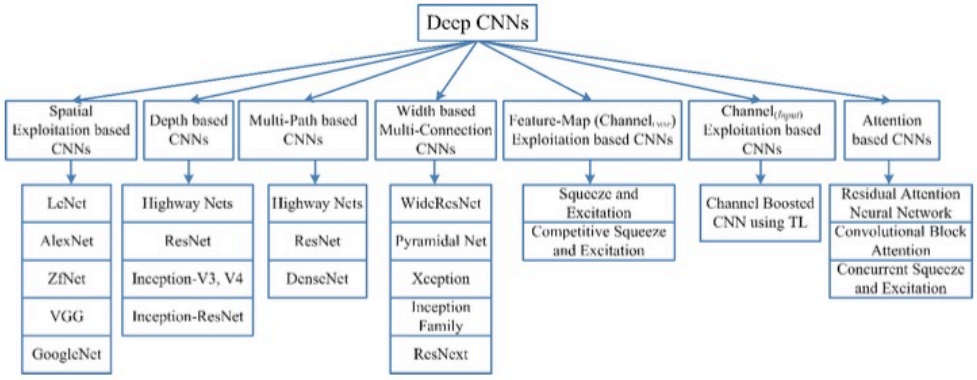
分类网络 阅读笔记



上图将主要算法分为了Spatial Exploitation based CNNs, Depth based CNNs, Muilti_Path based CNNs, Width based Muilti-Connection CNNs, Feature Map $(Channel_{FMap})$ Exploitation based CNNs, Attention based CNNs, 7个类别.

LeNet & Alexnet

LeNet由LeCuN等在1998年提出. 它以其历史重要性而闻名, 因为它是第一个CNN, 在手手指识别任务中显示了最先进的性能. 它具有对数字进行分类的能力, 而不会受到较小的失真的影响.而8层CNN的AlexNet以非常大的优势赢得了2012年ImageNet比赛. 该网络首次证明深度学习比手动提取特征的效果更好.AlexNet的缺点是深度加深之后会导致过拟合,它的解决方式是在开始的几层使用kernelsize=11的卷积,使用ReLU降低梯度消失的问题,使用dropout降低过拟合.

在LeNet提出后的将近20年里, 神经网络一度被其他机器学习方法超越, 如支持向量机. 虽然LeNet可以在早期的小数据集上取得好的成绩, 但是在更大的真实数据集上的表现并不尽如人意. 一方面, 神经网络计算复杂. 虽然20世纪90年代也有过一些针对神经网络的加速硬件, 但并没有像之后GPU那样大量普及. 因此, 训练一个多通道、多层和有大量参数的卷积神经网络在当年很难完成. 另一方面, 当年研究者还没有大量深入研究参数初始化和非凸优化算法等诸多领域, 导致复杂的神经网络的训练通常较困难.

神经网络可以直接基于图像的原始像素进行分类. 这种称为端到端(end-to-end)的方法节省了很多中间步骤. 然而, 在很长一段时间里更流行的是研究者通过勤劳与智慧所设计并生成的手工特征. 这类图像分类研究的主要流程是：

1. 获取图像数据集；
2. 使用已有的特征提取函数生成图像的特征；
3. 使用机器学习模型对图像的特征分类.

当时认为的机器学习部分仅限最后这一步. 如果那时候跟机器学习研究者交谈, 他们会认为机器学习既重要又优美. 优雅的定理证明了许多分类器的性质. 机器学习领域生机勃勃、严谨而且极其有用. 然而, 如果跟计算机视觉研究者交谈, 则是另外一幅景象. 他们会告诉你图像识别里“不可告人”的现实是：计算机视觉流程中真正重要的是数据和特征. 也就是说, 使用较干净的数据集和较有效的特征甚至比机器学习模型的选择对图像分类结果的影响更大.

$AlexNet$

AlexNet与LeNet的设计理念非常相似, 但也有显著的区别.

第一, 与相对较小的LeNet相比, AlexNet包含8层变换, 其中有5层卷积和2层全连接隐藏层, 以及1个全连接输出层. 下面我们来详细描述这些层的设计.

AlexNet第一层中的卷积窗口形状是 11×11. 因为ImageNet中绝大多数图像的高和宽均比MNIST图像的高和宽大10倍以上, ImageNet图像的物体占用更多的像素, 所以需要更大的卷积窗口来捕获物体. 第二层中的卷积窗口形状减小到 5×5, 之后全采用 3×3. 此外, 第一、第二和第五个卷积层之后都使用了窗口形状为3×3、步幅为2的最大池化层. 而且, AlexNet使用的卷积通道数也大于LeNet中的卷积通道数数十倍. 紧接着最后一个卷积层的是两个输出个数为4,096的全连接层. 这两个巨大的全连接层带来将近1 GB的模型参数. 由于早期显存的限制, 最早的AlexNet使用双数据流的设计使一块GPU只需要处理一半模型. 幸运的是, 显存在过去几年得到了长足的发展, 因此通常我们不再需要这样的特别设计了.

第二, AlexNet将sigmoid激活函数改成了更加简单的ReLU激活函数. 一方面, ReLU激活函数的计算更简单, 例如它并没有sigmoid激活函数中的求幂运算. 另一方面, ReLU激活函数在不同的参数初始化方法下使模型更容易训练. 这是由于当sigmoid激活函数输出极接近0或1时, 这些区域的梯度几乎为0, 从而造成反向传播无法继续更新部分模型参数；而ReLU激活函数在正区间的梯度恒为1. 因此, 若模型参数初始化不当, sigmoid函数可能在正区间得到几乎为0的梯度, 从而令模型无法得到有效训练.

第三, AlexNet使用dropout.

第四, AlexNet引入了大量的图像增广, 如翻转、裁剪和颜色变化, 从而进一步扩大数据集来缓解过拟合.

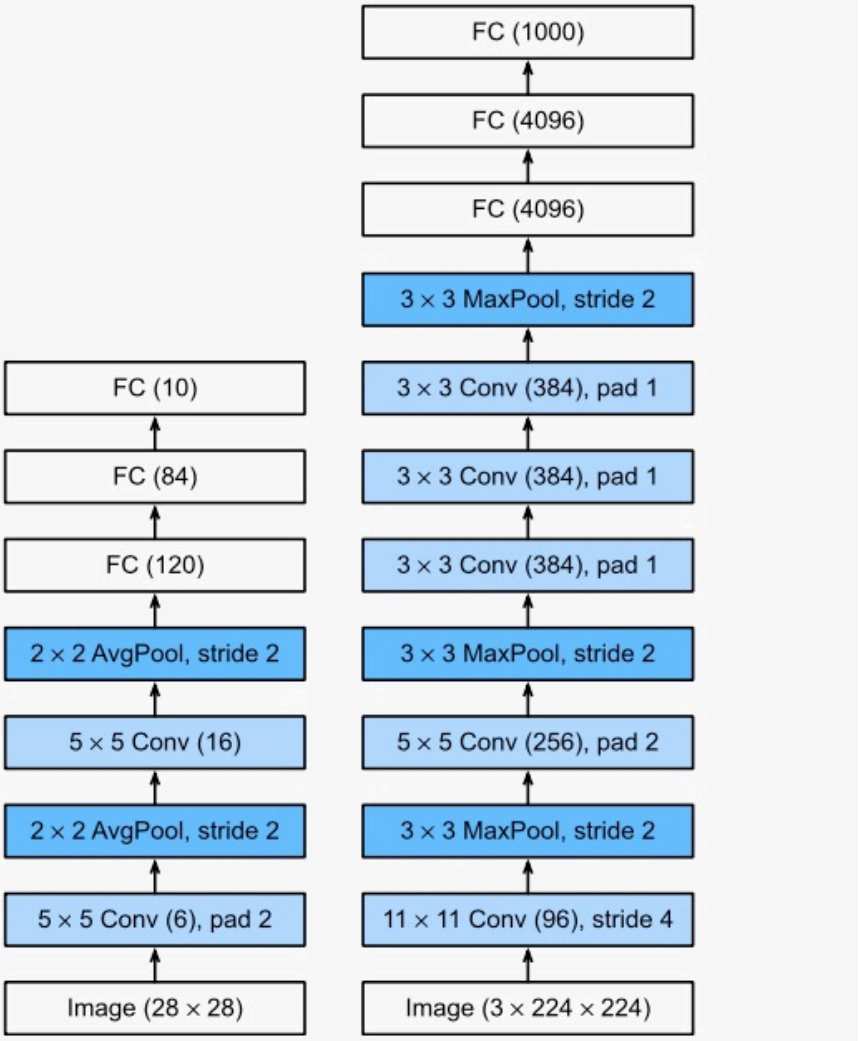


Fig. 7.1.2 From LeNet (left) to AlexNet (right).

ZfNet

对中间层(Conv, Relu)的计算状态进行了可视化,提出AlexNet只有很少的神经元都是active的.

VGG

第一个使用相同结构的模块不断迭代的构建网络，作为提取特征的单元，大大降低了大模型构建的难度，同时使用小的卷积核代替大的卷积核.

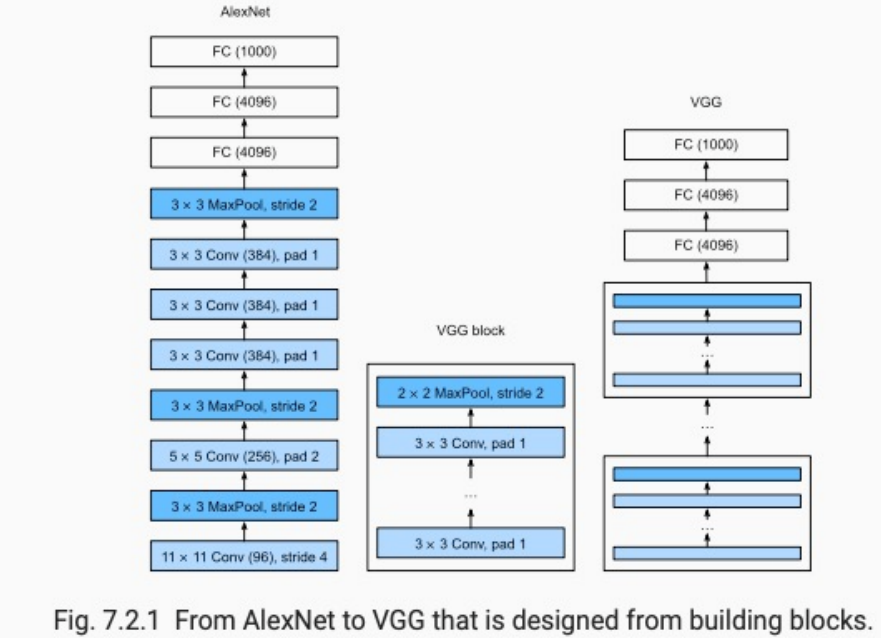


Fig. 7.2.1 From AlexNet to VGG that is designed from building blocks.

GoogLeNet

简介

$GoogleNet$主要贡献:

1. 使用包含$1 \times 1, 3 \times 3, 5 \times 5$的卷积,提取不同维度的空间特征,使得同一个目标在不同大小,不同形态的情况下都有比较好的识别效果.
2. 使用1×1的卷积层对图片进行预处理
3. 使用$GlobalAvgPooling$代替全连接层,降低了计算复杂度(138 million to 4 million).
4. $RmsProp$作为优化算法,以及$BatchNormalization$ 的应用.

主要缺陷：

1. 需要在模块之间自定义的异构拓扑.
2. 每一个block之间会大大减少特征参数,导致信息丢失,模型的表示能力降低.

直到GoogLeNet出来之前, 大家的主流的效果突破大致是网络更深, 网络更宽. 但是纯粹的增大网络有两个缺点:过拟合和计算量的增加. 同时还有梯度弥散问题. 方法当然就是增加网络深度和宽度的同时减少参数.

但结构稀疏性和运算能力有矛盾, 需要既能保持网络结构的稀疏性, 又能利用密集矩阵的高计算性能.基于的原理：相互独立的特征越多, 输入的信息就被分解的越彻底, 分解的子特征间相关性低, 子特征内部相关性高, 把相关性强的聚集在了一起会更容易收敛. 这点就是Hebbin原理.

Inception block

```python
class Inception(nn.Module):
    def __init__(self, in_channels, ch1x1, ch3x3red, ch3x3, ch5x5red, ch5x5, pool_proj):
        super(Inception, self).__init__()

        self.branch1 = BasicConv2d(in_channels, ch1x1, kernel_size=1)

        self.branch2 = nn.Sequential(
            BasicConv2d(in_channels, ch3x3red, kernel_size=1),
            BasicConv2d(ch3x3red, ch3x3, kernel_size=3, padding=1)   # 保证输出大小等于输入大小
        )

        self.branch3 = nn.Sequential(
            BasicConv2d(in_channels, ch5x5red, kernel_size=1),
            BasicConv2d(ch5x5red, ch5x5, kernel_size=5, padding=2)   # 保证输出大小等于输入大小
        )

        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            BasicConv2d(in_channels, pool_proj, kernel_size=1)
        )

    def forward(self, x):
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)

        outputs = [branch1, branch2, branch3, branch4]
        return torch.cat(outputs, 1)
```

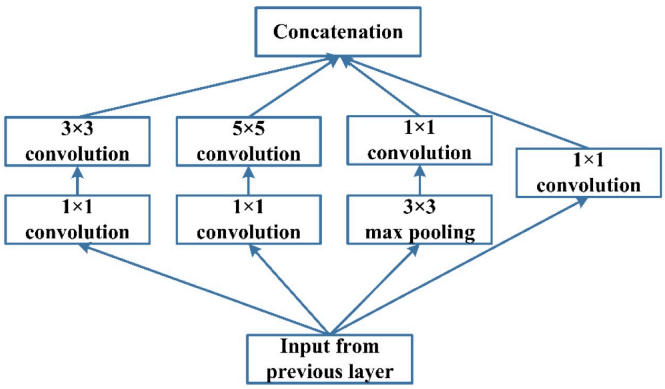
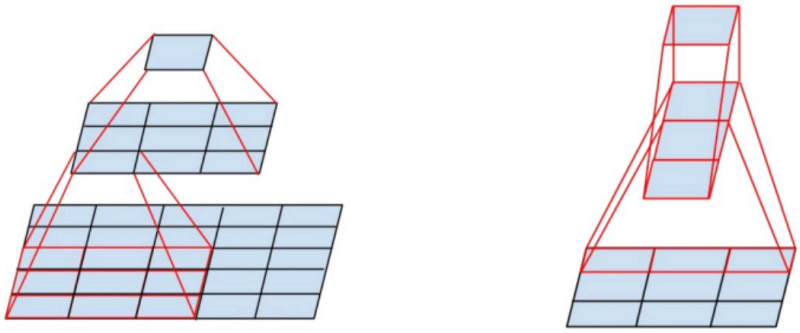


Fig. 6 Basic architecture of the inception block showing the split, transform, and merge concept.

1. 采用不同大小的卷积核意味着不同大小的感受野, 最后拼接意味着不同尺度特征的融合；
2. 之所以卷积核大小采用1、3和5, 主要是为了方便对齐. 设定卷积步长stride=1之后, 只要分别设定pad=0、1、2, 那么卷积之后便可以得到相同维度的特征, 然后这些特征就可以直接拼接在一起了；

Inception-V3

1. 使用小的卷积核代替大的卷积核,减少计算量,如下.



   5×5分解为两个3×3, 分解前和分解后的感受野是一样的. 两个3×3卷积的串联比一个5×5卷积的representation能力更强. 另外, 分解后多使用了一个激活函数, 增加了分线性能力(图2). 两个3×3卷积和一个5×5卷积的参数量的比为$\frac{9+9}{25}$, 分解减少了28%的参数,同时计算量也减少了28%.
   同时,通过卷积的非对称分解可以将3x3卷积分解为1×3和3×1卷积. 在输入输出filters数目一定的时候, 卷积的非对称分解可以将计算量减少33%.
2. 在通过卷积核比较大的层之前加入了1×1的卷积层调整维度.
3. feature map的size的高效减小(Efficient Grid Size Reduction):

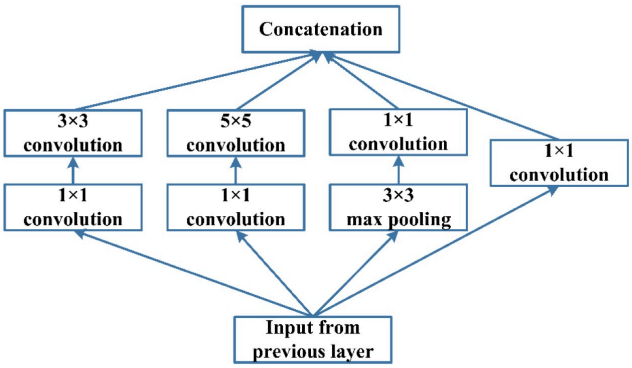


Fig. 6 Basic architecture of the inception block showing the split, transform, and merge concept.

4. Inception-ResNet中,Szegedy结合了Residual Block,使用了残差连接改进了Inception,将Concate操作改成了残差连接,发现在深度和宽度增加的同时,模型收敛速度更快了.

Depth based CNNs

Highway

第一个提出来跨层连接机制,但是性能不如ResNet,这里我认为原因是因为使用了Sigmoid激活函数.

ResNet

> 简介

深层的卷积神经网络给图像分类领域带来巨大的突破, 但是更深的神经网络往往很难训练, 会带来更高的训练和测试error, 同时还带来以下两个问题:
vanishing/exploding gradients：针对该问题有normalized initialization和intermediate normalization来解决, 但这两种方法也只在几十层内有效, 随着网络深度的增加, BP算法引起的梯度消失/爆炸问题愈发严重;
degradation problem：层数越深, 除了不好训练, 训练错误率不会减小, 反而也会增加, 这也说明并非所有的系统都能很好的优化.

> 基于以上两个问题(主要是第二个), 作者提出了一种残差网络:

Residual Block

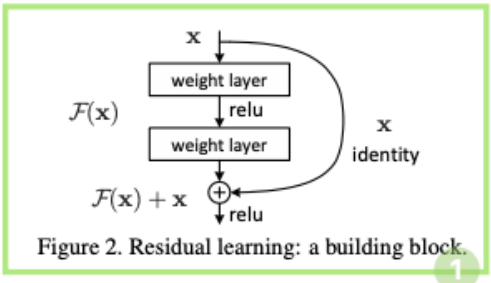


Figure 2. Residual learning: a building block.

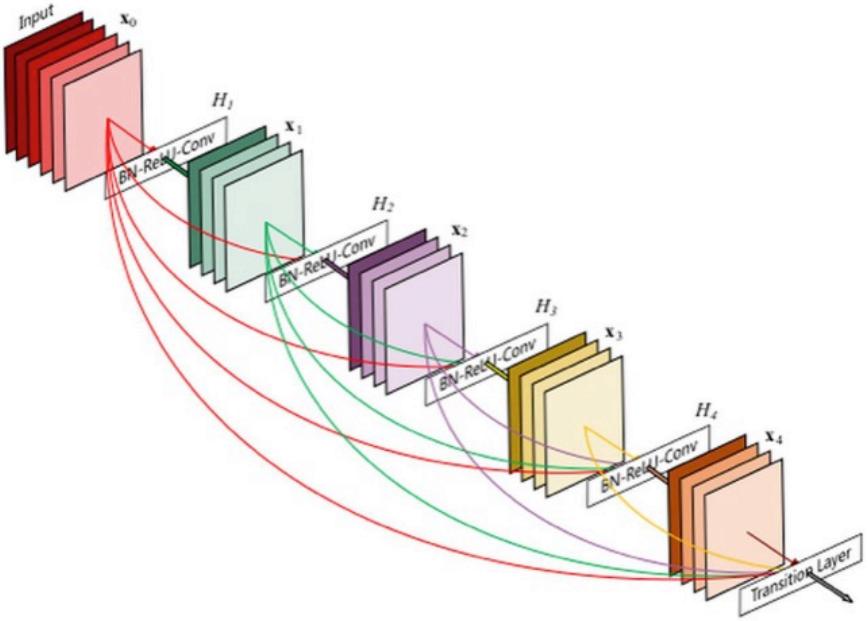
```python
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Optional[Callable[..., nn.Module]] = None
    ) -> None:
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
```

```
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        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)

        return out
```

Multi-Path based CNNs

DenseNet

> **简介:**
>
> 和ResNet 的主要区别是,这篇论文提出一个结构提炼了这个洞察$(insight)$到简单连通性模式：为了保证最大信息在网络的之间流动, 直接将所有的层与其他层连接(通过匹配特征层$feature\ map$的大小)为了保持前馈特性, 每层都有从前继层额外输入, 传递自己的$feature\ map$的信息. 与$ResNet$对比, 他们采用将特征连接$(concatenate)$. 因此, 第l层有l个输入,是所有前驱$feature\ map$块组成. 而它的特征是被送往$(L-l)$个后继层. 这样L层就有$L(L+1)/2$个连接, 而不是传统的L个连接. 所以它被命名为Dense Convolutional Network (DenseNet).它的优点是相比于ResNet,网络之间不同层的信息传递效率更高了.

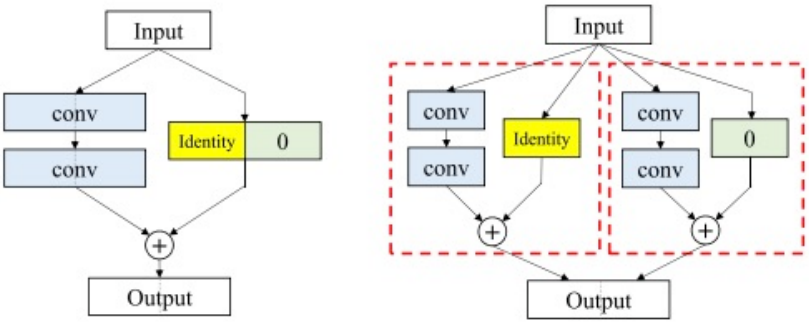


Width based Multi-Connection CNNs

Pyramidal Net

> 简介

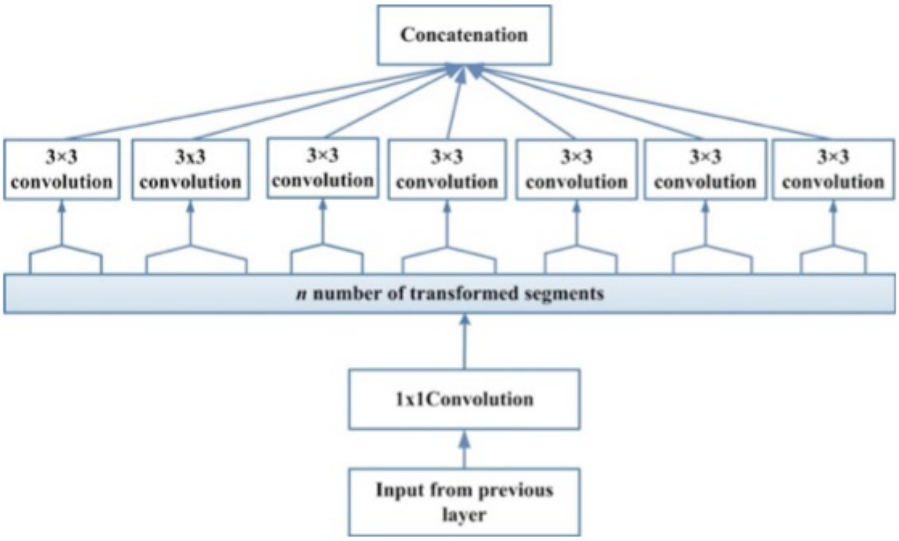
在之前的模型比如ResNet中,在层数增加时,特征图的深度通过卷积层增加了,但是空间维度下降了(使用降采样层),导致特征表示能力下降,导致性能下降. 于是此文作者舍弃了了$residual\ unit$, $Pyramidal\ Net$的特点是每层的宽度都会逐步增加,增加了模型的表示能力.下图左侧是$residual\ unit$,右侧是$Pyramidal\ Net$的实现方式.



Xception

> ### 简介
>
> 将 $Inception\,block$ 网络变得更宽了.传统模型一般使用一个变换操作,$Inceptionblock$ 使用三个变换操作,$Xceptionblock$ 的变换个数等于通道个数, 使用1×1卷积和通道空间卷积(对每个维度分别卷积),增加了模型的表示能力,而且降低了计算复杂度.
> Xception模块与深度可分离卷积之间的两个小区别是:

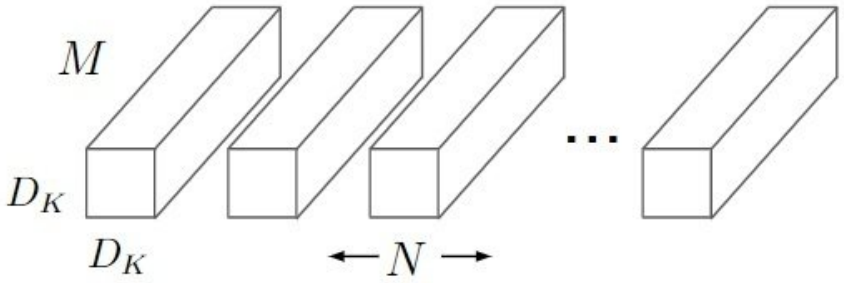
1. 操作顺序：通常实现的深度可分离卷积(例如在TensorFlow中)首先执行通道空间卷积(深度卷积), 然后执行1x1卷积, 而Inception首先执行1x1卷积.
2. 第一次操作后是否存在非线性. 在Inception中, 两个操作都跟随ReLU非线性, 但是通常在没有非线性的情况下实现深度可分离卷积.



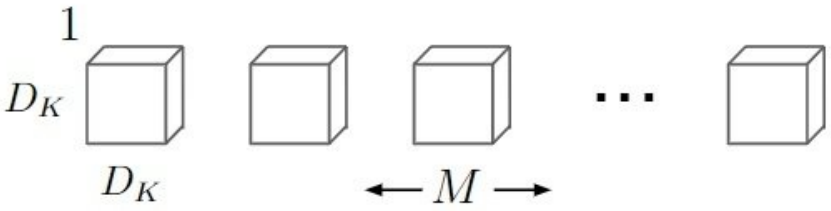
MobileNetV1-V2

> ### 简介

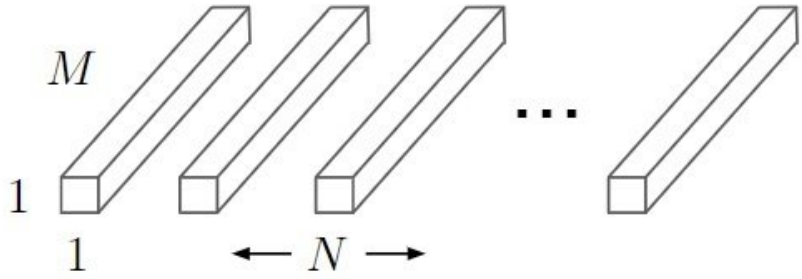
google 201704在archive上的论文.
采用**depthwise separable**卷积核, 减少计算量和模型大小.
MobileNet是为移动和嵌入式视觉应用设计的模型, 它是使用了深度可分离卷积的轻型流式结构, 如下:



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

原始卷积计算量:
$D_k \cdot D_k \cdot M \cdot N \cdot D_F \cdot D_F$
深度可分卷积计算量:
$D_k \cdot D_k \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$

其中引入了两个简单的全局超参数权衡速度和精度, 它们可以根据不同应用约束设置不同的模型规模.
宽度乘子：减小通道数
尽管基本的MobileNet体系结构已经很小并且延迟很短, 但是很多情况下, 特定的用例或应用程序可能要求模型变得更小, 更快. 为了构造这种模型, 引入了一个非常简单的参数 α称为宽度乘子. 宽度乘子的作用是使每一层的网络均匀地变薄. 通常设定为1, 0.75, 0.5, 0.25, 可以对网络做有效的计算量缩减.

分辨率乘子：减小特征

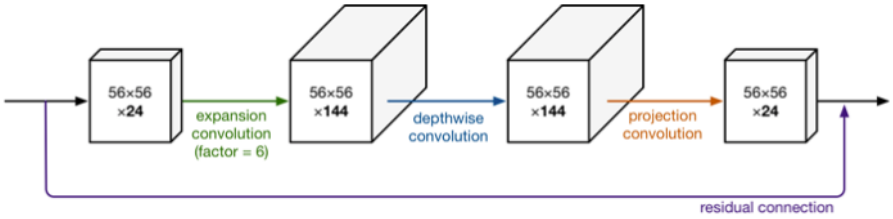
第二个减小计算量的超参数是分辨率乘子ρ , 通过将输入图片乘以乘子, 可以减小计算量.

```
class ConvBNReLU(nn.Sequential):
```

```
    def __init__(self, in_channel, out_channel, kernel_size=3, stride=1, groups=1):
        padding = (kernel_size - 1) // 2
        super(ConvBNReLU, self).__init__(
            nn.Conv2d(in_channel, out_channel, kernel_size, stride, padding, groups=groups, bias=False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU6(inplace=True)
        )


class InvertedResidual(nn.Module):
    def __init__(self, in_channel, out_channel, stride, expand_ratio):
        super(InvertedResidual, self).__init__()
        hidden_channel = in_channel * expand_ratio
        self.use_shortcut = stride == 1 and in_channel == out_channel

        layers = []
        if expand_ratio != 1:
            # 1x1 pointwise conv
            layers.append(ConvBNReLU(in_channel, hidden_channel, kernel_size=1))
        layers.extend([
            # 3x3 depthwise conv
            ConvBNReLU(hidden_channel, hidden_channel, stride=stride, groups=hidden_channel), # key !!!
            # 1x1 pointwise conv(linear)
            nn.Conv2d(hidden_channel, out_channel, kernel_size=1, bias=False),
            nn.BatchNorm2d(out_channel),
        ])

        self.conv = nn.Sequential(*layers)

    def forward(self, x):
        if self.use_shortcut:
            return x + self.conv(x)
        else:
            return self.conv(x)
```

Mobilenet V2:

1. $ReLU6(x) = min(max(x, 0), 6)$,
2. Inverted residual block: 先升维度再降维度, 保证了参数下降时信息能够更好的表示.
3. relu在低纬特征信息损失较大. 激活函数使用linear, 使用了shortcut链接(输入特征和输出特征相同时).
4. MobileNetV1网络主要思路就是深度可分离卷积的堆叠. 在V2的网络设计中, 我们除了继续使用深度可分离(中间那个)结构之外, 还使用了Expansion layer和 Projection layer. 这个projection layer也是使用 1×1 的网络结构, 他的目的是希望把高维特征映射到低维空间去.
5. Expansion layer的功能正相反, 使用1×1的网络结构, 目的是将低维空间映射到高维空间. 这里Expansion有一个超参数是维度扩展几倍. 可以根据实际情况来做调整的, 默认值是6, 也就是扩展6倍. 效果如下图:

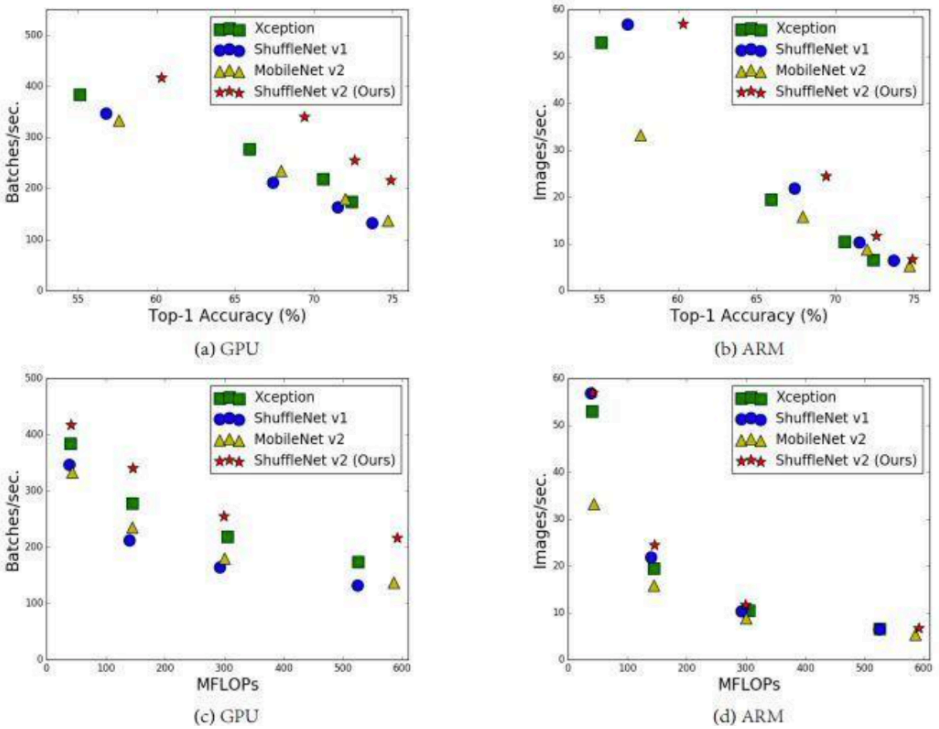


> 原因:如果tensor维度越低, 卷积层的乘法计算量就越小. 那么如果整个网络都是低维的tensor, 那么整体计算速度就会很快. 然而, 如果只是使用低维的tensor效果并不会好. 如果卷积层的过滤器都是使用低维的tensor来提取特征的话, 那么就没有办法提取到整体的足够多的信息. 所以, 如果提取特征数据的话, 我们可能更希望有高维的tensor来做这个事情. V2就设计这样一个结构来达到平衡.

---

shufflenet_v2:

> ## 简介:
>
> 在ShuffleNet v2这篇paper中, 作者们重点分析了影响在GPU/ARM两种平台上CNN网络计算性能的几个主要指标, 并提出了一些移动端CNN网络设计的指导准则, 最终将这些指导准则应用于ShuffleNet v1网络的改良就行成了ShuffleNet v2. 在分类与目标检测等通用任务时与mobilenet相比, 它都取得了不俗的性能.



高效CNN网络设计的四个准则:

下图中分析了ShuffleNet v1与MobileNet v2这两个移动端流行网络在GPU/ARM两种平台下的时间消耗分布.

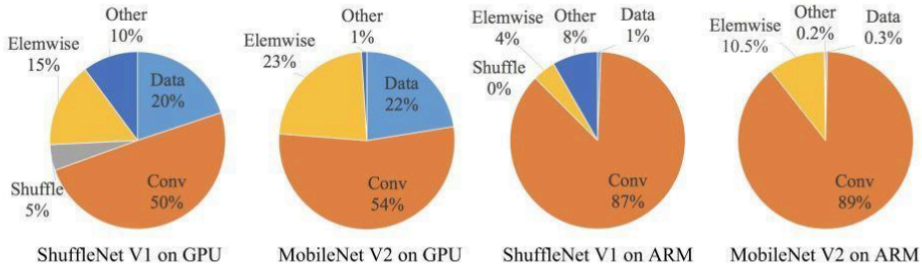
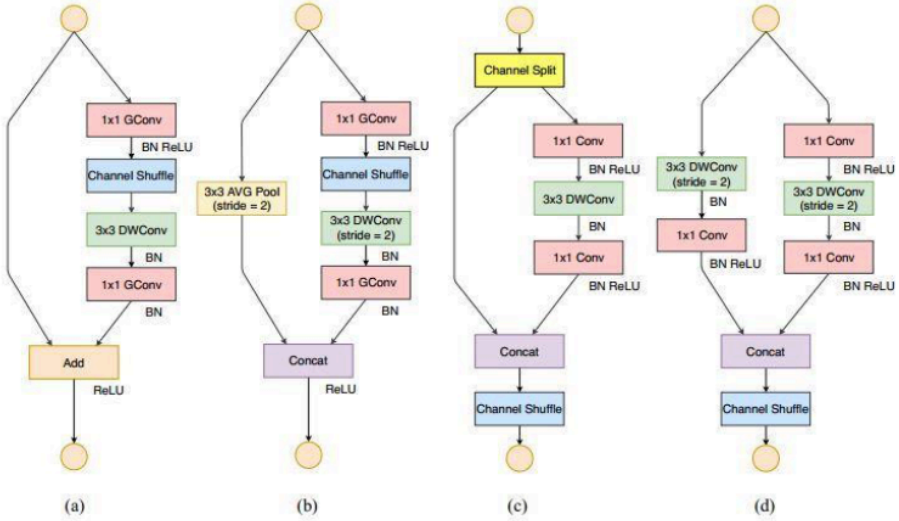


Fig. 2: Run time decomposition on two representative state-of-the-art network architectures, *ShuffeNet v1* [15] $(1\times, g = 3)$ and *MobileNet v2* [14] $(1\times)$.

从上图中可看出Conv等计算密集型操作占了其时间的绝大多数, 但其它像Elemwise/Data IO等内存读写密集型操作也占了相当比例的时间, 因此像以往那样一味以FLOPs来作为指导准则来设计CNN网络是不完备的, 虽然它可以反映出占大比例时间的Conv操作.

1. 当输入、输出channels数目相同时, conv计算所需的MAC(memory access cost)最为节省.
2. 过多的Group convolution会加大MAC开销.
3. 网络结构整体的碎片化会减少其可并行优化的程序.
4. Element-wise操作会消耗较多的时间, 不可小视.

下图中的a/b为原ShuffleNet v1中具有的两种模块结构. 图c/d则为ShuffleNet v2中的模块设计.



ShuffleNet v2中弃用了1x1的group convolution操作, 而直接使用了input/output channels数目相同的1x1普通conv. 它更是提出了一种ChannelSplit新的类型操作, 将module的输入channels分为两部分, 一部分直接向下传递, 另外一部分则进行真正的向后计算. 到了module的末尾, 直接将两分支上的output channels数目级连起来, 从而规避了原来ShuffleNet v1中Element-wise sum的操作. 然后我们再对最终输出的output feature maps进行RandomShuffle操作, 从而使得各channels之间的信息相互交通.
跟ShuffleNet v1一样, 它也提供了一种需要downsampling的模块变形. 为了保证在下采样的时候增加整体输出channels数目, 它取消了模块最开始时的RandomSplit操作, 从而将信处向下分别处理后再拼结, 使得最终output channels数目实现翻倍.

```
1
```

```
def conv_bn(inp, oup, stride):
    return nn.Sequential(
        nn.Conv2d(inp, oup, 3, stride, 1, bias=False),
        nn.BatchNorm2d(oup),
        nn.ReLU(inplace=True)
    )


def conv_1x1_bn(inp, oup):
    return nn.Sequential(
        nn.Conv2d(inp, oup, 1, 1, 0, bias=False),
        nn.BatchNorm2d(oup),
        nn.ReLU(inplace=True)
    )

def channel_shuffle(x, groups):
    batchsize, num_channels, height, width = x.data.size()

    channels_per_group = num_channels // groups

    # reshape
    x = x.view(batchsize, groups,
        channels_per_group, height, width)

    x = torch.transpose(x, 1, 2).contiguous()

    # flatten
    x = x.view(batchsize, -1, height, width)

    return x

class InvertedResidual(nn.Module):
    def __init__(self, inp, oup, stride, benchmodel):
        super(InvertedResidual, self).__init__()
        self.benchmodel = benchmodel
        self.stride = stride
        assert stride in [1, 2]

        oup_inc = oup//2

        if self.benchmodel == 1:
            #assert inp == oup_inc
          self.banch2 = nn.Sequential(
                # pw  key
                nn.Conv2d(oup_inc, oup_inc, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup_inc),
                nn.ReLU(inplace=True),
                # dw  key
                nn.Conv2d(oup_inc, oup_inc, 3, stride, 1, groups=oup_inc,
bias=False),
                nn.BatchNorm2d(oup_inc),
                # pw-linear
                nn.Conv2d(oup_inc, oup_inc, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup_inc),
                nn.ReLU(inplace=True),
```

```
            )
        else:
            self.banch1 = nn.Sequential(
                # dw
                nn.Conv2d(inp, inp, 3, stride, 1, groups=inp, bias=False),
                nn.BatchNorm2d(inp),
                # pw-linear
                nn.Conv2d(inp, oup_inc, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup_inc),
                nn.ReLU(inplace=True),
            )

            self.banch2 = nn.Sequential(
                # pw
                nn.Conv2d(inp, oup_inc, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup_inc),
                nn.ReLU(inplace=True),
                # dw
                nn.Conv2d(oup_inc, oup_inc, 3, stride, 1, groups=oup_inc,
bias=False),
                nn.BatchNorm2d(oup_inc),
                # pw-linear
                nn.Conv2d(oup_inc, oup_inc, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup_inc),
                nn.ReLU(inplace=True),
            )

    @staticmethod
    def _concat(x, out):
        # concatenate along channel axis
        return torch.cat((x, out), 1)

    def forward(self, x):
        if 1==self.benchmodel:
            x1 = x[:, :(x.shape[1]//2), :, :]
            x2 = x[:, (x.shape[1]//2):, :, :]
            out = self._concat(x1, self.banch2(x2))
        elif 2==self.benchmodel:
            out = self._concat(self.banch1(x), self.banch2(x))

        return channel_shuffle(out, 2)

```