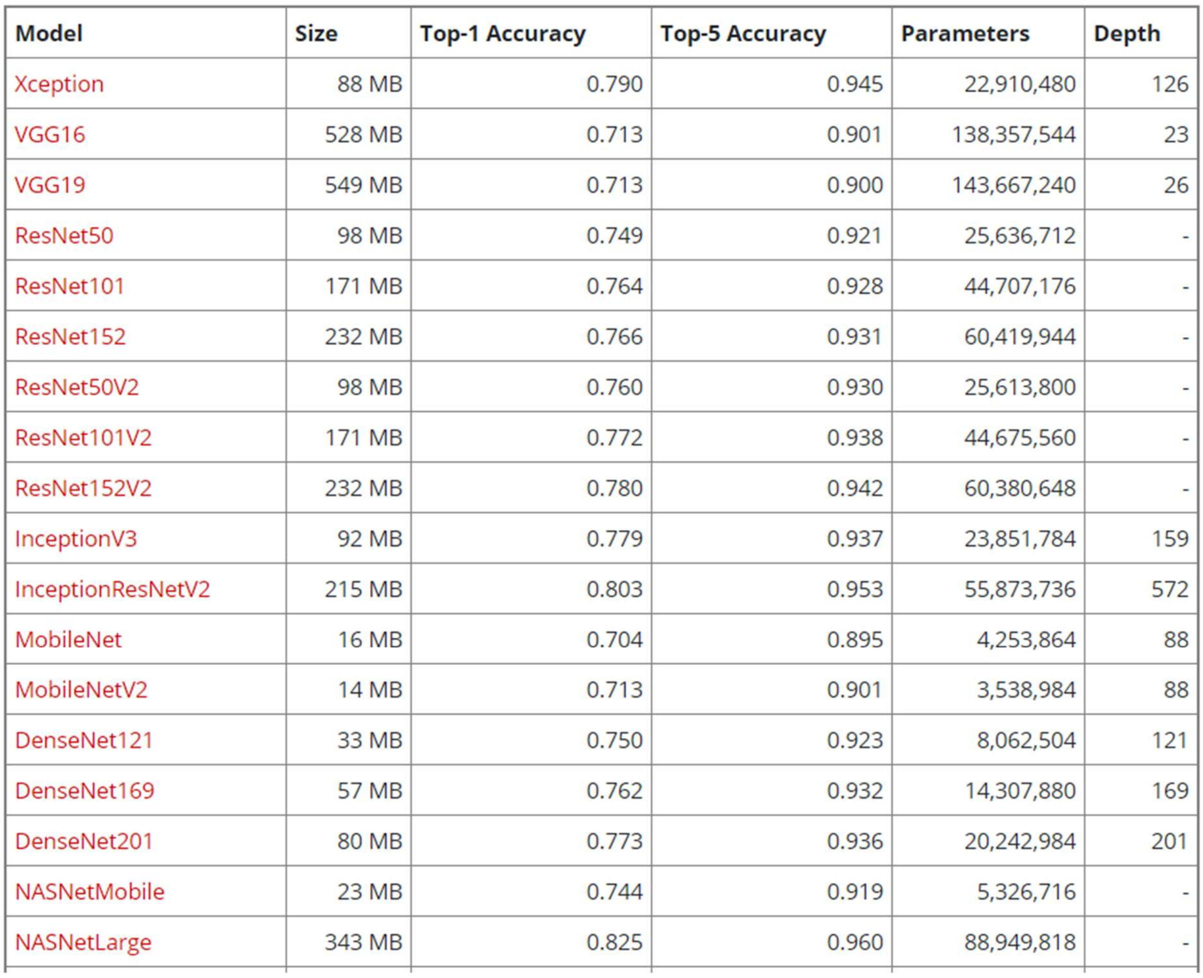
Convolutional Neural Networks (Part II)

1. Introduction

Keras has a number of pre-trained models whose parameters (weights and bias) can be used in other models. Here a list of some of them (all of them are models for image classification):

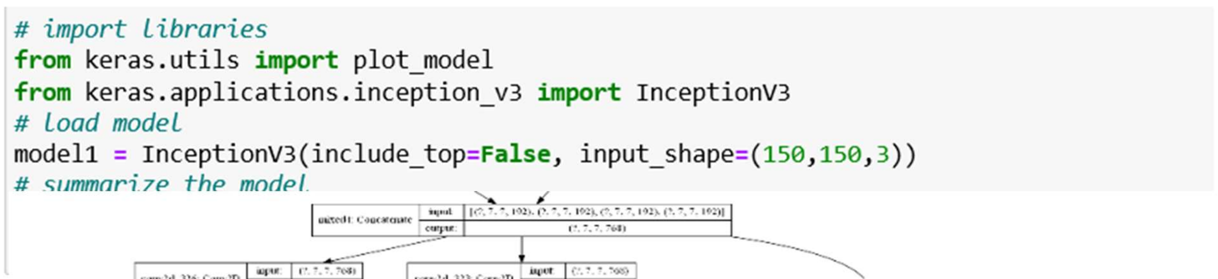


Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

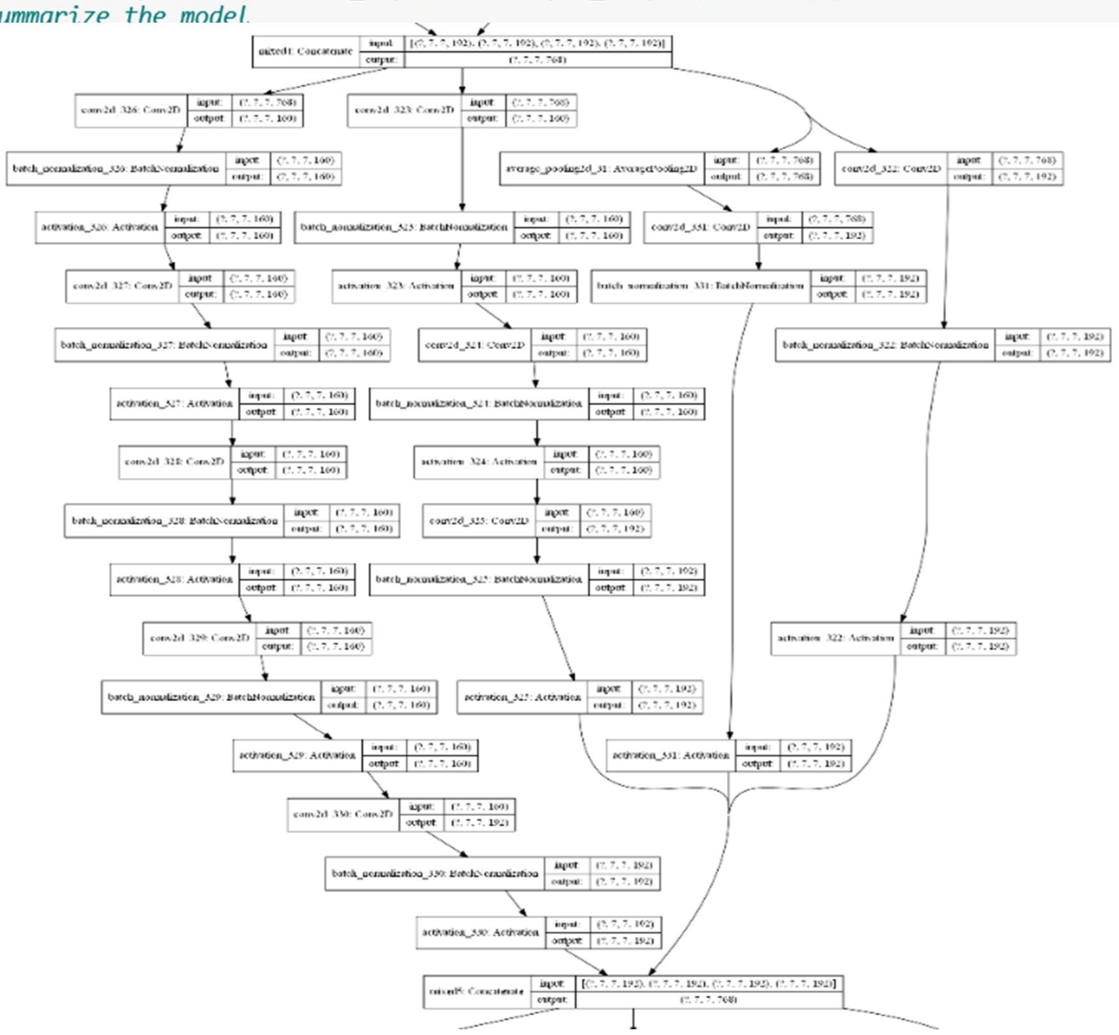
As you can see, it goes from "small" ones (with only 3 538 984 parameters) to big ones (with up to 143 667 240 parameters). Each of these models have been trained on general or specific imagery classification problems (multi class and/or multi label). When we solve an imagery classification problem, it's always a good idea to look for a pre-trained model that has been trained on similar images. A pre-trained model that has previously been trained on a dataset contains the weights and biases that represent the features of whichever dataset it was trained on. Learned features are often transferable to different data. For example, a model trained on a large dataset of bird images will contain learned features like edges or horizontal lines that you would be transferable on other dataset (boats, for example). Pre-trained models are beneficial for many reasons. By using a pre-trained model you are saving time. Someone else has already spent the time and compute resources to learn a lot of features and your model will likely benefit from it.

As we saw in the previous tutorial, convnets used for image classification have two parts: they

start with a series of convolution and pooling layers, and they end with a densely connected layer. The first part is called the **convolutional base** of the model. In the case of transfer learning, we only keep the weight and bias of the convolutional base. The reason is that the representations learned by the convolutional base are likely to be more generic and therefore more reusable. But the representations learned by the output dense layers will necessarily be specific to the set of classes on which the model was trained—they will only contain information about the presence probability of this or that class in the entire picture.
Let’s inspect two model. The first one is InceptionV3 which is the third iteration of the inception architecture, developed by researchers at Google in 2015. It can be imported with Keras with the following instructions:



```
# import libraries
from keras.utils import plot_model
from keras.applications.inception_v3 import InceptionV3
# load model
model1 = InceptionV3(include_top=False, input_shape=(150,150,3))
# summarize the model
```

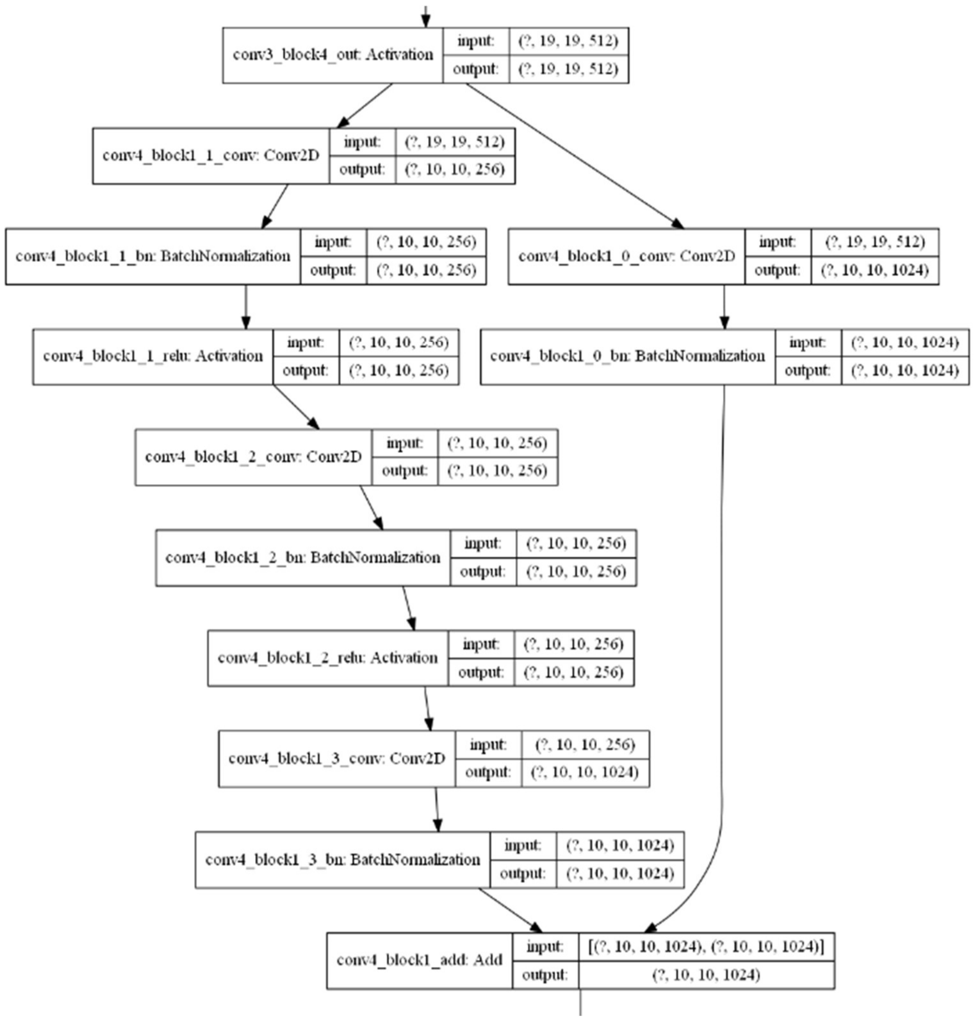


Here, we have not included the final dense layers of the model (*include_top=False*), since this is what we would do for transfer learning. We assume the image is a colour image (3 channels RGB) with a 150x150 resolution and therefore *input_shape=(150,150,3)*. The image below shows a small part of the network (this pattern is repeated several times in the InceptionV3 model).

Let’s now inspect the ResNet50 model. This model was developed by researchers at Microsoft in 2015 too.

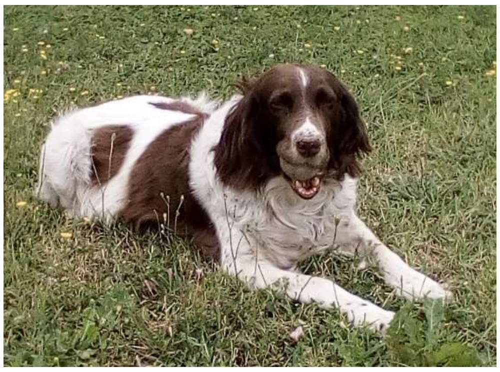
```python
# import libraries
from keras.utils import plot_model
from keras.applications.resnet50 import ResNet50
# load model
model2 = ResNet50(include_top=False, input_shape=(150,150,3))
# summarize the model
model2.summary()
plot_model(model2, show_shapes=True, to_file='model2_graph.png')
```

For this model, there are also repeated patterns with skip connexions. The image below shows one pattern.


conv3_block4_out: Activation | input: (?, 19, 19, 512) | output: (?, 19, 19, 512)
conv4_block1_1_conv: Conv2D | input: (?, 19, 19, 512) | output: (?, 10, 10, 256)
conv4_block1_1_bn: BatchNormalization | input: (?, 10, 10, 256) | output: (?, 10, 10, 256)
conv4_block1_0_conv: Conv2D | input: (?, 19, 19, 512) | output: (?, 10, 10, 1024)
conv4_block1_1_relu: Activation | input: (?, 10, 10, 256) | output: (?, 10, 10, 256)
conv4_block1_0_bn: BatchNormalization | input: (?, 10, 10, 1024) | output: (?, 10, 10, 1024)
conv4_block1_2_conv: Conv2D | input: (?, 10, 10, 256) | output: (?, 10, 10, 256)
conv4_block1_2_bn: BatchNormalization | input: (?, 10, 10, 256) | output: (?, 10, 10, 256)
conv4_block1_2_relu: Activation | input: (?, 10, 10, 256) | output: (?, 10, 10, 256)
conv4_block1_3_conv: Conv2D | input: (?, 10, 10, 256) | output: (?, 10, 10, 1024)
conv4_block1_3_bn: BatchNormalization | input: (?, 10, 10, 1024) | output: (?, 10, 10, 1024)
conv4_block1_add: Add | input: [(?, 10, 10, 1024), (?, 10, 10, 1024)] | output: (?, 10, 10, 1024)

2. Using pre-trained model as a classifier

In this section, we’ll see how to use a pre-trained model in its entirety. The model InceptionV3 has been trained with millions of images and 1000 classes. To test this model, we have to make sure the input image does not belong to the training images. This is why I’ve taken a picture of Biscotte (Biscotte is the name of my neighbor’s dog!):



We'd like to see if the model is able to recognize a dog and give its breed. The original picture resolution is 1191x875 pixels. The photo is loaded and reshaped to a 299×299 pixels size, which is the maximum resolution supported by the model. Then, the image must be converted to a numpy array. The model operates on an array of samples, therefore the dimensions of a loaded image need to be expanded by 1, for one image with 299×299 pixels and three channels.

```python
# import libraries
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.inception_v3 import preprocess_input
from keras.applications.inception_v3 import decode_predictions
from keras.applications.inception_v3 import InceptionV3

# load an image from file and set the resolution
image = load_img('biscotte.jpg', target_size=(299, 299,3))

# convert the image pixels to a numpy array
image = img_to_array(image)

# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
```

As you know, it is always necessary to apply scaling or to normalize the input of a neural network, this is what the *preprocess_input()* function does.

```python
# prepare the image for the InceptionV3 model
image = preprocess_input(image)
```

Next, the model can be loaded and a prediction made. This means that a predicted probability of the photo belonging to each of the 1,000 classes is made. In this example, we are only concerned with the most likely class, so we can decode the predictions and retrieve the label or name of the class with the highest probability.

```python
# load the model
model = InceptionV3()

# predict the probability across all output classes
yhat = model.predict(image)

# convert the probabilities to class labels
label = decode_predictions(yhat)

# retrieve the most likely result, e.g. highest probability
label = label[0][0]

# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

Let's tie all of this together:

```python
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.inception_v3 import preprocess_input
from keras.applications.inception_v3 import decode_predictions
from keras.applications.inception_v3 import InceptionV3

image = load_img('biscotte.jpg', target_size=(299, 299,3))
image = img_to_array(image)
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
image = preprocess_input(image)

model = InceptionV3()
yhat = model.predict(image)
label = decode_predictions(yhat)
label = label[0][0]

print('%s (%.2f%%)' % (label[1], label[2]*100))
```

Running this code is fast and it should return the highest probability 91,12% for an English Springer, which is the correct breed of dog!
Now you must test other models on this image. Be careful of the maximum resolution supported by the model you use. The processed image must not exceed this resolution and for some models, only specific resolutions are allowed.

3. Using the convolution base of a pre-trained model

In this section, we'll use the convolutional base of a pre-trained model and add dense layers for the output layers. Only the weights and bias of the added dense layers will have to be trained. We will show how to proceed on dataset that contains color images of cats and dogs. The problem we want to solve is a binary classification problem: the model should tell us if the image contains a cat or a dog. For that, we will use the convolutional base of the MobileNetV2 neural network. In Keras documentation, it is indicated that this model only accepts some specific image resolutions. One of them is 192x192 pixels and this is why the 2000 images of cats and dogs that are read on file have this resolution. Then 1600 images are used for training and the 400 remaining are for validation. Here the code doing this:

```
import numpy as np

# load and confirm the shape
photos = np.load('Cats_Dogs_photos.npy')
labels = np.load('Cats_Dogs_labels.npy')
print(photos.shape, labels.shape)

# specify training and validation data
X_train = photos[:1600,:,:,:]
X_test = photos[1600:,:,:,:]

y_train = labels[:1600]
y_test = labels[1600:]
```

```
(2000, 192, 192, 3) (2000,)
```

Prior to feeding the data to the convolutional base, it is necessary to preprocess our data the same way the pre-trained network expects it. For example, it may have to be normalized or scaled (by dividing the input value by 255). This must be done with the *preprocess_input()* function as follows:

```
# pre-processing input
from keras.applications.mobilenet_v2 import preprocess_input
X_train = preprocess_input(X_train)
X_test = preprocess_input(X_test)
```

Then, we can load the convolutional base model (i.e. the model without the output dense layers) by specifying *include_top=False* as argument in the following piece of code:

```
# load the convolutional base model
from keras.applications.mobilenet_v2 import MobileNetV2
base_model = MobileNetV2(include_top=False, input_shape=(192, 192, 3))
```

We have to make sure that Keras keeps the pre-trained weight and bias of the convolutional base model unchanged during the training process. This is why we set the *layer.trainable* of these layers as *False*:

```
# mark loaded layers as not trainable
for layer in base_model.layers:
    layer.trainable = False
```

Let’s inspect the output of the last layer of the convolutional base model:

```
base_model.layers[-1].output
```

```
<tf.Tensor 'out_relu/Relu6:0' shape=(None, 6, 6, 1280) dtype=float32>
```

We see that the size of the output is 6x6x1280. Before feeding this output to a dense layer, it is necessary to transform it to a vector with the *Flatten()* function:

```
# add new classifier layers
x = Flatten()(base_model.layers[-1].output)
x = Dense(128, activation='relu', kernel_initializer='he_uniform')(x)
output = Dense(1, activation='sigmoid')(x)
```

The rest of the added layers is standard: a fully connected dense layer with 128 neurons followed by the output layer with one neuron and a sigmoid as activation functions (compulsory since we are doing a binary classification).
Defining, compiling, and fitting the model is done in the same way as usual:

```
# define new model
model = Model(inputs=base_model.inputs, outputs=output)

# compile model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# fit model
model.fit(X_train, y_train, epochs=4, batch_size=64, validation_data=(X_test,y_test))
```

Only four epochs are necessary to obtain an accuracy of about 99% on the validation set. The complete code is given below in one piece.

```
import numpy as np
from keras.models import Model
from keras.layers import Dense
from keras.layers import Flatten

# load data and specify training and validation data
photos = np.load('Cats_Dogs_photos.npy')
labels = np.load('Cats_Dogs_labels.npy')

X_train = photos[:1600,:,:,:]
X_test = photos[1600:,:,:,:]
y_train = labels[:1600]
y_test = labels[1600:]

# pre-processing input
from keras.applications.mobilenet_v2 import preprocess_input
X_train = preprocess_input(X_train)
X_test = preprocess_input(X_test)

# load the convolutional base model and set layers as not trainable
from keras.applications.mobilenet_v2 import MobileNetV2
base_model = MobileNetV2(include_top=False, input_shape=(192, 192, 3))

for layer in base_model.layers:
    layer.trainable = False

# add new classifier layers
x = Flatten()(base_model.layers[-1].output)
x = Dense(128, activation='relu', kernel_initializer='he_uniform')(x)
output = Dense(1, activation='sigmoid')(x)

# define new model, compile and fit
model = Model(inputs=base_model.inputs, outputs=output)
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=4, batch_size=64, validation_data=(X_test,y_test))
```

You must build a neural network similar to the one of the previous tutorial and apply it to the cats and dogs dataset. Then, you should compare your result with the 99% accuracy of this tutorial.