AIM 5056-41: Homework 1

2020712702 Gahyung Kim(김가형)

1. Over-smoothing problem of GCN

Since GCN trains certain node's representation by multiplying it with neighboring node's representations, after repeating this process for multiple times, certain node's representation become similar and goes indistinguishable with other nodes. This problem is called over-smoothing problem and is caused of major performance drop in deep graph convolutional networks.

The code I used to observe over-smoothing problem in GCN is "pygcn", which is a Pytorch implemented code of Graph Convolutional Networks (GCNs) from the paper Thomas Kipf published in ICLR 2017 called "Semi-supervised Classification with Graph Convolutional Networks [1]."

```python
class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, num_layers, dropout):
        super(GCN, self).__init__()
        self.convs = torch.nn.ModuleList()
        self.convs.append(GraphConvolution(nfeat, nhid))
        for _ in range(num_layers -2):
            self.convs.append(GraphConvolution(nhid, nhid))
        self.convs.append(GraphConvolution(nhid, nclass))
        self.dropout = dropout
        self.num_layers = num_layers

    def reset_parameters(self):
        for conv in self.convs:
            conv.reset_parameters()

    def forward(self, x, adj):
        for conv in self.convs[:-1]:
            x = conv(x,adj)
            x = F.relu(x)
            x = F.dropout(x, p = self.dropout, training = self.training)
        x = self.convs[-1](x,adj)
        np.save(f'embedding/GCN_{self.num_layers}',x.cpu().detach().numpy())

        return F.log_softmax(x, dim=1)
```

To observe the embeddings of GCN with different hidden layer numbers, I changed the model.py code in pygcn like the picture on the left side. By getting num_layers arguments from the main.py, first made self.convs with empty torch.nn.ModuleList() and append the GraphConvolution function with the respect to number of hidden layers. And after finishing the forward part, I saved the embeddings as .npy file format in embedding folder. I decided to compare six different number of hidden layers from 2 to 7.

After saving embeddings of GCN in six different number of hidden layers into .npy file, I visualized this embedding space through using UMAP library. The result is shown in Figure 1.

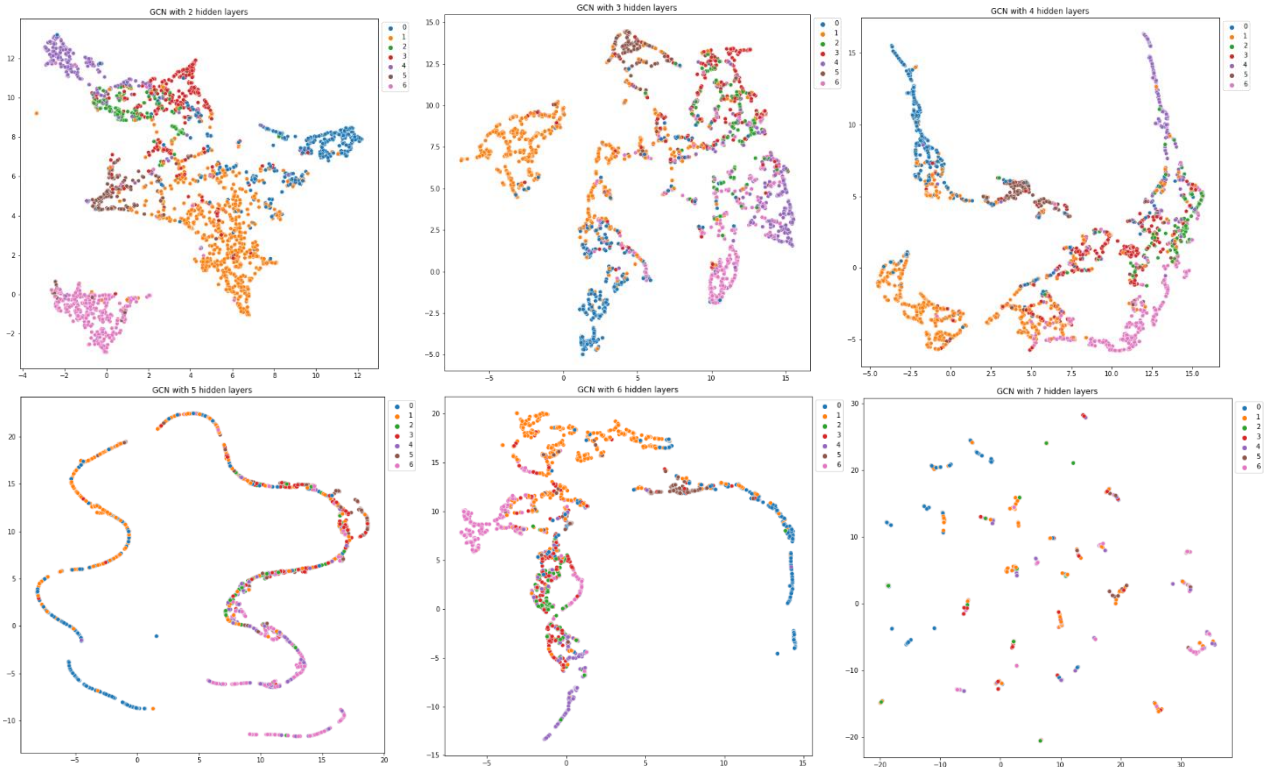

Figure 1. UMAP of Embeddings from different hidden layers. From the upper left, it is the UMAP of GCN with 2 hidden layers and the lower right UMAP is from GCN with 7 hidden layers. It is clear that as the layers go deeper, the embedding of each label spreads through the space and does not cluster well.

As we can see in Figure 1. embeddings in first row (2, 3, and four hidden layers) seem well clustered depending on labels and well separated throughout the space. However, starting from 4 hidden layers, embeddings are hard to distinguish between labels and especially in the last picture, almost all the embeddings of labels are scattered throughout the space, meaning that representations of each node do not well train the latent characteristics between labels and became indistinguishable. This is due to over-smoothing problem in GCN.

In order to compare the test accuracy of GCN with different number of hidden layers, I used tensorboard library to save scaler values of accuracy and loss changes in each epoch. The results are shown in Figure 2 and Figure 3.

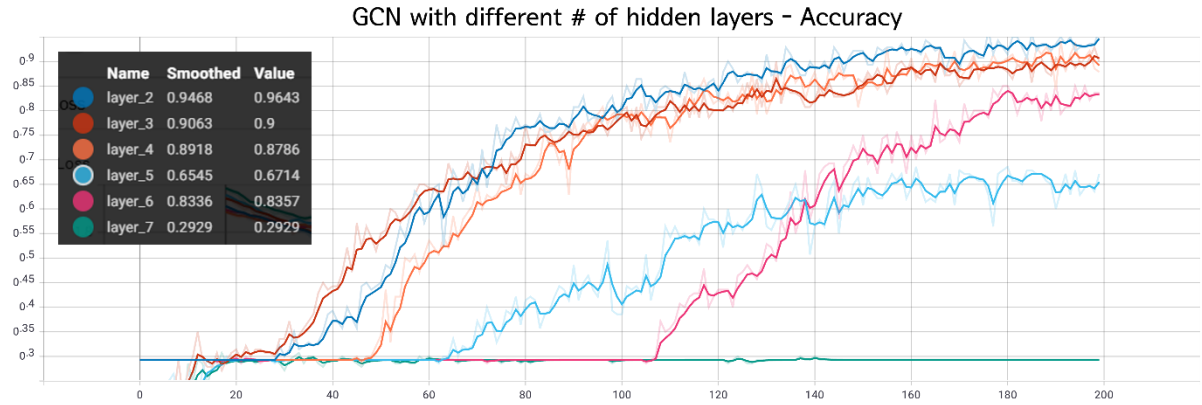

Figure 2. Accuracy of GCN with different number of hidden layers. The final test accuracy of GCN with different number of hidden layers. The dark blue line (layer_2) has the highest accuracy and as the number of hidden layers increases, the accuracy drops drastically.

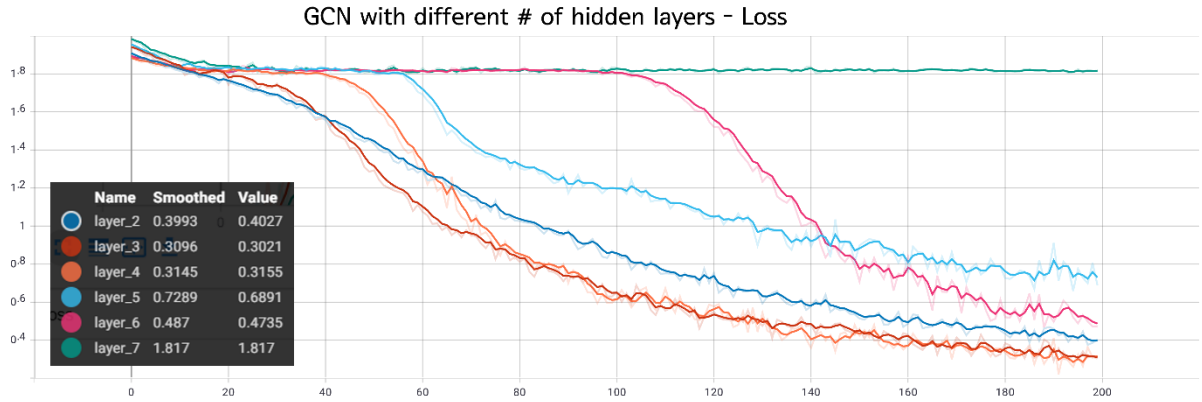

Figure 3. Loss of GCN with different number of hidden layers. The loss of GCN with different number of hidden layers. Loss of GCN with seven hidden layers does not change much during training, meaning that the model barely learns and does not coverage well.

As you can see, the accuracy of the test dataset was highest when using GCN with two hidden layers and lowest when using GCN with seven hidden layers. Also, in the case of Loss value, GCN with three hidden layers has the lowest loss value and GCN with seven hidden layers shows the highest value. Actually, GCN with seven hidden layers is barely trained because the loss and accuracy value does not change much even after several epochs.

GCN	2 Layers	3 Layers	4 Layers	5 Layers	6 Layers	7 Layers
Test Loss	0.7972	0.8472	1.3352	1.4345	1.0713	1.8897
Test Acc.	0.821	0.766	0.549	0.47	0.711	0.309

Table 1. Test Result of GCN with different number of hidden layers.

The final loss and accuracy of different hidden layer numbers are shown in Table 1 as well.

2. Skip-connection

In order to make my own skip-connection model, I first tried several experiments in different GCN models and different hyperparameter setting. The results are shown in Table 3 in Appendix. The first attempt I tried was a simple skip-connection in every layer change, adding previous h_i to current h_{i+1}. The result was great among other attempts. Test accuracy of Plus-GCN stays in between 0.75 to 0.83 even in seven hidden layers. This shows why skip-connections could be the great solution to mitigate over-smoothing problem in GCN. Then what about multiplication? I tried element-wise product (Hadamard Product) with h_i and h_{i+1} and realized that it is not helpful at all.

Then, I read "Simple and Deep Graph Convolutional Networks [2]" paper published in ICML 2020 which proposed a novel model called GCNII which could effectively relieve over-smoothing problem. They proposed two simple methods called Initial Residual Connection and Identity mapping. The equation is shown below.

$$\mathbf{H}^{(\ell+1)} = \sigma\Big(\underbrace{\big((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)} + \alpha_\ell \mathbf{H}^{(0)}\big)}_{\text{Initial Residual Connection}}\underbrace{\big((1-\beta_\ell)\mathbf{I}_n + \beta_\ell \mathbf{W}^{(\ell)}\big)}_{\text{Identity Mapping}}\Big)$$

They used two hyperparameter in this equation, alpha and beta. I find Initial Residual Connection interesting because, rather than adding previous representation, it adds initial representation to keep remind the node to contain its own original information. However, I was curious why they used fixed value of hyperparameter in Initial Residual Connection when they argued what fixed coefficients limit the expressive power of multi-layer GCN model. Thus, I decided to change alpha from hyperparameter into weight parameter by applying linear layer to alpha in calculating Initial Residual Connection, so that the amount of initial representation to be added can be arbitrary depending on model's learning. I called this model alpha-free GCNII and tested its ability to mitigate over-smoothing problem. The embedding space of alpha-free GCNII in six different number of hidden layers, from 2 to 7, are shown in Figure 4. As you can see, even though the clustering is not clearly shown, the embeddings of nodes are well separated regardless of number of hidden layers. Accuracy and Loss value of Alpha-free GCNII are shown in Figure 5 and 6.

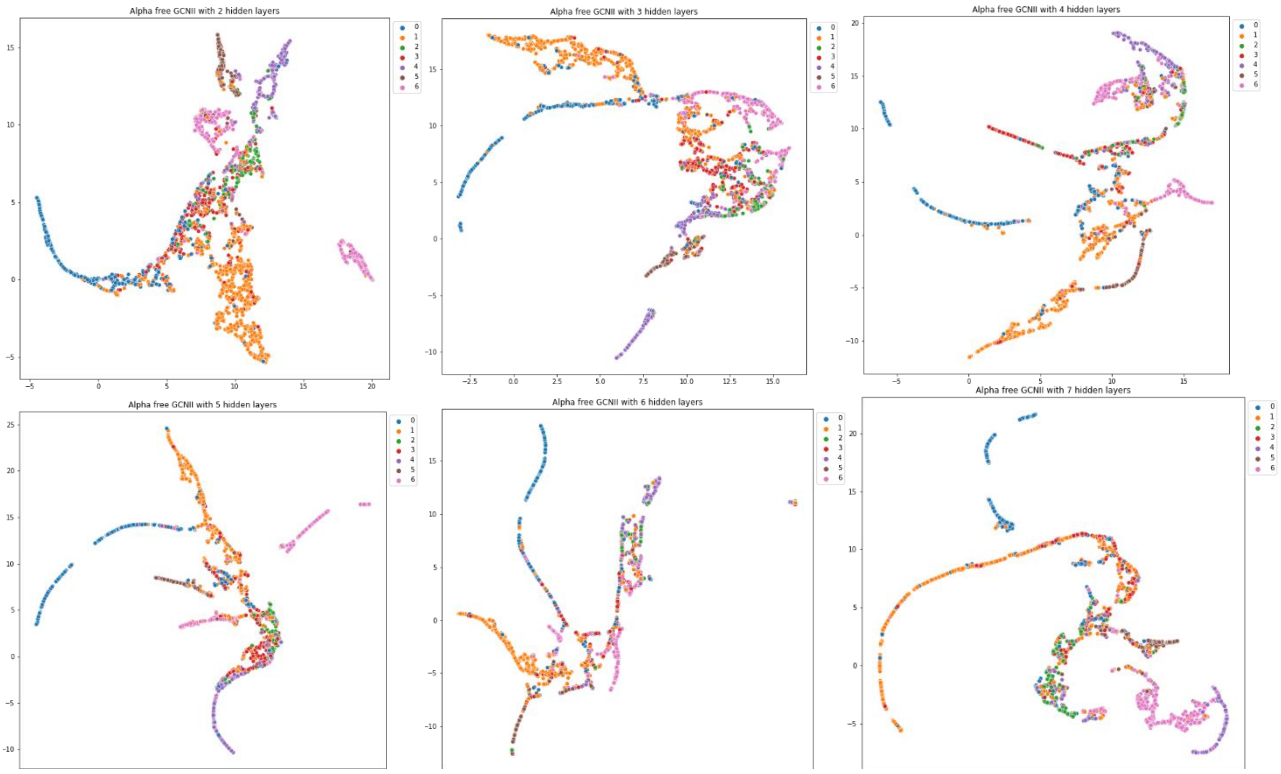

Figure 4. UMAP of Embeddings from different hidden layers. From the upper left, it is the UMAP of alpha-free GCNII with 2 hidden layers and the lower right UMAP is from alpha-free GCNII with 7 hidden layers.

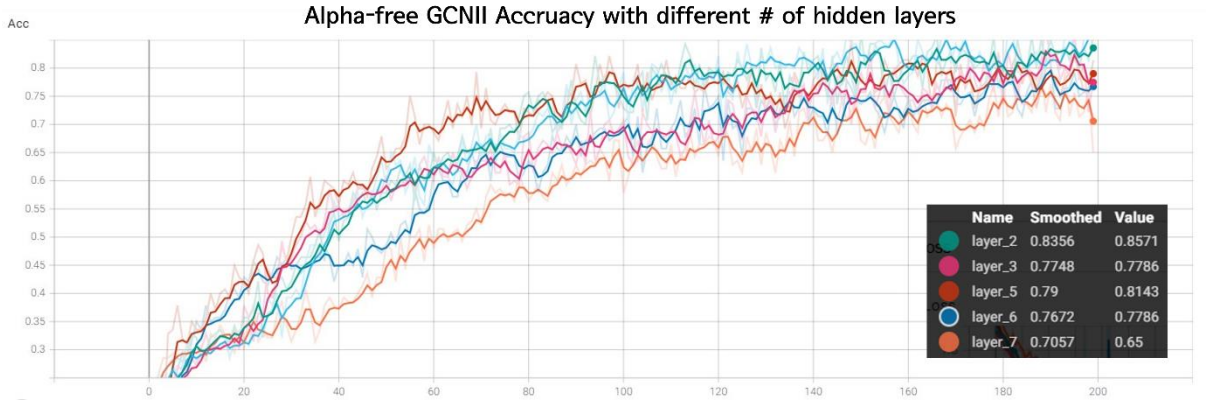

Figure 5. Accuracy of alpha-free GCNII with different number of hidden layers. The final test accuracy of alpha-free GCNII with different number of hidden layers.

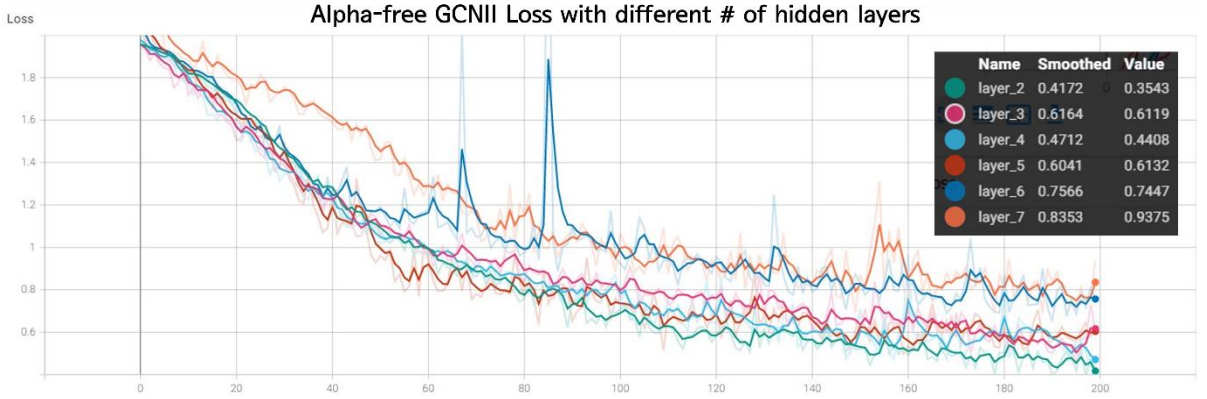

Figure 6. Loss of alpha-free GCNII with different number of hidden layers. The loss of alpha-free GCNII with different number of hidden layers.

Even though the accuracy drops as the model deepens, it sustains test accuracy in between 0.83 and 0.75 without drastic decrease. Also in the case of loss value the value decreases well as the training goes on, meaning that alpha-free GCNII trains well even when the model deepens. The specific values of loss and accuracy is shown in Table 2.

Alpha-free GCNII	2 Layers	3 Layers	4 Layers	5 Layers	6 Layers	7 Layers
Test Loss	0.9032	0.9679	1.1195	0.884	0.8927	1.0957
Test Acc.	0.733	0.753	0.713	0.752	0.731	0.665

Table 2. Test Result of Alpha-free GCNII

Lastly, I tried two different hyperparameter setting in GCNII because in "Design space for graph neural networks [3]" paper, SiLU activation function and 0.6 dropout rate shows better performance than ReLU and 0.5 dropout rate. However, the accuracy and loss value of GCNII+SiLu and GCNII+dropout 0.6 showed lower performance than alpha-free GCNII model, so maintain using the initial hyper-parameter setting in GCNII except alpha which I changed into model's parameter which can adaptively change depending on models' training.

Appendix

Model	Loss/Acc	2 Layers	3 Layers	4 Layers	5 Layers	6 Layers	7 Layers
Plus GCN	Test Loss	0.5988	0.8243	1.2243	1.3893	1.7667	3.1652
	Test Acc	0.833	0.784	0.762	0.762	0.746	0.754
Hadamard product GCN	Test Loss	0.7251	0.8463	1.4693	1.8833	1.8865	1.8897
	Test Acc	0.824	0.749	0.549	0.309	0.309	0.309
GCNII	Test Loss	0.7495	0.96	1.0282	0.9149	0.9897	1.0228
	Test Acc	0.774	0.746	0.704	0.757	0.712	0.712
GCNII+SiLU	Test Loss	0.8703	0.8505	0.9682	0.9369	1.02	0.8537
	Test Acc	0.722	0.74	0.741	0.728	0.66	0.749
GCNII+dropout=0.6	Test Loss	0.8296	1.1265	1.303	1.2277	1.2677	1.3684
	Test Acc	0.767	0.61	0.58	0.542	0.529	0.478

Table 3. Test Result of various models and hyper-parameters

Reference

[1] Kipf, Thomas and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks.” *ArXiv* abs/1609.02907 (2017)

[2] Chen, Ming, et al. "Simple and deep graph convolutional networks." *International Conference on Machine Learning*. PMLR, (2020).

[3] You, Jiaxuan, Zhitao Ying, and Jure Leskovec. "Design space for graph neural networks." *Advances in Neural Information Processing Systems* 33 (2020).