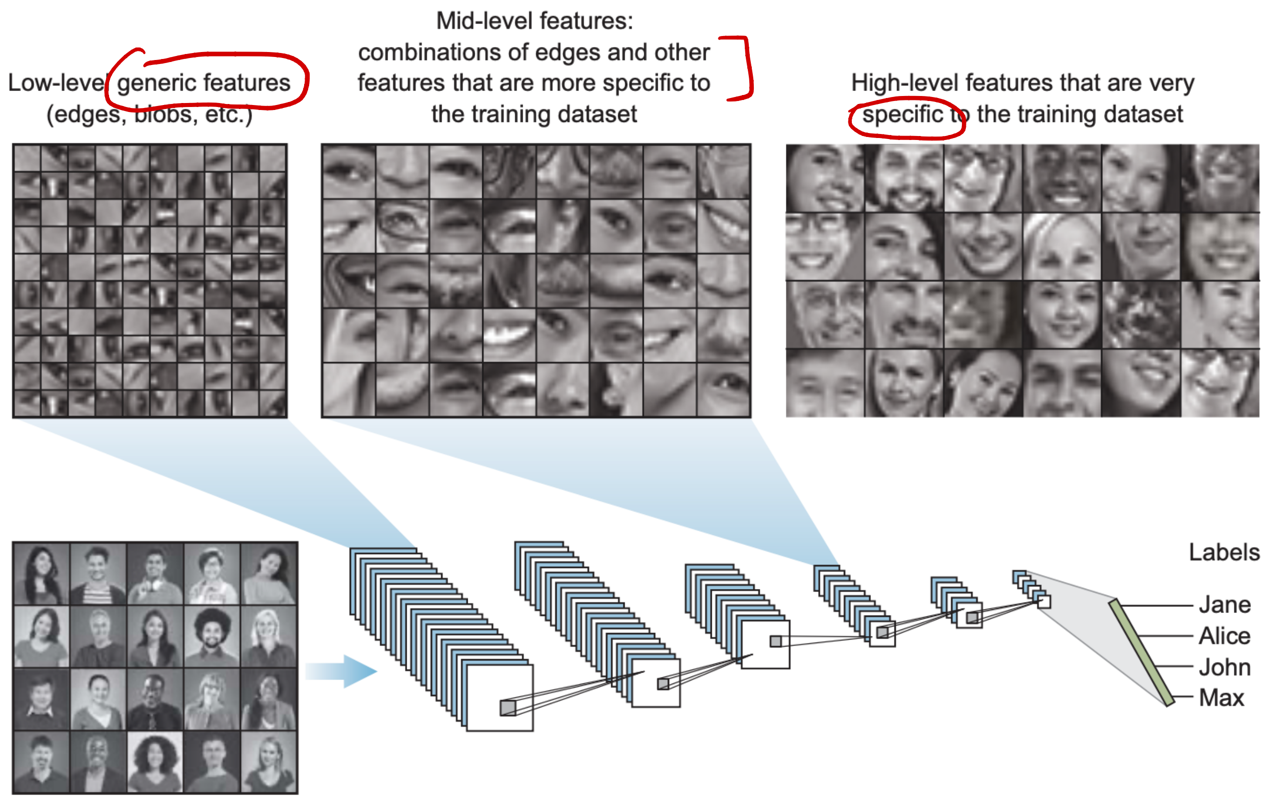
2021/12/21

Transfer Learning for Image Classification: Part 1

A CNN learns the features in a dataset step by step in increasing levels of complexity, one layer after another. These are called feature maps. The deeper you go through the network layers, the more image-specific features are learned. In the figure below, the first layer detects low-level features such as edges and curves. The output of the first layer becomes input to the second layer, which produces higher-level features like semicircles and squares. The next layer assembles the output of the previous layer into parts of familiar objects, and a subsequent layer detects the objects. As we go through more layers, the network yields a feature map that represents more complex features. As we go deeper into the network, the filters begin to be more responsive to a larger region of the pixel space. Higher-level layers amplify aspects of the received inputs that are important for discrimination and suppress irrelevant variations.

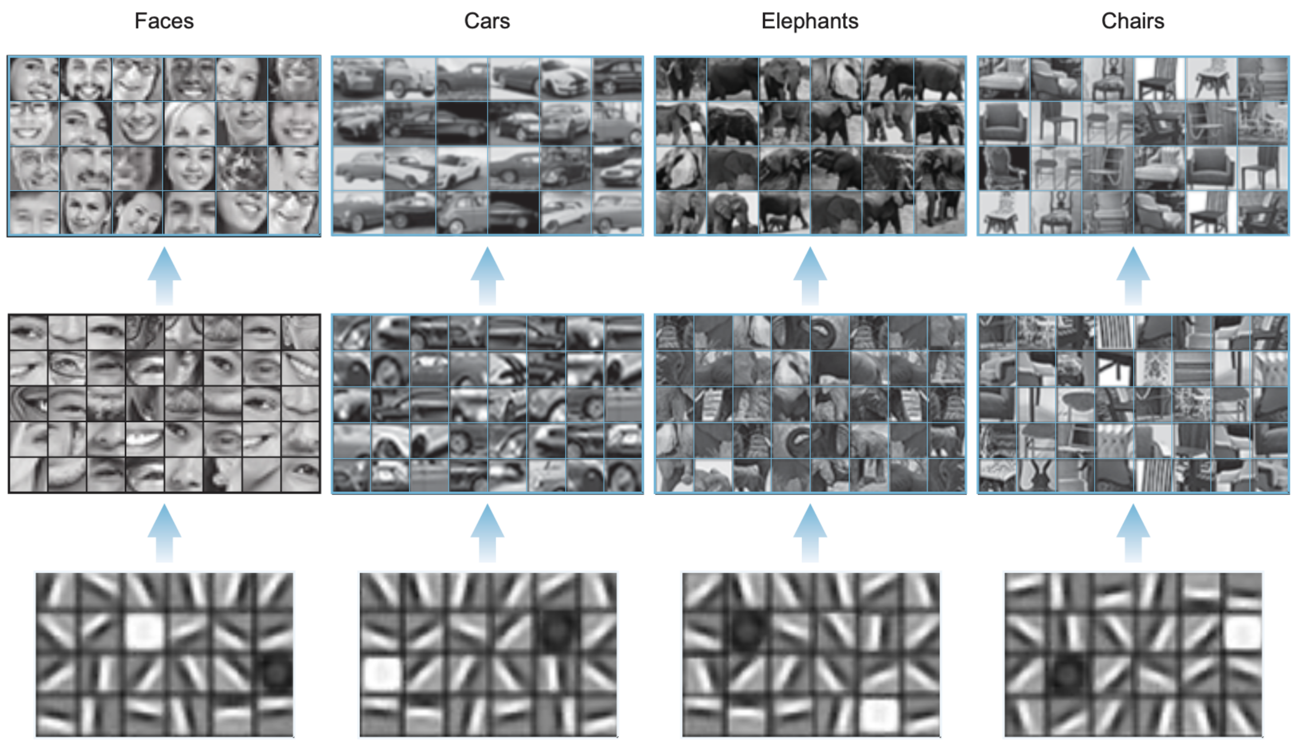

Low-level generic features (edges, blobs, etc.)
Mid-level features: combinations of edges and other features that are more specific to the training dataset
High-level features that are very specific to the training dataset
Labels
Jane
Alice
John
Max

A common and highly effective approach to deep learning on small image datasets is to use transfer learning with a **pretrained network**. A pretrained network is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task, for instance, a large CNN (e.g., ResNet) trained on the ImageNet dataset (1.4 million labeled images and 1,000 different classes). Top1 Top5

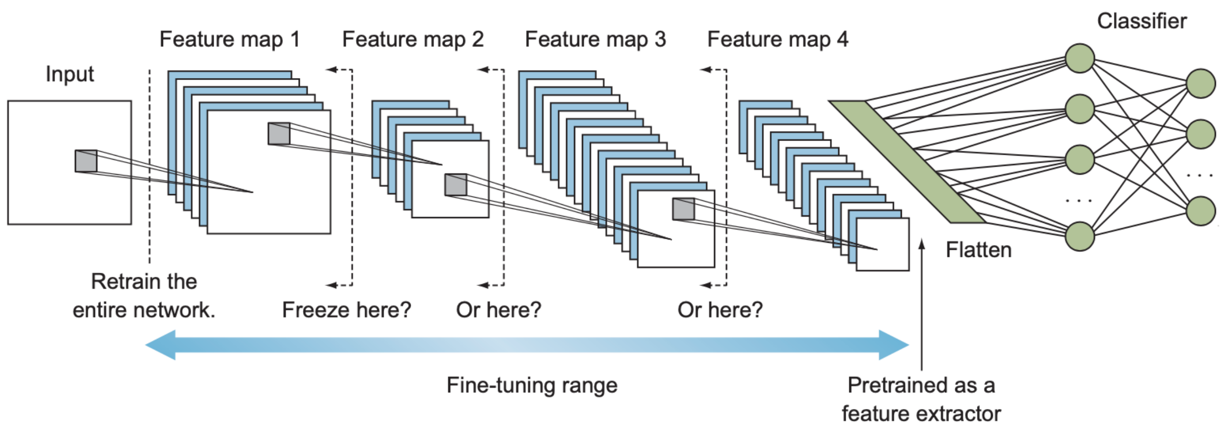
Note that the level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model. Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), whereas layers that are higher up extract more-abstract concepts (such as "cat ear" or "dog eye"). So if your new dataset differs a lot from the dataset on which the original model was trained, you may be better off using only the first few layers of the model to do feature extraction, rather than using the entire convolutional base.

Let's compare the feature maps extracted from four models that are trained to classify faces, cars, elephants, and chairs (see figure below). Notice that the earlier layers' features are very similar for all the models. They represent low-level features like edges, lines, and blobs. This means models that are trained on one task capture similar relations in the data types in the earlier layers of the network and can easily be reused for different problems in other domains. The deeper we go into the network, the more specific the features.

The lower-level features are almost always transferable from one task to another because they contain generic information like the structure and nature of how images look. Transferring information like lines, dots, curves, and small parts of objects is very valuable for the network to learn faster and with less data on the new task.

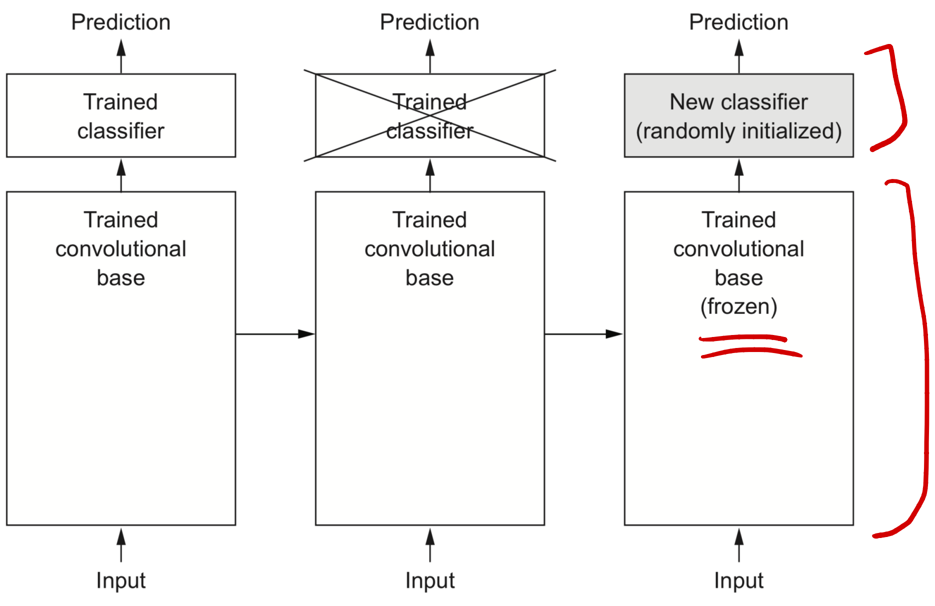


There are two ways to use a pretrained network: feature extractor and fine-tuning. We'll cover feature extractor in part 1 and fine-tuning in part 2 of the notes.



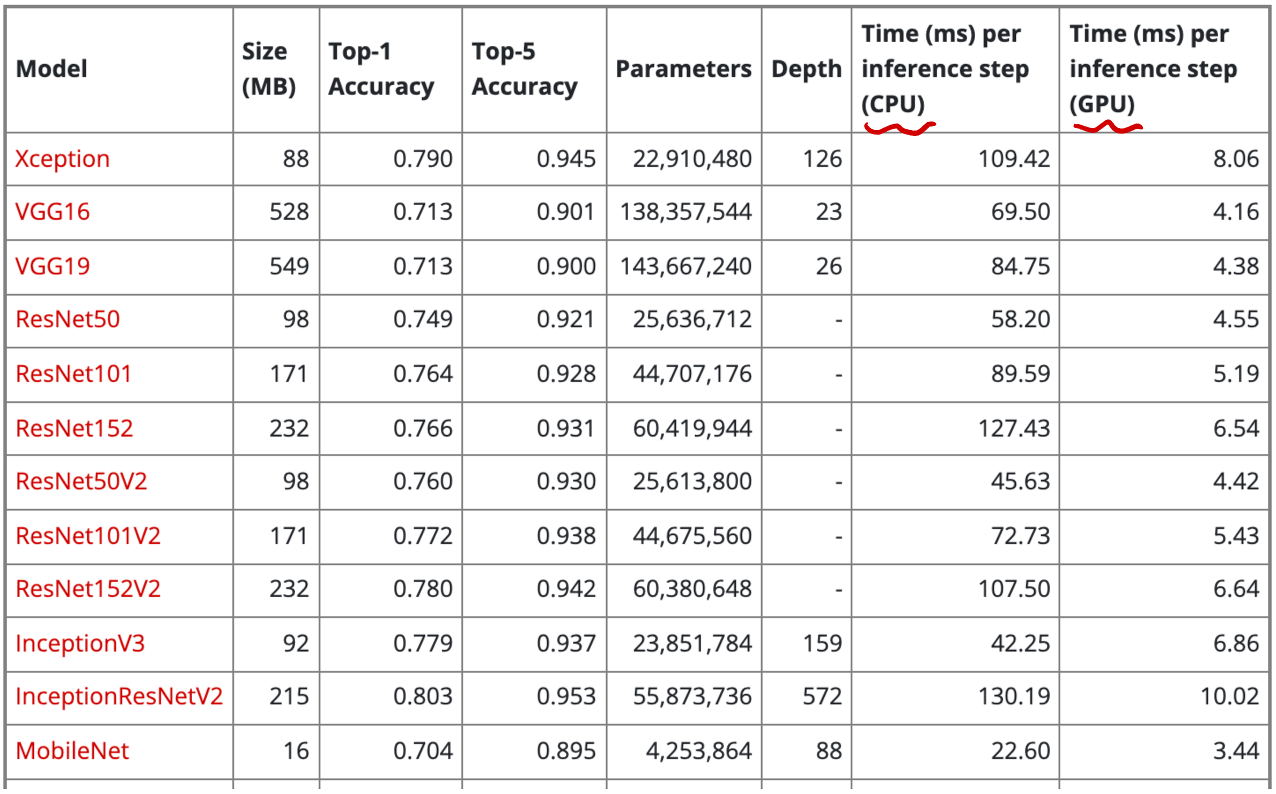
1. Feature extractor

Feature extraction consists of using the representations learned by a pretrained network to extract interesting features. As you saw previously, CNNs used for image classification comprise two parts: they start with a series of convolution and pooling layers, and they end with a densely connected classifier. The first part is called *the convolutional base* of the model. In the case of CNNs, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output (see figure below).


Prediction
Trained classifier
Trained convolutional base
Input
Prediction
~~Trained classifier~~
Trained convolutional base
Input
Prediction
New classifier (randomly initialized)
Trained convolutional base (frozen)
Input

1.1 Project: Using Pretrained Network As a Feature Extractor

Let's put this in practice by using the convolutional base of the VGG16 network, trained on ImageNet. The VGG16 model, among others, comes prepackaged with `Keras`. You can import it from the `keras.applications` module. Below are a few image-classification models (all pretrained on the ImageNet dataset) that are available as part of `keras.applications`:



Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	0.790	0.945	22,910,480	126	109.42	8.06
VGG16	528	0.713	0.901	138,357,544	23	69.50	4.16
VGG19	549	0.713	0.900	143,667,240	26	84.75	4.38
ResNet50	98	0.749	0.921	25,636,712	-	58.20	4.55
ResNet101	171	0.764	0.928	44,707,176	-	89.59	5.19
ResNet152	232	0.766	0.931	60,419,944	-	127.43	6.54
ResNet50V2	98	0.760	0.930	25,613,800	-	45.63	4.42
ResNet101V2	171	0.772	0.938	44,675,560	-	72.73	5.43
ResNet152V2	232	0.780	0.942	60,380,648	-	107.50	6.64
InceptionV3	92	0.779	0.937	23,851,784	159	42.25	6.86
InceptionResNetV2	215	0.803	0.953	55,873,736	572	130.19	10.02
MobileNet	16	0.704	0.895	4,253,864	88	22.60	3.44

- The top-1 and top-5 accuracy refers to the model's performance on the ImageNet validation dataset.
- Depth refers to the topological depth of the network. This includes activation layers, batch normalization layers etc.
- Time per inference step is the average of 30 batches and 10 repetitions. - CPU: AMD EPYC Processor (with IBPB) (92 core) - Ram: 1.7T - GPU: Tesla A100 - Batch size: 32

In this project, we will use a very small amount of data to train a classifier that detects images of dogs and cats (202 training samples, 103 validation samples, 451 testing samples).

This is a pretty simple project, but the goal of the exercise is to see how to implement transfer learning when you have a very small amount of data and the target domain is similar to the source domain. We will use the pretrained convolutional network as a feature extractor. This means we are going to freeze the feature extractor part of the network, add our own classifier, and then retrain the network on our new small dataset.

One other important takeaway from this project is learning how to preprocess custom data and make it ready to train your neural network. In previous projects, we used the CIFAR and MNIST datasets: they are preprocessed by `Keras`, so all we had to do was download them from the `Keras` library and use them directly to train the network. This project provides a tutorial of how to structure your data repository and use the `Keras` library to get your data ready.

The process to use a pretrained model as a feature extractor is well established:

1. Import the necessary libraries.
2. Preprocess the data to make it ready for the neural network.
3. Load pretrained weights from the VGG16 network trained on a large dataset.
4. Freeze all the weights in the convolutional layers (feature extraction part).
5. Remember, the layers to freeze are adjusted depending on the similarity of the new task to the original dataset. In our case, we observed that ImageNet has a lot of dog and cat images, so the network has already been trained to extract the detailed features of our target object.
6. Replace the fully connected layers of the network with a custom classifier. You can add as many fully connected layers as you see fit, and each can have as many hidden units as you want. For simple problems like this, we will just add one hidden layer with 64 units. You can observe the results and tune up if the model is underfitting or down if the model is overfitting. For the `softmax` layer, the number of units must be set equal to the number of classes (two units, in our case).
7. Compile the network, and run the training process on the new data of cats and dogs to optimize the model for the smaller dataset.
8. Evaluate the model.

1. Import the necessary libraries

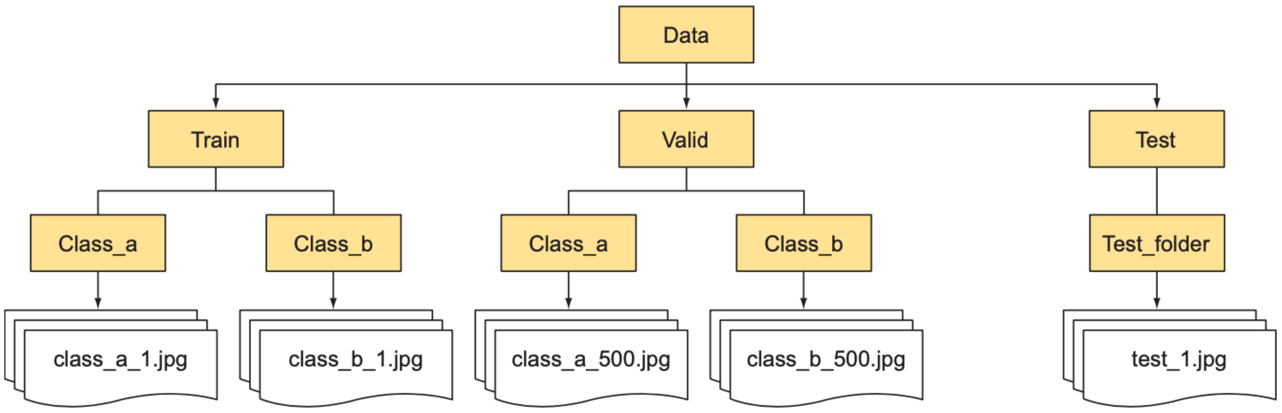
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications import imagenet_utils
from tensorflow.keras.applications import vgg16
from tensorflow.keras.applications import mobilenet
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.metrics import categorical_crossentropy

from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Preprocess the data to make it ready for the neural network

`Keras` has an `ImageDataGenerator` class that allows us to easily perform image augmentation on the fly; you can read about it at https://keras.io/api/preprocessing/image. In this example, we use

`ImageDataGenerator` to generate our image tensors, but for simplicity, we will not implement image augmentation.

The `ImageDataGenerator` class has a method called `flow_from_directory()` that is used to read images from folders containing images. This method expects your data directory to be structured as in figure below.



You can load the data into `train_path`, `valid_path`, and `test _path` variables, and then generate the train, valid, and test batches:

```
train_path  = 'data/train'
valid_path  = 'data/valid'
test_path  = 'data/test'

train_batches = ImageDataGenerator().flow_from_directory(train_path,
target_size=(224,224), batch_size=30)

valid_batches = ImageDataGenerator().flow_from_directory(valid_path,
target_size=(224,224), batch_size=30)

test_batches = ImageDataGenerator().flow_from_directory(test_path,
target_size=(224,224), batch_size=30)
```

3. Load in pretrained weights from the VGG16 network trained on a large dataset

We can download the VGG16 network from `Keras` and download its weights after they are pretrained on the ImageNet dataset. Remember that we want to remove the classifier part from this network, so we set the parameter `include_top=False`:
dense

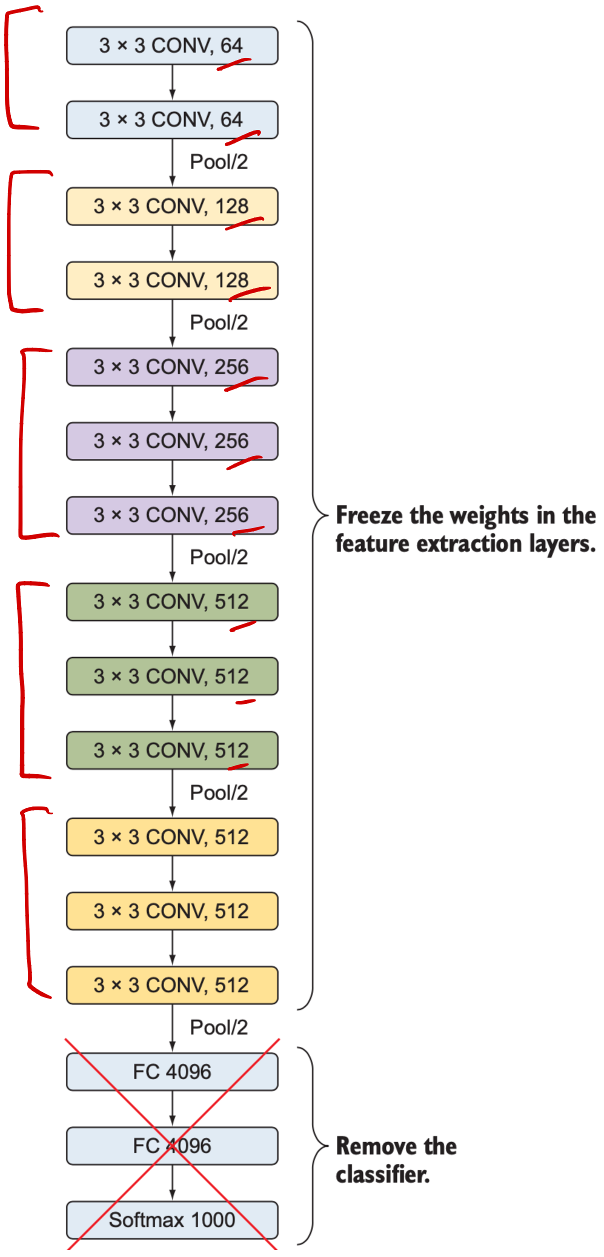
```
base_model = vgg16.VGG16(weights = "imagenet", include_top=False,
input_shape = (224,224, 3))
```

You pass three arguments to the `VGG16` pretrained network constructor:

- `weights` specifies the weight checkpoint from which to initialize the model.

- `include_top` refers to including (or not) the densely connected classifier on top of the network. By default, this densely connected classifier corresponds to the 1,000 classes from ImageNet. Because you intend to use your own densely connected classifier (with only two classes: `cat` and `dog`), you don't need to include it.
- `input_shape` is the shape of the image tensors that you'll feed to the network. This argument is purely optional: if you don't pass it, the network will be able to process inputs of any size.

Here's the detail of the architecture of the VGG16 convolutional base. It's similar to the simple convnets you're already familiar with:



```
base_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Remark: The VGG16 architecture strictly uses 3x3 convolution kernels with intermediate max-pooling layers for feature extraction. The design choice of using smaller kernels leads to a relatively reduced number of parameters, and therefore an efficient training and testing.

The final feature map has shape $(\overset{7}{\cancel{4}}, \overset{7}{\cancel{4}}, 512)$. That's the feature on top of which we will stick a densely-connected classifier.

4. Freeze all the weights in the convolutional layers (feature extraction part).

```
# iterate through its layers and lock them to make them not trainable
with this code
for layer in base_model.layers:
    layer.trainable = False
base_model.summary()
```

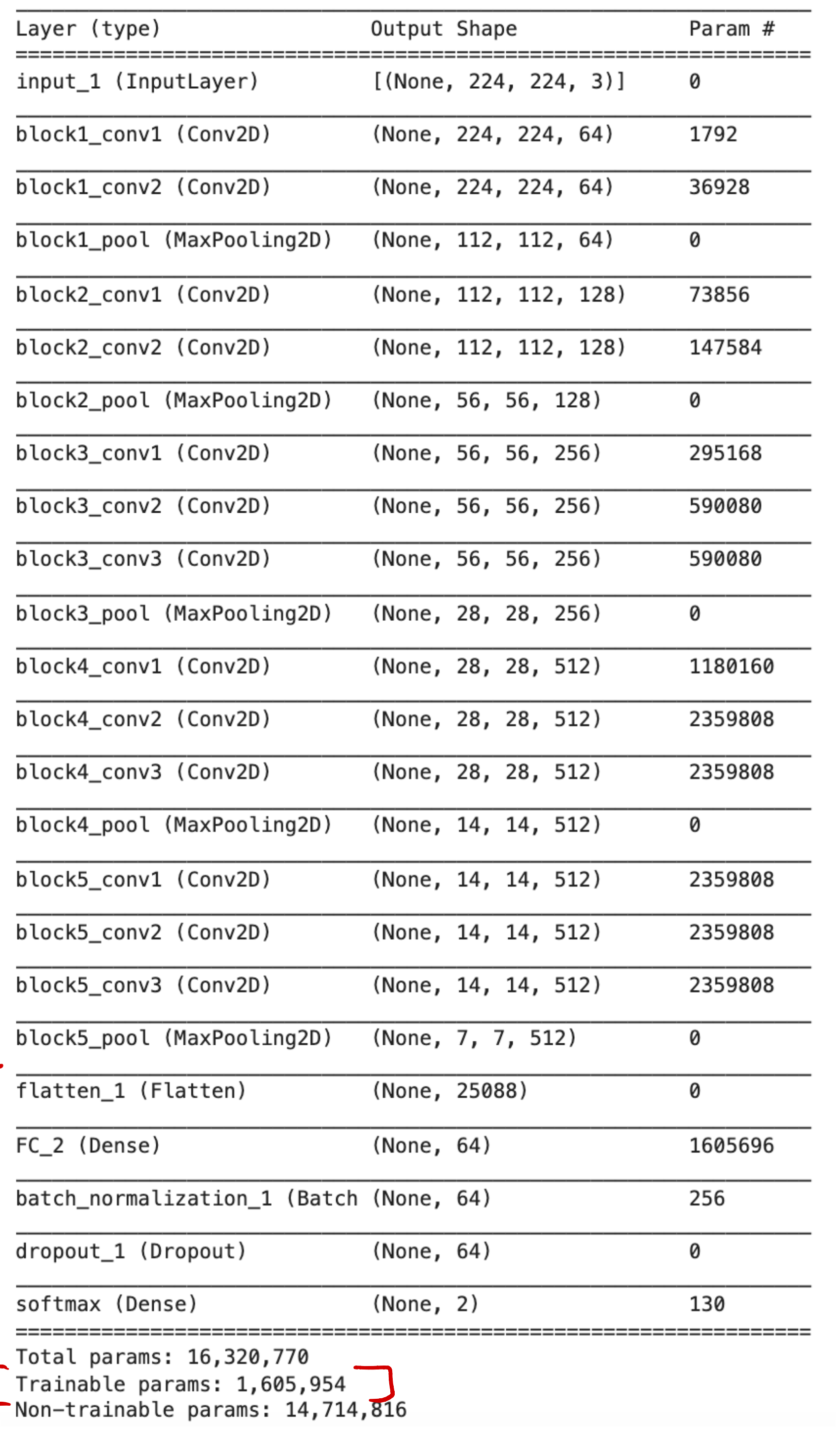
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0         
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792      
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928     
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0         
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856     
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584    
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0         
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168    
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080    
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080    
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0         
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160   
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808   
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808   
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0         
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808   
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808   
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808   
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0         
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

Fun Time: How many trainable parameters we have after freezing all the weights? [✓ (1) 0] (2) 14,714,688.

5. Add the new classifier, and build the new model.

We add a few layers on top of the base model. In this example, we add one fully connected layer with 64 units and a softmax 2 hidden units. We also add batch norm and dropout layers to avoid overfitting:

```
from tensorflow.keras.layers import Dense, Flatten, Dropout,
BatchNormalization
from tensorflow.keras.models import Model

# use "get_layer" method to save the last layer of the network
# save the output of the last layer to be the input of the next layer
last_layer = base_model.get_layer('block5_pool')
last_output = last_layer.output

# flatten the classifier input which is output of the last layer of
VGG16 model
x = Flatten()(last_output)

# add a 64 unit FC layer and relu activation
x = Dense(64, activation='relu', name='FC_2')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
# add our new softmax layer with 2 units
x = Dense(2, activation='softmax', name='softmax')(x)

# instantiate a new_model using keras's Model class
new_model = Model(inputs=base_model.input, outputs=x)

# print the new_model summary
new_model.summary()
```



```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
FC_2 (Dense)                 (None, 64)                1605696
_________________________________________________________________
batch_normalization_1 (Batch (None, 64)                256
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
softmax (Dense)              (None, 2)                 130
=================================================================
Total params: 16,320,770
Trainable params: 1,605,954
Non-trainable params: 14,714,816
```

Remark: The number of training parameters for FC_2 can be calculated from 25088*64+64 = 1,605,696

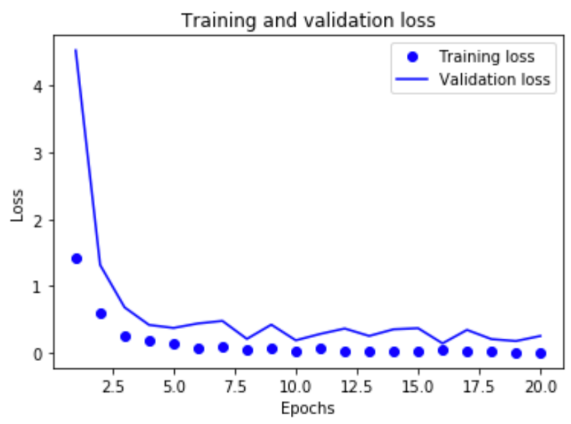
6. Compile the model and run the training process

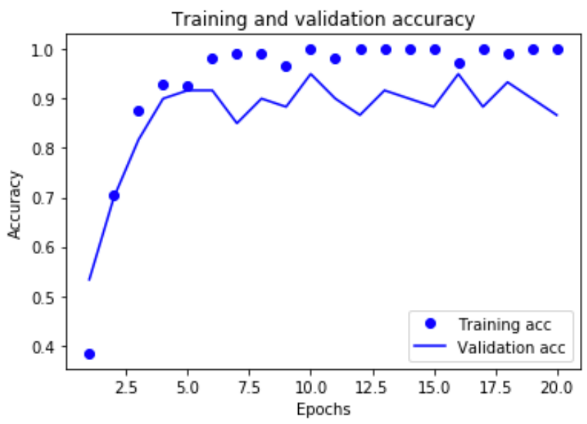
```
new_model.compile(Adam(lr=0.0001), loss='categorical_crossentropy',
metrics=['accuracy'])

from tensorflow.keras.callbacks import ModelCheckpoint
```

```
checkpointer = ModelCheckpoint(filepath='model.20epochs.hdf5',
verbose=1, save_best_only=True)

history = new_model.fit(train_batches,
steps_per_epoch=4,callbacks=[checkpointer],
validation_data=valid_batches, validation_steps=2, epochs=20,
verbose=2)
```

Remark: notice that the model was trained very quickly using regular CPU computing power.

We can then plot the learning curves as usual:





7. <u>Evaluate the model</u>

First, let's define the `load_dataset()` method that we will use to convert our dataset into tensors:

```
from sklearn.datasets import load_files
from tensorflow.keras.utils import to_categorical
import numpy as np

def load_dataset(path):
    data = load_files(path)
    paths = np.array(data['filenames'])
    targets = to_categorical(np.array(data['target']))
    return paths, targets

test_files, test_targets = load_dataset('data/test')
```

Then, we create `test_tensors` to evaluate the model on them:

```
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input
from tqdm import tqdm

def path_to_tensor(img_path):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224,
3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and
return 4D tensor
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in
tqdm(img_paths)]
    return np.vstack(list_of_tensors)

test_tensors = preprocess_input(paths_to_tensor(test_files))
```

Now we can run Keras's `evaluate()` method to calculate the model accuracy:

```
# evaluate and print test accuracy
# load the weights that yielded the best validation accuracy
new_model.load_weights('model.20epochs.hdf5')
score = new_model.evaluate(test_tensors, test_targets)
print('\n', 'Test accuracy:', score[1])
```

The model has achieved an accuracy of 96% in less than 10 minutes of training. This is very good, given our very small dataset.

You can download the package `dogs_vs_cats_project.zip` from the course website to reproduce and play around the results.