import pandas as pd

import numpy as np

import scipy

import scipy.io

import cv2

from matplotlib import pyplot as plt

from scipy.ndimage import gaussian\_filter

import torch

import torch.nn as nn

from torch.utils.data import DataLoader, Dataset

from PIL import Image

from pytorch\_lightning import LightningModule, Trainer

from pytorch\_lightning import seed\_everything

import os

def show(im):

plt.figure(figsize=(10, 10))

plt.imshow(cv2.cvtColor(im, cv2.COLOR\_BGR2RGB))

im = cv2.imread('../input/shanghaitech/ShanghaiTech/part\_B/train\_data/images/IMG\_1.jpg', cv2.IMREAD\_COLOR)

show(im)

m = scipy.io.loadmat('../input/shanghaitech/ShanghaiTech/part\_B/train\_data/ground-truth/GT\_IMG\_1.mat')

ps = m['image\_info'][0][0][0][0][0]

for x, y **in** ps:

x = int(x)

y = int(y)

cv2.drawMarker(im, (x, y), (0, 255, 0))

show(im)

from sklearn.model\_selection import train\_test\_split

train = [p.path for p **in** os.scandir('../input/shanghaitech/ShanghaiTech/part\_B/train\_data/images/')]

valid\_full = [p.path for p **in** os.scandir('../input/shanghaitech/ShanghaiTech/part\_B/test\_data/images/')]

*## use a small subset for validation*

\_, valid = train\_test\_split(valid\_full, test\_size=64, random\_state=42)

len(train), len(valid)

from torchvision import transforms

import albumentations as A

from albumentations.pytorch import ToTensorV2

im\_size = 512

aug\_train = A.Compose([

A.RandomCrop(im\_size, im\_size),

A.HorizontalFlip(p=0.5),

A.RandomBrightnessContrast(),

A.Normalize((0.5), (0.5)),

], keypoint\_params=A.KeypointParams(format='xy', angle\_in\_degrees=False))

aug\_val = A.Compose([

A.Resize(768, 1024),

A.Normalize((0.5), (0.5)),

], keypoint\_params=A.KeypointParams(format='xy', angle\_in\_degrees=False))

class MyDataset(Dataset):

def \_\_init\_\_(self, files, aug):

self.files = files

self.aug = aug

def \_\_len\_\_(self):

return len(self.files)

def \_\_getitem\_\_(self, idx):

fn = self.files[idx]

im = cv2.imread(fn, cv2.IMREAD\_COLOR)

im = cv2.cvtColor(im, cv2.COLOR\_BGR2GRAY)

m = scipy.io.loadmat(fn.replace('images', 'ground-truth').replace('IMG', 'GT\_IMG').replace('.jpg', '.mat'))

ps = m['image\_info'][0][0][0][0][0]

rst = self.aug(image=im, keypoints=ps)

im = rst['image']

ps = rst['keypoints']

dm = np.zeros((im.shape[0], im.shape[1]), dtype=np.float32)

for x, y **in** ps:

x = int(x)

y = int(y)

dm[y, x] = 1

sigma = 4

dm = gaussian\_filter(dm, sigma=sigma, truncate=4\*sigma)

dm = cv2.resize(dm, (im.shape[1] // 4, im.shape[0] // 4), interpolation=cv2.INTER\_LINEAR)

dm \*= 16

im = torch.from\_numpy(im)

dm = torch.from\_numpy(dm)

return im, dm

ds = MyDataset(train, aug\_train)

im, dm = ds[0]

plt.imshow(dm)

dm.sum()

class Conv2d(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride=1, relu=True, same\_padding=False, bn=False):

super(Conv2d, self).\_\_init\_\_()

padding = int((kernel\_size - 1) / 2) if same\_padding else 0

self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, padding=padding)

self.bn = nn.BatchNorm2d(out\_channels, eps=0.001, momentum=0, affine=True) if bn else None

self.relu = nn.ReLU(inplace=True) if relu else None

def forward(self, x):

x = self.conv(x)

if self.bn **is** **not** None:

x = self.bn(x)

if self.relu **is** **not** None:

x = self.relu(x)

return x

class FRCNN(LightningModule):

def \_\_init\_\_(self, lr, batch\_size, max\_steps, bn=False):

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

self.lr = lr

self.save\_hyperparameters()

self.use = 0

self.branch1 = nn.Sequential(Conv2d( 1, 16, 9, same\_padding=True, bn=bn),

nn.MaxPool2d(2),

Conv2d(16, 32, 7, same\_padding=True, bn=bn),

nn.MaxPool2d(2),

Conv2d(32, 16, 7, same\_padding=True, bn=bn),

self.branch2 = nn.Sequential(Conv2d( 1, 20, 7, same\_padding=True, bn=bn),

nn.MaxPool2d(2),

Conv2d(20, 40, 5, same\_padding=True, bn=bn),

nn.MaxPool2d(2),

Conv2d(40, 20, 5, same\_padding=True, bn=bn),

Conv2d(20, 10, 5, same\_padding=True, bn=bn))

self.branch3 = nn.Sequential(Conv2d( 1, 24, 5, same\_padding=True, bn=bn),

nn.MaxPool2d(2),

Conv2d(24, 48, 3, same\_padding=True, bn=bn),

nn.MaxPool2d(2),

Conv2d(48, 24, 3, same\_padding=True, bn=bn),

Conv2d(24, 12, 3, same\_padding=True, bn=bn))

self.fuse = nn.Sequential(Conv2d( 30, 1, 1, same\_padding=True, bn=bn))

self.out1 = nn.Sequential(Conv2d( 8, 1, 1, same\_padding=True, bn=bn))

self.out2 = nn.Sequential(Conv2d( 10, 1, 1, same\_padding=True, bn=bn))

self.out3 = nn.Sequential(Conv2d( 12, 1, 1, same\_padding=True, bn=bn))

self.crit = nn.MSELoss()

def forward(self, im\_data):

im\_data = im\_data.unsqueeze(1)

x1 = self.branch1(im\_data)

x2 = self.branch2(im\_data)

x3 = self.branch3(im\_data)

if self.use == 0:

x = torch.cat((x1,x2,x3),1)

x = self.fuse(x)

elif self.use == 1:

x = self.out1(x1)

elif self.use == 2:

x = self.out2(x2)

elif self.use == 3:

x = self.out3(x3)

return x.squeeze(1)

def training\_step(self, batch, batch\_idx):

self.train()

x, y = batch

pred = self(x)

loss = self.crit(pred, y)

pred\_sum = torch.round(pred.sum(dim=(1,2))).int()

gt\_sum = torch.round(y.sum(dim=(1,2))).int()

acc = (pred\_sum == gt\_sum).float().mean()

mae = torch.abs(pred\_sum - gt\_sum).float().mean()

self.log('train\_loss', loss)

self.log('train\_acc', acc)

self.log('train\_mae', mae)

return loss

def validation\_step(self, batch, batch\_idx):

with torch.no\_grad():

self.eval()

x, y = batch

pred = self(x)

loss = self.crit(pred, y)

acc = (pred\_sum == gt\_sum).float().mean()

mae = torch.abs(pred\_sum - gt\_sum).float().mean()

self.log('val\_loss', loss)

self.log('val\_acc', acc)

self.log('val\_mae', mae)

def configure\_optimizers(self):

optimizer = torch.optim.AdamW(self.parameters(), lr=self.lr, weight\_decay=1e-4)

scheduler = {

'scheduler': torch.optim.lr\_scheduler.OneCycleLR(optimizer, max\_lr=self.lr, total\_steps=self.hparams.max\_steps, pct\_start=0.1, cycle\_momentum=False),

'interval': 'step',

'frequency': 1

}

return [optimizer], [scheduler]

def weights\_normal\_init(model, dev=0.01):

if isinstance(model, list):

for m **in** model:

weights\_normal\_init(m, dev)

else:

for n, m **in** model.named\_modules():

if isinstance(m, nn.Conv2d):

m.weight.data.normal\_(0.0, dev)

if m.bias **is** **not** None:

m.bias.data.fill\_(0.0)

elif isinstance(m, nn.Linear):

m.weight.data.normal\_(0.0, dev)

batch\_size = 32

epochs = 300

max\_steps = epochs \* len(train) // batch\_size

In [18]:

linkcode

train\_loader = DataLoader(MyDataset(train, aug\_train), batch\_size=batch\_size, shuffle=True, drop\_last=True, pin\_memory=True, num\_workers=4)

val\_loader = DataLoader(MyDataset(valid, aug\_val), batch\_size=batch\_size, shuffle=False, drop\_last=False, pin\_memory=True, num\_workers=4)

from pytorch\_lightning.callbacks import ModelCheckpoint, LearningRateMonitor

seed\_everything(42)

checkpoint\_cb = ModelCheckpoint(

save\_top\_k=1,

save\_last=True,

verbose=True,

monitor='val\_mae',

mode='min',

prefix=''

)

trainer = Trainer(gpus=1, max\_steps=max\_steps, precision=16, benchmark=True, callbacks=[checkpoint\_cb, LearningRateMonitor()])

lr = 3e-4

model = FRCNN(lr, batch\_size, max\_steps)

weights\_normal\_init(model, dev=0.01)

model.use = 0

trainer.fit(model, train\_dataloader=train\_loader, val\_dataloaders=val\_loader)

import argparse

import cv2

import torch

import torchvision.transforms as T

import numpy as np

from torchvision.models.detection import fasterrcnn\_resnet50\_fpn

from torchvision.models.detection.faster\_rcnn import FastRCNNPredictor

def resize\_image(img, img\_size):

"""Resize image to the specified size."""

return cv2.resize(img, (img\_size, img\_size))

def detect\_single\_image(img\_path, weights, img\_size=640, conf\_thres=0.1, iou\_thres=0.45):

# Initialize

set\_logging()

device = select\_device('cpu') # Force CPU

half = False # half precision not supported on CPU

# Load model

model = attempt\_load(weights, map\_location=device) # load FP32 model

stride = int(model.stride.max()) # model stride

imgsz = check\_img\_size(img\_size, s=stride) # check img\_size

if half:

model.half() # to FP16

# Read image

img0 = cv2.imread(img\_path) # BGR

img = resize\_image(img0, img\_size) # Resize image

# Normalize RGB

img = img[:, :, ::-1].transpose(2, 0, 1) # BGR to RGB, to 3x416x416

img = np.ascontiguousarray(img)

img = torch.from\_numpy(img).to(device)

img = img.half() if half else img.float() # uint8 to fp16/32

img /= 255.0 # 0 - 255 to 0.0 - 1.0

if img.ndimension() == 3:

img = img.unsqueeze(0)

# Inference

t1 = time\_synchronized()

pred = model(img, augment=False)[0]

t2 = time\_synchronized()

# Apply NMS

pred = non\_max\_suppression(pred, conf\_thres, iou\_thres, classes=None, agnostic=False)

t3 = time\_synchronized()

# Count of humans detected

human\_count = 0

# Process detections

for i, det in enumerate(pred): # detections per image

gn = torch.tensor(img0.shape)[[1, 0, 1, 0]] # normalization gain whwh

if len(det):

# Rescale boxes from img\_size to im0 size

det[:, :4] = scale\_coords(img.shape[2:], det[:, :4], img0.shape).round()

# Class labels for YOLOv7

class\_labels = model.module.names if hasattr(model, 'module') else model.names

# Check if human is detected

for \*xyxy, conf, cls in reversed(det):

if class\_labels[int(cls)] == 'person':

human\_count += 1

return human\_count

def draw\_heatmap(img\_path, conf\_thres=0.5):

# Load Faster R-CNN model

model = fasterrcnn\_resnet50\_fpn(pretrained=True)

# Replace the classifier with a new one, which has the number of classes equal to 2 (person and background)

num\_classes = 2

in\_features = model.roi\_heads.box\_predictor.cls\_score.in\_features

model.roi\_heads.box\_predictor = FastRCNNPredictor(in\_features, num\_classes)

# Load image

img = cv2.imread(img\_path)

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

img\_tensor = T.ToTensor()(img)

img\_tensor = img\_tensor.unsqueeze(0)

# Forward pass

model.eval()

with torch.no\_grad():

prediction = model(img\_tensor)

# Extract bounding boxes and scores

boxes = prediction[0]['boxes'].cpu().numpy()

scores = prediction[0]['scores'].cpu().numpy()

# Filter detections based on confidence threshold

boxes = boxes[scores >= conf\_thres]

# Initialize heatmap

heatmap = np.zeros((img.shape[0], img.shape[1]))

# Draw detections on heatmap

for box in boxes:

x1, y1, x2, y2 = box.astype(int)

heatmap[y1:y2, x1:x2] += 1

return heatmap

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

parser = argparse.ArgumentParser()

parser.add\_argument('--img', type=str, help='path to input image')

parser.add\_argument('--conf-thres', type=float, default=0.5, help='confidence threshold')

opt = parser.parse\_args()

human\_count = detect\_single\_image(opt.img, opt.weights, opt.img\_size, opt.conf\_thres, opt.iou\_thres)

print("Number of humans detected:", human\_count)

# Draw heatmap

heatmap = draw\_heatmap(opt.img, opt.conf\_thres)

# Normalize heatmap

heatmap = cv2.normalize(heatmap, None, alpha=0, beta=255, norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

# Apply colormap

heatmap\_colormap = cv2.applyColorMap(heatmap, cv2.COLORMAP\_HOT)

# Resize heatmap to half the screen

heatmap\_resized = cv2.resize(heatmap\_colormap, (int(heatmap\_colormap.shape[1] / 2), int(heatmap\_colormap.shape[0] / 2)))

# Display heatmap in a window taking half the screen

cv2.namedWindow('Heatmap', cv2.WINDOW\_NORMAL)

cv2.resizeWindow('Heatmap', heatmap\_resized.shape[1], heatmap\_resized.shape[0])

cv2.imshow('Heatmap', heatmap\_resized)

cv2.waitKey(0)

cv2.destroyAllWindows()