Image Classification with EfficientNet: Better performance with computational efficiency

Anand Borad
Dec 14, 2019 · 8 min read

In May 2019, two engineers from Google brain team named Mingxing Tan and Quoc V. Le published a paper called "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". The core idea of publication was about strategically scaling deep neural networks but it also introduced a new family of neural nets, *EfficientNets*.

EfficientNets, as the name suggests are very much efficient computationally and also achieved state of art result on ImageNet dataset which is 84.4% top-1 accuracy.

So, in this article, we will discuss EfficientNets in detail but first, we will talk about the core idea introduced in the paper, model scaling.

Model scaling is about scaling the existing model in terms of model depth, model width, and less popular input image resolution to improve the performance of the model. Depth wise scaling is most popular amongst all, e.g. ResNet can be scaled from Resnet18 to ResNet200. Here ResNet10 has 18 residual blocks and can be scaled for depth to have 200 residual blocks.

ResNet200 delivers better performance than ResNet18 and thus, manually scaling works pretty well. But there is one problem with traditional manual scaling method, after a certain level, scaling doesn't improve performance. It starts to affect adversely by degrading performance.

The scaling method introduced in paper is named *compound scaling* and suggests that instead of scaling only one model attribute out of depth, width, and resolution; strategically scaling all three of them together delivers better results.

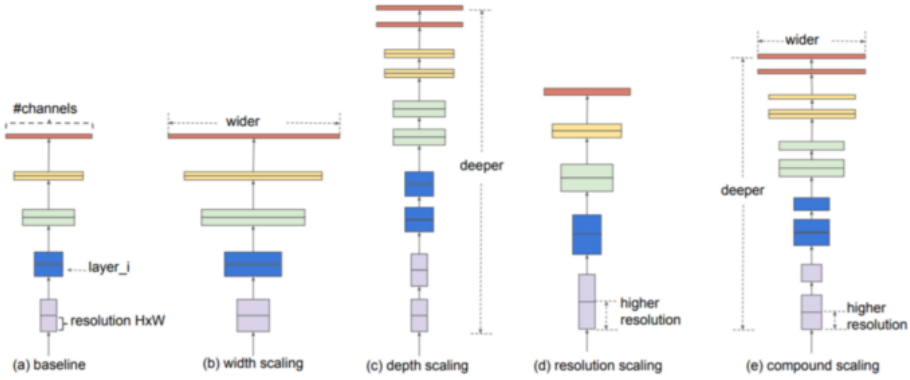
Compound scaling

Compound scaling method uses a compound co-efficient ø to scale width, depth, and resolution together. Below is the formula for scaled attributes:

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

Here, alpha, beta, and gamma are scaling multiplier for depth, width and resolution respectively and be obtained using grid search. Let's say we got alpha =1.2 after solving the above equation, then new depth = 1.2 * old depth.

ø is a user-specific co-efficient which takes real numbers like and controls resources which is 2ø. So if we have double resources available than what a model is currently using, we can take find ø using 2ø = 2 and hence ø is 1 for such cases.



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.

Using this compound scaling method, they achieved brilliant improvement for MobileNets and ResNet model architecture as shown in below image.

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%**

Scaling Up MobileNets and ResNet

Authors observed that mobile scaling can be used on any CNN architecture and it works just fine but the overall performance very much depends on baseline architecture. With that observation in mind, they came up with the brand new base architecture and named it EfficientNet-B0.

The base model of EfficientNet family, EfficientNet-B0

The EfficientNet-B0 architecture wasn't developed by engineers but by the neural network itself. They developed this model using a multi-objective neural architecture search that optimizes both accuracy and floating-point operations.

Taking B0 as a baseline model, the authors developed a full family of EfficientNets from B1 to B7 which achieved state of the art accuracy on ImageNet while being very efficient to its competitors.

Below is a table showing the performance of EfficientNets family on ImageNet dataset.

Model	Top-1 Acc.	Top-5 Acc.	#Params	Ratio-to-EfficientNet	#FLOPS	Ratio-to-EfficientNet
EfficientNet-B0	**77.3%**	**93.5%**	**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	**79.2%**	**94.5%**	**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	**80.3%**	**95.0%**	**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	**81.7%**	**95.6%**	**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	**83.0%**	**96.3%**	**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	**83.7%**	**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	**84.2%**	**96.8%**	**43M**	**1x**	**19B**	**1x**
EfficientNet-B7	**84.4%**	**97.1%**	**66M**	**1x**	**37B**	**1x**
GPipe (Huang et al., 2018)	84.3%	97.0%	557M	8.4x	-	-

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 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.

Here, we will deep dive into EfficientNet-B0 architecture. B0 is mobile sized architecture having 11M trainable parameters.

Before moving ahead, let's see how this new architecture looks like:

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

One can see that architecture uses 7 inverted residual blocks but each is having different settings. These blocks also use squeeze & excitation block along with swish activation. We will discuss all three in detail in this article. let's start with Swish.

Swish Activation

ReLu works pretty well but it got a problem, it nullifies negative values and thus derivatives are zero for all negative values. There are many known alternatives to tackle this problem like leaky ReLu, Elu, Selu etc., but none of them has proven consistent.

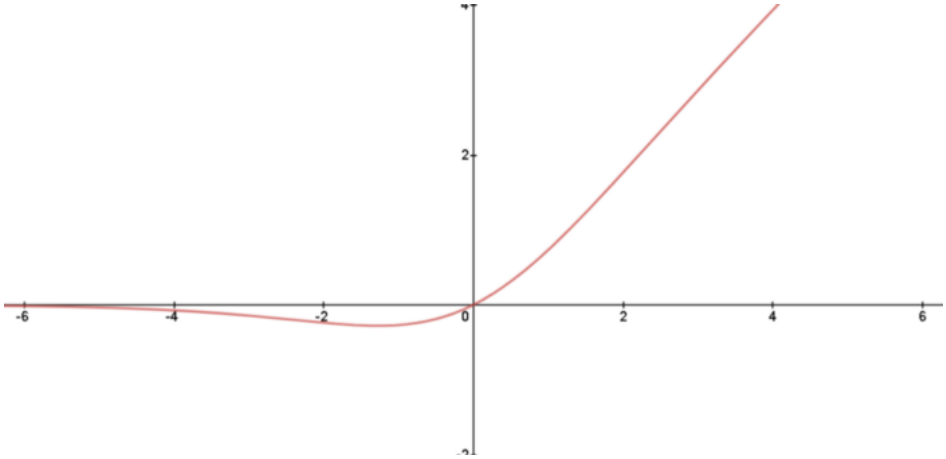
Google Brain team suggested a newer activation that tends to work better for deeper networks than ReLU which is a Swish activation. They proved that if we replace Swish with ReLu on InceptionResNetV2, we can achieve 0.6% more accuracy on ImageNet dataset.

Swish is a multiplication of a linear and a sigmoid activation.

Swish(x) = x * sigmoid(x)

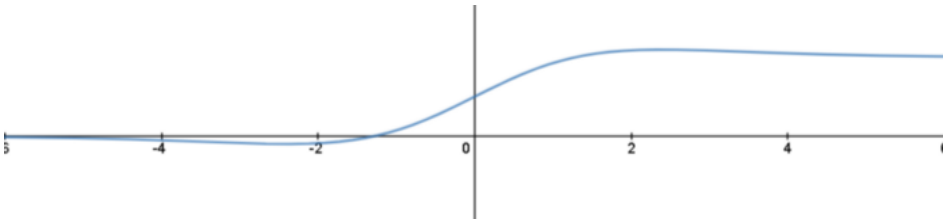
```
from keras import backend as K

def swish_activation(x):
    return x * K.sigmoid(x)
```

Swish looks as shown in the below image:



Swish activation

It's gradient looks as shown in the below image:

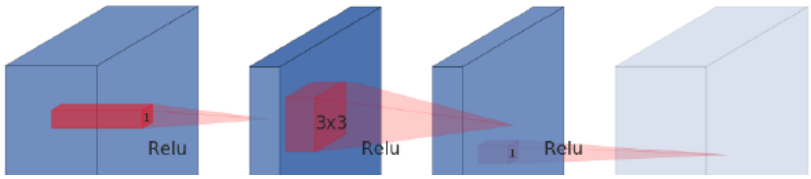


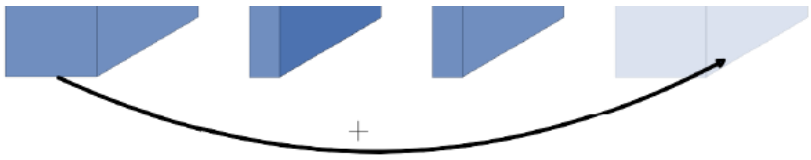
Derivatives of swish

Inverted Residual Block

The idea of a residual block was introduced in MobileNet architecture. MobileNet uses depthwise separable convolution inside the residual block which uses depthwise convolution first and then pointwise convolution. This approach decreases trainable parameters by a large number.

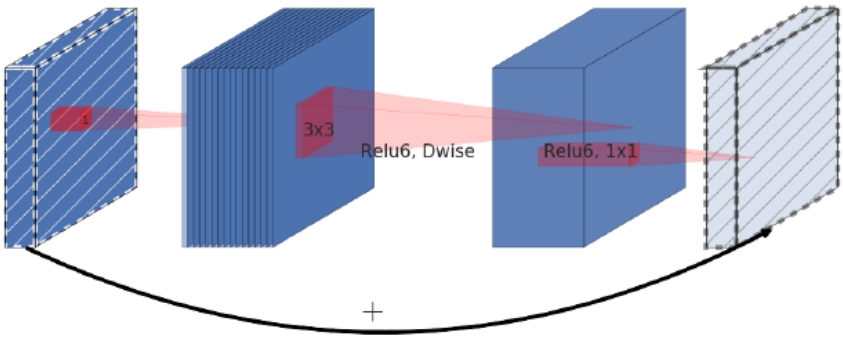
In an original residual block (introduced in ResNet), skip connections are used to connect wide layers (aka layers with a large number of channels) and there are fewer numbers of channels inside a block (aka narrow layers).





Residual block

The inverted residual block does the opposite, skip connections connects narrow layers while wider layers are between skip connections.



Inverted residual block

The code for the inverted residual block is as below:

```
from keras.layers import Conv2D, DepthwiseConv2D, Add

def 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])
```

Learn more about inverted residual block here:

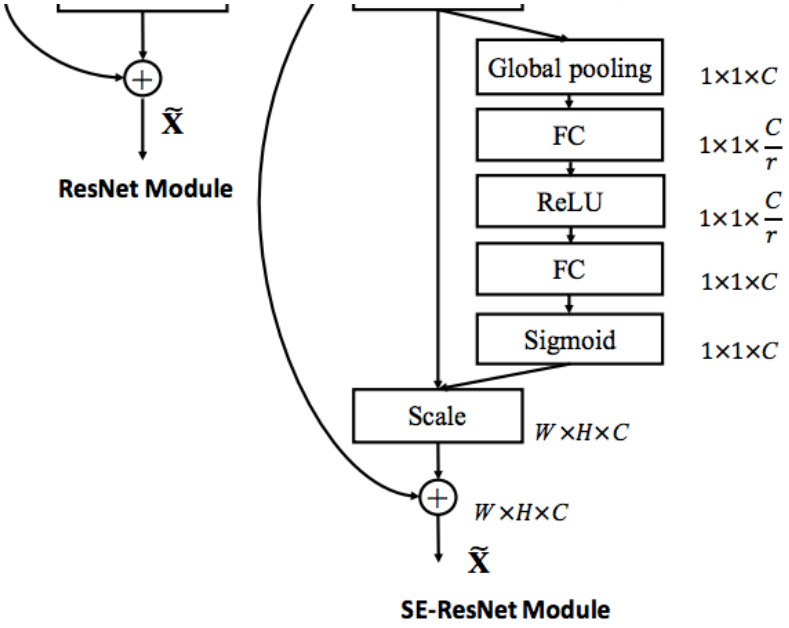
Squeeze and Excitation Block

When CNN creates output feature map from a convolutional layer, it gives equal weightage to each of channels. Squeeze and excitation (SE) block is a method to give weightage to each channel instead of treating them all equally.

SE block gives the output of shape (1 x 1 x channels) which specifies the weightage for each channel and the great thing is that neural network can learn this weightage by itself like other parameters.

below is the code:

```
from keras.layers import GlobalAveragePooling2D, Reshape, Conv2D

def se_block(x, filters, squeeze_ratio=0.25):
    x_ = GlobalAveragePooling2D()(x)
    x_ = Reshape((1,1,filters))(x_)
    squeezed_filters = max(1, int(filters * squeeze_ratio))
    x_ = Conv2D(squeezed_filters , activation='relu')(x_)
    x_ = Conv2D(filters, activation='sigmoid')(x_)
    return multiply()([x, x_])
```



Learn more here about SE networks.

EfficientNet' MBConv Block

Now, we had a brief introduction about all three building blocks used EfficientNets, let's see how an MBConv block feels like.

Below is the code inspired from this brilliant repository on Github about EfficientNet.

MBConv block takes two inputs, first is data and the other is block arguments. The data is output from the last layer. A block argument is a collection of attributes to be used inside an MBConv block like input filters, output filters, expansion ratio, squeeze ratio etc. Argument blocks for B0 model as below:

```
argument_block = [
BlockArgs(kernel_size=3, num_repeat=1, input_filters=32,
output_filters=16, expand_ratio=1, id_skip=True, strides=[1, 1],
se_ratio=0.25),

BlockArgs(kernel_size=3, num_repeat=2, input_filters=16,
output_filters=24, expand_ratio=6, id_skip=True, strides=[2, 2],
se_ratio=0.25),

BlockArgs(kernel_size=5, num_repeat=2, input_filters=24,
output_filters=40, expand_ratio=6, id_skip=True, strides=[2, 2],
se_ratio=0.25),

BlockArgs(kernel_size=3, num_repeat=3, input_filters=40,
output_filters=80, expand_ratio=6, id_skip=True, strides=[2, 2],
se_ratio=0.25),

BlockArgs(kernel_size=5, num_repeat=3, input_filters=80,
output_filters=112, expand_ratio=6, id_skip=True, strides=[1, 1],
se_ratio=0.25),

BlockArgs(kernel_size=5, num_repeat=4, input_filters=112,
output_filters=192, expand_ratio=6, id_skip=True, strides=[2, 2],
se_ratio=0.25),

BlockArgs(kernel_size=3, num_repeat=1, input_filters=192,
output_filters=320, expand_ratio=6, id_skip=True, strides=[1, 1],
se_ratio=0.25)
]
```

EfficientNet uses 7 MBConv blocks and above is specifications (argument block) for each of those blocks respectively.

- kernel_size is kernel size for convolution e.g. 3 x 3
- num_repeat specifies how many times a particular block needs to be repeated, must be greater than zero
- input_filters and output_filters are numbers of specified filters
- expand_ratio is input filter expansion ratio
- id_skip suggests whether to use skip connection or not
- se_ratio provides squeezing ratio for squeeze and excitation block

```
def mbConv_block(input_data, block_arg):
    """Mobile Inverted Residual block along with Squeeze
    and Excitation block."""

    kernel_size = block_arg.kernel_size
    num_repeat= block_arg.num_repeat
    input_filters= block_arg.input_filters
    output_filters= output_filters.kernel_size
    expand_ratio= block_arg.expand_ratio
    id_skip= block_arg.id_skip
    strides= block_arg.strides
    se_ratio= block_arg.se_ratio

    # continue...
```

Expansion phase: we will expand our layer and make them wide as mentioned in Inverted residual block (connected blocks are narrow and inner blocks are wider, here we are making layer wider just by increasing the number of channels).

```
    # continue...

    expanded_filters = input_filters * expand_ratio
    x = Conv2D(expanded_filters, 1, padding='same', use_bias=False)
(input_data)
    x = BatchNormalization()(x)
    x = Activation(swish_activation)(x)

    # continue...
```

Depthwise convolution phase: after expansion, we perform depthwise convolution with kernel size mentioned in block argument.

```
# continue...

    x = DepthwiseConv2D(kernel_size, strides, padding='same',
use_bias=False)(x)
    x = BatchNormalization()(x)
    x = Activation(swish_activation)(x)

# continue...
```

Squeeze and excitation phase: now, we extract global features with global average pooling and squeeze numbers of channels using se_ratio.

```
# continue...

    se = GlobalAveragePooling2D()(x)
    se = Reshape((1, 1, expanded_filters ))(x)
    squeezed_filters = max (1, int(input_filters * se_ratio))
    se = Conv2D(squeezed_filters , 1, activation=swish_activation,
```

```
padding='same')(se)
    se = Conv2D(expanded_filters, 1, activation='sigmoid',
padding='same')(se)
    x = multiply([x, se])

# continue...
```

Here shape of se block would be (1, 1, expanded_filters) and the shape of x would be (h, w, expanded_filters). thus, the output of se block can be considered as weightage for each channel in the output of x. To give weightage, we simply multiply se output with x output.

Output phase: after the se block, we apply convolution operation that gives output filters mention in the argument block.

```
# continue...

    x = Conv2D(output_filters, 1, padding='same', use_bias=False)
    x = BatchNormalization()(x)
    return x
```

below is the complete code by putting all together.

```
def mbConv_block(input_data, block_arg):
    """Mobile Inverted Residual block along with Squeeze
    and Excitation block."""

    kernel_size = block_arg.kernel_size
    num_repeat= block_arg.num_repeat
    input_filters= block_arg.input_filters
    output_filters= output_filters.kernel_size
    expand_ratio= block_arg.expand_ratio
    id_skip= block_arg.id_skip
    strides= block_arg.strides
    se_ratio= block_arg.se_ratio

    # expansion phase

    expanded_filters =  input_filters * expand_ratio
    x = Conv2D(expanded_filters, 1,  padding='same',  use_bias=False)
(input_data)
    x = BatchNormalization()(x)
    x = Activation(swish_activation)(x)

    # Depthwise convolution phase

    x = DepthwiseConv2D(kernel_size, strides,  padding='same',
use_bias=False)(x)
    x = BatchNormalization()(x)
    x = Activation(swish_activation)(x)

    # Squeeze and excitation phase

    se = GlobalAveragePooling2D()(x)
    se = Reshape((1, 1, expanded_filters ))(x)
    squeezed_filters = max (1, int(input_filters * se_ratio))
    se = Conv2D(squeezed_filters , 1, activation=swish_activation,
padding='same')(se)
    se = Conv2D(expanded_filters, 1, activation='sigmoid',
padding='same')(se)
    x = multiply([x, se])

    # Output phase

    x = Conv2D(output_filters, 1, padding='same', use_bias=False)
    x = BatchNormalization()(x)
    return x
```

Note that above very simplistic representation of MBConv block, an actual block is bit complex and considers few constraints. Learn more here.

EfficientNet can take smaller images as input also but it will be overkill for a dataset like MNIST. EfficientNets are advisable to use for complex datasets. We will be using EfficientNet B0 on CIFAR10 data and will train the model for 10 epochs. I have put the code for EfficientNetB0 on CIFAR10 on this google colab notebook so that you can play there.

References:

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

EfficientNet on GitHub from Qubvel

MobileNetV2: Inverted Residuals and Linear Bottlenecks

Squeeze-and-Excitation Networks

Swish: Searching for Activation Functions

EfficientNet on CIFAR10

Machine Learning Image Classification Efficientnet AI