Dog Breed Classifier with PyTorch, by André Vargas

Project Overview:

- What is Machine Learning?

On modern days, the algorithms are all around us. Language assistants that instantly translates from one language to another, self-driving cars, personalized movie suggestions in your favourite streaming platform, personalized shopping suggestions and automatic price setting, diagnosis of a disease based on collected data. All these tasks that a few years ago would be science fiction are possible today thanks to Machine Learning algorithms. A technology that allow computers and other devices to learn and solve different tasks by themselves without explicitly programming them how to do so.

The Collins English Dictionary defines Intelligence as “the ability to think, reason, and understand instead of doing things automatically or by instinct” [1]. For example, before lifting a vase we are able to predict its weight based on our intelligence, and know whether or not we are able to do it. This reasoning to predict certain situations is possible because of our memory from previous experiences, and it’s exactly the same principles applied to Machine Learning.

Computers can learn trough associations of different types of data, which can be numbers, images, patterns, classes, and so on. And beyond a massive amount of problems to solve, image classification is one of the biggest areas of study in the domain of Machine Learning.

- What is Image Classification?

There are hundreds of applications for image classification with Machine Learning. For example, let’s say we want a machine to analyse pictures of recent harvested strawberries to see which of them are good or bad for humans to eat, or let's imagine a doctor that uses a software capable of looking at an x-ray image and labeling as cancer or not.

Regardless of being a simple classification problem with two categories, good strawberries and bad strawberries, or a more complex one, they are easy problems for humans simply because we just know in our brain how a good strawberry looks and how a bad strawberry looks, but it’s impossible to tell a computer how to do it. And one of the solutions is to use Machine Learning algorithms to teach the machines how to identify specific patterns in these images that allow them to be classified.

Image 1: A bad and a good strawberry [2]

Problem Statement:

The goal of this project is to use an Image Classification algorithm to identify dog breeds from image inputs. This is a very complex problem because dogs are very similar among them in therms of body features and overall structure.

Dogs are one of the most diverse species on the planet, there are over 340 dog breeds known around the world. The American Kennel Club recognize over 170 official dog breeds, not counting mixed breeds and mutts.[3] Differentiating between all these categories is not an easy task for a machine to do, and a very large dataset is required.

Image 2: Can you tell which one is a German Shepherd Dog and which one is a Belgian Malinois? [4]

The basic premise of this project is that given an image of a dog, the algorithm must identify an estimate of the canine's breed. If supplied an image of a human, the code will identify the resembling dog breed. Pretty straight forward. We will create a small web application running in Flask that asks an input image from the user and returns the breed contained in that image.

This is the final project for Udacity's Machine Learning Engineer Nanodegree and all steps to accomplish this task will be covered on this report.

Data Exploration

Before discussing any algorithm possibilities, let's take a look at the dataset available for this project. Since it's an image classification problem, it's not surpring that the dataset is made of lots and lots of images. We will be using two datasets to accomplish this project. Both of them were provided by Udacity and can be downloaded in the links below.

- Dog Dataset:

There are a total of 8351 dog images separated in train (6680 images), test (836 images) and valid (835 images), and each group separated in 133 different dog breeds. The dataset can be downloaded here. It's interesting to note how balanced the classes are within each sub-dataset. This will certainly help to improve the algorithm results.

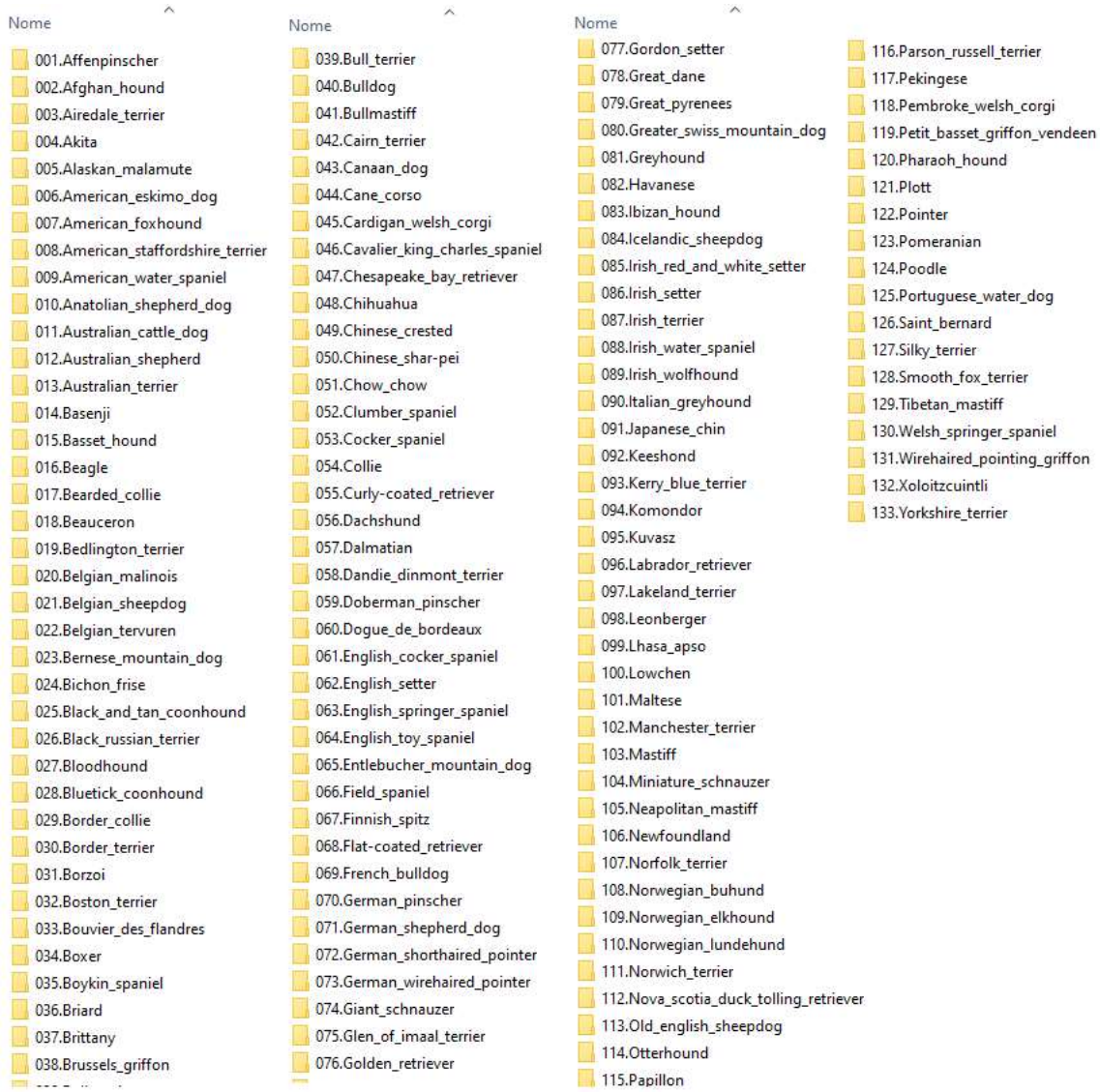


Image 3: All 133 classes available in the dataset.

Let's take a look at some examples from the dog dataset:

Image 4: Classes 072.German_shorthaired_pointer and 099.Lhasa_apso.

- Human Dataset:

And since this project should be able to identify and give a breed to a human image, there are 13233 human images as well which will be used for testing. This dataset can be downloaded here.

Some examples from the human dataset:

Image 5: Harrison Ford and Roger Federer.

Algorithms and Techniques

There are several Machine Learning algorithms to deal with image processing and image classification, but considering the fact that we are expecting to classify dogs with several specificities, a good approach is to use one that identify specific patterns in these images that allow them to be classified. One way to do this job is using a Convolutional Neural Network (CNN).

- What is a Convolutional Neural Network?

A CNN is a specific type of neural network and its largely used for image classification. To understand a little bit better how a CNN works, it helps if we look at how the computers "see" an image.

A black and white image is seen by the computer as a 2D matrix, and each position of this matrix represents a pixel of this image. The values for each element vary between 0 (black) and 255 (white). This can be seen in the example below:

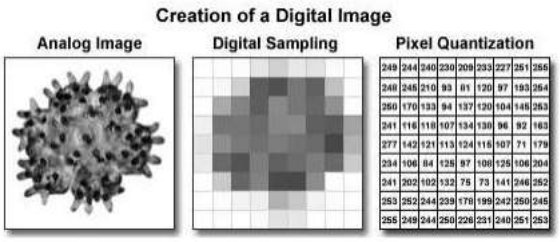


Image 6: Creation of a Digital Image [5]

A coloured image, on the other hand, is represented by a 3D matrix in which we can store a combination of the colours red, green and blue (RGB) from 0 to 255. The image below in an example of that:

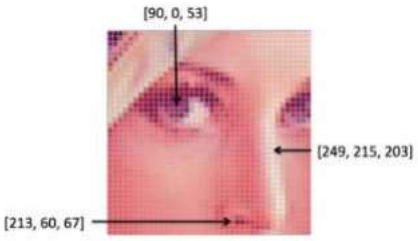


Image 7: Digitalization of a coloured picture [6]

So, considering that a machine can only see groups of numbers, one way to analyse and determine whether that image is a "cat" or a "dog" is by looking for specific patterns and characteristics for each image class. These operation of "reading" the matrices showed above, assign some weights and biases to some aspects of the image, and based on that differentiate the images is the basic premise of a CNN. [7]

This is obviously an oversimplification, there is a lot of complexity to add to it, but essentially a CNN can be seen as a class of neural networks in which a series of operations are performed from one layer to another until an output is given. They extract features from images.

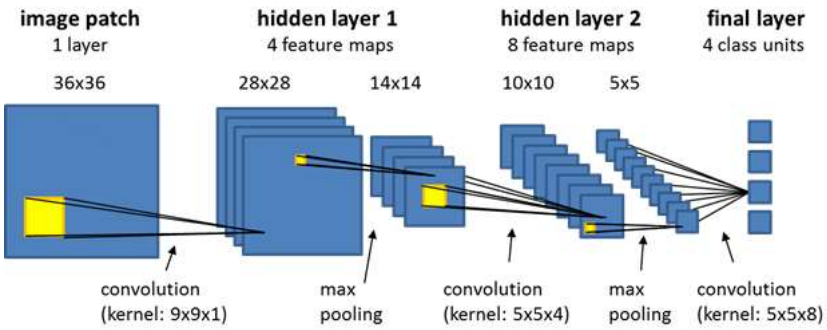


Image 8: Basic CNN scheme [8]

CNNs are also used in a variety of other fields like face recognition, document analysis, understanding climate changes, speech recognition, and so on. [9]

In order to create and test our CNNs, we will be using PyTorch. PyTorch is a Python-based library that provides functionalities such as:

- Creating serializable and optimizable models;
- Distributed training to parallelize computations;
- Dynamic Computation graphs which enable to make the computation graphs on the go, and many more.

PyTorch tensors in general are similar to NumPy's n-dimensional arrays which can also be used with GPUs. Performing operations on these tensors is almost similar to performing operations on NumPy arrays. This makes PyTorch very user-friendly and easy to learn.

Benchmark

In this project, we will test two different CNNs: the first one created from scratch, making the choices for each layer, each configuration, and defining the behaviour of the forward function, and the second CNN will be created using transfer learning. Transfer learning is a very simple concept: A model trained on a large dataset has its knowledge transfered to a smaller dataset. This allow us to use a network trained in a massive dataset and apply it to our problem, expecting a better performance.

So, given this idea, our benchmark model will be the CNN created from scratch, and our goal is to achieve an accuracy of 40%.

As for the transfer learning CNN, the goal is to achieve an accuracy of at least 85%, and it will be the model tested and used on the web application.

Implementation

We will split the implementation of this project into 7 separated steps:

Step 1) Import the datasets;
Step 2) Create a human face detector;
Step 3) Create a dog detector;
Step 4) Preprocess the data;
Step 5) Create a CNN to classify dog breeds from scratch;
Step 6) Create a CNN to classify dog breeds using transfer learning;
Step 7) Write the full algorithm;

Let's go through all these steps, in detail, one by one.

- Step 0: Import the datasets

After downloading both datasets from the links provided before, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays human_files and dog_files. This will make testing much easier later.

In [1]:
```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Image 9: Importing Libraries

- Step 1: Create a human face detector

Since we are supposed to identify when a human is present in the image, we can use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors stored as XML files on GitHub. We have downloaded one of these detectors and are going to use on this project. The XML file can be downloaded from this project repository on Github.

Let's take a look on how to use this detector to find human faces in a sample image.

In [2]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# load color (BGR) image
img = cv2.imread(human_files[0])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```

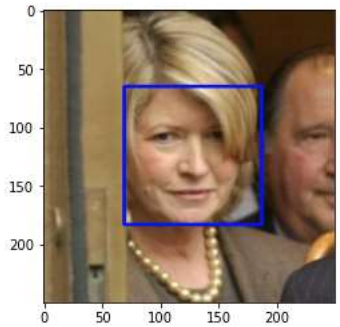


Image 10: Detecting a face with OpenCV

In the above code, faces is a numpy array of detected faces, where each row corresponds to a detected face. Which means that if we check the lenght of faces, we can tell if a face has been detected or not, in other words, a function named face_detector that takes a string-valued file path to an image as input and returns True or False, depending on the result.

In [4]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

Image 11: Face detector function

Now, as a side mission, let's test the performance of the face_detector function. Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. The algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays human_files_short and dog_files_short.

In [5]:
```
from tqdm import tqdm

human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

#-#-# Do NOT modify the code above this line. #-#-#

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.

# Passing a numpy array to a function
# Source: https://stackoverflow.com/questions/52118745/pass-numpy-array-as-function-argument

# Test on human_files_short
humans = np.array([face_detector(file) for file in human_files_short])
perc_h = 100*np.sum(humans)/len(human_files_short)

# Test on dog_files_short
dogs = np.array([face_detector(file) for file in dog_files_short])
perc_d = 100*np.sum(dogs)/len(dog_files_short)

# Results
print('Percentage of human faces detected in the first 100 images in human_files: {}'.format(perc_h))
print('Percentage of human faces detected in the first 100 images in dog_files: {}'.format(perc_d))
```

```
Percentage of human faces detected in the first 100 images in human_files: 99.0
Percentage of human faces detected in the first 100 images in dog_files: 12.0
```

Image 12: Face detector performance

As we've already mentioned, it's not perfect, and it’s interesting trying to understand why. Some of pictures from the dog dataset have humans in it, so the function will return true because it’s not necessarily a dog only image. Like this example:

Image 13: From the dataset "dog_images/train/103.Mastiff"

In other cases, the algorithm will simply identify some human features in specific patterns of the image and will consider as a face.

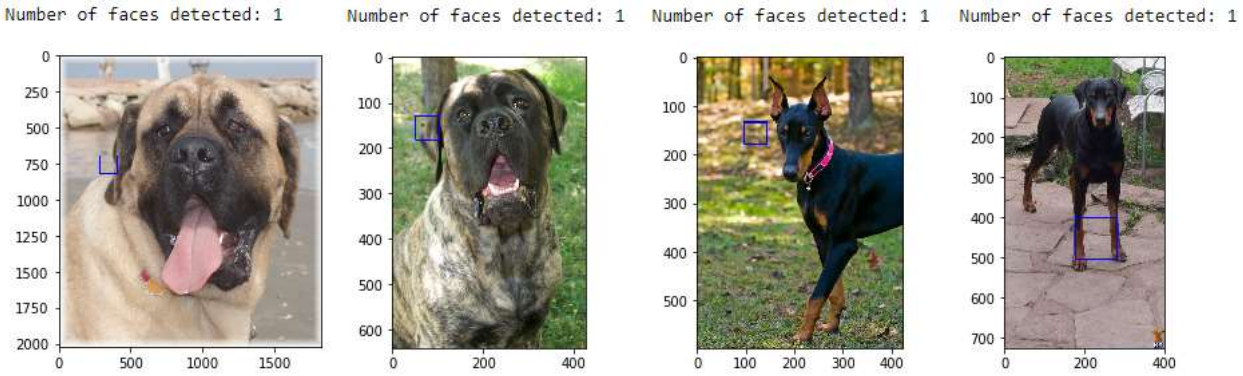


Image 14: From the dataset

And in the human dataset, there are a few images with lower quality that the classifier was unable to understand:

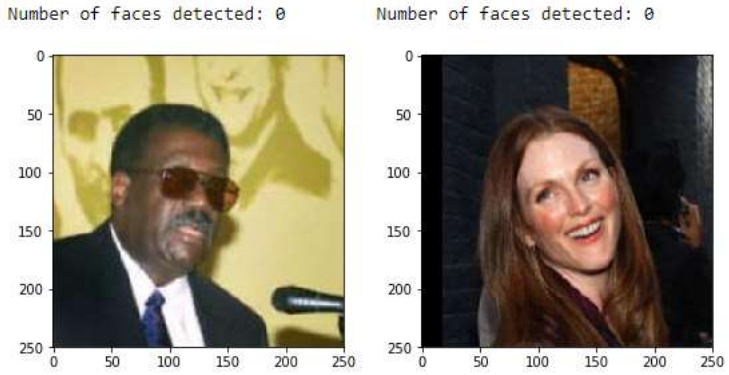


Image 15: From the dataset

But still, the results were very acceptable.

There are more face algorithms out there, and we can take a look at another one to compare the results with OpenCV. Let's check the performance of the Multi-Task Cascaded Convolutional Neural Network (MTCNN)[10], a CNN by itself.

MTCNN stores the attributes of any faces detected in a list, as it can be seen from the example below, which has 8 faces.

In [8]:
```
from mtcnn.mtcnn import MTCNN

filename = "images/group_of_people.jpg"

# draw an image with detected objects
def draw_facebox(filename, result_list):
    # load the image
    # plot the image
    plt.imshow(data)
    # get the context for drawing boxes
    ax = plt.gca()
    # plot each box
    # for each face, draw a rectangle based on coordinates
    for face in result_list:
        x, y, width, height = face['box']
        face_border = plt.Rectangle((x, y), width, height, fill=False, color='red')
        ax.add_patch(face_border)
    plt.show()

data = plt.imread(filename)
detector = mtcnn.MTCNN()
faces = detector.detect_faces(data)
```

In [9]:
```
print("Attributes:\n")
print(faces)
print("\nFaces highlighted:")
draw_facebox(filename, faces)
```

```
Attributes:

[{'box': [390, 59, 49, 65], 'confidence': 1.0, 'keypoints': {'left_eye': (405, 81), 'right_eye': (428, 84), 'nose': (417, 94),
'mouth_left': (402, 104), 'mouth_right': (425, 107)}}, {'box': [146, 105, 42, 57], 'confidence': 0.9999904632568359, 'keypoint
s': {'left_eye': (157, 126), 'right_eye': (177, 128), 'nose': (167, 138), 'mouth_left': (156, 145), 'mouth_right': (176, 14
6)}}, {'box': [177, 56, 44, 55], 'confidence': 0.9999825954437256, 'keypoints': {'left_eye': (191, 76), 'right_eye': (211, 77),
'nose': (202, 88), 'mouth_left': (190, 95), 'mouth_right': (211, 96)}}, {'box': [36, 34, 46, 62], 'confidence': 0.9999793767929
077, 'keypoints': {'left_eye': (53, 58), 'right_eye': (74, 60), 'nose': (64, 71), 'mouth_left': (49, 76), 'mouth_right': (72, 7
9)}}, {'box': [251, 84, 43, 55], 'confidence': 0.999962329864502, 'keypoints': {'left_eye': (259, 108), 'right_eye': (277, 10
3), 'nose': (268, 117), 'mouth_left': (263, 126), 'mouth_right': (282, 121)}}, {'box': [84, 120, 47, 64], 'confidence': 0.99990
51094055176, 'keypoints': {'left_eye': (103, 147), 'right_eye': (123, 145), 'nose': (118, 158), 'mouth_left': (103, 167), 'mout
h_right': (125, 165)}}, {'box': [490, 100, 45, 65], 'confidence': 0.9997772574424744, 'keypoints': {'left_eye': (499, 123), 'ri
ght_eye': (520, 125), 'nose': (506, 136), 'mouth_left': (497, 145), 'mouth_right': (520, 147)}}, {'box': [321, 37, 43, 56], 'co
nfidence': 0.9984403252601624, 'keypoints': {'left_eye': (333, 56), 'right_eye': (353, 55), 'nose': (342, 62), 'mouth_left': (3
31, 76), 'mouth_right': (354, 76)}}]

Faces highlighted:
```

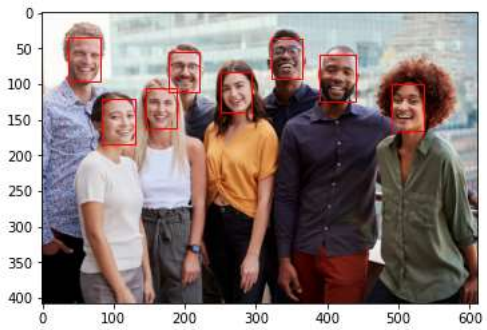


Image 16: MTCNN detection

So, using the same logic from our face_detector function, a face_detector using MTCNN could be done by reading the result list after looking an image, if there are any attributes stored, there is a face detected.

In [10]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector_MTCNN(img_path):
    data = plt.imread(img_path)
    detector = mtcnn.MTCNN()
    faces = detector.detect_faces(data)
    if len(faces) > 0:
        return 1
    else:
        return 0
```

In [12]:
```
# Testing face_detector_MTCNN in a human image
is_human = face_detector_MTCNN(human_files[0])

# Testing face_detector_MTCNN in a dog image
is_dog = face_detector_MTCNN(dog_files[0])

print('Face detected in a human image: {}'.format(is_human))
print('Face detected in a dog image: {}'.format(is_dog))
```

```
Face detected in a human image: 1
Face detected in a dog image: 0
```

Image 17: MTCNN face_detector function

And again, running in a sample of both datasets, we can see how well this model performs.

In [13]:
```
# Test on human_files_short
MTCNNhumans = np.array([face_detector_MTCNN(file) for file in human_files_short])
MTCNNperc_h = 100*np.sum(MTCNNhumans)/len(human_files_short)

# Test on dog_files_short
MTCNNdogs = np.array([face_detector_MTCNN(file) for file in dog_files_short])
MTCNNperc_d = 100*np.sum(MTCNNdogs)/len(dog_files_short)

# Results
print('Percentage of human faces detected in the first 100 images in human_files: {}'.format(MTCNNperc_h))
print('Percentage of human faces detected in the first 100 images in dog_files: {}'.format(MTCNNperc_d))
```

```
Percentage of human faces detected in the first 100 images in human_files: 100.0
Percentage of human faces detected in the first 100 images in dog_files: 27.0
```

Image 18: MTCNN performance

MTCNN was able to identify all humans in the human dataset sample, but found more faces in the dog dataset than the OpenCV method.

Algorithm	Image Type	%
OpenCV	Humans	99
OpenCV	Dogs	12
MTCNN	Humans	100
MTCNN	Dogs	27

Image 19: Comparative Table

This concludes Step 2 of this project, we will be using OpenCV's face_detector function later in the final code.

- Step 3: Create a dog detector

In this section, we will use a pre-trained model to detect dogs in images.

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

In [16]:
```
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()

# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /home/ec2-user/.cache/torch/hub/checkpoints/vgg16-3979
23af.pth
```

Image 20: Downloading VGG-16 model

The ideia is simple, given an image, this pre-trained VGG-16 model returns a prediction (an integer from 0 to 999 derived from the 1000 possible categories in ImageNet) for the object that is contained in the image. If we take a look at ImageNet categories, we'll see that categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from 'Chihuahua' to 'Mexican hairless'. Thus, to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

In order to write the dog_detector function, it's important to take a look at PyTorch's documentation to understand how to appropriately pre-process tensors for pre-trained models

In the code cell bellow, we have a little spoiler of what kind of preprocessing we're gonna have to do to our dataset when we train the model. We'll discuss these steps with more detail on Step 4. The relevant information is that the function reads an image from its path, preprocess it, convert to a tensor accepted by the model, and returns the index prediction of what that image is. We tested on a golden retriever image and got the result 207, which is golden retriever in ImageNet classification.

In [17]:
```python
from PIL import Image
import torchvision.transforms as transforms

from torch.autograd import Variable

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

# Predict function
def VGG16_predict(img_path):

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    # Load the image
    img = Image.open(img_path)

    # Normalize image
    normalize = transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                     std=[0.5, 0.5, 0.5])

    # Resize to a size accepted by VGG16 model
    preprocess = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     normalize])

    # Image tensor
    img_tensor = preprocess(img).float()

    # Add a new dimention
    img_tensor.unsqueeze_(0)

    # Convert tensor to a Pytorch Variable
    img_tensor = Variable(img_tensor)

    # Check cuda availability
    if use_cuda:
        img_tensor = Variable(img_tensor.cuda())

    VGG16.eval()

    # Output tensor
    output = VGG16(img_tensor)
    output = output.cpu()

    # Get the class label with the biggest value
    predict_index = output.data.numpy().argmax()

    # Return predicted index
    return predict_index
```

```
205: 'flat-coated retriever',
206: 'curly-coated retriever',
207: 'golden retriever',
208: 'Labrador retriever',
209: 'Chesapeake Bay retriever',
```

Image 21: VGG-16 prediction test

Now with the VGG16_predict funcion, to wrap up our dog_detector, all we have to do is create a logic that gives True if the predicted index is between 151 and 268 (inclusive), and False for every other value.

In [19]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.

    index = VGG16_predict(img_path)

    if ((index >= 151) & (index <= 268)):
        # True -> Dog class found
        return 1

    else:
        # False -> Didn't found a dog class
        return 0
```

In [20]:
```
# Test on a dog image:
is_dog = dog_detector(dog_files[2])

# Test on a human image:
is_human = dog_detector(human_files[2])

print('Dog class detected in a dog image: {}'.format(is_dog))
print('Dog class detected in a human image: {}'.format(is_human))
```

```
Dog class detected in a dog image: 1
Dog class detected in a human image: 0
```

Image 22: dog_detector function test

Now we can test the performance in the first 100 images of each dataset, like we did with the face_detector function.

In [13]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

# Test on human_files_short
DDhumans = np.array([dog_detector(file) for file in human_files_short])
ddperc_h = 100*np.sum(DDhumans)/len(human_files_short)

# Test on dog_files_short
DDdogs = np.array([dog_detector(file) for file in dog_files_short])
ddperc_d = 100*np.sum(DDdogs)/len(dog_files_short)

print('Percentage of human faces detected in the first 100 images in human_files: {}'.format(ddperc_h))
print('Percentage of human faces detected in the first 100 images in dog_files: {}'.format(ddperc_d))
```

```
Percentage of human faces detected in the first 100 images in human_files: 0.0
Percentage of human faces detected in the first 100 images in dog_files: 100.0
```

It works perfectly.

As an exercise, let's try a different dog detector using the ResNet50 pre-trained model[11].

In [23]:
```
# Source code: https://towardsdatascience.com/dog-breed-classification-using-cnns-and-transfer-learning-e36259b29925

from keras.applications.resnet import ResNet50, preprocess_input
from keras.preprocessing.image import load_img, img_to_array

def dog_detector_RESNET50(img_path):

    # Create a Redidual-network already trained in the IMAGENET
    ResNet50_model = ResNet50(weights='imagenet')

    # Load the image in the size expected by the ResNet50_model
    img = load_img(img_path, target_size=(224, 224))

    # Transform the image into an array
    preimg_array = img_to_array(img)
    img_array = np.expand_dims(preimg_array, axis=0)

    # Pre-process the image according to IMAGENET standarts
    img_ready = preprocess_input(img_array)

    # Predicts
    probs = ResNet50_model.predict(img_ready)

    # Find the position with the maximum probability value
    position_of_max = np.argmax(probs)

    # Verify if the position of max corresponds to a dog class
    dog_class = ((position_of_max >= 151) & (position_of_max <= 268))

    if dog_class == True:
        return 1
    else:
        return 0
```

In [25]:
```
# Testing the function on individual images

is_dog = dog_detector_RESNET50(dog_files[4])
is_human = dog_detector_RESNET50(human_files[5])

print('ResNet50 Dog_detector() output from a dog image: {}'.format(is_dog))
print('ResNet50 Dog_detector() output from a human image: {}'.format(is_human))
```

```
ResNet50 Dog_detector() output from a dog image: 1
ResNet50 Dog_detector() output from a human image: 0
```

Image 23: dog_detector function using ResNet50

We can test its performance and compare to VGG-16.

In [26]:
```
### Testing the performance of the ResNet50 dog_detector function on the images in human_files_short and dog_files_short.

# Human_files_short
RNhumans = np.array([dog_detector_RESNET50(file) for file in human_files_short])
RNperc_h = 100*np.sum(RNhumans)/len(human_files_short)

# Test on dog_files_short

RNdogs = np.array([dog_detector_RESNET50(file) for file in dog_files_short])
RNperc_d = 100*np.sum(RNdogs)/len(dog_files_short)

print('Percentage of dogs detected in the images in dog_files_short: {}'.format(RNperc_d))
print('Percentage of dogs detected in the images in human_files_short: {}'.format(RNperc_h))
```

```
Percentage of dogs detected in the images in dog_files_short: 100.0
Percentage of dogs detected in the images in human_files_short: 0.0
```

Image 23: ResNet50 dog_detector performance

Model	Image Type	% of dog faces
VGG16	Humans	0
VGG16	Dogs	100
RESNET50	Humans	0
RESNET50	Dogs	100

Image 24: Comparative Table

ResNet50 also works perfectly to our needs, but it's a lot slower than VGG-16. Therefore, we'll keep our dog_detector function with VGG-16, which will be used later in the final algorithm. This concludes Step 3 of this project.

- Step 4: Preprocess the data

The first thing we need to do is to write three separate data loaders for the training, validation, and test datasets of dog images. A Data loader combines a dataset and a sampler, and provides an iterable over the given dataset. This is where the preprocessing of the data takes place.

We've already had a glimpse of what needs to be done on this step. On the definition of VGG16_predict function, we needed to do some processing of the image in order to convert to an acceptable tensor, so the pre-trained model could understand and give a valid output. This is exactly what we're going to do to create our dataloaders.

Before we start going through this process, we need to introduce the concept of data augmentation.

When training a machine learning model, what is happening behind the curtains is a tuning of its parameters in a way that it can map a particular input (an image of a dog) to some output (its breed). The optimization goal is to chase that sweet spot where the model's loss is low, which happens the parameters are tuned in the right way. But if there are a lot of parameters, we would need to show the machine learning model a proportional amount of examples to get good performance. Also, the number of parameters needed is proportional to the complexity of the task the model has to perform.

But do we get more data, if I don't have "more data"? This is when data augmentation shows up. It's not necessary to hunt for novel new images that can be added to your dataset because neural networks understands minor alterations to an image such as flips, translations or rotations as a totally new and unique image. So, to get more data, we just need to make small changes to our existing dataset. That is the basic idea behind data augmentation.

Luckily, PyTorch has a model called transforms, which allows us to perform different transformations to our images. By using the class transforms.Compose, we can chain several transforms together. So now, let's finally discuss how we're going to preprocess our dataset.

For the train dataset, we are going to chain these transformations:
- Random Resized Crop(224) - Crop a random portion of image and resize it to 224x224, which is the size of the tensor accepted by the model;
- Random Horizontal Flip() - Horizontally flip the given image randomly with a given probability. This was a choice made to augment our data.
- Random Rotation (15) - Rotate the image by angle of 15°. Another choice made to augment the data.
- To Tensor() - Convert a PIL Image or numpy.ndarray to tensor.
- Normalize a float tensor image with mean and standard deviation.

For the valid dataset, we are going to chain these transformations:
- Random Resize(256) - Resize the input image to the given size. - Center Crop (224) - Crops the given image at the center with the given size (224 is the size of tensor accepted by the model).
- To Tensor() - Convert a PIL Image or numpy.ndarray to tensor.
- Normalize a float tensor image with mean and standard deviation.

For the test dataset, we are going to chain these transformations:
- Random Resize(256) - Resize the input image to the given size. - Center Crop (224) - Crops the given image at the center with the given size (224 is the size of tensor accepted by the model).
- To Tensor() - Convert a PIL Image or numpy.ndarray to tensor.
- Normalize a float tensor image with mean and standard deviation.

To put all together, let's define a dictionary with all the transformations:

In [28]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

# Check cuda availability
use_cuda = torch.cuda.is_available()

# Data preprocess dict
data_preprocess  = {

    # Source: https://pytorch.org/vision/stable/transforms.html
    #
    # Train dataset:
    # Transforms Compose - Composes several transforms together
    # Source: https://pytorch.org/vision/stable/transforms.html
    #
    # Random Resized Crop(224) - Crop a random portion of image and resize it to 224x224
    # Random Horizontal Flip() - Horizontally flip the given image randomly with a given probability (Data augmentation)
    # Random Rotation (15) - Rotate the image by angle of 15° (Data augmentation)
    # To Tensor() - Convert a PIL Image or numpy.ndarray to tensor
    # Normalize a float tensor image with mean and standard deviation
    #
    # What numbers select to transforms.Normalize?
    # Source: https://stackoverflow.com/questions/65467621/what-are-the-numbers-in-torch-transforms-normalize-and-how-to-select-

    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(15),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ]),

    # Valid dataset:
    # Transforms Compose - Composes several transforms together
    #
    # Random Resize(256) - Resize the input image to the given size
    # Center Crop (224) - Crops the given image at the center with the given size
    # To Tensor() - Convert a PIL Image or numpy.ndarray to tensor
    # Normalize a float tensor image with mean and standard deviation

    'valid': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ]),

    # Test dataset:
    # Transforms Compose - Composes several transforms together
    #
    # Random Resize(256) - Resize the input image to the given size
    # Center Crop (224) - Crops the given image at the center with the given size
    # To Tensor() - Convert a PIL Image or numpy.ndarray to tensor
    # Normalize a float tensor image with mean and standard deviation

    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ]),

}
```

Image 25: Transforms Dictionary

We finish it up by loading all the preprocessed images using torch.utils.data.DataLoader class, an iterable over a dataset. This way, we can fill our train, valid and load datasets.

```
path = "dogImages/"

# Load all pre-processed images
images = {x: datasets.ImageFolder(os.path.join(path, x), data_preprocess[x])
          for x in ['train', 'valid', 'test']}

# PyTorch data loading utility has the torch.utils.data.DataLoader class.
# It represents a Python iterable over a dataset, and it's used to iterate over a dataset
# Source: https://pytorch.org/docs/stable/data.html
dataloaders = {x: torch.utils.data.DataLoader(images[x], batch_size = 20, shuffle = True, num_workers = 0)
               for x in ['train', 'valid', 'test']}

# Lenght of the datasets (train, valid and test)
sizes = {x: len(images[x]) for x in ['train', 'valid', 'test']}

print('Train images: {}'.format(sizes['train']))
print('Valid images: {}'.format(sizes['valid']))
print('Test images: {}'.format(sizes['test']))
```

```
Train images: 6680
Valid images: 835
Test images: 836
```

Image 26: Dataloaders

Now the dataloaders are ready with all the images preprocessed, and we have access to the names of all dog classes within the dataset. It feels like all this work so far is starting to get us somewhere.

In [29]:
```
# Get all class names (from train class)
names = images['train'].classes
clean_names = []

# Get rid of the numbers and the "_"
for name in names:
    clean_names.append(name[4:].replace('_', ' '))

# Number of classes
n_classes = len(clean_names)

# Print Sample of classes:
for i in range(5):
    print (clean_names[i])
```

```
Affenpinscher
Afghan hound
Airedale terrier
Akita
Alaskan malamute
```

Image 27: Classes sample

- Step 5: Create a CNN to classify dog breeds from scratch

To create the CNN, first we need to define the Constructor, the basic components of the network, which consists of several layers: the convolutional layers, the pooling layer, the dropout layer, and the full connected layers. So we will define a class which includes the methods defining all the components of a CNN network.

In [30]:
```
import torch.nn as nn
import torch.nn.functional as F

# Source: https://pytorch.org/docs/stable/generated/torch.nn.Module.html

# define the CNN architecture
class Net(nn.Module):

    ### TODO: choose an architecture, and complete the class

    # Definition of the constructor
    def __init__(self):

        super(Net, self).__init__()

        # Convolutional Layers
        # Sources: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html
        #          https://androidkt.com/use-the-batchnorm-layer-in-pytorch/
        #
        # 5 Convolutional Layers and 6 BatchNorm Layers
        self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
        self.conv_bn1 = nn.BatchNorm2d(224,3)

        self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
        self.conv_bn2 = nn.BatchNorm2d(16)

        self.conv3 = nn.Conv2d(32, 64, 3, padding = 1)
        self.conv_bn3 = nn.BatchNorm2d(32)

        self.conv4 = nn.Conv2d(64, 128, 3, padding = 1)
        self.conv_bn4 = nn.BatchNorm2d(64)

        self.conv5 = nn.Conv2d(128, 256, 3, padding = 1)
        self.conv_bn5 = nn.BatchNorm2d(128)

        self.conv_bn6 = nn.BatchNorm2d(256)
        # Pooling Layer
        # Source: https://deeplizard.com/learn/video/ZjM_XQa5s6s
        #
        # Max pooling is a type of operation that is typically added to CNNs following individual convolutional layers.
        # When added to a model, max pooling reduces the dimensionality of images by reducing the number of pixels
        # in the output from the previous convolutional layer.
        self.pool = nn.MaxPool2d(2, 2)

        # Dropout Layer
        # Source: https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html
        #
        # Dropout is a technique used to prevent a model from overfitting
        self.dropout = nn.Dropout(0.5)

        # Fully Connected Layers
        # Sources: https://stackoverflow.com/questions/56660546/how-to-select-parameters-for-nn-linear-layer-while-training-a-cn
        #          https://pythonprogramming.net/building-deep-learning-neural-network-pytorch/
        #
        # Fully Connected layers in a neural networks are those layers where all the inputs from one layer are connected
        # to every activation unit of the next layer
        self.fc1 = nn.Linear(256*7*7, 512)
        self.fc2 = nn.Linear(512, n_classes)
```

Image 28: CNN Constructor

We are using:

Convolutional Layers - Five layers defined with respective 16, 32, 64, 128 and 256 filters, and all of them have kernel size = 3, stride = 1 and padding = 1.
Pooling Layer - I picked method MaxPooling2D for the pooling layer. When added to a model, max pooling reduces the dimensionality of images by reducing the number of pixels in the output from the previous convolutional layer.
Dropout Layer - Dropout Layer with a probability of 2% to prevent overfitting.
Fully Connected Layers - 2 fully connected layers, the first has a 256 input, and output to 512, which is the input to the second FC layer, and its output is the number of classes n_classes.

And second, we need to define the forward function, which is how the model is going to be run, from input to output.

```
    # Definition of the forward function
    def forward(self, x):

        # Apply the activation function (relu) to each layer
        # Relu Conv Layer 1
        x = self.pool(F.relu(self.conv1(x)))

        # Relu Conv Layer 2
        x = self.conv_bn2(x)
        x = self.pool(F.relu(self.conv2(x)))

        # Relu Conv Layer 3
        x = self.conv_bn3(x)
        x = self.pool(F.relu(self.conv3(x)))

        # Relu Conv Layer 4
        x = self.conv_bn4(x)
        x = self.pool(F.relu(self.conv4(x)))

        # Relu Conv Layer 5
        x = self.conv_bn5(x)
        x = self.pool(F.relu(self.conv5(x)))

        x = self.conv_bn6(x)

        # Apply view function
        # Avoids explicit data copy, thus allows us to do fast and memory efficient reshaping,
        # slicing and element-wise operations.
        # Source: https://pytorch.org/docs/stable/tensor_view.html
        x = x.view(-1, 256 * 7 * 7)

        # Add dropout layer
        x = self.dropout(x)

        # Pass the first fully connected layer forward to the second
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        # Return output
        return x
```

Image 29: CNN Forward function

The forward function works as follows:

A Relu activation function is applied to each layer to make the network non-linear.
The view function is applyed to prevent explicit data copy.
The FC layers are connected.

Now we need to specify the loss function and optimizer:

In [31]:
```
import torch.optim as optim

### TODO: select loss function
# Source: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html#torch.nn.CrossEntropyLoss
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
# Source: https://pytorch.org/docs/stable/optim.html
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001, momentum=0.9)
```

Image 30: Loss function and optimizer

And now we train and validate the model using the train function below:

In [32]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
import time
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    since = time.time()

    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0
```

```
        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()

            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

            # zero grad
            optimizer.zero_grad()

            # forward pass
            output = model(data)

            # batch loss
            loss = criterion(output, target)

            # backward pass
            loss.backward()

            # optimization step
            optimizer.step()

            # training loss
            train_loss = train_loss + (1 / (batch_idx + 1)) * (loss.data - train_loss)
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss

             # forward pass
            output = model(data)

            # batch loss
            loss = criterion(output, target)

            # average validation loss
            valid_loss = valid_loss + (1 / (batch_idx + 1)) * (loss.data - valid_loss)

        # Training Data
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss <= valid_loss_min:
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss
            print('Validation Loss decreased! Model saved.')

    # Elapsed training time
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))

    # return trained model
    return model
```

Image 31: Train and validate function

Since this project works with images, if there is no GPU available, it could be quite slow to do all the training and processing required. This notebook that I'm using to test the entire code was not GPU supported, so I only ran a few epochs to show the results, and then trained the model in a GPU environment with 100 epochs. This new model can be imported into our current project by using torch.load, an extremely useful tool that will also helps us to import the final model to our Flask application.

In [31]:
```
# train the model
loaders_scratch = dataloaders

# Trained model_scratch with only 3 epochs
model_scratch = train(3, loaders_scratch, model_scratch, optimizer_scratch, criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1        Training Loss: 4.711918         Validation Loss: 4.357176
Validation Loss decreased! Model saved.
Epoch: 2        Training Loss: 4.425651         Validation Loss: 4.116408
Validation Loss decreased! Model saved.
Epoch: 3        Training Loss: 4.269604         Validation Loss: 3.974433
Validation Loss decreased! Model saved.
Training complete in 25m 18s
```

In [32]:
```
# Loading a 100 epoch trained model_scratch
model_scratch.load_state_dict(torch.load('model_scratch100.pt'))
```

Out[32]:
```
<All keys matched successfully>
```

Image 32: Trained model and 100 epoch loaded model

To finish it up this step, we need a function to test the accuracy of our model_scratch:

In [39]:
```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))
    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (100. * correct / total, correct, total))
```

In [34]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.487851


Test Accuracy: 43% (362/836)
```

Image 33: Testing model_scratch

43% accuracy is better than our established 40% goal mark, so this step was finished successfully.

- Step 6: Create a CNN to classify dog breeds using transfer learning

We've already discussed the concept of transfer learning, so we can jump straight indo coding. For this step, we'll use DenseNet161 pretrained model. And we don't have to do the preprocess again because we'll use the same dataloaders defined on the previous step. We'll also use the same train and test functions. So all it has do be done is download the model, define the loss function and optmizer and then start training.

In [33]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

# Sources: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
#          https://pytorch.org/hub/pytorch_vision_densenet/

model_transfer = models.densenet161(pretrained=True)
for param in model_transfer.parameters():
    param.requires_grad = False
num_ftrs = model_transfer.classifier.in_features
model_transfer.classifier = nn.Linear(num_ftrs, n_classes)

if use_cuda:
    model_transfer.cuda()
```

```
Downloading: "https://download.pytorch.org/models/densenet161-8d451a50.pth" to /home/ec2-user/.cache/torch/hub/checkpoints/dens
enet161-8d451a50.pth
```

Image 34: Testing model_scratch

In [36]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001, momentum=0.9)
loaders_transfer = dataloaders
```

Image 35: Loss and optimizer model_transfer

If 100 epochs takes a long time to train with a CNN from scratch, imagine how long a CNN from transfer learning would take without. In the next cell I can give an estimate, just to run 1 epoch it took more than 2 hours. So, again, I opened up a GPU environment, trained an 100 epoch model_transfer and brought to the project.

In [38]:
```
# train the model
n_epochs = 1
loaders_transfer = dataloaders
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'model_tran
```

```
Epoch: 1        Training Loss: 4.058557         Validation Loss: 2.852604
Validation Loss decreased! Model saved.
Training complete in 123m 41s
```

In [37]:
```
# Loading a 100 epoch trained model_transfer
model_transfer.load_state_dict(torch.load('model_transfer100.pt'))
```

Out[37]:
```
<All keys matched successfully>
```

Image 36: model_transfer train

Using the same test function, we can see how well this model performed.

In [40]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.309930


Test Accuracy: 89% (750/836)
```

Image 36: model_transfer test

As expected, model_transfer100 easily surpass model_scratch100. This shows how powerful transfer learning is. And also, 89% is a great accuracy, higher than our goal of 85%.

Now, using model_transfer100, we can write a predict_breed_function that recieves an image input and returns the predicted breed of that image.

In [65]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# Get all class names (from train class)
names = images['train'].classes

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed

    # Load the image
    img = Image.open(img_path)

    # Normalize image
    normalize = transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                     std=[0.5, 0.5, 0.5])

    # Define preprocess with transforms.Compose, the same applyed to the dataset
    preprocess = transforms.Compose([transforms.Resize(224),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     normalize])

    img_tensor = preprocess(img).float()

    # Insert the new axis (extra dimension)
    img_tensor.unsqueeze_(0)

    # Convert image to tensor
    img_tensor = Variable(img_tensor)

    if use_cuda:
        img_tensor = Variable(img_tensor.cuda())

    model_transfer.eval()

    output = model_transfer(img_tensor)
    output = output.cpu()

    # Get the prediction index
    predicted_index = output.data.numpy().argmax()

    # Returns the predicted class
    return names[predicted_index]
```

Image 37: Predict Breed function

Let's test it with a few images from the valid dog dataset:

In [66]:
```
# Testing the predict function:
for i in range(10):
    result = predict_breed_transfer(dog_files[i*15])
    print("Test file: {}".format(dog_files[i*15]))
    print("Predicted breed: {}".format(result))
    print("")
```

```
Test file: dogImages/valid/076.Golden_retriever/Golden_retriever_05245.jpg
Predicted breed: 076.Golden_retriever

Test file: dogImages/valid/044.Cane_corso/Cane_corso_03176.jpg
Predicted breed: 044.Cane_corso

Test file: dogImages/valid/012.Australian_shepherd/Australian_shepherd_00847.jpg
Predicted breed: 012.Australian_shepherd

Test file: dogImages/valid/033.Bouvier_des_flandres/Bouvier_des_flandres_02356.jpg
Predicted breed: 033.Bouvier_des_flandres

Test file: dogImages/valid/003.Airedale_terrier/Airedale_terrier_00182.jpg
Predicted breed: 003.Airedale_terrier

Test file: dogImages/valid/026.Black_russian_terrier/Black_russian_terrier_01859.jpg
Predicted breed: 026.Black_russian_terrier

Test file: dogImages/valid/109.Norwegian_elkhound/Norwegian_elkhound_07178.jpg
Predicted breed: 109.Norwegian_elkhound

Test file: dogImages/valid/049.Chinese_crested/Chinese_crested_03504.jpg
Predicted breed: 049.Chinese_crested

Test file: dogImages/valid/042.Cairn_terrier/Cairn_terrier_02966.jpg
Predicted breed: 111.Norwich_terrier

Test file: dogImages/valid/118.Pembroke_welsh_corgi/Pembroke_welsh_corgi_07640.jpg
Predicted breed: 118.Pembroke_welsh_corgi
```

Image 37: Predict Breed function test

Looks pretty decent! With this function, we finish Step 6 of our project. And now it's time to put everything together.

- Step 7: Write the full algorithm

It's been a long journey, but it's finally reaching its climax. Now that we have the face_detector function, the dog_detector function, and the predict_breed_function, we can create the full code that will work based on these terms:

- Reads an input image;
- If a dog is detected in the image, return the predicted breed;
- If a human is detected in the image, return the resembling dog breed;
- If neither is detected in the image, return an error.

I added a few extra lines just to show the input image along with a sample image with the resulting predicted breed.

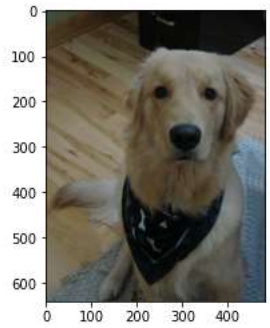
In [85]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
import matplotlib.image as mpimg
import random

def run_app(img_path):
    ## handle cases for a human face, dog, and neither

    is_dog = dog_detector(img_path)
    is_human = face_detector(img_path)
    prediction = predict_breed_transfer(img_path)

    # Is it a dog?
    if(is_dog):

        # Say that it's a dog and show the image analysed
        print("Based on my knowledge I can definetely tell that this is a DOG")
        img = mpimg.imread(img_path)
        imgplot = plt.imshow(img)
        plt.show()

        # Give the predicted breed
        breed = prediction[4:].replace("_", " ")
        print("And to me it looks like a {}".format(breed))

        # Show a random sample image from the valid dataset
        path = '/'.join(['dogImages/valid', str(prediction)])
        pick = random.choice(os.listdir(path))
        sample_path = '/'.join([path, pick])
        sample_image = mpimg.imread(sample_path)
        sample_plot = plt.imshow(sample_image)

        plt.show()

    # Is it a human?
    elif(is_human):

        # Say that it's a human and show the image analysed
        print("Based on my knowledge I can definetely tell that this is a HUMAN")
        img = mpimg.imread(img_path)
        imgplot = plt.imshow(img)
        plt.show()

        # Give the predicted breed
        breed = prediction[4:].replace("_", " ")
        print("And to me it looks like a {}".format(breed))

        # Show a random sample image from the valid dataset
        path = '/'.join(['dogImages/valid', str(prediction)])
        pick = random.choice(os.listdir(path))
        sample_path = '/'.join([path, pick])
        sample_image = mpimg.imread(sample_path)
        sample_plot = plt.imshow(sample_image)

        plt.show()

    # No human or dog
    else:
        print("ERROR: I can't see a dog or a human on this image. Please try another one")
        img = mpimg.imread(img_path)
        imgplot = plt.imshow(img)
        plt.show()

    print("-----------------------------------------------------------------")
```

Image 38: Final algorithm

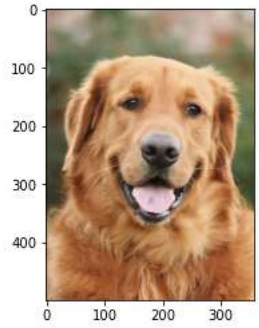
Let's test it in a dog image from the dataset:

In [86]:
```
# Testing on dog
run_app(dog_files[0])
```

```
Based on my knowledge I can definetely tell that this is a DOG
```



```
And to me it looks like a Golden retriever
```



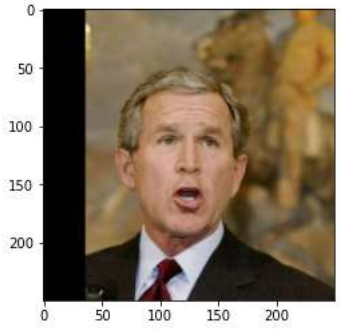
```
------------------------------------------------------------------
```

Image 39: Testing the algorithm on a dog image

Let's test it in a human image from the dataset:

In [87]:
```
# Testing on human
run_app(human_files[345])
```

```
Based on my knowledge I can definetely tell that this is a HUMAN
```



```
And to me it looks like a Irish wolfhound
```

```
------------------------------------------------------------------
```

Image 40: Testing the algorithm on a human image

And let's try it in a image that should give an error:

```
In [88]: # Neither option
         run_app("images/strawberry.jpg")
```

```
ERROR: I can't see a dog or a human on this image. Please try another one
```

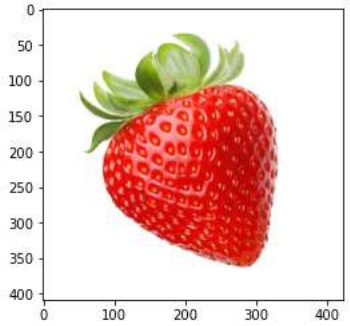


Image 41: Testing the algorithm on a error image

It's working like a charm! Now all we have to do is test it on images that are not in the dataset to check the results!.

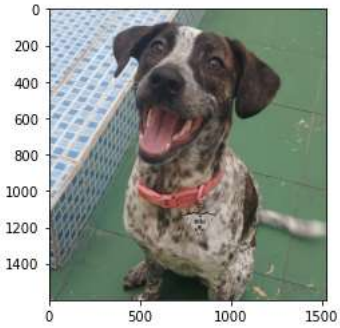
Results:

I've chosen 6 images to test it out:

- A picture of my dog (a mixed breed that contains a lot of German Shoirthaired Pointer);
- A picture of myself :)
- A picture of my friend's dog (another mixed breed with a lot of Labrador Retriever);
- A picture from a googled random woman;
- A picture of a cartoon dog;
- A picture of a cat;

Let's see what was produced by each image.

```
Based on my knowledge I can definetely tell that this is a DOG
```




```
And to me it looks like a German shorthaired pointer
```

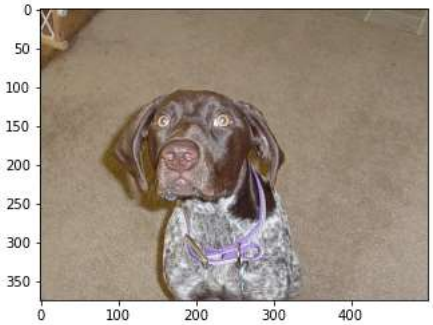
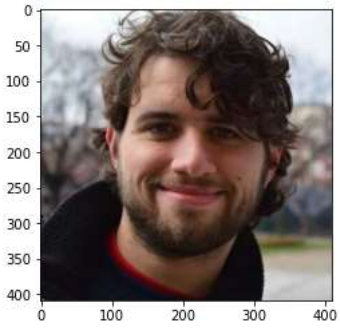


Image 42: Testing the algorithm on my dog

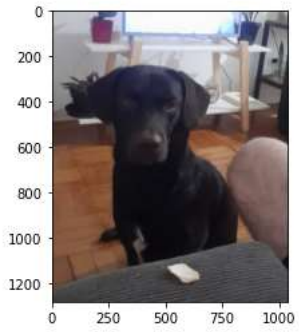
```
Based on my knowledge I can definetely tell that this is a HUMAN
```



```
And to me it looks like a Curly-coated retriever
```

Image 43: Testing the algorithm on myself

```
---------------------------------------------------------------------
Based on my knowledge I can definetely tell that this is a DOG
```



```
And to me it looks like a Labrador retriever
```

Image 44: Testing the algorithm on my friends dog

```
---------------------------------------------------------------------
Based on my knowledge I can definetely tell that this is a HUMAN
```

```
And to me it looks like a Havanese
```

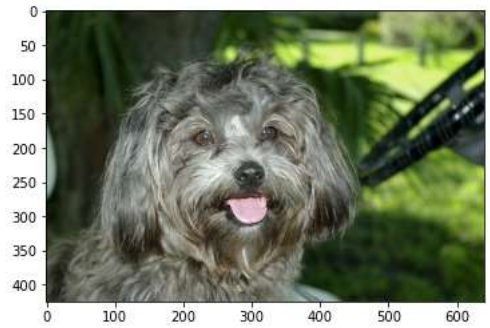


Image 45: Testing the algorithm on a random woman image

```
---------------------------------------------------------------------
ERROR: I can't see a dog or a human on this image. Please try another one
```

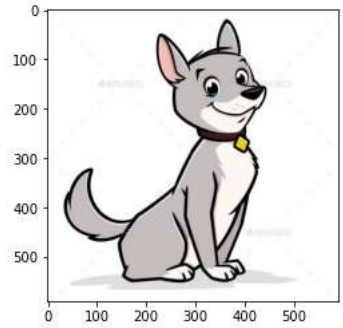
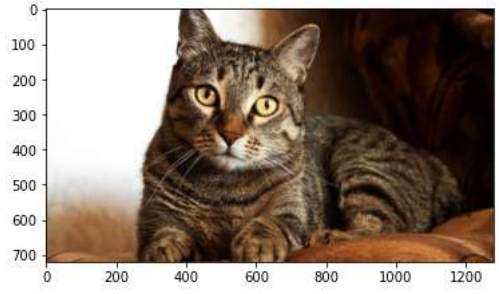


Image 46: Testing the algorithm on a cartoon dog

```
---------------------------------------------------------------------
ERROR: I can't see a dog or a human on this image. Please try another one
```



```
---------------------------------------------------------------------
```

Image 47: Testing the algorithm on a cat

After analyzing these results, I can say that the outputs were exactly what I expected. I tried two mixed breed dogs, mine which has a strong presence of a German Shorthaired Pointer, and another one from a friend of mine which has a lot of a Labrador Retriever, and those were exactly the results I got. For my personal picture, it was rather humourous that the output was a curly haired dog because of my curly-ish hair, and a simular result happened with

the random woman picture, making me believe that the hair has a strong weight in the model output. As for the other two images, I tried a cartoon dog and a cat, and the algorithm gave errors for both images.

This makes sence, because the index returned by the VGG16 model for these two pictures are "comic book" and "tiger cat", which are correct, but ignored by my application.

Deploy in a Flask Web Application:

When we're studying Machine Learning, we are often overwhelmed by lots of tools, algorithms, techniques and possibilities, but we forget one of the most important steps: deployment. What is the point of working hard on a Machine Learning application if nobody gets to use it? So to finish it up this project, I'll launched a small web application using Flask, and I'll leave this entire reported accessible from there.

In terms of Python programming there isn't too much secrets, we can use torch.load to load our model_transfer100 model, and all it needs to be done is redefine (or import) the same functions we did for the project, face_detector, dog_detector and predict_breed, and call these functions in specific routes of the application.

There is also a few html and css magic to create and customize input buttons for the images, but nothing too complex. In this project Github, there is a folder with the Flask Application and instructions to run it properly.

The final web site is hosted on andrevargas22.pythonanywhere.com

I also had an ideia of coding a Twitter bot using Tweepy, a python library for Twitter API, but unfortunately, due to a tight schedule, I didn't have enough time to pull it off before the deadline. It's certainly a challenge that I'll try eventually.

Conclusion:

This is was a very fun project to work on. It helped me understand a lot of concepts in the subject of Convolutional Neural Networks, Transfer Learning, Image Processing and even Flask Web Design, which is something I'm not used to do.

There are a few points of improvement to this project:

1) There are a few breeds not classified on this pretrained model which will give an error. For example, a picture of a Shiba Inu (japanese dog breed) will return index 273 - Canis Dingo, a type of australian wild canid, and since it's not within the index range accepted by the algorithm, it will give an error. So perhaps the usage of a different model with more classified breeds and a larger dataset would fix this problem.

2) I was able to get my model work with transfer learning from DenseNet161, which worked very well, but I wish to have tested with some other models for comparison. I know that ResNet50, InceptionV3 and Xception are interesting options, and maybe one of this models has an even better perfomance then DenseNet161.

3) I didn't make too many tests with the hyperparameters, I used the same values in the documentation examples, so I believe there is probably some better tuning to do.

References

[1]. Definition of Intelligence by Collins English Dictionary.

[2]. Image taken from "This Strawberry Hack Will Instantly Make Your Mushy Fruit as Good as New Again"

[3]. American Kennel Club: Dog Breeds.

[4]. Image taken from "American Kennel Club: Can You Tell These Dog Breed Look-Alikes Apart?".

[5]. Image taken from "Concepts in Digital Imaging Technology".

[6]. Image taken from "Stanford AI Lab Tutorial 1: Image Filtering".

[7]. Training Convolutional Neural Networks to Categorize Clothing with Pytorch.

[8]. Convolutional Neural Network Algorithms.

[9]. 7 Applications of Convolutional Neural Networks

[10]. Code taken from "How to Perform Face Detection with Deep Learning".

[11]. Code taken from "Applying transfer-learning in CNNs for dog breed classification".