Adversial Attack on Graph Data by Node Insertion: A New Attack Scenario Based on KDD Cup 2020 Competition

Xinhao Zhu*
zhuxinhao00@gmail.com
School of Artificial Intelligence, Nanjing University

June 19, 2020

Abstract

Graph convolutional networks (GCNs), with their superiority in the node classification problem on graph data, are vulnerable to adversarial attacks as has been proved in recent studies. However, current studies primarily focus on making unnoticeable perturbations on the original graph, which is sometimes unreasonable given the fact that it's difficult to make perturbations on existing nodes in real-life graph data like social networks. Additionaly, the assumption that the target classifier is accessable, which is typically made by the majority of studies, is hard to hold true in real-life circumstances. As a result, we proposed an algorithm to attack graph data by inserting vicious nodes to the graph with the original graph unaltered. Besides, we designed an approach to solving a Black-box attack problem by solving a corresponding White-box attack problem with a surrogate model, which could degrade accuracy of the classifier used in KDD Cup 2020 Competition Phase 1 from more than 60% to less than 30%.

Keywords: Graph Convolutional Networks . Adversial Attack . Node Insertion

1 Introduction

1.1 Introduction to node classification problem

Graph data has been widely used in many real world applications, such as social networks (Twitter and Weibo) and attribute graphs (arXiv). Node classification is one of the most important tasks on graphs — given a graph with labels associated with a subset of nodes, we want to predict the labels for the rest of the nodes. To solve this problem, graph convolutional networks (GCNs), which were first introduced in [1], have gained great research interests for their outstanding prediction performance[2].

However, according to recent researches, graph mining algorithms (e.g., GCNs) are quite vulnerable to adversial attacks, which could greatly compromise the classification accuracy of learning models[3]. As a result, considerable efforts from the data mining and machine learning community have been devoted to various attacking scenarios in the node classification problem, where they aimed at reducing the classification accuracy on the total dataset (**Global attack**) or just some specific node (**Targeted attack**), attack at training-time (**Poisoning attack**) or test-time (**Evasion attack**), and with different levels of knowledge about the system under attack (**White-box/Grey-box/Black-box attack**).

In this report, we focus on a new attack scenario in node classification problem, where we attack a node classifier by injecting some vicious nodes at test-time, manipulating their features and connectivities so that the classifier has a compromised prediction performance on a specific subset of nodes.

*Student ID: 181220080

1.2 Related Work

In recent years, various researches about adversarial attacks in graph-related problems has been conducted with different attacking scenarios. **Nettack**, which was introduced in [4], exploited incremental computations to make unnoticeable perturbations on the graph data, targeted at making specific target node being misclassified by the classifier. Another popular model **Mettack** achieved improved training-time attack (i.e., Poisoning attack) performance by applying meta learning[3], leading to significantly reduced prediction accuracy of the classifier on the total dataset (i.e. Global attack). Additionaly, **RL-S2V** used reinforced learning to tackle the Targeted attack problem[5], where their perturbations are limtied to the graph structure using test-time attack (i.e., Evasion attack).

However, none of these popular models are designed for compromising classifier performance by adding vicious nodes to the graph, which is a more realistic attack scenario since the attacker rarely has access to existing nodes in the graph. For instance, when attacking a social network classifier, it's far more easy to add new nodes to the graph by creating new accounts, while getting access and manipulating existing accounts is almost impossible. Only few researches like [6], [7] and [8] are digging into this nodes-adding scenario. Nevertheless, all the three researches mentioned above focus on training-time attack (i.e., Poisoning attack), where the classifier will be retrained on the modified graph. Since model training on large-scaled graph data is expensive, and model re-training is a relatively infrequent action in the production environment, it's more practical to deploy test-time attack (i.e., Evasion attack) in real-life problems. Besides, the attacker can rarely have access to the classifier in reality, which means all the gradient-based White-box attack methods out of action.

As is shown in [6], directly adapting these attacking methods to the new attack scenario is not promising due to high complexity, hence we are motivated to devise a new attack strategy tailored for the vicious node setting with a low computation cost.

1.3 Contributions

In this report, we:

- Designed an algorithm to attack GCNs at test-time by injecting vicious nodes to the original graph, which significantly reduces the performance of GCN classifiers on a fixed target set, with existing edges and node features kept intact.
- Proposed a method to achieve Black-box attack by doing White-box attack with a pre-trained surrogate model, which can degrade an inaccessable classifier's accuracy to a fair extent.
- Conducted experiments on the KDD Cup 2020 dataset, where we used both the local classifier trained by ourselves and the online classifier provided by the competition committe. Our best method could degrade the online classifier's accuracy from $> 60\%$ to $< 30\%$.

2 Model

2.1 Notation and Preliminary

In this section, we first introduce the notations used throughout this report and preliminaries of GCNs. Here, following the standard notations in the literature [9], we suppose that $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ denotes an undirected featured network (graph) with n nodes (e.g., $v_i \in \mathcal{V}$), m edges (e.g., $e_{ij} = (v_i, v_j) \in \mathcal{E}$), and d features (e.g., $f_i \in \mathcal{F}$).

The features of of these n nodes are given by $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n]^\top \in \mathbb{R}^{n \times d}$, where $\mathbf{x}_i \in \mathbb{R}^d$ denotes the feature of the i-th node v_i. The adjacency matrix $\mathbf{A} \in \{0, 1\}^{n \times n}$ contains the information about how the nodes are connected, where each component $\mathbf{A}_{ij}$ denotes whether the edge e_{ij} exists in the graph.

Hence, we denote the graph as $\mathcal{G} = (\mathbf{A}, \mathbf{X})$ for simplicity.

Besides, these n nodes are labeled as P classes, denoted as $1, \ldots, P$, and a label matrix $\mathbf{L} \in \{0, 1, \ldots, P\}^n$ is provided to store the class information. Note that GCNs are typically designed for semi-supervised node classification, i.e., some of these n nodes are unlabeled, so we use class 0 to mark a node as unlabeled.

There are several variations of GCNs, but we consider one of the most common approaches introduced in [1]. Starting from $H^0 = \mathbf{X}$, GCN uses the following rule to iteratively aggregate features from neighbors:

$$H^{(k+1)} = \sigma(\hat{\mathbf{A}} H^{(k)} W^{(k)}), \tag{1}$$

where $\hat{\mathbf{A}}$ is the normalized adjacency matrix, σ is the activation function and $W^{(k)}$ is the weights of GCN at the k-th layer.

As has been discussed in [7], there are two common ways for normalizing $\mathbf{A}$: **symmetric normalization** and **row-wise normalization**. The original GCN paper applied symmetric normalization, which sets $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ (adding 1 to all diagonal entries) and then normalize it by $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}$ where $\tilde{\mathbf{D}}$ is a diagonal matrix with $\tilde{D}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$. At the same time, row-wise normalization is done by computing $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-1} \tilde{\mathbf{A}}$, namely normalizing the adjacency matrix by rows.

We set $\sigma(x) = \mathbf{ReLU}(x) = \max(0, x)$, which is the most common choice in practice. For a GCN with K layers, after getting the top-layer feature $H^{(K)}$, a fully connected layer with soft-max loss is used for classification.

A commonly used approach is to apply two-layer GCN to semi-supervised node classification on graphs[1]. The model could be simplified as:

$$\begin{aligned} Z &= \operatorname{softmax}(\tilde{\mathbf{A}} \sigma(\tilde{\mathbf{A}} \mathbf{X} W^{(0)}) W^{(1)}) \\ &= \operatorname{softmax}(f(\mathbf{A}, \mathbf{X})) \end{aligned} \tag{2}$$

For simplicity, we assume that our target network is structured as 2, but generally our algorithm can be used to attack GCNs with more layers.

2.2 Problem Definition and Methodology

In this report, we consider an attack scenario where we attack t specific nodes $\mathcal{V}_t \subseteq \mathcal{V}$ by inserting n_{in} vicious nodes $\mathcal{N}$, denoted by $|\mathcal{N}| = n_{in}$ and $\mathcal{N} \cap \mathcal{V} = \emptyset$. We suppose that nodes in $\mathcal{V}_t$ are indexed by $s_1, s_2, \ldots, s_t$, i.e., $\mathcal{V}_t = \{v_{s_1}, \ldots, v_{s_t}\}$, and each inserted node have a maximum degree h (i.e., a new node can connect to at most h different nodes).

Formally, let $\mathcal{G}' = (\mathbf{A}', \mathbf{X}')$ be the new graph after performing perturbations on the original gragh $\mathcal{G}$, then we have

$$\mathbf{A}' = \begin{bmatrix} \mathbf{A} & \mathbf{E} \\ \mathbf{E}^\top & \mathbf{O} \end{bmatrix}, \quad \mathbf{X}' = \begin{bmatrix} \mathbf{X} \\ \mathbf{X}_{in} \end{bmatrix}. \tag{3}$$

Note that $\mathbf{E} \in \{0,1\}^{n \times n_{in}}$ denotes the adjacency matrix between original nodes and vicious nodes, and symmetric matrix $\mathbf{O} \in \{0,1\}^{n_{in} \times n_{in}}$ denotes the relationship between vicious nodes $\mathcal{N}$. Additionaly, we would use $\mathcal{N}[i]$ to represent the $(i+1)$-th node in $\mathcal{N}$, and $\mathcal{N}[i:j]$ to represent the nodes indexed from $i+1$ to j.

In the proposed attack scenario, we're supposed to degrade a classifier's accuracy on the target set $\mathcal{V}_t$. The classifier is pre-trained on the original graph $\mathcal{G}$, and its weights are fixed after the training stage, i.e., a test-time attack is deployed. Note that we assume that the target set nodes are labeled. Hence our task:

$$\begin{aligned} &\text{minimize} \sum_{i=s_1}^{s_t} \mathbf{1}_{[\mathbf{L}[i] = \arg\max \operatorname{softmax}(f(\mathbf{A}', \mathbf{X}'))[i]]} \\ &\text{s.t.} \sum_{j=1}^{n+n_{in}} A'[i][j] \le h, \ i = 1, \ldots, n_{in}. \end{aligned} \tag{4}$$

Besides, there are three kinds of attacking scenario, classified by the level of information we know when attacking:

1. White-box attack: $\mathcal{G} = (\mathbf{A}, \mathbf{X})$, labels $\mathbf{L}$ of the target set, and classifier function f (including information like gradient) are known to the attacker.

2. Grey-box attack: $\mathcal{G} = (\mathbf{A}, \mathbf{X})$, labels $\mathbf{L}$ of the target set are known to the attacker, but classifier function f is hidden.

3. Black-box attack classifier function f is hidden, and part of the graph $\mathcal{G}$ or target set labels are hidden.

We will show in the next section about how these concepts work in our task.

2.3 The KDD Cup 2020 Problem

Our primary objective in this report is to design an algorithm tailored for the KDD Cup 2020 problem.

There are 593486 nodes in the KDD Cup 2020 dataset (i.e., $n = 593486$), with every node associated with a 100-dimensional feature (i.e., $d = 100$). The first 543486 nodes are labeled, while the last 50000 nodes are unlabeled. The target set $\mathcal{V}_t$ in this problem is the last 50000 nodes, and we are supposed to degrade a fixed online classifier's accuracy on the target set. The classifier would not be made public, which means our task is more of a **Black-box attack** since we don't know the labels for the target set, neither parameters of the online classifier.

However, we can use the following method to transform a Black-box problem into a White-box problem:

1. For hidden classifier: We can train a local classifier f_{local} on the original graph to approximate f_{online}.

2. For incomplete information: We can use the labels of nodes in $\mathcal{V}_t$ predicted by f_{local} to approximate ture labels of nodes in $\mathcal{V}_t$.

Some experiments are made by us in section 5, showing that this approximation method can actually help in degrading the accuracy of the hidden online classifier.

3 The Proposed Method

We would like to present our method as two independent stages: **training** and **attacking**. Note that we work sorely on local classifiers in this section, i.e. a White-box attack is performed.

3.1 Training Stage

First, we would train a GCN classifier on the original graph $\mathcal{G} = (\mathbf{A}, \mathbf{X})$, using the method introduced in [1]. This GCN classifier works as a surrogate model for the online classifier. No specific constant are made to the surrogate classifier, but in this project we will use the model described in 2 for simplicity.

Since the training of GCNs is not the key point in this project, it will not be elaborated in this section. We will give an example of GCN training in section 5. For detailed information about training GCN, please refer to [1].

3.2 Attacking Stage

Then, we would like to attack the classifier trained in the previous section. We would degrade its classification accuracy on a specific subset of the graph by adding new nodes to the graph. Due to the complexity of large-scaled attribute graph, the attacking stage is separated into two discrete steps: **node insertion** and **feature optimization**.

3.2.1 Node Insertion

First, some modifications are made to the graph structure $\mathbf{A}$ so that new nodes are added to the original graph. Since the degrees of new nodes are limited to h in this problem, we can choose whether to attack **directly** (i.e., new nodes are directly connected to the nodes being attacked), or attack **indirectly**. As is discussed in [6], direct attack are generally more effective than indirect attack, so we would use direct attack and connect new nodes to nodes in $\mathcal{V}_t$ directly.

Hence, we proposed three approaches to connecting new nodes to nodes in $\mathcal{V}_t$ directly:

1. Sequential Nodes are connected sequentially according to their index, i.e., $\mathcal{N}[0]$ is connected with $\mathcal{V}_t[0:h]$, $\mathcal{N}[1]$ is connected with $\mathcal{V}_t[h+1:2h]$, etc.

2. Same New nodes will preferentially connect original nodes with identical labels.

3. Different New nodes will preferentially connect original nodes with different labels.

A detailed algorithm for the node insertion method is included in section 4.

3.2.2 Feature Optimization

After node insertion, we would optimize the features of new nodes $\mathcal{N}$. Similar to the method used in [10], we could iteratively compute the class probablity:

$$Z' = \text{softmax}(f(\mathbf{A}', \mathbf{X}'_{(i)})),$$

where $\mathbf{X}'_{(i)} = [\mathbf{X}, \mathbf{X}_{in(i)}]^\top$ denotes the feature matrix at the i-th iteration.

Then we can compute the cross-entropy loss on $\mathcal{V}_t$:

$$\mathcal{L}_{(i)} = \text{CrossEntropyLoss}(Z'[s_1, \dots, s_t], \text{Labels}[s_1, \dots, s_t]).$$

Afterwards, we can update the feature matrix of vicious nodes by:

$$\mathbf{X}_{in(i+1)} = \mathbf{X}_{in(i)} + \alpha \times \frac{\partial \mathcal{L}_{(i)}}{\partial \mathbf{X}_{in(i)}}$$

where α is a positive constant to maximize the loss function.

To increase the stability of this method, we can make an additional clipping process after each iteration, i.e., assigning a positive constant Δ_{max}, making sure that every entry of $\mathbf{X}_{in(i+1)}$ lies in $[-\Delta_{max}, \Delta_{max}]$. The constant Δ_{max} can be chosen according to the dataset, with a good empirical choice being the maximum of the corresponding feature in $\mathbf{X}$.

4 Attack Algorithm

Here we present a detailed algorithm for attack GCN classifiers, as is described in section 3.2.

Algorithm 1 Attack Algorithm Against GCN by Inserting Vicious Nodes

```
Input: original graph $\mathcal{G} = (\mathbf{A}, \mathbf{X})$, node labels $\mathbf{L}$, targeted set $\mathcal{V}_t = \{v_{s_1}, \dots, v_{s_t}\}$, GCN classifier
    function $f$, vicious node number $t$, vicious node degree $h$, positive constant $\alpha, \Delta_{max}$, insertion
    mode $mode$.
1:  $\mathbf{A}' = \text{node\_insertion}(\mathbf{A}, \mathbf{L}, \mathcal{V}_t, t, h, mode)$
2:  Initialize $\mathbf{X}_{in}$ randomly
3:  $\mathbf{X}' = [\mathbf{X}, \mathbf{X}_{in}]$
4:  for iteration = 1 to max_iter do
5:      $Z' = \text{softmax}(f(\mathbf{A}', \mathbf{X}'))$
6:      $\mathcal{L} = \text{CrossEntropyLoss}(Z'[s_1, \dots, s_t], \mathbf{L}[s_1, \dots, s_t])$
7:      $\mathbf{X}_{in} = \mathbf{X}_{in} + \alpha \times \frac{\partial \mathcal{L}}{\partial \mathbf{X}_{in}}$
8:      Clipping each entry of $\mathbf{X}_{in}$ into $[-\Delta_{max}, \Delta_{max}]$
9:      $\mathbf{X}' = [\mathbf{X}, \mathbf{X}_{in}]$
10: end for
Output: $\mathcal{G}' = (\mathbf{A}', \mathbf{X}')$
```

where the node insertion algorithm is defined as follows:

Algorithm 2 Node Insertion Algorithm

Input: original adjacency matrix $\mathbf{A}$, node labels $\mathbf{L}$, targeted set $\mathcal{V}_t = \{v_{s_1}, \dots, v_{s_t}\}$, vicious node number t, vicious node degree h, insertion mode *mode*.

```
1:  $\mathbf{A}' = \mathbf{A}$
2:  if $mode$ = Sequential then
3:    OptList = $[s_1, \dots, s_t]$
4:  else
5:    for $c = 1$ to $P$ do
6:      ClassList[$c$] = new List()
7:      ClassList[$c$].add($s_i$), for all $\mathbf{L}[s_i]$=$c$
8:    end for
9:    OptList = new List()
10:   if $mode$ = Same then
11:     for $c = 1$ to $P$ do
12:       OptList.extend(ClassList[$c$])
13:     end for
14:   else
15:     while len(ClassList) > 0 do
16:       for list in ClassList do
17:         OptList.add(list.pop())
18:         if len(list) = 0 then
19:           delete list from ClassList
20:         end if
21:       end for
22:     end while
23:   end if
24: end if
25: for $i = 1$ to $n_{in}$ do
26:   for $j = 1$ to $h$ do
27:     $\mathbf{A}'[i+n]$[OptList$[i*h+j]$] = 1
28:     $\mathbf{A}'$[**OptList**$[i*h+j]$]$[i+n]$ = 1
29:   end for
30: end for
```

Output: $\mathbf{A}'$

where we made a assumption that $|\mathcal{V}_t| = n_{in} \times h$, which holds true in KDD Cup 2020 dataset. However, this algorithm can be easily extended to other cases when $|\mathcal{V}_t| \neq n_{in} \times h$.

5 Experiments

5.1 Environment

Due to the high complexity of the KDD Cup 2020 dataset, we used the Google Colabroatory Service to run our model, including the training stage and attacking stage. Most of the time we used a signle NVIDIA Tesla K80 GPU for computation, but some work are done on a NVIDIA Tesla T4 GPU or NVIDIA Tesla V100-PCIE-16GB GPU.

Code of our model is built based on Pytorch Framework. For a introduction to the code we used in this project, please refer to the Appendix.

5.2 Local Experiments

In this section, we would explore the performance of our method in a White-box attack scenario, i.e., we would use a local classifier as a surrogate, trying to degrade its classification accuracy on the target set. All of the experiments are done on the KDD Cup 2020 Dataset.

The local classifier is a GCN network with two hidden layers, with node number 64 and 32 respectively, as is described in 2. We used **ReLU** as the activation function, and set a 50% dropout rate for all the layers except the last to avoid over-fitting. The dataset is splited into train/val/test set by 0.65/0.25/0.1. After 200 epoches of training, our local classifier gets a 40.56% classification accuracy on the validation set. According to our method described in section 3, we used the predicted labels of the target set as their true labels.

5.2.1 Node Insertion

First, we would discuss the three different node insertion methods proposed in section 3. To make a comparison between the performance of these methods, we used our attack algorithm to degrade the classification accuracy of loacl classifier. All the attack setting are the same except for insertion methods. The result is illustrated as follows.

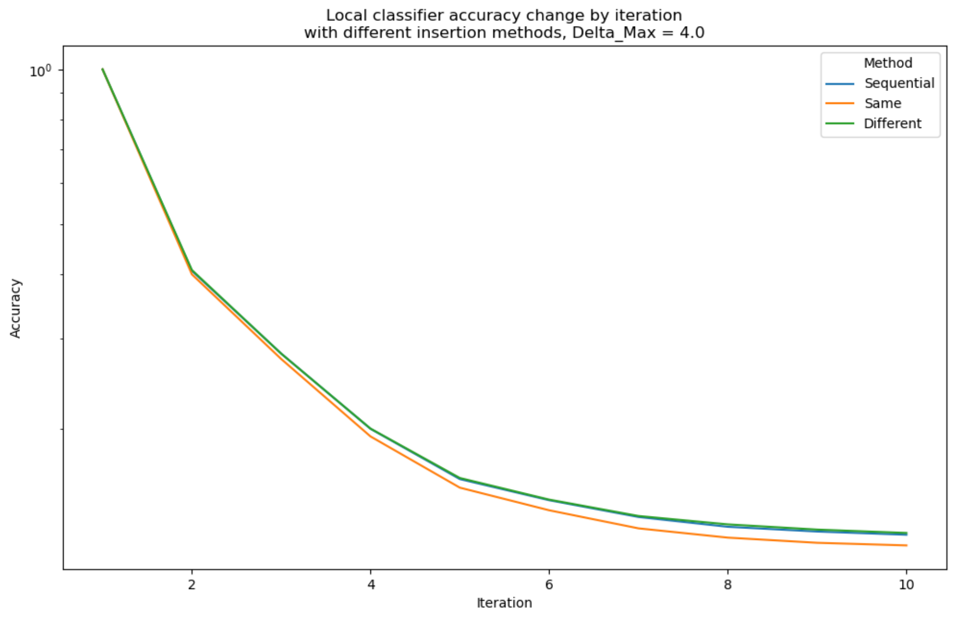


Figure 1: Local classifier accuracy change by iteration, with different insertion methods

As is shown in the figure, **Same** slightly outperforms the other methods, while there's no noticeable difference between **Sequential** and **Different**. This can be explained by the fact that nodes are more likely to be influenced by their **direct** neighbors, not **indirect** neighbors[6]. Hence, if a vicious node is connected with multiple original nodes labeled 1, we just optimize its feature so that its neighbors can be re-classified as some class other than 1. However, it a vicious node has neighbors with different labels, we may have different optimization targets for different neighbors so that the optimization becomes difficult and ineffective.

5.2.2 Feature Optimization

Then, we would discuss how the feature optimization step influences the performance of our algorithm. Specifically, we would explore how the **Delta_Max** constant influences the performance of our model. The following is the experiment reuslts with different Delta_Max constant.

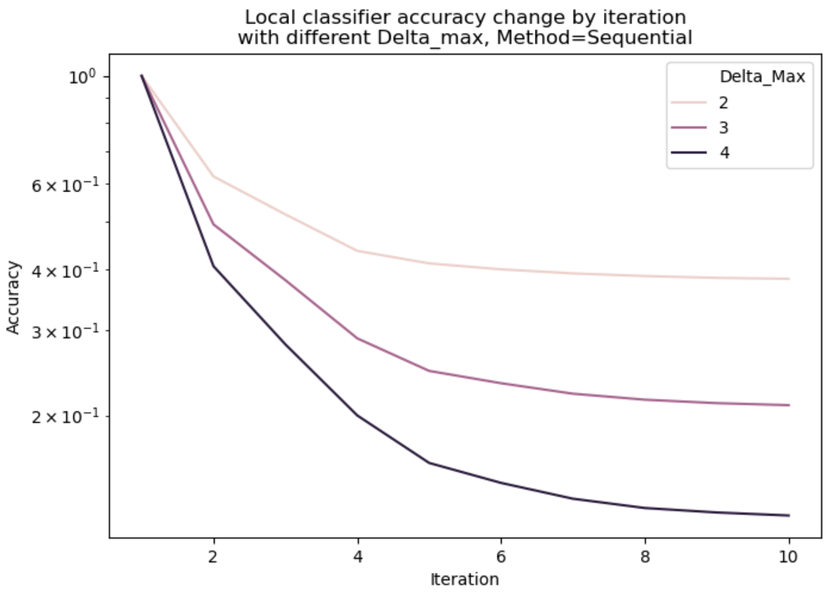


Figure 2: Local classifier accuracy change by iteration, with different Delta_Max

As is shown in the picture, greater Delta_Max constant makes for greater accuracy reduction. This is primarily because of the fact that **we didn't make normalization on the feature matrix**. Since the original features are between $[-2, 2]$, a vicious node with feature out of this range would greatly interfere with the convolution process, leading to dramatically decreased classification accuracy. Once the feature matrix is normalized, different Delta_Max constant bigger than 2 would make almost no difference on the KDD Cup 2020 dataset.

Here, we want to present this seemingly trivial discovery because **it could happen in real life**. At the beginning of the competition. the committe used a classifier which would not make normalization on the feature matrix. Consequently, a randomly inserted 100 nodes with all the features valued 99 could easily degrade the classifier's accuracy from 60% to 4%. Though this problem was quickly fixed by the committe several days later, it tells us that an un-normalized feature matrix could pose great threat to a GCN classifier.

5.3 Online Experments

In this section, our target is to degrade the classification accuracy of an unknown online classifier on the target set.

The greatest difficulty we encountered is that we are only allowed to submit two times a day, which means we can not make extensive experiments to fully exploit the similarity between the online classifier and our local classifier.

As a result, we used a surrogate model, as is described in the previous section, to approximate the online classifier. We got three models with different validation accuracy, and used identical methods to attack them (attack 15 iterations, using **Same** as insertion method). The final accuracy of local classifier are all around 5%, and the corresponding online accuracy are listed as follows:

Table 1: Online accuracy using surrogate with different accuracy

Index	Method	Epoches	Validation Accuracy	Online Accuracy
1	Clean	N/A	N/A	0.60506
2	GCN surrogate	200	0.4056	**0.28641**
3	GCN surrogate	300	0.4192	0.29908
4	GCN surrogate	400	**0.4252**	0.30715

Note that over-fitting is not likely to happen in these experiments: the validation accuracy keeps increasing. This is probably because that there is some fundemantal gap between the surrogate classifier and online classifier, be it the model structure (e.g., number of nodes, layers), training method (e.g., dataset split), or model type (e.g. the online classifier is based on other algorithms like random walk).

Hence, directly maximizing the accuracy of local classifier is not likely to increase its similarity with the online classifier, and we are supposed to fully utilize the attack method to get best online performance.

After multiple experiments, we developed a fair surrogate model (refer to the Appendix for more information) of the online classifier, and tested different attacking parameters to maximize the performance of our model. In the end, we managed to degrade the online classifier's accuracy on target set from around 60% to 28.641%. We set Delta_Max = 3.05 and attack_iter = 15 in this experiment, and imployed **Same** as the node insertion method.

6 Conclusion

There are several drabacks in our project:

1. We did not make comparisons with existing methods like Nettack. We planned to make comparisons between our method and other existing methods, and we actually implemented Mettack to this new attack scenario, but the complexity of KDD Cup 2020 dataset makes that almost all the data structure related to adjacency matrix be transformed to sparse tensor, and lots of operations need to be adjusted accordingly. Moreover, the speed of the algorithm is greatly affected such that a single iteration takes more than 1 minute. Since at least 50000 (500 nodes * 100 features/egdes) iterations are required to finish the attack, we have to skip these content for lack of time.

2. We failed to develop our algorithm to different classifier models. As has been discussed in section 5, we are not sure that whether the online classifier is based on GCN or not. We are glad to make more experiments and comparisons on multiple classifier models if more time is granted.

3. We did not make detailed algorithm analysis and extensive experiments. We, to the best of our ability, tried to create a well-structured report with substantial content in a limited time (2-days), which unavoidably makes us unable to cover some significant content in this report. It's a pity that we can't elaborate in these aspects on the basis of studies like [6].

In conclusion, we designed a new "vicious nodes injection" scenario in graph classification probelm, where a vicious attacker attacks a fixed model at test-time, aiming at degrade its classification accuracy on a specific subset of nodes. This scenario is more realistic than existing ones since it makes a much lower assumption on the attacker's ability to access and modify graph data.

Besides, we proved by this project that a Black-box attack in the former scenario can be partly achieved by training a local surrogate model and solving a typical White-box attack problem. We proposed a simple, gradient-based algorithm to solve this problem, and discussed how to utilize this algorithm to get better scores in the KDD Cup 2020 Phase 1 competition.

7 Acknowledgement

First, we would like to express our gratitude to **Zhenghai Xue** (STUID: 181220063) and **Ruice Rao** (STUID: 181220041) from the **XSR-Attackers** group. We got plenty of ideas during our discussion with them, and some of these ideas are proved to be quite helpful. We're more than glad to cooperate with them in the future.

Besides, our Pytorch code is partly built on the basis of this page and this repo. They greatly helped us as beginners in Pytorch. Thank the authors of these two projects for their kind action.

References

[1] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," *arXiv:1609.02907 [cs, stat]*, Feb. 2017. arXiv: 1609.02907.

[2] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.

[3] D. Zügner and S. Günnemann, "Adversarial Attacks on Graph Neural Networks via Meta Learning," *arXiv:1902.08412 [cs, stat]*, Feb. 2019. arXiv: 1902.08412.

[4] D. Zügner, A. Akbarnejad, and S. Günnemann, "Adversarial Attacks on Neural Networks for Graph Data," *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2847–2856, July 2018. arXiv: 1805.07984.

[5] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song, "Adversarial Attack on Graph Structured Data," *arXiv:1806.02371 [cs, stat]*, June 2018. arXiv: 1806.02371.

[6] J. Wang, M. Luo, F. Suya, J. Li, Z. Yang, and Q. Zheng, "Scalable Attack on Graph Data by Injecting Vicious Nodes," *arXiv:2004.13825 [cs]*, Apr. 2020. arXiv: 2004.13825.

[7] X. Wang, M. Cheng, J. Eaton, C.-J. Hsieh, and F. Wu, "Attack Graph Convolutional Networks by Adding Fake Nodes," *arXiv:1810.10751 [cs]*, Nov. 2019. arXiv: 1810.10751.

[8] Y. Sun, S. Wang, X. Tang, T.-Y. Hsieh, and V. Honavar, "Node Injection Attacks on Graphs via Reinforcement Learning," *arXiv:1909.06543 [cs, stat]*, Sept. 2019. arXiv: 1909.06543.

[9] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph Neural Networks: A Review of Methods and Applications," *arXiv:1812.08434 [cs, stat]*, July 2019. arXiv: 1812.08434.

[10] J. Chen, Y. Wu, X. Xu, Y. Chen, H. Zheng, and Q. Xuan, "Fast Gradient Attack on Network Embedding," *arXiv:1809.02797 [physics]*, Sept. 2018. arXiv: 1809.02797.

Appendix

Introduction to Our Code

Usage Recommendation: Upload the whole code directory into Google Colabroatory, placing KDD Cup 2020 dataset into ./dataset/, and open KDD.ipynb to run the model.
The following is a brief introduction to all the files in the code directory.

1) **./dataset/** Where the KDD CUP 2020 dataset is stored. Note that the dataset itself is not included in the submmited file.

2) **./log/** Surrogate model and submit files for previous submissions, denoted by the corresponding online score. Besides, several leaderboard screenshots in these days are provided for reference.

3) **./model/** Where the surrogate model trained in train.py are stored, with:

   a) **./model/model_64_32_best.pth** Our best surrogate model.

4) **./output/** Where the submission file (adjacency matrix, feature matrix) generated in attack.py are stored.

5) **./utils/** Utility model used by attack.py and train.py, with:

   a) **./utils/dataset.py** Functions about data loading,

b) **./utils/gcn.py** Implementation of GCN classifiers,

c) **./utils/poison.py** Functions about node insertion,

4) **./utils/utils.py** Utility function shared by all the models, with basic function like tensor creating and normalization.

6) **./KDD.ipynb** An interactive Jupyter notebook file, can be used as a general entry script.

7) **./attack.py** An entry script for the attacking stage, as is described in section 3.

8) **./train.py** An entry script for the training stage, as is described in section 3.

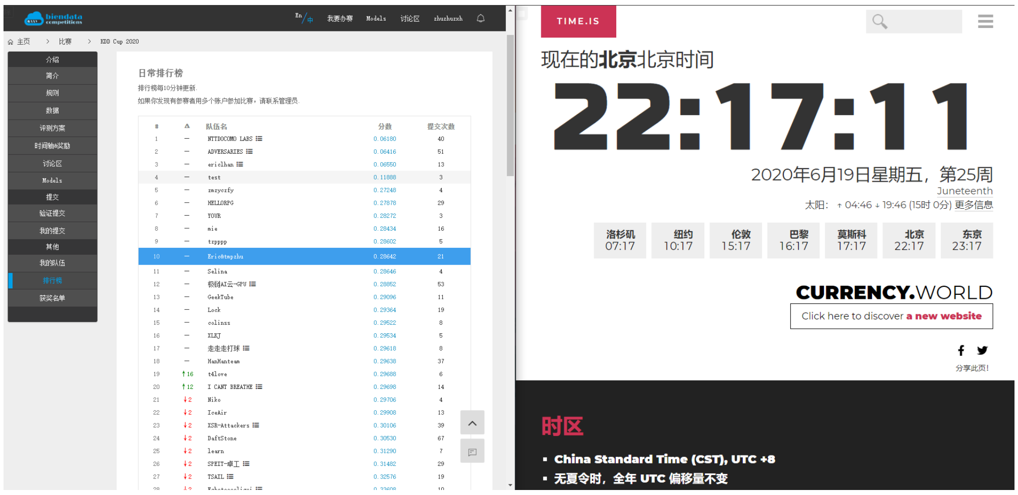
End Note

Team Information

Xinhao Zhu (STUID: 181220080, zhuxinhao00@gmail.com), as the only member of group "Eric@tmpzhu", finished this project on his own. Please pardon him for being addicted to using "we" in this report.

Grade Information

We achieved the **10-th** grade in the KDD Cup 2020 Phase 1, with a final score **0.28642**. The average score for current top-10 models is **0.20011**. We are still wondering how the top-3 model achieved a **0.06** score — maybe it's a good choice to read someariticles from KDD 2020.



日常排行榜

排行榜每10分钟更新.

如果你发现有参赛者用多个账户参加比赛，请联系管理员.

#	Δ	队伍名	分数	提交次数
1	—	NTTDOCOMO LABS	0.06180	40
2	—	ADVERSARIES	0.06416	51
3	—	ericlhan	0.06550	13
4	—	test	0.11888	3
5	—	zmzyczfy	0.27248	4
6	—	HELLORPG	0.27878	29
7	—	YOUR	0.28272	3
8	—	mie	0.28434	16
9	—	tzpppp	0.28602	5
10	—	Eric&tmpzhu	0.28642	21
11	—	Selina	0.28646	4
12	—	极链AI云-GPU	0.28852	53
13	—	GeekTube	0.29096	11
14	—	Lock	0.29364	19
15	—	colinss	0.29522	8
16	—	XLKJ	0.29534	5
17	—	走走走打球	0.29618	8
18	—	HanHanteam	0.29638	37
19	↑16	t4love	0.29688	6
20	↑12	I CANT BREATHE	0.29698	14
21	↓2	Niko	0.29706	4
22	↓2	IceAir	0.29908	13
23	↓2	XSR-Attackers	0.30106	39
24	↓2	DaftStone	0.30530	67
25	↓2	learn	0.31290	7
26	↓2	SPEIT-卓工	0.31482	29
27	↓2	TSAIL	0.32576	19

现在的**北京**北京时间

22:17:11

2020年6月19日星期五，第25周

Juneteenth

太阳: ↑04:46 ↓19:46 (15时 0分) 更多信息

时区

- China Standard Time (CST), UTC +8
- 无夏令时，全年 UTC 偏移量不变

Figure 3: Our grades at 06/19/2020, 22:17, GMT+8

Note that we achieved this score at June 14, with a best 3-rd grade on the leaderboard, but failed to get better score in the following days because we spent about 3 days implementing Mettack in this new attack scenario (which turned out not to be helpful), and two days writing this report. We're considering about getting more team members and having better time management in case of another competition like this.

Please feel free to contact if there is something wrong with the project.