Project 3 Understanding Network Mining: Attacking

Final Report

Long Ma
longma2
University of Illinois Urbana-Champaign,
MCS student at Department of Computer Science
longma2@illinois.edu

Haoyu Su
haoyus2
University of Illinois Urbana-Champaign,
MCS student at Department of Computer Science
haoyus2@illinois.edu

ABSTRACT

Deep learning on graph neural networks has pivotal applications in many fields, and thus, attacks on graph neural networks are valuable research topics. We hope to find out the attack methods that are applicable under different access permission through research and experiments. In this paper, we first construct a GCN model classifier, and then experiment and analyze the results of adding pooling layers to the classifier, different data split methods, and attacking the graph using DICE, Random, and NodeEmbedding. Hoping this paper could help pave the way for subsequent research on defending against the attacks.

Keywords: Graph, Classification, Neural Network, Attack

1 INTRODUCTION

Deep learning models for graphs have a wide application today, ranging from the analysis of social and shopping networks, and biology information collection, to marketing research and management[6].

The most significant advantage of graph neural networks over other methods for node classification problems is that they contain not only the feature data of each node itself but also the edge data, which visually represent the correlation between nodes. The three main data, node features, labels, and edges, complement each other and greatly increase the stability of the graph. Even if one of them is attacked, the other content contains important information about the node and, therefore, brings less changes to the whole node.

In recent years of investigation, researchers have found that although the graphs are very powerful in detecting anomalies and combating perturbations, there is still some room for improvement[1]. For example, anomaly detection on dynamic graphs and research on adversarial robustness are very promising topics for research. And also, there are underlying network perturbations due to random noise interference that can also affect the mining results[5].

In graph networks, relationships between nodes are highly considered. In many classic node classification tasks, some nodes that have higher credit or more connected nodes play a more important role in the network, but also bear a higher risk at the same time. [3] The advantage of this feature is that even after some nodes are attacked, nodes with high weights and credits can provide more and effective information to the classifier, thus reducing the impact of the attack on the graph and keeping the accuracy at a high level. At the same time, the downside of this feature is that if these nodes with higher credit are attacked, e.g., the label is modified, it will greatly affect the training results of the other nodes they are associated with.

In this project, we aim to work out and improve an efficient graph adversarial attack for the classical models, in order to improve the original traditional model with higher robustness by studying the methods and models of these attacks in future research. We will mainly focus on node classification models for better targeting and efficiency reasons, but we also want to verify that our solution can have a high efficiency on other similar models. We will study and implement several different attack models, analyze the effectiveness of the models under different conditions and classifier iterations, and improve them or give targeted recommendations for their selection.

There are two opposing features of attacks on graph neural networks. The more the attack changes and affects the graph, the greater the impact on the accuracy of the classifier, but at the same time the easier it is to detect the attack. When we follow some characteristics of the graph itself to slightly change the nature of the graph, the attack is not easy to detect, but the perturbation of the accuracy of the team medical classifier also becomes smaller. In view of this study, we only focus on the attack model and not on the obviousness of the attack, but the stealthiness of the attack method will also be one of the factors that cannot be ignored in the practical application and the next research.

2 PROBLEM DEFINITION

The problems we face have been boiled down to two main components. First, before we start attacking the models of GNN (Graph Neural Network), we should first build our own GNN model. The model we build should reach high accuracy and some degree of robustness so that it can better show the effect of the attacking methods. We want our attack to be effective not because it attacks an already vulnerable graph, but because the attack can have an impact on a system that is fine on its own. At the same time, the accuracy of this model needs to be clearly visible so that we can compare the performance of the model before and after the attack.

In view of the running time, we will use a relatively small dataset for the main exploration, and we will also prepare some larger datasets that require longer running time, to try to do further exploration of methods that work relatively well. Also, we need to process the provided dataset and the dataset we find to process them to fit the method we have chosen.

Second, we need to understand and implement the attacking methods introduced in the guidance papers or found on our own and analyze their performance by comparing the precision of networks after attacking with the original networks. In the next few sections of this report, we will first introduce the datasets and libraries in Python that we are using to build our own networks, then introduce the principles of the attacking methods and how we implemented them.

Variable	Meaning
$\mathcal{G}$	The graph data
V	Set of Vertexes
E	Set of edges
A	Adjacent matrix
X	Node attribution matrix
$\mathcal{N}$	Node IDs set
f_θ	Prediction function of Neural Network
$\mathcal{L}$	Loss function of Neural Network
θ	Parameters of the prediction function
θ^*	Trained parameters of the prediction function
D	Degree matrix of graph
$\widetilde{A}$	A+I
$\widetilde{D}$	Degree matrix of $\widetilde{A}$
H^l	Feature of l_{th} layer of Neural Network
Φ	the limitation function of modifying the graph

Table 1: Table of Variables

3 BACKGROUND

Based on how much information can be obtained from the target classifier, we know that we can classify the adversarial attack settings into the following cases.[3] For classifiers, we have: **white box attack (WBA)** meaning all the information of the target classifier is open to the attacker; **practical black box attack (PBA)** meaning only predictions are available; **restrict black box attack (RBA)** meaning only part of the samples are available. Based on the availability of prediction confidence, the PBAs were also classified into PBA-C, which can obtain it, and PBA-D, which can only obtain discrete prediction labels. Different attack methods require different permissions, have different effects, and require different defenses, so they need to be analyzed separately.

Since many of the attack models we use act on the graph data itself, modifying the edges of the graph by adding or removing, etc., to affect the effect of the fixed classifier in the graph. With reference to the permission classification model above, we also classify the different attack methods for the permission of the graph neural network as follows. We make the permission to know only edge data as **black box graph permission (BGP)**, the permission to know all graph information including node features and labels as **white box graph permission (WGP)**, and the permission to know some graph information as **partial black box graph permission (PBGP)**. In the next presentation, we will clarify the permission for each model of attacking edges.

4 METHOD

For ease of reading, we have summarized the names of all the variables mentioned in this paper in Table 1.

4.1 Model Selection

GNN (Graph Neural Network) can help us to solve plenty of problems such as graph classification, node classification and so on. As mentioned above, we need to select models that have short run times, low data requirements, and easily observable results. After doing lots of research, we decided to solve the node classification problems, which only need one large graph as the input data (then we can split this graph by using cross-validation method), and we can easily use the accuracy of the classification as the performance of the model.

After identifying the problem to be solved, we need to pick a model to solve the problem. Currently, there are various variants of graph neural networks according to graph type, training method, and propagation method, such as GCN (Graph Convolutional Network), GraphSAGE, Graph LSTM (Long-Short Term Memory), and so on. Intuitively, researchers often use CNN (Convolutional Neural Network) to solve the classification problem, and the results are generally better. Here, we try to use GCN to finish our node classification tasks. After some experiments, GCN did well on our mainly used dataset. Therefore, we decided to use GCN as our attack model (victim model).

4.2 Dataset Introduction

To implement a node classification problem, the dataset must be a labeled graph and the information contained in the node feature should ideally be rich enough and have the clarity to classify. And in order to satisfy the training speed requirement, this picture better not be too big. In the beginning, in order to test whether the neural network we built worked properly, we found the citeseer dataset, which is well known in the field of graph neural networks and small enough to ensure that it takes very little time to run each training code. But we soon ran into a problem. Because this dataset does not have sufficient and clear node features, (we could even say that its nodes have no feature data, so we take the indexes of the node as their features), in the case where we ensure that the neural network is sufficiently complex and complete, we can only achieve an accuracy of about 60% on the training set, not to mention the results on the test set, which are less than 40%. We used various means to strengthen the accuracy of our initial neural network, such as adding convolutional layers, replacing the optimizer, tweaking the hyper-parameters, etc., but there was never a significant improvement in accuracy. We realized that there might be a problem with the dataset itself, which is not suitable for use as a node classification problem. Therefore, we decided to change the dataset.

In the snap dataset provided by the course, we quickly found the Facebook dataset which consists of 'circles' (or 'friends lists') from the Facebook app. This dataset has higher dimensional node features than the citeseer dataset. We performed our models on this dataset and evaluated the performance of the model. The good news is that it does yield a very significant improvement in accuracy, about 10% on the test set, the bad news is that it still falls short of our expectations and we still need to select a more suitable dataset.

But we finally know what direction we should go to improve the classification accuracy, and we should next pick datasets with enough dimensionality of node features. In the end, we chose the Cora dataset, which has 1433 dimensional node features, nearly a hundred times larger than the previous dataset, which seems to be sufficient. And indeed, we were not disappointed. We ran our model on this dataset and the accuracy easily reached 90% on the training set in about 500 epochs and 85% on the test set at the end, which was in line with our expectations and requirements.

For convenience to train and attack the classifier, we decide to use *PyTorch* which is an external Python library to store the dataset. And it has a block called "*torch_geometric*" designed for GNN, and it can use GPU (Graphic Processing Unit) to do the calculation, which speeds up the training process to a great extent. Now, we have finished the process of setting the developing environment for applying PyTorch and have finished the pre-processing process of the dataset to store it in a tensor structure which is built in the PyTorch Python library.

4.3 Classifier

Now, we have a data set that we can use to train the classifier. We can divide it into three parts when we build the classifier. The first one is the training set, which is used in the process of training the model. We traverse the whole set according to some certain algorithm and finish the calculation of the weights (parameters) in our model. The next one is the validation set, We applied the trained model to the validation set to observe the performance of the model, and adjusted the hyper-parameters to make the model in the best state. The final one is the test set, we use this set to finish the process of the final evaluation. The effect of the attack strategy will also be evaluated based on this set.

Euclidean data can be simply divided but graph data can not. We need to completely eliminate the validation set and test set on the whole graph when training. It means that the vertexes and the edges both need to be eliminated. And when we start to evaluate the performance of the models, the eliminated features in the training process should be completely added back.

There are so many kinds of classifiers, and we should choose the most suitable one. At present, because GCN (Graph Convolutional Neural Networks) is basically used in our reference papers provided in the guidance slides, we also choose this model first.

CNN (Convolutional neural network) is a very famous and practical classifier. We decided to use this kind of classifier to do the classification of graph data. Because of the difference between Euclidean data and non-Euclidean data, some modifications should be made when we apply CNN which was originally used to classify Euclidean data on graph data. Here we introduce a new model called GCN and try to apply this model to our pre-processed dataset.

4.4 Classification Problem

In this subsection, I will give a systematical introduction to the node classification problems and our GCN models. GCN is a kind of GNN, which means a neural network model built based on the graph structure. Most of the existing machine learning models based on computer vision or natural language processing treat the input as Euclidean data which can be represented by a matrix with different dimensions, such as images, voice signals, and so on. However, there are still a large number of non-Euclidean data in real processing problems, such as social media network data, chemical composition structure data, biological gene protein data, and citation relationship data. We need new technology and structure to handle this kind of data. That is the reason why *graphical* data appears. We can represent a graph $\mathcal{G}$ by its vertexes set and edges set,

$$\mathcal{G} = (V, E)$$

V is the set of vertexes, and its shape is $n \times d$, where n is the number of vertexes (nodes) and d is the number (dimension) of node features. E is the set of edges and its shape is $m \times w$, where m is the number of edges and w is the number (dimension) of edge features. Each vertex (node) contains its own features and each edge connects two vertexes. Edges can be directed or undirected depending on whether there is a directional dependence between vertices. We can also classify graphs into directed and undirected graphs based on the kind of their edges. The graph structure also can be defined by other metrics. In the node classification problems, we often use its adjacent matrix and the node feature matrix to define the graph,

$$\mathcal{G} = (A, X)$$

A is the adjacent matrix of the graph and can be defined as $A \in \{0,1\}^{n \times n}$. If two nodes whose indexes are i_1 and i_2 are connected, $A[i_1][i_2] = 1$, otherwise $A[i_1][i_2] = 0$. X is the node attribute matrix of the graph and $A \in \mathbb{R}^{n \times d}$. Suppose the ID set of the nodes is $\mathcal{N} = \{1, ...n\}$. Given part of nodes $\mathcal{N}_{\mathcal{L}} \subset \mathcal{N}$ and their labels $l_1, ...l_k$. Each node is assigned exactly one label. Our goal is to train a function f_θ, which can map the node $n \in \mathcal{N}$ to a label. And we can define a loss function to evaluate the performance of the model's mapping. Mathematically, our goal is to calculate the parameters θ for the function f_θ to make loss function the minimum, in formula,

$$\theta^* = argmin\mathcal{L}_{train}(f_\theta(\mathcal{G}))$$

As the function of CNN in Euclidean data, GCN can also do a wonderful job of classification on graph data. After doing a lot of research about the theorem of GCN, we decided to use "dgl (DEEP GRAPH LIBRARY)" package in Python to implement our own GCN model. Generally, there are three kinds of layers in the GCN, which are "Convolutional Layer", "Pooling Layer" and "Fully-Connected Layer". As mentioned above, after we did an experiment on the Citeseer and Facebook datasets considering the accuracy requirements of the model with the running speed requirements and memory cost, we finally settled on the structure of our model: ten layers of convolutional layers in series.

First, let's introduce how the convolutional layer works in a graph convolutional neural network. Like a normal CNN, GCN still uses forward and backward propagation to train the internal parameters. The backward propagation process also uses the SGD (Stochastic gradient descent) method as CNN, and the forward formula of GCN is,

$$H^{l+1} = \sigma(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^l W^l)$$

in which, $\widetilde{A} = A + I$, and $\widetilde{D}$ is the degree matrix calculated by the modified adjacent matrix $\widetilde{A}$, and σ is the activation function, H is the feature of each layer, for instance, $H_{input} = X$.

Also, common GCN has pooling layers and fully-connect layers like CNN, But in our own model, they are all discarded. The reason is as follows.

The reason for discarding the pooling layer is based on experimental results. When we test on the Citeseer dataset, no matter how we modify the hyperparameters of the convolution layer, the accuracy of our own designed convolutional neural network has not been improved obviously (we later found out that this phenomenon was caused by the missing node feature in the dataset). We then thought of adding a little pooling layer to improve the performance of the model. But the results are clearly unsatisfactory, and in the experiments, we have done, adding pooling layers only makes the model less and less accurate. So we then gave up on adding pooling layers to the neural network. However, since the ultimate goal of this paper is to launch an attack on the graph neural network, the reduced accuracy of the model fits our requirements instead, so we save the addition of the pooling layer as an attack method, which will be explained in detail in the next subsection.

We use the Cross-Validation method to train our model, and split our input graph to three small subset, training set, validation set and testing set. After some experiment, we take out 60% of the nodes in the graph as the training set, 30% of the nodes as the validation set, and 10% of the nodes as the test set. The reason for abandoning the fully-connected Layer is that the redundant parameters of the fully-connected Layer are too many, which does not significantly help the model performance and greatly increases the time and space complexity of the model training.

In summary, we finally decided that our model consists of ten convolutional layers in series, and as shown in the schematic below, this neural network also achieves an accuracy of over 85% on the Cora dataset that we mainly use for our attacks, which we consider to be sufficient.

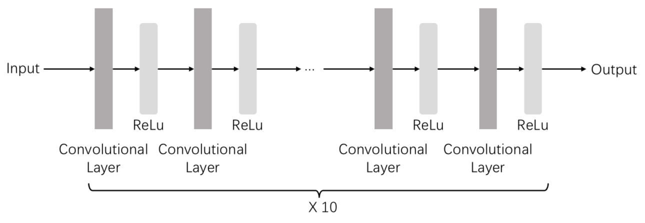

Figure 1: Our Neural Network Model

4.5 Attacking Models

In this subsection, I will give the systematic introduction to the attacking methods we have used, and the result of each of them will be shown in the evaluation section.

The basic attack methods can be roughly divided into two categories: attacking on the structure of the graphic dataset, and attacking on the structure of the neural network [4]. In our work, the attacking goals is to increase the misclassification rate of the GCN and slow down the speed of training, in other word, if after the attacking, the accuracy of the GCN decreases or the speed of convergence decreases, we can then say that our attack was effective. Mathematically, based on the two attacking categories, we build corresponding two models which can be represented by the following formulas.

$$min_{\hat{\mathcal{G}} \in \Phi(\mathcal{G})} Accuracy(f_{\theta^*}(\hat{\mathcal{G}})) s.t. \theta^* = argmin \mathcal{L}_{train}(f_\theta(\hat{\mathcal{G}}))$$

or,

$$minAccuracy(\hat{f}_{\theta^*}(\mathcal{G})) s.t. \theta^* = argmin \mathcal{L}_{train}(\hat{f}_\theta(\mathcal{G}))$$

The first one is the formula for attacking the graph structure itself. $\hat{\mathcal{G}}$ means the graph data after attacking. Φ means the limitation of modifying the graph. $\mathcal{L}$ is the loss function of the training model. f is the function that the model used to predict the labels of the new inputted nodes. and θ is the parameters of f. The model training process is to determine θ which makes the error of the prediction on the training set the smallest.

And the second one is the formula for the attacking method for the model. $\hat{f}$ is the modified function that the model after attacking to use to predict the label of new inputted nodes. Also, θ is the parameter of it.

4.5.1 Pooling. First, the method "Pooling" is to attack the GCN structure itself. As we mentioned in the previous section, when we tried to improve the performance of our GCN model, we found that adding pooling layers in GCN will cause an accuracy decrease. Therefore, we decided to see this as a pattern of attack as well. After each convolutional layer, we will add a max-pooling layer or an average-pooling layer to try to decrease the accuracy of the prediction.

4.5.2 Splitting. Changing the ratio of the three sets can also make the prediction less accurate. The general idea is that the lower the percentage of the training set, the less training the model gets, and the less accurate the training is. We will test this idea experimentally in the next section.

4.5.3 DICE (delete internally, connect externally). This method [7] is attacking the graphical dataset. Users should input the number of perturbations, and for each perturbation, we randomly choose whether to insert or remove an edge. Edges are only removed between nodes from the same classes and only inserted between nodes from different classes. This attack method can destroy the transferability of information between nodes of the same class while adding interference information between nodes of a different class, which will cause a little degree of deception to the prediction model. We passed a different number of perturbations in the test and observed the change in model performance separately, which will be analyzed in the next section. It is worth noting that the DICE attack model assumes that the attacker has full access to all datasets. To be more specific, the attacker knows the class of all nodes and can determine nodes of the same class as well as nodes of different classes. And it needs to know the edge information to ensure that it can accurately add or remove the edge. Therefore, DICE is a WGP (White Box Graph Permission) attacker.

4.5.4 Random. This method will randomly add edges to the original graph, to disrupt the flow of information in the original graph, which can decrease the accuracy of the prediction of the trained model. Because this attack model adds edges to the original graph, the attacker needs to know the nodes of the original graph as well as the edge information but node labels are not needed. Therefore, the Random attack is a PBGP (Partial Black Box Graph Permission) attacker.

4.5.5 NodeEmbedding. This method [2] is a random walk based graph weakness attack method. By random walk sampling, the algorithm selects some nodes in the original graph as the target, and adds, deletes and reverses edges between them to disrupt the information flow in the original graph. One of the more significant advantages of this attack method is that it is more stealthy and less detectable. In other words, the maintainer of the graph data is not easily aware that the data is under attack. But this comes at the expense of some attack efficiency. This method is also a PBGP (Partial Black Box Graph Permission) attacker because it only needs the edges information, while do not need the label information of these attacked nodes.

5 EVALUATION

In the evaluation phase, we generate line graphs to compare the accuracy of the test set of each model with the increasing number of epochs to see the accuracy of the data more intuitively.

5.1 Pooling

From Figure 2 we can see that when we add a pooling layer after every convolutional layer anyway, the accuracy of the whole neural network completely converges to 0 after a few jumps.

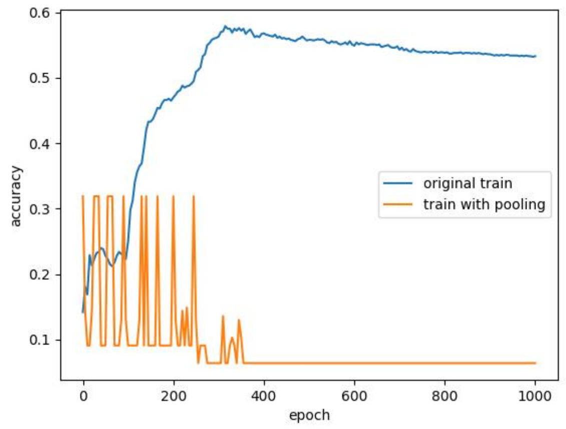

Figure 2: Performance of trained classifier before and after adding pooling layer

The purpose of using pooling layers itself is to reduce the number of parameters and computations, thus helping to reduce the occurrence of overfitting, but since there is not much data for the training itself, we add pooling layers at each step of training, thus leaving too little data that can actually be useful in training, and the neural network-based use this limited data to have a classification effect.

5.2 Splitting

At the beginning of building the classifier, we used the grouping ratio that comes with the classifier model itself, where the training set contains 140 nodes, the validation set contains 500 nodes, and the test set has the remaining 1000 nodes. The consequence is that no matter how we train, the accuracy of our model can never be improved, and it can only reach about 60% at the highest. Later, we changed the ratio for the data set to 6:3:1, and the accuracy improved significantly. The comparison can be visualized in Figure 3.

In fact, the problem caused by the ratio is that we would artificially cut the connection between many nodes. Let's say it is possible that a node itself has a very large number of neighboring nodes, but when we select only a small number of nodes as training nodes, only this node is selected for training, then its neighboring

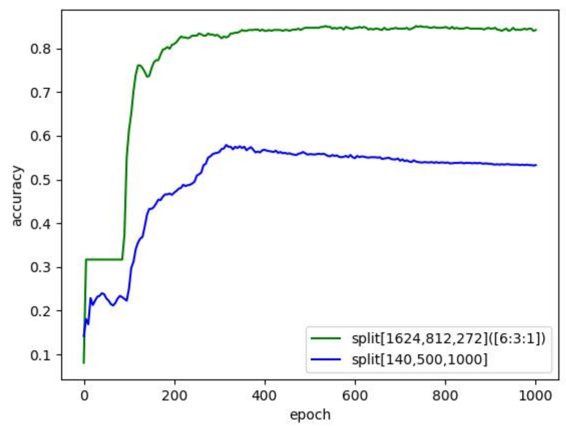

Figure 3: Performance of trained classifier with different splitting

nodes that should play a very important role in the label selection of this node during training are now dropped. The result is that the set of nodes taken for training is not similar to the original dataset that we want the classifier to work properly, and the accuracy after training is of course low.

5.3 DICE

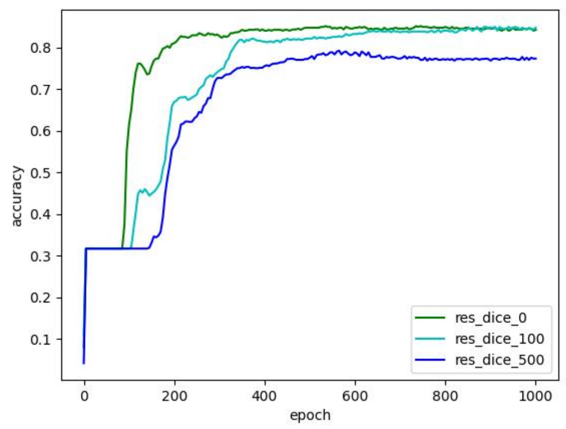

Figure 4: Performance of DICE with different Number of Perturbation

For the DICE model, we set up a control group with 0 perturbations and an experimental group with 100 and 500, respectively, and the data, as shown in Figure 4, shows a decreasing trend in accuracy for the first 0-700 epochs for the three cases, confirming the setting "the higher the number of perturbations, the greater the interference with the classifier". In the next 700-1000 epochs, we find that perturbing 100 nodes does not interfere with the accuracy of the classifier anymore.

This phenomenon indicates that for 3.5% of nodes being attacked, the classifier is able to perform a better adjustment by itself, but once the number increases to nearly 20%, the attack has a significant effect.

5.4 Random

The Random method is also a very effective method for our dataset and classifier. After trying the Random method of adding and removing edges, we present the results in Figure 5. It can be seen that generally, the accuracy decreases for all perturbations from 0 to 1000. The random edge addition method has a more significant effect on the accuracy when the number of perturbations is higher. For the random edge removal method, when the number of perturbations is lower, also produces a certain decrease in accuracy, which is better than the no change in previous models.

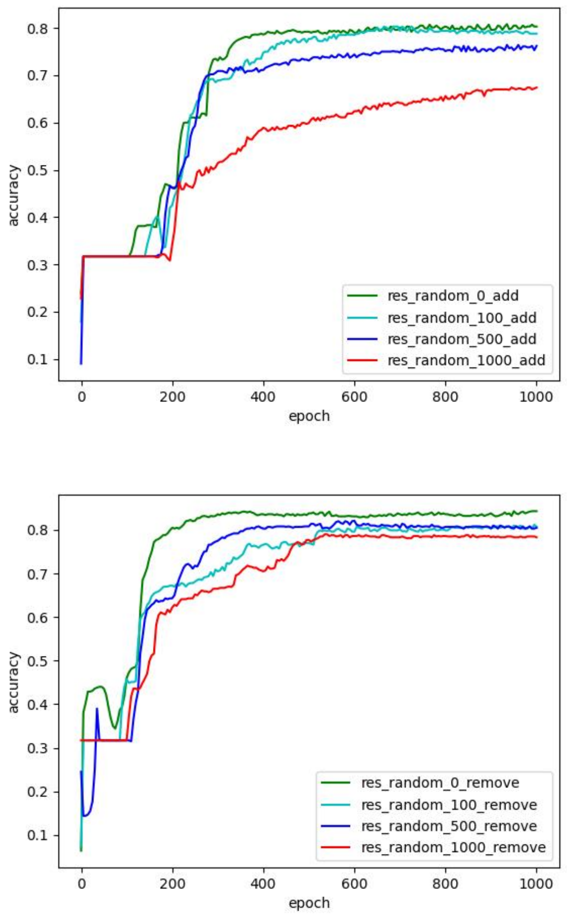

Figure 5: Performance of Random Models

This phenomenon indicates that random perturbation has a very obvious attack effect on the graph data, and the attack effect of adding edges is better than removing them. For this phenomenon, we offer our own speculation. As revealed by the graph convolutional neural network model formulation, the completion of node prediction by a GCN is very dependent on the information transfer between graph nodes. Just removing some edges, even if the nodes at both ends of these edges belong to the same class, does not play a big role in perturbing the information flow of the whole graph. What's more, the removed edges are randomly decided, and it is possible that the nodes on both sides do not communicate very important information. But adding some edges can serve to disrupt the flow of information to a great extent. This method can make a connection between two otherwise unrelated nodes, and even if there happen to be many edges added around these nodes, the originally unrelated nodes may be determined to be very importantly connected after the attack. Shunting the information they pass between otherwise connected points, or linking up nodes that are otherwise unconnected, are ways that can be used to good effect to attack. So, in this approach, adding edges can form a useful attack with a higher probability than removing edges.

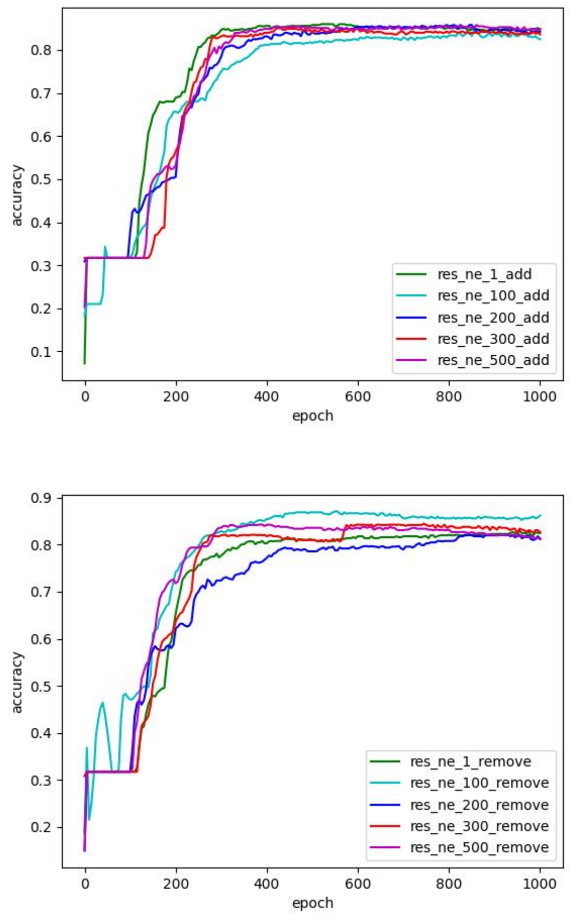

Figure 6: Performance of NodeEmbedding Models

5.5 NodeEmbedding

When trying to NodeEmbedding model, we found that in Figure 6, whether the perturbation is 1 node or 500 nodes, it seems that the training results do not change much, and the fluctuations on the figure are entirely brought about by the randomness of each run.

As we analyzed in Method before, since NodeEmbedding adds and removes edges based on a Random Walk of existing edges, it will prefer to modify between interconnected nodes. Then, when it links two points that are originally in the same class but have not yet established a direct connection, or when it removes an edge, but the two nodes of the edge are found to be related either through feature analysis or indirect connection of other nodes, it has little predictive perturbation for the whole model.

But this does not mean that the attack method is worthless, but quite the opposite. It does make changes to the model and is not easily detectable because it does not affect the accuracy. This attack method will be harder to detect and thus harder to defend. This is a noteworthy point if the next study involves this approach.

6 CONCLUSION AND DISCUSSION

In this project, we tackled two main challenges, one was the learning and implementation of convolutional neural networks, finding the right dataset, and choosing the appropriate hyperparameters to enable convolutional neural networks to achieve more satisfactory performance in the task of node classification. Another is studying the attack models introduced in the papers, clarifying the principles of attacks on graph neural networks, and implementing these models on your own neural networks. And count the convergence speed of the models and the final accuracy on the test set, and use these data to analyze the results of our attacks.

In terms of the final results, I believe that several attack models we implemented and analyzed performed as we expected. We will learn more about graph neural networks in the future and try to apply our attack models to several other graph neural networks, like GAT (Graph Attention Networks) or GLSTM (Graph Long-Short Term Memory) and so on.

REFERENCES

[1] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2014. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery* 29 (2014), 626–688.

[2] Aleksandar Bojchevski and Stephan Günnemann. 2019. Adversarial Attacks on Node Embeddings via Graph Poisoning. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 695–704.

[3] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, L. Wang, Jun Zhu, and Le Song. 2018. Adversarial Attack on Graph Structured Data. In *ICML*.

[4] Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. 2019. Topology attack and defense for graph neural networks: An optimization perspective. *arXiv preprint arXiv:1906.04214* (2019).

[5] Qinghai Zhou, Liangyue Li, Nan Cao, Lei Ying, and Hanghang Tong. 2019. ADMIRING: Adversarial Multi-network Mining. In *2019 IEEE International Conference on Data Mining (ICDM)*. 1522–1527. https://doi.org/10.1109/ICDM.2019.00201

[6] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. 2018. Adversarial Attacks on Neural Networks for Graph Data. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018).

[7] Daniel Zügner and Stephan Günnemann. 2019. Adversarial Attacks on Graph Neural Networks via Meta Learning.

7 DIVISION OF WORK

Our group has two members: Haoyu Su(haoyus2) and Long Ma(longma2).

Haoyu Su is mainly responsible for learning the attacking method of the GCN model and implementing them by using the "DeepRobust" library in Python. And also accomplished the results evaluation part.

Long Ma is mainly responsible for learning the general structure of the GCN model and pre-processing the selected dataset to fit the requirement of the "dgl" library's requirement. And Long also helped Haoyu to implement the attacking methods.