目标检测：口罩佩戴检测

1.实验介绍

1.1 实验背景

今年一场席卷全球的新型冠状病毒给人们带来了沉重的生命财产的损失。有效防御这种传染病毒的方法就是积极佩戴口罩。我国对此也采取了严肃的措施，在公共场合要求人们必须佩戴口罩。在本次实验中，我们要建立一个目标检测的模型，可以识别图中的人是否佩戴了口罩。

1.2 实验要求

1）建立深度学习模型，检测出图中的人是否佩戴了口罩，并将其尽可能调整到最佳状态。

2）学习经典的模型 MTCNN 和 MobileNet 的结构。

3）学习训练时的方法。

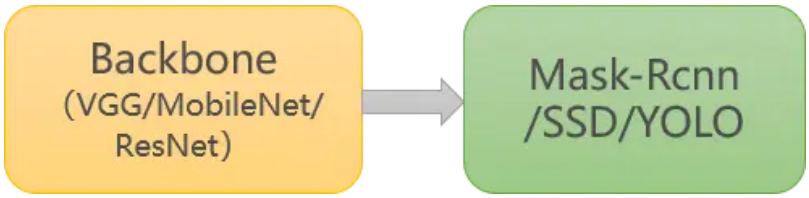
1.3 实验环境

可以使用基于 Python 的 OpenCV 、PIL 库进行图像相关处理，使用 Numpy 库进行相关数值运算，使用 Pytorch 等深度学习框架训练模型等。

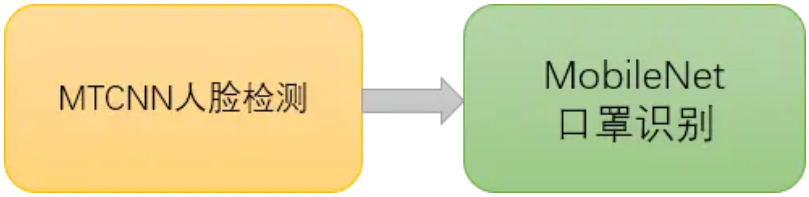
2. 实验思路与实验过程

针对目标检测的任务，可以分为两个部分：目标识别和位置检测。通常情况下，特征提取需要由特有的特征提取神经网络来完成，如 VGG、MobileNet、ResNet 等，这些特征提取网络往往被称为 Backbone 。而在 BackBone 后面接全连接层(FC)就可以执行分类任务。但 FC 对目标的位置识别乏力。经过算法的发展，当前主要以特定的功能网络来代替 FC 的作用，如 Mask-Rcnn、SSD、YOLO 等。我们选择充分使用已有的人脸检测的模型，再训练一个识别口罩的模型，从而提高训练的开支、增强模型的准确率。

常规目标检测：



本次案例：



2.1 数据集介绍与预处理

数据信息存放在 `/datasets/5f680a696ec9b83bb0037081-momodel/data` 文件夹下。本次实验中我们选择使用pytroch进行训练。

我们尝试读取数据集中戴口罩的图片及其名称。以下是训练集中的正样本：

```
mask_num = 4
fig = plt.figure(figsize=(15, 15))
for i in range(mask_num):
    sub_img = cv2.imread(data_path + "/image/mask/mask_" + str(i + 101) +
".jpg")
    sub_img = cv2.cvtColor(sub_img, cv2.COLOR_RGB2BGR)
    ax = fig.add_subplot(4, 4, (i + 1))
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title("mask_" + str(i + 1))
    ax.imshow(sub_img)
```

以下是训练集中的负样本：

```
nomask_num = 4
fig1 = plt.figure(figsize=(15, 15))
for i in range(nomask_num):
    sub_img = cv2.imread(data_path + "/image/nomask/nomask_" + str(i + 130) +
".jpg")
    sub_img = cv2.cvtColor(sub_img, cv2.COLOR_RGB2BGR)
    ax = fig1.add_subplot(4, 4, (i + 1))
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title("nomask_" + str(i + 1))
    ax.imshow(sub_img)
```

经过测试样例代码，正样本为戴口罩，负样本反之。同时观察数据集信息，正样本标签为0，负样本为1。

2.1.1调整图片尺寸

```
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
```

查看图片尺寸调整前后的对比，经过指导中给出的样例程序测试如下：

```
调整前图片的尺寸: (257, 495, 3)
调整前图片的尺寸: (50, 50, 3)
```

通过设置调用函数时的尺寸，可统一图片大小。

2.1.2 制作所需的批量数据集

Pytorch 读取数据虽然特别灵活，但是还是具有特定的流程的，它的操作顺序为：

- 创建一个 `Dataset` 对象，该对象如果现有的 `Dataset` 不能够满足需求，我们也可以自定义 `Dataset`，通过继承 `torch.utils.data.Dataset`。在继承的时候，需要 `override` 三个方法。
  - `__init__`： 用来初始化数据集
  - `__getitem__`：给定索引值，返回该索引值对应的数据；它是python built-in方法，其主要作用是能让该类可以像list一样通过索引值对数据进行访问
  - `__len__`：用于len(Dataset)时能够返回大小
- 创建一个 `DataLoader` 对象
- 不停的 循环 这个 `DataLoader` 对象

第一步：创建一个 `Dataset` 对象

torchvision.datasets.ImageFolder 是一个通用的数据加载器，常见的用法如下：

> dataset=torchvision.datasets.ImageFolder(root, transform=None, target_transform=None, loader=<function default_loader>, is_valid_file=None)

- **参数详解**：
  - root：图片存储的根目录，即各类别文件夹所在目录的上一级目录。
  - transform：对图片进行预处理的操作（函数），原始图片作为输入，返回一个转换后的图片。
  - target_transform：对图片类别进行预处理的操作，输入为 target，输出对其的转换。如果不传该参数，即对 target 不做任何转换，返回的顺序索引 0,1, 2...
  - loader：表示数据集加载方式，通常默认加载方式即可。
  - is_valid_file：获取图像文件的路径并检查该文件是否为有效文件的函数(用于检查损坏文件)
- 返回的 dataset 都有以下三种属性：
  - dataset.classes：用一个 list 保存类别名称
  - dataset.class_to_idx：类别对应的索引，与不做任何转换返回的 target 对应
  - dataset.imgs：保存(img-path, class) tuple 的列表

第二步：创建一个 `DataLoader` 对象

`DataLoader` 是 `torch` 用来包装我们的数据的工具，所以要将(numpy array 或其他) 数据形式装换成 Tensor, 然后再放进这个包装器中。 使用 `DataLoader` 帮助我们对数据进行有效地迭代处理。

> torch.utils.data.DataLoader(dataset,batch_size=1,shuffle=False,
> sampler=None,
> batch_sampler=None,
> num_workers=0,
> collate_fn=<function default_collate>,
> pin_memory=False,
> drop_last=False,
> timeout=0,
> worker_init_fn=None)

- **常用参数解释**：
  - dataset (Dataset): 是一个 DataSet 对象，表示需要加载的数据集
  - batch_size (int, optional): 每一个 batch 加载多少组样本，即指定 batch_size ， 默认是 1

- shuffle (bool, optional): 布尔值 True 或者是 False ， 表示每一个 epoch 之后是否对样本进行随机打乱，默认是 False
- sampler (Sampler, optional): 自定义从数据集中抽取样本的策略，如果指定这个参数，那么 shuffle 必须为 False
- batch_sampler (Sampler, optional): 与 sampler 类似，但是一次只返回一个 batch 的 indices（索引），需要注意的是，一旦指定了这个参数，那么 batch_size,shuffle,sampler,drop_last 就不能再制定了（互斥）
- num_workers (int, optional): 这个参数决定了有几个进程来处理 data loading 。0 意味着所有的数据都会被 load 进主进程，默认为0
- collate_fn (callable, optional): 将一个 list 的 sample 组成一个 mini-batch 的函数（这个还不是很懂）
- pin_memory (bool, optional): 如果设置为True，那么 data loader 将会在返回它们之前，将 tensors 拷贝到 CUDA 中的固定内存（CUDA pinned memory）中
- drop_last (bool, optional): 如果设置为 True：这个是对最后的未完成的 batch 来说的，比如 batch_size 设置为 64，而一个 epoch只有 100 个样本，那么训练的时候后面的 36 个就被扔掉了，如果为 False（默认），那么会继续正常执行，只是最后的 batch_size 会小一点。
- timeout (numeric, optional): 如果是正数，表明等待从 worker 进程中收集一个 batch 等待的时间，若超出设定的时间还没有收集到，那就不收集这个内容。这个 numeric 应总是大于等于0，默认为0。

我们采用以上 2 步进行数据处理，代码展示如下:

```
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
```

使用给出的测试程序，结果如下：

```
0
feature: tensor([[[0.0000, 0.0118, 0.2510,  ..., 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0196, 0.2549,  ..., 0.0078, 0.0078, 0.0039],
         [0.0000, 0.0118, 0.2627,  ..., 0.0510, 0.0471, 0.0510],
         ...,
         [0.2510, 0.2745, 0.3098,  ..., 0.1569, 0.0118, 0.0000],
         [0.0000, 0.0000, 0.0078,  ..., 0.1333, 0.0118, 0.0000],
         [0.0039, 0.0000, 0.0000,  ..., 0.1098, 0.0118, 0.0000]],

        [[0.0078, 0.0000, 0.1765,  ..., 0.0000, 0.0000, 0.0000],
         [0.0157, 0.0078, 0.1804,  ..., 0.0000, 0.0000, 0.0000],
         [0.0078, 0.0000, 0.1882,  ..., 0.0392, 0.0353, 0.0392],
         ...,
         [0.2510, 0.2745, 0.3098,  ..., 0.0902, 0.0000, 0.0118],
         [0.0000, 0.0000, 0.0078,  ..., 0.0784, 0.0000, 0.0039],
         [0.0039, 0.0000, 0.0000,  ..., 0.0549, 0.0000, 0.0078]],

        [[0.0000, 0.0000, 0.1490,  ..., 0.0078, 0.0078, 0.0078],
         [0.0000, 0.0000, 0.1529,  ..., 0.0039, 0.0039, 0.0000],
         [0.0000, 0.0000, 0.1647,  ..., 0.0196, 0.0157, 0.0196],
         ...,
         [0.2510, 0.2745, 0.3098,  ..., 0.0588, 0.0000, 0.0275],
         [0.0000, 0.0000, 0.0078,  ..., 0.0431, 0.0000, 0.0196],
         [0.0039, 0.0000, 0.0000,  ..., 0.0196, 0.0000, 0.0235]]])
labels: tensor([0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0,
        1, 1, 0, 1, 1, 1, 1, 0])
```

程序实现了图片分析以及相应的标签对应，满足数据处理要求。

2.2 MTCNN：人脸检测（解读与使用）

MTCNN论文的主要内容有：

- **三阶段的级联（cascaded）架构**
- **coarse-to-fine 的方式**
- **new online hard sample mining 策略**
- **同时进行人脸检测和人脸对齐**
- **state-of-the-art 性能**

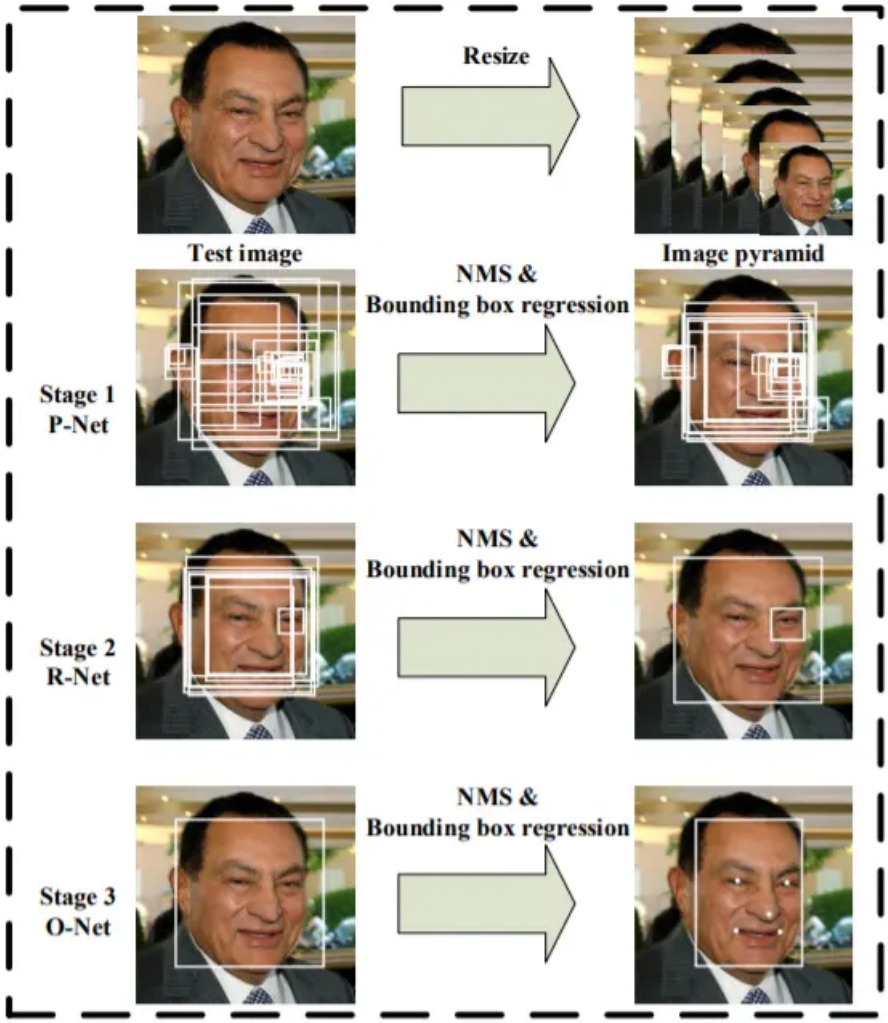


Fig. 1. Pipeline of our cascaded framework that includes three-stage multi-task deep convolutional networks. Firstly, candidate windows are produced through a fast Proposal Network (P-Net). After that, we refine these candidates in the next stage through a Refinement Network (R-Net). In the third stage, The Output Network (O-Net) produces final bounding box and facial landmarks position.

这里直接使用现有的表现较好的 MTCNN 的三个权重文件，它们已经保存在 `torch_py/MTCNN/weights` 文件夹下，路径如下：

```
pnet_path = "./torch_py/MTCNN/weights/pnet.npy"
rnet_path = "./torch_py/MTCNN/weights/rnet.npy"
onet_path = "./torch_py/MTCNN/weights/onet.npy"
```

通过搭建 MTCNN 网络实现人脸检测（搭建模型 py 文件在 torch_py/MTCNN 文件夹）

```
torch.set_num_threads(1)
# 读取测试图片
img = Image.open("test.jpg")
# 加载模型进行识别口罩并绘制方框
recognize = Recognition()
draw = recognize.face_recognize(img)
plot_image(draw)
```

程序测试结果如下：

2.3 口罩识别

首先加载预训练模型 MobileNet，代码如下：

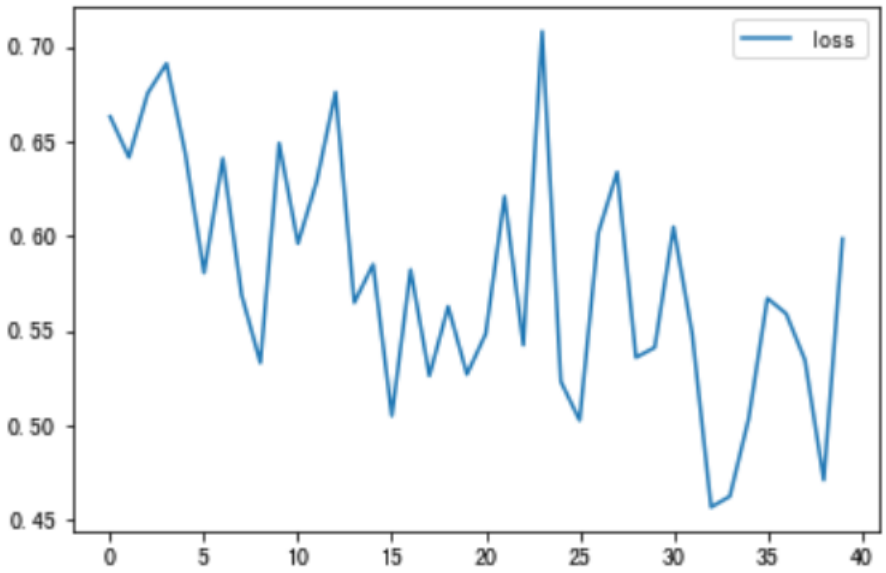
```
# 加载 MobileNet 的预训练模型权
device = torch.device("cuda:0") if torch.cuda.is_available() else
torch.device("cpu")
train_data_loader, valid_data_loader = processing_data(data_path=data_path,
height=160, width=160, batch_size=32)
modify_x, modify_y = torch.ones((32, 3, 160, 160)), torch.ones((32))

epochs = 2
model = MobileNetV1(classes=2).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)  # 优化器
print('加载完成...')
```

学习率（可调参）的手动设置可以使模型训练更加高效。这里我们设置当模型在两轮迭代后，准确率没有上升，就调整学习率。

```
# 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()
```

2.4 训练模型与结果分析

我们使用指导中给出的模型样例，使用给定的测试程序，测试结果如下：




```
all_num: 4 mask_num 0
```

整体上我们发现，模型训练过程中loss虽然整体逐渐下降，但是出现了较大的抖动。同时口罩检测效果也并不理想，需要进一步调试。

3. 实验过程与实验结果及分析

3.1 初次训练模型

实验要求通过训练模型完成人脸识别、戴口罩检测等功能。最初，我们采用给出的程序训练模型作为基础，设置参数为：

```
epochs = 20
lr = 1e-3
```

实验测试结果为：

测试详情

测试点	状态	时长	结果
在 5 张图片上测试模型	✓	12s	得分:77.5

确定

在模型训练中我们也发现，由于实验需要读取图像数据，数据处理量巨大，导致模型训练过程经常卡机中断。因此，为了保证模型训练正常，后续我们采用了GPU训练模型。使用GPU训练极大降低了训练时间。

3.2 调整参数

上述测试分数并不理想，我们观测GPU训练日志信息可见，整体的测试准确率早早地达到了较大值，部分中间时刻的准确率甚至达到了1.0，推测可能有过拟合的影响。因此我们降低训练轮数，使得`epcho = 10`，继续重复训练，得到以下结果：

测试详情

测试点	状态	时长	结果
在 5 张图片上测试模型	✓	5s	得分:87.5

确定

测试详情

测试点	状态	时长	结果
在 5 张图片上测试模型	✓	5s	得分:77.5

确定

测试详情

测试点	状态	时长	结果
在 5 张图片上测试模型	✓	5s	得分:90.0

确定

在这一段训练中我们没有改动代码，但出现了不一样的结果，因此我们推测为训练的随机性导致的结果。我们特殊保存了90分对应的模型。但同时，在后续的多次测试中，我们并没有发现更好的结果。

3.3 模型优化

继续查看GPU日志信息可见，在模型训练后的识别情况输出如下：

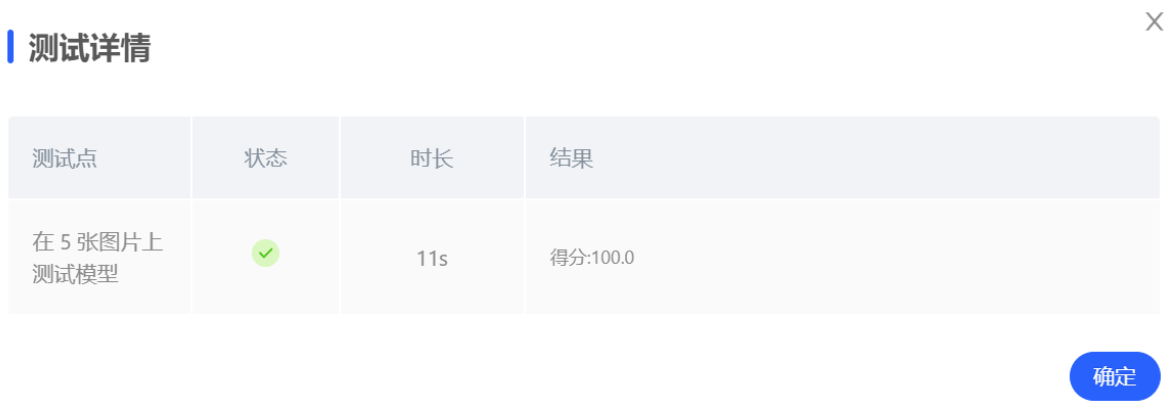
```
2024-12-30 14:29:19.281900 step:20/20 || Total Loss: 0.4086
2024-12-30 14:29:19.294700 Finish Training.
2024-12-30 14:29:19.573200 all_num: 4 mask_num 3
2024-12-30 14:29:19.816300 2 2
2024-12-30 14:29:20.490000 SYSTEM: Finishing...
```

而测试程序中使用了test1.jpg图片，具体如下：

可见，在程序中存在人脸识别不足的问题，人数与实际不匹配，因此实验模型得到的结果也必然正确率不足。我们考虑优化模型使得其能够应对这些情况。仿照程序`MobileNetV1`与相应的程序，我们设计类似的模块，继续使用GPU训练模型，得到新的结果。经过多次调参，我们使用新的函数训练模型的参数如下：

```
epcho = 30
lr = 1e-3
patience = 4
```

相应的模型测试结果如下：



可见模型训练结果良好，实现了满分的测试成绩。

4. 心得与体会

这次程序中主要包括人脸检测与口罩识别两部分，其中口罩识别为模型训练的主要问题对象。在实验过程中我首先利用原有的架构实现，并通过分析过拟合问题实现了一个良好的结果，但并无达到预期。在整体过程中，我通过多次分析GPU日志信息获取灵感，观察到人脸识别不足的问题并进行了相应的改进，最终调整参数实现了良好的结果。

通过这次实验，我也体会到对于数据预处理的关键性。就像本次实验中发现的，对于人脸检测的不足将会影响到最终的结果。因此我们对于这一问题进行了一定的改进，最终实现了很好的结果。

5. 附录：模型训练代码

```
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.MobileNetV2 import MobileNetV2
from torch_py.FaceRec2 import Recognition
# 数据集路径
data_path = "./datasets/5f680a696ec9b83bb0037081-momodel/data/"
mask_num = 4
fig = plt.figure(figsize=(15, 15))
for i in range(mask_num):
    sub_img = cv2.imread(data_path + "/image/mask/mask_" + str(i + 101) +
".jpg")
    sub_img = cv2.cvtColor(sub_img, cv2.COLOR_RGB2BGR)
    ax = fig.add_subplot(4, 4, (i + 1))
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title("mask_" + str(i + 1))
    ax.imshow(sub_img)
nomask_num = 4
fig1 = plt.figure(figsize=(15, 15))
for i in range(nomask_num):
    sub_img = cv2.imread(data_path + "/image/nomask/nomask_" + str(i + 130) +
".jpg")
    sub_img = cv2.cvtColor(sub_img, cv2.COLOR_RGB2BGR)
    ax = fig1.add_subplot(4, 4, (i + 1))
    ax.set_xticks([])
    ax.set_yticks([])
```

```
    ax.set_title("nomask_" + str(i + 1))
    ax.imshow(sub_img)
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"
torch.set_num_threads(1)
# 读取测试图片
img = Image.open("test.jpg")
# 加载模型进行识别口罩并绘制方框
recognize = Recognition()
draw = recognize.face_recognize(img)
plot_image(draw)
# 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=224, width=224, batch_size=32)
modify_x, modify_y = torch.ones((32, 3, 160, 160)), torch.ones((32))

epochs = 30
model = MobileNetV2(num_class=2).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)  # 优化器
print('加载完成...')
# 学习率下降的方式，acc四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=4)
# 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()
img = Image.open("test.jpg")
detector = FaceDetector()
recognize = Recognition(model_path='results/temp.pth')
draw, all_num, mask_nums = recognize.mask_recognize(img)
plt.imshow(draw)
plt.show()
print("all_num:", all_num, "mask_num", mask_nums)
```