Mo Lab3 Report - Mask Detection

万晨阳 3210105327

1. 实验内容介绍

1.1 实验背景

2020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

- 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。
- 学习经典的模型 mtcnn 和 mobilenet 的结构。
- 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 **MindSpore、Keras** 等框架建立深度学习模型等。

1.4 参考资料

- 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/
- mindspore：https://www.mindspore.cn/tutorial/training/zh-CN/master/index.html
- tensorflow：https://www.tensorflow.org/api_docs/python/tf?hl=zh-cn
- keras：https://keras.io/zh/

2. 模型训练与调参

2.1 模型介绍

2.1.1 MTCNN

- **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**

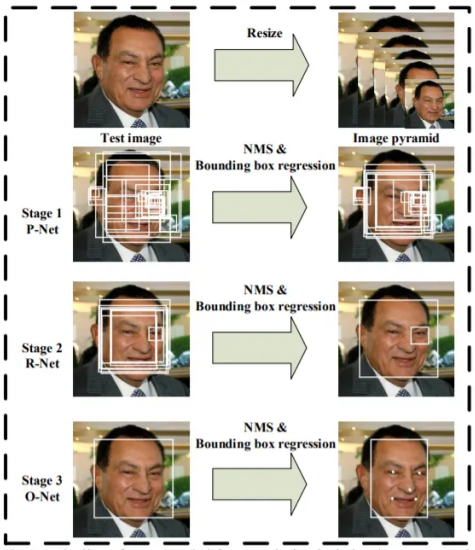


Fig. 1. Pipeline of our cascaded framework that includes three-stage multi-task deep convolutional networks. Firstly, candidate windows are produced through a fast Proposal Network (P-Net). After that, we refine these candidates in the next stage through a Refinement Network (R-Net). In the third stage,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

2.1.2 MobileNet V1

MobileNet是基于深度级可分离卷积构建的网络，它是将标准卷积拆分为了两个操作：深度卷积 `depthwise convolution` 和 逐点卷积 `pointwise convolution` ， `depthwise convolution` 和标准卷积不同，对于标准卷积其卷积核是用在所有的输入通道上，而 `depthwise convolution` 针对每个输入通道采用不同的卷积核，就是说一个卷积核对应一个输入通道，所以说 `depthwise convolution` 是depth级别的操作。而 `pointwise convolution` 其实就是普通的卷积，只不过其采用1x1的卷积核。

网络主体结构如下。首先是一个 3x3 的标准卷积，然后后面就是堆积 `depthwise separable convolution` ，并且可以看到其中的部分 `depthwise convolution` 会通过 `strides=2` 进 行 `down sampling` 。然后采用 `average pooling` 将 feature 变成 1x1 ，根据预测类别大小加上全连接层，最后是一个 `softmax` 层。

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Table 2. Resource Per Layer Type

Type	Mult-Adds	Parameters
Conv 1×1	94.86%	74.59%
Conv DW 3×3	3.06%	1.06%
Conv 3×3	1.19%	0.02%
Fully Connected	0.18%	24.33%

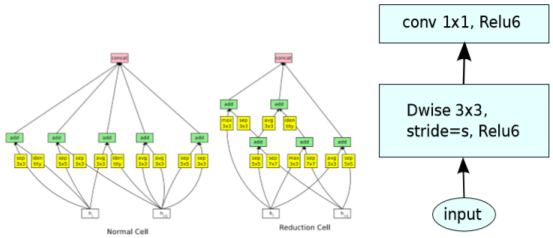
2.1.3 MobileNet V2

MobileNet V1是Mo平台给出的版本，MobileNet V2是升级版。MobileNetV2网络是由google团队在2018年提出的，相比MobileNet V1网络，准确率更高，模型更小。

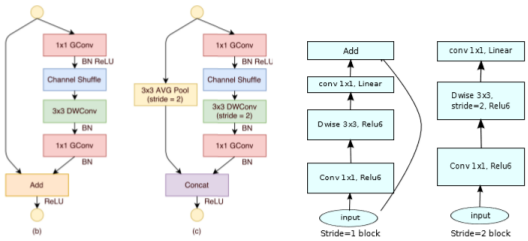
网络中的亮点：

- Inverted Residuals （倒残差结构 ）

- Linear Bottlenecks
  论文中称网络层中的激活特征为兴趣流形（mainfold of interest），我们设计网络结构的时候，想要减少运算量，就需要尽可能将网络维度设计的低一些但是维度如果低的话，激活变换ReLU函数可能会滤除很多有用信息。然后我们就想到了，既然ReLU另外一部分就是一个线性映射。那么如果我们全用线性分类器，会不会就不会丢失一些维度信息，同时可以设计出维度较低的层呢？**V2针对这个问题使用linear bottleneck(即不使用ReLU激活，做了线性变换)的来代替原本的非线性激活变换。**到此，优化网络架构的思路也出来了：通过在卷积模块中后插入linear bottleneck来捕获兴趣流形。 实验证明，使用linear bottleneck可以防止非线性破坏太多信息。

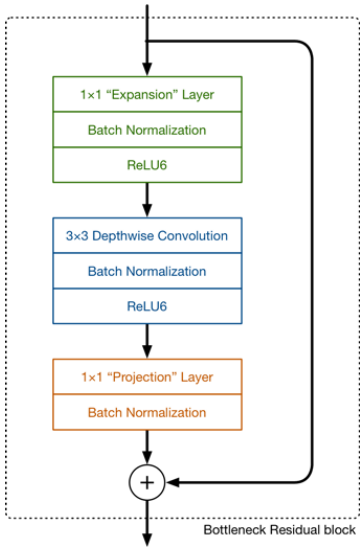


(a) NasNet[23] (b) MobileNet[27]



(c) ShuffleNet [20] (d) Mobilenet V2

网络主要结构如下：



Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

V1与V2的性能对比如下：

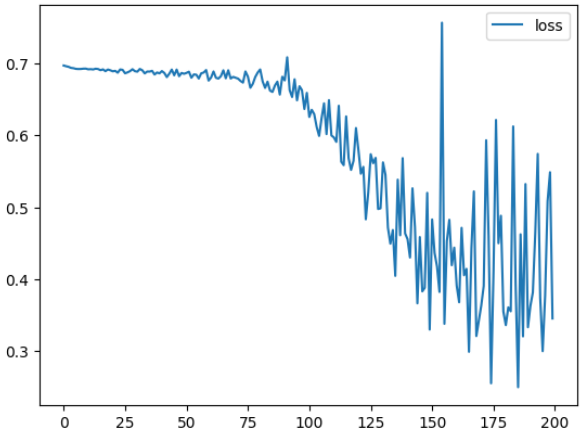
Network	Top 1	Params	MAdds	CPU
MobileNetV1	70.6	4.2M	575M	113ms
ShuffleNet (1.5)	71.5	**3.4M**	292M	-
ShuffleNet (x2)	73.7	5.4M	524M	-
NasNet-A	74.0	5.3M	564M	183ms
MobileNetV2	**72.0**	**3.4M**	**300M**	**75ms**
MobileNetV2 (1.4)	**74.7**	6.9M	585M	**143ms**

2.2 模型训练

首先，本任务中的object detection的任务为两部分，MCTNN负责对人脸进行识别，而MobileNet负责实现对得到的bbox中的口罩进行进一步的分类工作，所以我们的工作是要求对于MTCNN以及MobileNet的任务效果分别进行改进。这两部分的工作室可以解耦的：对于MTCNN来说，平台上为我们提供了预训练好的权重，使得模型效果满足要求，所以我们考虑不做进一步的修改，而把重心放在MobileNet网络的学习和训练上。以下的内容主要针对MobileNet。

2.2.1 MobileNet V1 模型的训练

使用模型中原有的代码进行参数调整进行训练，发现训练集准确率在70%上下波动。我们通过调整参数（学习率、门限函数等）结果没有得到很明显的改善。其中一次MobileNetV1训练10个epoch后的效果如下，可以看到loss在0.4左右波动。



因此我们思考改善网络结构。选择使用MobileNet V2 模型。

2.2.2 MobileNet V2 模型的训练

对于MobileNet V2的训练过程和V1基本相同。我们基于给出的`FaceDec.py`和`MobileNetV1.py`，编写`FaceDec2.py`和`MobileNetV2.py`供我们的模型训练使用。以下为代码，其中做出修改的部分以注释标识说明。

- `FaceDec2.py`中完成的是对于调用模型进行人脸识别输出bbox和bbox内mask与no mask的二分类任务。我们主要修改的内容是**对输入进行图片大小的调整以适合模型的输入。**

```
import torch
import numpy as np

from PIL import Image
from PIL import Image, ImageDraw, ImageFont
from matplotlib import pyplot as plt
from torchvision.transforms import transforms
import cv2

try:
    from MTCNN.detector import FaceDetector
    from MobileNetV2 import MobileNetV2
except:
    from .MTCNN.detector import FaceDetector
    from .MobileNetV2 import MobileNetV2

def plot_image(image, image_title="", is_axis=False):
    plt.imshow(image)
    if not is_axis:
        plt.axis('off')
```

```
    plt.title(image_title)
    plt.show()

class Recognition(object):
    classes = ["mask", "no_mask"]

    def __init__(self, model_path=None):
        self.detector = FaceDetector()
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.mobilenet = MobileNetV2(num_classes=2)
        if model_path:
            self.mobilenet.load_state_dict(
                torch.load(model_path, map_location=device))

    def face_recognize(self, image):
        drawn_image = self.detector.draw_bboxes(image)
        return drawn_image

    def mask_recognize(self, image):
        b_boxes, landmarks = self.detector.detect(image)
        detect_face_img = self.detector.draw_bboxes(image)
        face_num = len(b_boxes)
        mask_num = 0
        for box in b_boxes:
            face = image.crop(tuple(box[:4]))
            face = np.array(face)
            # reshape size of the image to fit the model input request
            face = cv2.resize(face, (224, 224),
                              interpolation=cv2.INTER_AREA)
            face = transforms.ToTensor()(face).unsqueeze(0)
            self.mobilenet.eval()

            with torch.no_grad():
                predict_label = self.mobilenet(face).cpu().data.numpy()
            current_class = self.classes[np.argmax(predict_label).item()]

            draw = ImageDraw.Draw(detect_face_img)
            if current_class == "mask":
                mask_num += 1
                draw.text((200, 50), u'yes', 'fuchsia')
            else:
                draw.text((200, 50), u'no', 'fuchsia')

        return detect_face_img, face_num, mask_num
```

- `MobileNetV2.py` 定义了我们网络的结构，是主要的任务部分。其中参数量主要集中在1x1的卷积层，因为每个倒残差层都是先1x1卷积升维，再1x1卷积降维。body中每个层都是由倒残差层堆叠而成，除此之外前面加上纯卷积层，后面加上池化和全连接层。基本大致是按照MobileNet V2的方式建立了一个网络（部分输入输出维度不完全相同，对参数量也有简化）。

```
import torch
import torch.nn as nn
import torchvision

# DW卷积
def Conv3x3BNReLU(in_channels, out_channels, stride, groups):
    return nn.Sequential(
        # stride=2 wh减半，stride=1 wh不变
        nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                  kernel_size=3, stride=stride, padding=1, groups=groups),
        nn.BatchNorm2d(out_channels),
        nn.ReLU6(inplace=True)
    )

# PW卷积
def Conv1x1BNReLU(in_channels, out_channels):
```

```
    return nn.Sequential(
        nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                  kernel_size=1, stride=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU6(inplace=True)
    )

# PW卷积(Linear) 没有使用激活函数
def Conv1x1BN(in_channels, out_channels):
    return nn.Sequential(
        nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                  kernel_size=1, stride=1),
        nn.BatchNorm2d(out_channels)
    )

class InvertedResidual(nn.Module):

    def __init__(self, in_channels, out_channels, expansion_factor, stride):
        super(InvertedResidual, self).__init__()
        self.stride = stride
        self.in_channels = in_channels
        self.out_channels = out_channels
        # expansion_factor 扩展因子 expansion layer的通道大小和输入bottleneck层的通道大小之比
        mid_channels = (in_channels * expansion_factor)

        # 先1x1卷积升维，再1x1卷积降维
        self.bottleneck = nn.Sequential(
            Conv1x1BNReLU(in_channels, mid_channels),
            Conv3x3BNReLU(mid_channels, mid_channels,
                          stride, groups=mid_channels),
            Conv1x1BN(mid_channels, out_channels)
        )

        if self.stride == 1:
            self.shortcut = Conv1x1BN(in_channels, out_channels)

    def forward(self, x):
        out = self.bottleneck(x)
        out = (out + self.shortcut(x)) if self.stride == 1 else out
        return out

class MobileNetV2(nn.Module):

    # classes为分类个数，t为expansion factor
    def __init__(self, num_classes=num_class, t=6):

        super(MobileNetV2, self).__init__()

        # 3 -> 32 groups=1
        self.first_conv = Conv3x3BNReLU(3, 32, 2, groups=1)

        # 32 -> 16 stride=1 wh不变
        self.layer1 = self.make_layer(
            in_channels=32, out_channels=16, stride=1, factor=1, block_num=1)

        # 16 -> 24 stride=2 wh减半
        self.layer2 = self.make_layer(
            in_channels=16, out_channels=24, stride=2, factor=t, block_num=2)

        # 24 -> 32 stride=2 wh减半
        self.layer3 = self.make_layer(
            in_channels=24, out_channels=32, stride=2, factor=t, block_num=3)

        # 32 -> 64 stride=2 wh减半
        self.layer4 = self.make_layer(
            in_channels=32, out_channels=64, stride=2, factor=t, block_num=4)
```

```
        # 64 -> 96 stride=1 wh不变
        self.layer5 = self.make_layer(
            in_channels=64, out_channels=96, stride=1, factor=t, block_num=3)

        # 96 -> 160 stride=2 wh减半
        self.layer6 = self.make_layer(
            in_channels=96, out_channels=160, stride=2, factor=t, block_num=3)

        # 160 -> 320 stride=1 wh不变
        self.layer7 = self.make_layer(
            in_channels=160, out_channels=320, stride=1, factor=t, block_num=1)

        # 320 -> 1280 单纯的升维操作
        self.last_conv = Conv1x1BNReLU(320, 1280)
        self.avgpool = nn.AvgPool2d(kernel_size=7, stride=1)
        self.dropout = nn.Dropout(p=0.2)
        self.linear = nn.Linear(in_features=1280, out_features=num_classes)
        self.init_params()

    # 将一定数量的倒残差层进行堆叠
    def make_layer(self, in_channels, out_channels, stride, factor, block_num):
        layers = []
        layers.append(InvertedResidual(
            in_channels, out_channels, factor, stride))

        for i in range(1, block_num):
            layers.append(InvertedResidual(
                out_channels, out_channels, factor, 1))

        return nn.Sequential(*layers)

    # 初始化权重
    def init_params(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear) or isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    # 前向传播
    def forward(self, x):
        x = self.first_conv(x)  # torch.Size([1, 32, 112, 112])
        x = self.layer1(x)      # torch.Size([1, 16, 112, 112])
        x = self.layer2(x)      # torch.Size([1, 24, 56, 56])
        x = self.layer3(x)      # torch.Size([1, 32, 28, 28])
        x = self.layer4(x)      # torch.Size([1, 64, 14, 14])
        x = self.layer5(x)      # torch.Size([1, 96, 14, 14])
        x = self.layer6(x)      # torch.Size([1, 160, 7, 7])
        x = self.layer7(x)      # torch.Size([1, 320, 7, 7])
        x = self.last_conv(x)   # torch.Size([1, 1280, 7, 7])
        x = self.avgpool(x)     # torch.Size([1, 1280, 1, 1])
        x = x.view(x.size(0), -1)    # torch.Size([1, 1280])
        x = self.dropout(x)
        x = self.linear(x)      # torch.Size([1, 5])
        return x
```

2.2.3 参数调整

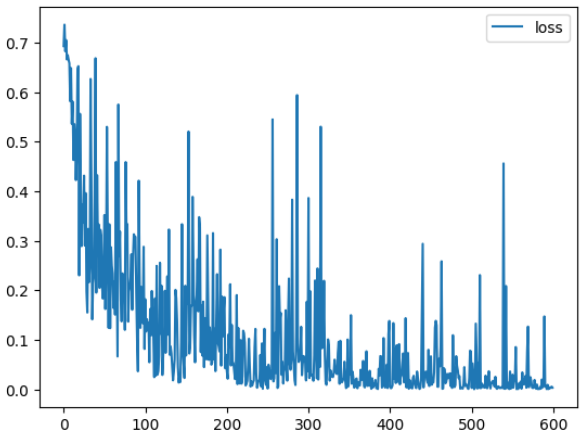
为了与论文方法保持一致，我们调整输入图片大小为`224*224`。取`batch_size=32`，`learning_rate=1e-3`，`epochs=30`。优化器我尝试了使用`Adam`、`SGD`、`RMSprop`，结果相差不大，因为任务相对简单都能在10个epoch左右收敛到loss在0.1以下，所以我选择使用`Adam`。另外对于`patience`我选择设置为4。以下为相关的训练参数配置代码：

```
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(classes=2).to(device)
optimizer = optim.Adamgrad(model.parameters(), lr=1e-3)

# 学习率下降的方式，acc 4 次不下降就改变学习率继续训练，衰减学习率
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                                 'max',
                                                 factor=0.5,
                                                 patience=4)
criterion = nn.CrossEntropyLoss()
```

训练loss结果如下，可以发现后期其loss降到了0.1以下。同时准确度接近0.97。



2.3 测试结果

将训练好的MobileNet V2 模型在mo平台进行测试，结果如下。效果比较好。

3. 总结与体会

- 这是我第一次利用机器学习的知识处理图片信息。图片信息相对于之前的纯数据而言，信息量更多，对于硬件的要求也更高。另外，我在本次实验一开始是使用的本地环境，但是在运行过程中出现了OOM的现象，通过调整网络结构（减少未使用的张量）、清理cuda缓存以及调整batch size，我成功解决了这一问题。这一经历让我对于机器学习中的对训练过程的硬件控制有了更深的了解。
- 本次实验主要是对于网络结构进行学习，而对于参数的设置调整相对比较少（事实上，由于网络足够复杂，各种训练参数对于最终效果的影响不是很明显，loss能够较快的收敛）。通过阅读MobileNet V2的论文以及自己搭建和调整网络架构，我对于pytorch的深度学习框架的使用更加熟练。