Compare DeepGCN, APPNP and MixHop Models on Github Dataset (CS886)

Allen Zhang[1]

Abstract

Graph neural networks (GNNs) have seen a lot of recent developments and applications in the real world, such as social networks, organic chemistry, and recommendation engines. One of the area of focus has been training GNNs that are deeper and thus more powerful. This paper aims to compare DeepGCN, APPNP, and MixHop models which were designed to allow GNNs to be deeper, and test their performance on a node classification task. Specifically, the Github dataset is used, which contains Web and Machine Learning (ML) developers on Github. The task for the models is to predict whether a developer is web based or ML based, and there is some class imbalance. The main finding is that all three models perform similarly on this dataset, with DeepGCN using max-relative convolution having an edge in terms of prediction speed.

1. Introduction

Graph Neural Networks (GNNs) are neural networks that focus on graph-related problems. One type of graph problem is node classification where GNNs are trained to predict which class each node belongs (Scarselli et al., 2009). A problem when using GNNs on node classification is that deeper models do not guarantee a better prediction, and deeper here means the models have a lot of iterations or layers. The basic versions of GNNs suffer from the over-smoothing problem caused by vanishing gradients in back propagation and often resulting in reaching the same value for all nodes (Li et al., 2018). There have been a lot of different approaches trying to tackle this over-smoothing problem.

This paper intends to compare three different approaches on a node classification task and contrast the different results. The three models are DeepGCN, APPNP, and MixHop which have all been shown to not suffer from the vanishing gradients problem. We compare these three models on the same dataset and test if the results can be replicated. We used a Github dataset with 2 classes and trained all three models with some hyperparameter tuning. We found that all three models perform similarly on this dataset with around 86.7% prediction accuracy given their best model configuration. The prediction time of all models are relatively fast, and they scale linearly with depth and width of the model, with the exception of DeepGCN models using max-relative convolution only scales with depth of the model.

[1]University of Waterloo, Waterloo, Ontario, Canada. Correspondence to: Allen Zhang <allen.zhang@uwaterloo.ca>.

Submitted as part of the final project for CS886 in Winter term of 2024. Do not copy or distribute.

2. Models Tested

We tested the DeepGCN (Li et al., 2019), APPNP (Gasteiger et al., 2019) and MixHop (Abu-El-Haija et al., 2019) models in this paper. We will briefly describe the models in this section and present their strengths and weaknesses.

2.1. DeepGCN

DeepGCN uses two key ideas to prevent over-smoothing. One of them is residual connections, where the output at current layer is added to the current graph to form the input for the next layer, instead of using the output of the current layer as the input for the next layer directly.

The other idea is using a dilated k-Nearest Neighbour (k-NN) function for dynamic edges. At the dth layer, the dilation rate is also d. The model uses k-NN to find the k nearest neighbors in the d-dilated neighborhood, a new idea introduced by Li et al. Specifically, for any node v, we find $k \times d$ number of nearest neighbours of v and sort them by distance, say they are $u_1, u_2, ..., u_{k\times d}$ in the ascending order by distance, then the d-dilated neighborhood of v is $\{u_1, u_{1+d}, u_{1+2d}, ..., u_{1+(k-1)d}\}$.

These two ideas allowed DeepGCN to eliminate the over-smoothing problem. When the model is deeper with more layers, the performance does not suffer and can be stably trained (Li et al., 2019).

ResGCN and DenseGCN are two sub-types of DeepGCN. The main difference is that ResGCN only uses the previous

layer's outputs for calculations in the current layer, while DenseGCN uses all layers' outputs before the current layer for calculation in the current layer. This paper focuses on testing ResGCN as it generally performs better than DenseGCN, and it is less computationally intensive (Li et al., 2019). The paper will use DeepGCN to refer to ResGCN in order not to create confusion.

2.2. APPNP

APPNP stands for approximated personalized propagation of neural predictions. This model utilizes ideas from both personalized PageRank and Graph Convolution Networks. It reduces the computational complexity to linear by approximating topic-sensitive PageRank (Gasteiger et al., 2019). Specifically, it achieves this by power iterations, where at each iteration, there is a restart probability that is used to calculate a weighted sum of the original prediction probability from a neural network and the output of the current layer, and this weighted sum is used in the next layer. The iterative equations by Gasteiger et al. are presented below:

$$Z^{(0)} = H = f_\theta(X), \tag{1}$$

$$Z^{(k+1)} = (1-\alpha)\hat{\tilde{A}}Z^{(k)} + \alpha H, \tag{2}$$

$$Z^{(K)} = \text{softmax}\left((1-\alpha)\hat{\tilde{A}}Z^{(k)} + \alpha H\right) \tag{3}$$

where $X \in \mathbb{R}^{n\times f}$ is the feature matrix, f_θ is a neural network with parameter set θ, $H \in \mathbb{R}^{n\times c}$ is the prediction, $\alpha \in [0,1]$ is a constant that $\hat{\tilde{A}} \in \mathbb{R}^{n\times n}$ is the adjacency matrix with added self-loops and matrix normalization, $Z^{(k)}$ is the output at the kth iteration, and K is the total number of power iterations (with n being the number of nodes, c being the number of classes, and f is the number of features, $k \in [0, K-2]$).

2.3. MixHop

MixHop uses the idea of neighborhood mixing. In each layer, it can use each set of zero-hop, one-hop, and two-hop neighbors of each vertex inside each convolution layer. If the model is using only zero-hop and one-hop neighbors, we say the power of this model equals 1. If the model uses zero-hop, one-hop, and two-hop neighbors, we say the power of this model equals 2. Each different set of i-hop neighbors uses a different number of filters and all of the different outputs are combined to become the input for the next layer. See the Figure 1 for an illustration of this idea.

2.4. Similarity and Differences

Mixhop's mixing neighborhood is somewhat similar to the dilated k-NN method in DeepGCN where we want to expand the neighborhood used in graph convolution in each layer. However, since MixHop only goes as far as two-hop, it does not consider neighbors as far as DeepGCN can consider. On the other hand, an advantage maybe that all of the closer neighbours are considered, instead of DeepGCN having to skip some dilated neighbors to avoid the neighbor expansion problems. APPNP does not use larger neighborhoods at all. The convolution only involves the adjacency matrix and message passing between neighboring nodes.

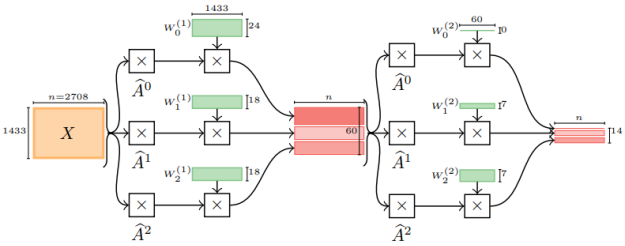

Figure 1. Example of network architecture illustrated by Abu-El-Haija et al. Input X is fed into a MixHop layer. Then, for each i-hop neighbors, there is an adjacency matrix $\hat{A}^i$ being used that is multiplied to X. Then weights $w_i^(j)$ are also multiplied to it, where j is the index of the layer. The three results from the different adjacency matrix and weights are concatenated together to form the output latent features. The number of output latent features depends on each trainable weight.

On the other hand, both MixHop and DeepGCN use the idea of inputting the current graph to the next layer to avoid the problem of diminishing gradients. APPNP slightly differs by using a weighted approach and can partially keep the values of the initial latent graph representation. It uses a constant multiplier α such that after each iteration, the new latent graph representation is equal to the original latent graph features weighted by α and summed with the calculated latent graph based on the previous layer weighted by $1-\alpha$.

Given these similarities and differences, contrasting how these three different models would perform on a real-world dataset can enable a better understanding of how to utilize the models better.

3. Experiment

The purpose of the experiment is to compare these three models on a real-world dataset. We would consider the two metrics for the comparison. The first is prediction accuracy, where the models will be tested on unseen data and see if they can make the correct predictions. We also use the micro-F1 score along with classification accuracy, as micro-F1 score is a better indication of classification accuracy when there is class imbalance in the data set. Equation (4) shows how to calculate the F1 score, where TP is the number of true positive, FP is the number of false positives,

and FN is the number of false negatives. The second is how long these models take to make predictions in milliseconds. This metric is useful for using the models in a real-world application that might need instant predictions. If a model can predict quickly, it will allow the application that uses it to be faster and smoother. Another important factor is that if the model takes too long to generate predictions, the model architecture may be inefficient and would not scale well.

$$\text{F1} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FP} + \text{FN}} \tag{4}$$

The real-world dataset invoked is called the Github dataset. It has 37,300 nodes, 578,006 edges, 128 node features, and 2 classes. It does not have edge weights. The graph is sparse considering the amount of nodes and edges, as the fully connected graph will have more than 1 trillion edges. The graph represents a social network with developers on Github as nodes and mutual follower relationships between them as edges. Some node features include locations, starred repositories, current employer, and email address. The labels are binary, where each node is either a web developer or a machine learning developer.

This dataset creates three challenges for these three models. The first is that the graph is sparse with a lot of nodes, and GCN often perform poorly on such graphs. The second is that each node has 128 features which is not a small amount of features either, so the ability to use all the information in these features is crucial. The third is that there are some class imbalances in the data, where the approximate ratio between the classes is $1:4$. These three challenges reflect real-world datasets, and thus the dataset is useful when testing these three models.

To train the model, we use transductive learning, where all of the nodes and edges in the original graph are given to the model and only the labels of nodes in the training data are known to the model. To split the data into train, validation, and test set, we opted for a random split so that the percentage of train, validation, and test set comprises of 50%, 20%, and 30% of the nodes in the Github dataset. The models are supervised using the training set, where the labels are known to the model and used to calculate losses during training. The validation set is used to help the model to not overfit to the training data, and provide a reference for early stopping so that the optimal model is saved and used on the test data to gather the different metrics.

We have also tested different hyperparameters in this paper. For DeepGCN, we focused on four tunable parameters: the number of filters, the number of layers, the type of convolution layer used, and the dropout rate between the layers. The number of filters used were 32, 64, and 128. The number of layers used were 4, 16, and 32. Two types of convolution were used: max-relative convolution layer (Li et al., 2019) and graph attention networks (Veličković et al., 2018). The dropout rates experimented with were 0.2 and 0.

For APPNP, we focused on four tunable parameters as well: the number of hidden features, the number of iteration, α, and dropout. The number of hidden features used were 32, 64, and 128. The number of iterations used were 4, 16, abd32. We tested α equals to 0.2 or 0.05. We experimented with dropout rates of 0.2 and 0.

Algorithm 1 Training and Testing Model pipeline

Input: graph $G = (V, E)$, node label y, model configs c, total number of epochs e, patience p
Split nodes to train set R, val set S, and test set T
Node labels are split to y_R, y_S, and y_T corresponding to which set the nodes are in
for model config m **in** c **do**
  Initialize model M based on m Initialize best validation f1-score $f = 0$ Initialize counter for patience $j = 0$
  for epoch $i = 1$ **to** e **do**
    Train M with G, but only given y_R to perform backpropagation
    Calculate F1-score on both R and S with current M, and store it as f_{iR} and f_{iS}
    if $f_{iS} < f$ **then**
      $f = f_{iS}$
      save the current best model M
      $j = 0$
    end if$j = j + 1$
    if $j > p$ **then**
      break loop
    end if
  end for
  Load saved best model as M
  Calculate F1-score, accuracy score, and prediction time on T with M and store the results for config m
end for

For MixHop, we focused on three tunable parameters: the number of hidden features, the number of layers, and power. The number of layers used were 4, 16, and 32. The parameter power is the i for the i-hop neighborhood the model uses in each layer, where 1 and 2 are the two choices. Since there is a equal amount of hidden features being combined at each level, the total number of hidden features needed to be divisible by the number of hops used. When power equals to 1, the 1-hop and 0-hop features are combined, the number of hops used is 2 and so we choose hidden features to be 32, 64, and 128. When power equals to 2, the 2-hop, 1-hop, and 0-hop features are combined, the number of hops used is 3, and so we choose the total number of hidden features to be close to the previous case, thus 33, 63, and 129.

All models used the same Adam optimizer (Kingma & Ba, 2015) with adaptive learning rate that reduces on plateau,

and the loss function used is binary cross entropy with logits loss.

The hyperparameters were chosen for mainly two reasons. The first is the ease of comparison between the models. The number of hidden features to be the same amongst all three models except when the power is 2 for MixHop; Even when the power is 2, the number of hidden features are only off by 1, allowing us to compare how all the models behave when there is an increase or decrease in the number of hidden features. In addition, the number of layers are chosen to be the same amongst all three models as well, to see how the model depth affects the performance, and a fair comparison between the models can be made.

The second is the limitation of the hardware. We were unable to get big enough memory to train deeper models with more layers and wider models with more hidden features. However, since we tested all possible combinations of model hyperparameters employed, we will be able to make useful generalizations when analyzing the trend of the results.

The actual training and testing pipeline for each model is summarized in the algorithm in Algorithm 1.

4. Results

We now present our experiment results. We first will compare the F1-score of the different models given different hyperparameters. The micro-F1 score is better than the accuracy score for this dataset because there is class imbalance, and the F1-score provides a better representation of how well the model predicts than the accuracy score. Figure 2 illustrates how the depth of the model (the number of iterations or layers) affects the model's micro F1-score given the best model with that depth. Figure 3 illustrates how the width of the model (the number of hidden features) affects the model's F1-score given the best model with that number of features. The scores are quite close but we can still see some insights.

In Figure 2, we see that for DeepGCN, the model can be better when the depth increases. This is in line with claims by Li et al where the DeepGCN model does not suffer from added depth to the model, hence enabling a workable deep graph neural network. On the other hand, MixHop drops in performance when there is depth added. However, it performs the best out of all tested model configurations when the depth is only 4, meaning it's a model that does not need to be deep to perform well, and it may not be the ideal architecture for a deep model. APPNP's results show that its architecture lies between the two, where its performance can increase when depth is added, but only to a certain point before depth is not helpful to the performance anymore.

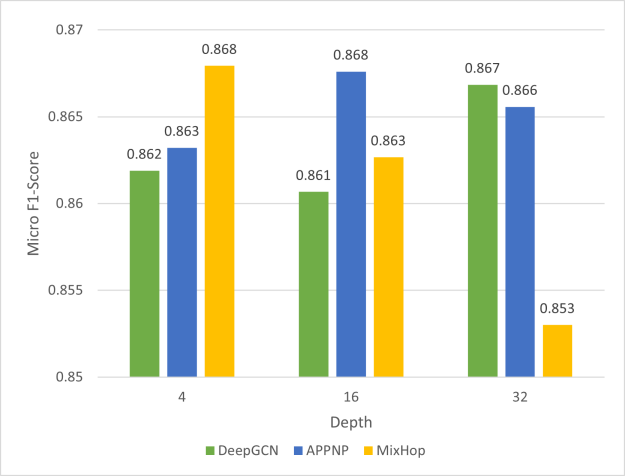

Figure 2. Best micro F1-score of different models with the same depth compared. DeepGCN best scores are in green, APPNP best scores are in blue, and MixHop best scores are in yellow.

In Figure 3, we can observe that MixHop obtains the best results when the model is wider. In contrast, APPNP does not require the width to be as high, and the lowest widths result in the best performance. DeepGCN, on the other hand, requires a moderate amount of width to balance between model complexity and generalizability.

So in terms of micro F1-score, the best model for DeepGCN is with 64 hidden features, 32 layers, graph attention networks for convolution, and a dropout rate equals to 0.2, which achieves a score of 0.8668. The best model for APPNP is with 32 hidden features, 16 iterations, an alpha equals to 0.2, and a dropout rate equals to 0, which achieves a score of 0.8676. The best model for MixHop is with 128 hidden features, 4 layers, power equals 2 (only 0 and 1 hop used), which achieves a score of 0.8679.

In Figure 4, Figure 5, and Figure 6, we present the training loss and validation loss from the different models. We see that DeepGCN's train loss and val loss diverge quickly, showing that the model is overfitting, but with early stopping, the best model before overfitting is found. The overfitting is mainly caused by DeepGCN being a more complicated models than the other two and has the most layers and thus parameters. MixHop and APPNP showed similar convergence of training and validation loss. This is because the models have less layers or iterations and thus have less variance.

In addition, we also compare the prediction time of the different models with different hyperparameters. In Table 1 and , we can see that for DeepGCN models, the number of layers and the prediction time have a linear relationship. We can see that the number of hidden features and the prediction

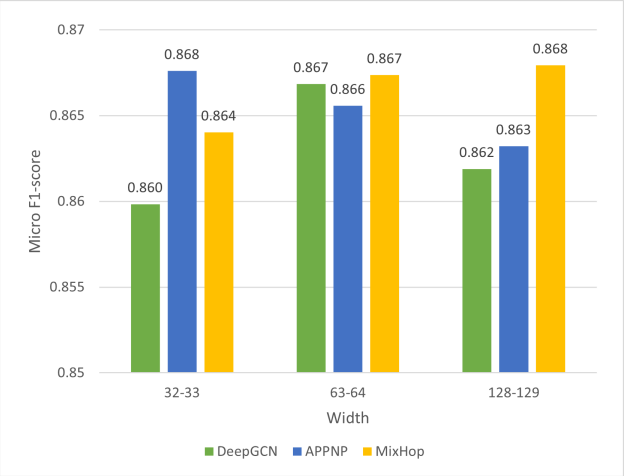

Figure 3. Best micro F1-score of different models with the same width compared. DeepGCN best scores are in green, APPNP best scores are in blue, and MixHop best scores are in yellow.

time do not correlate when using max-relative convolution, and have a linear relationship when using graph attention networks.

In Table 3, we can see that the times are very similar to DeepGCN with graph attention networks. The prediction time is increase linearly when number of iteration increase, and it increase linearly when the number of hidden features increase. MixHop time results are in Table 6 in the appendix, but the general ideas are the same. The prediction time increases linearly with an increase in number of layers, increases linearly with an increase in hidden features, and also increases linearly with the number of hops used.

Table 1. Comparing the prediction time of DeepGCN given that max-relative convolution is used.

	NUM OF HIDDEN FEATURES		
NUM LAYERS	32	64	128
4	0.00058	0.00057	0.00059
16	0.00157	0.00157	0.00168
32	0.00293	0.00302	0.00295

Table 2. Comparing the prediction time of DeepGCN given that graph attention networks is used.

	NUM OF HIDDEN FEATURES		
NUM LAYERS	32	64	128
4	0.00258	0.00413	0.00666
16	0.01097	0.01776	0.03115
32	0.02255	0.03915	0.06457

In terms of prediction time, if we require a model that is

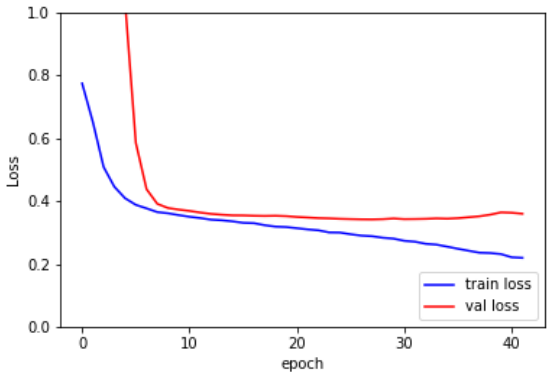

Figure 4. Training and validation loss of the DeepGCN model with best micro F1-score on test set

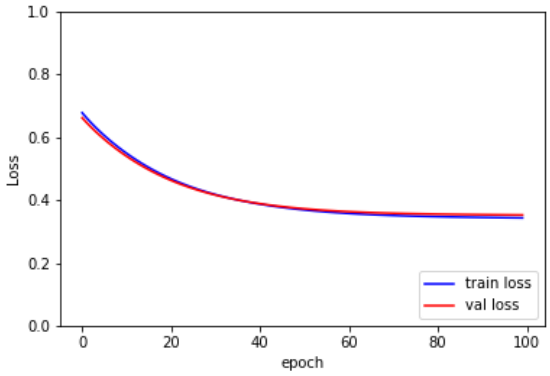

Figure 5. Training and validation loss of the APPNP model with best micro F1-score on test set

can generate prediction relatively fast with good prediction accuracy, DeepGCN with max-relative convolution seems to be the best choice. Suppose the the number of layers is l, the number of hidden layers is h. The model prediction scales linearly only with l, compared to the prediction time of APPNP and MixHop which scales linearly with $l \times h$. Power is at most 2 for MixHop so it can be considered as a constant.

All micro F1-score, accuracy, and prediction time results for the different model configurations of DeepGCN, APPNP and MixHop can be found in Table 4, Table 5, and Table 6 in the appendix. There were three interesting observations overall:

- APPNP models with a dropout rate of 0.2 and a high number of iterations suffer from vanishing gradient, where the F1-score is almost 0, and the models were

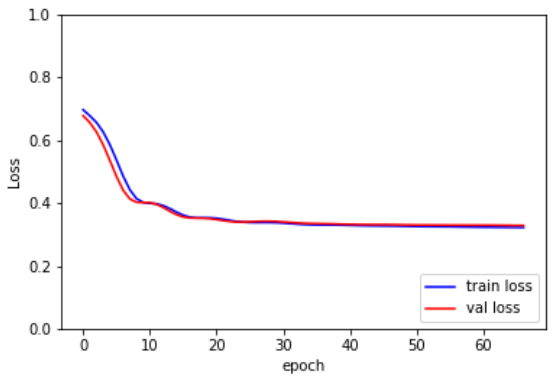

Figure 6. Training and validation loss of the MixHop model with best micro F1-score on test set

Table 3. Comparing the prediction time of APPNP.

	NUM OF HIDDEN FEATURES		
NUM ITERATIONS	32	64	128
4	0.00264	0.00478	0.00970
16	0.00803	0.01530	0.03082
32	0.01581	0.03006	0.06185

predicting only 1 class. These models cause exploding gradients and losses, which could be because of the message lost from dropouts.

- The MixHop model F1-score drops with an increase in the number of layers, meaning that it cannot be a very deep model. There was also no notable difference between setting power equals 1 versus 2 on this dataset, so the effectiveness of adding 2-hop neighbors is questionable and requires further investigation.
- DeepGCN showed consistency in training and testing with all hyperparameters tested where deeper models can be trained with good performance, although the difference in F1-score for this dataset is small as well.

5. Conclusion

To summarize the findings, we had 3 key observations. Firstly, all three models performed similarly on this dataset with only a little bit of variations given the their respective hyperparameters. Secondly, DeepGCN performs better when the model is deeper, while APPNP performs better when the depth is moderate, and MixHop performs better when there is less depth. Finally, the prediction time of all three models scales with depth linearly. The prediction time of APPPNP and MixHop scales with width linearly. On the other hand, DeepGCN has choices of types of convolution layers, and the prediction time when using max-relative convolution does not scale with width, while using graph attention networks scale with width linearly. Given the time constraint, the models were tested on only one dataset, so it can be interesting to see if the trends are replicated on other datasets as well. The depth of these models can be further investigated to figure out the best number of layers or iterations for these three models. It is also noted that the difference in performance amongst the models seems to be small, so better model architectures may be required to increase prediction accuracy.

Software and Data

In the experiment, overall code is Python based. We mostly used the PyTorch and Pytorch Geometric library for developing the models. Assisting libraries include Pandas, Numpy, and Matplotlib for table creation, data processing, and plotting. The DeepGCN architecture is borrowed from the GitHub repository maintained by the original authors of the paper: https://github.com/lightaime/deep_gcns_torch. The data is provided in Pytorch Geometric. Hardware used to train the model is Nvidia RTX 4090 with 24GB VRAM and Intel i9-13900KF with 32 threads running on Ubuntu Linux system. The project code is uploaded to the following GitHub repository: https://github.com/physics2001/CS886-project

References

Abu-El-Haija, S., Perozzi, B., Kapoor, A., Alipourfard, N., Lerman, K., Harutyunyan, H., Steeg, G. V., and Galstyan, A. MixHop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 21–29. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/abu-el-haija19a.html.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Gasteiger, J., Bojchevski, A., and Günnemann, S. Combining neural networks with personalized pagerank for classification on graphs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gL-2A9Ym.

Haveliwala, T. H. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge & Data Engineering*, 15(04):784–796, jul 2003. ISSN 1558-2191. doi: 10.1109/TKDE.2003.1208999.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *ICLR (Poster)*, 2015. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2015.html#KingmaB14.

Li, G., Müller, M., Thabet, A., and Ghanem, B. Deepgcns: Can gcns go as deep as cnns? In *The IEEE International Conference on Computer Vision (ICCV)*, 2019.

Li, Q., Han, Z., and Wu, X.-M. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018. ISBN 978-1-57735-800-8.

Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking : Bringing order to the web. In *The Web Conference*, 1999. URL https://api.semanticscholar.org/CorpusID:1508503.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Rozemberczki, B., Allen, C., and Sarkar, R. Multi-Scale attributed node embedding. *Journal of Complex Networks*, 9(2):cnab014, 05 2021. ISSN 2051-1329. doi: 10.1093/comnet/cnab014. URL https://doi.org/10.1093/comnet/cnab014.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJXMpikCZ. accepted as poster.

Veličković, P., Buesing, L., Overlan, M., Pascanu, R., Vinyals, O., and Blundell, C. Pointer graph networks. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 2232–2244. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/176bf6219855a6eb1f3a30903e34b6fb-Paper.pdf.

A. Appendix 1

Table 4. All hyperparmeters tuning results for DeepGCN model.

HIDDEN_FEATURES	NUM_LAYERS	CONV	DROPOUT	F1_SCORE	ACCURACY	PREDICTION TIME
32	4	MR	0.2	0.85597	0.85765	0.000570
32	4	MR	0	0.85595	0.85561	0.000582
32	4	GAT	0.2	0.85984	0.86039	0.002587
32	4	GAT	0	0.85189	0.85270	0.002578
32	16	MR	0.2	0.85514	0.85526	0.001578
32	16	MR	0	0.85543	0.85738	0.001570
32	16	GAT	0.2	0.84340	0.84677	0.010650
32	16	GAT	0	0.85789	0.85809	0.011285
32	32	MR	0.2	0.85596	0.85393	0.002903
32	32	MR	0	0.84959	0.85225	0.002960
32	32	GAT	0.2	0.85842	0.85827	0.021651
32	32	GAT	0	0.85810	0.85738	0.023454
64	4	MR	0.2	0.85462	0.85703	0.000545
64	4	MR	0	0.85129	0.85270	0.000586
64	4	GAT	0.2	0.85424	0.85393	0.004305
64	4	GAT	0	0.85780	0.85844	0.003959
64	16	MR	0.2	0.85682	0.85615	0.001543
64	16	MR	0	0.85709	0.85668	0.001591
64	16	GAT	0.2	0.82240	0.82166	0.017860
64	16	GAT	0	0.85819	0.85853	0.017663
64	32	MR	0.2	0.85384	0.85332	0.002977
64	32	MR	0	0.85497	0.85402	0.003067
64	32	GAT	0.2	0.86684	0.86684	0.036337
64	32	GAT	0	0.84917	0.84916	0.041963
128	4	MR	0.2	0.86189	0.86463	0.000627
128	4	MR	0	0.85518	0.85570	0.000553
128	4	GAT	0.2	0.83128	0.82944	0.006662
128	4	GAT	0	0.86023	0.86065	0.006665
128	16	MR	0.2	0.84354	0.84279	0.001655
128	16	MR	0	0.85175	0.85137	0.001696
128	16	GAT	0.2	0.86019	0.86012	0.031148
128	16	GAT	0	0.86067	0.86136	0.031143
128	32	MR	0.2	0.86039	0.85915	0.002942
128	32	MR	0	0.85976	0.85738	0.002959
128	32	GAT	0.2	0.86183	0.86136	0.065297
128	32	GAT	0	0.86034	0.86074	0.063841

Table 5. All hyperparmeters tuning results for APPNP model.

HIDDEN_FEATURE	NUM_ITERATION	ALPHA	DROPOUT	F1_SCORE	ACCURACY	PREDICTION TIME
32	4	0.2	0.2	0.86204	0.86251	0.00264
32	4	0.2	0	0.86000	0.86004	0.00265
32	4	0.05	0.2	0.85578	0.85570	0.00265
32	4	0.05	0	0.85690	0.85721	0.00263
32	16	0.2	0.2	0.84193	0.84562	0.00818
32	16	0.2	0	0.86761	0.86720	0.00781
32	16	0.05	0.2	0.83962	0.84598	0.00828
32	16	0.05	0	0.80105	0.81874	0.00784
32	32	0.2	0.2	0.00000	0.73943	0.01548
32	32	0.2	0	0.86110	0.86154	0.01604
32	32	0.05	0.2	0.00000	0.74156	0.01541
32	32	0.05	0	0.85732	0.85915	0.01631
64	4	0.2	0.2	0.86119	0.86065	0.00477
64	4	0.2	0	0.86298	0.86251	0.00479
64	4	0.05	0.2	0.85929	0.85906	0.00478
64	4	0.05	0	0.85837	0.85871	0.00479
64	16	0.2	0.2	0.85510	0.86127	0.01521
64	16	0.2	0	0.86282	0.86295	0.01528
64	16	0.05	0.2	0.82800	0.82599	0.01555
64	16	0.05	0	0.85960	0.85906	0.01514
64	32	0.2	0.2	0.00000	0.74191	0.02928
64	32	0.2	0	0.86336	0.86375	0.03134
64	32	0.05	0.2	0.00000	0.73996	0.03058
64	32	0.05	0	0.86557	0.86676	0.02904
128	4	0.2	0.2	0.85943	0.85968	0.00930
128	4	0.2	0	0.86321	0.86278	0.00980
128	4	0.05	0.2	0.86269	0.86295	0.00991
128	4	0.05	0	0.86179	0.86207	0.00981
128	16	0.2	0.2	0.84304	0.84235	0.03183
128	16	0.2	0	0.85906	0.85924	0.02964
128	16	0.05	0.2	0.84401	0.84713	0.02968
128	16	0.05	0	0.81057	0.82219	0.03212
128	32	0.2	0.2	0.00000	0.73882	0.06238
128	32	0.2	0	0.86072	0.86039	0.06163
128	32	0.05	0.2	0.00000	0.74103	0.06028
128	32	0.05	0	0.79773	0.78002	0.06312

Table 6. All hyperparmeters tuning results for MixHop model.

HIDDEN_FEATURE	NUM_LAYERS	POWER	F1_SCORE	ACCURACY	PREDICTION TIME
32	4	1	0.86313	0.86295	0.00320
32	16	1	0.85954	0.85871	0.00972
32	32	1	0.85123	0.85066	0.01920
64	4	1	0.85897	0.85942	0.00403
64	16	1	0.85571	0.85473	0.01579
64	32	1	0.85300	0.85208	0.03335
128	4	1	0.86793	0.86852	0.00590
128	16	1	0.86209	0.86110	0.02812
128	32	1	0.74541	0.74545	0.05958
33	4	2	0.86403	0.86278	0.00585
33	16	2	0.85820	0.85791	0.01897
33	32	2	0.74545	0.74545	0.03637
63	4	2	0.86736	0.86667	0.00745
63	16	2	0.85975	0.85880	0.03047
63	32	2	0.74253	0.74253	0.06086
129	4	2	0.86547	0.86463	0.01124
129	16	2	0.86267	0.86357	0.05789
129	32	2	0.73415	0.73422	0.11858