MobileNetV2 : Inverted Residuals and Linear Bottlenecks

2018.09.17
김보섭

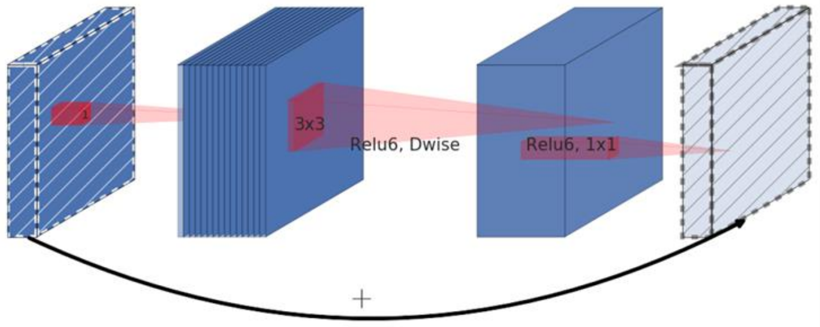
Agenda

Intro

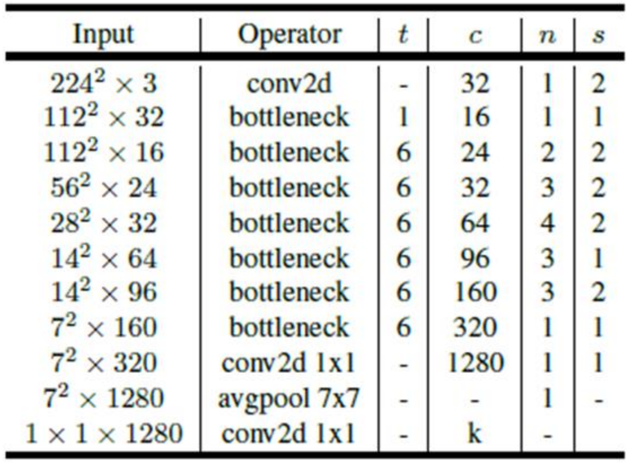
Resource가 제한된 환경 (eg. Mobile)에서 활용할 수 있는 새로운 Neural Net 구조와 이의 토대가 되는 Convolution block을 소개함

- 제한된 환경에서 활용할 수 있으려면?
  - Latency가 낮아야 (# of operation ↓)
  - Model의 크기가 작아야 (# of parameters ↓)
  - Inference 속도가 빨라야 (the memory footprint of intermediate tensor ↓)

Bottleneck residual block



MobileNetV2 architecture



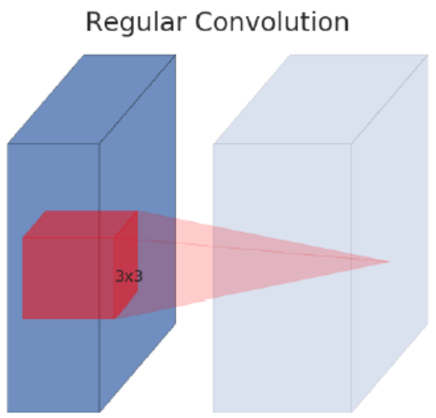
Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck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

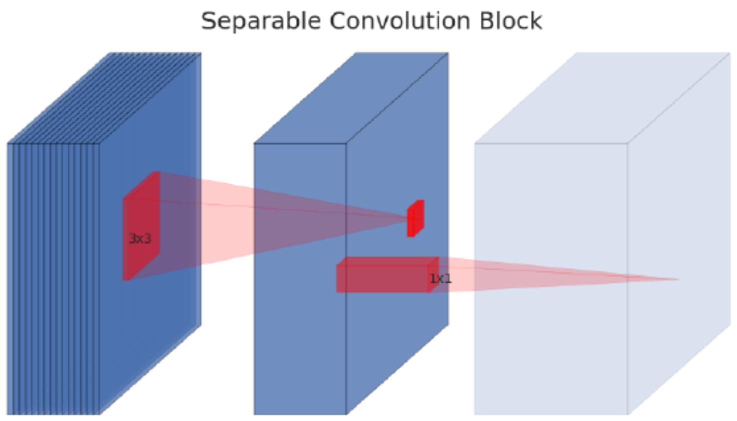
Preliminary

Depthwise Separable Convolution (1/2)

MobileNet, Xception 등 효율성을 추구하는 많은 Network에서 사용하는 Building block으로 MobileNet V2에서도 사용됨

- Key idea
  - Regular Convolution → Depthwise Convolution + Pointwise Convolution
    - Depthwise Convolution : Channel 별로 feature를 filtering
    - Pointwise Convolution : Channel 별로 filtering된 feature들을 combining

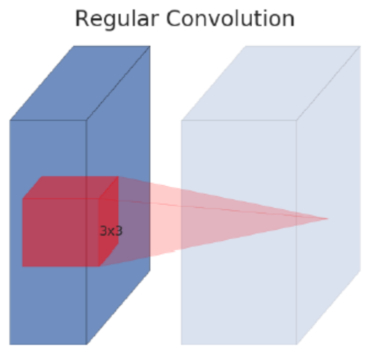


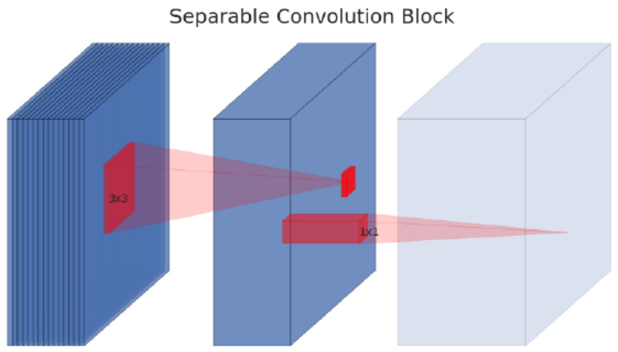


Preliminary

Depthwise Separable Convolution (2/2)

기존의 Regular Convolution과 비교시 Depthwise Separable Convolution는 계산 비용이 대략 Kernel의 Receptive field인 k^2 비율만큼 감소함





$input\ tensor\ L_i : h_i \times w_i \times d_i$
$output\ tensor\ L_j : h_j \times w_j \times d_j$
$kernel : R^{k \times k \times d_i \times d_j}$
$cost : h_i \times w_i \times d_i \times d_j \times k \times k$

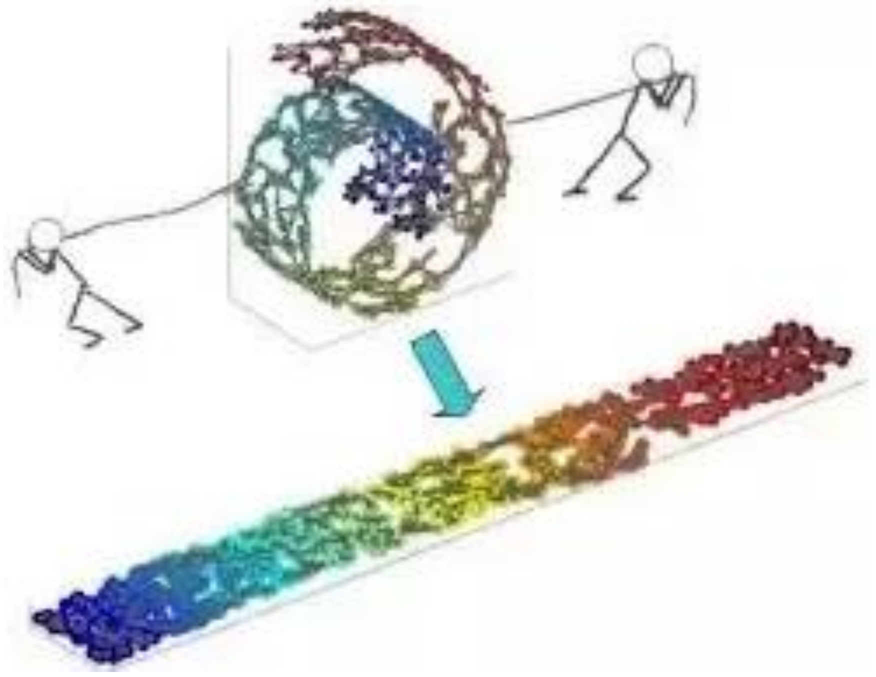
$input\ tensor\ L_i : h_i \times w_i \times d_i$
$output\ tensor\ L_j : h_j \times w_j \times d_j$
$depthwise\ kernel : R^{k \times k \times d_i}$
$pointwise\ kernel : R^{1 \times 1 \times d_i \times d_j}$
$cost : h_i \times w_i \times d_i \times k \times k + h_i \times w_i \times d_i \times d_j$

$$\frac{h_i w_i d_i (k^2 + d_j)}{h_i w_i d_i (d_j k^2)} \rightarrow \frac{k^2 + d_j}{d_j k^2}$$

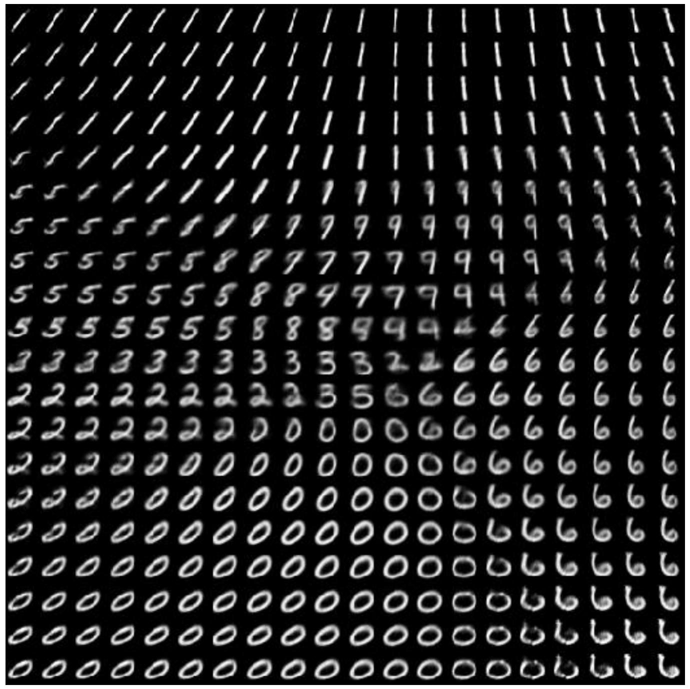
Architecture

Inverted Residuals and Linear Bottlenecks (1/7)

일반적으로 Manifold는 Representation에서 Low-dimensional subspace에 존재함



Representation : 3d
Manifold : 2d



Representation : 784d
Manifold : 2d

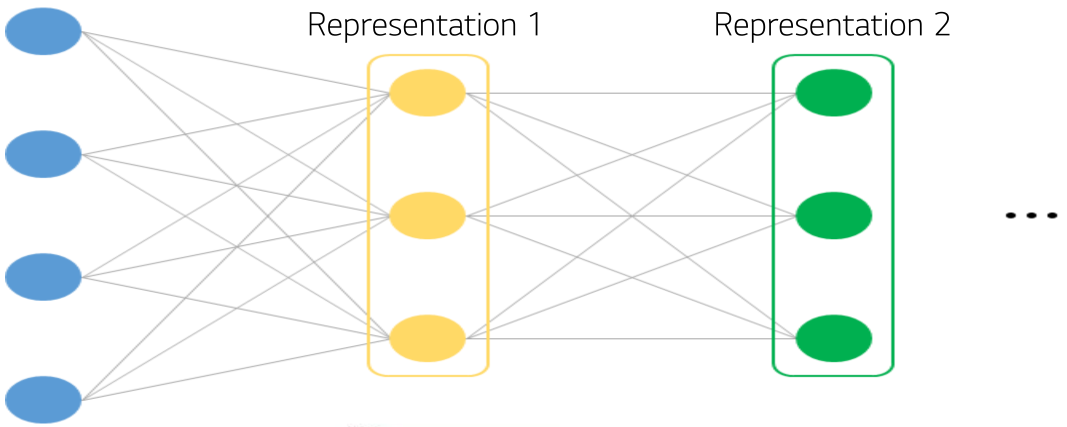
Representation : 어떤 data/feature가 표현되는 공간/초평면
Manifold : 어떤 data/feature가 고유로 가지고 있는 내재적 공간/초평면

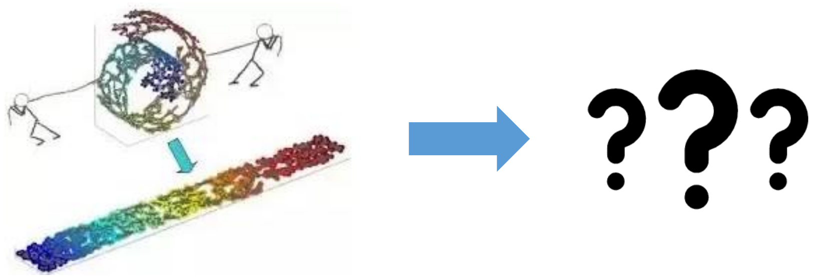
Architecture

Inverted Residuals and Linear Bottlenecks (2/7)

Neural Net (eg. CNN)의 layer (eg. Conv layer)의 Representation (Activation 거친 후)에도 Manifold가 존재한다면…

Question. Representation 1이 Low-dimension일 때, Representation 2의 Dimension이 Representation 2과 비슷하다면 Representation 1에 존재하는 Manifold는 잘 보존되는가?





Architecture

Inverted Residuals and Linear Bottlenecks (3/7)

Representation 1이 Low-dimension일 때, Representation 2는 Representation 1보다 High-dimension이어야 Manifold를 보존할 수 있음

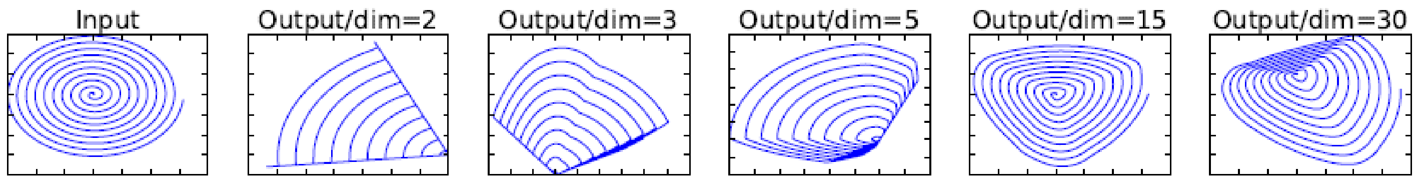
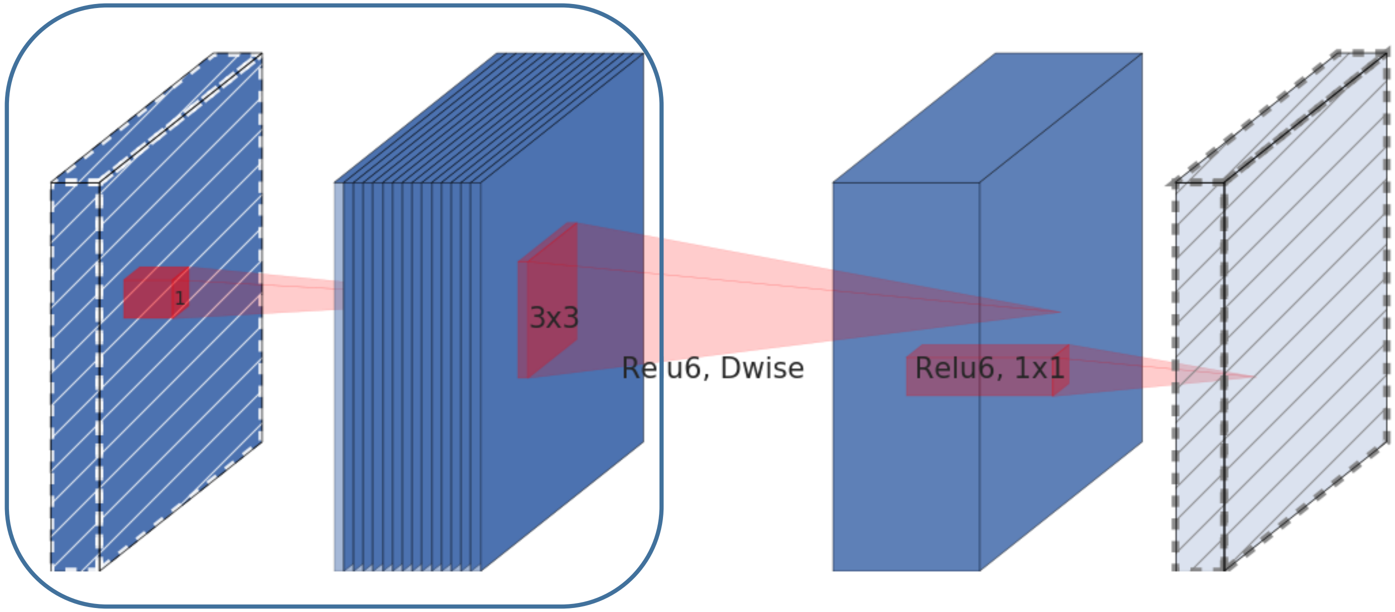


Figure 1: Examples of ReLU transformations of low-dimensional manifolds embedded in higher-dimensional spaces. In these examples the initial spiral is embedded into an n-dimensional space using random matrix T followed by ReLU, and then projected back to the 2D space using T^{-1}. In examples above $n = 2, 3$ 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.

Architecture

Inverted Residuals and Linear Bottlenecks (4/7)

Bottleneck (low-dimensional representation)에 embedded 되어있는 Manifold를 계속 전달하기위해서 Expansion을 활용



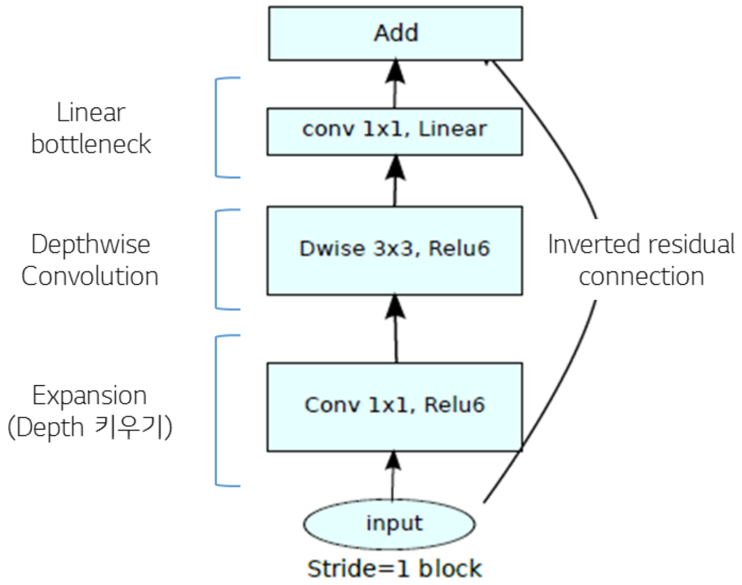
Pointwise Convolution (1 x1 Conv2d)로
Depth를 expansion

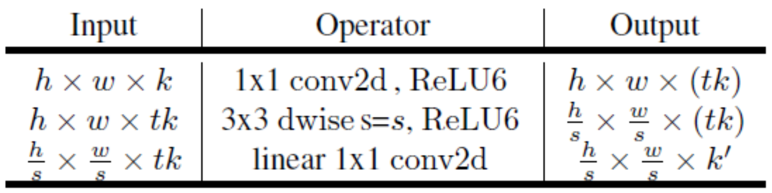
Architecture

Inverted Residuals and Linear Bottlenecks (5/7)

Expansion을 함과 동시에 아래의 두 가지를 가정을 토대로, “Bottleneck residual block” 을 생성하고, 이를 실험결과로 증명함

- Bottleneck (low-dimensional representation)이 Manifold가 잘 보존하고 있다고 가정, skip-connection을 Bottleneck간에 연결
- Non-linearity가 Expansion (pointwise convolution) 그리고 Depthwise Convolution에서 들어가므로, 마지막에는 Non-linearity를 제거





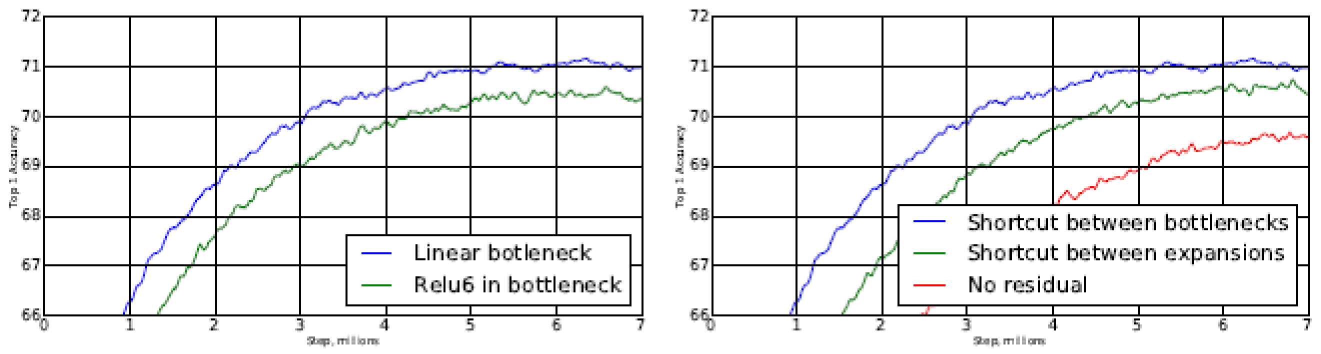
Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.

Architecture

Inverted Residuals and Linear Bottlenecks (6/7)

실험결과 Bottleneck간의 Short-cut을 연결했을 때, Bottleneck 생성시 Non-linearity를 제거하는 것이 성능이 좋음



(a) Impact of non-linearity in the bottleneck layer. (b) Impact of variations in residual blocks.

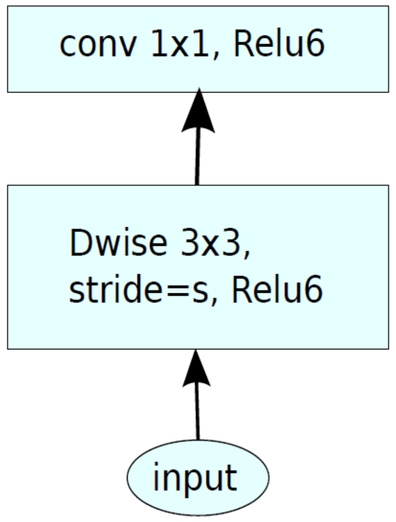
Figure 6: The impact of non-linearities and various types of shortcut (residual) connections.

Architecture

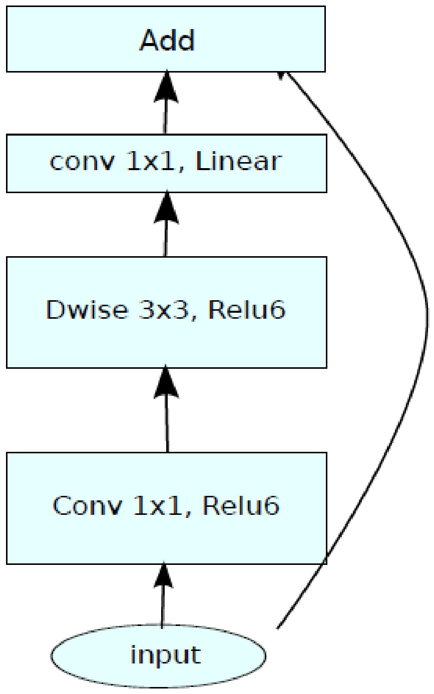
Inverted Residuals and Linear Bottlenecks (7/7)

기존의 MobileNet에서 쓰이던 Depthwise Separable Convolution block과 MobileNet V2에서 사용되는 Bottleneck residual block을 비교하면 아래와 같음

Depthwise Separable Convolution block



Bottleneck residual block

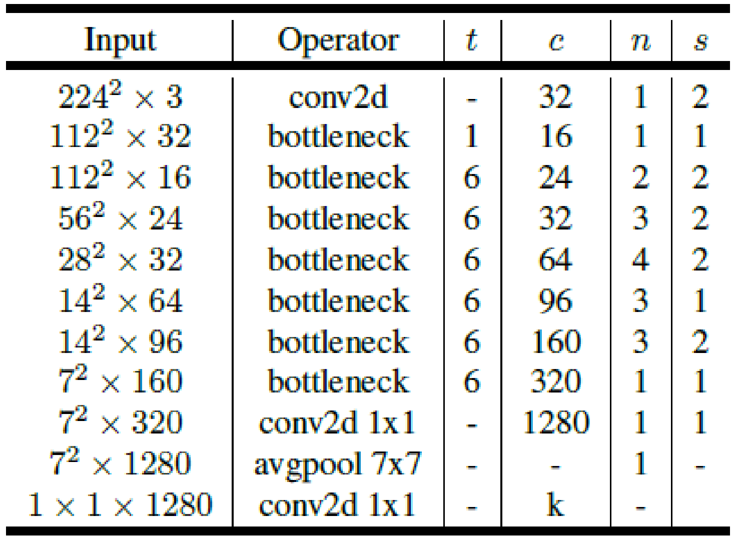


Stride=1 block

Architecture

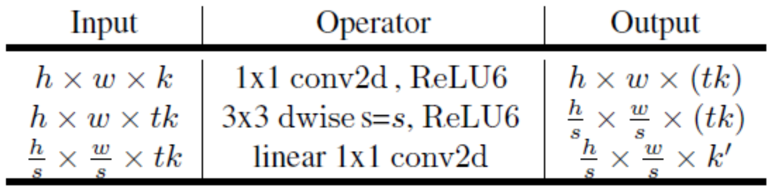
MobileNetV2 (1/2)

MobileNetV2는 Bottleneck residual block을 활용하여 아래와 같이 구성함



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.



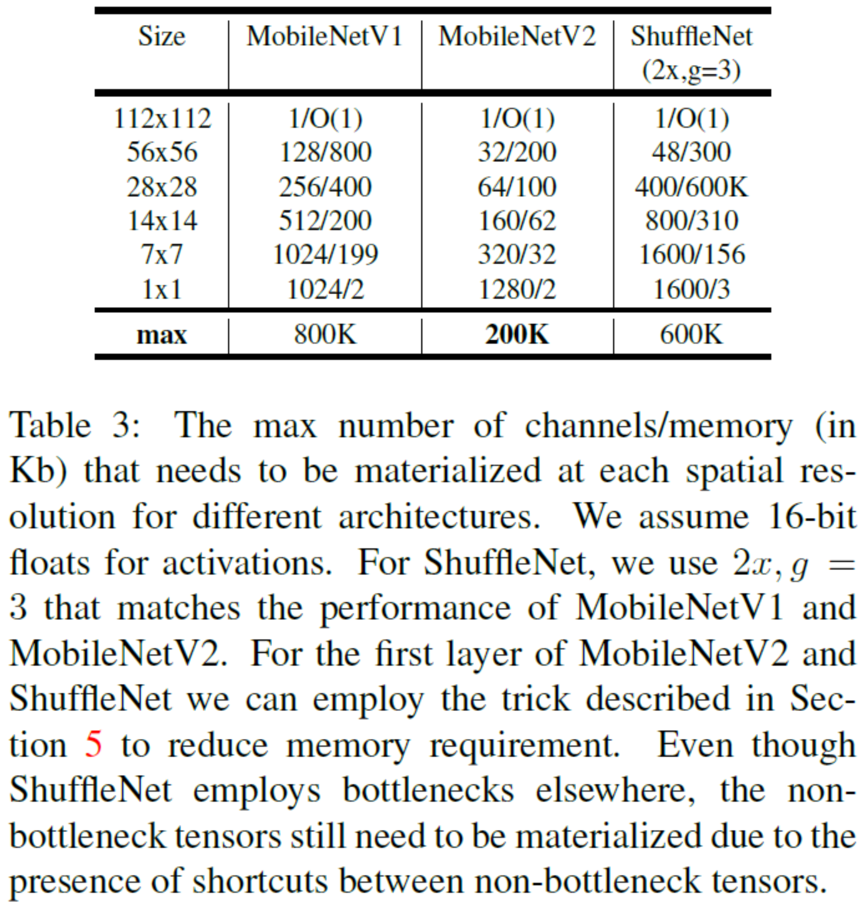
Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.

Architecture

MobileNetV2 (2/2)

다른 Architecture에 비해서 Inference시 Intermediate tensor의 최대 크기가 200k이므로 Cache memory를 잘 활용할 수 있음 (main memory access ↓)



Size	MobileNetV1	MobileNetV2	ShuffleNet (2x,g=3)
112x112	1/O(1)	1/O(1)	1/O(1)
56x56	128/800	32/200	48/300
28x28	256/400	64/100	400/600K
14x14	512/200	160/62	800/310
7x7	1024/199	320/32	1600/156
1x1	1024/2	1280/2	1600/3
max	800K	**200K**	600K

Table 3: The max number of channels/memory (in Kb) that needs to be materialized at each spatial resolution for different architectures. We assume 16-bit floats for activations. For ShuffleNet, we use $2x, g = 3$ that matches the performance of MobileNetV1 and MobileNetV2. For the first layer of MobileNetV2 and ShuffleNet we can employ the trick described in Section 5 to reduce memory requirement. Even though ShuffleNet employs bottlenecks elsewhere, the non-bottleneck tensors still need to be materialized due to the presence of shortcuts between non-bottleneck tensors.

Experiment results

ImageNet Classification

MobileNetV2의 Variant들이 Frontier를 형성하는 것을 볼 수 있음.

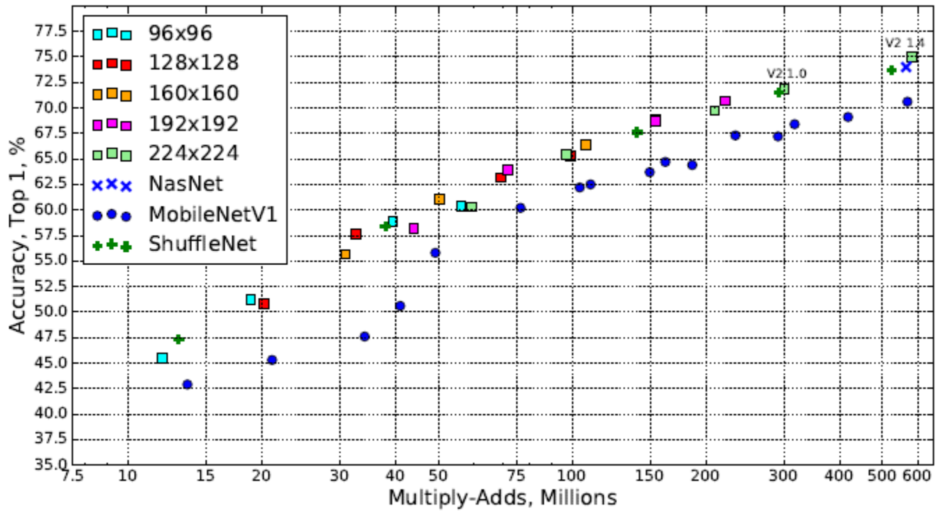
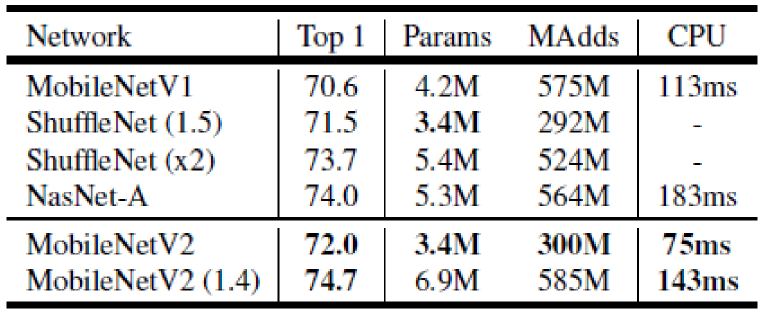


Figure 5: Performance curve of MobileNetV2 vs MobileNetV1, ShuffleNet, NAS. For our networks we use multipliers 0.35, 0.5, 0.75, 1.0 for all resolutions, and additional 1.4 for for 224. Best viewed in color.



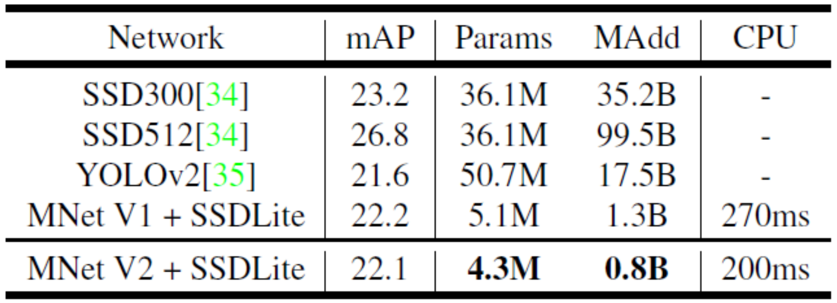
Network	Top 1	Params	MAdds	CPU
MobileNetV1	70.6	4.2M	575M	113ms
ShuffleNet (1.5)	71.5	**3.4M**	292M	-
ShuffleNet (x2)	73.7	5.4M	524M	-
NasNet-A	74.0	5.3M	564M	183ms
MobileNetV2	**72.0**	**3.4M**	**300M**	**75ms**
MobileNetV2 (1.4)	**74.7**	6.9M	585M	**143ms**

Table 4: Performance on ImageNet, comparison for different networks. As is common practice for ops, we count the total number of Multiply-Adds. In the last column we report running time in milliseconds (ms) for a single large core of the Google Pixel 1 phone (using TF-Lite). We do not report ShuffleNet numbers as efficient group convolutions and shuffling are not yet supported.

Experiment results

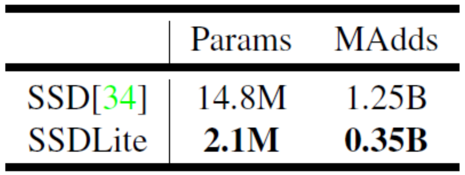
Object Detection

MS COCO 데이터에 대하여 MobileNetV2 + SSDLite가 Model이 훨씬 작으면서 Operation도 적지만 성능이 더 좋음



Network	mAP	Params	MAdd	CPU
SSD300[34]	23.2	36.1M	35.2B	-
SSD512[34]	26.8	36.1M	99.5B	-
YOLOv2[35]	21.6	50.7M	17.5B	-
MNet V1 + SSDLite	22.2	5.1M	1.3B	270ms
MNet V2 + SSDLite	22.1	**4.3M**	**0.8B**	200ms

Table 6: Performance comparison of MobileNetV2 + SSDLite and other realtime detectors on the COCO dataset object detection task. MobileNetV2 + SSDLite achieves competitive accuracy with significantly fewer parameters and smaller computational complexity. All models are trained on `trainval35k` and evaluated on `test-dev`. SSD/YOLOv2 numbers are from [35]. The running time is reported for the large core of the Google Pixel 1 phone, using an internal version of the TF-Lite engine.



	Params	MAdds
SSD[34]	14.8M	1.25B
SSDLite	**2.1M**	**0.35B**

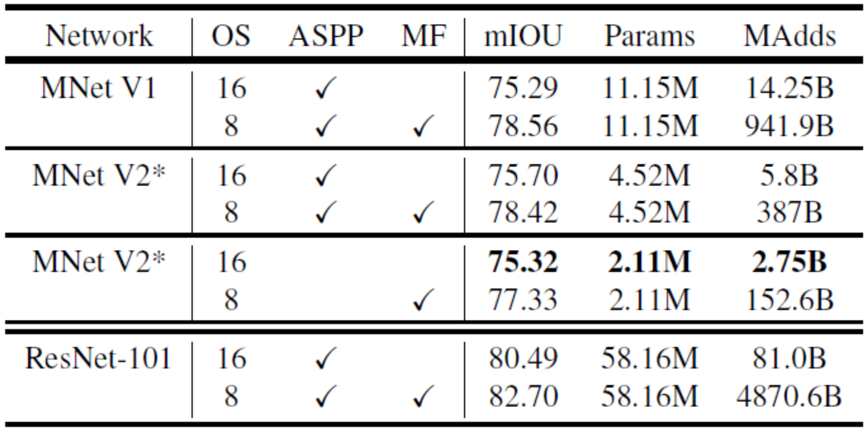
Table 5: Comparison of the size and the computational cost between SSD and SSDLite configured with MobileNetV2 and making predictions for 80 classes.

SSDLite : 기존의 SSD (Single Shot Multibox Detector)에서 기존의 Regular Convolution을 Depthwise-Separable Convolution으로 대체한 model

Experiment results

Semantic Segmentation

MobileNet (MNet V1), MobileNet V2 (MNet V2), ResNet을 Feature extractor 로 활용하여 DeepLabv3를 구성하여 비교한 결과는 아래와 같음



Network	OS	ASPP	MF	mIOU	Params	MAdds
MNet V1	16	✓		75.29	11.15M	14.25B
	8	✓	✓	78.56	11.15M	941.9B
MNet V2*	16	✓		75.70	4.52M	5.8B
	8	✓	✓	78.42	4.52M	387B
MNet V2*	16			**75.32**	**2.11M**	**2.75B**
	8		✓	77.33	2.11M	152.6B
ResNet-101	16	✓		80.49	58.16M	81.0B
	8	✓	✓	82.70	58.16M	4870.6B

Table 7: MobileNet + DeepLabv3 inference strategy on the PASCAL VOC 2012 *validation* set. **MNet V2***: Second last feature map is used for DeepLabv3 heads, which includes (1) Atrous Spatial Pyramid Pooling (**ASPP**) module, and (2) 1×1 convolution as well as image-pooling feature. **OS**: *output_stride* that controls the output resolution of the segmentation map. **MF**: Multi-scale and left-right flipped inputs during test. All of the models have been pretrained on COCO. The potential candidate for on-device applications is shown in bold face. PASCAL images have dimension 512×512 and atrous convolution allows us to control output feature resolution without increasing the number of parameters.

Conclusion

상용 framework (eg. Tensorflow, Pytorch) 등에 있는 operation으로 mobile application에 적합한 module과 architecture를 설계함

- 성능측면
  - Image Classification, Object Detection, Semantic Segmentation 등의 Task에 대해서 Mobile 환경에서도 기존의 Non-mobile architecture만큼 성능을 확보
- 이론측면
  - 제안한 Bottleneck residual block은 기존의 Convolution과는 달리 expressiveness (encoded by expansion layer)와 capacity (encoded by bottleneck input)을 구분함 --> 향후 연구거리

Q & A

감사합니다.