24-04-2020

Dynamic Graph CNN for Learning on Point Clouds

Group 843

Moaaz Mohamed Jamal Allahham

Kasper Schøn Henriksen

Mathias Stouggard Lynge

Tabel of content

Introduction

Handling point clouds with deep learning can be a very heavy task due to their irregularity as they are not placed in grids like images. Therefore it can be difficult to use deep learning methods like the traditional CNN which requires grids.

A solution to this is the PointNet[2] that came out in 2017, which is a network to handle point clouds in order to classify them. Later work and improvement of the network has also been conducted, for example the Dynamic Graph Convolutional Neural Network (DGCNN) [3]. This network, compared to the basic PointNet network, can do neighbour processing.

Network

DGCNN proposes the method EdgeConv, which is presented in figure 2, that dynamically change its graphs during training. This method utilizes k-nearest-neighbours by computing Euclidean distances between points and uses a fully connected layer that contains weights to identify edge features (e_{ij}) from pair of points x_i and x_j . To achieve permutation invariance, the EdgeConv utilizes the operation max-pooling to symmetrically aggregate edge features.

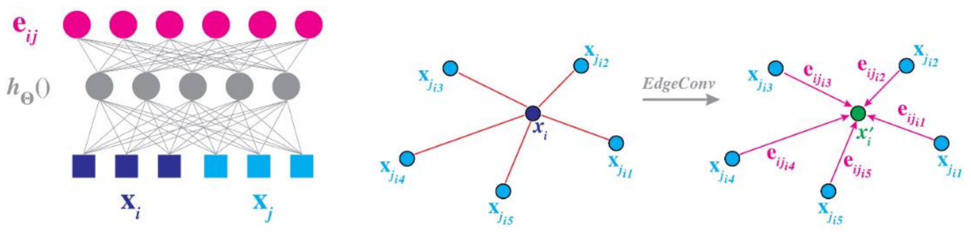

Figure 2: The proposed EdgeConv method which shows how graph features (e_{ij}) can be found using a pair of points x_i and x_j. [3]

The presented network below is split into two main branches. The top branch, which is responsible for the classification part, and the bottom branch, which is responsible for the segmentation part.

The classification part takes an input of the size n x 3 (x, y, z), where n is the number of points in the point cloud. The input is then passed further to an EdgeConv layer, which calculates the edge feature set of size k for each point, and it will be explained in more details in the next section. The output features of the first EdgeConv layer are aggregated by passing them into 3 more EdgeConv layers. Those 4 EdgeConv layers share three fully connected layers where the output features of each individual layer are concatenated to get a 512-dimensional point cloud. A max pooling operation is then applied at each point cloud to form a 1D global descriptor, after which two fully connected layers (512, 256) are used to transform the global feature.

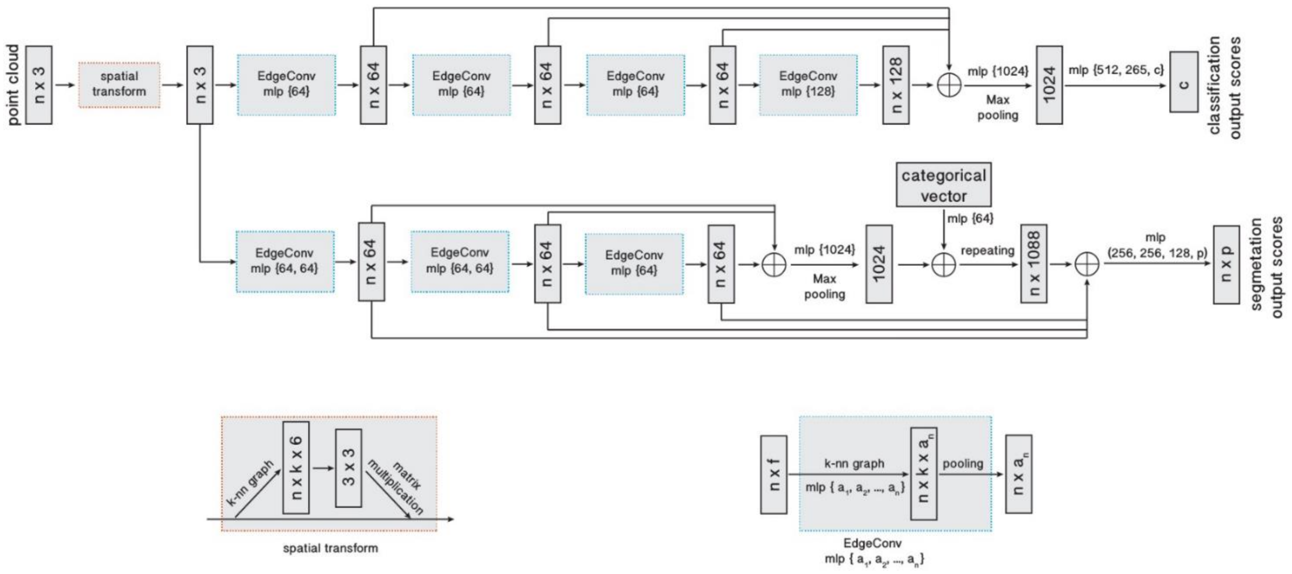

Figure 3: Model architectures for the DGCNN [3]. The top part is the classification part, which is implemented.

Implementation

The code can be found in:
https://github.com/Schonsteren/SemesterProject_843/tree/master/ComputerVision_miniproject

In this project, only the classification part of DGCNN was implemented in PyTorch. As presented in the architecture, the classification part includes spatial transformation. However, the authors of DGCNN have later commented that spatial transform only made a little difference, and it is therefore not implemented [1]. As previously mentioned, the model forward propagates through the four layers of EdgeConv. As presented in the code snippet below, the EdgeConv contains three parts; a get graph feature, a convolutional layer and a pooling at the end. The get graph feature extracts features from the data based upon k-nearest neighbours which is computed using Euclidean distances between points. The convolutional layer utilizes batch normalization which accelerates the training process and enhances regulation to prevent overfitting. Moreover, this layer uses the activation function LeakyReLU, which is a leaky version of the Rectified Linear Unit with a negative slope of 0.2 which prevents dead neurons. The pooling layer is a max pooling layer, which pools the maximum of the neighbouring edge features.

```
#EdgeConv1
        data = get_graph_feature(data, k=self.k)
        data = self.conv1(data)
        data1= data.max(dim=-1, keepdim=False)[0]


self.conv1 = nn.Sequential(nn.Conv2d(6,64,kernel_size=1,bias=False),
                                   nn.BatchNorm2d(64),
                                   nn.LeakyReLU(negative_slope=0.2))
```

As presented below, after each of fully connected layer in the classification part, dropout of 0.5 is applied, to further enhance regulation and prevent overfitting:

```
data = F.leaky_relu(self.bn1(self.fc1(data)), negative_slope=0.2) #Fully Connected
data = self.dp1(data) #Dropout of 0.5
```

For the training and testing we used batch sizes of 16 and 8 respectively. However, for the real implementation of DGCNN batch sizes of 32 and 16 is utilized. But due to only having one GPU available (GeForce GTX 1080) and not having their equipment (2x NVIDIA TITAN X) we had to decrease the number of batch sizes, else our GPU would run out of memory. Moreover, 50 epochs were utilized for training the model. For the hyperparameters, the optimizer SGD was utilized to optimize the parameters of the model to find the lowest possible minima and the learning rate of the optimizer is 0.1, a momentum of 0.9 and a weight decay of 1e-4. To calculate loss, the cross-entropy loss is computed by combining the log softmax and negative log likelihood of loss, which is used to update the parameters.

Evaluation

For the training we ran 50 epochs, due to computation time we were not able to do more. All the hyperparameters can be seen in table 1.

Hyper parameters	
Nearest Neighbors (K)	20
Initial learning rate	0.1
Dropout rate	0.5
Batch size	16/8
Momentum	0.9

Table 1: All the hyper parameters used for the training

Data

The data set used is the modelnet40 [4], which consists of 40 different classes of point clouds. Each point cloud consists of 1024 points, in total we used 12308 point clouds, 80% (9840) for training and 20% (2468) for testing.

Results

Results for our model can be seen on the graphs underneath, figure 1 and 2. Looking at the accuracy slopes for both the training and the testing set, the slopes are following each other which can be interpreted as not having overfitting or underfitting.

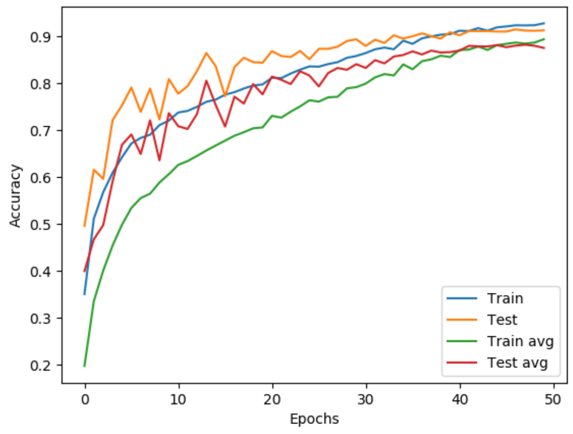

Figure 4: Graph showing the accuracy for our training and testing

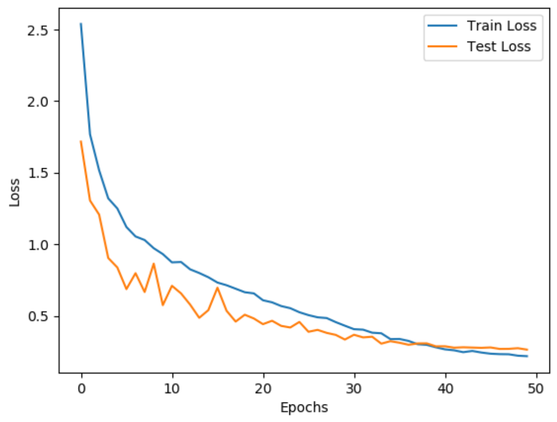

Figure 5: Graph showing the loss for our training and testing

The model with the best accuracy was from iteration 46 which can be seen in table 2 underneath.

Best model (Epoch number 46)	
Test accuracy	0.9145
Train accuracy	0.9233
Train loss	0.23
Test loss	0.27
Test mean class accuracy	0.88
Train mean class accuracy	0.89

Table 2: Showing the results of the best model from our training

Discussion

The original paper gets an accuracy of 92.9% from 250 epochs, which is close to what we achieved which is 91.5% from 50 epochs. Also, they achieve a mean class accuracy of 90.2% where we achieved 88.0%. Considering the lower amount of training we find these results acceptable, and our implementation to be working. To get more accuracy, we would need to train more epochs, and maybe use the point clouds with more points, which worked for the authors of the paper. However, it is not for sure that we will be able to reproduce the results. Next step to the implementation would also be to do the segmentation part, which we did not implement.

References

[1] Dynamic Graph CNN for Learning on Point Clouds: *https://github.com/WangYueFt/dgcnn/issues/32*. Accessed: 2020-04-24.

[2] Qi, C.R. 2017. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. (2017).

[3] Wang, Y.U.E. et al. 2019. Dynamic Graph CNN for Learning on Point Clouds. 1, 1 (2019).

[4] Wu, Z. and Song, S. 2015. 3D ShapeNets : A Deep Representation for Volumetric Shapes. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (2015). DOI:https://doi.org/10.1109/CVPR.2015.7298801.