1 实验介绍

1.1 实验背景

前几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

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。

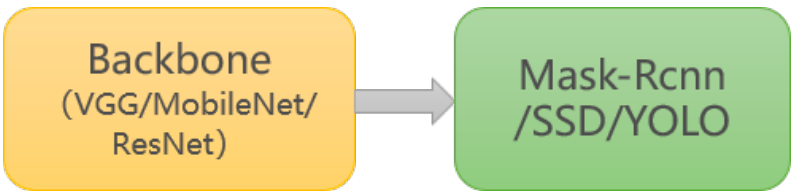
1.4 实验思路

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。

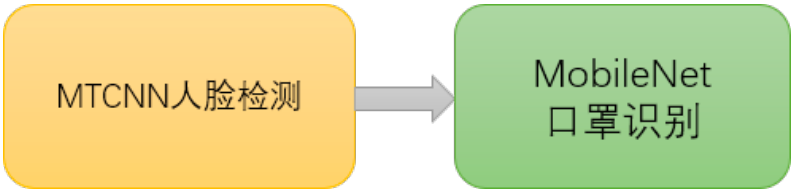
通常情况下，特征提取需要由特有的特征提取神经网络来完成，如 VGG、MobileNet、ResNet 等，这些特征提取网络往往被称为 Backbone 。而在 BackBone 后面接全连接层 (FC) 就可以执行分类任务。但 FC 对目标的位置识别乏力。经过算法的发展，当前主要以特定的功能网络来代替 FC 的作用，如 Mask-Rcnn、SSD、YOLO 等。

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



本次实验



2 数据预处理

2.1 数据集介绍

数据信息存放在 '/datasets/5f680a696ec9b83bb0037081-momodel/data' 文件夹下。

- 'image' 文件夹：'mask' 图片 313 张，'no mask' 图片 443 张
- 'train.txt': 图片对应的 label

2.2 预处理

预处理部分分为以下几步：

(1). 统一大小，随机翻转并归一化

(2). 按比例划分训练集和数据集

(3). 创建 **Dataset** 和 **Dataloader** 对象

```
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,num_workers=4,pin_memory=True)
    valid_data_loader = DataLoader(test_dataset, batch_size=batch_size,
        shuffle=True,num_workers=4,pin_memory=True)

    return train_data_loader, valid_data_loader
```

3 MTCNN: 人脸检测

3.1 MTCNN 原理

MTCNN（Multi-task Cascaded Convolutional Networks）是一种用于人脸检测的深度学习模型，它通过三个级联的卷积神经网络（CNN）来实现高效且准确的检测。MTCNN 的优势在于，它可以同时进行人脸检测、关键点检测（如眼睛、鼻子、嘴巴等关键部位）和人脸对齐，因此被广泛用于人脸识别、面部表情分析等任务。

MTCNN 由三个主要的网络组成，分别为：

(1). P-Net（Proposal Network）：人脸候选区域生成网络

(2). R-Net（Refine Network）：人脸候选区域精细化网络

(3). O-Net（Output Network）：人脸最终检测及关键点回归网络

MTCNN 的每个网络都依赖于前一个网络的输出。P-Net 负责初步筛选，R-Net 精细化这些候选框，O-Net 则进一步优化框的位置，并进行关键点回归。通过这种逐步筛选和精细化的方式，MTCNN 能够高效且准确地检测到人脸，并提供关键点位置。

3.2 检测效果

这里我们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。对示例图片进行检测：

可以看到，使用预训练的权重文件的准确率很高，因此我们不再进行训练调整。

4 训练 MobileNet

4.1 初始网络结构

MobileNet 是一个轻量级的卷积神经网络架构，旨在移动设备和嵌入式设备上运行时，提供良好的性能，同时保持较低的计算复杂度和内存消耗。MobileNet 通过深度可分离卷积（Depthwise Separable Convolution）来降低计算量，是其主要的创新之一。

初始的网络结构由以下几部分构成：

(1) 卷积层：

```
self._conv_bn(3, 32, 2)
```

MobileNetV1 的网络结构一开始通常是一个普通的卷积层，后面会紧跟一个批归一化层（Batch Normalization）。它接受输入通道数、输出通道数以及步幅作为参数。这种初始化是网络的开始部分，通常在这里进行下采样，同时增加特征的深度。

(2) 深度可分离卷积层：

```
self._conv_dw(32, 64, 1)
```

这个层的作用是通过深度卷积和逐点卷积来进一步提取特征。输入和输出通道分别为 32 和 64。

- 深度卷积（Depthwise Convolution）：每个输入通道与其对应的滤波器进行卷积。
- 逐点卷积（Pointwise Convolution）：使用 1x1 卷积来混合深度卷积的输出。

(3) 自适应平均池化：

`self.avg_pool = nn.AdaptiveAvgPool2d(1)`

MobileNetV1 使用了全局平均池化（Global Average Pooling）来减少特征图的尺寸。将每个通道的特征图压缩成一个单一的数值，即使输入图像的尺寸不同，输出的特征图形状也会一致，变成 [batch_size, channel, 1, 1]。

(4) 全连接层：

`self.fc = nn.Linear(64, classes)`

这个全连接层将池化后的特征图展平，并进行最终的分类输出。64 是通道数，classes 是输出的类别数。作用是根据提取到的特征进行分类。

4.2 调整学习率

学习率的适当调整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，在验证集上的 loss 没有下降，就调整学习率。注意到教程中实际没有在训练上应用这一点，应使用 step() 进行调用

```
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                                     'max',
                                                     factor=0.8,
                                                     patience=2)
    ...
    # 调用学习率调度器
    scheduler.step(avg_val_loss)
```

4.3 训练指标记录

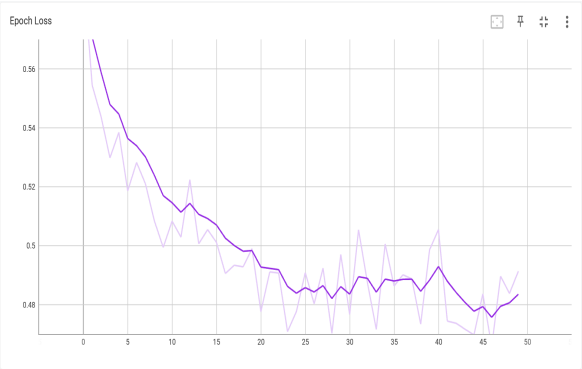
由于教程中的 lost_list 占用内存开销较大且过于单一，这里使用 tensorboard 对训练的每一 batch 的 Loss，每一 epoch 的平均 Loss，以及每一 epoch 后在 validation_set 上的 Loss 和准确率都进行了记录。

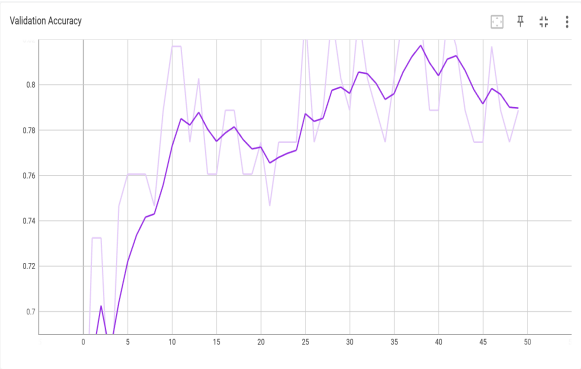
```
# 初始化 TensorBoard 记录器
writer = SummaryWriter(log_dir='./runs/experiment256_100_batch8')

for epoch in range(epochs):
    model.train()
    epoch_loss = 0
    ...
        epoch_loss += loss.item()
        # 每个批次记录到 TensorBoard
```

```
        writer.add_scalar('Batch Loss', loss.item(), epoch * len(
            train_data_loader) + batch_idx)

    # 记录每个 epoch 的平均损失
    avg_loss = epoch_loss / len(train_data_loader)
    writer.add_scalar('Epoch Loss', avg_loss, epoch)

    # 验证阶段
    model.eval()
    val_loss = 0
    with torch.no_grad():
        correct = 0
        total = 0
        for x_val, y_val in valid_data_loader:
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            pred_y_val = model(x_val)

            val_loss += criterion(pred_y_val, y_val).item()
            _, predicted = torch.max(pred_y_val, 1)
            correct += (predicted == y_val).sum().item()
            total += y_val.size(0)

    avg_val_loss = val_loss / len(valid_data_loader)
    val_accuracy = correct / total

    # 记录验证损失和准确率
    writer.add_scalar('Validation Loss', avg_val_loss, epoch)
    writer.add_scalar('Validation Accuracy', val_accuracy, epoch)
```

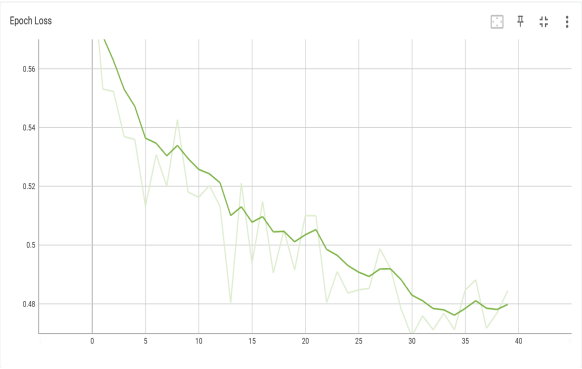
4.4 训练效果

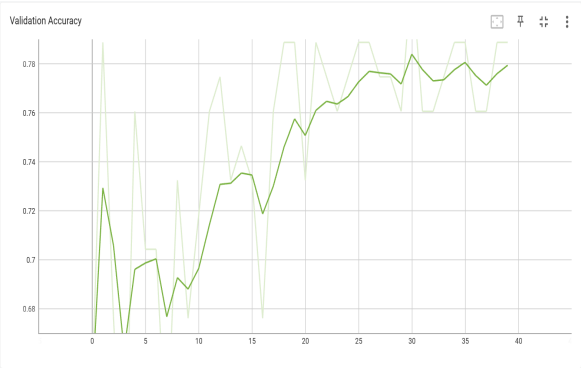
设置 batch size 为 8，训练 50 个 epoch，观察训练集上的 Loss 和验证集上的 accuracy 的变化曲线：



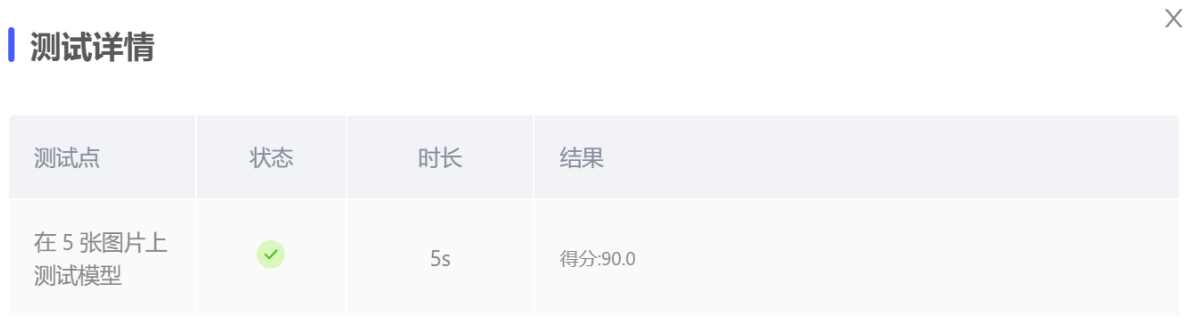


可以看到，在 40 个 epoch 后在验证集上的 accuracy 出现下降，而在训练集上的 Loss 基本达到稳定，因此后面极有可能为过拟合。故选择训练 40 个 epoch：





可以看到训练效果较好，在 mo 平台上进行测试，成绩达到 90 分：



不过，我们可以看到训练的 loss 实际上还比较大，在 0.4 左右，因此在下一章节我们继续进行优化。

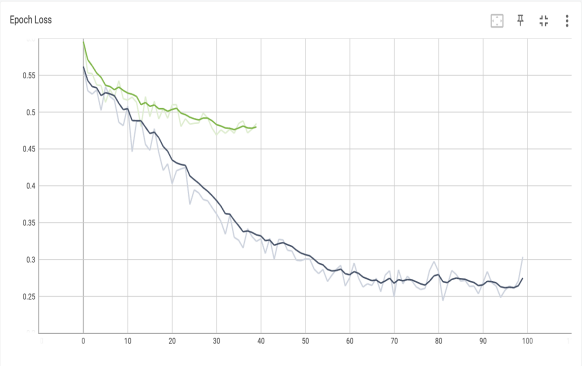
5 优化模型与参数

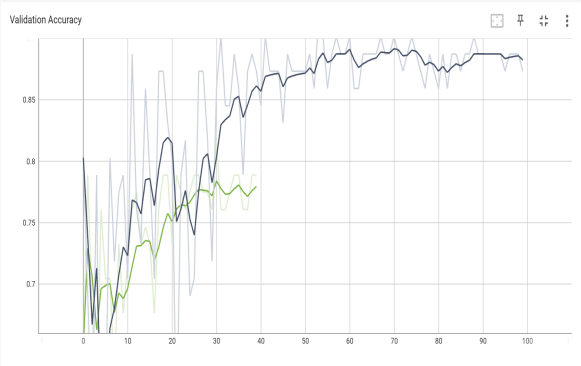
5.1 增加模型深度

原先的 mobilebone 只有一层卷积层和一层深度可分离卷积层，输出通道为 64，我们添加几层深度可分离卷积层，使输出通道数为 256：

```
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),
        )
```

仍取 $batchsize = 8$，训练 100 个 epoch：

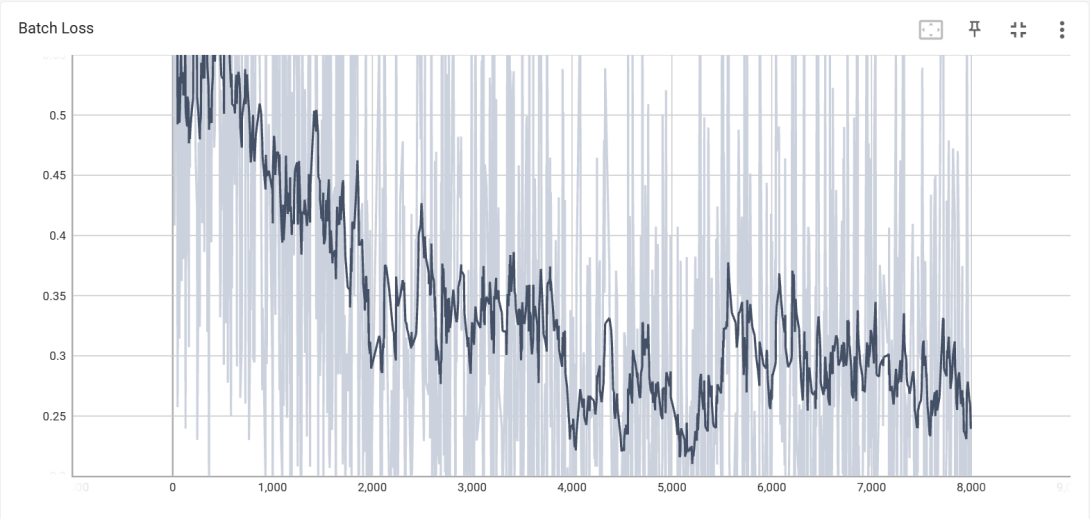




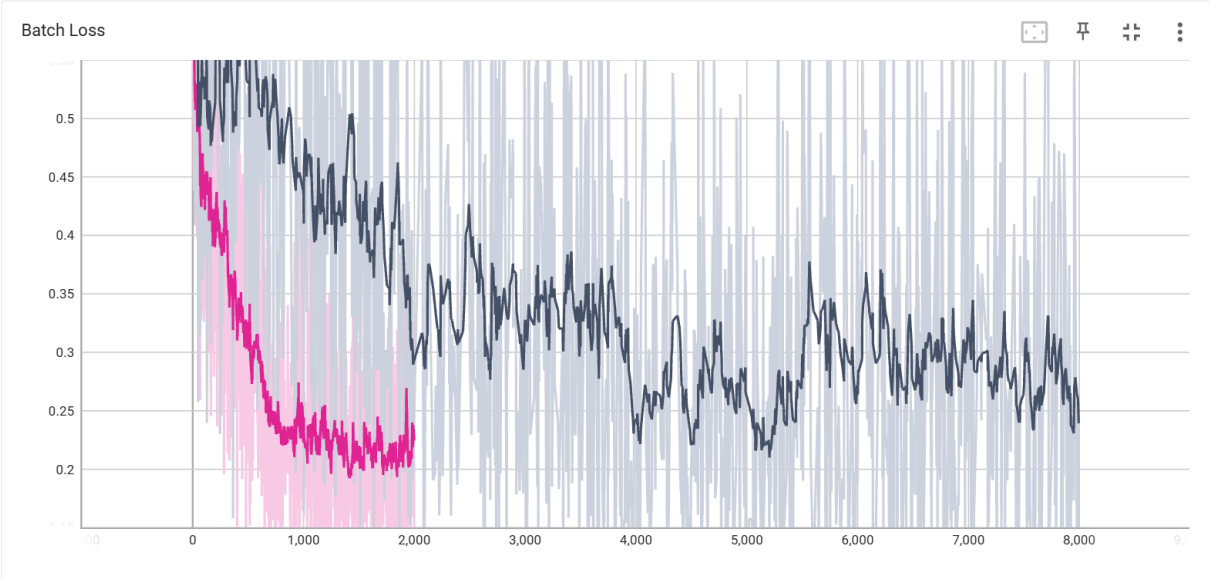
可以看到对比原先的模型结构，Loss 显著下降，到达 0.25 左右，在验证集上的准确率也显著提高。

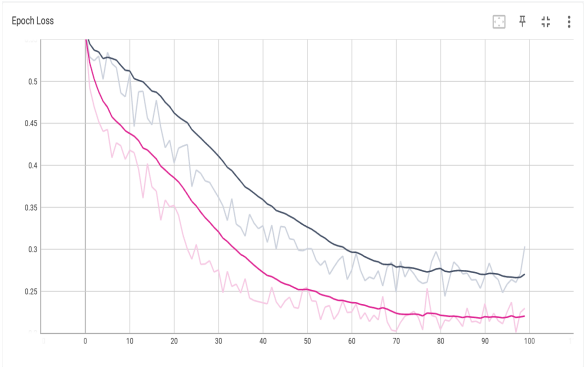
5.2 增大 batch size

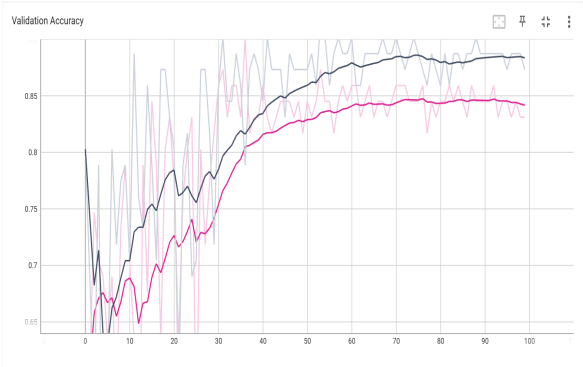
当我们观察在逐个 batch 上的 Loss 时，可以看到较大的震荡，因此考虑增大 batch size



由于本次实验样本数较少，考虑将 batch size 取为 32，发现稍有改善：







在 mo 平台上进行测试，达到 90 分

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:90.0

由于在 60 个 epoch 后 Loss 下降幅度很小，且 accuracy 略有下降，因此推测可能有过拟合现象。将 epoch 调整为 60 重新训练，并上传到 mo 平台进行测试，发现分数可以达到 97.5 分，验证了前面过拟合的猜想：

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:97.5

6 最终模型

根据前面的思想，经过多次调试，最终将模型结构调整如下，输出通道数为 512：

```
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._conv_dw(512, 512, 1),
        )
```

训练的 epoch 定为 60 个回合，在 mo 平台上测试达到 100 分

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

7 总结

通过这次实验我加深了对 pytorch 框架的掌握，同时通过观察 Loss、accuracy，我一步步地调整 batch size，模型结构等，使得模型的表现逐渐提升，是一次很好地实践机会。

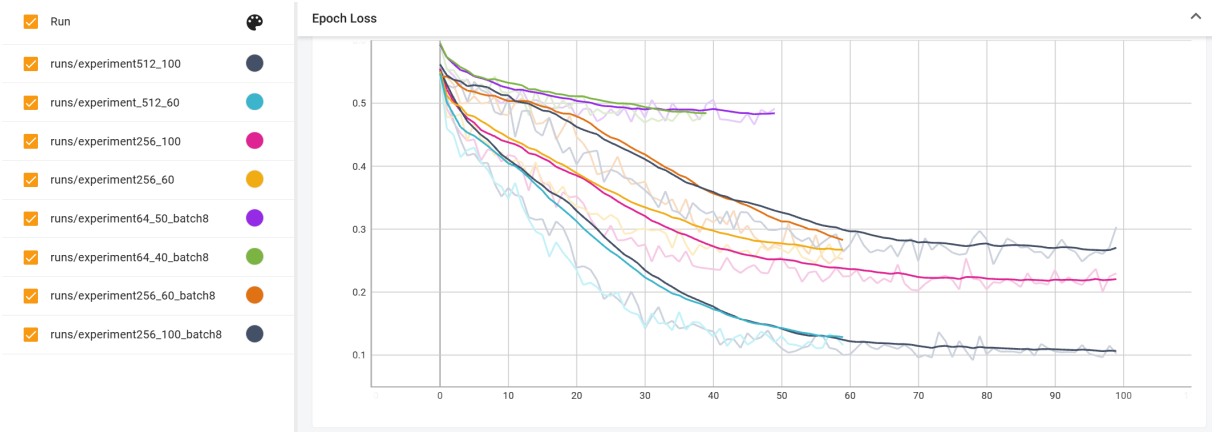


Figure 1: 不同模型结构和参数的 Loss 变化曲线

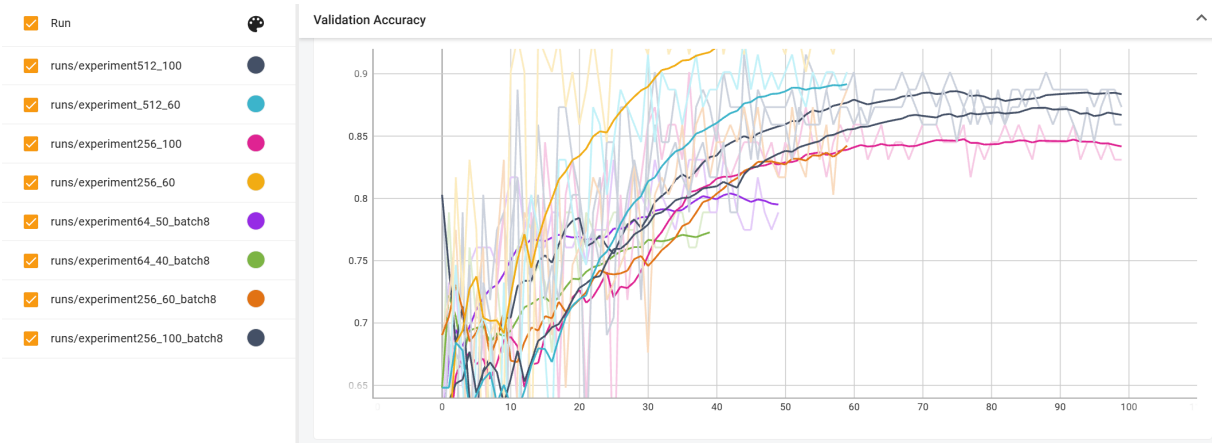


Figure 2: 不同模型结构和参数的 Accuracy 变化曲线

8 代码附录

```python
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

from torch.utils.tensorboard import SummaryWriter # 导入 TensorBoard

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,num_workers=4,pin_memory=True)
    valid_data_loader = DataLoader(test_dataset, batch_size=batch_size,
        shuffle=True,num_workers=4,pin_memory=True)

    return train_data_loader, valid_data_loader

# 数据集路径
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=8)

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.
    device("cpu")
print(device)

epochs = 100
model = MobileNetV1(classes=2).to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001) # 优化器

# 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=0.8,
                                                 patience=2)
```

```
70  # 损失函数
71  criterion = nn.CrossEntropyLoss()
72
73  # 初始化 TensorBoard 记录器
74  writer = SummaryWriter(log_dir='./runs/experiment256_100_batch8')
75
76  for epoch in range(epochs):
77      model.train()
78      epoch_loss = 0
79      for batch_idx, (x, y) in enumerate(train_data_loader, 1):
80          x = x.to(device)
81          y = y.to(device)
82          pred_y = model(x)
83
84          # print(pred_y.shape)
85          # print(y.shape)
86
87          loss = criterion(pred_y, y)
88          optimizer.zero_grad()
89          loss.backward()
90          optimizer.step()
91
92          epoch_loss += loss.item()
93          # 每个批次记录到 TensorBoard
94          writer.add_scalar('Batch Loss', loss.item(), epoch * len(
                train_data_loader) + batch_idx)
95
96      # 记录每个 epoch 的平均损失
97      avg_loss = epoch_loss / len(train_data_loader)
98      writer.add_scalar('Epoch Loss', avg_loss, epoch)
99
100     # 验证阶段
101     model.eval()
102     val_loss = 0
103     with torch.no_grad():
104         correct = 0
105         total = 0
106         for x_val, y_val in valid_data_loader:
```

```
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            pred_y_val = model(x_val)

            val_loss += criterion(pred_y_val, y_val).item()
            _, predicted = torch.max(pred_y_val, 1)
            correct += (predicted == y_val).sum().item()
            total += y_val.size(0)

    avg_val_loss = val_loss / len(valid_data_loader)
    val_accuracy = correct / total

    # 记录验证损失和准确率
    writer.add_scalar('Validation Loss', avg_val_loss, epoch)
    writer.add_scalar('Validation Accuracy', val_accuracy, epoch)

    # 调用学习率调度器
    scheduler.step(avg_val_loss)

    print(f'Epoch: {epoch + 1}/{epochs}, Average Loss: {avg_loss:.4f}')

torch.save(model.state_dict(), './results/temp.pth')
print('Finish Training.')
writer.close()
```