dog_app

October 31, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

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

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

# 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.
```

Step 1: Detect Humans

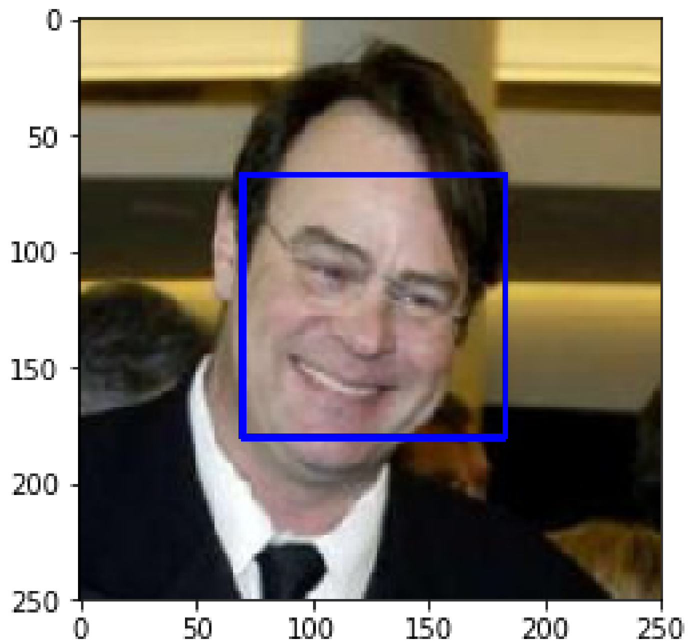
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate 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))
```

```python
# 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
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [3]: # 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
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our 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`.

Answer: (You can print out your results and/or write your percentages in this cell)

```
In [4]: 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.
        lista = []
        count_human_faces = 0
        count_dog = 0
        my_range = range(100)


        for i in tqdm(my_range) :
            if face_detector(human_files_short[i]):
                count_human_faces += 1
            if face_detector(dog_files_short[i]):
                count_dog += 1
```

```
100%|| 100/100 [00:31<00:00,  3.13it/s]
```

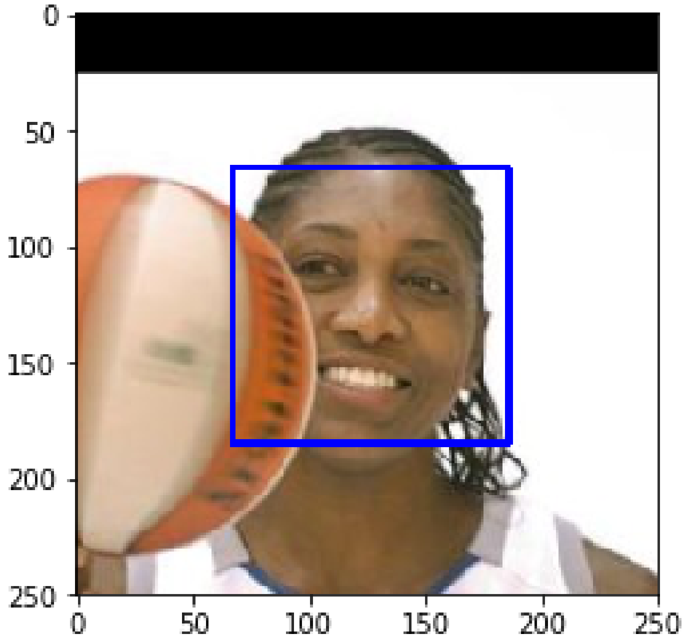
```
In [5]: print("Percentage of human faces in human dataset: {0:.2f}% \nPercentage of human faces
              .format( (count_human_faces/100)*100 , (count_dog/100)*100  ))
```

```
Percentage of human faces in human dataset: 98.00%
Percentage of human faces in dog dataset: 17.00%
```

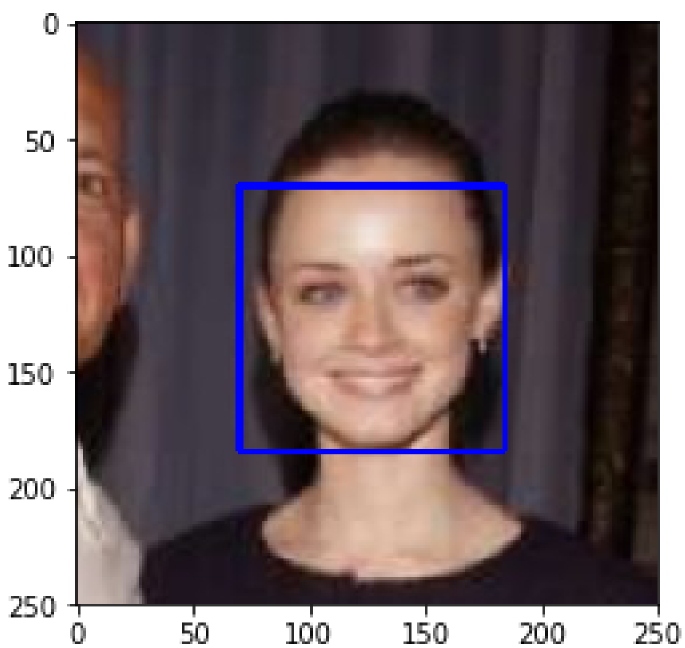
We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [6]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.

        def my_face_detector(img_path, detector):

            # convert BGR image to grayscale
            gray = cv2.cvtColor(img_path, cv2.COLOR_BGR2GRAY)

            # find faces in image
            faces = detector.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_path,(x,y),(x+w,y+h),(255,0,0),2)

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

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

In [7]: for ii in range(110,115):
            test_img = cv2.imread(human_files[ii])
            my_face_detector(test_img, face_cascade)
```

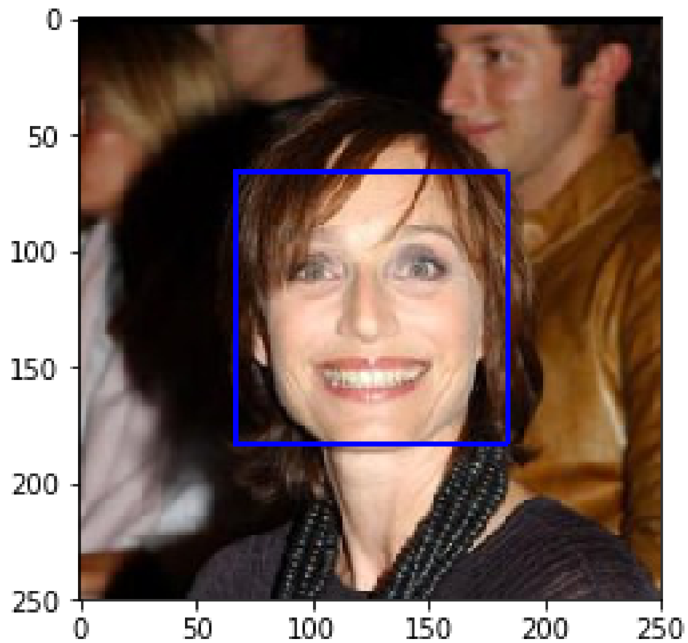
```
Number of faces detected: 1
```



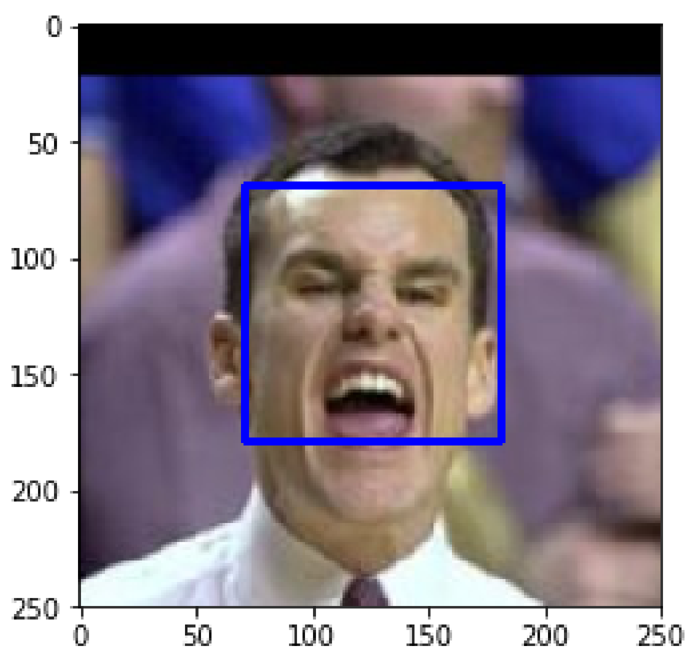

```
Number of faces detected: 1
```



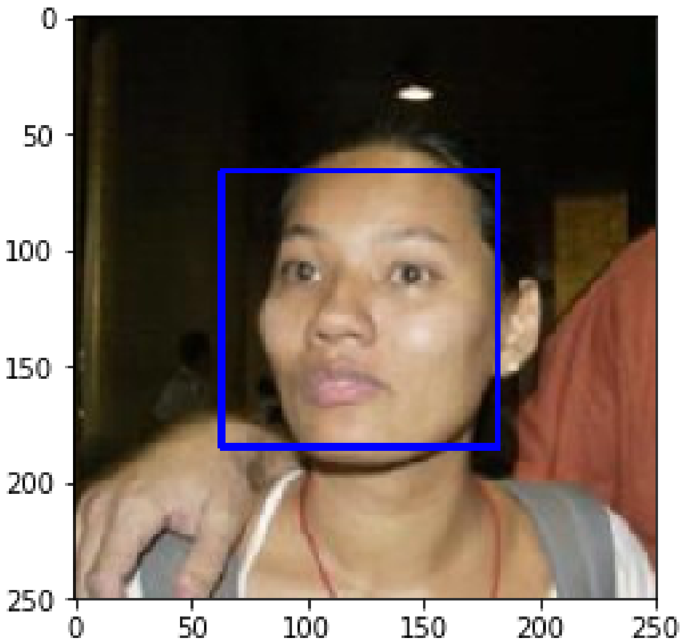
```
Number of faces detected: 1
```



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



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



Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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 [8]: 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 /root/.torch/models/vgg
100%|| 553433881/553433881 [01:01<00:00, 9007952.24it/s]
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as 'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg') as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [9]: from PIL import Image
        import torchvision.transforms as transforms
```

```
In [10]: def VGG16_predict(img_path):
             '''
             Use pre-trained VGG-16 model to obtain index corresponding to
             predicted ImageNet class for image at specified path

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''

             ## 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

             picture = Image.open(img_path).convert('RGB')



             #Tranfosmartion steps
             transform = transforms.Compose ([transforms.Resize(size=224),
                                             transforms.CenterCrop((224,224)),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406],
```

```
                                                                std=[0.229, 0.224, 0.225]
                                          ])

            image_ = transform(picture)[:3,:,:].unsqueeze(0)


            if use_cuda:
                image_ = image_.cuda()

            out = VGG16(image_)

            _, prediction = torch.max(out, 1)
            pred = np.squeeze(prediction.numpy())  if not use_cuda else np.squeeze(prediction.c

            return int(pred) # predicted class index
```

```
In [11]: test_prediction = VGG16_predict('/data/dog_images/train/001.Affenpinscher/Affenpinscher
         pred_class = int(test_prediction)

         print(f"Predicted class id: {pred_class}")
```

```
Predicted class id: 252
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the 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, in order 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).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [12]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.

             pred = VGG16_predict(img_path)

             #result = [True if (i >= 151) and (pred <= 268) else False for i in pred]

             return (pred >= 151) & (pred <= 268) # true/false
```

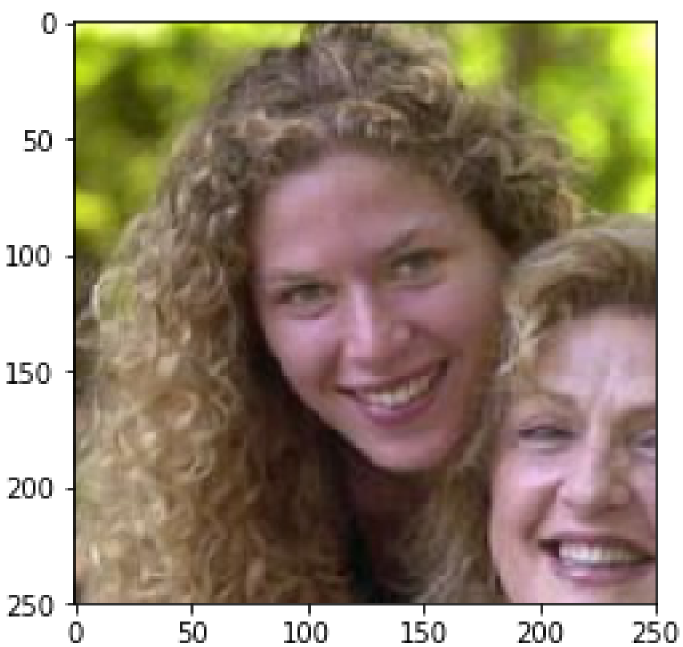
1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

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

detect_dogs_in_human = 0
detect_dogs_in_dogs = 0


for i in range(100):
    if dog_detector(human_files_short[i]):
        detect_dogs_in_human += 1
        human_img = Image.open(human_files_short[i])
        print("This person was identified as a dog")
        plt.imshow(human_img)
        plt.show()
    if dog_detector(dog_files_short[i]):
        detect_dogs_in_dogs += 1


hm_percentage = (detect_dogs_in_human/len(human_files_short)) * 100
dog_percentage = (detect_dogs_in_dogs/len(dog_files_short))*100

print("Percentage of humans faces identified as a dog specie: {0:.2f}%, \n Percentage o
```

```
This person was identified as a dog
```



```
Percentage of humans faces identified as a dog specie: 1.00%,
 Percentage of dogs detected as dog: 100.00%
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [14]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

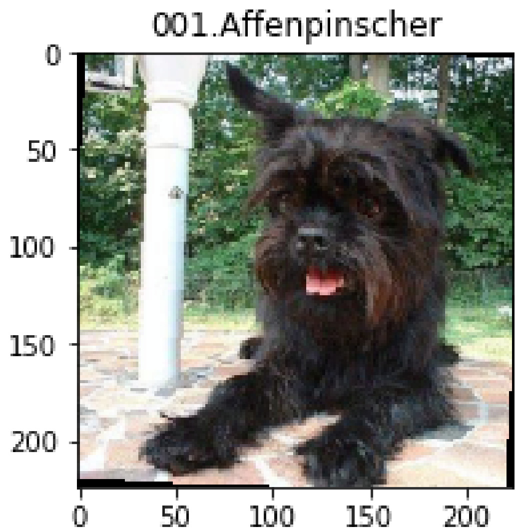
Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

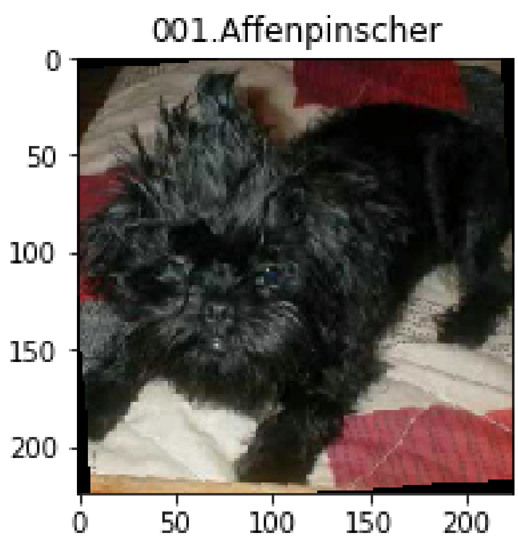
In [15]:
```
import os
from torchvision import datasets
import random
import requests
import ast
import time
import cv2
from PIL import Image, ImageFile

import torch
from torchvision import datasets
import torchvision.transforms as transforms
import torch.nn.functional as F
import torch.optim as optim
import torchvision.models as models



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

use_cuda = torch.cuda.is_available()
```

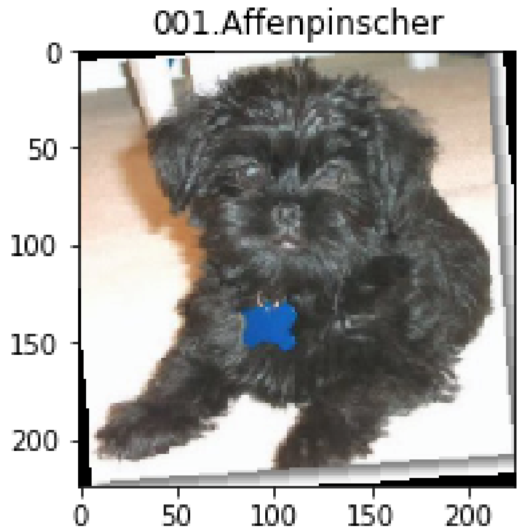
In [16]:
```
#Quantity of batch sizes sample per load
batch_size = 16

#Data loading subprocess number
num_workers = 2

valid_size = 0.2

#Transforming and normalizing the data
transform_train = transforms.Compose([transforms.Resize(size=224),
                                      transforms.CenterCrop((224,224)),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation(12),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.22


transform_test = transforms.Compose([transforms.Resize(size=224),
```

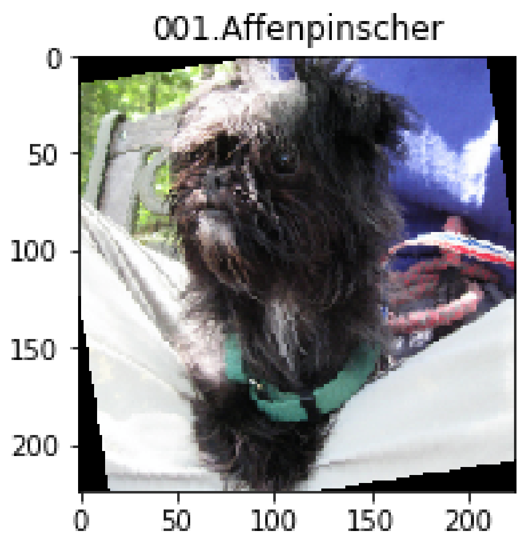
```
                                transforms.CenterCrop((224,224)),
                                transforms.ToTensor(),
                                transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.22


# Creating training, test and validation path
directory = '/data/dog_images/'

#img_dataset = { i : datasets.ImageFolder(os.path.join(directory, i), transform_train)

# #Ajustar aqui os data loaders
# loader = {l : torch.utils.data.DataLoader(img_dataset[l], shuffle=True, batch_size=ba

# Importing datasets
train_data = datasets.ImageFolder(os.path.join(directory,'train'),transform_train)

test_data = datasets.ImageFolder(os.path.join(directory,'test'),transform_test)

valid_data = datasets.ImageFolder(os.path.join(directory,'valid'),transform_test)

#Creating dataloaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,num_worker

test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,num_workers=

valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,num_worker
```

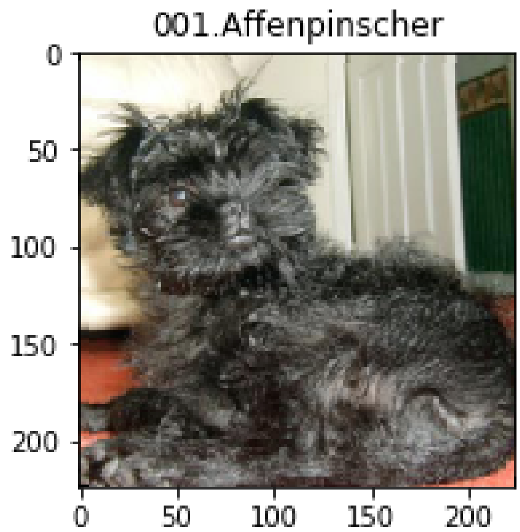
In [17]:
```
#Getting class names
cls_nm = train_data.classes

#Interating to get images into dataloader
inputs,classes = next(iter(train_loader))


#Interating over data and re-normalize it to standrd pattern to show the image as it is
for img, label in zip(inputs[:5], classes[:5]):
    img = img.to("cpu").clone().detach()
    img = img.numpy().squeeze()
    img = img.transpose(1,2,0)
    img = img * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    img = img.clip(0,1)


    fig = plt.figure(figsize=(12,3))
    plt.imshow(img)
    plt.title(cls_nm[label])
```


001.Affenpinscher
0
50
100
150
200
0
50
100
150
200


001.Affenpinscher
0
50
100
150
200
0
50
100
150
200


001.Affenpinscher
0
50
100
150
200
0
50
100
150
200


001.Affenpinscher
0
50
100
150
200
0
50
100
150
200



Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:

I tried to use the same methodology that I learn using MNIST and CIFAR.

I looking first to the shape of images to know how is it dimmensions to could treat it correctly. I choose flipping and rotate images to try amplify the dataset, to try make the model learn many kind of patterns for each classes to try avoid wrong classifications. Also, I tryed to normalize images following the best pratices showed during the course.

1.1.8 (IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. Use the template in the code cell below.

```
In [18]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
                 self.conv2 = nn.Conv2d(16,32,3, padding = 1)
                 self.conv3 = nn.Conv2d(32,64,3, padding=1)

                 #Max pooling layer
                 self.pool = nn.MaxPool2d(2,2)
```

```
        #Linear
        self.fc1 = nn.Linear(64 * 28 * 28, 500)
        self.fc2 = nn.Linear(500, 133)

        #Dropout layer
        self.dropout = nn.Dropout(0.25)
        self.batch_norm = nn.BatchNorm1d(num_features=500)



    def forward(self, x):
        ## Define forward behavior
        x = self.dropout(self.pool(F.relu(self.conv1(x))))
        x = self.dropout(self.pool(F.relu(self.conv2(x))))
        x = self.dropout(self.pool(F.relu(self.conv3(x))))


        #Flattening image input
        x = x.view(x.shape[0], -1)

        #Using linear model and ReLu activation function
        x = self.dropout(F.relu(self.batch_norm(self.fc1(x))))

        x = self.fc2(x)

        return x

#-#-# You so NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer:

I tried to use the first layer with the same shape of the image (224,224,3), outputting at the last layer the same shape of axisting class in the dataset 133.

As we're dealing with images I use Convolutional layers and Max pooling Layers reducing the size of my input, trying to keep only the most active pixels from the previous layer. At the end I use linear models to classify the image using dropout methods to avoid overfitting.

Looking for papers at the internet and based on my work experience I choose to use MaxPooling2D cus this method is an common choice to downsample.

I tried to define one model structure to get most complex patterns un boundaries and colors trying to make easy to the model get these patterns.

The filters were configured ith hight and width if 3, and during the convolution I'd like that the filter jump 1 by 1 pixel at time.

Also I tried to adding pool layers to try downsample the inputs dimensions by a factor of 2.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [19]:
```
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [20]:
```
import torch.optim as optim

### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr= 0.02)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

In [23]:
```
def train(n_epochs, loader_train, loader_valid, model, optimizer, criterion, use_cuda,
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = 3.808923

    if os.path.exists(save_path):
        model.load_state_dict(torch.load(save_path))

    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(train_loader):
            # 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_lo
```

```
        #Clear gradients
        optimizer.zero_grad()

        # Calc batch sizes
        output = model(data)

        #calculating loss
        loss = criterion(output, target)

        #Backward function
        loss.backward()

        #Perform optimization step
        optimizer.step()


        #Updating training loss
        train_loss += loss.item() * data.size(0)



    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, (data, target) in enumerate(valid_loader):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## update the average validation loss
        output = model(data)
        loss = criterion(output,target)
        valid_loss += loss.item()*data.size(0)

    # Calculating average losses
    train_loss = train_loss / len(train_loader.dataset)
    valid_loss = valid_loss / len(valid_loader.dataset)

    # print training/validation statistics
    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:
```

```
                print("Validation loss decreased and stay beeing saved!")
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss


        # return trained model
        return model
```

```
In [24]: # train the model
         model_scratch = train(17, train_loader,valid_loader, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')

         # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1         Training Loss: 4.981034         Validation Loss: 4.803396
Epoch: 2         Training Loss: 4.841687         Validation Loss: 4.738155
Epoch: 3         Training Loss: 4.862842         Validation Loss: 4.715117
Epoch: 4         Training Loss: 4.867002         Validation Loss: 4.648629
Epoch: 5         Training Loss: 4.843553         Validation Loss: 4.606935
Epoch: 6         Training Loss: 4.838650         Validation Loss: 4.597627
Epoch: 7         Training Loss: 4.800616         Validation Loss: 4.543097
Epoch: 8         Training Loss: 4.749176         Validation Loss: 4.502219
Epoch: 9         Training Loss: 4.769213         Validation Loss: 4.481428
Epoch: 10         Training Loss: 4.748585         Validation Loss: 4.463416
Epoch: 11         Training Loss: 4.669261         Validation Loss: 4.488988
Epoch: 12         Training Loss: 4.641216         Validation Loss: 4.468628
Epoch: 13         Training Loss: 4.660910         Validation Loss: 4.442299
Epoch: 14         Training Loss: 4.636487         Validation Loss: 4.441143
Epoch: 15         Training Loss: 4.595410         Validation Loss: 4.473638
Epoch: 16         Training Loss: 4.523386         Validation Loss: 4.469984
Epoch: 17         Training Loss: 4.447789         Validation Loss: 4.470791
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [25]: def test(test_loader, 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(test_loader):
                 # 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))

    # call test function
    test(test_loader, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.782564


Test Accuracy: 15% (128/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [26]: ## TODO: Specify data loaders

         #Creating dataloaders
         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,num_worker

         test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,num_workers=

         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,num_worker
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [27]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture

         model_transfer = models.resnet50(pretrained=True)


         if use_cuda:
             model_transfer = model_transfer.cuda()
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 70691970.57it/s]
```

```
In [28]: model_transfer
```

```
Out[28]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
           (layer1): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
               (downsample): Sequential(
                 (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
                 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               )
             )
             (1): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
```

```
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
```

```
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
```

```
        )
      )
      (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
      (fc): Linear(in_features=2048, out_features=1000, bias=True)
    )
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer:

I believe that the capacity of use well known models is very usefull and important to grow the capacity of develop great systems using models that won some kind of problems. The possibility of use this models and adapt them to working with many kind of issues is the key for grow this market at the world.

In this case I tryed to use Residual Model because it is knowing as an great model to recognize patterns in hard situations, and these dataset seams to be harder once we have many similarities classes, making an search at the internet I could saw that the community got used this model to solve problems like this one that we are dealing with.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [29]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(),lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [30]: # train the model
         n_epochs = 9

         model_transfer = train(n_epochs, train_loader, test_loader, model_transfer, optimizer_t
```

```
Epoch: 1         Training Loss: 16.597474         Validation Loss: 1.245253
Validation loss decreased and stay beeing saved!
Epoch: 2         Training Loss: 15.642812         Validation Loss: 1.270732
Epoch: 3         Training Loss: 15.584217         Validation Loss: 1.660278
Epoch: 4         Training Loss: 15.420693         Validation Loss: 1.630149
Epoch: 5         Training Loss: 15.247258         Validation Loss: 1.770450
Epoch: 6         Training Loss: 14.940275         Validation Loss: 1.819852
Epoch: 7         Training Loss: 14.820871         Validation Loss: 1.810162
Epoch: 8         Training Loss: 14.616376         Validation Loss: 1.895951
Epoch: 9         Training Loss: 14.511545         Validation Loss: 2.027465
```

```
In [31]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

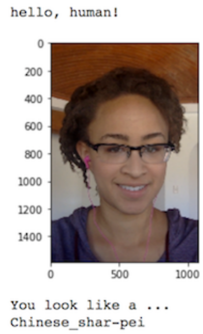
In [32]:
```
test(test_loader, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.227847


Test Accuracy: 73% (618/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

In [33]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in train_data.classes]

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

    picture = Image.open(img_path).convert('RGB')



    #Tranfosmartion steps
    transform = transforms.Compose ([transforms.Resize(size=224),
                                     transforms.CenterCrop((224,224)),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                          std=[0.229, 0.224, 0.225]
                                     ])

    img_tensor = transform(picture)[:3,:,:].unsqueeze(0)



    if use_cuda:
        img_tensor = img_tensor.cuda()

    pred = model_transfer(img_tensor)

    _, pred_tensor = torch.max(pred,1)
```



Sample Human Output

```
predict = np.squeeze(pred_tensor.numpy()) if not use_cuda else np.squeeze(pred_tens

return class_names[predict]
```

```
In [34]: def display_image(img_path, title="Title"):
             image = Image.open(img_path)
             plt.title(title)
             plt.imshow(image)
             plt.show()
```

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - 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, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [35]: , ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

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

             try:

                 if face_detector(img_path):
                     print("Human Here!")
```

```
            predicted_breed = predict_breed_transfer(img_path)
            display_image(img_path, predicted_breed)

            print("Hey human, you look like ...")
            print(predicted_breed)

        elif dog_detector(img_path):
            print("Hi, mad dog!!")
            predicted_breed = predict_breed_transfer(img_path)
            display_image(img_path,predicted_breed)

            print("This dog look most like a...")
            print(predicted_breed)

        else:

            print("Sorry my friend, we couldn't identify dog or an human in this pictur
            display_image(img_path, title="Try again please!")

    except Exception as e:
        print(e)
```

Step 6: Test Your Algorithm
In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

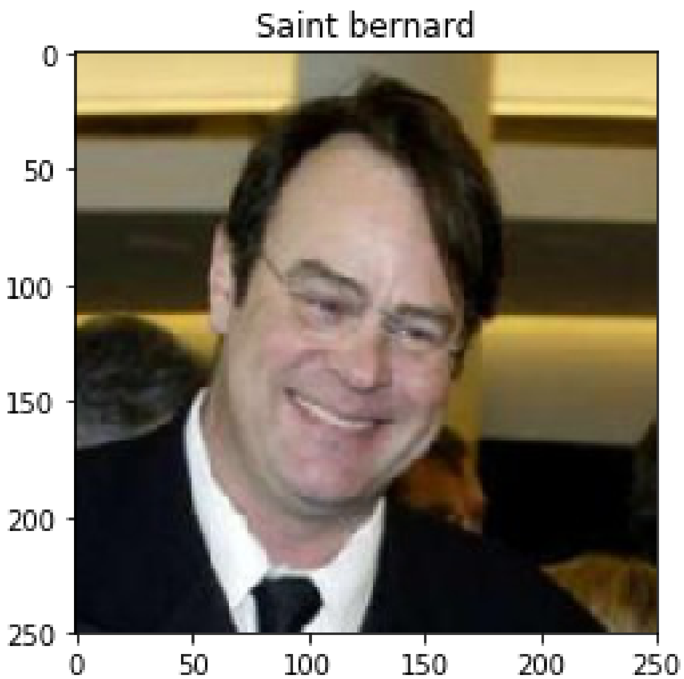
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

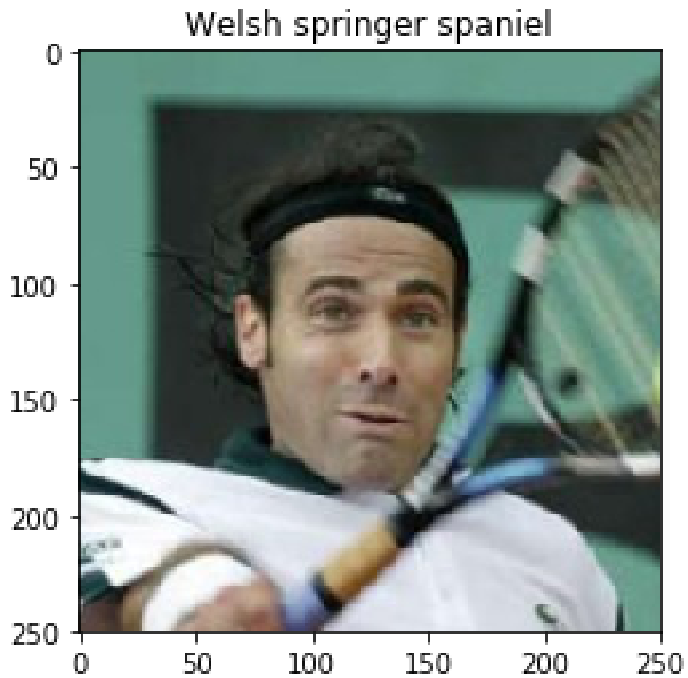
The output was better than I expected, I guessed that the similarity of some breends turned the work hard beeing diffucult to the model to got good accuracies. To improve the model we could: * 1) Implement more transformation methos to grow our dataset variability turnning easier to identify different patterns * 2) Improve our network with more layers, turning the model more specialized in discovering patterns * 3) Grow the number of training steps * 4) Trying another configuration for the filter at convolutional model.

```
In [36]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[:3], dog_files[:3])):
             run_app(file)
```

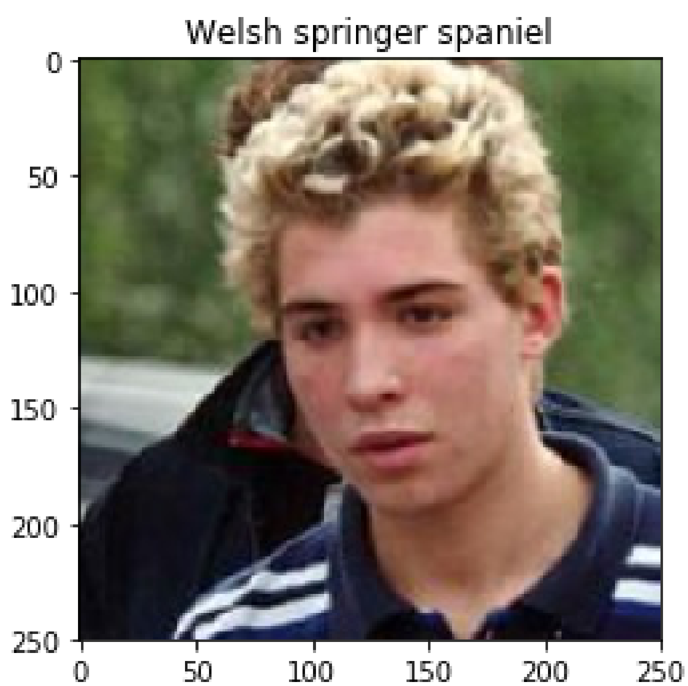
```
Human Here!
```



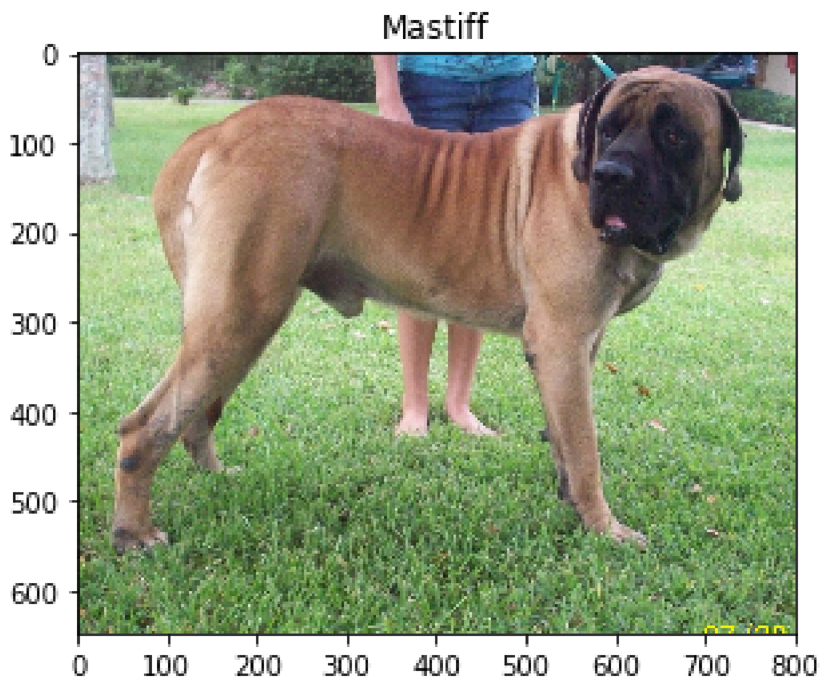

```
Hey human, you look like ...
Saint bernard
Human Here!
```



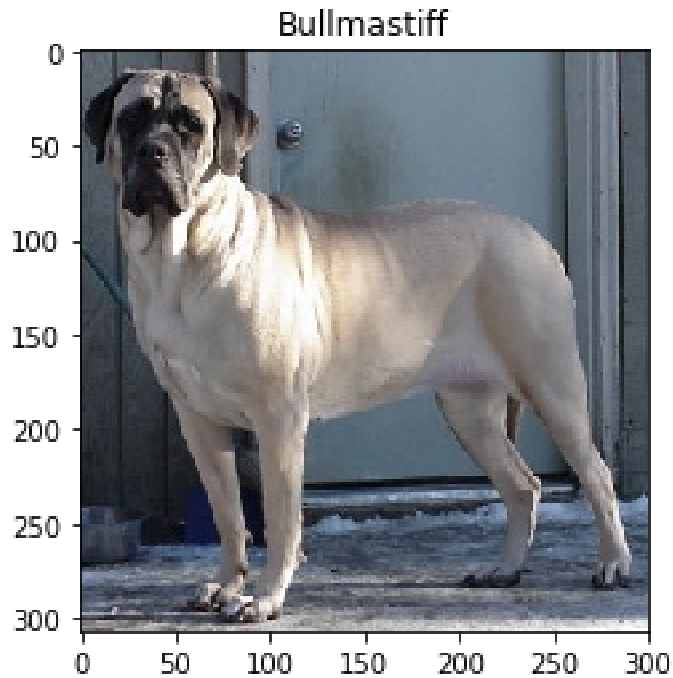

```
Hey human, you look like ...
Welsh springer spaniel
Human Here!
```



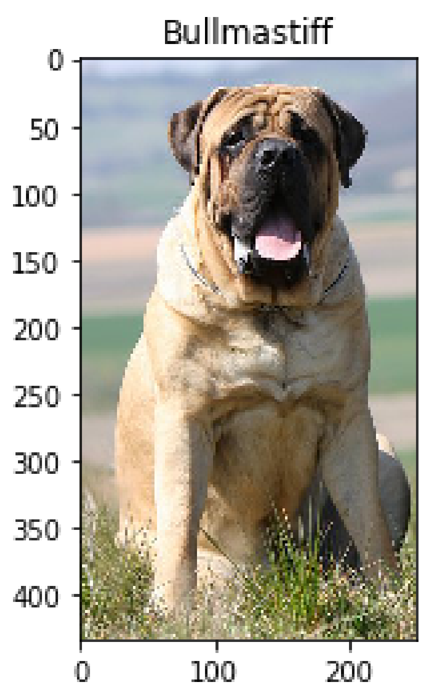
```
Hey human, you look like ...
Welsh springer spaniel
Hi, mad dog!!
```




```
This dog look most like a...
Mastiff
Hi, mad dog!!
```



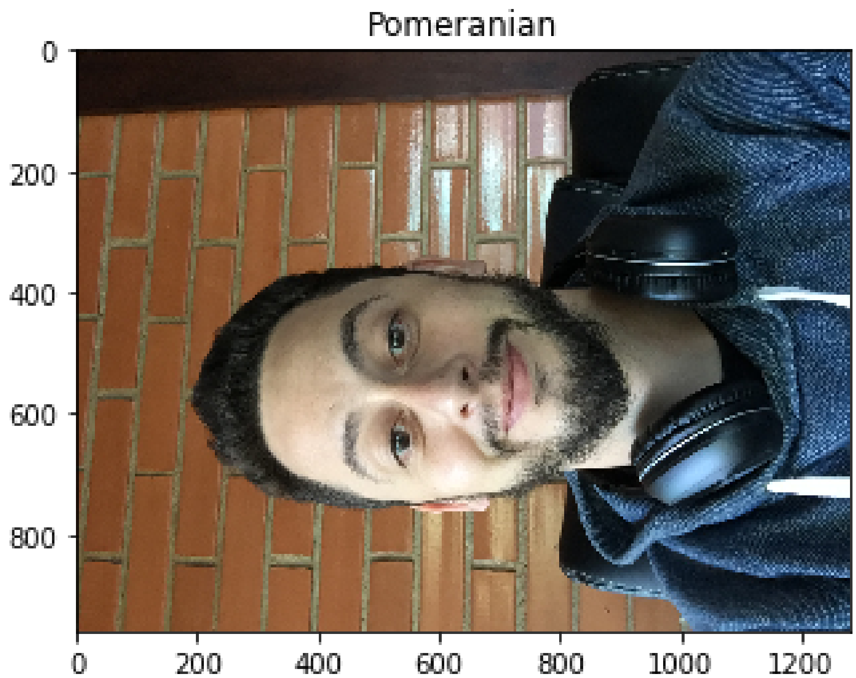

```
This dog look most like a...
Bullmastiff
Hi, mad dog!!
```



```
This dog look most like a...
Bullmastiff
```

```
In [43]: run_app("../my_img/eu.jpg")
```

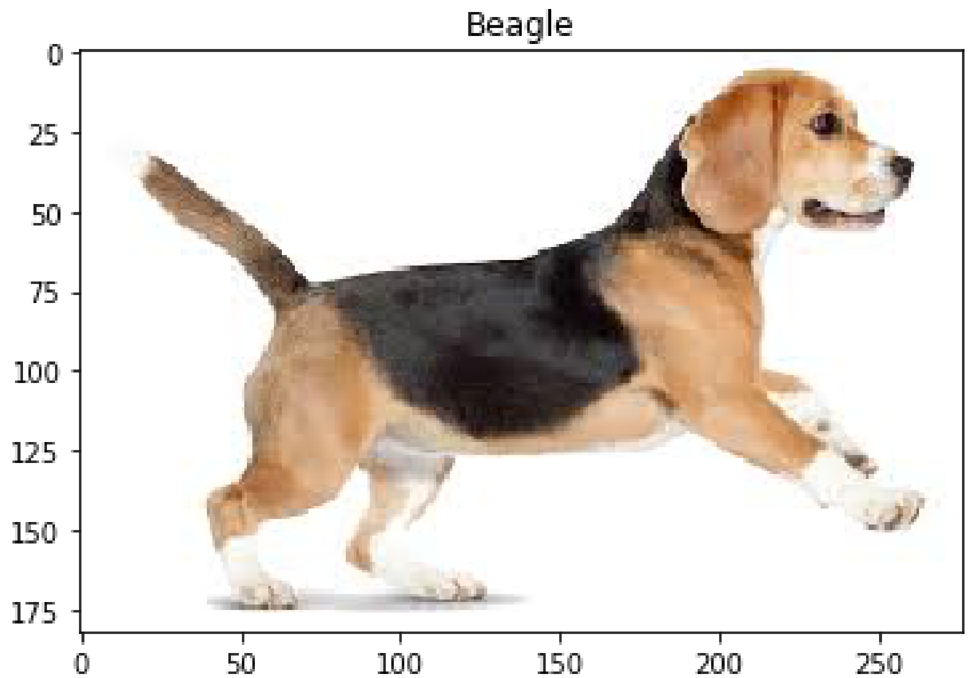
```
Human Here!
```




```
Hey human, you look like ...
Pomeranian
```

```
In [38]: run_app("../my_img/beagle.jpg")
```

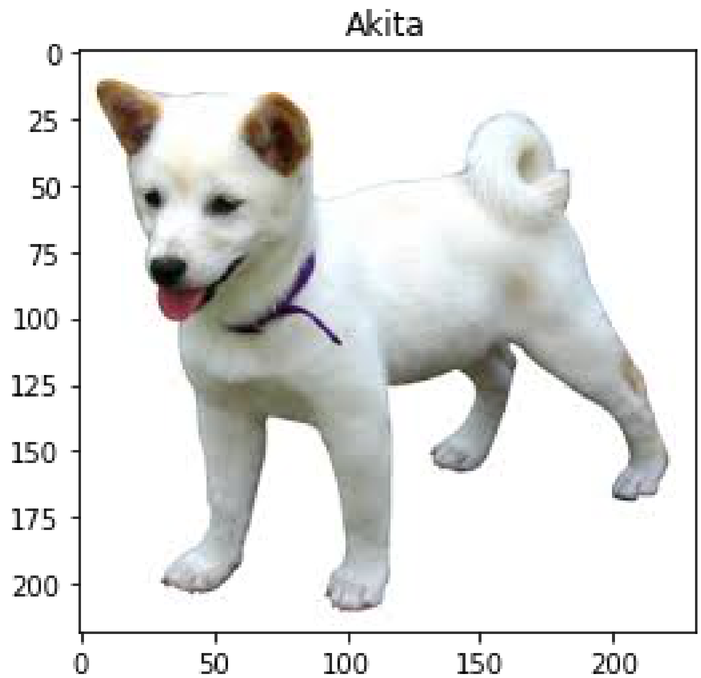
```
Hi, mad dog!!
```




```
This dog look most like a...
Beagle
```

```
In [39]: run_app("../my_img/chow.jpg")
```

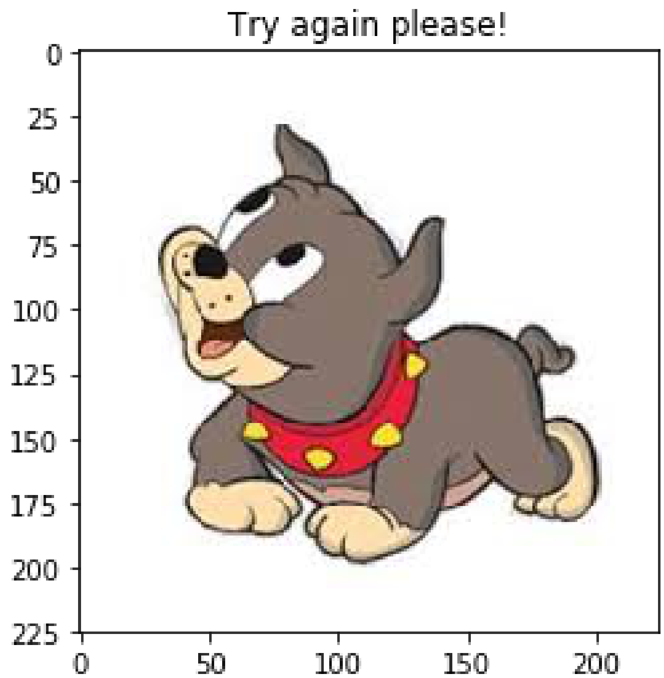
```
Human Here!
```




```
Hey human, you look like ...
Akita
```

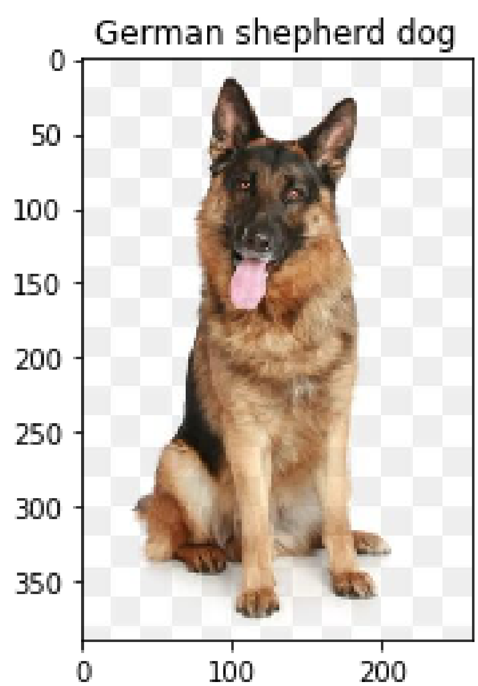
```
In [40]: run_app("../my_img/bulldog.jpg")
```

```
Sorry my friend, we couldn't identify dog or an human in this picture, please try it again with
```



In [41]: run_app("../my_img/pastor.jpg")

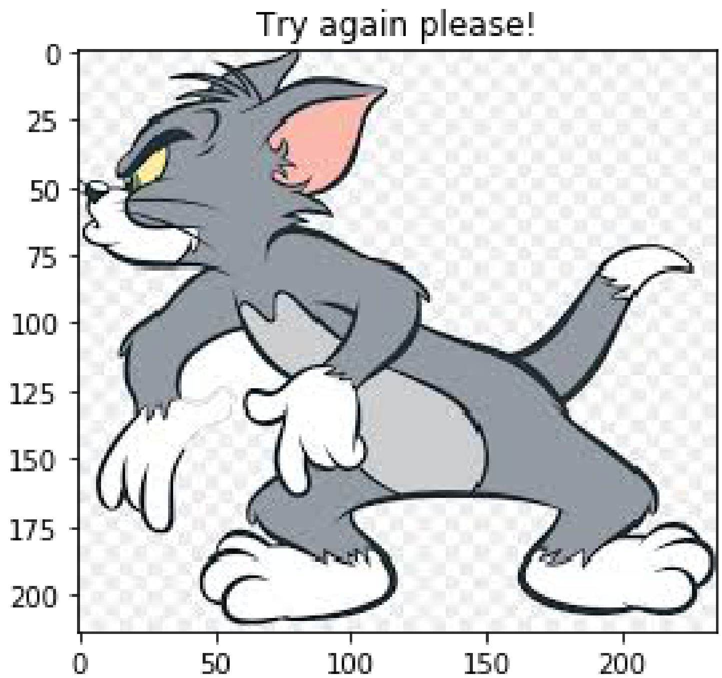
Hi, mad dog!!



```
This dog look most like a...
German shepherd dog
```

```
In [42]: run_app("../my_img/tom.jpg")
```

```
Sorry my friend, we couldn't identify dog or an human in this picture, please try it again with
```




```
In [ ]:
```