**网络发展脉络：**

**2012 AlexNet**

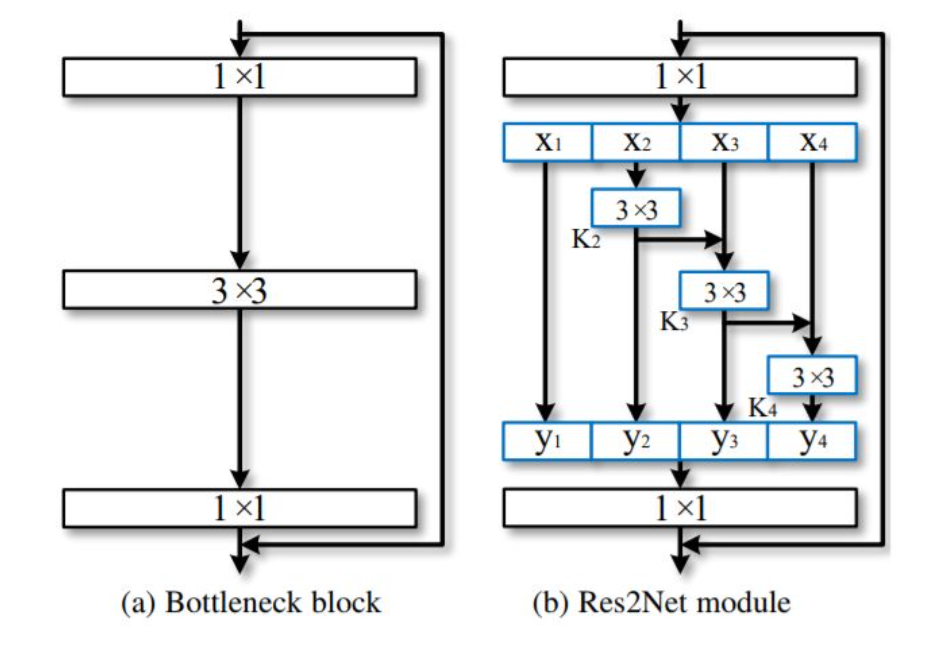
**2014 VGGNet， GoodleNet**

**2015 ResNet**

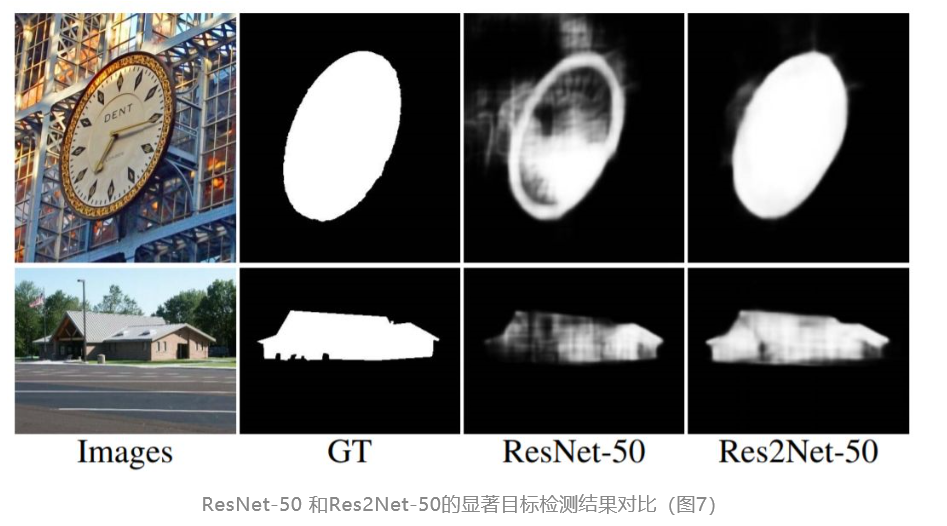
1. **DenseNet, SeNet**
2. **ResNeXt**
3. **Res2Net**

论文：[**https://arxiv.org/pdf/1904.01169.pdf**](https://arxiv.org/pdf/1904.01169.pdf)

多尺度表示特征，以更细粒度（granular level）表示多尺度特征，并增加每个网络层的感受野（receptive fields）范围。



Res2Net性能上的优越性已经在几个具有代表性的计算机视觉任务体现出来，包括类激活映射，对象检测和显着对象检测等。多尺度表示对于未来开拓更广泛的应用领域至关重要。



1. **SENet（Squeeze-and-Excitation Networks）**

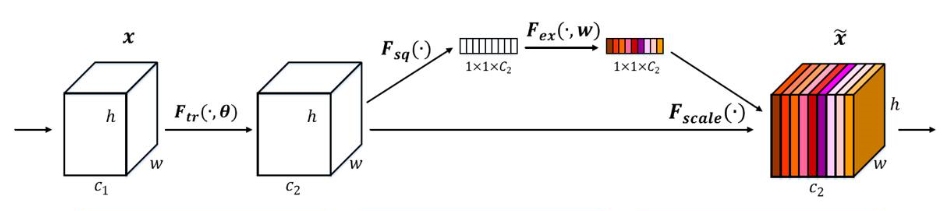
**论文：**[**https://arxiv.org/abs/1709.01507**](https://arxiv.org/abs/1709.01507)

**代码：**[**https://github.com/moskomule/senet.pytorch**](https://github.com/moskomule/senet.pytorch)

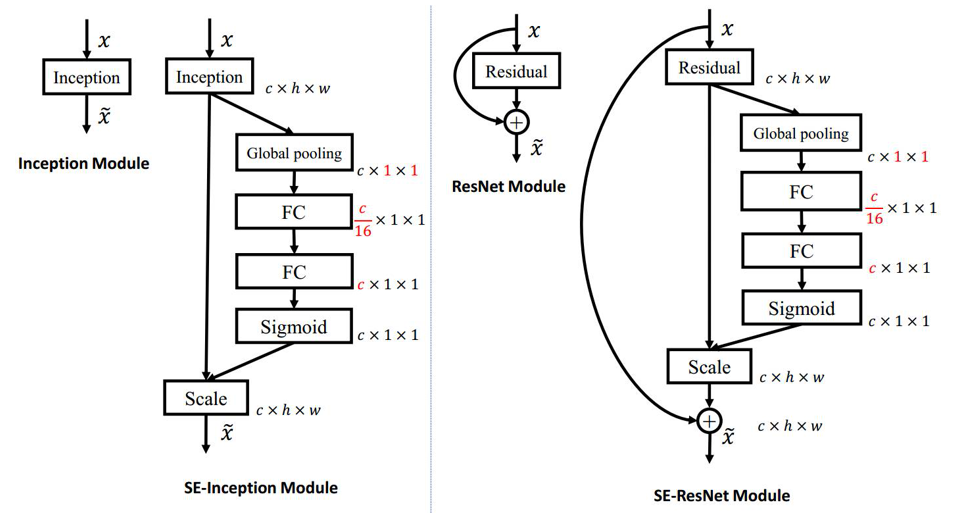
压缩激活网络(Squeeze-and-Excitation Networs)，从特征通道维度上来提升网络性能（首次提出），

Feature recalibration: Selectively enhance useful features and suppress less useful ones（特征重校准：选择性地提高重要特征和抑制不重要的特征）

本质是在提取特征后对特征通道做加权，权重由网络自动学习，可将SE模块嵌入到Inception、ResNet等结构中，提升网络对特征的学习能力，引入的额外参数量和计算量很小



1. Squeeze 操作：顺着空间维度来进行特征压缩，将每个二维的特征通道变成一个实数，即用Globalpooling对计算每个通道的特征的均值来表示这个通道的所有特征，
2. Excitation操作：用fc->relu->fc->sigmoid来获得每个通道的权重
3. Scale操作：将学好的权重加权到对应通道上



SEblock pytorch实现：

from torch import nn

class SELayer(nn.Module):

def \_\_init\_\_(self, channel, reduction=16):

super(SELayer, self).\_\_init\_\_()

self.avg\_pool = nn.AdaptiveAvgPool2d(1)

self.fc = nn.Sequential(

nn.Linear(channel, channel//reduction, bias=False),

nn.ReLU(inplace=True),

nn.Linear(channel//reduction, channel, bias=False),

nn.Sigmoid()

)

def forward(self, x):

b, c, \_, \_ = x.size()

y = self.avg\_pool(x).view(b, c) # squeeze

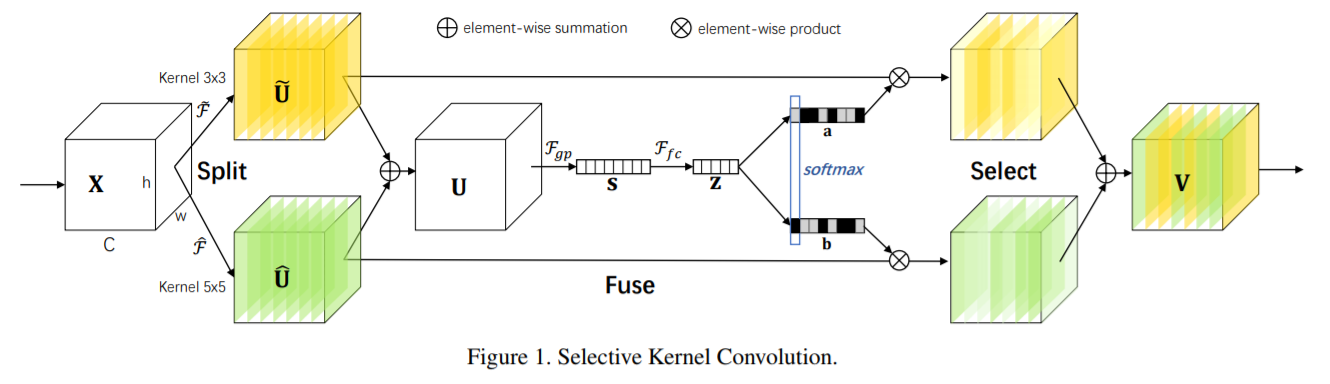
y = self.fc(y).view(b, c, 1, 1) # excitation

return x \* y.expand\_as(x) # scale

1. **SKNet（Selective Kernel Networks）**

论文：<https://arxiv.org/pdf/1903.06586.pdf>

代码：https://github.com/implus/SKNet



**轻量级模型：**

2016.02 SqueezeNet

2016.04 MobileNet

2016.06 ShuffleNet

* 1. ception

**1. SqueezeNet**

提出 fire module；fire module 包含两部分：squeeze 层+expand 层

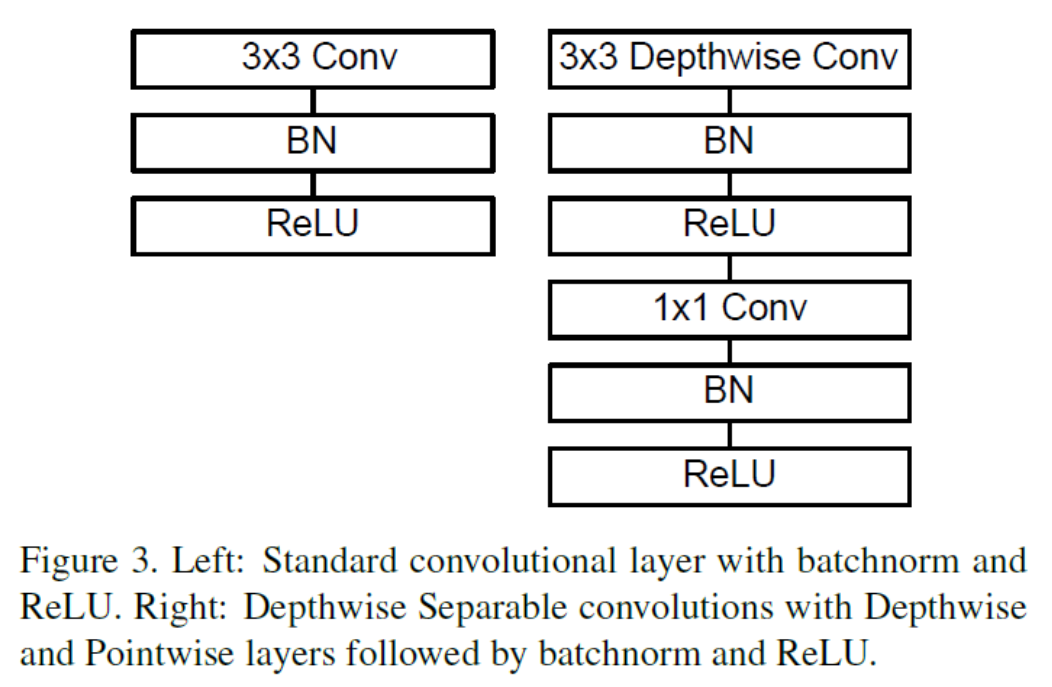
创新点与 inception 系列的思想非常接近，squeeze 层是一个 1\*1 卷积核的卷积层，其卷积核数要少于上一层feature map数。expand 层分别用 1\*1 和 3\*3 卷积，然后 concat，这两个个操作在 inception 系列里面也有。fire module有三个可调参数：S1，e1，e3，在文中提出的 SqueezeNet 结构中，e1=e3=4s1

1. **mobileNet-v1**

把普通卷积分解成深度卷积(depthwise convolution，首次提出)和逐点卷积(pointwise convolution),假设对尺寸为WxW,通道数为M的特征图做卷积，得到输出为N通道的特征图，需要大小为KxK通道为M的标准卷积核N个，即(K,K,M,N)可将其拆分为深度卷积和点卷积：

a.深度卷积负责滤波作用,尺寸为 (K,K​,1,M)，输出的特征通道为M(与输入特征通道数一样)

b.点卷积负责转换通道，尺寸为 (1,1,M,N)，得到最终输出通道数为N



普通卷积参数量为：K \* K \*M \* N

深度卷积+点卷积参数量为：K \* K \* M + M \* N

Pytorch 实现：

**class** MobileNetV1(nn.Module):  
 *"""Depthwise conv + Pointwise Conv """* **def** \_\_init\_\_(self, in\_channel, out\_channel, stride=1):  
 super(MobileNetV1, self).\_\_init\_\_()  
 self.mobilenet\_block = nn.Sequential(  
 nn.Conv2d(in\_channel, in\_channel, kernel\_size=3, stride=stride,  
 padding=1, groups=in\_channel, bias=**False**),  
 nn.BatchNorm2d(in\_channel),  
 nn.ReLU(**True**),  
 nn.Conv2d(in\_channel, out\_channel, kernel\_size=1, stride=1,   
 padding=0, bias=**False**),  
 nn.BatchNorm2d(out\_channel),  
 nn.ReLU(**True**)  
 )  
   
 **def** forward(self, x):  
 out = self.mobilenet\_block(x)  
 **return** out

1. **MobileNet-v2**

论文：<https://arxiv.org/pdf/1801.04381v4.pdf>

1. **ShuffleNet-v1**

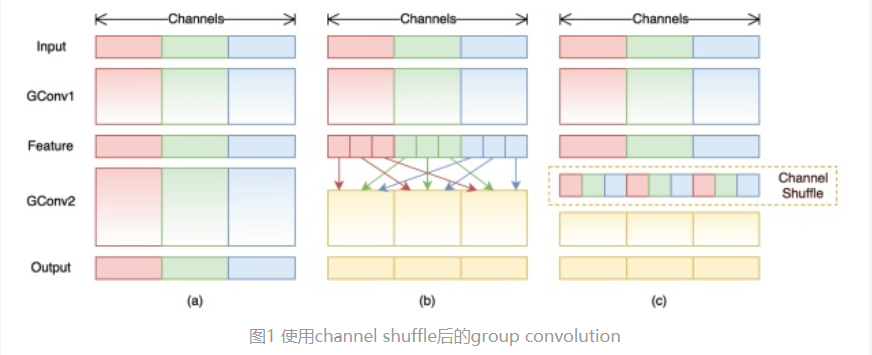
资料：<https://zhuanlan.zhihu.com/p/32304419>

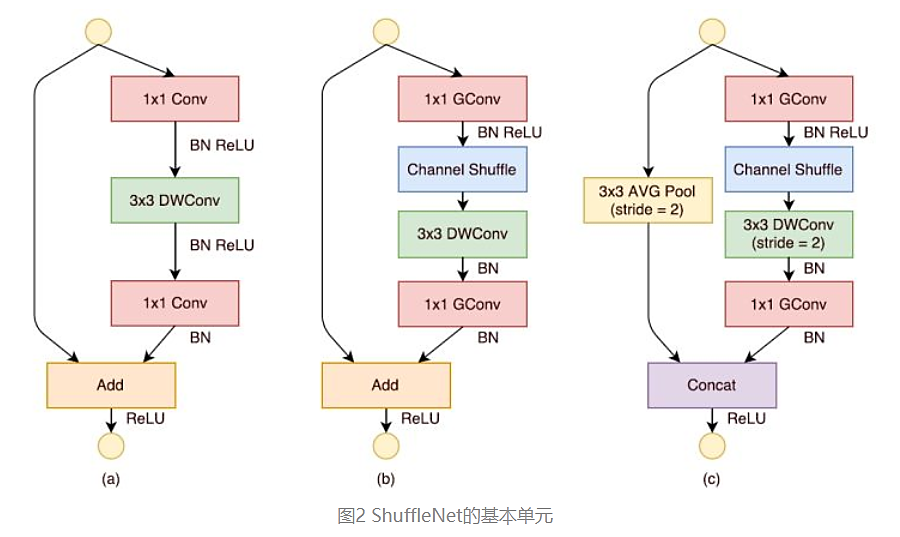
论文：<https://arxiv.org/pdf/1707.01083.pdf>

主要采用两种操作：pointwise group convolution 和 channel shuffle,在保持精度的同时大大降低了模型的计算量，

贡献：在group convolution后接上channel shuffle来打乱通道（这里是均匀的打乱），解决group convolution的各组通道直接没有通信的弊端，

在程序上实现channel shuffle：假定将输入层分为g组，每组n个通道，则总通道数为gxn，首先将通道的size reshape为[g,n]，然后转置成[n,g]，最后reshape成一个维度





Pytorch实现：

import torch

import torch.nn as nn

import torch.nn.functional as F

*# channel shuffle*

**class** Channel\_Shuffle(nn.Module):  
 **def** \_\_init\_\_(self, groups):  
 super(Channel\_Shuffle, self).\_\_init\_\_()  
 self.groups = groups  
  
 **def** forward(self, x):  
 batch\_size, channels, height, width = x.size()  
 **assert** channels % self.groups == 0  
 channels\_per\_group = channels // self.groups  
 *# split into groups* x = x.view(batch\_size, self.groups, channels\_per\_group, height, width)  
 *# transpose 1, 2 axis* x = x.transpose(1, 2).contiguous()  
 *# reshape into original* x = x.view(batch\_size, channels, height, width)  
 **return** x  
  
*# ShuffleNet中stride=1的基本单元***class** ShuffleNetV1Unit1(nn.Module):  
 *"""ShuffleNet unit for stride=1"""* **def** \_\_init\_\_(self, in\_channels, out\_channels, groups=2):  
 super(ShuffleNetV1Unit1, self).\_\_init\_\_()  
 **assert** in\_channels == out\_channels  
 **assert** out\_channels % 4 ==0  
 bottleneck\_channels = out\_channels // 4  
  
 self.shufflenet\_block = nn.Sequential(  
 *# 1x1 GConv* nn.Conv2d(in\_channels, bottleneck\_channels, kernel\_size=1, stride=1, padding=0,groups=groups),  
 nn.BatchNorm2d(bottleneck\_channels),  
 nn.ReLU(**True**),  
 *# channel shuffle* Channel\_Shuffle(groups=groups),  
 *# 3x3 DWConv* nn.Conv2d(bottleneck\_channels, bottleneck\_channels, kernel\_size=3, padding=1, stride=1,  
 groups=bottleneck\_channels),  
 nn.BatchNorm2d(bottleneck\_channels),  
 *# 1x1 GConv* nn.Conv2d(bottleneck\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, groups=groups),  
 nn.BatchNorm2d(out\_channels)  
 )  
 self.relu = nn.ReLU(**True**)  
  
 **def** forward(self, x):  
 out = self.relu(x + self.shufflenet\_block(x))  
 **return** out

# ShuffleNet中stride=2的基本单元

class ShuffleNetV1Unit2(nn.Module):

"""ShuffleNetV1 unit for stride=2"""

def \_\_init\_\_(self, in\_channels, out\_channels, groups=3):

super(ShuffleNetV1Unit2, self).\_\_init\_\_()

out\_channels -= in\_channels

assert out\_channels % 4 == 0

bottleneck\_channels = out\_channels // 4

self.groups = groups

self.group\_conv1 = nn.Conv2d(in\_channels, bottleneck\_channels,

kernel\_size=1, stride=1, groups=groups)

self.bn1 = nn.BatchNorm2d(bottleneck\_channels)

self.depthwise\_conv = nn.Conv2d(bottleneck\_channels,

bottleneck\_channels,

kernel\_size=3, padding=1, stride=2,

groups=bottleneck\_channels)

self.bn2 = nn.BatchNorm2d(bottleneck\_channels)

self.group\_conv2 = nn.Conv2d(bottleneck\_channels, out\_channels,

kernel\_size=1, stride=1, groups=groups)

self.bn3 = nn.BatchNorm2d(out\_channels)

def forward(self, x):

out = self.group\_conv1(x)

out = F.relu(self.bn1(out))

out = shuffle\_channels(out, groups=self.groups)

out = self.depthwise\_conv (out)

out = self.bn2(out)

out = self.group\_conv2(out)

out = self.bn3(out)

x = F.avg\_pool2d(x, kernel\_size=3, stride=2, padding=1)

out = F.relu(torch.cat([x, out], dim=1))

return out

1. **ShuffleNet-v2**

资料：<https://zhuanlan.zhihu.com/p/48261931>

论文：<https://arxiv.org/abs/1807.11164>

衡量模型复杂度的指标是FLOPs（multiply-add的数量）

影响模型速度的因素不仅仅是FLOPs,与MAC（memory access cost）也有关：

1. 输入输出的feature map通道数相等时，内存访问量（MAC）最小，速度最快
2. Group convlution的分组增加时，MAC也同时增加
3. 网络碎片化会降低并行速度（比如Inception倾向于采用多路结果）
4. 元素级操作（如ReLU,Add）虽然FLOPs较小，但是MAC较大，会降低速度

优化网络：

1. 用1x1卷积平衡输入和输出的通道大小
2. 注意Group convolution的分组数
3. 避免网络碎片化
4. 减少元素级运算

根据以上4条准则改进ShuffleNet-v1得到ShuffleNet-v2

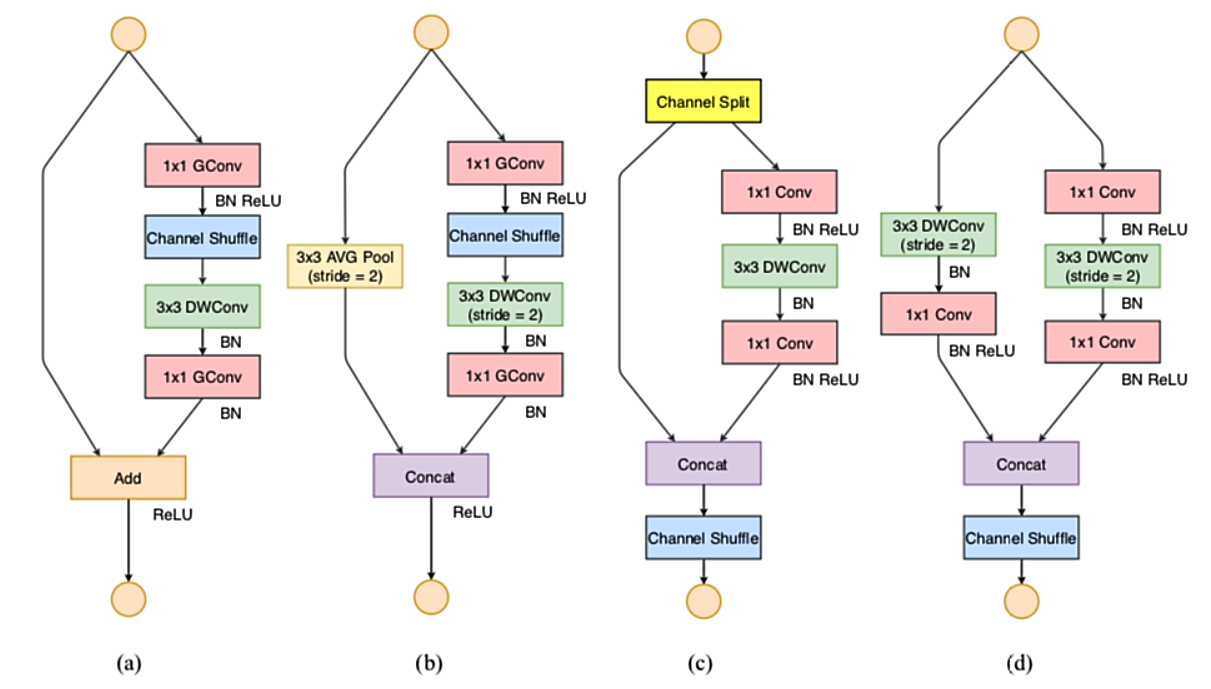


图1 (a)(b)ShuffleNet-v1 (c)(d)ShuffleNst-v2

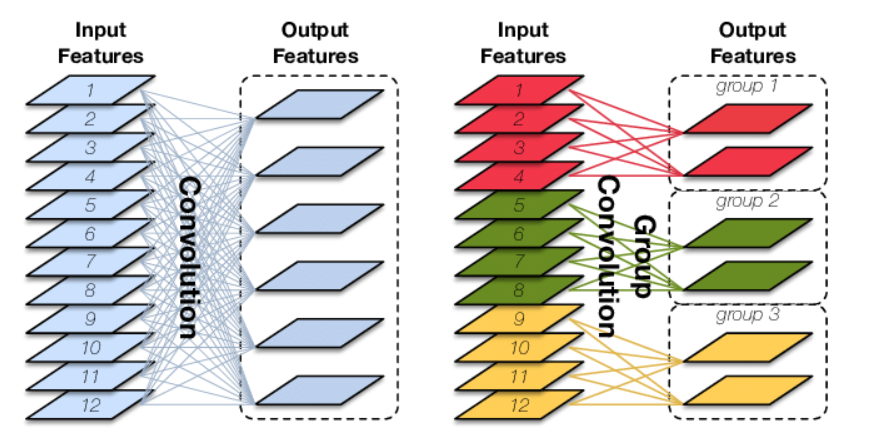
Python实现：

**class** Conv2d\_1x1(nn.Module):  
 *"""Conv2d\_1x1 -> BN -> ReLU """* **def** \_\_init\_\_(self, channel):  
 super(Conv2d\_1x1, self).\_\_init\_\_()  
 self.conv2d\_1x1 = nn.Sequential(  
 nn.Conv2d(channel, channel, kernel\_size=1, stride=1),  
 nn.BatchNorm2d(channel),  
 nn.ReLU(**True**)  
 )  
  
 **def** forward(self, x):  
 out = self.conv2d\_1x1(x)  
 **return** out  
  
**class** DepthwiseConv2d(nn.Module):  
 *"""DepthwiseConv2d -> BN """* **def** \_\_init\_\_(self, channel, kernel\_size=3, stride=1):  
 super(DepthwiseConv2d, self).\_\_init\_\_()  
 padding = kernel\_size // 2  
 self.depthwise\_conv2d = nn.Sequential(  
 nn.Conv2d(channel, channel, kernel\_size=kernel\_size,  
 stride=stride, padding=padding, groups=channel),  
 nn.BatchNorm2d(channel)  
 )  
 **def** forward(self, x):  
 out = self.depthwise\_conv2d(x)  
 **return** out  
  
**class** ShuffleNetV2Uint1(nn.Module):  
 **def** \_\_init\_\_(self, channel):  
 *""" ShuffleNetV2 unit for stride=1"""* super(ShuffleNetV2Uint1, self).\_\_init\_\_()  
 **assert** channel % 2 == 0  
 self.model = nn.Sequential(  
 Conv2d\_1x1(channel // 2),  
 DepthwiseConv2d(channel // 2, 3, 1),  
 Conv2d\_1x1(channel // 2)  
 )  
 self.channel\_shuffle = Channel\_Shuffle(groups=2)  
  
 **def** forward(self, x):  
 \_, c, \_, \_ =x.size()  
 shortcut, out = torch.split(x, c // 2, dim=1)

out = self.model(out)  
 out = torch.cat((shortcut, out), 1)  
 out = self.channel\_shuffle(out)  
 **return** out  
  
**class** ShuffleNetV2Uint2(nn.Module):  
 **def** \_\_init\_\_(self, in\_channel, out\_channel):  
 *"""ShuffleNetV2 unit for stride=2"""* super(ShuffleNetV2Uint2, self).\_\_init\_\_()  
 **assert** out\_channel % 2 == 0  
 self.model = nn.Sequential(  
 Conv2d\_1x1(out\_channel // 2),  
 DepthwiseConv2d(out\_channel // 2, 3, 2),  
 Conv2d\_1x1(out\_channel // 2)  
 )  
 self.shortcut = nn.Sequential(  
 DepthwiseConv2d(in\_channel, 3, 2),  
 Conv2d\_1x1(in\_channel)  
 )  
 self.channel\_shuffle = Channel\_Shuffle(groups=2)  
  
 **def** forward(self, x):  
 shortcut = x  
 out = x  
 out = self.model(out)  
 shortcut = self.shortcut(shortcut)  
 out = torch.cat((shortcut, out), 1)  
 out = self.channel\_shuffle(out)  
 **return** out

1. **Group Convlution**

Group Convolution就是对输入feature map进行分组，每组分别卷积。假设输入feature map尺寸为C∗H∗W，输出feature map的数量（通道）为N，分成G个groups，则每组的输入feature map数量为C/G，每组的输出feature map数量为N/G，每个卷积核的尺寸为C/G∗K∗K，卷积核的总数仍为N个，每组卷积核数量为N/G，卷积核只与其同组的输入feature map进行卷积，卷积核的总参数量为N∗C/G∗K∗K，总参数量减少为原来的 1/G倍



作用和特点：

* 减少参数量，将特征图分成G组，参数量减少为原来的1/G
* Group Convolution可以看成是结构稀疏（structured sparse），每个卷积核的尺寸由C∗K∗K变为C/G∗K∗K，可以将其余(C−C/G)∗K∗K的参数视为0，有时甚至可以在减少参数量的同时获得更好的效果（相当于正则）。
* 当分组数量G等于输入map数量C等于输出map数量N，即G==C==N，N个卷积核每个尺寸为1∗K∗K时，Group Convolution就成了Depthwise Convolution，