Graph coarsening with neural networks

Peter Buttaroni
peter.buttaroni@usi.ch

Shubhayu Das
shubhayu.das@usi.ch

Giorgio Macauda
giorgio.macauda@usi.ch

Abstract

Due to their flexibility, graphs are becoming common in many fields. However, concurrently with their diffusion, also their sizes are increasing. This is a challenge in terms of both computational power and time. That is why techniques to reduce the size of the graph while preserving properties of our interest are becoming essential tools. However, current graph coarsening algorithms are rigid and have no space for adjustments to preserve those properties we want to study. Motivated by these observations, Cai et al. (2021) proposed a coarsening method based on Graph Neural Networks that can be adapted to different loss functions, reduction ratios, graph sizes, and types. The goal of this work is to reproduce both the methodology and the results of their paper.

1 Introduction

Graph are ubiquitous in many fields. They are flexible structures that can be used to model almost every system. In the last years the size of these graphs has increased dramatically. This poses severe limits in terms of computation when we want to extract from them useful information. That's why nowadays a way of reducing the size of graphs while preserving some useful properties is crucial.

A very common method to achieve this task is *graph coarsening* (Fig.1). This method reduces the number of nodes by contracting disjoint sets of connected vertices. The traditional graph coarsening approaches have two main restrictions that limit their flexibility. First of all, the algorithm of coarsening is fixed and does not take into account which properties of the original graph we want to retain in the coarse one. The second issue is that we cannot decide the weights of the edges on the coarse graph. These weights are a product of the algorithm and there is no space for adjustments.

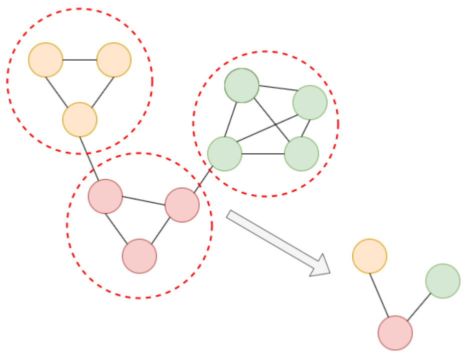


Figure 1: Coarsening graph example

In order to overcome this issues Cai et al. (1) proposed an approach to adjust the weights of the coarsen graph using a type of graph neural network (GNN) called `GOREN`. The goal is to preserve in the coarse graph a quantity of interest (namely the Laplace operator). The authors were able to improve the results wrt the traditional coarsening methods. The aim of this work is to reproduce their results. The report will be structured as follows. First, we will describe what graph coarsening is. Second, we will present the `GOREN` model, the methodology and the dataset used. Finally we will describe the results of our implementation of the model, the challenges we faced during the process, and the discrepancies we have found wrt the paper results.

2 Background

Graph Coarsening. The original idea of the graph coarsening is not new. It was first introduced in the algebraic multigrid literature (Ruge & Stüben, 1987) (2). More recently, Loukas has focused his researches on preserving properties related to the spectrum of the original graph and coarse graph. (4) (5).

Graph sparsification. Graph sparsification was firstly proposed to solve linear systems involving combinatorial graph Laplacian efficiently. In 2008 Spielman & Srivastava (3) showed that for any undirected graph G

of N vertices, we can create a spectral sparsifier in nearly-linear time. This spectral sparsifier has $O(N \log^c N/\epsilon^2)$ edges.

3 Methodology

In this section we will present the methodology used in the paper. We followed the same approach in order to replicate their results.

3.1 A two steps approach

The approach used in the paper is articulated in the two following steps.

- First, we construct the coarse graph by using a traditional coarsening technique (in our attempt of reproducing the original paper we have used the baseline approach discussed in the section 4.2).
- Second, we estimate the weights of the coarse graph edges using the `GOREN` network. This second step allows us to partially model the coarse graph. It is in this step that we adapt the coarsening method to our purposes using a GNN.

3.2 The traditional coarsening step

Assume we have a graph $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_N\}$ is the set of vertices and E is the set of edges connecting these vertices.
The first thing that we need to do is to apply a reduction algorithm in order to get the coarse graph $\hat{G} = (\hat{V}, \hat{E})$. There are many possible coarsening methods that we can use. We have decided to use the baseline algorithm presented in the paper. We choose randomly n nodes in the original graph. These will be the supernodes of our coarse graph. Then, we map all the remaining nodes to the closest supernodes. In case of tie, we randomly choose one among the closest. All the graph considered are connected.

3.3 The edge weights estimation

This is the phase in which GNN plays its role. We start from the coarse graph structure defined in the previous step and we predict the weight of each edge using the `GOREN` model.
In the Figure 2 you can see a scheme of the functioning of the `GOREN`. We choose an edge in the coarse graph and we extract in the original graph the corresponding subgraph. We feed this subgraph to the `GOREN` and we obtain a scalar. This scalar will be the new weight of the coarse graph edge.

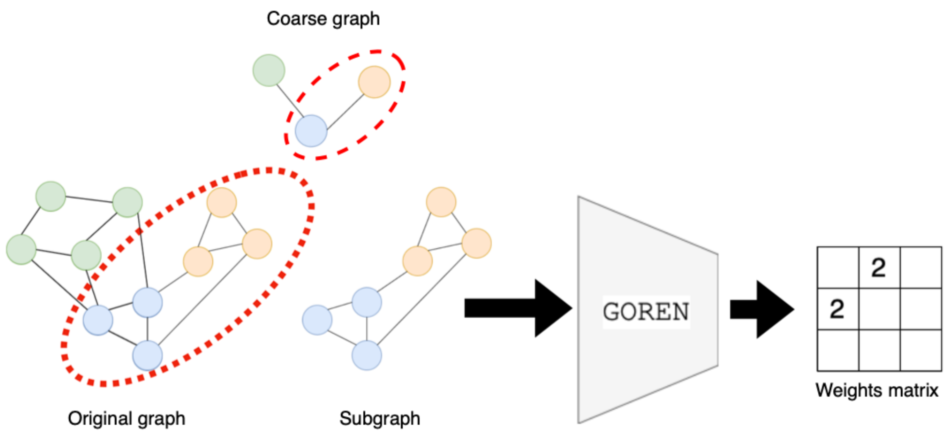


Figure 2: Predicting edge weights of the coarse graph

In Figure 3 we can see the architecture of the `GOREN`. The model takes in input the nodes features $h(v_i)$ and the edges features $h(e_{i,j})$ of the entire subgraph. It transform them to vectors of size 50 using a MLP. After this step we pass the data in 3 GINConv layers. Finally the result is passed through a fully connected layer to get a single value.

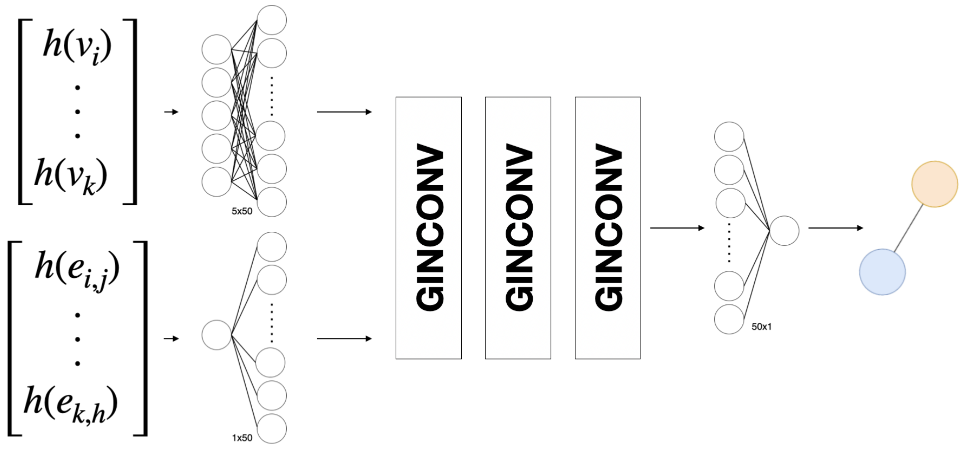


Figure 3: `GOREN` architecture

The edge features are just scalar of value 1. Instead, each node has 5 feature defined using a simple local degree profile (LDP)(Cai & Wang, 2018)(6). This means that for each node $v \in G(V)$, let $DN(v)$ denote the set of the degree of the neighbors of v (i.e. $DN(v) = \{degree(u)|(u,v) \in E\}$), we have that the features are:

$$h(v) = (degree(v), min(DN(v)), max(DN(v)), mean(DN(v)), std(DN(v)))$$

The GINConv layer is the main component of the `GOREN` architecture. This layer is based on a modification of the Graph Isomorphism Network (GIN) proposed by Xu (7).
Equation (1) shows the exact structure of the GINConv layer of `GOREN`.

$$h_v^{(k)} = ReLU\left(MLP^{(k)}\left(\sum_{u\in\mathcal{N}(v)\cup(v)} h_u^{(k-1)} + \sum_{e=(v,u):u\in\mathcal{N}(v)\cup(v)} h_e^{(k-1)}\right)\right) \tag{1}$$

After the three GIN Conv layers, the outputs of the fully connected layer (that are the pooled features of the nodes) are averaged (Eq. 2) and passed through a ReLu (Eq. 3) (in this way we ensure that the edge weights of the coarse graph are positive).

$$h_G = MEAN\left(\{h_v^{(K)}|v \in G\}\right) \tag{2}$$

$$w = 1 + ReLu(\Phi(h_G)) \tag{3}$$

Is important to highlight that in our implementation we have used the LeakyReLU function instead of the ReLU. The reason was that using the latter we incurred in the Dying ReLU problem. In other words, the backward becomes ineffective when the output of the neural network is always 0.

4 Dataset

In order to test the presented approach, the authors use a large amount of data. The choosen dataset are either synthetic or real, ranging from proteins to social network graphs. Since the purpose of the paper is to increase the quantity of information retained by the coarse graph, they needed to select also a coarsening algorithm. Considering they also wanted a general procedure, they had to chose and test many different coarsening algorithms. For more details, and the complete list of the datasets and algorithms used, you are invited to see the reference to the original paper.

Our goal is to replicate the results claimed by the authors. Hence, we decided to start with one of the presented dataset given that all of them should be replicable, and eventually we could expand our tests to the others.

4.1 Choosen dataset

We selected a synthetic graph dataset created using the **Erdős–Rényi model** which is already implemented in pythorch geometric. The probability of having an edge between two nodes can be customized as the authors did. In fact, it is setted as $p = (512 * 0.1)/size$. Towards generalization, they created 25 graphs of size $512, 612, 712, ..., 2912$ where the first 5 are used as training set, the following 5 as validation set and the remaining as test set. Note that the test set is on average 2.6x bigger than the training set, this because one of the goal of the authors is to handle the increasing size of graphs with a limited computational power. So they want it to generalize well after training on portions of the graphs.

4.2 Coarsening algorithm

As mentioned before, the coarsening algorithm we selected is called **Baseline algorithm**. It is a simple algorithm that randomly select k super nodes and then assign all the other nodes to the closest super node previously created. After that, an edge is placed between two super nodes if there is at least one edge between the two portions of graphs in the original graph, and the weight is just the sum of all weights crossing the two portions. Finally, the number of super nodes must be decided. In this case they suggest different reduction ratios, $0.3, 0.5, 0.7$, we picked 0.5 in order to accomplish our tests.

5 Evaluation

The main quality measure the authors use is called eigen error and its improvement after training. Given a Laplacian matrix of a graph L_G, that could be either combinatorial, doubly-weighted or normalized as described in the paper, we can extract the eigenvalues with a simple eigen analysis and compute the eigen error wrt to another graph Laplacian $L_{\widehat{G}}$, in this case, considering the one of the reduced graph, it can be computed as:

$$EE = \frac{1}{k} \sum_{i=1}^{k} \frac{\left|\widehat{\lambda}_i - \lambda_i\right|}{\lambda_i} \tag{4}$$

In particular, they use the first k smallest eigenvalues, where k is an hyper parameter and depends on the specific type of graph used. For our choosen dataset $k = 40$.

Once the model is trained, it should be able to predict the optimal weights of the reduced graph, this reflects in changes in the Laplacian $L_{\widehat{G}}$ that we can call $L_{\widehat{G_t}}$ for better differentiation. Calculated the improved Laplacian $L_{\widehat{G_t}}$ we can compute again the eigen error EE_t wrt the original L_G, and finally the improvement in percentage is given by:

$$improvement = \frac{EE - EE_t}{EE} \tag{5}$$

Unfortunately, in practice, this quality measure is not differentiable so they had to use another function as a proxy loss in order to train the model. They proposed two loss functions called Rayleigh loss and Quadratic loss. They also mention that GOREN uses the Rayleigh Loss in training even though they never report that quantity as they did with the quadratic loss. So we report here the quadratic loss formula for simplicity, since we used this one.

$$\text{Loss}\left(L_G, L_{\widehat{G_t}}\right) = \frac{1}{k} \sum_{i=1}^{k} \left| f_i^T L_G f_i - (P f_i)^T L_{\widehat{G_t}} (P f_i) \right| \tag{6}$$

Where f_i is the i-th eigenvector of the combinatorial Laplacian of the original graph L_G and P is the projector operator that simply scales down the eigenvectors of the original graph to the same size of the reduced graph. For more detail we refer you to the original article.

Note that the Reyleigh loss is associated with the doubly-weighted Laplacian for the reduced graph and the quadratic loss with the combinatorial Laplacian. Since we are using the quadratic loss we decided to use the combinatorial Laplacian for both the reduced graph and the original graph. For the sake of completeness we report here the formula of the combinatorial Laplacian:

$$L = D - W \tag{7}$$

where D is a diagonal matrix with each value i on the main diagonal is the sum of the i-th row of the adjacency matrix of the graph. The formula is true also for the reduced graph using the hat version of the variables.

Finally, one more thing to notice is that in all our plots and computation of the eigen error we always exclude the first eigenvalue since by definition, when there is one connected component, the lowest eigenvalue is always equal to zero. Moreover, we guess for a machine precision error, it is never exactly zero. This leads to an explosion of the eigen error which is undesirable and does not reflect the truth.

6 Training

Before describing the training session we need to describe how a batch is defined. The paper is inconveniently vague about it and we had to contact the authors to understand it better. As shown in 6, in order to compute the loss, the weights matrix of the reduced graph must be entirely predicted otherwise there would be a zero for the edges not yet processed by the network, which means, no edge at all. Furthermore, every forward require a subgraph which may have, and often has, different shape wrt the others, so they cannot be stacked together and fed to the network.

In order to solve this problem, for the first epoch of a graph they initialize the weights matrix $\widehat{W_t}^{(0)} = \widehat{W}$. Where $\widehat{W}$ is the weights matrix of the reduced graph computed as the heuristic coarsening algorithm does. Then, since every batch has size 600, they do 600 forwards replacing the weight for the edges just predicted. Then the backward and optimization step, using $\widehat{W_t}^{(k)}$ to compute the loss, is executed. Since they train 50 epoch on every graph, at the end of the epoch, the training start again using the last version $\widehat{W_t}^{(K)}$ obtained from the last batch of the previous epoch.

At the end of every epoch, the model is evaluated on the validation set. Indeed, every graph of the validation set is entirely forwarded through the network, edge by edge, and using the predicted weights matrix $\widehat{W_t}$, the improvement of the eigen error is computed. If it is better than the previous best one we store the model. Then the training set edges are also shuffled. At end of 50 epochs, the subsequent graph training start from the previous best model, again initializing $\widehat{W_t}^{(0)} = \widehat{W}$ with his own original weights.

6.1 Hyper parameters recap

We provide a concise list of parameters and hyper-parameters used:

1. Weights update: every batch
2. Epochs: 50 on each graph
3. Learning rate: 0.001
4. Laplacian: Both combinatorial
5. Loss: Quadratic loss
6. Batch size: 600
7. Reduction ratio: 0.5

6.2 Time complexity

The authors claims they can train on the same dataset in a few minutes. In our, case the model takes about 8 hours to complete the training. This could be due to the lack of memory that led us to compute the batches on the fly many times while they could be all pre-processed. Furthermore, it may be due to an inefficient computation from our side despite our attention on that.

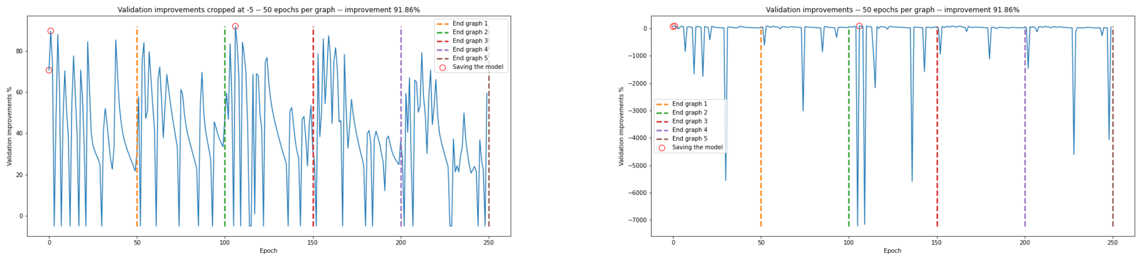
6.3 Doubts

Note that this process is not explained with this level of details by the authors, so we had to interpret their words. There are many things that could be done differently. For example, at the beginning of every epoch $\widehat{W_t}^{(0)}$ could be restored to $\widehat{W}$, or it could also be restored at end of every batch, ending the epoch with an extra forward and no back-propagation to provide a new $\widehat{W_t}^{(0)}$ for the next epoch. Another example concern the storing of the model, it is not given to know whether it should be saved wrt the validation eigen error improvements or the validation loss. Finally, we do not know if they start the training of the graphs following the first from the best model so far or the the last model used.

7 Results

For the given dataset, and as said before, the authors do no provide any eigen error evaluation using the combinatorial Laplacian for both original and reduced graph. However, since our model is stored when the average validation improvement increases, and we used the combinatorial Laplacian to compute it, we believe it is a necessary step to measure the quality of the model at the end using this Laplacian.

The following plots shows how the eigen error goes during training:

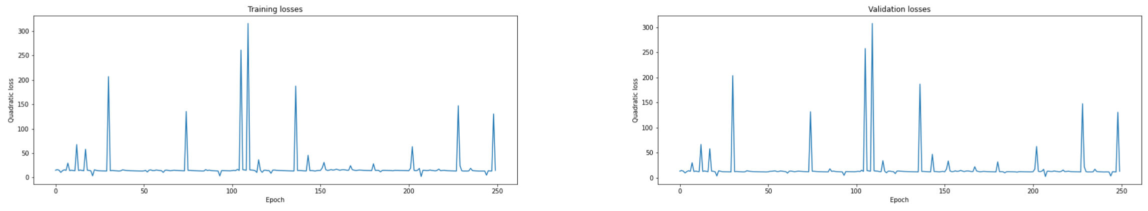


(a) Validation improvement cropped at -5%

(b) Uncropped validation improvements

Figure 4: Validation improvement in percentage, computed using the combinatorial Laplacian for both orginal and reduced graph

In our opinion the function looks too oscillating, perhaps for some hidden bug in our implementation of GOREN, more analysis is needed to understand that. First thing to notice is how many times the model is saved, just three, highlighted by a red circle in figure 4. This might mean that 250 epochs are useless since we are wasting a huge amount of computational power to get almost the best improvement within the first 3 epochs. For the sake of completeness we also include here the training loss, computed at end of every epoch with an extra forward and without back-propagation and the validation loss computed at the same time:

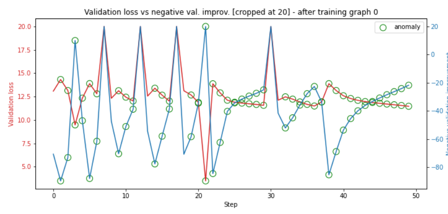


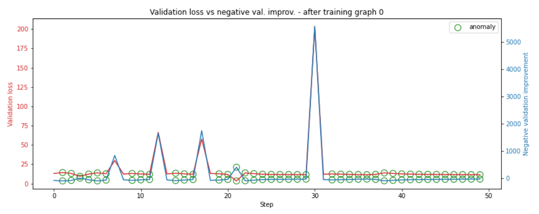
(a) Training losses

(b) Validation losses

Figure 5: Training and validation loss computed at the end of every epoch with no gradient with an extra forward. So $\widehat{W}_t$ is computed from scratch.

Even though very slowly, the loss is going down but something still looks weird. Assuming that it is acceptable, we wonder whether the loss is really a good proxy for the eigen error. We did it empirically looking at the following plot:



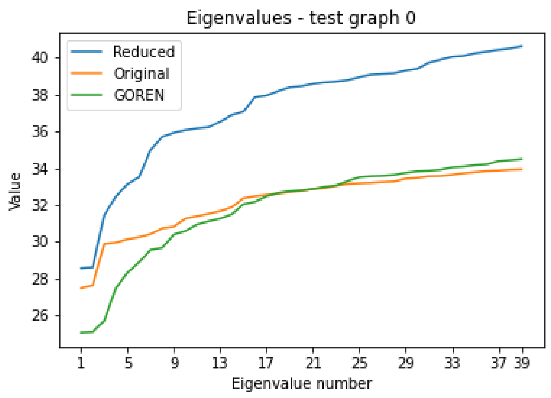


(a) Validation improvements versus validation losses cropped at 20 (b) Validation improvements versus validation losses uncropped

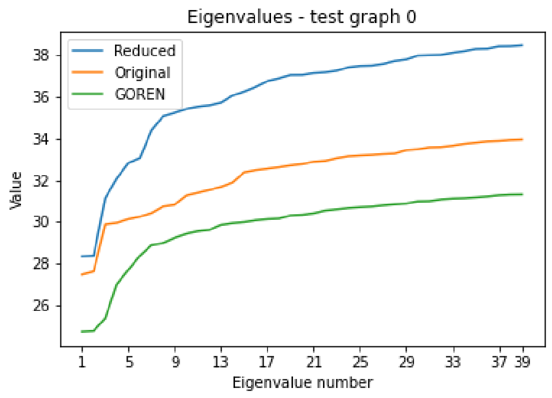
Figure 6: Validation losses vs negative improvements. The same plots for all the other graphs can be found in APPENDIX A

In these two graphs we can see the validation losses vs negative improvements. By default these two functions should have opposite slope to be good, for the sake of visualization the improvements are negative such that they must have the same slope to be good. The green circle highlight when it is not happening. As we can see whenever the loss is lower than 25, the improvement behaves strangely. Net of our mistakes, this could mean that the minimum of the loss function does not correspond to the one of the improvement function, at least the local one. In order to make this result statistically solid one should run this experiment many times and more importantly on different types of dataset. This could also be significant in explaining why the improvement is so oscillating.

Going further with the analysis, it remains plausible a mistake from our side in the implementation of the network, so we decided to plot all the eigen values in order to visualize better if they look more aligned with the original graph:



(a) Combinatorial Laplacian for both



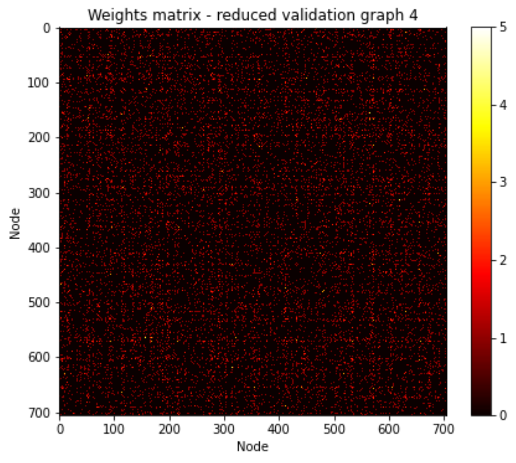
(b) Combinatorial Laplacian for the original graph and doubly-weighted Laplacian for reduced graph

Figure 7: Eigenvalues of the graph and reduced graph of the first test graph

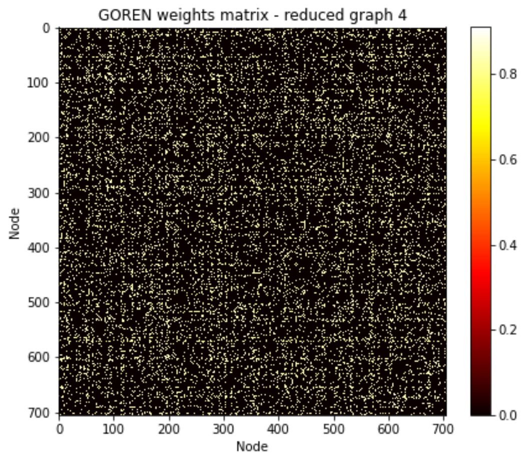
Using the combinatorial Laplacian for both graphs, as shown in figure 7a, the results are impressive. The best average improvement on the test set is 87%, but since we want to directly compare the result with the one provided by the authors, in figure 7b we show the eigen values of the original graph combinatorial Laplacian and the reduced graph doubly-weighted Laplacian, and surprisingly we get an average improvement of 60% while the authors claim a 1% for this specific dataset.

The very good alignment of the eigenvalues reduces the probability of having a bug in our GOREN implementation since the computation of the Laplacian and the eigen values is pretty straight forward and indipendent from the network (once the matrix $\widehat{W}$ is computed). Still from our side there could be a misinterpretation in the way the process they Laplacian before doing the eigen analysis.

One more thing to check is the heatmap describing the values of $\widehat{W}$ before and after training:



(a) Min=1; Max=5; Std=0.32



(b) Min=0.8113; Max=0.9095; Std=0.0111

Figure 8: Respectively Heatmap of $\widehat{W}$ and Heatmap of $\widehat{W_t}$ (predicted)

The previous plot is interesting since it seems that all values converge to just a specific one. In fact, the standard deviation is $std = 0.0111$, essentially zero. We wonder what would happen if we started with values already one as weights of the reduced graph edges. In the following plot we compare the eigen values of three Laplacians. The first one is computed starting from the adjacency matrix (weight = 1) of the reduce graph, the second one from the weights matrix obtained with the given heuristic and, finally, the third one from the original graph weights matrix.

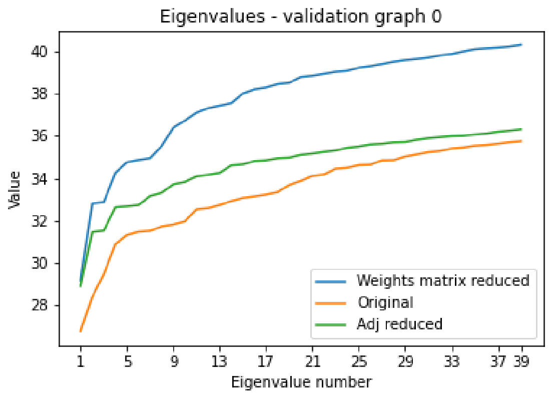
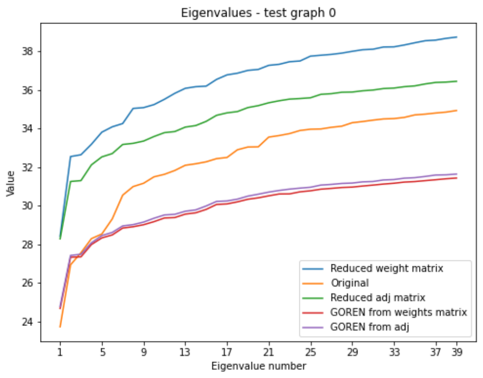


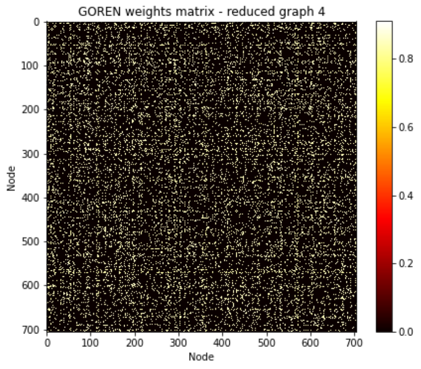
Figure 9: Eigen values of the adjacency matrix of the reduced graph compared with the weights matrix (Doubly weighted Laplacian used). All the plots for other graphs can be found in APPENDIX B

Unexpectedly, this new weights matrix is already better. It results in an improvement higher than 1% obtained by the authors using the network. In order to explore this possibility one should do the same plot and compute the improvement for all the other datasets proposed by the authors and see if it is a pattern or just a random case that appeared in our specific dataset. Furthermore, note that the same reasoning is also true if we use the combinatorial Laplacian for both original graph and reduced graph.

Finally, in order to complete our set of test we decided to train the same model starting from the weights matrix of reduced graph initialized with weights = 1.



(a) Eigen values of the two GOREN trained and also the original and starting reduced graphs.



(b) Prediction of the model on validation graph 4. Min=0.8186; Max=0.9067; Std=0.0088

Figure 10: Only the plot for test graph zero is provided but all the others are equivalent.

Note that the average improvement on the test set, in the second model is 28%, however, the final result is completely comparable to the previous trained model. Since the final results are equivalent, and it is not said that this approach can be generalized to other types of graphs, we cannot recommend to always start with the adjacency matrix. If in our case, our model is learning exactly the same thing, we guess also the author's model should remain the same, but if the adjacency matrix improve the initial reduction more than 1%, we wonder if their model is really learning.

7.1 Matrix Scaling

Since we checked that using an adjacency matrix instead of a weights matrix obtained by the coarsening algorithm gives us better results, we run some experiments in order to check what happens if we scale the adjacency matrix of a scalar c. Indeed, the equation 3 tells us that `GOREN` always outputs values bigger or equal to one (with the LeakyReLU this constraint is relaxed a bit, but still is difficult to go below such threshold). A natural question is whether we can achieve better results by lowering the values. We know that the definition of eigenvalues is the following.

$$Av = \lambda v \tag{8}$$

This means that if we multiply A by a constant c we have:

$$cAv = \lambda cv \tag{9}$$

In other words, when we scale A by a factor c we are scaling also the eigenvalues of A by the same factor.
In Figure 11 we plot the first 40 eigenvalues (except the first one because it was 0 in all the cases) of the Laplacian in the case of: original graph, the coarse graph using weights, the coarse graph using the adjacency matrix, the coarse graph using the adjacency matrix scaled by a factor of 0.85.

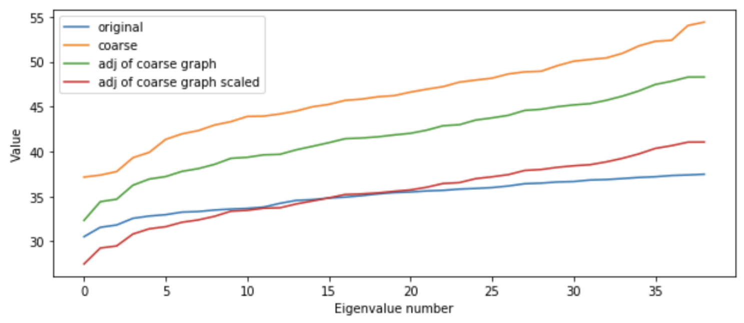


Figure 11: Caption

We can notice that the eigenerror (the error in the alignment of the eigenvalues) of the coarse graph using weights matrix is bigger than the one of the coarse graph using the adjacency matrix.
However, is also clear that by reducing of a factor of 0.85 the values of the adjacency matrix (i.e. scaling all the edges from 1 to 0.85) we are able to improve the result. If we continue to reduce the scaling factor we will worsening the alignment of the eigenvalues.
Since the eigenvalues of the Laplacian predicted by the `GOREN` are already well aligned with the eigenvalues of the initial graph, we cannot improve the results by just scaling the adjacency matrix.

8 Conclusion

We can say that the replication trial has not been successfully accomplished. However, the result we get, on this specific dataset, are much better of the authors claims. Clearly this may be due to our mistakes, even though it is important to notice that the evaluation of the eigen error is something that has nothing to do with the model, so if there is some mistake, it should be related either in the way we compute the Laplacian or the way we interpreted the authors suggestions. Assuming there is no error from our side, we could say that the eigenvalues of the combinatorial Laplacian are very aligned to the original ones, and this is a huge improvement from our point of view. To make this work more solid we aim to replicate the same test multiple times and on all datasets used.

References

[1] Chen Cai, Dingkang Wang, Yusu Wang. Graph coarsening with neural networks. *arXiv:2102.01350*, 2021.

[2] Ruge J. W., Stüben K. Algebraic Multigrid. 1987

[3] Spielman Daniel A., Srivastava Nikhil. Graph Sparsification by Effective Resistances. *arXiv:0803.0929* 2008

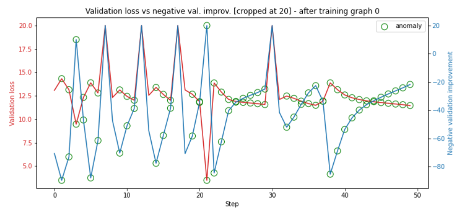
[4] Loukas Andreas. Graph Reduction with Spectral and Cut Guarantees. *arXiv:1808.10650*, 2019.

[5] Loukas Andreas, Vandergheynst Pierre. Spectrally approximating large graphs with smaller graphs. *arXiv:1802.07510*, 2018.

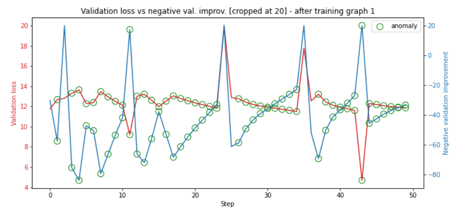
[6] Chen Cai, Yusu Wang. A simple yet effective baseline for non-attribute graph classification. *arXiv:1811.03508*, 2018.

[7] Keyulu Xu, Weihua Hu, Jure Leskovec, Stefanie Jegelka. How Powerful are Graph Neural Networks?. *arXiv:1810.00826*, 2019.

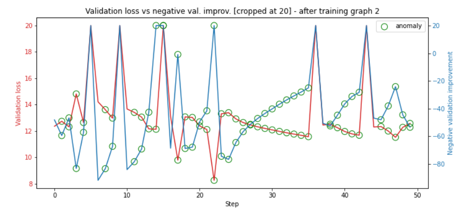
9 APPENDIX A



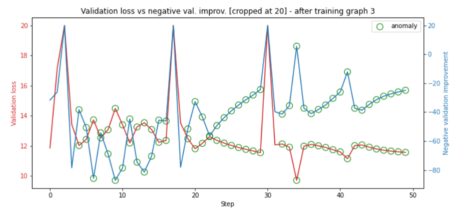
(a) After training graph 0 - cropped at 20



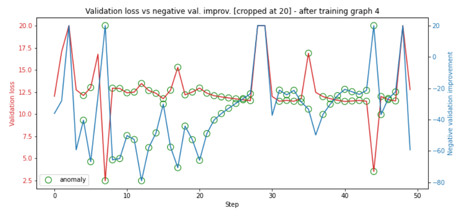
(b) After training graph 1 - cropped at 20



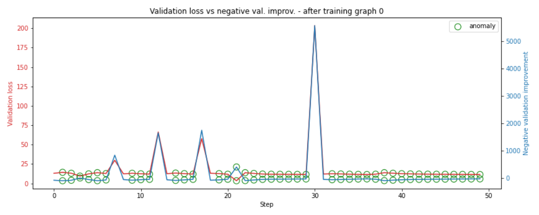
(c) After training graph 2 - cropped at 20



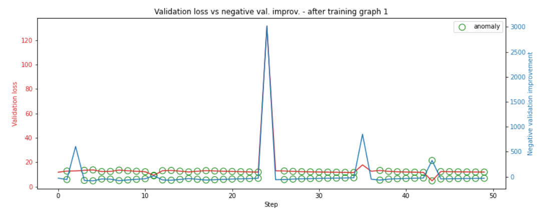
(d) After training graph 3 - cropped at 20



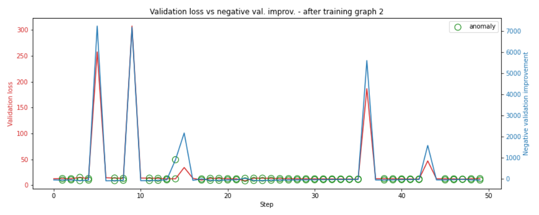
(e) After training graph 4 - cropped at 20



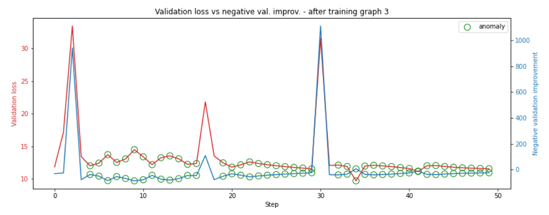
(f) After training graph 0 - uncropped



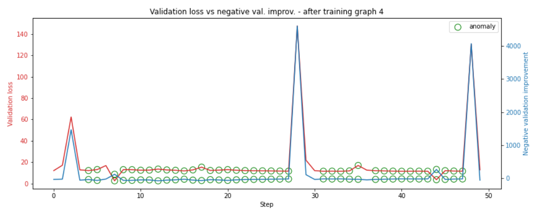
(g) After training graph 1 - uncropped



(h) After training graph 2 - uncropped



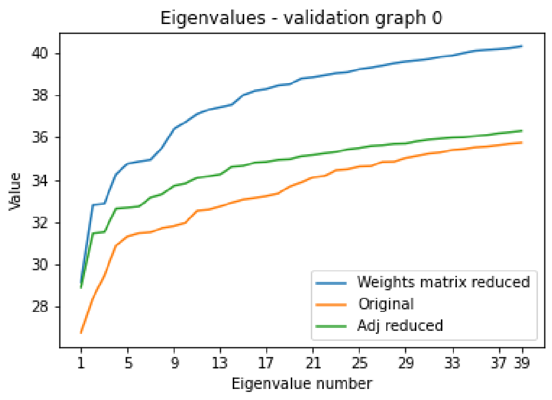
(i) After training graph 3 - uncropped



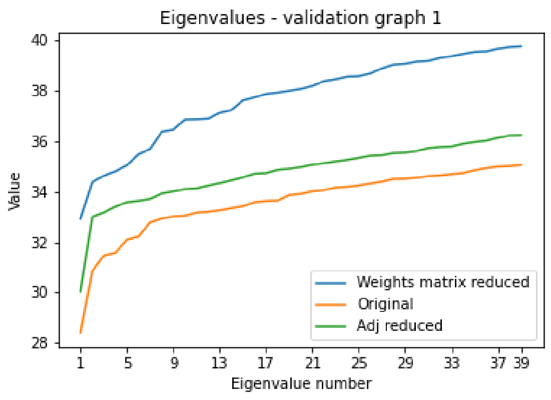
(j) After training graph 4 - uncropped

Figure 12: Validation improvements vs validation losses after training each graph

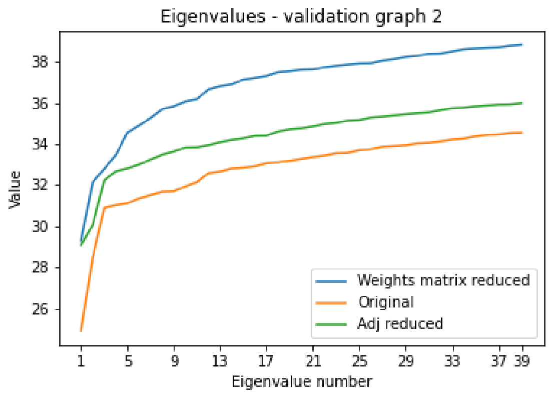
10 APPENDIX B



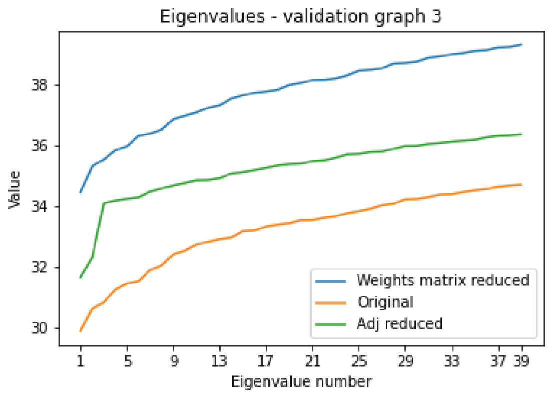
(a) Validation graph 0



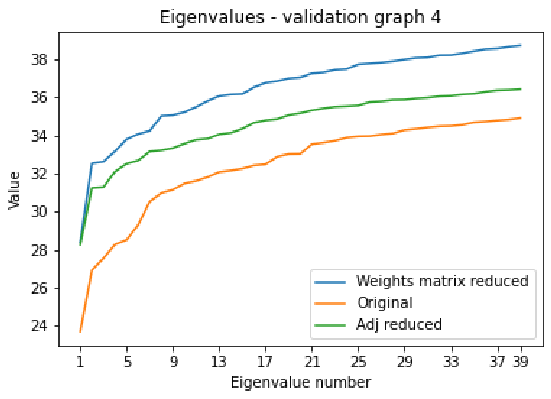
(b) Validation graph 1



(c) Validation graph 2



(d) Validation graph 3



(e) Validation graph 4

Figure 13: Eigen values of the reduced graph computed starting from the adjacency matrix (weights = 1) compared to the original graph and the original weights computed by the heuristic (Doubly weighted Laplacian for the reduced graph and combinatorial for the original graph.