Week 5: 3D Feature Matching

December 4, 2021

1 Introduction

In this task, I tried to implement 3d feature matching by training a Neural Network. Unlike in the past few tasks, which used mostly CNN for training, GNN is used in this task. The input of the GNN are two 3d pointclouds, the GNN encode these points into embedding space. The goal of training is that corresponding points have very similar embeddings while non-corresponding points have very different ones. After getting the embeddings for the both pointclouds from the model, the matching points can be computed based on the similarity of embeddings.

There are 4 scripts in the folder. 'task5.ipynb' is the training process with GraphUNET. 'task5_MyNet .ipynb' are the training processes with all other 4 models. 'inference.py' is used to test the output of the trained models. And 'task5_inference_example.ipynb' shows an example for 'inference.py'. The test results are stored in 'test_outputs.pdf'. Trained models are stored under the folder 'models'.

2 Prepare the dataset

The dataset used in this task is from ShapeNet, specifically I picked the 'Knife' category, since most of the objects in this category is non-symmetric, which reduce the probability of mismatching.

To load the dataset, first I tried to read the 3d points directly from the obj files stored locally. Then I found that it is also possible to load the dataset directly from torch_geometric.datasets. The samples in the dataset has a specific datatype called torch_geometric.data and the information of the points in each sample are stored in the data, including the features, the positions and category of this sample. In this task, only the position information is used.

Since the original 3d pointcloud is quite dense, which makes the matching difficult since there are many points with very position, the pointclouds are downsampled with the built-in function torch_cluster.fps. Since each sample has different number of points, the downsampling is applied to each sample with different ratio. After this step, each sample has on average 50-150 points. The edges are created for the downsampled points with torch_cluster.knn_graph and the number of knn is set to be 6.

The pointcloud should then be augmented arbitrarily with both random rotation and transformation. To test the if the model is capable of the random transformation, first I augment the pointcloud with only rotation on x-axis and add a very small rigid translation between 0 and 0.001 to the points to test which model fit this easy problem on the best. After this pre-test the points are augmented with random rotation on all 3 axes and a larger rigid translation between 0 and 1.

3 Design the loss function

The loss function used in this task is the contrastive loss, which aims at push the embeddings of corresponding points closer (1) and embeddings of non-corresponding points further (2).

$$\mathcal{L}_{\text{pos}}\left(\mathbf{F}_a, \mathbf{F}_b, l\right) = \frac{1}{N_{\text{pos}}} \sum_{N_{\text{pos}}} D_{\text{feat}}^2 \tag{1}$$

$$\mathcal{L}_{\text{neg}}\left(\mathbf{F}_a, \mathbf{F}_b, l\right) = \frac{1}{N_{\text{neg}}} \sum_{N_{\text{neg}}} \max\left(0, M - D_{\text{feat}}\right)^2 \tag{2}$$

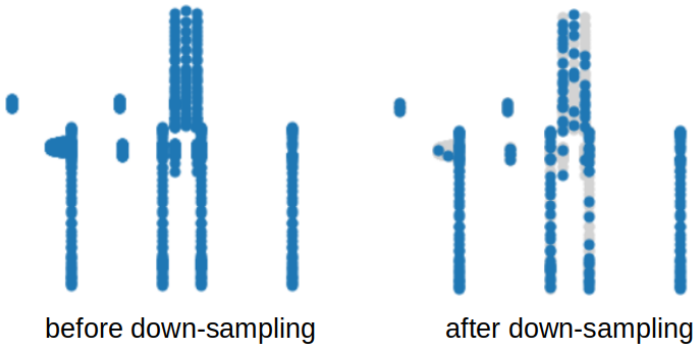

Figure 1: Downsample the pointclouds.

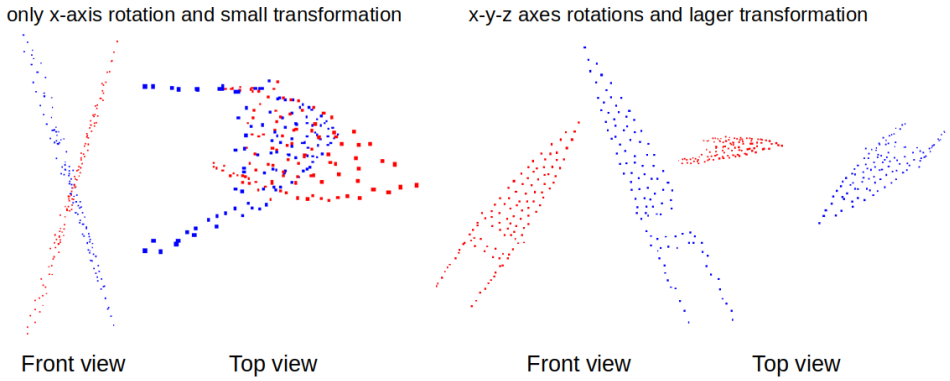

Figure 2: Transformations on pointclouds.

The cosine distance is used to compute the distances between the embeddings. The M in the equation (2) represents the margin, which means if the embeddings of non-corresponding points are as far as the size of the margin, this pair is considered to be far enough and make no contribution to the loss. After several trials, I found a margin of 1 is reasonable.

4 Design the matching function

With the cosine distance function, the distances between each two points are computed and stored in a distance matrix. Under ideal conditions, the diagonal distance should be very close to 0, while all other distances much larger.

First I tried to find the corresponding points simply by searching the smallest value in each row of the cosine distance matrix with torch.argmin. However, in this way, more than one point may be assigned to the same corresponding point. What's more, for some points, the distance of the ground truth corresponding point is very close or a little bit larger than the distance of an other point, like 0.025 and 0.021, which means the ground truth point maybe not the closest point but the second or the third.

To solve this problem, I perform the Hungarian matching method on the cosine distance matrix. With this method, each point is assigned to different point, the repetition is avoided. The result shows that the Hungarian matching has a large contribution for assigning corresponding points. Even if only half of the points are assigned correctly with torch.argmin, the Hungarian matching can find the true corresponding points for all points.

5 Design different models

There are many different convolutional layers in GNN and I designed several models with these layers

First I tried the GraphUNet, which can be directly imported from torch_geometric.nn. This model implements a U-Net like architecture with graph pooling and unpooling operations. The depth is set to 4.

Then I designed other four models, each with one of the following convolutional layers: GATConv, EdgeConv, GraphConv and PPFConv. It should be noticed that the GATConv, GraphConv are very different from EdgeConv and PPFConv. In both GATConv and GraphConv, the embeddings are computed first and then propagated to the nodes as messages. While in EdgeConv and PPFConv, messages are first propagated to the nodes then the embeddings are computed.

```
GATGCN(
  (conv1): GATConv(3, 128, heads=1)
  (conv2): GATConv(128, 128, heads=1)
  (conv3): GATConv(128, 512, heads=1)
)
```

```
GraphGCN(
  (conv1): GraphConv(3, 128)
  (conv2): GraphConv(128, 128)
  (conv3): GraphConv(128, 512)
)
```

```
(conv1): PPFConv(local_nn=Sequential(
  (0): Linear(in_features=7, out_features=512, bias=True)
  (1): ReLU()
  (2): Linear(in_features=512, out_features=512, bias=True)
  (3): ReLU()
  (4): Linear(in_features=512, out_features=512, bias=True)
), global_nn=Sequential(
  (0): Linear(in_features=512, out_features=512, bias=True)
  (1): ReLU()
  (2): Linear(in_features=512, out_features=512, bias=True)
  (3): ReLU()
  (4): Linear(in_features=512, out_features=512, bias=True)
))
```

```
(conv1): EdgeConv(nn=Sequential(
  (0): Linear(in_features=6, out_features=512, bias=True)
  (1): ReLU()
  (2): Linear(in_features=512, out_features=512, bias=True)
  (3): ReLU()
  (4): Linear(in_features=512, out_features=512, bias=True)
))
```

Figure 3: Four different models.

6 The training procedure

There are 312 samples in all, 265 are used for training and 47 for validation/test. The learning rate is set to be 5e-6 and the batch size is 1. For each model, the dimension of output embeddings are all set to be 512.

For training, the contrastive loss is computed for each training batch. In the validation step, the cosine distance is computed based on the 2 output embeddings and the Hungarian matching is applied on the computed distance to find the corresponding points. The accuracy of matching is computed for each epoch.

First I train the models with the small-transformation problem where only x-axis rotation and small translation are applied to the point clouds, then I selected 3 models to train with the large-transformation problem with x-y-z-axes rotations and larger translations.

6.1 Output with rotation only on x-axis and small translation dataset

Table 1: Loss and Accuracy for the small-transformation problem

GraphUNET	GATConv	EdgeConv	GraphConv	PPFConv
0.130	0.170	0.046	0.202	0.035
79%	88%	99%	89%	99%

After training each model for 300 epochs, learning rate 5e-6, the loss and accuracy are shown in Table 1. All the 5 models seem to perform well on this simplified problem since all the accuracy are at least around 80%. The training of GraphConv took the least time, but the loss is also the largest. Both models with EdgeConv and PPFConv fit the problem very well with the accuracy almost 100% after training. The difference between these two models was that the model with PPFConv took less epoch to converge to a very small loss, while the model with EdgeConv took twice more epochs. Unlike in the task 4, the UNET-like architecture GraphUNET does not have an advantage either on the speed or the outcome.

It should be noticed that for GraphUNET, GATConv and GraphConv models, the accuracies with torch.argmin and Hungarian matching are very different with the accuracy after Hungarian matching much higher, but for EdgeConv and PPFConv, the accuracy with torch.argmin is already very high, so the Hungarian matching does not play an important role in these two models. This difference also means that the EdgeConv and PPFConv have a much better performance then the other three models.

6.2 Output with 3-axes-rotation and larger translation dataset

Table 2: Loss and Accuracy for the large-transformation problem

GraphUNET	EdgeConv	PPFConv
0.27 (300 epochs)	0.11 (600 epochs)	0.0583 (400 epochs)
5.44% (argmin matching)	62.16% (argmin matching)	67.46% (argmin matching)
14.73% (Hungarian matching)	79.24% (Hungarian matching)	84.33% (Hungarian matching)

After training with the learning rate 5e-6, the loss and accuracy are shown in Table 2. The models with EdgeConv and PPFConv layers are selected since they performed best in the small-transformation problem. The GraphUNET is also trained here as a reference.

As can be seen in the Table 2. Both models with Edgeconv and PPFConv layers have good performance with the accuracy around 80%. The model with PPFConv performs better since the accuracy reached 84.33% with only 400 epochs while the model with EdgeConv reached an accuracy of 79.24% with 600 epochs. And the GraphUNET seems to be incapable of fitting this problem with the accuracy stuck by 14%.

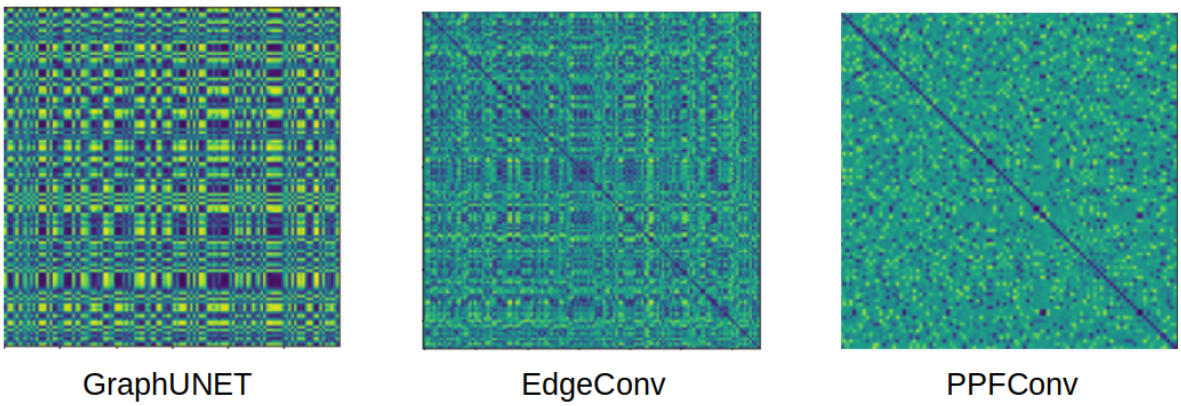

Figure 4: Cosine distance matrices after training.

7 Test with the trained models

To test the models, both the trained EdgeConv and PPFConv models are put into the inference.py script. The script takes a data from the ShapeNet-Knife as input. The 3d pointcloud is downsampled and augmented randomly in the script. The original pointcloud and the augmented pointcloud are then put into the pretrained EdgeConv/ PPFConv models to compute the embeddings. Based on the output embeddings, the cosine distance matrix is computed and then both the torch.argmin matching and Hungarian matching methods are performed on the distance matrix. The script print the both accuracies. 6 random points are chosen and their matching points are searched with Hungarian matching. The accuracy of these 6 points are also printed. Each random point and its matching point is colored differently from other points.

I made three tests with both the trained EdgeConv model and PPFConv model and the results are stored in test_output.pdf.

8 Conclusion and future work

In this task, I tried to train the GNN models for the first time. The GNN models propagate messages between neighboring nodes with edges. It seems that the GNN models are capable by compute embeddings of the nodes. I tried different architectures and the models with EdgeConv and PPFConv layers work much better than the others. And the Hungarian matching improves the matching accuracy dramatically.

As for future work, maybe different convolutional layers can be combined together to see if they can perform better.