Amazon Internship Report

Yan Hao

September 2020

1 Introduction

During my internship, I mainly work on implementing the algorithm of "Grid-GCN for Fast and Scalable Point Cloud Learning, CVPR2020" to our codebase Deep Graph Library (DGL).

Our implementation should be localized under folder examples → pytorch → pointcloud → pointnet, which is equivalent to the implementation of pointnet, pointnet2.

Code is available at https://github.com/honeyhaoyan/python_version_Grid_GCN.

2 Code

2.1 DataLoader

I add a normalization function to ModelNetDataLoader.py, which could limit the coordinates of all points in the point cloud in a range from 0 to 1.

```
# This function is in ModelnNetDataLoader.py
def normalization(points):
    size = points.size()
    xyz = torch.zeros_like(points)
    for i in range(size[0]): # per batch
        xyz_max = torch.max(points[i,:,:], 0)
        xyz_min = torch.min(points[i,:,:], 0)
        xyz[i] = (points[i,:,:]-xyz_min.values)/(xyz_max.values-xyz_min.values)
    return xyz
```

2.2 Grid_GCN

2.2.1 Voxel Module

This module defines the class of voxel struction for per batch of point cloud.

In the initialization function, we created neighbour voxel list using defined voxel size. After that, we find all neighbour voxels for every voxel by a three-layer for loop. Finally, we assign the caculated neighbour voxel list to every point cloud in a batch. By caculating the neighbour list in advance in the initalization function, we could save the time of caculating neighbour voxel list repeatedly for every point cloud in a batch.

```
# This function is in grid_gcn_final.py
def __init__(self, voxel_size, batch_size):
        super(VoxelModule, self).__init__()
        self.voxel_size = voxel_size
```

```
        self.voxels = None
        self.neighbour_voxel_list = torch.empty(voxel_size,voxel_size,voxel_size
            ,27,3)
        for i in range(voxel_size):
            for j in range(voxel_size):
                for k in range(voxel_size):
                    center_voxel_list = torch.from_numpy(np.array([[i,j,k]]).repeat
                        (27,axis=0))
                    neighbour_movement_list = torch.tensor
                        ([[-1,-1,-1],[-1,-1,0],[-1,-1,1],[-1,0,-1],[-1,0,0],[-1,0,1],

                        [-1,1,-1],[-1,1,0],[-1,1,1],[0,-1,-1],[0,-1,0],[0,-1,1],[0,0,-1],

                        [0,0,0],[0,0,1],[0,1,-1],[0,1,0],[0,1,1],[1,-1,-1],[1,-1,0],
                        [1,-1,1],[1,0,-1],[1,0,0],[1,0,1],[1,1,-1],[1,1,0],[1,1,1]])
                    neighbour_list = center_voxel_list + neighbour_movement_list
                    neighbour_list = neighbour_list.int()
                    self.neighbour_voxel_list[i][j][k] = neighbour_list
        self.neighbour_voxel_list = self.neighbour_voxel_list.repeat([batch_size
            ,1,1,1,1,1]).int()
```

After initialization, the forward function caculated the corressponding voxel id for every point in a point cloud.

for every point cloud, we save the voxel message into a dictionary, the key is the voxel number caculated by voxel id. For every point in the point cloud, the voxel id is caculated by

$$voxel\_id = point\_value * (voxel\_size - 1)$$

$$voxel\_number = voxel\_x\_id * 10000 + voxel\_y\_id * 100 + voxel\_z\_id$$

The value in the dictionary is the point index of all points in certain voxel. To save time of assigning values to the dictionary, we decieded to update the dictionary once we collected all points in one voxel. In order to achieve this goal, firstly we sorted the voxel number list from small to large; secondly we find all the point in the list which is different from its previous one and call such point as cut point; lastly we assign all points between two cut points into one voxel.

In the same time, since most of the voxels in one point cloud are empty, it will save time in further caculation if we could mark which are empty and which are not in advance. So here we created a list called mask and memorized whether the voxels are empty or not.

```
def forward(self, point_cloud):
        size = point_cloud.size()
        index_voxels = []
        mask = []
        for i in range(size[0]): # batch
            index_voxels_tmp = dict()
            mask_tmp = np.zeros([self.voxel_size, self.voxel_size, self.voxel_size
                ])
            point_to_voxels = (point_cloud[i]*(self.voxel_size-1)).int()
            new_point_to_voxels = point_to_voxels[:,0]*10000+point_to_voxels
                [:,1]*100+point_to_voxels[:,2]
            sorted_point_to_voxels, sorted_point_indexes = torch.sort(
                new_point_to_voxels)
            current_list = []
            length = len(sorted_point_to_voxels)
            array1 = sorted_point_to_voxels[0:(length-1)]
            array2 = sorted_point_to_voxels[1:length]
```

```
            index = torch.arange(length-1)
            difference = index[array1!=array2]
            pre_item = 0
            for item in difference:
                cut_point_index = (point_cloud[i,sorted_point_indexes[item],:]*(
                    self.voxel_size-1)).int()
                self.set_voxel_value(index_voxels_tmp, sorted_point_indexes[
                    pre_item:(item+1)], cut_point_index,mask_tmp)
                pre_item = item + 1
            index_voxels.append(index_voxels_tmp)
            mask.append(mask_tmp)

        return index_voxels, self.neighbour_voxel_list, mask
```

2.2.2 RVS

We create a class named RVS for Random Voxel Sampling (RVS), so that each occupied voxel will have the same probability to be selected.

Given the generated voxel representation, the forward function returns a tensor of shape (B, self.npoints), where B is the batch_size and self.npoints indicates the number of sampled points. If the number of required center points is larger than the number of occupied voxels, then we will randomly search again all the occupied voxels. However, if a point has been selected, it will not be selected again.

```
# RVS Module
class RVS(nn.Module):
    def __init__(self, npoints):
        super(RVS, self).__init__()
        self.npoints = npoints

    #@profile
    def forward(self, pos, index_voxels):
        B = len(index_voxels) # batch_size
        device = pos.device
        vs = int(np.cbrt(len(index_voxels[0]))) # 64 -> 4, voxel_size
        centroids = torch.zeros(B, self.npoints, dtype=torch.long).to(device)
        centroids_index = []
        for batch in range(B):
            voxels_per_batch = index_voxels[batch]
            indexes = []
            dict_keys = voxels_per_batch.keys()
            len_key = len(dict_keys)
            if self.npoints <= len_key:
                selected_keys = random.sample(dict_keys,self.npoints)
                i = 0
                for per_key in selected_keys:
                    indexes.append([batch, per_key[0],per_key[1],per_key[2]])
                    val = voxels_per_batch.get(per_key)
                    length = len(val)
                    if (length == 1):
                        tem = 0
                    else:
                        tem = random.randint(0, len(val)-1)
                    index = int(val[tem])
                    centroids[batch, i] = index
                    i = i + 1
            else:
                selected_keys = dict_keys
```

```
                i = 0
                added = []
                for per_key in selected_keys:
                    indexes.append([batch, per_key[0],per_key[1],per_key[2]])
                    val = voxels_per_batch.get(per_key)
                    length = len(val)
                    if (length == 1):
                        tem = 0
                    else:
                        tem = random.randint(0, len(val)-1)
                    index = int(val[tem])
                    centroids[batch, i] = index
                    added.append(index)
                    i = i + 1
                add_num = 0
                while add_num < (self.npoints-len_key):
                    index = int(random.sample(range(pos.shape[1]), 1)[0])

                    if index not in added:
                        centroids[batch, len_key+add_num] = index
                        indexes.append(index)
                        add_num += 1
                        added.append(index)
            centroids_index.append(indexes)
            i = 0
        return centroids, centroids_index # centroid_index is not used

    # get value from self.index_voxels
    def get_voxel_value(self, index_voxels, voxel_size, batch, key):
        return index_voxels[batch].get(key)
```

2.2.3 FixedRadiusNearNeighbors

In this part, we use the context points as neighbour points. The context points for certain voxel are the points in the neighbour voxels. It has to be mentioned that since it is possible that several centroid points are in a same voxel. So repeatly caculate context points per center point will result in unnecessary time consuming. So here we use the same strategy as the voxel module forward function. we caculate voxel number for every center point and sort the voxel number list. In the sorted list, we could see clearly which center points are in the same voxel. In this way, the code caculates neighbour points for every voxel which has center points and assign context points to the center points in this voxel at the same time.

```
class FixedRadiusNNGraph(nn.Module):
    '''
    Build NN graph
    '''
    def __init__(self, radius, n_neighbor):
        super(FixedRadiusNNGraph, self).__init__()
        self.radius = radius
        self.n_neighbor = n_neighbor
        self.frnn = FixedRadiusNearNeighbors(radius, n_neighbor)
    def forward(self, pos, centroids, index_voxels, centroids_index, voxel_size,
        context_points, mask, feat=None):
        dev = pos.device
        group_idx = self.frnn(pos, centroids, centroids_index, index_voxels,
            voxel_size, context_points, mask)
        B, N, _ = pos.shape
```

```
        glist = []
        for i in range(B):
            center = torch.zeros((N)).to(dev)
            center[centroids[i]] = 1
            src = group_idx[i].contiguous().view(-1)
            src = src.to(dev)
            dst = centroids[i].view(-1, 1).repeat(1, self.n_neighbor).view(-1).
                float()
            dst = dst.to(dev)
            unified = torch.cat([src, dst])
            uniq, idx, inv_idx = np.unique(unified.cpu().numpy(), return_index=True
                , return_inverse=True)
            src_idx = inv_idx[:src.shape[0]]
            dst_idx = inv_idx[src.shape[0]:]
            g = dgl.DGLGraph((src_idx, dst_idx), readonly=True)
            g.ndata['pos'] = pos[i][uniq]
            g.ndata['center'] = center[uniq]
            if feat is not None:
                g.ndata['feat'] = feat[i][uniq]
            glist.append(g)
        bg = dgl.batch(glist)
        return bg
```

2.2.4 RelativePositionMessage

In this part, I add the concatenation of edges.src['pos'] and edges.dst['pos'] as the geometric feature and return the feature named as 'geo_feat'.

```
    def forward(self, edges):
        pos = edges.src['pos'] - edges.dst['pos']
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        pos = pos.to(device)
        if 'feat' in edges.src:
            res = torch.cat([pos, edges.src['feat']], 1)
        else:
            res = pos
        geo_feat = torch.cat([edges.src['pos'], edges.dst['pos']], 1)
        return {'agg_feat': res, 'geo_feat': geo_feat}
```

2.2.5 Grid_GCN_Conv

In this part, I add the geometric stream similar to the original stream, and then concatenate these two streams as h_all. In detail, nodes.mailbox['geo_feat'] corresponds to the return value of function RelativePositionMessage.

```
class Grid_GCN_Conv(nn.Module):
    '''
    Feature aggregation
    '''
    def __init__(self, sizes, batch_size):
        super(Grid_GCN_Conv, self).__init__()
        self.batch_size = batch_size
        self.conv = nn.ModuleList()
        self.bn = nn.ModuleList()
        self.sizes = sizes
        for i in range(1, len(sizes)):
            self.conv.append(nn.Conv2d(sizes[i-1], sizes[i], 1))
```

```
            self.bn.append(nn.BatchNorm2d(sizes[i]))
        # geometric
        self.conv_geo = nn.ModuleList()
        self.bn_geo = nn.ModuleList()
        for i in range(1, len(sizes)):
            if i == 1:
                self.conv_geo.append(nn.Conv2d(6, sizes[i], 1))
                self.bn_geo.append(nn.BatchNorm2d(sizes[i]))
            else:
                self.conv_geo.append(nn.Conv2d(sizes[i-1], sizes[i], 1))
                self.bn_geo.append(nn.BatchNorm2d(sizes[i]))

    def forward(self, nodes):
        shape = nodes.mailbox['agg_feat'].shape
        h = nodes.mailbox['agg_feat'].view(self.batch_size, -1, shape[1], shape[2])
            .permute(0, 3, 1, 2)
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        h = h.to(device)
        for conv, bn in zip(self.conv, self.bn):
            h = conv(h)
            h = bn(h)
            h = F.relu(h)
        h = torch.max(h, 3)[0]
        feat_dim = h.shape[1]
        h = h.permute(0, 2, 1).reshape(-1, feat_dim)
        # geometric
        shape = nodes.mailbox['geo_feat'].shape
        h_geo = nodes.mailbox['geo_feat'].view(self.batch_size, -1, shape[1], shape
            [2]).permute(0, 3, 1, 2)
        h_geo = h_geo.to(device)
        for conv, bn in zip(self.conv_geo, self.bn_geo):
            h_geo = conv(h_geo)
            h_geo = bn(h_geo)
            h_geo = F.relu(h_geo)
        h_geo = torch.max(h_geo, 3)[0]
        feat_dim = h_geo.shape[1]
        h_geo = h_geo.permute(0, 2, 1).reshape(-1, feat_dim)
        h_all = torch.cat([h, h_geo], 0)
        return {'new_feat': h_all}

    def group_all(self, pos, feat):
        '''
        Feature aggretation and pooling for the non-sampling layer
        '''
        if feat is not None:
            h = torch.cat([pos, feat], 2)
        else:
            h = pos
        shape = h.shape
        h = h.permute(0, 2, 1).view(shape[0], shape[2], shape[1], 1)
        # h_geo =
        # h_sematic =
        for conv, bn in zip(self.conv, self.bn):
            h = conv(h)
            h = bn(h)
            h = F.relu(h)
        h = torch.max(h[:, :, :, 0], 2)[0]
        # Do the same thing to h_geo and h_semantic
        return h
```

3 Future works

Since the biggest problem we are facing is that this code is too time-consuming using pure python, so the next step for this project is to do optimization using c code.

As for as I consider, we should change the VoxelModule and FixedRadiusNearNeighbor part of the code into C code, since the most time-consuming part in this code is how to cacu-lated the dictionary of voxel-to-point for every point cloud and get all neighbouring points for certain centroid point.

To caculate the speed of the code per line, please refer to the readme in the repo and run the code of train_tmp.py.