DGM and MPR pooling report

Simone Eandi

May 15, 2021

1 Introduction

The following is a report describing the process of implementing in the *Pytorch Geometric*[1] framework two modules proposed in the paper *"Deep Graph Mapper: Seeing Graphs through the Neural Lens"*[2], namely Deep Graph Mapper(`DGM`) and `MPR Pooling`. The first module provides a way for generating compact and flexible visualizations of large graphs, while the second is a dense pooling layer. Our implementation of `DGM` and `MPR Pooling` follows, at least in logic structure, the official implementation released by the authors [3]. For this reason, when describing implementation, the will be focus on the parts that differ from the original code.

This report is structured as follows. Section 2 is devoted to the Deep Graph Mapper and contains a quick overview of how the module works, followed by the description of the relevant implementation choices. Section 3 is dedicated to `MPR Pooling` , instead, and follows the same structure of Section 2. Finally, section 4 contains few conclusive thoughts on the whole implementation process.

2 Deep Graph Mapper

Deep Graph Mapper is a visualization tool for large graphs that is based on Mapper [4] and uses a GNN as the lens function instead of a graph theoretic function as common in previous works. Prior mapper-based visualization tools have two main limitations that derives from the use of these functions: `(i)` they do not scale to large graphs and `(ii)` cannot properly leverage the node and edge features for building the compact representation. Using a trainable GNN as lens function solves both these problems.

2.1 Overview

Mapper for graphs Let $G = (V, E)$ be the input graph, $f : V -> I^d$ the lens function mapping the graph nodes to a d-dimensional embedding space and let $U = \{U_i\}_{i=1}^N$ be an open cover of I, that is, a set of sub-intervals U_i such that their union is the interval I. Given these inputs, Mapper first builds the *pull back cover* $f^*(U)$ of V induced by f and U, which is defined as the collection of N open sets, such that the i-th set is given by the subset of nodes of G that belongs to the preimage of f with respect to the interval U_i, i.e. $f^{-1}(U_i)$. Then Mapper builds the *refined pull back cover* $\bar{U} = \{U_j\}_{j=1}^{J \geq N}$ given by the connected components of each set in $f^*(U)$. Finally, Mapper outputs a graph that has exactly one node u_j for each U_j in the refined pull back cover $\bar{U}$ (whose features are the average features of nodes in U_j) and such that nodes u_i and u_j are connected if $\bar{U}_i$ and $\bar{U}_j$ have a non-empty intersection (**figure 1**).

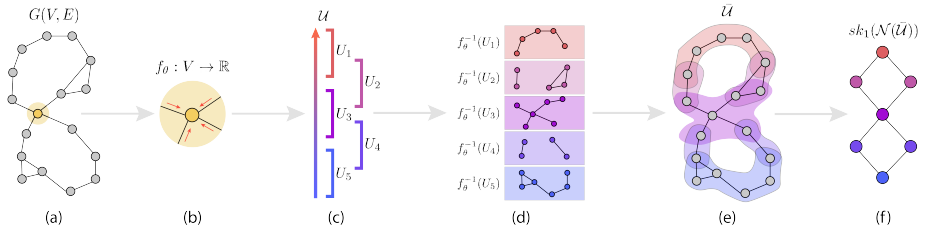


Figure 1: The Mapper work flow [2]. Given an input graph (a) and open cover (c), Mapper computes the pull back cover (d) and produces the mapped graph (f).

Deep Graph Mapper `DGM` builds on top of Mapper and uses as lens function f a trained GNN for node classification/regression (supervised or unsupervised). This allows great flexibility in how the mapped graph is constructed, since the GNN can be trained with different loss functions to highlight different features of the input graph in the visualization, scales well and can make use of all the information contained in the graph. If the GNN helps in producing more understandable visualizations, the authors also suggests using `DGM` for doing the inverse: using the `DGM` visualization to better understand the GNN model itself. For example, by plotting two visualizations, one where the nodes are colored by the node features predicted by the model and one using the nodes true label used for training, allows to better understand when the model fails to make correct predictions (**figure 2**).

2.2 Implementation

We implemented the Deep Graph Mapper tool in *Pytorch Geometric* as a module in *nn.models*, see **algorithm 1** for the pseudo code of our implementation.

Algorithm 1 Deep Graph Mapper

Require: Lens f, embed dims d, num intervals n, overlap o
  procedure INIT(f, n, o, d)
    Store f
    Make intervals $U = \{U_i\}^n$ in $[0, 1]^d$
  procedure VISUALIZE(Graph X)
    Compute embedding $f(X)$
    Reduce and normalize $f(X)$
    Compute pull back cover $f^*(U)$ from X, f, U
    Compute refined pull back cover $\bar{U}$
    for $U_j \in \bar{U}$ **do**
      Add new node v_j to mapped graph M
    Add edges to M
    Filter out small components from M
    Plot M

Generating the open cover U The precise definition of the interval overlap ratio o of sub-intervals in the open cover U of $[0, 1]$ used by the authors was ambiguous both in the paper and in their code. Furthermore, the procedure used in the authors code for generating the open cover of $[0, 1]$ appeared to produce intervals outside this range. We suspect this to be a bug that went unnoticed due to its marginal impact in the visualization. For this reason, a different procedure for making the open cover was used, which is based on the definition of overlap o as the ratio between the length of overlap of any two consecutive intervals and the length of an interval.

One lens, multiple graphs The authors implemented `DGM` as a class with almost no state and a single "public" method that expects an input graph X and the pre-computed embedding $f(X)$ and produces the visualization. Since the same GNN lens f can be trained over a wide set of graphs belonging to the same family/dataset, we opted for a 2-steps stateful approach where at initialization the lens f is stored and the open cover U is pre-computed and the visualization procedure is invoked for each input graph. In this way, the

visualization of a dataset of graphs supporting the same lens can be generated without redundant operations and instantiating a singe `DGM` object.

Comparison See **figure 3** for a comparison between the visualizations generated by our implementation (top) and the authors' (bottom) for the Cora dataset using similar hyper-parameters. The two visualizations are quite close, with few differences that can be attributed to the different procedure used for making the cover U.

3 MPR Pooling

Other than the `DGM` , the paper also proposes to combine Mapper and the PageRank algorithm to make a new pooling technique that they name Mapper-based PageRank Pooling.

3.1 Overview

The `MPR Pooling` layer works as follows: first it computes the PageRank scores $pr(V)$ of the nodes of the input graph $G = (V, E)$ received from the upper layers and normalizes them, such that they span the whole $[0, 1]$ interval, instead of just a subset of it. Then, it builds a pull back cover $pr^*(U)$ using these scores and an open cover U of $[0, 1]$ with n sub-intervals, in a way analogous to that described in section 2.1. Finally, the pooled graph is obtained by adding to an empty graph one vertex u_i for each subset in $pr^*(U)$ and connecting together two nodes u_i and u_j if $U_i \cap U_j \neq \emptyset$. The final pooled graph will have exactly as many nodes as the number n of sub-intervals in the open cover U. Notably, differently from `DGM` , the pull-back cover is directly used for creating the output graph, instead of refining it first (i.e. dividing each subset by its connected components). It is not explained why the authors made this choice, but a possible reason is to control exactly how many nodes will have the pooled graph.

3.2 Implementation

The same custom procedure for generating the open cover adopted for `DGM` has also been used for the `MPR Pooling` layer for the same reasons explained above. Other than that, the following are the few parts where the new implementation (located in *nn.dense*) differs significantly from the original one.

Computing the PageRank scores The PageRank scores are computed in the original implementation by converting the input graph into a *netoworkx* graph and applying the scipy PageRank algorithm. This process turned out to be a major bottleneck for the pooling process and, surprisingly, was not due to PageRank itself, but due to the conversion between the graph adjacency matrix and the *networkx* internal representation. For this reason, a sparse iterative PageRank implementation based on [5, 6] that could operate directly from a sparse adjacency matrix was used. This resulted in a large speedup of the pooling layer, with the torch geometric implementation being almost twice as fast as the original one.

Pooling the node features The features of the n pooled nodes X_p are obtained from the features of the m nodes of the input graph X as $X_p = SX$, where S is the usual n by m selection matrix. This matrix is built by first taking

$$S_{ji} = \begin{cases} 1 & if i \in pr^{-1}(U_j) \\ 0 & otherwise \end{cases}$$

and then normalizing it. The choice of normalization method, however, differs between the paper and the official implementation: in the first they normalize each column of S_{ji} by the number of sub-intervals of U in which the i-th node of X appears, while in they second they use column-wise softmax activation. The latter was used in our implementation in Pytorch Geometric.

Comparison Our implementation was tested on the D&D dataset using the same module and training hyper-parameters reported by the authors, with and without using pooling layers. The results are shown in **table 1**, where it can be seen that our code achieves the same accuracy reported by the authors, moreover, one can also see the large beneficial impact the using pooling layers can have on a node classification task.

	reported accuracy	our accuracy	our accuracy w/out pooling
D&D	$7.82\% \pm 3.4\%$	$7.85\% \pm 3.7\%$	$71.6\% \pm 3.3\%$

Table 1: Accuracy comparison on the D&D dataset.

4 Conclusions

Although there were a few ambiguities both in the paper and in the code, the work done by the authors was overall very well documented and highly

reproducible making the process of creating a new implementation easier. This also allowed to focus more on the performance of the code, instead of its correctness, resulting in a new implementation of `MPR Pooling` that is, on average, twice as fast as the original one.

References

[1] https://github.com/rusty1s/pytorch_geometric.

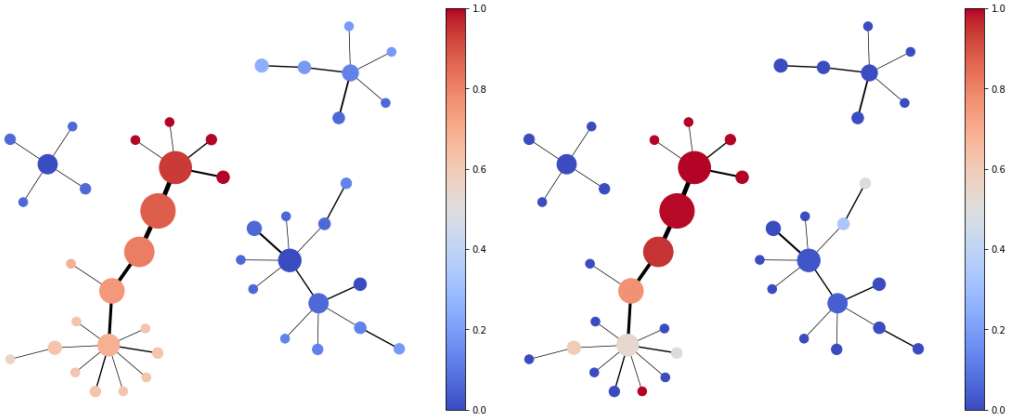
[2] Cristian Bodnar, Cătălina Cangea, and Pietro Liò. Deep graph mapper: Seeing graphs through the neural lens, 2020.

[3] https://github.com/crisbodnar/dgm.

[4] Singh Gurjeet, Memoli Facundo, and Carlsson Gunnar. Topological methods for the analysis of high dimensional data sets and 3d object recognition, 2007.

[5] https://it.mathworks.com/moler/exm.html.

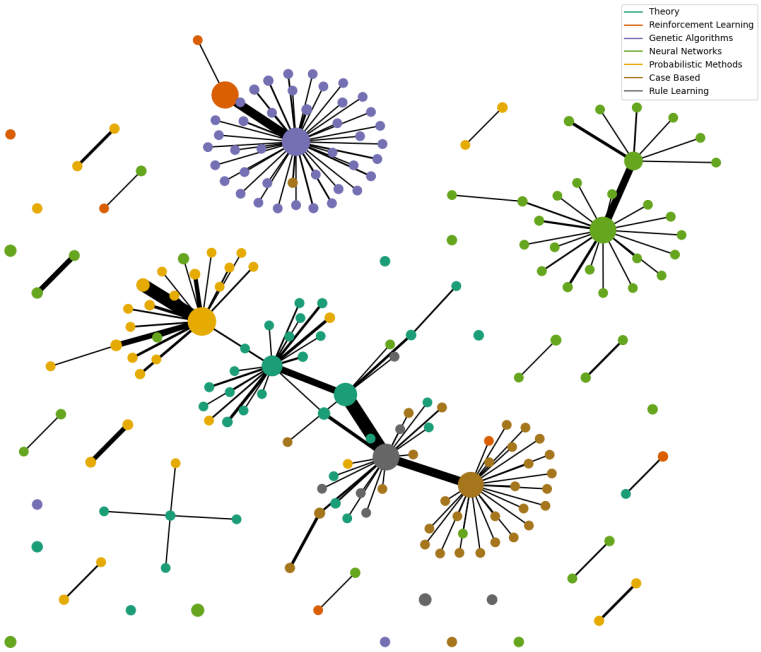
[6] https://github.com/asajadi/fast-pagerank.git.



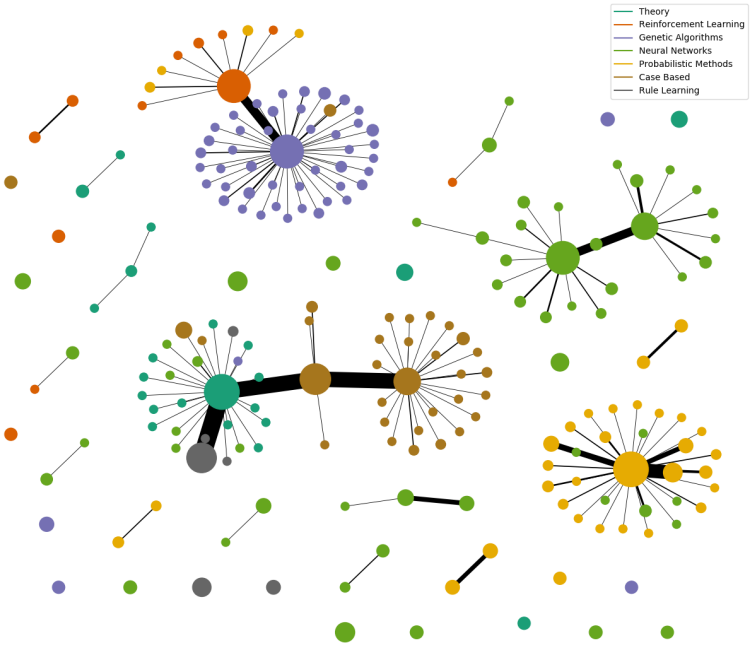
(a) Colored with GNN predictions

(b) Colored with true labels

Figure 2: DGM visualizations for the spam dataset. The same GNN is used in both cases, but on the left nodes are colored according the lens node features predictions, while on the right the true node labels are used for coloring. The comparison allows to better understand how the GNN classification model works.



(a) Our implementation



(b) Official implementation

Figure 3: Comparison of the visualizations for the Cora dataset generated using our implementation of DGM and the original one.