Parcours Data Scientist, projet n°7, Notebook d'exploration et des modèles

Christophe Coudé, 2017

Plan.

I- Objectifs

La base de données des pensionnaires d'une association de protection des animaux commence à s'agrandir et ils n'ont pas toujours le temps de référencer les images des animaux qu'ils ont accumulées depuis plusieurs années. Ils aimeraient réaliser un index de l'ensemble de la base de données d'images qu'ils possèdent, pour classer les chiens par race.

- Le but de ce projet est de réaliser un algorithme de détection de la race d'un chien sur une photo, afin d'accélérer leur travail d'indexation.
- Un programme devra prendre en entrée une photo et retourner la race la plus probable du chien présent sur cette photo.

Base de données : http://vision.stanford.edu/aditya86/ImageNetDogs/

Ces images sont extraites de la base de donnée ImageNet (http://www.image-net.org/), projet de recherche développant une importante base de données regroupant des images et leur description.

Remarque : la plupart des réseaux de neurones effectuant beaucoup d'opérations, ils nécessitent une puissance de calcul importante et beaucoup de mémoire. Il faudrait donc disposer de machines possédant des GPU, ce qui n'est pas mon cas, la mienne n'ayant que 4 CPU. Je n'ai donc pas pu faire beaucoup de tests (et avec peu d'itérations), ce qui limite grandement les possibilités de comparaison entre les modèles. J'ai donc décrit un peu de théorie dans les différents modèles en attendant de pouvoir disposer d'ordinateurs plus puissants permettant de faire plus de tests et d'expérimentations.

II- Exploration

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings(action='ignore', category=UserWarning)
%matplotlib inline
import logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
from os.path import join
import sys
import os
import cv2
import datetime
import time
from random import shuffle
from scipy import io
from itertools import zip_longest
import seaborn as sns
from lxml import etree
from mpl_toolkits.axes_grid1 import ImageGrid

import keras
from keras.applications import imagenet_utils
from keras.applications.imagenet_utils import preprocess_input
from keras.preprocessing import image
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Flatten, Activation
from keras.models import Model # basic class for specifying and training a neural network
from keras.layers import Input, Convolution2D, MaxPooling2D, Dense, Dropout, Flatten
from keras.utils import np_utils # utilities for one-hot encoding of ground truth values
from keras.optimizers import SGD
from keras.callbacks import EarlyStopping
from sklearn.metrics import log_loss
from keras import __version__ as keras_version
from keras.constraints import maxnorm
from keras.initializers import RandomNormal
from keras.regularizers import l1, l2
```

```
Using TensorFlow backend.
```

In [2]:
```
# répertoire où j'ai mis localement tous les fichiers des images et des descriptifs, à modifier si nécessaire
localProjectDirectory = "...spyder/notebooks/projet7/"
```

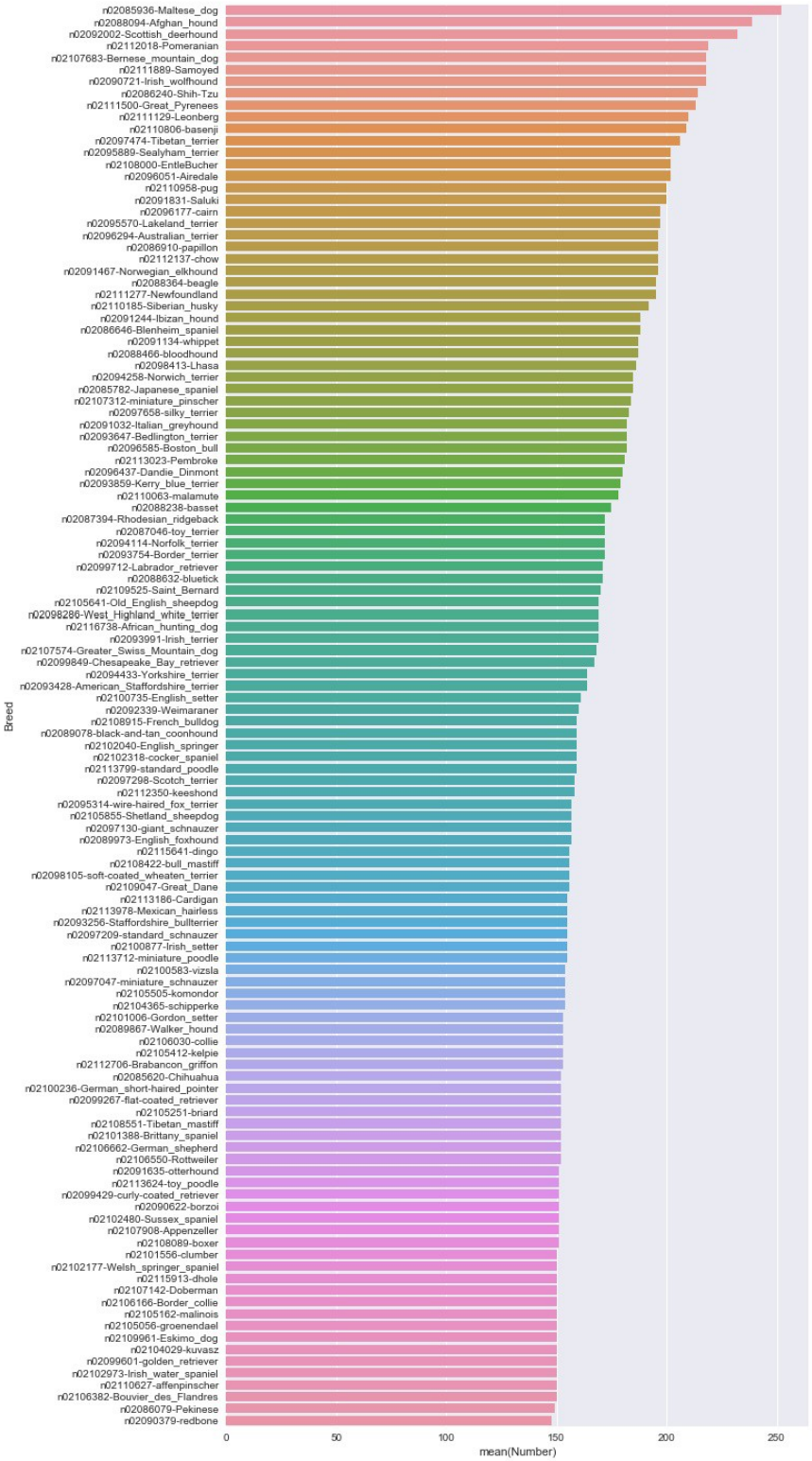
In [3]:
```
data1=io.loadmat(localProjectDirectory+'train_data.mat')
data2=io.loadmat(localProjectDirectory+'test_data.mat')

# les fichiers train_list.mat et test_list.mat contiennent la description des répertoires et noms des images d'entraînement e
t de test.
# J'ai toutefois téléchargé et déployé les images réelles en local sur mon ordinateur.
trainList=io.loadmat(localProjectDirectory+'train_list.mat')
testList=io.loadmat(localProjectDirectory+'test_list.mat')
dogImagesDir = localProjectDirectory+"Images/"

# séparation des répertoires des données d'entraînement et de validation
train_data_dir = localProjectDirectory+'train'
validation_data_dir = localProjectDirectory+'validation'

# le dataset du projet contient 20 580 images 224x224 réparties dans 120 catégories. C'est beaucoup trop pour les pauvres CPU
de
# mon ordinateur lors de certains modèles, j'ai donc parfois restreint le nombre de catégories de chien.
someBreeds = ['n02085620-Chihuahua', 'n02087394-Rhodesian_ridgeback', 'n02089973-English_foxhound', 'n02092002-Scottish_deerh
ound', 'n02096585-Boston_bull',
              'n02092339-Weimaraner', 'n02095570-Lakeland_terrier', 'n02099429-curly-coated_retriever', 'n02106030-collie', 'n
02107908-Appenzeller']
# toutes les catégories
allBreeds = os.listdir(dogImagesDir)
```

Nombre d'images par race

In [4]:
```
def getTrainBreedDogDico(_trainList, _breedsChoice):
    """ Méthode qui crée le dictionnaire des noms de répertoire auxquels on a associé la liste des noms de fichiers des image
s des chiens pour les images d'entraînement
    Arguments:
    _trainList -- la liste de la description des données d'entraînement
    _breedsChoice -- la liste des races de chien retenues
    Retour:
    trainDico -- le dictionnaire des noms de répertoire auxquels on a associé la liste des noms de fichiers des images des ch
iens pour les images d'entraînement
    """
    trainDico = {}
    for dog in _trainList['file_list']:
        temp = dog[0][0].split('/')
        if temp[0] in _breedsChoice:
            if temp[0] not in trainDico:
                trainDico[temp[0]] = []
            trainDico[temp[0]].append(temp[1])

    # trainDico : {'n02085620-Chihuahua': ['n02085620_5927.jpg', 'n02085620_4441.jpg', ... ], 'n02089973-English_foxhound':
...}
    return trainDico

def getTestBreedDogDico(_testList, _breedsChoice):
    """ Méthode qui crée le dictionnaire des noms de répertoire auxquels on a associé la liste des noms de fichiers des image
s des chiens pour les images de test
    Arguments:
    _testList -- la liste de la description des données de test
    _breedsChoice -- la liste des races de chien retenues
    Retour:
    testDico -- le dictionnaire des noms de répertoire auxquels on a associé la liste des noms de fichiers des images des chi
ens pour les images de test
    """
    testDico = {}
    for dog in _testList['file_list']:
        temp = dog[0][0].split('/')
        if temp[0] in _breedsChoice:
            if temp[0] not in testDico:
                testDico[temp[0]] = []
            testDico[temp[0]].append(temp[1])
    return testDico

def somme(_tup):
    """ Méthode qui somme tous les éléments d'un tuple
    Arguments:
    _tup -- le tuple dont on veut la somme des éléments
    Retour:
    res -- la somme de tous les éléments du tuple d'entrée
    """
    res = 0
    for n in _tup:
        res += int(n)
    return res

def getDogBreedIdFromFileName(_fileName):
    """ Méthode qui détermine l'identifiant d'un répertoire/race à partir du nom de fichier d'un chien
    Arguments:
    _fileName -- le nom de fichier d'un chien
    Retour:
    identifiant -- l'identifiant du répertoire/race à partir de l'image du chien
    """
    identifiant = _fileName.split("_")[0]
    # n02085620_5927.jpg --> n02085620
    return identifiant
```

```python
def getDogBreedDirFromFileName(_fileName, _dicoBreedIdName):
    """ Méthode qui détermine l'identifiant/nom de la race du chien contenu dans l'image du fichier donné en entrée
    Arguments:
    _fileName -- le nom de fichier d'un chien
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    Retour:
    breedIdName -- l'identifiant/nom de la race du chien
    """
    breedId = getDogBreedIdFromFileName(_fileName)
    breedName = _dicoBreedIdName[breedId]
    breedIdName = breedId + "-" + breedName
    # n02085620_5927.jpg --> n02085620-Chihuahua
    return breedIdName

def getTrainBreedDirAndImageNameList(_trainList, _breedsChoice):
    """ Méthode qui crée les listes des noms de répertoire (id/nom) des races et des noms de fichiers des images des chiens p
our les images d'entraînement
    Arguments:
    _trainList -- la liste de la description des données d'entraînement
    _breedsChoice -- la liste des races de chien retenues
    Retour:
    idBreedDirTrainList -- la liste des noms de répertoire (id/nom) des races des images d'entraînement
    idImageTrainList -- la liste des noms de fichiers des images des chiens pour les images d'entraînement
    """
    idBreedDirTrainList = []
    idImageTrainList = []
    for dog in _trainList['file_list']:
        temp = dog[0][0].split('/')
        if temp[0] in _breedsChoice:
            idBreedDirTrainList.append(temp[0])
            idImageTrainList.append(temp[1])
    # idBreedDirTrainList : ['n02085620-Chihuahua', 'n02085620-Chihuahua',
    # idImageTrainList : ['n02085620_5927.jpg', 'n02085620_4441.jpg',
    return idBreedDirTrainList, idImageTrainList
def getTestBreedImageNameList(_testList, _breedsChoice):
    """ Méthode qui crée la liste des noms de fichiers des images des chiens pour les images de test
    Arguments:
    _testList -- la liste de la description des données de test
    _breedsChoice -- la liste des races de chien retenues
    Retour:
    idImageTestList -- la liste des noms de fichiers des images des chiens pour les images de test
    """
    idImageTestList = []
    for dog in _testList['file_list']:
        temp = dog[0][0].split('/')
        if temp[0] in _breedsChoice:
            idImageTestList.append(temp[1])
    # idImageTestList : ['n02085620_2650.jpg', 'n02085620_4919.jpg',
    return idImageTestList

def createDicoBreedIdName(_idBreedDirTrainList):
    """ Méthode qui crée le dictionnaire des noms de race pour chaque identifiant de race
    Arguments:
    _idBreedDirTrainList -- la liste des noms de répertoire (id/nom) des races des images d'entraînement
    Retour:
    dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    """
    dicoBreedIdName = {}
    for idName in _idBreedDirTrainList:
        indexFirstDash = idName.index('-') # il peut y avoir plusieurs '-' dans le nom de la race...
        if indexFirstDash > 0:
            dicoBreedIdName[idName[:indexFirstDash]] = idName[indexFirstDash+1:]
    # dicoBreedIdName : {'n02085620': 'Chihuahua', 'n02087394': 'Rhodesian_ridgeback',
    return dicoBreedIdName


def createDicoDogBreedIdNameCateg(_breedsChoice):
    """ Méthode qui crée les dictionnaires des positions des répertoire/races des chiens et des positions des identifiants de
s races des chiens
    Arguments:
    _breedsChoice -- la liste des races retenues
    Retour:
    dicoDogBreedIdNameCateg -- le dictionnaire des positions des répertoires/races des chiens
    dicoDogBreedIdCateg -- le dictionnaire des positions des identifiants des races des chiens
    """
    dicoDogBreedIdNameCateg = {}
    dicoDogBreedIdCateg = {}
    for i, breedIdName in enumerate(_breedsChoice):
        dicoDogBreedIdNameCateg[breedIdName] = i
        indexFirstDash = breedIdName.index('-')
        if indexFirstDash > 0:
            dicoDogBreedIdCateg[breedIdName[:indexFirstDash]] = i
    # dicoDogBreedIdNameCateg : {'n02085620-Chihuahua': 0, 'n02085782-Japanese_spaniel': 1,
    # dicoDogBreedIdCateg : {'n02085620': 0, 'n02085782': 1,
    return dicoDogBreedIdNameCateg, dicoDogBreedIdCateg
```

```python
def getDogBreedNameFromFileName(_fileName, _dicoBreedIdName):
    """ Méthode qui détermine le nom de la race du chien contenu dans l'image du fichier donné en entrée
    Arguments:
    _fileName -- le nom de fichier d'un chien
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    Retour:
    breedName -- le nom de la race du chien
    """
    breedName = _dicoBreedIdName[getDogBreedIdFromFileName(_fileName)]
    # n02085620_5927.jpg --> Chihuahua
    return breedName

def getDogBreedCategFromFileName(_dicoDogBreedIdCateg, _fileName):
    """ Méthode qui détermine l'index du répertoire de la race du chien contenu dans l'image du fichier donné en entrée
    Arguments:
    _dicoDogBreedIdCateg -- dictionnaire des positions des identifiants des races des chiens
    _fileName -- le nom de fichier d'un chien
    Retour:
    index -- l'index du répertoire de la race du chien
    """
    # n02087394_36.jpg --> 8 (n02087394-Rhodesian_ridgeback = 9ième répertoire)
    index = _dicoDogBreedIdCateg[getDogBreedIdFromFileName(_fileName)]
    return index

def readImage(_dogImagesDir, _dicoBreedIdName, _fileName, _size):
    """ Méthode qui lit une image d'un répertoire et la transforme en tableau de scalaires
    Arguments:
    _dogImagesDir -- chemin vers le répertoire local des images
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    _fileName -- le nom de fichier d'un chien
    _size -- la taille que doit prendre l'image, sous format (hauteur, largeur), ex : (224, 224)
    Retour:
    img -- image transformée en tableau de scalaires
    """
    dataDir = getDogBreedDirFromFileName(_fileName, _dicoBreedIdName)
    img = image.load_img(_dogImagesDir+dataDir+"/"+_fileName+".jpg", target_size=_size)
    img = image.img_to_array(img)
    return img

def showImage(_dogImagesDir, _dicoBreedIdName, _fileName, _size):
    """ Méthode qui affiche à l'écran l'image d'un chien
    Arguments:
    _dogImagesDir -- chemin vers le répertoire local des images
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    _fileName -- le nom de fichier d'un chien
    _size -- la taille que doit prendre l'image, sous format (hauteur, largeur), ex : (224, 224)
    """
    img = readImage(_dogImagesDir, _dicoBreedIdName, _fileName, _size)
    plt.axis('off')
    plt.imshow(img / 255.)

def showExternalImage(_imagePath, _size):
    """ Méthode qui affiche à l'écran l'image d'un chien ne provenant pas d'ImageNet
    Arguments:
    _imagePath -- chemin vers l'image extérieure d'un chien
    _size -- la taille que doit prendre l'image, sous format (hauteur, largeur), ex : (224, 224)
    """
    img = image.load_img(_imagePath, target_size=_size)
    img = image.img_to_array(img)
    plt.axis('off')
    plt.imshow(img / 255.)
idAllBreedDirTrainList, _ = getTrainBreedDirAndImageNameList(trainList, allBreeds)

# dictionnaire des identifiants de race pour chaque nom de race
# dicoBreedIdName : {'n02085620': 'Chihuahua', 'n02087394': 'Rhodesian_ridgeback', ...
dicoBreedIdName = createDicoBreedIdName(idAllBreedDirTrainList)

trainDico = getTrainBreedDogDico(trainList, allBreeds)
testDico = getTestBreedDogDico(testList, allBreeds)

trainBreeds = []
trainHoundsNumber = []
testBreeds = []
testHoundsNumber = []

# nombres de chiens par race pour les données d'entraînement
for breed in trainDico.keys():
    trainBreeds.append(breed)
    trainHoundsNumber.append(len(trainDico[breed]))

# nombres de chiens par race pour les données de test
for breed in testDico.keys():
    testBreeds.append(breed)
    testHoundsNumber.append(len(testDico[breed]))

tabs = (trainHoundsNumber, testHoundsNumber)
houndsNumber = [somme(val) for val in zip_longest(*tabs, fillvalue = '0')]
dogsDataframe = pd.DataFrame({'Breed': trainBreeds, 'Number': houndsNumber})
dogsDataframeSorted = dogsDataframe.sort_values(by = 'Number', ascending=False)

f, ax = plt.subplots(figsize=(10, 25))
sns.barplot(x='Number', y='Breed', data=dogsDataframeSorted)
plt.show();
```



Il y a toujours au moins presque 150 images par race, ce qui fait que pour chaque race, 100 images servent constamment d'entraînement, les autres images servant de test. La nombre d'images de test varie donc suivant la race, la taille de l'entraînement étant, lui, fixe.

Il serait intéressant de voir quelles sont les catégories de chiens que les modèles ont le plus de mal à classer.

Répartition des tailles des images

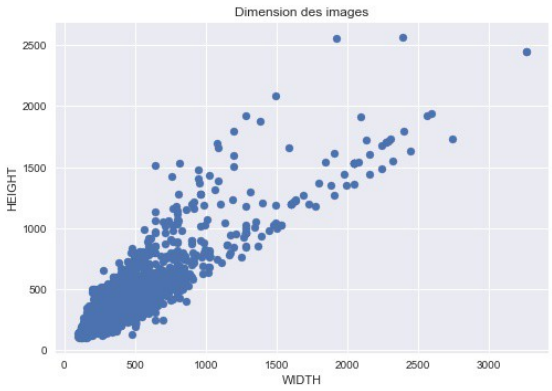
Les images sont de tailles très diverses, allant de 97 à 3264 pixels en largeur et de 100 à 2562 en hauteur. On est donc loin de la taille des images utilisées dans les différents modèles pré-entraînés qui sont en général de 224x224 ou 229x229. On devra donc restreindre le nombre de pixels de ces images, avec potentiellement un problème de reconnaissance due à cette perte d'information.

In [6]:
```
# les informations relatives aux images se trouvent dans le dossier Annotation
# on va donc faire pour chaque image un parsing XML pour récupérer la largeur (width) et la hauteur (height),
# la profondeur étant la même pour toutes les images, 3 (RGB).

breeds = []
widths = []
heights = []
dogRealWidths = []
dogRealHeights = []
dogs = []

# répertoire des fichiers d'annotation
annotDir = localProjectDirectory+"Annotation"

allBreedsAnnot = os.listdir(annotDir)
for breed in allBreedsAnnot:
    allDogsAnnotFromBreedDir = os.listdir(annotDir+"/"+breed)
    for dog in allDogsAnnotFromBreedDir:
        dogs.append(dog)
        breeds.append(breed)
        tree = etree.parse(annotDir+"/"+breed+"/"+dog)
        width = int(tree.xpath("/annotation/size/width")[0].text)
        widths.append(width)
        height = int(tree.xpath("/annotation/size/height")[0].text)
        heights.append(height)
        xmin = int(tree.xpath("/annotation/object/bndbox/xmin")[0].text)
        ymin = int(tree.xpath("/annotation/object/bndbox/ymin")[0].text)
        xmax = int(tree.xpath("/annotation/object/bndbox/xmax")[0].text)
        ymax = int(tree.xpath("/annotation/object/bndbox/ymax")[0].text)
        dogRealWidths.append(xmax-xmin)
        dogRealHeights.append(ymax-ymin)
print("minWidth: {0}, maxWidth: {1}, minHeight: {2}, maxHeight: {3}".format(min(widths), max(widths), min(heights), max(heigh
ts)))
```

```
minWidth: 97, maxWidth: 3264, minHeight: 100, maxHeight: 2562
```

On voit ci-dessous la répartition de la taille des images pour tout le dataset:

In [7]:
```
plt.scatter(widths, heights)
plt.xlabel('WIDTH')
plt.ylabel('HEIGHT')
plt.title('Dimension des images')
plt.show()
```



On est très souvent nettement au-delà des tailles d'image utilisées par les modèles pré-entraînés.

Répartition particulière de la taille des images des chiens seuls dans les images

Les fichiers du répertoire annotation contiennent des informations sur la localisation précise des chiens dans les images (sous le chemin /annotation/object /bndbox). Elles peuvent permettre de se faire éventuellement une idée sur la taille moyenne des chiens suivant leur race et voir ultérieurement si cela a une influence sur le résultat des prévisions.

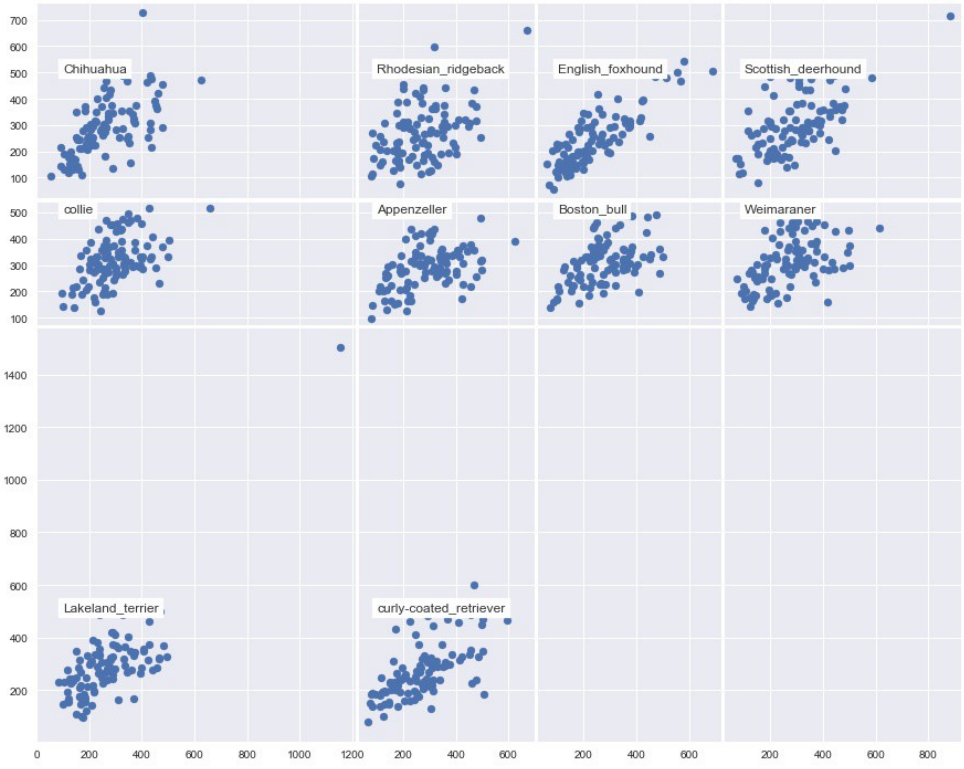
In [8]:
```
print(breeds[10], dogs[10], widths[10], heights[10], dogRealWidths[10], dogRealHeights[10])
```

```
n02085620-Chihuahua n02085620_1152 500 375 153 144
```

Par exemple, le chihuahua n° 1152 se trouve dans une image 500x375 mais le contour du chien est de taille plus réduite, 153x144.

In [9]:
```
trainBreedsList = {}
for breed in trainDico.keys():
    trainBreedsList[breed] = {}
    trainDogAloneWidths = []
    trainDogAloneHeights = []
    for dog in trainDico[breed]:
        ind = dogs.index(dog.split('.')[0])
        trainDogAloneWidths.append(dogRealWidths[ind])
        trainDogAloneHeights.append(dogRealHeights[ind])
    trainBreedsList[breed]['width'] = trainDogAloneWidths
    trainBreedsList[breed]['height'] = trainDogAloneHeights
```

In [10]:
```
someBreedsDiplay = ['n02085620-Chihuahua', 'n02087394-Rhodesian_ridgeback', 'n02089973-English_foxhound', 'n02092002-Scottish
_deerhound', 'n02106030-collie', 'n02107908-Appenzeller', 'n02096585-Boston_bull',
            'n02092339-Weimaraner', 'n02095570-Lakeland_terrier', 'n02099429-curly-coated_retriever']

fig = plt.figure(1, figsize=(15, 30))
from mpl_toolkits.axes_grid1 import ImageGrid
grid = ImageGrid(fig, 111, nrows_ncols=(3, 4), axes_pad=0.05)
for i, breed in enumerate(someBreedsDiplay):
    ax = grid[i]
    ax.scatter(trainBreedsList[breed]['width'], trainBreedsList[breed]['height'])
    ax.text(100, 500, '%s' % breed[10:], color='k', backgroundcolor='w', alpha=0.8)
    ax.axis('on')
plt.show()
```



On observe que la taille réelle des chiens dans les images est globalement limitée à moins de 500x500.

In [11]:
```
# recherche des plus grandes images
for breed in allBreedsAnnot:
    allDogsAnnotFromBreedDir = os.listdir(annotDir+"/"+breed)
    for dog in allDogsAnnotFromBreedDir:
        breeds.append(breed[10:])
        tree = etree.parse(annotDir+"/"+breed+"/"+dog)
        width = int(tree.xpath("/annotation/size/width")[0].text)
        height = int(tree.xpath("/annotation/size/height")[0].text)
        if width > 2000 and height > 2000:
            print(breed, dog, width, height)
```

```
n02085620-Chihuahua n02085620_4602 3264 2448
n02089973-English_foxhound n02089973_3037 2388 2562
n02093647-Bedlington_terrier n02093647_2743 3264 2448
```

In [12]:
```
showImage(dogImagesDir, dicoBreedIdName, "n02085620_4602", (3264, 2448))
```

In [13]:
```
showImage(dogImagesDir, dicoBreedIdName, "n02085620_4602", (224, 224))
```

In [14]:
```
showImage(dogImagesDir, dicoBreedIdName, "n02085620_4602", (28, 28))
```

"A première vue", il ne semble pas y avoir de perte flagrante dans le fait de passer de la taille normale des images à la taille utilisée par les modèles, par exemple 224x224. Les contours sont moins nets, mais les formes restent quand même assez proches. Mais ce n'est pas le cas avec le passage à 28x28 où l'image est bien trop "pixélisée" pour qu'on puisse reconnaître la race du chien. Cela force donc garder des images au minimum en 224x224, ce qui entraîne des temps de calculs bien plus elevés qu'avec des images en 28x28.

III- Modèles

In [15]:
```
def shuffleTrainIdImageList(_idImageTrainList):
    """ Méthode qui mélange les images d'entraînement afin que les races des images ne se suivent pas
    Arguments:
    _idImageTrainList -- la liste des noms de fichiers des images des chiens pour les images d'entraînement
    Retour:
    idImageTrainListShuffled -- la liste mélangée des noms de fichiers des images des chiens pour les images d'entraînement
    """
    idImageTrainListShuffled = idImageTrainList.copy()
    shuffle(idImageTrainListShuffled)
    return idImageTrainListShuffled

def shuffleTestIdImageList(_idImageTestList):
    """ Méthode qui mélange les images de test afin que les races des images ne se suivent pas
    Arguments:
    _idImageTestList -- la liste des noms de fichiers des images des chiens pour les images de test
    Retour:
    idImageTestListShuffled -- la liste mélangée des noms de fichiers des images des chiens pour les images de test
    """
    idImageTestListShuffled = idImageTestList.copy()
    shuffle(idImageTestListShuffled)
    return idImageTestListShuffled
```

```python
def load_train(_dogImagesDir, _idImageTrainList, _dicoDogBreedIdCateg, _dicoBreedIdName, _size):
    """ Méthode qui charge les données d'entraînement en différents tableaux (images, classes etc.)
    Arguments:
    _dogImagesDir -- chemin vers le répertoire local des images
    _idImageTrainList -- la liste des noms de fichiers des images des chiens pour les images d'entraînement
    _dicoDogBreedIdCateg -- le dictionnaire des positions des répertoires/races des chiens
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    _size -- la taille des images avec laquelle on veut voir les images chargées
    Retour:
    X_train -- le tableau des images d'entraînement
    y_train -- le tableau des classes des images d'entraînement
    """
    X_train = []
    y_train = []
    start_time = time.time()

    print('Read train images')
    for i, dog in enumerate(_idImageTrainList):
        dogBreedDir = getDogBreedDirFromFileName(dog, _dicoBreedIdName)
        imagePath = _dogImagesDir+dogBreedDir+"/"+dog
        img = image.load_img(imagePath, target_size=_size)
        img = image.img_to_array(img)
        imgExpendedPreProcessed= preprocess_input(img)
        X_train.append(imgExpendedPreProcessed)              # variables explicatives les images
        y_train.append(getDogBreedCategFromFileName(_dicoDogBreedIdCateg, dog))  # variable à expliquer, la catégorie, ie la
race du chien

    print('Read train data time: {} seconds'.format(round(time.time() - start_time, 2)))
    return X_train, y_train

def load_test(_dogImagesDir, _idImageTestList, _dicoDogBreedIdCateg, _dicoBreedIdName, _size):
    """ Méthode qui charge les données de test en différents tableaux (images, classes etc.)
    Arguments:
    _dogImagesDir -- chemin vers le répertoire local des images
    _idImageTestList -- la liste des noms de fichiers des images des chiens pour les images de test
    _dicoDogBreedIdCateg -- le dictionnaire des positions des répertoires/races des chiens
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    _size -- la taille des images avec laquelle on veut voir les images chargées
    Retour:
    X_test -- le tableau des images de test
    y_test -- le tableau des classes des images de test
    """
    X_test = []
    y_test = []
    start_time = time.time()

    print('Read test images')
    for i, dog in enumerate(_idImageTestList):
        dogBreedDir = getDogBreedDirFromFileName(dog, _dicoBreedIdName)
        imagePath = _dogImagesDir+dogBreedDir+"/"+dog
        img = image.load_img(imagePath, target_size=_size)
        img = image.img_to_array(img)
        imgExpendedPreProcessed= preprocess_input(img)
        X_test.append(imgExpendedPreProcessed)               # variables explicatives les images
        y_test.append(getDogBreedCategFromFileName(_dicoDogBreedIdCateg, dog))  # variable à expliquer, la catégorie, ie la r
ace du chien

    print('Read test data time: {} seconds'.format(round(time.time() - start_time, 2)))
    return X_test, y_test

def reshape_and_normalize_data(_X_, _y_, _flat, _dim, _preprocessed):
    """ Méthode qui normalise les données et les reformate
    Arguments:
    _X_ -- tableau des images en entrée (entraînement ou test) généralement de format (nb_samples, width, height, depth)
    _y_ -- tableau des classes des images (nb_samples, nb_classes)
    _flat -- détermine si l'on veut "applatir" ou non les dimensions (nécessaire suivant le modèle utilisé)
    _dim -- indique si nombre de canaux est présent (2 = pas de canal couleur pour les images en noir et blanc, 3 pour les im
ages couleur RGB)
    _preprocessed -- indique si les images ont déjà été préprocessées
    Retour:
    X_ -- le tableau des images formatées
    y_ -- le tableau des cibles/classes formatées
    num_classes -- le nombre de classes
    """
    start_time = time.time()

    print('Convert to numpy...')
    X_ = np.array(_X_, dtype=np.uint8)
    y_ = np.array(_y_, dtype=np.uint8)

    print("X_.shape", X_.shape)
    print("y_.shape", y_.shape)

    num_classes = np.unique(y_).shape[0]
    print("num_classes = ", num_classes)

    if _flat: # si _flat == True, on veut, comme pour le perceptron, un input "plat" (on multiplie les "dimensions" entre ell
es)
        if _dim == 2:# si _dim == 2, il n'y a qu'un seul canal comme avec le MNIST qui utilise les nuances de noir/gris/blanc
            nb_data_sample, picture_width, picture_height = X_.shape
            X_ = X_.reshape(nb_data_sample, picture_width*picture_height) # 60000, 784 = 28*28
        elif _dim == 3: # si _dim = 3, on utilise une 3 dimensions dont la 3ème est celle des 3 canaux RGB
            nb_data_sample, picture_width, picture_height, nb_channels = X_.shape
            X_ = X_.reshape(nb_data_sample, picture_width*picture_height*nb_channels) # 500, 150528 = 224*224*3
    # si _flat = False, on ne redimensionne pas
```

```
    if not _preprocessed:
        print('Convert to float...')
        X_ = X_.astype('float32')
        X_ = X_ / 255
    # convert class vectors to multi-categorical class matrices
    # when using the categorical_crossentropy loss, your targets should be in categorical format (e.g. if you have 10 classe
s,
    # the target for each sample should be a 10-dimensional vector that is all-zeros expect for a 1 at the index correspondin
g
    # to the class of the sample). In order to convert integer targets into categorical targets,
    # you can use the Keras utility to_categorical.
    y_ = keras.utils.to_categorical(y_, num_classes)

    print('Data shape:', X_.shape)
    print(X_.shape[0], 'data samples')
    print('Target data shape:', y_.shape)
    print(y_.shape[0], 'target samples')
    print('Read and process data time: {} seconds'.format(round(time.time() - start_time, 2)))
    return X_, y_, num_classes


def drawModelAccuracyAndLoss(_history, _size):
    """ Méthode qui trace l'évolution de l'accuracy et du loss
    Arguments:
    _history -- l'historique de l'accuracy et du loss
    _size -- la taille des graphes
    """
    plt.figure(1, figsize=_size)

    # summarize history for accuracy

    plt.subplot(121)
    plt.plot(_history.history['acc'])
    plt.plot(_history.history['val_acc'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'validation'], loc='lower right')

    # summarize history for loss

    plt.subplot(122)
    plt.plot(_history.history['loss'])
    plt.plot(_history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'validation'], loc='upper left')

    plt.show()
def fitAndEvaluate(_model, _train_data, _train_target, _test_data, _test_target, _batch_size, _epochs, _verbose):
    """ Méthode qui entraîne et évalue un modèle
    Arguments:
    _model -- le modèle à entraîner et à évaluer
    _train_data -- les données d'entraînement (images)
    _train_target -- les cibles d'entraînement (classes)
    _test_data -- les données de test
    _test_target -- les cibles de test
    _batch_size -- la taille du batch (en nombre de neurones) pour l'entraînement
    _epochs -- le nombre d'itération
    _verbose -- 1 pour afficher l'évolution de l'entraînement, 0 sinon
    Retour:
    _model -- le modèle entraîné
    history -- historique (pour suivre l'évolution de l'accuracy et du loss)
    score -- les scores d'accuracy et de loss
    """
    # entraînement du modèle, validation avec les données de test
    history = _model.fit(_train_data, _train_target, batch_size=_batch_size, epochs=_epochs, verbose=_verbose, validation_dat
a=(_test_data, _test_target), shuffle=True)
    score = _model.evaluate(_test_data, _test_target, verbose=_verbose)
    print('\nTest loss:', score[0])
    print('Test accuracy:', score[1])
    return _model, history, score
```

Réseaux de neurones réguliers, perceptrons multicouches.

Les réseaux de neurones réguliers dont font partie les perceptrons, suivent globalement le schéma suivant: le réseau

- reçoit en entrée un unique vecteur ("aplatissement des dimensions")
- transforme ce vecteur à travers un certain nombre de couches cachées (hidden layers)
- chaque couche cachée est faite d'un ensemble de neurones où chaque neurone est complètement connecté à tous les neurones de la couche précédente (couche Dense ou FC/Fully-Connected)
- chaque neurone dans une couche est indépendant des autres neurones de sa couche
- la dernière couche complètement connectée est appelée « couche de sortie » (output layer) et représente les scores des catégories/classes

1- Perceptron multicouche avec les données du MNIST en 28x28 pour comparaison

Pour fin de comparaison, voici un exemple de perceptron multicouche qui donne de très bons résultats sur les données MNIST (Mixed National Institute of Standards and Technology), alors même que le modèle ne comporte que 3 couches. Les images MNIST sont toutefois assez légères car ce sont des images de 28 pixels de côté en noir et blanc, donc avec seulement 1 seul canal (contre 3 pour les couleurs RGB). Nous verrons que le même algorithme utilisé pour les images d'ImageNet de tailles bien supérieures ne répond pas aussi bien.

- Un perceptron monocouche ne pouvant apprendre que des relations linéaires, il est nécessaire pour les images d'utiliser des perceptrons multicouches utilisant l'algorithme de rétropropagation du gradient de l'erreur afin de déterminer les meilleurs poids permettant de classer les images, puisqu'il s'agit ici d'un problème de classification.

In [17]:
```
def initialiseMultiLayerPerpectronMNIST(_input_shape, _num_classes):
    """ Méthode qui initialise un modèle de type perceptron multicouche (destiné ici par exemple à classer les images du MNIS
T)
    Arguments:
    _input_shape -- la dimension de l'image en entrée
    _num_classes -- le nombre de classes possibles dans lesquelles les images en entrée doivent être classées
    Retour:
    model -- le modèle du Perceptron
    """
    model = Sequential()
    # input couche d'entrée correspondant à l'image, de dimension 28x28x1 = 784 pour les images du MNIST
    # première couche Dense/Fully-Connected à 256 neurones avec une fonction d'activation ReLU
    model.add(Dense(256, kernel_initializer=RandomNormal(mean=0.0, stddev=0.05), bias_initializer=RandomNormal(mean=0.0, stdd
ev=0.05), activation='relu', input_shape=(_input_shape,)))
    # désactivation de 20% des neurones de la couche précédente afin d'éviter l'overfitting (cf. [1])
    model.add(Dropout(0.2))
    model.add(Dense(256, kernel_initializer=RandomNormal(mean=0.0, stddev=0.05), bias_initializer=RandomNormal(mean=0.0, stdd
ev=0.05), activation='relu'))
    # désactivation de 50% des neurones
    model.add(Dropout(0.5))
    # 3ème et dernière couche, une FC/Dense avec 10 classes pour les images du MNIST et la fonction d'activation softmax
    # afin d'obtenir une hiérarchie des probabilités de chaque classe
    model.add(Dense(_num_classes, kernel_initializer=RandomNormal(mean=0.0, stddev=0.05), bias_initializer=RandomNormal(mean=
0.0, stddev=0.05), activation='softmax'))
    # compilation du modèle avec un Stochastic Gradient Descent au learning rate à 0.1
    sgd = SGD(lr=0.01, decay=0.0005, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

    return model
```

Remarques sur le modèle:

- Pour instancier ce perceptron à 3 couches, j'utilise la librairie Keras, basée elle-même sur TensorFlow.
- La première couche, l'input (non comptabilisée dans le décompte des couches d'un réseau de neurone) ne supporte aucun poid, elle ne contient que les données en input, c'est-à-dire chaque pixel de l'image d'entrée. Pour les images du MNIST qui sont de taille 28x28 en noir et blanc, on a donc comme dimension 28*28*1 = 784.
- Puisque ces images représentent des chiffres de 0 à 9, il s'agit donc de classer les images en 10 classes différentes.
- La "vraie" première couche est une couche FC (Fully-Connected), ou Dense, avec 256 neurones et comme méthode d'activation ReLU utilisant la fonction f(x) = max(0, x). D'après [2], il est préférable d'utiliser ReLu plutôt que tanh ou sigmoid car elle accélère grandement la convergence de la descente de gradient du fait de sa forme linéaire et non-saturable. Il est possible d'utiliser d'autres méthode comme leaky-ReLU, PReLU ou Maxout qui n'ont pas les inconvénients de ReLU ("mort des neurones", évitable en ne prenant pas un learning rate trop élevé), c'est à tester.
- A la sortie de cette couche se trouve un dropout qui est une technique de régularisation qui permet d'éviter le surapprentissage (overfitting) dû à un trop grand nombre de paramètres (relativement à un contexte précis). D'après [3], durant l'entraînement, le dropout peut être interprété comme prenant un échantillon du réseau à l'intérieur de tout le réseau et en mettant à jour uniquement les paramètres du réseau échantilloné en se basant sur ses inputs. Dans Keras, cet hyperparamètre est décrit par une fraction (entre 0 et 1) des neurones de la couche d'entrée à rendre inactif. Dans notre exemple, suivant les conseils de [1], pour la première couche intégrant les pixels de l'input je choisis d'inactiver 20% des neurones, puis 50% pour la couche suivante.
- La dernière couche possède exactement le même nombre de neurones qu'il y a de classes recherchée, dans le cas de MNIST, c'est 10. Dans le cas d'ImageNet, cela dépendra du nombre de races de chien que l'on veut utiliser, 120 pour la totalité du dataset, ou moins si on veut diminuer les temps de calcul.
- La fonction d'activation de la dernière couche est le classifieur Softmax qui est la généralisation à plusieurs catégories du classifieur de régression logistique binaire. On pourrait utiliser SVM mais Softmax fournit un score un peu plus intuitif (grâce à des probabilités normalisées) et une interprétation probabiliste (même si ce sont plutôt des probabilités relatives entre elles). Softmax permet donc d'obtenir une hiérachie de scores compris entre 0 et 1 et donc la somme vaut 1. Il suffit donc de prendre comme prévision de la classe d'une image, la classe dont le score est le plus élevé.
- J'ai ajouté kernel_initializer=RandomNormal(mean=0.0, stddev=0.05) et bias_initializer=RandomNormal(mean=0.0, stddev=0.05) car (cf. [3]) il ne faut pas initialiser tous les paramètres (poids, biais) à 0, car sans différence, tous les neurones vont calculer les mêmes résultats, puis les mêmes gradients en sens inverse etc. L'idée est de rendre tous les neurones aléatoires et différents mais petits au début afin qu'ils calculent des mises-à-jour différentes et s'intègrent eux-mêmes dans des parties distinctes de l'ensemble du réseau.
- J'utilise pour minimiser la fonction de coût (loss) la descente de gradient stochastique SGD qui permet d'augmenter la vitesse de l'entraînement en estimant le gradient de la fonction pour tous les inputs de l'entraînement par le gradient de la fonction pour un petit échantillon (minibatch) d'input d'entraînement choisis au hasard.
- La taux d'apprentissage est un des hyperparamètres du modèle puisqu'en fonction de sa valeur, on peut énormément améliorer la vitesse d'apprentissage, mais aussi la diminuer fortement (par exemple en prenant un taux beaucoup trop élévé).
- Suivant [4], j'ai mis le moment de la SGD à 0.9. (Permet de garder en mémoire la mise à jour à chaque étape des paramètres et réintroduit un certain nombre de ces valeurs dans le calcul de la dernière mise à jour. Cela empêche entre autres les oscillations.)
- Nesterov est une l'auteur d'une méthode qui permet d'accélérer la descente de gradient.
- J'utilise la fonction de coût categorical_crossentropy qui permet de relativement éviter le ralentissement de l'apprentissage (par exemple vis-à-vis de la fonction de coût MeanSquarredError) [5] (categorical afin d'utiliser les catégories en output). En effet, alors que les dérivées partielles de la fonction de coût par rapport aux poids et aux biais dépendent de la dérivée de la fonction d'activation pour MSE, ce n'est pas le cas pour Cross Entropy. Ainsi, le problème d'applatissement (dérivée tendant vers 0) abaissant les dérivées partielles du coût n'a pas lieu avec la Cross entropy. C'est d'ailleurs pour cela qu'elle a été choisie.

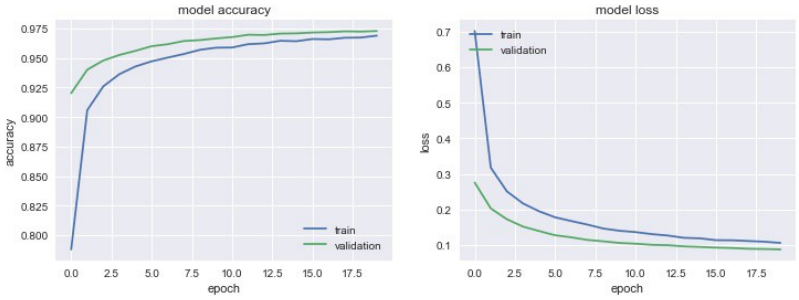
In [18]:
```
# the data, shuffled and split between train and test sets
(X_train_MNIST, y_train_MNIST), (X_test_MNIST, y_test_MNIST) = mnist.load_data()

# normalisation des images afin de pouvoir être utilisées par les réseaux
# il est par exemple nécessaire de diviser les données des images par 255 afin de les normaliser car certains algorithmes
# n'apprécient pas trop les grands nombres
# pour MNIST, _flat = True et _dim = 2 car on doit applatir les dimensions en un format plat du genre 28x28 = 784 au lieu
# de rester en format(28, 28, 1) comme ce sera le cas pour ImageNet en (224, 224, 3)
train_data_MNIST, train_target_MNIST, num_classes_MNIST = reshape_and_normalize_data(X_train_MNIST, y_train_MNIST, True, 2, F
alse)
test_data_MNIST, test_target_MNIST, num_classes_MNIST = reshape_and_normalize_data(X_test_MNIST, y_test_MNIST, True, 2, Fals
e)

# initialisation du modèle du perceptron multicouche avec les images du MNIST
model_MNIST = initialiseMultiLayerPerpectronMNIST(train_data_MNIST.shape[1], num_classes_MNIST) # 784 = 28*28
model_MNIST.summary()

# entraînement des données et évaluation du modèle avec des batches de 128 neurones et 20 epochs
# (5 suffisent pour obtenir déjà un très bon résultat, mais 20 pour voir l'évolution de l'accuracy et du loss)
model_MNIST, historyMNIST, scoreMNIST = fitAndEvaluate(model_MNIST, train_data_MNIST, train_target_MNIST, test_data_MNIST, te
st_target_MNIST, 128, 20, 1)
```

```
Convert to numpy...
X_.shape (60000, 28, 28)
y_.shape (60000,)
num_classes =  10
Convert to float...
Data shape: (60000, 784)
60000 data samples
Target data shape: (60000, 10)
60000 target samples
Read and process data time: 0.16 seconds
Convert to numpy...
X_.shape (10000, 28, 28)
y_.shape (10000,)
num_classes =  10
Convert to float...
Data shape: (10000, 784)
10000 data samples
Target data shape: (10000, 10)
10000 target samples
Read and process data time: 0.03 seconds
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               200960
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               65792
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                2570
=================================================================
Total params: 269,322
Trainable params: 269,322
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s - loss: 0.7018 - acc: 0.7879 - val_loss: 0.2755 - val_acc: 0.9203
Epoch 2/20
60000/60000 [==============================] - 2s - loss: 0.3174 - acc: 0.9059 - val_loss: 0.2025 - val_acc: 0.9402
Epoch 3/20
60000/60000 [==============================] - 2s - loss: 0.2506 - acc: 0.9261 - val_loss: 0.1722 - val_acc: 0.9480
Epoch 4/20
60000/60000 [==============================] - 2s - loss: 0.2167 - acc: 0.9365 - val_loss: 0.1515 - val_acc: 0.9527
Epoch 5/20
60000/60000 [==============================] - 2s - loss: 0.1946 - acc: 0.9430 - val_loss: 0.1390 - val_acc: 0.9562
Epoch 6/20
60000/60000 [==============================] - 2s - loss: 0.1776 - acc: 0.9473 - val_loss: 0.1271 - val_acc: 0.9601
Epoch 7/20
60000/60000 [==============================] - 2s - loss: 0.1671 - acc: 0.9504 - val_loss: 0.1213 - val_acc: 0.9618
Epoch 8/20
60000/60000 [==============================] - 2s - loss: 0.1572 - acc: 0.9535 - val_loss: 0.1140 - val_acc: 0.9645
Epoch 9/20
60000/60000 [==============================] - 2s - loss: 0.1459 - acc: 0.9570 - val_loss: 0.1100 - val_acc: 0.9653
Epoch 10/20
60000/60000 [==============================] - 2s - loss: 0.1397 - acc: 0.9588 - val_loss: 0.1054 - val_acc: 0.9667
Epoch 11/20
60000/60000 [==============================] - 2s - loss: 0.1360 - acc: 0.9590 - val_loss: 0.1034 - val_acc: 0.9678
Epoch 12/20
60000/60000 [==============================] - 2s - loss: 0.1303 - acc: 0.9618 - val_loss: 0.1002 - val_acc: 0.9698
Epoch 13/20
60000/60000 [==============================] - 2s - loss: 0.1264 - acc: 0.9625 - val_loss: 0.0991 - val_acc: 0.9696
Epoch 14/20
60000/60000 [==============================] - 2s - loss: 0.1199 - acc: 0.9647 - val_loss: 0.0959 - val_acc: 0.9708
Epoch 15/20
60000/60000 [==============================] - 2s - loss: 0.1184 - acc: 0.9643 - val_loss: 0.0943 - val_acc: 0.9710
Epoch 16/20
60000/60000 [==============================] - 2s - loss: 0.1133 - acc: 0.9662 - val_loss: 0.0923 - val_acc: 0.9717
Epoch 17/20
60000/60000 [==============================] - 2s - loss: 0.1130 - acc: 0.9659 - val_loss: 0.0912 - val_acc: 0.9720
Epoch 18/20
60000/60000 [==============================] - 2s - loss: 0.1109 - acc: 0.9672 - val_loss: 0.0890 - val_acc: 0.9727
```

```
Epoch 19/20
60000/60000 [==============================] - 2s - loss: 0.1090 - acc: 0.9674 - val_loss: 0.0883 - val_acc: 0.9725
Epoch 20/20
60000/60000 [==============================] - 2s - loss: 0.1056 - acc: 0.9690 - val_loss: 0.0873 - val_acc: 0.9730
 9760/10000 [============================>.] - ETA: 0s
Test loss: 0.0873307392024
Test accuracy: 0.973
```

- On voit donc dans le résumé du modèle la succession des couches et des techniques avec leurs nombres respectifs de paramètres et leur format.
- Par exemple, le dropout, s'il désactive des neurones, ne les supprime pas puisqu'ils seront réutilisés à l'itération suivante. La "shape" reste donc identique.

In [19]:
```
drawModelAccuracyAndLoss(historyMNIST, (12, 4))
```



- On voit qu'avec ce modèle et ces données, on atteint assez vite des bornes pour l'accuracy et le loss des données de test.
- Il s'est passé quelque chose de bizarre, car lors de mon premier test, les courbes étaient "inversées", c'est-à-dire que (si l'on suit [6]) on avait un léger overfitting pour ce modèle car la courbe d'accuracy des données de test était légèrement inférieure à celle de l'entraînement qui, elle, frôlait 1 vers la fin. Or on relançant le programme, il y a une inversion, les courbes de loss et d'accuracy sont meilleures que celle de l'entraînement. Peut-être y a-t-il eu une mise en mémoire des paramètres qui ont été entraînés précédemment, et la nouvelle exécution du programme en tient compte...
- D'après la courbe de loss, on a un bon learning rate.
- Les courbes s'applatissant nettement au bout de 10 itérations, on peut envisager d'arrêter l'entraînement à ce moment-là. (Compromis temps d'exécution / exactitude des prévisions.)

2- Perceptron multicouche avec les données d'ImageNet

Je vais donc maintenant récupérer les images d'ImageNet et lancer ce même modèle dessus.

In [20]:
```
breedsChoice = someBreeds

# liste des races pour chaque image d'entraînement suivant un format linéaire
# idBreedDirTrainList : ['n02085620-Chihuahua', 'n02085620-Chihuahua', ...
# et liste des noms des fichiers image d'entraînement suivant un format linéaire
# idImageTrainList : ['n02085620_5927.jpg', 'n02085620_4441.jpg', ...
idBreedDirTrainList, idImageTrainList = getTrainBreedDirAndImageNameList(trainList, breedsChoice)

# liste des noms des fichiers image de test suivant un format linéaire
# idImageTestList : ['n02085620_2650.jpg', 'n02085620_4919.jpg', ...
idImageTestList = getTestBreedImageNameList(testList, breedsChoice)

# dictionnaire des positions des répertoires/races des chiens
# dicoDogBreedIdNameCateg --> {'n02085620-Chihuahua': 0, 'n02085782-Japanese_spaniel': 1,
# dictionnaire des positions des identifiants des races des chiens
# dicoDogBreedIdCateg) --> {'n02085620': 0, 'n02085782': 1,
dicoDogBreedIdNameCateg, dicoDogBreedIdCateg = createDicoDogBreedIdNameCateg(breedsChoice)

# shuffle des images d'entraînement et de test afin de ne pas avoir les images d'une même race à se suivre et "dégrouper" ainsi les chiens
idImageTrainListShuffled = shuffleTrainIdImageList(idImageTrainList)
idImageTestListShuffled = shuffleTestIdImageList(idImageTestList)
```

Chargement des images et séparation des datasets d'entraînement et de test.

In [21]:
```
X_train, y_train = load_train(dogImagesDir, idImageTrainListShuffled, dicoDogBreedIdCateg, dicoBreedIdName, (224, 224))
X_test, y_test = load_test(dogImagesDir, idImageTestListShuffled, dicoDogBreedIdCateg, dicoBreedIdName, (224, 224))
```

```
Read train images
Read train data time: 3.81 seconds
Read test images
Read test data time: 2.91 seconds
```

In [22]:
```
# Formatage des données pour obtenir un input plat pour les perceptrons
# on est toujours en format plat (multiplication des dimensions, _flat=True), mais cette fois-ci on utilise le nombre de canaux
train_data, train_target, num_classes = reshape_and_normalize_data(X_train, y_train, True, 3, True)
test_data, test_target, num_classes = reshape_and_normalize_data(X_test, y_test, True, 3, True)
```

```
Convert to numpy...
X_.shape (1000, 224, 224, 3)
y_.shape (1000,)
num_classes =  10
Data shape: (1000, 150528)
1000 data samples
Target data shape: (1000, 10)
1000 target samples
Read and process data time: 0.3 seconds
```

```
Convert to numpy...
X_.shape (707, 224, 224, 3)
y_.shape (707,)
num_classes =  10
Data shape: (707, 150528)
707 data samples
Target data shape: (707, 10)
707 target samples
Read and process data time: 0.18 seconds
```

Cette fois-ci, la dimension à l'entrée est de 224x224x3 = 150528 et non plus 28x28x1 = 784 comme pour le MNIST, ce qui fait beaucoup de paramètres en plus, comme on peut le voir ci-dessous dans le nombre total de paramètres du modèle:

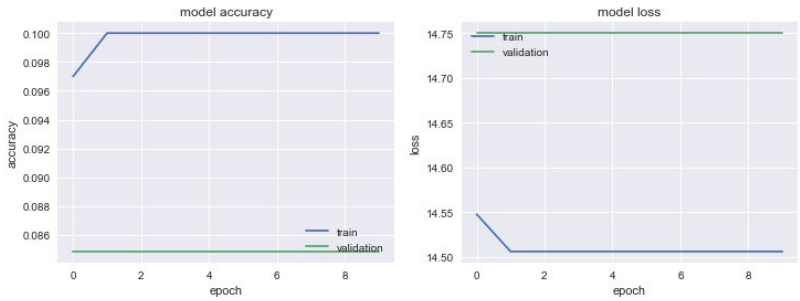
Modèle du MNIST avec les images d'ImageNet

In [23]:
```
modelMNISTForImageNet = initialiseMultiLayerPerpectronMNIST(train_data.shape[1], num_classes)
modelMNISTForImageNet.summary()
modelMNISTForImageNet, historyMNISTForImageNet, scoreMNISTForImageNet = fitAndEvaluate(modelMNISTForImageNet, train_data, tra
in_target, test_data, test_target, 64, 10, 1)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 256)               38535424
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               65792
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                2570
=================================================================
Total params: 38,603,786
Trainable params: 38,603,786
Non-trainable params: 0
_________________________________________________________________
Train on 1000 samples, validate on 707 samples
Epoch 1/10
1000/1000 [==============================] - 7s - loss: 14.5480 - acc: 0.0970 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 2/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 3/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 4/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 5/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 6/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 7/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 8/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 9/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 10/10
1000/1000 [==============================] - 6s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.7502 - val_acc: 0.0849
707/707 [==============================] - 1s

Test loss: 14.7502228711
Test accuracy: 0.0848656294201
```

- Alors qu'on a utilisé le même modèle, on voit la différence de résultat, 0.973 / 0.087 pour MNIST et 0.084 / 14.75 pour ImageNet. Le modèle n'est clairement pas adapté pour les images d'ImageNet. En fait, les réseaux de neurones réguliers (RNN), s'ils conviennent bien pour les petites images (ici 28x28x1), ne sont pas adaptés pour les images plus lourdes [7]. Par exemple, si pour un neurone de la première couche d'une petite image du MNIST on a 28x28x1 = 784 poids, pour celles d'ImageNet on aura 224x224x3 = 150 528 poids, difficile à gérer.
- J'ai testé avec la totalité des 120 races, et le résultat est encore pire. (Les temps de calcul trop importants m'ont toutefois empêché d'aller au-delà de 5 itérations.)

In [78]:
```
drawModelAccuracyAndLoss(historyMNISTForImageNet, (12, 4))
```



Ces "courbes" témoignent d'un problème qui sera discuté en conclusion.

Si on veut vraiment utiliser le même modèle, on peut restreindre la taille des images d'ImageNet à 28x28 comme pour MNIST.

In [25]:
```
X_train2, y_train2 = load_train(dogImagesDir, idImageTrainListShuffled, dicoDogBreedIdCateg, dicoBreedIdName, (28, 28))
X_test2, y_test2 = load_test(dogImagesDir, idImageTestListShuffled, dicoDogBreedIdCateg, dicoBreedIdName, (28, 28))
train_data2, train_target2, num_classes2 = reshape_and_normalize_data(X_train2, y_train2, True, 3, True)
test_data2, test_target2, num_classes2 = reshape_and_normalize_data(X_test2, y_test2, True, 3, True)
modelMNISTForImageNet2 = initialiseMultiLayerPerpectronMNIST(train_data2.shape[1], num_classes2)
modelMNISTForImageNet2.summary()
modelMNISTForImageNet2, historyMNISTForImageNet2, scoreMNISTForImageNet2 = fitAndEvaluate(modelMNISTForImageNet2, train_data
2, train_target2, test_data2, test_target2, 64, 20, 1)
```

```
Read train images
Read train data time: 2.75 seconds
Read test images
Read test data time: 2.27 seconds
Convert to numpy...
X_.shape (1000, 28, 28, 3)
y_.shape (1000,)
num_classes =  10
Data shape: (1000, 2352)
1000 data samples
Target data shape: (1000, 10)
1000 target samples
Read and process data time: 0.01 seconds
Convert to numpy...
X_.shape (707, 28, 28, 3)
y_.shape (707,)
num_classes =  10
Data shape: (707, 2352)
707 data samples
Target data shape: (707, 10)
707 target samples
Read and process data time: 0.01 seconds
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 256)               602368
_________________________________________________________________
dropout_5 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_8 (Dense)              (None, 256)               65792
_________________________________________________________________
dropout_6 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                2570
=================================================================
Total params: 670,730
Trainable params: 670,730
Non-trainable params: 0
_________________________________________________________________
Train on 1000 samples, validate on 707 samples
Epoch 1/20
1000/1000 [==============================] - 0s - loss: 14.7159 - acc: 0.0870 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 2/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 3/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 4/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 5/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 6/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 7/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 8/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 9/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 10/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 11/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 12/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 13/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 14/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 15/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 16/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 17/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 18/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 19/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
Epoch 20/20
1000/1000 [==============================] - 0s - loss: 14.5063 - acc: 0.1000 - val_loss: 14.2487 - val_acc: 0.1160
 32/707 [>.............................] - ETA: 0s
Test loss: 14.2486697552
Test accuracy: 0.115983026874
```

On observe une amélioration notable, toutefois le modèle reste inadapté.On peut alors essayer d'augmenter le nombre de couches pour mieux prendre en compte le grand nombre des pixels.

3- Perceptron multicouche avec différents nombres de couches

In [26]:
```
def initialiseMultiLayerPerpectron(_input_shape, _num_classes, _nb_layer):
    """ Méthode qui initialise un perceptron multicouche
    Arguments:
    _input_shape -- la forme de l'input
    _num_classes -- le nombre de classes
    _nb_layer -- le nombre de couches Dense + Dropout à ajouter aux couches d'input et d'output
    Retour:
    model -- le modèle généré
    """
    model = Sequential()

    model.add(Dense(512, activation='relu', input_shape=(_input_shape,)))
    model.add(Dropout(0.2))

    for i in range(_nb_layer):
        model.add(Dense(512, activation='relu'))
        model.add(Dropout(0.5))

    model.add(Dense(_num_classes, activation='softmax'))

    sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

    return model
```

In [27]:
```
lossList = []
accuracyList = []
for n in [1, 4, 7, 10, 15, 20, 30]:
    print("N = ", n)
    modelNL = initialiseMultiLayerPerpectron(train_data.shape[1], num_classes, n) # 150528 = 224 * 224 * 3
    modelNL, historyNL, scoreNL = fitAndEvaluate(modelNL, train_data, train_target, test_data, test_target, 64, 10, 0)
    lossList.append(scoreNL[0])
    accuracyList.append(scoreNL[1])
```

```
N =  1

Test loss: 14.9554037717
Test accuracy: 0.0721357850071
N =  4

Test loss: 14.9326057164
Test accuracy: 0.0735502121641
N =  7

Test loss: 14.7502228711
Test accuracy: 0.0848656294201
N =  10

Test loss: 13.1087763778
Test accuracy: 0.186704384808
N =  15

Test loss: 14.9098079633
Test accuracy: 0.0749646393211
N =  20

Test loss: nan
Test accuracy: 0.0735502121641
N =  30

Test loss: nan
Test accuracy: 0.0735502121641
```

Comme nous le verrons plus loin avec l'introduction des Resnet, ce n'est pas parce qu'on augmente le nombre de couches qu'on obtient de meilleurs résultats. Pour ces tests, le meilleur résultat correspond à 3 + 2x10 = 23 couches (N = 10).

4- Réseau de neurone convolutif

Nous avons donc vu que les réseaux réguliers ne pouvaient guère prendre en charge les images d'ImageNet. On va donc utiliser des réseaux plus adaptés, les ConvNets / CNN ou réseaux de neurones convolutifs. Voici quelques renseignements importants sur ces réseaux:

- Les ConvNets/CNN ont l'avantage sur les RNN d'avoir affaire à des images pour contraindre l'architecture. Si pour les RNN on avait une seule dimension en entrée (par exemple pour les images MNIST, 1 dimension en "applatissant" le format des images (28, 28, 1) --> 784, pour les CNN, les couches ont des neurones arrangés en 3 dimension (height pour la hauteur, width pour la largeur et depth pour la profondeur correspondant aux 3 channels/canaux des couleurs RGB). Pour les images d'ImageNet, on a donc comme format d'entrée (height, width, depth), soit (224, 224, 3).
- Dans les ConvNets, contrairement aux Dense/FC des RNN, les neurones ne sont pas connectés à tous les neurones de la couche précédente, mais seulement à une petite région de la couche précédente.
- Chaque couche d'un ConvNet transforme le volume 3D en entrée en un volume 3D de sortie d'activations de neurone.
- Comme nous le verrons avec le modèle ci-dessous, il y a 3 types principaux de couches, les convolutional layer (Convolution2D sous Keras), les pooling layer (MaxPooling2D avec max comme fonction du pooling) et les fully-connected layer (Dense/FC identiques aux RNN).
- Les couches CONV et FC effectuent des transformations qui sont des fonctions des activations dans le volume d'entrée mais aussi des paramètres (poids et biais des neurones), alors que les "couches" ReLU et POOL implémentent une fonction fixe.
- Les paramètres de CONV et FC sont entraînés avec la descente de gradient afin que les scores des catégories que le ConvNet calcule soient consistent avec les libellés des catégories du dataset d'entraînement pour chaque image.
- Paramètres: les couches CONV et FC ont des paramètres alors que ReLU et POOL n'en ont pas.
- Hyperparamètres: les couches CONV, FC et POOL ont des hyperparamètres, mais pas la fonction d'activation ReLU.
- Si pendant un temps la méthode Dropout était "réservée", destinées à suivre une couche Dense/FC, il s'est avéré qu'elles pouvaient également être bénéfiques après une couche CONV.

In [28]:
```
def convolutionModel(_train_data, _dicoModelParam, _num_classes):
    """ Méthode qui initialise un modèle de type convolutif (destiné ici par exemple à classer les images d'ImageNet)
    Arguments:
    _train_data -- les données d'entraînement
    _dicoModelParam -- le dictionnaire des paramètres utiles au modèle
    _num_classes -- le nombre de classes possibles dans lesquelles les images en entrée doivent être classées
    Retour:
    model -- le modèle du Perceptron
    """
    _, height, width, depth = _train_data.shape # _ pour le nombre d'enregistrements dans le dataset
    inp = Input(shape=(height, width, depth)) # depth en dernière position pour TensorFlow, avec Theano, elle devrait être mi
se en premier

    # deux séries Conv -> Conv -> Pool + dropout
    conv_1 = Convolution2D(_dicoModelParam['conv_depth_1'], (_dicoModelParam['kernel_size'], _dicoModelParam['kernel_size']),
padding='same', activation='relu')(inp)
    conv_2 = Convolution2D(_dicoModelParam['conv_depth_1'], (_dicoModelParam['kernel_size'], _dicoModelParam['kernel_size']),
padding='same', activation='relu')(conv_1)
    pool_1 = MaxPooling2D(pool_size=(_dicoModelParam['pool_size'], _dicoModelParam['pool_size']))(conv_2)
    drop_1 = Dropout(_dicoModelParam['drop_prob_1'])(pool_1)

    conv_3 = Convolution2D(_dicoModelParam['conv_depth_2'], (_dicoModelParam['kernel_size'], _dicoModelParam['kernel_size']),
padding='same', activation='relu')(drop_1)
    conv_4 = Convolution2D(_dicoModelParam['conv_depth_2'], (_dicoModelParam['kernel_size'], _dicoModelParam['kernel_size']),
padding='same', activation='relu')(conv_3)
    pool_2 = MaxPooling2D(pool_size=(_dicoModelParam['pool_size'], _dicoModelParam['pool_size']))(conv_4)
    drop_2 = Dropout(_dicoModelParam['drop_prob_1'])(pool_2)

    # applatissement en 1D avant la/les FC -> Dense + ReLU (et dropout) -> Dense + softmax
    flat = Flatten()(drop_2)
    hidden = Dense(_dicoModelParam['hidden_size'], activation='relu')(flat)
    drop_3 = Dropout(_dicoModelParam['drop_prob_2'])(hidden)
    out = Dense(_num_classes, activation='softmax')(drop_3)
    model = Model(inputs=inp, outputs=out)

    sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', # cross-entropy loss function
                  optimizer=sgd, # pour garder celle des perceptrons afin de comparaison
                  metrics=['accuracy']) # pour garder celle des perceptrons afin de comparaison

    return model
```

Description :

- Input: fonction qui intègre l'image avec le format adéquat shape=(height, width, depth)
- Convolution2D: couche qui calcule les output des neurones qui sont connectés à des régions locales de l'image d'entrée, chaque neurone calculant un produit entre leurs poids et la petite région à laquelle ils sont connectés dans l'image. Cela donne un volume par exemple de dimension [224x224x12] s'il y a 12 filtres. Dans notre exemple, il y a _dicoModelParam['conv_depth_1'] filtres (par ex. 32 pour les deux premières, 64 pour les deux suivantes).
- Toujours pour la couche de convolution, kernel_size est l'hyperparamètre correspondant au "receptive field" (c'est la taille du filtre), par exemple 3, c'est-à-dire que chaque "fibre" du réseau (ou depth column) observera une zone spatiale (hauteurxlargeur) de 3x3 du volume d'entrée.
- ReLU: fonction d'activation max(0, x) appliquée à ce produit. (La dimension reste inchangée.)
- MaxPooling2D: couche qui effectue un échantillonnage le long des dimensions spatiales (width / height) pouvant donner par exemple une dimension [112x112x12], et qui applique la fonction Max
- Flatten: applatit le volume 3D en un vecteur de dimension 1
- Dense (FC) / softmax: couche qui calcule le score des catégories, avec comme dimension finale le "volume" [1x1x10]. Pour cette couche, tous les neurones sont connectés à tous les neurones de la couche précédente.

Les hyperparamètres du modèle (instantiation + entraînement) : (cf. [8])

- batch size (entraînement): représente le nombre d'images d'entraînement utilisées simultanément pendant une itération de l'algorithme de descente de gradient
- epochs (entraînement): représente le nombre de fois que l'algorithme d'entraînement va itérer sur la totalité des données d'entraînement avant de se terminer
- kernel size (modèle): dans la couche convolutive, correspond à la taille du filtre (receptive field)
- pooling size (modèle): dans la couche de pooling, correspond à la taille du filtre (souvent 2x2, parfois 3x3)
- nombre de kernels (modèle): dans la couche convolutive, correspond à la profondeur (depth) de la couche, c'est-à-dire le nombre de neurones qui regardent vers la même zone du volume d'entrée
- probabilité de dropout (modèle): probabilité qu'à un neurone d'être rendu inactif le temps d'une itération
- nombre de neurones (modèle): dans la couche Dense/FC son nombre de neurones

In [29]:
```
X_train, y_train = load_train(dogImagesDir, idImageTrainListShuffled, dicoDogBreedIdCateg, dicoBreedIdName, (224, 224))
X_test, y_test = load_test(dogImagesDir, idImageTestListShuffled, dicoDogBreedIdCateg, dicoBreedIdName, (224, 224))
train_data, train_target, num_classes = reshape_and_normalize_data(X_train, y_train, False, 0, True)
test_data, test_target, num_classes = reshape_and_normalize_data(X_test, y_test, False, 0, True)
```

```
Read train images
Read train data time: 3.15 seconds
Read test images
Read test data time: 2.53 seconds
Convert to numpy...
X_.shape (1000, 224, 224, 3)
y_.shape (1000,)
num_classes =  10
Data shape: (1000, 224, 224, 3)
1000 data samples
Target data shape: (1000, 10)
1000 target samples
Read and process data time: 0.31 seconds
Convert to numpy...
X_.shape (707, 224, 224, 3)
y_.shape (707,)
num_classes =  10
Data shape: (707, 224, 224, 3)
707 data samples
Target data shape: (707, 10)
707 target samples
Read and process data time: 0.18 seconds
```

In [30]:
```
dicoModelParam = {
    'batch_size': 32, # à chaque itération, 32 images d'entraînement à la fois
    'num_epochs': 10, # 10 itérations sur la totalité des images d'entraînement
    'kernel_size': 3, # filtre de 3x3
    'pool_size': 2, # filtre de 2x2
    'conv_depth_1': 32, # d'abord 32 neurones de profondeur par couche
    'conv_depth_2': 64, # puis 64 neurones de profondeur après le premier pooling
    'drop_prob_1': 0.25, # probabilité de 0.25 pour les deux premiers dropout
    'drop_prob_2': 0.5, # probabilité de 0.50 dans la couche Dense/FC
    'hidden_size': 512 # 512 neurones dans la couche Dense/FC
}
modelConvolution = convolutionModel(train_data, dicoModelParam, num_classes)
modelConvolution.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 224, 224, 32)      896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 224, 224, 32)      9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 112, 112, 32)      0
_________________________________________________________________
dropout_101 (Dropout)        (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 112, 112, 64)      18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 112, 112, 64)      36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
dropout_102 (Dropout)        (None, 56, 56, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 200704)            0
_________________________________________________________________
dense_111 (Dense)            (None, 512)               102760960
_________________________________________________________________
dropout_103 (Dropout)        (None, 512)               0
_________________________________________________________________
dense_112 (Dense)            (None, 10)                5130
=================================================================
Total params: 102,831,658
Trainable params: 102,831,658
Non-trainable params: 0
_________________________________________________________________
```

- Comme on peut le voir, si les précédents modèles de perceptrons possédaient un nombre de paramètres comme 269 322 pour MNIST 28x28 et 38 603 786 pour le PMC 3 couches ImageNet 224x224), avec ce modèle convolutif, on dépasse les 100 millions de paramètres.
- On peut voir que le pooling avec un filtre 2x2 (ici, le stride/pas est de 2) a divisé par 2 les dimensions spatiales (height/width) tout en préservant la profondeur du volume de sortie. Par exemple pour le premier pooling, on obtient bien : W2 = (W1 - F) / S + 1 = (224 - 2) / 2 + 1 = 112 et pour le second : (112 - 2) / 2 + 1 = 56. On supprime alors 75% des activations.

In [31]:
```
modelConvolution, historyConvolution, scoreConvolution = fitAndEvaluate(modelConvolution, train_data, train_target, test_dat
a, test_target, 128, 5, 1)
```

```
Train on 1000 samples, validate on 707 samples
Epoch 1/5
1000/1000 [==============================] - 234s - loss: 14.5706 - acc: 0.0940 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 2/5
1000/1000 [==============================] - 219s - loss: 14.4902 - acc: 0.1010 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 3/5
1000/1000 [==============================] - 221s - loss: 14.5224 - acc: 0.0990 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 4/5
1000/1000 [==============================] - 218s - loss: 14.4257 - acc: 0.1050 - val_loss: 14.7502 - val_acc: 0.0849
Epoch 5/5
1000/1000 [==============================] - 230s - loss: 14.6191 - acc: 0.0930 - val_loss: 14.7502 - val_acc: 0.0849
707/707 [==============================] - 41s

Test loss: 14.7502228711
Test accuracy: 0.0848656294201
```

Normalement, on devrait faire au moins 200 epochs... mais comme on peut le voir, les itérations prennent beaucoup plus de temps que pour les modèles précédents. Le score n'est pas très bon, assez similaire aux perceptrons.

Comme les CNN entraînés sur des petits dataset souffrent habituellement du problème d'overfitting (par exemple nos 1000 images est un petit dataset), une des solutions pour y remédier est d'initialiser son CNN avec des poids appris sur un dataset beaucoup plus fourni, et ensuite adapter les poids à son propre dataset. On peut utiliser pour cela des modèles pré-entraînés.

5- Modèle pré-entraîné RESNET50

RESNET (resnet pour Residual Network) a pour origine du residual learning. Il est le gagnant du ILSVRC 2015.
Au fur et à mesure des recherches, avec l'approfondissement des réseaux, l'entraînement est devenu de plus en plus difficile et l'accuracy a commencé à saturer et à se dégrader. On pouvait penser qu'en ajoutant toujours plus de couches, on obtiendrait toujours de meilleurs résultats, ou du moins égaux. Ce n'est pas le cas. Le residual learning a tenté de résoudre ces problèmes. Ses auteurs sont partis de l'hypothèse que les mappings directs entre deux entités étaient difficiles à apprendre. Ils ont donc proposé, plutôt que d'apprendre les liens sous-jacents, de tenter d'apprendre les différences, c'est-à-dire les résidus. Il ne reste plus alors qu'à ajouter ces résidus à l'entité initiale pour obtenir l'autre.

En residual learning, le modèle effectue des raccourcis en connectant directement les input d'une couche plus basse (ex. la n-ième) à une couche plus élevée (ex. la (n+x)-ième). Il a été prouvé qu'entraîner ce genre de réseaux était plus facile qu'entraîner un simple CNN et que cela résolvait le problème de dégradation de l'accuracy.

On peut voir le résumé de sa structure ci-dessous, mais le graphe de ses 50 couches est plus lisible dans cette page:
http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006

In [76]:
```
modelrestnet50_ = ResNet50(weights='imagenet', include_top=True)
print(modelrestnet50_.summary())
```

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
input_12 (InputLayer)            (None, 224, 224, 3)   0
____________________________________________________________________________________________________
conv1 (Conv2D)                   (None, 112, 112, 64)  9472        input_12[0][0]
____________________________________________________________________________________________________
bn_conv1 (BatchNormalization)    (None, 112, 112, 64)  256         conv1[0][0]
____________________________________________________________________________________________________
activation_340 (Activation)      (None, 112, 112, 64)  0           bn_conv1[0][0]
____________________________________________________________________________________________________
max_pooling2d_12 (MaxPooling2D)  (None, 55, 55, 64)    0           activation_340[0][0]
____________________________________________________________________________________________________
res2a_branch2a (Conv2D)          (None, 55, 55, 64)    4160        max_pooling2d_12[0][0]
____________________________________________________________________________________________________
bn2a_branch2a (BatchNormalizatio (None, 55, 55, 64)    256         res2a_branch2a[0][0]
____________________________________________________________________________________________________
activation_341 (Activation)      (None, 55, 55, 64)    0           bn2a_branch2a[0][0]
____________________________________________________________________________________________________
res2a_branch2b (Conv2D)          (None, 55, 55, 64)    36928       activation_341[0][0]
____________________________________________________________________________________________________
bn2a_branch2b (BatchNormalizatio (None, 55, 55, 64)    256         res2a_branch2b[0][0]
____________________________________________________________________________________________________
activation_342 (Activation)      (None, 55, 55, 64)    0           bn2a_branch2b[0][0]
____________________________________________________________________________________________________
res2a_branch2c (Conv2D)          (None, 55, 55, 256)   16640       activation_342[0][0]
____________________________________________________________________________________________________
res2a_branch1 (Conv2D)           (None, 55, 55, 256)   16640       max_pooling2d_12[0][0]
____________________________________________________________________________________________________
bn2a_branch2c (BatchNormalizatio (None, 55, 55, 256)   1024        res2a_branch2c[0][0]
____________________________________________________________________________________________________
bn2a_branch1 (BatchNormalization (None, 55, 55, 256)   1024        res2a_branch1[0][0]
```

add_105 (Add)	(None, 55, 55, 256)	0	bn2a_branch2c[0][0] bn2a_branch1[0][0]
activation_343 (Activation)	(None, 55, 55, 256)	0	add_105[0][0]
res2b_branch2a (Conv2D)	(None, 55, 55, 64)	16448	activation_343[0][0]
bn2b_branch2a (BatchNormalizatio	(None, 55, 55, 64)	256	res2b_branch2a[0][0]
activation_344 (Activation)	(None, 55, 55, 64)	0	bn2b_branch2a[0][0]
res2b_branch2b (Conv2D)	(None, 55, 55, 64)	36928	activation_344[0][0]
bn2b_branch2b (BatchNormalizatio	(None, 55, 55, 64)	256	res2b_branch2b[0][0]
activation_345 (Activation)	(None, 55, 55, 64)	0	bn2b_branch2b[0][0]
res2b_branch2c (Conv2D)	(None, 55, 55, 256)	16640	activation_345[0][0]
bn2b_branch2c (BatchNormalizatio	(None, 55, 55, 256)	1024	res2b_branch2c[0][0]
add_106 (Add)	(None, 55, 55, 256)	0	bn2b_branch2c[0][0] activation_343[0][0]
activation_346 (Activation)	(None, 55, 55, 256)	0	add_106[0][0]
res2c_branch2a (Conv2D)	(None, 55, 55, 64)	16448	activation_346[0][0]
bn2c_branch2a (BatchNormalizatio	(None, 55, 55, 64)	256	res2c_branch2a[0][0]
activation_347 (Activation)	(None, 55, 55, 64)	0	bn2c_branch2a[0][0]
res2c_branch2b (Conv2D)	(None, 55, 55, 64)	36928	activation_347[0][0]
bn2c_branch2b (BatchNormalizatio	(None, 55, 55, 64)	256	res2c_branch2b[0][0]
activation_348 (Activation)	(None, 55, 55, 64)	0	bn2c_branch2b[0][0]
res2c_branch2c (Conv2D)	(None, 55, 55, 256)	16640	activation_348[0][0]
bn2c_branch2c (BatchNormalizatio	(None, 55, 55, 256)	1024	res2c_branch2c[0][0]
add_107 (Add)	(None, 55, 55, 256)	0	bn2c_branch2c[0][0] activation_346[0][0]
activation_349 (Activation)	(None, 55, 55, 256)	0	add_107[0][0]
res3a_branch2a (Conv2D)	(None, 28, 28, 128)	32896	activation_349[0][0]
bn3a_branch2a (BatchNormalizatio	(None, 28, 28, 128)	512	res3a_branch2a[0][0]
activation_350 (Activation)	(None, 28, 28, 128)	0	bn3a_branch2a[0][0]
res3a_branch2b (Conv2D)	(None, 28, 28, 128)	147584	activation_350[0][0]
bn3a_branch2b (BatchNormalizatio	(None, 28, 28, 128)	512	res3a_branch2b[0][0]
activation_351 (Activation)	(None, 28, 28, 128)	0	bn3a_branch2b[0][0]
res3a_branch2c (Conv2D)	(None, 28, 28, 512)	66048	activation_351[0][0]
res3a_branch1 (Conv2D)	(None, 28, 28, 512)	131584	activation_349[0][0]
bn3a_branch2c (BatchNormalizatio	(None, 28, 28, 512)	2048	res3a_branch2c[0][0]
bn3a_branch1 (BatchNormalization	(None, 28, 28, 512)	2048	res3a_branch1[0][0]
add_108 (Add)	(None, 28, 28, 512)	0	bn3a_branch2c[0][0] bn3a_branch1[0][0]
activation_352 (Activation)	(None, 28, 28, 512)	0	add_108[0][0]
res3b_branch2a (Conv2D)	(None, 28, 28, 128)	65664	activation_352[0][0]
bn3b_branch2a (BatchNormalizatio	(None, 28, 28, 128)	512	res3b_branch2a[0][0]
activation_353 (Activation)	(None, 28, 28, 128)	0	bn3b_branch2a[0][0]
res3b_branch2b (Conv2D)	(None, 28, 28, 128)	147584	activation_353[0][0]
bn3b_branch2b (BatchNormalizatio	(None, 28, 28, 128)	512	res3b_branch2b[0][0]
activation_354 (Activation)	(None, 28, 28, 128)	0	bn3b_branch2b[0][0]
res3b_branch2c (Conv2D)	(None, 28, 28, 512)	66048	activation_354[0][0]
bn3b_branch2c (BatchNormalizatio	(None, 28, 28, 512)	2048	res3b_branch2c[0][0]
add_109 (Add)	(None, 28, 28, 512)	0	bn3b_branch2c[0][0] activation_352[0][0]

activation_355 (Activation)	(None, 28, 28, 512)	0	add_109[0][0]
res3c_branch2a (Conv2D)	(None, 28, 28, 128)	65664	activation_355[0][0]
bn3c_branch2a (BatchNormalizatio	(None, 28, 28, 128)	512	res3c_branch2a[0][0]
activation_356 (Activation)	(None, 28, 28, 128)	0	bn3c_branch2a[0][0]
res3c_branch2b (Conv2D)	(None, 28, 28, 128)	147584	activation_356[0][0]
bn3c_branch2b (BatchNormalizatio	(None, 28, 28, 128)	512	res3c_branch2b[0][0]
activation_357 (Activation)	(None, 28, 28, 128)	0	bn3c_branch2b[0][0]
res3c_branch2c (Conv2D)	(None, 28, 28, 512)	66048	activation_357[0][0]
bn3c_branch2c (BatchNormalizatio	(None, 28, 28, 512)	2048	res3c_branch2c[0][0]
add_110 (Add)	(None, 28, 28, 512)	0	bn3c_branch2c[0][0] activation_355[0][0]
activation_358 (Activation)	(None, 28, 28, 512)	0	add_110[0][0]
res3d_branch2a (Conv2D)	(None, 28, 28, 128)	65664	activation_358[0][0]
bn3d_branch2a (BatchNormalizatio	(None, 28, 28, 128)	512	res3d_branch2a[0][0]
activation_359 (Activation)	(None, 28, 28, 128)	0	bn3d_branch2a[0][0]
res3d_branch2b (Conv2D)	(None, 28, 28, 128)	147584	activation_359[0][0]
bn3d_branch2b (BatchNormalizatio	(None, 28, 28, 128)	512	res3d_branch2b[0][0]
activation_360 (Activation)	(None, 28, 28, 128)	0	bn3d_branch2b[0][0]
res3d_branch2c (Conv2D)	(None, 28, 28, 512)	66048	activation_360[0][0]
bn3d_branch2c (BatchNormalizatio	(None, 28, 28, 512)	2048	res3d_branch2c[0][0]
add_111 (Add)	(None, 28, 28, 512)	0	bn3d_branch2c[0][0] activation_358[0][0]
activation_361 (Activation)	(None, 28, 28, 512)	0	add_111[0][0]
res4a_branch2a (Conv2D)	(None, 14, 14, 256)	131328	activation_361[0][0]
bn4a_branch2a (BatchNormalizatio	(None, 14, 14, 256)	1024	res4a_branch2a[0][0]
activation_362 (Activation)	(None, 14, 14, 256)	0	bn4a_branch2a[0][0]
res4a_branch2b (Conv2D)	(None, 14, 14, 256)	590080	activation_362[0][0]
bn4a_branch2b (BatchNormalizatio	(None, 14, 14, 256)	1024	res4a_branch2b[0][0]
activation_363 (Activation)	(None, 14, 14, 256)	0	bn4a_branch2b[0][0]
res4a_branch2c (Conv2D)	(None, 14, 14, 1024)	263168	activation_363[0][0]
res4a_branch1 (Conv2D)	(None, 14, 14, 1024)	525312	activation_361[0][0]
bn4a_branch2c (BatchNormalizatio	(None, 14, 14, 1024)	4096	res4a_branch2c[0][0]
bn4a_branch1 (BatchNormalization	(None, 14, 14, 1024)	4096	res4a_branch1[0][0]
add_112 (Add)	(None, 14, 14, 1024)	0	bn4a_branch2c[0][0] bn4a_branch1[0][0]
activation_364 (Activation)	(None, 14, 14, 1024)	0	add_112[0][0]
res4b_branch2a (Conv2D)	(None, 14, 14, 256)	262400	activation_364[0][0]
bn4b_branch2a (BatchNormalizatio	(None, 14, 14, 256)	1024	res4b_branch2a[0][0]
activation_365 (Activation)	(None, 14, 14, 256)	0	bn4b_branch2a[0][0]
res4b_branch2b (Conv2D)	(None, 14, 14, 256)	590080	activation_365[0][0]
bn4b_branch2b (BatchNormalizatio	(None, 14, 14, 256)	1024	res4b_branch2b[0][0]
activation_366 (Activation)	(None, 14, 14, 256)	0	bn4b_branch2b[0][0]
res4b_branch2c (Conv2D)	(None, 14, 14, 1024)	263168	activation_366[0][0]
bn4b_branch2c (BatchNormalizatio	(None, 14, 14, 1024)	4096	res4b_branch2c[0][0]
add_113 (Add)	(None, 14, 14, 1024)	0	bn4b_branch2c[0][0] activation_364[0][0]
activation_367 (Activation)	(None, 14, 14, 1024)	0	add_113[0][0]
res4c_branch2a (Conv2D)	(None, 14, 14, 256)	262400	activation_367[0][0]
bn4c_branch2a (BatchNormalizatio	(None, 14, 14, 256)	1024	res4c_branch2a[0][0]
activation_368 (Activation)	(None, 14, 14, 256)	0	bn4c_branch2a[0][0]
res4c_branch2b (Conv2D)	(None, 14, 14, 256)	590080	activation_368[0][0]
bn4c_branch2b (BatchNormalizatio	(None, 14, 14, 256)	1024	res4c_branch2b[0][0]

activation_369 (Activation)	(None, 14, 14, 256)	0	bn4c_branch2b[0][0]
res4c_branch2c (Conv2D)	(None, 14, 14, 1024)	263168	activation_369[0][0]
bn4c_branch2c (BatchNormalizatio	(None, 14, 14, 1024)	4096	res4c_branch2c[0][0]
add_114 (Add)	(None, 14, 14, 1024)	0	bn4c_branch2c[0][0] activation_367[0][0]
activation_370 (Activation)	(None, 14, 14, 1024)	0	add_114[0][0]
res4d_branch2a (Conv2D)	(None, 14, 14, 256)	262400	activation_370[0][0]
bn4d_branch2a (BatchNormalizatio	(None, 14, 14, 256)	1024	res4d_branch2a[0][0]
activation_371 (Activation)	(None, 14, 14, 256)	0	bn4d_branch2a[0][0]
res4d_branch2b (Conv2D)	(None, 14, 14, 256)	590080	activation_371[0][0]
bn4d_branch2b (BatchNormalizatio	(None, 14, 14, 256)	1024	res4d_branch2b[0][0]
activation_372 (Activation)	(None, 14, 14, 256)	0	bn4d_branch2b[0][0]
res4d_branch2c (Conv2D)	(None, 14, 14, 1024)	263168	activation_372[0][0]
bn4d_branch2c (BatchNormalizatio	(None, 14, 14, 1024)	4096	res4d_branch2c[0][0]
add_115 (Add)	(None, 14, 14, 1024)	0	bn4d_branch2c[0][0] activation_370[0][0]
activation_373 (Activation)	(None, 14, 14, 1024)	0	add_115[0][0]
res4e_branch2a (Conv2D)	(None, 14, 14, 256)	262400	activation_373[0][0]
bn4e_branch2a (BatchNormalizatio	(None, 14, 14, 256)	1024	res4e_branch2a[0][0]
activation_374 (Activation)	(None, 14, 14, 256)	0	bn4e_branch2a[0][0]
res4e_branch2b (Conv2D)	(None, 14, 14, 256)	590080	activation_374[0][0]
bn4e_branch2b (BatchNormalizatio	(None, 14, 14, 256)	1024	res4e_branch2b[0][0]
activation_375 (Activation)	(None, 14, 14, 256)	0	bn4e_branch2b[0][0]
res4e_branch2c (Conv2D)	(None, 14, 14, 1024)	263168	activation_375[0][0]
bn4e_branch2c (BatchNormalizatio	(None, 14, 14, 1024)	4096	res4e_branch2c[0][0]
add_116 (Add)	(None, 14, 14, 1024)	0	bn4e_branch2c[0][0] activation_373[0][0]
activation_376 (Activation)	(None, 14, 14, 1024)	0	add_116[0][0]
res4f_branch2a (Conv2D)	(None, 14, 14, 256)	262400	activation_376[0][0]
bn4f_branch2a (BatchNormalizatio	(None, 14, 14, 256)	1024	res4f_branch2a[0][0]
activation_377 (Activation)	(None, 14, 14, 256)	0	bn4f_branch2a[0][0]
res4f_branch2b (Conv2D)	(None, 14, 14, 256)	590080	activation_377[0][0]
bn4f_branch2b (BatchNormalizatio	(None, 14, 14, 256)	1024	res4f_branch2b[0][0]
activation_378 (Activation)	(None, 14, 14, 256)	0	bn4f_branch2b[0][0]
res4f_branch2c (Conv2D)	(None, 14, 14, 1024)	263168	activation_378[0][0]
bn4f_branch2c (BatchNormalizatio	(None, 14, 14, 1024)	4096	res4f_branch2c[0][0]
add_117 (Add)	(None, 14, 14, 1024)	0	bn4f_branch2c[0][0] activation_376[0][0]
activation_379 (Activation)	(None, 14, 14, 1024)	0	add_117[0][0]
res5a_branch2a (Conv2D)	(None, 7, 7, 512)	524800	activation_379[0][0]
bn5a_branch2a (BatchNormalizatio	(None, 7, 7, 512)	2048	res5a_branch2a[0][0]
activation_380 (Activation)	(None, 7, 7, 512)	0	bn5a_branch2a[0][0]
res5a_branch2b (Conv2D)	(None, 7, 7, 512)	2359808	activation_380[0][0]
bn5a_branch2b (BatchNormalizatio	(None, 7, 7, 512)	2048	res5a_branch2b[0][0]
activation_381 (Activation)	(None, 7, 7, 512)	0	bn5a_branch2b[0][0]
res5a_branch2c (Conv2D)	(None, 7, 7, 2048)	1050624	activation_381[0][0]
res5a_branch1 (Conv2D)	(None, 7, 7, 2048)	2099200	activation_379[0][0]
bn5a_branch2c (BatchNormalizatio	(None, 7, 7, 2048)	8192	res5a_branch2c[0][0]
bn5a_branch1 (BatchNormalization	(None, 7, 7, 2048)	8192	res5a_branch1[0][0]
add_118 (Add)	(None, 7, 7, 2048)	0	bn5a_branch2c[0][0] bn5a_branch1[0][0]

```
activation_382 (Activation)      (None, 7, 7, 2048)    0           add_118[0][0]
res5b_branch2a (Conv2D)          (None, 7, 7, 512)     1049088     activation_382[0][0]
bn5b_branch2a (BatchNormalizatio (None, 7, 7, 512)     2048        res5b_branch2a[0][0]
activation_383 (Activation)      (None, 7, 7, 512)     0           bn5b_branch2a[0][0]
res5b_branch2b (Conv2D)          (None, 7, 7, 512)     2359808     activation_383[0][0]
bn5b_branch2b (BatchNormalizatio (None, 7, 7, 512)     2048        res5b_branch2b[0][0]
activation_384 (Activation)      (None, 7, 7, 512)     0           bn5b_branch2b[0][0]
res5b_branch2c (Conv2D)          (None, 7, 7, 2048)    1050624     activation_384[0][0]
bn5b_branch2c (BatchNormalizatio (None, 7, 7, 2048)    8192        res5b_branch2c[0][0]
add_119 (Add)                    (None, 7, 7, 2048)    0           bn5b_branch2c[0][0]
                                                                   activation_382[0][0]
activation_385 (Activation)      (None, 7, 7, 2048)    0           add_119[0][0]
res5c_branch2a (Conv2D)          (None, 7, 7, 512)     1049088     activation_385[0][0]
bn5c_branch2a (BatchNormalizatio (None, 7, 7, 512)     2048        res5c_branch2a[0][0]
activation_386 (Activation)      (None, 7, 7, 512)     0           bn5c_branch2a[0][0]
res5c_branch2b (Conv2D)          (None, 7, 7, 512)     2359808     activation_386[0][0]
bn5c_branch2b (BatchNormalizatio (None, 7, 7, 512)     2048        res5c_branch2b[0][0]
activation_387 (Activation)      (None, 7, 7, 512)     0           bn5c_branch2b[0][0]
res5c_branch2c (Conv2D)          (None, 7, 7, 2048)    1050624     activation_387[0][0]
bn5c_branch2c (BatchNormalizatio (None, 7, 7, 2048)    8192        res5c_branch2c[0][0]
add_120 (Add)                    (None, 7, 7, 2048)    0           bn5c_branch2c[0][0]
                                                                   activation_385[0][0]
activation_388 (Activation)      (None, 7, 7, 2048)    0           add_120[0][0]
avg_pool (AveragePooling2D)      (None, 1, 1, 2048)    0           activation_388[0][0]
flatten_7 (Flatten)              (None, 2048)          0           avg_pool[0][0]
fc1000 (Dense)                   (None, 1000)          2049000     flatten_7[0][0]
==================================================================================================
Total params: 25,636,712
Trainable params: 25,583,592
Non-trainable params: 53,120
__________________________________________________________________________________________________
None
```

In [37]:

```python
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input as resnet50preprocess_input
from keras.applications.resnet50 import decode_predictions as resnet50decode_predictions

def predictImageBreedWithResnet50(_model, _dogImagesDir, _dicoBreedIdName, _fileName):
    """ Méthode qui prédit la race du chien dont l'image est en entrée avec le modèle Resnet50
    Arguments:
    _model -- le modèle Resnet50
    _dogImagesDir -- chemin vers le répertoire local des images
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    _fileName -- le nom de fichier d'un chien
    Retour:
    predicted -- la prédiction de la race du chien
    """
    img = readImage(_dogImagesDir, _dicoBreedIdName, _fileName, (224, 224))
    imgExpended = np.expand_dims(img, axis=0) # sert à quoi ?
    imgExpendedPreProcessed= resnet50preprocess_input(imgExpended)
    preds = _model.predict(imgExpendedPreProcessed)
    predicted = resnet50decode_predictions(preds, top=1)[0][0]
    return predicted

def scoreModeleResNet50(_idImageTestListShuffled, _dogImagesDir, _dicoDogBreedIdCateg, _dicoDogBreedIdNameCateg, _dicoBreedId
Name):
    """ Méthode qui calcule le score du modèle Resnet50 sur toutes les images de test
    Arguments:
    _idImageTestListShuffled -- liste des noms de fichier des images de test mélangées
    _dogImagesDir -- chemin vers le répertoire local des images
    _dicoDogBreedIdCateg -- dictionnaire des positions des identifiants des races des chiens
    _dicoDogBreedIdNameCateg -- le dictionnaire des positions des répertoires/races des chiens
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    Retour:
    score -- le score du modèle Resnet50 sur toutes les images de test
    """
    start_time = time.time()
    model = ResNet50(weights='imagenet', include_top=True) # inclut la dernière FC en haut de la pile des couches
    count = 0
    for idImage in _idImageTestListShuffled:
```

```
        predicted = predictImageBreedWithResnet50(model, _dogImagesDir, _dicoBreedIdName, idImage.split(".")[0])
        realCateg = getDogBreedCategFromFileName(_dicoDogBreedIdCateg, idImage)
        categPredicted = -1
        try:
            categPredicted = _dicoDogBreedIdNameCateg[predicted[0]+'-'+predicted[1]]
        except:
            pass # quand la Breed n'est pas trouvée dans dicoDogBreedIdNameCateg car en dehors du choix des races
        if realCateg == categPredicted:
            count += 1
    score = 100 * count / len(_idImageTestListShuffled)
    print('ScoreModeleResNet50 time: {} seconds'.format(round(time.time() - start_time, 2)))
    return score
```

Résultats :

In [38]:
```
scoreModeleResNet50(idImageTestListShuffled, dogImagesDir, dicoDogBreedIdCateg, dicoDogBreedIdNameCateg, dicoBreedIdName)
```

```
ScoreModeleResNet50 time: 199.93 seconds
```

Out[38]:
```
72.84299858557284
```

On obtient ici un très bon score de 72.84 % de bonnes prévisions.

Pour les autres modèles pré-entraînés, j'ai mis le code en commun:

Modèles pré-entraînés, code mis en commun

In [39]:
```
from keras.applications import ResNet50
from keras.applications import InceptionV3
from keras.applications import Xception # TensorFlow ONLY
from keras.applications import VGG16
from keras.applications import VGG19
from keras.applications import MobileNet
from keras.applications.inception_v3 import preprocess_input as preprocess_input_ception
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img

MODELS = {
    "vgg16": VGG16,
    "vgg19": VGG19,
    "inception": InceptionV3,
    "xception": Xception, # TensorFlow ONLY
    "resnet": ResNet50,
    "mobileNet": MobileNet
}
```

In [64]:
```
def getModel(_chosenModel, _print):
    """ Méthode qui instancie un modèle pré-entraîné et qui charge ses poids
    Arguments:
    _chosenModel -- le libellé du modèle
    _print -- choix d'afficher les informations de chargement du modèle
    Retour:
    model -- le modèle instancié
    """
    # si c'est la première fois que le modèle est instancié, le chargement des paramètres peut prendre du temps.
    if _print:
        print("[INFO] loading {}...".format(_chosenModel))

    Network = MODELS[_chosenModel]
    model = Network(weights="imagenet")

    if _print:
        print("[INFO] model loaded {}...".format(_chosenModel))

    return model
def getInputShapeAndPreprocessFromTypeOfModel(_chosenModel):
    """ Méthode qui détermine l'input shape et le préprocessing adapté au modèle choisi
    Arguments:
    _chosenModel -- le libellé du modèle
    Retour:
    inputShape -- l'input shape adapté au modèle choisi
    preprocess -- le préprocessing adapté au modèle choisi
    """

    if _chosenModel in ("inception", "xception"):
        # pour InceptionV3 et Xception, la taille en input doit être de (299x299) plutôt que (224x224)
        # le prépocessing est également différent
        inputShape = (299, 299)
        preprocess = preprocess_input_ception
    else:
        inputShape = (224, 224)
        preprocess = imagenet_utils.preprocess_input

    return inputShape, preprocess
```

```
def predict5MostLikelyBreedsFromImage(_model, _pathImage, _inputShape, _chosenModel, _preprocess, _print):
    """ Méthode qui prédit les 5 races les plus probables d'un chien
    Arguments:
    _model -- le modèle choisi
    _pathImage -- le chemin entier vers le fichier d'un chien
    _inputShape -- la forme de l'input
    _chosenModel -- le libellé du modèle
    _preprocess -- le préprocessing du modèle
    _print -- affiche ou non des informations
    Retour:
    result -- la race la plus probable
    probResult -- le score de la race la plus probable
    """

    if _print:
        print("[INFO] loading and pre-processing image...")

    image = load_img(_pathImage, target_size=_inputShape)
    image = img_to_array(image)

    # l'image est présentée sous forme d'un tableau Numpy de forme (inputShape[0], inputShape[1], 3),
    # cependant, on doit étendre la dimension de ce tableau vers le format (1, inputShape[0], inputShape[1], 3)
    # afin qu'il puisse être utilisé par le réseau
    image = np.expand_dims(image, axis=0)

    # préprocessing de l'image avec la méthode adaptée au modèle pré-entraîné (normalisation)
    image = _preprocess(image)

    if _print:
        print("[INFO] classifying image with '{}'...".format(_chosenModel))

    preds = _model.predict(image)
    P = imagenet_utils.decode_predictions(preds)

    # Affichage des 5 races les plus probables
    if _print:
        for (i, (imagenetID, label, prob)) in enumerate(P[0]):
            print("{}. {}: {:.2f}%".format(i + 1, label, prob * 100))

    # race la plus probable
    result = P[0][0][1]
    probResult = P[0][0][2]

    return result, probResult

def scoreModele(_model, _idImageTestListShuffled, _dogImagesDir, _dicoDogBreedIdNameCateg, _dicoBreedIdName, _chosenModel):
    """ Méthode qui calcule le score d'un modèle sur toutes les images de test
    Arguments:
    _model -- le modèle choisi
    _idImageTestListShuffled -- liste des noms de fichier des images de test mélangées
    _dogImagesDir -- chemin vers le répertoire local des images
    _dicoDogBreedIdNameCateg -- le dictionnaire des positions des répertoires/races des chiens
    _dicoBreedIdName -- le dictionnaire des noms de race pour chaque identifiant de race
    _chosenModel -- le libellé du modèle
    Retour:
    score -- le score du modèle
    """
    start_time = time.time()
    count = 0
    inputShape, preprocess = getInputShapeAndPreprocessFromTypeOfModel(_chosenModel)
    for idImage in _idImageTestListShuffled:
        pathImage = _dogImagesDir+getDogBreedDirFromFileName(idImage, _dicoBreedIdName)+"/"+idImage
        predicted = predict5MostLikelyBreedsFromImage(_model, pathImage, inputShape, _chosenModel, preprocess, False)
        realCateg = getDogBreedNameFromFileName(idImage, _dicoBreedIdName)
        if predicted[0] == realCateg:
            count += 1
    score = 100 * count / len(_idImageTestListShuffled)
    print('ScoreModele time: {} seconds'.format(round(time.time() - start_time, 2)))
    print('Score pour le modèle {0} : {1}'.format(_chosenModel, score))
    return score
```

Test avec une image extérieure à ImageNet

Pour ne pas reprendre une image d'ImageNet, je suis allé chercher une photo d'un boston-bull ailleurs sur Internet afin de voir ce que donnent les modèles avec autre chose qu'une photo d'ImageNet car dans les poids pré-entraînés, il doit sans doute y avoir également les photos des tests.
Photo récupérée sur : http://1petloversworld.com/boston-terrier.html (Un boston-terrier se dit aussi boston-bull en américain.)

In [77]:
```
pathImage = dogImagesDir+"boston13.jpg"
showExternalImage(pathImage, (224,224))
```

Modèles pré-entraînés, RESNET50

In [49]:
```
chosenModelResnet = "resnet"
modelResnet = getModel(chosenModelResnet, True)
```

```
[INFO] loading resnet...
[INFO] model loaded resnet...
```

In [50]:
```
# Image extérieure téléchargée dans le dossier Images localement
pathImageExternalDog = dogImagesDir+"boston13.jpg"
# si on voulait une image d'ImageNet, il suffirait de la récupérer ainsi:
# fileName = "n02096585_9534.jpg"
# pathImage = dogImagesDir+getDogBreedDirFromFileName(fileName, dicoBreedIdName)+"/"+fileName

inputShapeResnet, preprocessResnet = getInputShapeAndPreprocessFromTypeOfModel(chosenModelResnet)
predict5MostLikelyBreedsFromImage(modelResnet, pathImageExternalDog, inputShapeResnet, chosenModelResnet, preprocessResnet, T
rue)
```

```
[INFO] loading and pre-processing image...
[INFO] classifying image with 'resnet'...
1. Boston_bull: 84.29%
2. French_bulldog: 14.37%
3. Cardigan: 0.46%
4. boxer: 0.08%
5. muzzle: 0.06%
```

Out[50]:
```
('Boston_bull', 0.84291404)
```

- Dans cet exemple, on peut voir que le chien dont l'image apparaît ci-dessus a été classé en première position comme étant un boston-bull avec un résultat de 84.29, loin devant les autres classes. Et c'est en effet un boston-bull. (Les % ne sont pas de réels pourcentages puisque ce ne sont pas de réelles probabilités).
- Le modèle a mis le bulldog français en deuxième position, et en effet, boston-bull et bulldog français se ressemblent énormément.
- Je ne l'ai pas mis ici mais quand on fait un test avec une image d'ImageNet, même de test, on obtient un meilleur résultat.

6- Modèle pré-entraîné VGG16

VGG16 (Visual Geometry Group) est un CNN à 16 couches, il fut 2ème du ILSVRC 2014 :
http://book.paddlepaddle.org/03.image_classification/image/vgg16.png
Il contient essentiellement des groupes de couches de convolution 3x3 entre lesquels s'intercalent des couches de pooling 2x2, et se termine par 3 FC. Il semble avoir été le premier à surpasser l'être humain dans la reconnaissance des images d'ImageNet.
C'est un modèle qui au début utilisait beaucoup de paramètres, toutefois il s'est avéré qu'on pouvait retirer les FC sans grande baisse de performance.

In [44]:
```
chosenModelVGG16 = "vgg16"
modelVGG16 = getModel(chosenModelVGG16, True)
```

```
[INFO] loading vgg16...
[INFO] model loaded vgg16...
```

In [45]:
```
inputShapeVGG16, preprocessVGG16 = getInputShapeAndPreprocessFromTypeOfModel(chosenModelVGG16)
predict5MostLikelyBreedsFromImage(modelVGG16, pathImageExternalDog, inputShapeVGG16, chosenModelVGG16, preprocessVGG16, True)
```

```
[INFO] loading and pre-processing image...
[INFO] classifying image with 'vgg16'...
1. Boston_bull: 91.67%
2. French_bulldog: 4.43%
3. toy_terrier: 1.84%
4. Cardigan: 0.49%
5. space_heater: 0.36%
```

Out[45]:
```
('Boston_bull', 0.91667604)
```

In [65]:
```
scoreModele(modelVGG16, idImageTestListShuffled, dogImagesDir, dicoDogBreedIdNameCateg, dicoBreedIdName, chosenModelVGG16)
```

```
ScoreModele time: 320.29 seconds
Score pour le modèle vgg16 : 68.31683168316832
```

Out[65]:
```
68.31683168316832
```

Le résultat est un peu moins bon qu'avec Resnet50, toutefois, il est difficile de comparer avec si peu d'images d'entraînement. Mais il est quand même à noter que Resnet50, plus récent et plus profond et utilisant la technique de residual learning est connu pour donner de meilleurs résultats que VGG16.

7- Modèle pré-entraîné Inception V3

Si Resnet a exploré les profondeurs, la famille Inception s'est intéressé à la largeur, c'est-à-dire en cherchant à élargir les modèles sans pour autant augmenter les temps de calcul. Avec Inception V1 (ou GoogLeNet, vainqueur du ILSVRC 2014 avec 22 couches), ils ont ainsi développé des modules appelés "Inception" (d'autres versions ont suivi, V2, V3 et V4). Un module Inception calcule en parallèle plusieurs transformations différentes du même input et concatène ses résultats en un unique output. Pour chaque couche, Inception effectue des transformations convolutives 5x5, 3x3 et un maxpooling 3x3, et c'est à la couche suivante du modèle de décider quelle information utiliser. Même si cela semble augmenter drastiquement le nombre de paramètres, ils ont ajouté des convolutions 1x1 pour réduire les dimensions.
Schéma : https://cdn-images-1.medium.com/max/1400/1*aq4tcBl9t5Z36kTDeZSOHA.png

In [66]:
```
chosenModelInception = "inception"
modelInception = getModel(chosenModelInception, True)
```

```
[INFO] loading inception...
[INFO] model loaded inception...
```

In [67]:
```
inputShapeInception, preprocessInception = getInputShapeAndPreprocessFromTypeOfModel(chosenModelInception)
predict5MostLikelyBreedsFromImage(modelInception, pathImageExternalDog, inputShapeInception, chosenModelInception, preprocess
Inception, True)
```

```
[INFO] loading and pre-processing image...
[INFO] classifying image with 'inception'...
1. Boston_bull: 82.46%
2. French_bulldog: 11.14%
3. toy_terrier: 0.69%
4. Cardigan: 0.14%
5. basenji: 0.09%
```

Out[67]: ('Boston_bull', 0.82463211)

In [68]:
```
scoreModele(modelInception, idImageTestListShuffled, dogImagesDir, dicoDogBreedIdNameCateg, dicoBreedIdName, chosenModelIncep
tion)
```

```
ScoreModele time: 208.92 seconds
Score pour le modèle inception : 86.28005657708628
```

Out[68]: 86.28005657708628

On obtient bien toujours la bonne catégorie, et un score plus élevé avec 86.28% de bonnes prévisions.

8- Modèle pré-entraîné Xception

Xception est un Inception "Xtrême".
Dans un ConvNet traditionnel, les couches convolutives recherchent des corrélations à travers l'espace (height/width) et la profondeur (depth) grâce aux filtres. Inception sépare la région de l'input et les canaux à l'aide de convolutions 1x1 qui séparent l'input original en plusieurs petits espaces sur lesquels sont appliqués différents types de filtres. Xception, de son côté, utilise des convolutions de séparation le long de la profondeur et recherche les corrélations spatiales à travers tous les canaux indépendamment. [9]

In [69]:
```
chosenModelXception = "xception"
modelXception = getModel(chosenModelXception, True)
```

```
[INFO] loading xception...
[INFO] model loaded xception...
```

In [70]:
```
inputShapeXception, preprocessXception = getInputShapeAndPreprocessFromTypeOfModel(chosenModelXception)
predict5MostLikelyBreedsFromImage(modelXception, pathImageExternalDog, inputShapeXception, chosenModelXception, preprocessXce
ption, True)
```

```
[INFO] loading and pre-processing image...
[INFO] classifying image with 'xception'...
1. Boston_bull: 88.12%
2. French_bulldog: 3.10%
3. toy_terrier: 0.29%
4. Cardigan: 0.24%
5. Chihuahua: 0.09%
```

Out[70]: ('Boston_bull', 0.88115096)

In [71]:
```
scoreModele(modelXception, idImageTestListShuffled, dogImagesDir, dicoDogBreedIdNameCateg, dicoBreedIdName, chosenModelXcepti
on)
```

```
ScoreModele time: 375.1 seconds
Score pour le modèle xception : 87.27015558698727
```

Out[71]: 87.27015558698727

On obtient bien toujours la bonne catégorie, et un score encore plus élevé avec 87.27% de bonnes prévisions.

9- Modèles pré-entraînés, MobileNet

In [72]:
```
chosenModelMobileNet = "mobileNet"
modelMobileNet = getModel(chosenModelMobileNet, True)
```

```
[INFO] loading mobileNet...
[INFO] model loaded mobileNet...
```

In [73]:
```
inputShapeMobileNet, preprocessMobileNet = getInputShapeAndPreprocessFromTypeOfModel(chosenModelMobileNet)
predict5MostLikelyBreedsFromImage(modelMobileNet, pathImageExternalDog, inputShapeMobileNet, chosenModelMobileNet, preprocess
MobileNet, True)
```

```
[INFO] loading and pre-processing image...
[INFO] classifying image with 'mobileNet'...
1. pillow: 32.68%
2. Christmas_stocking: 27.50%
3. handkerchief: 3.40%
4. Chihuahua: 2.88%
5. Scotch_terrier: 2.20%
```

Out[73]: ('pillow', 0.32679021)

In [74]:
```
scoreModele(modelMobileNet, idImageTestListShuffled, dogImagesDir, dicoDogBreedIdNameCateg, dicoBreedIdName, chosenModelMobil
eNet)
```

```
ScoreModele time: 68.36 seconds
Score pour le modèle mobileNet : 0.9900990099009901
```

Out[74]: 0.9900990099009901

Je ne pense pas qu'un chien puisse ressembler à un coussin (à moins qu'il ne dorme dessus...) ni à un "bas de Noël"... Ce modèle fait pour les applications Mobile n'est sans doute pas adapté, ce qui semble normal puisque les images des mobiles sont d'un format différent de celles d'ImageNet.

IV- Code de prédiction de la race d'un chien

In [75]:
```
import keras
from keras.applications import imagenet_utils
from keras.applications.imagenet_utils import preprocess_input
from keras.preprocessing import image
from keras.applications import ResNet50
from keras.applications import InceptionV3
from keras.applications import Xception # TensorFlow ONLY
from keras.applications import VGG16
from keras.applications import VGG19
from keras.applications import MobileNet
from keras.applications.inception_v3 import preprocess_input as preprocess_input_ception
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

MODELS = {
    "vgg16": VGG16,
    "vgg19": VGG19,
    "inception": InceptionV3,
    "xception": Xception, # TensorFlow ONLY
    "resnet": ResNet50,
    "mobileNet": MobileNet
}
def getModel(_chosenModel, _print):
    """ Méthode qui instancie un modèle pré-entraîné et qui charge ses poids
    Arguments:
    _chosenModel -- le libellé du modèle
    _print -- choix d'afficher les informations de chargement du modèle
    Retour:
    model -- le modèle instancié
    """
    # si c'est la première fois que le modèle est instancié, le chargement des paramètres peut prendre du temps.
    if _print:
        print("[INFO] loading {}...".format(_chosenModel))

    Network = MODELS[_chosenModel]
    model = Network(weights="imagenet")

    if _print:
        print("[INFO] model loaded {}...".format(_chosenModel))

    return model

def getInputShapeAndPreprocessFromTypeOfModel(_chosenModel):
    """ Méthode qui détermine l'input shape et le préprocessing adapté au modèle choisi
    Arguments:
    _chosenModel -- le libellé du modèle
    Retour:
    inputShape -- l'input shape adapté au modèle choisi
    preprocess -- le préprocessing adapté au modèle choisi
    """

    if _chosenModel in ("inception", "xception"):
        # pour InceptionV3 et Xception, la taille en input doit être de (299x299) plutôt que (224x224)
        # le préprocessing est également différent
        inputShape = (299, 299)
        preprocess = preprocess_input_ception
    else:
        inputShape = (224, 224)
        preprocess = imagenet_utils.preprocess_input

    return inputShape, preprocess

def predict5MostLikelyBreedsFromImage(_model, _pathImage, _inputShape, _chosenModel, _preprocess, _print):
    """ Méthode qui prédit les 5 races les plus probables d'un chien
    Arguments:
    _model -- le modèle choisi
    _pathImage -- le chemin entier vers le fichier d'un chien
    _inputShape -- la forme de l'input
    _chosenModel -- le libellé du modèle
    _preprocess -- le préprocessing du modèle
    _print -- affiche ou non des informations
    Retour:
    result -- la race la plus probable
    probResult -- le score de la race la plus probable
    """

    if _print:
        print("[INFO] loading and pre-processing image...")

    image = load_img(_pathImage, target_size=_inputShape)
    image = img_to_array(image)
```

```
    # l'image est présentée sous forme d'un tableau Numpy de forme (inputShape[0], inputShape[1], 3),
    # cependant, on doit étendre la dimension de ce tableau vers le format (1, inputShape[0], inputShape[1], 3)
    # afin qu'il puisse être utilisé par le réseau
    image = np.expand_dims(image, axis=0)

    # préprocessing de l'image avec la méthode adaptée au modèle pré-entraîné (normalisation)
    image = _preprocess(image)

    if _print:
        print("[INFO] classifying image with '{}'...".format(_chosenModel))

    preds = _model.predict(image)
    P = imagenet_utils.decode_predictions(preds)

    # Affichage des 5 races les plus probables
    if _print:
        for (i, (imagenetID, label, prob)) in enumerate(P[0]):
            print("{}. {}: {:.2f}%".format(i + 1, label, prob * 100))

    # race la plus probable
    result = P[0][0][1]
    probResult = P[0][0][2]

    return result, probResult

def showExternalImageWithScore(_imagePath, _size, _breedName, _breedScore, _modelName):
    """ Méthode qui affiche à l'écran l'image d'un chien avec l'affichage de sa race la plus probable et de son score
    Arguments:
    _imagePath -- chemin vers l'image
    _size -- la taille que doit prendre l'image, sous format (hauteur, largeur), ex : (224, 224)
    _breedName -- la race la plus probable du chien
    _breedScore -- le score de la race la plus probable du chien
    _modelName -- le nom du modèle
    """
    img = image.load_img(_imagePath, target_size=_size)
    img = image.img_to_array(img)
    plt.text(100, 180, 'SCORE %s: %.2f' % (_modelName , _breedScore), color='w', backgroundcolor='k', alpha=1)
    plt.text(100, 200, 'LABEL: %s' % _breedName, color='w', backgroundcolor='k', alpha=1)
    plt.axis('off')
    plt.imshow(img / 255.)


# Xception étant le modèle le plus prometteur, on peut l'utiliser

localProjectDirectory = "C:/Users/gewurz14/.spyder/notebooks/projet7/"

# Image d'entrée
dogImagesDir = localProjectDirectory+"Images/"
codePathImage = dogImagesDir+"boston13.jpg"
# Choix du modèle
codeChosenModel = "xception"

codeModel = getModel(codeChosenModel, False)
codeInputShape, codePreprocess = getInputShapeAndPreprocessFromTypeOfModel(codeChosenModel)
resultat, probResult = predict5MostLikelyBreedsFromImage(codeModel, codePathImage, codeInputShape, codeChosenModel, codePrepr
ocess, False)

# Affiche de l'image avec le résultat
showExternalImageWithScore(codePathImage, (224,224), resultat, probResult, codeChosenModel)
```

V- Conclusion

Dans la capacité à reconnaître et distinguer des races de chiens dans des images, les réseaux de neurones ont depuis quelques années obtenu de meilleurs résultats que les êtres humains. Cela a été rendu possible grâce à un approfondissement des réseaux et à de nouvelles techniques de communication non linéaire entre couches. L'utilisation des capacités de fonctionnement en parallèle des GPU a aussi permis un net gain de temps et une amélioration des performances de ces réseaux, et donc leur développement et un approfondissement des recherches qui leur sont dédiées.

J'ai donc implémenté quelques modèles allant des plus anciens et des plus sommaires comme le perceptron aux plus récents et performants comme l'Xception. Les recherches sur le sujet allant très rapidement, d'autres modèles encore plus complexes ont été développés, AlexNet, SqueezeNet, Resnet152 etc. mais il était difficile de les implémenter sur ma machine.
Quelques enseignements tirés de mes tests:

- Le principal, les réseaux de neurones possèdent un grand nombre d'hyperparamètres qu'il est difficile de calibrer car de petites modifications de certains peuvent aboutir à des pertes de performance sans qu'il soit aisé d'en connaître la raison.
- Le choix de ces hyperparamètres dépend de son expérience, des bonnes pratiques (rules of thumb) qui sont surtout des règles empiriques qui peuvent marcher pour la plupart des modèles mais souvent avec des exceptions et qui peuvent servir de point de départ à l'expérimentation dans un contexte personnel de projet.
- Malgré la grande difficulté à tirer des conclusions de mes tests, ceux-ci étant limités par les temps de calculs trop importants à chaque test, le fait d'avoir rajouté des couches dans le perceptron n'a pas permis d'améliorer les performances des perceptrons avec peu de couches. Cela rejoint l'idée que sans adaptations supplémentaires (utiliser des convolutions, du pooling, des batchs de normalisation, des modules Inception, du residual learning etc.), il y a une limite à l'amélioration des performances quand on augmente le nombre de couches. Augmenter le nombre de couche doit donc s'accompagner d'une modification de l'architecture du réseau et de la communication entre couches.
- Les courbes d'évolution du coût (loss) et de l'exactitude (accuracy) des prévisions permettent, en temps normal, d'observer un éventuel overfitting (ou même underfitting) et de voir quand arrêter l'entraînement. Cela a bien été le cas lors des tests avec le perceptron multicouche et les données du MNIST. Cela a bien moins été le cas pour ce même modèle appliqué aux données d'ImageNet. J'avoue ne pas avoir bien compris pourquoi, les courbes dénotent un apprentissage quasi-inexistant (courbe constante) et de très mauvais résultats. Cela ne semble pas dû au préprocessing des données car c'est à peu près le même que celui utilisé pour les modèles pré-entraînés or ils ont bien donné de bons résultats avec les mêmes images à classer. Peut-être est-ce dû au fait que je n'ai pas utilisé beaucoup de catégories, 10 sur les 120 disponibles et que les itérations d'entraînement n'étaient pas suffisantes. Cela a été également le cas avec le modèle convolutif qui était encore plus gourmand en temps de calculs.
- Les modèles pré-entraînés ont heureusement permis de s'affranchir de ces contraintes de recherche des poids et biais. J'ai pu observer qu'effectivement, plus les modèles possédaient un grand nombre de paramètres (par exemple VGG16), plus le téléchargement des poids prenait du temps (VGG16 a mis deux heures). Mais une fois ce téléchargement effectué, l'exécution du modèle était assez rapide.
- Finalement, j'ai retenu le modèle Xception qui est assez rapide et qui semble donner les meilleurs résultats (en concordance avec la hiérarchie qu'on peut trouver dans les articles sur Internet).

Voici quelques pistes d'améliorations possibles:

- Récupérer les attributs des images pour tenter de voir dans quels cas la classification ne marche pas, par exemple avec les attributs téléchargeables sur : http://image-net.org/download-attributes
- Utiliser les positions précises des chiens à l'intérieur des images que l'on peut trouver dans les annotations (sous la balise bndbox) afin de supprimer le "bruit" extérieur. Toutefois, la restriction de l'image entière à celle du chien seul n'est avantageuse que si on propose au modèle de nouvelles images ne contenant également que le chien. Une nouvelle image avec d'autres personnages que le chien poserait problème dans le sens où le modèle n'aurait pas été entraîné à apprendre des "compositions". Cela peut aussi poser des problèmes lors de l'utilisation des modèles pré-entraînés qui ont été entraînés avec la totalité des images et non juste le chien.
- Faire des statistiques pour regarder parmi les erreurs de classification, quelles sont les races les plus impactées, et si par exemple cela a un rapport avec le nombre de chiens de ces catégories.
- Ne pas se servir des données de test comme validation, mais réserver par exemple 10% des données d'entraînement (validation_split=0.1 dans la méthode model.fit), puis faire l'évaluation sur les données de test. (Dans notre contexte, les données d'entraînement / validation / test provenant toutes du même échantillon, cela n'est pas trop gênant et cela m'a permis de suivre l'historique des données de test pour les courbes de loss/accuracy. Mais dans un contexte plus général, on devrait bien séparer validation et test.) Faire une cross-validation.
- Faire du transfer learning. Même si dans notre contexte, cela a peut-être un intérêt limité du fait que toutes les images d'entraînement et de test proviennent de la même source, ImageNet, utilisée par les modèles pré-entraînés, il pourrait être intéressant d'utiliser les paramètres pré-entraînés sur la totalité de la base de données, puis "étêter" le modèle (ne pas utiliser sa partie top) et remplacer les dernières couches de la pile par un autre modèle qui s'adapterait à notre projet particulier afin de calibrer plus finement les paramètres spécifiquement au context local plutôt qu'à l'enseignement général de la tâche précédente, c'est-à-dire du modèle précédent. On devrait ainsi avoir un gain de temps mais sans forcément perdre beaucoup de performance relativement à une version développée entièrement sur notre contexte local.

- [1] https://arxiv.org/abs/1207.0580 (G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, R. R. Salakhutdinov, 2012)
- [2] http://cs231n.github.io/neural-networks-1/ (Stanford, 2017)
- [3] http://cs231n.github.io/neural-networks-2/ (Stanford, 2017)
- [4] http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/ (Stanford, 2017)
- [5] http://neuralnetworksanddeeplearning.com/chap3.html (Michael Nielsen, 2015)
- [6] http://cs231n.github.io/neural-networks-3/ (Stanford, 2017)
- [7] http://cs231n.github.io/convolutional-networks/ (Stanford, 2017)
- [8] https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html (Petar Veličković, Cambridge, 2017)
- [9] http://www.shortscience.org/paper?bibtexKey=journals/corr/1610.02357#qureai (François Chollet, 2016)