Cats/Dogs classification using a reduced training dataset

1. Objectives

This laboratory is focused on the issue of cats/dogs classification using convolutional neural networks (CNN) trained with very little data (common situation encounter in practical application). Very little data in the context of deep neural networks can mean anything from a hundred to thousands of images. As a practical example, we'll focus on classifying images into categories: dogs or cats, using a dataset with 4.000 pictures (2.000 for each class). The system will use 2.000 pictures for training, 1.000 for validation and 1.000 for testing. By using the programming language Python and some dedicated machine learning platforms (*i.e.*, Tensorsflow with Keras API) the students will implement cats/dogs recognition framework trained from scratch (a new model will be developed) and will determine the system performances with respect to objective evaluation metrics such as: accuracy and loss.

2. Theoretical aspects

It is commonly said that deep learning frameworks can correctly predict the output label only when lots of training data is available in advance. This is valid in part: one fundamental characteristic of deep learning is that it can find interesting features in the training data on its own, without any need for manual feature engineering, and this can only be achieved when lots of training examples are available. But what constitutes lots of samples is relative and dependent of the size and depth of the CNN used for training.

In order to work with limited amount of data we can explore two approaches (designed to address the problem of overfitting) dropout regularization and data augmentation.

Data augmentation: Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset. Training deep learning neural network models on more data can result in more skillful models, and the augmentation techniques can create variations of the images that can improve the ability of the fit models to generalize, what they have learned, to new images.

Data augmentation can also act as a regularization technique, adding noise to the training data, and encouraging the model to learn the same features, invariant to their position in the input. Small changes to the input photos of dogs and cats might be useful for this problem, such as small shifts and horizontal flips (Fig. 1).

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by the `ImageDataGenerator` instance. The parameters that can be adjusted are presented below:

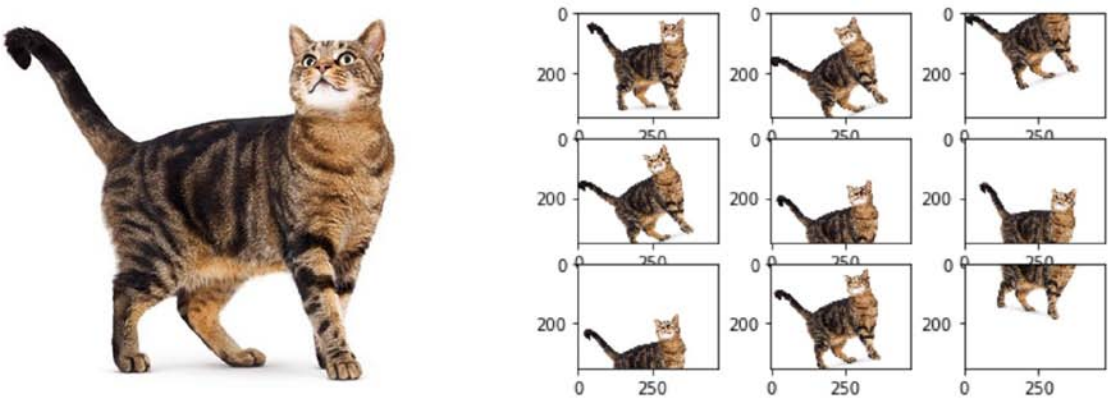

Fig.1. Data augmentation for small datasets

- `rescale` – a parameter that allow to rescale the pixel values (usually between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values);
- `rotation_range` – a parameter that allows to rotate the image and takes values between (0–180 degrees), a range within which to randomly rotate pictures;
- `width_shift_range` and `height_shift_range` - are ranges (as a fraction of total height or width) within which to randomly translate pictures vertically or horizontally axes;
- `shear_range` – a parameter that allows to randomly apply shearing transformations;
- `zoom_range` – a parameter that allows randomly zooming inside pictures;
- `horizontal_flip` – a parameter allowing randomly flipping images horizontally;
- `fill_mode` – is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Dropout regularization - is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly (Fig. 2). This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of the neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data.

You can imagine that if neurons are randomly dropped out of the network during training that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network. The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

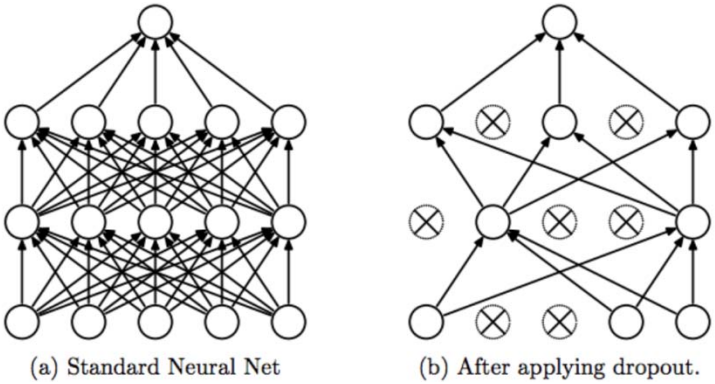

Fig.2. Dropout example

Transfer learning: A pre-trained model is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. The intuition behind transfer learning for image classification is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model of the visual world. You can then take advantage of these learned feature maps without having to start from scratch by training a deep model on a large dataset (Fig. 3).

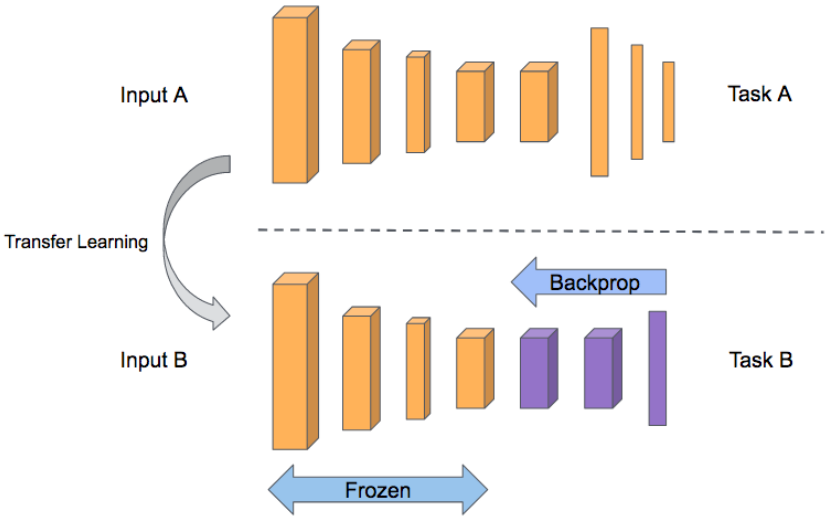

Fig.3. CNN transfer learning

In this laboratory, you will try two ways to customize a pre-trained model in order to perform relevant feature extraction from images. Use the representations learned by a previous network to extract meaningful features from new samples. You simply add a new classifier, which will be trained from scratch, on top of the pre-trained model so that you can repurpose the feature maps learned previously for the dataset.

You do not need to (re)train the entire model. The base convolutional network already contains features that are generically useful for classifying pictures. However, the final, classification part of the pre-trained model is specific to the original classification task, and subsequently specific to the set of classes on which the model was trained. The most common approach of transfer learning in the context of deep learning consists in the following steps:

1. Take layers from a previously trained model.
2. Freeze them, in order to avoid destroying any of the information they contain during future training rounds.
3. Add some new, trainable layers on top of the frozen layers.
4. Train the new layers on your dataset.

A last, optional step, is **fine-tuning**, which consists of unfreezing the entire model you obtained above (or some part of it), and re-training it on the new data with a very low learning rate. This can potentially achieve meaningful improvements, by incrementally adapting the pre-trained features to the new data.

3. Practical implementation

The following example aims to determine the optimal solution for classifying cats/dogs using the Kaggle dataset. The application will demonstrate that is possible to train a CNN from scratch on a very small image dataset which will still yield reasonable results despite a relative lack of data.

Different from the MNIST Digits or MNIST Fashion datasets used in the previous laboratories, the Dogs vs. Cats dataset is not packaged with Keras. However, the dataset is made available by Kaggle as part of a computer-vision competition in late 2013. For your convenience the Dogs vs. Cats dataset can be found on the folder Kaggle_Cats_And_Dogs_Dataset. The pictures are medium-resolution color JPEGs. This dataset contains 25.000 images of dogs and cats (12.500 for each class) and is 543 MB (compressed). After downloading and uncompressing it, you'll create a new dataset containing three subsets: a training set with 1.000 samples for each class, a validation set with 500 samples for each class, and a test set with 500 samples for each class.

Pre-requirements: Copy the CatsAndDogsClassification.py Python scripts in the project "ArtificialIntelligenceProject" main directory (developed in the laboratory: "Handwritten digit classification using ANN/CNN").

Application 1: Consider the Dogs vs. Cats dataset, complete the CatsAndDogsClassification.py script (using the dedicated image machine learning platforms (*i.e.,* Tensorsflow with Keras API)) able to recognize the category for an unknown image applied as input. There are 2 classes to predict. The system performance needs to be evaluated using the classification accuracy.

Step 1: *Prepare the training/validation and testing dataset* – From the complete Kaggle Dogs vs. Cats dataset create a novel directory denoted Kaggle_Cats_And_Dogs_Dataset_Small that will store the reduced testing dataset. As mentioned previously the training set will contain 1.000 samples of each class, the validation set 500 samples of each class and testing set with 500 samples of each class.

Write a Python function denoted `def prepareDatabase(original_directory, base_directory)` that takes as input the `original_directory` (containing the complete Kaggle dataset) and location where the reduced size dataset will be stored (`base_directory`). Within the `base_directory` folder the dataset will be structure in three independent directories denoted: train, validation and test (Fig. 4):

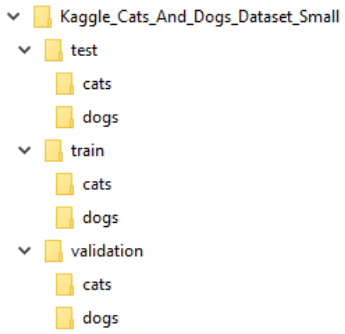

Fig.4. The directories structure in the small dataset

Step 1a: In the `prepareDatabase()` function create using Python code the *train* folder:

```
def prepareDatabase(original_directory, base_directory):

      train_directory = os.path.join(base_directory, 'train')
      os.mkdir(train_directory)
```

Step 1b: In the `prepareDatabase()` function create using Python code the *validation* folder.

Step 1c: In the `prepareDatabase()` function create using Python code the *test* folder.

Step 1d: In each directory (train, validation and test) create another two subdirectories denoted: cats and dogs.

Step 1e: Copy the first 1000 images with cats' images from the `original_directory` into the cats training folder (`train_cats_directory`).

```
#Copy the first 1000 cat images into the training directory
(train_cats_directory)

original_directory_cats = str(original_directory + '/cats/')
fnames = ['{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_directory_cats, fname)
    dst = os.path.join(train_cats_directory, fname)
    shutil.copyfile(src, dst)
```

Step 1f: Copy the next 500 cats' images from the `original_directory` into the cats validation folder (`validation_cats_directory`).

Step 1g: Copy the following 500 images into the cats testing folder (`test_cats_directory`).

Step 1h: Copy the first 1000 images with dogs' images from the `original_directory` into the dogs training folder (`train_dogs_directory`).

Step 1i: Copy the next 500 dogs' images from the `original_directory` into the dogs' validation folder (`validation_dogs_directory`).

Step 1j: Copy the following 500 images into the dogs' testing folder (`test_dogs_directory`).

Step 1k: *Visualize the training/validation/testing folders structure* – As a sanity check, let's count how many pictures are in each training folder (train/validation/test):

```
#As a sanitary check verify how many pictures are in each directory
print('Total number of CATS used for training =
{}'.format(len(os.listdir(train_cats_directory))))
print('Total number of CATS used for validation =
{}'.format(len(os.listdir(validation_cats_directory))))
print('Total number of CATS used for testing =
{}'.format(len(os.listdir(test_cats_directory))))

print('Total number of DOGS used for training =
{}'.format(len(os.listdir(train_dogs_directory))))
print('Total number of DOGS used for validation =
{}'.format(len(os.listdir(validation_dogs_directory))))
print('Total number of DOGS used for testing =
{}'.format(len(os.listdir(test_dogs_directory))))
```

You will need to have 2.000 training images, 1.000 validation images, and 1.000 test images. Each split contains the same number of samples from each class: this is a balanced binary-classification problem, which means classification accuracy will be an appropriate measure of success.

Observation: Once the dataset is created successfully (see Fig. 4), comment the `prepareDatabase()` function so you will not repeat the process each time the code is run.

Step 2: *Image preprocessing* – As you know by now, data should be formatted into appropriately preprocessed floating point tensors before being fed into the network. Currently, the data from the different training, validation and testing folders (*cf.* Step 1) contains JPEG files. In this stage we will define a function denoted `def image_preprocessing(base_directory)` that takes as input the `base_directory` directory and will perform the following operations:

- Read the picture files;
- Decode the JPEG content to RGB grids of pixels;
- Convert these into floating-point tensors;

- Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

This process can be performed somehow automatically by using the Keras image-processing module, located in the `keras.preprocessing.image` library. In particular, the library contains the class ImageDataGenerator, which lets operate with Python generators that can automatically turn image files located on the disk into batches of preprocessed tensors.

```
def imagePreprocessing(base_directory):

    train_directory = base_directory + '/train'
    validation_directory = base_directory + '/validation'

    train_datagen = ImageDataGenerator(rescale=1./255)
    validation_datagen = ImageDataGenerator(rescale=1./255)

    train_generator = train_datagen.flow_from_directory(train_directory,
target_size = (150, 150), batch_size = 20, class_mode='binary')
    validation_generator =
validation_datagen.flow_from_directory(validation_directory,target_size = (150,
150), batch_size = 20, class_mode='binary')
```

Let’s look at the output of one of these generators: it yields batches of 150 × 150 RGB images (shape (20, 150, 150, 3)) and binary labels (shape (20,)). There are 20 samples in each batch (the batch size).

Note: the generator yields these batches of files indefinitely (it loops endlessly over the images in the target folder). For this reason, you need to break the iteration loop at some point.

```
#Application 1 - Step 2 - Analyze the output of the train and validation generators

    for data_batch, labels_batch in train_generator:
        print('Data batch shape in train: ', data_batch.shape)
        print('Labels batch shape in train: ', labels_batch.shape)
        break

    for data_batch, labels_batch in validation_generator:
        print('Data batch shape in validation: ', data_batch.shape)
        print('Labels batch shape in validation: ', labels_batch.shape)
        break
```

The function will return the two generators.

```
    return train_generator, validation_generator
```

Step 3: *Build your CNN architecture* – We will define our model in a function denoted `defineCNNModelFromScratch()`. This is handy if you want to extend the example later and try to get a better accuracy score. We will define a simple structure that uses all of the elements for state of the art results.

Step 3a: Start by instantiating the `Sequential` model with:

```
model = models.Sequential()
```

We will consider input images of size 150 × 150 (or somewhat arbitrary choice). The layers of the CNN architecture are presented below:

Step 3b: The first hidden layer is a convolutional layer called **Convolution2D** (Conv2D). The layer has 32 kernels, of size 3×3. This layer expects as input images with the structure: [width][height][channels] (`input_shape=(150, 150, 3)`). The activation function is Relu (`activation='relu'`).

Step 3c: Next we define a pooling layer called **MaxPooling2D** that takes the maximum value of a block of size 2x2. The layer is configured by default with a pool size of 2×2.

Step 3d: The third hidden layer is a convolutional layer called **Convolution2D** (Conv2D). The layer has 64 feature maps of size of 3×3. The activation function is Relu (`activation='relu'`).

Step 3e: Next we define a pooling layer called **MaxPooling2D** that takes the maximum value of a block of size 2x2. The layer is configured by default with a pool size of 2×2.

Step 3f: A convolutional layer called **Convolution2D** (Conv2D). The layer has 128 feature maps of size of 3×3. The activation function is Relu (`activation='relu'`).

Step 3g: A pooling layer called **MaxPooling2D** that takes the maximum value of a block of size 2x2. The layer is configured by default with a pool size of 2×2.

Step 3h: A convolutional layer called **Convolution2D** (Conv2D). The layer has 128 feature maps of size of 3×3. The activation function is Relu (`activation='relu'`).

Step 3i: A pooling layer called **MaxPooling2D** that takes the maximum value of a block of size 2x2. The layer is configured by default with a pool size of 2×2.

Step 3j: The next layer is called **Flatten** and converts the 2D matrix data to a vector. It allows the output to be processed by standard fully connected layers.

Step 3k: Next a fully connected (**Dense**) layer with 512 neurons and rectifier activation function is added (`activation='relu'`).

Step 3l: The problem is a binary classification task, requiring the prediction of one value of either 0 or 1. In this case, the output layer with contain only 1 node and a sigmoid activation (`activation='sigmoid'`) will be used.

Step 3m: Let's look at how the dimensions of the feature maps change with every successive layer: `model.summary()`

Step 3n: Before we can begin training, we need to configure the training process. We set 3 key factors for the compilation: the optimizer (*i.e.*, the RMSprop optimizer), the loss function (*i.e.*, the cross-entropy) and the performance metric (*i.e.*, since this is a classification problem, the accuracy metric will be used).

```
model.compile(optimizer=rmsprop_v2.RMSprop(learning_rate=0.0001),
loss='binary_crossentropy', metrics=['accuracy'])
```

Step 4: *Train the model* – The model training will be performed using the Keras function `fit_generator`. The function takes as input the training generator, the validation generator together with some training hyper-parameters as: the number of epochs (iterations over the entire dataset), the number of batches in an epoch (`steps_per_epoch`) and number of batches for the validation dataset (when the epoch ends the validation generator will yield `validation_steps` batches).

```
    #train the model
    history = model.fit_generator(train_generator, steps_per_epoch=100, epochs=10,
validation_data=validation_generator, validation_steps=50)
```

This method will be called in the main function (*cf.* Step 6)!

Step 5: *Display the system performance* – In this step we will plot the loss and accuracy function of the model over the training and validation data. The system performance will be define within a function denoted `def visualizeTheTrainingPerformance(history)` that takes as input the `history` (that contains the model accuracy and loss obtained on the test and training dataset at the end of each epoch).

```
def visualizeTheTrainingPerformances(history):

    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']

    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs = range(1, len(acc) + 1)
    pyplot.title('Training and validation accuracy')
    pyplot.plot(epochs, acc, 'bo', label = 'Training accuracy')
    pyplot.plot(epochs, val_acc, 'b', label = 'Validation accuracy')
    pyplot.legend()
```

```
    pyplot.figure()
    pyplot.title('Training and validation loss')
    pyplot.plot(epochs, loss, 'bo', label = 'Training loss')
    pyplot.plot(epochs, val_loss, 'b', label = 'Validation loss')
    pyplot.legend

    pyplot.show()

    return
```

Exercise 1: Save the figures generated by the function `visualizeTheTrainingPerformance()` when performing the training for 100 epochs.

Exercise 2: Evaluate the model accuracy on the testing dataset.

Hint: First, create a data generator for the testing dataset as for the training and the validation datasets (*cf.* Step 2). Next, use the `model.evaluate_generator` function to obtain the loss and accuracy.

Exercise 3: Once fit, we can save the final model to an *.h5 file by calling the *save()* function on the model and pass in the selected filename (`model.save('Model_cats_dogs_small_dataset.h5')`). We can use our saved model to make a prediction on new images. Using the pre-trained model (saved above) write a Python script able to make an automatic prediction regarding the category for the following images "test1.jpg" and "test2.jpg". The prediction will be performed simultaneously for the two images using a batch of images.

Hint: First, load the images with OpenCV, and then convert them to RGB images. Resize the images at 150×150 pixels (as for the other images used for training). Normalize the pixel values. This process should be implemented within a `def loadImages()` function that will return a tensor of size`(2, 150, 150, 3)`, ready for classification purposes.

Exercise 4: Try to further increase the system performances (by reducing the overfitting) by adding a dropout layer to your model that randomly excludes 50% of neurons. The layer should be added right before the densely connected classifier.

In order to further improve the system performance increase the dataset used for training by performing some data augmentation techniques. Let's train the network for 100 epochs using data augmentation techniques as presented bellow. How is the system accuracy influenced by the data augmentation techniques? Compare the graphs with the figure saved at Exercise 1.

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
```

```
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

Application 2: In this example, you will need to classify images of cats and dogs by using transfer learning from a pre-trained network. In the **defineCNNModelVGGPretrained** method, load the VGG16 network, previously trained on ImageNet, to extract interesting features from cat and dog images, and then (using the pretrained filters) train a dogs-versus-cats classifier on top of these features.

The VGG16 architecture comes prepackaged with Keras library (`from keras.applications.vgg16 import VGG16`).

Step 1: *Load the VGG16 architecture*– First, load the `VGG16()` model pre-trained on the ImageNet dataset (`weights='imagenet'`), in a variable called `baseModel.` The top layers should be omitted (`include_top=False`), while the input shape is structured as: [width][height][channels] (`input_shape=(150, 150, 3)`).

Step 2: *Visualize the CNN architecture* – Display the structure of the VGG16 network architecture and determine the total number of trainable parameters.

Step 3: *Freeze the convolutional layers of the CNN architecture* – The VGG16 convolutional layers parameters should not be modified during training:

```
for layer in baseModel.layers:
    layer.trainable = False
```

Step 4: *Create the final CNN architecture* **–** The novel CNN model will be defined as sequential model `VGG_model = models.Sequential(),` and will contain the corresponding layers from the `baseModel` (`VGG_model.add(baseModel)`) and finally add the following layers:

Step 4a: The next layer is called **Flatten** and converts the 3D tensor data to a vector. It allows the output to be processed by standard fully connected layers.

Step 4b: Add a dropout (**Dropout**) layer to your model that randomly excludes 50% of neurons.

Step 4c: Next a fully connected (**Dense**) layer with 512 neurons and rectifier activation function is added (`activation='relu'`).

Step 4d: The problem is a binary classification task, requiring the prediction of one value of either 0 or 1. In this case, the output layer with contain only 1 node and a sigmoid activation (`activation='sigmoid'`) will be used.

Step 4e: Before we can begin training, we need to configure the training process. We set 3 key factors for the compilation: the optimizer (*i.e.*, the RMSprop optimizer), the loss function (*i.e.*, the cross-entropy) and the performance metric (*i.e.*, since this is a classification problem, the accuracy metric will be used).

```
VGG_model.compile(optimizer=rmsprop_v2.RMSprop(learning_rate=0.0001),
loss='binary_crossentropy', metrics=['accuracy'])

return VGG_model
```

Finally, the model will be trained and the performance curves will be displayed.

Exercise 5: In this exercise, you will need to classify images of cats and dogs by using transfer learning and fine-tunning. Write a Python script that uses the VGG16 network, previously trained on ImageNet, to extract interesting features from cat and dog images, unfreeze some of the top layers of the frozen model and jointly train both the newly-added classifier layers and the last layers of the base model. This allows us to "fine-tune" the higher-order feature representations in the base model in order to make them more relevant for the specific task.

You'll fine-tune the last three convolutional layers, which mean that all layers up to `block4_pool` should be frozen, and the other layers `block5_conv1`, `block5_conv2`, and `block5_conv3` should be trainable.

Exercise 6: Write a Python script that uses the network architectures presented in Table 1, previously trained on ImageNet, to extract interesting features from cat and dog images, and then (using the pretrained filters) train a dogs-versus-cats classifier on top of these features. How is the system accuracy influenced by this network topology type?

Table 1. System performance evaluation for various numbers of neurons on the hidden layers

CNN architecture	VGG16	Xception	Inception	ResNet50	MobileNet
System accuracy					

You can import in Keras multiple pre-trained models (on ImageNet dataset) such as: VGG16, Xception, Inception V3, ResNet50, VGG19 or MobileNet.