Node Classification in Citation Graph Dataset

Jihwan Oh
School of Industrial Engineering
UNIST
dhwlghks00@unist.ac.kr

Abstract

These days, various method of Graph Neural Network (GNN) is being studied. Graph Contrastive Learning with Augmentations (GraphCL, Nips 2020) introduced contrastive learning based on Deep Graph Infomax (DGI, ICLR 2019). This is a state-of-the-art methodology of graph unsupervised learning. It performed well in many graph classification tasks beyond the supervised method. We study and implement the GraphCL with 4 augmentation functions: Node drop, Edge perturbation, Attribute masking, and Subgraph. Our objective is node classification in the citation graph dataset. Additionally, we introduce a new method: Adopt the PageRank algorithm to the Subgraph function. The result shows that our existing implementation success and our new method is effective.

I. Introduction

Until these days, many types of neural network have been studied. Especially, Convolution Neural Network (CNN) with image data is the hottest topic in these days. Many deep learning techniques are developed through CNN. However, only a few methods have been experimentally introduced in the field of graph machine learning. This is because graph data has some characteristic properties unlike image data. Such properties sometimes have a limited effect on designing machine learning structures. The permutation invariant property is typical. Like this, graph data cannot be handled at will like image data. Graph convolution network (GCN), the beginning of graph machine learning, is also an applied model from CNN's idea. Recently in 2017, GraphSAGE paper effectively enhance GNN encoders based on aggregation of graph data. Since then, studies introducing effective GNN encoders in various ways have attracted attention. Meanwhile, a Deep infomax paper introducing unsupervised continuous learning based on contrastive learning is published. This method is then optimized for graph data in the Deep graph infomax paper. Finally, the method is drawing attention once again in 2020 by announcing amazing performance using graph augmentation in Graph Contrastive Learning with Augmentation (GraphCL). Therefore, in this report, we will test and verify once again whether GraphCL actually performs well in the real world. The four graph data augmentation functions presented in the paper are directly implemented and applied to the data. In addition, a newly modified augmentation function is implemented by applying PageRank algorithm in addition to the presented augmentation function. In Graph data, there are tasks such as node classification, edge classification, and graph classification. Among them, we will perform a node classification task that predicts the class of nodes in representative Graph data. For the data, we will use the citation data, which is processed as graph data, will be used for the citation relationship of the papers.

II. Literature Review

1. Graph Contrastive Learning with Augmentations *(NIPS 2020)*

This is the main paper of our project. It introduced the state of art method of graph representation learning in 2020. The method is GraphCL which means graph contrastive learning. Additionally, they introduced 4 types of graph augmentation functions: Node drop, Edge perturbation, Attribute masking, and Subgraph. This model is made for improving the performance of graph representation learning. Especially, focused on semi-supervised learning and unsupervised learning. GraphCL is based on the previous well-known representation learning method: Deep Graph Infomax. It will be explained in detail in Method section.

2. Deep Graph Infomax *(ICLR 2019)*

Deep Graph Infomax (DGI) is a graph representation learning method that aims to learn meaningful and useful representations of nodes in a graph. It is designed to capture both local and global structural information in the graph, allowing for effective downstream tasks such as node classification, link prediction, and graph clustering. It is the fundamental and state of the art model of graph contrastive learning.

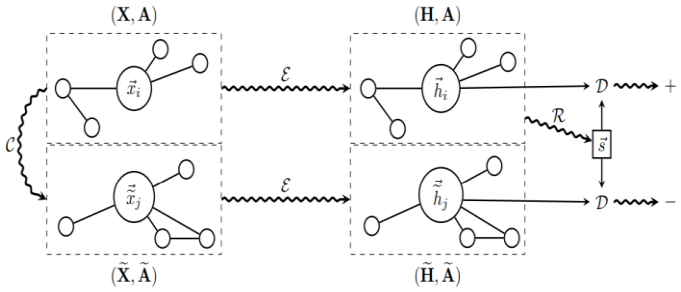

Figure 1: Deep Graph Infomax architecture

DGI consists of two main components: the encoder and the discriminator. The encoder is responsible for learning node representations by encoding both local and global information. It takes as input a graph with its node features and outputs a set of node embeddings. The encoder can be implemented using graph convolutional networks (GCNs) or other graph neural network (GNN) architectures.

The discriminator, on the other hand, aims to distinguish between positive (true) samples and negative (false) samples. Positive samples are pairs of nodes that are connected in the graph, while negative samples are pairs of nodes that are randomly selected. The discriminator is trained to classify whether a given pair of node representations comes from positive or negative samples. This architecture is applied to various graph represent learning tasks.

III. DATA

As we mentioned earlier, the graph data is slightly different with other data such as time series data or image data.

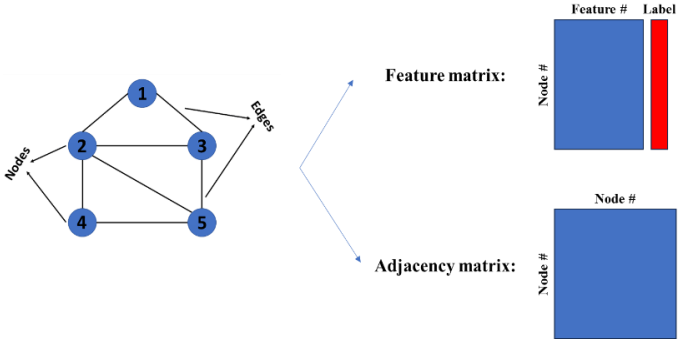

Figure 2: Graph data structure

Figure 2 shows that the graph data structure. Basically, graph data include feature matrix and adjacency matrix. Feature matrix is same with other time series data or image data. However, it is correlated with adjacency matrix. It represents the relationship between nodes. Now, we will show how graph data look like.

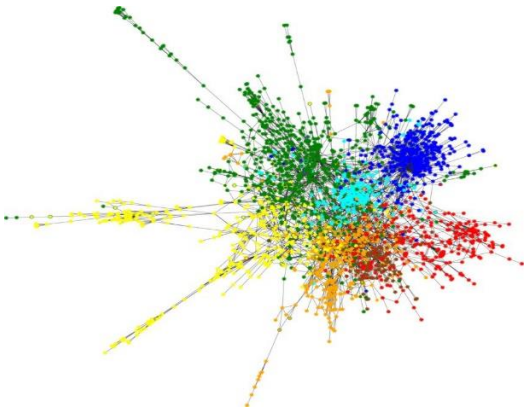

Figure 3: Citation graph data visualization

Figure 3 shows that the citation graph data structure represented in 3D vector space. Each node means publication and edge means citation information between the papers. The node color means the category of each paper. There are 6 class in this dataset. Our objective is to classify these papers into 6 categories correctly.

Citeseer Dataset:

```
citeseer = CitationGraphDataset('citeseer')

NumNodes: 3327
NumEdges: 9228
NumFeats: 3703
NumClasses: 6
NumTrainingSamples: 120
NumValidationSamples: 500
NumTestSamples: 1000
```

Cora Dataset:

```
cora = CoraGraphDataset('cora')

NumNodes: 2708
NumEdges: 10556
NumFeats: 1433
NumClasses: 7
NumTrainingSamples: 140
NumValidationSamples: 500
NumTestSamples: 1000
```

Figure 4: Citeseer and Cora dataset information

Citesser and Cora, these two datasets are most popular citation dataset. Figure 4 shows the Citeseer and Cora dataset information. There are number of nodes, number of edge, number of classes and number of train/valid/test samples. They have different number of classes.

IV. METHOD

1. Basic Graph Neural Network (GNN)

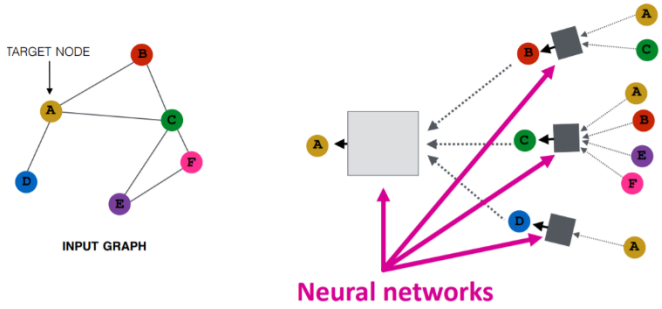

Figure 5: Graph Neural Network architecture

The Graph Neural Network (GNN) architecture consists of multiple layers, each performing message passing and aggregation operations on the graph data. Here is a brief overview of the typical GNN architecture:

1. Input Layer: The initial graph representation is provided as input to the GNN. Each node in the graph has associated features or attributes.

2. Graph Convolutional Layers: These layers are the core components of a GNN. Each layer performs message passing

between nodes, updating their representations based on the features of neighboring nodes. The update is typically performed by aggregating information from neighboring nodes and combining it with the current node's features. This process is iterated over multiple layers to capture complex patterns and dependencies.

3. Activation Function: Non-linear activation functions, such as ReLU (Rectified Linear Unit), are applied to the updated node representations to introduce non-linearity into the model.

4. Readout/Pooling Layer: In some cases, after the graph convolutional layers, a readout or pooling layer is used to aggregate node representations into a graph-level representation. This step is useful for tasks such as graph classification.

5. Output Layer: The final layer of the GNN produces the desired output, which depends on the specific task being performed. For example, it could be node labels, link predictions, or graph-level predictions.

Throughout the GNN architecture, the parameters (weights and biases) are learned through backpropagation and optimization algorithms, like traditional neural networks, to minimize a given loss function. This enables the GNN to learn and make predictions based on the input graph data.

2. Graph Contrastive Learning with Augmentations

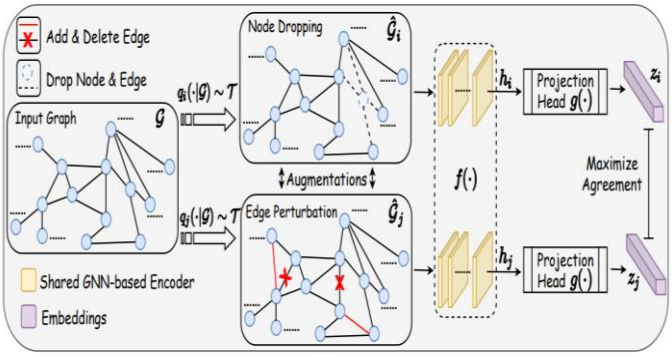

Figure 6: A framework of graph contrastive learning (GraphCL).

Figure 6 shows that two graph augmentations $q_i(\cdot|G)$ and $q_j(\cdot|G)$ are sampled from an augmentation pool T and applied to input graph G. A shared GNN-based encoder $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize the agreement between representations z_i and z_j via a contrastive loss. In this figure, two augmentation functions are used: Node dropping and Edge perturbation. At this time, we used GNN encoder as GIN (Graph Isomorphism Network) which is state-of-the-art method. The contrastive learning architecture is based on deep graph infomax. Therefore, this is unsupervised graph representation learning method.

Data augmentation	Type	Underlying Prior
Node dropping	Nodes, edges	Vertex missing does not alter semantics.
Edge perturbation	Edges	Semantic robustness against connectivity variations.
Attribute masking	Nodes	Semantic robustness against losing partial attributes.
Subgraph	Nodes, edges	Local structure can hint the full semantics.

Figure 7: Overview of data augmentations for graphs

Figure 7 shows that 4 graph augmentation functions: Node dropping, Edge perturbation, Attribute masking, and Subgraph. Also, there are underlying prior: these augmentation functions do not change the global semantics and do not affect original information. Our objective is classifying the negative samples. Therefore, augmented graph should keep original information. We explain these 4 graph augmentation functions in detail with code.

```python
def aug_drop_node(input_fea, input_adj, drop_percent=0.2):

    input_adj = torch.tensor(input_adj.todense().tolist())
    input_fea = input_fea.squeeze(0)

    node_num = input_fea.shape[0]
    drop_num = int(node_num * drop_percent)    # number of drop nodes
    all_node_list = [i for i in range(node_num)]

    drop_node_list = sorted(random.sample(all_node_list, drop_num))

    aug_input_fea = delete_row_col(input_fea, drop_node_list, only_row=True)
    aug_input_adj = delete_row_col(input_adj, drop_node_list)

    aug_input_fea = aug_input_fea.unsqueeze(0)
    aug_input_adj = sp.csr_matrix(np.matrix(aug_input_adj))

    return aug_input_fea, aug_input_adj
```

Figure 8: Node Dropping Function

Figure 8 shows that the node dropping function. First, we use the original input feature matrix, original input adjacency matrix and drop percent. Then, we set the drop node list using random sampling with drop percent. Then remove the node. As a result, both the dimension of feature matrix and adjacency matrix are augmented. Finally, we get the augmented feature matrix and augmented adjacency matrix.

```python
def aug_random_edge(input_adj, drop_percent=0.2):

    percent = drop_percent / 2
    row_idx, col_idx = input_adj.nonzero()

    index_list = []
    for i in range(len(row_idx)):
        index_list.append((row_idx[i], col_idx[i]))

    single_index_list = []
    for i in list(index_list):
        single_index_list.append(i)
        index_list.remove((i[1], i[0]))


    edge_num = int(len(row_idx) / 2)       # 9228 / 2
    add_drop_num = int(edge_num * percent / 2)
    aug_adj = copy.deepcopy(input_adj.todense().tolist())

    edge_idx = [i for i in range(edge_num)]
    drop_idx = random.sample(edge_idx, add_drop_num)


    for i in drop_idx:
        aug_adj[single_index_list[i][0]][single_index_list[i][1]] = 0
        aug_adj[single_index_list[i][1]][single_index_list[i][0]] = 0

    '''
    above finish drop edges
    '''
    node_num = input_adj.shape[0]
    l = [(i, j) for i in range(node_num) for j in range(i)]
    add_list = random.sample(l, add_drop_num)

    for i in add_list:

        aug_adj[i[0]][i[1]] = 1
        aug_adj[i[1]][i[0]] = 1

    aug_adj = np.matrix(aug_adj)
    aug_adj = sp.csr_matrix(aug_adj)
    return aug_adj
```

Figure 9: Edge Perturbation Function

Figure 9 shows that the edge perturbation function. First, we use the original input adjacency matrix and drop percent. Then, we set the remove/add edge list using random sampling with drop percent. There are two loops in this code. Upper loop is for dropping edge. On the other hand, below loop is for adding edge. Then, change 1 to 0 in adjacency matrix for dropping edge and change 0 to 1 in adjacency matrix for adding edge. As a result, only dimension of adjacency matrix is augmented. Finally, we get the augmented adjacency matrix.

```python
def aug_random_mask(input_feature, drop_percent=0.2):

    node_num = input_feature.shape[1]
    mask_num = int(node_num * drop_percent)
    node_idx = [i for i in range(node_num)]
    mask_idx = random.sample(node_idx, mask_num)
    aug_feature = copy.deepcopy(input_feature)
    zeros = torch.zeros_like(aug_feature[0][0])
    for j in mask_idx:
        aug_feature[0][j] = zeros
    return aug_feature
```

Figure 10: Attribute Masking Function

Figure 10 shows that the attribute masking function. First, we use the original input feature matrix, and drop percent. Then, we set the mask node list using random sampling with drop percent. Then, we change the feature value to 0 in feature matrix. As a result, only feature matrix is augmented. Finally, we get the augmented feature matrix.

```python
def aug_subgraph(input_fea, input_adj, drop_percent=0.2):

    input_adj = torch.tensor(input_adj.todense().tolist())
    input_fea = input_fea.squeeze(0)
    node_num = input_fea.shape[0]

    all_node_list = [i for i in range(node_num)]
    s_node_num = int(node_num * (1 - drop_percent))
    center_node_id = random.randint(0, node_num - 1)
    sub_node_id_list = [center_node_id]
    all_neighbor_list = []

    for i in range(s_node_num - 1):

        all_neighbor_list += torch.nonzero(input_adj[sub_node_id_list[i]], as_tuple=False).squeeze(1).tolist()

        all_neighbor_list = list(set(all_neighbor_list))
        new_neighbor_list = [n for n in all_neighbor_list if not n in sub_node_id_list]
        if len(new_neighbor_list) != 0:
            new_node = random.sample(new_neighbor_list, 1)[0]
            sub_node_id_list.append(new_node)
        else:
            break


    drop_node_list = sorted([i for i in all_node_list if not i in sub_node_id_list])

    aug_input_fea = delete_row_col(input_fea, drop_node_list, only_row=True)
    aug_input_adj = delete_row_col(input_adj, drop_node_list)

    aug_input_fea = aug_input_fea.unsqueeze(0)
    aug_input_adj = sp.csr_matrix(np.matrix(aug_input_adj))

    return aug_input_fea, aug_input_adj
```

Figure 11: Subgraph Function

Figure 11 shows that the subgraph function. First, we use the original input feature matrix, original input adjacency matrix and drop percent. Then, we set the sub-node list using random sampling with drop percent. Additionally, we set the one center node randomly. Then, we sequentially add the sub-nodes from the center node. As a result, both the dimension of feature matrix and adjacency matrix are augmented. Finally, we get the augmented feature matrix and augmented adjacency matrix. Subgraph function is similar to node dropping function, but it has strategies to drop node.

3. Subgraph Augmentation with PageRank

We will adopt PageRank algorithm to subgraph augmentation function. Original subgraph augmentation function selects the center node randomly. However, the center node is important in subgraph algorithm. Because if the central node is selected as a node with little alternation, the subgraph may not represent the entire graph. Therefore, we adopt new strategy PageRank to select center node, not random. First of all, we will study basic PageRank algorithm.

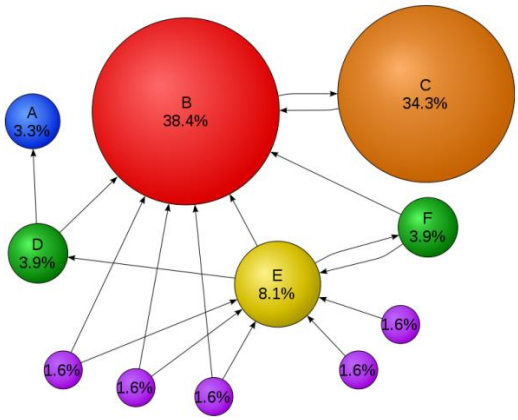

Figure 12: PageRank visualization

Figure 12 shows that the graph structure and its PageRank. As shown in figure, the larger the number of connected edges, the larger the PageRank. Now we study the PageRank algorithm in detail.

- Stochastic adjacency matrix $\mathbf{M}$
  - Let page i has d_i out-links

$$\mathbf{M}_{ji} = \begin{cases} \frac{1}{d_i} & \text{if } i \to j \\ 0 & \text{otherwise} \end{cases}$$

  - $\mathbf{M}$ is a column stochastic matrix (column sum to 1)
- Rank vector $\mathbf{r}$: vector with an entry per page
  - r_i is the importance score of page i
  - $\sum_i r_i = 1$
- The flow equation $r_j = \sum_{i \to j} \frac{r_i}{d_i}$ can be written as

$$\mathbf{r} = \mathbf{M}\mathbf{r}$$

Figure 13: PageRank: Matrix formulation

Figure 13 shows the PageRank algorithm. First, we need adjacency matrix M. Then, flow equation can be written as r=Mr.

- NOTE: x is an eigenvector of A with the corresponding eigenvalue λ if: $Ax = \lambda x$
- Flow equation in the matrix form: $\mathbf{M}\mathbf{r} = \mathbf{r}$
- The rank vector $\mathbf{r}$ is an eigenvector of the stochastic web matrix $\mathbf{M}$
  - In fact, its first or principal eigenvector, with corresponding eigenvalue 1
  - Largest eigenvalue of $\mathbf{M}$ is 1 since $\mathbf{M}$ is column stochastic. We know $\mathbf{r}$ is unit length and each column of $\mathbf{M}$ sums to one, so $\mathbf{M}\mathbf{r} \leq 1$
- We can now efficiently solve for $\mathbf{r}$ through *power iteration*

Figure 14: PageRank: Eigenvector method

As a result, rank vector r is an eigenvector of M. At this time, we can efficiently solve for r through power iteration.

- Power iteration: a simple iterative scheme
  - Suppose there are N web pages
  - Initialize: $\mathbf{r}^{(0)} = [1/N, \ldots, 1/N]^\top$
  - Iterate: $\mathbf{r}^{(t+1)} = \mathbf{M}\mathbf{r}^{(t)}$, i.e.,

$$\mathbf{r}_j^{t+1} = \sum_{i \to j} \frac{\mathbf{r}_i^{(t)}}{d_i}, \quad d_i : \text{out-degree of node i}$$

  - Stop when $\|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}\|_1 \leq \epsilon$

Figure 15: PageRank: Power iteration

Figure 15 show the power iteration method. It is effective way to derive eigenvector in only small size M. Because it has the disadvantage of not converging. The larger the size of M, the higher the probability. Therefore, we should use original way Gauss elimination to derive eigenvector.

At this time, we define the PageRank as eigenvector which derived from maximum value of eigenvalue. Because we can think that maximum eigenvalue means the most representative eigenvector.

Our strategy is that we adopt the maximum PageRank node to center node. The prior is this strategy will make good subgraph sample that represents the entire graph well. Out strategy will be proved through experiment in result section.

```python
#subgraph 수정
def aug_subgraph(input_fea, input_adj, drop_percent=0.2):

    matrix_D_array = input_adj.todense()
    matrix_D_array = np.nan_to_num(matrix_D_array)

    # L 에 대한 eigenvalue, eigenvector 계산
    eigen_value , eigen_vector = np.linalg.eig( matrix_D_array )

    # eigen value 내림차순 정렬
    order = np.absolute(eigen_value).argsort()[::-1]

    # 정렬 순서에 따라 재정렬
    eigen_value = eigen_value[order]
    eigen_vector = eigen_vector[:,order]

    # 첫번째 eigen value 에 대한 eigenvector 추출 및 비중확인
    r = eigen_vector[:,0]  # 0번째 열
    value = 100*np.real(r/np.sum(r)) ## np.real : 복소수 인수의 실수부를 반환
    print(value) #pagerank

    #-------------------------------------------------------------------------------
    input_adj = torch.tensor(input_adj.todense().tolist())
    input_fea = input_fea.squeeze(0)
    node_num = input_fea.shape[0]

    all_node_list = [i for i in range(node_num)]
    s_node_num = int(node_num * (1 - drop_percent))
    center_node_id = value.tolist().index(max(value)) ### 수정됨
    #center_node_id = random.randint(0, node_num - 1) ### 오리지널
    sub_node_id_list = [center_node_id]
    all_neighbor_list = []

    for i in range(s_node_num - 1):

        all_neighbor_list += torch.nonzero(input_adj[sub_node_id_list[i]], as_tuple=False).squeeze(1).tolist()

        all_neighbor_list = list(set(all_neighbor_list))
        new_neighbor_list = [n for n in all_neighbor_list if not n in sub_node_id_list]
        if len(new_neighbor_list) != 0:
            new_node = random.sample(new_neighbor_list, 1)[0]
            sub_node_id_list.append(new_node)
        else:
            break
```

Derive PageRank (Using eigenvector of adjacency matrix)

Adopt maximum PageRank to center node

Figure 16: Subgraph augmentation function with PageRank

Figure 16 shows the implementation of subgraph augmentation function with PageRank. Upper code is for derive PageRank from input adjacency matrix. After derive PageRank, we adopt it to center node. Originally, center node was determined by random selection.

4. Adopt Softmax regression for evaluation

As we mentioned before, Grpah CL is unsupervised model. Therefore, we cannot use directly to classification task. To solve this problem, the paper introduces adopting the simple Softmax regression which has only one hidden layer after GraphCL. Final output of GraphCL is low dimensional representative vector **z**. The idea is that input **z** to input layer of Softmax regression. Then we can classify the categories of nodes.

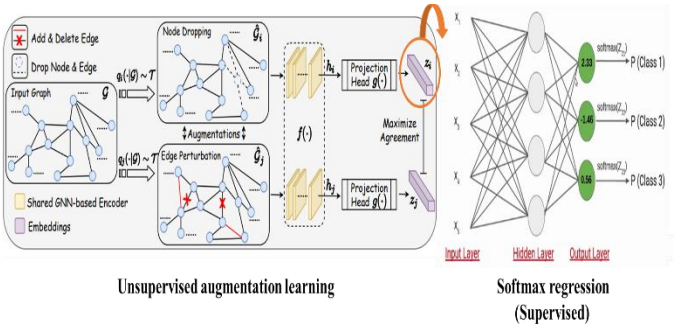

Figure 17: GraphCL with Softmax regression

Figure 17 shows the final GrpahCL classification model structure with Sotfmax regression. Of course, there are learning parameters in Softmax regression. However, our objective is not focus on learning the Softmax regression. Moreover, the effect Softmax regression has on classification accuracy is very small due to only one hidden layer. Therefore, we can say that this final method is almost affected by GraphCL algorithm. In conclusion, final classification accuracy is almost depending on GraphCL performance.

V. Experiment Results

1. GraphCL with Original Augmentations

1-1. Citeseer Dataset

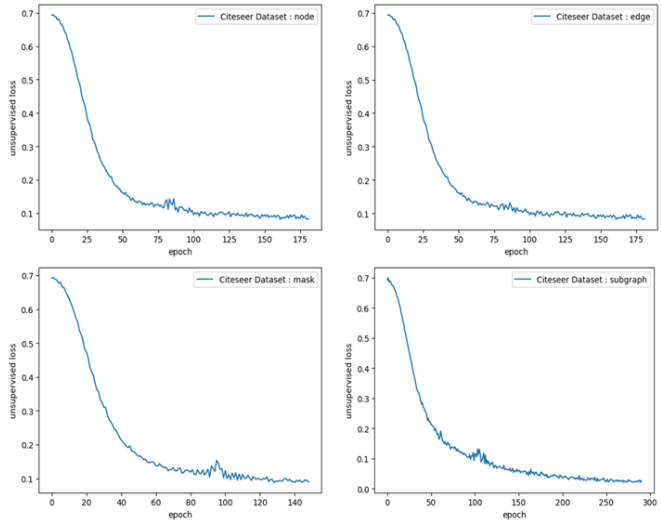

Figure 18: GraphCL unsupervised loss in Citeseer dataset

Figure 18 shows the GraphCL performance in Citeseer dataset. All augmentation functions have similar unsupervised performance.

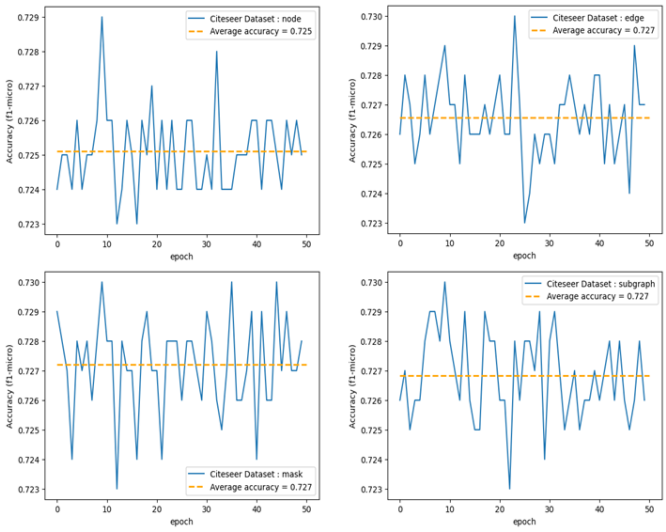

Figure 19: Final classification average accuracy in Citeseer dataset

Figure 19 shows the average classification accuracy. Node dropping has a little bit low performance as 0.725. Other functions have about 0.727. It is almost same with original "Graph Contrastive Learning with Augmentations" paper results of 0.74. It means our implementation was successful.

1-2. Cora Dataset

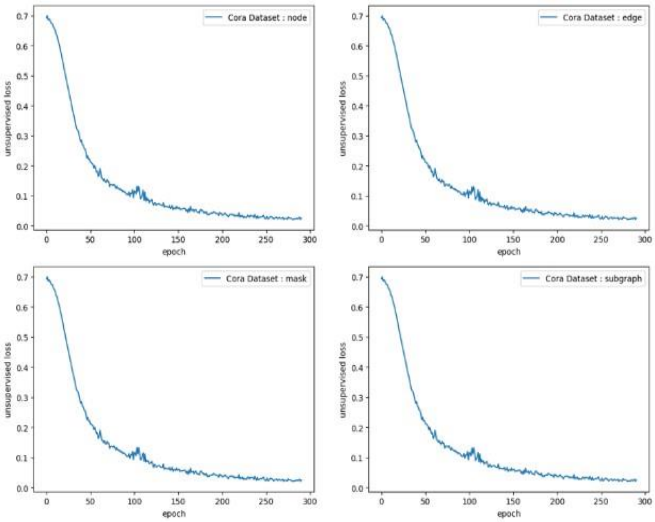

Figure 20: GraphCL unsupervised loss in Cora dataset

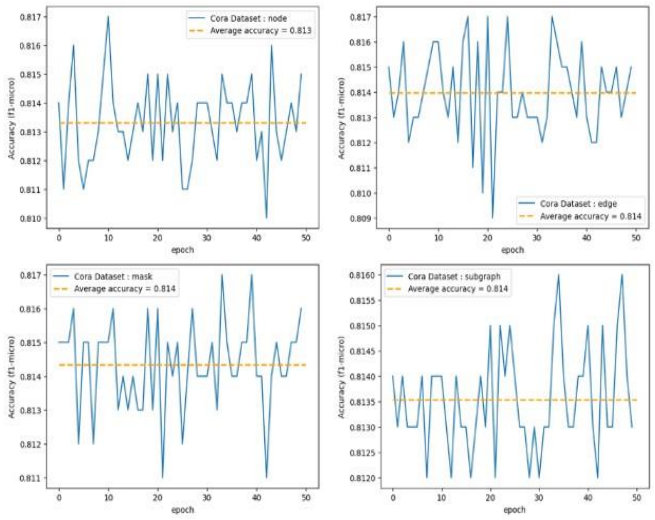

Figure 21: Final classification average accuracy in Cora dataset

Figure 20 and Figure 21 show the loss function and average accuracy in Cora dataset. The shape is almost same with Citation dataset. Average accuracies are close to 0.814. It is also same with original paper result of 0.83.

2. Modified GraphCL: Subgraph with PageRank

2-1. Citeseer Dataset

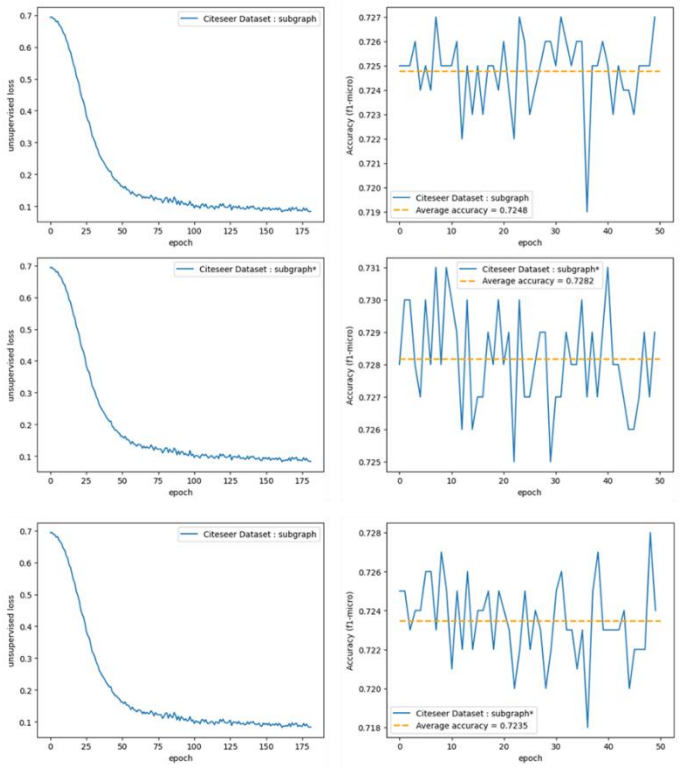

Figure 21: Modified GraphCL accuracy in Citeseer dataset (Subgraph with PageRank, From top to bottom: Random, Max PageRank, Min PageRank)

Now, we check our modified Grpah CL performance. Figure 21 shows that loss function and average accuracy of modified Grpah CL model with PageRank algorithm in the Citeseer daaset. From top to bottom, original model, adopt maximize PageRank to center node and adopt minimize PageRank to centre node. Accuracy of minimize PageRank was lowest and accuracy of maximize PageRank was highest. It means our PageRank subgraph sampling method is effective. The performance became better. However, the change amount is very small. Therefore, we need to check the performance with the bigger dataset. It is for future work section.

2-2. Cora Dataset

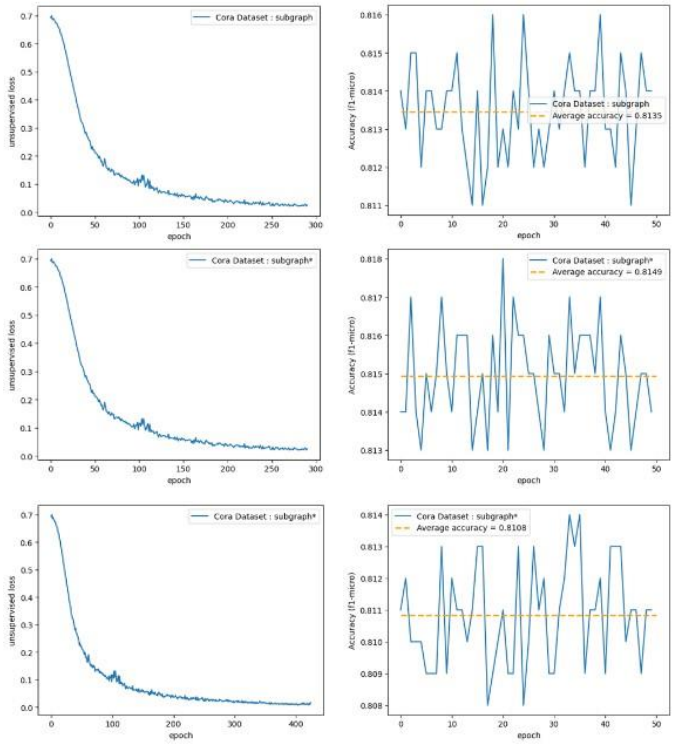

Figure 22: Modified GraphCL accuracy in Cora dataset (Subgraph with PageRank, From top to bottom: Random, Max PageRank, Min PageRank)

Figure 22 shows that loss function and average accuracy of modified Grpah CL model with PageRank algorithm in the Cora daaset. The result was almost same with Citeseer dataset.

VI. FUTURE WORK

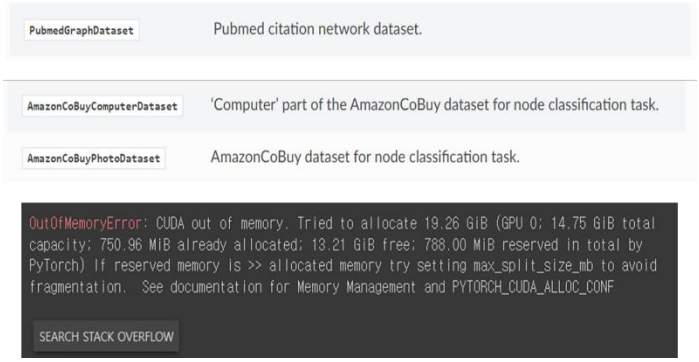

Figure 23. Example of bigger graph dataset and error code

We couldn't make implementation on more extensive datasets such as 'PubMed' or 'Amazon' by memory issue. We want to experiment our revised model on bigger dataset than citation data.

VII. CONCLUSION

We studied unsupervised graph representation leaning based on Graph Contrastive Learning with Augmentations paper. We implemented 4 types of graph augmentation functions: Node drop, Edge perturbation, Attribute masking and Subgraph. The result was almost same with original paper result. Additionally, we focused on subgraph functions to improve the performance. We adopt PageRank algorithm to subgraph sampling method. We adopt maximum and minimum PageRank node to canter node. Originally, random node was selected. Then, we compared the three results. The performance became better when using maximum PageRank. Therefore, we verified our PageRank algorithm is effective to subgraph sampling. However, we need to experiment with the bigger dataset to prove our method.

REFERENCES

[1] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, Yang Shen, "Graph Contrastive Learning with Augmentations" *(NeurIPS 2020)*

[2] Petar Veličković, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, R Devon Hjelm, "Deep Graph Infomax" *(ICLR 2019)*

[3] Link Analysis Lecture Note, PKU.edu
http://faculty.bicmr.pku.edu.cn/~wenzw/bigdata/lect-link.pdf