

# Contents

<b>1</b>	<b>1. Case studies</b>	<b>1</b>
1.1	1.1 Why look at case studies? . . . . .	1
1.2	1.2 Classic Networks . . . . .	2
1.2.1	1.2.1 LeNet-5 (1998 年) . . . . .	2
1.2.2	1.2.2 AlexNet (2012 年) . . . . .	3
1.2.3	1.2.3 vgg-16(2015 年) . . . . .	4
1.3	1.3 ResNets . . . . .	5
1.4	1.4 Why ResNets Work . . . . .	7
1.5	1.5 Networks in Networks and 1x1 Convolutions (2013 年) . . . . .	7
1.6	1.6 Inception Network Motivation (2014 年) . . . . .	8
1.7	1.7 Inception Network . . . . .	11
<b>2</b>	<b>2. Practical advices for using ConvNets</b>	<b>12</b>
2.1	2.1 Using Open-Source Implementation . . . . .	12
2.2	2.2 Transfer Learning . . . . .	12
2.2.1	2.2.1 如果训练集较小 . . . . .	12
2.2.2	2.2.2 如果训练集很大 . . . . .	12
2.2.3	2.2.3 如果训练集特别大 . . . . .	13
2.3	2.3 Data Augmentation . . . . .	13
2.4	2.4 State of Computer Vision . . . . .	14

contents

- 1. Case studies
  - 1.1 Why look at case studies?
  - 1.2 Classic Networks
    - 1.2.1 LeNet-5 (1998 年)
    - 1.2.2 AlexNet (2012 年)
    - 1.2.3 vgg-16(2015 年)
  - 1.3 ResNets
  - 1.4 Why ResNets Work
  - 1.5 Networks in Networks and 1x1 Convolutions (2013 年)
  - 1.6 Inception Network Motivation (2014 年)
  - 1.7 Inception Network
- 2. Practical advices for using ConvNets
  - 2.1 Using Open-Source Implementation
  - 2.2 Transfer Learning
    - 2.2.1 如果训练集较小
    - 2.2.2 如果训练集很大
    - 2.2.3 如果训练集特别大
  - 2.3 Data Augmentation
  - 2.4 State of Computer Vision

## 1 1. Case studies

### 1.1 1.1 Why look at case studies?

其中，ResNet 可以达到 152 层

# Outline

## Classic networks:

- LeNet-5 [←](#)
- AlexNet [←](#)
- VGG [←](#)

ResNet [\(152\)](#)

Inception



Figure 1: case-study-outline.png

## 1.2 1.2 Classic Networks

### 1.2.1 1.2.1 LeNet-5 (1998 年)

- 因为输入是灰度图，所以 channel=1，维度是

$$n_H^{[0]} \times n_W^{[0]} \times n_C^{[0]} = 32 \times 32 \times 1$$

- stride=1，有 6 个 filters，每个是 5x5，所以输出是，

$$n_H^{[1]} \times n_W^{[1]} \times n_C^{[1]} = \left(\frac{(32 - 5 + 2 \times 0)}{1} + 1\right) \times \left(\frac{(32 - 5 + 2 \times 0)}{2} + 1\right) \times 6 = 28 \times 28 \times 6$$

- 接下来接的是 avg pooling(当时 avg 很流行，后来大部分是 max pooling 了)，2x2 的 filter，stride=2，而 pooling 的 filter 并没有第三维，输入和输出的第三维是一样的。所以输出是，

$$\left(\frac{(28 - 2 + 2 \times 0)}{2} + 1\right) \times \left(\frac{(28 - 2 + 2 \times 0)}{2} + 1\right) \times 6 = 14 \times 14 \times 6$$

- 接下来接 16 个 5x5(应该是 5x5x6，maybe 是同一个 5x5，给 6 个 channel 用，所以简记为 5x5) 的 filter，stride=1，所以：

$$\left(\frac{14 - 5 + 2 \times 0}{1} + 1\right) \times \left(\frac{14 - 5 \times 0}{1} + 1\right) \times 16 = 10 \times 10 \times 16$$

- 然后接一个 avg pooling，2x2 的，输出是

$$\left(\frac{10 - 2 + 2 \times 0}{2} + 1\right) \times \left(\frac{10 - 2 + 2 \times 0}{2} + 1\right) \times 16 = 5 \times 5 \times 16$$

- 然后把这 5x5x16=400 个神经元 flatten，变成一个 400 维的向量，再接一个 120 维的 fc，再接一个 84 维的 fc。

- 目标是一个 10 分类的结果，现代的方法是用一个 10 维的 softmax，而在当时，用的是 gaussian connections，现在已经不用啦。

小结：

- lenet-5 总共有大约 60k 的参数。
- 因为都没有用 padding，所以随着层数的增加，height 和 width 都会变小。
- channels 是增加的。
- 形成了 pattern: conv+pool+conv+pool+fc+fc+output

看这篇论文时可以关注的点

- 当时的激活函数还是 sigmoid/tanh
- 由于当时的计算速度比较慢，对  $n_H \times n_W \times n_C$  的输入，使用  $f \times f \times n_C$  的 filter，每个 filter 只对对应的通道起作用
- 当时在 pooling 后会接非线性

可以关注论文的 section II(网络结构) 和 section III (实验和结论)。

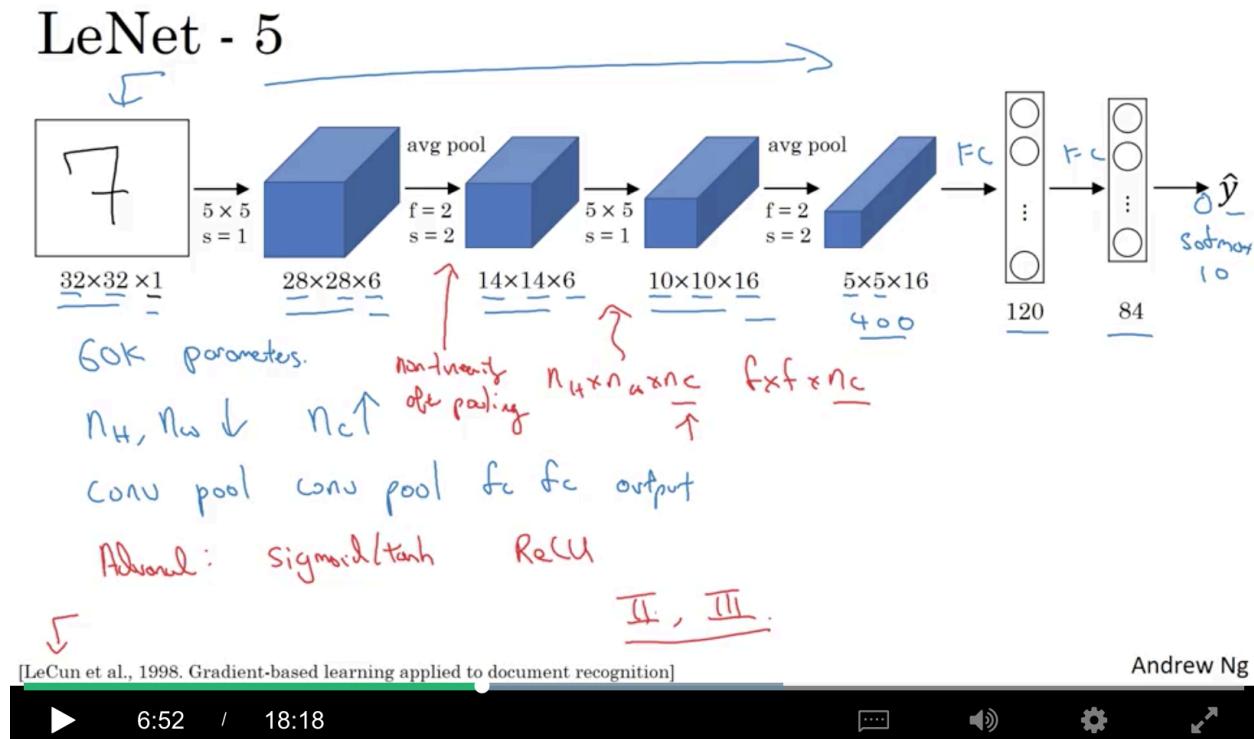


Figure 2: paper-lenet-5.png

### 1.2.2 1.2.2 AlexNet (2012 年)

- 输入是 227x227x3 的图片
- 第一层是 96 个 11x11 的 filters, stride=4, 所以得到 55x55x96
- 然后接一个 3x3 的 max pooling, stride=2, 所以得到 27x27x96
- 然后经过 256 个 5x5 的卷积, 做一个 same conv, 使得得到的宽和高保持不变 [即  $(25+2p-5)/s+1=25 \Rightarrow (20+2p)/s=24$ , 可以设置 stride=1, padding=2], 得到 27x27x256。
- 然后再做一个 3x3, stride=2 的 max pooling, 得到 13x13x256。
- 再接 384 个 3x3 的卷积, 做 same conv, 得到 13x13x384
- 再接 384 个 3x3 的卷积, 做 same conv, 同样得到 13x13x384
- 再接 256 个 3x3 的卷积, 做 same conv, 同样得到 13x13x256
- 再接 3x3, stride=2 的 max pooling, 得到 6x6x256

- 然后 flatten 成一个  $6 \times 6 \times 256 = 9216$  维的向量
- 再接两个 4096 维的 fc
- 最后接一个 1000 分类的 softmax

小结：

- 结构和 lenet-5 相似，但参数上，lenet-5 只有 60k 即 6w 的参数，而 alexnet 有 60m 即 6kw 的参数
- 使用了 relu

看这篇论文时可以关注的点：

- 当时的 gpu 比较慢，所以当时用很复杂的方式在 2 个 gpu 上进行训练。大概的思想就是把一些 layer 拆开到不同 gpu 上进行训练和通信。
- 使用了一些特殊的层，例如 local response normalization(LRN)。大概思想是，例如针对一个  $13 \times 13 \times 256$  的 block，看  $13 \times 13$  中的某一个点时，对这个点在 256 个通道上的值进行 normalization。动机在于，在  $13 \times 13$  上的某一个位置，或许不需要太多的神经元有太高的激活值。但后来，大多数研究者发现这么做其实没什么用...

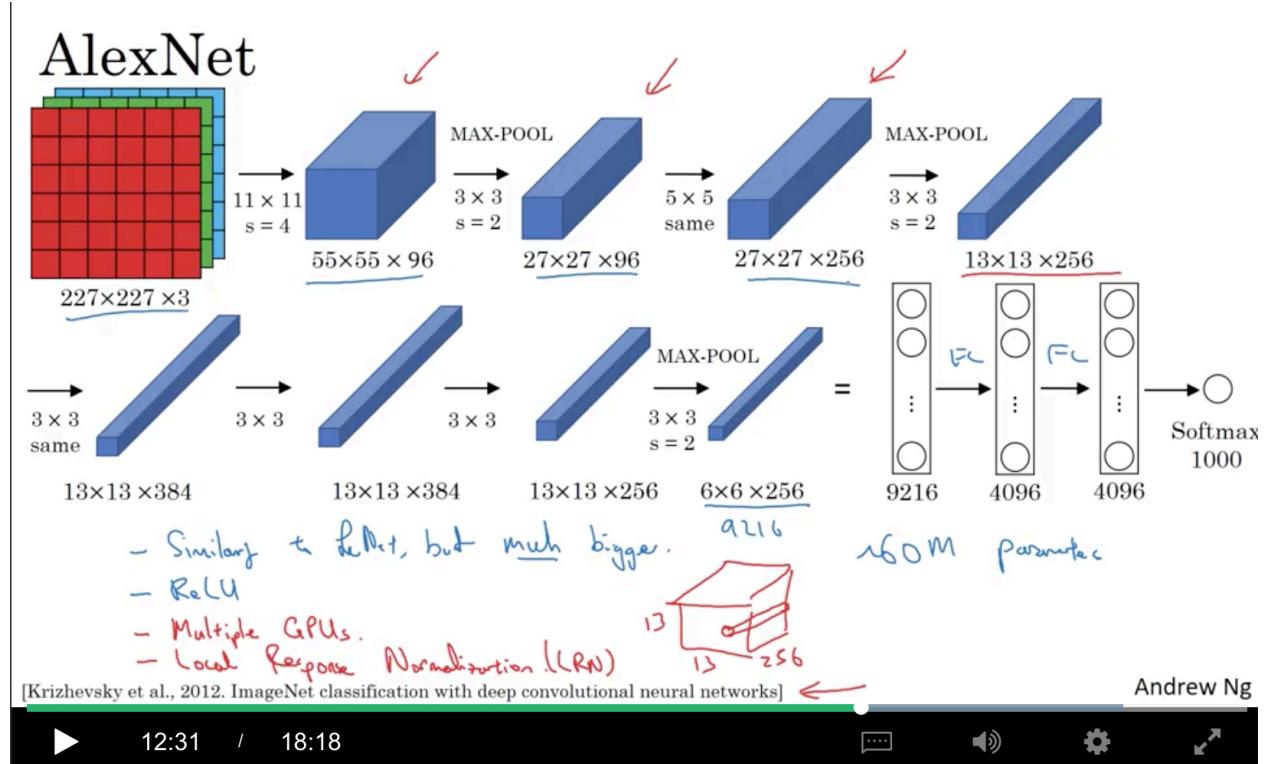


Figure 3: paper-alexnet.png

### 1.2.3 1.2.3 vgg-16(2015 年)

CONV 都是  $3 \times 3$  的 filter, stride=1, same padding MAX-POOL 都是  $2 \times 2$ , stride=2(刚好可以把宽度和高度缩小一半:  $(x-2)/2+1=x/2$ )

- 输入是  $224 \times 224 \times 3$  的图片
- 然后接两个 CONV 64 层 (每层有 64 个 filter) [右上角的图]，得到  $224 \times 224 \times 64$
- 然后接一个 POOL，得到  $112 \times 112 \times 64$
- 再接两个 CONV 128 层，得到  $112 \times 112 \times 128$
- 再接一个 POOL，得到  $56 \times 56 \times 128$
- 再接 3 个 CONV 256，得到  $56 \times 56 \times 256$
- 再接一个 POOL，得到  $28 \times 28 \times 256$

- 再接 3 个 CONV 512，得到  $28 \times 28 \times 512$
- 再接一个 POOL，得到  $14 \times 14 \times 512$
- 再接 3 个 CONV 512，得到  $14 \times 14 \times 512$
- 再接一个 POOL，得到  $7 \times 7 \times 512$
- 然后 flatten，并接 2 个 4096 的 FC，最后接 1000 的 softmax

小结：

- 不把 POOL 当做 layer，就总共有  $2+2+3+3+3+1+1+1$ ，所以叫 vgg-16
- 可以看到，POOL 会把高度和宽度减半，而 channels 则是逐层翻倍 (64-128-256-512)
- 缺点就是参数太多，参数总量有 138M，即 13.8kw，即 1.38 亿
- 还有 vgg-19，但效果其实差不多，所以大部分人还是用 vgg-16

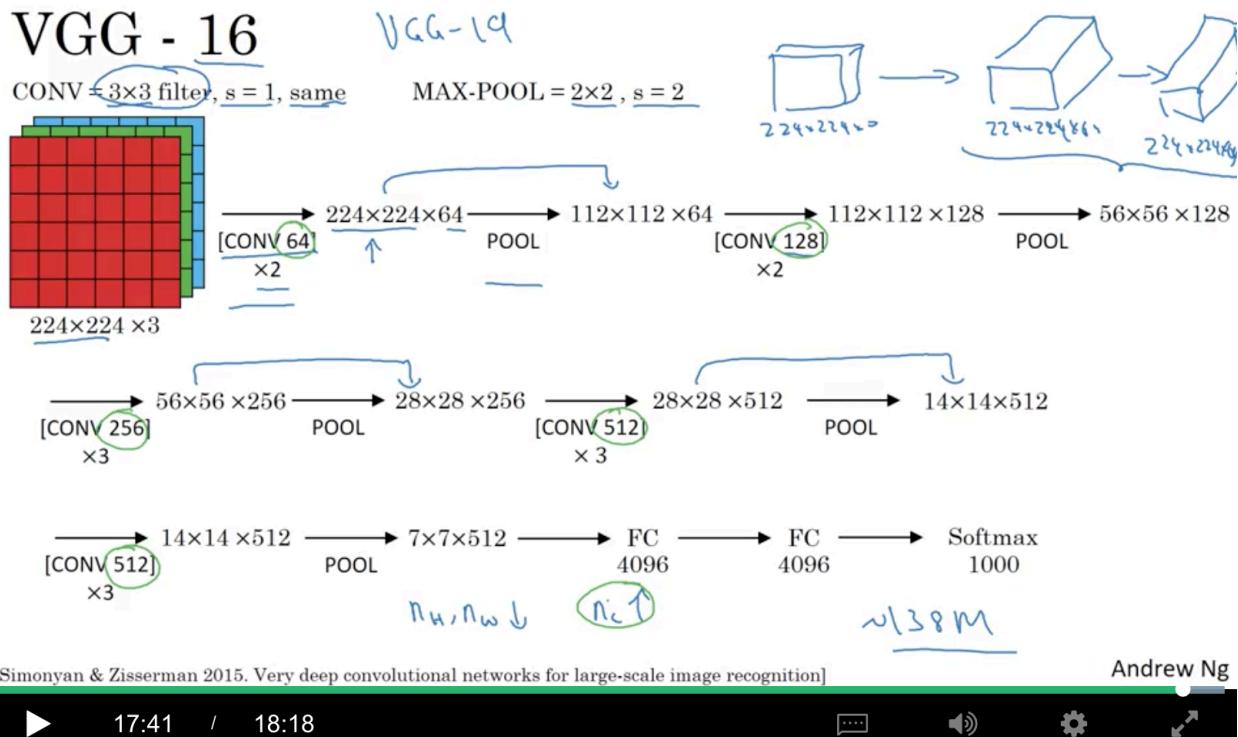


Figure 4: paper-vgg16.png

### 1.3 1.3 ResNets

核心思想是 skip connections/shortcut。

基本组成单元是 Residual Block。所谓的 Residual，就是把图中的  $a^{[l]}$  直接『加』到  $z^{[l+2]}$  上，即：

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

当然，这种 shortcut 不仅可以跨越一层，还可以跨越好几层

如下图，有 5 个 residual blocks。

理论上，层数加深，training error 会一直变小，如左图中的绿线，但实际应用中，一个很深的普通 dnn (没有残差连接)，训练误差反而会变大。而 ResNet 能让更深的网络训练效果更好，可能后面会出现平原 (怎么加深效果也不会怎么变化)。

## Residual block

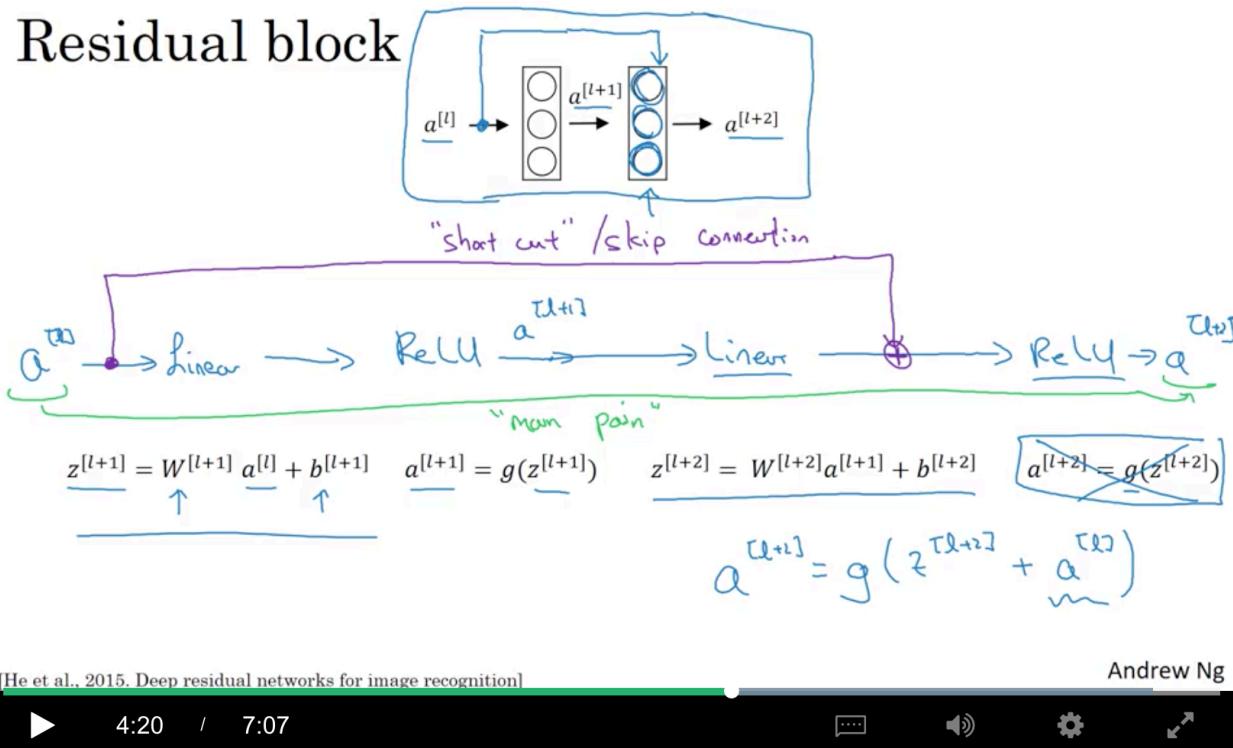
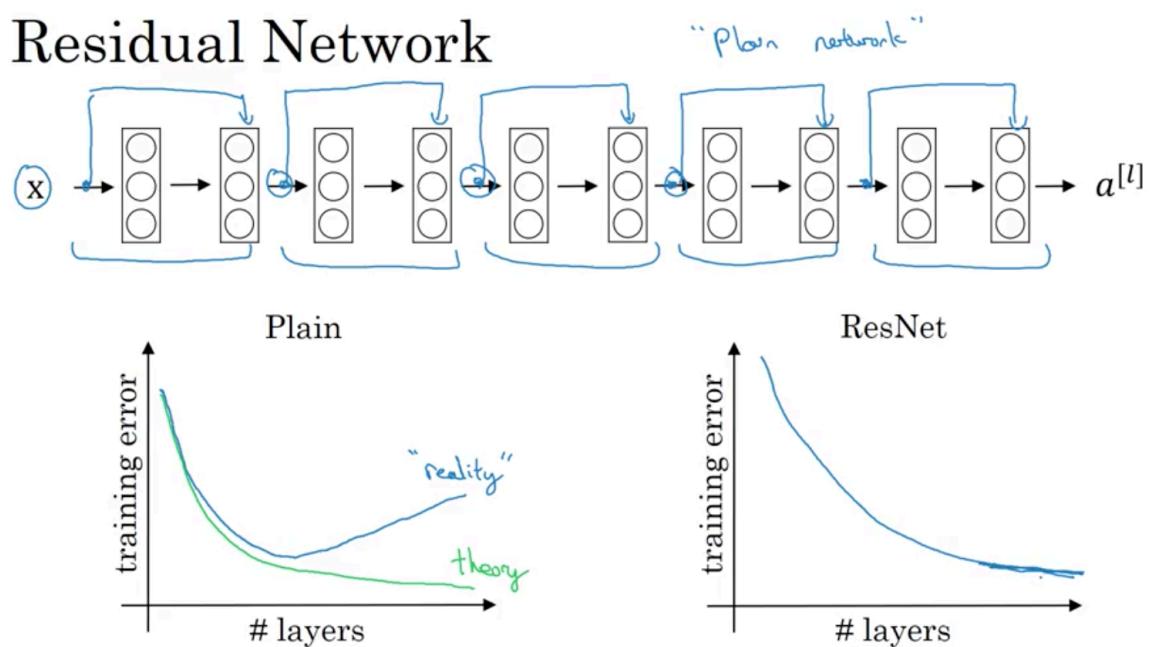


Figure 5: paper-resnet.png

## Residual Network



[He et al., 2015. Deep residual networks for image recognition]

Andrew Ng

Figure 6: paper-resnet-performance.png

## 1.4 1.4 Why ResNets Work

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(W^{[l+2]}a^{[l+1]} + b^{[l+1]} + a^{[l]})$$

如果使用 l2-regularization, 即 weight decay, 会减小  $W^{[l+2]}$ 。如果  $W^{[l+2]}$  减小到 0, 方便起见, 假设  $b^{[l+2]}$  也是 0, 那么,

$$a^{[l+2]} = g(W^{[l+2]}a^{[l+1]} + b^{[l+1]} + a^{[l]}) = g(a^{[l]})$$

如果用的 relu, 那就是  $a^{[l+2]} = a^{[l]}$ 。所以, Residual block 能够很容易地学习恒等变换。相当于这两层在神经网络中白加了, 其实就是减少了网络的深度。

另外一点需要注意的是, 这里使用了『加』, 所以要求  $z^{[l+2]}$  和  $a^{[l]}$  的维度是一样的, 所以在 resnet 中会有很多的相同 conv。

假设输入和输出维度不同, 例如  $a^{[l]}$  是 128 维的, 而  $a^{[l+2]}$  是 256 维的, 这个时候, 可以加一个  $256 \times 128$  的  $W_s$ :

$$a^{[l+2]} = g(z^{[l+2]} + W_s a^{[l]})$$

这里的  $W_s$  可以是一个可学习的矩阵, 也可以是一个固定的矩阵, 例如对  $a^{[l]}$  进行 zero-padding。

## Why do residual networks work?

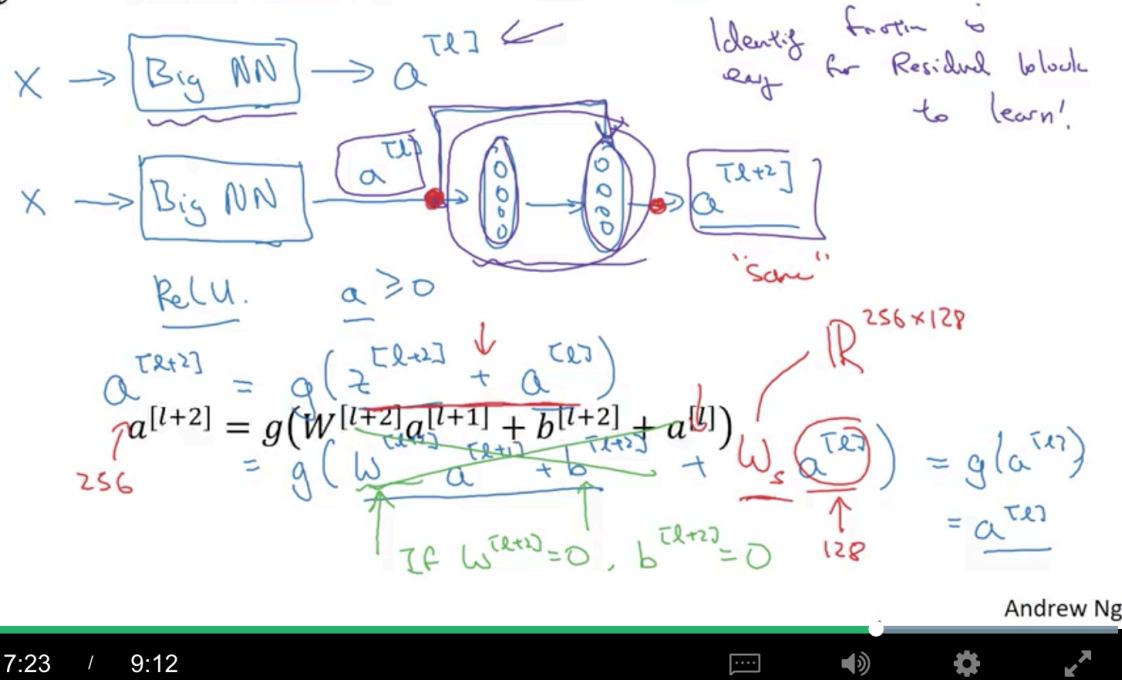


Figure 7: why-resnet-works.png

如何把一个 plain network (普通网络) 变成一个 resnet 呢? 如图中有大量的 same conv, 直接变就行了。而图中的 pool, 输入和输出维度不同, 这个时候就需要上面提到的  $W_s$  了, 当然也可以不加 residual。

## 1.5 1.5 Networks in Networks and 1x1 Convolutions (2013 年)

1x1 其实就是乘了一个数, 有啥用呢?

- 如果 channels (即  $n_C$ ) = 1, 确实只是乘了一个数

# ResNet

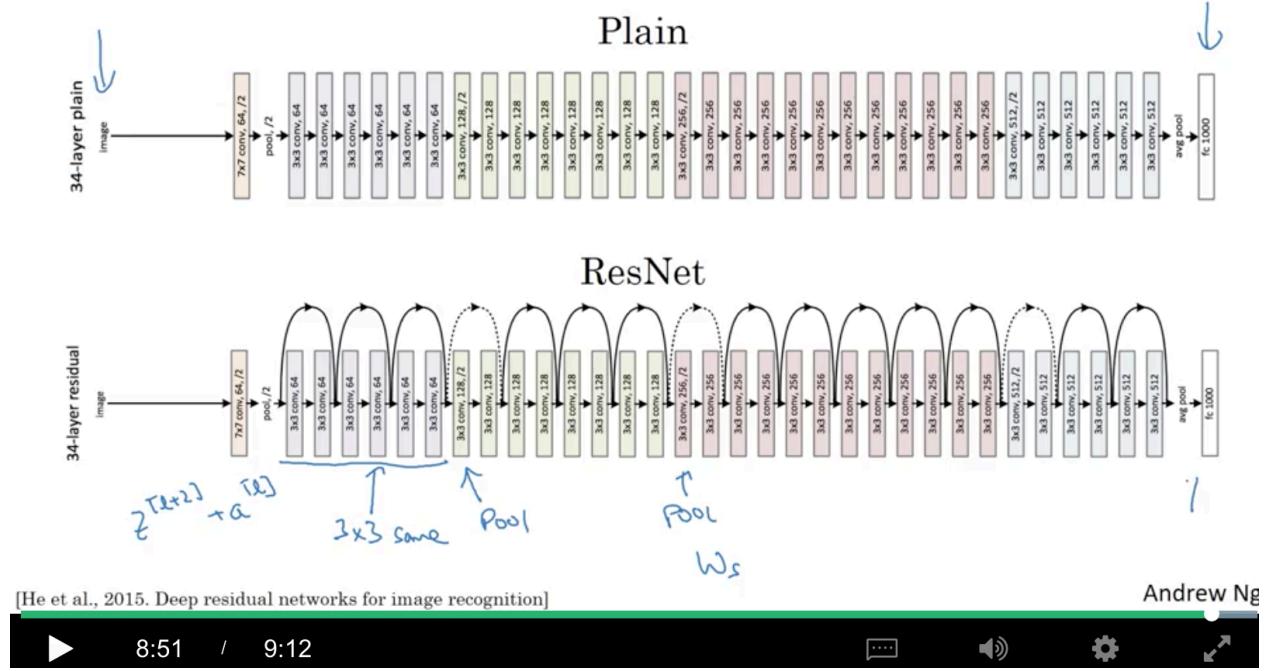


Figure 8: trans-plain-to-resnet.png

- 如果  $n_C > 1$ , 例如,  $n_C = 32$ , 那么相当于  $6 \times 6$  图像中的一个  $32$  维的点接受了一个  $32$  个数的向量, 分别对应相乘再相加, 再过一个 Relu

所以本质上,  $1 \times 1$  的卷积是一个全连接的神经网络, 逐一作用于这  $6 \times 6 = 36$  个不同的位置。这个全连接网络输入是  $32$  个数, 输出 filter 个数个值 ( $\#filters = n_C^{[l+1]}$ ), 得到  $6 \times 6 \times \#filters$  的输出。

因此,  $1 \times 1$  卷积有时也被称作 Network in Network(当然, 2013 年的这篇 paper 就叫这个名字。。)。

对于一个  $28 \times 28 \times 192$  的输入, 如果想把 height 和 width 变小, 可以直接用 pooling:

- 如何把 channels 变小呢? 可以用  $32$  个  $1 \times 1$  的卷积, 这样就得到了  $28 \times 28 \times 32$ 。 $1 \times 1$  可以减小 channels 数, 从而减小计算量
- 而如果使用  $192$  个  $1 \times 1$  卷积呢? 这个时候  $1 \times 1$  的效果就是增加非线性了

## 1.6 1.6 Inception Network Motivation (2014 年)

如果输入是  $28 \times 28 \times 192$ , + 经过  $64$  个  $1 \times 1$  之后, 得到  $28 \times 28 \times 64$ (绿色部分), + 经过  $128$  个  $3 \times 3$  的 same conv 之后, 得到  $28 \times 28 \times 128$ (蓝色), 把蓝色堆在绿色的上面。+ 同样地, 堆一个  $32$  个  $5 \times 5$  的 same conv, 得到紫色的, 堆上去 + 当然, 还可以用 MAX-POOL, 得到黄色的  $28 \times 28 \times 32$ 。当然, 为了维数能 match( $28 \times 28$ ), 需要对 pool 做 padding, 且 stride=1 + 最后得到的就是一个  $28 \times 28 \times (64+128+32+32)=28 \times 28 \times 256$

而 inception network 有个缺点, 就是计算成本问题。这是由图中的  $5 \times 5$  卷积导致的。

来看看上面说到的  $5 \times 5$  的计算成本:

每个 filter 的大小是  $5 \times 5 \times 192$ , 输出大小是  $28 \times 28 \times 32$ , 所以需要计算  $28 \times 28 \times 32$  这么多个数, 而每个数, 需要做  $5 \times 5 \times 192$  的乘法, 所以总共要做的乘法有  $28 \times 28 \times 32 \times 5 \times 5 \times 192 = 120M$ , 也就是 1.2 亿。。。而接下来, 会讲  $1 \times 1$  卷积, 能够将这个计算成本大概降低到  $1/10$  左右!!

## Why does a $1 \times 1$ convolution do?

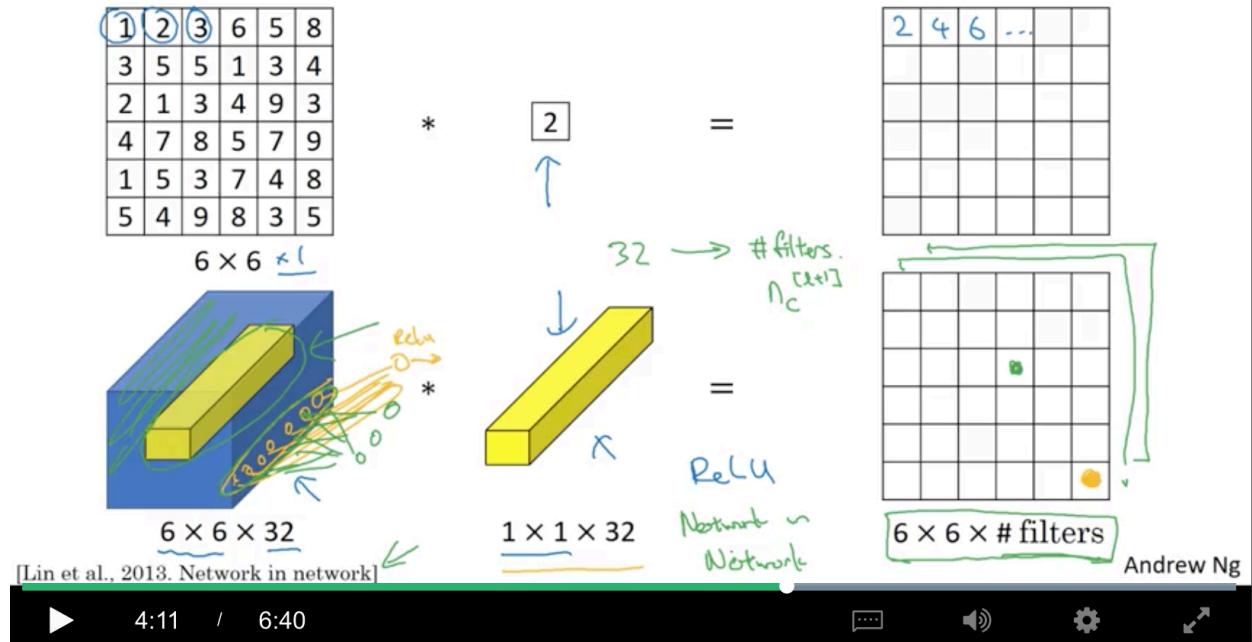
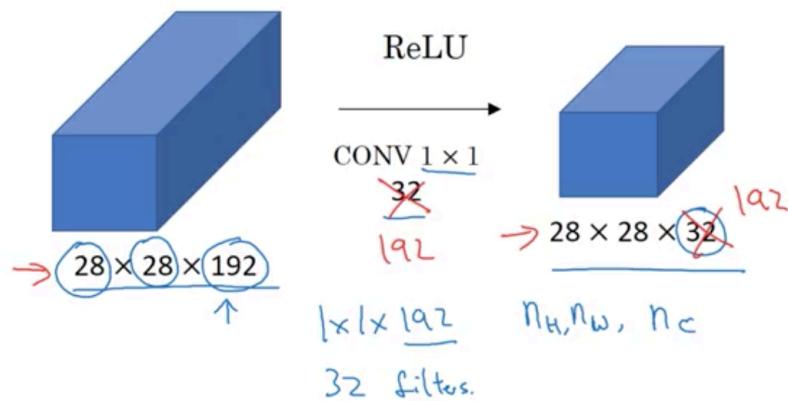


Figure 9: why-1x1-convs-do.png

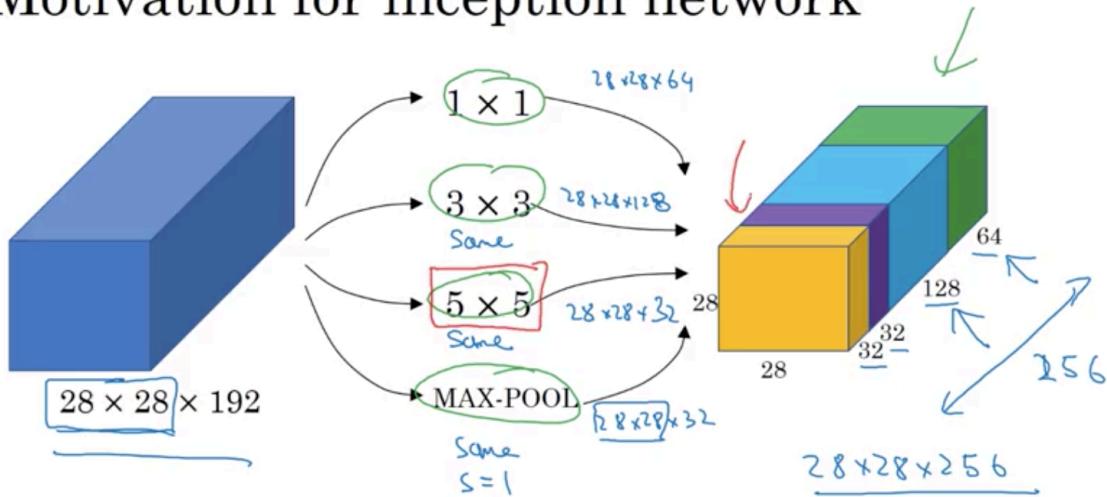
## Using $1 \times 1$ convolutions



Andrew Ng

Figure 10: using-1x1-conv.png

## Motivation for inception network



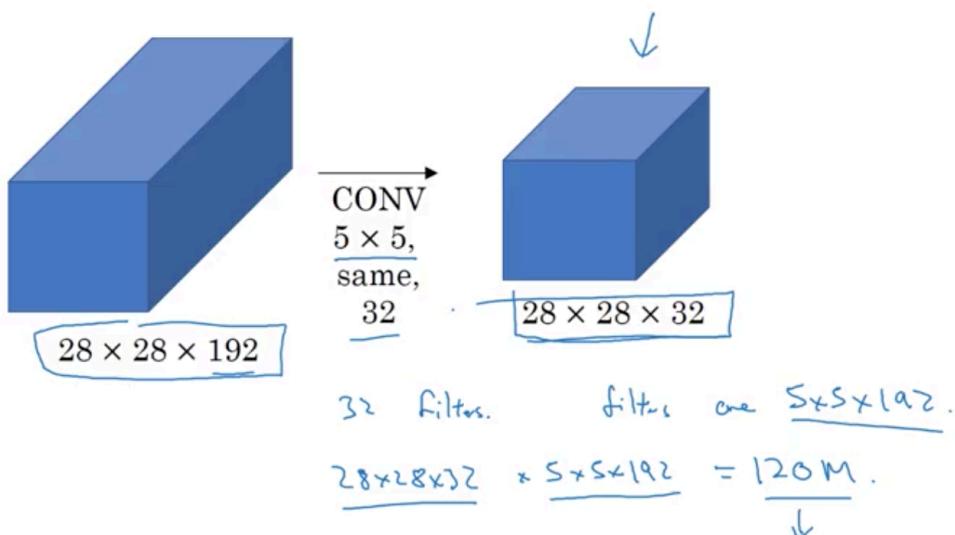
[Szegedy et al. 2014. Going deeper with convolutions]

Andrew Ng



Figure 11: motivation-for-inception-network.png

### The problem of computational cost



Andrew Ng



Figure 12: inception-computation-cost.png

而同样地由  $28 \times 28 \times 192$  变成  $28 \times 28 \times 32$ , 可以中间先接 16 个  $1 \times 1$ , 得到  $28 \times 28 \times 16$ , 再接 32 个  $5 \times 5$ , 得到  $28 \times 28 \times 32$ 。这个中间层, 有时被称作 bottleneck layer, 因为很细 ( $n_C$  很小咯)。。是这个网络中最小的部分。

看看计算成本:

- 计算 bottleneck 的成本: 输出是  $28 \times 28 \times 16$ , 对于每一个, 需要做  $192$  次乘法, 所以是  $28 \times 28 \times 16 \times 192 = 2.4M$
- 对于后半部分, 需要  $28 \times 28 \times 32 \times 5 \times 5 \times 16 = 10M$
- 二者相加, 就是  $12.4M$ , 所以大概减少为原来的  $1/10$  左右

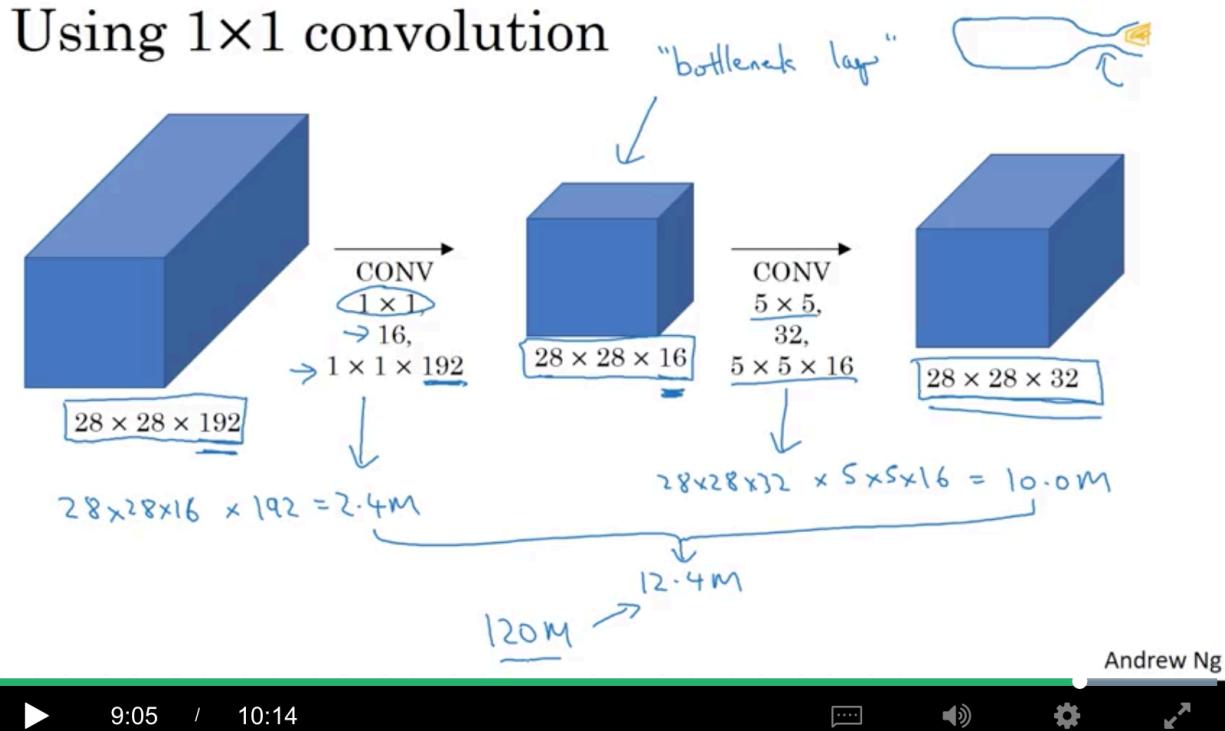


Figure 13: using-1x1-conv-for-5x5.png

合理地使用  $1 \times 1$ , 可以减小计算成本, 并且可以尽量不对网络的最终效果造成损失。

## 1.7 1.7 Inception Network

一个 inception module 如下 (例如, 输入是  $28 \times 28 \times 192$ ),

- 和上一节讲到的一样, 这里把  $5 \times 5$  拆成了  $1 \times 1$  和  $5 \times 5$ , 中间是 16 个 channel
- 对  $3 \times 3$  也做类似的操作,  $1 \times 1$  中间是 96 个 channel, 输出是  $28 \times 28 \times 128$
- 然后单独接一个  $1 \times 1$ , 输出  $28 \times 28 \times 64$
- 然后来个 pooling,  $3 \times 3$ , stride=1, same, 得到  $28 \times 28 \times 192$ , 可见这里的 channels 还是挺多的, 所以加多一个  $1 \times 1$ ~ 得到一个  $28 \times 28 \times 32$
- 最后把上面的输出全部 concat 到一起, 得到  $28 \times 28 \times (64+128+32+32)=28 \times 28 \times 256$

所以 2014 年的 googlenet, 其实就是一系列的 inception module, 可能有一些 block 不太一样 (例如图中有箭头那个, 就是先 pooling 了一下, 改变维数, 再接的 inception block)。

还有一点值得注意的是, inception network 还有一些 side branches(旁枝)。因为最后是一些 FC, 再接一个 softmax。而这些 side branches 是试着把一些中间层拿出来, 经过一些全连接 +softmax 来做预测。作用是确保计算的特征值, 即使在中间层, 在做预测时也不会表现得太差。这其实是一种对 inception network 起到一种正则化的效果, 避免 overfitting。

# Inception network

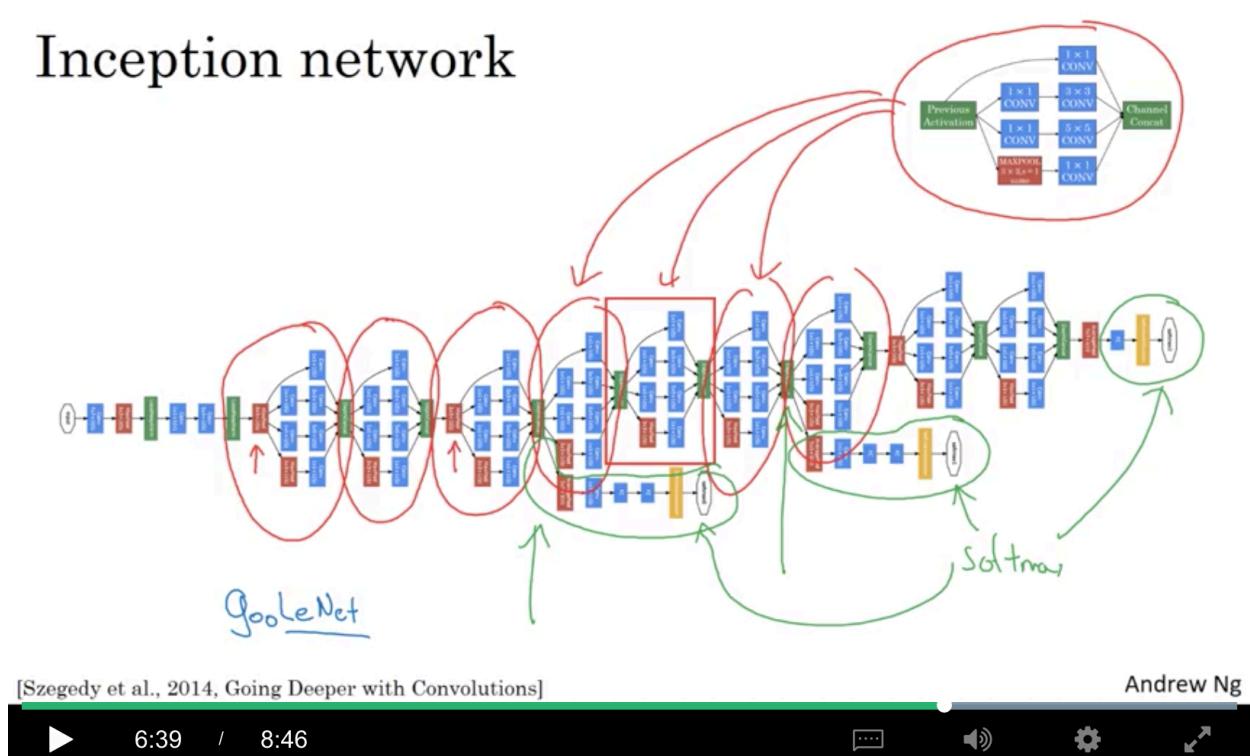


Figure 14: inception-network.png

## 2 2. Practical advices for using ConvNets

### 2.1 2.1 Using Open-Source Implementation

很多神经网络的效果很难复现，因为很多超参的调整（例如学习率衰减）等很多细节都会对最终结果带来影响，所以还是从github上找开源实现吧~。

### 2.2 2.2 Transfer Learning

例如需要搞一个三分类问题，可以找一个开源的实现，并把训练好的参数搞下来。

#### 2.2.1 2.2.1 如果训练集较小

然后把最后的 softmax 层（例如 Imagenet 就是 1000 分类）去掉，改成一个三分类的 softmax。而前面的参数直接 freeze，只训练与 softmax 层有关的参数。

如何固定参数呢？有的框架可以设置 `trainableParameters=0` 之类的。

也可以将前面层的当然一个固定的函数，输入一个  $X$ ，计算其特征向量，把所有特征向量直接存放到磁盘中，然后拿这些向量去经过一个浅层神经网络，训练这个网络。

#### 2.2.2 2.2.2 如果训练集很大

可以只 freeze 前面的部分层，然后有几种做法：

- 可以把最后几层的参数拿来初始化，然后进行训练

- 也可以直接去掉最后几层，接自己的 softmax

如果数据越多，所 freeze 的层数可以越少 ~

### 2.2.3 如果训练集特别大

甚至可以拿下载下来的权重做初始化，直接训练整个网络。

在 cv 领域，迁移学习往往能取得很好的效果。

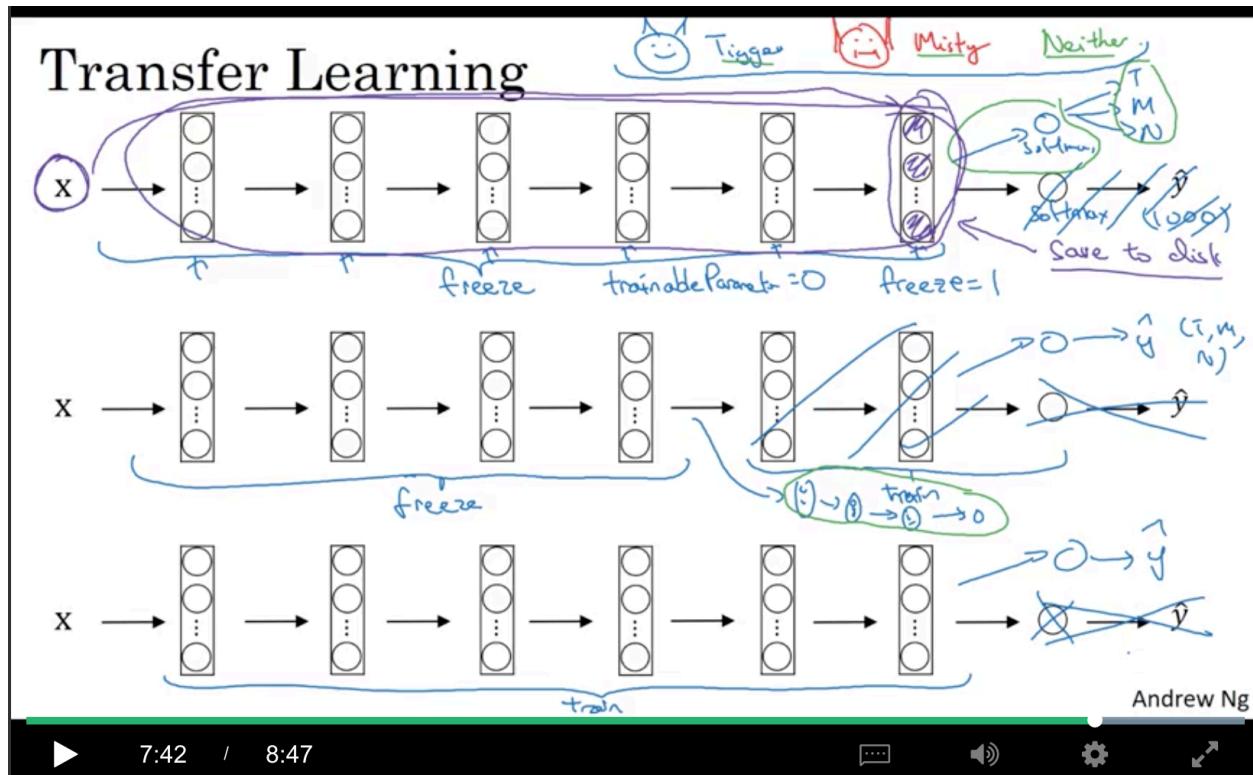


Figure 15: transfer-learning.png

### 2.3 2.3 Data Augmentation

最常用的 data augmentation 如下：

- mirroring: 镜像，垂直翻转
- random cropping: 随机裁剪

还有其他几个，不太常用：

- Rotation: 旋转一定角度
- Shearing: 剪切，使图像变形，例如图中的变成平行四边形。。
- Local Warping: 局部弯曲

另一种是 Color shifting(色彩变化)，在 R/G/B 分别加上不同的扰动，图中  $+/-20$  只是为了看得更明显一点，实际的扰动可能很小。

PCA (主成分分析) 也可以用在色彩变化上，在 Alexnet 的文章中有介绍，被称为 PCA 色彩增强 (PCA color augmentation)：如果图片是偏紫色的，也就是说 R 和 B 比较多，G 较少，PCA color augmentation 会对 R 和 B 做比较大的加减，而对 G 只做微小的变化，从而保持整体与之前有相同的着色。

# Color shifting

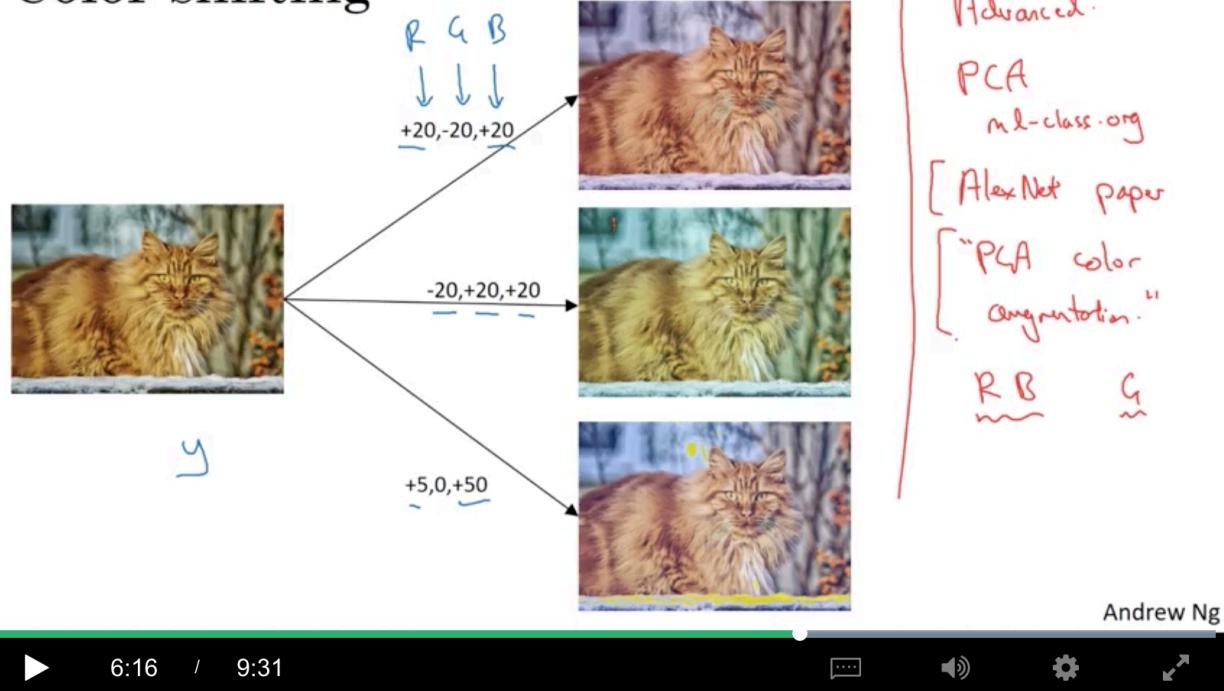


Figure 16: color-shifting.png

数据增强同样也有很多的超参数，所以可以直接拿开源版本的实现来搞，当然，也可以针对具体任务，自己调一下超参。

## 2.4 2.4 State of Computer Vision

从容易获取的数据量来讲，object detection < image recognition < speech recognition,

- 数据足够多时，用的是更简单的算法，更好的人工特征
- 数据较少时，用的是更多的人工特征

knowledge 有两种来源：

- 标注数据
- 手工构造的 feature/网络架构/其他组成部分

在 cv 领域，标注数据并不多，所以经常需要设置很复杂的网络结构。object detection 任务的数据比图像识别更少，所以它们的架构更复杂，而且有更多的 specialized components。

当然，如果数据集不够大，transfer learning 是一个很好的方法。

另外，如何在 benchmark 或者比赛中取得比较好的成绩呢？有以下几种方法（在实际工程领域并不一定有用...）：

- ensembling:
  - 独立地训练一些（一般有 3-15 个）网络，并将他们的输出取平均。注意，是输出，不是权重
  - 在 test 的时候使用 multi-crop
  - 针对 test image，进行多种 crop，对他们分别做预测，然后对结果求平均

使用开源代码：

- 使用已 published 的网络架构
- 使用开源实现

- 使用 pretrained 的模型，并在你的数据集上 finetune