ResNet 解决什么问题？

自从深度神经网络 AlexNet 在 ImageNet 取得优异效果之后，深度神经网络就朝着网络层数越来越深的方向发展。增加网络深度，网络可以提取更加复杂的特征，直觉上应该可以取得更好的结果。

但事实并非如此，人们发现随着网络深度的增加，模型精度并不总是提升，并且这个问题显然不是由过拟合（overfitting）造成的，如果是过拟合，训练误差会不断变低，但测试误差会很高，目前的情况明显不是。因为网络加深后不仅测试误差变高了，它的训练误差也变高了。**如图所示，比较 56 层和 20 层的网络训练结果，发现 56 层的训练误差和测试误差都高于 20 层。**

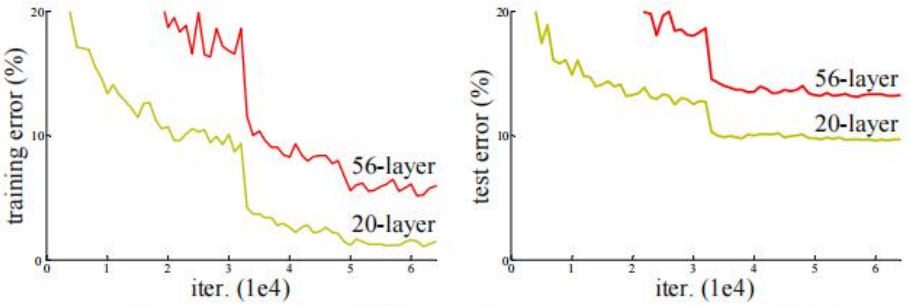


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

作者提出，这可能是因为更深的网络会伴随**梯度消失/爆炸**问题，从而阻碍网络的收敛，并将这种加深网络深度但网络性能却下降的现象称为**退化问题（degradation problem），**

ResNet 网络框架结构

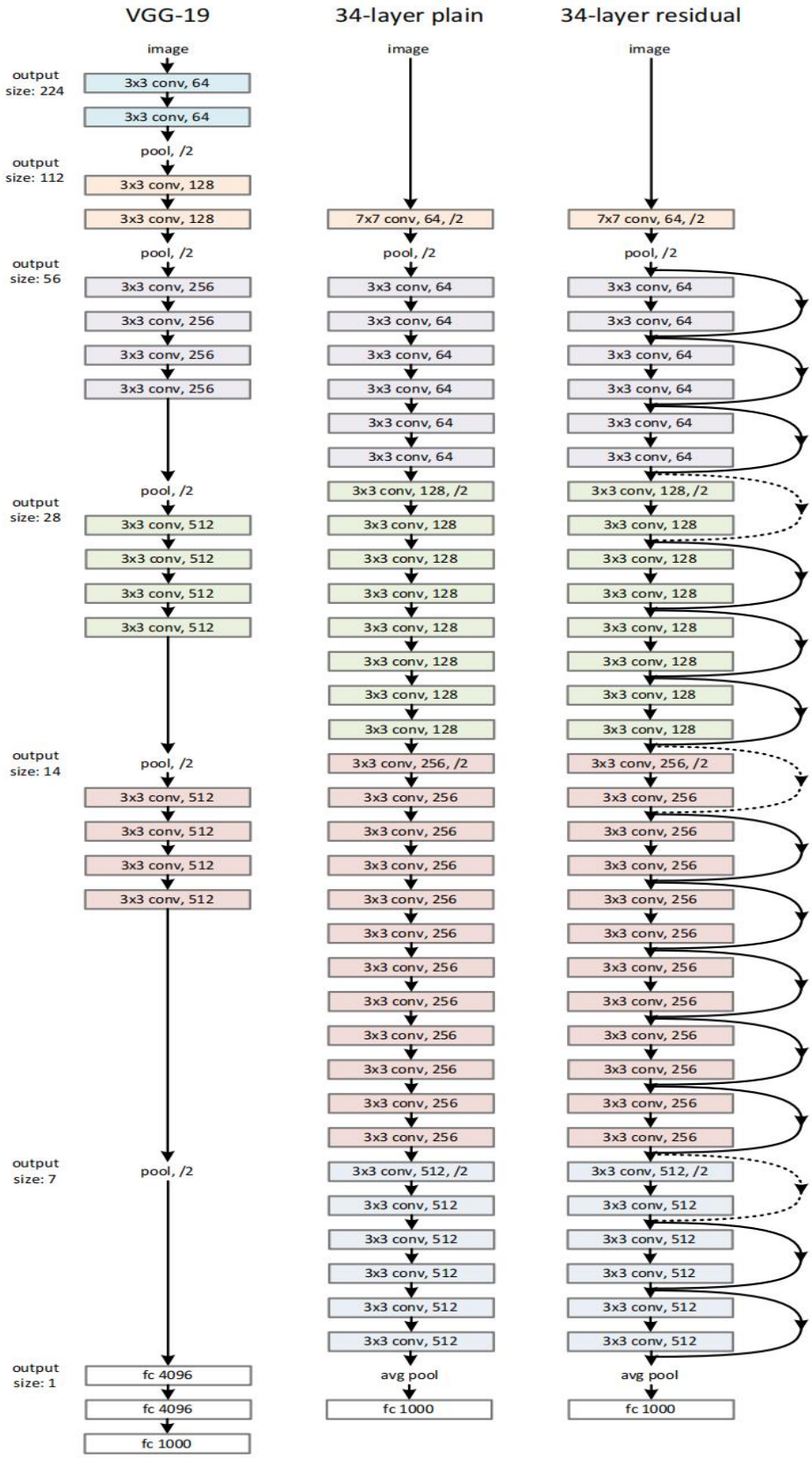


图 1

ResNet 网络的主要框架是基于 VGG 网络搭建而成的，分为两步，第一步是搭建的 plain 网络结构，这一步相比于 VGG 网络只是优化了深度和网络层的通道数，以及最后取代了 2 个全连接层；第二步是提出了论文的核心创新点，在 plain 网络的结构基础上进行了残差连接，将普通网络层模块变成了残差模块。

ResNet 的骨架网络与 VGG 网络的对比

论文的骨架 plain 网络结构(图 1，中)基于 VGG 网络 (图 1，左)改编搭建的，

（1）网络最开始先用一个大的 7*7 卷积核搭配一个池化层降低特征图尺寸；

（2）卷积层主要为 3*3 的卷积核，输出特征图尺寸相同的层含有相同数量的卷积核（主要保证通道数一致）；

（3）不同于 VGG 每个小模块最后使用最大池化层下采样降低特征图尺寸，ResNet 直接通过 stride 为 2 的卷积层（不填充）下采样。注意特征图尺寸减半，则卷积核的组数（通道数）增加一倍来保证每层的时间复杂度相同。

（4）全局平均池化层：网络的最后阶段不再是连续的三个全连接层，而是使用一个全局的平均 pooling 层和一个 1000 类的包含 softmax 的全连接层。全局平均池化是在 NIN 网络中提出来的，简单来说就是最后卷积层输出的特征图尺寸是[Batch,Channel,Height,Width]，经过全局平均池化后，尺寸将变为[Batch,Channel,1,1]，也就是说，全局平均池化其实就是对每一个通道特征图所有像素（特征）值求平均值，得到一个 1*1 的特征图。在代码里使用

self.avgpool=nn.AdaptiveAvgPool2d((1,1))初始化

（5）ResNet 模型比 VGG 网络有更少的卷积核和更低的计算复杂度。可以看到全局平均池化层取代了全连接层，而全连接层是参数量最大的部分。

残差网络

如果仅是上述网络结构的优化，ResNet 网络不会这么经典且效果显著，ResNet 网络的核心是在以上 plain 网络的基础上，插入 shortcut 连接(图 1，右)，将网络变成了对应的残差版本。接下来详细介绍一下。

残差基本单元

前面讲到 ResNet 网络主要是为了解决网络退化问题的，实际上，在此之前研究人员通过提出包括 **Batch normalization** 在内的方法，已经一定程度上缓解了这个问题，但依然不足以满足需求。

作者想到了**构建恒等映射**（Identity mapping），即通过在网络层之间加一个 shortcut 短接，可以在实际训练中将两个连接层之间的网络层忽略。举个例子来说，对于 56 层的网络，因为比 20 层网络更深，在不加任何处理的情况下，因为

梯度消失/爆炸等问题导致网络难以训练，训练误差反而降低。那么我们就在最后 36 层网络全部短接，也就意味着最后 36 层事实上是不工作的，输入是什么，输出还是什么，就等效成 20 层网络了。在实际应用中当然不是这么简单粗暴的把最后 36 层全部短接，而是在整个网络结构中设置一个个小的残差模块，在训练中决定哪些是短接的。

问题解决的标志是：增加网络层数，但训练误差不增加。

残差模块：残差模块的输入为 x，输出为 H(x)，若使用以前网络拟合的方法，直接训练网络拟合 H(x)；但残差模块将输出 H(x)分解为两部分，一部分是原始输入特征 x,另一部分是“残差”F(x)，即 H(x)=F(x)+x。

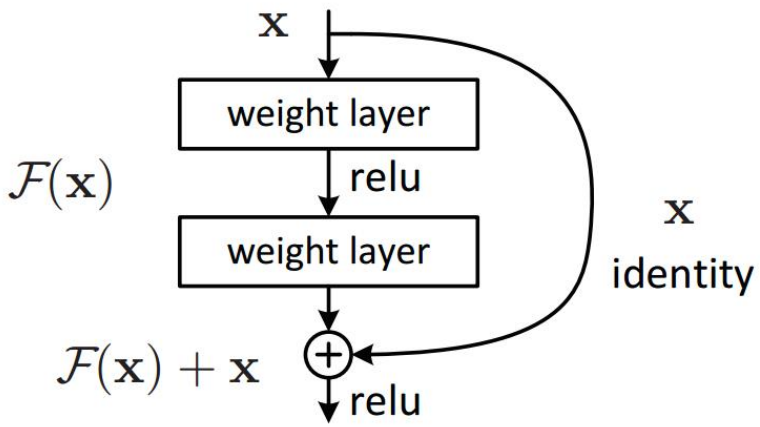


图 2 残差学习基本单元

如上图，原始输入特征 x 可以直接经过恒等映射连接从输入传到输出，但值得注意的是，根据输入和输出通道以及尺寸的不同，这部分也可能需要经过线性映射的。下面再仔细介绍一下是如何“恒等”映射的。而残差 F(x)就是正常网络训练拟合的函数了，不过，作者提出，网络学习残差 F(x)会比直接学习原始特征 H(x)简单的多。然后原始输入特征 x 和残差 F(x)加在一起作为该模块的最终输出 H(x)=F(x)+x 并传递给下一层网络。

这样做有什么优势呢？或者说怎么体现短接呢？如果 F(x)对于提高模型的训练精度无作用，自然梯度下降算法就调整该部分的参数，使该部分的效果趋近于 0，也就意味着主线路的 F(x)变为 0，这时网络的特征图流向就是只走“跳接”曲线（shortcut connection），此时 H(x)=F(x)+x=0+x=x 了，不必再去拟合 H(x)=x，很明显，将网络的主干拟合函数 F(x)优化为 0 比将最终输出拟合成恒等变换 H(x)=x 要容易得多。

刚刚讲了 ResNet 网络不是在最后 36 层一次性短接，而是一个一个的残差基本单元，那么残差基本单元 F（x）里面一般有多少层呢？论文中设置了两种方式，一中是卷积核通道数完全相同的 2 层网络；第二种是卷积核为 1*1，3*3,1*1，通道数也不完全相同的 3 层网络。

恒等映射（Identity mapping）的方式

前面讲到残差基本单元里的恒等映射有 2 种方式，区别就是原始输入特征图 x 和最终输出特征图 H（x）的尺寸与通道是否有差异，

根据连接前后的

（1）若残差基本单元原始输入特征图 x 和最终输出特征图 H（x）的尺寸与通道数相同，如果输入和输出的尺寸和通道相同时，可以直接使用恒等 shortcuts 映射，（ResNet 网络残差连接中的实线部分），即直接把 F(x)与 x 每个通道逐元素（长宽）相加：

H（x）=F(x,Wi)+x

其中 x 和 H（x）分别表示残差基本单元的输入和输出。函数 F(x,Wi)代表着学到的残差映射，根据图中的包含 2 层的基本单元，F=W2 σ (W1x)，若三层，则为 F=W3 σ （W2 σ (W1x)），σ 代表 ReLU

这种方式通过 shortcuts 直接传递输入 x，不会引入额外的参数也不会增加模块的计算复杂性，因此可以公平地将残差网络和 plain 网络作比较。

（2）如果两者尺寸或通道不同，需要给 x 执行一个线性映射（代码里的下采样 downsample）来匹配通道和尺寸。

H（x）=F(x,Wi)+Wsx.

Ws 这种方式的目的仅仅是为了保持 x 与 H(x)之间的尺寸和通道一致，所以通常只在相邻残差块之间通道数改变时使用，绝大多数情况下仅使用第一种方式。

当维度增加时（ResNet 网络残差连接中的虚线部分），考虑两个选项：

(A) 零填充：shortcut 仍然使用恒等映射，在增加的维度上使用 0 来填充，这样做不会增加额外的参数；

(B) 线性投影变换：使用 Eq.2 的映射 shortcut 来使维度保持一致（通过 1*1 的卷积，参数需要学习，精度比 zero-padding 更好，但是耗时更长，占用更多内存）。

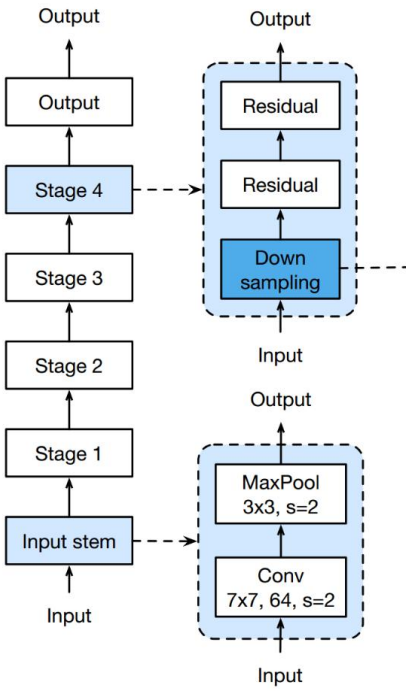
对于这两个选项，当 shortcut 跨越两种尺寸的特征图时，均使用 stride 为 2 的卷积。

图像维度变化

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
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

如图是 5 种不同的 ResNet 网络层架构，可以看到最开始是统一的 7*7*64 的卷积核，步长为 2，填充为 3，将 224*224 的原始输入特征图下采样到 112*112*64，再经过一个尺寸 3*3，步长为 2 的最大池化层，输出特征图为 56*56*64，接下来进入最关键的残差网络结构。

残差网络结构包括 4 个残差模块，conv2_x,conv3_x,conv4_x 和 conv5_x,主要分为 2 大类，一大类是 18 层和 34 层网络配置的基础模块 basicblock，另一大类就是 50,101,152 层网络配置的组合模块 bottleblock，如图所示。二者的区别主要是残差模块内部的基础卷积层的数量以及卷积核的尺寸和通道数。



明确几个概念：

（1）子结构：图中的 conv2_x,conv3_x,conv4_x 和 conv5_x；

（2）残差模块：每个残差块会有多个重复的子结构，图中的*2 这些系数，每个子结构有多个卷积层

（3）卷积层：子结构中是由不同尺寸和通道数的卷积层组成的，如下图，34 层中每个子结构有 2 个 3*3 的卷积层；50 层往上是 1*1,3*3,1*1 三个卷积层组成的。

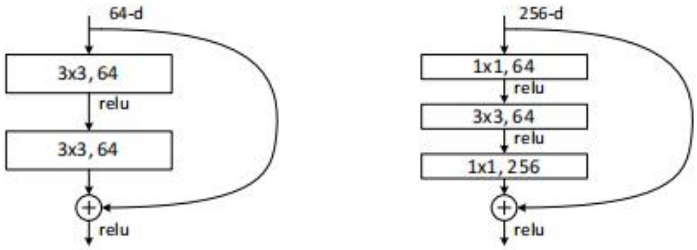


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

下图是 34 层的网络特征图的通道和尺寸变化，

Layer34	in_channel*hegiht*width	out-channel*hegiht*width	stride	padding	down sample	卷积核尺寸&通道数
conv2.1	64*56*56	64*56*56	1	1	None	64*64*3*3
conv2.x	64*56*56	64*56*56	1	1	None	64*64*3*3
conv3.1	64*56*56	128*28*28	2	1	TRUE	128*64*3*3
conv3.x	128*28*28	128*28*28	1	1	none	128*128*3*3
conv4.1	128*28*28	256*14*14	2	1	TRUE	256*128*3*3
conv4.x	256*14*14	256*14*14	1	1	none	256*256*3*3
conv5.1	256*14*14	512*7*7	2	1	TRUE	512*256*3*3
conv5.x	512*7*7	512*7*7	1	1	none	512*512*3*3

从中可以看到：

（1）本层的卷积核的意义是，上一层的特征图经过本层的卷积核得到本层的特征图，本层的卷积核数对应着本层的输出通道数；在创建模型的时候我们只需要考虑每层卷积核的尺寸及通道数变化。

（2）每个子结构（图中的 conv2_x 等）中的残差模块是相同的，每个残差模块内部的卷积层也都是相同的，都是 2 个 3*3 的卷积层，在后面乘上重复的系数。但是不同子结构的卷积层通道数是不同的，这就意味着跨子结构时，输入输出通道会发生变化。

（3）残差模块从 3_x 开始，在模块的第一个子结构的第一个卷积层上 stride=2，特征图的尺寸会缩小一倍，输出通道数会扩大 1 倍，同一个子结构后面的卷积层不会再改变特征图的尺寸和通道数。这也就意味着最开始输入的特征图尺寸和该子结构最后输出的特征图尺寸是不同的，在计算残差连接时需要先将 x 进行下采样 downsample。

（4）输入通道是一个全局变量，每个子结构最后更改其值，输出通道每个子结构都一样，传递参数赋值。

下图是 50 层的网络特征图的通道和尺寸变化，

Layer50	in_channel*hegiht*width	out-channel*hegiht*width	stride	padding	down sample	卷积核尺寸&通道数
conv2.1.1	64*56*56	64*56*56	1	0	None	64*64*1*1
conv2.1.2	64*56*56	64*56*56	1	1	None	64*64*3*3
conv2.1.3	64*56*56	256*56*56	1	0	None	256*64*1*1
conv2.x.1	256*56*56	64*56*56	1	0	None	64*256*1*1
conv2.x.2	64*56*56	64*56*56	1	1	None	64*64*3*3
conv2.x.3	64*56*56	256*56*56	1	0	None	256*64*1*1
conv3.1.1	256*56*56	128***56*56**	1	0	None	128*256*1*1
conv3.1.2	128***56*56**	128***28*28**	2	1	TRUE	128*128*3*3
conv3.1.3	128***28*28**	512***28*28**	1	0	None	512*128*1*1
conv3.x.1	512***28*28**	128***28*28**	1	0	None	128*512*1*1
conv3.x.2	128***28*28**	128***28*28**	1	1	None	128*128*3*3
conv3.x.3	128***28*28**	512***28*28**	1	0	None	512*128*1*1
conv4.1.1	512***28*28**	256*28*28	1	0	None	256*512*1*1
conv4.1.2	256*28*28	256*14*14	2	1	TRUE	256*256*3*3
conv4.1.3	256*14*14	1024*14*14	1	0	None	1024*256*1*1
conv4.x.1	1024*14*14	256*14*14	1	0	None	256*1024*1*1
conv4.x.2	256*14*14	256*14*14	1	1	None	256*256*3*3
conv4.x.3	256*14*14	1024*14*14	1	0	None	1024*256*1*1
conv5.1.1	1024*14*14	512*14*14	1	0	None	512*1024*1*1
conv5.1.2	512*14*14	512*7*7	2	1	TRUE	512*512*3*3
conv5.1.3	512*7*7	2048*7*7	1	0	None	2048*512*1*1
conv5.x.1	2048*7*7	512*7*7	1	0	None	512*2048*1*1
conv5.x.2	512*7*7	512*7*7	1	1	None	512*512*3*3
conv5.x.3	512*7*7	2048*7*7	1	0	None	2048*512*1*1

同样地，相比 34 层的网络结构，50 层的特殊之处在于：

（1）每个子结构残差模块的内部是由 1*1,3*3,1*1 三个卷积层组成的。

（2）conv 子结构每个残差模块的第三层 1*1 卷积层通道数会扩大 4 倍，但因为子结构会重复多次，所以同一子结构下一个残差模块第一层的输入通道数是会是扩大 4 倍的值，但输出通道数是不变；

（3）50 层以上也是在子结构 conv3 开始降低特征图尺寸，且是在第一个子结构的 3*3 卷积层实现，stride=2，其它网络层（所有子结构）都是 stride=1；

代码

https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

ResNet 主要有五种主要形式：Res18，Res34，Res50，Res101，Res152；

如下图所示，每个网络都包括三个主要部分：**输入部分**、**输出部分**和**中间残差模块部分**（中间残差模块部分包括 4 个 layer）。尽管 ResNet 的变种形式丰富，但都遵循上述的结构特点，**网络之间的不同主要在于中间残差模块部分的 block 参数和个数存在差异**。

接下来的四个阶段，也就是代码中的 layer1,layer2,layer3,layer4，这也是网络的主干部分。这里用 make_layer 函数产生四个 layer，需要用户输入每个 layer 的 block 数目（即 layers 列表)以及采用的 block 类型（basicblock 还是 bottleblock）

model.py

```
import torch
import torch.nn as nn
import random
import torch.nn.functional as F

# 定义基础残差模块，基础残差模块的主要变化点就是不同模块之间的切换时的输入通道变化，其它的都可以复用，下采样也发生在模块的第一个子结构的第一个卷积层
class BasicBlock(nn.Module):
    expansion=1
    #change_channels：输入通道数。out_channels：输出通道数。
    # stride：默认为 1，第一个卷积层的 stride 才会变化。downsample：从 make_layer 中传入的 downsample 层。
    def __init__(self,change_channels,out_channels,stride=1,downsample=None):
        super(BasicBlock,self).__init__()
        #定义第 1 组 conv3x3 -> norm_layer -> relu，这里使用传入的 stride 和 change_channels。
        # （如果是 layer2 ，layer3 ，layer4 里的第一个 BasicBlock，那么 stride=2，这里会降采样和改变通道数）。

self.conv1=nn.Conv2d(in_channels=change_channels,out_channels=out_channels,kernel_size=3,stride=stride,padding=1)
        self.bn1=nn.BatchNorm2d(num_features=out_channels)
        #注意，2 个卷积层都需要经过 relu 层，但它们使用的是同一个 relu 层。
        self.relu=nn.ReLU()
        #定义第 2 组 conv3x3 -> norm_layer -> relu，这里不使用传入的 stride（默认为 1），
        # 输入通道数和输出通道数使用 out_channels，也就是不需要降采样和改变通道数。
        self.conv2 = nn.Conv2d(in_channels=out_channels,
out_channels=out_channels, kernel_size=3, stride=1,padding=1)
        self.bn2 = nn.BatchNorm2d(num_features=out_channels)
```

```
            self.downsample=downsample
# 输入数据分成两条路，一条路经过两个 3 * 3 卷积，另一条路直接短接，二者相加经过 relu 输出。
# 核心就是判断短接的路要不要 downsample，下采样同时则增加通道数，这点在 make_layer 里面通过 stride 是否！=1（即特征尺寸变化）或者输入输出通道数是否一致来判断
        def forward(self,x):
            # x 赋值给 identity，用于后面的 shortcut 连接。
            identity=x
            # x 经过第 1 组 conv3x3 -> norm_layer，得到 out。
            # 如果是 layer2 ，layer3 ，layer4 里的第一个 BasicBlock，那么 downsample 不为空，会经过 downsample 层，得到 identity。
            if self.downsample:
                identity=self.downsample(x)
#x 经过第 1 组 conv3x3 -> norm_layer -> relu，如果是 layer2 ，layer3 ，layer4 里的第一个 BasicBlock，那么 stride=2，第一个卷积层会降采样。
            x=self.conv1(x)
            x=self.bn1(x)
            x=self.relu(x)

            x=self.conv2(x)
            x=self.bn2(x)
            print("identity",identity.size())
            print("x",x.size())
#最后将 identity 和 out 相加，经过 relu ，得到输出。
            x=x+identity
            x=self.relu(x)
            return x
class BottleBlock(nn.Module):
        expansion=4
        #in_channel：输入通道数。out_channel：输出通道数。
        #stride:第一个卷积层的 stride。downsample:从 layer 中传入的 downsample 层。
        def __init__(self,in_channel,out_channel,stride=1,downsample=None):
            super(BottleBlock,self).__init__()
            #定义子结构里的 1*1 卷积层 conv1x1 -> norm_layer，注意这里的 out_channel 会降维。将上一子结构最后的通道数降下来，stride 默认为 1。

self.conv1=nn.Conv2d(in_channels=in_channel,out_channels=out_channel,kernel_size=1, stride=1)
            self.bn1=nn.BatchNorm2d(num_features=out_channel)
           #定义子结构里的 3*3 卷积层 conv3x3 -> norm_layer，这里使用传入的 stride，这里输出通道数不发生变化。
            # （如果是 layer2 ，layer3 ，layer4 里的第一个 Bottleneck，那么
```

stride=2，这里会降采样）。

```
self.conv2=nn.Conv2d(in_channels=out_channel,out_channels=out_channel,kernel_size=3,stride=stride,padding=1)
            self.bn2=nn.BatchNorm2d(num_features=out_channel)
          #定义子结构里的第二个 1*1 卷积层，conv1x1 -> norm_layer，使用 out_channel * self.expansion 作为输出通道数升维,stride 默认为 1。
            self.conv3 = nn.Conv2d(in_channels=out_channel, out_channels=out_channel*self.expansion, kernel_size=1, stride=1)
            self.bn3 = nn.BatchNorm2d(num_features=out_channel*self.expansion)
            self.relu=nn.ReLU(inplace=True)
            self.downsample=downsample


      def forward(self,x):
            identity=x
            if self.downsample:
                  identity=self.downsample(x)
           #1x1  的卷积是为了降维，减少通道数
            x=self.conv1(x)
            x=self.bn1(x)
            x = self.relu(x)
            # 3x3  的卷积是为了改变图片大小，不改变通道数
            x=self.conv2(x)
            x=self.bn2(x)
            x = self.relu(x)
            # 1x1  的卷积是为了升维，增加通道数，增加到  out_channel * 4
            x=self.conv3(x)
            x=self.bn3(x)

            print("identity", identity.size())
            print("x", x.size())

            x=x+identity
            x = self.relu(x)
            return x
class ResNet(nn.Module):
      def __init__(self,block,blocks_num,num_classes):
            super(ResNet,self).__init__()
            #定义残差模块最开始的输入通道数为 64
            self.change_channels=64
            #输入部分首先经过为一个 size=7x7，stride 为 2 的卷积处理，得到 112*112*64 的特征图，
```

```
self.conv1=nn.Conv2d(in_channels=3,out_channels=64,kernel_size=7,stride=2,padding=3)
        self.bn1=nn.BatchNorm2d(num_features=64)
        self.relu=nn.ReLU()
        # 然后经过 size=3x3, stride=2 的最大池化处理，一个 224x224 的输入图像就会变 56x56*64 大小的特征图，极大减少了存储所需大小。并将此特征图传到残差模块
        self.maxpool=nn.MaxPool2d(kernel_size=3,stride=2,padding=1)

        self.layer1=self._make_layer(block,blocks_num[0],64,stride=1)

        self.layer2=self._make_layer(block,blocks_num[1],128,stride=2)

        self.layer3=self._make_layer(block,blocks_num[2],256,stride=2)

        self.layer4=self._make_layer(block,blocks_num[3],512,stride=2)
       #通过全局自适应平滑池化，把所有的特征图拉成 1*1，
        #网络的最后阶段不再是连续的三个全连接层，而是使用一个全局的平均 pooling 层和一个 1000 类的包含 softmax 的全连接层。
        # 全局平均池化是在 NIN 网络中提出来的，简单来说就是最后卷积层输出的特征图尺寸是[Batch,Channel,Height,Width]，
        # 经过全局平均池化后，尺寸将变为[Batch,Channel,1,1]，也就是说，全局平均池化其实就是对每一个通道特征图所有像素（特征）值求平均值，得到一个 1*1 的特征图。
        # 对于 res18 来说，就是 1x512x7x7 的输入数据拉成 1x512x1x1，然后接全连接层输出，输出节点个数与预测类别个数一致。
        self.avgpool=nn.AdaptiveAvgPool2d((1,1))
        self.flateen=nn.Flatten()
        self.fc=nn.Linear(512*block.expansion,num_classes)

        # 遍历网络中的每一层
        # 继承 nn.Module 类中的一个方法:self.modules(), 他会返回该网络中的所有 modules
        for m in self.modules():
            # isinstance(object, type)：如果指定对象是指定类型，则 isinstance() 函数返回 True
            # 如果是卷积层
            if isinstance(m, nn.Conv2d):
                # kaiming 正态分布初始化，使得 Conv2d 卷积层反向传播的输出的方差都为 1
                # fan_in：权重是通过线性层（卷积或全连接）隐性确定
                # fan_out：通过创建随机矩阵显式创建权重
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
```

```
    def forward(self,x):
        x=self.conv1(x)
        x=self.bn1(x)
        x=self.relu(x)
        x=self.maxpool(x)
        print("layer1")
        x=self.layer1(x)
        print("layer2")
        x=self.layer2(x)
        print("layer3")
        x=self.layer3(x)
        print("layer4")
        x=self.layer4(x)
        x=self.avgpool(x)
        x=self.flateen(x)
        x=self.fc(x)

        return x

    #block：每个 layer 里面使用的 block，可以是 BasicBlock，BottleBlock。
    #block_num,一个整数，表示该层 layer 有多少个 block，根据给定的 blocks_num 里遍历的
    #out_channels 输出的通道数
    #stride：第一个 block 的卷积层的 stride，默认为 1。注意，BasicBlock 只有在每个 layer 的第一个子结构的卷积层使用该参数。
    #BottleBlock 只有在每个 layer 的第一个子结构的第二个卷积层使用该参数。
    def _make_layer(self, block, block_num, out_channels, stride):
        layers = []
#判断 stride 是否为 1，输入通道和输出通道是否相等。如果这两个条件都不成立，那么表明需要建立一个 1 X 1 的卷积层将原始输入 x 下采样升维变成和输出尺寸通道一致，
        # 来改变通道数和改变图片大小。具体是建立 downsample 层，包括 1x1 卷积层和 norm_layer，1x1 卷积层注意输出通道数和 stride。
        downsample=None
        if stride!=1 or self.change_channels!=out_channels*block.expansion:
            #建立第一个 block，把 downsample 传给 block 作为降采样的层，并且 stride 也使用传入的 stride（stride=2）
            downsample = nn.Sequential(

nn.Conv2d(in_channels=self.change_channels,out_channels=out_channels*block.expansion,kernel_size=1, stride=stride),
                nn.BatchNorm2d(num_features=out_channels*block.expansion)
            )
```

```
        #第一个子结构需要单独传参创建，后面的子结构才可以遍历创建
        layers.append(block(self.change_channels, out_channels, stride, downsample))
        #改变输入通道数 self.change_channels=out_channels*block.expansion，这个变量是整个类的全局变量
        #o 在 BasicBlock 里，expansion=1，因此这一步不会改变通道数。在 BottleBlock 里，expansion=4，因此这一步会改变通道数。
        self.change_channels=out_channels*block.expansion
        #图片经过第一个 block 后，就会改变通道数和图片大小。接下来 for 循环添加剩下的 block。
        # 从第 2 个 block 起，输入和输出通道数是相等的，因此就不用传入 downsample 和 stride（那么 block 的 stride 默认使用 1，
        for i in range(1, block_num):
            layers.append(block(self.change_channels, out_channels))

        return nn.Sequential(*layers)

# 测试代码
        # 每个 python 模块（python 文件）都包含内置的变量 __name__，当该模块被直接执行的时候，__name__ 等于文件名（包含后缀 .py ）
        # 如果该模块 import 到其他模块中，则该模块的 __name__ 等于模块名称（不包含后缀.py）
        #“__main__” 始终指当前执行模块的名称（包含后缀.py）
        # if 确保只有单独运行该模块时，此表达式才成立，才可以进入此判断语法，执行其中的测试代码，反之不行
if __name__=="__main__":
    x=torch.randn([1,3,224,224])
    # [3, 4, 6, 3] 等则代表了 bolck 的重复堆叠次数
    # blocks_num=[3,4,6,3]
    # model=ResNet(BasicBlock,blocks_num,num_classes=7)
    blocks_num = [3, 4, 6, 3]
    model = ResNet(BottleBlock, blocks_num, num_classes=7)
    y=model(x)
    print(y.size())
    print(model)
```

train.py

```
#从自己创建的 models 库里导入 ResNet 模块
#import ResNet  仅仅是把 ResNet.py 导入进来,当我们创建 ResNet 的实例的时候需要通过指定 ResNet.py 中的具体类.
#例如:我的 ResNet.py 中的类名是 ResNet,则后面的模型实例化 ResNet 需要通过
```

ResNet.ResNet()来操作
#还可以通过 from 还可以通过 from ResNet import * 直接把 ResNet.py 中除了以 _ 开头的内容都导入
from models.cv.ResNet import *
# torchvision：PyTorch 的一个图形库，服务于 PyTorch 深度学习框架的，主要用来构建计算机视觉模型
# transforms：主要是用于常见的一些图形变换
# datasets：包含加载数据的函数及常用的数据集接口
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
# os：operating system（操作系统），os 模块封装了常见的文件和目录操作
import os
#导入画图的库，后面将主要学习使用 axes 方法来画图
import matplotlib.pyplot as plt

设置数据转化方式，如数据转化为 Tensor 格式，数据切割等
Compose()：将多个 transforms 的操作整合在一起
ToTensor(): 将 numpy 的 ndarray 或 PIL.Image 读的图片转换成形状为(C,H, W)的 Tensor 格式，且归一化到[0,1.0]之间
#compose 的参数为列表[]
train_transform=transforms.Compose([
# RandomResizedCrop(224)：将给定图像随机裁剪为不同的大小和宽高比，然后缩放所裁剪得到的图像为给定大小
    transforms.RandomResizedCrop(224),
RandomVerticalFlip()：以 0.5 的概率竖直翻转给定的 PIL 图像
    transforms.RandomHorizontalFlip(),
    # ToTensor()：数据转化为 Tensor 格式
    transforms.ToTensor(),
# Normalize()：将图像三个通道的像素值归一化到[-1,1]之间，使模型更容易收敛
    transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])
])
test_transform=transforms.Compose([transforms.Resize((224, 224)),
                                   transforms.ToTensor(),
                                   transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

#ImageFolder(root, transform``=``None``, target_transform``=``None``, loader``=``default_loader)
#root 指定路径加载图片； transform：对 PIL Image 进行的转换操作，transform 的输入是使用 loader 读取图片的返回对象
#target_transform：对 label 的转换 loader：给定路径后如何读取图片，默认读取为 RGB 格式的 PIL Image 对象
#label 是按照文件夹名顺序排序后存成字典，即{类名:类序号(从 0 开始)}，一般

```
来说最好直接将文件夹命名为从 0 开始的数字，举例来说，两个类别，
#狗和猫，把狗的图片放到文件夹名为 0 下；猫的图片放到文件夹名为 1 的下面。
# 这样会和 ImageFolder 实际的 label 一致， 如果不是这种命名规范，建议看看
self.class_to_idx 属性以了解 label 和文件夹名的映射关系
#python 中\是转义字符，Windows 路径如果只有一个\，会把它识别为转义字符。
#可以用 r"把它转为原始字符，也可以用\\,也可以用 Linux 的路径字符/。
train_dataset=ImageFolder(r"E:\计算机\data\fer2013_数据增强版本
\train",train_transform)
test_dataset=ImageFolder(r"E:\计算机\data\fer2013_数据增强版本
\test",test_transform)

# DataLoader：将读取的数据按照 batch size 大小封装并行训练
# dataset (Dataset)：加载的数据集
# batch_size (int, optional)：每个 batch 加载多少个样本(默认: 1)
# shuffle (bool, optional)：设置为 True 时会在每个 epoch 重新打乱数据(默认: False)
train_dataloader=DataLoader(train_dataset,batch_size=32,shuffle=True)

test_dataloader=DataLoader(test_dataset,batch_size=32,shuffle=True)

device='cuda' if torch.cuda.is_available() else 'cpu'

blocks_num = [3, 4, 6, 3]
model = ResNet(BottleBlock, blocks_num, num_classes=7).to(device)
# 定义损失函数（交叉熵损失）
loss_fn=nn.CrossEntropyLoss()
# 定义 adam 优化器
# params(iterable)：要训练的参数，一般传入的是 model.parameters()
# lr(float)：learning_rate 学习率，也就是步长，默认：1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# 迭代次数（训练次数）
epochs = 30
# 用于判断最佳模型
best_acc = 0.0
# 最佳模型保存地址
#save_path = './{}Net.pth'.format(model_name)
train_steps = len(train_dataloader)

def train(train_dataloader,model,loss_fn,optimizer):
    loss,acc,n=0.0,0.0,0
    # dataloader: 传入数据（数据包括：训练数据和标签）
    # enumerate()：用于将一个可遍历的数据对象(如列表、元组或字符串)组合
为一个索引序列，一般用在 for 循环当中
    # enumerate 返回值有两个：一个是序号，一个是数据（包含训练数据和标
签）
```

```
    # x：训练数据（inputs）(tensor 类型的），y：标签（labels）(tensor 类型的）
    # 和 dataloader 结合使用时返回数据下标是 batch（在创建 dataloader 时会把 batch size 作为参数传入），
    # 从 0 开始，最大数为样本总数除以 batch size 大小，数据是一 batch 的数据和标签
    for batch,(x,y) in enumerate(train_dataloader):
        x,y=x.to(device),y.to(device)
        output=model(x)
        cur_loss=loss_fn(output,y)
        # torch.max(input, dim)函数
        # input 是具体的 tensor，dim 是 max 函数索引的维度，0 是每列的最大值，1 是每行的最大值输出
        # 函数会返回两个 tensor，第一个 tensor 是每行的最大值；第二个 tensor 是每行最大值的索引
        _,pred=torch.max(output,axis=1)
        # 计算每批次的准确率
        # output.shape[0]一维长度为该批次的数量
        # torch.sum()对输入的 tensor 数据的某一维度求和
        cur_acc=torch.sum(pred==y)/output.shape[0]
        # 清除过往梯度值，防止上个 batch 的数据的梯度值累积
        optimizer.zero_grad()
        cur_loss.backward()
        optimizer.step()
        #.item()将张量转化为标量
        loss+=cur_loss.item()
        acc+=cur_acc.item()
        n=n+1
    train_loss=loss/n
    train_acc=acc/n
    print('train_loss==' + str(train_loss))
    # 计算训练的准确率
    print('train_acc' + str(train_acc))
    return train_loss, train_acc

#测试函数里参数无优化器，不需要再训练，只需要测试和验证即可
def test(test_dataloader,model,loss_fn):
    loss,acc,n=0.0,0.0,0
    # with torch.no_grad()：将 with 语句包裹起来的部分停止梯度的更新，从而节省了 GPU 算力和显存，但是并不会影响 dropout 和 BN 层的行为
    with torch.no_grad():
        for batch,(x,y) in enumerate(test_dataloader):
            x,y=x.to(device),y.to(device)
            output=model(x)
            cur_loss=loss_fn(output,y)
```

```
            _,pred=torch.max(output,axis=1)
            cur_acc=torch.sum(pred==y)/output.shape[0]
            optimizer.zero_grad()
            cur_loss.backward()
            optimizer.step()
            loss+=cur_loss.item()
            acc+=cur_acc.item()
            n=n+1
        test_loss=loss/n
        test_acc=acc/n
        print('test_loss==' + str(test_loss))
        # 计算训练的准确率
        print('test_acc' + str(test_acc))
        return test_loss, test_acc

def matplot_loss(train_loss, test_loss):
    fig, ax = plt.subplots(1, 1)
    # 参数 label = "传入字符串类型的值，也就是图例的名称
    ax.plot(train_loss, label='train_loss')
    ax.plot(test_loss, label='test_loss')
    # loc 代表了图例在整个坐标轴平面中的位置（一般选取'best'这个参数值）
    ax.legend(loc='best')
    ax.set_xlabel('loss')
    ax.set_ylabel('epoch')
    ax.set_title("训练集和验证集的 loss 值对比图")
    plt.show()

    # 准确率


def matplot_acc(train_acc, test_acc):
    fig, ax = plt.subplots(1, 1)
    ax.plot(train_acc, label='train_acc')
    ax.plot(test_acc, label='test_acc')
    ax.legend(loc='best')
    ax.set_xlabel('acc')
    ax.set_ylabel('epoch')
    ax.set_title("训练集和验证集的 acc 值对比图")
    plt.show()

epochs=20
min_acc=0.0

loss_train=[]
```

```
acc_train=[]
loss_test=[]
acc_test=[]

for t in range(epochs):
      #不同的优化函数不同的使用方法
     # lr_scheduler.step()
      print(f"{t+1}\n------")
      train_loss,train_acc=train(train_dataloader,model,loss_fn,optimizer)
      test_loss,test_acc=test(test_dataloader,model,loss_fn)
      loss_train.append(train_loss)
      acc_train.append(train_acc)
      loss_test.append(test_loss)
      acc_test.append(test_acc)

      if test_acc>min_acc:
            folder="save_model"
            if not os.path.exists(folder):
                  os.mkdir(folder)
            min_acc=test_acc
            print(f"保存{t+1}轮")
            torch.save(model.state_dict(),"save_model/resnet-best-model.pth")
      if t==epochs-1:
            # torch.save(model.state_dict(),path)只保存模型参数，推荐使用的方式
            torch.save(model.state_dict(), 'save_model/resnet-best-model.pth')
matplot_loss(loss_train,loss_test)
matplot_acc(acc_train,acc_test)
```

参考资料

[1]https://zh.d2l.ai/index.html
[2]https://arxiv.org/pdf/1312.4400

[3]He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).