理解

1. 本题类型：一个有监督学习的分类问题。
2. 本题考察能力：考察：机器学习基本知识、根据数据集的特点进行预处理、搭建神经网络、调参、微调的能力。由于数据集都是工作室的佬们直接找好的，对于收集数据的能力考察得不多。
3. 我主要干了什么：自己搭了一个ResNet，调了 `torchvision.models` 中预训练的ResNet18、VGG13，微调，调参，训练。相当于做到了3.要求（进阶）的c题（选做），但没做完

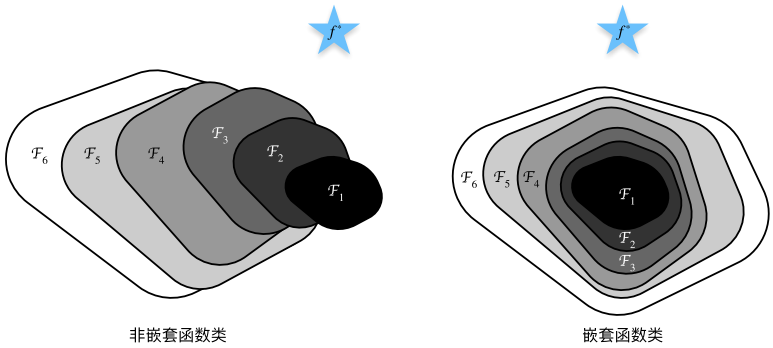
学习笔记（理论）

函数类

首假设有一类特定的神经网络架构$\mathcal{F}$，它包括学习速率和其他超参数设置。
对于所有$f \in \mathcal{F}$，存在一些参数集（例如权重和偏置），这些参数可以通过在合适的数据集上进行训练而获得。
现在假设f^*是我们真正想要找到的函数，如果是$f^* \in \mathcal{F}$，那我们可以轻而易举的训练得到它，但通常我们不会那么幸运。
相反，我们将尝试找到一个函数$f^*_\mathcal{F}$，这是我们在$\mathcal{F}$中的最佳选择。
例如，给定一个具有$\mathbf{X}$特性和$\mathbf{y}$标签的数据集，我们可以尝试通过解决以下优化问题来找到它：

$$f^*_\mathcal{F} := \operatorname{argmin}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$

那么，怎样得到更近似真正f^*的函数呢?
唯一合理的可能性是，我们需要设计一个更强大的架构$\mathcal{F}'$。
换句话说，我们预计$f^*_{\mathcal{F}'}$比$f^*_\mathcal{F}$“更近似”。
然而，如果$\mathcal{F} \not\subseteq \mathcal{F}'$，则无法保证新的体系“更近似”。
事实上，$f^*_{\mathcal{F}'}$可能更糟：
对于非嵌套函数（non-nested function）类，较复杂的函数类并不总是向“真”函数f^*靠拢（复杂度由$\mathcal{F}_1$向$\mathcal{F}_6$递增）。
虽然$\mathcal{F}_3$比$\mathcal{F}_1$更接近f^*，但$\mathcal{F}_6$却离的更远了。
对于右侧的嵌套函数（nested function）类$\mathcal{F}_1 \subseteq \ldots \subseteq \mathcal{F}_6$，我们可以避免上述问题。



非嵌套函数类　　嵌套函数类

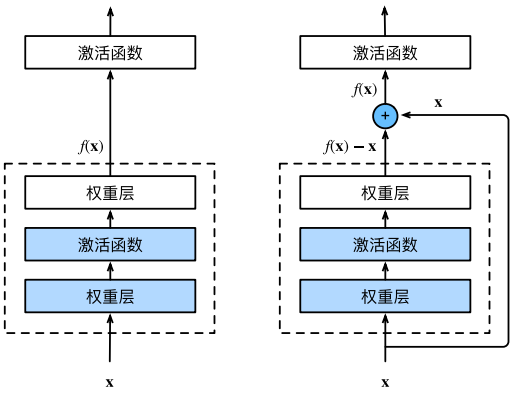
因此，只有当较复杂的函数类包含较小的函数类时，我们才能确保提高它们的性能。
对于深度神经网络，如果我们能将新添加的层训练成*恒等映射*（identity function）$f(\mathbf{x}) = \mathbf{x}$，新模型和原模型将同样有效。
同时，由于新模型可能得出更优的解来拟合训练数据集，因此添加层似乎更容易降低训练误差。

针对这一问题，ResNet横空出世。
残差网络的核心思想是：每个附加层都应该更容易地包含原始函数作为其元素之一。*残差块*（residual blocks）诞生了，这个设计对如何建立深层神经网络产生了深远的影响。

残差块

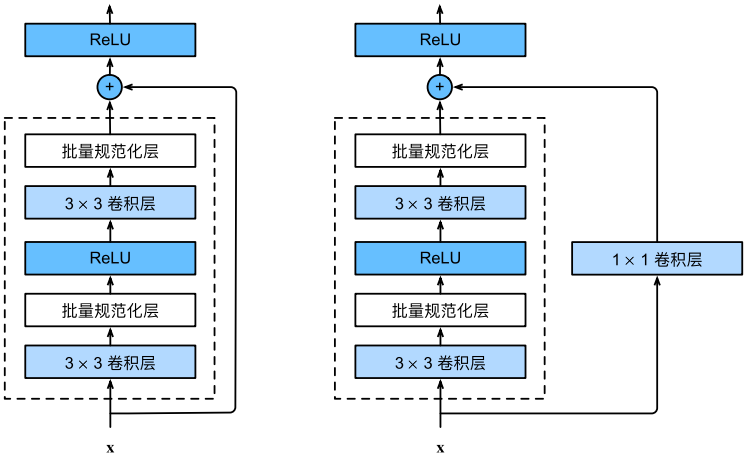
我们聚焦于神经网络局部：如图所示，假设我们的原始输入为x，而希望学出的理想映射为$f(\mathbf{x})$（作为激活函数的输入）。
左图虚线框中的部分需要直接拟合出该映射$f(\mathbf{x})$，而右图虚线框中的部分则需要拟合出残差映射$f(\mathbf{x}) - \mathbf{x}$。
残差映射在现实中往往更容易优化。
以本节开头提到的恒等映射作为我们希望学出的理想映射$f(\mathbf{x})$，我们只需将右图虚线框内上方的加权运算（如仿射）的权重和偏置参数设成0，那么$f(\mathbf{x})$即为恒等映射。
实际中，当理想映射$f(\mathbf{x})$极接近于恒等映射时，残差映射也易于捕捉恒等映射的细微波动。

右图是ResNet的基础架构--*残差块*（residual block）。
在残差块中，输入可通过跨层数据线路更快地向前传播。



ResNet沿用了VGG完整的3×3卷积层设计。
残差块里首先有2个有相同输出通道数的3×3卷积层。
每个卷积层后接一个批量规范化层和ReLU激活函数。
然后我们通过跨层数据通路，跳过这2个卷积运算，将输入直接加在最后的ReLU激活函数前。
这样的设计要求2个卷积层的输出与输入形状一样，从而使它们可以相加。
如果想改变通道数，就需要引入一个额外的1×1卷积层来将输入变换成需要的形状后再做相加运算。

有两种类型的Residual块：
一种是当 `use_1x1conv=False` 时，应用ReLU非线性函数之前，将输入添加到输出。
另一种是当 `use_1x1conv=True` 时，添加通过1×1卷积调整通道和分辨率。



下面我们来查看**输入和输出形状一致**的情况。

ResNet的前两层跟之前介绍的GoogLeNet中的一样：
在输出通道数为64、步幅为2的7×7卷积层后，接步幅为2的3×3的最大汇聚层。
不同之处在于ResNet每个卷积层后增加了批量规范化层。

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.BatchNorm2d(64), nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet在后面接了4个由Inception块组成的模块。
ResNet则使用4个由残差块组成的模块，每个模块使用若干个同样输出通道数的残差块。
第一个模块的通道数同输入通道数一致。
由于之前已经使用了步幅为2的最大汇聚层，所以无须减小高和宽。
之后的每个模块在第一个残差块里将上一个模块的通道数翻倍，并将高和宽减半。
下面我们来实现这个模块。注意，我们对第一个模块做了特别处理。

```
def resnet_block(input_channels, num_channels, num_residuals,
                 first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(input_channels, num_channels,
                                use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels))
    return blk
```

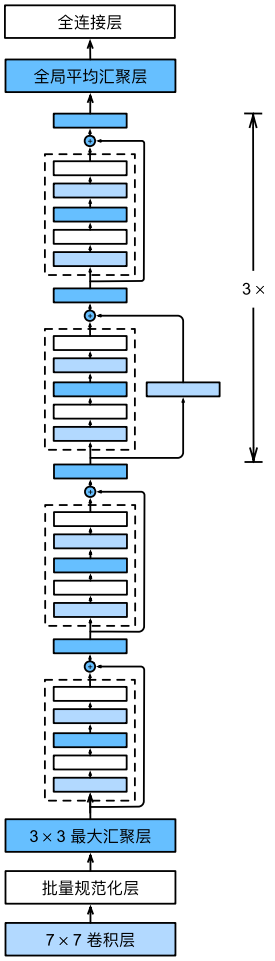
接着在ResNet加入所有残差块，这里每个模块使用2个残差块。

```
b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))
```

最后，与GoogLeNet一样，在ResNet中加入全局平均汇聚层，以及全连接层输出。

```
net = nn.Sequential(b1, b2, b3, b4, b5,
                    nn.AdaptiveAvgPool2d((1,1)),
                    nn.Flatten(), nn.Linear(512, 10))
```

每个模块有4个卷积层（不包括恒等映射的1×1卷积层）。
加上第一个7×7卷积层和最后一个全连接层，共有18层。
因此，这种模型通常被称为ResNet-18。
通过配置不同的通道数和模块里的残差块数可以得到不同的ResNet模型，例如更深的含152层的ResNet-152。
虽然ResNet的主体架构跟GoogLeNet类似，但ResNet架构更简单，修改也更方便。这些因素都导致了ResNet迅速被广泛使用。
下图描述了完整的ResNet-18。



ResNet中不同模块的输入形状的如何变化：分辨率降低，通道数量增加，直到全局平均池化层聚集所有特征。

以下是ChatGPT对残差神经网络的形象讲解

想象一下你正在用乐高积木搭建一座非常高的塔。你搭建得越高，就越担心塔会倒塌。但是，如果你添加的每一个新积木都能让塔更加坚固，而不仅仅是更高呢？ResNet的工作原理有点像这样。

在传统的神经网络中，每一层都试图从前一层传下来的信息中学到新东西。随着你增加更多的层，网络变得更深，理论上应该帮助它学得更好。但实际上，太多的层反而可能使网络的性能变差，有点像我们的塔变得更加摇摇欲坠。这个问题被称为梯度消失问题，即每一层应该学到的课程在途中丢失了。

ResNet引入了一个改变游戏规则的解决方案：快捷方式。这些快捷方式允许层跳过一些其他层，直接向下传递信息。就像一些乐高积木能够神奇地加固塔，不仅仅是在顶部，而是一直到底部。这有助于解决梯度消失问题，因为学习可以跳过那些不添加太多信息的问题层。

机器学习中，科学家们总是对解释模型的结果乐此不疲。以下是我认为ResNet优点的阐释：

1. ResNet最大的特点在于“可选”，这个可选是对于模型复杂度、推理过程的各个阶段的，就是它为了让loss最小，可以自己选择：我再这一阶段，是直接跳到下一阶段呢；还是要先提取一次特征，再到下一阶段呢？这个特点有两个主要的好处：
   i. 若不用Residual块，常常面临这样一个问题：loss function对靠近输出端的参数的偏导数，往往比对靠近数据端的参数的偏导数要大很多。假如我们调大学习率，就容易梯度爆炸。假如我们调小学习率，就容易梯度消失。Residual块可以让这两种参数的偏导之间的差距减小，从而解决梯度爆炸、消失问题。
   ii. 在相同深度下，ResNet比VGG可以训练得更快，准确率更高。因为ResNet是“有选择”的，而VGG“莫得选择”。
2. ResNet还有一个特点：最后用的是一个全局平均池化层+一个全连接层；而VGG用的是三个全连接层。这样，ResNet的参数要少很多，且训练更快。

流程记录

1. 数据集的特点：
   样本数量非常小，直接用训练集训练好一个模型可能比较困难。在测试之前可以预见：用torchvision中预训练好的ResNet18模型，比自己搭一个神经网络来训练精度高一些。

```
#实例化
net = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
num_ftrs = net.fc.in_features
net.fc = nn.Linear(num_ftrs, out_features: 10)
```

```
#分开特征提取层的参数、前面的参数
output_id = list(map(id, net.fc.parameters()))#特征提取层的参数的地址
feature_params = filter(lambda p: id(p) not in output_id, net.parameters())
#以下为超参数：
lr, num_epochs, weight_decay=0.001,10,0.001
optimizer = optim.SGD( params: [{'params': feature_params},
                                {'params': net.fc.parameters(), 'lr': lr * 10}],
                       lr=lr, weight_decay=weight_decay)
```

2. 预处理：
   预处理主要工作是数据增广。由于单个图片实在太小，我们用自己搭的、预训练的神经网络来训练时，我采取的增广的办法是不同的。
   i. 自己的ResNet&VGG13:（以下操作均针对训练集）
      a. 不做随机切割 `transforms.RandomResizedCrop(28)` 、缩放 `transforms.Resize()`
      b. 只用水平翻转 `transforms.RandomHorizontalFlip()` （考虑到有轮船、货车这样的上下翻转后就会很奇怪的）
   ii. 预训练的ResNet18:
      a. 对于训练集：先随机切割 `transforms.RandomResizedCrop(28)` ，取其中的28×28的部分；缩放 `transforms.Resize(224)` ，把图片的高宽放大，以适应预训练模型；用水平翻转 `transforms.RandomHorizontalFlip()`
      b. 对于测试集：先用 `transforms.CenterCrop(28)` 取其中间的28×28的部分，再用 `transforms.Resize(224)` 放大为224×224的图片，来适应预训练模型。

```python
def get_set(size=224):
    train_dir = "../data/train"
    test_dir = "../data/test"
    mean = [0.485, 0.456, 0.406]
    std = [0.229, 0.224, 0.225]
    if(size==224):
        train_augs = transforms.Compose([
            transforms.RandomResizedCrop(28),
            transforms.Resize(size),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ])
        test_augs = transforms.Compose([
            transforms.CenterCrop(28),
            transforms.Resize(size),
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ])
    else:
        train_augs = transforms.Compose([
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ])
        test_augs = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ])
```

3. 自己搭建和调试残差神经网络：

   考虑用经典的ResNet18架构。由于此训练集比较小，考虑后来再通过各种方法降低模型复杂度。但是后来，我试了减少层数、尝试减小通道数，反而发现了test accuracy减小了！最后用回了经典的ResNet18，确实发现是最好的。现在想来减少层数其实是反ResNet的设计初衷的。Residual块的就是让模型可以选择：在哪些阶段该复杂，哪些阶段该简单。

```python
class Residual(nn.Module):#Residual
    def __init__(self, input_channels, num_channels,
                 use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels,
                               kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels,
                               kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                   kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

```python
def resnet_block(input_channels, num_channels, num_residuals,first_block=False): #residual块
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block: #意思是：不是第一个块，都需要通道数翻倍、高宽减半。
            blk.append(Residual(input_channels, num_channels,
                               use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels)) #意思是：第一个块不需要通道数翻倍、高宽减半。
    return blk


b1 = nn.Sequential(nn.Conv2d( in_channels: 3, out_channels: 64, kernel_size=7, stride=2, padding=2),
                   nn.BatchNorm2d(64), nn.ReLU(),
                   nn.MaxPool2d(kernel_size=2, stride=2, padding=1))
b2 = nn.Sequential(*resnet_block( input_channels: 64, num_channels: 64, num_residuals: 2, first_block=True))
b3 = nn.Sequential(*resnet_block( input_channels: 64, num_channels: 128, num_residuals: 2))
b4 = nn.Sequential(*resnet_block( input_channels: 128, num_channels: 256, num_residuals: 2))
b5 = nn.Sequential(*resnet_block( input_channels: 256, num_channels: 512, num_residuals: 2))

net = nn.Sequential(b1, b2, b3, b4, b5,
                    nn.AdaptiveAvgPool2d((1,1)),
                    nn.Flatten(), nn.Linear( in_features: 512, out_features: 10))

train_set, test_set = get_set(size=32)
batch_size = 32
train_iter = DataLoader(train_set, batch_size=batch_size, shuffle=True)
test_iter = DataLoader(test_set, batch_size=batch_size)
```

4. 调参：
   i. 学习率：每次训练后，记得把参数保存下来，下次调小学习率，在此基础上训练。有两种在训练中调整学习率的办法：
      a. 指数函数衰退：指定经过多少个epoch后要减少学习率，以及学习率降低为原来的多少。
      b. 余弦退火：把区间[1,num_epochs+1]映射到区间[0,π/2]上，对于每轮epoch：取其次数的映射的余弦值，乘以学习率，作为新的学习率。
   ii. 批量大小：设为32

iii. 权重衰退：都设为1e-3

iv. 优化算法：全用SGD。本来用了一次Adam，后来放弃了这次训练的参数，Adam也没用了。

```
batch_size = 32
lr,num_epochs = 0.01,10
#以下optimizer是我自己搭建的ResNet的optimizer
optimizer = optim.SGD(net.parameters(), lr=lr, weight_decay=0.001)
#以下optimizer用于预训练的ResNet18：
optimizer = optim.SGD([{'params': feature_params},
                       {'params': net.fc.parameters(), 'lr': lr * 10}],
                      lr=lr, weight_decay=weight_decay)
#以下optimizer用于预训练的VGG13，本质上和上面一个差别不大：
#下一行若取消注释#{'params': net.features.parameters()},的话，就要训练特征提取层。
optimizer=optim.SGD([#{'params': net.features.parameters()},
                       {'params': net.classifier.parameters(), 'lr': lr * 10}],lr=lr,weight_decay=1e-3)
#下面是两种scheduler
scheduler=torch.optim.lr_scheduler.StepLR(optimizer,step_size,gamma)
scheduler=torch.optim.lr_scheduler.CosineAnnealingLR(optimizer = optimizer,T_max =  num_epochs)
```

5. 微调：

i. ResNet18：

```
#实例化
net = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
num_ftrs = net.fc.in_features
net.fc = nn.Linear(num_ftrs, out_features: 10)
```

```
#分开特征提取层的参数、前面的参数
output_id = list(map(id, net.fc.parameters()))#特征提取层的参数的地址
feature_params = filter(lambda p: id(p) not in output_id, net.parameters())
#以下为超参数：
lr, num_epochs, weight_decay=0.001,10,0.001
optimizer = optim.SGD( params: [{'params': feature_params},
                        {'params': net.fc.parameters(), 'lr': lr * 10}],
                       lr=lr, weight_decay=weight_decay)
```

ii. VGG13：

```
net=models.vgg13_bn(weights=models.VGG13_BN_Weights.DEFAULT)
for param in net.features.parameters():
    param.requires_grad_(False)
net.classifier[6]=nn.Linear( in_features: 4096, out_features: 10, bias: True)
```

6. 很大的小问题

第一次自己搭建比线性回归更复杂的深度学习框架，由于缺乏经验，最耗时、最棘手的问题不是模型选择、调参问题，而是工程中的一些很细碎的问题。而这些问题中，把我弄得最惨的是一个看起来弱鸡的问题：在运行过程中保存最优参数。

这个问题困扰了我整整半天。

一开始我没有想到过在训练过程中保存最优参数，都是在文件末尾加一句: `torch.save({'net': net.state_dict()}, 'my_resnet.pth')` 。这样，每一次运行.py文件时，都是可以重新到上次的。后来我突发奇想，想到了这个操作。我在 `train_show.py` 的 `train()` 函数中定义形参 `save_path` ，把用来存参数的.pth文件名作为实参传给它。但是！噩梦就开始了：每次重新运行程序，无法读取上一次训练完的参数！我退一步，回到了在程序最后保存参数的情况后，惊恐万状地发现：也无法读取参数了！

最后倒腾了一天，改了很多地方，终于解决了这个问题。我现在都不知道我当时哪里错了，因为改的地方实在太多了。也许只是一个细节出了问题，但我永远找不到它了。

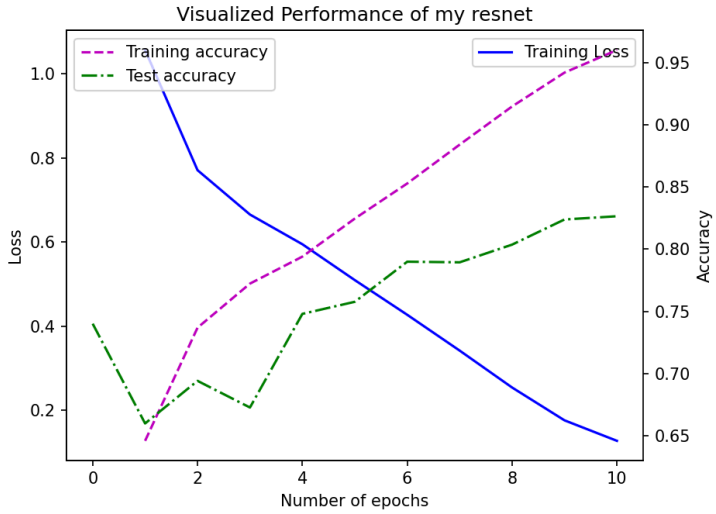
7. 训练结果

i. 我自己搭的ResNet：

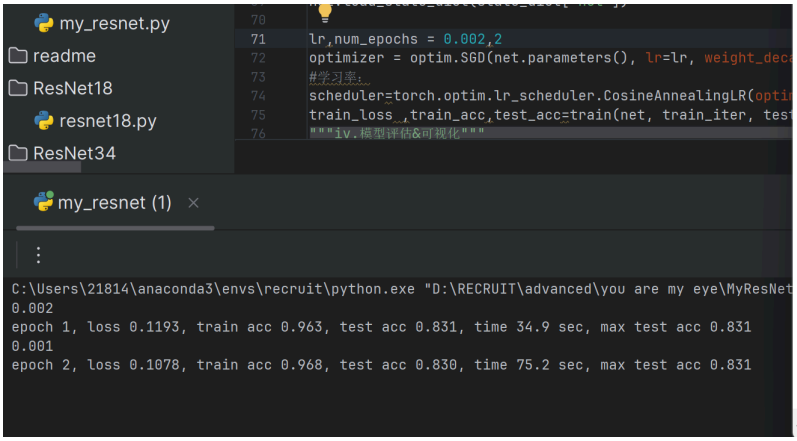
最后最高的精度反而是乱调出来的，没有什么技巧。两个ResNet都训练了多次，其中大部分的结果在这里未呈现。而VGG13训练得比较少

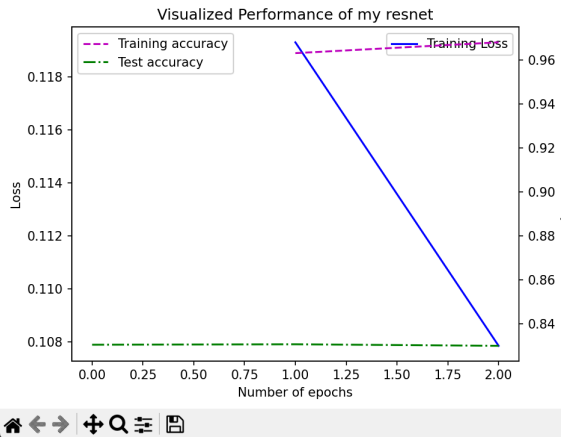
第一次：

```
0.08
epoch 1, loss 1.0558, train acc 0.646, test acc 0.660,
0.07804226065180615
epoch 2, loss 0.7706, train acc 0.737, test acc 0.694,
0.0723606797749979
epoch 3, loss 0.6653, train acc 0.772, test acc 0.673,
0.06351141009169893
epoch 4, loss 0.5947, train acc 0.794, test acc 0.748,
0.0523606797749979
epoch 5, loss 0.5095, train acc 0.825, test acc 0.758,
0.04
epoch 6, loss 0.4272, train acc 0.853, test acc 0.790,
0.027639320225002106
epoch 7, loss 0.3422, train acc 0.884, test acc 0.789,
0.016488589908301078
epoch 8, loss 0.2545, train acc 0.915, test acc 0.804,
0.007639320225002106
epoch 9, loss 0.1766, train acc 0.942, test acc 0.824,
0.0019577393481938587
epoch 10, loss 0.1279, train acc 0.960, test acc 0.827,
```



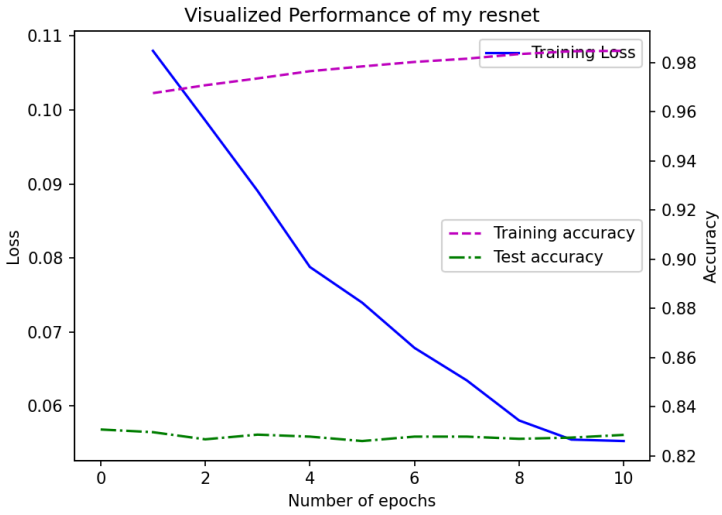
第二次：





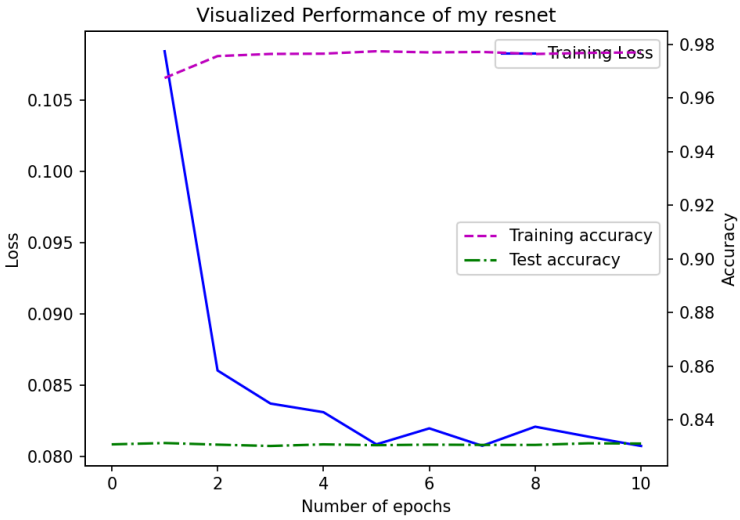
第三次：

```
C:\Users\21814\anaconda3\envs\recruit\python.exe "D:\REC
0.002
epoch 1, loss 0.1080, train acc 0.968, test acc 0.830, t
0.0019510565162951536
epoch 2, loss 0.0986, train acc 0.971, test acc 0.827, t
0.0018090169943749473
epoch 3, loss 0.0891, train acc 0.974, test acc 0.829, t
0.0015877852522924731
epoch 4, loss 0.0788, train acc 0.976, test acc 0.828, t
0.0013090169943749475
epoch 5, loss 0.0740, train acc 0.978, test acc 0.826, t
0.001
epoch 6, loss 0.0679, train acc 0.980, test acc 0.828, t
0.0006909830056250527
epoch 7, loss 0.0635, train acc 0.982, test acc 0.828, t
0.000412214747707527
epoch 8, loss 0.0581, train acc 0.983, test acc 0.827, t
0.00019098300562505268
epoch 9, loss 0.0555, train acc 0.985, test acc 0.828, t
4.894348370484647e-05
epoch 10, loss 0.0553, train acc 0.985, test acc 0.829,
```



第四次（可以看出test acc进步空间很小了）：

```
0.0001
epoch 1, loss 0.1084, train acc 0.967, test acc 0.831,
9.755282581475769e-05
epoch 2, loss 0.0860, train acc 0.976, test acc 0.831,
9.045084971874737e-05
epoch 3, loss 0.0837, train acc 0.976, test acc 0.830,
7.938926261462366e-05
epoch 4, loss 0.0831, train acc 0.976, test acc 0.831,
6.545084971874737e-05
epoch 5, loss 0.0808, train acc 0.977, test acc 0.831,
4.9999999999999996e-05
epoch 6, loss 0.0819, train acc 0.977, test acc 0.831,
3.454915028125263e-05
epoch 7, loss 0.0807, train acc 0.977, test acc 0.831,
2.0610737385376345e-05
epoch 8, loss 0.0821, train acc 0.976, test acc 0.831,
9.549150281252631e-06
epoch 9, loss 0.0814, train acc 0.977, test acc 0.831,
2.447174185242323e-06
epoch 10, loss 0.0807, train acc 0.977, test acc 0.831,
```



最高test acc：83.1%。已保存参数。

ii. 预训练的ResNet18：

第一次：

```
epoch 1, loss 1.1309, train acc 0.605, test acc 0.836, time 187.9 sec, max test acc 0.836
0.001
epoch 2, loss 0.8068, train acc 0.720, test acc 0.876, time 381.7 sec, max test acc 0.876
0.001
epoch 3, loss 0.7248, train acc 0.746, test acc 0.898, time 562.1 sec, max test acc 0.898
0.001
epoch 4, loss 0.6775, train acc 0.764, test acc 0.901, time 733.1 sec, max test acc 0.901
0.001
epoch 5, loss 0.6472, train acc 0.774, test acc 0.916, time 896.9 sec, max test acc 0.916
0.001
epoch 6, loss 0.6177, train acc 0.784, test acc 0.915, time 1075.9 sec, max test acc 0.916
0.001
epoch 7, loss 0.5954, train acc 0.791, test acc 0.922, time 1258.4 sec, max test acc 0.922
0.001
epoch 8, loss 0.5870, train acc 0.796, test acc 0.926, time 1444.5 sec, max test acc 0.926
0.001
epoch 9, loss 0.5705, train acc 0.803, test acc 0.918, time 11464.3 sec, max test acc 0.926
0.001
epoch 10, loss 0.5525, train acc 0.808, test acc 0.926, time 11620.8 sec, max test acc 0.926

进程已结束，退出代码为 0
```

第二次加上了scheduler，用指数衰退：

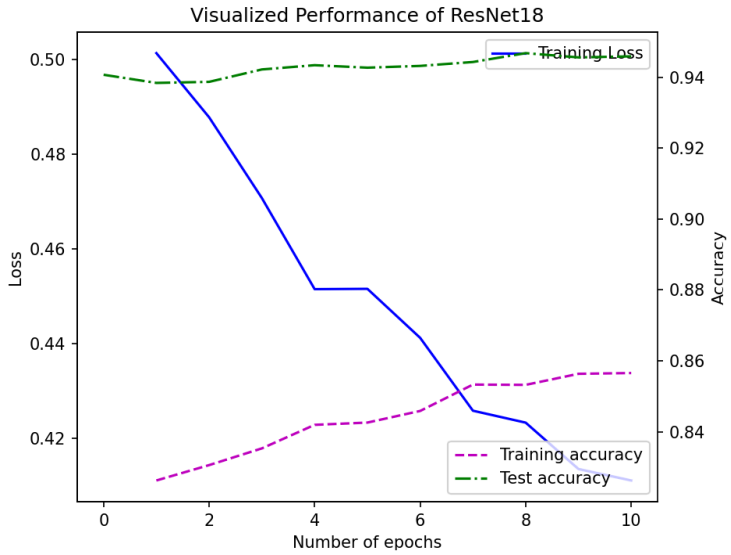
```
0.001
epoch 1, loss 0.5408, train acc 0.811, test acc 0.928, time 169.8 sec, max test acc 0.928
0.0008
epoch 2, loss 0.5361, train acc 0.814, test acc 0.931, time 347.6 sec, max test acc 0.931
0.00064
epoch 3, loss 0.5195, train acc 0.821, test acc 0.933, time 532.3 sec, max test acc 0.933
```

第三次：把scheduler去掉：

```
C:\Users\21814\anaconda3\envs\recruit\python.exe "D:\RE
0.001
epoch 1, loss 0.5242, train acc 0.817, test acc 0.933,
0.001
epoch 2, loss 0.5160, train acc 0.820, test acc 0.932,
0.001
epoch 3, loss 0.5028, train acc 0.826, test acc 0.935,
0.001
epoch 4, loss 0.5021, train acc 0.826, test acc 0.938,
0.001
epoch 5, loss 0.4893, train acc 0.830, test acc 0.938,
0.001
epoch 6, loss 0.4936, train acc 0.825, test acc 0.941,
0.001
epoch 7, loss 0.4746, train acc 0.836, test acc 0.941,
0.001
epoch 8, loss 0.4691, train acc 0.836, test acc 0.938,
0.001
epoch 9, loss 0.4716, train acc 0.835, test acc 0.940,
0.001
epoch 10, loss 0.4680, train acc 0.839, test acc 0.941,
```

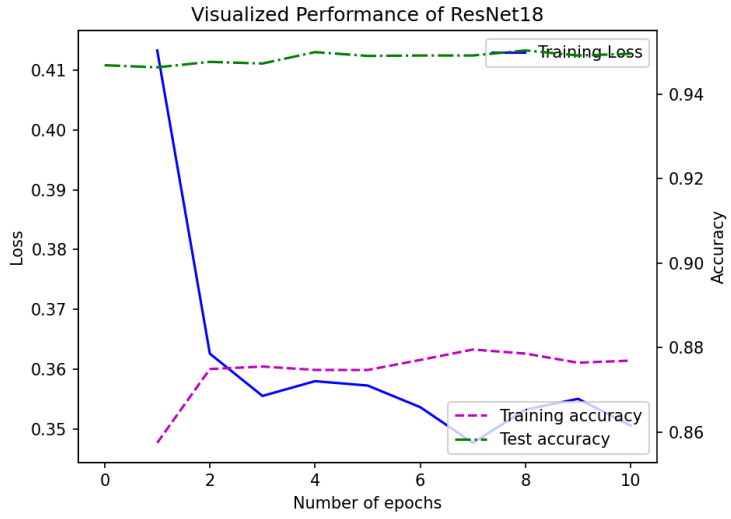
第四次：把scheduler换成余弦退火：

```
C:\Users\21814\anaconda3\envs\recruit\python.exe "D:\RE
0.002
epoch 1, loss 0.5013, train acc 0.826, test acc 0.938,
0.0019510565162951536
epoch 2, loss 0.4878, train acc 0.831, test acc 0.939,
0.0018090169943749473
epoch 3, loss 0.4707, train acc 0.835, test acc 0.942,
0.0015877852522924731
epoch 4, loss 0.4515, train acc 0.842, test acc 0.943,
0.0013090169943749475
epoch 5, loss 0.4516, train acc 0.843, test acc 0.943,
0.001
epoch 6, loss 0.4412, train acc 0.846, test acc 0.943,
0.0006909830056250527
epoch 7, loss 0.4259, train acc 0.853, test acc 0.944,
0.000412214747707527
epoch 8, loss 0.4234, train acc 0.853, test acc 0.947,
0.00019098300562505268
epoch 9, loss 0.4136, train acc 0.856, test acc 0.946,
4.894348370484647e-05
epoch 10, loss 0.4111, train acc 0.857, test acc 0.946,
```



第五次：把学习率初始化为0.0001，其他不变：

```
C:\Users\21814\anaconda3\envs\recruit\python.exe "D:\RE
0.0001
epoch 1, loss 0.4133, train acc 0.858, test acc 0.946,
9.755282581475769e-05
epoch 2, loss 0.3626, train acc 0.875, test acc 0.948,
9.045084971874737e-05
epoch 3, loss 0.3555, train acc 0.876, test acc 0.947,
7.938926261462366e-05
epoch 4, loss 0.3580, train acc 0.875, test acc 0.950,
6.545084971874737e-05
epoch 5, loss 0.3573, train acc 0.875, test acc 0.949,
4.9999999999999996e-05
epoch 6, loss 0.3536, train acc 0.877, test acc 0.949,
3.454915028125263e-05
epoch 7, loss 0.3477, train acc 0.880, test acc 0.949,
2.0610737385376345e-05
epoch 8, loss 0.3532, train acc 0.879, test acc 0.950,
9.549150281252631e-06
epoch 9, loss 0.3550, train acc 0.876, test acc 0.949,
2.447174185242323e-06
epoch 10, loss 0.3506, train acc 0.877, test acc 0.950,
```



（看起来用余弦退火有点浪费算力，因为后来学习率就太小了）

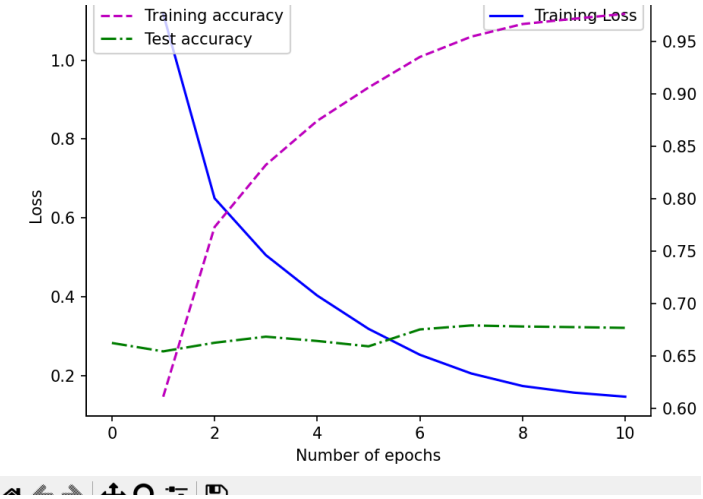
最高test acc：95.0%。已保存参数。

iii. 预训练的VGG13：

a. 不训练特征提取层

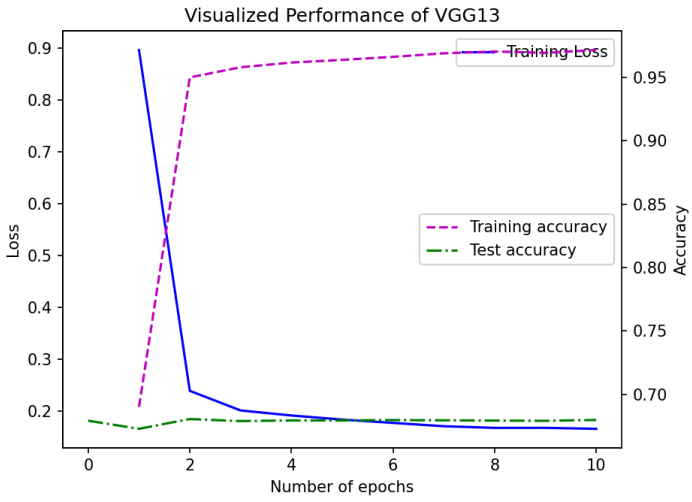
第一次：用余弦退火训了1次。可以看出：train acc已经很高，但test acc很难继续升高了

```
0.002
epoch 1, loss 1.1178, train acc 0.611, test acc 0.655, t
0.0019510565162951536
epoch 2, loss 0.6498, train acc 0.773, test acc 0.663, t
0.0018090169943749473
epoch 3, loss 0.5054, train acc 0.832, test acc 0.669, t
0.0015877852522924731
epoch 4, loss 0.4028, train acc 0.874, test acc 0.664, t
0.0013090169943749475
epoch 5, loss 0.3183, train acc 0.906, test acc 0.659, t
0.001
epoch 6, loss 0.2526, train acc 0.935, test acc 0.675, t
0.0006909830056250527
epoch 7, loss 0.2051, train acc 0.954, test acc 0.679, t
0.000412214747707527
epoch 8, loss 0.1733, train acc 0.966, test acc 0.678, t
0.00019098300562505268
epoch 9, loss 0.1565, train acc 0.971, test acc 0.678, t
4.894348370484647e-05
epoch 10, loss 0.1462, train acc 0.976, test acc 0.677,
```



第二次：从头开始，再训了1次。与第一次的结果差不多。

```
0.0002
epoch 1, loss 0.8969, train acc 0.690, test acc 0.673,
0.00019510565162951537
epoch 2, loss 0.2388, train acc 0.950, test acc 0.681,
0.00018090169943749473
epoch 3, loss 0.2009, train acc 0.958, test acc 0.679,
0.00015877852522924732
epoch 4, loss 0.1912, train acc 0.962, test acc 0.679,
0.00013090169943749474
epoch 5, loss 0.1832, train acc 0.964, test acc 0.679,
9.999999999999999e-05
epoch 6, loss 0.1768, train acc 0.966, test acc 0.680,
6.909830056250526e-05
epoch 7, loss 0.1704, train acc 0.969, test acc 0.680,
4.122147477075269e-05
epoch 8, loss 0.1672, train acc 0.970, test acc 0.679,
1.9098300562505263e-05
epoch 9, loss 0.1673, train acc 0.969, test acc 0.679,
4.894348370484646e-06
epoch 10, loss 0.1654, train acc 0.971, test acc 0.680,
```

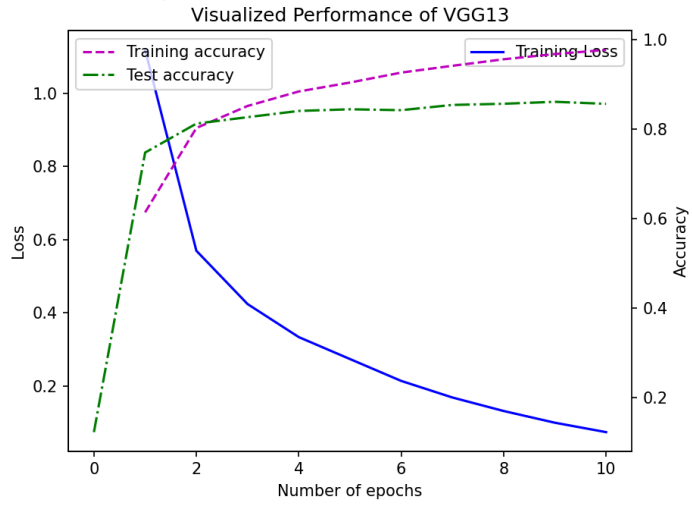


最高test acc:68.1%。未保存参数。

b. 训练特征提取层。

第一次：从头开始训一次。可以看出：在第一轮epoch后，就得到了比上面两次好得多的结果。

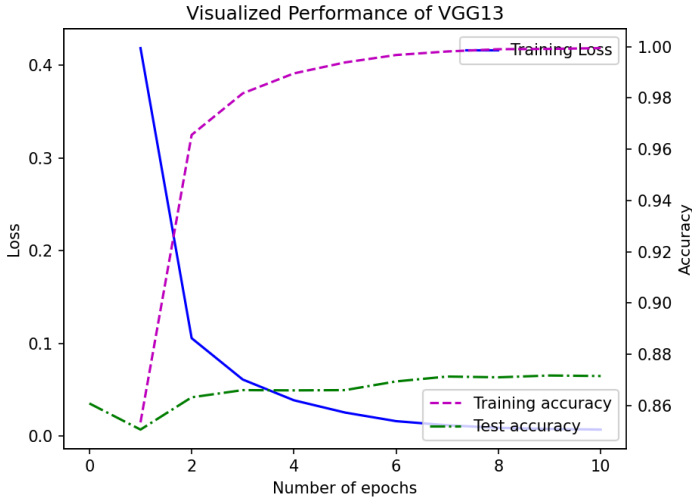
```
0.001
epoch 1, loss 1.1182, train acc 0.614, test acc 0.748,
0.001
epoch 2, loss 0.5699, train acc 0.802, test acc 0.812,
0.001
epoch 3, loss 0.4244, train acc 0.851, test acc 0.827,
0.001
epoch 4, loss 0.3342, train acc 0.884, test acc 0.840,
0.001
epoch 5, loss 0.2742, train acc 0.904, test acc 0.844,
0.001
epoch 6, loss 0.2145, train acc 0.926, test acc 0.842,
0.001
epoch 7, loss 0.1692, train acc 0.941, test acc 0.854,
0.001
epoch 8, loss 0.1324, train acc 0.956, test acc 0.856,
0.001
epoch 9, loss 0.1004, train acc 0.967, test acc 0.861,
0.001
epoch 10, loss 0.0747, train acc 0.977, test acc 0.856,
```



最高test acc:86.1%

第二次：在之前保存的参数上训。scheduler用余弦退火。最终train acc已经非常高了，而test acc趋于稳定，进步空间不大了。这个模型肯定是有点过拟合的。

```
0.001
epoch 1, loss 0.4183, train acc 0.853, test acc 0.851,
0.0009755282581475768
epoch 2, loss 0.1056, train acc 0.966, test acc 0.863,
0.0009045084971874736
epoch 3, loss 0.0610, train acc 0.982, test acc 0.866,
0.0007938926261462366
epoch 4, loss 0.0387, train acc 0.990, test acc 0.866,
0.0006545084971874737
epoch 5, loss 0.0255, train acc 0.994, test acc 0.866,
0.0005
epoch 6, loss 0.0162, train acc 0.997, test acc 0.869,
0.00034549150281252633
epoch 7, loss 0.0118, train acc 0.998, test acc 0.871,
0.00020610737385376635
epoch 8, loss 0.0090, train acc 0.999, test acc 0.871,
9.549150281252634e-05
epoch 9, loss 0.0078, train acc 0.999, test acc 0.872,
2.4471741852423235e-05
epoch 10, loss 0.0071, train acc 0.999, test acc 0.872,
```



最高test acc:87.2%。已保存参数。
两次结果说明这个预训练的VGG13由于自身结构，局限性还是在那，在这个数据集上不可能达到太大的精度。不排除我没有注意某些细节，导致限制了VGG13的更大潜力的可能；但即使再训一下，最终也很难比得上预训练的ResNet18。

小小的思考

现在，ResNet已经占据了图片分类训练的半壁江山。

但是，引入了重参数化方法之后，在台前表演的可能越来越多地变成了VGG了。训练时，用ResNet；在台前表演的，是VGG。没有ResNet的训练，VGG就不会推理得那么准；没有结构重参数化为VGG，ResNet的推理速度也没有那么快。ResNet是在对VGG的否定性中产生，却仍然要回到VGG。

在哲学上看，可以把ResNet看成VGG发展的一个阶段。VGG对自身单结构的扬弃，反而可以看作VGG为了维持自身同一性而不得不排出自己的单分支结构。变，才活；不变，就死。这是辩证法。

谢谢观看！