

口罩佩戴检测

实战实验报告

控制科学与工程学院 3200100259 沈骏一

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。

1.4 注意事项

- Python 与 Python Package 的使用方式，可在 [API 文档](#) 中查阅。
- 当右上角的『Python 3』长时间指示为运行中的时候，造成代码无法执行时，可以重新启动 Kernel 解决（左上角『Kernel』 - 『Restart Kernel』）。

1.5 参考资料

- 论文 Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks : https://kpzhang93.github.io/MTCNN_face_detection_alignment/
- OpenCV : https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html
- PIL : <https://pillow.readthedocs.io/en/stable/>
- Numpy : <https://www.numpy.org/>

- Scikit-learn : <https://scikit-learn.org/>
- PyTorch : <https://pytorch.org/>

1.6 实验思路

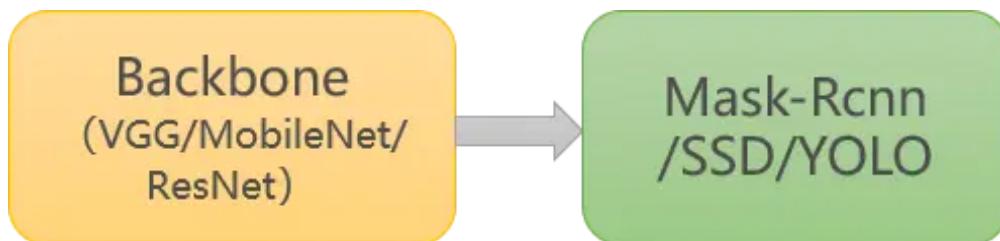
针对目标检测的任务，可以分为两个部分：目标识别和位置检测。

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。

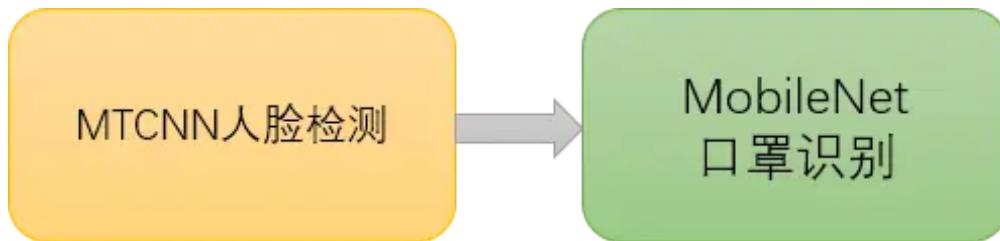
但 FC 对目标的位置识别乏力。经过算法的发展，当前主要以特定的功能网络来代替 FC 的作用，如 Mask-Rcnn、SSD、YOLO 等。

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.数据集介绍

2.1 导入 Python 第三方库（包）

```

In [ ]: 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
```

2.2 导入已经写好的 Python 文件

```
In [ ]: 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
```

```
In [ ]: # 数据集路径
data_path = "./data/"
```

现在我们尝试读取数据集中戴口罩的图片及其名称

以下是训练集当中的正样本：

```
In [ ]: 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_BGR2RGB)
    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)
```



以下是训练集当中的负样本：

```
In [ ]: 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_BGR2RGB)
    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)
```



2.4 调整图片尺寸

```
In [ ]: 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
```

可以用以下代码查看图片尺寸调整前后的对比

```
In [ ]: # 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("调整后图片的尺寸:", read_img.shape)
```

但是，由于我们直接使用 Transformer 库进行调整，以上函数并没有用到。

2.5 制作训练时所需的批量数据集

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

2.5.1 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=, 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

2.5.2 创建一个 `DataLoader` 对象

`DataLoader` 是 `torch` 给你用来包装你的数据的工具，所以你要将(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=, 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：

```
In [ ]: def processing_data(data_path, height=112, width=112, batch_size=256,
                           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
```

```
In [ ]: # 进行一次读取的试验，为节省篇幅不展示结果
train_data_loader, valid_data_loader = processing_data(data_path=data_path, height=112, width=112, batch_size=256, test_split=0.1)

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
```

3. MTCNN：人脸检测

3.1 MTCNN 解读

参考文献：《Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks》

文献与代码地址：https://kpzhang93.github.io/MTCNN_face_detection_alignment/

论文的主要贡献：

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

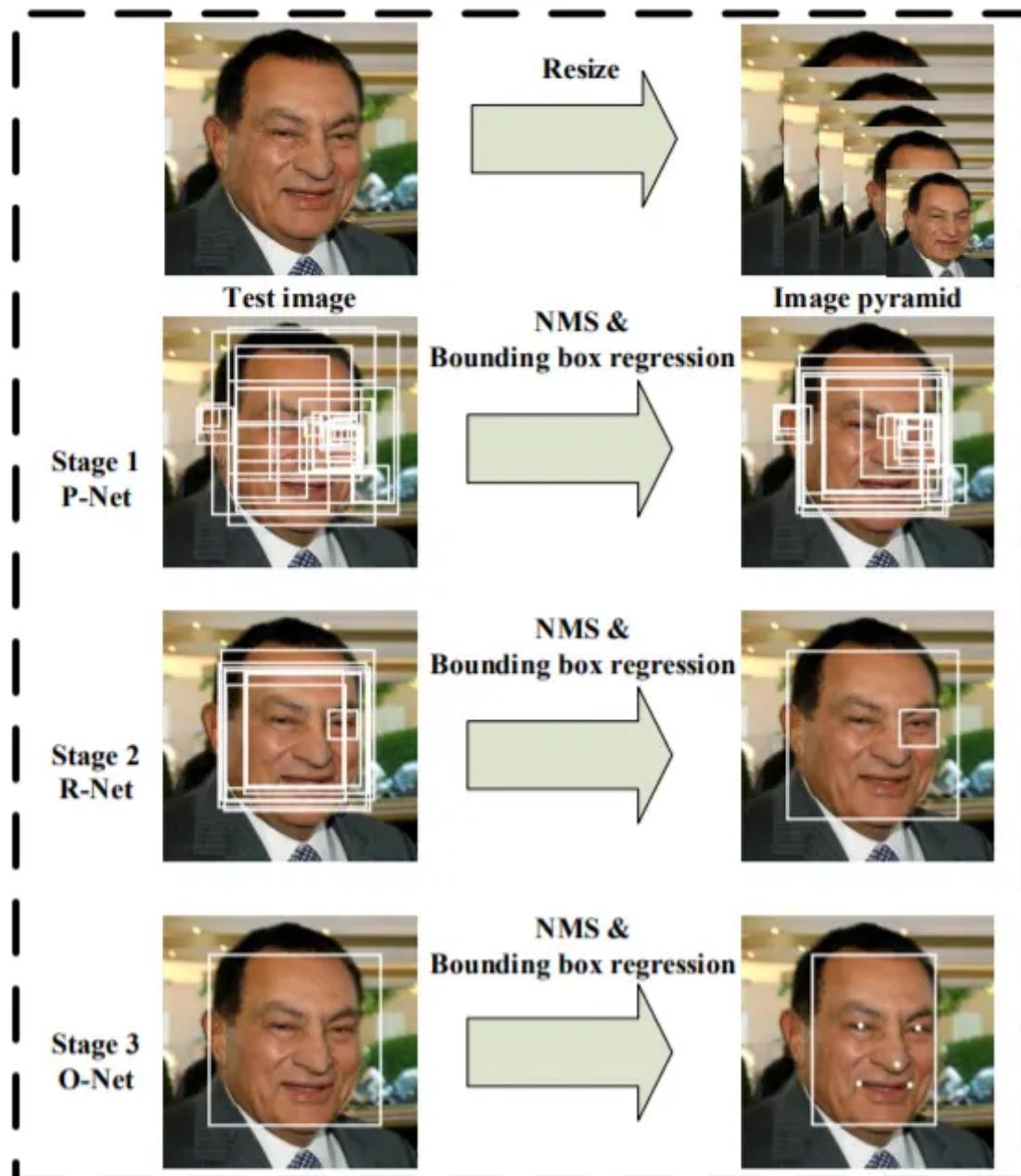


Fig. 1. Pipeline of our cascaded framework that includes three-stage multi-task deep convolutional networks. Firstly, candidate windows are produced through a fast Proposal Network (P-Net). After that, we refine these candidates in the next stage through a Refinement Network (R-Net). In the third stage,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

3.2 MTCNN 的使用

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：

```
In [ ]: 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）

```
In [ ]: 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)
```



4. 口罩识别

在提交的模型中，笔者受到 `Arcface` ECCV等文章的启发，决定舍弃提供的MobileNet框架，

而采用 Resnet50 的50层残差神经网络进行口罩佩戴的识别工作 具体模型结构在 `model.py` 中给出，
以下给出部分模型摘要

```
In [ ]: class ArcfaceNet(torch.nn.Module):
    def __init__(self, classes=2):
        super(ArcfaceNet, self).__init__()
        self.Arcnet = Backbone(num_layers=50, drop_ratio=0.6, mode='ir_se')
        #选择是否加载预训练模型
        #self.Arcnet.load_state_dict(torch.load('model_ir_se50.pth'))
        self.fc = torch.nn.Linear(512, classes)
        self.softmax = torch.nn.Softmax()
```

```

def forward(self, x):
    x = self.Arcnet(x)
    out = self.fc(x)
    out = self.softmax(out)
    return out

```

```

In [ ]: class Backbone(Module):
    def __init__(self, num_layers, drop_ratio, mode='ir'):
        super(Backbone, self).__init__()
        assert num_layers in [50, 100, 152], 'num_layers should be 50,100, 152'
        assert mode in ['ir', 'ir_se'], 'mode should be ir or ir_se'
        blocks = get_blocks(num_layers)
        if mode == 'ir':
            unit_module = bottleneck_IR
        elif mode == 'ir_se':
            unit_module = bottleneck_IR_SE
        self.input_layer = Sequential(Conv2d(3, 64, (3, 3), 1, 1, bias=False),
                                       BatchNorm2d(64),
                                       PReLU(64))
        self.output_layer = Sequential(BatchNorm2d(512),
                                       Dropout(drop_ratio),
                                       Flatten(),
                                       Linear(512 * 7 * 7, 512),
                                       BatchNorm1d(512))

        modules = []
        for block in blocks:
            for bottleneck in block:
                modules.append(
                    unit_module(bottleneck.in_channel,
                                bottleneck.depth,
                                bottleneck.stride))
        self.body = Sequential(*modules)

    def forward(self, x):
        x = self.input_layer(x)
        x = self.body(x)
        x = self.output_layer(x)
        return l2_norm(x)

```

主模型由 Resnet50 的Backbone加上一层线性的输出层组成
 转换成Tensor的图片向量经过50层残差神经网络生成一个512维的特征向量
 再输入到线性层形成2维的预测，经过 softmax 操作形成输出

4.1 加载预训练模型

```

In [ ]: from model import ArcfaceNet
#选择设备训练
device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
#加载数据，根据硬件条件选择合适的Batch_size
train_data_loader, valid_data_loader = processing_data(data_path='./data/images',
                                                       batch_size=16, shuffle=True)
#根据训练效果选择epochs
epochs = 500
model = ArcfaceNet(classes=2).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=5e-4) # 优化器
print('加载完成... ')

```

加载完成...

4.2 手动调整学习率(可调参)

学习率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。

```
In [ ]: # 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()
```

4.3 训练模型

考虑到 MTCNN 已经是一个比较成熟的人脸识别网络，故本次试验主要训练的是第二个网络

```
In [ ]: 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()
    #model.Arcnet.requires_grad = False
    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)

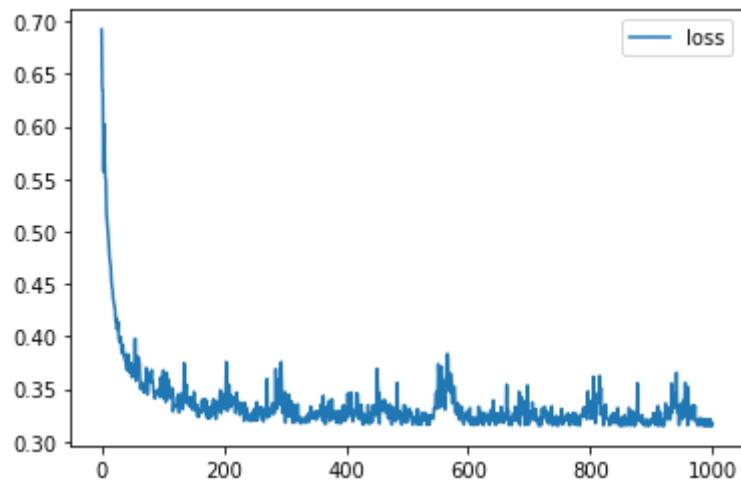
    print('step:' + str(epoch + 1) + '/' + str(epochs) + ' || Total Loss: %'
torch.save(model.state_dict(), 'temp.pth')
print('Finish Training.')
```

为了缩减报告长度，这里删除了上述训练过程的结果，具体可以看4.4中的Loss图像发现大概趋势

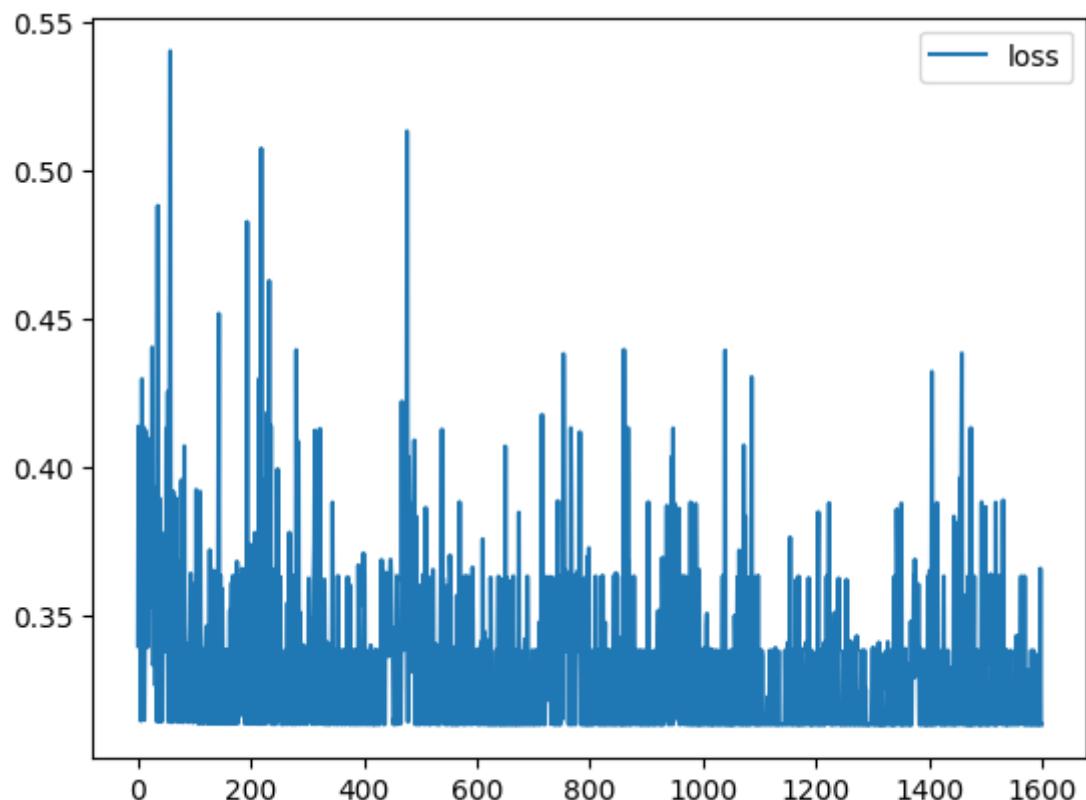
4.4 展示模型训练过程

```
In [ ]: train_result = []
for loss in loss_list :
    train_result.append(loss.detach().to('cpu').numpy())
plt.plot(train_result,label = "loss")
```

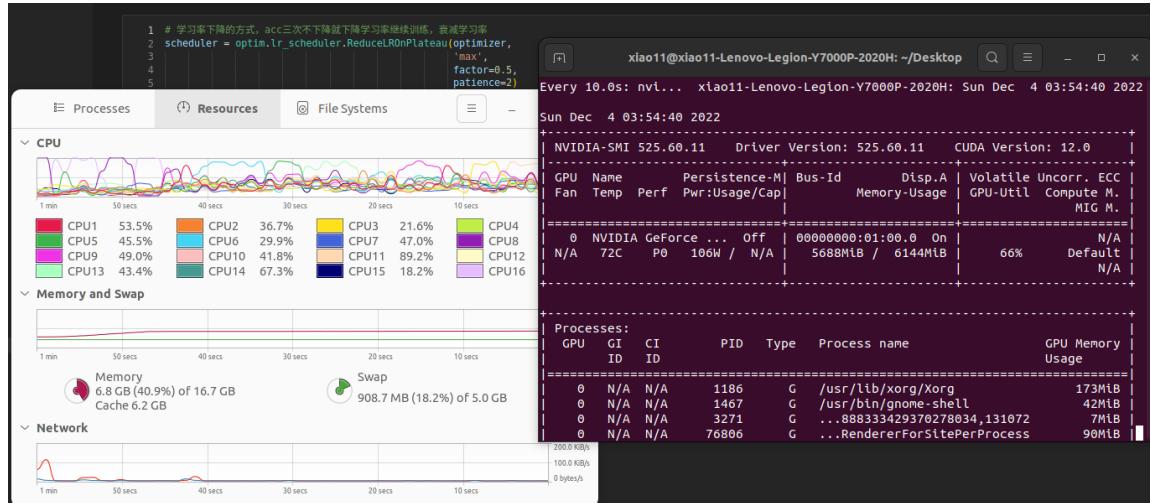
```
plt.legend()  
plt.show()
```



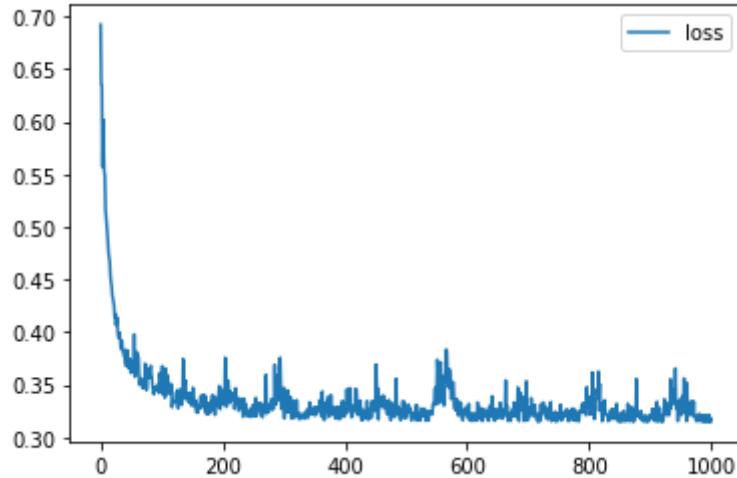
再训练发现Loss很难下降：



训练过程中电脑资源的使用情况：



在 opendl 平台上尝试放开 Resnet50，
进行整个模型的训练：



但从测试结果来看，识别准确度与得分并没有得到很大的提升。

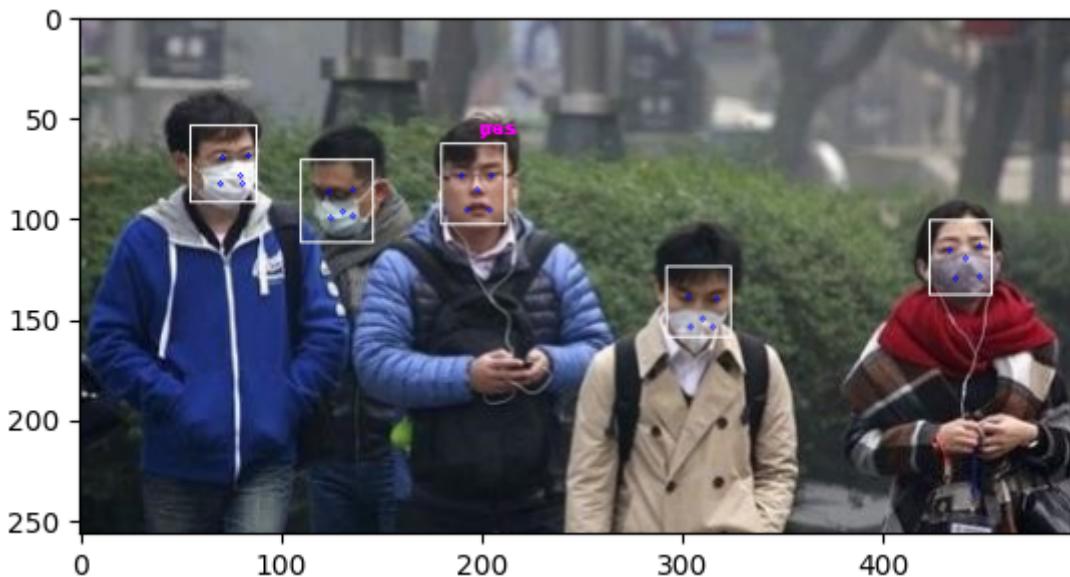
4.5 检测图片中人数及戴口罩的人数

```
In [ ]: img = Image.open("test.jpg")
detector = FaceDetector()
recognize = Recognition(model_path='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)
```



all_num: 4 mask_num 3

```
In [ ]: img = Image.open("test1.jpg")
detector = FaceDetector()
recognize = Recognition(model_path='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)
```



all_num: 5 mask_num 4

可以发现模型已经达到了比较好的预测效果。

4.6 其他尝试的模型

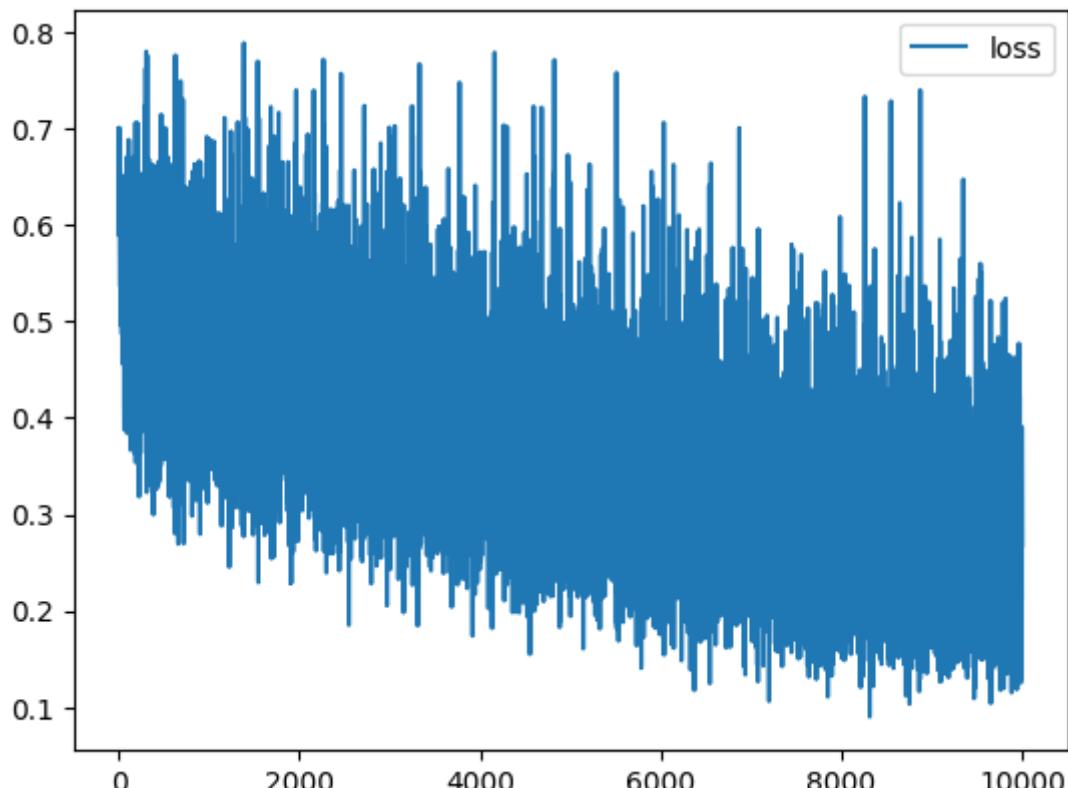
除了上述比较成功的模型之外，笔者也尝试了项目给出的 MobileNet 进行测试

```
In [ ]: class MobileNetV1(nn.Module):
    def __init__(self, classes=2):
        super(MobileNetV1, self).__init__()
        self.mobilebone = nn.Sequential(
            self._conv_bn(3, 32, 2),
            self._conv_dw(32, 64, 1),
            #self._conv_dw(64, 128, 2),
            #self._conv_dw(128, 128, 1),
            #self._conv_dw(128, 256, 2),
            #self._conv_dw(256, 256, 1),
            #self._conv_dw(256, 512, 2),
            #self._top_conv(512, 512, 5),
            #self._conv_dw(512, 1024, 2),
            #self._conv_dw(1024, 1024, 1),
        )
        # self.avgpool = nn.AvgPool2d(kernel_size=7, stride=1)
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(64, classes)
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, (2. / n) ** .5)
            if isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

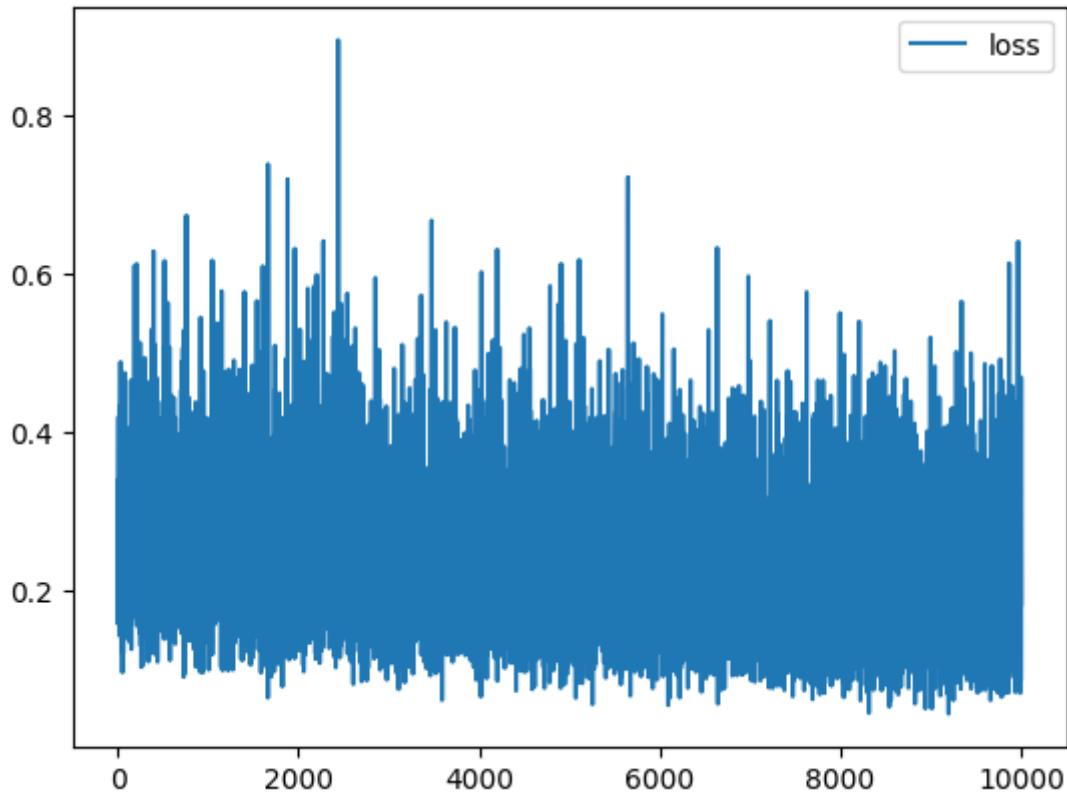
    def forward(self, x):
        x = self.mobilebone(x)
        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        out = self.fc(x)
        return out
```

训练结果：

500epoch：



1000epoch :



在系统测试后，由于模型过于简单化，并没有达到预期的效果与分数。

故修改原本的模型继续测试：

```
In [ ]: class MobileNetV1(nn.Module):
    def __init__(self, classes=2):
        super(MobileNetV1, self).__init__()

        self.mobilebone = nn.Sequential(
            self._conv_bn(3, 32, 2),
            self._conv_dw(32, 64, 1),
            self._conv_bn(64, 128, 2),
            self._conv_dw(128, 128, 1),
            self._conv_bn(128, 256, 2),
            self._conv_dw(256, 256, 1),
            self._conv_dw(256, 512, 2),
            self._top_conv(512, 512, 5),
        )
```

训练完成后，在后续的测试环节中也取得了比较好的效果。

5. 模型预测

口罩佩戴检测模型训练流程，包含数据处理、创建模型、训练模型、模型保存、评价模型等。
训练模型可以参考第 4.3 部分训练模型代码

如果对训练出来的模型不满意，你可以通过调整模型的参数等方法重新训练模型，直至训练出你满意的模型。

如果你对自己训练出来的模型非常满意，则可以提交作业！

注意：

1. 你可以在我门准好的接口中实现深度学习模型（若使用可以修改函数接口），也可以自己实现深度学习模型。
2. 写好代码后可以在 Py 文件中使用 [离线任务](#) 进行模型训练。
3. **使用离线训练模型必须保存在 results 文件夹。**
4. 将自己认为最佳模型保存在 result 文件夹，其余模型备份在项目中其它文件夹，方便您加快测试通过。

===== 实现自己的深度学习模型代码答题区域

双击下方区域开始编写 **数据处理、创建模型、训练模型、保存模型** 和 **评估模型** 等部分的代码，请勿在别的位置作答

以上部分代码见第四模块

===== 模型预测代码答题区域

在下方的代码块中编写 **模型预测** 部分的代码，请勿在别的位置作答

```
In [ ]: import warnings
# 忽视警告
warnings.filterwarnings('ignore')

import cv2
from PIL import Image
import numpy as np
import torch
import torch.nn as nn
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 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.
model_path = None
# -----


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
    # 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))
    detector = FaceDetector()
    recognize = Recognition(model_path='test.pth')
```

```

recognize.mobilenet.eval()
draw, all_num, mask_num = recognize.mask_recognize(img)

#
#-----#
return all_num,mask_num

```

In []: #预测测试
predict(cv2.imread('test.jpg'))

Out[]: (4, 3)

In []: #预测测试
predict(cv2.imread('test1.jpg'))

Out[]: (5, 4)

经验证，以上测试结果都是准确的

In []: #转换模型
state_dict = torch.load('temp2.pth', map_location="cpu")
torch.save(state_dict, 'test2.pth', _use_new_zipfile_serialization=False)

在完成模型转换后，上传至Mo平台进行测试，比较好的结果如下：

测试详情

测试点	状态	时长	结果
在 5 张图片上 测试模型	✓	13s	得分:98.33333333333333

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	✓	6s	得分:100.0

确定

6. 实验总结与感想

在本次实验中，我通过阅读样例代码，逐渐掌握了基本图像的神经网络学习方式。并通过自我学习，逐渐掌握了CNN卷积神经网络的结构与原理，实现了模型的搭建与训练。选择最终的合适模型也是一个比较复杂的过程，在整个过程中，我尝试了MobileNet以及Resnet50等多种模型，并通过调节epochs等多种超参数不断提升模型在Mo平台上的准确

度。

最终的结果是令人可喜的，对于日后的学习是一个有用的经历。