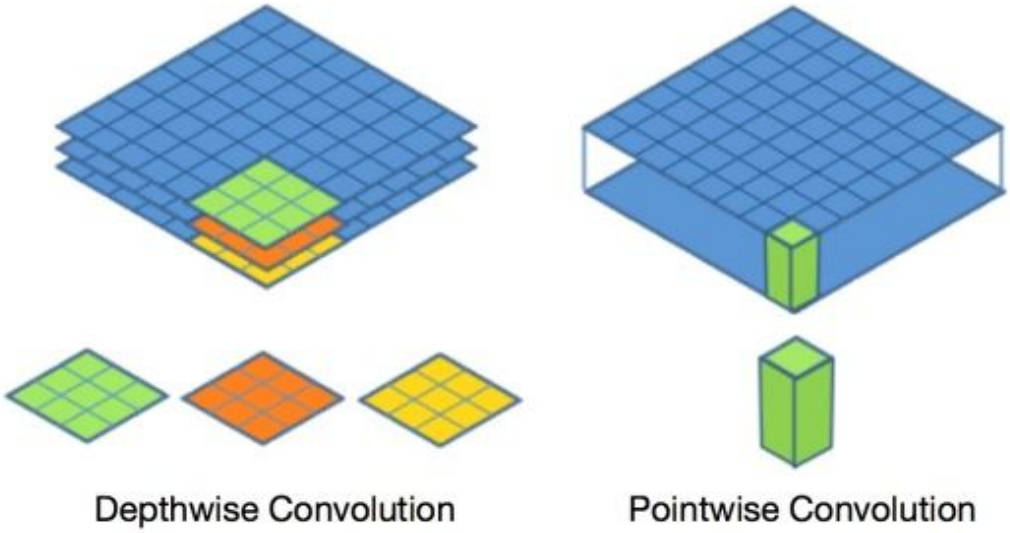
Evolution of MobileNets and ShuffleNets

MobileNets are the benchmarks in lightweight networks. They have relatively small sizes (fewer parameters) and comparable efficiencies, yet they don't have very complicated combinations across blocks.

Besides, we will also cover ShuffleNets, which bring many creative and valuable ideas to this field.

MobileNetV1

In 2017, Google proposed the first generation MobileNet: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. It introduces a new type of convolution named ***depthwise separable convolution***.



As we can see in the figure, its computational cost:

$$DepthwiseSeparableConv = DepthwiseConv + PointwiseConv$$
$$= (N * N * K * K * C_{in}) + (N * N * C_{in} * C_{out})$$

compared with standard convolutions':

$$StandardConv = N * N * K * K * C_{in} * C_{out}$$

and the ratio is:

$$Ratio = C_{out} + K * K$$

Normally $K = 3 \; or \; 5$, but C_{out} could be much bigger, a significant speedup in convolutions!

Enhanced with this, we can build rather efficient blocks and stack them together. In addition, two useful hyper-parameters were also introduced: *Width Multiplier* α and *Resolution Multiplier* ρ. The former mainly control the number of output channels, and the latter is to reduce input images' sizes (224, 192, 160...).

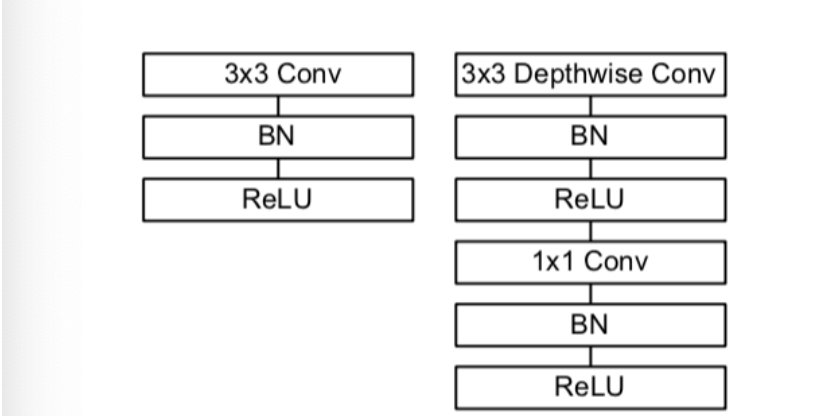


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

ShuffleNetV1

A few months later, Face++ introduced another efficient network ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices, which employs two novel operations, ***channel shuffle*** and ***pointwise group convolution***, and it outperforms MobileNetV1 on ImageNet classification task.

The metric issue must be emphasized. At that time, we were using indirect metrics ($MFLOPs$ or $MAdds$) to measure computational cost, instead of the direct metric $speed$. As you can imagine, some operations' time may be underestimated, like *Channel Shuffle* (extra overheads in memory copy).

Channel Shuffle for Group Convolutions

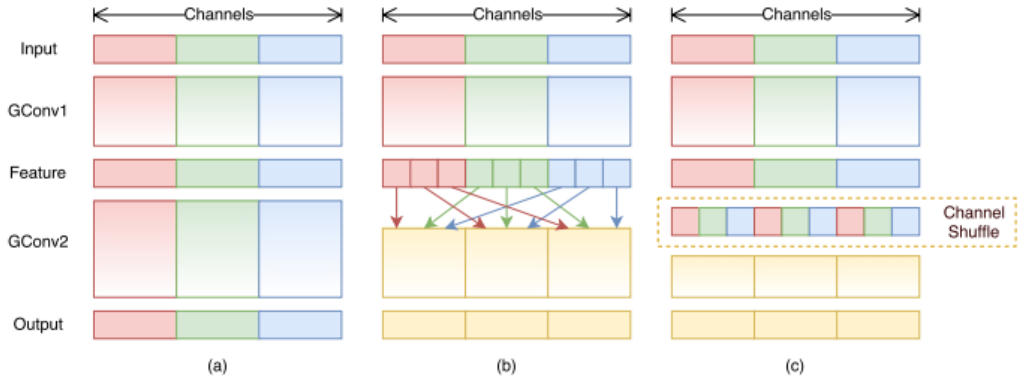


Figure 1. Channel shuffle with two stacked group convolutions. GConv stands for group convolution. a) two stacked convolution layers with the same number of groups. Each output channel only relates to the input channels within the group. No cross talk; b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; c) an equivalent implementation to b) using channel shuffle.

For *group convolution*, the authors split channels into different groups, then apply convolutions separately. *Channel shuffle* is the bridge across convolution groups, making it possible to utilize information from other groups (hmmm... just sounds reasonable).

ShuffleNetV1 Units

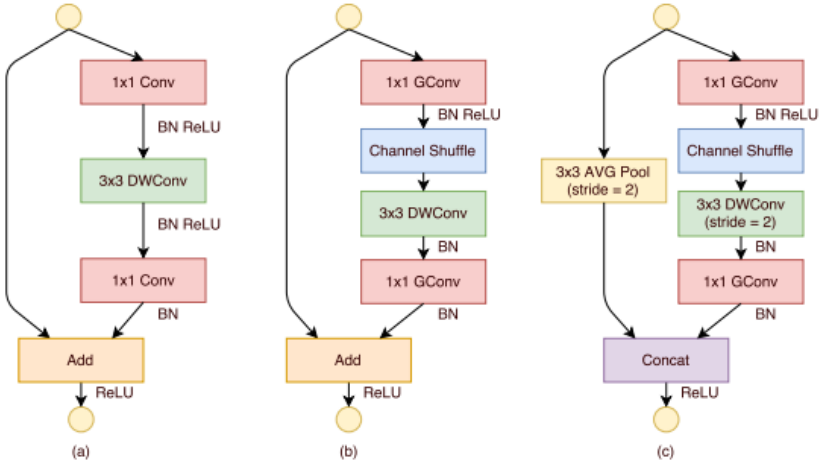


Figure 2. ShuffleNet Units. a) bottleneck unit [9] with depthwise convolution (DWConv) [3, 12]; b) ShuffleNet unit with pointwise group convolution (GConv) and channel shuffle; c) ShuffleNet unit with stride = 2.

There are two types of units for different $stride(=1\ or\ 2)$, and they are quite similar to **ResNet units**. We can stack them to be an efficient network.

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

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.

We should also note that *Channel Shuffle* is not a built-in operation in many frameworks, so you may need to combine *slice* and *gather* operations to mimic it, and be aware of its cost.

MobileNetV2

In January 2018, MobileNetV2: Inverted Residuals and Linear Bottlenecks was proposed. The key feature is applying **Inverted Residuals** where the shortcut connections are between the thin bottleneck layers. Within this structure, the intermediate expansion layer uses lightweight depthwise convolutions followed by non-linearity activations. Meanwhile, they found non-linearities should not be used in the narrow layers.

Non-linearities in narrow layers

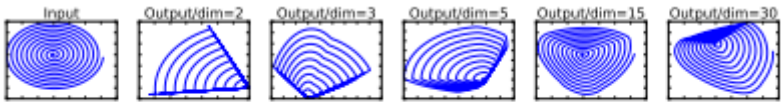


Figure 1: Examples of ReLU transformations of low-dimensional manifolds embedded in higher-dimensional spaces. In these examples the initial spiral is embedded into an n-dimensional space using random matrix T followed by ReLU, and then projected back to the 2D space using T^{-1}. In examples above $n = 2, 3$ result in information loss where certain points of the manifold collapse into each other, while for $n = 15$ to 30 the transformation is highly non-convex.

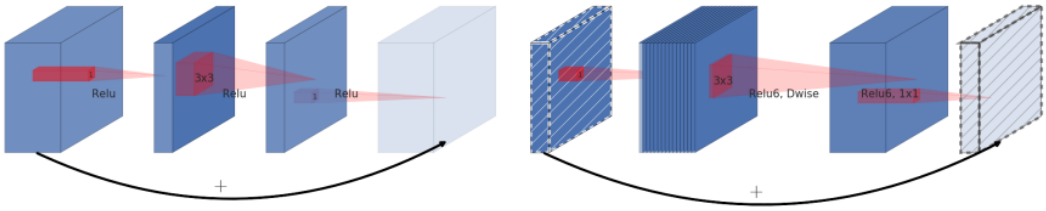
Why shouldn't we use non-linearities in narrow layers? Let's discuss *ReLU* to get some intuitions.

1. *ReLU* sets all negative parts to 0, which definitely lose some source information.
2. In wide layers, they should have redundant information from sources, so the removal of negative parts won't hurt them too much.
3. As the above figure shows, once a 2D spiral is projected and activated by *ReLU*, we can hardly reconstruct it from low-dimensional spaces.

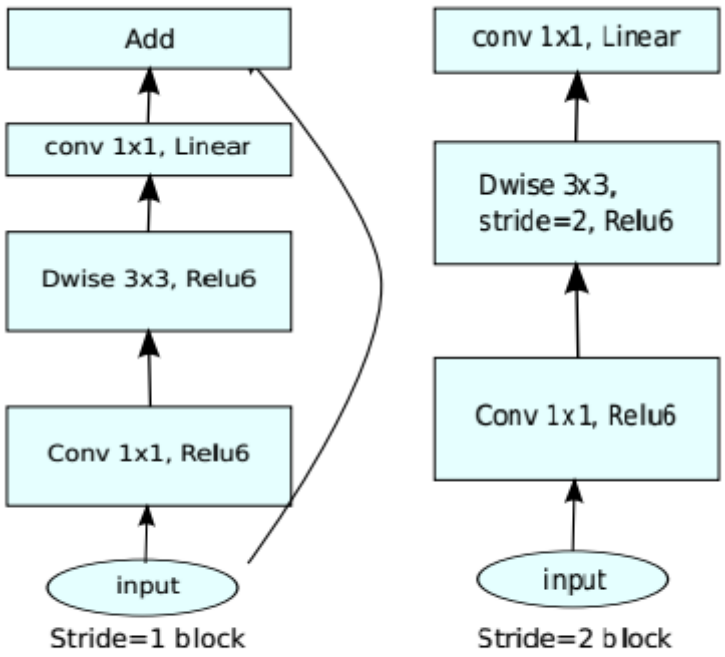
Inverted Residuals

The classical *residual blocks* use bottleneck layers to compress input features to compact features (may contain all the necessary information), while *inverted residuals* use expansion layers to produce higher-dimensional features (*no activations*, no hurt), then use *depthwide convolution (activation = ReLU)*, to prevent *ReLU* from destroying too much input's information. According to the paper, the expansion factor $t = 6$ except in the first bottleneck.

(a) Residual block (b) Inverted residual block



Still, the block structures:



(d) Mobilenet V2

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Table 2: MobileNetV2 : Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3×3 kernels. The expansion factor t is always applied to the input size as described in Table 1.

ShuffleNetV2

In the middle of 2018, Face++ and THU upgraded ShuffleNetV1 to ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. The paper is highly recommended, which derives several practical and detailed guidelines for efficient network architecture design and shows convincing experimental results.

Prior to ShuffleNetV2, most works only consider $FLOPs$ or $MAdds$. However, in engineering, ***memory access cost (MAC)*** and ***degree of parallelism*** also contribute to our final inference time. For example, in (c) and (d) networks with similar $FLOPs$ have different $speeds$.

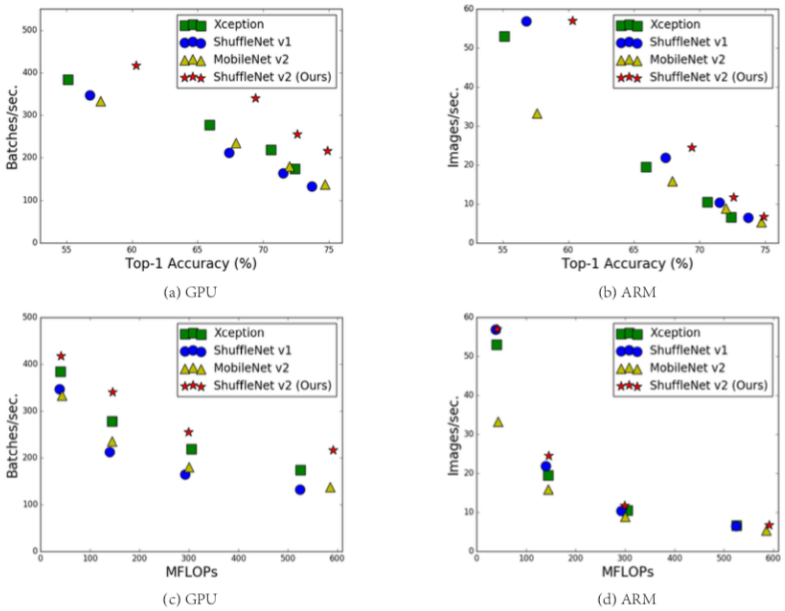


Fig. 1: Measurement of accuracy (ImageNet classification on validation set), speed and FLOPs of four network architectures on two hardware platforms with four different level of computation complexities (see text for details). (a, c) GPU results, $batchsize = 8$. (b, d) ARM results, $batchsize = 1$.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

Let's set up our parameters before discussing guidelines. c_1 = the number of input channels, c_2 = the number of input channels, h = the height of the feature map, w = the width of the feature map.

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

For simplicity, we just consider the 1×1 convolution. The computational cost ($FLOPs$):

$$B = hwc_1c_2$$

and memory access cost:

$$\begin{aligned} MAC &= MAC_{in} + MAC_{out} + MAC_{kernel} \\ &= hwc_1 + hwc_2 + c_1c_2 \end{aligned}$$

From *Inequality of arithmetic and geometric means*:

$$x + y \geq 2\sqrt{xy}$$

we have:

$$\begin{aligned} MAC &= hwc_1 + hwc_2 + c_1c_2 \\ &\geq hw * 2\sqrt{c_1c_2} + c_1c_2 \\ &= 2\sqrt{hwB} + \frac{B}{hw} \end{aligned}$$

and therefore given fixed B, MAC reaches the minimum when $c_1 = c_2$.

		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

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.

G2) Excessive group convolution increases MAC

Group convolution is to reduce $FLOPs$ in convoluting feature maps (e.g., *depthwise convolution*). But when given a fixed $FLOPs$, as the number of groups g going up, MAC increases.

Now we focus on 1×1 *group convolution*,

$$B = \frac{hwc_1 c_2}{g}$$

and

$$\begin{aligned} MAC &= hwc_1 + hwc_2 + \frac{c_1 c_2}{g} \\ &= hwc_1 + \frac{Bg}{c_1} + \frac{B}{hw} \end{aligned}$$

		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.

G3) Network fragmentation reduces degree of parallelism

This is easily understandable. **Multi-path** is a common technique to generate different sets of features and improve accuracy. On the other hand, compared with "one big operation", using "one set of separating operations" will surely introduce extra overheads such as synchronization, like *group convolution*. It may slow down strong parallel computing devices like GPU.

	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.

G4) Element-wise operations are non-negligible

The element-wise operators have small $FLOPs$ but relatively heavy MAC, e.g., *ReLU*, *AddTensor*, *AddBias*. Specially, the authors also consider *depthwise convolution* as an element-wise operator because of the high $MAC/FLOPs$ ratio.

		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

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.

Conclusion

1. Use "balanced" convolutions (equal channel width); Base case: ShuffleNetV1 and MobileNetV2
2. Be aware of the cost of using group convolution; Base case: ShuffleNetV1
3. Reduce the degree of fragmentation; Base case: Inception Series and ShuffleNetV1
4. Reduce element-wise operations; Base case: MobileNetV2

Let's consider building units. The guidelines are only beneficial for efficiency, but the capacity of models is equally important. ShuffleNetV1 has been proven as an efficient model. Therefore, we use it as the base model, and try to improve it within our guidelines.

1. To Follow G1, ShuffleNetV1 has bottlenecks (such as 256-d -> 64-d -> 256-d), so we remove these bottlenecks by changing all 1×1 $GConv(c_{in} \neq c_{out})$ to 1×1 $GConv$ $(c_{in} = c_{out})$.
2. To Follow G2, we further change 1×1 $GConv$ $(c_{in} = c_{out})$ to 1×1 $Conv$ $(c_{in} = c_{out})$, making *channel shuffle* will not exchange extra information, thus move it to the bottom.
3. To Follow G3, the raw 1×1 $GConv$ should be blamed again. Meanwhile, we shall keep the number of branches low ($= 2$).
4. To Follow G4, replace *add* with *concat* while merging two branches.

Now, we are facing a contradiction that the number of output channels is larger than that of input channels in the units. To fix it, a simple operator called *channel split* was introduced, which equally split input channels into two branches. Nevertheless, while doing spatial downsampling, *channel split* is removed in order to double the number of output channels.

Also, the three successive element-wise operations, *concat*, *channel shuffle* and *channel split* , can be merged into a single element-wise operation.

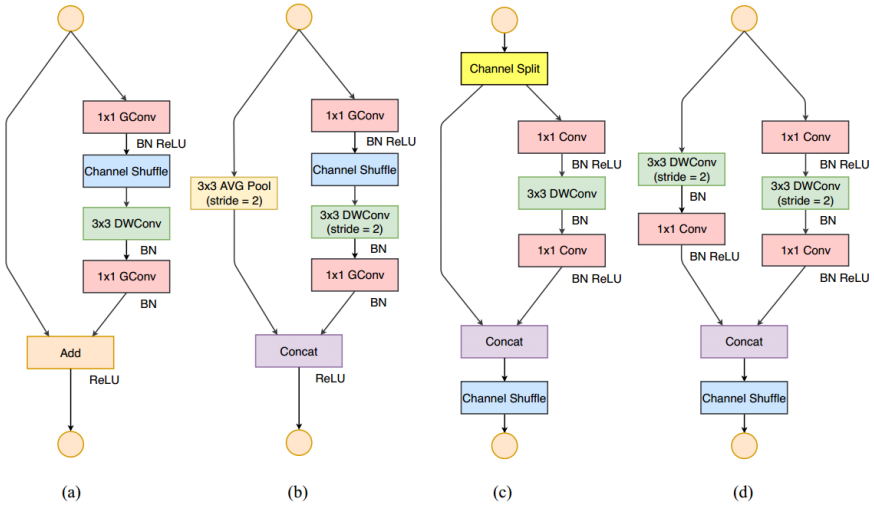


Fig. 3: Building blocks of ShuffleNet v1 [15] and this work. (a): the basic ShuffleNet unit; (b) the ShuffleNet unit for spatial down sampling (2×); (c) our basic unit; (d) our unit for spatial down sampling (2×). **DWConv**: depthwise convolution. **GConv**: group convolution.

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.

MobileNetV3

Time flies to June 2019. The most cutting-edge and GPU-consuming technology, **network architecture search (NAS)** has been being applied widely, and gradually reducing job positions in the DL market. This generation of MobileNets Searching for MobileNetV3 is tuned to mobile phone CPUs.

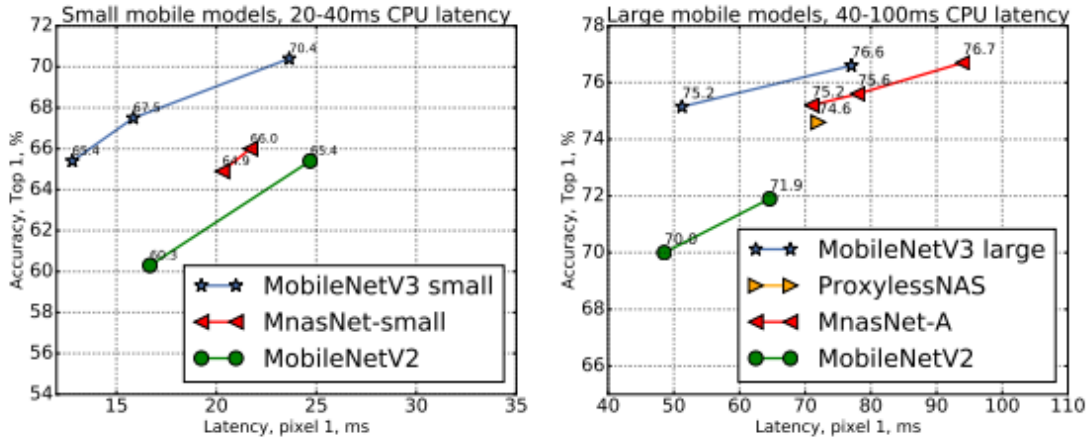


Figure 1. The trade-off between Pixel 1 latency and top-1 ImageNet accuracy. All models use the input resolution 224. V3 large and V3 small use multipliers 0.75, 1 and 1.25 to show optimal frontier. All latencies were measured on a single large core of the same device using TFLite[1]. MobileNetV3-Small and Large are our proposed next-generation mobile models.

Block-wise search

First, For block-wise search in **Large** mobile model, we employ a hardware-aware NAS approach which is similar to **MnasNet**, and get similar results. Therefore, we simply reuse the same **MnasNet-A1** as our initial Large model.

Then, for **Small** mobile model, we change multi-objective reward as follow:

$$ACC(m) \times [LAT(T)/TAR]^w$$

where $ACC(m)$ is accuracy, $LAT(m)$ is model's latency, and TAR stands for target latency. Specially, $w = -0.15$ (vs $w = -0.07$ in MnasNet) to compensate for the larger accuracy change for different latencies. Upon this, we apply NetAdapt and other optimizations to obtain the final **MobileNetV3-Small** model.

Layer-wise search

For layer-wise search, we modify this **NetAdapt** algorithm in MnasNet and minimize the ratio between latency change and accuracy change ($\frac{\Delta Acc}{\Delta latency}$). In short the technique proceeds as follows:

> 1. Starts with a seed network architecture found by platform-aware NAS.
> 2. For each step: (a) Generate a set of new *proposals*. Each proposal represents a modification of an architecture that generates at least $\delta = 0.01|L|$(L is the latency of the seed model) reduction in latency compared to the previous step.
>
>   (b) For each proposal we use the pre-trained model from the previous step and populate the new proposed architecture, truncating and randomly initializing missing weights as appropriate. Finetune each proposal for $T = 1000$ steps to get a coarse estimate of the accuracy.
>
>   (c) Selected best proposal according to some metric.

> 3. Iterate previous steps until target latency is reached.

Redesigning Expensive Layers and Nonlinearities

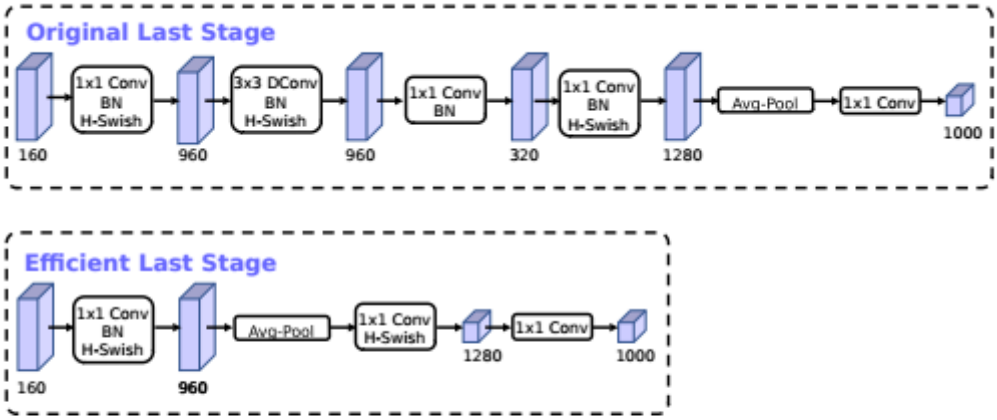


Figure 5. Comparison of original last stage and efficient last stage. This more efficient last stage is able to drop three expensive layers at the end of the network at no loss of accuracy.

The original last stage uses *inverted residual* to expand features, followed by *average pooling* and 1×1 convolution (dense layer), to get the final prediction. Now, we move *average pooling* forward, and the final set of features is now computed at 1×1 spatial resolution instead of 7×7 spatial resolution. This approach reduces 7ms latency and almost no hurt in accuracy.

Beside, we replace the filter number of the first 3×3 convolution layer (to produce initial features from image) from 32 to 16.

Swish is an effective substitute for *ReLU*, which can improve accuracy in many neural networks. However, the sigmoid part in *swish* is quite expensive to compute.

$$swish(x) = x * \sigma(x)$$

We replace the original sigmoid function to hard-sigmoid: $\frac{ReLU6(x+3)}{6}$, then we have:

$$h\text{-}swish(x) = x * \frac{ReLU6(x+3)}{6}$$

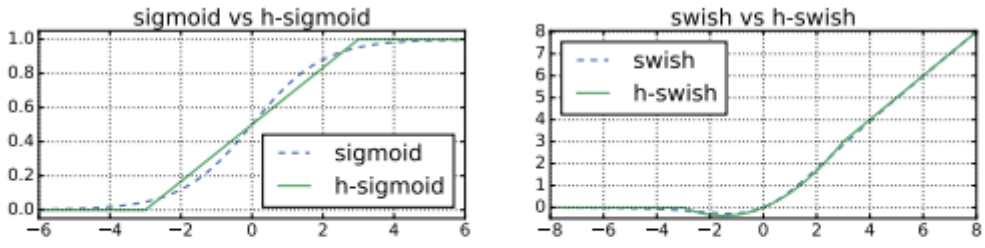


Figure 6. Sigmoid and swish nonlinearities and ther "hard" counterparts.

Large squeeze-and-excite

We set all the size of the **squeeze-and-excite** bottleneck to fixed to be $1/4$ of the number of channels in expansion layer, without discernible latency cost. We should also note that the *FCs* can be replaced by convolution layers.

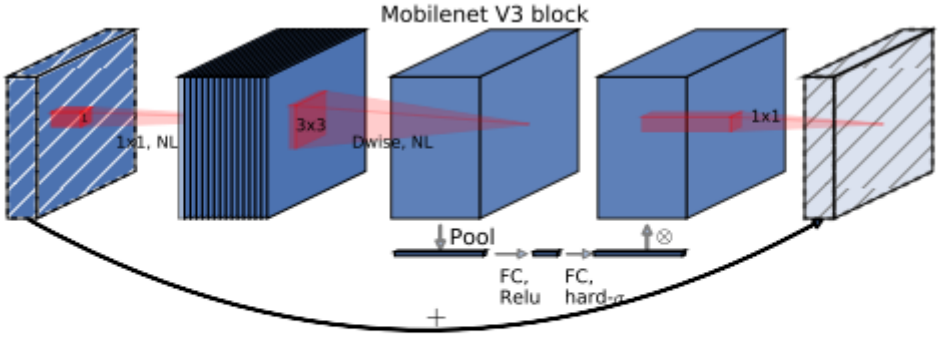


Figure 4. MobileNetV2 + Squeeze-and-Excite [20]. In contrast with [20] we apply the squeeze and excite in the residual layer. We use different nonlinearity depending on the layer, see section 5.2 for details.

The final MobileNetV3-Large and MobileNetV3-Small model:

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Table 1. Specification for MobileNetV3-Large. SE denotes whether there is a Squeeze-And-Excite in that block. NL denotes the type of nonlinearity used. Here, HS denotes h-swish and RE denotes ReLU. NBN denotes no batch normalization. s denotes stride.

Input	Operator	exp size	$\#out$	SE	NL	s
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1024	-	HS	1
$1^2 \times 1024$	conv2d 1x1, NBN	-	k	-	-	1

Table 2. Specification for MobileNetV3-Small. See table 1 for notation.