


Keras

 Star 57,369

About Keras

Getting started

Developer guides

Keras API reference

Code examples

Computer Vision

Image classification from scratch

Simple MNIST convnet

Image classification via fine-tuning with EfficientNet

Image classification with Vision Transformer

Image Classification using BigTransfer (BiT)

Classification using Attention-based Deep Multiple Instance Learning

Image classification with modern MLP models

A mobile-friendly Transformer-based model for image classification

Pneumonia Classification on TPU

Compact Convolutional Transformers

Image classification with ConvMixer

Image classification with EANet (External Attention Transformer)

Involutional neural networks

Image classification with Perceiver

Few-Shot learning with Reptile

Semi-supervised image classification using contrastive pretraining with SimCLR

Image classification with Swin Transformers

Train a Vision Transformer on small datasets

A Vision Transformer without Attention

Image segmentation with a U-Net-like architecture

Multiclass semantic segmentation using DeepLabV3+

Object Detection with RetinaNet

Keypoint Detection with Transfer Learning

Object detection with Vision Transformers

3D image classification from CT scans

Monocular depth estimation

» [Code examples](#) / [Computer Vision](#) / Image classification via fine-tuning with EfficientNet


Image classification via fine-tuning with EfficientNet

Author: [Yixing Fu](#)


Date created: 2020/06/30

Last modified: 2020/07/16

Description: Use EfficientNet with weights pre-trained on imagenet for Stanford Dogs classification.

 [View in Colab](#)

•

 [GitHub source](#)

Introduction: what is EfficientNet

EfficientNet, first introduced in [Tan and Le, 2019](#) is among the most efficient models (i.e. requiring least FLOPS for inference) that reaches State-of-the-Art accuracy on both imagenet and common image classification transfer learning tasks.

The smallest base model is similar to [MnasNet](#), which reached near-SOTA with a significantly smaller model. By introducing a heuristic way to scale the model, EfficientNet provides a family of models (B0 to B7) that represents a good combination of efficiency and accuracy on a variety of scales. Such a scaling heuristics (compound-scaling, details see [Tan and Le, 2019](#)) allows the efficiency-oriented base model (B0) to surpass models at every scale, while avoiding extensive grid-search of hyperparameters.

A summary of the latest updates on the model is available at [here](#), where various augmentation schemes and semi-supervised learning approaches are applied to further improve the imagenet performance of the models. These extensions of the model can be used by updating weights without changing model architecture.

B0 to B7 variants of EfficientNet

(This section provides some details on "compound scaling", and can be skipped if you're only interested in using the models)

Based on the [original paper](#) people may have the impression that EfficientNet is a continuous family of models created by arbitrarily choosing scaling factor in as Eq.(3) of the paper. However, choice of resolution, depth and width are also restricted by many factors:

- Resolution: Resolutions not divisible by 8, 16, etc. cause zero-padding near boundaries of some layers which wastes computational resources. This especially applies to smaller variants of the model, hence the input resolution for B0 and B1 are chosen as 224 and 240.
- Depth and width: The building blocks of EfficientNet demands channel size to be multiples of 8.
- Resource limit: Memory limitation may bottleneck resolution when depth and width can still increase. In such a situation, increasing depth and/or width but keep resolution can still improve performance.

As a result, the depth, width and resolution of each variant of the EfficientNet models are hand-picked and proven to produce good results, though they may be significantly off from the compound scaling formula. Therefore, the keras implementation (detailed below) only provide these 8 models, B0 to B7, instead of allowing arbitray choice of width / depth / resolution parameters.

Keras implementation of EfficientNet

An implementation of EfficientNet B0 to B7 has been shipped with tf.keras since TF2.3. To use EfficientNetB0 for classifying 1000 classes of images from imagenet, run:

- 3D volumetric rendering with NeRF
- Point cloud classification
- OCR model for reading Captchas
- Handwriting recognition
- Convolutional autoencoder for image denoising
- Low-light image enhancement using MIRNet
- Image Super-Resolution using an Efficient Sub-Pixel CNN
- Enhanced Deep Residual Networks for single-image super-resolution
- Zero-DCE for low-light image enhancement
- CutMix data augmentation for image classification
- MixUp augmentation for image classification
- RandAugment for Image Classification for Improved Robustness
- Image captioning
- Natural language image search with a Dual Encoder
- Visualizing what convnets learn
- Model interpretability with Integrated Gradients
- Investigating Vision Transformer representations
- Grad-CAM class activation visualization
- Near-duplicate image search
- Semantic Image Clustering
- Image similarity estimation using a Siamese Network with a contrastive loss
- Image similarity estimation using a Siamese Network with a triplet loss
- Metric learning for image similarity search
- Metric learning for image similarity search using TensorFlow Similarity
- Video Classification with a CNN-RNN Architecture
- Next-Frame Video Prediction with Convolutional LSTMs
- Video Classification with Transformers
- Video Vision Transformer
- Semi-supervision and domain adaptation with AdaMatch
- Barlow Twins for Contrastive SSL
- Class Attention Image Transformers with LayerScale
- Consistency training with supervision
- Distilling Vision Transformers
- FixRes: Fixing train-test resolution discrepancy

```
from tensorflow.keras.applications import EfficientNetB0
model = EfficientNetB0(weights='imagenet')
```

This model takes input images of shape (224, 224, 3), and the input data should range [0, 255]. Normalization is included as part of the model.

Because training EfficientNet on ImageNet takes a tremendous amount of resources and several techniques that are not a part of the model architecture itself. Hence the Keras implementation by default loads pre-trained weights obtained via training with [AutoAugment](#).

For B0 to B7 base models, the input shapes are different. Here is a list of input shape expected for each model:

Base model	resolution
EfficientNetB0	224
EfficientNetB1	240
EfficientNetB2	260
EfficientNetB3	300
EfficientNetB4	380
EfficientNetB5	456
EfficientNetB6	528
EfficientNetB7	600

When the model is intended for transfer learning, the Keras implementation provides a option to remove the top layers:

```
model = EfficientNetB0(include_top=False, weights='imagenet')
```

This option excludes the final **Dense** layer that turns 1280 features on the penultimate layer into prediction of the 1000 ImageNet classes. Replacing the top layer with custom layers allows using EfficientNet as a feature extractor in a transfer learning workflow.

Another argument in the model constructor worth noticing is **drop_connect_rate** which controls the dropout rate responsible for [stochastic depth](#). This parameter serves as a toggle for extra regularization in finetuning, but does not affect loaded weights. For example, when stronger regularization is desired, try:

```
model = EfficientNetB0(weights='imagenet', drop_connect_rate=0.4)
```

The default value is 0.2.

Example: EfficientNetB0 for Stanford Dogs.

EfficientNet is capable of a wide range of image classification tasks. This makes it a good model for transfer learning. As an end-to-end example, we will show using pre-trained EfficientNetB0 on [Stanford Dogs](#) dataset.

```
# IMG_SIZE is determined by EfficientNet model choice
IMG_SIZE = 224
```

Setup and data loading

This example requires TensorFlow 2.3 or above.

To use TPU, the TPU runtime must match current running TensorFlow version. If there is a mismatch, try:

Focal Modulation: A replacement for Self-Attention
Using the Forward-Forward Algorithm for Image Classification
Gradient Centralization for Better Training Performance
Knowledge Distillation
Learning to Resize in Computer Vision
Masked image modeling with Autoencoders
Self-supervised contrastive learning with NNCLR
Augmenting convnets with aggregated attention
Point cloud segmentation with PointNet
Semantic segmentation with SegFormer and Hugging Face Transformers
Self-supervised contrastive learning with SimSiam
Supervised Contrastive Learning
Learning to tokenize in Vision Transformers
Natural Language Processing
Structured Data
Timeseries
Generative Deep Learning
Audio Data
Reinforcement Learning
Graph Data
Quick Keras Recipes

Why choose Keras?
Community & governance
Contributing to Keras
KerasTuner
KerasCV
KerasNLP

```
from cloud_tpu_client import Client
c = Client()
c.configure_tpu_version(tf.__version__, restart_type="always")
```

```
import tensorflow as tf

try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver.connect()
    print("Device:", tpu.master())
    strategy = tf.distribute.TPUStrategy(tpu)
except ValueError:
    print("Not connected to a TPU runtime. Using CPU/GPU strategy")
    strategy = tf.distribute.MirroredStrategy()
```

```
Not connected to a TPU runtime. Using CPU/GPU strategy
INFO:tensorflow:Using MirroredStrategy with devices
('/job:localhost/replica:0/task:0/device:GPU:0',)
```

Loading data

Here we load data from [tensorflow datasets](#) (hereafter TFDS). Stanford Dogs dataset is provided in TFDS as [stanford_dogs](#). It features 20,580 images that belong to 120 classes of dog breeds (12,000 for training and 8,580 for testing).

By simply changing `dataset_name` below, you may also try this notebook for other datasets in TFDS such as [cifar10](#), [cifar100](#), [food101](#), etc. When the images are much smaller than the size of EfficientNet input, we can simply upsample the input images. It has been shown in [Tan and Le, 2019](#) that transfer learning result is better for increased resolution even if input images remain small.

For TPU: if using TFDS datasets, a [GCS bucket](#) location is required to save the datasets. For example:

```
tfds.load(dataset_name, data_dir="gs://example-bucket/datapath")
```

Also, both the current environment and the TPU service account have proper [access](#) to the bucket. Alternatively, for small datasets you may try loading data into the memory and use `tf.data.Dataset.from_tensor_slices()`.

```
import tensorflow_datasets as tfds

batch_size = 64

dataset_name = "stanford_dogs"
(ds_train, ds_test), ds_info = tfds.load(
    dataset_name, split=["train", "test"], with_info=True, as_supervised=True
)
NUM_CLASSES = ds_info.features["label"].num_classes
```

When the dataset include images with various size, we need to resize them into a shared size. The Stanford Dogs dataset includes only images at least 200x200 pixels in size. Here we resize the images to the input size needed for EfficientNet.

```
size = (IMG_SIZE, IMG_SIZE)
ds_train = ds_train.map(lambda image, label: (tf.image.resize(image, size), label))
ds_test = ds_test.map(lambda image, label: (tf.image.resize(image, size), label))
```

Visualizing the data

The following code shows the first 9 images with their labels.

```
import matplotlib.pyplot as plt

def format_label(label):
    string_label = label_info.int2str(label)
    return string_label.split("-")[1]

label_info = ds_info.features["label"]
for i, (image, label) in enumerate(ds_train.take(9)):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(image.numpy().astype("uint8"))
    plt.title("{}".format(format_label(label)))
    plt.axis("off")
```



Data augmentation

We can use the preprocessing layers APIs for image augmentation.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

img_augmentation = Sequential(
    [
        layers.RandomRotation(factor=0.15),
        layers.RandomTranslation(height_factor=0.1, width_factor=0.1),
        layers.RandomFlip(),
        layers.RandomContrast(factor=0.1),
    ],
    name="img_augmentation",
)
```

This `Sequential` model object can be used both as a part of the model we later build, and as a function to preprocess data before feeding into the model. Using them as function makes it easy to visualize the augmented images. Here we plot 9 examples of augmentation result of a given figure.

```
for image, label in ds_train.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        aug_img = img_augmentation(tf.expand_dims(image, axis=0))
        plt.imshow(aug_img[0].numpy().astype("uint8"))
        plt.title("{}".format(format_label(label)))
        plt.axis("off")
```



Prepare inputs

Once we verify the input data and augmentation are working correctly, we prepare dataset for training. The input data are resized to uniform **IMG_SIZE**. The labels are put into one-hot (a.k.a. categorical) encoding. The dataset is batched.

Note: **prefetch** and **AUTOTUNE** may in some situation improve performance, but depends on environment and the specific dataset used. See this [guide](#) for more information on data pipeline performance.

```
# One-hot / categorical encoding
def input_preprocess(image, label):
    label = tf.one_hot(label, NUM_CLASSES)
    return image, label

ds_train = ds_train.map(
    input_preprocess, num_parallel_calls=tf.data.AUTOTUNE
)
ds_train = ds_train.batch(batch_size=batch_size, drop_remainder=True)
ds_train = ds_train.prefetch(tf.data.AUTOTUNE)

ds_test = ds_test.map(input_preprocess)
ds_test = ds_test.batch(batch_size=batch_size, drop_remainder=True)
```

Training a model from scratch

We build an EfficientNetB0 with 120 output classes, that is initialized from scratch:

Note: the accuracy will increase very slowly and may overfit.

```
from tensorflow.keras.applications import EfficientNetB0

with strategy.scope():
    inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
    x = img_augmentation(inputs)
    outputs = EfficientNetB0(include_top=True, weights=None, classes=NUM_CLASSES)(x)

    model = tf.keras.Model(inputs, outputs)
    model.compile(
        optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"]
    )

model.summary()

epochs = 40 # @param {type: "slider", min:10, max:100}
hist = model.fit(ds_train, epochs=epochs, validation_data=ds_test, verbose=2)
```

```
Model: "functional_1"

Layer (type)                 Output Shape              Param #
=====
input_1 (InputLayer)         [(None, 224, 224, 3)]    0
img_augmentation (Sequential (None, 224, 224, 3)    0
efficientnetb0 (Functional)  (None, 120)              4203291
=====
Total params: 4,203,291
Trainable params: 4,161,268
Non-trainable params: 42,023

Epoch 1/40
187/187 - 66s - loss: 4.9221 - accuracy: 0.0119 - val_loss: 4.9835 - val_accuracy: 0.0104
Epoch 2/40
187/187 - 63s - loss: 4.5652 - accuracy: 0.0243 - val_loss: 5.1626 - val_accuracy: 0.0145
Epoch 3/40
187/187 - 63s - loss: 4.4179 - accuracy: 0.0337 - val_loss: 4.7597 - val_accuracy: 0.0237
Epoch 4/40
187/187 - 63s - loss: 4.2964 - accuracy: 0.0421 - val_loss: 4.4028 - val_accuracy: 0.0378
Epoch 5/40
187/187 - 63s - loss: 4.1951 - accuracy: 0.0540 - val_loss: 4.3048 - val_accuracy: 0.0443
Epoch 6/40
187/187 - 63s - loss: 4.1025 - accuracy: 0.0596 - val_loss: 4.1918 - val_accuracy: 0.0526
Epoch 7/40
187/187 - 63s - loss: 4.0157 - accuracy: 0.0728 - val_loss: 4.1482 - val_accuracy: 0.0591
Epoch 8/40
187/187 - 62s - loss: 3.9344 - accuracy: 0.0844 - val_loss: 4.1088 - val_accuracy: 0.0638
Epoch 9/40
187/187 - 63s - loss: 3.8529 - accuracy: 0.0951 - val_loss: 4.0692 - val_accuracy: 0.0770
Epoch 10/40
187/187 - 63s - loss: 3.7650 - accuracy: 0.1040 - val_loss: 4.1468 - val_accuracy: 0.0719
Epoch 11/40
187/187 - 63s - loss: 3.6858 - accuracy: 0.1185 - val_loss: 4.0484 - val_accuracy: 0.0913
Epoch 12/40
187/187 - 63s - loss: 3.5942 - accuracy: 0.1326 - val_loss: 3.8047 - val_accuracy: 0.1072
Epoch 13/40
187/187 - 63s - loss: 3.5028 - accuracy: 0.1447 - val_loss: 3.9513 - val_accuracy: 0.0933
Epoch 14/40
187/187 - 63s - loss: 3.4295 - accuracy: 0.1604 - val_loss: 3.7738 - val_accuracy: 0.1220
Epoch 15/40
187/187 - 63s - loss: 3.3410 - accuracy: 0.1735 - val_loss: 3.9104 - val_accuracy: 0.1104
Epoch 16/40
187/187 - 63s - loss: 3.2511 - accuracy: 0.1890 - val_loss: 3.6904 - val_accuracy: 0.1264
Epoch 17/40
187/187 - 63s - loss: 3.1624 - accuracy: 0.2076 - val_loss: 3.4026 - val_accuracy: 0.1769
Epoch 18/40
187/187 - 63s - loss: 3.0825 - accuracy: 0.2229 - val_loss: 3.4627 - val_accuracy: 0.1744
Epoch 19/40
187/187 - 63s - loss: 3.0041 - accuracy: 0.2355 - val_loss: 3.6061 - val_accuracy: 0.1542
Epoch 20/40
187/187 - 64s - loss: 2.8945 - accuracy: 0.2552 - val_loss: 3.2769 - val_accuracy: 0.2036
Epoch 21/40
187/187 - 63s - loss: 2.8054 - accuracy: 0.2710 - val_loss: 3.5355 - val_accuracy: 0.1834
Epoch 22/40
187/187 - 63s - loss: 2.7342 - accuracy: 0.2904 - val_loss: 3.3540 - val_accuracy:
```

```
0.1973
Epoch 23/40
187/187 - 62s - loss: 2.6258 - accuracy: 0.3042 - val_loss: 3.2608 - val_accuracy:
0.2217
Epoch 24/40
187/187 - 62s - loss: 2.5453 - accuracy: 0.3218 - val_loss: 3.4611 - val_accuracy:
0.1941
Epoch 25/40
187/187 - 63s - loss: 2.4585 - accuracy: 0.3356 - val_loss: 3.4163 - val_accuracy:
0.2070
Epoch 26/40
187/187 - 62s - loss: 2.3606 - accuracy: 0.3647 - val_loss: 3.2558 - val_accuracy:
0.2392
Epoch 27/40
187/187 - 63s - loss: 2.2819 - accuracy: 0.3801 - val_loss: 3.3676 - val_accuracy:
0.2222
Epoch 28/40
187/187 - 62s - loss: 2.2114 - accuracy: 0.3933 - val_loss: 3.6578 - val_accuracy:
0.2022
Epoch 29/40
187/187 - 62s - loss: 2.0964 - accuracy: 0.4215 - val_loss: 3.5366 - val_accuracy:
0.2186
Epoch 30/40
187/187 - 63s - loss: 1.9931 - accuracy: 0.4459 - val_loss: 3.5612 - val_accuracy:
0.2310
Epoch 31/40
187/187 - 63s - loss: 1.8924 - accuracy: 0.4657 - val_loss: 3.4780 - val_accuracy:
0.2359
Epoch 32/40
187/187 - 63s - loss: 1.8095 - accuracy: 0.4874 - val_loss: 3.5776 - val_accuracy:
0.2403
Epoch 33/40
187/187 - 63s - loss: 1.7126 - accuracy: 0.5086 - val_loss: 3.6865 - val_accuracy:
0.2316
Epoch 34/40
187/187 - 63s - loss: 1.6117 - accuracy: 0.5373 - val_loss: 3.6419 - val_accuracy:
0.2513
Epoch 35/40
187/187 - 63s - loss: 1.5532 - accuracy: 0.5514 - val_loss: 3.8050 - val_accuracy:
0.2415
Epoch 36/40
187/187 - 63s - loss: 1.4479 - accuracy: 0.5809 - val_loss: 4.0113 - val_accuracy:
0.2299
Epoch 37/40
187/187 - 62s - loss: 1.3885 - accuracy: 0.5939 - val_loss: 4.1262 - val_accuracy:
0.2158
Epoch 38/40
187/187 - 63s - loss: 1.2979 - accuracy: 0.6217 - val_loss: 4.2519 - val_accuracy:
0.2344
Epoch 39/40
187/187 - 62s - loss: 1.2066 - accuracy: 0.6413 - val_loss: 4.3924 - val_accuracy:
0.2169
Epoch 40/40
187/187 - 62s - loss: 1.1348 - accuracy: 0.6618 - val_loss: 4.2216 - val_accuracy:
0.2374
```

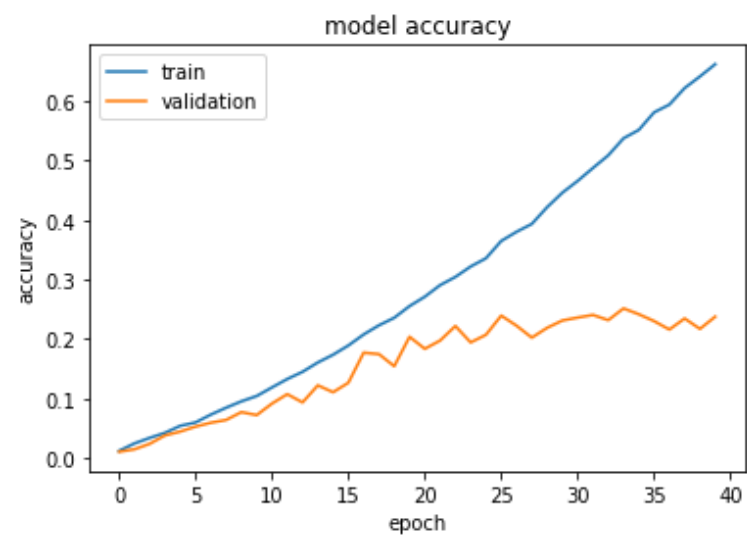
Training the model is relatively fast (takes only 20 seconds per epoch on TPUv2 that is available on Colab). This might make it sounds easy to simply train EfficientNet on any dataset wanted from scratch. However, training EfficientNet on smaller datasets, especially those with lower resolution like CIFAR-100, faces the significant challenge of overfitting.

Hence training from scratch requires very careful choice of hyperparameters and is difficult to find suitable regularization. It would also be much more demanding in resources. Plotting the training and validation accuracy makes it clear that validation accuracy stagnates at a low value.

```
import matplotlib.pyplot as plt

def plot_hist(hist):
    plt.plot(hist.history["accuracy"])
    plt.plot(hist.history["val_accuracy"])
    plt.title("model accuracy")
    plt.ylabel("accuracy")
    plt.xlabel("epoch")
    plt.legend(["train", "validation"], loc="upper left")
    plt.show()
```

```
plot_hist(hist)
```



Transfer learning from pre-trained weights

Here we initialize the model with pre-trained ImageNet weights, and we fine-tune it on our own dataset.

```
def build_model(num_classes):
    inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
    x = img_augmentation(inputs)
    model = EfficientNetB0(include_top=False, input_tensor=x, weights="imagenet")

    # Freeze the pretrained weights
    model.trainable = False

    # Rebuild top
    x = layers.GlobalAveragePooling2D(name="avg_pool")(model.output)
    x = layers.BatchNormalization()(x)

    top_dropout_rate = 0.2
    x = layers.Dropout(top_dropout_rate, name="top_dropout")(x)
    outputs = layers.Dense(NUM_CLASSES, activation="softmax", name="pred")(x)

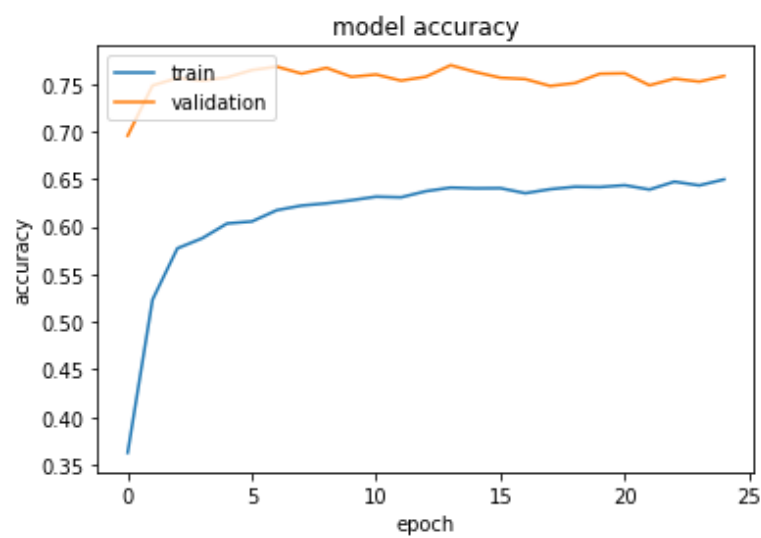
    # Compile
    model = tf.keras.Model(inputs, outputs, name="EfficientNet")
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-2)
    model.compile(
        optimizer=optimizer, loss="categorical_crossentropy", metrics=["accuracy"]
    )
    return model
```

The first step to transfer learning is to freeze all layers and train only the top layers. For this step, a relatively large learning rate (1e-2) can be used. Note that validation accuracy and loss will usually be better than training accuracy and loss. This is because the regularization is strong, which only suppresses training-time metrics.

Note that the convergence may take up to 50 epochs depending on choice of learning rate. If image augmentation layers were not applied, the validation accuracy may only reach ~60%.


```
with strategy.scope():  
    model = build_model(num_classes=NUM_CLASSES)  
  
epochs = 25 # @param {type: "slider", min:8, max:80}  
hist = model.fit(ds_train, epochs=epochs, validation_data=ds_test, verbose=2)  
plot_hist(hist)
```

Epoch 1/25
187/187 - 33s - loss: 3.5673 - accuracy: 0.3624 - val_loss: 1.0288 - val_accuracy: 0.6957
Epoch 2/25
187/187 - 31s - loss: 1.8503 - accuracy: 0.5232 - val_loss: 0.8439 - val_accuracy: 0.7484
Epoch 3/25
187/187 - 31s - loss: 1.5511 - accuracy: 0.5772 - val_loss: 0.7953 - val_accuracy: 0.7563
Epoch 4/25
187/187 - 31s - loss: 1.4660 - accuracy: 0.5878 - val_loss: 0.8061 - val_accuracy: 0.7535
Epoch 5/25
187/187 - 31s - loss: 1.4143 - accuracy: 0.6034 - val_loss: 0.7850 - val_accuracy: 0.7569
Epoch 6/25
187/187 - 31s - loss: 1.4000 - accuracy: 0.6054 - val_loss: 0.7846 - val_accuracy: 0.7646
Epoch 7/25
187/187 - 31s - loss: 1.3678 - accuracy: 0.6173 - val_loss: 0.7850 - val_accuracy: 0.7682
Epoch 8/25
187/187 - 31s - loss: 1.3286 - accuracy: 0.6222 - val_loss: 0.8142 - val_accuracy: 0.7608
Epoch 9/25
187/187 - 31s - loss: 1.3210 - accuracy: 0.6245 - val_loss: 0.7890 - val_accuracy: 0.7669
Epoch 10/25
187/187 - 31s - loss: 1.3086 - accuracy: 0.6278 - val_loss: 0.8368 - val_accuracy: 0.7575
Epoch 11/25
187/187 - 31s - loss: 1.2877 - accuracy: 0.6315 - val_loss: 0.8309 - val_accuracy: 0.7599
Epoch 12/25
187/187 - 31s - loss: 1.2918 - accuracy: 0.6308 - val_loss: 0.8319 - val_accuracy: 0.7535
Epoch 13/25
187/187 - 31s - loss: 1.2738 - accuracy: 0.6373 - val_loss: 0.8567 - val_accuracy: 0.7576
Epoch 14/25
187/187 - 31s - loss: 1.2837 - accuracy: 0.6410 - val_loss: 0.8004 - val_accuracy: 0.7697
Epoch 15/25
187/187 - 31s - loss: 1.2828 - accuracy: 0.6403 - val_loss: 0.8364 - val_accuracy: 0.7625
Epoch 16/25
187/187 - 31s - loss: 1.2749 - accuracy: 0.6405 - val_loss: 0.8558 - val_accuracy: 0.7565
Epoch 17/25
187/187 - 31s - loss: 1.3022 - accuracy: 0.6352 - val_loss: 0.8361 - val_accuracy: 0.7551
Epoch 18/25
187/187 - 31s - loss: 1.2848 - accuracy: 0.6394 - val_loss: 0.8958 - val_accuracy: 0.7479
Epoch 19/25
187/187 - 31s - loss: 1.2791 - accuracy: 0.6420 - val_loss: 0.8875 - val_accuracy: 0.7509
Epoch 20/25
187/187 - 30s - loss: 1.2834 - accuracy: 0.6416 - val_loss: 0.8653 - val_accuracy: 0.7607
Epoch 21/25
187/187 - 30s - loss: 1.2608 - accuracy: 0.6435 - val_loss: 0.8451 - val_accuracy: 0.7612
Epoch 22/25
187/187 - 30s - loss: 1.2780 - accuracy: 0.6390 - val_loss: 0.9035 - val_accuracy: 0.7486
Epoch 23/25
187/187 - 30s - loss: 1.2742 - accuracy: 0.6473 - val_loss: 0.8837 - val_accuracy: 0.7556
Epoch 24/25
187/187 - 30s - loss: 1.2609 - accuracy: 0.6434 - val_loss: 0.9233 - val_accuracy: 0.7524
Epoch 25/25
187/187 - 31s - loss: 1.2630 - accuracy: 0.6496 - val_loss: 0.9116 - val_accuracy: 0.7584



The second step is to unfreeze a number of layers and fit the model using smaller learning rate. In this example we show unfreezing all layers, but depending on specific dataset it may be desirable to only unfreeze a fraction of all layers.

When the feature extraction with pretrained model works good enough, this step would give a very limited gain on validation accuracy. In our case we only see a small improvement, as ImageNet pretraining already exposed the model to a good amount of dogs.

On the other hand, when we use pretrained weights on a dataset that is more different from ImageNet, this fine-tuning step can be crucial as the feature extractor also needs to be adjusted by a considerable amount. Such a situation can be demonstrated if choosing CIFAR-100 dataset instead, where fine-tuning boosts validation accuracy by about 10% to pass 80% on **EfficientNetB0**. In such a case the convergence may take more than 50 epochs.

A side note on freezing/unfreezing models: setting **trainable** of a **Model** will simultaneously set all layers belonging to the **Model** to the same **trainable** attribute. Each layer is trainable only if both the layer itself and the model containing it are trainable. Hence when we need to partially freeze/unfreeze a model, we need to make sure the **trainable** attribute of the model is set to **True**.

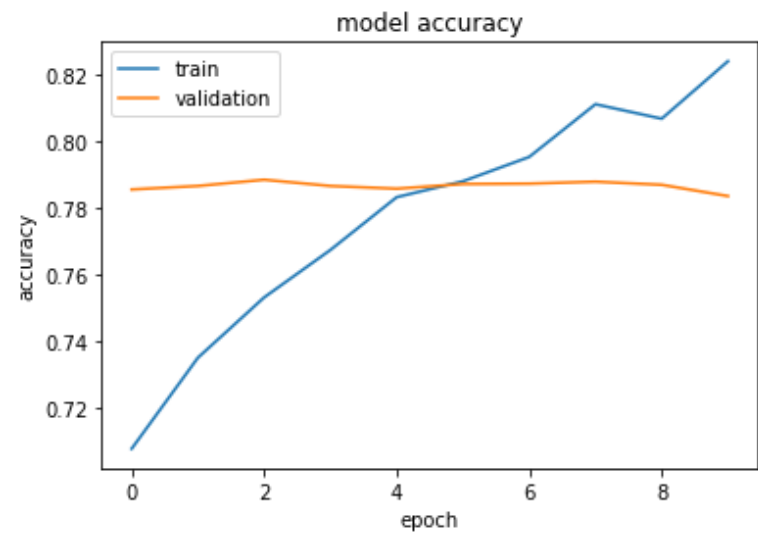
```
def unfreeze_model(model):
    # We unfreeze the top 20 layers while leaving BatchNorm layers frozen
    for layer in model.layers[-20:]:
        if not isinstance(layer, layers.BatchNormalization):
            layer.trainable = True

    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
    model.compile(
        optimizer=optimizer, loss="categorical_crossentropy", metrics=["accuracy"]
    )

unfreeze_model(model)

epochs = 10 # @param {type: "slider", min:8, max:50}
hist = model.fit(ds_train, epochs=epochs, validation_data=ds_test, verbose=2)
plot_hist(hist)
```

```
Epoch 1/10
187/187 - 33s - loss: 0.9956 - accuracy: 0.7080 - val_loss: 0.7644 - val_accuracy:
0.7856
Epoch 2/10
187/187 - 31s - loss: 0.8885 - accuracy: 0.7352 - val_loss: 0.7696 - val_accuracy:
0.7866
Epoch 3/10
187/187 - 31s - loss: 0.8059 - accuracy: 0.7533 - val_loss: 0.7659 - val_accuracy:
0.7885
Epoch 4/10
187/187 - 32s - loss: 0.7648 - accuracy: 0.7675 - val_loss: 0.7730 - val_accuracy:
0.7866
Epoch 5/10
187/187 - 32s - loss: 0.6982 - accuracy: 0.7833 - val_loss: 0.7691 - val_accuracy:
0.7858
Epoch 6/10
187/187 - 31s - loss: 0.6823 - accuracy: 0.7880 - val_loss: 0.7814 - val_accuracy:
0.7872
Epoch 7/10
187/187 - 31s - loss: 0.6536 - accuracy: 0.7953 - val_loss: 0.7850 - val_accuracy:
0.7873
Epoch 8/10
187/187 - 31s - loss: 0.6104 - accuracy: 0.8111 - val_loss: 0.7774 - val_accuracy:
0.7879
Epoch 9/10
187/187 - 32s - loss: 0.5990 - accuracy: 0.8067 - val_loss: 0.7925 - val_accuracy:
0.7870
Epoch 10/10
187/187 - 31s - loss: 0.5531 - accuracy: 0.8239 - val_loss: 0.7870 - val_accuracy:
0.7836
```



Tips for fine tuning EfficientNet

On unfreezing layers:

- The **BatchNormalization** layers need to be kept frozen ([more details](#)). If they are also turned to trainable, the first epoch after unfreezing will significantly reduce accuracy.
- In some cases it may be beneficial to open up only a portion of layers instead of unfreezing all. This will make fine tuning much faster when going to larger models like B7.
- Each block needs to be all turned on or off. This is because the architecture includes a shortcut from the first layer to the last layer for each block. Not respecting blocks also significantly harms the final performance.

Some other tips for utilizing EfficientNet:

- Larger variants of EfficientNet do not guarantee improved performance, especially for tasks with less data or fewer classes. In such a case, the larger variant of EfficientNet chosen, the harder it is to tune hyperparameters.
- EMA (Exponential Moving Average) is very helpful in training EfficientNet from scratch, but not so much for transfer learning.
- Do not use the RMSprop setup as in the original paper for transfer learning. The momentum and learning rate are too high for transfer learning. It will easily corrupt the pretrained weight and blow up the loss. A quick check is to see if loss (as categorical cross entropy) is getting significantly larger than $\log(\text{NUM_CLASSES})$ after the same epoch. If so, the initial learning rate/momentum is too high.
- Smaller batch size benefit validation accuracy, possibly due to effectively providing regularization.

Using the latest EfficientNet weights

Since the initial paper, the EfficientNet has been improved by various methods for data preprocessing and for using unlabelled data to enhance learning results. These improvements are relatively hard and computationally costly to reproduce, and require extra code; but the weights are readily available in the form of TF checkpoint files. The model architecture has not changed, so loading the improved checkpoints is possible.

To use a checkpoint provided at [the official model repository](#), first download the checkpoint. As example, here we download noisy-student version of B1:

```
!wget https://storage.googleapis.com/cloud-tpu-checkpoints/efficientnet\
/noisystudent/noisy_student_efficientnet-b1.tar.gz
!tar -xf noisy_student_efficientnet-b1.tar.gz
```

Then use the script [efficientnet_weight_update_util.py](#) to convert ckpt file to h5 file.

```
!python efficientnet_weight_update_util.py --model b1 --notop --ckpt \
efficientnet-b1/model.ckpt --o efficientnetb1_notop.h5
```

When creating model, use the following to load new weight:

```
model = EfficientNetB1(weights="efficientnetb1_notop.h5", include_top=False)
```

[Terms](#) | [Privacy](#)