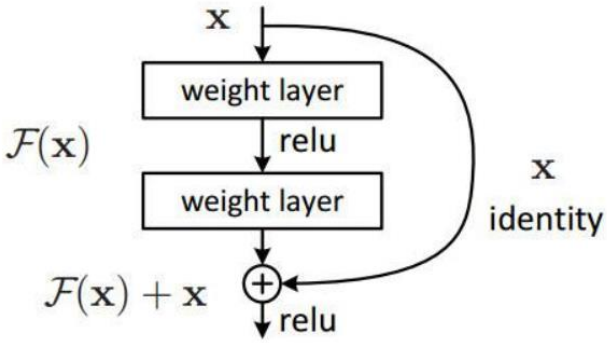
1. ResNet 的创新点

梯度消失问题：基于反向传播法计算梯度优化的神经网络，由于反向传播求隐藏层梯度时利用了链式法则，梯度值会进行一系列的连乘，导致浅层隐藏层的梯度会出现剧烈的衰减，这也就是梯度消失问题的本源。（另一种说法：随着网络的训练，导致反向传播的损失 loss 变得越来越小，最后小到为 0，这就会导致网络输入层前面的神经元学习不到任何特征，由于其每次训练变化的梯度都是 0，其参数不会改变。这就使其失去了意义。）

由于梯度消失问题，随着传统的卷积网络结构越来越深，分类准确率反而下降了，这限制了网络层数的加深。而 ResNet 本质上是为了缓解梯度消失问题而设计的 CNN。

上述问题表明，直接的映射很难学习，因此，ResNet 不是学习层的输出和它的输入之间的映射，而是学习它们之间的差异 - 学习残差 residual。比较官方的残差网络原理说明：“若将输入设为 x，将某一有参网络层设为 H，那么以 x 为输入的此层的输出将为 H(x)。一般的 CNN 网络如 Alexnet/VGG 等会直接通过训练学习出参数函数 H 的表达，从而直接学习 x -> H(x) 。而残差学习则是致力于使用多个有参网络层来学习输入、输出之间的残差即 H(x) - x，即学习 x -> (H(x) - x) + x。其中 x 这一部分为直接的 identity mapping，而 H(x) - x 则为有参网络层要学习的输入输出间残差。”



上图即是 ResNet 的核心部分，其中的 weight layer 可以看作是卷积层。如果用数学公式来规范一下，设 F(x)为卷积操作，则普通的网络规范为 Net = F(x)，而 ResNet 则规范为 Net = F(x) +x。（F(x) = 上一段所说的 H(x) - x）

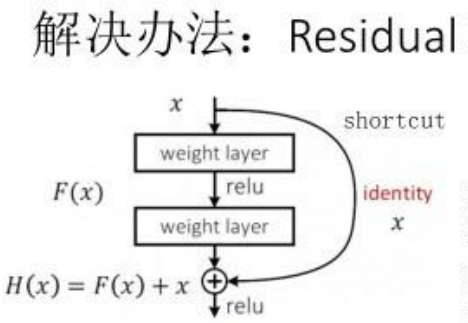
直观地说，ResNet 的核心就是右边的 shortcut，也就是公式中的 x。这样网络就能做出选择，如果层数过深的时候，网络没有作用了，那么网络学习过程中就把 F(x)学习为 0，这样网络就变为了 Net = x，其实就相当于过于深的层不起作用了，但也不会造成负影响，所以 ResNet 最终的结果只会大于等于传统的网络，而不会小于。

此外还有一个原因，学习残差 F(x)=H(x)-x 会比直接学习原始特征 H(x)简单的多，因此选择优化 F(x)能够使得网络训练起来更加容易。（这个的解释在后文 ppt 截图中有）

可以这样说，skip connections 或 shortcuts 为前面的层提供了梯度的捷径，跳过了一堆层。这一设计虽然简单，但是作用却重大，在 ResNet 出现之前网络最多十几层，比如 Vgg16，但是 ResNet 出现后，网络能达到 100 层，比如 ResNet 101。

以上说明可能依然不够全面，作为补充，我贴几张网上找到的 ppt 截图：

(1) ResNet 的核心 residual net：



(1) 增加了一个shortcut
(2) 我们就拟合的对象就变成F（x），F（x）就是残差
(3) 拟合目标变成使F（x）趋近于0

rather than expect stacked layers to approximate $\mathcal{H}(\mathbf{x})$, we explicitly let these layers approximate a residual function $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$. The original function thus becomes $\mathcal{F}(\mathbf{x}) + \mathbf{x}$. Although both forms should be able to asymptotically approximate the desired functions (as hypothesized), the ease of learning might be different.

图6 残差模块

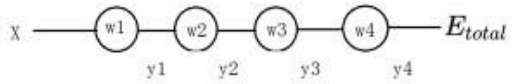
The degradation problem suggests that the solvers might have difficulties in approximating identity mappings by multiple nonlinear layers. With the residual learning reformulation, if identity mappings are optimal, the solvers may simply drive the weights of the multiple nonlinear layers toward zero to approach identity mappings.

(2) 梯度问题的来源：

什么是梯度弥散（爆炸）简单例子

$$w_1^+ = w_1 - \eta \frac{\partial E_{total}}{\partial w_1}$$

$$y_i = \sigma(z_i) = \sigma(w_i x_i + b_i)$$

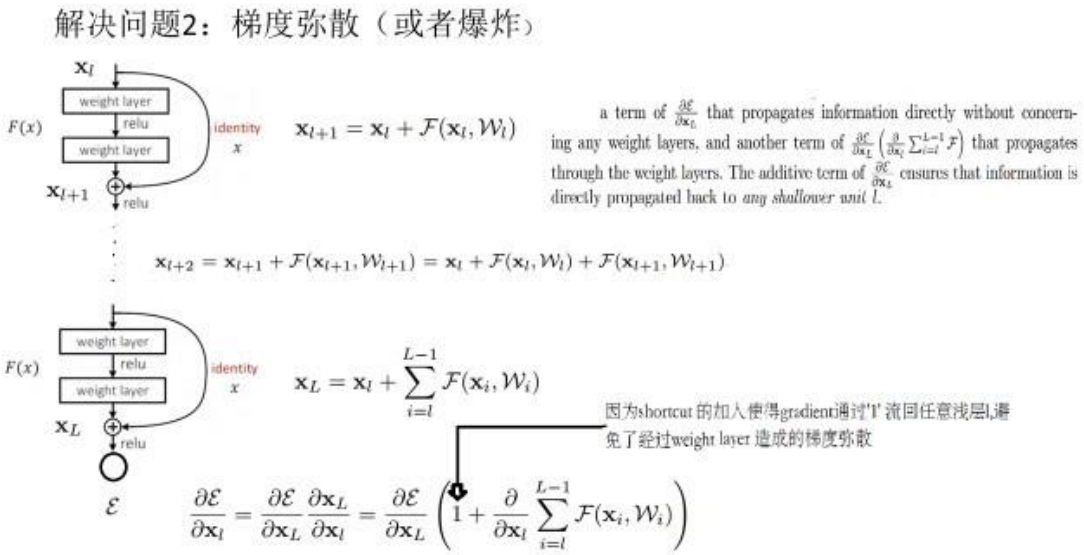


$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial y_4}\frac{\partial y_4}{\partial z_4}\frac{\partial z_4}{\partial x_4}\frac{\partial x_4}{\partial z_3}\frac{\partial z_3}{\partial x_3}\frac{\partial x_3}{\partial z_2}\frac{\partial z_2}{\partial x_2}\frac{\partial x_2}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$

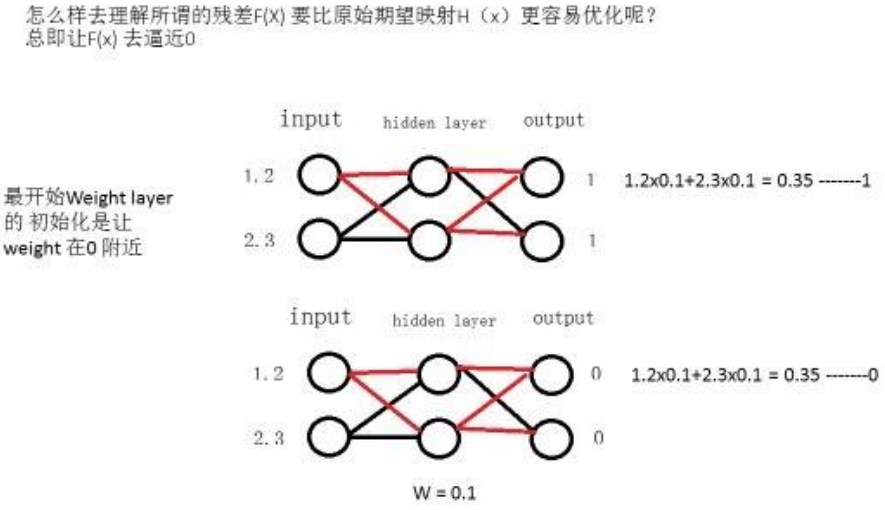
$$= \frac{\partial E_{total}}{\partial y_4}\sigma'(z_4)\, w_4 \sigma'(z_3)\, w_3 \sigma'(z_2)\, w_2 \sigma'(z_1) x_1$$

因此对于上面的链式求导，层数越多：
(1) 我们初始化的网络权值 W 通常都在0 附近，W2*W3*W4 越小，因而导致梯度消失的情况出现。
(2) 如果 W 都大于 1，或者部分比较大的情况下，就会产生梯度爆炸。

(3) ResNet 为什么能解决梯度问题：

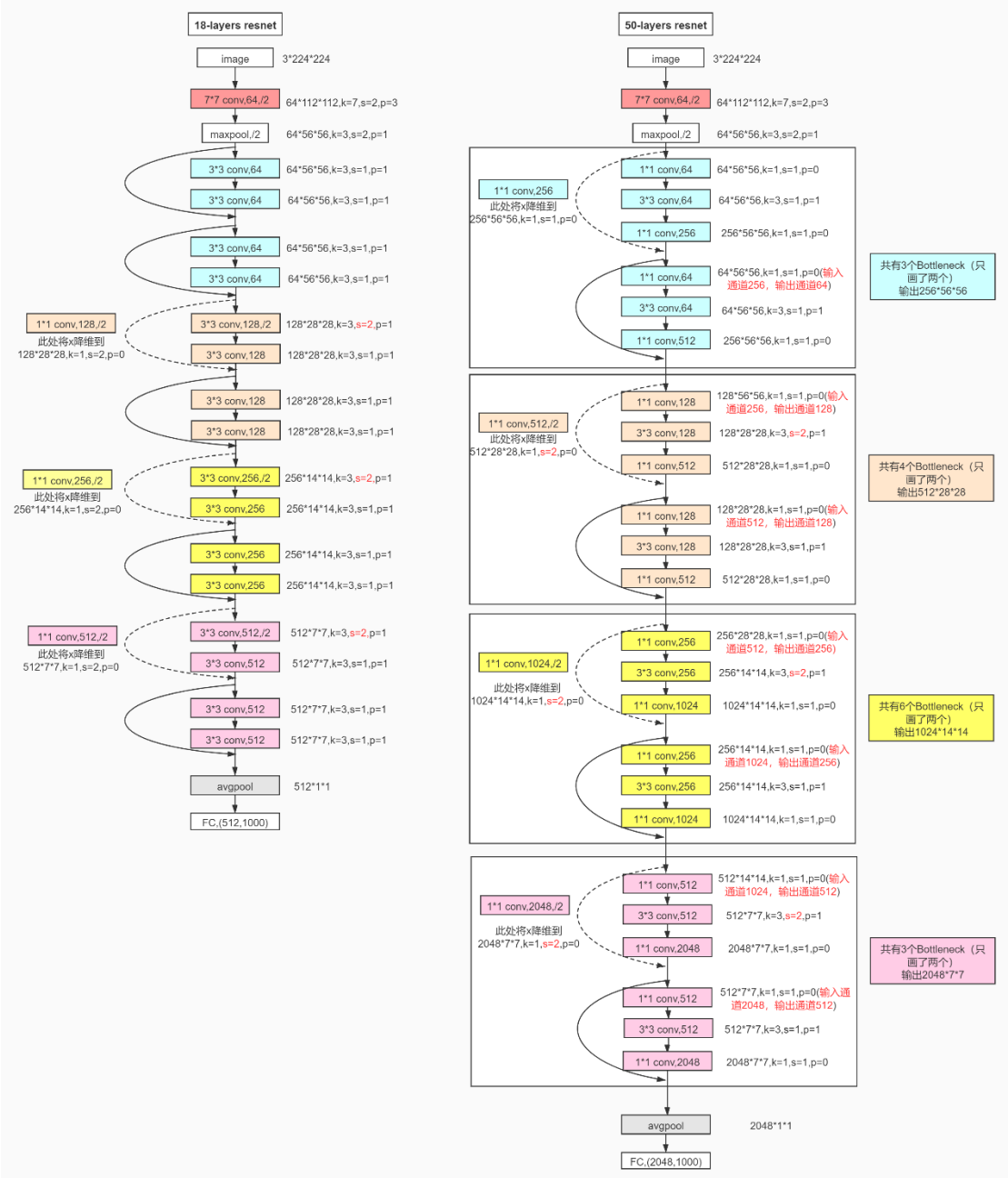


(4) 为什么学习残差 F(x)=H(x)-x 会比直接学习原始特征 H(x)简单的多：



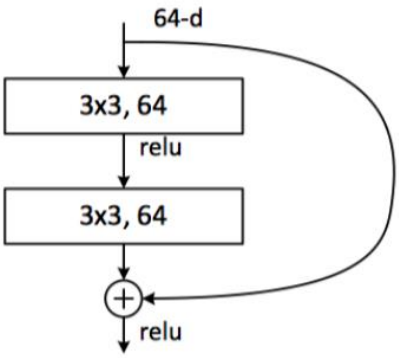
现在的通用方法是，将 weight 初始化为一个接近于 0 的很小的数字，这样 weight 离 0 的距离本来就比离 1 的距离小很多。

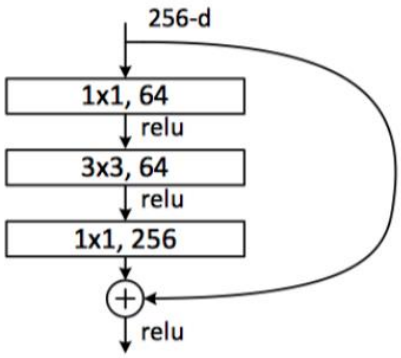
2. ResNet 的具体结构（下图左边为 ResNet18，右边为 ResNet50）



ResNet 网络的总体结构为：首先对输入图像进行预处理（7x7 的 conv），然后经过 4 个 layer（每个 layer 里有不同数量的 block），最后经过 FC 层得到输出。尽管 ResNet 的变种形式丰富，但都遵循这种结构特点，网络之间的不同主要在于中间卷积部分的 block 参数和个数存在差异。

两种不同的 block 如下图所示（左侧为 BasicBlock，右侧为 Bottleneck）：





在较为浅层的 ResNet 中（ResNet18，ResNet34）中使用的基本模块叫 BasicBlock，它由两个 3x3 的 Conv2d 堆叠而成。在较为深层的 resnet 中（ResNet50，ResNet101，ResNet152），使用的是一种瓶颈结构 Bottleneck，它由（1,1），（3,3），（1,1）的 Conv2d 堆叠而成，第一个 1x1 的卷积层用于降维，第二个 3x3 层用于处理，第三个 1x1 层用于升维，这样既能增加模块深度，又能减少参数量和计算量。

中间的 4 个 layer 为 ResNet 的主体部分，不同 ResNet 在这 4 个 layer 中的 block 数量为：ResNet18：[2,2,2,2]；ResNet34：[3,4,6,3]；ResNet50：[3,4,6,3]；ResNet101：[3,4,23,3]；ResNet152：[3,8,36,3] 。

注意：ResNet18，ResNet50 中的 18 和 50 等数字表示整个网络中 conv 层和 FC 层的层数（由于 relu、BN、pooling 都不带 weight，因此不纳入计算），而不是 block 的数目，具体来说，18 = 1+(2+2+2+2)*2+1，50 = 1 +(3+4+6+3)*3+1。

3.代码实现（以 ResNet18 为例）

首先，在图像进入 CNN 前，需要进行归一化处理：

```
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225])
])
```

原始 input 图像 size 为(64, 64, 3)，像素值区间为[0, 255]。先进行缩放，将原图像尺寸改为 224x224；再经过 transforms.ToTensor()的处理，也可能是专门用于 imagenet 数据集的处理方法，总之像素数值区间变为[0.0, 1.0]；在此基础上，再进行正规化，就是 x = (x - mean)/std，得到能够直接输入 CNN 的 input；而 mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]则是 imagenet 根据数据集抽样计算出来的数值。

接下来正式进入 ResNet。无论是哪种类型的 ResNet，将 input 数据输入后，首先都要经过以下 4 步处理：

```
(conv1): Conv2d(3, 64, kernel_size=(7, 7), 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)
```

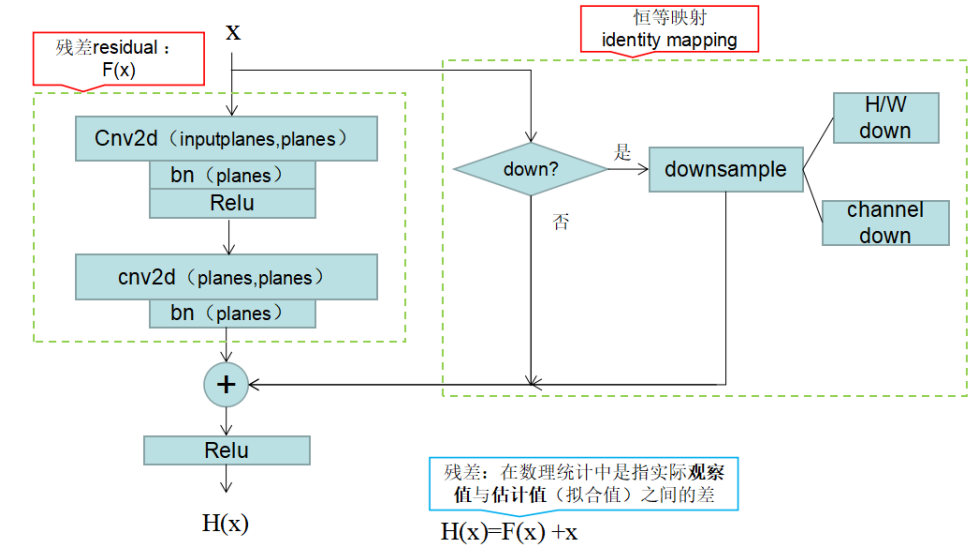
conv 和 maxpool 没有必要再进行解释了，大家都很熟悉。relu 为是一种人工神经网络中常用的激活函数，f(x) = max(0, x)。相对来说，我们不太了解的是 BN，因此补充说明一下：

BN（Batch Normalization）就是批量标准化，简而言之，就是对于每一层网络的输出，对其做一个归一化，使其服从标准的正态分布，这样后一层网络的输入也是一个标准的正态分布，所以能够比较好的进行训练，加快收敛速度。BN 是将 feature map 标准化为在 0 附近，一般位于卷积层或者 pooling 层之后，激活函数之前。因为有些常用激活函数在 0 附近变化率最大，BN 之后，特征图经过激活函数，比如 sigmoid 或 relu，被映射到新的特征空间，由于激活函数在 0 附近变化率大，因此被映射到新的特征空间的特征图也更容易被区别开来，这样既帮助了分类，也同时缓解了梯度消失问题。

经过以上 4 步后，正式进入 ResNet 的主体：layer1~layer4。ResNet18 的 4 个 layer 大同小异，因此以 layer2 为例（layer1 有点特殊，后文会讲）：

```
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
```

这只是网络的结构，跟实际处理时的流程还有一点点区别，因此下面的流程图更为准确：



以上流程图表示的是一个 BasicBlock 的处理过程，与直接 print 出来的结构相比，主要是最后相加之后还会有一个 relu，不知道为什么 print 出来的网络结构不显示这个 relu……

此外需要注意的是 downsample 的处理。根据前面 part2 的 ResNet 结构图可以发现，ResNet18 中，每经过两个 BasicBlock，output 的尺寸（维度）会降低为原来的一半（例如从 64*56*56 到 128*28*28）。而如果如果上一个 BasicBlock 的输出维度和当前的 BasicBlock 的维度不一样，那么上一个 BasicBlock 的输出 x（同时也是当前 BasicBlock 的输入）是不能和当前计算出的 output 直接相加的，这时就需要对 x 进行 downsample 处理。如果维度一致，例如 layer1 的 output 必然和最初的 input 一致；或者在后面的 layer 中，第二个 BasicBlock 维度必然与上一个 BasicBlock 一致，这种情况下直接相加就行了，out+=x。

经过 4 个 layer 后，就是网络输出部分：

```
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=512, out_features=1000, bias=True)
```

网络输出部分很简单，通过全局自适应平滑池化（avgpool），把所有的特征图拉成 1*1，对于 ResNet18 来说，就是 1x512x7x7 的输入数据拉成 1x512x1x1，即对其中每 7x7=49 个数字做一次平均，然后接全连接层输出，输出节点个数与预测类别个数一致。

关于这个 avgpool：传统的 CNN 中（例如最早的 LeNet），在网络的输出部分使用了两个 FC（全连接）层，但 FC 层的参数实在太多了，因此之前有研究者意识到了这一点，并把其中一层 FC 层更改为 pooling（池化）层，ResNet 就沿用了这种方法。池化是不需要参数的，相比于全连接层可以砍去大量的参数。对于一个 7x7 的特征图，直接池化和改用全连接层相比，可以节省将近 50 倍的参数，作用有二：一是节省计算资源，二是防止模型过拟合，提升泛化能力。

最后是困扰了我挺久的 FC 层：其作用就是把特征 representation 整合到一起，输出为一个值。正常情况下的 FC 层，例如 VGG-16 全连接层中，对 224x224x3 的输入，最后一层卷积可得输出为 7x7x512，如果后层是一层含 4096 个神经元的 FC，那么可以用卷积核为 7x7x512x4096 的全局卷积来实现这一全连接运算过程。而 ResNet 中，FC 层前面是 avgpool 层，已经将 7x7x512 的卷积输出变成了 1 x512 的池化输出，此时应该对这个 1 x512 运用全局卷积，由于 imagenet 的输出是 1000 类，因此就是用了 1000 个 1 x512 的卷积，得到最后的 1 x1000 的输出结果。

至于 FC 层的具体操作，我查看代码发现其核心在于 torch.nn.Parameter()这个函数。对于 self.v = torch.nn.Parameter(torch.FloatTensor(hidden_size))这个用法，首先可以把这个函数理解为类型转换函数，将一个不可训练的类型 Tensor 转换成可以训练的类型 parameter 并将这个 parameter 绑定到这个 module 里面(net.parameter()中就有这个绑定的 parameter，所以在参数优化的时候可以进行优化的)，所以经过类型转换这个 self.v 变成了模型的一部分，成为了模型中根据训练可以改动的参数了。使用这个函数的目的也是想让某些变量在学习的过程中不断的修改其值以达到最优化。也就是说，FC 层的全局卷积操作的具体参数，都是系统自己训练出来的，属于我们看不到的部分。

经过以上流程，ResNet 的处理就全部结束了。对于 imagenet 数据集，最终的输出是 FC 层的 1 x1000 的 output。

在这里还是贴一下具体的代码，左边是 ResNet18 的整体流程（分为 输入-主体-输出 三个部分），右边是每个 BasicBlock 的流程（核心代码是 out += identity 这一句，这就是 resnet 的中心思想）：

```
x = self.conv1(x)
x = self.bn1(x)
x = self.relu(x)
x = self.maxpool(x)

x = self.layer1(x)
x = self.layer2(x)
x = self.layer3(x)
x = self.layer4(x)

x = self.avgpool(x)
x = torch.flatten(x, 1)
x = self.fc(x)
```

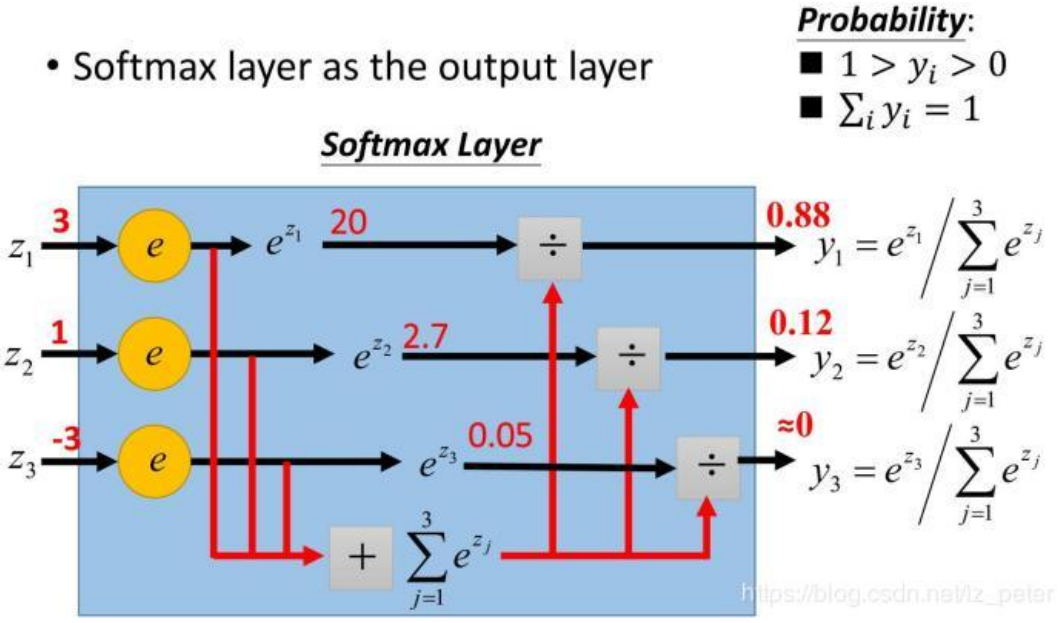
```
identity = x

out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)

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

if self.downsample is not None:
    identity = self.downsample(x)
out += identity
out = self.relu(out)
```

最后，当得到了 FC 层的输出时，还需要根据这个(1, 1000)的 output 得到图像分类的具体结果，这里使用了 softmax 函数（也叫归一化指数函数），计算过程如下图所示：



对于(1, 1000)的 output 来说，也就是先分别对这 1000 个数字取指数，y=exp(x)，将模型的预测结果转化到指数函数上，这样保证了概率的非负性。然后对转换后得到的 1000 个新数字进行归一化处理，目的是确保各个预测结果的概率之和等于 1；具体方法是将转化后的结果除以所有转化后结果之和，可以理解为转化后结果占总数的百分比，这样就得到近似的概率。最后选择概率最大的数字，其对应位置的标签就是该图像的分类结果。