

ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design

Ningning Ma ^{*1,2} Xiangyu Zhang ^{*1} Hai-Tao Zheng² Jian Sun¹

¹ Megvii Inc (Face++)

² Tsinghua University

{maningning,zhangxiangyu,sunjian}@megvii.com

zheng.haitao@sz.tsinghua.edu.cn

Abstract. Currently, the neural network architecture design is mostly guided by the *indirect* metric of computation complexity, i.e., FLOPs. However, the *direct* metric, e.g., speed, also depends on the other factors such as memory access cost and platform characteristics. Thus, this work proposes to evaluate the direct metric on the target platform, beyond only considering FLOPs. Based on a series of controlled experiments, this work derives several practical *guidelines* for efficient network design. Accordingly, a new architecture is presented, called *ShuffleNet V2*. Comprehensive ablation experiments verify that our model is the state-of-the-art in terms of speed and accuracy tradeoff.

Keywords: CNN architecture design, efficiency, practical

1 Introduction

The architecture of deep convolutional neural networks (CNNs) has evolved for years, becoming more accurate and faster. Since the milestone work of AlexNet [1], the ImageNet classification accuracy has been significantly improved by novel structures, including VGG [2], GoogLeNet [3], ResNet [4,5], DenseNet [6], ResNeXt [7], SE-Net [8], and automatic neural architecture search [9,10,11], to name a few.

Besides accuracy, computation complexity is another important consideration. Real world tasks often aim at obtaining best accuracy under a limited computational budget, given by target platform (e.g., hardware) and application scenarios (e.g., auto driving requires low latency). This motivates a series of works towards light-weight architecture design and better speed-accuracy trade-off, including Xception [12], MobileNet [13], MobileNet V2 [14], ShuffleNet [15], and CondenseNet [16], to name a few. Group convolution and depth-wise convolution are crucial in these works.

To measure the computation complexity, a widely used metric is the number of float-point operations, or *FLOPs*¹. However, FLOPs is an *indirect* metric. It is an approximation of, but usually not equivalent to the *direct* metric that we really care about, such as speed or latency. Such discrepancy has been noticed

^{*} Equal contribution.

¹ In this paper, the definition of *FLOPs* follows [15], i.e. the number of multiply-adds.

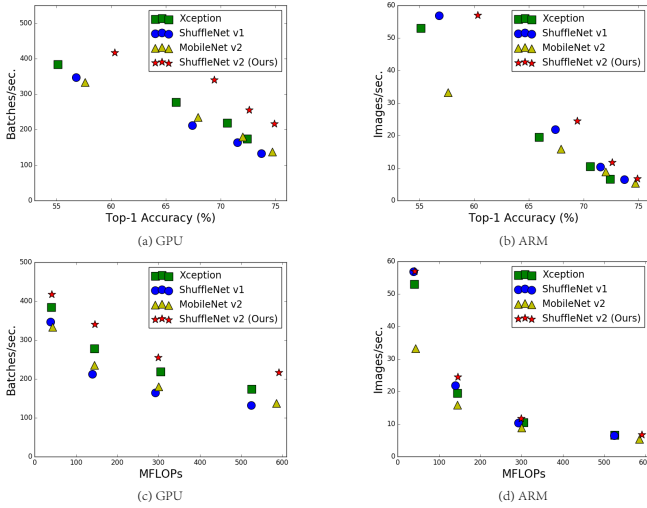


Fig. 1: Measurement of accuracy (ImageNet classification on validation set), speed and FLOPs of four network architectures on two hardware platforms with four different level of computation complexities (see text for details). (a, c) GPU results, $batchsize = 8$. (b, d) ARM results, $batchsize = 1$. The best performing algorithm, our proposed ShuffleNet v2, is on the top right region, under all cases.

in previous works [17,18,14,19]. For example, *MobileNet v2* [14] is much faster than *NASNET-A* [9] but they have comparable FLOPs. This phenomenon is further exemplified in Figure 1(c)(d), which show that networks with similar FLOPs have different speeds. Therefore, using FLOPs as the only metric for computation complexity is insufficient and could lead to sub-optimal design.

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.

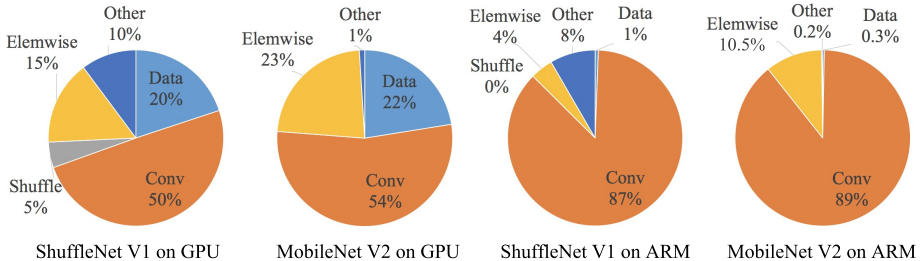


Fig. 2: Run time decomposition on two representative state-of-the-art network architectures, *ShuffleNet v1* [15] ($1\times$, $g = 3$) and *MobileNet v2* [14] ($1\times$).

With these observations, we propose that two principles should be considered for effective network architecture design. First, the direct metric (e.g., speed) should be used instead of the indirect ones (e.g., FLOPs). Second, such metric should be evaluated on the target platform.

In this work, we follow the two principles and propose a more effective network architecture. In Section 2, we firstly analyze the runtime performance of two representative state-of-the-art networks [15,14]. Then, we derive four guidelines for efficient network design, which are beyond only considering FLOPs. While these guidelines are platform independent, we perform a series of controlled experiments to validate them on two different platforms (GPU and ARM) with dedicated code optimization, ensuring that our conclusions are state-of-the-art.

In Section 3, according to the guidelines, we design a new network structure. As it is inspired by ShuffleNet [15], it is called *ShuffleNet V2*. It is demonstrated much faster and more accurate than the previous networks on both platforms, via comprehensive validation experiments in Section 4. Figure 1(a)(b) gives an overview of comparison. For example, given the computation complexity budget of 40M FLOPs, ShuffleNet v2 is 3.5% and 3.7% more accurate than ShuffleNet v1 and MobileNet v2, respectively.

2 Practical Guidelines for Efficient Network Design

Our study is performed on two widely adopted hardware with industry-level optimization of CNN library. We note that our CNN library is more efficient than most open source libraries. Thus, we ensure that our observations and conclusions are solid and of significance for practice in industry.

- *GPU*. A single NVIDIA GeForce GTX 1080Ti is used. The convolution library is CUDNN 7.0 [23]. We also activate the benchmarking function of CUDNN to select the fastest algorithms for different convolutions respectively.
- *ARM*. A Qualcomm Snapdragon 810. We use a highly-optimized Neon-based implementation. A single thread is used for evaluation.

		GPU (Batches/sec.)				ARM (Images/sec.)		
c1:c2	(c1,c2) for $\times 1$	$\times 1$	$\times 2$	$\times 4$	(c1,c2) for $\times 1$	$\times 1$	$\times 2$	$\times 4$
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

Table 1: Validation experiment for **Guideline 1**. Four different ratios of number of input/output channels (c1 and c2) are tested, while the total FLOPs under the four ratios is fixed by varying the number of channels. Input image size is 56×56 .

Other settings include: full optimization options (e.g. tensor fusion, which is used to reduce the overhead of small operations) are switched on. The input image size is 224×224 . Each network is randomly initialized and evaluated for 100 times. The average runtime is used.

To initiate our study, we analyze the runtime performance of two state-of-the-art networks, *ShuffleNet v1* [15] and *MobileNet v2* [14]. They are both highly efficient and accurate on ImageNet classification task. They are both widely used on low end devices such as mobiles. Although we only analyze these two networks, we note that they are representative for the current trend. At their core are group convolution and depth-wise convolution, which are also crucial components for other state-of-the-art networks, such as ResNeXt [7], Xception [12], MobileNet [13], and CondenseNet [16].

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.

Based on this observation, we perform a detailed analysis of runtime (or speed) from several different aspects and derive several practical guidelines for efficient network architecture design.

G1) Equal channel width minimizes memory access cost (MAC).

The modern networks usually adopt *depthwise separable convolutions* [12,13,15,14], where the pointwise convolution (i.e., 1×1 convolution) accounts for most of the complexity [15]. We study the kernel shape of the 1×1 convolution. The shape is specified by two parameters: the number of input channels c_1 and output channels c_2 . Let h and w be the spatial size of the feature map, the FLOPs of the 1×1 convolution is $B = hwc_1c_2$.

For simplicity, we assume the cache in the computing device is large enough to store the entire feature maps and parameters. Thus, the memory access cost (MAC), or the number of memory access operations, is $MAC = hw(c_1 + c_2) + c_1c_2$. Note that the two terms correspond to the memory access for input/output feature maps and kernel weights, respectively.

From mean value inequality, we have

		GPU (Batches/sec.)					CPU (Images/sec.)		
g	c for $\times 1$	$\times 1$	$\times 2$	$\times 4$	c for $\times 1$	$\times 1$	$\times 2$	$\times 4$	
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 .

$$\text{MAC} \geq 2\sqrt{hwB} + \frac{B}{hw}. \quad (1)$$

Therefore, *MAC has a lower bound given by FLOPs. It reaches the lower bound when the numbers of input and output channels are equal.*

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.

G2) Excessive group convolution increases MAC. Group convolution is at the core of modern network architectures [7,15,25,26,27,28]. It reduces the computational complexity (FLOPs) by changing the dense convolution between all channels to be sparse (only within groups of channels). On one hand, it allows usage of more channels given a fixed FLOPs and increases the network capacity (thus better accuracy). On the other hand, however, the increased number of channels results in more MAC.

Formally, following the notations in **G1** and Eq. 1, the relation between MAC and FLOPs for 1×1 group convolution is

$$\begin{aligned} \text{MAC} &= hw(c_1 + c_2) + \frac{c_1 c_2}{g} \\ &= hwc_1 + \frac{Bg}{c_1} + \frac{B}{hw}, \end{aligned} \quad (2)$$

where g is the number of groups and $B = hwc_1 c_2 / g$ 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 .

To study the affection in practice, a benchmark network is built by stacking 10 pointwise group convolution layers. Table 2 reports the running speed of using

	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

Table 3: Validation experiment for **Guideline 3**. c denotes the number of channels for *1-fragment*. The channel number in other fragmented structures is adjusted so that the FLOPs is the same as *1-fragment*. Input image size is 56×56 .

		GPU (Batches/sec.)			CPU (Images/sec.)		
		c=32	c=64	c=128	c=32	c=64	c=128
ReLU	short-cut						
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

Table 4: Validation experiment for **Guideline 4**. The ReLU and shortcut operations are removed from the “bottleneck” unit [4], separately. c is the number of channels in unit. The unit is stacked repeatedly for 10 times to benchmark the speed.

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.*

G3) Network fragmentation reduces degree of parallelism. In the GoogLeNet series [29,30,3,31] and auto-generated architectures [9,11,10]), a “multi-path” structure is widely adopted in each network block. A lot of small operators (called “fragmented operators” here) are used instead of a few large ones. For example, in *NASNET-A* [9] the number of fragmented operators (i.e. the number of individual convolution or pooling operations in one building block) is 13. In contrast, in regular structures like ResNet [4], this number is 2 or 3.

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.

To quantify how network fragmentation affects efficiency, we evaluate a series of network blocks with different degrees of fragmentation. Specifically, each

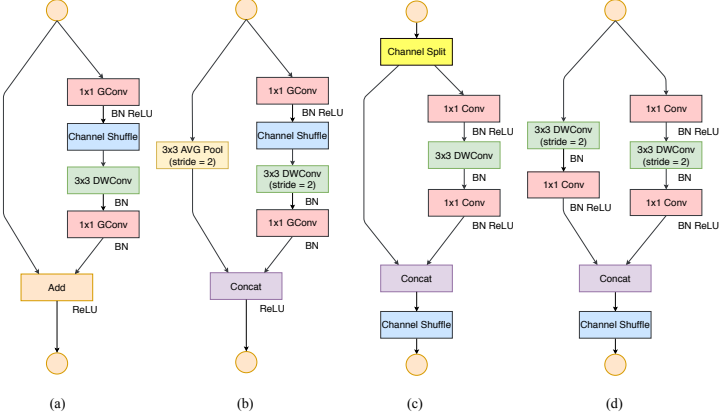


Fig. 3: Building blocks of ShuffleNet v1 [15] and this work. (a): the basic ShuffleNet unit; (b) the ShuffleNet unit for spatial down sampling ($2\times$); (c) our basic unit; (d) our unit for spatial down sampling ($2\times$). **DWConv**: depthwise convolution. **GConv**: group convolution.

building block consists of from 1 to 4 1×1 convolutions, which are arranged in sequence or in parallel. The block structures are illustrated in appendix. Each block is repeatedly stacked for 10 times. Results in Table 3 show that fragmentation reduces the speed significantly on GPU, e.g. 4-fragment structure is $3\times$ slower than 1-fragment. On ARM, the speed reduction is relatively small.

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.

For validation, we experimented with the “bottleneck” unit (1×1 conv followed by 3×3 conv followed by 1×1 conv, with ReLU and shortcut connection) in ResNet [4]. The ReLU and shortcut operations are removed, separately. Runtime 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.

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 characteristics (such as memory manipulation and code optimization) that are beyond theoretical FLOPs. They should be taken into account for practical network design.

Recent advances in light-weight neural network architectures [15,13,14,9,11,10,12] are mostly based on the metric of FLOPs and do not consider these properties above. For example, *ShuffleNet v1* [15] heavily depends group convolutions

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

Table 5: Overall architecture of ShuffleNet v2, for four different levels of complexities.

(against **G2**) and bottleneck-like building blocks (against **G1**). *MobileNet v2* [14] uses an inverted bottleneck structure that violates **G1**. It uses depthwise convolutions and ReLUs on “thick” feature maps. This violates **G4**. The auto-generated structures [9,11,10] are highly fragmented and violate **G3**.

3 ShuffleNet V2: an Efficient Architecture

Review of ShuffleNet v1 [15]. ShuffleNet is a state-of-the-art network architecture. It is widely adopted in low end devices such as mobiles. It inspires our work. Thus, it is reviewed and analyzed at first.

According to [15], the main challenge for light-weight networks is that only a limited number of feature channels is affordable under a given computation budget (FLOPs). To increase the number of channels without significantly increasing FLOPs, two techniques are adopted in [15]: pointwise group convolutions and bottleneck-like structures. A “channel shuffle” operation is then introduced to enable information communication between different groups of channels and improve accuracy. The building blocks are illustrated in Figure 3(a)(b).

As discussed in Section 2, both pointwise group convolutions and bottleneck structures increase MAC (**G1** and **G2**). This cost is non-negligible, especially for light-weight models. Also, using too many groups violates **G3**. The element-wise “Add” operation in the shortcut connection is also undesirable (**G4**). Therefore, in order to achieve high model capacity and efficiency, the key issue is how to maintain a large number and equally wide channels with neither dense convolution nor too many groups.

Channel Split and ShuffleNet V2 Towards above purpose, we introduce a simple operator called *channel split*. It is illustrated in Figure 3(c). At the

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] is then used to enable information communication between the two branches.

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 depth-wise 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**.

stride=2
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.

The proposed building blocks (c)(d), as well as the resulting networks, are called *ShuffleNet V2*. Based the above analysis, we conclude that this architecture design is highly efficient as it follows all the guidelines.

The building blocks are repeatedly stacked to construct the whole network. For simplicity, we set $c' = c/2$. The overall network structure is similar to ShuffleNet 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 $0.5\times$, $1\times$, etc.

Analysis of Network Accuracy ShuffleNet v2 is not only efficient, but also accurate. There are two main reasons. First, the high efficiency in each building block enables using more feature channels and larger network capacity.

Second, in each block, half of feature channels (when $c' = c/2$) directly go through the block and join the next block. This can be regarded as a kind of *feature reuse*, in a similar spirit as in *DenseNet* [6] and *CondenseNet* [16].

In *DenseNet*[6], to analyze the feature reuse pattern, the l_1 -norm of the weights between layers are plotted, as in Figure 4(a). It is clear that the connections between the adjacent layers are stronger than the others. This implies that the dense connection between all layers could introduce redundancy. The recent *CondenseNet* [16] also supports the viewpoint.

In ShuffleNet V2, it is easy to prove that the number of “directly-connected” channels between i -th and $(i+j)$ -th building block is $r^j c$, where $r = (1 - c')/c$. In other words, the amount of feature reuse decays exponentially with the distance between two blocks. Between distant blocks, the feature reuse becomes much weaker. Figure 4(b) plots the similar visualization as in (a), for $r = 0.5$. Note that the pattern in (b) is similar to (a).

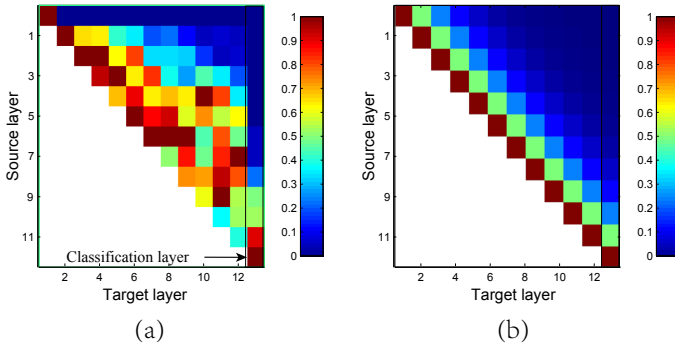


Fig. 4: Illustration of the patterns in feature reuse for *DenseNet* [6] and *ShuffleNet V2*. (a) (courtesy of [6]) the average absolute filter weight of convolutional layers in a model. The color of pixel (s, l) encodes the average $l1$ -norm of weights connecting layer s to l . (b) The color of pixel (s, l) means the number of channels *directly* connecting block s to block l in *ShuffleNet v2*. All pixel values are normalized to $[0, 1]$.

Thus, the structure of *ShuffleNet V2* realizes this type of feature re-use pattern *by design*. It shares the similar benefit of feature re-use for high accuracy as in *DenseNet* [6], but it is much more efficient as analyzed earlier. This is verified in experiments, Table 8.

4 Experiment

Our ablation experiments are performed on ImageNet 2012 classification dataset [32,33]. Following the common practice [15,13,14], all networks in comparison have four levels of computational complexity, i.e. about 40, 140, 300 and 500+ MFLOPs. Such complexity is typical for mobile scenarios. Other hyper-parameters and protocols are exactly the same as *ShuffleNet v1* [15].

We compare with following network architectures [12,14,6,15]:

- *ShuffleNet v1* [15]. In [15], a series of group numbers g is compared. It is suggested that the $g = 3$ has better trade-off between accuracy and speed. This also agrees with our observation. In this work we mainly use $g = 3$.
- *MobileNet v2* [14]. It is better than *MobileNet v1* [13]. For comprehensive comparison, we report accuracy in both original paper [14] and our reimplementation, as some results in [14] are not available.
- *Xception* [12]. The original *Xception* model [12] is very large (FLOPs >2G), which is out of our range of comparison. The recent work [34] proposes a modified light weight *Xception* structure that shows better trade-offs between accuracy and efficiency. So, we compare with this variant.
- *DenseNet* [6]. The original work [6] only reports results of large models (FLOPs >2G). For direct comparison, we reimplement it following the architecture settings in Table 5, where the building blocks in Stage 2-4 consist

of *DenseNet* blocks. We adjust the number of channels to meet different target complexities.

Table 8 summarizes all the results. We analyze these results from different aspects.

Accuracy vs. FLOPs. It is clear that the proposed ShuffleNet v2 models outperform all other networks by a large margin², especially under smaller computational budgets. Also, we note that MobileNet v2 performs poorly at 40 MFLOPs level with 224×224 image size. This is probably caused by too few channels. In contrast, our model do not suffer from this drawback as our efficient design allows using more channels. Also, while both of our model and DenseNet [6] reuse features, our model is much more efficient, as discussed in Sec. 3.

Table 8 also compares our model with other state-of-the-art networks including *CondenseNet* [16], *IGCV2* [27], and *IGCV3* [28] where appropriate. Our model performs better consistently at various complexity levels.

Inference Speed vs. FLOPs/Accuracy. For four architectures with good accuracy, ShuffleNet v2, MobileNet v2, ShuffleNet v1 and Xception, we compare their actual speed vs. FLOPs, as shown in Figure 1(c)(d). More results on different resolutions are provided in Appendix Table 1.

ShuffleNet v2 is clearly faster than the other three networks, especially on GPU. For example, at 500MFLOPs ShuffleNet v2 is 58% faster than MobileNet v2, 63% faster than ShuffleNet v1 and 25% faster than Xception. On ARM, the speeds of ShuffleNet v1, Xception and ShuffleNet v2 are comparable; however, MobileNet v2 is much slower, especially on smaller FLOPs. We believe this is because MobileNet v2 has higher MAC (see **G1** and **G4** in Sec. 2), which is significant on mobile devices.

Compared with *MobileNet v1* [13], *IGCV2* [27], and *IGCV3* [28], we have two observations. First, although the accuracy of MobileNet v1 is not as good, its speed on GPU is faster than all the counterparts, including ShuffleNet v2. We believe this is because its structure satisfies most of proposed guidelines (e.g. for **G3**, the fragments of MobileNet v1 are even fewer than ShuffleNet v2). Second, IGCv2 and IGCv3 are slow. This is due to usage of too many convolution groups (4 or 8 in [27,28]). Both observations are consistent with our proposed guidelines.

Recently, automatic model search [9,10,11,35,36,37] has become a promising trend for CNN architecture design. The bottom section in Table 8 evaluates some auto-generated models. We find that their speeds are relatively slow. We believe this is mainly due to the usage of too many fragments (see **G3**). Nevertheless, this research direction is still promising. Better models may be obtained, for example, if model search algorithms are combined with our proposed guidelines, and the direct metric (speed) is evaluated on the target platform.

² As reported in [14], MobileNet v2 of 500+ MFLOPs has comparable accuracy with the counterpart ShuffleNet v2 (25.3% vs. 25.1% top-1 error); however, our reimplemented version is not as good (26.7% error, see Table 8).

Finally, Figure 1(a)(b) summarizes the results of accuracy vs. speed, the direct metric. We conclude that ShuffleNet v2 is best on both GPU and ARM.

Compatibility with other methods. ShuffleNet v2 can be combined with other techniques to further advance the performance. When equipped with *Squeeze-and-excitation* (SE) module [8], the classification accuracy of ShuffleNet v2 is improved by 0.5% at the cost of certain loss in speed. The block structure is illustrated in Appendix Figure 2(b). Results are shown in Table 8 (bottom section).

Generalization to Large Models. Although our main ablation is performed for light weight scenarios, ShuffleNet v2 can be used for large models (e.g, FLOPs $\geq 2\text{G}$). Table 6 compares a 50-layer ShuffleNet v2 (details in Appendix) with the counterpart of ShuffleNet v1 [15] and *ResNet-50* [4]. ShuffleNet v2 still outperforms ShuffleNet v1 at 2.3GFLOPs and surpasses ResNet-50 with 40% fewer FLOPs.

For very deep ShuffleNet v2 (e.g. over 100 layers), for the training to converge faster, we slightly modify the basic ShuffleNet v2 unit by adding a residual path (details in Appendix). Table 6 presents a ShuffleNet v2 model of 164 layers equipped with *SE* [8] components (details in Appendix). It obtains superior accuracy over the previous state-of-the-art models [8] with much fewer FLOPs.

Object Detection To evaluate the generalization ability, we also tested COCO object detection [38] task. We use the state-of-the-art light-weight detector – *Light-Head RCNN* [34] – as our framework and follow the same training and test protocols. Only backbone networks are replaced with ours. Models are pretrained on ImageNet and then finetuned on detection task. For training we use *train+val* set in COCO except for 5000 images from *minival* set, and use the *minival* set to test. The accuracy metric is COCO standard *mmAP*, i.e. the averaged mAPs at the box IoU thresholds from 0.5 to 0.95.

ShuffleNet v2 is compared with other three light-weight models: *Xception* [12,34], *ShuffleNet v1* [15] and *MobileNet v2* [14] on four levels of complexities. Results in Table 7 show that ShuffleNet v2 performs the best.

Compared the detection result (Table 7) with classification result (Table 8), it is interesting that, on classification the accuracy rank is ShuffleNet v2 \geq MobileNet v2 $>$ ShuffleNet v1 $>$ Xception, while on detection the rank becomes ShuffleNet v2 $>$ Xception \geq ShuffleNet v1 \geq MobileNet v2. This reveals that Xception is good on detection task. This is probably due to the larger receptive field of Xception building blocks than the other counterparts (7 *vs.* 3). Inspired by this, we also enlarge the receptive field of ShuffleNet v2 by introducing an additional 3×3 depthwise convolution before the first pointwise convolution in each building block. This variant is denoted as *ShuffleNet v2**. With only a few additional FLOPs, it further improves accuracy.

We also benchmark the runtime time on GPU. For fair comparison the batch size is set to 4 to ensure full GPU utilization. Due to the overheads of data copying (the resolution is as high as 800×1200) and other detection-specific

Model	FLOPs	Top-1 err. (%)
ShuffleNet v2-50 (ours)	2.3G	22.8
ShuffleNet v1-50 [15] (our impl.)	2.3G	25.2
ResNet-50 [4]	3.8G	24.0
SE-ShuffleNet v2-164 (ours, with residual)	12.7G	18.56
SENet [8]	20.7G	18.68

Table 6: Results of large models. See text for details.

operations (like *PSRoI Pooling* [34]), the speed gap between different models is smaller than that of classification. Still, ShuffleNet v2 outperforms others, e.g. around 40% faster than ShuffleNet v1 and 16% faster than MobileNet v2.

Furthermore, the variant ShuffleNet v2* has best accuracy and is still faster than other methods. This motivates a practical question: how to increase the size of receptive field? This is critical for object detection in high-resolution images [39]. We will study the topic in the future.

Model	mmAP(%)				GPU Speed (Images/sec.)			
FLOPs	40M	140M	300M	500M	40M	140M	300M	500M
Xception	21.9	29.0	31.3	32.9	178	131	101	83
ShuffleNet v1	20.9	27.0	29.9	32.9	152	85	76	60
MobileNet v2	20.7	24.4	30.0	30.6	146	111	94	72
ShuffleNet v2 (ours)	22.5	29.0	31.8	33.3	188	146	109	87
ShuffleNet v2* (ours)	23.7	29.6	32.2	34.2	183	138	105	83

Table 7: Performance on COCO object detection. The input image size is 800×1200 . *FLOPs* row lists the complexity levels at 224×224 input size. For GPU speed evaluation, the batch size is 4. We do not test ARM because the *PSRoI Pooling* operation needed in [34] is unavailable on ARM currently.

5 Conclusion

We propose that network architecture design should consider the direct metric such as speed, instead of the indirect metric like FLOPs. We present practical guidelines and a novel architecture, ShuffleNet v2. Comprehensive experiments verify the effectiveness of our new model. We hope this work could inspire future work of network architecture design that is platform aware and more practical.

Acknowledgements Thanks Yichen Wei for his help on paper writing. This research is partially supported by National Natural Science Foundation of China (Grant No. 61773229).

Model	Complexity (MFLOPs)	Top-1 err. (%)	GPU Speed (Batches/sec.)	ARM Speed (Images/sec.)
ShuffleNet v2 0.5× (ours)	<u>41</u>	39.7	<u>417</u>	57.0
0.25 MobileNet v1 [13]	41	49.4	502	36.4
0.4 MobileNet v2 [14] (our impl.)*	43	43.4	333	33.2
0.15 MobileNet v2 [14] (our impl.)	39	55.1	351	33.6
ShuffleNet v1 0.5× (g=3) [15]	38	43.2	347	56.8
DenseNet 0.5× [6] (our impl.)	42	58.6	366	39.7
Xception 0.5× [12] (our impl.)	40	44.9	384	52.9
IGCV2-0.25 [27]	46	45.1	183	31.5
ShuffleNet v2 1× (ours)	<u>146</u>	30.6	<u>341</u>	24.4
0.5 MobileNet v1 [13]	149	36.3	382	16.5
0.75 MobileNet v2 [14] (our impl.)**	145	32.1	235	15.9
0.6 MobileNet v2 [14] (our impl.)	141	33.3	249	14.9
ShuffleNet v1 1× (g=3) [15]	140	32.6	213	21.8
DenseNet 1× [6] (our impl.)	142	45.2	279	15.8
Xception 1× [12] (our impl.)	145	34.1	278	19.5
IGCV2-0.5 [27]	156	34.5	132	15.5
IGCV3-D (0.7) [28]	210	31.5	143	11.7
ShuffleNet v2 1.5× (ours)	<u>299</u>	27.4	<u>255</u>	11.8
0.75 MobileNet v1 [13]	325	31.6	314	10.6
1.0 MobileNet v2 [14]	300	28.0	180	8.9
1.0 MobileNet v2 [14] (our impl.)	301	28.3	180	8.9
ShuffleNet v1 1.5× (g=3) [15]	292	28.5	164	10.3
DenseNet 1.5× [6] (our impl.)	295	39.9	274	9.7
CondenseNet (G=C=8) [16]	274	29.0	-	-
Xception 1.5× [12] (our impl.)	305	29.4	219	10.5
IGCV3-D [28]	318	27.8	102	6.3
ShuffleNet v2 2× (ours)	<u>591</u>	25.1	<u>217</u>	6.7
1.0 MobileNet v1 [13]	569	29.4	247	6.5
1.4 MobileNet v2 [14]	585	25.3	137	5.4
1.4 MobileNet v2 [14] (our impl.)	587	26.7	137	5.4
ShuffleNet v1 2× (g=3) [15]	524	26.3	133	6.4
DenseNet 2× [6] (our impl.)	519	34.6	197	6.1
CondenseNet (G=C=4) [16]	529	26.2	-	-
Xception 2× [12] (our impl.)	525	27.6	174	6.7
IGCV2-1.0 [27]	564	29.3	81	4.9
IGCV3-D (1.4) [28]	610	25.5	82	4.5
ShuffleNet v2 2x (ours, with SE [8])	<u>597</u>	24.6	<u>161</u>	5.6
NASNet-A [9] (4 @ 1056, our impl.)	564	26.0	130	4.6
PNASNet-5 [10] (our impl.)	588	25.8	115	4.1

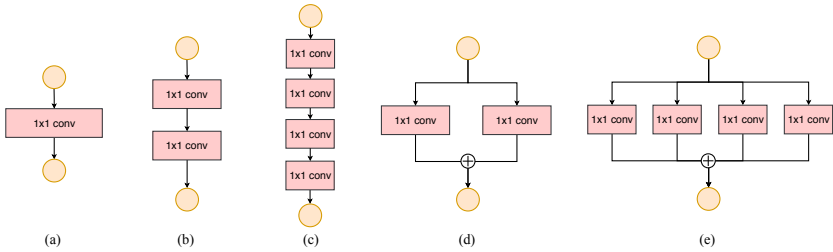
Table 8: Comparison of several network architectures over classification error (on validation set, single center crop) and speed, on two platforms and four levels of computation complexity. Results are grouped by complexity levels for better comparison. The batch size is 8 for GPU and 1 for ARM. The image size is 224×224 except: [*] 160×160 and [**] 192×192 . We do not provide speed measurements for *CondenseNets* [16] due to lack of efficient implementation currently.

References

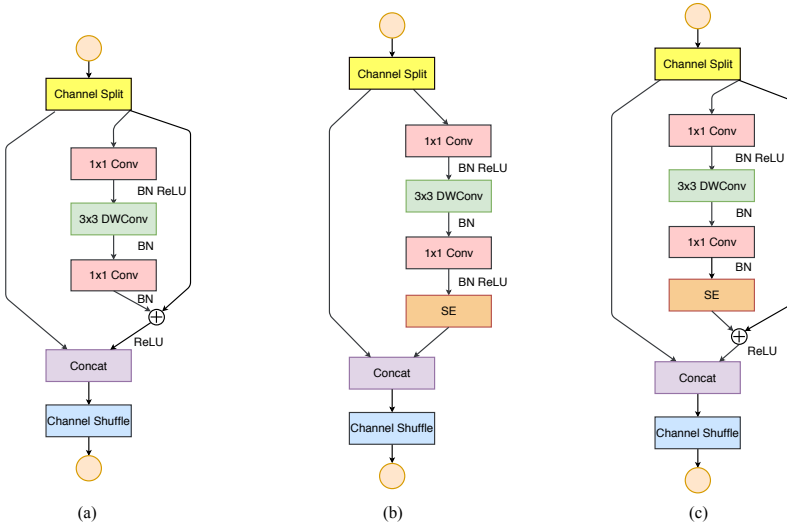
1. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. (2012) 1097–1105
2. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
3. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., et al.: Going deeper with convolutions, Cvpr (2015)
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2016) 770–778
5. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: European Conference on Computer Vision, Springer (2016) 630–645
6. Huang, G., Liu, Z., Weinberger, K.Q., van der Maaten, L.: Densely connected convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. Volume 1. (2017) 3
7. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. In: Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on, IEEE (2017) 5987–5995
8. Hu, J., Shen, L., Sun, G.: Squeeze-and-excitation networks. arXiv preprint arXiv:1709.01507 (2017)
9. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. arXiv preprint arXiv:1707.07012 (2017)
10. Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.J., Fei-Fei, L., Yuille, A., Huang, J., Murphy, K.: Progressive neural architecture search. arXiv preprint arXiv:1712.00559 (2017)
11. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. arXiv preprint arXiv:1802.01548 (2018)
12. Chollet, F.: Xception: Deep learning with depthwise separable convolutions. arXiv preprint (2016)
13. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
14. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. arXiv preprint arXiv:1801.04381 (2018)
15. Zhang, X., Zhou, X., Lin, M., Sun, J.: Shufflenet: An extremely efficient convolutional neural network for mobile devices. arXiv preprint arXiv:1707.01083 (2017)
16. Huang, G., Liu, S., van der Maaten, L., Weinberger, K.Q.: Condensenet: An efficient condensenet using learned group convolutions. arXiv preprint arXiv:1711.09224 (2017)
17. Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., Zhang, C.: Learning efficient convolutional networks through network slimming. In: 2017 IEEE International Conference on Computer Vision (ICCV), IEEE (2017) 2755–2763
18. Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H.: Learning structured sparsity in deep neural networks. In: Advances in Neural Information Processing Systems. (2016) 2074–2082
19. He, Y., Zhang, X., Sun, J.: Channel pruning for accelerating very deep neural networks. In: International Conference on Computer Vision (ICCV). Volume 2. (2017) 6

20. Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. arXiv preprint arXiv:1405.3866 (2014)
21. Zhang, X., Zou, J., Ming, X., He, K., Sun, J.: Efficient and accurate approximations of nonlinear convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2015) 1984–1992
22. Zhang, X., Zou, J., He, K., Sun, J.: Accelerating very deep convolutional networks for classification and detection. IEEE transactions on pattern analysis and machine intelligence **38**(10) (2016) 1943–1955
23. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
24. Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., Kaul, B., Dubey, P.: Distributed deep learning using synchronous stochastic gradient descent. arXiv preprint arXiv:1602.06709 (2016)
25. Ioannou, Y., Robertson, D., Cipolla, R., Criminisi, A.: Deep roots: Improving cnn efficiency with hierarchical filter groups. arXiv preprint arXiv:1605.06489 (2016)
26. Zhang, T., Qi, G.J., Xiao, B., Wang, J.: Interleaved group convolutions for deep neural networks. In: International Conference on Computer Vision. (2017)
27. Xie, G., Wang, J., Zhang, T., Lai, J., Hong, R., Qi, G.J.: Igcv 2: Interleaved structured sparse convolutional neural networks. arXiv preprint arXiv:1804.06202 (2018)
28. Sun, K., Li, M., Liu, D., Wang, J.: Igcv3: Interleaved low-rank group convolutions for efficient deep neural networks. arXiv preprint arXiv:1806.00178 (2018)
29. Szegedy, C., Ioffe, S., Vanhoucke, V., Alemi, A.A.: Inception-v4, inception-resnet and the impact of residual connections on learning. In: AAAI. Volume 4. (2017) 12
30. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2016) 2818–2826
31. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: International conference on machine learning. (2015) 448–456
32. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, IEEE (2009) 248–255
33. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al.: Imagenet large scale visual recognition challenge. International Journal of Computer Vision **115**(3) (2015) 211–252
34. Li, Z., Peng, C., Yu, G., Zhang, X., Deng, Y., Sun, J.: Light-head r-cnn: In defense of two-stage object detector. arXiv preprint arXiv:1711.07264 (2017)
35. Xie, L., Yuille, A.: Genetic cnn. arXiv preprint arXiv:1703.01513 (2017)
36. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Le, Q., Kurakin, A.: Large-scale evolution of image classifiers. arXiv preprint arXiv:1703.01041 (2017)
37. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)
38. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft coco: Common objects in context. In: European conference on computer vision, Springer (2014) 740–755
39. Peng, C., Zhang, X., Yu, G., Luo, G., Sun, J.: Large kernel matters—improve semantic segmentation by global convolutional network. arXiv preprint arXiv:1703.02719 (2017)

Appendix



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*.



Appendix Fig. 2: Building blocks of ShuffleNet v2 with SE/residual. (a) *ShuffleNet v2 with residual*. (b) *ShuffleNet v2 with SE*. (c) *ShuffleNet v2 with SE and residual*.

Input size	FLOPs	GPU (Batches/sec.)				CPU (Images/sec.)			
		40M	140M	300M	500M	40M	140M	300M	500M
320x320	ShuffleNet v2	315*	525	474	422	28.1	12.5	6.1	3.4
	ShuffleNet v1	236*	414	344	275	27.2	11.4	5.1	3.1
	MobileNet v2	187*	460	389	335	11.4	6.4	4.6	2.7
	Xception	279*	463	408	350	31.1	10.1	5.6	3.5
640x480	ShuffleNet v2	424	394	297	250	9.3	4.0	1.9	1.1
	ShuffleNet v1	396	269	198	156	8.0	3.7	1.6	1.0
	MobileNet v2	338	248	208	165	3.8	2.0	1.4	0.8
	Xception	399	326	244	209	9.6	3.2	1.7	1.1
1080x720	ShuffleNet v2	248	197	141	115	3.5	1.5	0.7	0.4
	ShuffleNet v1	203	131	96	77	2.9	1.4	0.4	0.3
	MobileNet v2	159	117	99	78	1.4	0.7	0.3	0.3
	Xception	232	160	124	106	3.6	1.2	0.5	0.4

(a) Comparison of actual speeds (whole architecture).

Input size	Channel (c) for ShuffleNet v2	GPU (Batches/sec.)			CPU (Images/sec.)		
		c=64	c=128	c=256	c=64	c=128	c=256
56x56	ShuffleNet v2	216	142	81	34.8	12.3	3.9
	ShuffleNet v1	127	73	45	24.3	9.4	3.0
	MobileNet v2	89	125	69	25.8	10.0	3.0
	Xception	185	52	68	27.0	9.7	3.1
28x28	ShuffleNet v2	407	313	237	174.5	53.4	16.6
	ShuffleNet v1	298	222	60	139.7	43.9	13.2
	MobileNet v2	381	286	189	118.3	46.2	13.3
	Xception	254	238	169	117.0	45.8	14.0

(b) Comparison of actual speeds (units).

Appendix Table 1: Table (a) compares the speed of each network (whole architecture). Table (b) compares the speed of each network’s unit, we stack 10 network units of each network; the value of c means the number of channels for ShuffleNet v2, we adjust the number of channels to keep the FLOPs unchanged for other network units. Please refer to Section 4 for details. [*] For the models of 40M FLOPs with input size of 320×320 , the *batchsize* is set to 8 to ensure the GPU utilization rate, and we set *batchsize* = 1 otherwise.

layer	output size	ShuffleNet v1-50 (group=3)	ShuffleNet v2-50	Resnet-50	SE-ShuffleNet v2-164
conv1_x	112×112	3×3, 64, stride 2	3×3, 64, stride 2	7×7, 64, stride 2	3×3, 64, stride 2 3×3, 64 3×3, 128
conv2_x	56×56	3×3 max pool, stride 2			
		$\begin{bmatrix} 1\times 1, 360 \\ 3\times 3, 360 \\ 1\times 1, 360 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 244 \\ 3\times 3, 244 \\ 1\times 1, 244 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 64 \\ 3\times 3, 64 \\ 1\times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 340 \\ 3\times 3, 340 \\ 1\times 1, 340 \end{bmatrix} \times 10$
conv3_x	28×28	$\begin{bmatrix} 1\times 1, 720 \\ 3\times 3, 720 \\ 1\times 1, 720 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1, 488 \\ 3\times 3, 488 \\ 1\times 1, 488 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1, 128 \\ 3\times 3, 128 \\ 1\times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1, 680 \\ 3\times 3, 680 \\ 1\times 1, 680 \end{bmatrix} \times 10$
conv4_x	14×14	$\begin{bmatrix} 1\times 1, 1440 \\ 3\times 3, 1440 \\ 1\times 1, 1440 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1, 976 \\ 3\times 3, 976 \\ 1\times 1, 976 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1, 256 \\ 3\times 3, 256 \\ 1\times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1, 1360 \\ 3\times 3, 1360 \\ 1\times 1, 1360 \end{bmatrix} \times 23$
conv5_x	7×7	$\begin{bmatrix} 1\times 1, 2880 \\ 3\times 3, 2880 \\ 1\times 1, 2880 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 1952 \\ 3\times 3, 1952 \\ 1\times 1, 1952 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 512 \\ 3\times 3, 512 \\ 1\times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 2720 \\ 3\times 3, 2720 \\ 1\times 1, 2720 \end{bmatrix} \times 10$
conv6	7×7	-	1×1, 2048	-	1×1, 2048
	1×1	average pool, 1000-d fc, softmax			
FLOPs		2.3G	2.3G	3.8G	12.7G

Appendix Table 2: Architectures for large models. Building blocks are shown in brackets, with the convolution kernel shapes and the numbers of blocks stacked. Downsampling is performed by *conv3.1*, *conv4.1*, and *conv5.1* with a stride of 2. For ShuffleNet v1-50 and ResNet-50, the bottleneck ratio is set to 1:4. For SE-ShuffleNet v2-164, we add the SE modules right before the residual *add-ReLUs* (details in Appendix Figure 2); we set the neural numbers in SE modules to the 1/2 of the channel numbers in the corresponding building blocks. See Section 4 for details.