目录

Week2 深度卷积网络：实例探究

2.1 为什么要进行实例探究

上周我们介绍了 CNN 的各个基本构建（如卷积层、池化层、全连接层等），事实上过去几年 CV 研究中大量的精力都集中在如何组合这些基本构建，从而形成有效的 CNN 网络。找感觉最好的方式就是去看一些实际案例，通过研究别人构建 CNN 的方法获得灵感。

而且，一些在 CV 任务中表现良好的网络结构往往也适用于其他的任务，所以我们也可以借鉴他们的网络来解决自己的问题。

后续将一一探究的 CNN 网络结构如下：

（1）经典 CNN 网络：LeNet-5，AlexNet，VGG

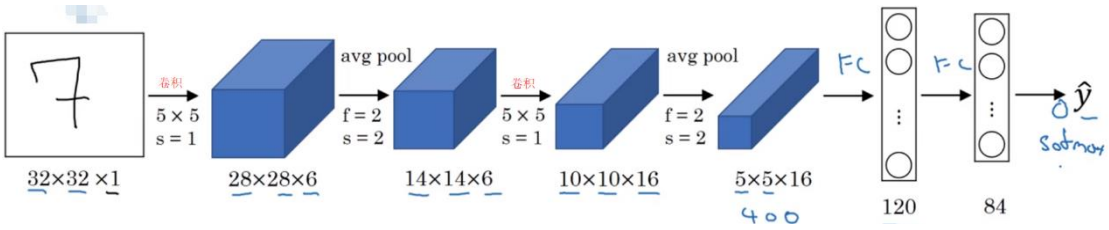
（2）ResNet

（3）Inception

2.2 经典网络

LeNet-5：

假设我们用 LeNet-5 来识别图片中的数字，设输入图片的维度是 32*32*1，也就是此 LeNet-5 是基于灰度图像来训练的。



Tip：经典的 LeNet-5 网络在池化后一般会用非线性函数来激活（这与目前的习惯也一致）

（1）使用 6 个 5*5 的过滤器（kernel），步幅（stride）为 1，padding 为 0，则本次卷积之后输出的维度是 28*28*6。

（2）使用平均池化（Average pooling），过滤器（kernel）的宽度为 2（2*2 的缩写），步幅为 2。于是 feature map 的宽度和高度都缩小为一半，故池化后输出的维度是 14*14*6。

（3）使用 16 个 5*5 的过滤器，步幅为 1，故卷积后输出的维度是 10*10*16。

（4）使用平均池化，过滤器宽度为 2，步幅为 2，故池化后输出维度变成 5*5*16。可视为一个 400 维的长向量。

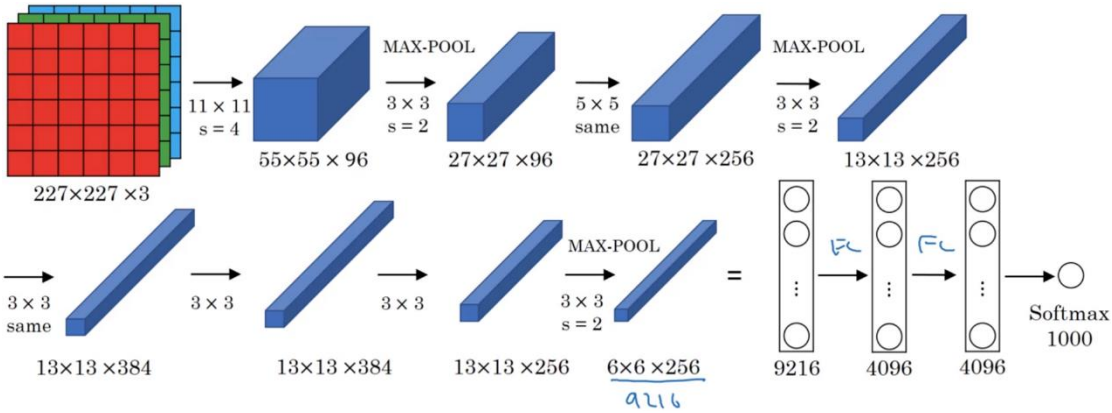
（5）把该向量作为全连接层的输入，我们或许得到一个 84 维的向量。再经过第二个全连接层，我们得到的输出 y^就可以用来预测数字了。通常对于该多分类问题会使用 softmax 函数。

由于当时人们并不使用 padding 或有效卷积，于是可见每次卷积/池化后图像的宽度和高度都会缩小。且一般随着深度增加，feature map 的通道数也逐渐增加。这个网络比较小，它只有 60K 个参数；现代网络通常有一千万到一亿个参数。。

一般我们设计的 CNN 网络都是如下形式：

AlexNet：

AlexNet 首先输入一张 227*227*3 的图片。过程如下：



（1）卷积：使用 96 个 11*11 的过滤器，步幅为 4，则本次卷积之后 feature map 的维度为 55*55*96（追踪 kernel 右上角：pos=4*k+7=227，故 k=55）。

（2）池化：使用 3*3 的最大池化层，步幅为 2，则池化后维度为 27*27*96（追踪 kernel 右上角：pos=2*k+1=55，故 k=27）。

（3）卷积：使用 256 个 5*5 的过滤器，在零扩充之后，维度变为 27*27*256。

（4）池化：使用 3*3 的最大池化，步幅为 2，则池化后维度为 13*13*256（追踪 kernel 右上角：pos=2*k+1=27，则 k=13）。

（5）卷积：使用 384 个 3*3 的过滤器，在零扩充之后，维度变为 13*13*384。

••••

（8）池化：使用 3*3 的最大池化，步幅为 2，则池化后维度为 6*6*256=9216。将其展开为 9216 个单元。

（9）一些列全连接层

（10）softmax：得到 1000 种识别结果的概率分布。

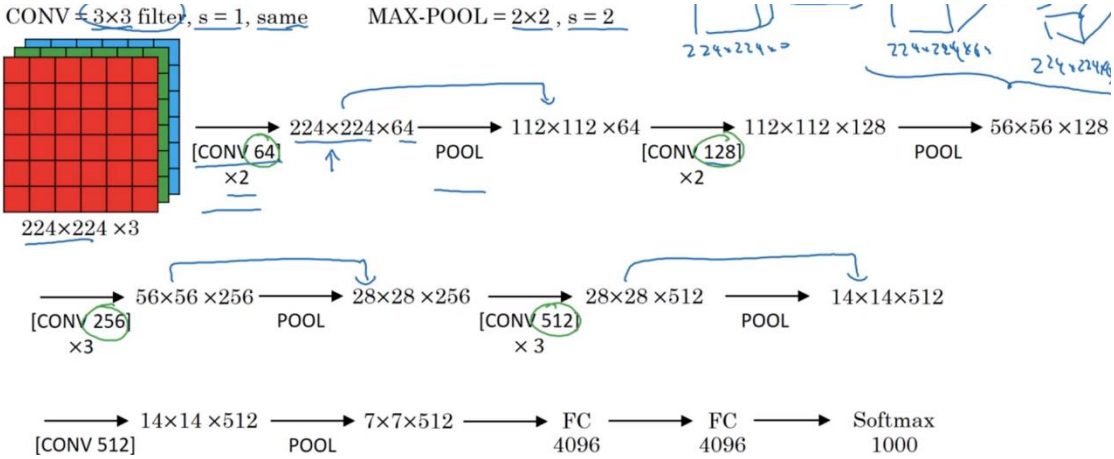
Tip：

（1）相比于 LeNet-5，本 AlexNet 与它结构相似，但网络庞大许多，故表现往往更好。

（2）AlexNet 表现更好的另一个原因是：使用了 ReLU 激活函数。

VGG-16：

相比之下，VGG 网络没有那么多的超参数，这是一种只需要专注于构建卷积层的简单网络。它的优点是简化了网络的结构。过程如下：



（1）卷积：使用了 64 个 3*3 的相同卷积（维度保持不变），步幅为 1，故输出的维度为 224*224*64。连续使用 2 次这个卷积。

（2）池化：使用 2*2 步幅为 2 的最大池化，对图片进行压缩。则输出为 112*112*64。

（3）卷积：使用 128 个 3*3 的相同卷积，步幅为 1，并连续卷积 2 次，则输出维度为 112*112*128。

• • •

（n）全连接层。

（m）softmax 层。得到预期输出的概率分布。

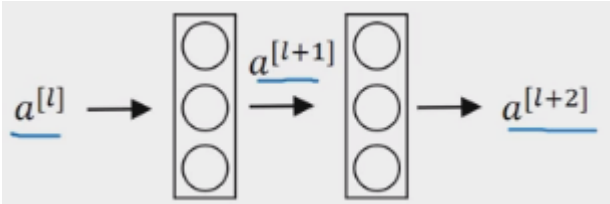
Tip：这是一个非常大的网络，总共包含 1.38 亿个参数。但它并不算复杂，因为这种网络的结构特别规整。因为它都是连续几次卷积+一次池化，且卷积时过滤器的个数都是翻倍增长。

2.3 残差网络 ResNet

我们已经知道，非常深度网络是难以训练的，因为存在梯度消失和梯度爆炸的问题。本节我们将学习 skip connections，它可以从某一层获取激活，然后迅速反馈给另外一层，甚至是网络的更深层。使用这种连接我们能构建可训练的非常深的 ResNets 网络。

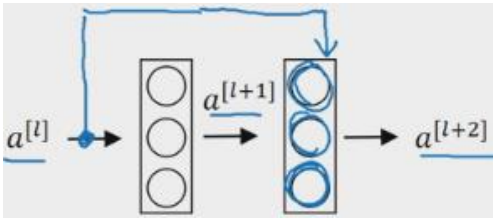
残差块 Residual block：

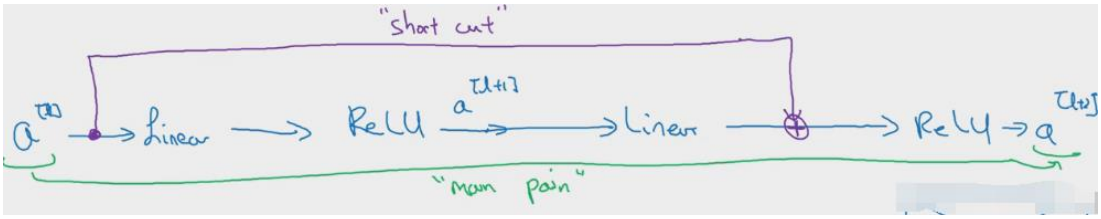
对于普通的神经网络：若要从 L 层的某个神经元传播到 L+2 层，需要连续两次的线性变换+非线性激活得到：



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

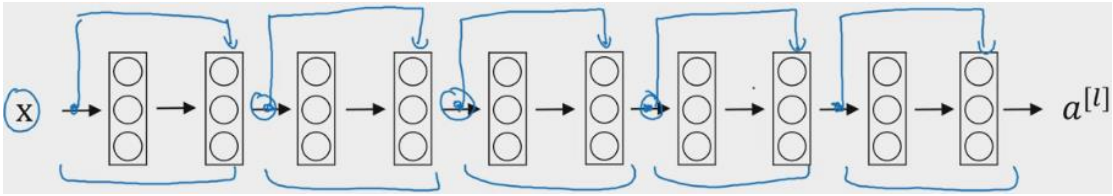
而对于残差网络 ResNets：将在 main path 的基础上，增加一条捷径 short cut，使得 $a^{[L]}$的拷贝值直接传到 ReLU 的入口：



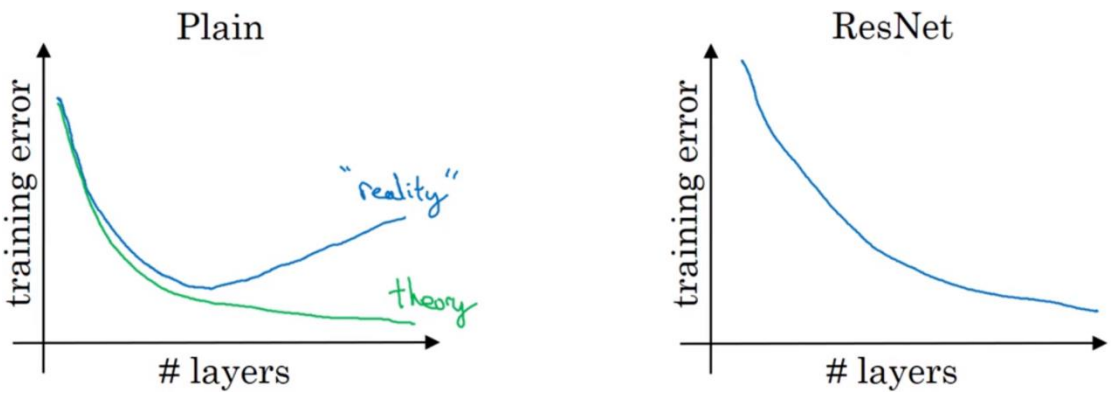


即 $a^{[L+2]}$=ReLU($Z^{[L+2]}$+$\mathbf{a^{[L]}}$)。所以 $a^{[L]}$能通过 shortcut 跳过一层或多层网络，直接把信息传递到网络的更深层。于是，使用残差块能训练更深的网络。

残差网络 Residual Networks：

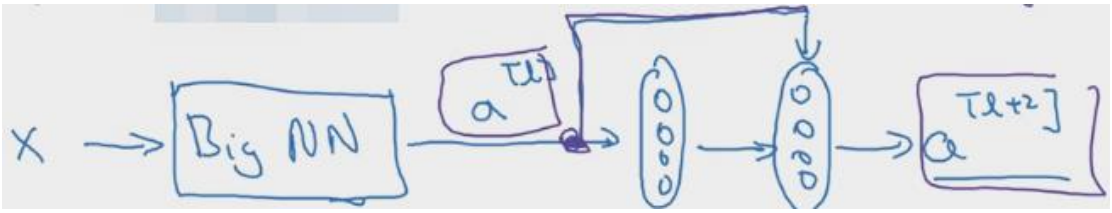


传统的网络随着深度加深，由于存在梯度消失和爆炸的问题，且误差居然越来越大（这是不应该的，因为网络越深拟合的函数应该就越复杂）；而有了 ResNets，即使网络很深，对应的误差也会逐渐变小，使得我们能够训练更深且保证性能的网络了：



2.4 残差网络为什么有用

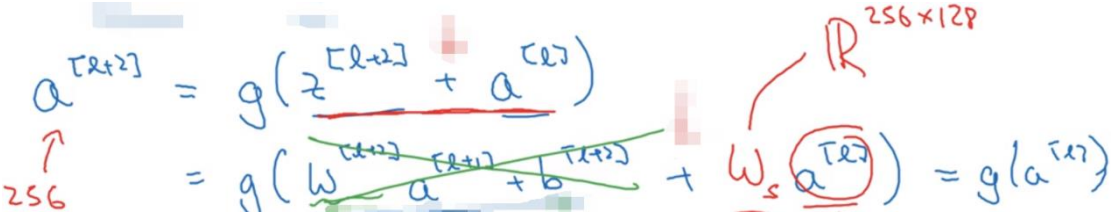
假设网络的输入为 x，经过 L 层的传播后得到 $a^{[L]}$，现在通过 skip connection 直接传给 L+2 层的入口：



故 $\mathbf{a^{[L+2]}=g(z^{[L+2]}+a^{[L]})=ReLU(w^{[L+2]}a^{[L+1]}+b^{[L+2]}+a^{[L]})}$，如果学得的 $w\approx 0$，则 $a^{[L+2]}\approx g(a^{[L]})\approx a^{[L]}$。由此可见，残差块学习这个恒等函数并不难，因为 skip connection 使我们很容易地得到 $a^{[L+2]}\approx a^{[L]}$。这也是残差网络有效的主要原因，有了它，我们能确定网络性能不怎么受影响，很多时候反而会学习到意料之外的惊喜。

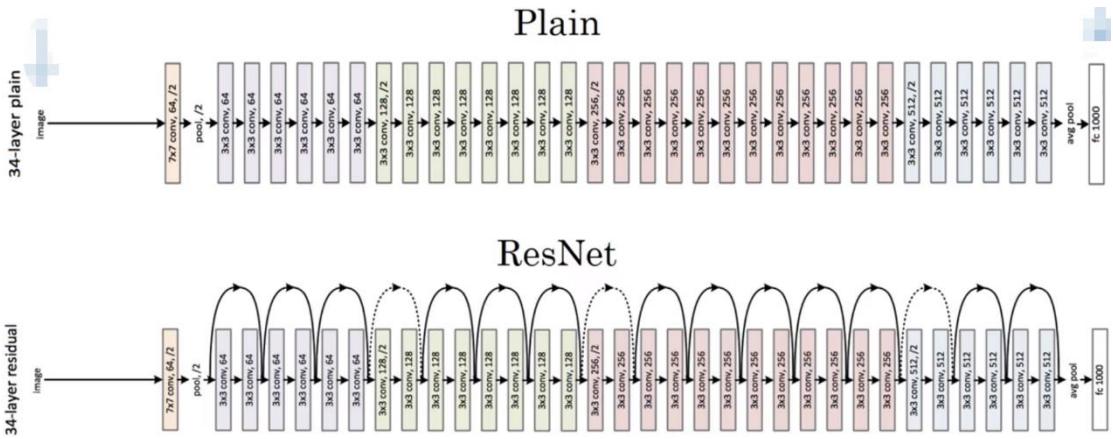
而对于普通网络，随着深度增加，就连学习一个恒等函数都是十分困难的。所以很多层之后，表现不但没有变好，可能反而更糟。

细节：假设 $z^{[L+2]}$和 $a^{[L]}$有相同的维度（即 L 层与 L+2 层有相同数量的神经元），所以 ResNets 使用了很多相同卷积；如果二者维度不相同，需增加一个参数矩阵 w_s，使得：



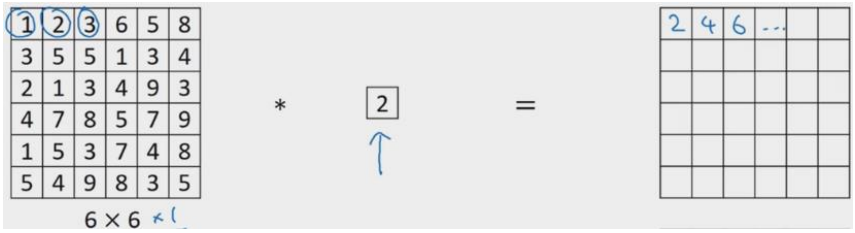
实例：

把一张图片作为输入，一个典型的 CNN 和 ResNets 如下：

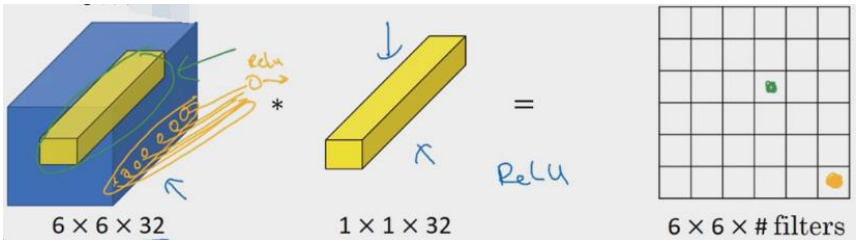


2.5 网络中的网络及 1×1 卷积

在设计架构的时候，我们很多时候会用到 1×1 卷积。对于一个 6*6 的图片，使用一个 1×1 卷积（kernel 值为 2），那么实质就是对整个矩阵用 2 进行数乘，并没有太大的实际意义：



但是对于多通道的情况，很多时候 1×1 卷积往往很有效。假设输入为 6*6*32，kernel 为 1*1*32。对它进行卷积可知：



对某一通道，提取到的切片维度是 1*1*32，用当前 kernel 对其进行卷积，计算各通道的值与对应权重乘积之和，得到卷积后的标量值。该步骤的本质类似两个向量的内积。

通过 kernel 在整个图片上滑动，不断自左向右、自上而下地扫描，最终我们得到整个输入的卷积结果。若假设总共有#filter 个 kernel，则卷积后输入的维度是 6*6*(#filter)。

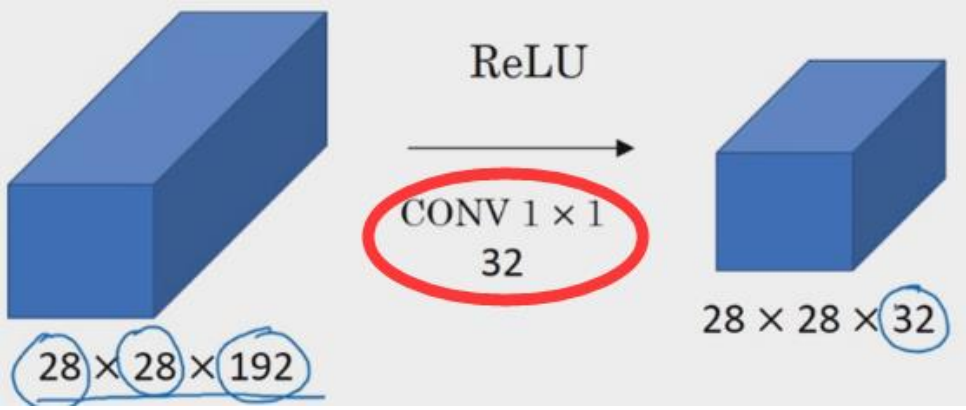
故 1×1 卷积可以理解为对每个切片的 32 个单元应用了一个全连接神经网络。FCN 的输入是 32 维的切片，输出#filter 个值。从而以便在输入层上实施一个非平凡（Non-trivial）计算。1×1 卷积也被称为网络中的网络（Network in

Network）。

尽管 1×1 卷积的使用并不是非常广泛，但是它对其他模型的出现产生了很大的影响，如 Inception。

1×1 卷积举例：

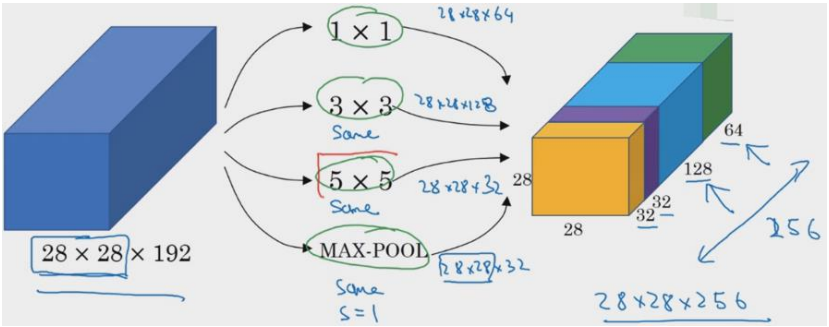
我们用池化可以压缩图片的长度和宽度，得到图片更抽象的特征表示，但是池化往往不能改变通道的数量。这里，我们使用含 32 个 kernel 的 1*1 卷积处理图片，得到的输出维度是 28*28*32，故它能保持长度宽度不变，而压缩通道的数量：



2.6 谷歌 Inception 简介

以前我们在构建卷积层的时候，经常需要考虑的是 kernel 的大小，或者要不要加池化层。而 Inception 网络的作用就是代替你来做决定。虽然网络因此变得很复杂，但是它却表现得很好。

假设 Inception 的输入为 28*28*192，依次让一系列的 kernel（如 1*1*64、3*3*128、5*5*32 等）对输入进行卷积。由此，我们可以得到一系列的卷积结果（如右图不同的色块），把它们堆积（Stack up）在一起，得到一个 28*28*256 的卷积结果：



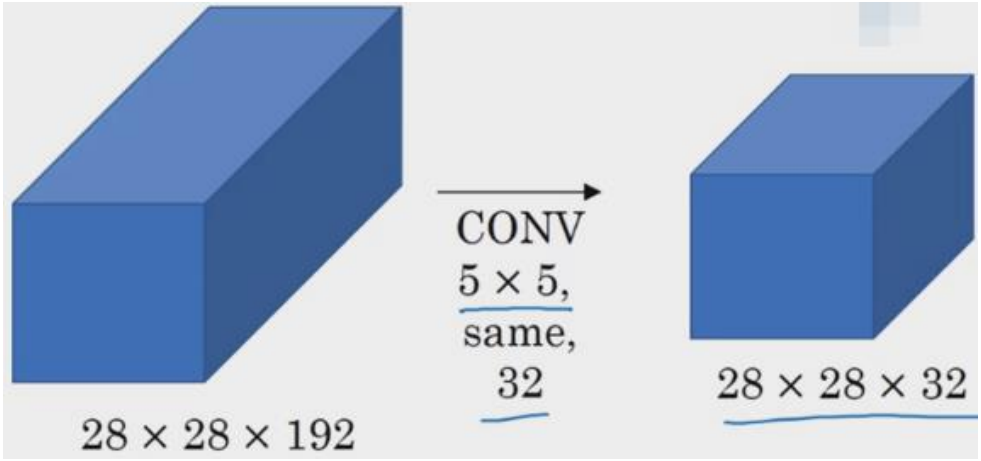
为使以上各色块能拼接在一起，需要让每个 kernel 的结果长、宽维度一致，故这里采用的是相同卷积。

而 Inception 的做法就是把上文的中间值作为其后全连接神经网络的输入，

让网络自己通过数据来学习所需要的参数，即网络自行决定需要的卷积/卷积的组合。

计算代价的缺陷：

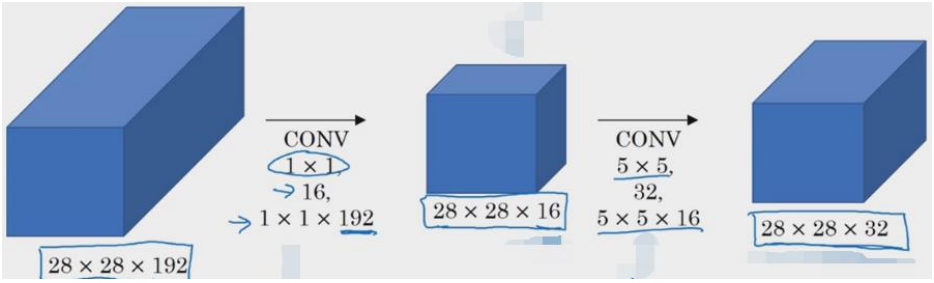
以上文 Inception 中 5*5*32 的 kernel 为例，分析对应的计算成本：



对于输出的每个 unit：计算次数为 5*5*192（首先 5*5 次乘法卷积单个通道，然后对输入的所有通道执行同样操作，最后相加得到一个 unit）；其次，输出总共有 28*28*32 个 unit，故该 kernel 总共需要（28*28*32）*（5*5*192）=120M 次计算。

使用 1×1 卷积减小计算量：

对于同样的输入，我们先让它通过一个 16 个 1*1 的 kernel，将得到 28*28*16 的中间结果；其次，再进行原本的 5*5 卷积（32 个核），得到最后 28*28*32 的输出：



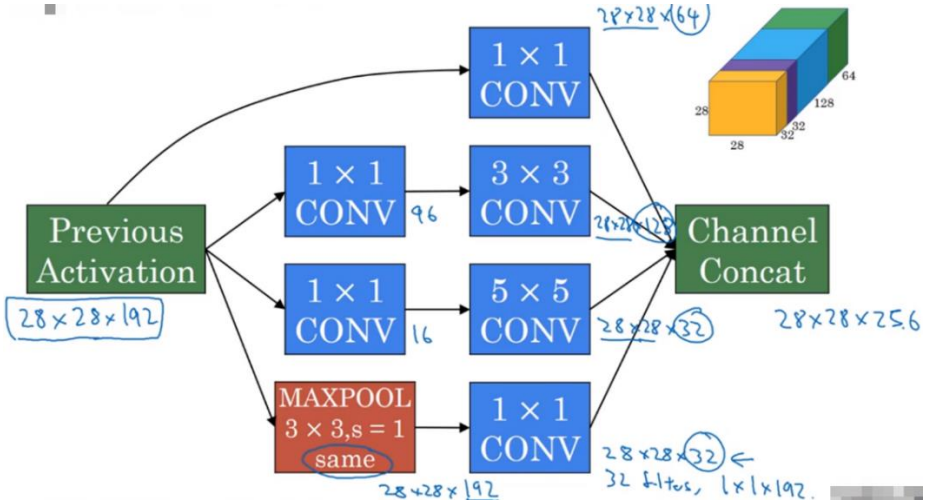
第一部分：计算量为（28*28*16）*（1*1*192）=2.4M；第二部分：计算量为（28*28*32）*（5*5*16）=12.4M。故使用 1×1 卷积后网络总的计算量为 12.4M<<120M。由此可见，计算量大大减少！根本原因是网络通过 1×1 卷积先大大降低了通道的数量！

我们也把这个中间层叫做瓶颈层（Bottleneck layer），通过构建瓶颈层我们可以降低网络的计算开销。且事实证明，如果选择合适的瓶颈层，我们既可以显著缩小表示层的规模，又不会降低网络的性能！

2.7 Inception 网络

Inception Module：

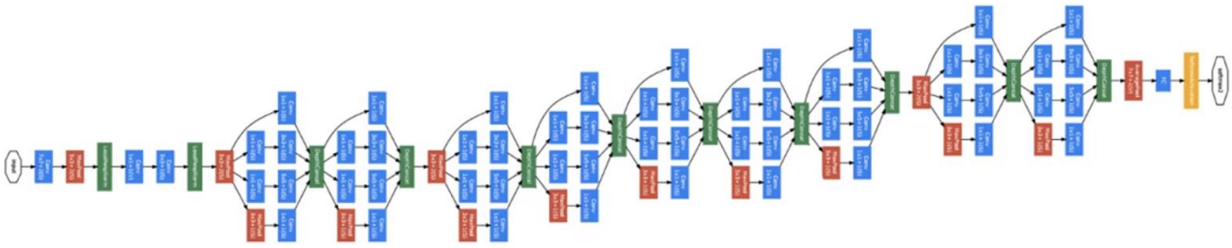
假设获取到的激活值维度为 28*28*192，对其分别使用几种不同的卷积，并将它们的结果拼接起来（在 channel 维度拼接），得到输出 28*28*256，称之为一个 Inception Module：



注意：红色部分是最大池化，使用 same 以保持维度不变，并紧接一个 1×1 卷积来压缩通道的维度。

Inception Network：

将大量的 Inception block 连接起来，得到 Inception Net：



网络末尾会有 FCN 来预测输出。通常，某些 Block 下面也会有分支，它们也是通过连接 FCN 和 softmax 来做出预测。

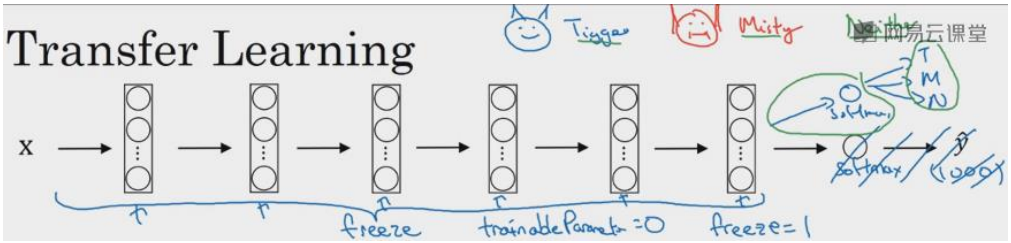
2.8 使用 CNN 的建议：借鉴开源的实现

通常，在阅读他人论文的时候，我们建议 github 直接搜索其开源的代码，这将比自己手动从零实现快得多。

通过直接下载别人已实现的代码，模型往往已经由他们提前训练好，不用再去训练模型参数（很多大的数据集甚至需要训练几周）。我们可以直接用它来迁移学习，把这个模型转移到自己感兴趣的任务上。

2.9 迁移学习

假设现在我们的任务是做一个识别猫的种类的分类器，但是收集的数据集十分有限。先去网上下载别实现的网络和权重，该网络或许是已经在 ImageNet 上训练好了的，它通过 softmax 有 1000 个输出：



我们可以去除原网络最后的 softmax 层，添加自己的 softmax 用来分类 Tigger 猫、Misty 猫和剩余猫（三分类问题）。在此过程中，我们把前面的网络层全部视为冻结的（freeze），而只修改其最后的部分，于是我们只用训练与 softmax 有关的参数。

由此，尽管我只有一个很小的数据集，通过训练这个合成的网络，也能最终得到很好的性能。幸运的是，目前大多数 DL 框架都支持这种操作。

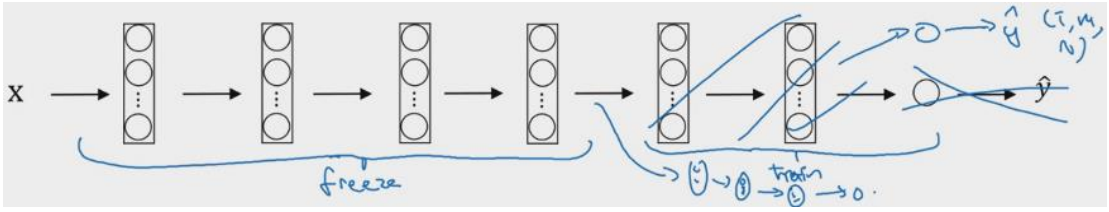
技巧：

由于前面的网络都已经训练好，我们不需再改动。所以我们可以先把训练集 x 输入给前面的网络，然后保存下这些 frozen 网络产生的激活值到磁盘中。每次需要训练 softmax 时，不用再重复计算前面庞大的网络产生的激活值；而是直接把文件中保存的值作为 softmax 的输入来训练。

这样，我们就可以只用训练一个很浅层的网络了，每次也不用再花时间重复劳动，来计算这些激活值。

经验：

假设我们的训练集比较大，此时应该冻结更少的中间层，让尾部一部分的网络层参与训练。



方法 1：随机初始化最后几层网络层的权重，并让它们参与后续的训练与参数更新。

方法 2：直接去掉最后几层，替换成自己的 hidden layer，结合最后的 softmax 统一训练和更新参数。

规律：数据集越大，冻结的网络层数越少。除非自己有个非常大的训练集，

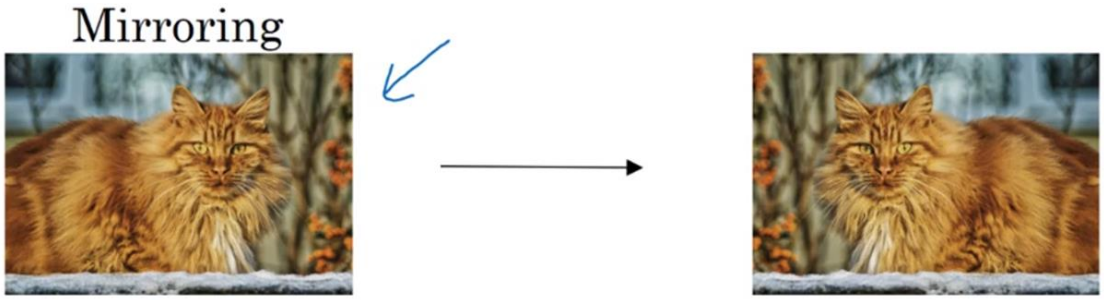
否则优先考虑使用别人开源的网络和权重，这将大大提升我们的模型性能。

2.10 数据扩充

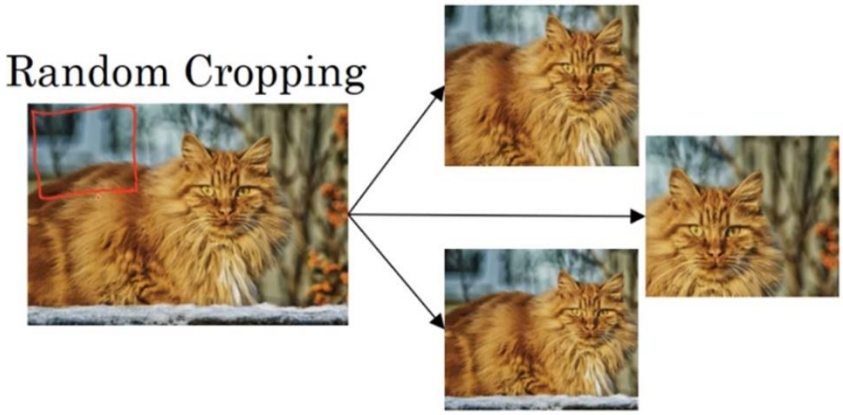
大部分 CV 任务都会用到大量的数据，数据集的扩充往往能带来 CV 模型性能的提升。所以数据增强（Data Augmentation）也是经常使用的技巧。对于 CV 问题，目前一大问题往往就是得不到足够的数据。

常用 Augmentation 方法：

1.镜像对称：对图片左右对称，由于待识别的物体仍然在图中，但是视角已经转变，故可以视为一个较为独立的训练样本。

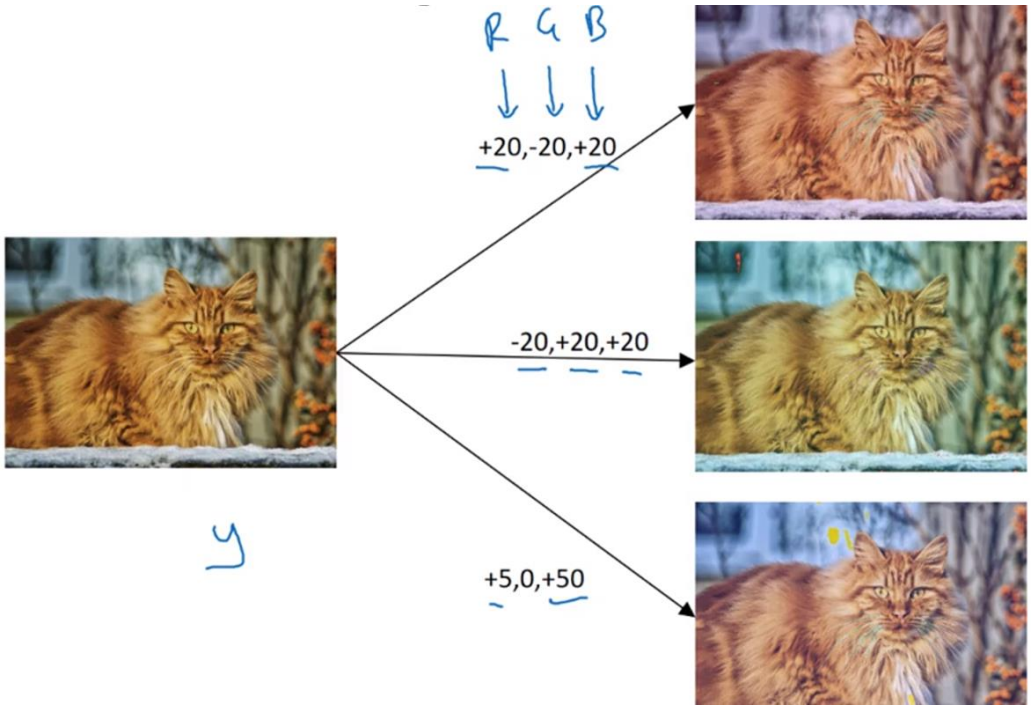


2.随机裁剪：用 Random Cropping 随机处理图片，生成各种局部图片作为新样本。虽然这不是一个完美的方案，但是这种方法在实际使用中效果还是挺不错的。



当然，理论上我们还可以使用旋转（Rotation）、剪切（Shearing）和局部弯曲（Local warping）等。但是它们比较复杂，在实践中使用比较少。

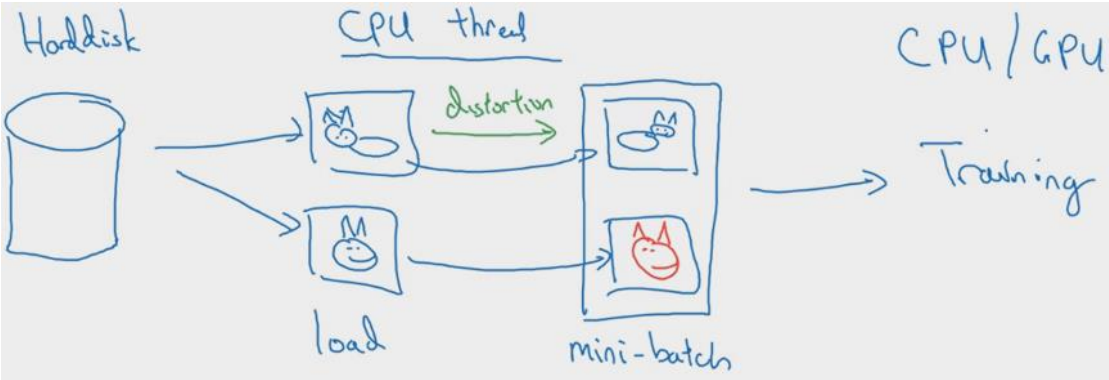
3.色彩转换（Color Shifting）：给定输入图片，对其 RGB 三个通道加上一定的失真值（distortion）。尽管对于人来说一眼就可以认出这几张照片是对应同一个物体，但是对于计算机来说，输入的图片实质是一个三维矩阵，故色彩转换后的图片也可以视为一个较为独立的样本。



当然，这种转换能模拟一些现实的场景：对于 R 值升高的图片，相比原图，或许我们可以认为它是阳光偏黄的时候拍到的照片。但是其内容并没有改变，所以可以用来参与训练分类器。所以，用 Data augmentation 训练后的分类器对照片颜色的更改更有鲁棒性（robust）。

在训练时实现 distortions：

假设数据集存放在磁盘中，我们用一个 CPU 线程不断地从里面读取图片，并采用各种 mirroring、cropping 等。从而构成一个 mini-batch 的 data。再把这些 batch 持续送给 CPU/GPU 来训练：



核心：用一个 CPU 线程专门处理图片，形成 mini-batch；用另外一个线程/进程专门训练模型。这样，二者就可以并行了！

当然，同样建议直接使用别人实现的开源代码。

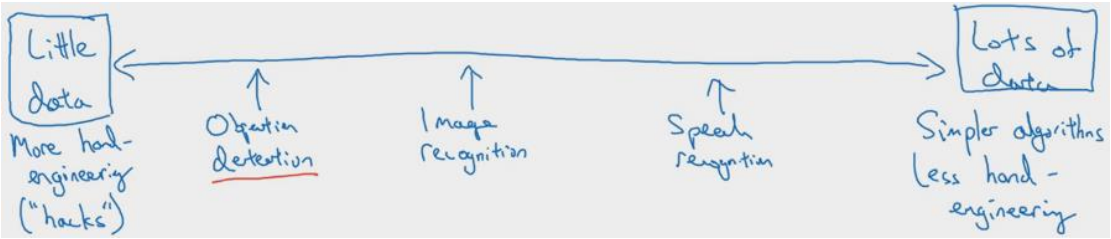
2.11 CV 的现状

Data vs. Hand-engineering：

Image Recognition：告诉你图片中是不是猫。

Object Detection：找出图片中有哪些物体以及它们的位置。

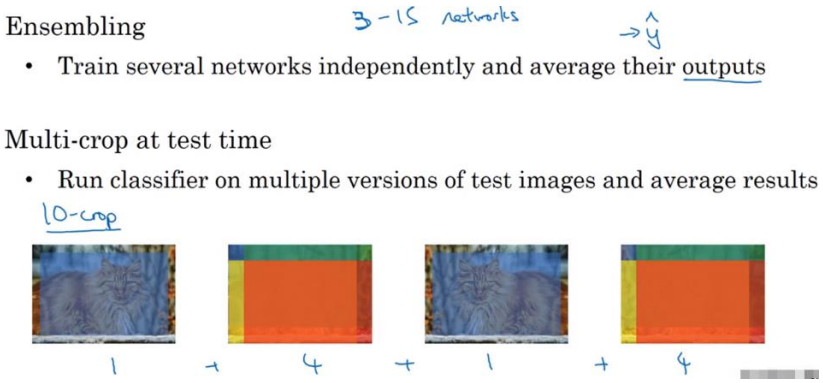
目前各个领域来说，其收集到的数据集与预期相比如下：



（1）对于语音识别，由于我们能收集到足够的数据，所以其模型更为简单，同时也只需要更少的手工工程；

（2）而对于物体检测，由于数据明显不足，所以我们往往会设计更复杂的模型，做更多的手工工程。

提升在 benchmarks 上性能的技巧：



使用开源代码：

使用别人实现的模型，往往他们已经解决了非常多繁琐的细节，比如设置学习速率衰减或超参数；或者他们已经花了非常多的时间来训练。

通过使用他人预训练（pretrain）的模型，我们只用在自己数据集上进行微调（fine tuning），往往能实现较好的迁移。这种方法能大大加快我们的研究进度。