Part3预训练模型实验报告

代码

下载MINIST数据集

```
# 导入MINIST数据集
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=True, download=True,
                               transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                       (0.1307,), (0.3081,))
                               ])),
    batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=False, download=True,
                               transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                       (0.1307,), (0.3081,))
                               ])),
    batch_size=64, shuffle=True)
```

导入数据并划分训练集和测试集

```
# 预处理
my_transform = transforms.Compose([
        # transforms.Resize((224, 224)),   # 修改图片尺寸以适应resnet的输入
        # transforms.Grayscale(3),
        transforms.ToTensor(),
        transforms.Normalize((0.1307), (0.3081)),
        # transforms.Normalize((0.1307,0.1307,0.1307), (0.3081,0.3081,0.3081)),
    ])

# 训练集
train_file = datasets.MNIST(
    root='data',
    train=True,
    transform=my_transform
)
# 测试集
test_file = datasets.MNIST(
    root='data',
    train=False,
    transform=my_transform
)

train_loader = torch.utils.data.DataLoader(train_file,batch_size=64)
test_loader = torch.utils.data.DataLoader(test_file,batch_size=64)
```

选择预训练模型、损失函数、优化器

```
resnet50 = models.resnet50(pretrained=True)  #调用预训练模型
loss_function = nn.CrossEntropyLoss()  #定义损失函数
optimizer = optim.SGD(resnet50.parameters(), lr=0.1)
# 修改模型尺寸以适应MINIST数据集，最后10种输出
resnet50.fc = nn.Linear(2048,10)
# MINIST图片是单通道的,修改resnet第一层卷积层为单通道输入
resnet50.conv1 = nn.Conv2d(1, 64,kernel_size=5, stride=2, padding=3, bias=False)
print(resnet50)
```

训练过程

```
def train(model,train_loader,loss_func,optim,epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        output = model(data)
        loss = loss_func(output,target)
        optim.zero_grad()
        loss.backward()
        optim.step()
        if batch_idx % 10 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.item()))
```

测试过程

```
def test(model, test_loader,loss_func):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            # if batch_idx == 10000:
            # break
            output = model(data)
            test_loss += loss_func(output, target).item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the
max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item() #compare
prediction and lable
            test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.8f}, Accuracy: {}/{}
({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

主函数部分

```
if __name__=="__main__":
    for epoch in range(2):
        train(resnet50,train_loader,loss_function,optimizer,epoch)
        test(resnet50,test_loader,loss_function)
    torch.save(resnet50.state_dict(),"minist_resnet.pt")
```

训练结果

我最后选用的学习率是0.08，epoch是3，训练准确率在验证集合上达到99%（9877/10000），如果调低学习率并增加轮数应该可以再提升一点，但是这个part不是为了单纯追求准确率，所以我没有再去追求调参

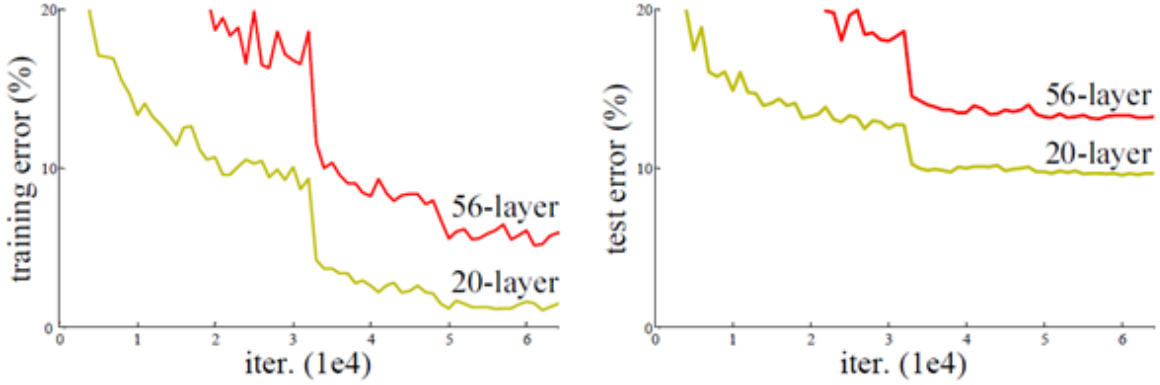
```
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.130773
Train Epoch: 2 [51840/60000 (86%)]      Loss: 0.006462
Train Epoch: 2 [52480/60000 (87%)]      Loss: 0.003891
Train Epoch: 2 [53120/60000 (88%)]      Loss: 0.018981
Train Epoch: 2 [53760/60000 (90%)]      Loss: 0.099901
Train Epoch: 2 [54400/60000 (91%)]      Loss: 0.056275
Train Epoch: 2 [55040/60000 (92%)]      Loss: 0.001427
Train Epoch: 2 [55680/60000 (93%)]      Loss: 0.134741
Train Epoch: 2 [56320/60000 (94%)]      Loss: 0.029842
Train Epoch: 2 [56960/60000 (95%)]      Loss: 0.002163
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.085821
Train Epoch: 2 [58240/60000 (97%)]      Loss: 0.001410
Train Epoch: 2 [58880/60000 (98%)]      Loss: 0.000971
Train Epoch: 2 [59520/60000 (99%)]      Loss: 0.000297

Test set: Average loss: 0.00000006, Accuracy: 9877/10000 (99%)
```

我选用的预训练模型——resnet50

模型介绍

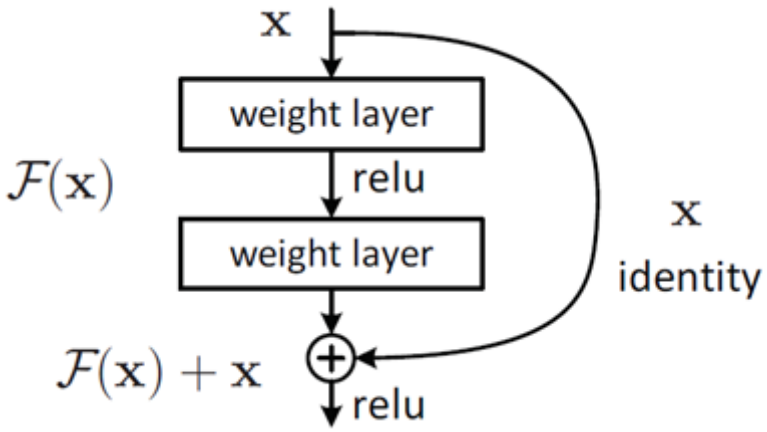
ResNet是深度残差网络（Deep residual network），网络深度对模型的性能影响很大，直观上，增加网络深度可以提高准确率，但是实验发现深度网络出现了退化问题（Degradation problem）：网络深度增加时，网络准确度出现饱和，甚至出现下降，比如下图所示的情况：



这个图中56层的网络比20层网络效果还要差，这不是过拟合问题，因为56层网络的训练误差同样高。深层网络存在着梯度消失或者爆炸的问题，这使得深度学习模型很难训练。

深度网络的退化问题说明深度网络不容易训练。但是我们考虑这样一个事实：现在有一个浅层网络，你想通过向上堆积新层来建立深层网络，一个极端情况是这些增加的层什么也不学习，仅仅复制浅层网络的特征，即这样新层是恒等映射（Identity mapping）。在这种情况下，深层网络应该至少和浅层网络性能一样，也不应该出现退化现象，所以是训练方法有问题，才使得深层网络很难去找到一个好的参数。

ResNet是何恺明博士提出的，利用残差学习来解决退化问题，使得深度网络的训练变得可行。对于一个堆积层结构（几层堆积而成）当输入为 x 时其学习到的特征记为 H(x)，现在我们希望其可以学习到残差 F(x)=H(x)−x，这样其实原始的学习特征是 F(x)+x 。之所以这样是因为残差学习相比原始特征直接学习更容易。当残差为0时，此时堆积层仅仅做了恒等映射，至少网络性能不会下降，实际上残差不会为0，这也会使得堆积层在输入特征基础上学习到新的特征，从而拥有更好的性能。残差学习的结构如下图所示：



残差学习相对容易的原因在于，残差一般比较小，需要学习的内容比较少，从数学的角度来分析如下：

首先残差单元可以表示为：

$$y_l = h(x_l) + F(x_l, W_l)$$
$$x_{l+1} = f(y_l)$$

其中 x_l 和 x_l+1 分别表示的是第 L 个残差单元的输入和输出，每个残差单元一般包含多层结构。 F 是残差函数，表示学习到的残差，而 h(x_l)=x_l 表示恒等映射， F 是ReLU激活函数。基于上式，我们求得从浅层 L 到深层 L+1 的学习特征为：

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i, W_i)$$

利用链式规则，可以求得反向过程的梯度：

$$\frac{\partial loss}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot \left(1 + \frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F(x_i, W_i)\right)$$

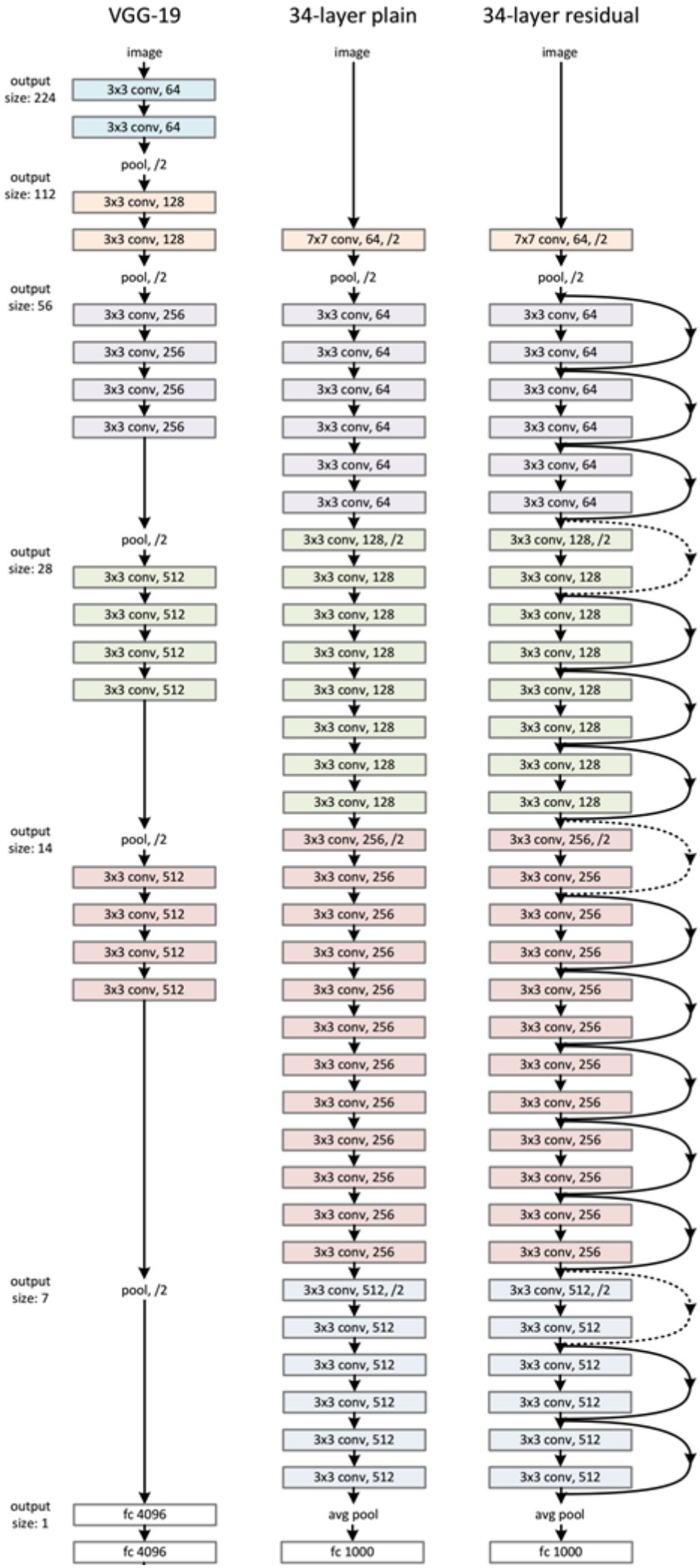
式子等号右边第一个因子表示的损失函数到达 L 的梯度，小括号中的1表明短路机制可以无损地传播梯度，而另外一项残差梯度则需要经过带有weights的层，梯度不是直接传递过来的。残差梯度不会那么巧全为-1，而且就算其比较小，有1的存在也不会导致梯度消失。所以残差学习会更容易。

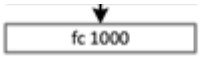
ResNet模型的网络结构

我选用的resnet50是深度为50层的ResNet模型，不同深度的ResNet模型结构汇总图如下：

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

ResNet网络结构图（以34层为例）和VGG结构图的对比如下：


VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
output size: 1
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000



ResNet网络是基于VGG19网络的基础上实现了，通过短路机制加入了残差单元，如上图所示。与VGG19相比，变化主要体现在ResNet直接使用stride=2的卷积做下采样，并且用global average pool层替换了全连接层。ResNet的一个重要设计原则是：当feature map大小降低一半时，feature map的数量增加一倍，这保持了网络层的复杂度。从上图中可以看到，ResNet相比普通网络每两层间增加了短路机制，这就形成了残差学习，其中虚线表示feature map数量发生了改变。上图中展示的34-layer的ResNet，还可以构建更深的网络，比如我选用的50层网络，还有101层、152层的网络。从上表中可以看到，对于18-layer和34-layer的ResNet，其进行的两层间的残差学习，当网络更深时（比如我选用的50层resnet50），其进行的是三层间的残差学习，三层卷积核分别是1x1，3x3和1x1，一个值得注意的是隐含层的feature map数量是比较小的，并且是输出feature map数量的1/4。

对于我们的MINIST数据集分类任务，我们不能直接调用torchvision.models里面的resnet50模型，这是因为MINIST数据集中的图片是单通道，大小为28×28，但是resnet50的输入是三通道，另一方面，`resnet50`是在`imagenet`上训练的，输出特征数是`1000`；而对于`mnist`来说，需要分`10`类，因此要改全连接层的输出为10：

```
resnet50 = models.resnet50(pretrained=True)  #调用预训练模型
resnet50.fc = nn.Linear(2048,10)  # 修改模型尺寸
resnet50.conv1 = nn.Conv2d(1, 64,kernel_size=5, stride=2, padding=3, bias=False)
print(resnet50)
```

我们可以打印出来resnet50的结构检验我们的修改：

```
C:\Users\84663\.conda\envs\pytorch\python.exe D:/aiproject1/part3/pretrain.py
ResNet(
  (conv1): Conv2d(1, 64, kernel_size=(5, 5), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=10, bias=True)
)
```

对”预训练+微调“的理解

“预训练“方法的诞生的背景是：标注资源稀缺而无标注资源丰富，即某种特殊的任务只存在非常少量的相关训练数据，以至于模型不能从中学习总结到有用的规律。

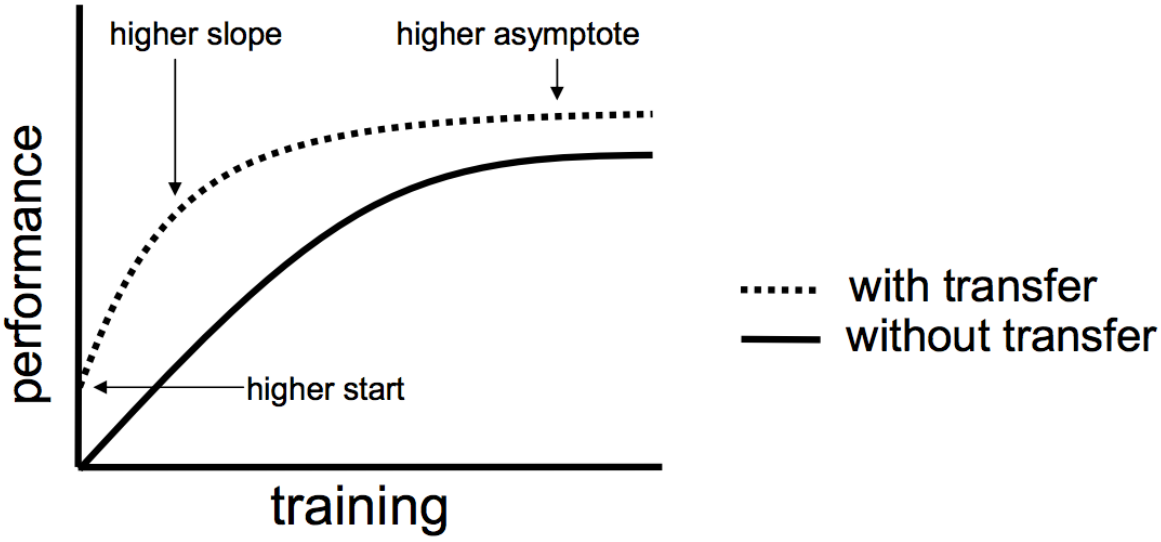
于是出现了先在大量非直接相关数据上训练模型学得“共性”，再在少量优质标注数据上训练进行微调的“**预训练+微调**”的模型训练范式，采用预训练模型是**迁移学习**的一种。基于这样的思想，“预训练“的做法一般是将大量低成本收集的训练数据放在一起，经过某种预训方法去学习其中的共性，然后将其中的共性“移植”到特定任务的模型中，再使用相关特定领域的少量标注数据进行“微调”，这样的话，模型只需要从”共性“出发，去“学习”该特定任务的“特殊”部分即可。相比参数随机赋值，完全从零开始训练，采用预训练不仅解决了优质标注数据稀缺的问题，而且收敛速度更快，这一点可以理解为预训练已经学好了任务的共性部分，只需要根据特定任务进行调整，比如我们要训练一个网络书写英文法律条文，但是英文法律条文数据较少，我们可以先让网络学习其他英文资料，先学会英语（这一步等于预训练），然后再用少量的英语法律条文训练已经学会英语的网络，对网络进行微调，因为网络已经学会了英语，在此基础上学习英文法律条文就会更快。

那么在我们的part3中，我们用了预训练模型resnet50，它是在ImageNet数据集上训练出来的，它在ImageNet学到了识别图片的一些共性知识，然后我们再用这个模型在MINIST数据集上进行训练以实现“微调”，达到我们的网络具备识别手写数字的能力。

利用预训练模型+微调这种迁移学习的方法，当我们要训练针对某种任务的网络时，我们可以调用类似任务的预训练模型，再用少量数据进行微调，提高我们训练网络的效率。

迁移学习主要有以下三个优点：

- 更高的起点。微调前，源模型的初始性能比不使用迁移学习高。
- 更高的斜率。训练中，源模型的提升速率比不使用迁移学习高。
- 更高的渐进。训练得到的模型的收敛性比不使用迁移学习更好。



实现迁移学习主要有两种常见的方法：

- Convnet微调：代替随机初始化，我们使用预训练的网络初始化我们的模型，例如在imagenet 1000数据集上训练的网络。其余的训练方法还是照旧。
- Convnet作为固定的特征提取器：冻结除全连接层外的所有网络的权重，最后的全连接层用一个具有随机权重的新层来替换，并且仅训练该层。

在我的代码中，采用的是Convnet微调的方法。