计算机视觉—Plant Pathology 2021 图片分类

20354240 帅灿宇

0. 引言

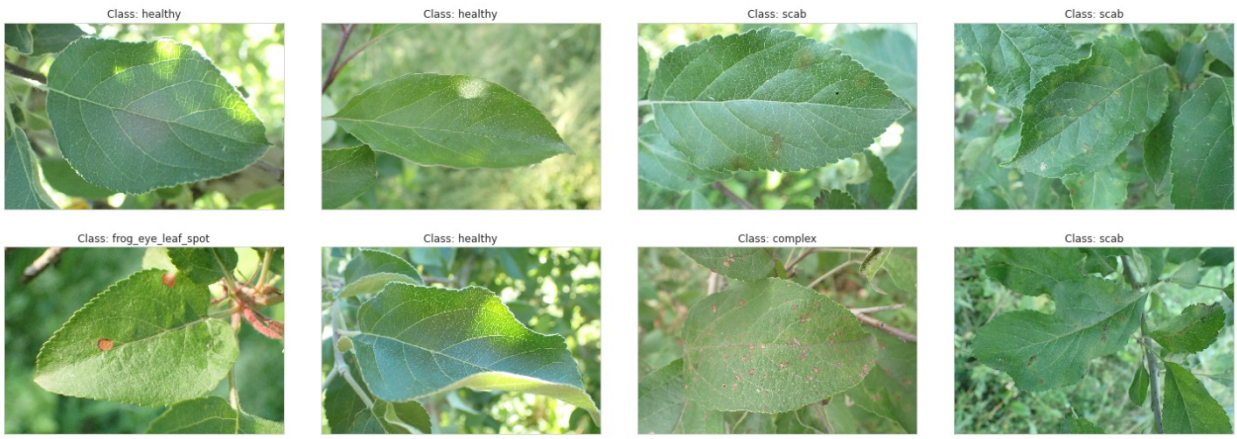
本任务是对 Plant Pathology 2021 数据集进行分类的任务。在该任务中，使用了 ResNet50 网络来进行分类。其中，在处理数据集时，将图片类别将 12 类转换为了 6 类进行分类，对图片标签进行了 Onehot 编码处理，并对图片进行裁剪等数据增强处理，在搭建的网络上进行数据分类。在训练中使用了多分类问题中的 MultiLabelSoftMarginLoss 损失函数，并计算分类的正确率及 F1 分数。

在最终训练出的模型上使用测试集进行验证，最后得出的 F1 分数及正确率为：

Name	Score
F1	0.840889
Accuracy	0.781667

1. 数据处理

Plant Pathology 2021 数据集由 3000 张训练集，600 张验证集及 600 张测试集组成，图片内容为不同树叶状况的分类。



图表 1PlantPathology 2021 数据集

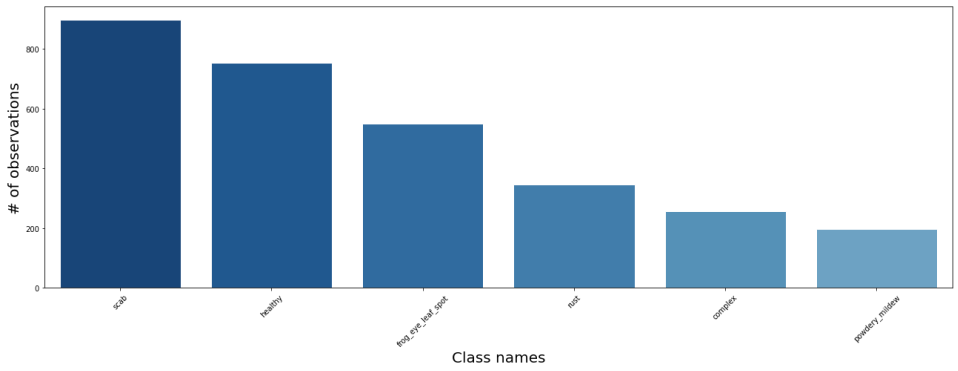
根据树叶不同的健康状况共分为了 12 类别，不同的类别之间有重复，例如 [存在于多个类别中，这会影响模型的训练精度。首先先把类别改为如下 6 类，转换成多标签分类问题，对标签进行 one hot 编码再训练。

图表 2 修改标签类别

修改类别后对训练集数据进行分析，得到各类别的数据个数及比例如下：

disease	count	%
scab	897	30.0
healthy	753	25.0
frog_eye_leaf_spot	550	18.0
rust	347	12.0
complex	256	9.0
powdery_mildew	197	7.0

图表 3 修改为 6 类后训练集数据情况



图表 4 训练集数据分布情况柱状图

对数据集标签进行 **Onehot** 编码：

images	labels	scab	frog_eye_leaf_spot	complex	rust	powdery_mildew	healthy
a8ab965f868fc44c.jpg	healthy	0	0	0	0	0	1
a5d8924f7fad18a0.jpg	healthy	0	0	0	0	0	1
fab3f2b1c0d2a982.jpg	scab	1	0	0	0	0	0
d9b283cd98b19d13.jpg	scab	1	0	0	0	0	0
852979c129dde25d.jpg	frog_eye_leaf_spot	0	1	0	0	0	0
b76b84406eb75545.jpg	healthy	0	0	0	0	0	1
c1a972977e88b49c.jpg	complex	0	0	1	0	0	0
ba3457285a4655e7.jpg	scab	1	0	0	0	0	0
f2d5d2adc18c6986.jpg	scab	1	0	0	0	0	0
e0f996008ddabf0d.jpg	frog_eye_leaf_spot	0	1	0	0	0	0

图表 5Onehot 编码后数据集标签

最后为增强网络的分辨能力，对数据集进行增强及裁剪：

图表 6 数据增强后的图片

2. ResNet50 网络搭建

ResNet50 中的 50 指有 50 个层。和上图一样，本图描述的 ResNet 也分为 5 个阶段。

ResNet50 网络如图：

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图表 7 ResNet 网络结构

如本图所示，ResNet 分为 5 个 stage（阶段），其中 Stage 0 的结构比较简单，可以视其为对 INPUT 的预处理，后 4 个 Stage 都由 Bottleneck 组成，结构较为相似。Stage 1 包含 3 个 Bottleneck，剩下的 3 个 stage 分别包括 4、6、3 个 Bottleneck。

Stage 0

(3,224,224)指输入 INPUT 的通道数(channel)、高(height)和宽(width)，即(C,H,W)。现假设输入的高度和宽度相等，所以用(C,W,W)表示。该 stage 中第 1 层包括 3 个先后操作

1. CONV

CONV 是卷积（Convolution）的缩写，7×7 指卷积核大小，64 指卷积核的数量（即该卷积层输出的通道数），/2 指卷积核的步长为 2。

2. BN

BN 是 Batch Normalization 的缩写，即常说的 BN 层。

3. RELU

RELU 指 ReLU 激活函数。

该 stage 中第 2 层为 MAXPOOL，即最大池化层，其 kernel 大小为 3×3、步长为 2。

(64,56,56)是该 stage 输出的通道数(channel)、高(height)和宽(width)，其中 64 等于该 stage 第 1 层卷积层中卷积核的数量，56 等于 224/2/2（步长为 2 会使输入尺寸减半）。

总体来讲，在 Stage 0 中，形状为(3,224,224)的输入先后经过卷积层、BN 层、ReLU 激活函数、MaxPooling 层得到了形状为(64,56,56)的输出。

Stage 1

Stage 1 的输入的形状为(64,56,56)，输出的形状为(64,56,56)。

Bottleneck

BTNK2

BTNK2 有 2 个可变的参数 C 和 W，即输入的形状(C,W,W)中的 c 和 W。令形状为(C,W,W)的输入为 x，令 BTNK2 左侧的 3 个卷积块（以及相关 BN 和 RELU）为函数 F(x)，两者相加（F(x)+x）后再经过 1 个 ReLU 激活函数，就得到了 BTNK2 的输出，该输出的形状仍为(C,W,W)，即上文所说的 BTNK2 对应输入 x 与输出 F(x)通道数相同的情况。

BTNK1

BTNK1 有 4 个可变的参数 C、W、C1 和 S。与 BTNK2 相比，BTNK1 多了 1 个右侧的卷积层，令其为函数 G(x)。BTNK1 对应了输入 x 与输出 F(x)通道数不同的情况，也正是这

个添加的卷积层将 x 变为 G(x)，起到匹配输入与输出维度差异的作用（G(x)和 F(x)通道数相同），进而可以进行求和 F(x)+G(x)。

简要分析

ResNet50 搭建代码:

```
from torch import nn
class Bottleneck(nn.Module):
    #每个 stage 维度中扩展的倍数
    extention=4
    def __init__(self,inplanes,planes,stride,downsample=None):
        '''

        :param inplanes: 输入 block 的之前的通道数
        :param planes: 在 block 中间处理的时候的通道数
                planes*self.extention:输出的维度
        :param stride:
        :param downsample:
        '''
        super(Bottleneck, self).__init__()

        self.conv1=nn.Conv2d(inplanes,planes,kernel_size=1,stride=stride,bias=False)
        self.bn1=nn.BatchNorm2d(planes)

        self.conv2=nn.Conv2d(planes,planes,kernel_size=3,stride=1,padding=1,bias=False)
        self.bn2=nn.BatchNorm2d(planes)

        self.conv3=nn.Conv2d(planes,planes*self.extention,kernel_size=1,stride=1,bias=False)
        self.bn3=nn.BatchNorm2d(planes*self.extention)

        self.relu=nn.ReLU(inplace=True)

        #判断残差有没有卷积
        self.downsample=downsample
        self.stride=stride

    def forward(self,x):
        #参差数据
        residual=x

        #卷积操作
        out=self.conv1(x)
        out=self.bn1(out)
        out=self.relu(out)

        out=self.conv2(out)
        out=self.bn2(out)
        out=self.relu(out)

        out=self.conv3(out)
        out=self.bn3(out)
        out=self.relu(out)
```

```
        #是否直连（如果 Indentity blobk 就是直连；如果 Conv2 Block 就需要对残差边就行卷积，改变通道数和 size
        if self.downsample is not None:
            residual=self.downsample(x)

        #将残差部分和卷积部分相加
        out+=residual
        out=self.relu(out)

        return out


class ResNet(nn.Module):
    def __init__(self,block,layers,num_class):
        #inplane=当前的 fm 的通道数
        self.inplane=64
        super(ResNet, self).__init__()

        #参数
        self.block=block
        self.layers=layers

        #stem 的网络层
        self.conv1=nn.Conv2d(3,self.inplane,kernel_size=7,stride=2,padding=3,bias=False)
        self.bn1=nn.BatchNorm2d(self.inplane)
        self.relu=nn.ReLU()
        self.maxpool=nn.MaxPool2d(kernel_size=3,stride=2,padding=1)

        #64,128,256,512 指的是扩大 4 倍之前的维度，即 Identity Block 中间的维度
        self.stage1=self.make_layer(self.block,64,layers[0],stride=1)
        self.stage2=self.make_layer(self.block,128,layers[1],stride=2)
        self.stage3=self.make_layer(self.block,256,layers[2],stride=2)
        self.stage4=self.make_layer(self.block,512,layers[3],stride=2)

        #后续的网络
        self.avgpool=nn.AvgPool2d(7)
        self.fc=nn.Linear(512*block.extention,num_class)

    def forward(self,x):
        #stem 部分：conv+bn+maxpool
        out=self.cuda().conv1(x)
        out=self.bn1(out)
        out=self.relu(out)
        out=self.maxpool(out)

        #block 部分
        out=self.stage1(out)
        out=self.stage2(out)
        out=self.stage3(out)
        out=self.stage4(out)

        #分类
        out=self.avgpool(out)
        out=torch.flatten(out,1)
        out=self.fc(out)
```

```
        return out

    def make_layer(self,block,plane,block_num,stride=1):
        '''
        :param block: block 模板
        :param plane: 每个模块中间运算的维度，一般等于输出维度/4
        :param block_num: 重复次数
        :param stride: 步长
        :return:
        '''
        block_list=[]
        #先计算要不要加 downsample
        downsample=None
        if(stride!=1 or self.inplane!=plane*block.extention):
            downsample=nn.Sequential(
                nn.Conv2d(self.inplane,plane*block.extention,stride=stride,kernel_size=1,bias=False),
                nn.BatchNorm2d(plane*block.extention)
            )

        # Conv Block 输入和输出的维度（通道数和 size）是不一样的，所以不能连续串联，他的作用是改变网络的维度
        # Identity Block 输入维度和输出（通道数和 size）相同，可以直接串联，用于加深网络
        #Conv_block
        conv_block=block(self.inplane,plane,stride=stride,downsample=downsample)
        block_list.append(conv_block)
        self.inplane=plane*block.extention

        #Identity Block
        for i in range(1,block_num):
            block_list.append(block(self.inplane,plane,stride=1))

        return nn.Sequential(*block_list)
```

3. 模型训练

模型训练部分使用多分类问题中的 MultiLabelSoftMarginLoss 损失函数，及 Adam 优化器进行训练。

训练代码：

```
class MetricMonitor:
    def __init__(self):
        self.reset()

    def reset(self):
        self.losses = []
        self.accuracies = []
        self.scores = []
        self.metrics = dict({
            'loss': self.losses,
            'acc': self.accuracies,
```

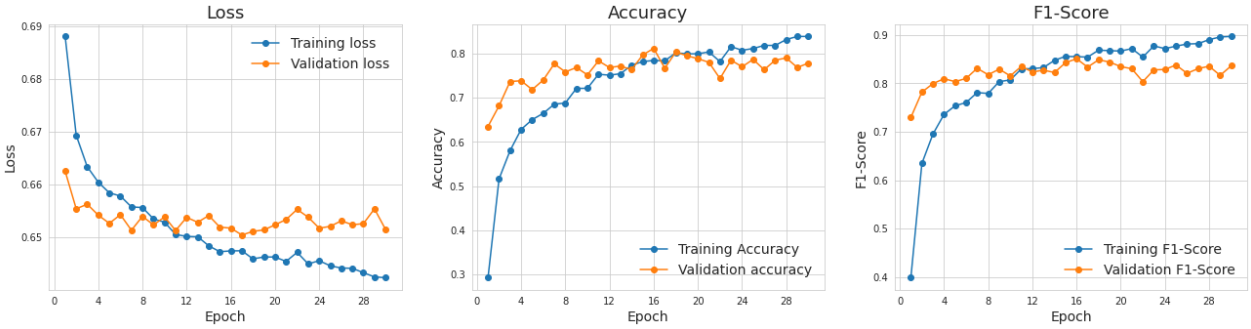
```
            'f1': self.scores
        })

    def update(self, metric_name, value):
        self.metrics[metric_name] += [value]

train_monitor = MetricMonitor()
test_monitor = MetricMonitor()

from sklearn.metrics import f1_score, accuracy_score

def get_metrics(
    y_pred_proba,
    y_test,
    labels=Config.CLASSES) -> None:
    """
    """
    threshold = 0.6
    y_pred = np.where(y_pred_proba > threshold, 1, 0)

    y1 = y_pred.round().astype(float)
    y2 = y_test.round().astype(float)

    f1 = f1_score(y1, y2, average='micro')
    acc = accuracy_score(y1, y2, normalize=True)

    return acc,f1

def to_numpy(tensor):
    """Auxiliary function to convert tensors into numpy arrays
    """
return tensor.detach().cpu().numpy() if tensor.requires_grad else tensor.cpu().numpy()


def training_loop(
    dataloader,
    model,
    loss_fn,
    optimizer,
    epoch,
    monitor = MetricMonitor(),
    is_train=True
) -> None:
    """
    """
    size = len(dataloader.dataset)

    loss_val = 0
    accuracy = 0
    f1score = 0

    if is_train:
        model.train()
    else:
        model.eval()

    stream = tqdm(dataloader)
```

```
    for batch, (X, y) in enumerate(stream, start=1):
        X = X.cuda()
        y = y.cuda()

        # compute prediction and loss
        pred_prob = model(X)
        loss = loss_fn(pred_prob, y)

        if is_train:
            # backpropagation
            optimizer.zero_grad()

            loss.backward()
            optimizer.step()

        loss_val += loss.item()
        acc, f1 = get_metrics(to_numpy(pred_prob), to_numpy(y))

        accuracy += acc
        f1score += f1

        phase = 'Train' if is_train else 'Val'
        stream.set_description(
            f'Epoch {epoch:3d}/{Config.N_EPOCH} - {phase} - Loss: {loss_val/batch:.4f}, ' +
            f'Acc: {accuracy/batch:.4f}, F1: {f1score/batch:.4f}'
        )

    monitor.update('loss', loss_val/batch)
    monitor.update('acc', accuracy/batch)
monitor.update('f1', f1score/batch)
# initialize the loss function
loss_fn = nn.MultiLabelSoftMarginLoss()

optimizer = torch.optim.Adam(
    model.parameters(),
    lr=Config.LEARNING_RATE
)

# %%time
for epoch in range(1, Config.N_EPOCH + 1):
    # training loop
    training_loop(
        train_loader,
        model,
        loss_fn,
        optimizer,
        epoch,
        train_monitor,
        is_train=True
    )

    # validation loop
    training_loop(
        valid_loader,
        model,
        loss_fn,
        optimizer,
```

```
        epoch,
        test_monitor,
        is_train=False
    )
```

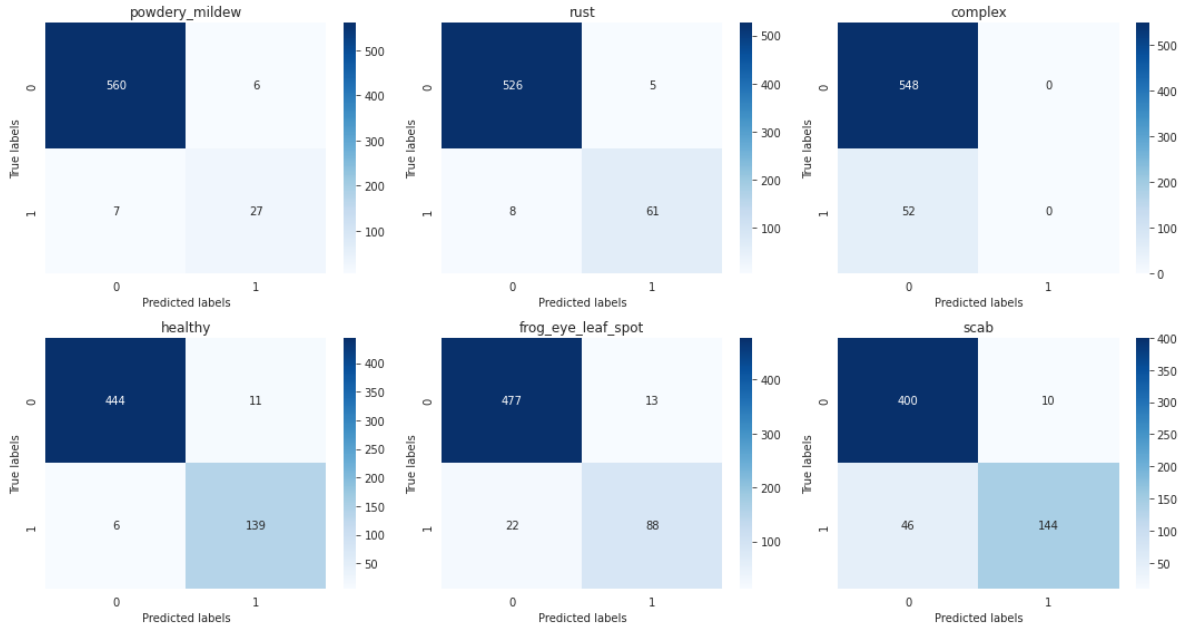
4. 训练结构

训练过程中损失函数 loss、正确率及 F1 分数岁迭代次数的变化曲线图如下：



图表 8 损失函数、正确率及 F1 分数的变化折线图

训练结果中，各类别的分类混淆矩阵如下：



图表 9 各类别分类结果混淆矩阵

5. 结果验证

在测试集图片上验证模型，得出正确率及 F1 分数如下：

Name	Score
F1	0.840889
Accuracy	0.781667