GDL Project Final Report: Towards Sparse Hierarchical Graph Classifiers

Alessia Ruggeri[1]

[1] *Università della Svizzera Italiana (USI)*

(Dated: Academic Year 2018/2019)

In the last years, there have been important advances in the domain of representation learning on graphs, especially regarding the successful use of graph convolutional networks. The new approaches in the field of graph deep learning have led to great results for the node embedding and classification tasks, but still need to be improved to adapt to the graph classification task. Typically, in order to aggregate node features, a global pooling is used; unfortunately this approach prevents the model from learning information about the topology of the graph. Lately, there have been important steps towards the creation of differentiable pooling layers which are able to coarsen the graph in a hierarchical manner, similarly to a local pooling on an image. This allows to preserve the graph topological information and to use them to perform the graph classification.

The focus of this report is on the paper *Towards Sparse Hierarchical Graph Classifiers* which present an interesting architecture for hierarchical graph classification. The aim of this project is to analyze, implement and try to reproduce the results presented by the paper.

I. INTRODUCTION

The paper *Towards Sparse Hierarchical Graph Classifiers* studies the problem of graph classification, where the goal is to predict a class label associated with an entire graph. This task is similar to the image classification task since an image can be considered a special case of a "grid-graph". For this reason, it is intuitive to look for a generalization of CNN to graphs.

In the last few years, there have been several proposals for an abstraction of the convolutional layer to graphs, leading to the creation of different *graph convolutional layers* which demonstrated to be very successful on many node classification benchmarks. The generalization of the pooling layer, however, received less attention and, consequently, the results on graph classification benchmarks aren't such promising. The common approach is to apply a *global pooling layer* to the graph's node features, which reduces the representation of the graph to a single "virtual" node that summarizes all the features of all the original nodes in the graph. The downside of this process is that, while summing up the node features, it isn't able to preserve the graph's topological information.

In order to take into account also the structure of the graph, the pooling layer should be similar to the local pooling layer applied to images, but with the difference that graphs don't have locality. As a replacement parameter, a measure of the importance of the nodes can be used in order to produce the new graph representation. With this approach, it is possible to coarsen the graph through a *hierarchical pooling layer*, which keeps only the most important nodes in each graph.

II. BACKGROUND

As already mentioned, the most common strategy to aggregate nodes representations is to apply a global pooling layer before passing the information to the MLP for the classification. In order to preserve the topology information of graphs, other approaches involving clustering techniques have been proposed. The assignment of nodes to clusters allows to coarsen the graph in a hierarchical manner. However, the majority of these approaches assumes a fixed and deterministic cluster assignment obtained by running a clustering algorithm on the graph nodes; therefore the cluster assignment isn't adaptive to the underlying data.

The paper *Hierarchical Graph Representation Learning with Differentiable Pooling* proposes the first end-to-end trainable CNN with a learnable pooling operator, named *DiffPool*. The main insight at the basis of DiffPool is that the assignment of nodes to clusters can be learned in a differentiable manner, leading to a cluster assignment tailor-made for the underlying data. DiffPool computes soft clustering assignments of nodes and, through some restriction and regularization applied to the assignment, the clustering that it learns eventually converges to an almost-hard clustering.

The main limitation of DiffPool is the quadratic $\mathcal{O}\left(kV^2\right)$ storage complexity with a fixed pooling ratio k. This is due to the need to store an entire assignment matrix which relates the nodes from the original graph to the nodes from the pooled graph in an all-pairs fashion. Consequently, the application of DiffPool is prohibitive for large graphs.

The aim of this project's paper is to demonstrate performance that is comparable to the one of DiffPool on graph classification benchmarks but with a linear $\mathcal{O}\left(V+E\right)$ storage complexity.

III. MODEL

For the graph classification task, the dataset is composed of a set of graphs, each one containing a variable number of nodes, with some associated features and connected in different ways. Each graph is represented by two matrices: the *node features matrix* $\mathbf{X} \in \mathbb{R}^{N \times F}$ and the *adjacency matrix* $\mathbf{A} \in \mathbb{R}^{N \times N}$, where N is the number of nodes in the graph and F is the number of features associated with each node. In our case, we assume the adjacency matrix to be binary and symmetric.

The neural network architecture presented in the paper makes use of a *graph convolutional layer*, a *hierarchical pooling layer* and a *readout layer*, which in turn makes use of some global pooling layers in order to flatten the information into a fixed-size representation, before being processed by the MLP for the final prediction.

A. Graph convolutional layer

The graph convolutional layer used is able to extract and rearrange the features of a graph's nodes in an inductive manner, i.e. without relying on a fixed and known graph structure. For this purpose, the paper's model uses a *mean-pooling propagation rule*:

$$\mathrm{MP}(\mathbf{X}, \mathbf{A}) = \sigma\left(\hat{\mathbf{D}}^{-1}\hat{\mathbf{A}}\mathbf{X}\mathbf{\Theta} + \mathbf{X}\mathbf{\Theta}'\right) \tag{1}$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ is the adjacency matrix with inserted self-loops; $\hat{\mathbf{D}}$ is the corresponding degree matrix, constructed as $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$; σ is the activation function, which in this case is the rectified linear (ReLU) activation; $\mathbf{\Theta}, \mathbf{\Theta}' \in \mathbb{R}^{F \times F'}$ are learnable linear transformations applied to every node. The right part of the formula, represented by the transformation of $\mathbf{X}$ through $\mathbf{\Theta}'$, is a simple *skip-connection*. Its role is to preserve the information about the central node against the information about the neighbourhood. While the input node features matrix $\mathbf{X}$ has dimensions $N \times F$, after the application of the graph convolutional layer the output will be an $N \times F'$ node features matrix.

B. Hierarchical pooling layer

The hierarchical pooling layer coarsens the graph by dropping some of its nodes, thus down-sampling the graph representation. The approach used in this paper is different from the one used for DiffPool and it has been proposed before in the *Graph U-Net* paper.

The hierarchical pooling layer reduces the graph size with a *pooling ratio* $k \in (0, 1]$, so that, after the application of the layer, the graph will remain with $\lceil kN \rceil$ nodes against the initial N nodes. The choice of the nodes to drop is made based on a *projection score* against a learnable vector $\vec{p} \in \mathbb{R}^F$. All the nodes features of a graph are projected to 1D through $\vec{p}$ and the resulting values measure how much information of each node can be retained when projected onto the direction of $\vec{p}$. Since we wish to preserve as much information as possible from the original graph, we can interpret the projection value for each node as a score of the importance of that node in the graph. Consequently, in order to coarsen the graph, we perform the *k-top* operation in order to select the indices of the nodes with a higher score. These indices are then used to accordingly slice the adjacency matrix and the node features matrix, keeping only the most important nodes and dropping the others.

In order to let the gradients flow into $\vec{p}$, the projection scores are also used as *gating values*: the sliced node features matrix is multiplied element-wise to the projection scores, so that the nodes with lower scores will experience less significant feature preservation, and vice-versa.

The whole pooling operation, with $(\mathbf{X}, \mathbf{A})$ as input, can be expressed mathematically as:

$$\vec{y} = \frac{\mathbf{X}\vec{p}}{\|\vec{p}\|} \qquad \vec{i} = \mathrm{top}-k(\vec{y}, k) \qquad \tilde{y} = \tanh(\vec{y}(\vec{i}))$$
$$\tilde{\mathbf{X}} = \mathbf{X}(\vec{i}, :) \qquad \mathbf{X}' = (\tilde{\mathbf{X}} \odot \tilde{y}) \qquad \mathbf{A}' = \mathbf{A}(\vec{i}, \vec{i}) \tag{2}$$

where $\|\cdot\|$ is the L_2 norm, *top-k* is the operation that select the indices of the $\lceil kN \rceil$ nodes with higher score, $\odot$ is the (broadcasted) element-wise multiplication. Unlike DiffPool, this hierarchical pooling layer drops rather than aggregate nodes and it doesn't need to store additionally matrices, since the projection and slicing operations are performed directly on the original input matrices.

The hierarchical pooling operation is performed right after each graph convolutional layer, representing together a *conv-pool block.*

C. Readout layer

The readout layer is a flattening operation that preserves the information about the input graph in a fixed-size representation. For this purpose, it makes use of three different types of global pooling operations. After each conv-pool block, the resulting information is stored through a concatenation of a *global average pooling* and a *global max pooling* operations applied to the output of the block. Lastly, just before the MLP for the prediction, all the concatenations of global pooling results obtained by the multiple conv-pool blocks are aggregated using a *global sum pooling* operation.

$$\vec{s}^{(l)} = \frac{1}{N^{(l)}} \sum_{i=1}^{N^{(l)}} \vec{x}_i^{(l)} \,\|\, \max_{i=1}^{N} \vec{x}_i^{(l)} \qquad \vec{s} = \sum_{l=1}^{L} \vec{s}^{(l)}$$
$$\text{predictions} = \mathbf{MLP}(\vec{s}) \tag{3}$$

where $\|$ denotes the concatenation operation, $N^{(l)}$ is the number of nodes in the graph and $\vec{x}_i^{(l)}$ is the i-th node's features vector.

The aggregation across layers is important since it allows to retain information on graphs at different scales of the processing; it also preserves some information about smaller graphs that may be quickly be pooled down to a too small number of nodes.

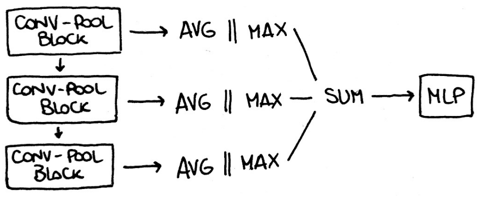

Figure 1: Readout layer.

D. Architecture

The entire model architecture is composed of *three conv-pool blocks*. Each one of them is represented by a graph convolutional layer followed by a hierarchical pooling layer and, for each conv-pool block, a concatenation of global average pooling and global max pooling operations on the output is stored. After the three conv-pool blocks, their representations through concatenations are aggregated by a global sum pooling layer, whose fixed-size output is submitted to an MLP to obtain final predictions.

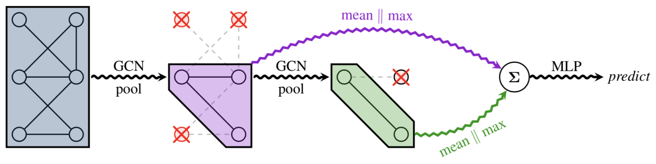

Figure 2: Full pipeline (with two conv-pool blocks, for $k = 0.5$) - from the original paper.

IV. EXPERIMENTS

A. Datasets

The paper's model has been evaluated on four well known graph classification benchmark tasks: the biological datasets *Enzymes*, *Proteins*, *D&D* and the scientific collaboration dataset *Collab*. The results reported in the paper have been achieved by performing 10-fold cross-validation on each dataset.

B. Model parameters

As already mentioned, the model is composed of three conv-pool blocks, each of them consisting of a graph convolutional layer with 128 (Enzymes and Collab) or 64 (Proteins and D&D) features, followed by a hierarchical pooling step. A pooling rate $k = 0.8$ ensures that, after the coarsening of each graph, there is enough information by preserving the 80% of the nodes in the graph. A learning rate of 0.005 was used for Proteins and 0.0005 for all other datasets. The model was trained using Adam optimizer for 100 epochs on Enzymes, 40 on Proteins, 20 on D&D and 30 on Collab.

C. My implementation

For this project, I replicated the exact same model architecture described in the paper to attempt to obtain the same results. I used Python 3.6 and Tensorflow 2.0 to implement all the layers from scratch and I ran the experiments on *Google Colab* GPUs [1].

In addition to the hierarchical model, I also implemented a much simpler GCN model to be used as a *baseline*. The baseline GCN consists of three graph convolutional layers, followed by a global max pooling layer just before the MLP for the classification.

For both the two models, I used the same parameters of the paper, except for the number of epochs, which in my case was higher. I also added some regularization parameters to the model. My model has been tested only on two of the four datasets, *Enzymes* and *Proteins*.

D. Results

Table I shows the results of the performances achieved by the paper's model compared to the ones of DiffPool variants (to see the comparison with other models, have a look at the original paper).

	Datasets			
Model	*Enzymes*	*Proteins*	*D&D*	*Collab*
DiffPool-Det	58.33	75.62	75.47	**82.13**
DiffPool	**64.23**	**78.10**	**81.15**	75.50
Paper's model	64.17	75.46	78.59	74.54
My model	49.16	69.66	\	\
My baseline	55.83	63.09	\	\

Table I: Classification accuracy percentages in comparison.

The paper's algorithm successfully competes with the variants of DiffPool, obtaining very close results to the state-of-the-art. Unlike DiffPool, the paper's model does not require quadratic memory, as shown by the experiment in Figure 3. Therefore, this method represents a scalable hierarchical graph classification algorithms applicable to large real-world datasets.

[1] My implementation can be found at: `https://github.com/HeapHop30/hierarchical-pooling`

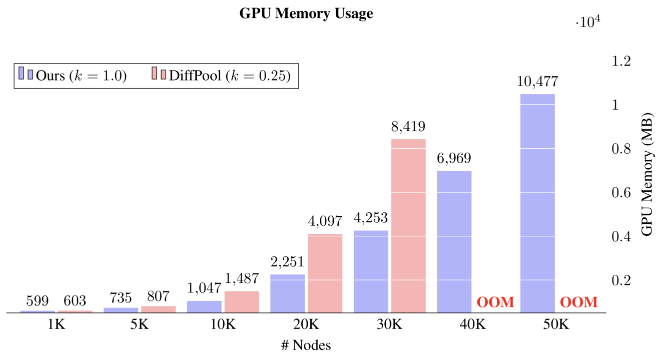

Figure 3: GPU memory usage of paper's method (with no pooling: $k = 1.0$) and DiffPool ($k = 0.25$) during training - from the original paper.

	Datasets	
Model	*Enzymes*	*Proteins*
Paper's model	64.17	75.46
My model	49.16	69.66
My baseline	55.83	63.09

Table II: Classification accuracy percentages in comparison between my model and the paper's model.

V. DISCUSSION

As we have seen from the classification accuracy percentages, the paper's model was able to achieve its purpose of obtaining performance comparable to DiffPool on graph classification benchmarks with a linear storage complexity.

Unfortunately, my implementation of the model wasn't able to reproduce the results presented in the paper. As shown in Table II, the accuracy value obtained on Proteins is not far away from the one indicated in the paper, but it isn't close enough to consider it similar. The result obtained on Enzymes, on the other hand, is even worse than the baseline, even though the numbers shown in the table are the best my model was able to achieve. In general, the accuracy values that I obtained training and testing my model were very variable and hugely influenced by the model parameters.

The model I implemented is not optimized to be fast, but I can confidently say to be correct and to accurately reflect the architecture described in the paper. Consequently, I have reason to believe that the authors omitted crucial details in the paper, which are essential for the proper functioning of the model. This assumption is confirmed also by the fact that there doesn't exist any implementation on the Internet that was able to replicate the paper's results. Therefore, it cannot be confirmed that the performances showed in the paper are achievable using the exact same architecture described in the paper, since the authors didn't publish any official implementation on the Internet.

REFERENCES

[1] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. Towards sparse hierarchical graph classifiers, November 2018. URL `https://arxiv.org/abs/1811.01287`. arXiv:1811.01287 [**stat.ML**].

[2] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling, February 2019 (first version in 2018). URL `https://arxiv.org/abs/1806.08804`. arXiv:1806.08804 [**cs.LG**].

[3] Hongyang Gao and Shuiwang Ji. Graph u-nets, May 2019 (first version in 2018). URL `https://arxiv.org/abs/1905.05178`. arXiv:1905.05178 [**cs.LG**].

[4] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, February 2017 (first version in 2016). URL `https://arxiv.org/abs/1609.02907`. arXiv:1609.02907 [**cs.LG**].

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, December 2015. URL `https://arxiv.org/abs/1512.03385`. arXiv:1512.03385 [**cs.CV**].