dim=3, c\_dim=32,paadding=0.1

**Encoder**

LocalPoolPointnet(

(fc\_pos): Linear(in\_features=3, out\_features=64, bias=True)

(blocks): ModuleList(

(0): ResnetBlockFC(

(fc\_0): Linear(in\_features=64, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

(shortcut): Linear(in\_features=64, out\_features=32, bias=False)

)

(1): ResnetBlockFC(

(fc\_0): Linear(in\_features=64, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

(shortcut): Linear(in\_features=64, out\_features=32, bias=False)

)

(2): ResnetBlockFC(

(fc\_0): Linear(in\_features=64, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

(shortcut): Linear(in\_features=64, out\_features=32, bias=False)

)

(3): ResnetBlockFC(

(fc\_0): Linear(in\_features=64, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

(shortcut): Linear(in\_features=64, out\_features=32, bias=False)

)

(4): ResnetBlockFC(

(fc\_0): Linear(in\_features=64, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

(shortcut): Linear(in\_features=64, out\_features=32, bias=False)

)

)

(fc\_c): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

(unet): UNet(

(down\_convs): ModuleList(

(0): DownConv(

(conv1): Conv2d(32, 32, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv2): Conv2d(32, 32, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(pool): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(1): DownConv(

(conv1): Conv2d(32, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(pool): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(2): DownConv(

(conv1): Conv2d(64, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv2): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(pool): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(3): DownConv(

(conv1): Conv2d(128, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv2): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

)

)

(up\_convs): ModuleList(

(0): UpConv(

(upconv): ConvTranspose2d(256, 128, kernel\_size=(2, 2), stride=(2, 2))

(conv1): Conv2d(256, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv2): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

)

(1): UpConv(

(upconv): ConvTranspose2d(128, 64, kernel\_size=(2, 2), stride=(2, 2))

(conv1): Conv2d(128, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

)

(2): UpConv(

(upconv): ConvTranspose2d(64, 32, kernel\_size=(2, 2), stride=(2, 2))

(conv1): Conv2d(64, 32, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv2): Conv2d(32, 32, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

)

)

(conv\_final): Conv2d(32, 32, kernel\_size=(1, 1), stride=(1, 1))

)

)

**Decoder**

LocalDecoder(

(fc\_c): ModuleList(

(0): Linear(in\_features=32, out\_features=32, bias=True)

(1): Linear(in\_features=32, out\_features=32, bias=True)

(2): Linear(in\_features=32, out\_features=32, bias=True)

(3): Linear(in\_features=32, out\_features=32, bias=True)

(4): Linear(in\_features=32, out\_features=32, bias=True)

)

(fc\_p): Linear(in\_features=3, out\_features=32, bias=True)

(blocks): ModuleList(

(0): ResnetBlockFC(

(fc\_0): Linear(in\_features=32, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

)

(1): ResnetBlockFC(

(fc\_0): Linear(in\_features=32, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

)

(2): ResnetBlockFC(

(fc\_0): Linear(in\_features=32, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

)

(3): ResnetBlockFC(

(fc\_0): Linear(in\_features=32, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

)

(4): ResnetBlockFC(

(fc\_0): Linear(in\_features=32, out\_features=32, bias=True)

(fc\_1): Linear(in\_features=32, out\_features=32, bias=True)

(actvn): ReLU()

)

)

(fc\_out): Linear(in\_features=32, out\_features=1, bias=True)

)

method: conv\_onet

data:

input\_type: pointcloud

classes: ["04256520", "03636649", "04401088", "03001627", "02933112","04379243","03211117"]

path: data/ShapeNet

pointcloud\_n: 3000

pointcloud\_noise: 0.005

points\_subsample: 2048

points\_file: points.npz

points\_iou\_file: points.npz

voxels\_file: null

model:

encoder: pointnet\_local\_pool

encoder\_kwargs:

hidden\_dim: 32

plane\_type: ['xz', 'xy', 'yz']

plane\_resolution: 64

unet: True

unet\_kwargs:

depth: 4

merge\_mode: concat

start\_filts: 32

decoder: simple\_local

decoder\_kwargs:

sample\_mode: bilinear # bilinear / nearest

hidden\_size: 32

c\_dim: 32

training:

out\_dir: out/pointcloud/shapenet\_3plane\_indoor

batch\_size: 32

model\_selection\_metric: iou

model\_selection\_mode: maximize

print\_every: 1000

visualize\_every: 10000

validate\_every: 5000 # 10000

checkpoint\_every: 2000

backup\_every: 10000

n\_workers: 8

n\_workers\_val: 4

test:

threshold: 0.2

eval\_mesh: true

eval\_pointcloud: false

model\_file: model\_best.pt

generation:

vis\_n\_outputs: 2

refine: false

n\_x: 128

n\_z: 1

import torch

import torch.nn as nn

import torch.nn.functional as F

from src.layers import ResnetBlockFC

from torch\_scatter import scatter\_mean, scatter\_max

from src.common import coordinate2index, normalize\_coordinate, normalize\_3d\_coordinate, map2local

from src.encoder.unet import Unet

class LocalPoolPointnet(nn.Module):

''' PointNet-based encoder network with ResNet blocks for each point.

Args:

c\_dim (int): dimension of latent code c

dim (int): input points dimension

hidden\_dim (int): hidden dimension of the network

scatter\_type (str): feature aggregation when doing local pooling

unet (bool): weather to use U-Net

unet\_kwargs (str): U-Net parameters

unet3d (bool): weather to use 3D U-Net

unet3d\_kwargs (str): 3D U-Net parameters

plane\_resolution (int): defined resolution for plane feature

grid\_resolution (int): defined resolution for grid feature

plane\_type (str): feature type, 'xz' - 1-plane, ['xz', 'xy', 'yz'] - 3-plane, ['grid'] - 3D grid volume

padding (float): conventional padding paramter of ONet for unit cube, so [-0.5, 0.5] -> [-0.55, 0.55]

n\_blocks (int): number of blocks ResNetBlockFC layers

'''

def \_\_init\_\_(self, c\_dim=128, dim=3, hidden\_dim=128, scatter\_type='max', n\_blocks=5

unet=False, unet\_kwargs=None, unet3d=False, unet3d\_kwargs=None,

plane\_resolution=None, grid\_resolution=None, plane\_type='xz', padding=0.1):

super().\_\_init\_\_()

self.c\_dim = c\_dim

self.fc\_pos = nn.Linear(dim, 2\*hidden\_dim)

self.blocks = nn.ModuleList([

ResnetBlockFC(2\*hidden\_dim, hidden\_dim) for i in range(n\_blocks) ])

self.fc\_c = nn.Linear(hidden\_dim, c\_dim)

self.actvn = nn.ReLU()

self.hidden\_dim = hidden\_dim

if unet: self.unet = UNet(c\_dim, in\_channels=c\_dim, \*\*unet\_kwargs)

else: self.unet = None

if unet3d: self.unet3d = UNet3D(\*\*unet3d\_kwargs)

else: self.unet3d = None

self.reso\_plane = plane\_resolution

self.reso\_grid = grid\_resolution

self.plane\_type = plane\_type

self.padding = padding

if scatter\_type == 'max':

self.scatter = scatter\_max

elif scatter\_type == 'mean':

self.scatter = scatter\_mean

else:

raise ValueError('incorrect scatter type')

def generate\_plane\_features(self, p, c, plane='xz'):

# acquire indices of features in plane

xy = normalize\_coordinate(p.clone(), plane=plane, padding=self.padding) # normalize to the range of (0, 1)

index = coordinate2index(xy, self.reso\_plane)

# scatter plane features from points

fea\_plane = c.new\_zeros(p.size(0), self.c\_dim, self.reso\_plane\*\*2)

c = c.permute(0, 2, 1) # B x 512 x T

fea\_plane = scatter\_mean(c, index, out=fea\_plane) # B x 512 x reso^2

fea\_plane = fea\_plane.reshape(p.size(0), self.c\_dim, self.reso\_plane, self.reso\_plane) # sparce matrix (B x 512 x reso x reso)

# process the plane features with UNet

if self.unet is not None:

fea\_plane = self.unet(fea\_plane)

return fea\_plane

def generate\_grid\_features(self, p, c):

p\_nor = normalize\_3d\_coordinate(p.clone(), padding=self.padding)

index = coordinate2index(p\_nor, self.reso\_grid, coord\_type='3d')

# scatter grid features from points

fea\_grid = c.new\_zeros(p.size(0), self.c\_dim, self.reso\_grid\*\*3)

c = c.permute(0, 2, 1)

fea\_grid = scatter\_mean(c, index, out=fea\_grid) # B x C x reso^3

fea\_grid = fea\_grid.reshape(p.size(0), self.c\_dim, self.reso\_grid, self.reso\_grid, self.reso\_grid) # sparce matrix (B x 512 x reso x reso)

if self.unet3d is not None: fea\_grid = self.unet3d(fea\_grid)

return fea\_grid

def pool\_local(self, xy, index, c):

bs, fea\_dim = c.size(0), c.size(2)

keys = xy.keys()

c\_out = 0

for key in keys:

# scatter plane features from points

if key == 'grid':

fea = self.scatter(c.permute(0, 2, 1), index[key], dim\_size=self.reso\_grid\*\*3)

else:

fea = self.scatter(c.permute(0, 2, 1), index[key], dim\_size=self.reso\_plane\*\*2)

if self.scatter == scatter\_max:

fea = fea[0]

# gather feature back to points

fea = fea.gather(dim=2, index=index[key].expand(-1, fea\_dim, -1))

c\_out += fea

return c\_out.permute(0, 2, 1)

def forward(self, p):

batch\_size, T, D = p.size()

# acquire the index for each point

coord = {}

index = {}

if 'xz' in self.plane\_type:

coord['xz'] = normalize\_coordinate(p.clone(), plane='xz', padding=self.padding)

index['xz'] = coordinate2index(coord['xz'], self.reso\_plane)

if 'xy' in self.plane\_type:

coord['xy'] = normalize\_coordinate(p.clone(), plane='xy', padding=self.padding)

index['xy'] = coordinate2index(coord['xy'], self.reso\_plane)

if 'yz' in self.plane\_type:

coord['yz'] = normalize\_coordinate(p.clone(), plane='yz', padding=self.padding)

index['yz'] = coordinate2index(coord['yz'], self.reso\_plane)

if 'grid' in self.plane\_type:

coord['grid'] = normalize\_3d\_coordinate(p.clone(), padding=self.padding)

index['grid'] = coordinate2index(coord['grid'], self.reso\_grid, coord\_type='3d')

net = self.fc\_pos(p)

net = self.blocks[0](net)

for block in self.blocks[1:]:

pooled = self.pool\_local(coord, index, net)

net = torch.cat([net, pooled], dim=2)

net = block(net)

c = self.fc\_c(net)

fea = {}

if 'grid' in self.plane\_type:

fea['grid'] = self.generate\_grid\_features(p, c)

if 'xz' in self.plane\_type:

fea['xz'] = self.generate\_plane\_features(p, c, plane='xz')

if 'xy' in self.plane\_type:

fea['xy'] = self.generate\_plane\_features(p, c, plane='xy')

if 'yz' in self.plane\_type:

fea['yz'] = self.generate\_plane\_features(p, c, plane='yz')

return fea

import torch

import torch.nn as nn

import torch.nn.functional as F

from src.layers import ResnetBlockFC

from src.common import normalize\_coordinate, normalize\_3d\_coordinate, map2local

class LocalDecoder(nn.Module):

''' Decoder.

Instead of conditioning on global features, on plane/volume local features.

Args:

dim (int): input dimension

c\_dim (int): dimension of latent conditioned code c

hidden\_size (int): hidden size of Decoder network

n\_blocks (int): number of blocks ResNetBlockFC layers

leaky (bool): whether to use leaky ReLUs

sample\_mode (str): sampling feature strategy, bilinear|nearest

padding (float): conventional padding paramter of ONet for unit cube, so [-0.5, 0.5] -> [-0.55, 0.55]

'''

def \_\_init\_\_(self, dim=3, c\_dim=128,

hidden\_size=256, n\_blocks=5, leaky=False, sample\_mode='bilinear', padding=0.1):

super().\_\_init\_\_()

self.c\_dim = c\_dim

self.n\_blocks = n\_blocks

if c\_dim != 0:

self.fc\_c = nn.ModuleList([

nn.Linear(c\_dim, hidden\_size) for i in range(n\_blocks)

])

self.fc\_p = nn.Linear(dim, hidden\_size)

self.blocks = nn.ModuleList([

ResnetBlockFC(hidden\_size) for i in range(n\_blocks)

])

self.fc\_out = nn.Linear(hidden\_size, 1)

if not leaky:

self.actvn = F.relu

else:

self.actvn = lambda x: F.leaky\_relu(x, 0.2)

self.sample\_mode = sample\_mode

self.padding = padding

def sample\_plane\_feature(self, p, c, plane='xz'):

# normalize to the range of (0, 1)

xy = normalize\_coordinate(p.clone(), plane=plane, padding=self.padding)

xy = xy[:, :, None].float()

vgrid = 2.0 \* xy - 1.0 # normalize to (-1, 1)

c = F.grid\_sample(c, vgrid, padding\_mode='border',

align\_corners=True, mode=self.sample\_mode).squeeze(-1)

return c

def sample\_grid\_feature(self, p, c):

# normalize to the range of (0, 1)

p\_nor = normalize\_3d\_coordinate(p.clone(), padding=self.padding)

p\_nor = p\_nor[:, :, None, None].float()

vgrid = 2.0 \* p\_nor - 1.0 # normalize to (-1, 1)

# acutally trilinear interpolation if mode = 'bilinear'

c = F.grid\_sample(c, vgrid, padding\_mode='border',

align\_corners=True, mode=self.sample\_mode).squeeze(-1).squeeze(-1)

return c

def forward(self, p, c\_plane, \*\*kwargs):

if self.c\_dim != 0:

plane\_type = list(c\_plane.keys())

c = 0

if 'grid' in plane\_type:

c += self.sample\_grid\_feature(p, c\_plane['grid'])

if 'xz' in plane\_type:

c += self.sample\_plane\_feature(p, c\_plane['xz'], plane='xz')

if 'xy' in plane\_type:

c += self.sample\_plane\_feature(p, c\_plane['xy'], plane='xy')

if 'yz' in plane\_type:

c += self.sample\_plane\_feature(p, c\_plane['yz'], plane='yz')

c = c.transpose(1, 2)

p = p.float()

net = self.fc\_p(p)

for i in range(self.n\_blocks):

if self.c\_dim != 0:

net = net + self.fc\_c[i](c)

net = self.blocks[i](net)

out = self.fc\_out(self.actvn(net))

out = out.squeeze(-1)

return out