model.py：

import torch

import torch.nn as nn

from torch.nn import init

from torchvision import models

from torch.autograd import Variable

from torch.nn import functional as F

######################################################################

class GeM(nn.Module):

# GeM zhedong zheng

def \_\_init\_\_(self, dim = 2048, p=3, eps=1e-6):

super(GeM, self).\_\_init\_\_()

self.p = nn.Parameter(torch.ones(dim)\*p, requires\_grad = True)

self.eps = eps

self.dim = dim

def forward(self, x):

return self.gem(x, p=self.p, eps=self.eps)

def gem(self, x, p=3, eps=1e-6):

x = torch.transpose(x, 1, -1)

x = x.clamp(min=eps).pow(p)

x = torch.transpose(x, 1, -1)

x = F.avg\_pool2d(x, (x.size(-2), x.size(-1)))

x = x.view(x.size(0), x.size(1))

x = x.pow(1./p)

return x

def \_\_repr\_\_(self):

return self.\_\_class\_\_.\_\_name\_\_ + '(' + 'p=' + '{:.4f}'.format(self.p.data.tolist()[0]) + ', ' + 'eps=' + str(self.eps) + ',' + 'dim='+str(self.dim)+')'

def weights\_init\_kaiming(m):

classname = m.\_\_class\_\_.\_\_name\_\_

if classname.find('Conv') != -1:

init.kaiming\_normal\_(m.weight.data, a=0, mode='fan\_in')

elif classname.find('Linear') != -1:

init.kaiming\_normal\_(m.weight.data, a=0, mode='fan\_out')

init.constant\_(m.bias.data, 0.0)

elif classname.find('BatchNorm1d') != -1:

init.normal\_(m.weight.data, 1.0, 0.02)

init.constant\_(m.bias.data, 0.0)

def weights\_init\_classifier(m):

classname = m.\_\_class\_\_.\_\_name\_\_

if classname.find('Linear') != -1:

init.normal\_(m.weight.data, std=0.001)

init.constant\_(m.bias.data, 0.0)

def fix\_relu(m):

classname = m.\_\_class\_\_.\_\_name\_\_

if classname.find('ReLU') != -1:

m.inplace=True

class ClassBlock(nn.Module):

def \_\_init\_\_(self, input\_dim, class\_num, droprate, relu=False, bnorm=True, num\_bottleneck=512, linear=True, return\_f = False):

super(ClassBlock, self).\_\_init\_\_()

self.return\_f = return\_f

add\_block = []

if linear:

add\_block += [nn.Linear(input\_dim, num\_bottleneck)]

else:

num\_bottleneck = input\_dim

if bnorm:

add\_block += [nn.BatchNorm1d(num\_bottleneck)]

if relu:

add\_block += [nn.LeakyReLU(0.1)]

if droprate>0:

add\_block += [nn.Dropout(p=droprate)]

add\_block = nn.Sequential(\*add\_block)

add\_block.apply(weights\_init\_kaiming)

classifier = []

classifier += [nn.Linear(num\_bottleneck, class\_num)]

classifier = nn.Sequential(\*classifier)

classifier.apply(weights\_init\_classifier)

self.add\_block = add\_block

self.classifier = classifier

def forward(self, x):

x = self.add\_block(x)

if self.return\_f:

f = x

x = self.classifier(x)

return x,f

else:

x = self.classifier(x)

return x

class ft\_net\_LPN(nn.Module):

def \_\_init\_\_(self, class\_num, droprate=0.5, stride=2, pool='avg', block=6):

super(ft\_net\_LPN, self).\_\_init\_\_()

model\_ft = models.resnet50(pretrained=True)

if stride == 1:

model\_ft.layer4[0].downsample[0].stride = (1,1)

model\_ft.layer4[0].conv2.stride = (1,1)

self.pool = pool

self.block = block

self.model = model\_ft

if pool == 'avg+max':

self.model.avgpool2 = nn.AdaptiveAvgPool2d((1,1))

self.model.maxpool2 = nn.AdaptiveMaxPool2d((1,1))

elif pool == 'avg':

self.model.avgpool2 = nn.AdaptiveAvgPool2d((1,1))

elif pool == 'max':

self.model.maxpool2 = nn.AdaptiveMaxPool2d((1,1))

elif pool == 'gem':

self.model.gem2 = GeM(dim=2048)

# 添加多个classifier（LPN核心）

self.classifiers = nn.ModuleList()

for \_ in range(block):

self.classifiers.append(ClassBlock(2048, class\_num, droprate))

def forward(self, x):

x = self.model.conv1(x)

x = self.model.bn1(x)

x = self.model.relu(x)

x = self.model.maxpool(x)

x = self.model.layer1(x)

x = self.model.layer2(x)

x = self.model.layer3(x)

x = self.model.layer4(x)

if self.pool == 'avg+max':

x1 = self.model.avgpool2(x)

x2 = self.model.maxpool2(x)

x = torch.cat((x1,x2), dim=1)

elif self.pool == 'avg':

x = self.model.avgpool2(x)

elif self.pool == 'max':

x = self.model.maxpool2(x)

elif self.pool == 'gem':

x = self.model.gem2(x)

# LPN处理：将特征图分割为多个水平条带

x = x.view(x.size(0), x.size(1), x.size(2), x.size(3))

features = []

parts = self.block

height = x.size(2)

for i in range(parts):

if i < (parts - 1):

split\_layer = x[:, :, int(i\*height/parts):int((i+1)\*height/parts), :]

else:

split\_layer = x[:, :, int(i\*height/parts):, :]

split\_layer = F.adaptive\_avg\_pool2d(split\_layer, (1,1))

split\_layer = split\_layer.view(split\_layer.size(0), -1)

features.append(split\_layer.unsqueeze(1))

features = torch.cat(features, 1)

outputs = []

for i in range(parts):

classifier = self.classifiers[i]

output = classifier(features[:,i,:])

outputs.append(output)

return torch.stack(outputs, dim=2)

class ft\_net(nn.Module):

def \_\_init\_\_(self, class\_num, droprate=0.5, stride=2, init\_model=None, pool='avg'):

super(ft\_net, self).\_\_init\_\_()

model\_ft = models.resnet50(pretrained=True)

if stride == 1:

model\_ft.layer4[0].downsample[0].stride = (1,1)

model\_ft.layer4[0].conv2.stride = (1,1)

self.pool = pool

if pool == 'avg+max':

model\_ft.avgpool2 = nn.AdaptiveAvgPool2d((1,1))

model\_ft.maxpool2 = nn.AdaptiveMaxPool2d((1,1))

elif pool == 'avg':

model\_ft.avgpool2 = nn.AdaptiveAvgPool2d((1,1))

elif pool == 'max':

model\_ft.maxpool2 = nn.AdaptiveMaxPool2d((1,1))

elif pool == 'gem':

model\_ft.gem2 = GeM(dim=2048)

self.model = model\_ft

if init\_model!=None:

self.model = init\_model.model

self.pool = init\_model.pool

def forward(self, x):

x = self.model.conv1(x)

x = self.model.bn1(x)

x = self.model.relu(x)

x = self.model.maxpool(x)

x = self.model.layer1(x)

x = self.model.layer2(x)

x = self.model.layer3(x)

x = self.model.layer4(x)

if self.pool == 'avg+max':

x1 = self.model.avgpool2(x)

x2 = self.model.maxpool2(x)

x = torch.cat((x1,x2), dim=1)

elif self.pool == 'avg':

x = self.model.avgpool2(x)

elif self.pool == 'max':

x = self.model.maxpool2(x)

elif self.pool == 'gem':

x = self.model.gem2(x)

x = x.view(x.size(0), x.size(1))

return x

# 其他原有模型定义（ft\_net\_VGG16, two\_view\_net, three\_view\_net）保持不变...

if \_\_name\_\_ == '\_\_main\_\_':

net = ft\_net\_LPN(751, block=6)

input = Variable(torch.FloatTensor(8, 3, 256, 256))

output = net(input)

print('LPN output shape:', output.shape)