薰风读论文：Deep Residual Learning 手把手带你理解ResNet

薰风初入弦
上海交通大学 计算机科学与技术博士在读

已关注

56 人赞同了该文章

> 这是薰风读论文的第2篇投稿，除了论文本身更多的是对文章思路的梳理和个人的思考。欢迎讨论~也很好奇你是喜欢只讲论文，还是喜欢多写讨论？

一、引言：为什么会有ResNet？ Why ResNet?

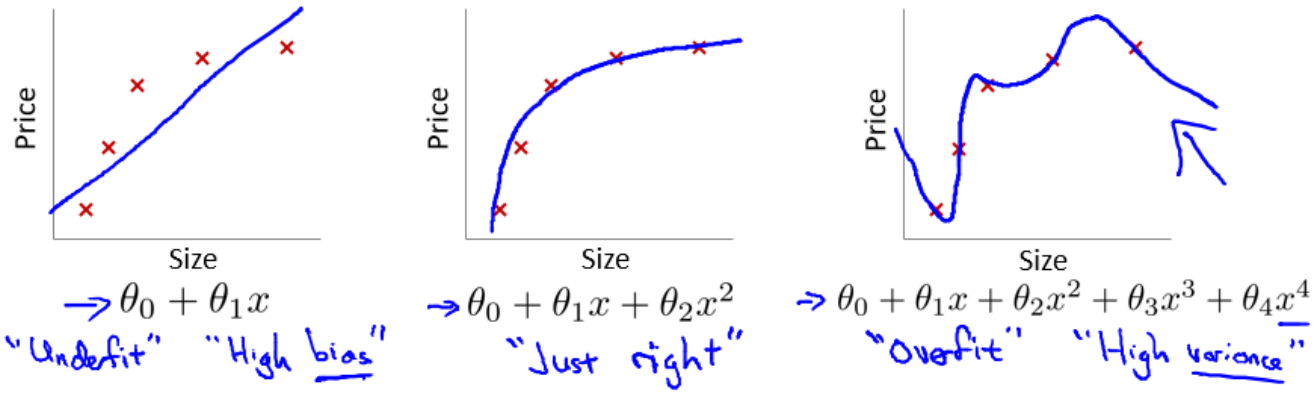
> 神经网络叠的越深，则学习出的效果就一定会越好吗？

答案无疑是否定的，人们发现当模型层数增加到某种程度，模型的效果将会不升反降。也就是说，深度模型发生了退化（degradation）情况。

那么，为什么会出现这种情况？

1. 过拟合？ Overfitting?

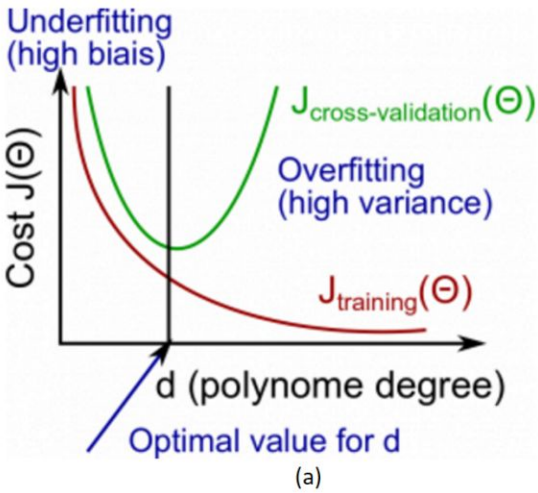
首先印入脑海的就是Andrew Ng机器学习公开课[1]的过拟合问题



Andrew Ng的课件截图

在这个多项式回归问题中，左边的模型是欠拟合（under fit）的此时有很高的偏差（high bias），中间的拟合比较成功，而右边则是典型的过拟合（overfit），此时由于**模型过于复杂**，导致了高方差（high variance）。

然而，很明显当前CNN面临的效果退化不是因为过拟合，因为过拟合的现象是"高方差，低偏差"，即测试误差大而训练误差小。但实际上，深层CNN的训练误差和测试误差都很大。



(a)

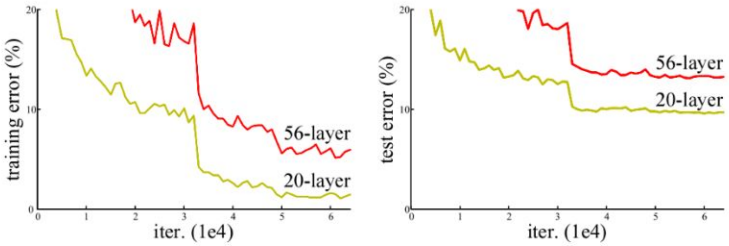


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

(b)

（a）欠拟合与过拟合 （b）模型退化

2. 梯度爆炸/消失? Gradient Exploding/Vanishing?

除此之外，最受人认可的原因就是“梯度爆炸/消失（弥散）”了。为了理解什么是梯度弥散，首先回顾一下反向传播的知识。

假设我们现在需要计算一个函数 $f(x,y,z)=(x+y)\times z$ ， $x=-2$, $y=5$, $z=-4$ 在时的梯度，那么首先可以做出如下所示的计算图。

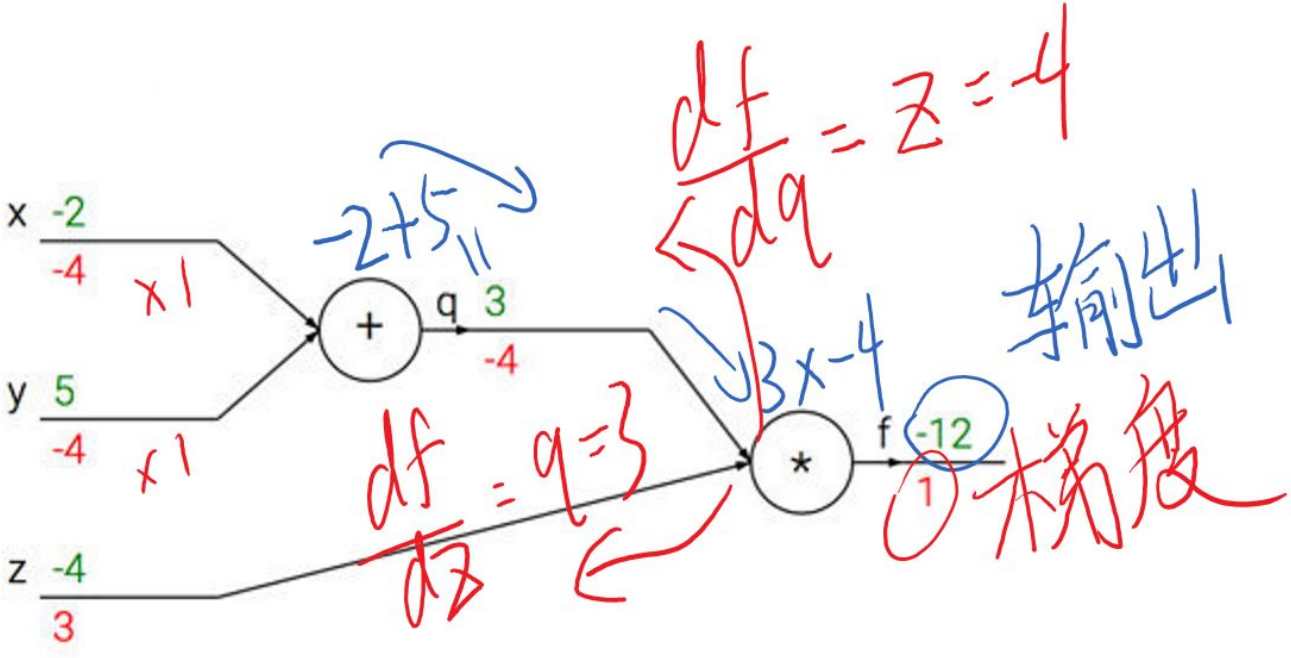
将 $x=-2$, $y=5$, $z=-4$ 带入，其中，令 $x+y=q$,一步步计算，很容易就能得出 $f(-2,5,-4)=-12$ 。

这就是前向传播（计算图上部分绿色打印字体与蓝色手写字体），即：

$$\begin{cases} f=q\cdot z \\ q=x+y \end{cases}$$

前向传播是从输入一步步向前计算输出，而反向传播则是从输出反向一点点推出输入的梯度(计算图下红色的部分)。

$$\begin{cases} \dfrac{df}{dq}=z \\ \dfrac{df}{dz}=q \\ \dfrac{df}{dx}=\dfrac{df}{dq}\cdot\dfrac{dq}{dx}, \dfrac{df}{dy}=\dfrac{df}{dq}\cdot\dfrac{dq}{dy} \end{cases}$$



原谅我字丑……

注：这里的反向传播假设输出端接受之前回传的梯度为1（也可以是输出对输出求导=1）

观察上述反向传播，不难发现，在输出端梯度的模值，经过回传扩大了3~4倍。

这是由于反向传播结果的**数值大小**不止取决于求导的式子，很大程度上也取决于**输入的模值。**当计算图每次输入的模值都大于1，那么经过很多层回传，梯度将不可避免地呈几何倍数增长（每次都变成3~4倍，重复上万次，想象一下310000有多大……），直到Nan。这就是梯度爆炸现象。

当然反过来，如果我们每个阶段输入的模恒小于1，那么梯度也将不可避免地呈几何倍数下降（比如每次都变成原来的三分之一，重复一万次就是3-10000），直到0。这就是梯度消失现象。值得一提的是，由于人为的参数设置，梯度更倾向于消失而不是爆炸。

由于至今神经网络都以反向传播为参数更新的基础，所以梯度消失问题听起来很有道理。然而，事实也并非如此，至少不止如此。

我们现在无论用Pytorch还是Tensorflow，都会自然而然地加上Bacth Normalization(简称BN)，**而BN的作用本质上也是控制每层输入的模值，**因此梯度的爆炸/消失现象理应在很早就被解决了（至少解决了大半）。

不是过拟合，也不是梯度消失，这就很尴尬了……CNN没有遇到我们熟知的两个老大难问题，却还是随着模型的加深而导致效果退化。无需任何数学论证，我们都会觉得这不符合常理。等等，不符合常理……

3. 为什么模型退化不符合常理?

按理说，当我们堆叠一个模型时，理所当然的会认为效果会越堆越好。因为，假设一个比较浅的网络已经可以达到不错的效果，**那么即使之后堆上去的网络什么也不做，模型的效果也不会变差。**

然而事实上，这却是问题所在。“什么都不做”恰好是当前神经网络最难做到的东西之一。

MobileNet V2的论文[2]也提到过类似的现象，由于非线性激活函数Relu的存在，每次输入到输出的过程都几乎是不可逆的（信息损失）。我们很难从输出反推回完整的输入。

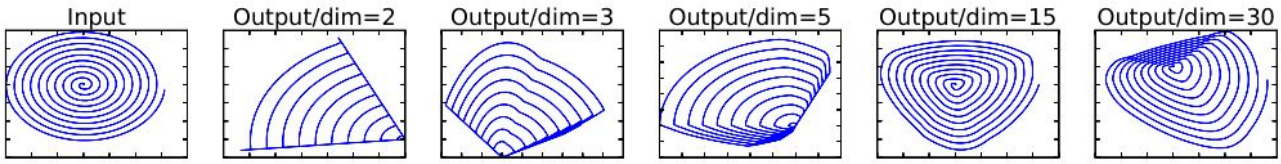


Figure 1: Examples of ReLU transformations of low-dimensional manifolds embedded in higher-dimensional spaces. In these examples the initial spiral is embedded into an n-dimensional space using random matrix T followed by ReLU, and then projected back to the 2D space using T^{-1}. In examples above $n = 2, 3$ result in information loss where certain points of the manifold collapse into each other, while for $n = 15$ to 30 the transformation is highly non-convex.

Mobilenet v2是考虑的结果是去掉低维的Relu以保留信息

也许赋予神经网络无限可能性的“非线性”让神经网络模型走得太远，却也让它忘记了为什么出发（想想还挺哲学）。这也使得特征随着层层前向传播得到完整保留（什么也不做）的可能性都微乎其微。

用学术点的话说，这种神经网络丢失的“不忘初心”/“什么都不做”的品质叫做**恒等映射（identity mapping）**。

因此，可以认为Residual Learning的初衷，其实是让模型的内部结构至少有恒等映射的能力。以保证在堆叠网络的过程中，网络至少不会因为继续堆叠而产生退化！

二、深度残差学习 Deep Residual Learning

1. 残差学习 Residual Learning

前面分析得出，如果深层网络后面的层都是是**恒等映射**，那么模型就可以转化为一个浅层网络。那现在的问题就是**如何得到恒等映射**了。

事实上，已有的神经网络很难拟合潜在的恒等映射函数H(x) = x。

但如果把网络设计为**H(x) = F(x) + x，即直接把恒等映射作为网络的一部分**。就可以把问题转化为**学习一个残差函数F(x) = H(x) - x.**

只要F(x)=0，就构成了一个恒等映射H(x) = x。 而且，拟合残差至少比拟合恒等映射容易得多。

于是，就有了论文[3]中的Residual block结构

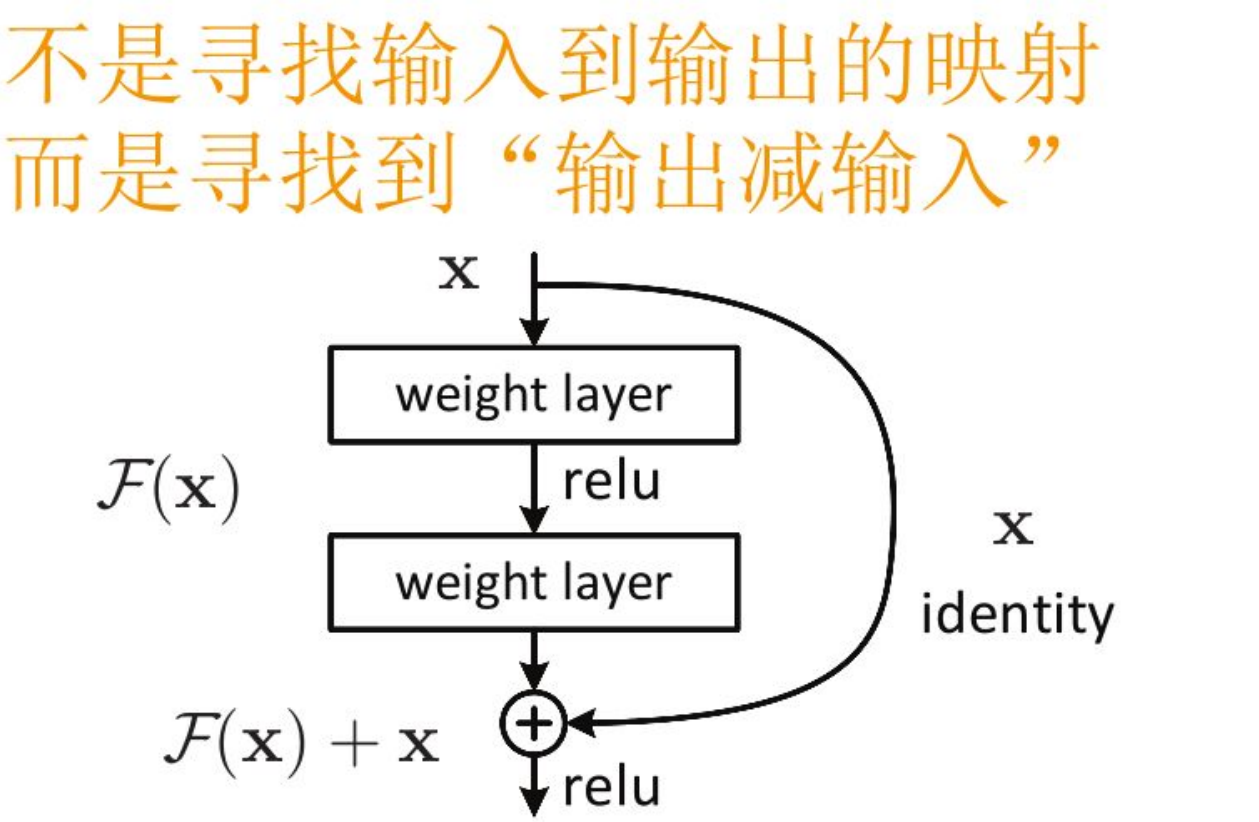


Figure 2. Residual learning: a building block.

Residual Block的结构

图中右侧的曲线叫做跳接（shortcut connection），通过跳接在**激活函数前，**将上一层（或几层）**之前的输出与本层**计算的**输出相加**，将求和的结果输入到激活函数中做为本层的输出。

用数学语言描述，假设Residual Block的输入为 $\boldsymbol{x}$ ，则输出 $\boldsymbol{y}$ 等于：

$$\mathbf{y} = \mathcal{F}(x, \{W_i\}) + x$$

其中 $\boldsymbol{\mathcal{F}(x, \{W_i\})}$ 是我们学习的目标，即输出输入的残差 $\boldsymbol{y - x}$ 。以上图为例，残差部分是中间有一个Relu激活的双层权重，即：

$$\mathcal{F} = W_2\sigma(W_1 x)$$

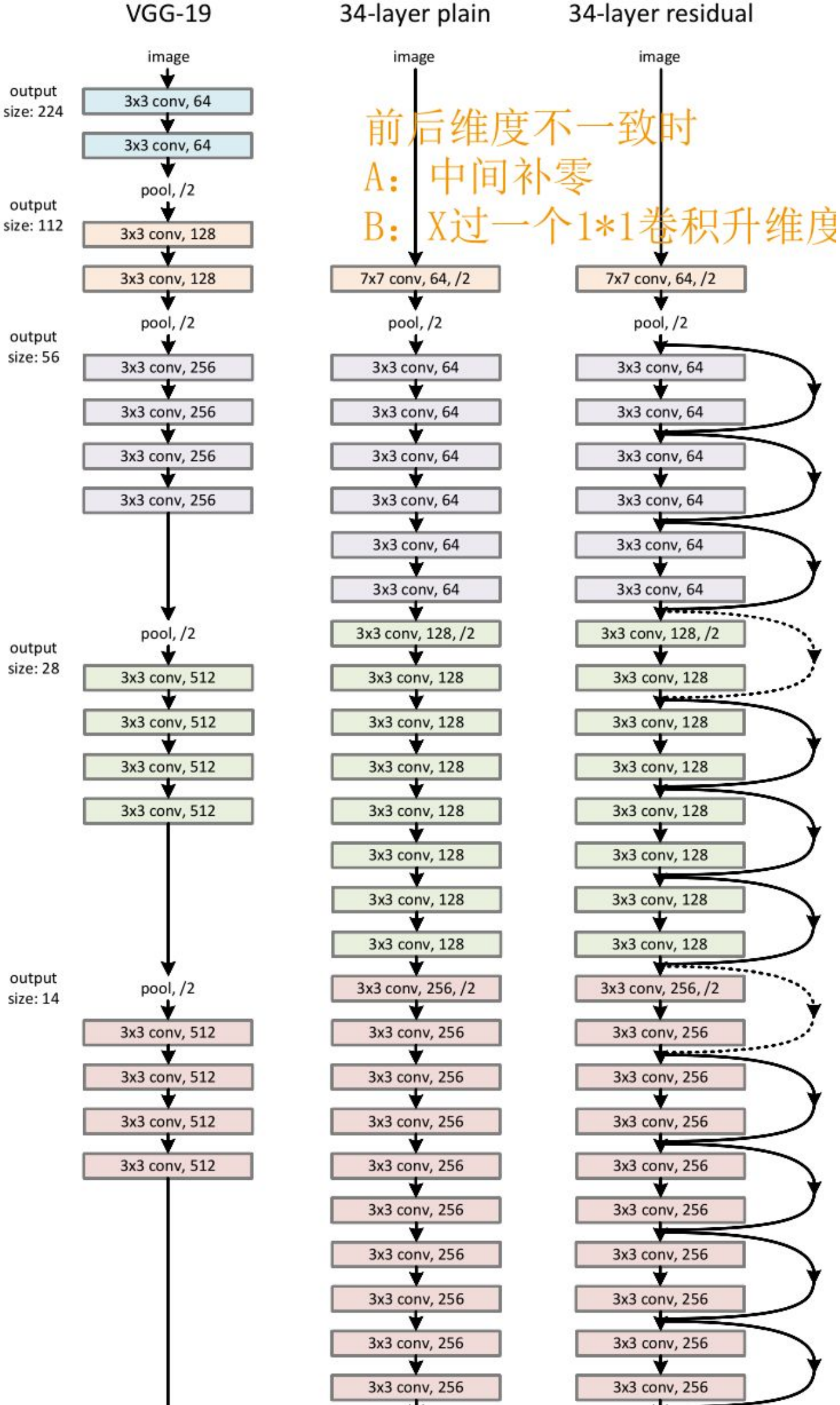
其中 σ 指代Relu，而 W_1, W_2 指代两层权重。

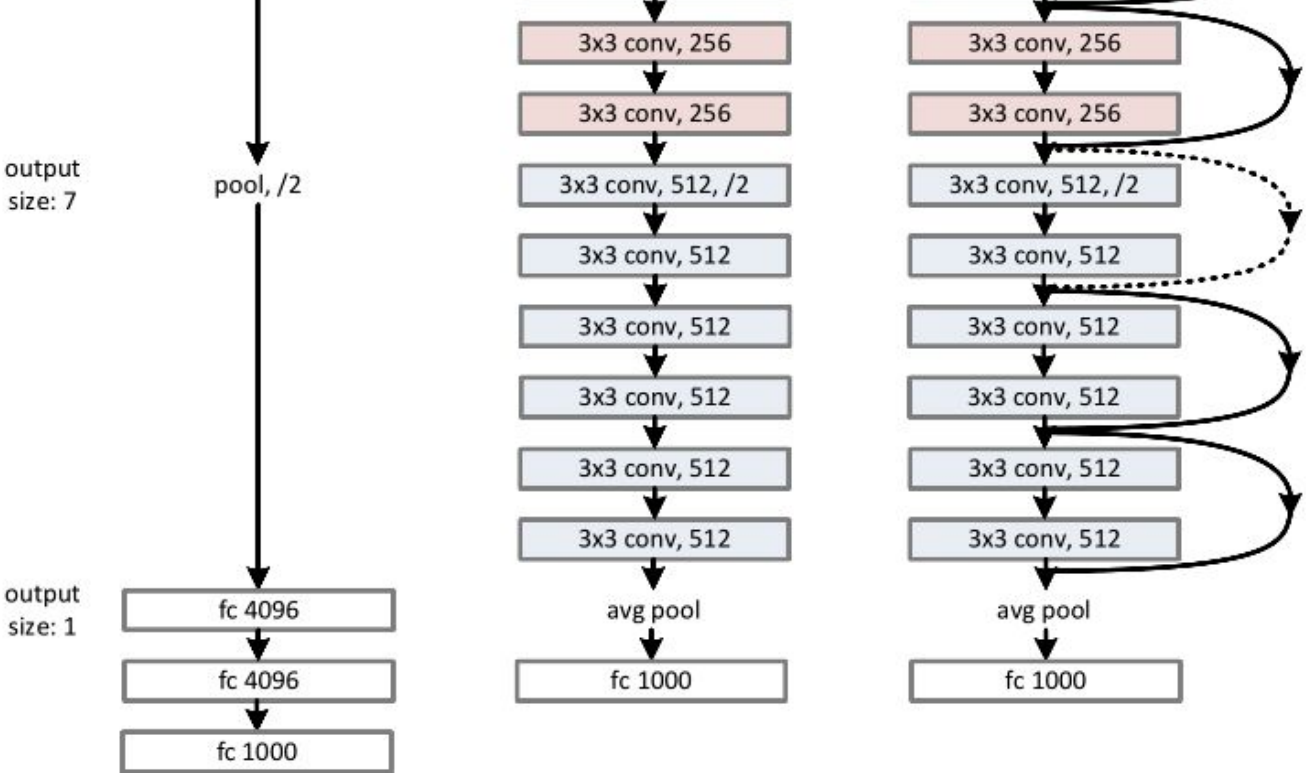
顺带一提，这里一个**Block中必须至少含有两个层**，否则就会出现很滑稽的情况：

$$\mathbf{y} = \mathcal{F}(x, \{W_i\}) + x = (W_1 x) + x = (W_1 + 1)x$$

显然这样加了和没加差不多……

2.**网络结构与维度问题**


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
7x7 conv, 64, /2
3x3 conv, 128, /2
3x3 conv, 256, /2
前后维度不一致时
A：中间补零
B：X过一个1*1卷积升维度



ResNet结构示意图（左到右分别是VGG，没有残差的PlainNet，有残差的ResNet）

论文中原始的ResNet34与VGG的结构如上图所示，可以看到即使是当年号称“Very Deep”的VGG，和最基础的Resnet在深度上相比都是个弟弟。

可能有好奇心宝宝发现了，跳接的曲线中大部分是实现，但也有少部分虚线。这些虚线的代表这些Block前后的维度不一致，因为去掉残差结构的Plain网络还是参照了VGG经典的设计思路：**每隔x层，空间上/2（下采样）但深度翻倍。**

也就是说，**维度不一致体现在两个层面**：

空间上不一致
深度上不一致

空间上不一致很简单，只需要在跳接的部分给输入x加上一个线性映射 W_s ，即：

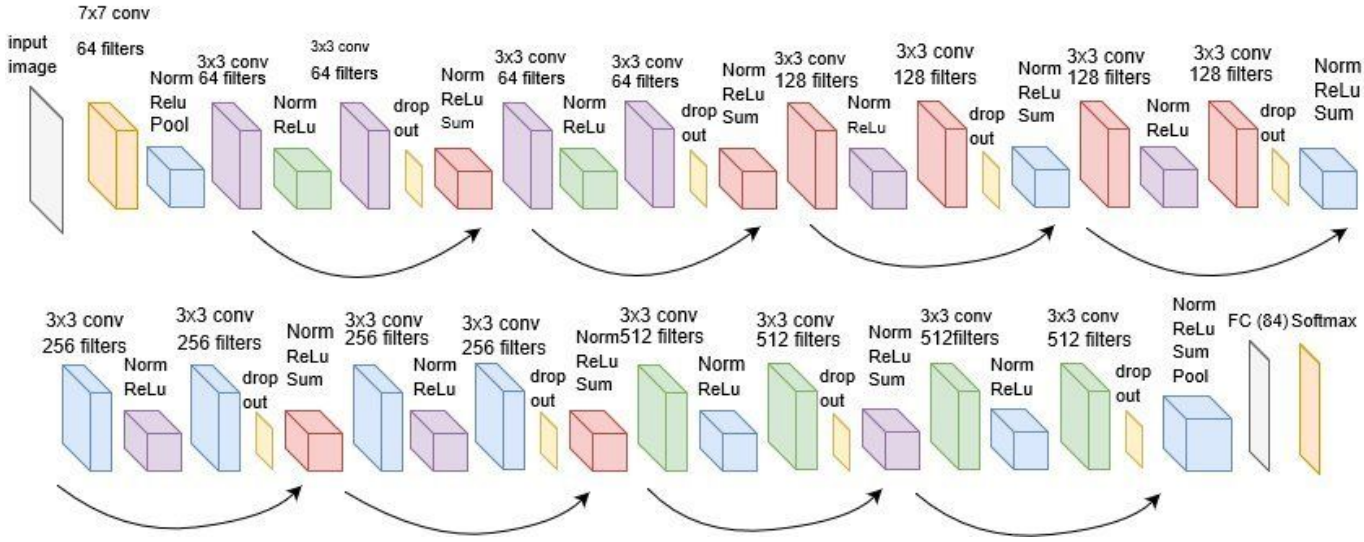
$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x} \quad \rightarrow \quad \mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x}$$

而对于**深度上**的不一致，则有两种解决办法，一种是在跳接过程中加一个1*1的卷积层进行升维，另一种则是直接简单粗暴地补零。事实证明两种方法都行得通。

注：深度上和空间上维度的不一致是分开处理的，但很多人将两者混为一谈（包括目前某乎一些高赞文章），这导致了一些人在模型的实现上感到困惑（比如当年的我）。

3. torchvision中的官方实现

事实上论文中的ResNet并不是最常用的，我们可以在Torchvision的模型库中找到一些很不错的例子，这里拿Resnet18为例：



运行代码：

```
import torchvision
model = torchvision.models.resnet18(pretrained=False) #我们不下载预训练权重
print(model)
```

得到输出：

```
ResNet(
  (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=Tru
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
```

```
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (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_stat
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  )
)
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bia
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  )
)
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bia
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    )
  )
  (1): BasicBlock(
```

```
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

薰风说 Thinkings

上述的内容是我以自己的角度思考作者提出ResNet的心路历程，我比作者蔡很多，所以难免出现思考不全的地方。

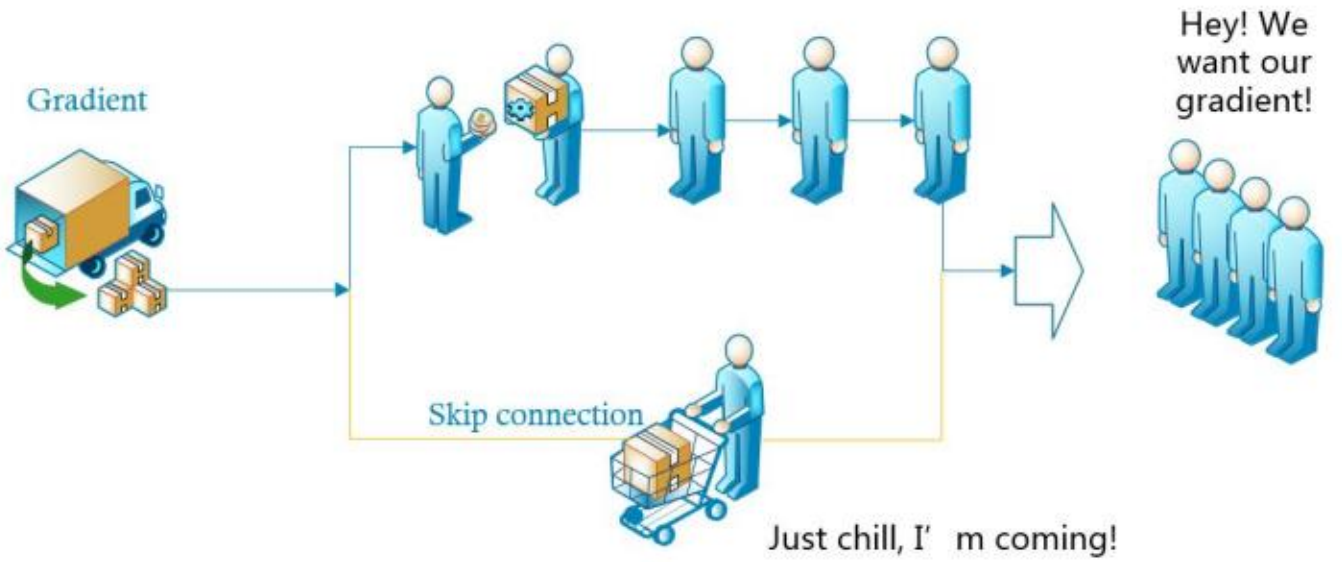
ResNet是如此简洁高效，以至于模型提出后还有无数论文讨论“ResNet到底解决了什么问题(The Shattered Gradients Problem: If resnets are the answer, then what is the question?)”[4]

论文[4]认为，即使BN过后梯度的模稳定在了正常范围内，但**梯度的相关性实际上是随着层数增加持续衰减的**。而经过证明，ResNet可以有效减少这种相关性的衰减。

对于 L 层的网络来说，没有残差表示的Plain Net梯度相关性的衰减在 $\frac{1}{2^L}$ ，而ResNet的衰减却只有 $\frac{1}{\sqrt{L}}$ 。这也验证了ResNet论文本身的观点，网络训练难度随着层数增长的速度不是线性，而至少是多项式等级的增长（如果该论文属实，则可能是指数级增长的）

而对于“梯度弥散”观点来说，在输出引入一个输入x的恒等映射，则梯度也会对应地引入一个常数1，这样的网络的确不容易出现梯度值异常，在某种意义上，起到了稳定梯度的作用。

除此之外，shortcut类似的方法也并不是第一次提出，之前就有“Highway Networks”。可以只管理解为，以往参数要得到梯度，需要快递员将梯度一层一层中转到参数手中（就像我取个快递，都显示要从“上海市”发往“闵行分拣中心”，闵大荒日常被踢出上海籍）。而跳接实际上给梯度开了一条“高速公路”（取快递可以直接用无人机空投到我手里了），效率自然大幅提高，不过这只是个比较想当然的理由。



上面的理解很多论文都讲过，但我个人最喜欢下面两个理解。

第一个已经由Feature Pyramid Network[5]提出了，那就是跳连接相加可以实现不同分辨率特征的组合，因为浅层容易有高分辨率但是低级语义的特征，而深层的特征有高级语义，但分辨率就很低了。

第二个理解则是说，引入跳接实际上让模型自身**有了更加“灵活”的结构**，即在训练过程本身，模型可以选择在每一个部分是“更多进行卷积与非线性变换”还是“更多倾向于什么都不做”，抑或是将两者结合。模型在训练便可以自适应本身的结构，这听起来是多么酷的一件事啊！

有的人也许会纳闷，我们已经知道一个模型的来龙去脉了，那么在一个客观上已经十分优秀的模型，强加那么多主观的个人判断有意思吗？

然而笔者还是相信，更多角度的思考有助于我们发现现有模型的不足，以及值得改进的点。比如我最喜欢的两个理解就可以引申出这样的问题“虽然跳接可以结合不同分辨率，但ResNet显然没有充分利用这个优点，因为每个shortcut顶多跨越一种分辨率（大部分还不会发生跨越）”。

那么“如果用跳接组合更多分辨率的特征，模型的效果会不会更好？”这就是DenseNet回答我们的问题了。