Published in Towards Data Science

Kenneth Leung Follow

Dec 27, 2022 · 14 min read · Listen

Practical Guide to Transfer Learning in TensorFlow for Multiclass Image Classification

Clearly-explained step-by-step tutorial for implementing transfer learning in image classification

Photo by Jason Yuen on Unsplash

Often we do not have access to a wealth of labeled data or computing power to build image classification deep learning models from scratch.

Fortunately, transfer learning empowers us to develop robust image classifiers for our specific classification tasks, even if we have limited resources.

In this easy-to-follow walkthrough, we will learn how to leverage pre-trained models as part of transfer learning in TensorFlow to classify images effectively and efficiently.

Table of Contents

The accompanying GitHub repo to this article can be found here.

(1) Motivation and Benefits of Transfer Learning

OPTIONAL READING

Transfer learning is a powerful method that uses pre-trained models as the starting point for creating new models instead of building them from scratch.

These pre-trained models are built by academic or big tech researchers, who develop novel deep-learning model architectures and train them on large datasets with the help of substantial computing power.

Several vital benefits drive the popularity of transfer learning:

- **Save plenty of time and resources** since training a deep learning model from scratch is time-consuming and resource-intensive
- **Better performance** because pre-trained models have already learned the important features after extensive training on large datasets

- **Less labeled data** is required because we can leverage the firm foundation of 'knowledge' that the pre-trained models possess
- **Reduced overfitting** since models built from scratch may be overly sensitive to specific characteristics of smaller training datasets

Photo by Tim Mossholder on Unsplash

(2) About the Dataset

We will work with the Oxford-IIIT pet dataset for this image classification task. It is a 37-category pet image dataset with ~200 images per class.

This dataset is ideal for this tutorial because:

- It is of relatively small size (~800Mb in total)
- It is available for commercial and research purposes under a Creative Commons Attribution-ShareAlike License

- The high number of classes is an excellent way to test out transfer learning for multiclass image classification
- It is one of many ready-to-use datasets in `tensorflow_datasets` (`tfds`) (which we will load directly in this tutorial)

Sample of the Oxford-IIIT pet dataset | Image by author

The following annotations are available for every image:

- species (cat or dog) and breed name
- tight bounding box around the head of the animal
- pixel-level foreground-background segmentation

This tutorial will focus on the **breed name** as the label we want to predict.

If we wish to get the data outside of TensorFlow, we can do either of the following:

- *BitTorrent with Academic Torrents*

- *Direct download of dataset (images.tar.gz) and ground-truth data (annotations.tar.gz)*

(3) Step-by-Step Guide

> Check out the completed notebook to follow along in this walkthrough.

Step 1 — Initial Setup

We will use Google Colab for this tutorial because it grants us free access to GPUs, and the default environment has the necessary Python dependencies.

While extensive computing power is unnecessary, running transfer learning on GPU is still vital because we are working with deep learning models.

> **Important**: Remember to switch the **Hardware Accelerator** to **GPU** from the top menu bar: Runtime > Change runtime type.

Upon launching a new Colab notebook, we import the following packages:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow import keras
from tensorflow.keras import layers
from keras import callbacks
```

Step 2 — Load Data with Train-Validation-Test Split

As mentioned earlier, we will use the TensorFlow Datasets component `tfds` to load the images and corresponding metadata.

```
(train_raw, val_raw, test_raw), ds_info = tfds.load(
                                          name='oxford_iiit_pet',
                                          split=['train[:90%]',
                                                 'train[90%:]',
                                                 'test'],
                                          shuffle_files=True,
                                          as_supervised=True,
                                          with_info=True
                                          )
```

The `tfds.load` function automatically downloads the data from the source and returns it as an `tf.data.Dataset` object.

- `name` : Dataset name in the TensorFlow Datasets collection. In this case, it is named `oxford_iiit_pet` .
- `split` : Parameter specifying the data splits. Since there are only two splits in the original data (train and test), we shall partition 10% of the train set to create a new validation set
- `shuffle_files` : If set to True, the order of input data is shuffled (to promote better model generalization)
- `as_supervised` : If set to True, the returned `tf.data.Dataset` object will have a tuple structure *(image, label)* where the label refers to the pet breed in the image.
- `with_info` : If set to True, the metadata is returned as a `DatasetInfo` object together with the image dataset object

Since we set `with_info=True` , we will get two output objects:

1. A set of three dataset objects (`train_raw` , `val_raw` , and `test_raw`) based on the three splits
2. A `DatasetInfo` object (`ds_info`) comprising metadata of the dataset

> *If we have custom datasets (e.g., CSV files) that we want to load into `tfds` , we can refer to the documentation here and here.*

Step 3 — Explore Data

There are various ways to understand the dataset better.

(i) Metadata

If available, we can take a closer look at the metadata stored in `ds_info` by running it in a cell.

```
ds_info
```

```
tfds.core.DatasetInfo(
    name='oxford_iiit_pet',
    full_name='oxford_iiit_pet/3.2.0',
    description="""
    The Oxford-IIIT pet dataset is a 37 category pet image dataset with roughly 200
    images for each class. The images have large variations in scale, pose and
    lighting. All images have an associated ground truth annotation of breed.
    """,
    homepage='http://www.robots.ox.ac.uk/~vgg/data/pets/',
    data_path='~/tensorflow_datasets/oxford_iiit_pet/3.2.0',
    file_format=tfrecord,
    download_size=773.52 MiB,
    dataset_size=774.69 MiB,
    features=FeaturesDict({
        'file_name': Text(shape=(), dtype=tf.string),
        'image': Image(shape=(None, None, 3), dtype=tf.uint8),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=37),
        'segmentation_mask': Image(shape=(None, None, 1), dtype=tf.uint8),
        'species': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=3669, num_shards=4>,
        'train': <SplitInfo num_examples=3680, num_shards=4>,
    },
    citation="""@InProceedings{parkhi12a,
      author       = "Parkhi, O. M. and Vedaldi, A. and Zisserman, A. and Jawahar, C.~V.",
      title        = "Cats and Dogs",
      booktitle    = "IEEE Conference on Computer Vision and Pattern Recognition",
      year         = "2012",
    }""",
)
```

Dataset information (aka metadata) of the Oxford-IIIT Pet dataset | Image by author

From the output, we can discover information such as the types of image features, number of classes, and size of splits.

To get the number of classes and split sizes directly, we can run the following code:

```
# Get number of classes
num_classes = ds_info.features['label'].num_classes
```

```
print('Number of classes:', num_classes)

# Get split sizes (aka cardinality)
num_train_examples = tf.data.experimental.cardinality(train_raw).numpy()
num_val_examples = tf.data.experimental.cardinality(val_raw).numpy()
num_test_examples = tf.data.experimental.cardinality(test_raw).numpy()

print('Number of training samples:', num_train_examples)
print('Number of validation samples:', num_val_examples)
print('Number of test samples:', num_test_examples)
```

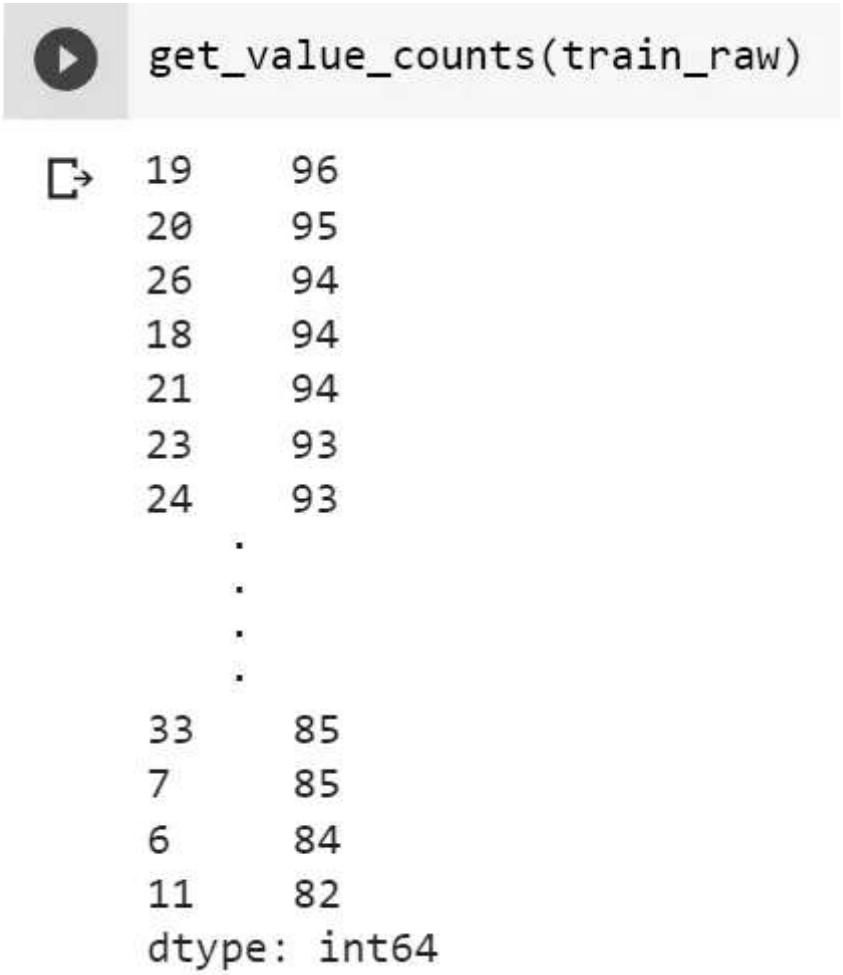
(ii) Get value counts

A quick way to understand the data is to examine the distribution of the image labels. We can do so with a custom value counts function:

```
def get_value_counts(ds):
  label_list = []
  for images, labels in ds:
    label_list.append(labels.numpy()) # Convert tensor to numpy array
  label_counts = pd.Series(label_list).value_counts(sort=True)

  print(label_counts)
```

If we run it on the training set, we can see that the images are relatively evenly distributed across each class.



Value counts of labels in the training set | Image by author

(iii) Image examples

Given that we are dealing with images, it is always a good idea to visualize them directly.

We can display a random image and its corresponding label using the custom function below, including converting labels from their integer form to the actual breed names.

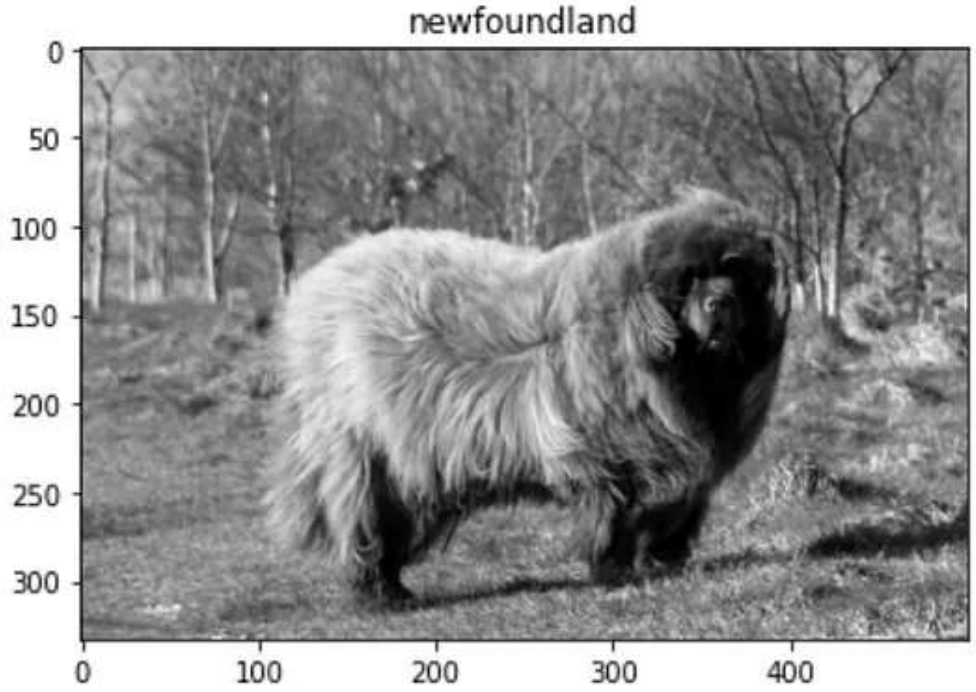
```
# Obtain name for label integer
get_label_name = ds_info.features['label'].int2str

def view_single_image(ds):
  image, label = next(iter(ds)) # Get next image (random)
  print('Image shape: ', image.shape) # Get shape of image
```

```
    plt.imshow(image)
    _ = plt.title(get_label_name(label))
```

```
view_single_image(train_raw)
```

```
Image shape:  (333, 500, 3)
```



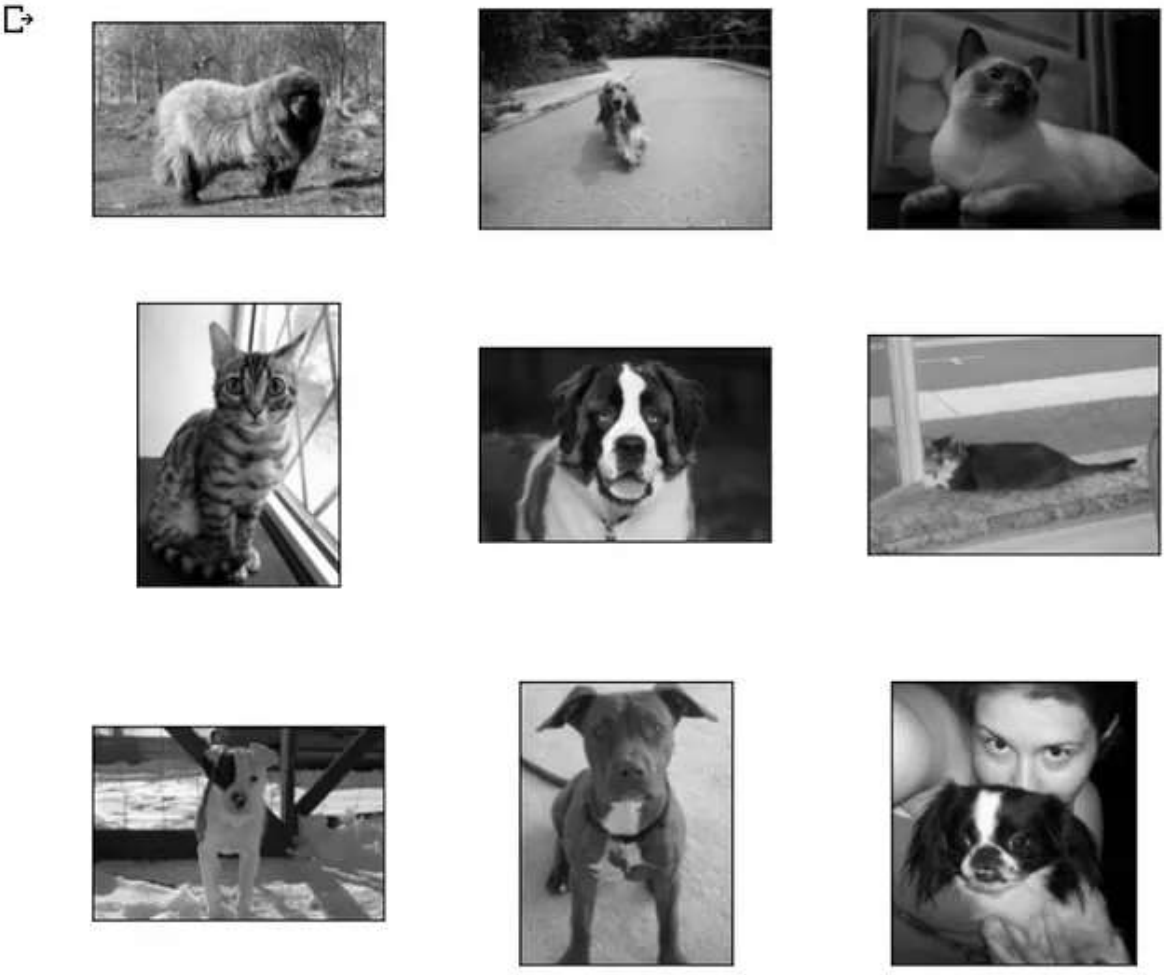
Single random image from the dataset | Image by author

The in-built `show_examples` method also allows us to visualize a random subset of images.

For this particular dataset, we need to specify `image_key='image'` to retrieve the actual images since multiple image features are present.

```
tfds.show_examples(train_raw, ds_info, image_key='image')
```

```
tfds.show_examples(train_raw, ds_info, image_key='image')
```



A random subset of images from tdfs.show_examples| Image by author

Step 4 — Prepare Data

This next step is crucial as it involves preprocessing the images into an appropriate format before transfer learning can occur.

(i) Image Resizing

Deep learning models expect input images to be of the same shape for consistency, compatibility, and training efficiency.

Given that our original images come in different sizes, we need to resize them to have the same length and width.

```
IMG_SIZE = 224
```

```
train_ds = train_raw.map(lambda x, y: (tf.image.resize(x, (IMG_SIZE, IMG_SIZE)),
val_ds = val_raw.map(lambda x, y: (tf.image.resize(x, (IMG_SIZE, IMG_SIZE)), y)
test_ds = test_raw.map(lambda x, y: (tf.image.resize(x, (IMG_SIZE, IMG_SIZE)),
```

The resized resolution of 224x224 is chosen because the pre-trained model we will use later expects input images to be of this particular shape.

(ii) One-hot Encoding of Labels

Our dataset has 37 classes for multiclass image classification. Thus, we one-hot-encode the labels to get an output tensor of length 37 for each class.

Illustration of one-hot encoded output tensor based on the number of classes | Image by author

```
def one_hot_encode(image, label):
    label = tf.one_hot(label, num_classes)
    return image, label

train_ds = train_ds.map(one_hot_encode)
val_ds = val_ds.map(one_hot_encode)
test_ds = test_ds.map(one_hot_encode)
```

This step is important as we will use categorical accuracy to measure model performance, calculating how often the predictions match these actual **one-hot** labels.

(iii) Image Augmentation

While transfer learning reduces the amount of data needed, a dataset of sufficient quality, quantity, and variety is still required for good performance.

Image augmentation is a technique that artificially increases the size of the training set by generating modified copies of the original images.

To perform augmentation, we use the Keras preprocessing layers API. Each type of image augmentation we want to apply is defined as a **layer** within a Keras Sequential class.

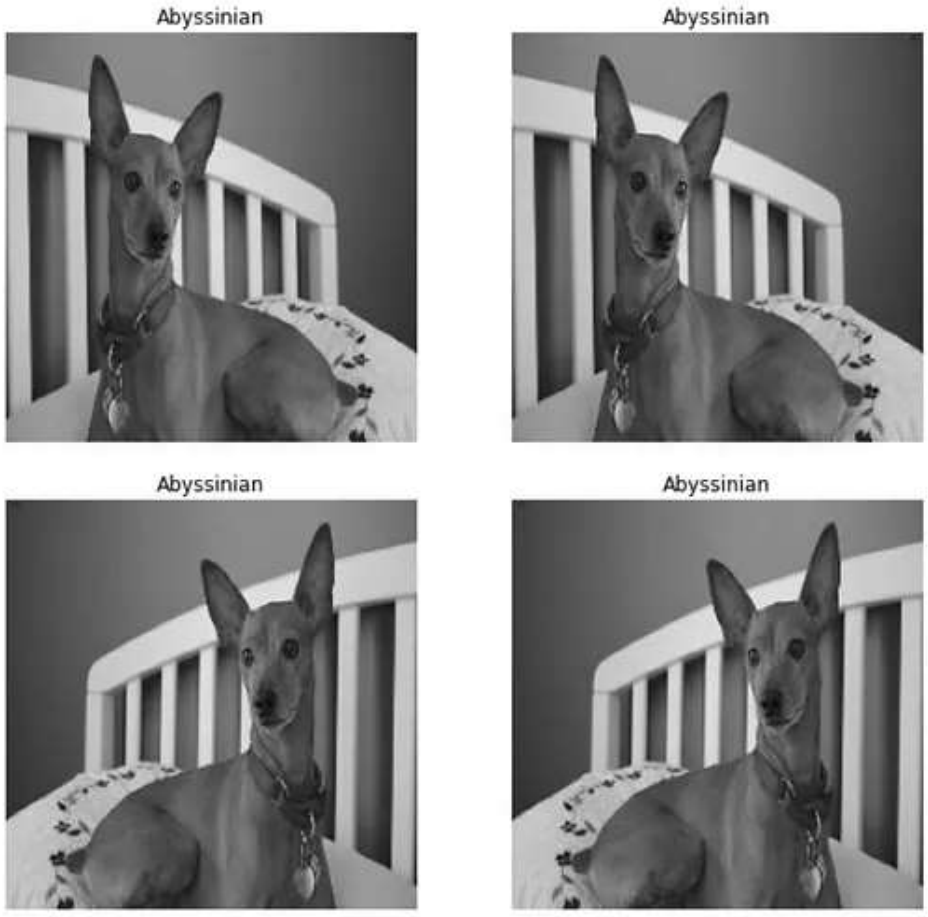
For simplicity, we will only perform a random horizontal flip for the images, but note that a wide range of augmentations is available.

```
data_augmentation = keras.Sequential(
                [layers.RandomFlip('horizontal'),
                #  layers.RandomRotation(factor=(-0.025, 0.025)),
                #  layers.RandomTranslation(height_factor=0.1, width_factor=0.1
                #  layers.RandomContrast(factor=0.1),
                ])
```

We can view the effects of augmentation with the following code:

```
for image, label in train_ds.take(1): # Iterate and get set of image and label
    plt.figure(figsize=(10, 10))
    for i in range(4):  # Display augmented images in 2x2 grid
        ax = plt.subplot(2, 2, i+1)
        aug_img = data_augmentation(tf.expand_dims(image, axis=0))
        plt.imshow(aug_img[0].numpy().astype('uint8')) # Retrieve raw pixel val
```

```
        plt.title(get_label_name(int(label[0]))) # Get corresponding label
        plt.axis('off')
```



Illustrating the impact of horizontal flip augmentation | Image by author

Top Python libraries for Image Augmentation in Computer Vision

Featuring the best augmentation libraries (with sample codes) for your next computer vision project

towardsdatascience.com

(iv) Batching and Prefetching

The next step is to set up batching and prefetching, which are techniques to optimize the efficiency of model training.

- **Batching** groups subsets of training data as mini-batches so that training can be parallelized (taking advantage of GPU hardware acceleration) while maintaining accurate gradient estimates.
- **Prefetching** overlaps model computation with the fetching of input data, allowing training to continue while the next batch of images is being retrieved.

While batch size is a hyperparameter that can be tuned, we can start by setting it at a standard default value of **32**.

```
BATCH_SIZE = 32

train_ds = train_ds.batch(batch_size=BATCH_SIZE,
                          drop_remainder=True).prefetch(tf.data.AUTOTUNE)

val_ds = val_ds.batch(batch_size=BATCH_SIZE,
                      drop_remainder=True).prefetch(tf.data.AUTOTUNE)

test_ds = test_ds.batch(batch_size=BATCH_SIZE,
                        drop_remainder=True).prefetch(tf.data.AUTOTUNE)
```

Let's take a look at the parameters of the batching/prefetching code above:

- `drop_remainder=True` means that the final batch of images smaller than a batch size of 32 (due to uneven division) is dropped. This is useful when the model depends on batches having the same outer dimension
- `tf.data.AUTOTUNE` in the `prefetch()` method means that the `tf.data` runtime will dynamically tune the number of elements to prefetch for each training step

Step 5 — Set up Model

The core concept behind transfer learning is to utilize pre-trained models as the foundation for building custom image classifiers.

Many pre-trained deep learning models (alongside the pre-trained weights) are available within Keras, and the complete list can be found here.

For this tutorial, we will use the **ResNet50V2** model because it offers an excellent balance of model size, accuracy, reputation, and inference speed.

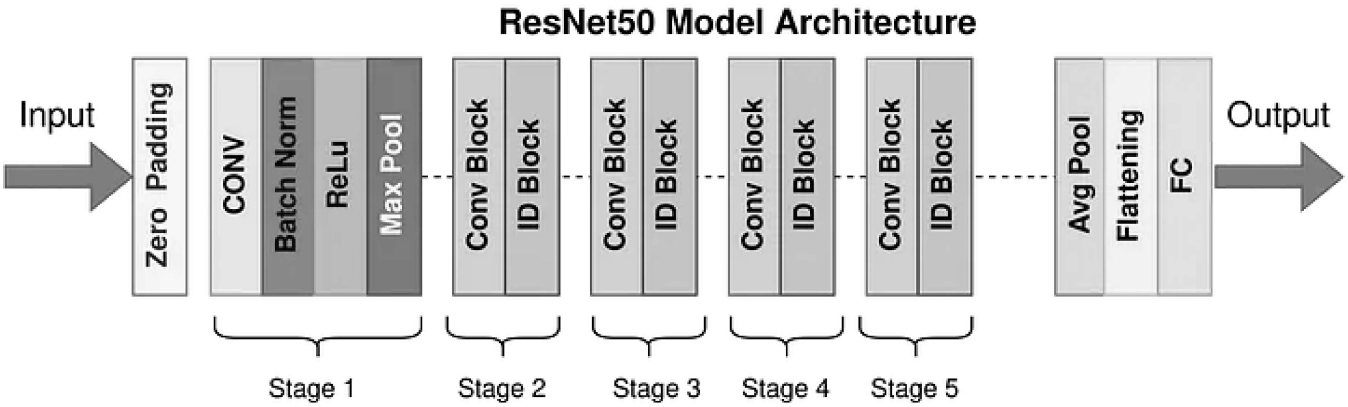


Illustration of ResNet50 model architecture | Image used under Wikimedia Commons license

(i) Setup base model

We first load the ResNet50V2 model from Keras applications using `keras.applications.ResNet50V2`.

Note that the ResNet50V2 pre-trained model was trained on the ImageNet dataset, a large-scale image dataset containing millions of images and thousands of classes.

```
base_model = keras.applications.ResNet50V2(
                        include_top=False, # Exclude ImageNet classifier at the
                        weights='imagenet',
                        input_shape=(IMG_SIZE, IMG_SIZE, 3)
                        )
```

Here is an explanation of the parameters used:

- `include_top` : If set to False, we exclude the ImageNet classifier as the top layer. This is important because we want to introduce our own top layers to classify pet breeds instead of the original ImageNet classes
- `weights` : By setting it to `'imagenet'` , we will use the pre-trained weights trained on ImageNet. It is what we want because we are performing image

classification, and the features learned from ImageNet are relevant

- `input_shape` : We set the input shape tuple to match what the ResNet architecture needs, i.e., `(224, 224, 3)` (where the last element of `3` represents the number of color channels)

(ii) Freeze pre-trained weights of base model

Once we have loaded the pre-trained model, we want to **fix** (**aka freeze**) the layers and the weights because we do not want to lose the valuable information that has already been learned.

```
# Freeze base_model
base_model.trainable = False
```

By setting the `trainable` attribute as `False` , the trainable weights become non-trainable and will not be updated during training. In general, all weights across the layers are considered trainable.

In fact, the **only built-in layer that has non-trainable weights** is `BatchNormalization` because it uses non-trainable weights to track the mean and variance of inputs during training.

(iii) Instantiate and Modify inputs

In this step, we preprocess the inputs (i.e., images) to be compatible with what the pre-trained ResNet50v2 architecture expects.

We first instantiate a Keras tensor using `keras.Input` to represent an 'object structure' for our input images.

```
# Setup inputs based on input image shape
inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
```

Next, we apply the data augmentation step set up earlier and store the output in a new variable `x` representing the new input transformations.

```
x = data_augmentation(inputs)
```

Furthermore, each pre-trained model in Keras requires a specific kind of input preprocessing, which is on top of the earlier data preparation.

We do this by running `resnet_v2.preprocess_input` on the inputs before passing them to the model.

```
# Apply specific pre-processing function for ResNet v2
x = keras.applications.resnet_v2.preprocess_input(x)
```

For ResNet50 V2 in particular, the preprocessing step scales the input pixels from a range of [0,255] to a range of **[-1, 1]** so that the input is compatible with the ResNet architecture.

(iv) Handling Batch Normalization layers

The `BatchNormalization` layers are slightly trickier, given that it has non-trainable weights for tracking the mean and variance of inputs.

Because we will unfreeze the entire model later (in **Step 8 — Finetune Model**), we need to include additional code to keep the `BatchNormalization` layers in inference mode (i.e., remain non-trainable).

```
# Keep base model batch normalization layers in inference mode (instead of tra
x = base_model(x, training=False)
```

Note that this step is specific to `BatchNormalization` layers and is on top of the base model freezing (`base_model.trainable=False`) done earlier.

If this step is omitted, the fine-tuning step later will be a mess since the non-trainable weights will be erased and undone.

(v) Rebuild top layers

Now that we have sorted out the earlier layers of the pre-trained model, we can add new trainable layers so that our model can effectively learn to classify the labels we want, i.e., pet breeds.

```
# Rebuild top layers
x = layers.GlobalAveragePooling2D()(x) # Average pooling operation
x = layers.BatchNormalization()(x) # Introduce batch norm
x = layers.Dropout(0.2)(x)  # Regularize with dropout

# Flattening to final layer - Dense classifier with 37 units (multi-class class
outputs = layers.Dense(num_classes, activation='softmax')(x)
```

The new layers include the following:

- Global average pooling
- Batch normalization
- Dropout (with a probability of 20%)
- Final output dense layer with softmax activation and with an output dimensionality of 37 (reflecting 37 classes of pet breeds)

(vi) Create new Keras model object

We instantiate a new Keras model with the updated inputs and outputs defined earlier.

```
# Instantiate final Keras model with updated inputs and outputs
model = keras.Model(inputs, outputs)
```

We can view the structure of our updated model with `model.summary()`:

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 224, 224, 3)]     0

 sequential (Sequential)     (None, 224, 224, 3)       0

 tf.math.truediv (TFOpLambda  (None, 224, 224, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 224, 224, 3)      0
 a)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 batch_normalization (BatchN  (None, 2048)             8192
 ormalization)

 dropout (Dropout)           (None, 2048)              0

 pred (Dense)                (None, 37)                75813

=================================================================
Total params: 23,648,805
Trainable params: 79,909
Non-trainable params: 23,568,896
_________________________________________________________________
```

Model summary | Image by author

(vii) Compile model

Lastly, we apply the finishing touches by compiling the model, which involves configuring settings like evaluation metrics and optimizer for stochastic gradient descent.

```
model.compile(optimizer=keras.optimizers.Adam(),
              loss=keras.losses.CategoricalCrossentropy(),
              metrics=[keras.metrics.CategoricalAccuracy()]
              )
```

Since we are dealing with multiclass classification, it makes sense to use categorical cross entropy as the loss metric and categorical accuracy as the performance metric.

We also include an early stopping callback to avoid overfitting during training.

```
earlystopping = callbacks.EarlyStopping(monitor='val_loss',
                                        mode='min',
                                        patience=5,
                                        restore_best_weights=True)
```

Step 6 — Run Model Training

With our model updated and compiled, we are ready to train it on the pet image dataset.

We start with an epoch count of 25, although the early stopping callback will help to cease training when signs of overfitting are detected.

We run `model.fit()` to initiate training, and we store the training output in a variable called `history`.

```
EPOCHS = 25

history = model.fit(train_ds,
```

```
                    epochs=EPOCHS,
                    validation_data=val_ds,
                    verbose=1,
                    callbacks =[earlystopping])
```

```
Epoch 1/25
103/103 [==============================] - 22s 141ms/step - loss: 1.1400 - categorical_accuracy: 0.6723 - val_loss: 0.5788 - val_categorical_accuracy: 0.8295
Epoch 2/25
103/103 [==============================] - 14s 131ms/step - loss: 0.2827 - categorical_accuracy: 0.9117 - val_loss: 0.4679 - val_categorical_accuracy: 0.8466
Epoch 3/25
103/103 [==============================] - 14s 130ms/step - loss: 0.1591 - categorical_accuracy: 0.9508 - val_loss: 0.4326 - val_categorical_accuracy: 0.8636
Epoch 4/25
103/103 [==============================] - 14s 131ms/step - loss: 0.1031 - categorical_accuracy: 0.9745 - val_loss: 0.4072 - val_categorical_accuracy: 0.8722
Epoch 5/25
103/103 [==============================] - 15s 145ms/step - loss: 0.0765 - categorical_accuracy: 0.9818 - val_loss: 0.4091 - val_categorical_accuracy: 0.8608
Epoch 6/25
103/103 [==============================] - 14s 139ms/step - loss: 0.0576 - categorical_accuracy: 0.9888 - val_loss: 0.4344 - val_categorical_accuracy: 0.8778
Epoch 7/25
103/103 [==============================] - 13s 129ms/step - loss: 0.0414 - categorical_accuracy: 0.9933 - val_loss: 0.4430 - val_categorical_accuracy: 0.8778
Epoch 8/25
103/103 [==============================] - 14s 131ms/step - loss: 0.0345 - categorical_accuracy: 0.9945 - val_loss: 0.4527 - val_categorical_accuracy: 0.8722
Epoch 9/25
103/103 [==============================] - 14s 132ms/step - loss: 0.0269 - categorical_accuracy: 0.9961 - val_loss: 0.4573 - val_categorical_accuracy: 0.8693
```

Epoch training output | Image by author

Step 7 — Evaluate Model

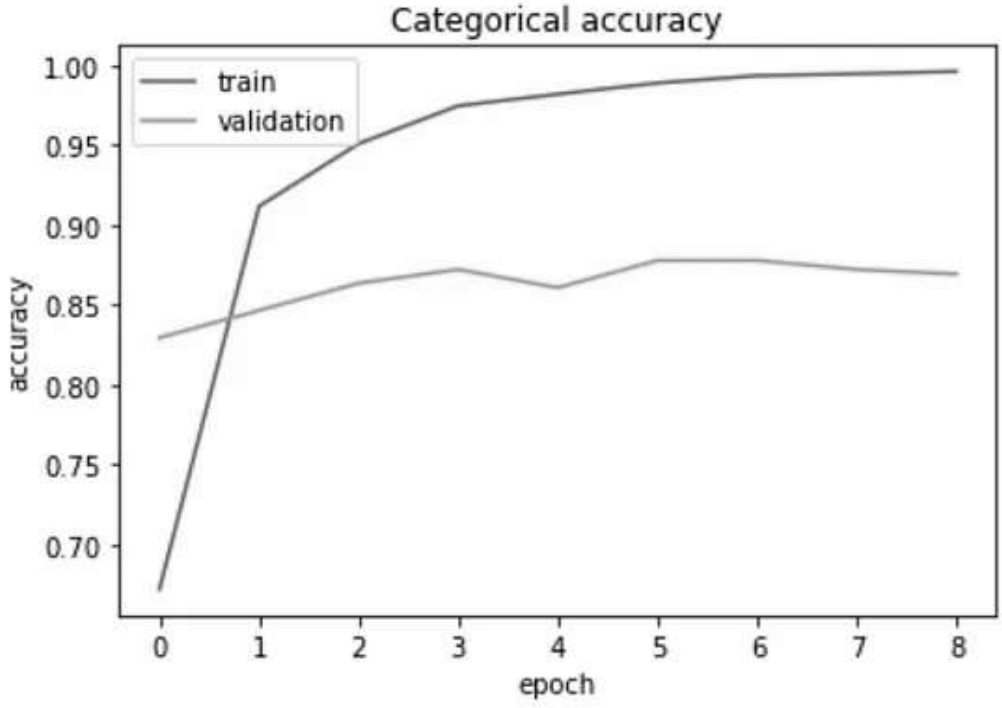
Once training is complete, it is time to understand how our model fared.

(i) Plot training and validation sets

We can plot the categorical accuracies on the training and validation sets over increasing epochs.

```
def plot_hist(hist):
    plt.plot(hist.history['categorical_accuracy'])
    plt.plot(hist.history['val_categorical_accuracy'])
    plt.title('Categorical accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'validation'], loc='upper left')
    plt.show()

plot_hist(history)
```



Training and validation accuracy scores over epochs | Image by author

We see from the plot that it only takes a few epochs to overfit the training dataset, and the validation accuracy did not improve after five epochs.

(ii) Predict and evaluate on test dataset

To evaluate the model predictions on the test set, we use `model.evaluate()`.

```
result = model.evaluate(test_ds)
dict(zip(model.metrics_names, result))
```

```
result = model.evaluate(test_ds)
```

```
114/114 [==============================] - 13s 112ms/step - loss: 0.4689 - categorical_accuracy: 0.8607
```

```
dict(zip(model.metrics_names, result))
```

```
{'loss': 0.46886688470840454, 'categorical_accuracy': 0.8607456088066101}
```

Evaluation results from model | Image by author

The results show that the categorical accuracy is at a fantastic value of **86.1%**.

And with that, we have completed transfer learning by building a deep learning model for image classification from the ResNet50 V2 pre-trained model.

Step 8 — Finetune Model (Optional)

As an optional step, we can further optimize the model by unfreezing and retraining all the weights in the model.

This process is called **fine-tuning**, where we unfreeze all or parts of the earlier pre-trained layers and retrain the weights on the new data.

While this technique may boost performance, it runs the risk of overfitting quickly.

A crucial part of fine-tuning is to retrain the model using a **very low learning rate** (e.g., 0.00001).

The rationale is that these pre-trained layers have already learned valuable features for image classification. Hence, a low learning rate ensures the model only makes subtle parameter adjustments without significant disruptions.

In our case, we unfreeze the top 15 layers of the entire model by setting the `trainable` attribute to `True`, excluding `BatchNormalization` layers which we need to keep frozen.

```
for layer in model.layers[-15:]:
    if not isinstance(layer, layers.BatchNormalization):
        layer.trainable = True
```

After these changes, we need to recompile the model, where we significantly lower the learning rate from default 0.001 to **0.00001**.

```
model.compile(optimizer=keras.optimizers.Adam(learning_rate=1e-5), # Set a very
              loss=keras.losses.CategoricalCrossentropy(),
```

```
                metrics=[keras.metrics.CategoricalAccuracy()]
                )
```

We retrain the model and retrieve the evaluation metrics in the final steps. Given the risk of overfitting, we run the training over fewer epochs.

```
EPOCHS = 5

history_2 = model.fit(train_ds,
                      epochs=EPOCHS,
                      validation_data=val_ds,
                      verbose=1,
                      callbacks =[earlystopping])

result_2 = model.evaluate(test_ds)

dict(zip(model.metrics_names, result_2))
```

```
result_2 = model.evaluate(test_ds)

dict(zip(model.metrics_names, result_2))
```

```
114/114 [==============================] - 13s 112ms/step - loss: 0.4471 - categorical_accuracy: 0.8717
{'loss': 0.44707489013671875, 'categorical_accuracy': 0.8717105388641357}
```

Evaluation results from fine-tuned model | Image by author

The evaluation output above shows that the fine-tuning method has helped boost the categorical accuracy from **86.1% to 87.2%**.

Finally, to retrieve the actual predictions on the test set, we can use the `model.predict()` function:

```
preds = model.predict(test_ds)
```

(4) Wrapping it up

This walkthrough covered the practical steps in getting started with transfer learning for image classification.

Transfer learning, coupled with fine-tuning, is an effective technique that empowers us to build robust image classifiers under limited time and resources.

Given its utility, it is no surprise that transfer learning is a widespread technique deployed in the industry to solve business problems.

The link to the complete notebook can be found **here**.

Before you go

I welcome you to **join me on a data science learning journey!** Follow this Medium page and check out my GitHub to stay in the loop of more exciting practical data science content. Meanwhile, have fun implementing transfer learning in TensorFlow for image classification!

Transfer Learning Tensor Flow Image Classification Machine Learning

Hands On Tutorials

Enjoy the read? Reward the writer.[Beta]

Your tip will go to Kenneth Leung through a third-party platform of their choice, letting them know you appreciate their story.

Give a tip

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Get this newsletter