《数据科学导引》期末project报告

薛宇燕 1600017814

张玉婷 1700017858

选题介绍

我们此次project选择的是「花朵分类」问题，采用`ResNet101`神经网络模型进行深度学习训练，具体代码利用`pytorch`里的相关代码包进行实现。

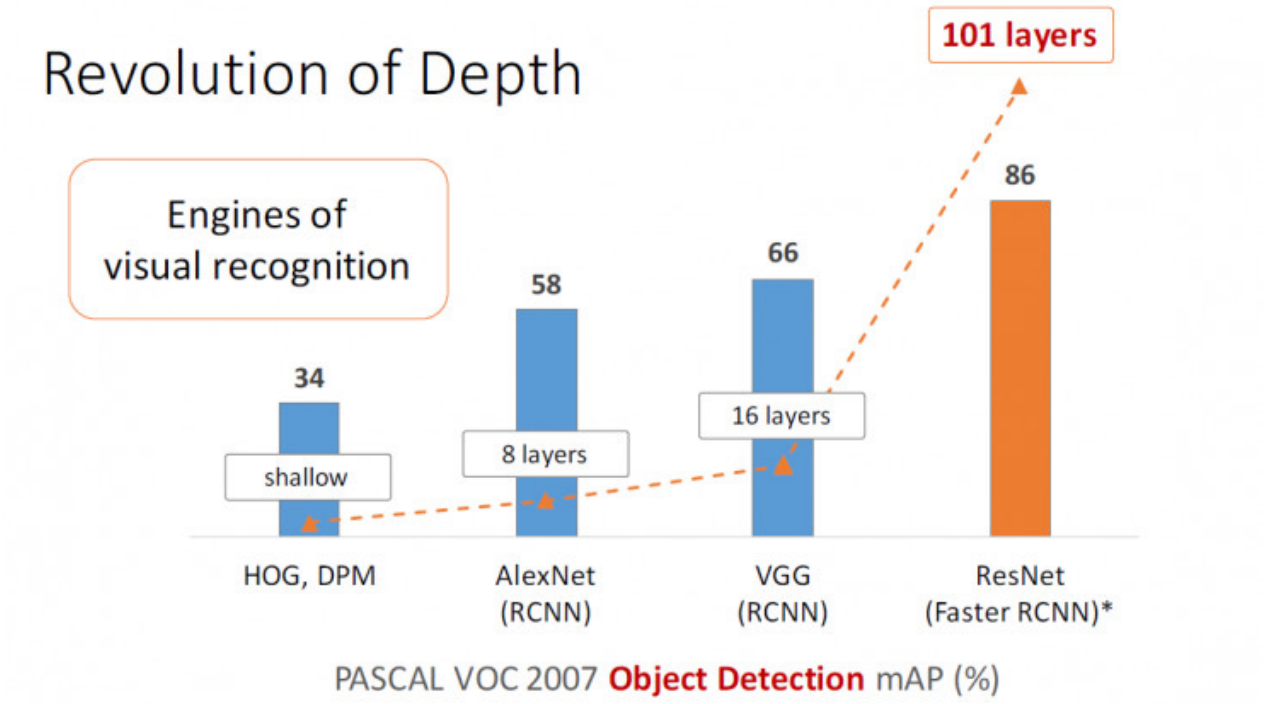
我们的`kaggle`小组名称为`TryTry`，排名2/4，成绩为0.93700。

小组分工

- 薛宇燕：选定模型，完成预处理、训练参数设置、训练、输出结果等部分的代码，训练模型，撰写部分实验报告。
- 张玉婷：理解ResNet原理，完成定义模型部分代码，撰写部分实验报告。

ResNet网络结构介绍

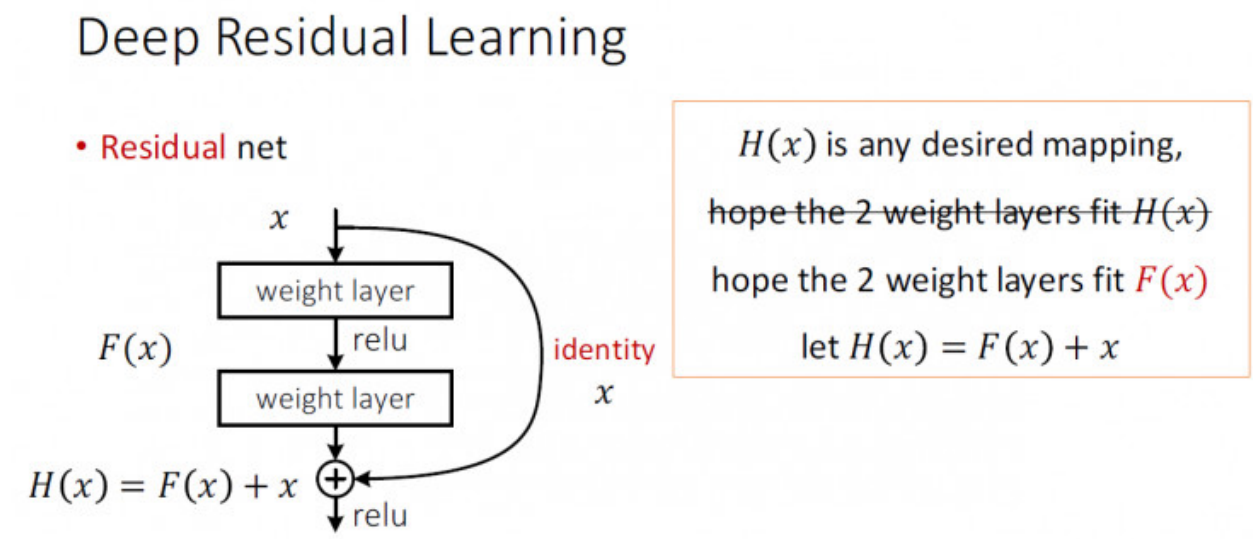
众所周知，在计算机视觉里，特征的"等级"随增网络深度的加深而变高。网络的深度是实现好的效果的重要因素。然而梯度弥散/爆炸成为训练深层次的网络的障碍，导致无法收敛。有一些方法可以弥补，如归一初始化，各层输入归一化，使得可以收敛的网络的深度提升为原来的十倍。然而，虽然收敛了，但网络却开始退化了，即增加网络层数却导致更大的误差。



正如上图所示，`ResNet`很好地权衡了这两方面的问题，使得我们得到一个“非常深”的深度网络的同时获得很高的准确率。尤其在图像识别领域，这一模型具有很好的表现。所以这次project我们采用了这一方法。

它的具体原理如下：

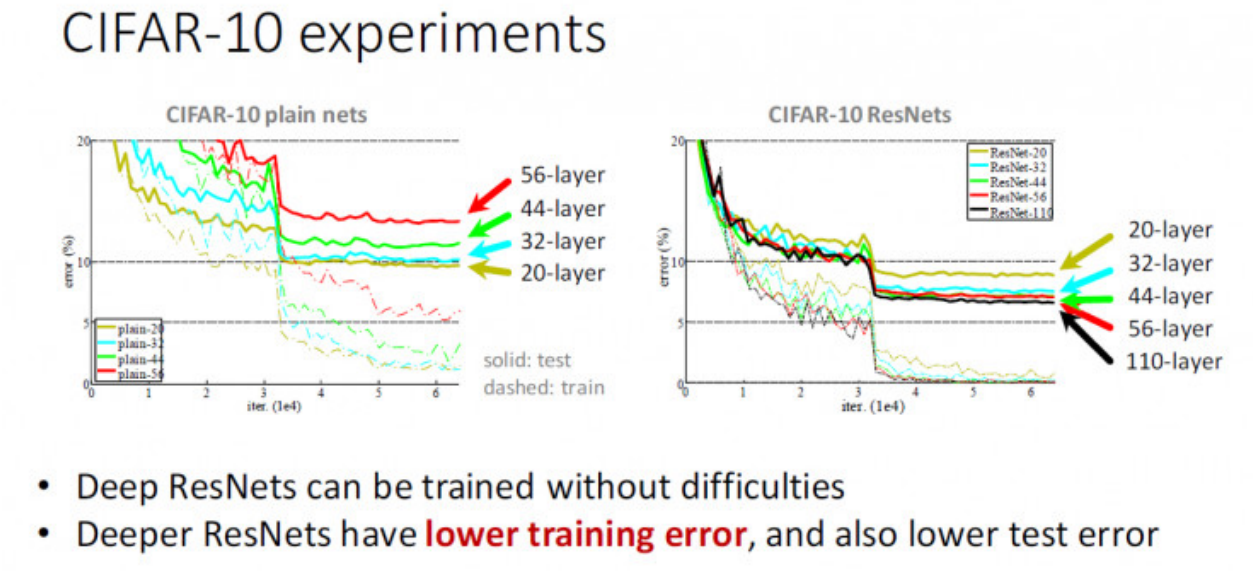
> **H(x)**是任意一种理想的映射
>
> 希望第2类权重层能够与**F(x)**拟合
>
> 使**H(x) = F(x) + x**



通过在一个浅层网络基础上叠加**y=x**的层（称`identity mappings`，恒等映射），可以让网络随深度增加而不退化。这反映了多层非线性网络无法逼近恒等映射网络。

但是，不退化不是我们的目的，我们希望有更好性能的网络。 `ResNet`学习的是残差函数**F(x) = H(x) - x**, 这里如果**F(x) = 0**, 那么就是上面提到的恒等映射。事实上，`ResNet`是`shortcut connections`的在connections是在恒等映射下的特殊情况，它没有引入额外的参数和计算复杂度。 假如优化目标函数是逼近一个恒等映射, 而不是零映射， 那么学习找到对恒等映射的扰动会比重新学习一个映射函数要容易。其实残差函数一般会有较小的响应波动，表明恒等映射是一个合理的预处理。

如下图所示，残差网络解决了退化的问题，在训练集和校验集上，都证明了的更深的网络错误率越小。



具体代码实现

本次project的具体代码分为`main.py`和`mymodel.py`两个文件，`main.py`里面主要写了数据加载及预处理以及训练参数的设置和正式训练调用模型的过程，`mymodel.py`里面主要为修改已有的`pytorch`实现的`ResNet`模型里的参数以及优化器的设置。

main.py

首先定义了整个文件所在的路径，字符串形式方便后续直接相加的操作，还给出了预训练的模型路径，预训练模型直接从`pytorch`官方的`ResNet`下载即可。`base_model`指定训练时选用的`ResNet`种类，本project选用`ResNet101`，也尝试了`ResNet50`，但发现训练效果不如`ResNet101`，精确度只有0.87401，所以最终还是坚持使用`ResNet101`。

然后是数据预处理部分，定义`data_loader`函数以及其相关的`mydataset`类，方便加载图片。`data_loader`里提供了训练集、验证集以及测试集，虽然由于已经选定了`ResNet101`因此不需要验证模型类别，但是为了保证深度学习的数据分类格式，本类别依旧保留，其中图片和训练集一样。`data_loader`函数负责预处理图片成后续训练可以直接使用的格式，采用`torchvision.transforms`包。训练集将图片随机旋转$[-30^\circ, 30^\circ]$、随机部分切割为224^3的格式、随机水平翻转、转换到`Tensor`格式，然后再正则化。验证集将图片调整为256^3的格式、从中心分割、转换到`Tensor`格式，然后再正则化。测试机的处理方法与验证集相同。最后返回三个图片集的`loader`。

核心代码块如下：

```
def data_loaders(data_dir):
...
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456,
0.406],
                                                            [0.229, 0.224,
0.225])])
...
return trainloader, validloader, testloader, train_data, valid_data, test_data
```

接下来定义训练和优化器的参数。使用GPU训练30个epoch，使用`Adam`优化器，`learning rate`为0.001，衰减率为每2步减半。然后加载数据以及调用`mymodel.py`里修改了部分参数的`ResNet`模型，使用之前下载好的预训练好的模型以及`cuda`处理器。

核心代码块如下：

```
epochs = 0
gpu = True
optimizer = 'Adam'
lr = 1e-3
lr_decay = 0.5
lr_decay_step = 2
criterion = 'NLLLoss'
```

下面的`for`循环是开始训练的过程。对每个`batch`，将训练集里的刚才已经处理好的数据及其标签放入处理器内，然后进行梯度下降法进行迭代，将梯度初始化为零，然后前向传播求出预测的值，然后求出`loss`值，最后反向传播求梯度并更新参数，均使用`pytorch`包里面的已经写好的函数。每隔100张图片，用验证集进行检验，并输出`training loss`、`valid loss`和`valid accuracy`。然后将此时训练的参数记录在一个字典内，并保存训练的模型。

核心代码块如下：

```
model.train()
running_loss = 0
lr_scheduler.step()
for ii, (inputs, labels) in enumerate(trainloader):
    steps += 1
    inputs, labels = inputs.to(processor), labels.to(processor)
    optimizer.zero_grad()
    outputs = model.forward(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
```

训练结束后，对测试集里的图片进行类别预测，使用`python`里自带的写`csv`文件的包，将预测结果写到`csv`文件里并输出保存。

mymodel.py

首先搭建`ResNet101`网络。

我们取`ResNet101`预训练好的参数，始终锁定`conv1`。等到loss不再变化时，带`ResNet101`中的参数一起训练。这样做可以使网络具有更好的稳定性。因为，`ResNet101`预训练的参数是比较好的，可以使训练刚开始就获得一个较高的准确率，大大降低了训练的难度，缩短了训练的时间。但是针对不同的问题也要有针对的迁移。所以，在loss稳定时，我们再连同`ResNet101`的参数一起训练，可以得到针对此问题更优的参数，使得准确率进一步提升。

```
self.conv1 = resnet.conv1  # H/2
    self.bn1 = resnet.bn1
    self.relu = resnet.relu
    self.maxpool = resnet.maxpool  # H/4

    self.encoder1 = resnet.layer1  # H/4
    self.encoder2 = resnet.layer2  # H/8
    self.encoder3 = resnet.layer3  # H/16
    self.encoder4 = resnet.layer4  # H/32

    self.avgpool = resnet.avgpool
```

接下来，我们接入自己的分类器。

将损失函数定义为交叉熵。因为这样的损失函数整体呈单调性，loss越大，梯度越大，便于梯度下降反向传播，利于优化，适用于本次的多分类问题。最终输出一个五维向量。

```
self.classifier = nn.Sequential(OrderedDict([
        ('dropout', nn.Dropout(0.25)),
        ('inputs', nn.Linear(512*4, 5))
         ]))
```

另外，我们前后采用两种不同的优化方式。

最开始的训练中，先锁住前面参数，使用`Adam`优化方式。`Adam`学习率使用了衰减。等到效果比较好了，解开除`conv1`层以外的所有参数，使用`Adam`继续训练。最后换为`momentum-SGD`训练两轮，学习率为1e-8。

```
if arguments['optimizer'] == 'Adam':
    optimizer = optim.Adam(model.parameters(), lr, weight_decay=0)
else:
    optimizer = optim.SGD(model.parameters(), lr, momentum=0.8,
weight_decay=0)
```

实验结果

由于计算资源有限（借用朋友的GPU进行训练），本模型训练了30个epoch，最后在`kaggle`上显示的成绩为0.93700，即精确度为93.70%。

实验反思

在得到0.93700的结果之后，我们很难再在此基础上有所提高。这应该有两方面的原因。一方面，我们网络结构较为简单，而测试集中的一些图片有很多干扰的区域，造成了误分类。另一方面，我们尝试过增加网络的层数来获得更精细的结果。但事实上，由于训练集较小，增加网络层数参数训练效果不好，容易造成过拟合。所以这种方法被舍弃。如果要进一步提高准确率，可能要在数据预处理上下更多的功夫，专门来应对一些对抗性样本。