SOTA deep CNN architectures and their principles: from AlexNet to EfficientNet (2)

EfficientNet

Artificial Intelligence

Creating the Future

Dong-A University

Division of Computer Engineering & Artificial Intelligence

References

Main

- https://theaisummer.com/cnn-architectures/

blog Sub

- https://deep-learning-study.tistory.com/
- https://hoya012.github.io/blog/DenseNet-Tutorial-1/
- https://kjhov195.github.io/2020-02-11-CNN_architecture_3/
- https://poddeeplearning.readthedocs.io/ko/latest/CNN/GoogLeNet/
- https://bskyvision.com/539
- https://m.blog.naver.com/laonple/220686328027

PyTorch tutorial

- website : https://pytorch.org/
- Korea website : https://pytorch.kr/

github

- https://github.com/pytorch
- https://github.com/9bow/PyTorch-tutorials-kr
- torchvision : https://github.com/pytorch/vision
- https://github.com/weiaicunzai/pytorch-cifar100

Nikolas Adaloglou

https://theaisummer.com/cnn-architectures/

SOTA of CNN Architecture

Part1

- AlexNet (2012)
- VGG (2014)
- InceptionNet / GoogleNet (2014)
- Inception V2 (2015)
- Inception V3 (2016)
- ResNet (2015)
- Inception V4, Inception-ResNet (2016)
- DenseNet (2017)
- BigTransfer (BiT) (2020)

Part2

- **EfficientNet (2019)**
- Noisy Student (2020)
- Meta - Pseudo Labels (2020)
- EfficientDet (2021)
- **EfficientNetV2 (2021)**

- GoogleNet (2014 ImageNet winner) : 74.8% top-1 accuracy, about 6.8M parameters
- SENet (2017 ImageNet winner) : 82.7% top-1 accuracy, about 145M parameters
- GPipe (2018, SOTA IMageNet) : 84.% top-1 accuracy, about 557M parameters

7 EfficientNet

Mingxing Tan, Quoc V. Le, "**EfficientNet : Rethinking Model Scaling for Convolutional Neural Networks**," arXiv:1905.11946, 2019 (**Google Research Brain Team**)

https://theaisummer.com/cnn-architectures/
https://ys-cs17.tistory.com/30
https://ys-cs17.tistory.com/34 PR-169 https://youtu.be/Vhz0quyvR7I

Pytorch
https://github.com/zylo117/Yet-Another-EfficientDet-Pytorch
https://github.com/lukemelas/EfficientNet-PyTorch

Official
https://github.com/google/automl
https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet

Abstract

- Convolutional Neural Networks (**ConvNets**) are commonly developed at a fixed resource budget, and then **scaled up for better accuracy if more resources are available**.
- In this paper, we systematically **study model scaling** and identify that carefully **balancing network depth, width, and resolution can lead to better performance**.
- Propose **a new scaling method that uniformly scales all dimensions of depth/width/resolution using a simple yet highly effective compound coefficient**. Demonstrate the effectiveness of this method on scaling up MobileNets and ResNet.
- **Use neural architecture search to design a new baseline network and scale it up to obtain a family of models**, called ***EfficientNets***, which achieve much better accuracy.
- **EfficientNet-B7** achieves state-of-the-art **84.3% top-1 accuracy on ImageNet**, while being **8.4x smaller** and **6.1x faster on inference** than the best existing ConvNet.
- Our EfficientNets also **transfer well** and achieve state-of-the-art accuracy on CIFAR-100 (91.7%), Flowers (98.8%), and 3 other transfer learning datasets, with an order of magnitude fewer parameters.
- Source code is at https: //github.com/tensorflow/tpu/tree/master/models/official/efficientnet.

7

EfficientNet

1. Introduction

- Scaling up ConvNets is widely used to achieve better accuracy.
  - ✓ **ResNet** (He et al., 2016) can be scaled up from ResNet-18 to ResNet-200 by using more layers
  - ✓ **GPipe** (Huang et al., 2018) achieved 84.3% ImageNet top-1 accuracy by scaling up a baseline model 4 times larger.

- The most common way is to scale up ConvNets by their **depth**, **width**, or image **resolution**.
  - ✓ In previous work, it is common to scale only one of the three dimensions.
  - ✓ Though it is possible to scale two or three dimensions arbitrarily, arbitrary scaling requires tedious manual tuning and still often yields sub-optimal accuracy and efficiency.

- Authors want to study and rethink the process of scaling up ConvNets.
  - ✓ **Question : Is there a Principled method to scale up ConvNets that can achieve better accuracy and efficiency?**

- Empirical study shows that it is critical to **balance all dimensions** of network width/depth/resolution, and surprisingly such balance can be achieved by **simply scaling each of them with constant ratio**.

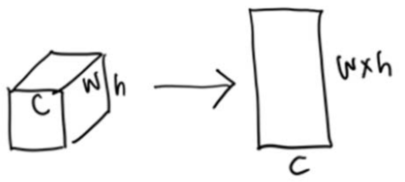
- Based on this observation, authors propose a **compound scaling methods**.

7

EfficientNet

Compound scaling method

- **Compound scaling method** : Uniformly scales network width / depth / resolution with a set of fixed scaling coefficients.



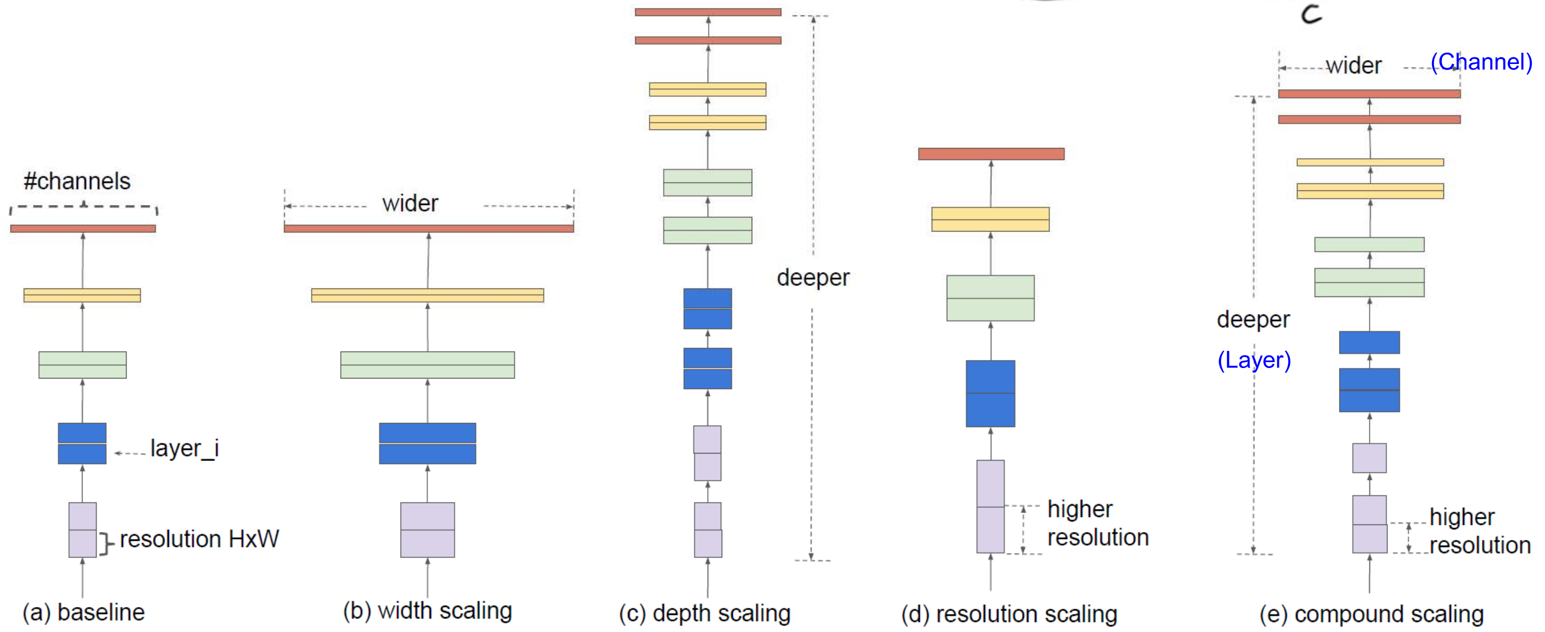


Figure 2. Model Scaling. (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

- Effectiveness of model scaling heavily depend on the baseline network; Use ***neural architecture search*** **to develop a new baseline network, and scale it up to obtain a family of models, called** ***EfficientNets*****.**

7 EfficientNet

2. Related Work - ConvNet Accuracy

- ConvNets have become increasingly more accurate by going bigger.
  - ✓ While the 2014 ImageNet winner **GoogleNet (Szegedy et al., 2015) achieves 74.8% top-1 accuracy with about 6.8M parameters**, the 2017 ImageNet winner **SENet (Hu et al., 2018) achieves 82.7% top-1 accuracy with 145M parameters**.
  - ✓ Recently, **GPipe** (Huang et al., 2018) further pushes the state-of-the-art **ImageNet top-1 validation accuracy to 84.3% using 557M parameters**.

⬇

- Although higher accuracy is critical for many applications, we have already **hit the hardware memory limit**, and thus **further accuracy gain needs better efficiency**.

2. Related Work - ConvNet Efficiency

- Deep ConvNets are often over-parameterized.
  - ✓ **Model compression** (Han et al., 2016; He et al., 2018; Yang et al., 2018) is a common way to reduce model size by trading accuracy for efficiency.
  - ✓ It is also common to handcraft efficient mobile-size ConvNets, such as **SqueezeNets, MobileNets, and ShuffleNets**.
- Recently, **neural architecture search becomes increasingly popular** in designing efficient mobile-size ConvNets (Tan et al., 2019; Cai et al., 2019) such as **MNasNet**.

⬇

- However, **it is unclear how to apply these techniques for larger models that have much larger design space and much more expensive tuning cost**.
- Authors aims to study model efficiency for super large ConvNets that surpass SOTA accuracy. Resort to **Model scaling**.

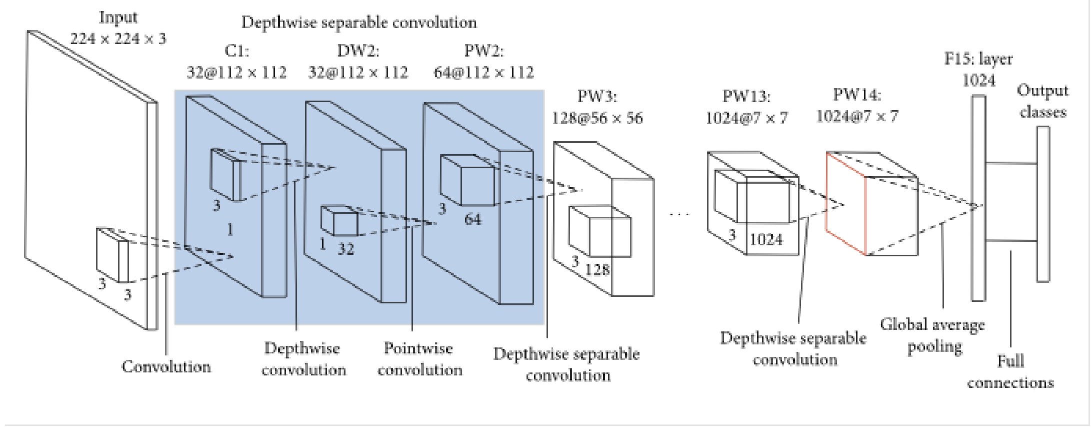
7 **EfficientNet**

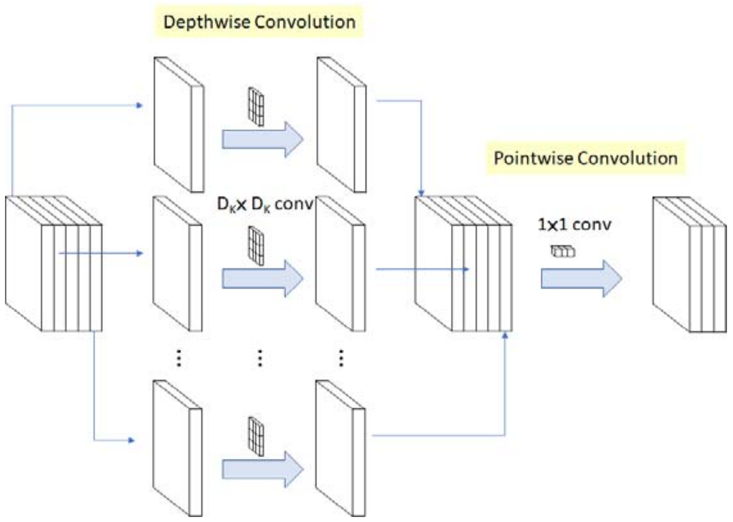
2. Related Work - Model Scaling

- There are many ways to scale a ConvNet for different resource constraints
  - ✓ **ResNet** (He et al., 2016) can be scaled down (e.g., ResNet-18) or up (e.g., ResNet-200) by **adjusting network depth (#layers)**
  - ✓ **WideResNet** (Zagoruyko & Komodakis, 2016) and **MobileNets** (Howard et al., 2017) can be scaled by **network width (#channels)**.
  - ✓ It is also well-recognized that **bigger input image size will help accuracy** with the overhead of more FLOPS

- The network **depth** and **width** are both important for ConvNets expressive power, it still remains an open question of **how to effectively scale a ConvNet to achieve better efficiency and accuracy**.

MobileNet body architecture





Depthwise Separable Convolution

7

EfficientNet

3.1 Problem Formulation

- A ConvNet Layer i can be defined as a function: $Y_i = \mathcal{F}_i(X_i)$, where $\mathcal{F}_i$ is operator, Y_i is output tensor, X_i is input tensor.
- A ConvNet $\mathcal{N}$ can be represented by a list of composed layer:

$$\mathcal{N} = \mathcal{F}_k \odot \cdots \odot \mathcal{F}_2 \odot \mathcal{F}_1(X_1) = \odot_{j=1,..,k} \mathcal{F}_j(X_1)$$

- Define a ConvNet $\mathcal{N}$ as:

Input tensor

Spatial Dimension

$$\mathcal{N} = \bigodot_{i=1,..,s} \mathcal{F}_i^{L_i}\left(X_{<H_i, W_i, C_i>}\right)$$

Channel Dimension

stage i

layer F_i is repeated L_i times in stage i

- Practically, ConvNet layers are often partitioned into multiple stage and all layers in each stage share the same architecture (ex. ResNet : 5 stage, 각 stage 상 모든 layers는 같은 아키텍처임, 첫번째 layer가 down-sampling하는 것을 제외하고)

- Unlike regular ConvNet designs that mostly focus on finding the best layer architecture F_i, **model scaling tries to expand the network length (L_i), width (C_i), and/or resolution (H_i, W_i) without changing F_i predefined in the baseline network**.
- By fixing F_i, model scaling simplifies the design problem for new resource constraints, but it still remains a large design space to explore different (L_i, C_i, H_i, W_i) for each layer.

- In order to further reduce the design space, we restrict that **all layers must be scaled uniformly with constant ratio**. Our target is to **maximize the model accuracy for any given resource constraints**.

$$\begin{aligned} \max_{d,w,r} \quad & Accuracy\big(\mathcal{N}(d,w,r)\big) \\ s.t. \quad & \mathcal{N}(d,w,r) = \bigodot_{i=1\ldots s} \hat{\mathcal{F}}_i^{d\cdot \hat{L}_i}\big(X_{\langle r\cdot \hat{H}_i,\, r\cdot \hat{W}_i,\, w\cdot \hat{C}_i\rangle}\big) \\ & \text{Memory}(\mathcal{N}) \leq \text{target\_memory} \\ & \text{FLOPS}(\mathcal{N}) \leq \text{target\_flops} \end{aligned} \quad (2)$$

Coefficients for scaling network depth d, width w, and resolution r

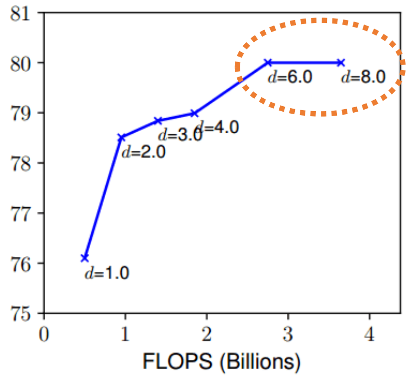
$\hat{\mathcal{F}}_i, \hat{L}_i, \hat{H}_i, \hat{W}_i, \hat{C}_i$ are predefined parameters in baseline network

7

EfficientNet

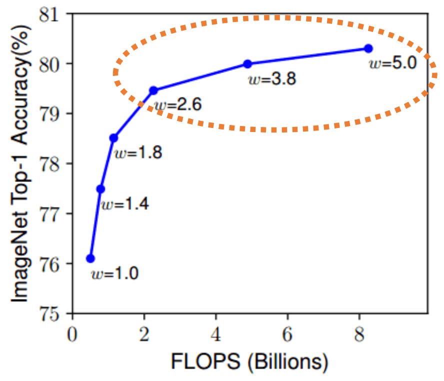
3.2 Scaling Dimensions – Depth (d)

- Scaling network depth is the most common way used by many ConvNets. The intuition is that **deeper ConvNet can capture richer and more complex features and generalize well on new tasks**.
- However, **the accuracy gain of very deep network diminishes** although several techniques, such as skip connection and batch normalization.
  - ✓ For example, ResNet-1000 has similar accuracy as ResNet-101 even though it has much more layers



3.2 Scaling Dimensions – Width (w)

- Scaling network width is commonly used for **small size models**.
- As discussed in WideResNet, **wider networks tend to be able to capture more fine-grained features and are easier to train.**
- However, **extremely wide but shallow networks tend to have difficulties in capturing higher level features**.



The accuracy quickly saturates when networks become much wider with larger w.

Figure 3. **Scaling Up a Baseline Model with Different Network Depth (*d*).**
Bigger networks with larger width, depth, or resolution tend to achieve higher accuracy, but the accuracy gain quickly saturate after reaching 80%, demonstrating the limitation of single dimension scaling. Baseline network is described in Table 1.

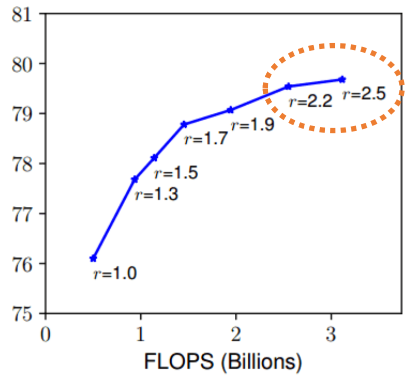
7

EfficientNet

3.2 Scaling Dimensions – Resolution (r)

- With higher resolution input images, ConvNets can potentially **capture more fine-grained patterns**.
  - ✓ Starting from 224x224 in early ConvNets, modern ConvNets tend to use 299x299 or 331x331 for better accuracy. Recently, **GPipe** (Huang et al., 2018) achieves state-of-the-art ImageNet accuracy with **480x480** resolution.
  - ✓ Higher resolutions, such as **600x600**, are also widely used in object detection ConvNets

Observation 1 : *Scaling up any dimension of network width, depth, or resolution improves accuracy, but* ***the accuracy gain diminishes for bigger models.***



Higher resolutions improve accuracy, but the accuracy gain diminishes for very high resolutions (r = 1.0 denotes resolution 224x224 and r = 2.5 denotes resolution 560x560).

Figure 3. **Scaling Up a Baseline Model with Different Network Depth (*d*).**
Bigger networks with larger width, depth, or resolution tend to achieve higher accuracy, but the accuracy gain quickly saturate after reaching 80%, demonstrating the limitation of single dimension scaling. Baseline network is described in Table 1.

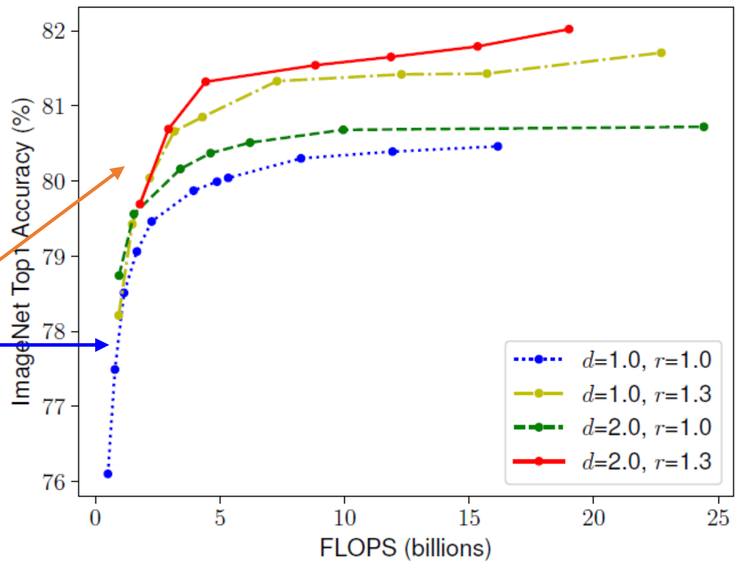
7

EfficientNet

3.3 Compound Scaling

- Empirically observe that **different scaling dimensions are not independent**. (예, resolution이 커지면 width와 depth가 증가)
- Intuitively, if **the input image is bigger**, then **the network needs more layers** to increase the receptive filed and **more channels** to capture more fine-grained patterns on the bigger image.

- If we only scale network width w without changing depth (d=1.0) and resolution (r=1.0), the accuracy saturates quickly.
- **With deeper (d=2.0) and higher resolution (r=2.0), width scaling achieves much better accuracy under the same FLOPS cost.**



Observation 2 : *In order to pursue better accuracy and efficiency, **it is critical to balance all dimensions of network width, depth, and resolution** during ConvNet scaling.*

Figure 4. **Scaling Network Width for Different Baseline Networks**. Each dot in a line denotes a model with different width coefficient (w). All baseline networks are from Table 1. The first baseline network (d=1.0, r=1.0) has 18 convolutional layers with resolution 224x224, while the last baseline (d=2.0, r=1.3) has 36 layers with resolution 299x299.

7 EfficientNet

3.3 Compound Scaling

- Compound scaling method : **Use a compound coefficient ϕ to uniformly scales network width, depth, and resolution** in a principled way:

$$\begin{aligned} \text{depth: } & d = \alpha^{\phi} \\ \text{width: } & w = \beta^{\phi} \\ \text{resolution: } & r = \gamma^{\phi} \\ \text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned} \tag{3}$$

- **α,β,γ are constants** that can be determined by a small grid search.
- Intuitively, ϕ is a user-specified coefficient that controls how many more resources are available for model scaling,
- while α,β,γ specify how to assign these extra resources to network width(w), depth(d), resolution(r) respectively.

- Notably, **FLOPs of a regular convolution op is proportional to d, w^2, r^2.**
  - ✓ Doubling d will double FLOPs but doubling w or r will increase FLOPs by four times.
  - ✓ Since convolution ops usually dominate the computation cost in ConvNets, **Scaling a ConvNet will approximately increase total FLOPs by $(\alpha \cdot \beta^2 \cdot \gamma^2)^{\phi}$.**
- We constraint $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$ s.t. for any new ϕ, **the total FLOPs will approximate increase by 2^{ϕ}.**

7 EfficientNet

4. EfficientNet Architecture

- **Model Scaling은 baseline network 상에 layer operator $\hat{\mathcal{F}}_i$을 변경하지 않으므로, 좋은 baseline network 가지는 것이 매우 중요함.**
- ***Accuracy와 FLOPS에 최적화한 Multi-object neural architecture search** 영향*에 따라 새로운 mobile-size baseline을 개발하고, 이를 *EfficientNet*이라 부름

➢ New baseline network by performing a neural architecture search using the **AutoML MNAS framework**, which optimizes both accuracy and efficiency (FLOPS).

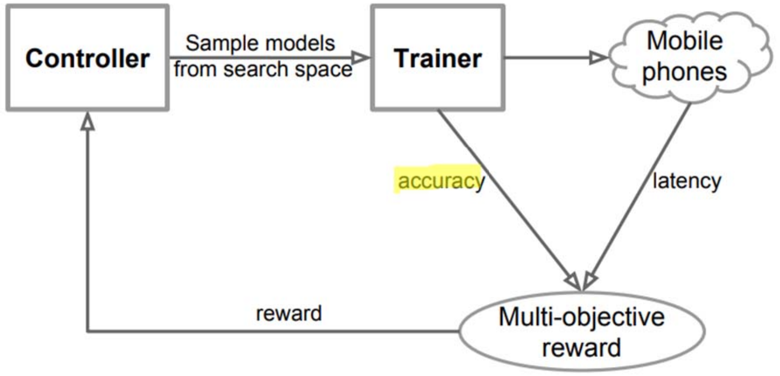
- **Optimization Goal**

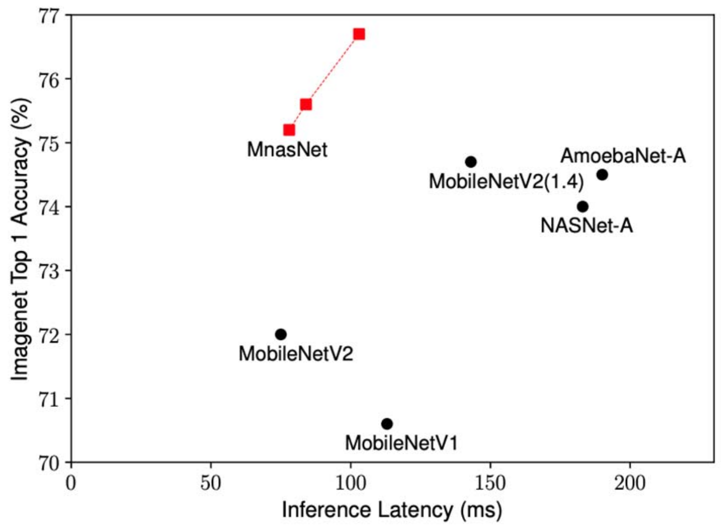
$$ACC(m) \times [FLOPS(m)/T]^{\omega}$$

- $ACC(m)$, $FLOPS(m)$: Model m의 accuracy와 FLOPS
- T : Target FLOPS
- $\omega = -0.07$: Accuracy와 FLOPS 간 trade-off를 조정하는 hyperparameter

- We optimize **FLOPS** rather than **latency** since we are not targeting any specific hardware device

MnasNet: Platform-Aware Neural Architecture Search for Mobile (CVPR2019) - Factorized Hierarchical Search Space 제안





7

EfficientNet

4. EfficientNet Architecture

- **EfficientNet-B0**라는 network 찾음 : **MnasNet과 유사한 MBConv 구조이나, FLOPS target (400M)이 커서, MnasNet보다 다소 큰 구조임 (MnasNet과 동일한 Search space 사용)**
- Main building block : **Mobile inverted Bottleneck Convolution (MBConv)**, to which add squeeze-and-excitation optimization

Table 1. **EfficientNet-B0 baseline network** – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $\langle \hat{H}_i, \hat{W}_i \rangle$ and output channels $\hat{C}_i$. Notations are adopted from equation 2.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

MnasNet : https://ai.googleblog.com/2018/08/mnasnet-towards-automating-design-of.html

EfficientNet B1 to B7

- Starting from the Baseline EfficientNet-B0, apply Compound scaling Method to scale it up with two steps.

Step 1

- **First fix ϕ=1, assuming twice more resource available and do a small grid search of α,β,γ** by Eq. (2),(3).
- The best values for Efficient-B0 are α=1.2, β=1.1, γ=1.15

Step 2

- We then **fix α,β,γ as constants and scale up baseline network (B0) (Eq. (3)) with different ϕ to obtain EfficientNet-B1 to B7**

$$\begin{aligned} \max_{d,w,r} \quad & Accuracy\big(\mathcal{N}(d,w,r)\big) \\ s.t. \quad & \mathcal{N}(d,w,r) = \bigodot_{i=1\ldots s} \hat{\mathcal{F}}_i^{d\cdot\hat{L}_i}\big(X_{\langle r\cdot\hat{H}_i, r\cdot\hat{W}_i, w\cdot\hat{C}_i\rangle}\big) \\ & \text{Memory}(\mathcal{N}) \leq \text{target\_memory} \\ & \text{FLOPS}(\mathcal{N}) \leq \text{target\_flops} \end{aligned}$$

$$\begin{aligned} \text{depth: } & d = \alpha^\phi \\ \text{width: } & w = \beta^\phi \\ \text{resolution: } & r = \gamma^\phi \\ \text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned}$$

7 EfficientNet

4. EfficientNet Architecture

- Baseline network : EfficientNet-B0

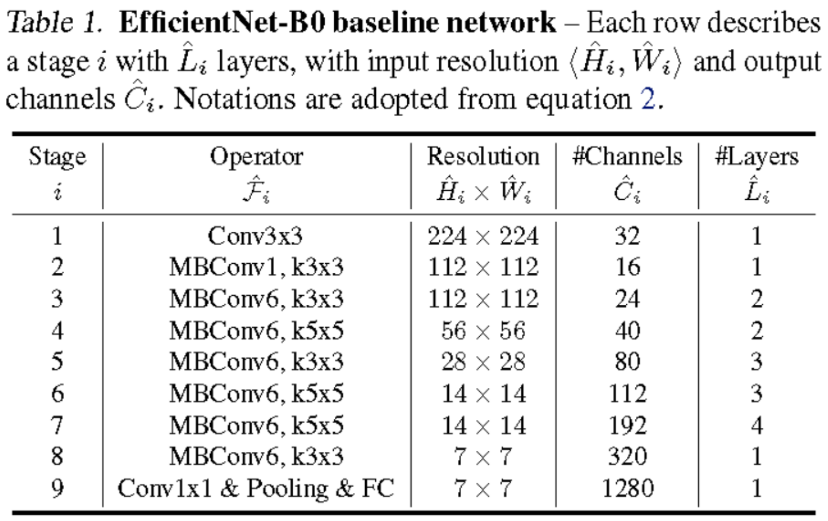
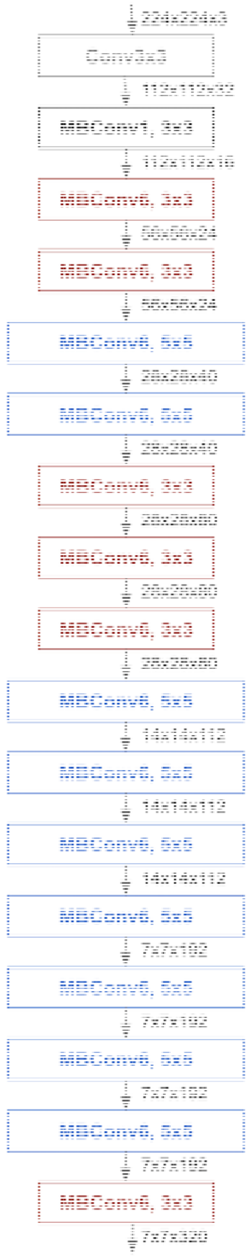


Table 1. **EfficientNet-B0 baseline network** – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $\langle \hat{H}_i, \hat{W}_i \rangle$ and output channels $\hat{C}_i$. Notations are adopted from equation 2.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1



MnasNet과 동일한 search space 사용

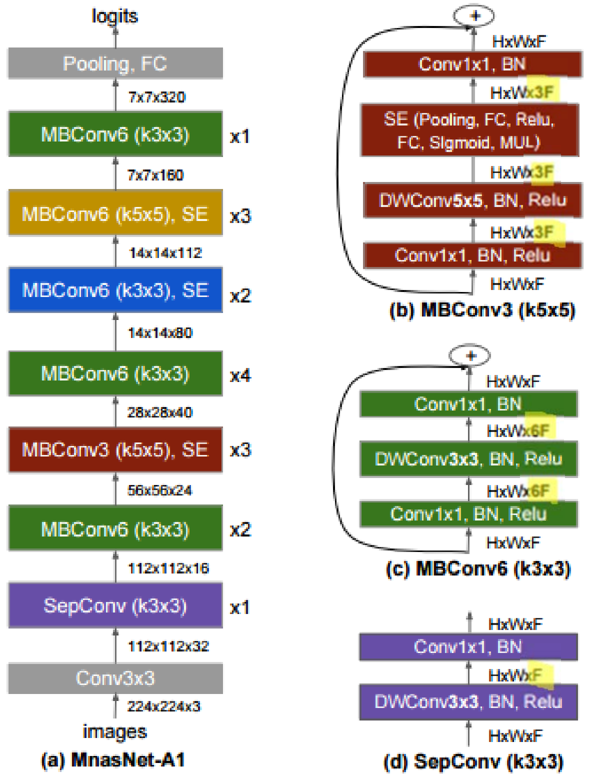
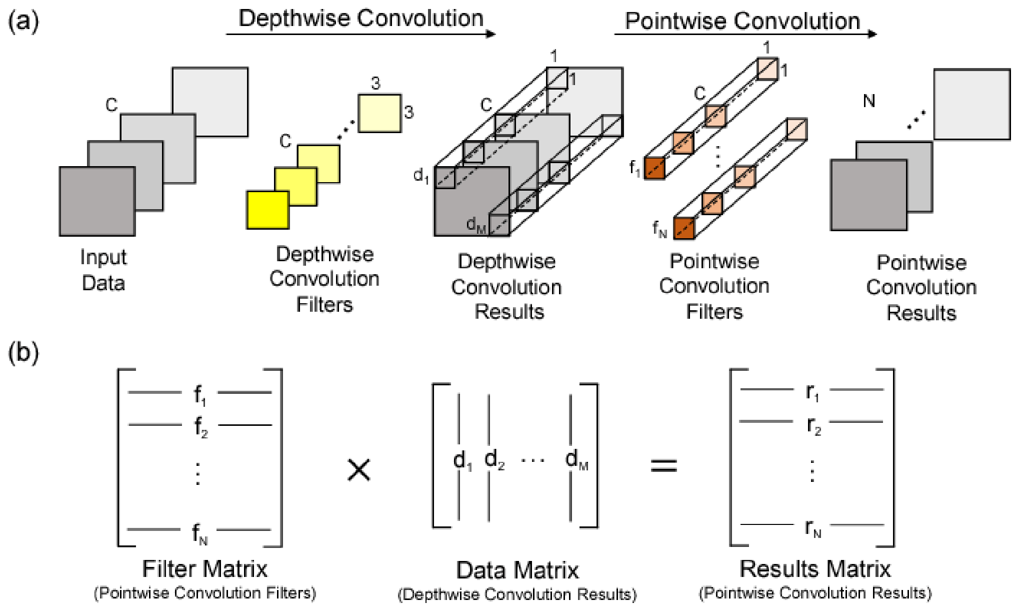
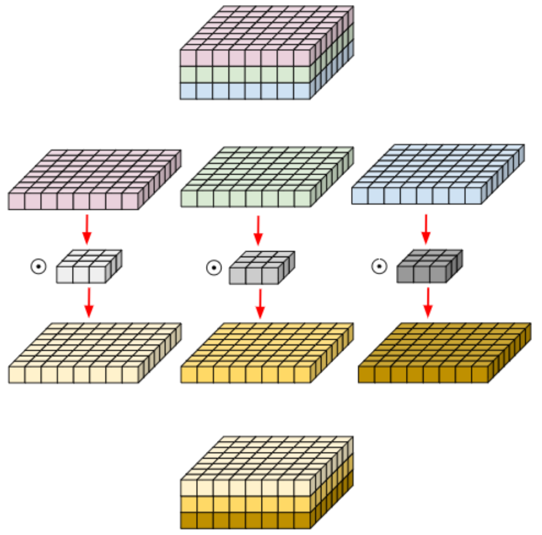


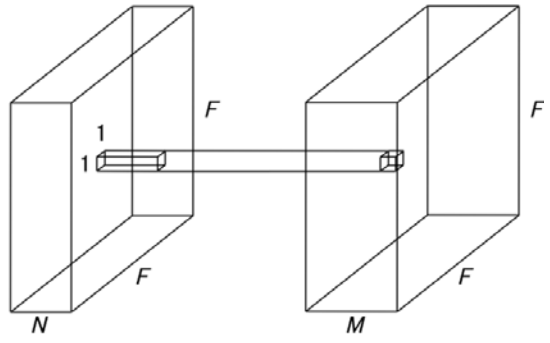
Figure 7: **MnasNet-A1 Architecture** – (a) is a representative model selected from Table 1; (b) - (d) are a few corresponding layer structures. *MBConv* denotes mobile inverted bottleneck conv, *DWConv* denotes depthwise conv, k3x3/k5x5 denotes kernel size, *BN* is batch norm, HxWxF denotes tensor shape (height, width, depth), and $\times 1/2/3/4$ denotes the number of repeated layers within the block.

3 basic steps why it is more efficient

Reviewing EfficientNet: Increasing the Accuracy and Robustness of CNNs
https://heartbeat.fritz.ai/reviewing-efficientnet-increasing-the-accuracy-and-robustness-of-cnns-6aaf411fc81d


(a)
Depthwise Convolution
Pointwise Convolution
C
3
3
1
1
d1
dM
f1
fN
N
Input Data
Depthwise Convolution Filters
Depthwise Convolution Results
Pointwise Convolution Filters
Pointwise Convolution Results
(b)
f1
f2
fN
×
d1 d2 ··· dM
=
r1
r2
rN
Filter Matrix
(Pointwise Convolution Filters)
Data Matrix
(Depthwise Convolution Results)
Results Matrix
(Pointwise Convolution Results)


Depthwise Convolution
(공간 방향의 Convolution)



Pointhwise Convolution (채널 방향의 Convolution - channel reduction)

Depthwise Convolution + Pointwise Convolution: Divides the original convolution into two stages to significantly reduce the cost of calculation, with a minimum loss of accuracy.

Inverse Res: The original **ResNet** blocks consist of a layer that squeezes the channels, then a layer that extends the channels. In this way, it links skip connections to rich channel layers. In **MBConv**, however, blocks consist of a layer that first extends channels and then compresses them, so that layers with fewer channels are skip connected.

Linear bottleneck: Uses linear activation in the last layer in each block to prevent loss of information from ReLU.

```
from keras.layers import Conv2D, DepthwiseConv2D, Adddef
inverted_residual_block(x, expand=64, squeeze=16):
    block = Conv2D(expand, (1,1), activation='relu')(x)
    block = DepthwiseConv2D((3,3), activation='relu')(block)
    block = Conv2D(squeeze, (1,1), activation='relu')(block)
    return Add()([block, x])
```

Mnas & MBCblock

Mobile Neural Architecture Search

- **Factorized Hierarchical Search Space**

CNN 모델을 미리 정의된 블록들로 나누고, 점차적으로 입력 해상도를 줄이고 필터 크기를 늘리는 구조

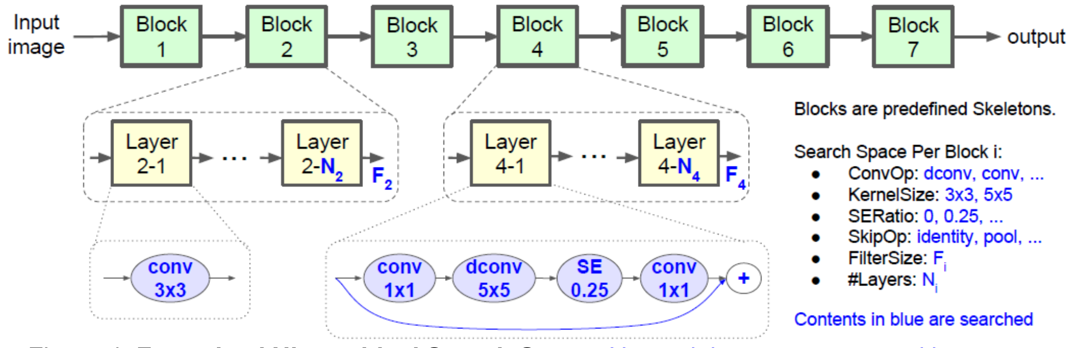


Figure 4: **Factorized Hierarchical Search Space.** Network layers are grouped into a number of predefined skeletons, called *blocks*, based on their input resolutions and filter sizes. Each block contains a variable number of repeated identical layers where only the first layer has stride 2 if input/output resolutions are different but all other layers have stride 1. For each block, we search for the operations and connections for a single layer and the number of layers N, then the same layer is repeated *N* times (e.g., Layer 4-1 to 4-N4 are the same). Layers from different blocks (e.g., Layer 2-1 and 4-1) can be different.

각 블록들은 sub search space를 통해 정의한 동일한 레이어들로 구성되어 있으며 다음과 같은 조건들을 고려하여 sub search space를 결정

- Convolutional ops $ConvOp$: regular conv (conv), depthwise conv (dconv), and mobile inverted bottleneck conv
- Convolutional kernel size $KernelSize$: 3x3, 5x5.
- Squeeze-and-excitation ratio $SERatio$: 0, 0.25.
- Skip ops $SkipOp$: pooling, identity residual, or no skip.
- Output filter size Fi.
- Number of layers per block Ni.

Search Algorithm은 강화학습 사용

Mobile Inverted Bottleneck Convolution

Residual Block　　　Inverted Residual Block

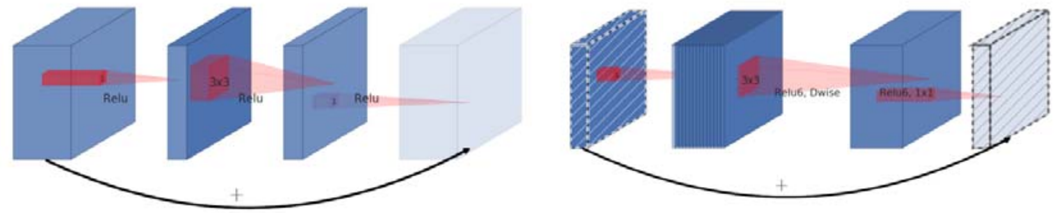
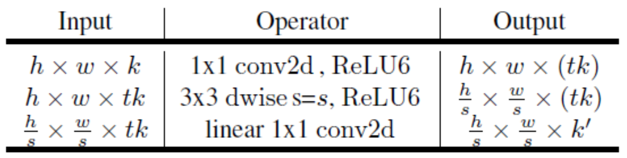


Figure 3: The difference between residual block [8, 30] and inverted residual. Diagonally hatched layers do not use non-linearities. We use thickness of each block to indicate its relative number of channels. Note how classical residuals connects the layers with high number of channels, whereas the inverted residuals connect the bottlenecks.

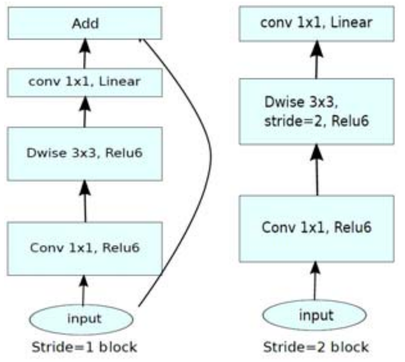
필요한 정보는 narrow layer에서 있기 때문에, skip connection으로 사용해도 필요한 정보를 더 깊은 layer에까지 잘 전달할 것이라는 기대와 narrow layer의 skip connection으로 메모리 사용량 줄임

Bottleneck Residual Block



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.



expansion factor t : 블록 중간에 t만큼 채널이 확장

7 EfficientNet

5.1 Experiment: Scaling Up MobileNets and ResNet

- Apply the scaling method to MobileNet and ResNet

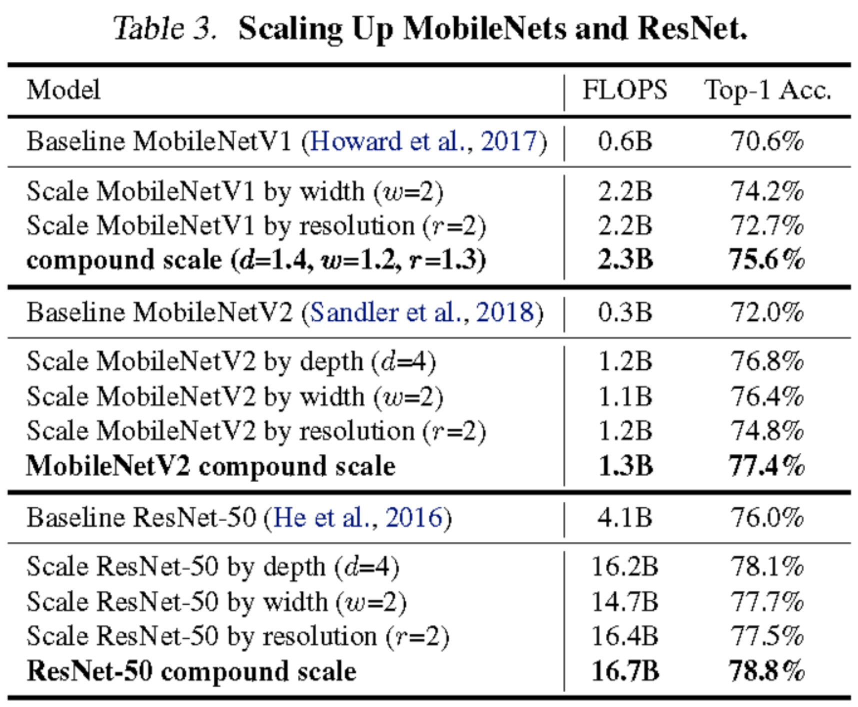
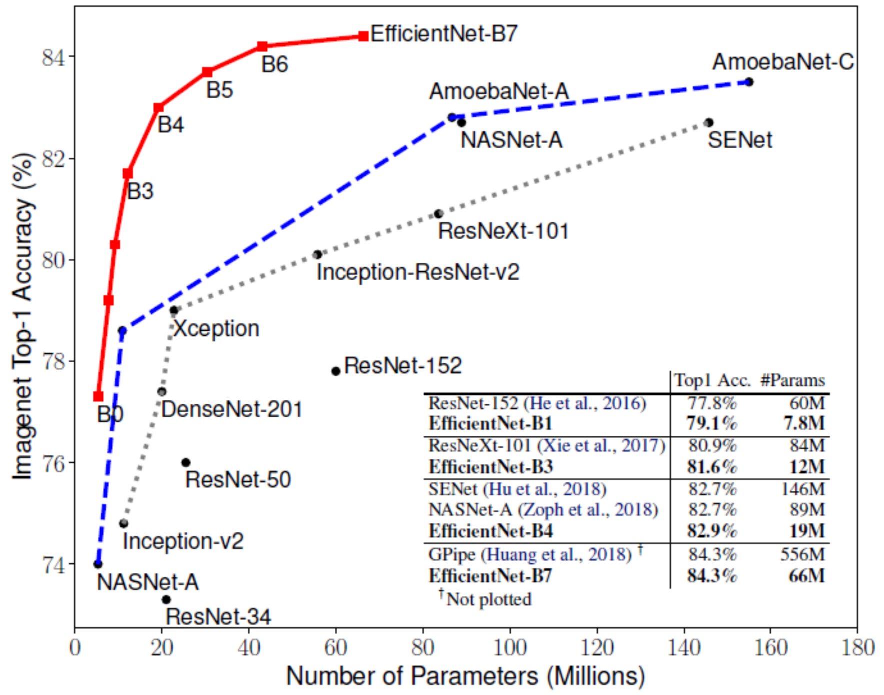


Table 3. **Scaling Up MobileNets and ResNet.**

Model	FLOPS	Top-1 Acc.
Baseline MobileNetV1 (Howard et al., 2017)	0.6B	70.6%
Scale MobileNetV1 by width (w=2)	2.2B	74.2%
Scale MobileNetV1 by resolution (r=2)	2.2B	72.7%
compound scale (d=1.4, w=1.2, r=1.3)	**2.3B**	**75.6%**
Baseline MobileNetV2 (Sandler et al., 2018)	0.3B	72.0%
Scale MobileNetV2 by depth (d=4)	1.2B	76.8%
Scale MobileNetV2 by width (w=2)	1.1B	76.4%
Scale MobileNetV2 by resolution (r=2)	1.2B	74.8%
MobileNetV2 compound scale	**1.3B**	**77.4%**
Baseline ResNet-50 (He et al., 2016)	4.1B	76.0%
Scale ResNet-50 by depth (d=4)	16.2B	78.1%
Scale ResNet-50 by width (w=2)	14.7B	77.7%
Scale ResNet-50 by resolution (r=2)	16.4B	77.5%
ResNet-50 compound scale	**16.7B**	**78.8%**

EfficientNet

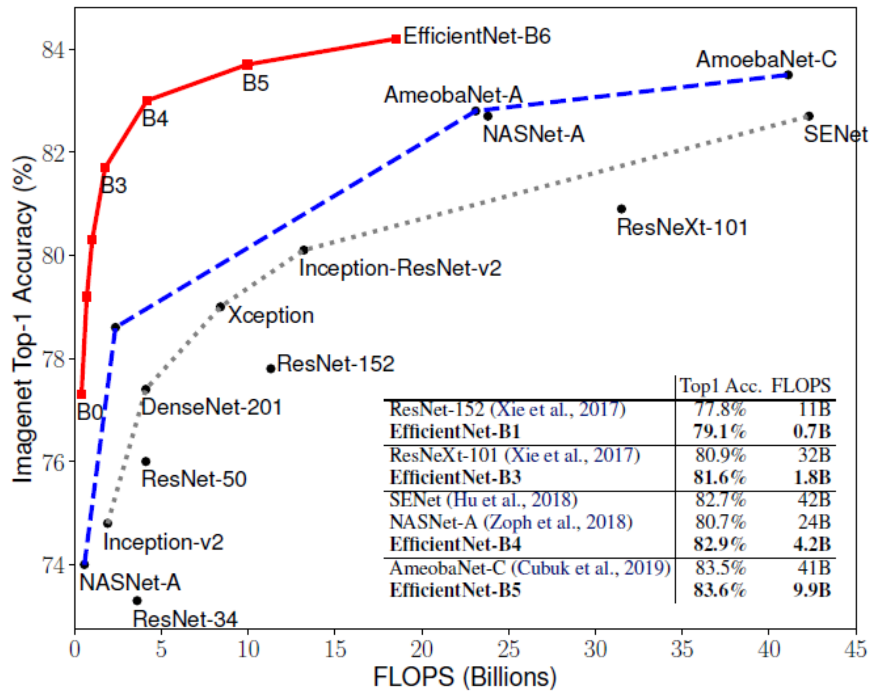
ImageNet Results



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) [†]	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

[†]Not plotted

Figure 1. **Model Size** vs. **ImageNet Accuracy**. All numbers are for single-crop, single-model. In particular, EfficientNet-B7 achieves new state-of-the-art 84.3% top-1 accuracy but being 8.4x smaller and 6.1x faster than GPipe. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152.



	Top1 Acc.	FLOPS
ResNet-152 (Xie et al., 2017)	77.8%	11B
EfficientNet-B1	**79.1%**	**0.7B**
ResNeXt-101 (Xie et al., 2017)	80.9%	32B
EfficientNet-B3	**81.6%**	**1.8B**
SENet (Hu et al., 2018)	82.7%	42B
NASNet-A (Zoph et al., 2018)	80.7%	24B
EfficientNet-B4	**82.9%**	**4.2B**
AmeobaNet-C (Cubuk et al., 2019)	83.5%	41B
EfficientNet-B5	**83.6%**	**9.9B**

Figure 5. **FLOPS** vs. **ImageNet Accuracy** – Similar to Figure 1 except it compares FLOPS rather than model size.

7

EfficientNet

5.2 ImageNet Results for Efficient-B0 to B7

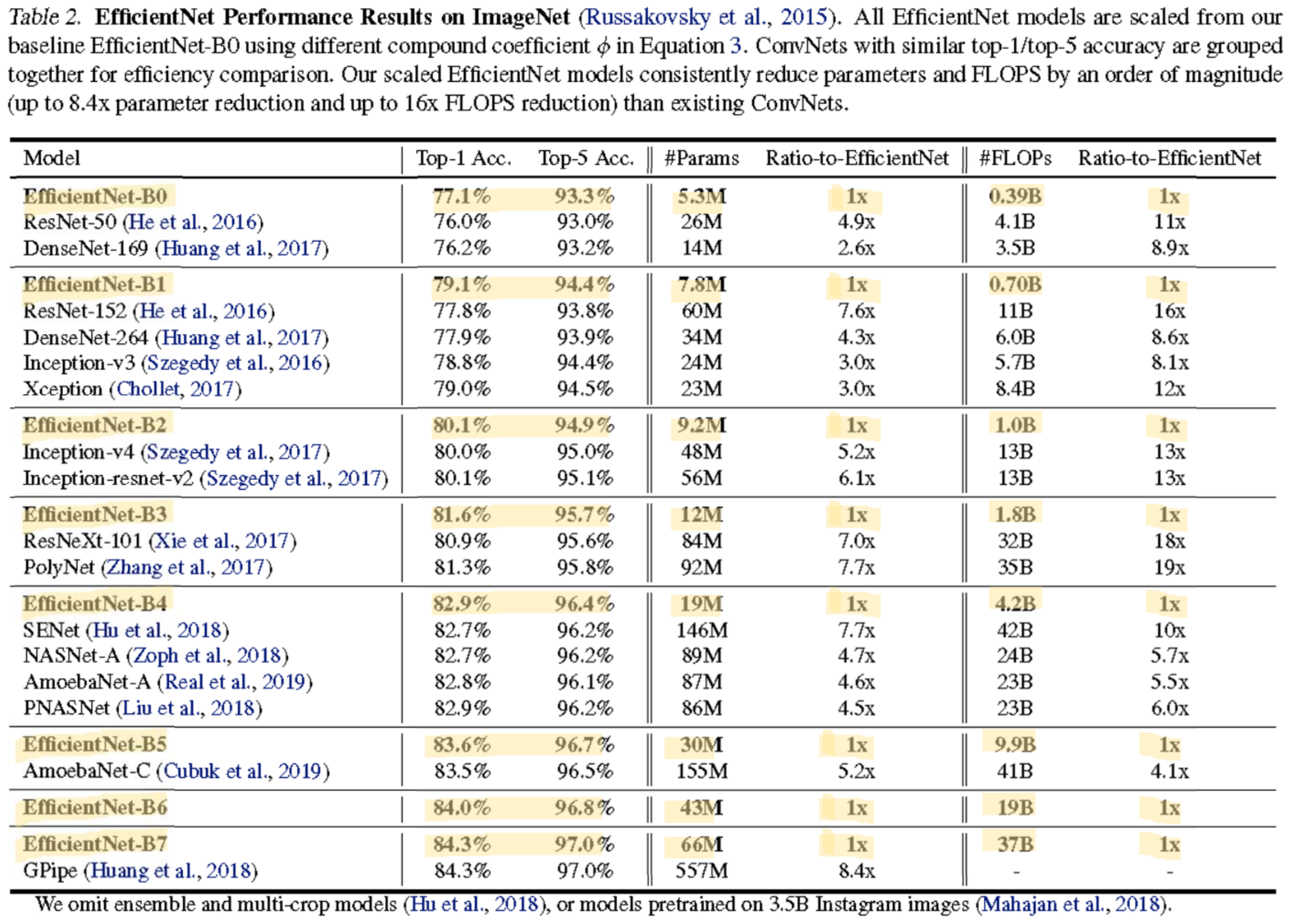


Table 2. **EfficientNet Performance Results on ImageNet** (Russakovsky et al., 2015). All EfficientNet models are scaled from our baseline EfficientNet-B0 using different compound coefficient ϕ in Equation 3. ConvNets with similar top-1/top-5 accuracy are grouped together for efficiency comparison. Our scaled EfficientNet models consistently reduce parameters and FLOPS by an order of magnitude (up to 8.4x parameter reduction and up to 16x FLOPS reduction) than existing ConvNets.

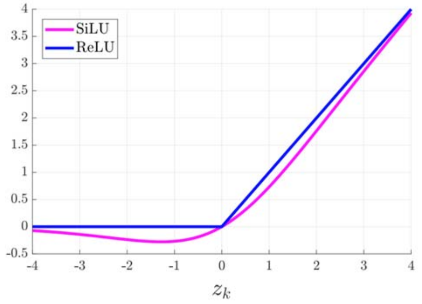
Model	Top-1 Acc.	Top-5 Acc.	#Params	Ratio-to-EfficientNet	#FLOPs	Ratio-to-EfficientNet
EfficientNet-B0	77.1%	93.3%	5.3M	1x	0.39B	1x
ResNet-50 (He et al., 2016)	76.0%	93.0%	26M	4.9x	4.1B	11x
DenseNet-169 (Huang et al., 2017)	76.2%	93.2%	14M	2.6x	3.5B	8.9x
EfficientNet-B1	79.1%	94.4%	7.8M	1x	0.70B	1x
ResNet-152 (He et al., 2016)	77.8%	93.8%	60M	7.6x	11B	16x
DenseNet-264 (Huang et al., 2017)	77.9%	93.9%	34M	4.3x	6.0B	8.6x
Inception-v3 (Szegedy et al., 2016)	78.8%	94.4%	24M	3.0x	5.7B	8.1x
Xception (Chollet, 2017)	79.0%	94.5%	23M	3.0x	8.4B	12x
EfficientNet-B2	80.1%	94.9%	9.2M	1x	1.0B	1x
Inception-v4 (Szegedy et al., 2017)	80.0%	95.0%	48M	5.2x	13B	13x
Inception-resnet-v2 (Szegedy et al., 2017)	80.1%	95.1%	56M	6.1x	13B	13x
EfficientNet-B3	81.6%	95.7%	12M	1x	1.8B	1x
ResNeXt-101 (Xie et al., 2017)	80.9%	95.6%	84M	7.0x	32B	18x
PolyNet (Zhang et al., 2017)	81.3%	95.8%	92M	7.7x	35B	19x
EfficientNet-B4	82.9%	96.4%	19M	1x	4.2B	1x
SENet (Hu et al., 2018)	82.7%	96.2%	146M	7.7x	42B	10x
NASNet-A (Zoph et al., 2018)	82.7%	96.2%	89M	4.7x	24B	5.7x
AmoebaNet-A (Real et al., 2019)	82.8%	96.1%	87M	4.6x	23B	5.5x
PNASNet (Liu et al., 2018)	82.9%	96.2%	86M	4.5x	23B	6.0x
EfficientNet-B5	83.6%	96.7%	30M	1x	9.9B	1x
AmoebaNet-C (Cubuk et al., 2019)	83.5%	96.5%	155M	5.2x	41B	4.1x
EfficientNet-B6	84.0%	96.8%	43M	1x	19B	1x
EfficientNet-B7	84.3%	97.0%	66M	1x	37B	1x
GPipe (Huang et al., 2018)	84.3%	97.0%	557M	8.4x	-	-

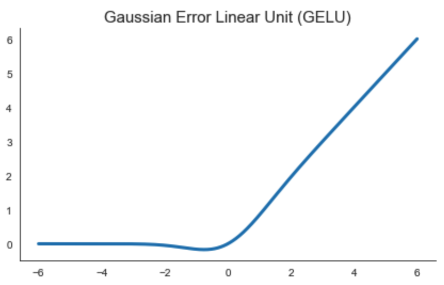
We omit ensemble and multi-crop models (Hu et al., 2018), or models pretrained on 3.5B Instagram images (Mahajan et al., 2018).

❖ Simulation Settings

- RMSProp optimizer with decay 0.9 and momentum 0.9; batch norm momentum 0.99; weight decay 1e-5; initial learning rate 0.256 that decays by 0.97 every 2.4 epochs
- SiLU(Swish-1) activation, AutoAugment, and stochastic depth with survival probability 0.8
- Dropout ratio from 0.2 for EffcientNet-B0 to 0.5 for B7
- Reserve 25K randomly picked images from the *training* set as a *minival* set, and perform early stopping on this *minival*
- Then evaluate the early-stopped checkpoint on the original *validation* set to report the final validation accuracy.

BERT, GPT, ViT – GeLU 사용



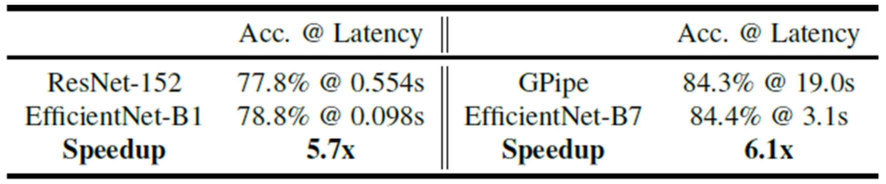


Modern Activation : https://towardsdatascience.com/activation-functions-you-might-have-missed-79d72fc080a5

7 EfficientNet

5.2 Inference Latency Comparison

Table 4. **Inference Latency Comparison** – Latency is measured with batch size 1 on a single core of Intel Xeon CPU E5-2690.

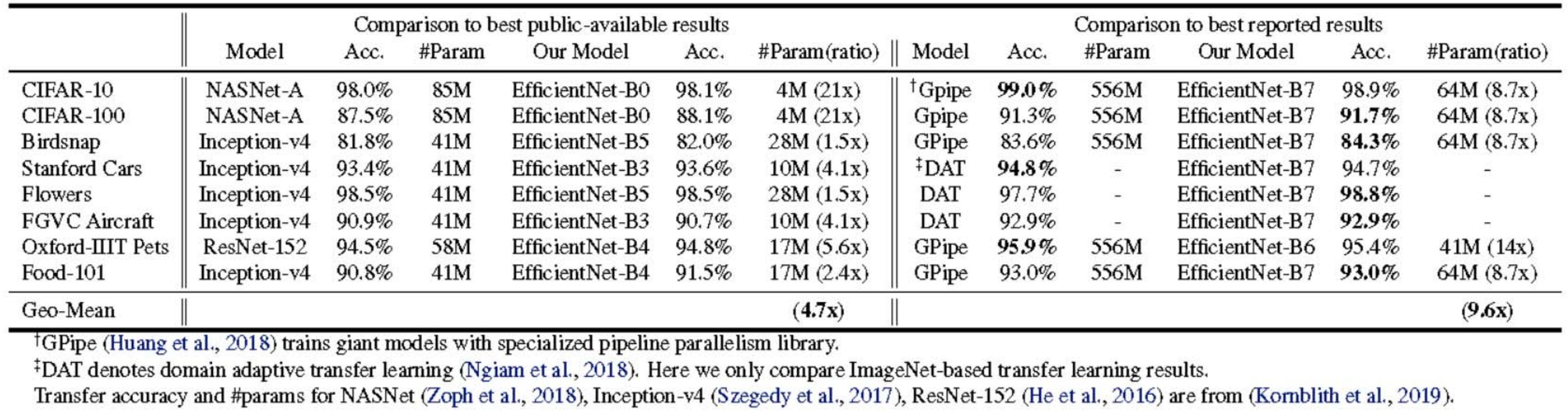


	Acc. @ Latency		Acc. @ Latency
ResNet-152	77.8% @ 0.554s	GPipe	84.3% @ 19.0s
EfficientNet-B1	78.8% @ 0.098s	EfficientNet-B7	84.4% @ 3.1s
Speedup	**5.7x**	**Speedup**	**6.1x**

7 EfficientNet

5.3 Experiment: Transfer Learning Results

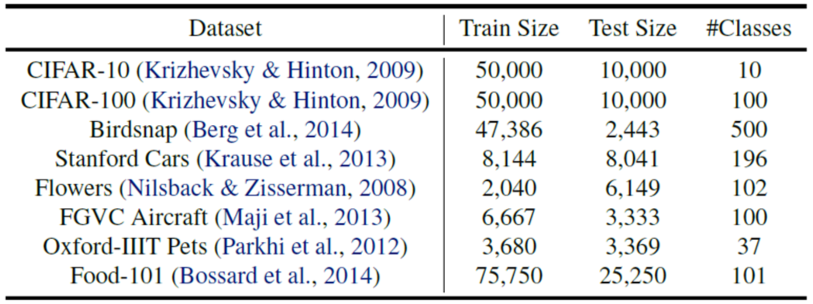
Table 5. **EfficientNet Performance Results on Transfer Learning Datasets**. Our scaled EfficientNet models achieve new state-of-the-art accuracy for 5 out of 8 datasets, with 9.6x fewer parameters on average.



	Comparison to best public-available results						Comparison to best reported results					
	Model	Acc.	#Param	Our Model	Acc.	#Param(ratio)	Model	Acc.	#Param	Our Model	Acc.	#Param(ratio)
CIFAR-10	NASNet-A	98.0%	85M	EfficientNet-B0	98.1%	4M (21x)	†Gpipe	**99.0%**	556M	EfficientNet-B7	98.9%	64M (8.7x)
CIFAR-100	NASNet-A	87.5%	85M	EfficientNet-B0	88.1%	4M (21x)	Gpipe	91.3%	556M	EfficientNet-B7	**91.7%**	64M (8.7x)
Birdsnap	Inception-v4	81.8%	41M	EfficientNet-B5	82.0%	28M (1.5x)	GPipe	83.6%	556M	EfficientNet-B7	**84.3%**	64M (8.7x)
Stanford Cars	Inception-v4	93.4%	41M	EfficientNet-B3	93.6%	10M (4.1x)	‡DAT	**94.8%**	-	EfficientNet-B7	94.7%	-
Flowers	Inception-v4	98.5%	41M	EfficientNet-B5	98.5%	28M (1.5x)	DAT	97.7%	-	EfficientNet-B7	**98.8%**	-
FGVC Aircraft	Inception-v4	90.9%	41M	EfficientNet-B3	90.7%	10M (4.1x)	DAT	92.9%	-	EfficientNet-B7	**92.9%**	-
Oxford-IIIT Pets	ResNet-152	94.5%	58M	EfficientNet-B4	94.8%	17M (5.6x)	GPipe	**95.9%**	556M	EfficientNet-B6	95.4%	41M (14x)
Food-101	Inception-v4	90.8%	41M	EfficientNet-B4	91.5%	17M (2.4x)	GPipe	93.0%	556M	EfficientNet-B7	**93.0%**	64M (8.7x)
Geo-Mean						**(4.7x)**						**(9.6x)**

†GPipe (Huang et al., 2018) trains giant models with specialized pipeline parallelism library.
‡DAT denotes domain adaptive transfer learning (Ngiam et al., 2018). Here we only compare ImageNet-based transfer learning results.
Transfer accuracy and #params for NASNet (Zoph et al., 2018), Inception-v4 (Szegedy et al., 2017), ResNet-152 (He et al., 2016) are from (Kornblith et al., 2019).

Table 6. **Transfer Learning Datasets**.



Dataset	Train Size	Test Size	#Classes
CIFAR-10 (Krizhevsky & Hinton, 2009)	50,000	10,000	10
CIFAR-100 (Krizhevsky & Hinton, 2009)	50,000	10,000	100
Birdsnap (Berg et al., 2014)	47,386	2,443	500
Stanford Cars (Krause et al., 2013)	8,144	8,041	196
Flowers (Nilsback & Zisserman, 2008)	2,040	6,149	102
FGVC Aircraft (Maji et al., 2013)	6,667	3,333	100
Oxford-IIIT Pets (Parkhi et al., 2012)	3,680	3,369	37
Food-101 (Bossard et al., 2014)	75,750	25,250	101

EfficientNet

5.3 Experiment: Transfer Learning Results

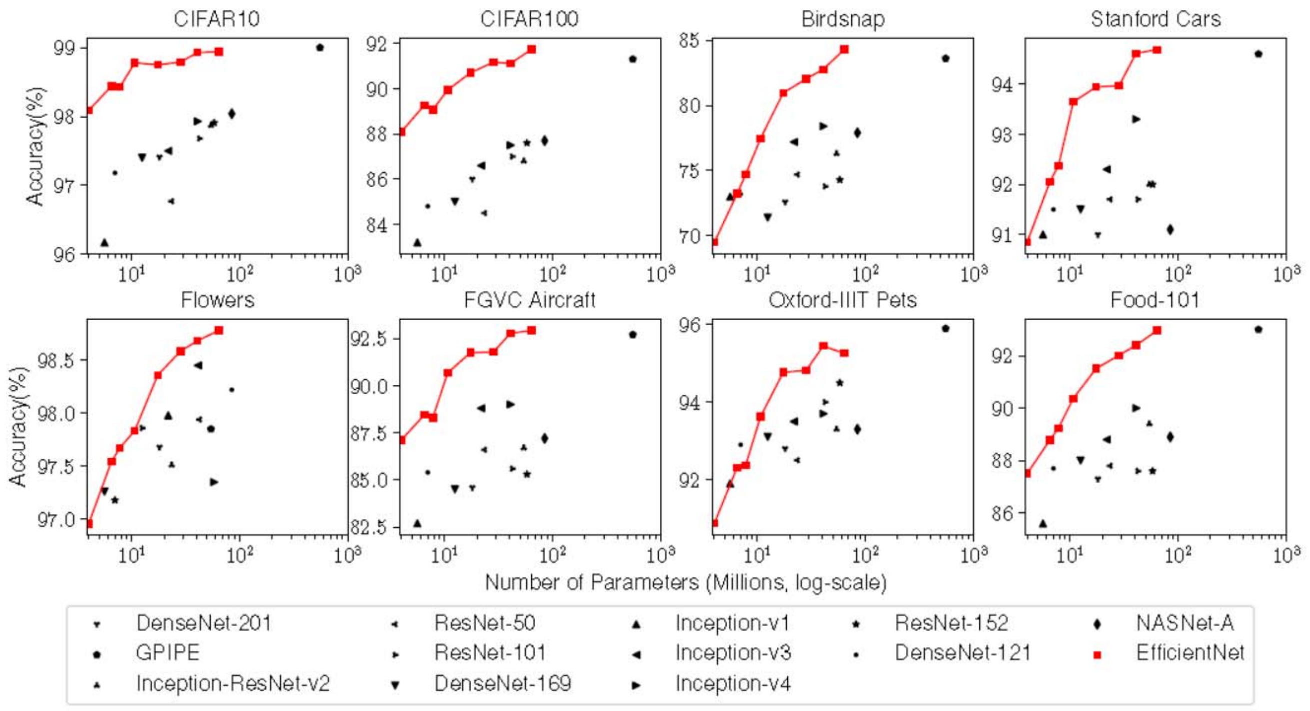


Figure 6. **Model Parameters vs. Transfer Learning Accuracy** – All models are pretrained on ImageNet and finetuned on new datasets.

7 EfficientNet

6. Discussion

- Compare the ImageNet performance of different scaling method for the same EfficientNet-B0 baseline network

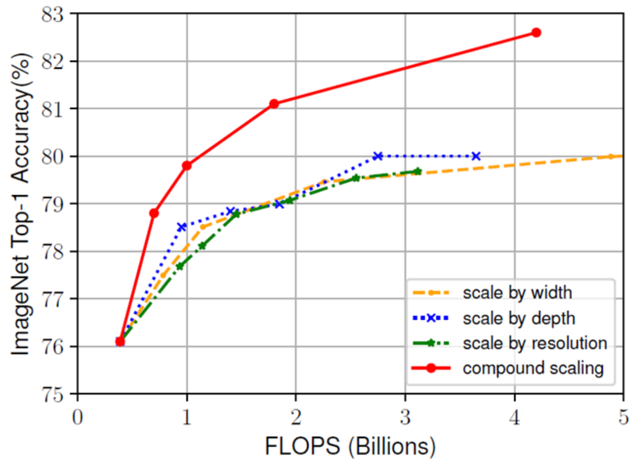


Figure 8. **Scaling Up EfficientNet-B0 with Different Methods.**

Table 7. **Scaled Models Used in Figure 7.**

Model	FLOPS	Top-1 Acc.
Baseline model (EfficientNet-B0)	0.4B	77.3%
Scale model by depth (d=4)	1.8B	79.0%
Scale model by width (w=2)	1.8B	78.9%
Scale model by resolution (r=2)	1.9B	79.1%
Compound Scale (d=1.4, w=1.2, r=1.3)	1.8B	81.1%

➢ The model with compound scaling tends to focus on more relevant regions with more object details than other models

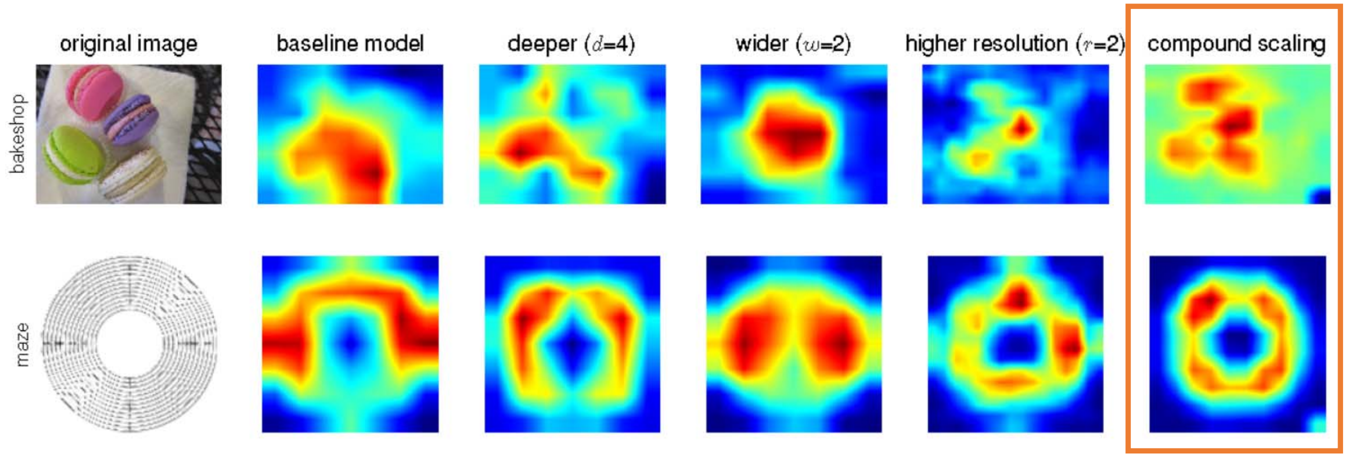


Figure 7. **Class Activation Map (CAM)** (Zhou et al., 2016) for Models with different scaling methods- Our compound scaling method allows the scaled model (last column) to focus on more relevant regions with more object details. Model details are in Table 7.

7

EfficientNet

EffieicntNet Pytorch

https://github.com/lukemelas/EfficientNet-PyTorch

Colab : https://deep-learning-study.tistory.com/563

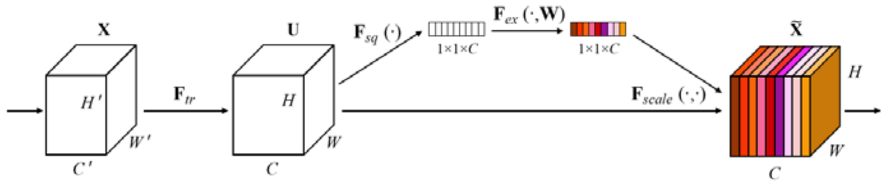
```
cd drive/MyDrive/Colab Notebooks/data
from google.colab import drive
drive.mount(/content/drive')

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary
from torch import optim

# dataset and transformation
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision import models
import os

# display images
from torchvision import utils
import matplotlib.pyplot as plt
%matplotlib inline

# utils
import numpy as np
from torchsummary import summary
import time
import copy
```

torchvision 패키지 STL10 dataset 이용
(10개 라벨, train dataset 5000개, test dataset 8000개로 구성됩니다)

```
# specify path to data
path2data = '/content/efficientnet/MyDrive/data'

# if not exists the path, make the directory
if not os.path.exists(path2data):
    os.mkdir(path2data)

# load dataset
train_ds = datasets.STL10(path2data, split='train', download=True,
transform=transforms.ToTensor())
val_ds = datasets.STL10(path2data, split='test', download=True,
transform=transforms.ToTensor())

print(len(train_ds))
print(len(val_ds))

# define transformation
transformation = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Resize(224)
])

# apply transformation to dataset
train_ds.transform = transformation
val_ds.transform = transformation

# make dataloade
train_dl = DataLoader(train_ds, batch_size=32, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=32, shuffle=True)
```

7 EfficientNet

모델 구축 코드출처:
[1] https://github.com/zsef123/EfficientNets-PyTorch/blob/master/models/effnet.py
[2] https://github.com/katsura-jp/efficientnet-pytorch/blob/master/model/efficientnet.py



Swish activation function

```
# Swish activation function
class Swish(nn.Module):
    def __init__(self):
        super().__init__()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return x * self.sigmoid(x)

# check
if __name__ == '__main__':
    x = torch.randn(3, 3, 224, 224)
    model = Swish()
    output = model(x)
    print('output size:', output.size())
```

SE block (squeeze-and-excitation optimization)

```
# SE Block
class SEBlock(nn.Module):
    def __init__(self, in_channels, r=4):
        super().__init__()

        self.squeeze = nn.AdaptiveAvgPool2d((1,1))
        self.excitation = nn.Sequential(
            nn.Linear(in_channels, in_channels * r),
            Swish(),
            nn.Linear(in_channels * r, in_channels),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.squeeze(x)                          # check :3x56x1x1 (17x17x -> 1x1)
        x = x.view(x.size(0), -1)                    # check : 3x56
        x = self.excitation(x)                       # check : 3x56
        x = x.view(x.size(0), x.size(1), 1, 1)       # check : 3x56x1x1
        return x

# check
if __name__ == '__main__':
    x = torch.randn(3, 56, 17, 17)
    model = SEBlock(x.size(1))                       # in_channels = 56 (x.size(1))
    output = model(x)
    print('output size:', output.size())
)
```

$$z_c = F_{sq}(u_c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j)$$

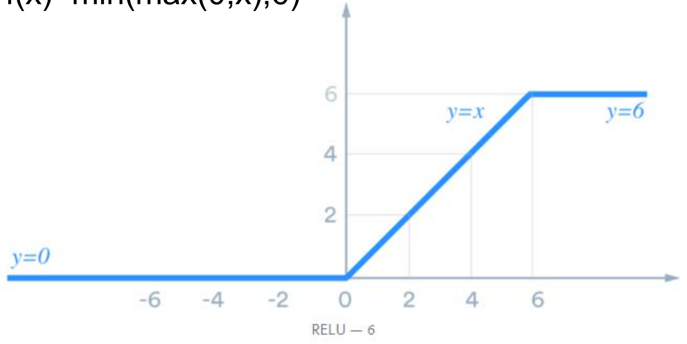
$$s = F_{ex}(z, W) = \sigma(W_2 \delta(W_1 z))$$

Activation Ft.

[출처] http://ai-hub.kr/post/111/

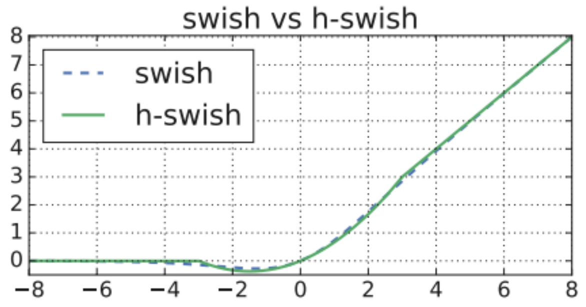
ReLU6

f(x)=min(max(0,x),6)



activation이 폭발적으로 증가하는 것을 막아 일반 ReLU에서 발생하는 문제를 막을 수 있음.

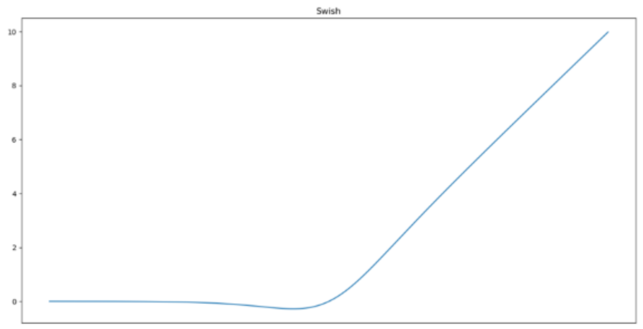
H-Swish



가장 좋은 점은 swish와 거의 비슷하지만 sigmoid를 ReLU로 대체하기 때문에 계산비용이 저렴함

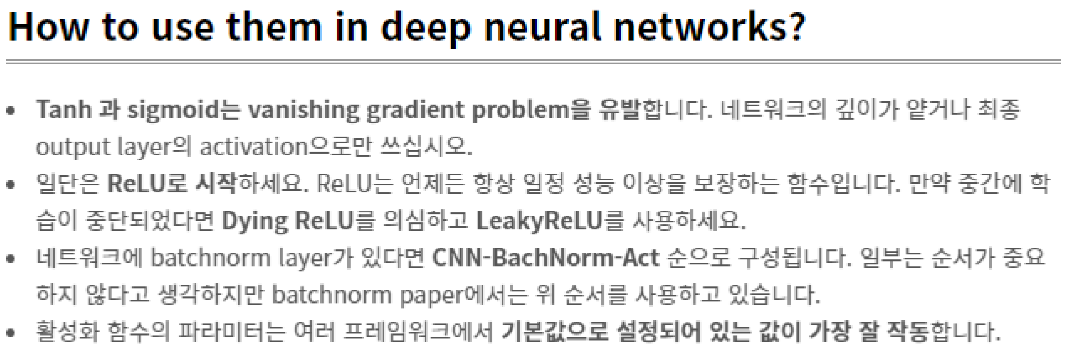
Swish

f(x)=x * sigmoid(x)



Google Brain Team에서 제안한 activation function으로 꽤 많은 양의 challenging dataset에 대해서 ReLU보다 성능이 더 좋음을 확인
ReLU와 유사하게 생겨 ReLU를 Swish unit으로 대체하기 쉬우나 계산비용이 많이 든다는 단점이 있음.

$$
\begin{aligned}
f(x) &= x * sigmoid(x) \\
&= x * (1 + e^{-x})^{-1}
\end{aligned}
$$



How to use them in deep neural networks?

- **Tanh 과 sigmoid는 vanishing gradient problem을 유발**합니다. 네트워크의 깊이가 얕거나 최종 output layer의 activation으로만 쓰십시오.
- 일단은 **ReLU로 시작**하세요. ReLU는 언제든 항상 일정 성능 이상을 보장하는 함수입니다. 만약 중간에 학습이 중단되었다면 **Dying ReLU**를 의심하고 **LeakyReLU**를 사용하세요.
- 네트워크에 batchnorm layer가 있다면 **CNN-BachNorm-Act** 순으로 구성됩니다. 일부는 순서가 중요하지 않다고 생각하지만 batchnorm paper에서는 위 순서를 사용하고 있습니다.
- 활성화 함수의 파라미터는 여러 프레임워크에서 **기본값으로 설정되어 있는 값이 가장 잘 작동**합니다.

SENet(Squeeze and excitation networks)

[출처] https://jayhey.github.io/deep%20learning/2018/07/18/SENet/

Jie Hu, Li Shen, Samuel Albanie, Gang Sun, Enhua Wu, "Squeeze-and-Excitation Networks," https://arxiv.org/abs/1709.01507, 2017

Squeeze-and-Excitation Blocks

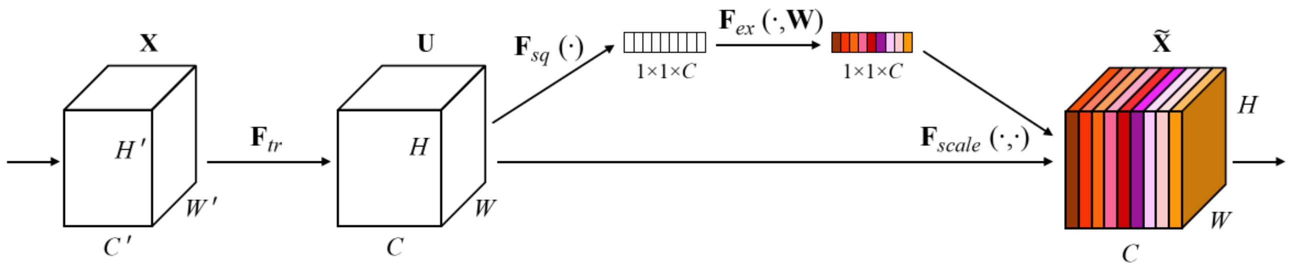
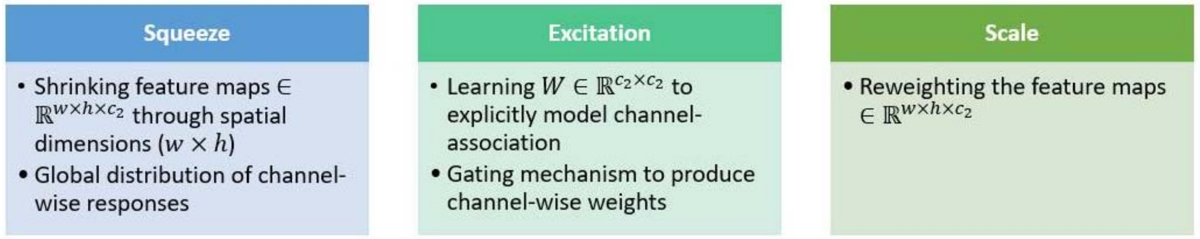


Figure 1: A Squeeze-and-Excitation block.



Squeeze
- Shrinking feature maps $\in \mathbb{R}^{w\times h\times c_2}$ through spatial dimensions $(w \times h)$
- Global distribution of channel-wise responses

Excitation
- Learning $W \in \mathbb{R}^{c_2\times c_2}$ to explicitly model channel-association
- Gating mechanism to produce channel-wise weights

Scale
- Reweighting the feature maps $\in \mathbb{R}^{w\times h\times c_2}$

- Our goal is to improve the representational power of a network by explicitly modelling the interdependencies between the channels of its convolutional features.
- SE block : 각 feature map에 대한 전체 정보를 요약하는 Squeeze operation, 이를 통해 각 feature map의 중요도를 스케일해주는 excitation operation

1) Squeeze : Global Information Embedding

✓ GAP(Global average pooling) 사용하여 중요 정보 추출

$$z_c = F_{sq}(u_c) = \frac{1}{H \times W}\sum_{i=1}^{H}\sum_{j=1}^{W} u_c(i,j)$$

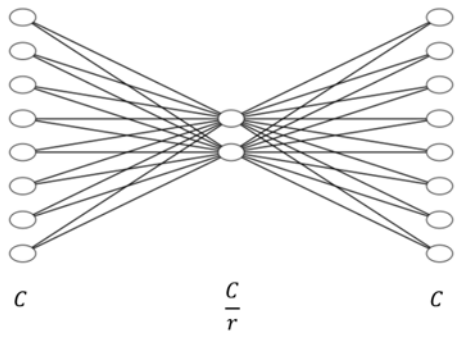
2) Excitation : Adaptive Recalibration

✓ 채널 간 의존성(channel-wise dependencies)을 계산하게 됨. 논문에서는 Fully connected layer와 비선형 함수를 조절하는 것으로 간단하게 이를 계산함

$$s = F_{ex}(z, W) = \sigma(W_2\delta(W_1 z))$$

σ는 ReLU 함수, W_1과 W_2는 각각 Fully connected layer

Reduction ratio r 을 통해서 W1의 노드 수를 줄인다는 것임. 그리고 W2에서 다시 피쳐맵의 수 C만큼 증가시킴. 만약 feature map의 수가 C개(그림에선 8개)이고 r값이 4라면 아래 그림처럼 FC layer가 생기게 됨



Deep Networks with Stochastic Depth

[출처] https://towardsdatascience.com/review-stochastic-depth-image-classification-a4e225807f4a

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, Kilian Weinberger, "Deep Networks with Stochastic Depth," https://arxiv.org/abs/1603.09382, 2016

Stochastic Depth

- Deep Network : vanishing gradient, diminishing feature (Forward 시 여러 번 multiplication, convolution computation 을 반복하면서 Feature가 손실)
- By using Stochastic Depth, the network is shorten during training, i.e. a subset of layers is randomly dropped and bypass them with the identity function. And a full network is used during testing/inference. By this mean:

1) Training time is reduced substantially
2) Test error is improved significantly as well
   - ✓ depth를 줄여서 training 중의 forward propagation 및 gradient computation의 chain을 감소시킴.
   - ✓ stochastic depth의 경우, implicit ensemble로 볼 수 있음. (network에 대해 다른 depth를 사용)

- Bernoulli random variable $b_l \sim Bernoulli(p_l) \in \{0,1\}$ 에 의해 정해짐. 여기에서 만약 l번째 레이어의 $b_l = 1$ active, 아니면 inactive 함.
- layer l 이 "survival" 하는 probability $p_l = \Pr(b_l{=}1) = 1 - \frac{l}{L}(1 - p_L)$
- $p_0 = 1$, 마지막 레이어 L의 $p_L = 0.5$

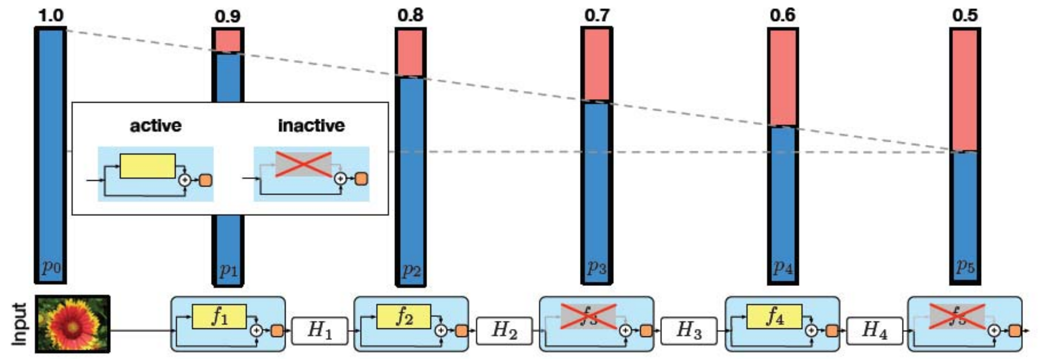
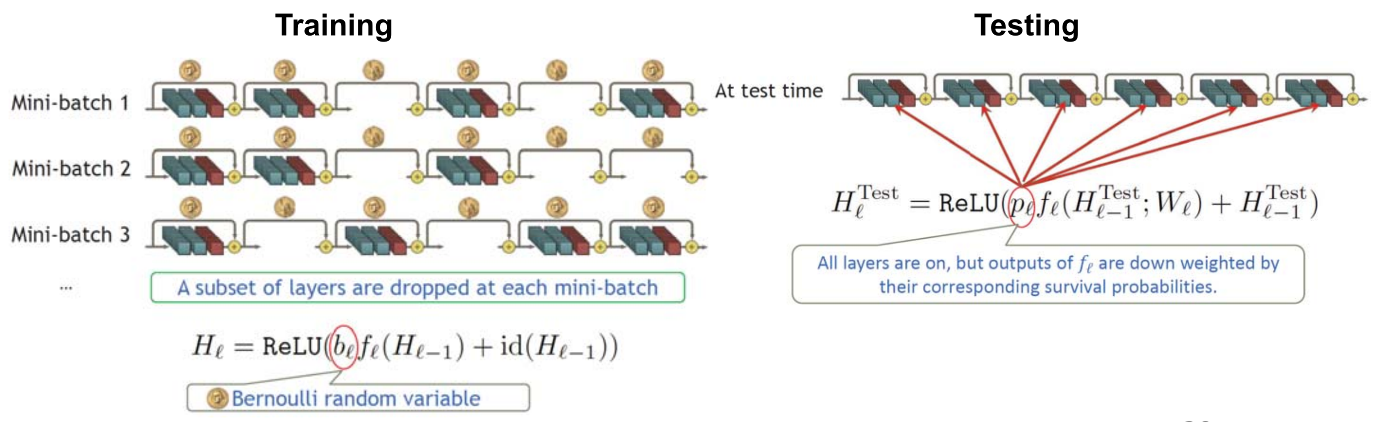


Fig. 2. The linear decay of p_ℓ illustrated on a ResNet with stochastic depth for $p_0 = 1$ and $p_L = 0.5$. Conceptually, we treat the input to the first ResBlock as H_0, which is always active.


Training
Testing
Mini-batch 1
Mini-batch 2
Mini-batch 3
...
A subset of layers are dropped at each mini-batch
$H_\ell = \mathrm{ReLU}(b_\ell f_\ell(H_{\ell-1}) + \mathrm{id}(H_{\ell-1}))$
Bernoulli random variable
At test time
$H_\ell^{\mathrm{Test}} = \mathrm{ReLU}(p_\ell f_\ell(H_{\ell-1}^{\mathrm{Test}}; W_\ell) + H_{\ell-1}^{\mathrm{Test}})$
All layers are on, but outputs of f_ℓ are down weighted by their corresponding survival probabilities.

7 EfficientNet

모델 구축

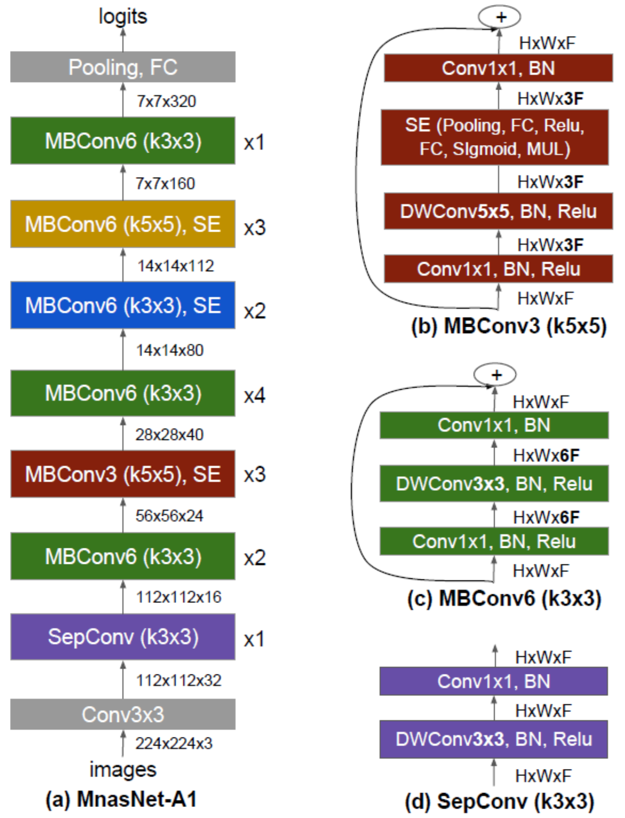


Figure 7: **MnasNet-A1 Architecture** – (a) is a representative model selected from Table 1; (b) - (d) are a few corresponding layer structures. *MBConv* denotes mobile inverted bottleneck conv, *DWConv* denotes depthwise conv, k3x3/k5x5 denotes kernel size, *BN* is batch norm, HxWxF denotes tensor shape (height, width, depth), and ×1/2/3/4 denotes the number of repeated layers within the block.

MBConv 클래스 정의 : 학습시 stochastic depth, expand=6, 3개 레이어

```
class MBConv(nn.Module):
    expand = 6
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, se_scale=4, p=0.5):
        super().__init__()
        # first MBConv is not using stochastic depth
        self.p = torch.tensor(p).float() if (in_channels == out_channels) else torch.tensor(1).float()

        self.residual = nn.Sequential(
            nn.Conv2d(in_channels, in_channels * MBConv.expand, 1, stride=stride, padding=0, bias=False),
            nn.BatchNorm2d(in_channels * MBConv.expand, momentum=0.99, eps=1e-3),
            Swish(),
            nn.Conv2d(in_channels * MBConv.expand, in_channels * MBConv.expand, kernel_size=kernel_size,
                      stride=1, padding=kernel_size//2, bias=False, groups=in_channels*MBConv.expand),
            nn.BatchNorm2d(in_channels * MBConv.expand, momentum=0.99, eps=1e-3),
            Swish()
        )

        self.se = SEBlock(in_channels * MBConv.expand, se_scale)

        self.project = nn.Sequential(
            nn.Conv2d(in_channels*MBConv.expand, out_channels, kernel_size=1, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(out_channels, momentum=0.99, eps=1e-3)
        )

        self.shortcut = (stride == 1) and (in_channels == out_channels)
```

Expansion & Depthwise Convolution

EfficientNet

MBConv 클래스 정의

```
    def forward(self, x):
        # stochastic depth
        if self.training:
            if not torch.bernoulli(self.p):
                return x

        x_shortcut = x
        x_residual = self.residual(x)
        x_se = self.se(x_residual)

        x = x_se * x_residual
        x = self.project(x)

        if self.shortcut:
            x= x_shortcut + x

        return x

# check
if __name__ == '__main__':
    x = torch.randn(3, 16, 24, 24)
    model = MBConv(x.size(1), x.size(1), 3, stride=1, p=1)
    model.train()
    output = model(x)
    x = (output == x)
    print('output size:', output.size(), 'Stochastic depth:', x[1,0,0,0])
```

SepConv 클래스 정의 : expand=1, 2개 layer

```
class SepConv(nn.Module):
    expand = 1
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, se_scale=4, p=0.5):
        super().__init__()
        # first SepConv is not using stochastic depth
        self.p = torch.tensor(p).float() if (in_channels == out_channels) else torch.tensor(1).float()

        self.residual = nn.Sequential(
            nn.Conv2d(in_channels * SepConv.expand, in_channels * SepConv.expand, kernel_size=kernel_size,
                      stride=1, padding=kernel_size//2, bias=False, groups=in_channels*SepConv.expand),
            nn.BatchNorm2d(in_channels * SepConv.expand, momentum=0.99, eps=1e-3),
            Swish()
        )

        self.se = SEBlock(in_channels * SepConv.expand, se_scale)

        self.project = nn.Sequential(
            nn.Conv2d(in_channels*SepConv.expand, out_channels, kernel_size=1, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(out_channels, momentum=0.99, eps=1e-3)
        )

        self.shortcut = (stride == 1) and (in_channels == out_channels)
```

7 EfficientNet

SepConv 클래스 정의

```
    def forward(self, x):
        # stochastic depth
        if self.training:
            if not torch.bernoulli(self.p):
                return x

        x_shortcut = x
        x_residual = self.residual(x)
        x_se = self.se(x_residual)

        x = x_se * x_residual
        x = self.project(x)

        if self.shortcut:
            x= x_shortcut + x

        return x

# check
if __name__ == '__main__':
    x = torch.randn(3, 16, 24, 24)
    model = SepConv(x.size(1), x.size(1), 3, stride=1, p=1)
    model.train()
    output = model(x)
    # stochastic depth check
    x = (output == x)
    print('output size:', output.size(), 'Stochastic depth:', x[1,0,0,0])
```

EfficientNet 정의

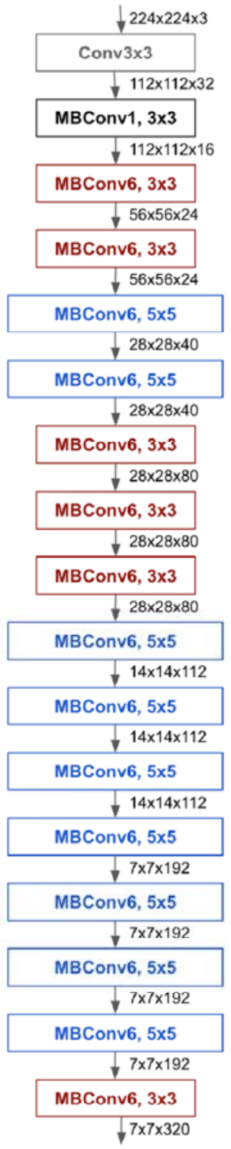
```
class EfficientNet(nn.Module):
    def __init__(self, num_classes=10, width_coef=1., depth_coef=1., scale=1., dropout=0.2, se_scale=4,
stochastic_depth=False, p=0.5):
        super().__init__()
        channels = [32, 16, 24, 40, 80, 112, 192, 320, 1280]
        repeats = [1, 2, 2, 3, 3, 4, 1]
        strides = [1, 2, 2, 2, 1, 2, 1]
        kernel_size = [3, 3, 5, 3, 5, 5, 3]
        depth = depth_coef
        width = width_coef

        channels = [int(x*width) for x in channels]
        repeats = [int(x*depth) for x in repeats]

        # stochastic depth
        if stochastic_depth:
            self.p = p
            self.step = (1 - 0.5) / (sum(repeats) - 1)
        else:
            self.p = 1
            self.step = 0
```

EfficientNet

EfficientNet 정의

```
        # efficient net
        self.upsample = nn.Upsample(scale_factor=scale, mode='bilinear',
align_corners=False)

        self.stage1 = nn.Sequential(
            nn.Conv2d(3, channels[0],3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(channels[0], momentum=0.99, eps=1e-3)
        )

        self.stage2 = self._make_Block(SepConv, repeats[0], channels[0], channels[1],
kernel_size[0], strides[0], se_scale)

        self.stage3 = self._make_Block(MBConv, repeats[1], channels[1], channels[2],
kernel_size[1], strides[1], se_scale)

        self.stage4 = self._make_Block(MBConv, repeats[2], channels[2], channels[3],
kernel_size[2], strides[2], se_scale)

        self.stage5 = self._make_Block(MBConv, repeats[3], channels[3], channels[4],
kernel_size[3], strides[3], se_scale)

        self.stage6 = self._make_Block(MBConv, repeats[4], channels[4], channels[5],
kernel_size[4], strides[4], se_scale)

        self.stage7 = self._make_Block(MBConv, repeats[5], channels[5], channels[6],
kernel_size[5], strides[5], se_scale)

        self.stage8 = self._make_Block(MBConv, repeats[6], channels[6], channels[7],
kernel_size[6], strides[6], se_scale)
```

```
        self.stage9 = nn.Sequential(
            nn.Conv2d(channels[7], channels[8], 1, stride=1, bias=False),
            nn.BatchNorm2d(channels[8], momentum=0.99, eps=1e-3),
            Swish()
        )

        self.avgpool = nn.AdaptiveAvgPool2d((1,1))
        self.dropout = nn.Dropout(p=dropout)
        self.linear = nn.Linear(channels[8], num_classes)

    def forward(self, x):
        x = self.upsample(x)
        x = self.stage1(x)
        x = self.stage2(x)
        x = self.stage3(x)
        x = self.stage4(x)
        x = self.stage5(x)
        x = self.stage6(x)
        x = self.stage7(x)
        x = self.stage8(x)
        x = self.stage9(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = self.linear(x)
        return x
```

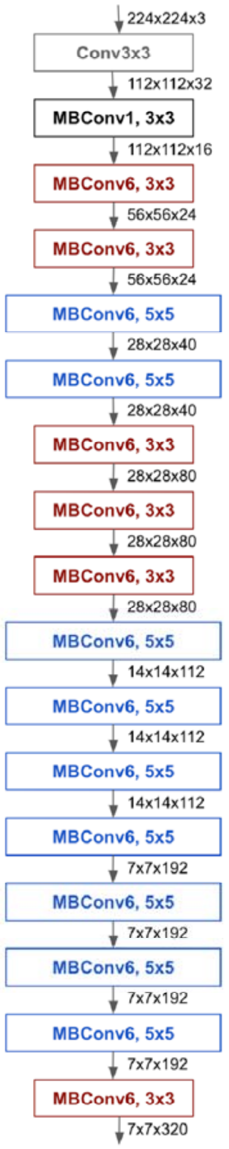


7 EfficientNet

EfficientNet 정의

```
    def _make_Block(self, block, repeats, in_channels, out_channels, kernel_size, stride, 
se_scale):
        strides = [stride] + [1] * (repeats - 1)
        layers = []
        for stride in strides:
            layers.append(block(in_channels, out_channels, kernel_size, stride, se_scale, 
self.p))
            in_channels = out_channels
            self.p -= self.step

        return nn.Sequential(*layers)

    def efficientnet_b0(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=1.0, depth_coef=1.0, 
scale=1.0,dropout=0.2, se_scale=4)

    def efficientnet_b1(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=1.0, depth_coef=1.1, 
scale=240/224, dropout=0.2, se_scale=4)

    def efficientnet_b2(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=1.1, depth_coef=1.2, 
scale=260/224., dropout=0.3, se_scale=4)

    def efficientnet_b3(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=1.2, depth_coef=1.4, 
scale=300/224, dropout=0.3, se_scale=4)
```

```
    def efficientnet_b4(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=1.4, 
depth_coef=1.8, scale=380/224, dropout=0.4, se_scale=4)

    def efficientnet_b5(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=1.6, 
depth_coef=2.2, scale=456/224, dropout=0.4, se_scale=4)

    def efficientnet_b6(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=1.8, 
depth_coef=2.6, scale=528/224, dropout=0.5, se_scale=4)

    def efficientnet_b7(num_classes=10):
        return EfficientNet(num_classes=num_classes, width_coef=2.0, 
depth_coef=3.1, scale=600/224, dropout=0.5, se_scale=4)


# check
if __name__ == '__main__':
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    x = torch.randn(3, 3, 224, 224).to(device)
    model = efficientnet_b0().to(device)
    output = model(x)
    print('output size:', output.size())
```



8 EfficientNetV2 (Smaller Models and Faster Training)

Mingxing Tan, Quoc V. Le, "EfficientNetV2: Smaller Models and Faster Training," arXiv:2104.00298, 2021 (**Google Research Brain Team**) (ICML'21)

https://github.com/google/automl/tree/master/efficientnetv2

Abstract

- EfficientNetV2, a new family of convolutional networks that have **faster training speed** and **better parameter efficiency** than previous models.
- **Use a combination of training-aware neural architecture search and scaling**, to jointly **optimize training speed and parameter efficiency**.
- The models were searched from the search space enriched with **new ops** such as **Fused-MBConv**.
- Our experiments show that EfficientNetV2 models train **4x faster** than SOTA models while being up to **6.8x smaller**.

➢ 논문에서 학습 속도를 느리게 하는 3가지 요소 설명 (Training Bottlenecks)

(1) 큰 이미지로 학습을 하면 학습 속도가 느리다.

(2) 초기 layer에서 depthwise convolution은 학습 속도에 악영향 준다.

(3) 모든 stage를 동일한 비율로 scailing up하는 것은 최적의 선택이 아니다.

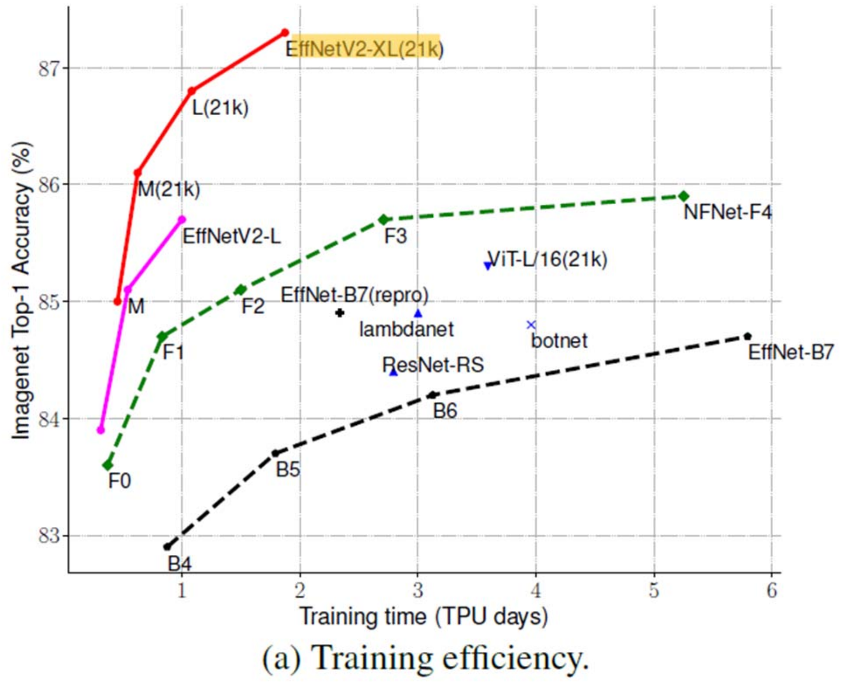
이 3가지를 해결하기 위해서 **progressive learning**과 **fused-MBConv, non-uniform scaling** 전략을 소개함.

Breakpoint

- **Neural architecture search (NAS)**: Use of random search/reinforcement learning to make optimal model design choices and find hyperparameters.
- **Scaling strategies:** Guidelines on how to upscale small networks into bigger ones effectively, e.g. compound scaling rule of EfficientNet.
- **Training strategies**: e.g. new regularization methods, guidelines for training efficiency.
- **Progressive learning**: Accelerating training by progressively increasing the image size and adaptively adjusts regularization (e.g. dropout and augmentation) along with image size
- **Various types of convolutions and building blocks**: e.g. depthwise conv, depthwise-separable conv, squeeze and excitation(SE), MB Conv, **Fused-MB Conv.**

8

EfficientNetV2 (Smaller Models and Faster Training)



(a) Training efficiency.

	EfficientNet (2019)	ResNet-RS (2021)	DeiT/ViT (2021)	EfficientNetV2 (ours)
Top-1 Acc.	84.3%	84.0%	83.1%	83.9%
Parameters	43M	164M	86M	24M

(b) Parameter efficiency.

Figure 1. **ImageNet ILSVRC2012 top-1 Accuracy vs. Training Time and Parameters** – Models tagged with 21k are pretrained on ImageNet21k, and others are directly trained on ImageNet ILSVRC2012. Training time is measured with 32 TPU cores. All EfficientNetV2 models are trained with progressive learning. Our EfficientNetV2 trains 5x - 11x faster than others, while using up to 6.8x fewer parameters.

Review of EfficientNet

- Optimized for **FLOPs** and **parameter efficiency**.
- Leverage NAS to search for the baseline EfficientNet-B0 model that has better trade-off on accuracy and FLOPs.
- The baseline model is then scaled up with a simple compound scaling strategy to obtain a family of models B1-B7.
- In this paper, we aim to improve the **training speed** while maintaining the parameter efficiency.

8

EfficientNetV2 (Smaller Models and Faster Training)

3.2 Understanding Training Efficiency

(1) Training with very large image sizes is slow

Progressive learning with adaptive regularization : Accelerating training by progressively increasing the image size by adding stronger regularization

- Image size plays an important role in training efficiency. Training할 때, 이미지의 크기를 점진적으로 증가 → 학습 속도를 빠르게 하기 위해 사용하지만 정확도가 감소 (입력 이미지 크기에 따라 동일한 정규화를 사용하기 때문)
- Smaller images : Lead to smaller network capacity, thus weaker regularization (augment 확률 낮게)
- Larger image size : Lead to larger capacity, thus strong regularization ; more vulnerable to overfitting (augment 확률 높게 또는 dropout 높게)

Table 5. ImageNet top-1 accuracy. We use RandAug (Cubuk et al., 2020), and report mean and stdev for 3 runs.

	Size=128	Size=192	Size=300
RandAug magnitude=5	78.3 ±0.16	81.2 ±0.06	82.5 ±0.05
RandAug magnitude=10	78.0 ±0.08	81.6 ±0.08	82.7 ±0.08
RandAug magnitude=15	77.7 ±0.15	81.5 ±0.05	83.2 ±0.09

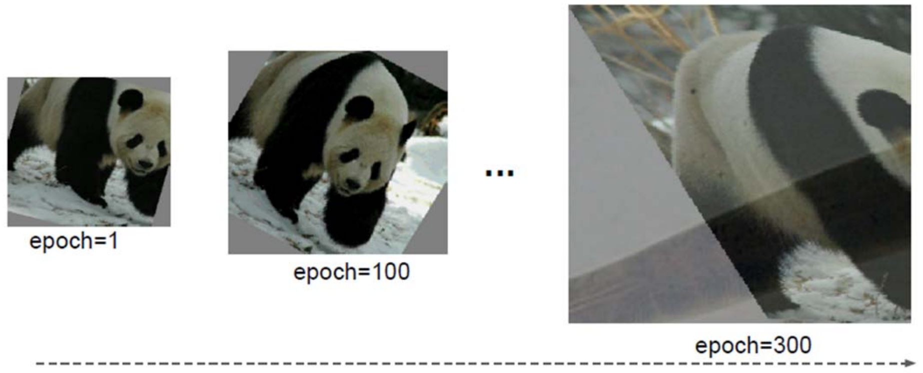


Figure 4. **Training process in our improved progressive learning** – It **starts with small image size and weak regularization (epoch=1)**, and then **gradually increase the learning difficulty with larger image sizes and stronger regularization**: larger dropout rate, RandAugment magnitude, and mixup ratio (e.g., epoch=300).

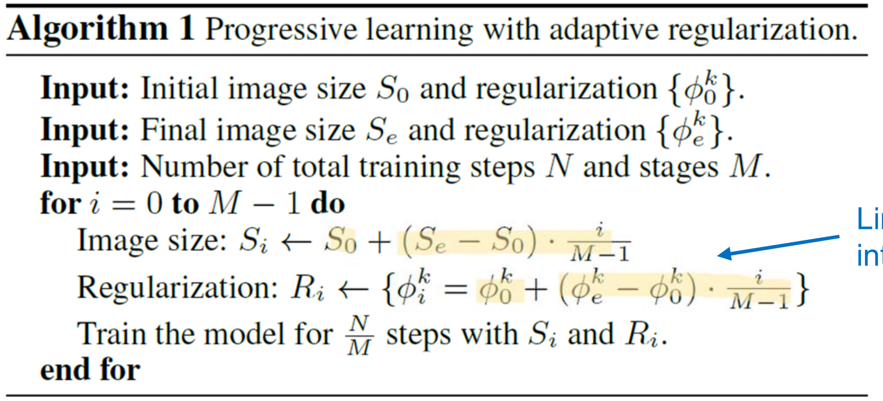
8

EfficientNetV2 (Smaller Models and Faster Training)

(1) Training with very large image sizes is slow

Progressive learning with adaptive regularization

- Target image size S_e with a list of regularization magnitude $\Phi_e = \{\phi_e^k\}$ where k represents a type of regularization (dropout, randaug, mixup)



Algorithm 1 Progressive learning with adaptive regularization.

```
**Input:** Initial image size $S_0$ and regularization $\{\phi_0^k\}$.
**Input:** Final image size $S_e$ and regularization $\{\phi_e^k\}$.
**Input:** Number of total training steps $N$ and stages $M$.
**for** $i = 0$ **to** $M - 1$ **do**
    Image size: $S_i \leftarrow S_0 + (S_e - S_0) \cdot \frac{i}{M-1}$
    Regularization: $R_i \leftarrow \{\phi_i^k = \phi_0^k + (\phi_e^k - \phi_0^k) \cdot \frac{i}{M-1}\}$
    Train the model for $\frac{N}{M}$ steps with $S_i$ and $R_i$.
**end for**
```

Linear interpolation

Simulation setup for progressive learning

- Divide the training process into 4 stages with about 87 epochs per stage
- min (first stage) and max (last stage)

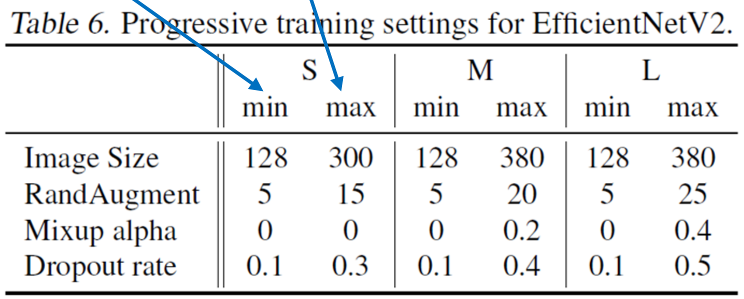
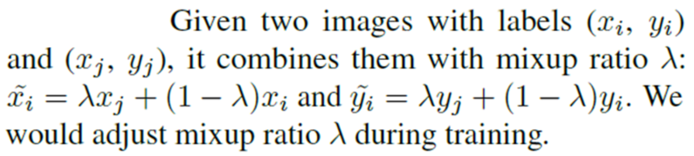


Table 6. Progressive training settings for EfficientNetV2.

	S		M		L	
	min	max	min	max	min	max
Image Size	128	300	128	380	128	380
RandAugment	5	15	5	20	5	25
Mixup alpha	0	0	0	0.2	0	0.4
Dropout rate	0.1	0.3	0.1	0.4	0.1	0.5

3types of regularization

- **Dropout** : network-level regularization, drop rate γ
- **RandAug** : a per-image data augmentation, with adjustable magnitude ϵ
- **Mixup** : a cross-image data augmentation.



Given two images with labels (x_i, y_i) and (x_j, y_j), it combines them with mixup ratio λ: $\tilde{x}_i = \lambda x_j + (1 - \lambda)x_i$ and $\tilde{y}_i = \lambda y_j + (1 - \lambda)y_i$. We would adjust mixup ratio λ during training.

8

EfficientNetV2 (Smaller Models and Faster Training)

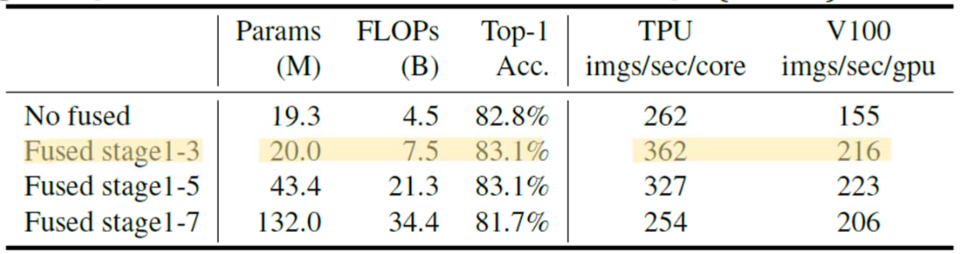
(2) Depthwise convolutions are slow in early layers

- **Depthwise convolution** (MobileNetV1, Xception에서 제안)
  - ✓ conv 연산량을 낮춰주어 제한된 연산량 내에 더 많은 filter를 사용할 수 있는 이점이 있음 (fewer parameters and FLOPs than regular conv.)
  - ✓ 하지만 modern accelerator를 활용하지 못하여 학습 속도를 느리게 함
- stage 1-3 에서 MBConv 대신에 Fused-MBConv를 사용

Fused-MBConv

- ✓ Replace the depthwise conv3x3 and expansion conv1x1 in MBConv with a single regular 3x3conv
- ✓ 모든 stage에 Fused-MBConv를 적용하니 오히려 학습 속도가 느려져서, 초기의 stage에만 Fused-MBConv를 사용

Table 3. Replacing MBConv with Fused-MBConv. No fused denotes all stages use MBConv, Fused stage1-3 denotes replacing MBConv with Fused-MBConv in stage {2, 3, 4}.



	Params (M)	FLOPs (B)	Top-1 Acc.	TPU imgs/sec/core	V100 imgs/sec/gpu
No fused	19.3	4.5	82.8%	262	155
Fused stage1-3	20.0	7.5	83.1%	362	216
Fused stage1-5	43.4	21.3	83.1%	327	223
Fused stage1-7	132.0	34.4	81.7%	254	206

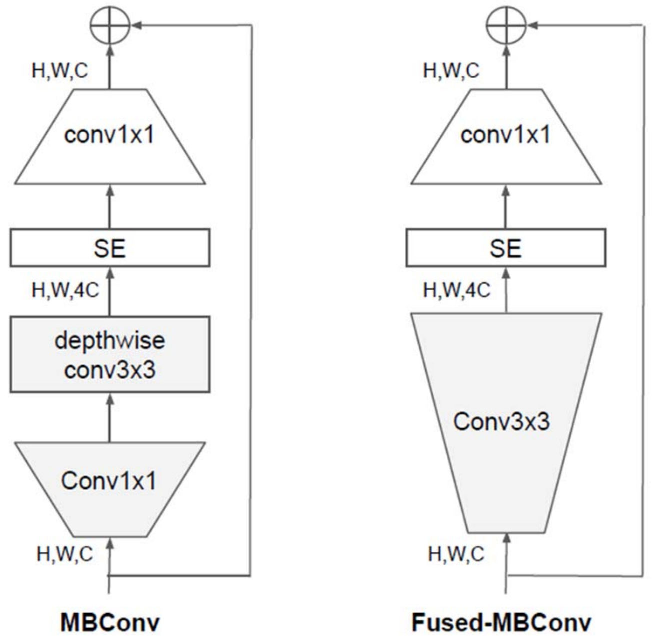


Figure 2. Structure of MBConv and Fused-MBConv.

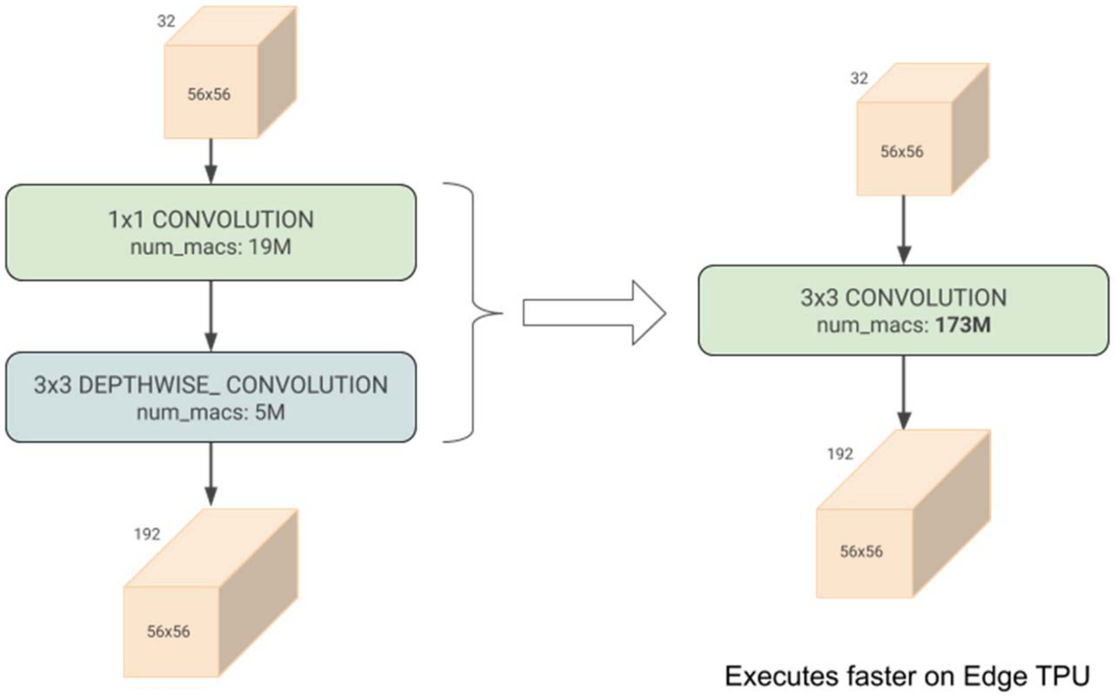
Params / FLOPS 자체는 단순 MBConv 보다 많지만,
실제 throughput은 Fused-MBConv 조합하는 것이 더 빠르다

8

EfficientNetV2 (Smaller Models and Faster Training)

(2) Depthwise convolutions are slow in early layers

https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html



A regular 3x3 convolution (right) has more compute (multiply-and-accumulate (mac) operations) than a depthwise-separable convolution (left), but for certain input/output shapes, executes faster on Edge TPU due to ~3x more effective hardware utilization.

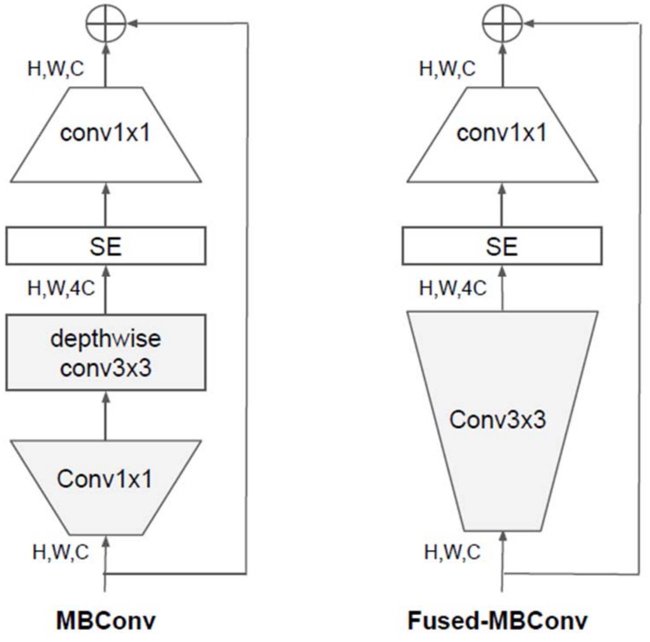


Figure 2. Structure of MBConv and Fused-MBConv.

Params / FLOPS 자체는 단순 MBConv 보다 많지만,
실제 throughput은 Fused-MBConv 조합하는 것이 더 빠르다

8 EfficientNetV2 (Smaller Models and Faster Training)

(3) Equally scaling up every stage is sub-optimal

- **EfficientNet equally scales up all stages** using a simple compound scaling rule.
  - ✓ For example, when depth coefficient is 2, then all stages in the networks would double the number of layers.
  - ✓ However, **these stages are not equally contributed to the training speed and parameter efficiency**.
- We will use a **non-uniform scaling strategy** to **gradually add more layers to later stages**.

- In addition, EfficientNets aggressively scale up image size, leading to large memory consumption and slow training.
- To address this issue, we slightly **modify the scaling rule and restrict the maximum image size to a smaller value**.

8 EfficientNetV2 (Smaller Models and Faster Training)

3.3 Training-Aware NAS and Scaling

- Haver learned multiple design choices for improving **training speed**

1) NAS Search

Training-Aware NAS framework is largely based on previous NAS but aims to jointly **optimize accuracy**, **parameter efficiency**, and **training efficiency**.

- Backbone : **EfficientNet**
- Search space : Conv. operation types {MBConv, Fused-MBConv}, number of layers, kernel size {3x3, 5x5), Expansion ratio {1, 4, 6}
- Sample up to 1,000 models and train each model about 10 epochs with reduced image size for training.
- Search reward : Combines the model accuracy A, Normalized training step time S, parameter size P, using a weighted product $A \cdot S^w \cdot P^v$ where w=-0.07 and v=-0.05

Searched model “EfficientNetV2-S”

Scale up EfficientNetV2-S to EfficientNetV2-M/L using similar compound scaling

2) EfficientNetV2 architecture

- Major distinctions with EfficientNet and our EfficientNetV2

1) EfficientNetV2 uses both MBConv and fuzed MBConv in early layers.
2) EfficientNetV2 prefers
   - smaller expansion ratios for MBConv
   - smaller 3x3 kernel sizes, but add more layers to compensate the reduced receptive field from the smaller kernel size.
3) Removes the last stride-1 stage.

Table 4. EfficientNetV2-S **architecture** – MBConv and Fused-MBConv blocks are described in Figure 2.

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	272	15
7	Conv1x1 & Pooling & FC	-	1792	1

8

EfficientNetV2 (Smaller Models and Faster Training)

3) EfficientNetV2 Scaling

- Scale up **EfficientNetV2-S** to **EfficientNetV2-M/L** using similar compound scaling with a few additional optimizations

1) Restrict the **maximum inference image size** to **480**, as very large images often lead to expensive memory and training speed overhead;
2) As a heuristic, **gradually add more layers to later stages** (e.g., stage 5 and 6 in Table4) in order to increase the network capacity without adding much runtime overhead.

4) Training Speed Comparison

- With our training-aware NAS and scaling, EfficientNetV2 model train much faster than the other recent models.

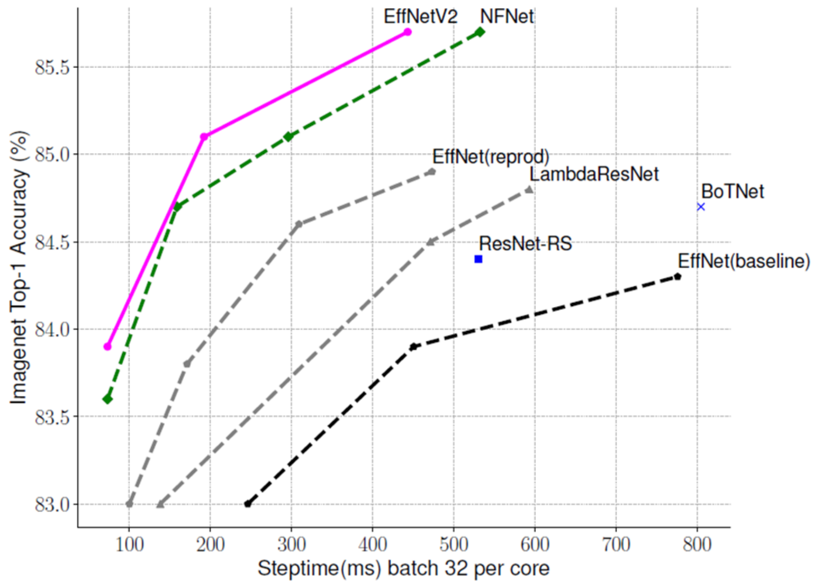


Figure 3. ImageNet accuracy and training step time on TPUv3 – Lower step time is better; all models are trained with fixed image size without progressive learning

8 EfficientNetV2

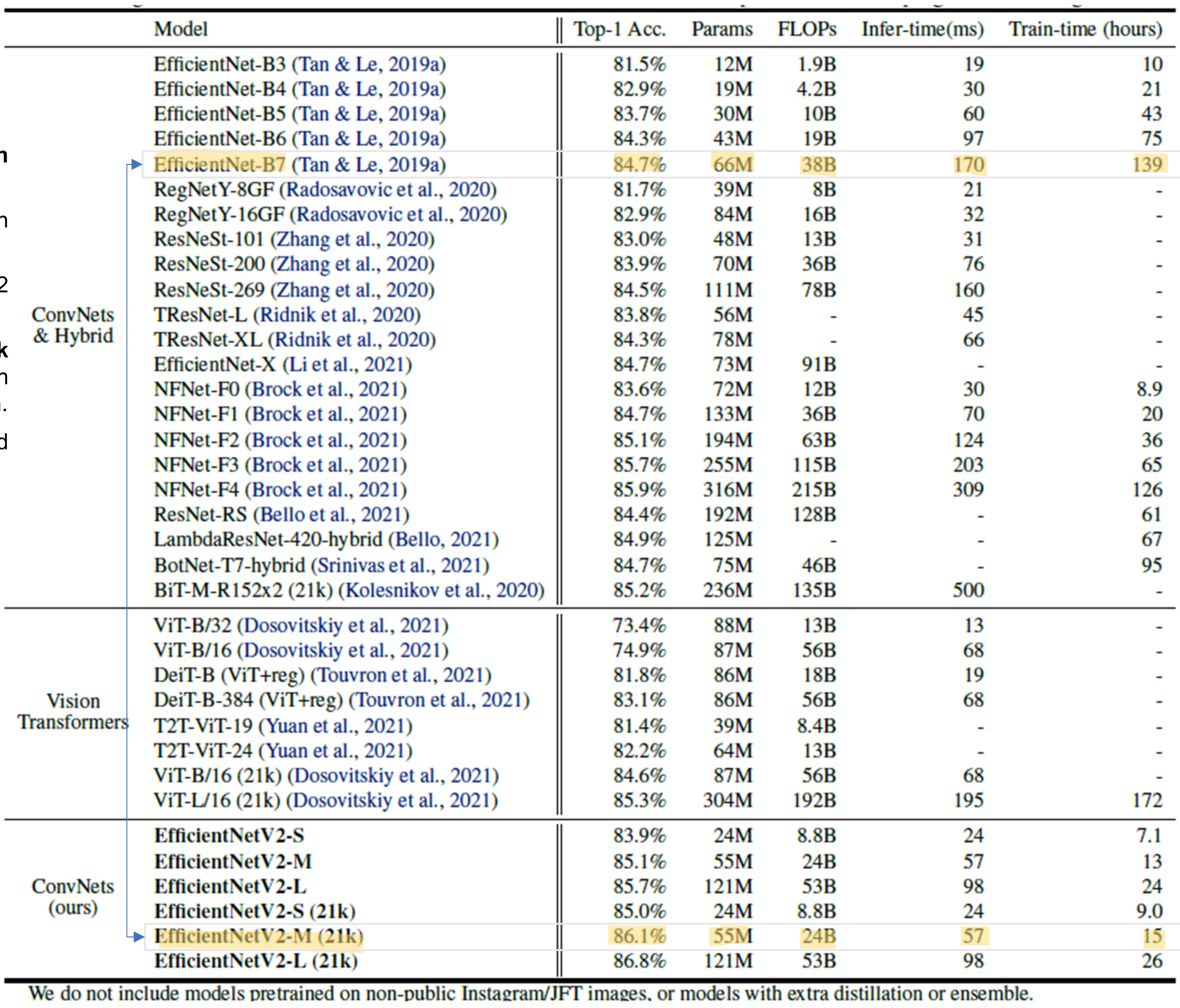
Table 7. **EfficientNetV2 Performance Results on ImageNet** (Russakovsky et al., 2015) –

Infer-time is measured on V100 GPU FP16 with batch size 16 using the same codebase (Wightman, 2021);

Train-time is the total training time normalized for 32 TPU cores.

Models marked with 21k are pretrained on **ImageNet21k** with 13M images, and others are directly trained on ImageNet ILSVRC2012 with 1.28M images from scratch.

All EfficientNetV2 models are trained with our improved method of progressive learning.



	Model	Top-1 Acc.	Params	FLOPs	Infer-time(ms)	Train-time (hours)
ConvNets & Hybrid	EfficientNet-B3 (Tan & Le, 2019a)	81.5%	12M	1.9B	19	10
	EfficientNet-B4 (Tan & Le, 2019a)	82.9%	19M	4.2B	30	21
	EfficientNet-B5 (Tan & Le, 2019a)	83.7%	30M	10B	60	43
	EfficientNet-B6 (Tan & Le, 2019a)	84.3%	43M	19B	97	75
	EfficientNet-B7 (Tan & Le, 2019a)	84.7%	66M	38B	170	139
	RegNetY-8GF (Radosavovic et al., 2020)	81.7%	39M	8B	21	-
	RegNetY-16GF (Radosavovic et al., 2020)	82.9%	84M	16B	32	-
	ResNeSt-101 (Zhang et al., 2020)	83.0%	48M	13B	31	-
	ResNeSt-200 (Zhang et al., 2020)	83.9%	70M	36B	76	-
	ResNeSt-269 (Zhang et al., 2020)	84.5%	111M	78B	160	-
	TResNet-L (Ridnik et al., 2020)	83.8%	56M	-	45	-
	TResNet-XL (Ridnik et al., 2020)	84.3%	78M	-	66	-
	EfficientNet-X (Li et al., 2021)	84.7%	73M	91B	-	-
	NFNet-F0 (Brock et al., 2021)	83.6%	72M	12B	30	8.9
	NFNet-F1 (Brock et al., 2021)	84.7%	133M	36B	70	20
	NFNet-F2 (Brock et al., 2021)	85.1%	194M	63B	124	36
	NFNet-F3 (Brock et al., 2021)	85.7%	255M	115B	203	65
	NFNet-F4 (Brock et al., 2021)	85.9%	316M	215B	309	126
	ResNet-RS (Bello et al., 2021)	84.4%	192M	128B	-	61
	LambdaResNet-420-hybrid (Bello, 2021)	84.9%	125M	-	-	67
	BotNet-T7-hybrid (Srinivas et al., 2021)	84.7%	75M	46B	-	95
	BiT-M-R152x2 (21k) (Kolesnikov et al., 2020)	85.2%	236M	135B	500	-
Vision Transformers	ViT-B/32 (Dosovitskiy et al., 2021)	73.4%	88M	13B	13	-
	ViT-B/16 (Dosovitskiy et al., 2021)	74.9%	87M	56B	68	-
	DeiT-B (ViT+reg) (Touvron et al., 2021)	81.8%	86M	18B	19	-
	DeiT-B-384 (ViT+reg) (Touvron et al., 2021)	83.1%	86M	56B	68	-
	T2T-ViT-19 (Yuan et al., 2021)	81.4%	39M	8.4B	-	-
	T2T-ViT-24 (Yuan et al., 2021)	82.2%	64M	13B	-	-
	ViT-B/16 (21k) (Dosovitskiy et al., 2021)	84.6%	87M	56B	68	-
	ViT-L/16 (21k) (Dosovitskiy et al., 2021)	85.3%	304M	192B	195	172
ConvNets (ours)	**EfficientNetV2-S**	83.9%	24M	8.8B	24	7.1
	EfficientNetV2-M	85.1%	55M	24B	57	13
	EfficientNetV2-L	85.7%	121M	53B	98	24
	EfficientNetV2-S (21k)	85.0%	24M	8.8B	24	9.0
	EfficientNetV2-M (21k)	86.1%	55M	24B	57	15
	EfficientNetV2-L (21k)	86.8%	121M	53B	98	26

We do not include models pretrained on non-public Instagram/JFT images, or models with extra distillation or ensemble.

8

EfficientNetV2

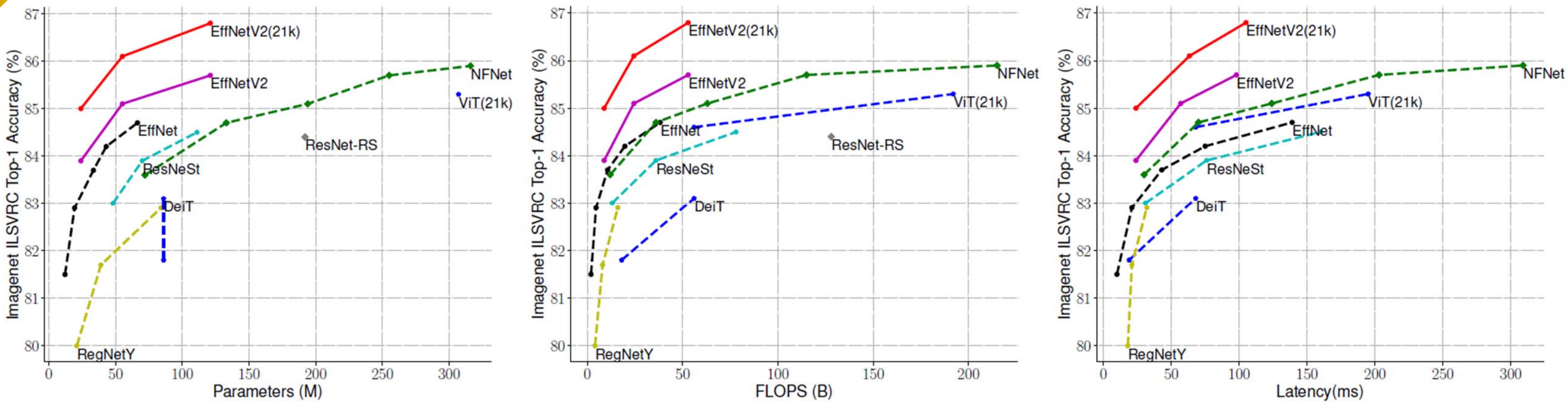


Figure 5. **Model Size**, **FLOPs**, and **Inference Latency** – Latency is measured with batch size 16 on V100 GPU. 21k denotes pretrained on ImageNet21k images, others are just trained on ImageNet ILSVRC2012. Our EfficientNetV2 has slightly better parameter efficiency with EfficientNet, but runs 3x faster for inference.

Scaling up data size is more effective than simply scaling up model size in high-accuracy regime: when the top-1 accuracy is beyond 85%, it is very difficult to further improve it by simply increasing model size due to the *severe overfitting*. However, the extra **ImageNet21K** pretraining can significantly improve accuracy.

Pretraining on ImageNet21k could be quite efficient. Although ImageNet21k has 10x more data, our training approach enables us to finish the pretraining of EfficientNetV2 within two days using 32 TPU cores (instead of weeks for ViT (Dosovitskiy et al., 2021)).