How to do Transfer learning with Efficientnet

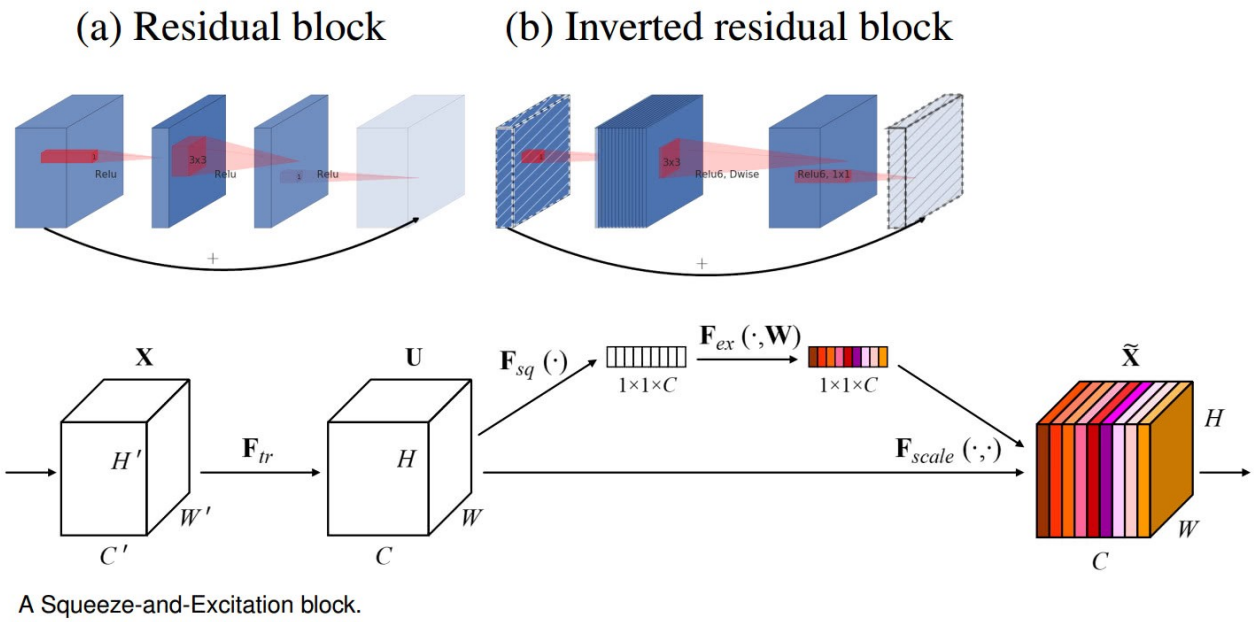
In this tutorial, you will learn how to create an image classification neural network to classify your custom images. The network will be based on the latest EfficientNet, which has achieved state of the art accuracy on ImageNet while being 8.4x smaller and 6.1x faster.

Why EfficientNet?

Compared to other models achieving similar ImageNet accuracy, EfficientNet is much smaller. For example, the ResNet50 model as you can see in Keras application has 23,534,592 parameters in total, and even though, it still underperforms the smallest EfficientNet, which only takes 5,330,564 parameters in total.

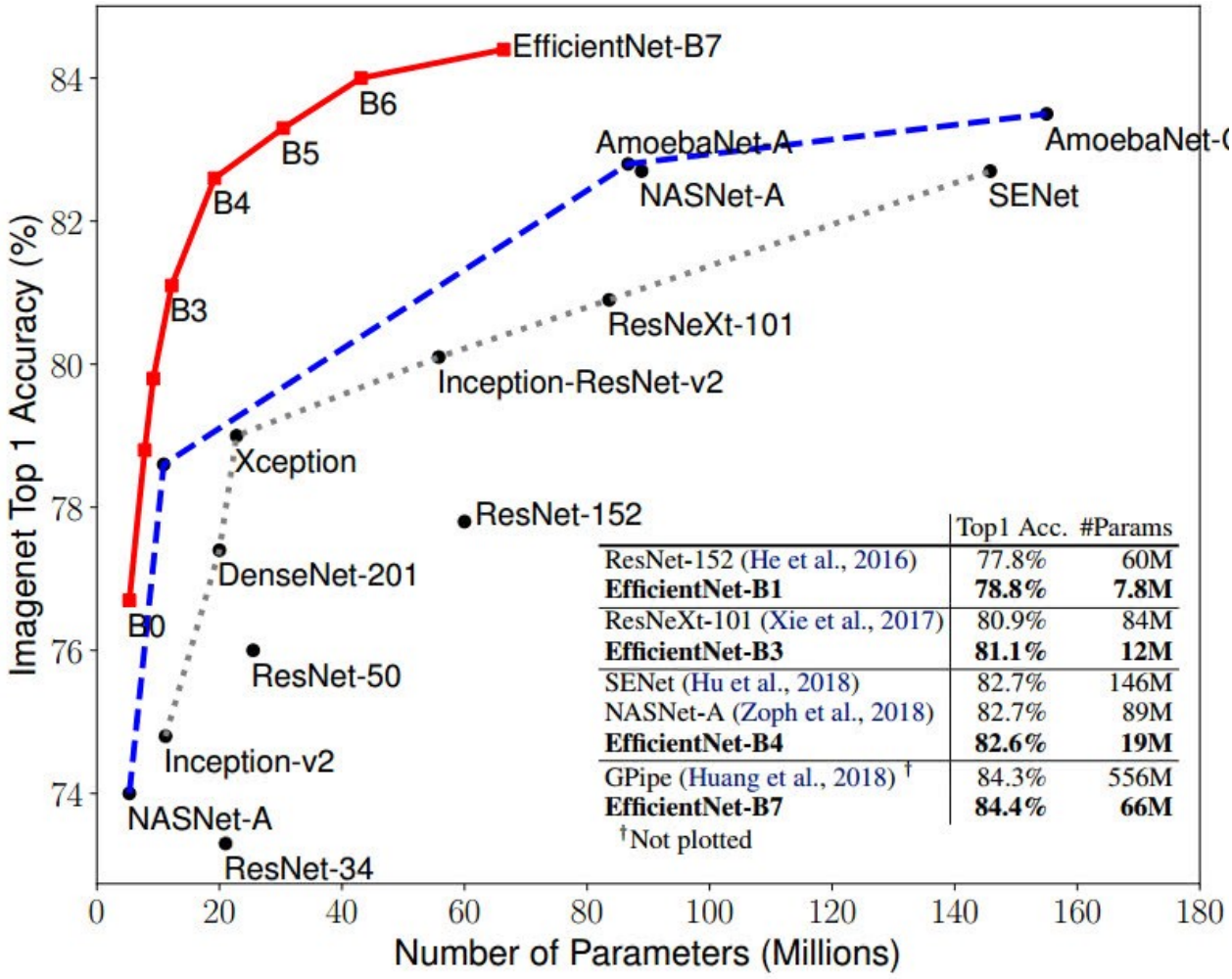
Why is it so efficient? To answer the question, we will dive into its base model and building block. You might have heard of the building block for the classical ResNet model is identity and convolution block.

For EfficientNet, its main building block is mobile, combined with **inverted bottleneck** MBConv, which was first introduced in MobileNetV2. By using shortcuts directly between the bottlenecks which connects a much fewer number of channels compared to expansion layers **depthwise separable convolution** which effectively reduces computation by almost a factor of k^2, compared to traditional layers. Where k stands for the kernel size, specifying the height and width of the 2D convolution window.



A Squeeze-and-Excitation block.

The authors also add squeeze-and-excitation(SE) optimization, which contributes to further performance improvements. The second benefit of EfficientNet, it scales more efficiently by carefully balancing network depth, width, and resolution, which lead to better performance.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**78.8%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.1%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.6%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

As you can see, starting from the smallest EfficientNet configuration B0 to the largest B7, accuracies are steady increasing while maintaining a relatively small size.

Transfer Learning with EfficientNet

It is fine if you are not entirely sure what I am talking about in the previous section. Transfer learning for image classification is more or less model agnostic. You can pick any other pre-trained ImageNet model such as MobileNetV2 or ResNet50 as a drop-in replacement if you want.

A pre-trained network is simply a saved network previously trained on a large dataset such as ImageNet. The learned features can prove useful for many different computer vision problems, even though these new problems might involve completely different classes from those of the original task. For instance, one might train a network on ImageNet (where classes are mostly animals and everyday objects) and then re-purpose this trained network for something as remote as identifying the car models in images. For this tutorial, we expect the model to perform well on our cat vs. dog classification problem with a relatively small number of samples.

The easiest way to get started is by opening this notebook in Colab, while I will explain more detail here in this post.

First clone my repository which contains the Tensorflow Keras implementation of the EfficientNet, then cd into the directory.

```
!git clone https://github.com/Tony607/efficientnet_keras_transfer_learning
%cd efficientnet_keras_transfer_learning/
```

The EfficientNet is built for ImageNet classification which contains 1000 classes labels. For our dataset, we only have 2. Which means the last few layers for classification is not useful for us. They can be excluded while loading the model by specifying the `include_top` argument to False, and this applies to other ImageNet models made available in Keras applications as well.

```
# Options: EfficientNetB0, EfficientNetB1, EfficientNetB2, EfficientNetB3
# Higher the number, the more complex the model is.
from efficientnet import EfficientNetB0 as Net
from efficientnet import center_crop_and_resize, preprocess_input
# loading pretrained conv base model
conv_base = Net(weights="imagenet", include_top=False, input_shape=input_shape)
```

To create our own classification layers stack on top of the EfficientNet convolutional base model. We adopt `GlobalMaxPooling2D` to convert 4D the `(batch_size, rows, cols, channels)` tensor into 2D tensor with shape `(batch_size, channels)`. `GlobalMaxPooling2D` results in a much smaller number of features compared to the `Flatten` layer, which effectively reduces the number of parameters.

```
from tensorflow.keras import models
from tensorflow.keras import layers
dropout_rate = 0.2
model = models.Sequential()
model.add(conv_base)
model.add(layers.GlobalMaxPooling2D(name="gap"))
# model.add(layers.Flatten(name="flatten"))
if dropout_rate > 0:
    model.add(layers.Dropout(dropout_rate, name="dropout_out"))
# model.add(layers.Dense(256, activation='relu', name="fc1"))
model.add(layers.Dense(2, activation="softmax", name="fc_out"))
```

To keep the convolutional base's weight untouched, we will freeze it, otherwise, the representations previously learned from the ImageNet dataset will be destroyed.

```
conv_base.trainable = False
```

Then you can download and unzip the `dog_vs_cat` data from Microsoft.

```
!wget https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-6DEBA77B919F/kagglecatsanddogs_3367a.zip
!unzip -qq kagglecatsanddogs_3367a.zip -d dog_vs_cat
```

There are several blocks of data in the Notebook dedicated to sample a subset of images from the original dataset to form train/validation/test sets after which you will see.

```
total training cat images: 1000
total training dog images: 1000
total validation cat images: 500
total validation dog images: 500
total test cat images: 500
total test dog images: 500
```

Then you can compile and train the model with Keras's `ImageDataGenerator`, which adds various data augmentation options during the training to reduce the chance of overfitting.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode="nearest",
)
# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1.0 / 255)
train_generator = train_datagen.flow_from_directory(
    # This is the target directory
    train_dir,
    # All images will be resized to target height and width.
    target_size=(height, width),
    batch_size=batch_size,
    # Since we use categorical_crossentropy loss, we need categorical labels
    class_mode="categorical",
)
validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(height, width),
```

```
    batch_size=batch_size,
    class_mode="categorical",
)
model.compile(
    loss="categorical_crossentropy",
    optimizer=optimizers.RMSprop(lr=2e-5),
    metrics=["acc"],
)
history = model.fit_generator(
    train_generator,
    steps_per_epoch=NUM_TRAIN // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=NUM_TEST // batch_size,
    verbose=1,
    use_multiprocessing=True,
    workers=4,
)
```

Another technique to make the model representation more relevant for the problem at hand is called fine-tuning. That is based on the following intuition.

Earlier layers in the convolutional base encode more generic, reusable features, while layers higher up encode more specialized features.

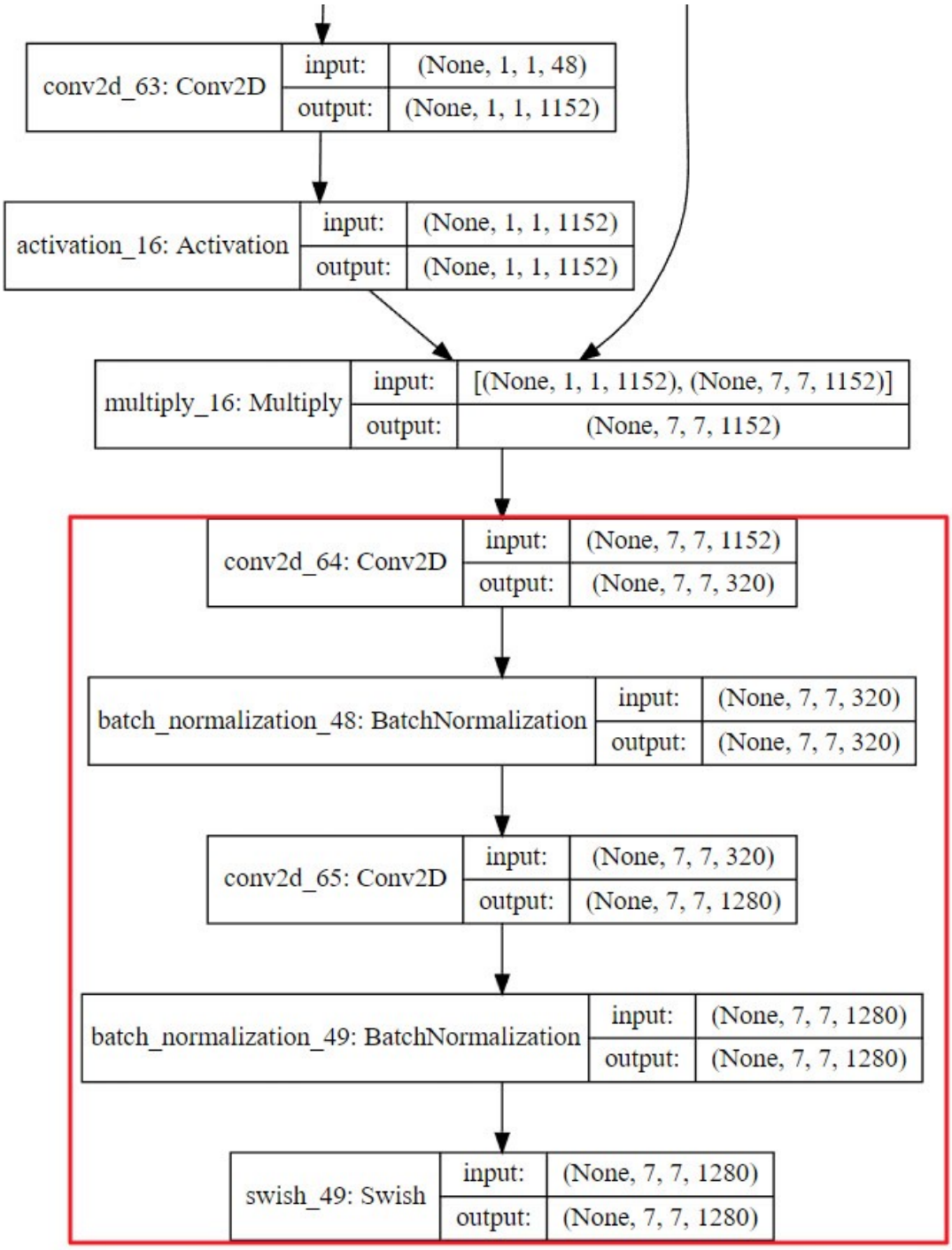
The steps for fine-tuning a network are as follow:

- 1) Add your custom network on top of an already trained base network.
- 2) Freeze the base network.
- 3) Train the part you added.
- 4) Unfreeze some layers in the base network.
- 5) Jointly train both these layers and the part you added.

We have already done the first three steps, to find out which layers to unfreeze, it is helpful to plot the Keras model.

```
from tensorflow.keras.utils import plot_model
plot_model(conv_base, to_file='conv_base.png', show_shapes=True)
from IPython.display import Image
Image(filename='conv_base.png')
```

Here is the zoom in view of the last several layers in the convolutional base model.



To set '`multiply_16`' and successive layers trainable.

```
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'multiply_16':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

Then you can compile and train the model again for some more epochs. Finally, you will have a fine-tuned model with a 9% increase in validation accuracy.

Conclusion and Further reading

This post starts with a brief introduction to EfficientNet and why its more efficient compare to classical ResNet model. An example is made runnable on Colab Notebook showing you how to build a model reusing the convolutional base of EfficientNet and fine-tuning last several layers on the custom dataset.

The full source code is available on my GitHub repo.

You might find the following resources helpful.

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

MobileNetV2: Inverted Residuals and Linear Bottlenecks

Squeeze-and-Excitation Networks

TensorFlow implementation of EfficientNet

Originally published at https://www.dlology.com.