural networks, Neural Networks 108 (2018) 248–259.

[17] M. Lukoševičius, H. Jaeger, Reservoir computing approaches to recurrent neural network training, Computer Science Review 3 (3) (2009) 127–149.

[18] H. Jaeger, H. Haas, Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication, Science 304 (5667) (2004) 78–80.

[19] C. Gallicchio, A. Micheli, Tree echo state networks, Neurocomputing 101 (2013) 319–337.

[20] C. Gallicchio, A. Micheli, Graph echo state networks, in: The 2010 International Joint Conference on Neural Networks (IJCNN), IEEE, 2010, pp. 1–8.

[21] C. Gallicchio, A. Micheli, Fast and deep graph neural networks., in: AAAI, 2020, pp. 3898–3905.

[22] D. Bacciu, L. Di Sotto, A non-negative factorization approach to node pooling in graph convolutional neural networks, in: AI*IA 2019 – Advances in Artificial Intelligence, Springer International Publishing, 2019, pp. 294–306.

[23] F. M. Bianchi, D. Grattarola, L. Livi, C. Alippi, Hierarchical representation learning in graph neural networks with node decimation pooling, IEEE Transaction on Neural Networks and Learning Systems.

[24] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, J. Leskovec, Hierarchical graph representation learning withdifferentiable pooling, arXiv preprint arXiv:1806.08804.

[25] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, G. E. Dahl, Message passing neural networks, in: Machine Learning Meets Quantum Physics, Springer, 2020, pp. 199–214.

[26] F. M. Bianchi, C. Gallicchio, A. Micheli, Pyramidal graph echo state networks, in: Proceedings of ESANN, 2020, pp. 573–578.

[27] C. Gallicchio, A. Micheli, L. Pedrelli, Deep reservoir computing: A critical experimental analysis, Neurocomputing 268 (2017) 87–99.

[28] S. Bai, J. Z. Kolter, V. Koltun, Deep equilibrium models, in: Advances in Neural Information Processing Systems, 2019, pp. 690–701.

[29] S. Bai, V. Koltun, J. Z. Kolter, Multiscale deep equilibrium models, Advances in Neural Information Processing Systems 33.

[30] F. M. Bianchi, D. Grattarola, C. Alippi, Spectral clustering with graph neural networks for graph pooling, in: International Conference on Machine Learning, PMLR, 2020, pp. 874–883.

[31] S. J. Hongyang Gao, Graph u-nets, in: Proceedings of the 36th International conference on Machine learning (ICML), 2019.

[32] J. Lee, I. Lee, J. Kang, Self-attention graph pooling, in: Proceedings of the 36th International conference on Machine learning (ICML-19), 2019.

[33] I. S. Dhillon, Y. Guan, B. Kulis, Kernel k-means: spectral clustering and normalized cuts, in: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2004, pp. 551–556.

[34] J. Bruna, W. Zaremba, A. Szlam, Y. LeCun, Spectral networks and locally connected networks on graphs, arXiv preprint arXiv:1312.6203.

[35] M. Fey, J. E. Lenssen, F. Weichert, H. Müller, Splinecnn: Fast geometric deep learning with continuous b-spline kernels, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 869–877.

[36] R. Levie, F. Monti, X. Bresson, M. M. Bronstein, Cayleynets: Graph convolutional neural networks with complex rational spectral filters, IEEE Transactions on Signal Processing 67 (1) (2018) 97–109.

[37] S. A. Vavasis, On the complexity of nonnegative matrix factorization, SIAM Journal on Optimization 20 (3) (2010) 1364–1377.

[38] F. Dorfler, F. Bullo, Kron reduction of graphs with applications to electrical networks, IEEE Transactions on Circuits and Systems I: Regular Papers 60 (1) (2013) 150–163. doi:10.1109/TCSI.2012.2215780.

[39] D. Shuman, M. J. Faraji, P. Vandergheynst, A multiscale pyramid transform for graph signals, IEEE Transactions on Signal Processing 64 (8) (2016) 2119–2134.

[40] S. Scardapane, D. Wang, Randomness in neural networks: an overview, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery.

[41] C. Gallicchio, A. Micheli, L. Pedrelli, Design of deep echo state networks, Neural Networks 108 (2018) 33–47.

[42] C. Gallicchio, A. Micheli, Deep reservoir neural networks for trees, Information Sciences 480 (2019) 174–193.

[43] C. Gallicchio, A. Micheli, Ring reservoir neural networks for graphs, arXiv preprint arXiv:2005.05294.

[44] Y. Orlova, M. Alamgir, U. von Luxburg, Graph kernel benchmark data sets are trivial!, in: ICML Workshop on Features and Structures FEATS, 2015.

[45] F. Errica, M. Podda, D. Bacciu, A. Micheli, A fair comparison of graph neural networks for graph classification, in: International Conference on Learning Representations (ICLR), 2019.

[46] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, M. Neumann, Benchmark data sets for graph kernels (2016).
URL http://graphkernels.cs.tu-dortmund.de

[47] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks?, in: International Conference on Learning Representations, 2019.
URL https://openreview.net/forum?id=ryGs6iA5Km

[48] S. Balakrishnama, A. Ganapathiraju, Linear discriminant analysis-a brief tutorial, in: Institute for Signal and information Processing, Vol. 18, 1998, pp. 1–8.

[49] D. Yu, H. Wang, P. Chen, Z. Wei, Mixed pooling for convolutional neural networks, in: Rough Sets and Knowledge Technology, Springer International Publishing, Cham, 2014, pp. 364–375.