DSC180B Report

Xinrui Zhan

Li Shang

3D Points Clouds Classification with Graph Neural Network

1. Abstract

This research focuses on 3D shape classification. Our goal is to predict the category of shapes consisting of 3D data points. We aim to implement Graph Neural Network models and compare the performances with the PointNet, a popular architecture for 3d points cloud classification tasks. Not only will we compare standard metrics such as accuracy and confusion matrix, we will also explore the model's resilience on data transformation. What's more, we tried combining PointNet with graph pooling layers. Our experiment shows that even though PointNet has a higher accuracy overall, GCN has much more reasonable misclassification and is much more robust to data augmentation.

2. Introduction

Currently, 3d data has been widely used in a lot of industries' fields such as automatic-driving, virtual-reality, gesture recognition, etc. There are two major structures to represent 3d data: the first one is 3d grids and the second one is 3d points cloud. 3d grids are created based on 2d images. Instead of using a 2d array to represent the pixel values in the 2d grid, the 3d grid uses a 3d array to represent 'true/false' or pixel value in a specific location. Since the similarities between the data structure, lots of researchers tried to apply Convolutional Neural Network which originally developed on images on 3D grids. There are lots of successful works. However, 3d grid is not a memory-efficient data structure for storing 3d shapes, and thus

3d CNN also wastes a lot of time/memory if used for predicting the 3d shapes. For example, we now only have 4 points {(100, 100), (100, -100), (-100, 100), (-100, -100)} representing a cube in 3d space. If we want to use 3d grids representing this shape, we need to create a 100*100*100 arrays for storing information and 99.99% of the memory used are waste.

The 3d points cloud is created to save 3d shapes which solves the problem that 3d grids have. That is the reason why most of the 3d radar in the industry use this structure, and why it is so important to have models that make predictions based on it. The 3d point cloud is just a set of 3d points where all points are like (x,y,z). A famous model for 3d points cloud data is PointNet. It is used for shapes classification, objects segmentation, and scene segmentations. It is a model which is designed directly based on the set of the 3d points. However, 3d points cloud could also be seen as a graph. The nodes are the points and the edges represent the connection between points. In this project, we aim to use Graph Convolutional Network to predict 3d shapes based on 3d point cloud data. We will compare our results with our baseline model - PointNet and we will also try to combine graph layers and Pointnet.

In this report, the Data section will discuss the dataset we used. The Graph Construct section will talk about how we construct graphs on points clouds. The Model section will discuss the models' architecture we designed, and finally the Result section will discuss the running outcomes and performance of our models. We also provide links for the code and the website in the appendix.

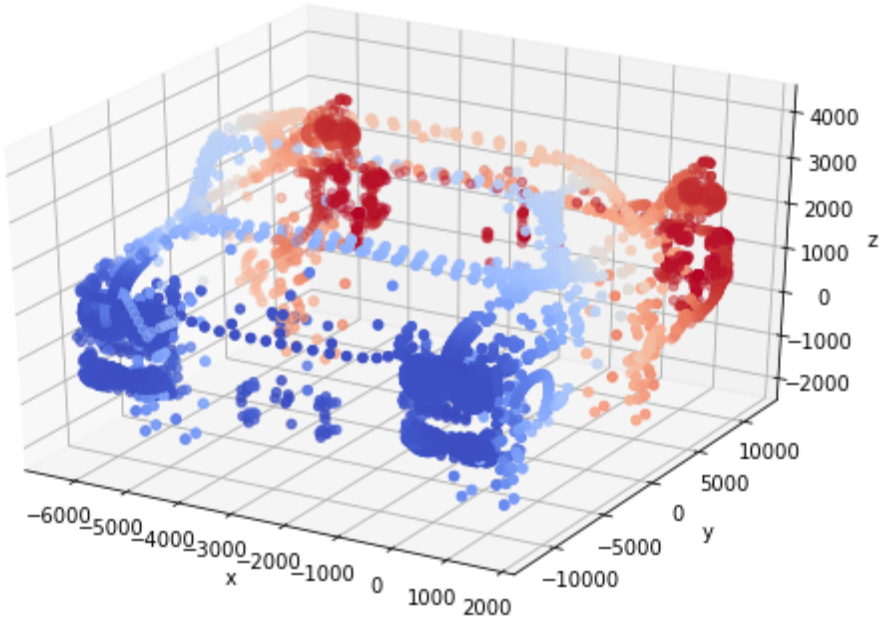
3. Data

In this project, we will mainly use the ModelNet40 as our dataset. This dataset contains 40 categories such as airplane, TV_stand, guitar, etc. Each category has around 500 samples and

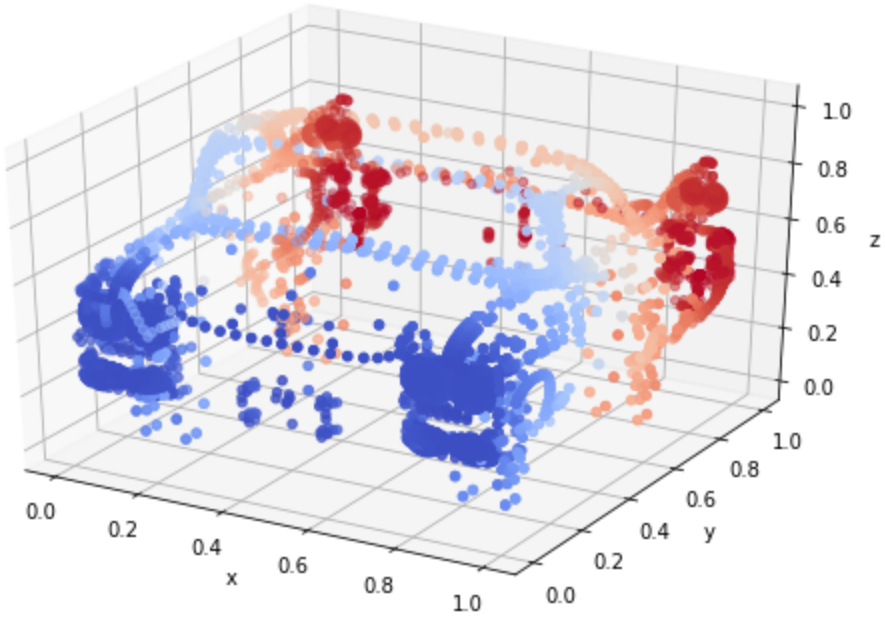
every sample contains 3k-80k points. Points are all in the format of euclidean space coordinates (x,y,z). The range of the coordinates is not unified and varied a lot. Some samples may have points all in the range in (-100, -100, -100) to (100, 100, 100) while some may have (-4000, -4000, -4000) to (4000, 4000, 4000). This difference could result in imbalance models' parameters training and thus we used min-max normalization to normalize all points in all samples to range (0, 0, 0) to (1, 1, 1). Not only could normalization help us train a balance model, but also it will be convenient for us with the graph construction. Since the fix-radius graph construction needs a unique hyper-parameter r, instead of finding a best choice of r for all samples, we could simply use one number if we normalized our data first.

Sampling is also a key procedure. Samples with categories like air_plane, cars, or guitar usually have 60k+ points. Using 60k+ points will construct a graph with 60k+ nodes and 600k-1200k+ edges. The data dimension is too big to train the model efficiently and at the same time, increase the running time for data-loading and graph-construction. Not to mention that most of the points are unnecessary in model's training. In lots of cases, datasets include the inner points, which do not contribute to the determination of the category's shape. What is matter are those surface points which construct the shape of the sample. In our project, we decide to randomly select 1000 points for all samples. Some may doubt that the size may be too small while our baseline model PointNet shows that 1000 points are indeed sufficient. What's more, since we regularize the number of points, we do find implementing the model's architecture more easily since we have a unified input shape.

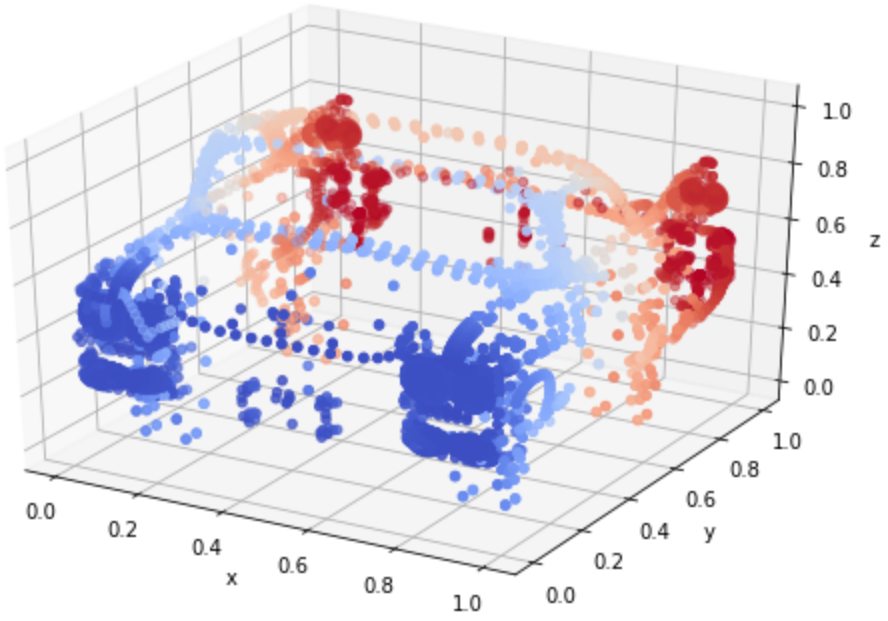
Below are three images of the raw data, the normalized data, and the sampled normalized data. We could see that even after the sampling process, we still could clearly tell the object's shape.



Graph 3.1: Raw data; “Car”



Graph 3.2: Normalized data; “Car”



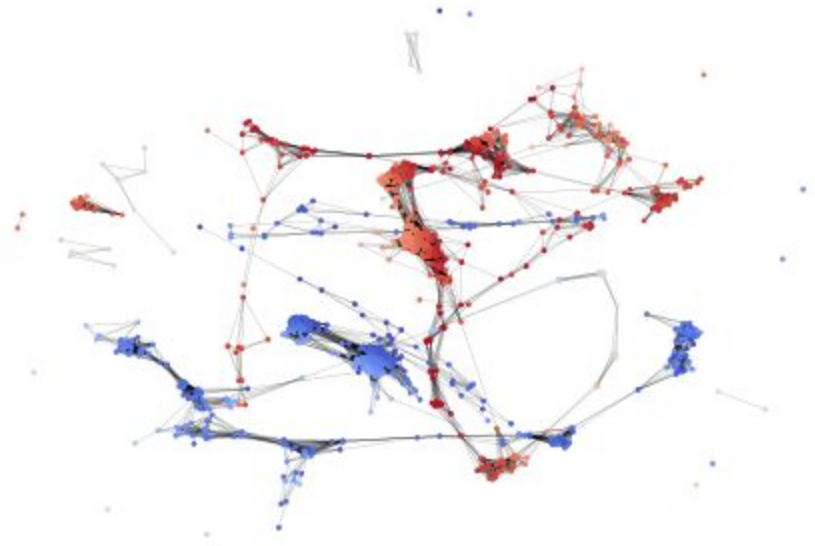
Graph 3.3: Normalized and Sampled data; 'Car'

After models-training, we also tested our model's performance on a different dataset ShapeNet. ShapeNet is a public 3d points cloud data and shares a lot of the same categories with ModelNet40. Through testing on samples that models have not seen in the training process, we could measure our model's elasticity.

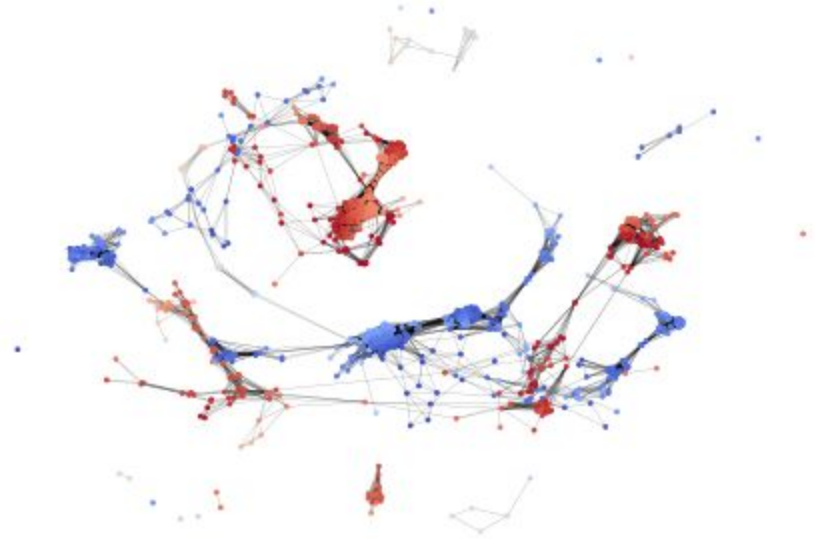
Data augmentation is also used in this project to improve the accuracy and measure models' resilience. Our goal is to train models predicting the shape of a point cloud. Therefore, if the incoming points: 1. Globally translate with one direction; 2. Rotate a certain angle; 3. Stretch or squeeze with a degree in certain ranges, our model should be able to predict the same result. In this project, normalization will help us deal with transformation and we used the data augmentation testing the left two.

4. Graph Construct

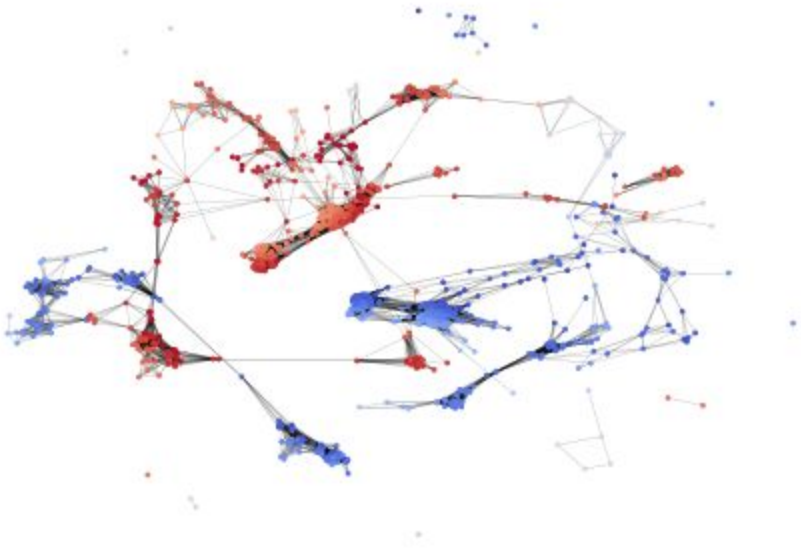
In order to train a Graph Neural Network, we need first convert our dataset into a collection of graphs. To be more specific, we need an adjacency matrix or adjacency list to represent the edges' information. In this project, we used two different ways to construct the graph. The first one is the fixed-radius distance. We calculated the euclidean distance between each node and set a threshold. Any node that is farther away from one node than this distance will not be considered as one's neighbor. For nodes that are the range of this distance, we will store the distance either by a distance matrix or an adjacency list. The second method is using k-nearest-neighbors to find k most nearest neighbors of one node. Then storing either the distance or simply binary information as a matrix or adjacency list. The advantage of using fix-radius is that for different nodes, we can have different connection density. Thus we could distinguish nodes' popularity. For example, a node with lots of neighbors may be located right in the center of the shape while a node with few neighbors is in the corner. From the model's training perspective, this could help us aggregate the node's features' information. The features of nodes with more neighbors will be aggregated more often and thus play a more important role. The drawback of fix-radius is that the value of the hyper-parameter 'r' is hard to define. Different r values could result in a highly different graph and thus affect our model's result. In general, a higher value of r could result in a more dense graph while a smaller number could result in a more sparse one. In our model's comparison process, we will compare our models' performance on datasets generated by different r values. Below are images of graphs that are constructed based on the same data points but with different r values.



Graph 4.1: Graph constructed with r = 0.1



Graph 4.2: Graph constructed with r = 0.2



Graph 4.3: Graph constructed with r = 0.3

The advantage of using k-nearest neighbors is that we could have the same neighbors for all points thus we do need to worry about the hyper-parameter setting in-class influence.However, it is also the drawback since we could no longer apply the connectivity of one node to others in our models. We will use both methods to create datasets and train our models on each one. The comparison results will be discussed in the Result section.

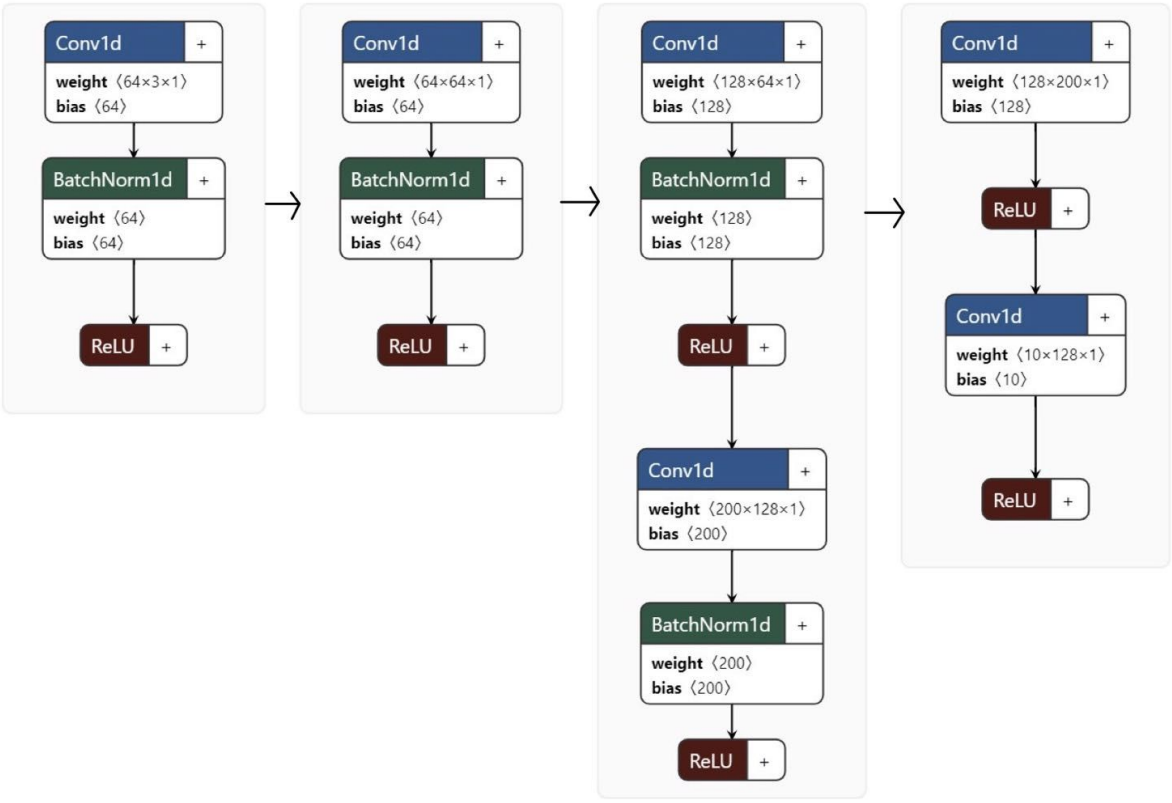
5. Models

In this section we will discuss models' implementation details. Graph Convolution Network (GCN) is our main focus. PointNet is used as a baseline model. Since it is only used for comparison, we designed it simply and similar with our GCN's architecture. In the last part, we will discuss the effects of combining graph pooling layers with PointNet. All three models are predicting on a points set. Thus, all models should be invariant to permutation. In other words, if we change the order in which we pass in our data points, the predicting results should always be

the same. Thus, maintaining this speciality is an important key in implementation of models. We will discuss the permutation invariance of the three models in the corresponding section.

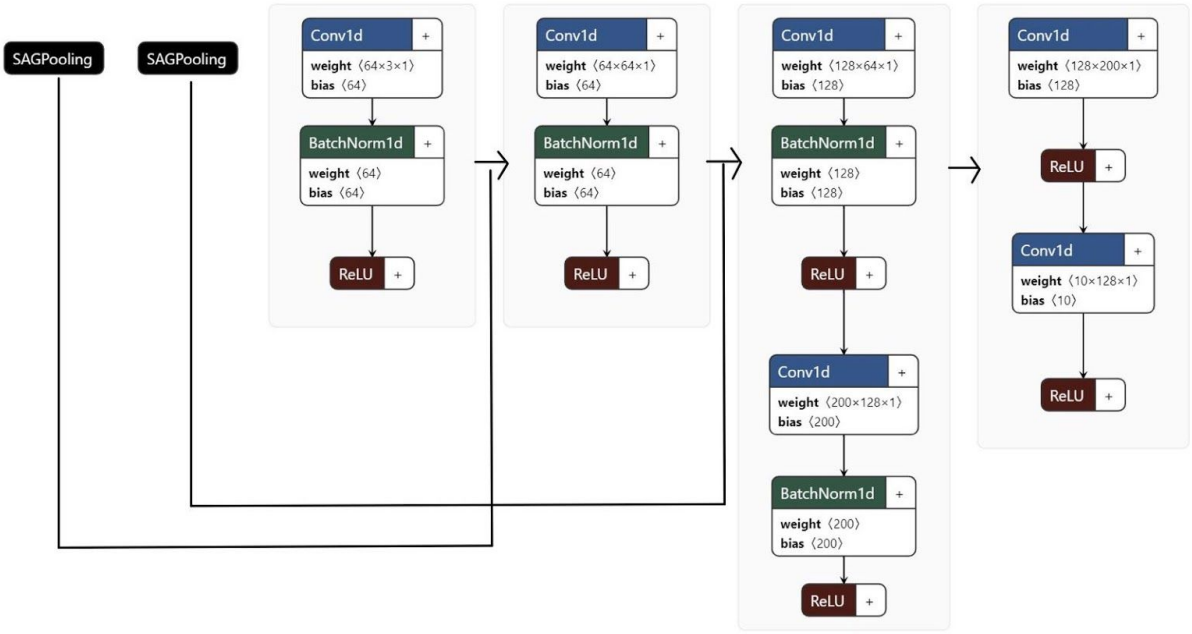
5.1 PointNet

PointNet is a well known model's architecture and widely-used in 3d points data machine learning tasks. It fully used the Multi-Layer-Perceptron (MLP) and symmetric functions to maintain the model's permutation invariance. More specifically, MLP is a layer that is trained to learn a function for all the points. Letting δ be the function MLP learned. For all points, the parameters α^* of δ are all the same. Thus, set $\{ \delta(\alpha_1), \delta(\alpha_2), ..., \delta(\alpha_n) \}$ should always be the same for all permutation of set of points $\{ \alpha_1, \alpha_2, ..., \alpha_n \}$. Connected MLP with symmetric functions such as max-pooling or mean-pooling will help us to reduce the data dimension while at the same time still keep the permutation invariance. After multiple blocks of MLP and pooling layer, we transform our data points to a feature vector and use them to predict the class labels. When implementing the code, we choose to use the convolution 1d layer to represent the MLP layer. The full architecture is shown below:



Graph 5.1: PointNet Architecture

PointNet is only used as a baseline model in our project. Thus we only keep its core layers and design it simply. The architecture is similar to the architecture of our Graph Convolutional Network. We also implemented a Graph-Pooling-PointNet model in which we added two graph pooling layers between the MLPs to reduce the dimension. The full architecture is shown below:



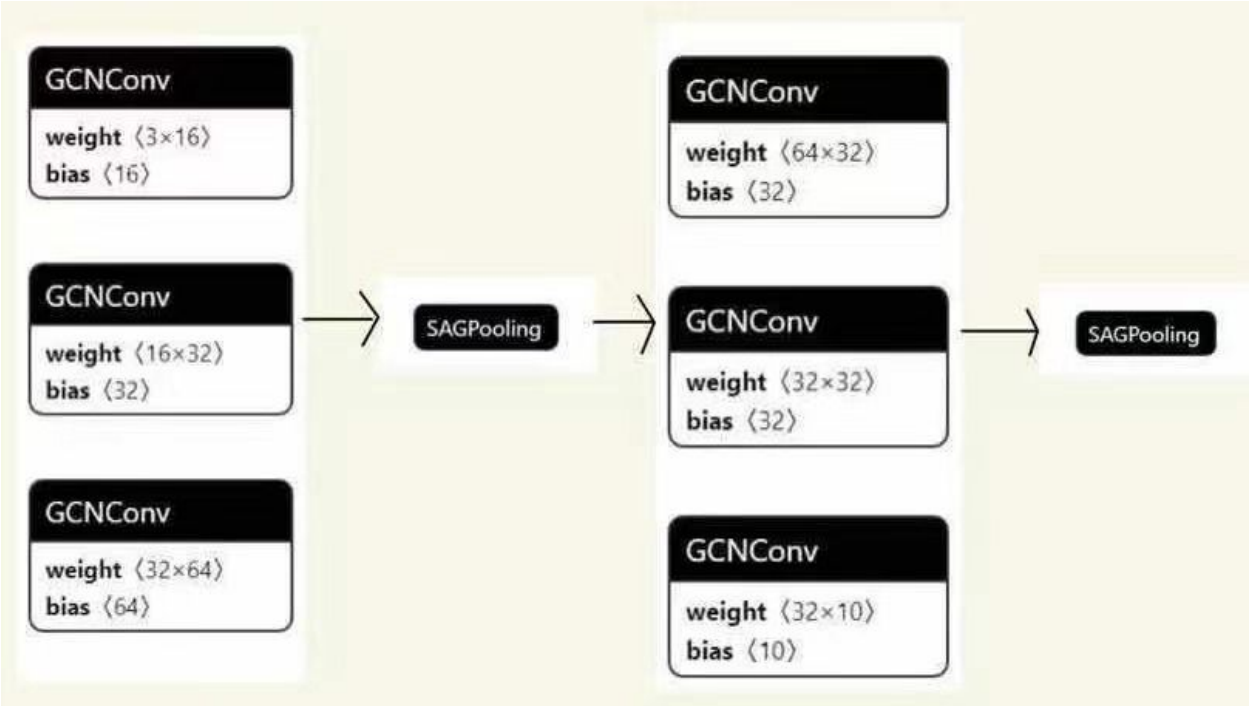
Graph 5.2: Graph-Pool Pointnet

5.2 Graph Convolutional Network

The core component of Graph Convolutional Network (GCN) is the graph convolutional layer. Traditional convolutional computation is based on matrix-shape data. Compared with traditional convolution, graph convolution computation is based on graph architecture, to be more specific, the edges. Either through multiply with adjacency matrix or one-by-one aggregating through edge list, graph convolution extracting features for a node from its neighbors. Similar to what we did in PointNet, we first transform our features to higher dimensions (3 to 32 and to 64) using GCN layer. Then by down-sampling through the graph pooling layer, we reduce the sample size. After pooling, we again use GCN layer to transform our features space back to the class-size dimension for predicting. The model is ended with a

softmax function to calculate the probability for each category. The full architecture is shown below.

Graph is indeed a permutation-invariant structure. No matter what order we passed in the points, we will always construct the same graph. What's more, GCN layer and graph pooling layer are all based on graph architecture. Through connecte them with non-linear layers, our GCN model should be permutation invariant as a whole.



Graph 5.3: GCN

5.3 Future Direction: GCN-PointNet

One further potential model architecture that may boost the accuracy is combining MLP layer with graph aggregation. To be more precise, after MLP learns a function that transforms all points through a same function with the same weights and bias, multiply the outcome with an adjacency matrix and distance matrix that was created from the graph construction method. This

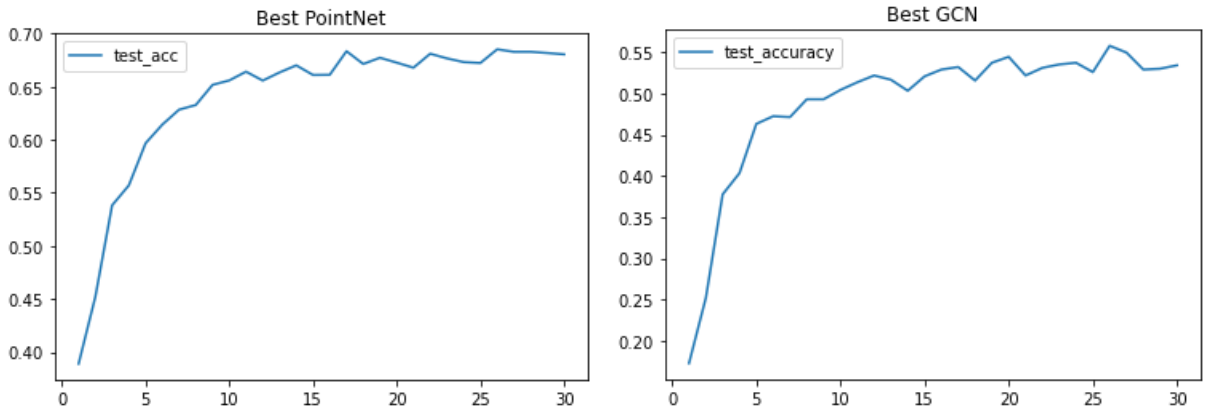
model's architecture combines the advantages of GCN layer and PointNet layer and we expect it will have better performance. This model is still in the WIP state and we will update to our code repository.

6. Results

In this section, we are going to: 1. Compare the accuracy and confusion matrix of GCN and PointNet; 2. Discuss hyper-parameters' effects. 3. Discuss the use of pooling layers. 4. Discuss GCN and PointNet's resilience on data augmentation.

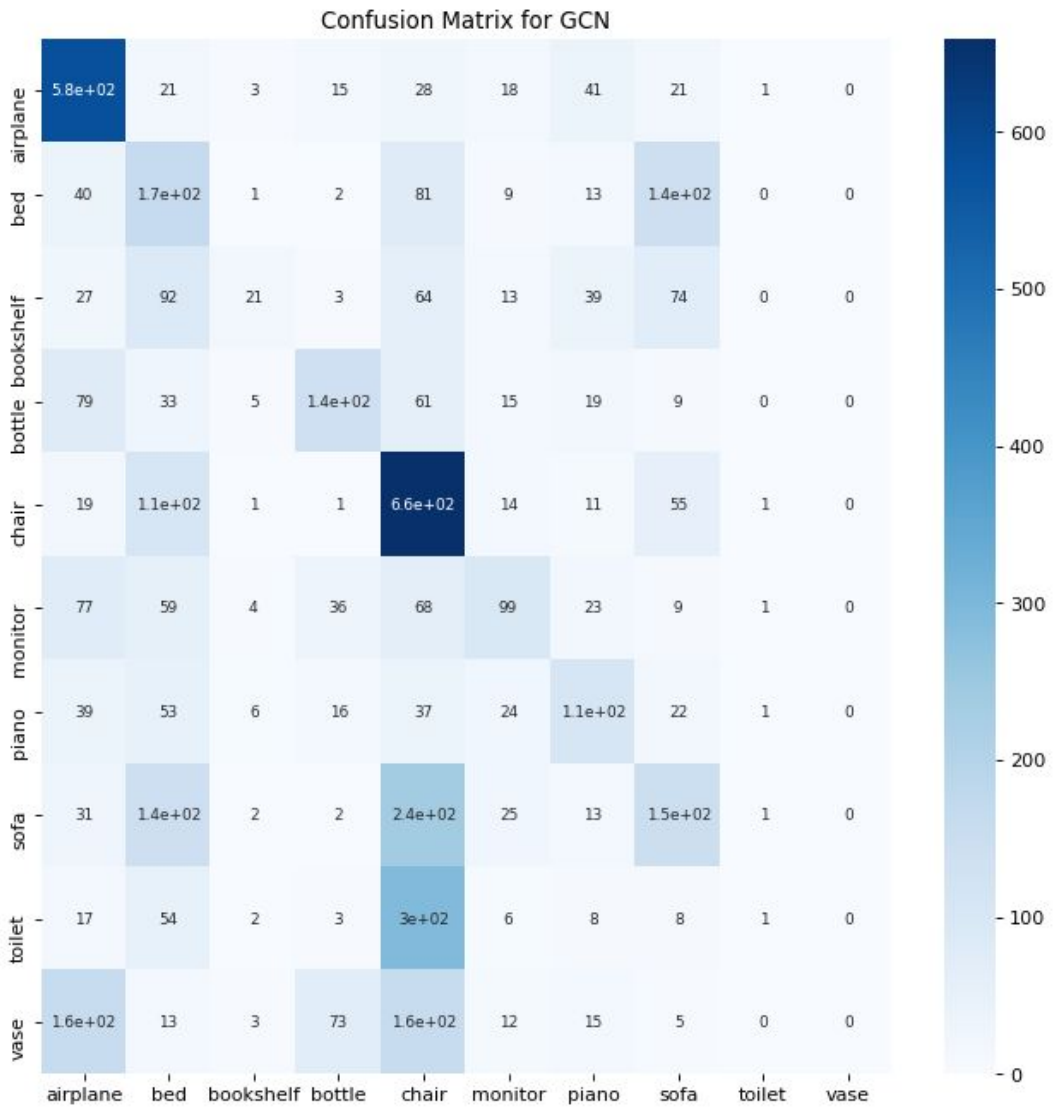
6.1 GCN vs PointNet:

In our experiment, the best accuracy we found using GCN on 10 categories classification is 57%±2.7%. The deviation is caused by the sampling difference happening in the graph construction process and as well as the model's initiation values. The parameters used for this result is {'pool': SAG, 'ratio': 0.4, 'val_size': 0.2, 'lr': 5e-4, 'epoch':30, 'batch_size': 32}. PointNet, on the other hand, got a best accuracy 68% on the same dataset. Training both two models on transformed 40-categories data will both get a much worse result - GCN will have a best accuracy of 27% and PointNet will have 45%. The reason is that we used 1000-points sampling to build our training dataset while in some categories, most of the samples could not meet this threshold and thus create a highly imbalanced dataset. In general, PointNet seems to have a better accuracy overall. Looking at the accuracy curve given below, we could find that both two models converge really fast. The accuracies will not change much after 10 epochs.

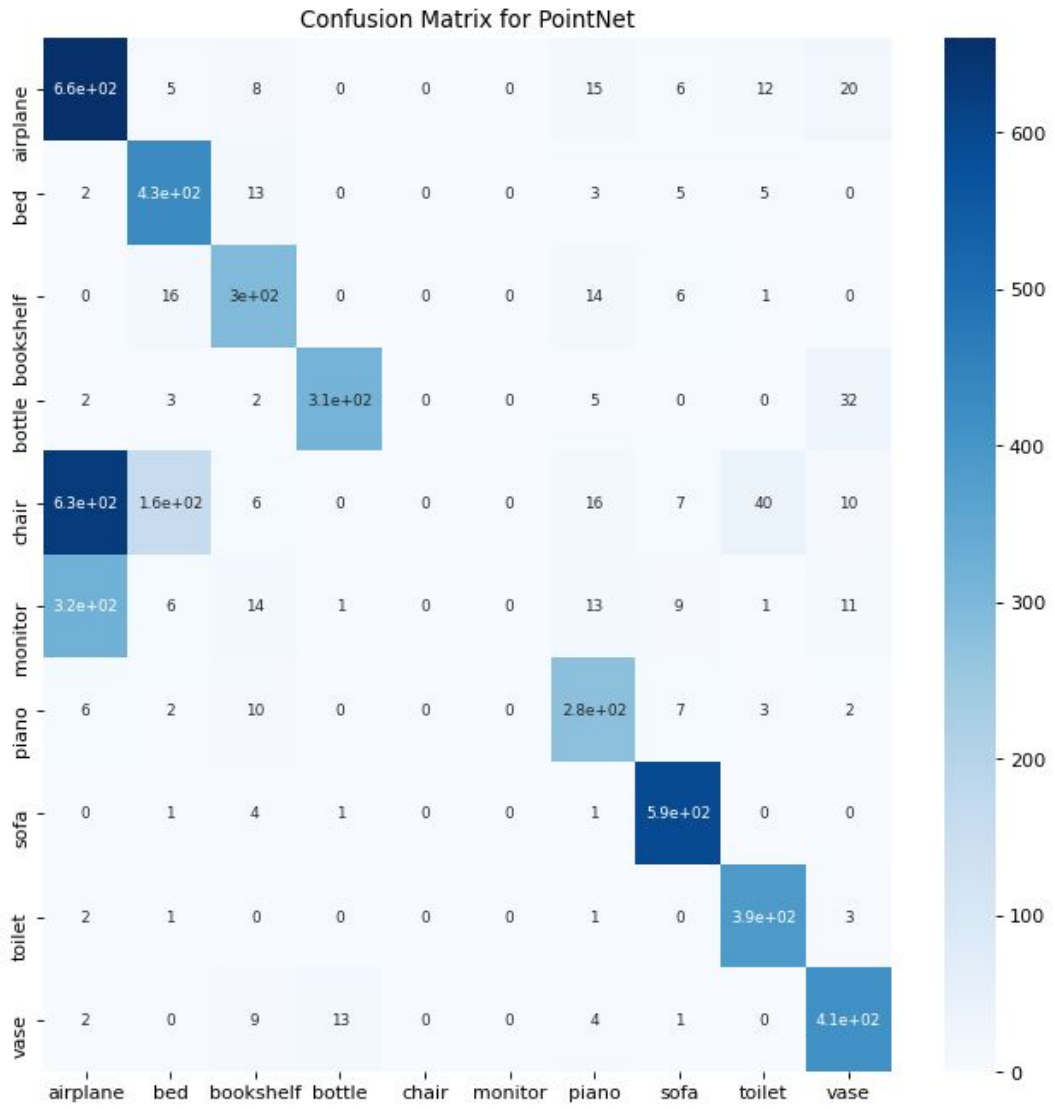


Graph 6.1: Accuracy

Accuracy is a major metric used in classification tasks but it could not show everything. We also plot two confusion matrices for each model. Below we provide the plots of each confusion matrix.



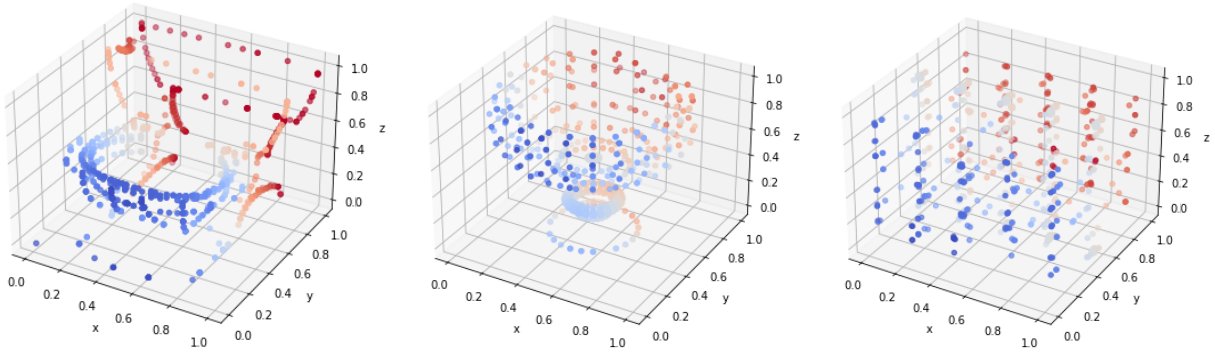
Graph 6.2: CF-GCN



Graph 6.3: CF-PointNet

Through the confusion matrix for GCN (Graph 6.2), we could see that: 1. GCN has a best performance on ‘airplane’ and ‘chair’ while has a bad prediction on ‘bookshelf’, ‘toilet’, and ‘vase’; 2. GCN tends to misclassify ‘toilet’ and ‘vase’ as ‘chair’, and misclassify ‘bookshelf’ as ‘bed’; 3. The misclassification is not caused by the imbalance numbers of training samples. This is because GCN predicts the ‘monitor’ which contains less samples pretty well while doing a bad job on ‘toilet’ and ‘vase’. Both of them contain a farewell number of samples. 4. Taking a look at

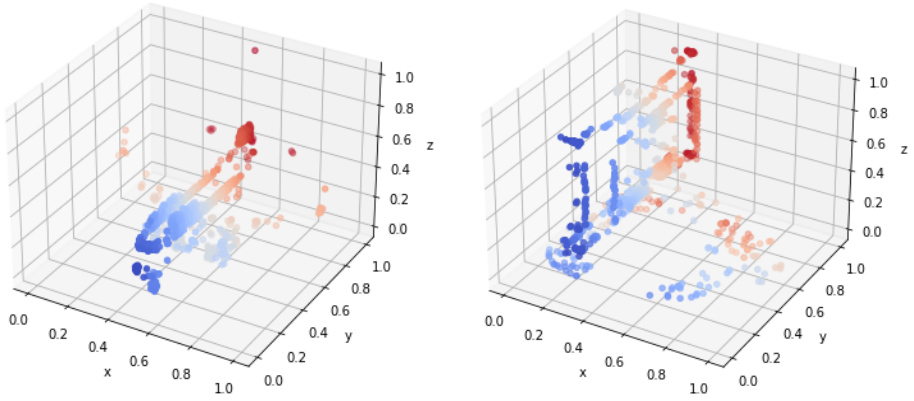
the misclassified samples of GCN, we can see that the misclassification is reasonable and related to the shapes of the data.



Original: Toilet; Predict: Chair | Original: Vase; Predict: Chair | Original: Bookshelf; Predict: Bed

Graph 6.4: Shapes

On the other hand, the confusion matrix for PointNet (Graph 6.3) shows that: 1.PointNet predicts all 8 classes very well except the ‘chair’ and ‘monitor’. 2. PointNet tends to misclassify ‘monitor’ as ‘airplane’. In the training dataset, ‘airplane’ has the second most training samples and ‘monitor’ has the third least training samples. This shows that PointNet’s misclassification is more related to the number of samples instead of the shape itself. We could also see from Graph 6.5 that it is hard to tell that ‘airplane’ is similar with ‘monitor’.



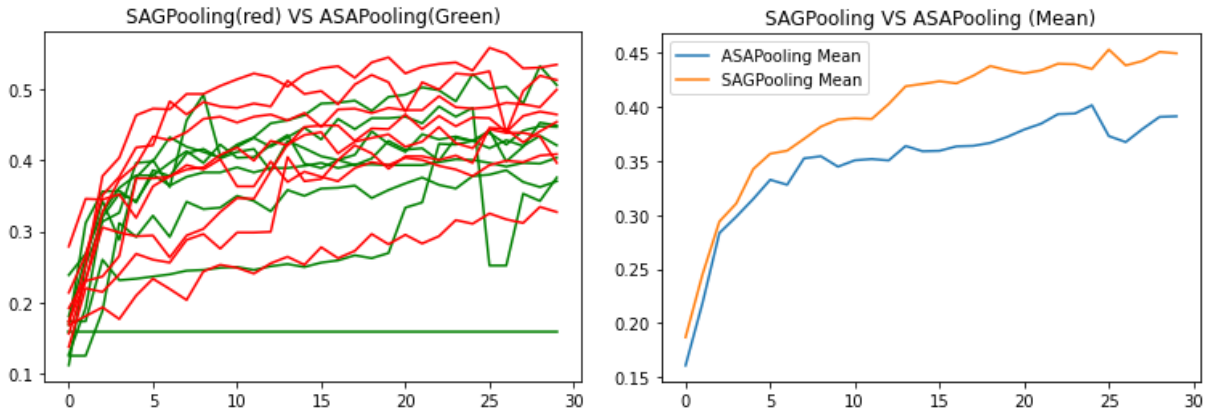
Graph 6.5: Airplane and Monitor

Putting it all together, we found that in general PointNet has a better accuracy than GCN while GCN's misclassification is more reasonable and more fit with the shape's similarity. We expect that if training on a more decent and large dataset, GCN could perform better than the PointNet.

6.2 Hyper-parameters:

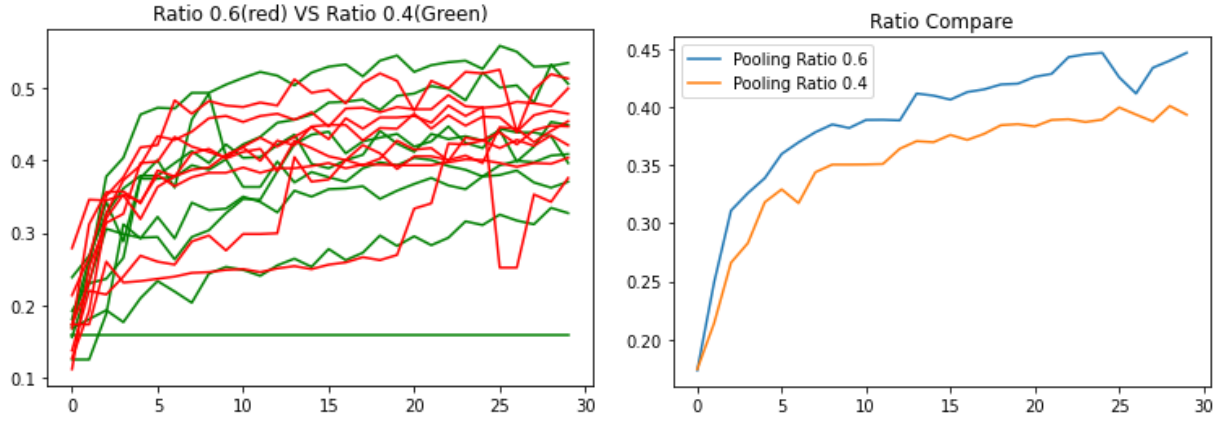
In this project, we used brute force to search the best parameters' combination. Due to the hardware and time limitations, we searched in a small range. However, we could still see some general trends from the result.

The most significant parameter which affects the results a lot is which pooling layer to use and the ratio used in the pooling layer. In this project, we tried SAG (Self-Attention Graph) pooling layer and ASA (Adaptive Structure Aware) pooling layer. Both pooling layers are based on graph architecture and they both use a hidden Graph Convolutional Layer (GCL) for scoring. The difference is that SAG used GCL to score the importance of nodes within a cluster while ASA used GCL to score the clusters. In other words, SAG is pooling nodes through all clusters while ASA is pooling clusters. You can find two corresponding papers in the appendix section. Our experiment shows that in our task, SAG pooling has a much better performance than ASA pooling. From the graph 6.6, we could see that almost every combination using SAGPooling performed better than ASAPooling. Also, the right graph shows that SAGPooling in general has a 10% higher max-accuracy.



Graph 6.6: Pooling Layer

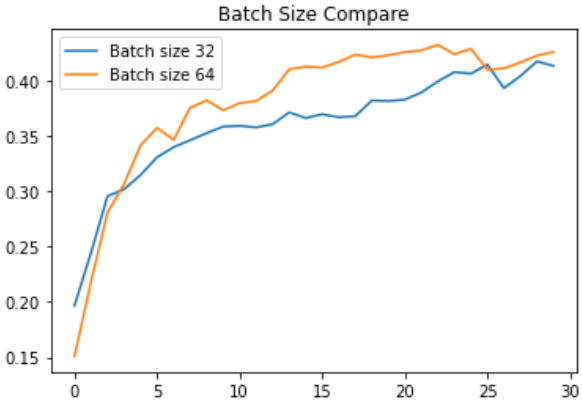
The parameter ‘ratio’ used in the pooling layer also plays an important role. In our experiments, the result shows that compared with a slight larger pooling ratio (0.6 means after pooling, the data size will be 0.6 of the original), a smaller ratio (0.4) could have a better accuracy. This is shown through Graph 6.7.



Graph 6.7: Ratio

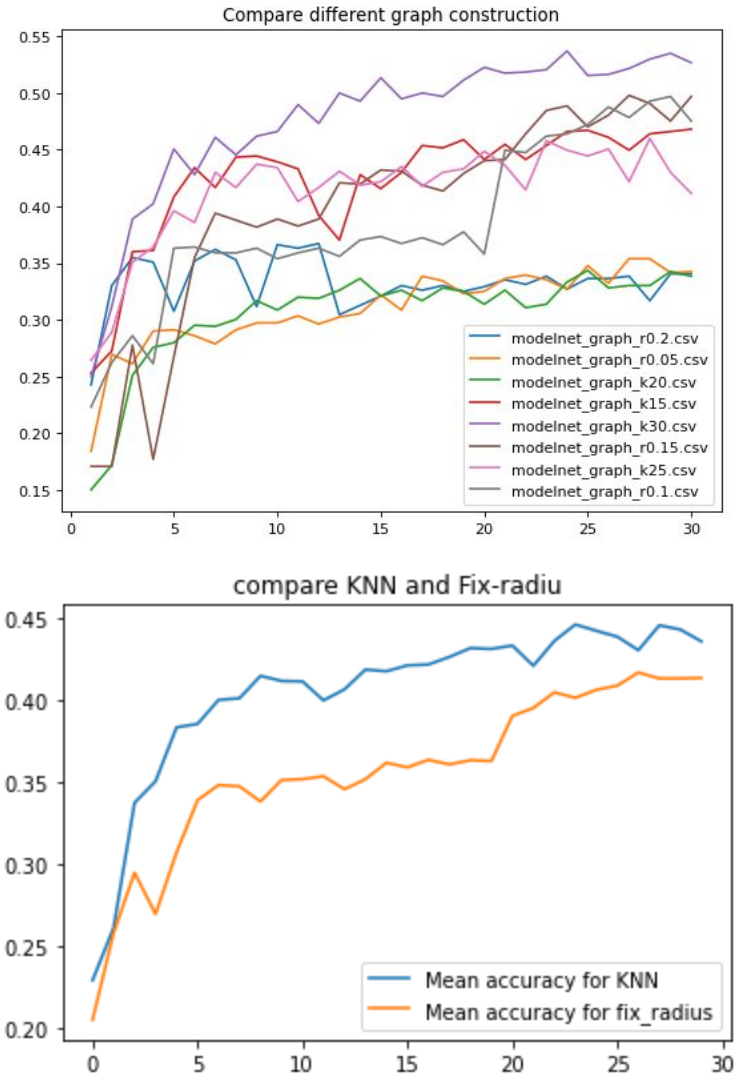
We also tried different settings of traditional hyper-parameters in deep learning such as learning rate and batch size. It turns out that learning rate does not have much effect on final results as long as it is smaller than 1e-3. Batch size, on the other hand, though did not have much influence on the best accuracy, it has effects on the efficiency. In Graph 6.8, we could see that with a larger batch size, the model converges much faster and reaches its best accuracy a few

epochs before small batch size. This is reasonable since with a larger batch size, we updated our weights with a larger batch data and thus have a more efficient training procedure.



Graph 6.8: Batch size

The last thing we want to talk about is the graph construct methods. We talked about two graph construct methods fix_radius and KNN in the graph construction part. We tried four different hyper-parameter settings (k and r) for each method. Through Graph 6.9, we could see that KNN is generally better than the fix_raius. The reason, from our perspective, is that KNN will construct a more balanced graph which basically will have zero disconnected nodes and every node will have an equal number of edges.



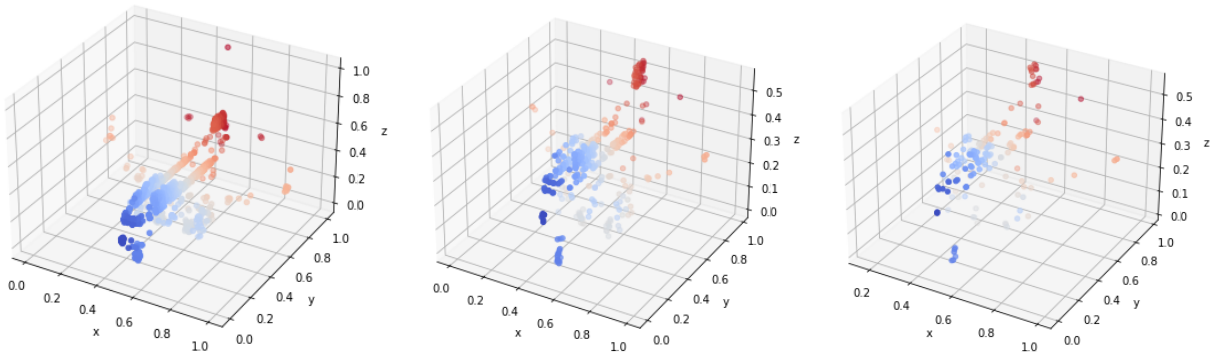
Graph 6.9: Graph Construct

6.3 Pooling layers

In the model section, we mentioned that we will also train a Graph-Pooling-PointNet model. However, this model has the worst result. The best accuracy is about 24% on 10-categories dataset. The reason why graph pooling works on GCN but not on pointnet is that GCN model used graph convolutional layers ahead of graph pooling layers. The graph convolutional layer will aggregate the information within a neighborhood/cluster. Thus when the SAG pooling layer pools out nodes that are considered as less significant nodes in the cluster,

their information will maintain in their neighbors. In other words, connecting graph convolutional layers with graph pooling layers will not lose much information while at the same time, reduce the sample size. However, MLPs in pointnet do not aggregate information within a cluster. Thus, if we connect MLPs with graph pooling layers, we actually lose every nodes' information we pool out. This also explains why the SAGpooling layer has a better result than the ASApooling layer in our project. The reason is that the graph convolutional layer and the SAGpooling layer make up for each other and have really good results.

The following graph shows the pooling results after the pooling layer.



Graph 6.10: Original-First Pooing-Second Pooling; Points

Graph 6.11: Original-First Pooing-Second Pooling; Graphs

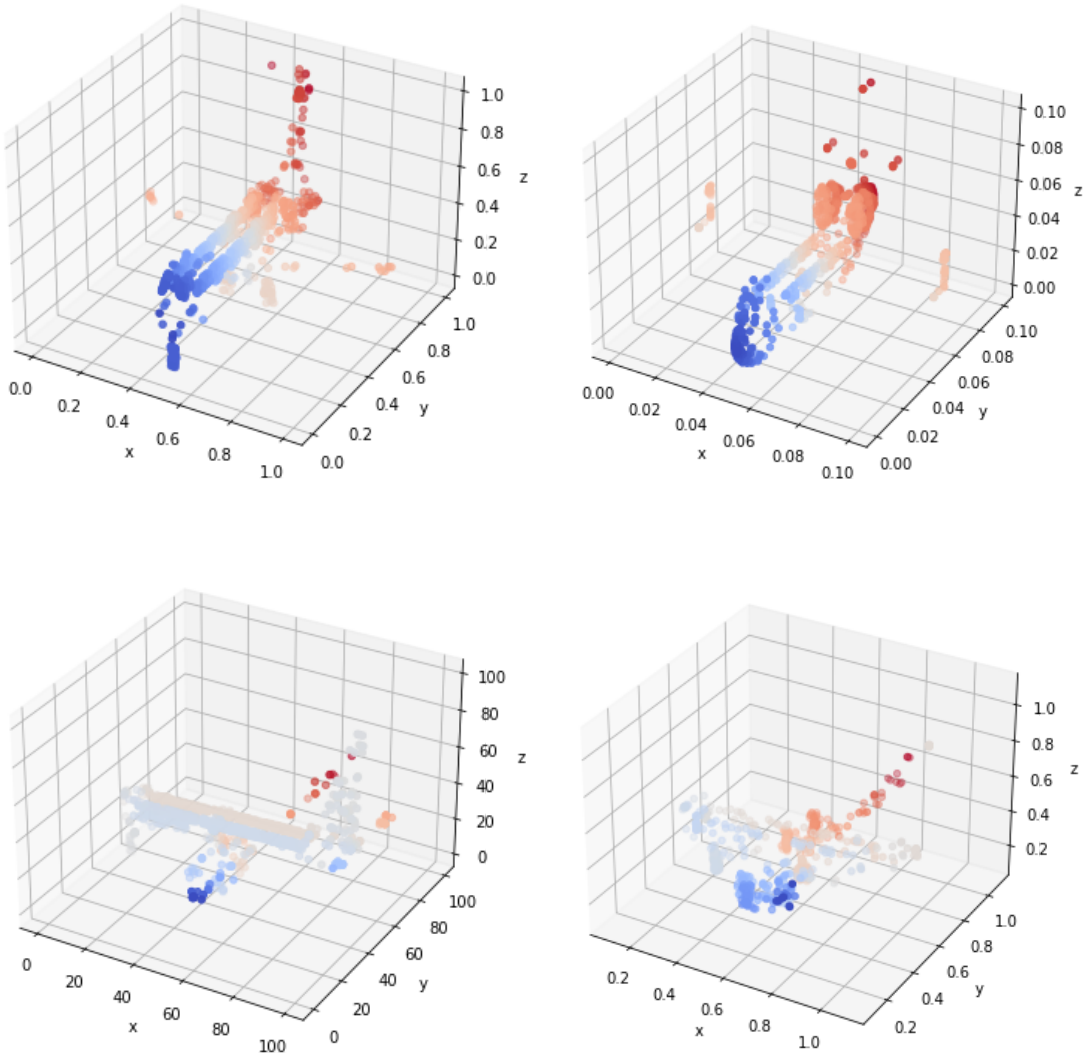
6.4 Resistance on data augmentation

Our models are trained to predict the shapes. Despite the permutation-invariance we talked before, a good model should also be invariant to different data transformations such as enlarge, shrink, and translation. In our project, we tried three different data augmentations and used them to test GCN and PointNet's resistance. The results is shown in the below table:

	name	GCN	PointNet
0	Strech	0.307163	0.000000
1	Shrink	0.000000	0.000000
2	Translate	0.811295	0.320937
3	Not Change	0.796143	0.909091

Table 6.1: Augmentation

We could see that PointNet has a much worse resistance than GCN. If the data is stretched or shrunk, the pointnet will no longer recognize it anymore. On the other hand, GCN model even did really badly on shrunk data, it did really well on translated data. The stretch procedure is done by multiplying the original data with 100, the shrink procedure is done by multiplying 1e-2, and the translate procedure is done though adding 0.1 to all points within a sample. Three transformations could be seen through the following graph.



Top Left: Original; Top Right: Shrink;

Bottom Left: Enlarge; Bottom Right: Translate

7. Conclusion

All in all, our project shows that: 1. In general, PointNet has 11% higher accuracy than GCN on ModelNet data. 2. GCN has more reasonable misclassification. 3. Pooling layers work well in GCN while not in pointnet. The reason is graph convolutional layers make up the loss information. 4. GCN has a much more robust resistance on data augmentation.

8. Appendix

“Self-Attention Graph Pooling”, Junhyun Lee, Inyeop Lee, Jaewoo Kang,

https://arxiv.org/pdf/1904.08082.pdf

“ASAP: Adaptive Structure Aware Pooling for Learning Hierarchical Graph Representations”,

Ekagra Ranjan , Soumya Sanyal , Partha Talukdar, https://arxiv.org/pdf/1911.07979.pdf

Our project code: https://github.com/ctwayen/GNN-Points-Cloud

Our project website: https://ctwayen.github.io/Graph-Neural-Network-on-3D-Points/