1. ShuffleNet v1

An Extremely Efficient Convolutional Neural Network for Mobile Devices

提出3 channel shuffle 的思想.

ShuffleNet Unit 中全是 Group Conv 和 DW Conv

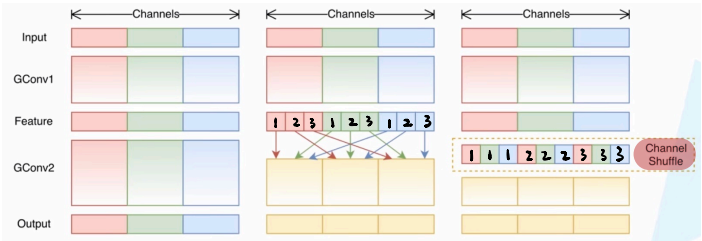
2. Performance

错误率 时间

Model	Cls err. (%)	FLOPs	224 × 224	480 × 640	720 × 1280
ShuffleNet 0.5× ($g = 3$)	43.2	38M	15.2ms	87.4ms	260.1ms
ShuffleNet 1× ($g = 3$)	32.6	140M	37.8ms	222.2ms	684.5ms
ShuffleNet 2× ($g = 3$)	26.3	524M	108.8ms	617.0ms	1857.6ms
AlexNet [21]	42.8	720M	184.0ms	1156.7ms	3633.9ms
1.0 MobileNet-224 [12]	29.4	569M	110.0ms	612.0ms	1879.2ms

错误率差不多 时间短 时间更大是 error 更小

3.



GConv 虽然能够减小参数与计算量

但 GConv 中不同组之间信息没有交流

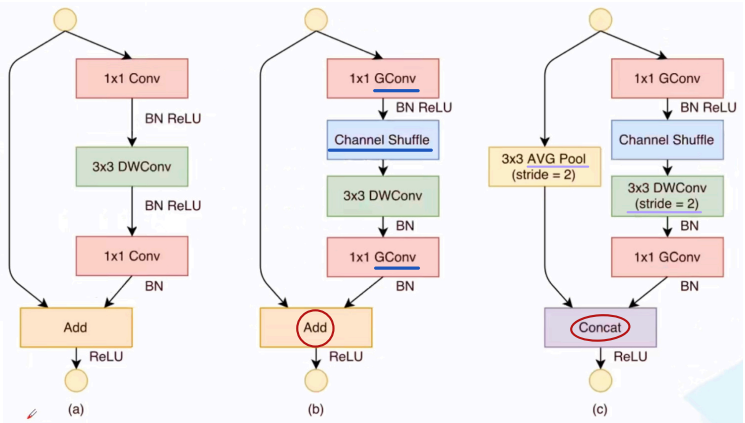
Channel Shuffle:

融合不同组信息.

4. ShuffleNet Unit

3.1. Channel Shuffle for Group Convolutions

Modern convolutional neural networks [30, 33, 34, 32, 9, 10] usually consist of repeated building blocks with the same structure. Among them, state-of-the-art networks such as *Xception* [3] and *ResNeXt* [40] introduce efficient depthwise separable convolutions or group convolutions into the building blocks to strike an excellent trade-off between representation capability and computational cost. However, we notice that both designs do not fully take the 1×1 convolutions (also called *pointwise convolutions* in [12]) into account, which require considerable complexity. For example, in ResNeXt [40] only 3×3 layers are equipped with group convolutions. As a result, for each residual unit in ResNeXt the pointwise convolutions occupy 93.4% multiplication-adds (cardinality = 32 as suggested in [40]). In tiny networks, expensive pointwise convolutions result in limited number of channels to meet the complexity constraint, which might significantly damage the accuracy.



之前在 ResNet, stride=2 时,

用 stride=2 Conv

这里使用 stride=2 AVG Pool

这里用 concat 拼接

94% 的计算都在 1×1 conv

↳ b) 1×1 换成 GConv

5.

Layer	Output size	KSize	Stride	Repeat	Output channels (g groups)				
					$g = 1$	$g = 2$	$g = 3$	$g = 4$	$g = 8$
Image	224 × 224				3	3	3	3	3
Conv1	112 × 112	3 × 3	2	1	24	24	24	24	24
MaxPool	56 × 56	3 × 3	2						
Stage2	28 × 28		2	1	144	200	240	272	384
	28 × 28		1	3	144	200	240	272	384
Stage3	14 × 14		2	1	288	400	480	544	768
	14 × 14		1	7	288	400	480	544	768
Stage4	7 × 7		2	1	576	800	960	1088	1536
	7 × 7		1	3	576	800	960	1088	1536
GlobalPool	1 × 1	7 × 7							
FC					1000	1000	1000	1000	1000
Complexity					143M	140M	137M	133M	137M

Built on ShuffleNet units, we present the overall ShuffleNet architecture in Table 1. The proposed network is mainly composed of a stack of ShuffleNet units grouped into three stages. The first building block in each stage is applied with stride = 2. Other hyper-parameters within a stage stay the same, and for the next stage the output channels are doubled. Similar to [9], we set the number of bottleneck channels to 1/4 of the output channels for each ShuffleNet

1) 每个 stage 的第 1 个 block, stride=2

2) 下个 stage # output_C 翻倍

3) [9]: ResNet

bottleneck_C = 1/4 # output_C

4) 第 1 个 24C 的 block 不使用 GConv

24 个 C 比较小

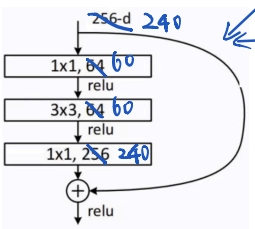


Table 1. ShuffleNet architecture. The complexity is evaluated with FLOPs, i.e. the number of floating-point multiplication-adds. Note that for Stage 2, we do not apply group convolution on the first pointwise layer because the number of input channels is relatively small.

6. FLOPs

ResNet: $hw(1\times1\times c\times m) + hw(3\times3\times m\times m) + hw(1\times1\times m\times c) = hw(2cm + 9m^2)$

ResNeXt: $hw(1\times1\times c\times m) + hw(3\times3\times m\times m)/g + hw(1\times1\times m\times c) = hw(2cm + 9m^2/g)$

ShuffleNet: $hw(1\times1\times c\times m)/g + hw(3\times3\times m) + hw(1\times1\times m\times c)/g = hw(2cm/g + 9m)$

<DW: m=g, 约去>

FLOPs: 全大写, 每秒浮点运算次数, 硬件指标

FLOPs: s 小写, 浮点运算数, 计算量, 衡量算法/模型复杂度 1 GFLOPs = 10^9 FLOPs

1. ShuffleNet v2 : Pratical Guidelines for Efficient CNN Architecture Design

- 计算复杂度不能只看 FLOPs
- 提出 4 条设计高效网络准则
- 提出新的 block 设计

2. Measurement:

FLOPs —— indirect metrics

speed —— direct metrics

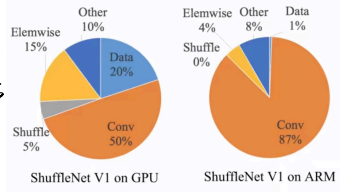
Factors:

Memory Access Cost (MAC) 内存访问时间成本

Degree of parallelism 并列等级

Platform <GPU vs 基于 ARM 的 CPU>

⇒ 不能只考虑 FLOPs



The discrepancy between the indirect (FLOPs) and direct (speed) metrics can be attributed to two main reasons. First, several important factors that have considerable affection on speed are not taken into account by FLOPs. One such factor is *memory access cost* (MAC). Such cost constitutes a large portion of runtime in certain operations like group convolution. It could be bottleneck on devices with strong computing power, e.g., GPUs. This cost should not be simply ignored during network architecture design. Another one is *degree of parallelism*. A model with high degree of parallelism could be much faster than another one with low degree of parallelism, under the same FLOPs.

Second, operations with the same FLOPs could have different running time, depending on the platform. For example, tensor decomposition is widely used in early works [20, 21, 22] to accelerate the matrix multiplication. However, the recent work [19] finds that the decomposition in [22] is even slower on GPU although it reduces FLOPs by 75%. We investigated this issue and found that this is because the latest CUDNN [23] library is specially optimized for 3×3 conv. We cannot certainly think that 3×3 conv is 9 times slower than 1×1 conv.

The overall runtime is decomposed for different operations, as shown in Figure 2. We note that the FLOPs metric only account for the convolution part. Although this part consumes most time, the other operations including data I/O, data shuffle and element-wise operations (AddTensor, ReLU, etc) also occupy considerable amount of time. Therefore, FLOPs is not an accurate enough estimation of actual runtime.

3. Several pratical guidelines for efficient network architecture design

3.1. Equal channel width minimizes memory access cost (MAC)

当卷积层的输入特征矩阵和输出特征矩阵保持不变时 MAC 最小 (保持 FLOPs 不变)

$$MAC \geq 2\sqrt{hwB} + \frac{B}{hw} \qquad B = hwc_1c_2 \ (FLOPs) \qquad \text{当 } c_1 = c_2 \text{ 取等号}$$

$$MAC = hw(c_1 + c_2) + c_1c_2 \quad ; \quad \frac{c_1 + c_2}{2} \geq \sqrt{c_1c_2}$$

$$MAC \geq 2hw\sqrt{c_1c_2} + c_1c_2 \geq 2\sqrt{hwB} + \frac{B}{hw} \qquad B = hwc_1c_2$$

		GPU (Batches/sec.)				ARM (Images/sec.)		
c1:c2	(c1,c2) for ×1	×1	×2	×4	(c1,c2) for ×1	×1	×2	×4
1:1	(128,128)	1480	723	232	(32,32)	76.2	21.7	5.3
1:2	(90,180)	1296	586	206	(22,44)	72.9	20.5	5.1
1:6	(52,312)	876	489	189	(13,78)	69.1	17.9	4.6
1:12	(36,432)	748	392	163	(9,108)	57.6	15.1	4.4

变慢 3　　变慢不明显

The conclusion is theoretical. In practice, the cache on many devices is not large enough. Also, modern computation libraries usually adopt complex blocking strategies to make full use of the cache mechanism [24]. Therefore, the real MAC may deviate from the theoretical one. To validate the above conclusion, an experiment is performed as follows. A benchmark network is built by stacking 10 building blocks repeatedly. Each block contains two convolution layers. The first contains c_1 input channels and c_2 output channels, and the second otherwise.

Table 1 reports the running speed by varying the ratio $c_1 : c_2$ while fixing the total FLOPs. It is clear that when $c_1 : c_2$ is approaching $1 : 1$, the MAC becomes smaller and the network evaluation speed is faster.

3.2. Excessively group convolution increase MAC

当 GConv 的 groups 增大时 (保持 FLOPs 一定), MAC 增大

$$MAC = hw(c_1 + c_2) + \frac{c_1c_2}{g} = hwc_1 + \frac{Bg}{c_1} + \frac{B}{hw} \qquad B = hwc_1c_2/g \ (FLOPs)$$

		GPU (Batches/sec.)				CPU (Images/sec.)		
g	c for ×1	×1	×2	×4	c for ×1	×1	×2	×4
1	128	2451	1289	437	64	40.0	10.2	2.3
2	180	1725	873	341	90	35.0	9.5	2.2
4	256	1026	644	338	128	32.9	8.7	2.1
8	360	634	445	230	180	27.8	7.5	1.8

Table 2: Validation experiment for **Guideline 2**. Four values of group number g are tested, while the total FLOPs under the four values is fixed by varying the total channel number c. Input image size is 56×56.

where g is the number of groups and $B = hwc_1c_2/g$ is the FLOPs. It is easy to see that, given the fixed input shape $c_1 \times h \times w$ and the computational cost B, MAC increases with the growth of g.

different group numbers while fixing the total FLOPs. It is clear that using a large group number decreases running speed significantly. For example, using 8 groups is more than two times slower than using 1 group (standard dense convolution) on GPU and up to 30% slower on ARM. This is mostly due to increased MAC. We note that our implementation has been specially optimized and is much faster than trivially computing convolutions group by group.

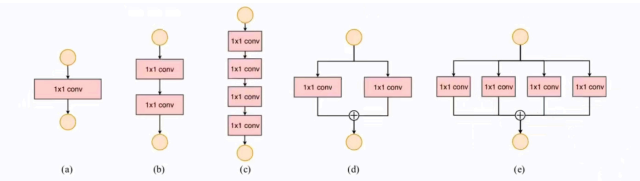
Therefore, we suggest that *the group number should be carefully chosen based on the target platform and task. It is unwise to use a large group number simply because this may enable using more channels, because the benefit of accuracy increase can easily be outweighed by the rapidly increasing computational cost.*

3.3. Network fragmentation reduces degree of parallelism

网络设计的碎片化程度越高, 速度越慢 ⇒ 分支: 串联 / 并联

Though such fragmented structure has been shown beneficial for accuracy, it could decrease efficiency because it is unfriendly for devices with strong parallel computing powers like GPU. It also introduces extra overheads such as kernel launching and synchronization.

尽管多分支可以很地地提高准确率, 但是对于像 GPU 这种非常强的并行运算能力设备, 碎片化的结构会降低效率. 并且还有 kernal 启动和同步的问题 (ie): 4 个 kernal, 如果运行时间相差大, 快的要等慢的结束后才能进行下一步)



Appendix Fig. 1: Building blocks used in experiments for guideline 3. (a) *1-fragment*. (b) *2-fragment-series*. (c) *4-fragment-series*. (d) *2-fragment-parallel*. (e) *4-fragment-parallel*.

	GPU (Batches/sec.)			CPU (Images/sec.)		
	c=128	c=256	c=512	c=64	c=128	c=256
1-fragment	2446	1274	434	40.2	10.1	2.3
2-fragment-series	1790	909	336	38.6	10.1	2.2
4-fragment-series	752	745	349	38.4	10.1	2.3
2-fragment-parallel	1537	803	320	33.4	9.1	2.2
4-fragment-parallel	691	572	292	35.0	8.4	2.1

反而变快了

3.4. Element-wise operations are non-negligible

Element-wise 操作带来的影响是不可忽视的

如果将 ReLU 和 short-cut 去除，大概能够快 20%

(eg. ReLU, AddTensor, AddBias, etc, small FLOPs but heavy MAC)

G4) Element-wise operations are non-negligible. As shown in Figure 2, in light-weight models like [15,14], element-wise operations occupy considerable amount of time, especially on GPU. Here, the element-wise operators include ReLU, AddTensor, AddBias, etc. They have small FLOPs but relatively heavy MAC. Specially, we also consider *depthwise convolution* [12,13,14,15] as an element-wise operator as it also has a high MAC/FLOPs ratio.

time of different variants is reported in Table 4. We observe around 20% speedup is obtained on both GPU and ARM, after ReLU and shortcut are removed.

		GPU (Batches/sec.)			CPU (Images/sec.)		
ReLU	short-cut	c=32	c=64	c=128	c=32	c=64	c=128
yes	yes	2427	2066	1436	56.7	16.9	5.0
yes	no	2647	2256	1735	61.9	18.8	5.2
no	yes	2672	2121	1458	57.3	18.2	5.1
no	no	2842	2376	1782	66.3	20.2	5.4

3.5 总结

Conclusion and Discussions Based on the above guidelines and empirical studies, we conclude that an efficient network architecture should 1) use "balanced" convolutions (equal channel width); 2) be aware of the cost of using group convolution; 3) reduce the degree of fragmentation; and 4) reduce element-wise operations. These desirable properties depend on platform characterics (such as memory manipulation and code optimization) that are beyond theoretical FLOPs. They should be taken into accout for practical network design.

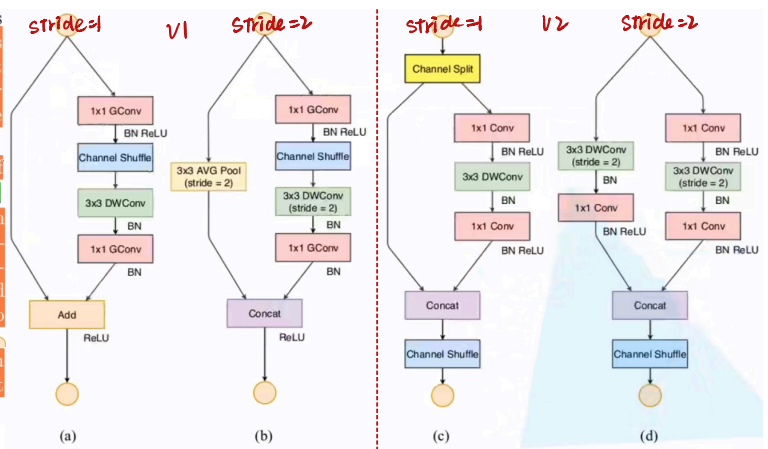
4. 根据准则设计 V2

beginning of each unit, the input of c feature channels are split into two branches with $c - c'$ and c' channels, respectively. Following **G3**, one branch remains as identity. The other branch consists of three convolutions with the same input and output channels to satisfy **G1**. The two 1×1 convolutions are no longer group-wise, unlike [15]. This is partially to follow **G2**, and partially because the split operation already produces two groups.

After convolution, the two branches are concatenated. So, the number of channels keeps the same (**G1**). The same "channel shuffle" operation as in [15].

After the shuffling, the next unit begins. Note that the "Add" operation in ShuffleNet v1 [15] no longer exists. Element-wise operations like ReLU and *depthwise convolutions* exist only in one branch. Also, the three successive element-wise operations, "Concat", "Channel Shuffle" and "Channel Split", are merged into a single element-wise operation. These changes are beneficial according to G4.

For spatial down sampling, the unit is slightly modified and illustrated in Figure 3(d). The channel split operator is removed. Thus, the number of output channels is doubled.



① channel split: c', $c-c'$ $<c'=\frac{c}{2}>$

② Following G3: 分支不做任何操作

<减少碎片化程度>

③ Following G1: input_c = output_c

④ Following G2: 2个 1×1 conv 不再使用 G conv

⑤ Following G1: 两个 branch concat, input c = output c

⑥ Following G4: 只对1个 branch ReLU, ReLU 移到3 Concat 之前

Concat, Channel Shuffle 和下一层 Channel Split 可以算做1个 Ele-wise operation

⑦ stride=2 时，没有 channel split，#output_c doubled

fleNet v1 [15] and summarized in Table 5. There is only one difference: an additional 1×1 convolution layer is added right before *global averaged pooling* to mix up features, which is absent in ShuffleNet v1. Similar to [15], the number of channels in each block is scaled to generate networks of different complexities, marked as 0.5×, 1×, etc.

Layer	Output size	KSize	Stride	Repeat	Output channels 0.5×	1×	1.5×	2×
Image	224×224				3	3	3	3
Conv1	112×112	3×3	2	1	24	24	24	24
MaxPool	56×56	3×3	2					
Stage2	28×28		2	1	48	116	176	244
	28×28		1	3				
Stage3	14×14		2	1	96	232	352	488
	14×14		1	7				
Stage4	7×7		2	1	192	464	704	976
	7×7		1	3				
Conv5	7×7	1×1	1	1	1024	1024	1024	2048
GlobalPool	1×1	7×7						
FC					1000	1000	1000	1000
FLOPs					41M	146M	299M	591M
# of Weights					1.4M	2.3M	3.5M	7.4M

V2 结构和 V1 类似，除了 Conv5

对于 stage2 的第1个 block，2个 branch 中 output_c ≠ input_c

而是直接设置为指定 output_c 的一半，

比如对于 1×，每分支 channel 应该是 58，不是 24