NAIC2020ReID

说明材料

队伍：神圣兽国游尾郡窝窝乡独行族妖侠
队长：刘音
队员：李碧、王军杰、程枫

目录

NAIC2020ReID..... 1
说明材料.....1
一、 参赛项目 Git 仓库链接地址.....3
二、 核心代码展示..... 3
2.1 文件读取代码部分..... 3
2.2 模型创建代码部分..... 7
2.3 训练 loss 代码部分..... 20
2.4 测试推理代码部分..... 28
三、算法思路、亮点解读、建模算计与环境说明..... 35
四、解题思路详情..... 36
4.1 获取数据集..... 36
4.2 创建网络结构、损失计算及度量方式..... 36
五、 项目运行环境和运行办法..... 37
六、 训练时说明..... 37

一、参赛项目 Git 仓库链接地址

https://github.com/ly-wenli/NAIC_ReID.git

二、核心代码展示

2.1 文件读取代码部分

```
from .bases import BaseImageDataset
import os.path as osp
from collections import defaultdict
class NAIC(BaseImageDataset):
    def __init__(self,cfg, root='../data', verbose = True):
        super(NAIC, self).__init__()
        self.cfg = cfg
        self.dataset_dir = root
        self.mydataset_dir = osp.join(self.dataset_dir,'MyDataSet')
        # self.dataset_dir_train = osp.join(self.mydataset_dir, 'train')
        # self.dataset_dir_train_2019_cs = osp.join(self.mydataset_dir, 'train_2019_cs')
        # self.dataset_dir_train_2019_fs = osp.join(self.mydataset_dir, 'train_2019_fs')
        self.dataset_dir_test = osp.join(self.mydataset_dir, 'image_B_v1_1')

        train = self._process_dir(self.mydataset_dir, relabel=True)
        query_green, query_normal = self._process_dir_test(self.dataset_dir_test,    query = True)
        gallery_green, gallery_normal = self._process_dir_test(self.dataset_dir_test, query = False)
        if verbose:
            print("=> NAIC Competition data loaded")
            self.print_dataset_statistics(train, query_green+query_normal, gallery_green+gallery_normal)

        self.train = train
        self.query_green = query_green
        self.gallery_green = gallery_green
        self.query_normal = query_normal
        self.gallery_normal = gallery_normal

        self.num_train_pids, self.num_train_imgs, self.num_train_cams = self.get_imagedata_info(self.train)


    def _process_dir(self, data_dir, relabel=True): # train_2020_cs_path

        train_2020_cs_path = osp.join(data_dir,'train_2020_cs')
        train_2020_fs_path = osp.join(data_dir,'train_2020_fs')
        train_2019_cs_path = osp.join(data_dir,'train_2019_cs')
```

```
        train_2019_fs_path = osp.join(data_dir,'train_2019_fs')
        train_2020_fs_unlabel_path = osp.join(data_dir,'unlabel')
        filename_train_2020_cs = osp.join(train_2020_cs_path, 'new_train_list.txt')
        filename_train_2020_fs = osp.join(train_2020_fs_path, 'train_list.txt')
        filename_train_2019_cs = osp.join(train_2019_cs_path, 'new_2019_cs_train_list.txt')
        filename_train_2019_fs = osp.join(train_2019_fs_path, 'train_list.txt')
        filename_train_2020_fs_unlabel = osp.join(train_2020_fs_unlabel_path, 'label.txt')
        dataset = []
        camid = 1
        count_image=defaultdict(list)
        #load 2020 cs dataset
        with open(filename_train_2020_cs, 'r') as file_to_read:
            while True:
                lines = file_to_read.readline()
                if not lines:
                    break
                img_name,img_label = [i for i in lines.split(':')]
                # if img_name == 'train/105180993.png' or img_name=='train/829283568.png' or
img_name=='train/943445997.png': # remove samples with wrong label
                #     continue
                img_label = 'train_2020_cs_' + str(img_label)
                img_name = osp.join('images',img_name)
                img_name = osp.join(train_2020_cs_path,img_name)
                count_image[img_label].append(img_name)
        ccil_2020 = len(count_image)
        print("wenli:2020cs len is",ccil_2020)
        # load 2020 fs dataset
        with open(filename_train_2020_fs, 'r') as file_to_read:
            while True:
                lines = file_to_read.readline()
                if not lines:
                    break
                img_name, img_label = [i for i in lines.split(':')]
                # if img_name == 'train/105180993.png' or img_name=='train/829283568.png' or
img_name=='train/943445997.png': # remove samples with wrong label
                #     continue
                img_label = 'train_2020_fs_' + str(img_label)
                img_name = osp.join('images', img_name)
                img_name = osp.join(train_2020_fs_path, img_name)
                count_image[img_label].append(img_name)
        fcil_2020 = len(count_image)-ccil_2020
        print("wenli:2020fs len is",fcil_2020)
        # load 2019 cs dataset
        with open(filename_train_2019_cs, 'r') as file_to_read:
```

```
            while True:
                lines = file_to_read.readline()
                if not lines:
                    break
                img_name,img_label = [i for i in lines.split(':')]
                # if img_name == 'train/105180993.png' or img_name=='train/829283568.png' or
img_name=='train/943445997.png': # remove samples with wrong label
                #     continue
                img_label = 'train_2019_cs_' + str(img_label)
                img_name = osp.join(train_2019_cs_path, img_name)
                count_image[img_label].append(img_name)
        ccil_2019 = len(count_image)-ccil_2020-fcil_2020
        print("wenli:2019cs len is",ccil_2019)



        # loade 2019 fs dataset
        with open(filename_train_2019_fs, 'r') as file_to_read:
            while True:
                lines = file_to_read.readline()
                if not lines:
                    break
                img_name,img_label = [i for i in lines.split(' ')]
                if img_name == 'train/105180993.png' or img_name=='train/829283568.png' or
img_name=='train/943445997.png': # remove samples with wrong label
                    continue
                img_label = 'train_2019_fs_' + str(img_label)
                img_name = osp.join(train_2019_fs_path, img_name)
                count_image[img_label].append(img_name)
        fcil_2019 = len(count_image)-ccil_2020-fcil_2020-ccil_2019
        print("wenli:2019fs len is",fcil_2019)

        if self.cfg.DATASETS.USE_UNLABEL:
            # load 2020 fs unlabel dataset
            with open(filename_train_2020_fs_unlabel, 'r') as file_to_read:
                while True:
                    lines = file_to_read.readline()
                    if not lines:
                        break
                    img_name, img_label = [i for i in lines.split(':')]
                    # if img_name == 'train/105180993.png' or img_name=='train/829283568.png' or
img_name=='train/943445997.png': # remove samples with wrong label
                    #     continue
                    img_label = 'train_2020_fs_unlabel_' + str(img_label)
```

```
                    img_name = osp.join('images', img_name)
                    img_name = osp.join(train_2020_fs_unlabel_path, img_name)
                    count_image[img_label].append(img_name)
            unlabel_2020fs = len(count_image)-ccil_2020-fcil_2020-ccil_2019-fcil_2019
            print("wenli:unlabel len is", unlabel_2020fs)

        val_imgs = {}
        pid_container = set()
        for pid, img_name in count_image.items():
            if len(img_name) < 2:
                pass
            else:
                val_imgs[pid] = count_image[pid]
                pid_container.add(pid)
        pid_container = sorted(pid_container)
        pid2label = {pid: label for label, pid in enumerate(pid_container)}
        for pid, img_name in val_imgs.items():
            pid = pid2label[pid]
            for img in img_name:
                dataset.append((img, pid, camid))

        return dataset

    def _process_dir_test(self, data_dir, query=True):
        if query:
            subfix = 'query'
        else:
            subfix = 'gallery'

        datatype = ['green', 'normal']
        for index, type in enumerate(datatype):
            filename = osp.join(data_dir, '{}_{}.txt'.format(subfix, type))
            dataset = []
            with open(filename, 'r') as file_to_read:
                while True:
                    lines = file_to_read.readline()
                    if not lines:
                        break
                    for i in lines.split():
                        img_name = i

                    dataset.append((osp.join(self.dataset_dir_test, subfix, img_name), 1, 1))
            if index == 0:
                dataset_green = dataset
```

```
        return dataset_green, dataset
```

2.2 模型创建代码部分

```
import torch
import torch.nn as nn
from .backbones.resnet import ResNet, BasicBlock, Bottleneck
from .backbones.resnest import resnest50,resnest50_ibn,resnest101,resnest101_ibn
from loss.metric_learning import Arcface, Cosface, AMSoftmax, CircleLoss
from .backbones.resnet_ibn_a import resnet50_ibn_a,resnet101_ibn_a
from .backbones.se_resnet_ibn_a import se_resnet101_ibn_a,se_resnet50_ibn_a
from .backbones.resnet_ibn_b import resnet101_ibn_b,resnet50_ibn_b
import torch.nn.functional as F
from torch.nn.parameter import Parameter
from efficientnet_pytorch import EfficientNet

class GeM(nn.Module):

    def __init__(self, p=3.0, eps=1e-6, freeze_p=True):
        super(GeM, self).__init__()
        self.p = p if freeze_p else Parameter(torch.ones(1) * p)
        self.eps = eps

    def forward(self, x):
        return F.adaptive_avg_pool2d(x.clamp(min=self.eps).pow(self.p),
                                     (1, 1)).pow(1. / self.p)

    def __repr__(self):
        if isinstance(self.p, float):
            p = self.p
        else:
            p = self.p.data.tolist()[0]
        return self.__class__.__name__ +\
                '(' + 'p=' + '{:.4f}'.format(p) +\
                ', ' + 'eps=' + str(self.eps) + ')'

class ClassBlock(nn.Module):
    def __init__(self,input_dim,num_features=512,relu=True):
        super(ClassBlock,self).__init__()
        add_block = []
        add_block += [nn.Conv2d(input_dim,num_features,kernel_size=1,bias=False)]
        add_block += [nn.BatchNorm2d(num_features)]

        add_block = nn.Sequential(*add_block)
```

```
            add_block.apply(weights_init_kaiming)
            self.add_block = add_block
        def forward(self,x):
            x = self.add_block(x)
            x = torch.squeeze(x)
            return x
class PCB(nn.Module):
    def __init__(self,cfg,num_features,num_classes,dropout,out_planes,cut_at_pooling=False):
        super(PCB,self).__init__()
        self.cfg = cfg
        self.num_features = num_features
        self.num_classes = num_classes
        self.dropout = dropout
        self.cut_at_pooling = cut_at_pooling
        self.bn = nn.BatchNorm2d(out_planes)
        self.relu = nn.ReLU(inplace=True)

        self.local_conv = nn.Conv2d(out_planes,self.num_features,kernel_size=1,padding=0,bias=False)
        self.local_conv_list = nn.ModuleList()
        for i in range(6):
            self.local_conv_list.append(ClassBlock(out_planes,self.num_features))
        nn.init.kaiming_normal_(self.local_conv.weight,mode='fan_out')
        self.feat_bn2d = nn.BatchNorm2d(self.num_features)
        nn.init.constant_(self.feat_bn2d.weight,1)
        nn.init.constant_(self.feat_bn2d.bias,0)

        self.dy_weight0 = nn.Sequential(
            nn.AdaptiveAvgPool2d((1,1)),
            nn.Conv2d(out_planes,self.num_features,kernel_size=1),
            nn.Conv2d(self.num_features,self.num_features//16,kernel_size=1),
            nn.Conv2d(self.num_features//16,self.num_features,kernel_size=1),
            nn.Flatten(),
            nn.Linear(self.num_features,1)
        )
        self.dy_weight1 = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Conv2d(out_planes, self.num_features, kernel_size=1),
            nn.Conv2d(self.num_features, self.num_features // 16, kernel_size=1),
            nn.Conv2d(self.num_features // 16, self.num_features, kernel_size=1),
            nn.Flatten(),
            nn.Linear(self.num_features, 1)
        )
        self.dy_weight2 = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
```

```
        nn.Conv2d(out_planes, self.num_features, kernel_size=1),
        nn.Conv2d(self.num_features, self.num_features // 16, kernel_size=1),
        nn.Conv2d(self.num_features // 16, self.num_features, kernel_size=1),
        nn.Flatten(),
        nn.Linear(self.num_features, 1)
)
self.dy_weight3 = nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)),
        nn.Conv2d(out_planes, self.num_features, kernel_size=1),
        nn.Conv2d(self.num_features, self.num_features // 16, kernel_size=1),
        nn.Conv2d(self.num_features // 16, self.num_features, kernel_size=1),
        nn.Flatten(),
        nn.Linear(self.num_features, 1)
)
self.dy_weight4 = nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)),
        nn.Conv2d(out_planes, self.num_features, kernel_size=1),
        nn.Conv2d(self.num_features, self.num_features // 16, kernel_size=1),
        nn.Conv2d(self.num_features // 16, self.num_features, kernel_size=1),
        nn.Flatten(),
        nn.Linear(self.num_features, 1)
)
self.dy_weight5 = nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)),
        nn.Conv2d(out_planes, self.num_features, kernel_size=1),
        nn.Conv2d(self.num_features, self.num_features // 16, kernel_size=1),
        nn.Conv2d(self.num_features // 16, self.num_features, kernel_size=1),
        nn.Flatten(),
        nn.Linear(self.num_features, 1)
)

##------------------------------stipe1-begin-------------------------------#
self.instance0 = nn.Linear(self.num_features,self.num_classes)
nn.init.normal_(self.instance0.weight,std=0.001)
nn.init.constant_(self.instance0.bias,0)
##------------------------------stipe1-end-------------    ------------------#

##------------------------------stipe1-begin-------------------------------#
self.instance1 = nn.Linear(self.num_features, self.num_classes)
nn.init.normal_(self.instance1.weight, std=0.001)
nn.init.constant_(self.instance1.bias, 0)
##------------------------------stipe1-end-------------    ------------------#

##------------------------------stipe1-begin-------------------------------#
```

```
        self.instance2 = nn.Linear(self.num_features, self.num_classes)
        nn.init.normal_(self.instance2.weight, std=0.001)
        nn.init.constant_(self.instance2.bias, 0)
        ##------------------------------stipe1-end-------------    ------------------#

        ##------------------------------stipe1-begin-------------------------------#
        self.instance3 = nn.Linear(self.num_features, self.num_classes)
        nn.init.normal_(self.instance3.weight, std=0.001)
        nn.init.constant_(self.instance3.bias, 0)
        ##------------------------------stipe1-end-------------    ------------------#

        ##------------------------------stipe1-begin-------------------------------#
        self.instance4 = nn.Linear(self.num_features, self.num_classes)
        nn.init.normal_(self.instance4.weight, std=0.001)
        nn.init.constant_(self.instance4.bias, 0)
        ##------------------------------stipe1-end-------------    ------------------#

        ##------------------------------stipe1-begin-------------------------------#
        self.instance5 = nn.Linear(self.num_features, self.num_classes)
        nn.init.normal_(self.instance5.weight, std=0.001)
        nn.init.constant_(self.instance5.bias, 0)
        ##------------------------------stipe1-end-------------    ------------------#

        ##------------------------------stipe1-begin-------------------------------#
        self.instance_merge = nn.Linear(self.num_features*6, self.num_classes)
        nn.init.normal_(self.instance_merge.weight, std=0.001)
        nn.init.constant_(self.instance_merge.bias, 0)
        ##------------------------------stipe1-end-------------    ------------------#
        self.drop = nn.Dropout(self.dropout)
        self.adavgpool = nn.AdaptiveAvgPool2d((6,1))
    def forward(self,x):
        # x = self.relu(x)
        # sx = int(x.size(2)/6)
        # kx = int(x.size(2) - sx*5)
        # x = F.avg_pool2d(x,kernel_size=(kx,x.size(3)),stride=(sx,x.size(3)))
        #x = self.bn(x)
        # x = self.adavgpool(x)
        # part = {}
        # part_feat = {}
        # for i in range(6):
        #         part[i] = torch.unsqueeze(x[:,:,i,:],3)
        #         part_feat[i] = self.local_conv_list[i](part[i])
        #
        # d_weight = {}
```

```
            # d_weight[0] = self.dy_weight0(x)
            # d_weight[1] = self.dy_weight1(x)
            # d_weight[2] = self.dy_weight2(x)
            # d_weight[3] = self.dy_weight3(x)
            # d_weight[4] = self.dy_weight4(x)
            # d_weight[5] = self.dy_weight5(x)
            # weight_part_feat = {}
            # for i in range(6):
            #         weight_part_feat[i] = part_feat[i] * d_weight[i]
            # c0 = self.instance0(self.relu(part_feat[0]))
            # c1 = self.instance0(self.relu(part_feat[1]))
            # c2 = self.instance0(self.relu(part_feat[2]))
            # c3 = self.instance0(self.relu(part_feat[3]))
            # c4 = self.instance0(self.relu(part_feat[4]))
            # c5 = self.instance0(self.relu(part_feat[5]))




            # is model is test ,use this feature
            # print("*"*100)
            # print(x.shape)
            # print(x)
            # print("*" * 100)
            if not self.cfg.TEST.PCB_GLOBAL_FEAT_ENSEMBLE:
                """
                x = self.drop(x)
                x = self.local_conv(x)
                x = self.feat_bn2d(x)

                out_t = x

                d_weight0 = self.dy_weight0(x)
                d_weight1 = self.dy_weight1(x)
                d_weight2 = self.dy_weight2(x)
                d_weight3 = self.dy_weight3(x)
                d_weight4 = self.dy_weight4(x)
                d_weight5 = self.dy_weight5(x)
                # print("six weight")
                # print(d_weight0.shape, d_weight1.shape, d_weight2.shape, d_weight3.shape, d_weight4.shape,
d_weight5.shape)
                test_x = x.chunk(6, 2)
                test_x0 = test_x[0].contiguous().view(test_x[0].size(0), -1)*d_weight0
                test_x1 = test_x[1].contiguous().view(test_x[1].size(0), -1)*d_weight1
```

```
test_x2 = test_x[2].contiguous().view(test_x[2].size(0), -1)*d_weight2
test_x3 = test_x[3].contiguous().view(test_x[3].size(0), -1)*d_weight3
test_x4 = test_x[4].contiguous().view(test_x[4].size(0), -1)*d_weight4
test_x5 = test_x[5].contiguous().view(test_x[5].size(0), -1)*d_weight5
x = F.relu(x)

x = x.chunk(6, 2)
x0 = x[0].contiguous().view(x[0].size(0), -1)
x1 = x[1].contiguous().view(x[1].size(0), -1)
x2 = x[2].contiguous().view(x[2].size(0), -1)
x3 = x[3].contiguous().view(x[3].size(0), -1)
x4 = x[4].contiguous().view(x[4].size(0), -1)
x5 = x[5].contiguous().view(x[5].size(0), -1)
# print(x0.shape)
weight_x0 = x0*d_weight0
weight_x1 = x1*d_weight1
weight_x2 = x2*d_weight2
weight_x3 = x3*d_weight3
weight_x4 = x4*d_weight4
weight_x5 = x5*d_weight5



# linear feat,dont use in test.
c0 = self.instance0(x0)
c1 = self.instance1(x1)
c2 = self.instance2(x2)
c3 = self.instance3(x3)
c4 = self.instance4(x4)
c5 = self.instance5(x5)
"""
x = self.adavgpool(x)
part = {}
part_feat = {}
for i in range(6):
    part[i] = torch.unsqueeze(x[:, :, i, :], 3)
    part_feat[i] = self.local_conv_list[i](part[i])

d_weight = {}
d_weight[0] = self.dy_weight0(x)
d_weight[1] = self.dy_weight1(x)
d_weight[2] = self.dy_weight2(x)
d_weight[3] = self.dy_weight3(x)
d_weight[4] = self.dy_weight4(x)
```

```
                d_weight[5] = self.dy_weight5(x)
                weight_part_feat = {}
                for i in range(6):
                    weight_part_feat[i] = part_feat[i] * d_weight[i]
                c0 = self.instance0(part_feat[0])
                c1 = self.instance1(part_feat[1])
                c2 = self.instance2(part_feat[2])
                c3 = self.instance3(part_feat[3])
                c4 = self.instance4(part_feat[4])
                c5 = self.instance5(part_feat[5])

                pcb_merge_feat = torch.cat([weight_part_feat[0],weight_part_feat[1],weight_part_feat[2],weight_part_feat[3],weight_part_feat[4],weight_part_feat[5]],dim=1)
                # pcb_merge_feat = x0 + x1 + x2 + x3 + x4 + x5
                pcb_merge_feat_train = self.instance_merge(pcb_merge_feat)
                if not self.training:
                    if self.cfg.TEST.USE_PCB_MERGE_FEAT:
                        return pcb_merge_feat
                    else:
                        return (weight_part_feat[0],weight_part_feat[1],weight_part_feat[2],weight_part_feat[3],weight_part_feat[4],weight_part_feat[5])
                    # val_c0 = c0.view(c0.shape[0], -1)
                    # val_c1 = c1.view(c1.shape[0], -1)
                    # val_c2 = c2.view(c2.shape[0], -1)
                    # val_c3 = c3.view(c3.shape[0], -1)
                    # val_c4 = c4.view(c4.shape[0], -1)
                    # val_c5 = c5.view(c5.shape[0], -1)
                    # return (val_c0, val_c1, val_c2, val_c3, val_c4, val_c5)
                if self.cfg.MODEL.MERGE_PCB_FEAT:

                    return (c0, c1, c2, c3, c4, c5, pcb_merge_feat_train)
                else:
                    return (c0, c1, c2, c3, c4, c5)
            else:
                # this if modoule is use pcb global feature to ensemble distmat.
                if not self.training:
                    # print("x.shape",x.shape)

                    # use local_conv can raise 0.1 sorce:0.38(dont use is 0.37)

                    x = self.local_conv(x)
                    out0 = x / x.norm(2, 1).unsqueeze(1).expand_as(x)
```

```
        # wenli:use this GeM pooling can raise 0.2(use local_conv) sorce:0.37(not use is 0.36)

        out0 = GeM()(out0)
        out0 = out0.view(out0.shape[0], -1)

        # wenli:the next score is 0.07,dont use next method
        # x = self.local_conv(x)
        # x = self.feat_bn2d(x)
        # out0 = F.relu(x)
        # out0 = out0.view(out0.shape[0], -1)

        return out0
    else:
        x = self.drop(x)
        x = self.local_conv(x)
        x = self.feat_bn2d(x)

        d_weight0 = self.dy_weight(x)
        d_weight1 = self.dy_weight(x)
        d_weight2 = self.dy_weight(x)
        d_weight3 = self.dy_weight(x)
        d_weight4 = self.dy_weight(x)
        d_weight5 = self.dy_weight(x)

        x = F.relu(x)

        x = x.chunk(6, 2)
        x0 = x[0].contiguous().view(x[0].size(0), -1)
        x1 = x[1].contiguous().view(x[1].size(0), -1)
        x2 = x[2].contiguous().view(x[2].size(0), -1)
        x3 = x[3].contiguous().view(x[3].size(0), -1)
        x4 = x[4].contiguous().view(x[4].size(0), -1)
        x5 = x[5].contiguous().view(x[5].size(0), -1)


        weight_x0 = x0 * d_weight0
        weight_x1 = x1 * d_weight1
        weight_x2 = x2 * d_weight2
        weight_x3 = x3 * d_weight3
        weight_x4 = x4 * d_weight4
        weight_x5 = x5 * d_weight5
```

```
                c0 = self.instance0(x0)*0.1
                c1 = self.instance1(x1)*0.2
                c2 = self.instance2(x2)*0.5
                c3 = self.instance3(x3)*0.33
                c4 = self.instance4(x4)*0.8
                c5 = self.instance5(x5)*0.22

                pcb_merge_feat = torch.cat([weight_x0, weight_x1, weight_x2, weight_x3, weight_x4,
weight_x5], dim=1)
                # pcb_merge_feat = x0 + x1 + x2 + x3 + x4 + x5
                pcb_merge_feat_train = self.instance_merge(pcb_merge_feat)
                if self.cfg.MODEL.MERGE_PCB_FEAT:

                    return (c0, c1, c2, c3, c4, c5, pcb_merge_feat_train)
                else:
                    return (c0, c1, c2, c3, c4, c5)




def weights_init_kaiming(m):
    classname = m.__class__.__name__
    if classname.find('Linear') != -1:
        nn.init.kaiming_normal_(m.weight, a=0, mode='fan_out')
        nn.init.constant_(m.bias, 0.0)

    elif classname.find('Conv') != -1:
        nn.init.kaiming_normal_(m.weight, a=0, mode='fan_in')
        if m.bias is not None:
            nn.init.constant_(m.bias, 0.0)
    elif classname.find('BatchNorm') != -1:
        if m.affine:
            nn.init.constant_(m.weight, 1.0)
            nn.init.constant_(m.bias, 0.0)


def weights_init_classifier(m):
    classname = m.__class__.__name__
    if classname.find('Linear') != -1:
        nn.init.normal_(m.weight, std=0.001)
        if m.bias:
            nn.init.constant_(m.bias, 0.0)
```

```
class Backbone(nn.Module):
    def __init__(self, num_classes, cfg):
        super(Backbone, self).__init__()
        last_stride = cfg.MODEL.LAST_STRIDE
        model_path = cfg.MODEL.PRETRAIN_PATH
        model_name = cfg.MODEL.NAME
        self.cfg = cfg
        self.model_name = model_name
        pretrain_choice = cfg.MODEL.PRETRAIN_CHOICE
        self.cos_layer = cfg.MODEL.COS_LAYER
        self.neck = cfg.MODEL.NECK
        self.neck_feat = cfg.TEST.NECK_FEAT

        # self.in_planes = 1280
        # model_weight_b0 = EfficientNet.from_pretrained('efficientnet-b0')
        # model_weight_b0.to('cuda')
        # mm = nn.Sequential(*model_weight_b0.named_children())
        # self.base = model_weight_b0.extract_features
        #
        # from IPython import embed
        # embed()
        #print('using efficientnet-b0 as a backbone')

        if model_name == 'resnet50':
            self.in_planes = 2048
            self.base = ResNet(last_stride=last_stride,
                               block=Bottleneck, frozen_stages=cfg.MODEL.FROZEN,
                               layers=[3, 4, 6, 3])
            print('using resnet50 as a backbone')
        elif model_name == 'resnet50_ibn_a':
            self.in_planes = 2048
            self.base = resnet50_ibn_a(last_stride)
            print('using resnet50_ibn_a as a backbone')
        elif model_name == 'resnet101_ibn_a':
            self.in_planes = 2048
            self.base = resnet101_ibn_a(last_stride, frozen_stages=cfg.MODEL.FROZEN)
            print('using resnet101_ibn_a as a backbone')
        elif model_name == 'se_resnet101_ibn_a':
            self.in_planes = 2048
            self.base = se_resnet101_ibn_a(last_stride)
            print('using se_resnet101_ibn_a as a backbone')
        elif model_name == 'se_resnet50_ibn_a':
```

```
        self.in_planes = 2048
        self.base = se_resnet50_ibn_a(last_stride)
        print('using se_resnet101_ibn_a as a backbone')
    elif model_name == 'resnet101_ibn_b':
        self.in_planes = 2048
        self.base = resnet101_ibn_b(last_stride)
        print('using resnet101_ibn_b as a backbone')
    elif model_name == 'resnet50_ibn_b':
        self.in_planes = 2048
        self.base = resnet50_ibn_b(last_stride)
        print('using resnet50_ibn_b as a backbone')
    elif model_name == 'resnest50':
        self.in_planes = 2048
        self.base = resnest50(last_stride)
        print('using resnest50 as a backbone')
    elif model_name == 'resnest50_ibn':
        self.in_planes = 2048
        self.base = resnest50_ibn(last_stride)
        print('using resnest50_ibn as a backbone')
    elif model_name == 'resnest101':
        self.in_planes = 2048
        self.base = resnest101(last_stride)
        print('using resnest101 as a backbone')
    elif model_name == 'resnest101_ibn':
        self.in_planes = 2048
        self.base = resnest101_ibn(last_stride)
        print('using resnest101_ibn as a backbone')
    elif model_name == 'efficientnet_b7':
        # self.in_planes = 1280
        #
        # model_weight_b0 = EfficientNet.from_pretrained('efficientnet-b0')
        # model_weight_b0.to('cuda')
        # self.base = model_weight_b0.extract_features
        self.base = EfficientNet.from_pretrained('efficientnet-b0')
        self.in_planes = self.base._fc.in_features
        print('using efficientnet-b0 as a backbone')
    else:
        print('unsupported backbone! but got {}'.format(model_name))

    if pretrain_choice == 'imagenet' and model_name != 'efficientnet_b7':
        # if model_name == 'efficientnet_b7':
        #         state_dict = torch.load(model_path)
        #         # self.base.load_state_dict(state_dict)
        #         if 'state_dict' in state_dict:
```

```
                #               param_dict = state_dict['state_dict']
                #         for i in param_dict:
                #               if 'fc' in i:
                #                     continue
                #               self.state_dict()[i.replace('module.', '')].copy_(param_dict[i])
                # else:
                self.base.load_param(model_path)
                print('Loading pretrained ImageNet model......from {}'.format(model_path))

        if cfg.MODEL.POOLING_METHOD == 'GeM':
                print('using GeM pooling')
                self.gap = GeM()
        else:
                self.gap = nn.AdaptiveAvgPool2d(1)
        if cfg.MODEL.IF_USE_PCB:
                self.pcb = PCB(cfg,256,num_classes,0.5,self.in_planes,cut_at_pooling=False)
        self.num_classes = num_classes
        self.ID_LOSS_TYPE = cfg.MODEL.ID_LOSS_TYPE
        if self.ID_LOSS_TYPE == 'arcface':
                print('using {} with s:{}, m: {}'.format(self.ID_LOSS_TYPE,cfg.SOLVER.COSINE_SCALE,cfg.SOLVER.COSINE_MARGIN))
                self.classifier = Arcface(self.in_planes, self.num_classes,
                                          s=cfg.SOLVER.COSINE_SCALE, m=cfg.SOLVER.COSINE_MARGIN)
        elif self.ID_LOSS_TYPE == 'cosface':
                print('using {} with s:{}, m: {}'.format(self.ID_LOSS_TYPE,cfg.SOLVER.COSINE_SCALE,cfg.SOLVER.COSINE_MARGIN))
                self.classifier = Cosface(self.in_planes, self.num_classes,
                                          s=cfg.SOLVER.COSINE_SCALE, m=cfg.SOLVER.COSINE_MARGIN)
        elif self.ID_LOSS_TYPE == 'amsoftmax':
                print('using {} with s:{}, m: {}'.format(self.ID_LOSS_TYPE,cfg.SOLVER.COSINE_SCALE,cfg.SOLVER.COSINE_MARGIN))
                self.classifier = AMSoftmax(self.in_planes, self.num_classes,
                                            s=cfg.SOLVER.COSINE_SCALE, m=cfg.SOLVER.COSINE_MARGIN)
        elif self.ID_LOSS_TYPE == 'circle':
                print('using {} with s:{}, m: {}'.format(self.ID_LOSS_TYPE,cfg.SOLVER.COSINE_SCALE,cfg.SOLVER.COSINE_MARGIN))
                self.classifier = CircleLoss(self.in_planes, self.num_classes,
                                            s=cfg.SOLVER.COSINE_SCALE, m=cfg.SOLVER.COSINE_MARGIN)
        else:
                self.classifier = nn.Linear(self.in_planes, self.num_classes, bias=False)
                self.classifier.apply(weights_init_classifier)

        self.bottleneck = nn.BatchNorm1d(self.in_planes)
        self.bottleneck.bias.requires_grad_(False)
```

```
        self.bottleneck.apply(weights_init_kaiming)
        # from IPython import embed
        # embed()
    def forward(self, x, label=None):    # label is unused if self.cos_layer == 'no'
        # device = 'cuda'
        # x.to(device)
        # from IPython import    embed
        # embed()
        #print("x.shape",x.shape)
        if 'efficientnet_b7' == self.model_name:
            x = self.base.extract_features(x)
        else:

            x = self.base(x)
        if self.cfg.MODEL.IF_USE_PCB:
            pcb_out = self.pcb(x)

        # print("x.shape",x.shape)
        global_feat = self.gap(x)
        # print("global_feat.shape",global_feat.shape)
        # print("pcb_out.shape",pcb_out.shape)
        global_feat = global_feat.view(global_feat.shape[0], -1)    # flatten to (bs, 2048)
        feat = self.bottleneck(global_feat)

        if self.neck == 'no':
            feat = global_feat
        elif self.neck == 'bnneck':
            feat = self.bottleneck(global_feat)

        if self.training:
            if self.ID_LOSS_TYPE in ('arcface', 'cosface', 'amsoftmax', 'circle'):
                cls_score = self.classifier(feat, label)
            else:
                cls_score = self.classifier(feat)
            if self.cfg.MODEL.IF_USE_PCB:
                return cls_score, global_feat, pcb_out
            else:
                return cls_score, global_feat
        else:
            if self.neck_feat == 'after':
                # print("Test with feature after BN")
                if self.cfg.MODEL.IF_USE_PCB:
                    return feat, pcb_out
                else:
```

```
                return feat
            else:
                # print("Test with feature before BN")
                if self.cfg.MODEL.IF_USE_PCB:
                    return global_feat, pcb_out
                else:
                    return global_feat

    def load_param(self, trained_path):
        param_dict = torch.load(trained_path)
        for i in param_dict:
            if 'classifier' in i or 'arcface' in i:
                continue
            self.state_dict()[i.replace('module.','')].copy_(param_dict[i])
        print('Loading pretrained model from {}'.format(trained_path))

    def load_param_finetune(self, model_path):
        param_dict = torch.load(model_path)
        for i in param_dict:
            self.state_dict()[i].copy_(param_dict[i])
        print('Loading pretrained model for finetuning from {}'.format(model_path))

def make_model(cfg, num_class):
    model = Backbone(num_class, cfg)
    return model
```

2.3 训练 loss 代码部分

```
import logging
import numpy as np
import os
import time
import torch
import torch.nn as nn
import cv2
from utils.meter import AverageMeter
from utils.metrics import R1_mAP, R1_mAP_Pseudo
import json
import datetime
from solver import make_optimizer, WarmupMultiStepLR
import torch.distributed as dist
try:
    from apex.parallel import DistributedDataParallel as DDP
    from apex.fp16_utils import *
    from apex import amp, optimizers
```

```
        from apex.multi_tensor_apply import multi_tensor_applier
except ImportError:
        raise ImportError("Please install apex from https://www.github.com/nvidia/apex to run this example.")



def pcb_loss_forward(pcb_feat,targets):
        # print("inputs.device",inputs.device)
        # print("next(model.parameters()).device",next(model.parameters()).device)
        #
        # outputs = model(inputs)
        criterion = nn.CrossEntropyLoss()
                #
                # global loss0
                # global loss1
                # global loss2
                # global loss3
                # global loss4
                # global loss5
        loss0 = criterion(pcb_feat[0],targets)
        loss1 = criterion(pcb_feat[1],targets)
        loss2 = criterion(pcb_feat[2],targets)
        loss3 = criterion(pcb_feat[3],targets)
        loss4 = criterion(pcb_feat[4],targets)
        loss5 = criterion(pcb_feat[5],targets)
        loss_merge = criterion(pcb_feat[6],targets)
        return loss0,loss1,loss2,loss3,loss4,loss5,loss_merge

def get_pcb_optimizer(model):
        if hasattr(model.module,'base'):
                base_param_ids = set(map(id,model.module.base.parameters()))
                new_params = [p for p in model.parameters() if
                                        id(p) not in base_param_ids]
                param_groups = [
                        {'params': model.module.base.parameters(),'lr_mult': 0.1},
                        {'params': new_params,'lr_mult': 1.0}
                ]
        else:
                param_groups = model.parameters()
        optimizers = torch.optim.SGD(param_groups,lr=0.1,momentum=0.9,weight_decay=5e-4,nesterov=True)
        return optimizers
```

```
def do_train(cfg,
                    model,
                    center_criterion,
                    train_loader,
                    val_loader,
                    optimizer,
                    optimizer_center,
                    scheduler,
                    loss_fn,
                    num_query):
    log_period = cfg.SOLVER.LOG_PERIOD
    checkpoint_period = cfg.SOLVER.CHECKPOINT_PERIOD

    device = "cuda"
    epochs = cfg.SOLVER.MAX_EPOCHS

    logger = logging.getLogger("reid_baseline.train")
    logger.info('start training')

    if device:
        # dist.init_process_group(backend='nccl',init_method='env://')

        model.to(device)
        if torch.cuda.device_count() > 1:
            print('Using {} GPUs for training'.format(torch.cuda.device_count()))

            model, optimizer = amp.initialize(model, optimizer, opt_level='O1')

            model = nn.DataParallel(model)
            # model = torch.nn.parallel.DistributedDataParallel(model,find_unused_parameters=True)
        else:
            if cfg.SOLVER.FP16:
                model, optimizer = amp.initialize(model, optimizer, opt_level='O1')

    loss_meter = AverageMeter()
    all_loss_meter = AverageMeter()
    acc_meter = AverageMeter()
    pcb_losses = AverageMeter()
    pcb_merge_losses = AverageMeter()

    pcb_optimizer = get_pcb_optimizer(model)
    pcb_scheduler = WarmupMultiStepLR(pcb_optimizer, cfg.SOLVER.STEPS, cfg.SOLVER.GAMMA,
                                      cfg.SOLVER.WARMUP_FACTOR,
                                      cfg.SOLVER.WARMUP_EPOCHS, cfg.SOLVER.WARMUP_METHOD)
```

```
    # train
    for epoch in range(1, epochs + 1):
        start_time = time.time()
        loss_meter.reset()
        all_loss_meter.reset()
        acc_meter.reset()
        pcb_losses.reset()
        pcb_merge_losses.reset()

        model.train()
        for n_iter, (img, vid) in enumerate(train_loader):
            optimizer.zero_grad()
            optimizer_center.zero_grad()
            img = img.to(device)
            target = vid.to(device)

            if cfg.MODEL.IF_USE_PCB:
                score, feat, pcb_out = model(img, target)

                loss = loss_fn(score, feat, target)
                loss0, loss1, loss2, loss3, loss4, loss5,loss_merge = pcb_loss_forward(pcb_feat=pcb_out,
targets=target)
                pcb_loss = (loss0 + loss1 + loss2 + loss3 + loss4 + loss5) / 6
                all_loss = loss + 0.5 * pcb_loss + 0.5 * loss_merge

                if cfg.SOLVER.FP16:
                    with amp.scale_loss(all_loss, optimizer) as scaled_loss:
                        scaled_loss.backward()
                else:
                    all_loss.backward()


                optimizer.step()
                if 'center' in cfg.MODEL.METRIC_LOSS_TYPE:
                    for param in center_criterion.parameters():
                        param.grad.data *= (1. / cfg.SOLVER.CENTER_LOSS_WEIGHT)
                    optimizer_center.step()

                acc = (score.max(1)[1] == target).float().mean()
                loss_meter.update(loss.item(), img.shape[0])
                all_loss_meter.update(all_loss.item(), img.shape[0])
                pcb_losses.update(pcb_loss.item(), img.shape[0])
                pcb_merge_losses.update(loss_merge.item(),img.shape[0])
```

```
                acc_meter.update(acc, 1)

                if (n_iter + 1) % log_period == 0:
                    logger.info("Epoch[{}] Iteration[{}/{}] All_Loss: {:.3f},Global_Loss: {:.3f},PCB_Loss: {:.3f},Merge_Loss: {:.3f}, Acc: {:.3f}, Base Lr: {:.2e}"
                                .format(epoch, (n_iter + 1), len(train_loader),
                                        all_loss_meter.avg,loss_meter.avg, pcb_losses.avg,pcb_merge_losses.avg,acc_meter.avg, scheduler.get_lr()[0]))
            else:
                score, feat = model(img, target)


                loss = loss_fn(score, feat, target)
                if cfg.SOLVER.FP16:
                    with amp.scale_loss(loss, optimizer) as scaled_loss:
                                scaled_loss.backward()
                else:
                    loss.backward()
                optimizer.step()
                if 'center' in cfg.MODEL.METRIC_LOSS_TYPE:
                    for param in center_criterion.parameters():
                        param.grad.data *= (1. / cfg.SOLVER.CENTER_LOSS_WEIGHT)
                    optimizer_center.step()

                acc = (score.max(1)[1] == target).float().mean()
                loss_meter.update(loss.item(), img.shape[0])
                # all_loss_meter.update(all_loss.item(), img.shape[0])
                # pcb_losses.update(pcb_loss.item(), img.shape[0])
                acc_meter.update(acc, 1)

                if (n_iter + 1) % log_period == 0:
                    logger.info("Epoch[{}] Iteration[{}/{}] Global_Loss: {:.3f}, Acc: {:.3f}, Base Lr: {:.2e}"
                                .format(epoch, (n_iter + 1), len(train_loader),
                                        loss_meter.avg,acc_meter.avg, scheduler.get_lr()[0]))


        # if cfg.SOLVER.FP16:
        #         with amp.scale_loss(loss, optimizer) as scaled_loss:
        #             scaled_loss.backward()
        # else:
        #         loss.backward(retain_graph=True)

        # loss0, loss1, loss2, loss3, loss4, loss5 = pcb_loss_forward(pcb_feat=pcb_out, targets=target)
        # pcb_loss = (loss0 + loss1 + loss2 + loss3 + loss4 + loss5) / 6
```

```
                # all_loss = 0.1 * loss + 0.9 * pcb_loss

                # if cfg.SOLVER.FP16:
                #         with amp.scale_loss(all_loss, optimizer) as scaled_loss:
                #                 scaled_loss.backward()
                # else:
                #         all_loss.backward()

                # wenli:if use mulit task to train ,may overfit.Deprecated use this method
                # pcb_optimizer.zero_grad()
                # torch.autograd.backward([loss0, loss1, loss2, loss3, loss4, loss5],
                #                                                 [torch.ones(1)[0].cuda(),  torch.ones(1)[0].cuda(),
torch.ones(1)[0].cuda(),
                #                                                  torch.ones(1)[0].cuda(),  torch.ones(1)[0].cuda(),
torch.ones(1)[0].cuda(),
                #                                         torch.ones(1)[0].cuda()])
                # pcb_optimizer.step()



                # optimizer.step()
                # if 'center' in cfg.MODEL.METRIC_LOSS_TYPE:
                #         for param in center_criterion.parameters():
                #                 param.grad.data *= (1. / cfg.SOLVER.CENTER_LOSS_WEIGHT)
                #         optimizer_center.step()

                # acc = (score.max(1)[1] == target).float().mean()
                # loss_meter.update(loss.item(), img.shape[0])
                # all_loss_meter.update(all_loss.item(), img.shape[0])
                # pcb_losses.update(pcb_loss.item(), img.shape[0])
                # acc_meter.update(acc, 1)

                # if (n_iter + 1) % log_period == 0:
                #         logger.info("Epoch[{}] Iteration[{}/{}] All_Loss: {:.3f},Global_Loss: {:.3f},PCB_Loss: {:.3f}, Acc:
{:.3f}, Base Lr: {:.2e}"
                #                         .format(epoch, (n_iter + 1), len(train_loader),
                #                                         all_loss_meter.avg,loss_meter.avg, pcb_losses.avg,acc_meter.avg,
scheduler.get_lr()[0]))

        #pcb_scheduler.step()
        scheduler.step()
        end_time = time.time()
        time_per_batch = (end_time - start_time) / (n_iter + 1)
```

```
            logger.info("Epoch {} done. Time per batch: {:.3f}[s] Speed: {:.1f}[samples/s]"
                        .format(epoch, time_per_batch, train_loader.batch_size / time_per_batch))

            if epoch % checkpoint_period == 0:
                torch.save(model.state_dict(), os.path.join(cfg.OUTPUT_DIR, cfg.MODEL.NAME +
'_{}.pth'.format(epoch)))


def do_inference(cfg,
                 model,
                 val_loader_green,
                 val_loader_normal,
                 num_query_green,
                 num_query_normal):
    device = "cuda"
    logger = logging.getLogger("reid_baseline.test")
    logger.info("Enter inferencing")

    if device:
        if torch.cuda.device_count() > 1:
            print('Using {} GPUs for inference'.format(torch.cuda.device_count()))
            model = nn.DataParallel(model)
        model.to(device)

    model.eval()
    val_loader = [val_loader_green, val_loader_normal]
    for index, loader in enumerate(val_loader):
        if index == 0:
            subfix = '1'
            reranking_parameter = [30, 2, 0.8]
            evaluator = R1_mAP(cfg,num_query_green, max_rank=200, feat_norm=cfg.TEST.FEAT_NORM,
                               reranking=cfg.TEST.RE_RANKING)
        else:
            subfix = '2'
            reranking_parameter = [30, 2, 0.8]
            evaluator = R1_mAP(cfg,num_query_normal, max_rank=200, feat_norm=cfg.TEST.FEAT_NORM,
                               reranking=cfg.TEST.RE_RANKING)

        evaluator.reset()
        DISTMAT_PATH = os.path.join(cfg.OUTPUT_DIR, "distmat_{}.npy".format(subfix))
        QUERY_PATH = os.path.join(cfg.OUTPUT_DIR, "query_path_{}.npy".format(subfix))
        GALLERY_PATH = os.path.join(cfg.OUTPUT_DIR, "gallery_path_{}.npy".format(subfix))

        for n_iter, (img, pid, camid, imgpath) in enumerate(loader):
```

```
            with torch.no_grad():
                img = img.to(device)

                if cfg.TEST.FLIP_FEATS == 'on':
                    # if model_name != efficientnet_b* ,use this feat
                    # feat = torch.FloatTensor(img.size(0), 2048).zero_().cuda()
                    feat = torch.FloatTensor(img.size(0), 2048).zero_().cuda()
                    for i in range(2):
                        if i == 1:
                            inv_idx = torch.arange(img.size(3) - 1, -1, -1).long().cuda()
                            img = img.index_select(3, inv_idx)
                        if cfg.MODEL.IF_USE_PCB:
                            #print("image shape is ", img.shape)
                            f,_ = model(img)
                        else:
                            f = model(img)
                        feat = feat + f
                else:
                    feat,_ = model(img)
                if cfg.MODEL.IF_USE_PCB:
                    _,pcb_feat = model(img)
                    if cfg.TEST.USE_PCB_MERGE_FEAT:
                        evaluator.update_pcb((feat,pcb_feat, imgpath))
                    else:
                        evaluator.update_pcb_split((feat,pcb_feat,imgpath))
                else:
                    evaluator.update((feat, imgpath))
        # if cfg.MODEL.IF_USE_PCB:
        #         if not cfg.TEST.USE_PCB_MERGE_FEAT:
        #             evaluator.update_split()

        data, distmat, img_name_q, img_name_g = evaluator.compute(reranking_parameter)
        np.save(DISTMAT_PATH, distmat)
        np.save(QUERY_PATH, img_name_q)
        np.save(GALLERY_PATH, img_name_g)

        if index == 0:
            data_1 = data

data_all = {**data_1, **data}
nowTime = datetime.datetime.now().strftime('%Y-%m-%d-%H-%M-%S')
with open(os.path.join(cfg.OUTPUT_DIR, 'result_{}.json'.format(nowTime)), 'w',encoding='utf-8') as fp:
    json.dump(data_all, fp)
```

```
def do_inference_Pseudo(cfg,
                        model,
                        val_loader,
                        num_query
                        ):
    device = "cuda"

    evaluator = R1_mAP_Pseudo(num_query, max_rank=200, feat_norm=cfg.TEST.FEAT_NORM)
    evaluator.reset()
    if device:
        if torch.cuda.device_count() > 1:
            print('Using {} GPUs for inference'.format(torch.cuda.device_count()))
            model = nn.DataParallel(model)
        model.to(device)

    reranking_parameter = [14, 4, 0.4]

    model.eval()
    for n_iter, (img, pid, camid, imgpath) in enumerate(val_loader):
        with torch.no_grad():
            img = img.to(device)
            if cfg.TEST.FLIP_FEATS == 'on':
                feat = torch.FloatTensor(img.size(0), 2048).zero_().cuda()
                for i in range(2):
                    if i == 1:
                        inv_idx = torch.arange(img.size(3) - 1, -1, -1).long().cuda()
                        img = img.index_select(3, inv_idx)
                    f = model(img)
                    feat = feat + f
            else:
                feat = model(img)

            evaluator.update((feat, imgpath))

    distmat, img_name_q, img_name_g = evaluator.compute(reranking_parameter)

    return distmat, img_name_q, img_name_g
```

2.4 测试推理代码部分

```
import torch
import numpy as np
import os
from utils.reranking import re_ranking
from scipy.spatial.distance import cdist
```

```
def euclidean_distance(qf, gf):
    m = qf.shape[0]
    n = gf.shape[0]
    dist_mat = torch.pow(qf, 2).sum(dim=1, keepdim=True).expand(m, n) + \
                   torch.pow(gf, 2).sum(dim=1, keepdim=True).expand(n, m).t()
    dist_mat.addmm_( qf, gf.t(),beta=1, alpha=-2)
    return dist_mat.cpu().numpy()

def cosine_similarity(qf, gf):
    epsilon = 0.00001
    dist_mat = qf.mm(gf.t())
    qf_norm = torch.norm(qf, p=2, dim=1, keepdim=True)    # mx1
    gf_norm = torch.norm(gf, p=2, dim=1, keepdim=True)    # nx1
    qg_normdot = qf_norm.mm(gf_norm.t())

    dist_mat = dist_mat.mul(1 / qg_normdot).cpu().numpy()
    dist_mat = np.clip(dist_mat, -1 + epsilon, 1 - epsilon)
    dist_mat = np.arccos(dist_mat)
    return dist_mat

def eval_func(distmat, q_pids, g_pids, q_camids, g_camids, max_rank=50):
    """Evaluation with market1501 metric
        Key: for each query identity, its gallery images from the same camera view are discarded.
        """
    num_q, num_g = distmat.shape
    # distmat g
    #       q       1 3 2 4
    #               4 1 2 3
    if num_g < max_rank:
        max_rank = num_g
        print("Note: number of gallery samples is quite small, got {}".format(num_g))
    indices = np.argsort(distmat, axis=1)
    #    0 2 1 3
    #    1 2 3 0
    matches = (g_pids[indices] == q_pids[:, np.newaxis]).astype(np.int32)
    # compute cmc curve for each query
    all_cmc = []
    all_AP = []
    num_valid_q = 0.    # number of valid query
    for q_idx in range(num_q):
        # get query pid and camid
        q_pid = q_pids[q_idx]
        q_camid = q_camids[q_idx]
```

```
            # remove gallery samples that have the same pid and camid with query
            order = indices[q_idx]    # select one row
            remove = (g_pids[order] == q_pid) & (g_camids[order] == q_camid)
            keep = np.invert(remove)

            # compute cmc curve
            # binary vector, positions with value 1 are correct matches
            orig_cmc = matches[q_idx][keep]
            if not np.any(orig_cmc):
                # this condition is true when query identity does not appear in gallery
                continue

            cmc = orig_cmc.cumsum()
            cmc[cmc > 1] = 1

            all_cmc.append(cmc[:max_rank])
            num_valid_q += 1.

            # compute average precision
            #                                                                                                    reference:
https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)#Average_precision
            num_rel = orig_cmc.sum()
            tmp_cmc = orig_cmc.cumsum()
            tmp_cmc = [x / (i + 1.) for i, x in enumerate(tmp_cmc)]
            tmp_cmc = np.asarray(tmp_cmc) * orig_cmc
            AP = tmp_cmc.sum() / num_rel
            all_AP.append(AP)

        assert num_valid_q > 0, "Error: all query identities do not appear in gallery"

        all_cmc = np.asarray(all_cmc).astype(np.float32)
        all_cmc = all_cmc.sum(0) / num_valid_q
        mAP = np.mean(all_AP)

        return all_cmc, mAP

class R1_mAP():
    def __init__(self,cfg, num_query, max_rank=200, feat_norm=True,    reranking=False):
        super(R1_mAP, self).__init__()
        self.cfg = cfg
        self.num_query = num_query
        self.max_rank = max_rank
        self.feat_norm = feat_norm
```

```
        self.reranking = reranking

    def reset(self):
        self.feats = []
        self.pcb_feat = []
        self.img_name_path = []
        self.pcb_feat_split0 = []
        self.pcb_feat_split1 = []
        self.pcb_feat_split2 = []
        self.pcb_feat_split3 = []
        self.pcb_feat_split4 = []
        self.pcb_feat_split5 = []
        self.pcb_feat_split = {}

    # wenli:have local feature use next method
    def update_pcb(self, output):    # called once for each batch
        feat,pcb_feat, imgpath = output
        self.feats.append(feat)
        self.pcb_feat.append(pcb_feat)
        self.img_name_path.extend(imgpath)

    # wenli:have local feature use next method
    def update_pcb_split(self, output):    # called once for each batch
        feat, pcb_feat, imgpath = output
        self.feats.append(feat)
        self.pcb_feat_split0.append(pcb_feat[0])
        self.pcb_feat_split1.append(pcb_feat[1])
        self.pcb_feat_split2.append(pcb_feat[2])
        self.pcb_feat_split3.append(pcb_feat[3])
        self.pcb_feat_split4.append(pcb_feat[4])
        self.pcb_feat_split5.append(pcb_feat[5])
        # for i in range(6):
        #         print("pcb_feat[i].shape",pcb_feat[i].shape)
        #         self.pcb_feat_split[i].append(pcb_feat[i])
        # self.pcb_feat.append(pcb_feat)
        self.img_name_path.extend(imgpath)
    def update_split(self):

        self.pcb_feat_split[0] = self.pcb_feat_split0
        self.pcb_feat_split[1] = self.pcb_feat_split1
        self.pcb_feat_split[2] = self.pcb_feat_split2
        self.pcb_feat_split[3] = self.pcb_feat_split3
        self.pcb_feat_split[4] = self.pcb_feat_split4
        self.pcb_feat_split[5] = self.pcb_feat_split5
```

```
    # wenli:have local feature use next method
    def update(self, output):    # called once for each batch
        feat, imgpath = output
        self.feats.append(feat)
        self.img_name_path.extend(imgpath)
    def compute(self,reranking_parameter=[20,6,0.3]):    # called after each epoch
        feats = torch.cat(self.feats, dim=0)

        # if use pcb global feature ,use next method
        if self.cfg.MODEL.IF_USE_PCB:
            if self.cfg.TEST.PCB_GLOBAL_FEAT_ENSEMBLE:
                pcb_feats = torch.cat(self.pcb_feat,dim=0)
                pcb_qf = pcb_feats[:self.num_query]
                pcb_gf = pcb_feats[self.num_query:]



        if self.feat_norm:
            print("The test feature is normalized")
            feats = torch.nn.functional.normalize(feats, dim=1, p=2)    # along channel
        # query
        qf = feats[:self.num_query]
        q_path = self.img_name_path[:self.num_query]
        # gallery
        gf = feats[self.num_query:]
        g_path = self.img_name_path[self.num_query:]
        if self.reranking:
            print('=> Enter reranking')
            print('k1={}, k2={}, lambda_value={}'.format(reranking_parameter[0], reranking_parameter[1],
                                                         reranking_parameter[2]))
            distmat = re_ranking(qf, gf, k1=reranking_parameter[0], k2=reranking_parameter[1],
lambda_value=reranking_parameter[2])
            qf = qf.cpu()
            gf = gf.cpu()
            torch.cuda.empty_cache()
            if self.cfg.MODEL.IF_USE_PCB:
                if self.cfg.TEST.PCB_GLOBAL_FEAT_ENSEMBLE:
                    pcb_distmat = re_ranking(pcb_qf, pcb_gf, k1=reranking_parameter[0],
k2=reranking_parameter[1], lambda_value=reranking_parameter[2])
                    # del pcb_qf
                    # del pcb_gf
                    # torch.cuda.empty_cache()
                else:
                    if self.cfg.TEST.USE_PCB_MERGE_FEAT:
                        pcb_feats = torch.cat(self.pcb_feat, dim=0)
```

```
                pcb_qf = pcb_feats[:self.num_query]
                pcb_gf = pcb_feats[self.num_query:]
                pcb_distmat = re_ranking(pcb_qf, pcb_gf, k1=reranking_parameter[0],
k2=reranking_parameter[1],
                                         lambda_value=reranking_parameter[2])
                '''
                all_pcb_distmat = []
                pcb_feats = torch.cat(self.pcb_feat, dim=0)

                pcb_qf = pcb_feats[:self.num_query]
                pcb_gf = pcb_feats[self.num_query:]
                print(pcb_qf.shape)
                m = pcb_qf.shape[0]
                n = pcb_gf.shape[0]
                for j in range(m//300+1):
                    temp_pcb_qf = pcb_qf[j * 300:j * 300 + 300]
                    temp_pcb_dist = []
                    for i in range(n // 600 + 1):
                        temp_pcb_gf = pcb_gf[i * 600:i * 600 + 600]
                        pcb_distmat_i = re_ranking(temp_pcb_qf, temp_pcb_gf,
k1=reranking_parameter[0],
                                                   k2=reranking_parameter[1],
                                                   lambda_value=reranking_parameter[2])
                        temp_pcb_dist.append(pcb_distmat_i)
                    all_pcb_distmat.append(np.concatenate(temp_pcb_dist,axis=1))
                pcb_distmat = np.concatenate(all_pcb_distmat, axis=0)
                '''


                # for pcb_qf in pcb_qf:
                #     # print("part pcb_shape",pcb_qf.shape)
                #     # print("pcb_gf shape is",pcb_gf.shape)
                #     pcb_qf = torch.unsqueeze(pcb_qf,0)
                #
                #         pcb_distmat = re_ranking(pcb_qf, pcb_gf, k1=reranking_parameter[0],
k2=reranking_parameter[1],
                #                                  lambda_value=reranking_parameter[2])
                #     all_pcb_distmat.append(pcb_distmat)
                # pcb_distmat = np.concatenate(all_pcb_distmat,axis=0)
                # del pcb_qf
                # del pcb_gf
                # torch.cuda.empty_cache()
            else:
                pcb_distmat = np.zeros_like(distmat)
```

```
                    pcb_feats0 = torch.cat(self.pcb_feat_split0,dim=0)
                    pcb_qf0 = pcb_feats0[:self.num_query]
                    pcb_gf0 = pcb_feats0[self.num_query:]
                    pcb_distmat = re_ranking(pcb_qf0, pcb_gf0, k1=reranking_parameter[0],
k2=reranking_parameter[1],
                                             lambda_value=reranking_parameter[2])

                    pcb_feats1 = torch.cat(self.pcb_feat_split1, dim=0)
                    pcb_qf1 = pcb_feats1[:self.num_query]
                    pcb_gf1 = pcb_feats1[self.num_query:]
                    pcb_distmat = pcb_distmat + re_ranking(pcb_qf1, pcb_gf1,
k1=reranking_parameter[0], k2=reranking_parameter[1],
                                             lambda_value=reranking_parameter[2])

                    pcb_feats2 = torch.cat(self.pcb_feat_split2, dim=0)
                    pcb_qf2 = pcb_feats2[:self.num_query]
                    pcb_gf2 = pcb_feats2[self.num_query:]
                    pcb_distmat = pcb_distmat + re_ranking(pcb_qf2, pcb_gf2,
k1=reranking_parameter[0], k2=reranking_parameter[1],
                                             lambda_value=reranking_parameter[2])

                    pcb_feats3 = torch.cat(self.pcb_feat_split3, dim=0)
                    pcb_qf3 = pcb_feats0[:self.num_query]
                    pcb_gf3 = pcb_feats0[self.num_query:]
                    pcb_distmat = pcb_distmat + re_ranking(pcb_qf3, pcb_gf3,
k1=reranking_parameter[0], k2=reranking_parameter[1],
                                             lambda_value=reranking_parameter[2])

                    pcb_feats4 = torch.cat(self.pcb_feat_split4, dim=0)
                    pcb_qf4 = pcb_feats4[:self.num_query]
                    pcb_gf4 = pcb_feats4[self.num_query:]
                    pcb_distmat = pcb_distmat + re_ranking(pcb_qf4, pcb_gf4,
k1=reranking_parameter[0], k2=reranking_parameter[1],
                                             lambda_value=reranking_parameter[2])

                    pcb_feats5 = torch.cat(self.pcb_feat_split5, dim=0)
                    pcb_qf5 = pcb_feats5[:self.num_query]
                    pcb_gf5 = pcb_feats5[self.num_query:]
                    pcb_distmat = pcb_distmat + re_ranking(pcb_qf5, pcb_gf5,
k1=reranking_parameter[0], k2=reranking_parameter[1],
                                             lambda_value=reranking_parameter[2])
                    """
                    print("self.pcb_feat.shape",np.array(self.pcb_feat_split[0]).shape)
```

```
                    for pcb_feat in self.pcb_feat_split:
                        pcb_feats = torch.cat(pcb_feat, dim=0)
                        pcb_qf = pcb_feats[:self.num_query]
                        pcb_gf = pcb_feats[self.num_query:]
                        pcb_distmat = pcb_distmat + re_ranking(pcb_qf, pcb_gf,
k1=reranking_parameter[0], k2=reranking_parameter[1],
                                                               lambda_value=reranking_parameter[2])
                    """
                    # del pcb_qf
                    # del pcb_gf
                    # torch.cuda.empty_cache()
        else:
            print('=> Computing DistMat with cosine similarity')
            distmat = cosine_similarity(qf, gf)
            if self.cfg.MODEL.IF_USE_PCB:
                if self.cfg.TEST.PCB_GLOBAL_FEAT_ENSEMBLE:
                    pcb_distmat = cosine_similarity(pcb_qf, pcb_gf)
                else:
                    pcb_distmat = np.zeros_like(distmat)
                    for pcb_feat in self.pcb_feat:
                        pcb_feats = torch.cat(pcb_feat, dim=0)
                        pcb_qf = pcb_feats[:self.num_query]
                        pcb_gf = pcb_feats[self.num_query:]
                        pcb_distmat = pcb_distmat + cosine_similarity(pcb_qf, pcb_gf)
        if self.cfg.MODEL.IF_USE_PCB:
            if self.cfg.TEST.USE_LOCAL:
                distmat = distmat + pcb_distmat
        print(distmat,'distmat')
        num_q, num_g = distmat.shape
        indices = np.argsort(distmat, axis=1)
        data = dict()
        print(len(g_path), 'self.img_name_q')
        print(len(q_path),'self.img_name_g')
        for q_idx in range(num_q):
            order = indices[q_idx]  # select one row
            result_query = np.array(g_path)[order[:self.max_rank]]
            data[q_path[q_idx]] = [str(i) for i in result_query]
        return data, distmat, q_path, g_path
```

三、算法思路、亮点解读、建模算计与环境说明

在 NAIC 行人重识别复赛中，整体设计思路是通过表征学习方式以及全局度量学习方式得到精确的行人特征 ID。同时我们也注意到，本次大赛给出的行人图像并非是在自然光下的行人图片，因此在公开数据集上通用的方法例如 Alinged（罗浩）都在本赛题中有比较差的效果。因此，我们借鉴去年罗浩团队在类似比赛中的公开代码（DMT）做了一些针对本赛题的修改，并且我们通过实验发现，对本赛题图片进行不同分辨率的训练，最终得到的结果

差距很大，所以在复赛最终提交方面，我们将得到的不同分辨率的特征进行了 ensemble 融合，这使得最终达到了复赛 B 榜第 13 名的成绩。

其中，由于今年行人重识别比赛数据量极大，最终复赛时使用的数据集达到了 88w 张，同时在测试榜里面的数量也比去年比赛中多了很多，所以，为了保证在有限的计算设备中能够完成如此大规模的运算量，我们对数据在推理部分的 rerank 做了更进一步的分割处理，这也保证了我们的代码能够正常获得最终的推理结果。同时，我们在 DMT 代码的基础上重构了局部特征的训练分支，其中局部特征使用的是 PCB_RPP 特征提取方式。但是对于这部分特征，由于机器设备以及时间关系，最终没能够在复赛环节使用上。但是，通过本地实验证明，本赛题中将全局特征融合局部特征是行之有效的方法。

我们的模型训练是在 linux 系统下进行，配合调用两块显卡才能够达到和我们同样的分数。如下是我们建模算力表：

GPU 型号	显存大小	平均使用率
Nvidia GTX2080Ti	11019MiB	94%
Nvidia GTX2080Ti	11019MiB	87%

最终我们在复赛 B 榜使用的模型大小如下：

模型名称	模型主干	模型参数大小
Resnet101_ibn_b_128_40.pth	Resnet101_ibn	590MB
Resnet101_ibn_b_128_50.pth	Resnet101_ibn	590MB
Resnet101_ibn_b_192_40.pth	Resnet101_ibn	590MB
Resnet101_ibn_b_192_50.pth	Resnet101_ibn	590MB
Resnet101_ibn_b_240_40.pth	Resnet101_ibn	590MB
Resnet101_ibn_b_240_50.pth	Resnet101_ibn	590MB

最终我们模型在本地设备运行的情况下使用的串行训练总时间大约为 59 小时。

四、解题思路详情

4.1 获取数据集

我们在本赛题中使用的数据集仅 NAIC2019 初赛数据集、NAIC2019 复赛数据集、NAIV2020 初赛数据集以及 NAIC2020 复赛数据集。通过代码级审查后首先剔除了部分 2020 初赛数据集中和 NAIC2020 复赛数据集相同的部分，这一步的操作，极大的降低了初赛数据集对 2020 复赛数据集的干扰。同时，为了保证在模型微调以后，模型能够在加载中间权重后继续训练，我们对数据集每个 ID 做了排序操作，保证了模型微调后再训练的可靠性。

4.2 创建网络结构、损失计算及度量方式

虽然在复赛过程中我们曾使用局部特征进行过模型的训练，但是由于本地算力限制，最终我们只能通过全局特征提取行人特征用以训练，但是我们在提交的代码中依然保留了局部特征的网络结构，如果需要使用，只需要将 MODEL.IF_USE_PCB 设置为 True 即可。

在我们的全局特征中，使用 resnet101_ibn_b 作为主干网络得到维度 x 为[b,2048,8,4]的输出张量，其中，b 表示 batchsize 大小，在训练过程中，由于我们根据不同分辨率将训练集分别进行了训练，所以针对分辨率为(256,128)的行人图片，batchsize 设置的是 256，针对分辨率为(384,192)的行人图片，batchsize 设置的是 240，针对分辨率为(480,240)的行人图片，batchsize 设置的是 200。得到特征 x 以后对 x 进行全局池化并打平维度后得到特征 gf 维度为[b,2048]，再在 gf 基础上进行一层 BN 操作，得到特征 feat，然后在训练阶段使用 gf 作为 tripletloss 的特征进行计算损失，feat 作为 arcface 的输入进行计算 idloss。而在测试部分，

则只需要拿到 feat 特征来计算 query 和 gallery 两个数据集的最小距离即可。

五、项目运行环境和运行办法

本项目由于使用了一些 linux 中特有的库，所以不能在 windows 环境下训练运行。

如果需要运行本项目，需要在 ubuntu16.04 版本以上，python3.6 版本以上，pytorch 版本在 1.6 版本，cuda 为 10.1 版本下运行。

同时本项目已经给出了自动化训练脚本文件，如果想要得到和复赛相同的结果，只需要在控制台运行 bash run.sh 即可。训练时间大概 60 小时。如训练过程出现任何错误，请及时联系 m17693280903@163.com。

六、训练时说明

本项目使用的数据集为 NAIC2019 初赛、NAIC2019 复赛、NAIC2020 初赛以及 NAIC2020 复赛四个组合起来的数据集。其中由于 NAIC2020 初赛数据集和 NAIC2020 复赛数据集有部分重复，因此需要剔除初赛中重复的行人图片，总计剔除 33814 条记录，这一部分内容可在项目根目录 cs_2020equal_path.txt 文件中看到。

因此，为了方便快速训练，需要将新的初赛标签目录代替旧的标签目录（新的标签目录在项目根目录下的 new_train_list.txt 文件中）。同时，需要将四个数据集的标签文本重命名为 train_list.txt。

项目结构树形图如下所示

```
-model
-data
    --MyDataSet
        --train_2019_cs
            -train
            --train_list.txt
        --train_2019_fs
            --train
            --train_list.txt
        --train_2020_cs
            --images
            --train_list.txt
        --train_2020_fs
            --images
            --train_list.txt
--NAIC_ReID
```