**程序报告**

学号：2213410 姓名：徐俊智

1. **问题重述**

（简单描述对问题的理解，从问题中抓住主干，必填）

====================================================================

本次实验的目标是学习经典模型MTCNN和MobileNet的结构以及训练方法，构建一个深度学习模型用于检测图像中的人物是否佩戴口罩。

1. **设计思想**

（所采用的方法，有无对方法加以改进，该方法有哪些优化方向（参数调整，框架调整，或者指出方法的局限性和常见问题），伪代码，理论结果验证等… **思考题，非必填**）

====================================================================

设计思想分两步：

1.使用MTCNN进行人脸检测，确定图像中人脸的位置和关键点。

2.对检测到的人脸区域应用MobileNet模型，判断是否佩戴口罩。

在实现过程中采用了以下方法：

1.对MTCNN的三个阶段（PNet、RNet、ONet）进行了权重加载，使用了预训练的模型来进行人脸检测。

2.对MobileNet模型进行了微调，将其最后一层的类别数调整为2（未佩戴口罩和佩戴口罩）。

3.使用了数据增强技术，如随机水平和垂直翻转，来增加模型的泛化能力。

4.应用了学习率衰减策略，当验证集上的性能在连续几个epoch内没有提升时，降低学习率。

优化方向：

1.调整MobileNet每次读取图片的数量batch\_size由原来的32修改为256。

2.调整优化器参数lr由原来的1e-3修改为0.0037

3.调整门限函数权重thresholds由原来的[0.6，0.7，0.8]修改为[0.55，0.65，0.8]

1. **代码内容**

（能体现解题思路的主要代码，有多个文件或模块可用多个"===="隔开，必填）

====================================================================

**detector.py**

import torch

import math

import numpy as np

from PIL import Image, ImageDraw

from torch.autograd import Variable

from .get\_nets import PNet, RNet, ONet

from .utils import (

try\_gpu,

nms,

calibrate\_box,

convert\_to\_square,

correct\_bboxes,

get\_image\_boxes,

generate\_bboxes,

preprocess,

)

class FaceDetector:

def \_\_init\_\_(self, device=try\_gpu()):

self.device = device

# LOAD MODELS

self.pnet = PNet().to(device)

self.rnet = RNet().to(device)

self.onet = ONet().to(device)

self.onet.eval()

def detect(

self,

image,

min\_face\_size=20.0,

thresholds=[0.55, 0.65, 0.8],

nms\_thresholds=[0.7, 0.7, 0.7],

):

"""

Arguments:

image: an instance of PIL.Image.

min\_face\_size: a float number.

thresholds: a list of length 3.

nms\_thresholds: a list of length 3.

Returns:

two float numpy arrays of shapes [n\_boxes, 5] and [n\_boxes, 10],

bounding boxes and facial landmarks.

"""

# BUILD AN IMAGE PYRAMID

width, height = image.size

min\_length = min(height, width)

min\_detection\_size = 12

factor = 0.707 # sqrt(0.5)

# scales for scaling the image

scales = []

# scales the image so that

# minimum size that we can detect equals to

# minimum face size that we want to detect

m = min\_detection\_size / min\_face\_size

min\_length \*= m

factor\_count = 0

while min\_length > min\_detection\_size:

scales.append(m \* factor \*\* factor\_count)

min\_length \*= factor

factor\_count += 1

# STAGE 1

# it will be returned

bounding\_boxes = []

# run P-Net on different scales

for s in scales:

boxes = self.\_\_run\_first\_stage(image, scale=s, threshold=thresholds[0])

bounding\_boxes.append(boxes)

# collect boxes (and offsets, and scores) from different scales

bounding\_boxes = [i for i in bounding\_boxes if i is not None]

bounding\_boxes = np.vstack(bounding\_boxes)

keep = nms(bounding\_boxes[:, 0:5], nms\_thresholds[0])

bounding\_boxes = bounding\_boxes[keep]

# use offsets predicted by pnet to transform bounding boxes

bounding\_boxes = calibrate\_box(bounding\_boxes[:, 0:5], bounding\_boxes[:, 5:])

# shape [n\_boxes, 5]

bounding\_boxes = convert\_to\_square(bounding\_boxes)

bounding\_boxes[:, 0:4] = np.round(bounding\_boxes[:, 0:4])

# STAGE 2

img\_boxes = get\_image\_boxes(bounding\_boxes, image, size=24)

with torch.no\_grad():

img\_boxes = Variable(torch.FloatTensor(img\_boxes).to(self.device))

output = self.rnet(img\_boxes)

offsets = output[0].cpu().data.numpy() # shape [n\_boxes, 4]

probs = output[1].cpu().data.numpy() # shape [n\_boxes, 2]

keep = np.where(probs[:, 1] > thresholds[1])[0]

bounding\_boxes = bounding\_boxes[keep]

bounding\_boxes[:, 4] = probs[keep, 1].reshape((-1,))

offsets = offsets[keep]

keep = nms(bounding\_boxes, nms\_thresholds[1])

bounding\_boxes = bounding\_boxes[keep]

bounding\_boxes = calibrate\_box(bounding\_boxes, offsets[keep])

bounding\_boxes = convert\_to\_square(bounding\_boxes)

bounding\_boxes[:, 0:4] = np.round(bounding\_boxes[:, 0:4])

# STAGE 3

img\_boxes = get\_image\_boxes(bounding\_boxes, image, size=48)

if len(img\_boxes) == 0:

return [], []

with torch.no\_grad():

img\_boxes = Variable(torch.FloatTensor(img\_boxes).to(self.device))

output = self.onet(img\_boxes)

landmarks = output[0].cpu().data.numpy() # shape [n\_boxes, 10]

offsets = output[1].cpu().data.numpy() # shape [n\_boxes, 4]

probs = output[2].cpu().data.numpy() # shape [n\_boxes, 2]

keep = np.where(probs[:, 1] > thresholds[2])[0]

bounding\_boxes = bounding\_boxes[keep]

bounding\_boxes[:, 4] = probs[keep, 1].reshape((-1,))

offsets = offsets[keep]

landmarks = landmarks[keep]

# compute landmark points

width = bounding\_boxes[:, 2] - bounding\_boxes[:, 0] + 1.0

height = bounding\_boxes[:, 3] - bounding\_boxes[:, 1] + 1.0

xmin, ymin = bounding\_boxes[:, 0], bounding\_boxes[:, 1]

landmarks[:, 0:5] = (

np.expand\_dims(xmin, 1) + np.expand\_dims(width, 1) \* landmarks[:, 0:5]

)

landmarks[:, 5:10] = (

np.expand\_dims(ymin, 1) + np.expand\_dims(height, 1) \* landmarks[:, 5:10]

)

bounding\_boxes = calibrate\_box(bounding\_boxes, offsets)

keep = nms(bounding\_boxes, nms\_thresholds[2], mode="min")

bounding\_boxes = bounding\_boxes[keep]

landmarks = landmarks[keep]

return bounding\_boxes, landmarks

def draw\_bboxes(self, image):

"""Draw bounding boxes and facial landmarks.

Arguments:

image: an instance of PIL.Image.

Returns:

an instance of PIL.Image.

"""

bounding\_boxes, facial\_landmarks = self.detect(image)

img\_copy = image.copy()

draw = ImageDraw.Draw(img\_copy)

for b in bounding\_boxes:

draw.rectangle([(b[0], b[1]), (b[2], b[3])], outline="white")

for p in facial\_landmarks:

for i in range(5):

draw.ellipse(

[(p[i] - 1.0, p[i + 5] - 1.0), (p[i] + 1.0, p[i + 5] + 1.0)],

outline="blue",

)

return img\_copy

def crop\_faces(self, image, size=112):

"""Crop all face images.

Arguments:

image: an instance of PIL.Image.

size: the side length of output images.

Returns:

a list of PIL.Image instances

"""

bounding\_boxes, \_ = self.detect(image)

img\_list = []

# convert bboxes to square

square\_bboxes = convert\_to\_square(bounding\_boxes)

for b in square\_bboxes:

face\_img = image.crop((b[0], b[1], b[2], b[3]))

face\_img = face\_img.resize((size, size), Image.BILINEAR)

img\_list.append(face\_img)

return img\_list

def \_\_run\_first\_stage(self, image, scale, threshold):

"""Run P-Net, generate bounding boxes, and do NMS.

Arguments:

image: an instance of PIL.Image.

scale: a float number,

scale width and height of the image by this number.

threshold: a float number,

threshold on the probability of a face when generating

bounding boxes from predictions of the net.

Returns:

a float numpy array of shape [n\_boxes, 9],

bounding boxes with scores and offsets (4 + 1 + 4).

"""

# scale the image and convert it to a float array

width, height = image.size

sw, sh = math.ceil(width \* scale), math.ceil(height \* scale)

img = image.resize((sw, sh), Image.BILINEAR)

img = np.asarray(img, "float32")

with torch.no\_grad():

img = Variable(torch.FloatTensor(preprocess(img)).to(self.device))

output = self.pnet(img)

probs = output[1].cpu().data.numpy()[0, 1, :, :]

offsets = output[0].cpu().data.numpy()

# probs: probability of a face at each sliding window

# offsets: transformations to true bounding boxes

boxes = generate\_bboxes(probs, offsets, scale, threshold)

if len(boxes) == 0:

return None

keep = nms(boxes[:, 0:5], overlap\_threshold=0.5)

return boxes[keep]

====================================================================

**torch\_main.py**

import warnings

# 忽视警告

warnings.filterwarnings('ignore')

import cv2

from PIL import Image

import numpy as np

import copy

import matplotlib.pyplot as plt

from tqdm.auto import tqdm

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision.datasets import ImageFolder

import torchvision.transforms as T

from torch.utils.data import DataLoader

from torch\_py.Utils import plot\_image

from torch\_py.MTCNN.detector import FaceDetector

from torch\_py.MobileNetV1 import MobileNetV1

from torch\_py.FaceRec import Recognition

# 数据集路径

data\_path = "./datasets/5f680a696ec9b83bb0037081-momodel/data/"

def letterbox\_image(image, size):

"""

调整图片尺寸

:param image: 用于训练的图片

:param size: 需要调整到网络输入的图片尺寸

:return: 返回经过调整的图片

"""

new\_image = cv2.resize(image, size, interpolation=cv2.INTER\_AREA)

return new\_image

# 使用 PIL.Image 读取图片

read\_img = Image.open("test1.jpg")

read\_img = np.array(read\_img)

print("调整前图片的尺寸:", read\_img.shape)

read\_img = letterbox\_image(image=read\_img, size=(50, 50))

read\_img = np.array(read\_img)

print("调整前图片的尺寸:", read\_img.shape)

def processing\_data(data\_path, height=224, width=224, batch\_size=32,

test\_split=0.1):

"""

数据处理部分

:param data\_path: 数据路径

:param height:高度

:param width: 宽度

:param batch\_size: 每次读取图片的数量

:param test\_split: 测试集划分比例

:return:

"""

transforms = T.Compose([

T.Resize((height, width)),

T.RandomHorizontalFlip(0.1), # 进行随机水平翻转

T.RandomVerticalFlip(0.1), # 进行随机竖直翻转

T.ToTensor(), # 转化为张量

T.Normalize([0], [1]), # 归一化

])

dataset = ImageFolder(data\_path, transform=transforms)

# 划分数据集

train\_size = int((1-test\_split)\*len(dataset))

test\_size = len(dataset) - train\_size

train\_dataset, test\_dataset = torch.utils.data.random\_split(dataset, [train\_size, test\_size])

# 创建一个 DataLoader 对象

train\_data\_loader = DataLoader(train\_dataset, batch\_size=batch\_size,shuffle=True)

valid\_data\_loader = DataLoader(test\_dataset, batch\_size=batch\_size,shuffle=True)

return train\_data\_loader, valid\_data\_loader

data\_path = './datasets/5f680a696ec9b83bb0037081-momodel/data/image'

train\_data\_loader, valid\_data\_loader = processing\_data(data\_path=data\_path, height=160, width=160, batch\_size=32)

def show\_tensor\_img(img\_tensor):

img = img\_tensor[0].data.numpy()

img = np.swapaxes(img, 0, 2)

img = np.swapaxes(img, 0, 1)

img = np.array(img)

plot\_image(img)

for index, (x, labels) in enumerate(train\_data\_loader):

print(index, "\nfeature:",x[0], "\nlabels:",labels)

show\_tensor\_img(x)

break

pnet\_path = "./torch\_py/MTCNN/weights/pnet.npy"

rnet\_path = "./torch\_py/MTCNN/weights/rnet.npy"

onet\_path = "./torch\_py/MTCNN/weights/onet.npy"

#========================================================================

# 加载 MobileNet 的预训练模型权

device = torch.device("cuda:0") if torch.cuda.is\_available() else torch.device("cpu")

train\_data\_loader, valid\_data\_loader = processing\_data(data\_path=data\_path, height=160, width=160, batch\_size=256)

modify\_x, modify\_y = torch.ones((32, 3, 160, 160)), torch.ones((32))

epochs = 60

model = MobileNetV1(classes=2).to(device)

optimizer = optim.Adam(model.parameters(), lr=0.0037) # 优化器

print('加载完成...')

# 学习率下降的方式，acc三次不下降就下降学习率继续训练，衰减学习率

scheduler = optim.lr\_scheduler.ReduceLROnPlateau(optimizer,

'max',

factor=0.5,

patience=2)

# 损失函数

criterion = nn.CrossEntropyLoss()

#========================================================================

best\_loss = 1e9

best\_model\_weights = copy.deepcopy(model.state\_dict())

loss\_list = [] # 存储损失函数值

for epoch in range(epochs):

model.train()

for batch\_idx, (x, y) in tqdm(enumerate(train\_data\_loader, 1)):

x = x.to(device)

y = y.to(device)

pred\_y = model(x)

# print(pred\_y.shape)

# print(y.shape)

loss = criterion(pred\_y, y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if loss < best\_loss:

best\_model\_weights = copy.deepcopy(model.state\_dict())

best\_loss = loss

loss\_list.append(loss)

print('step:' + str(epoch + 1) + '/' + str(epochs) + ' || Total Loss: %.4f' % (loss))

torch.save(model.state\_dict(), './results/temp.pth')

print('Finish Training.')

plt.plot(loss\_list,label = "loss")

plt.legend()

plt.show()

====================================================================

**main.py**

from torch\_py.Utils import plot\_image

from torch\_py.MTCNN.detector import FaceDetector

from torch\_py.MobileNetV1 import MobileNetV1

from torch\_py.FaceRec import Recognition

from torch\_py.FaceRec import Recognition

from PIL import Image

import cv2

# -------------------------- 请加载您最满意的模型 ---------------------------

# 加载模型(请加载你认为的最佳模型)

# 加载模型,加载请注意 model\_path 是相对路径, 与当前文件同级。

# 如果你的模型是在 results 文件夹下的 dnn.h5 模型，则 model\_path = 'results/temp.pth'

model\_path = 'results/temp.pth'

# ---------------------------------------------------------------------------

def predict(img):

"""

加载模型和模型预测

:param img: cv2.imread 图像

:return: 预测的图片中的总人数、其中佩戴口罩的人数

"""

# -------------------------- 实现模型预测部分的代码 ---------------------------

# 将 cv2.imread 图像转化为 PIL.Image 图像，用来兼容测试输入的 cv2 读取的图像（勿删！！！）

# cv2.imread 读取图像的类型是 numpy.ndarray

# PIL.Image.open 读取图像的类型是 PIL.JpegImagePlugin.JpegImageFile

if isinstance(img, np.ndarray):

# 转化为 PIL.JpegImagePlugin.JpegImageFile 类型

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

recognize = Recognition(model\_path)

img, all\_num, mask\_num = recognize.mask\_recognize(img)

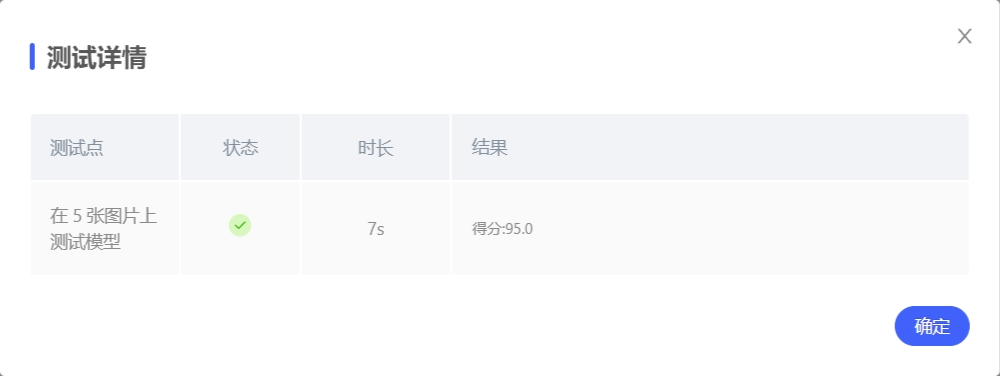
# -------------------------------------------------------------------------

return all\_num,mask\_num

1. **实验结果**

（实验结果，必填）

====================================================================



1. **总结**

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

====================================================================

在本次实验中，我主要使用了torch方法进行训练，最后成功构建了一个能够检测图像中人物是否佩戴口罩的模型。

可能改进的方向：

1.使用更大的数据集进行训练，以增强模型的泛化能力。

2.调整模型的超参数，如学习率、批量大小和训练轮数。

3.探索不同的模型架构和特征提取器，如尝试不同的预训练模型或调整MobileNet的深度和宽度。

4.实施更复杂的数据增强策略，以进一步提高模型对不同情况的适应性。

实现过程中的困难：模型的准确率提升缓慢和过拟合问题。我通过调整优化器和学习率来解决这些问题。

提升性能：模型的超参数和框架搜索在一定程度上是合理的，但仍有改进空间。通过更细致的调整和更深入的分析，我们可以进一步提升模型的性能。