dog_app

October 27, 2018

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: * 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.

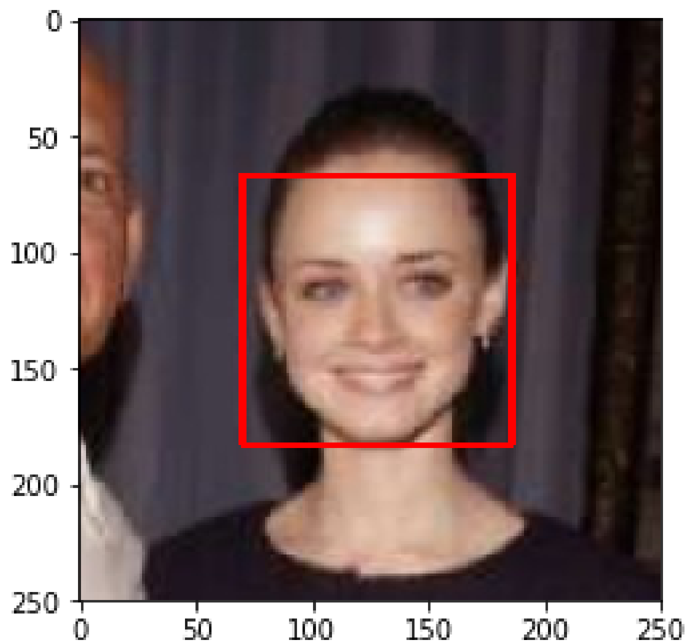
```
In [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [3]: def convertToRGB(img):
            return cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
In [4]: # extract pre-trained face detector
        haar_face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml'
        img = cv2.imread(human_files[111])

        def test_face_detector_on_image(img, face_detector):
            # convert BGR image to grayscale
            gray = convertToRGB(img)

            # find faces in image
            faces = face_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,(x,y-4),(x+w+4,y+h),(0,0,255),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()

test_face_detector_on_image(img, haar_face_cascade)
```

```
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 [5]: # returns "True" if face is detected in image stored at img_path
        def haar_face_detector(img_path):
            img = cv2.imread(img_path)
            gray = convertToRGB(img)
            faces = haar_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:

Using haarcascades face detector:

Detected human faces in the first 100 human images: 96%

Detected faces in the first 100 dogs images: 16%

```
In [6]: 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.
        detected_faces_in_humans = 0
        detected_faces_in_dogs = 0

        for ii in range(100):
            if haar_face_detector(human_files_short[ii]):
                detected_faces_in_humans += 1
            if haar_face_detector(dog_files_short[ii]):
                detected_faces_in_dogs +=1

        print (f"Detected human faces: {detected_faces_in_humans}%")
        print (f"Detected faces in dogs: {detected_faces_in_dogs}%")
```

```
Detected human faces: 96%
Detected faces in dogs: 16%
```

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`.

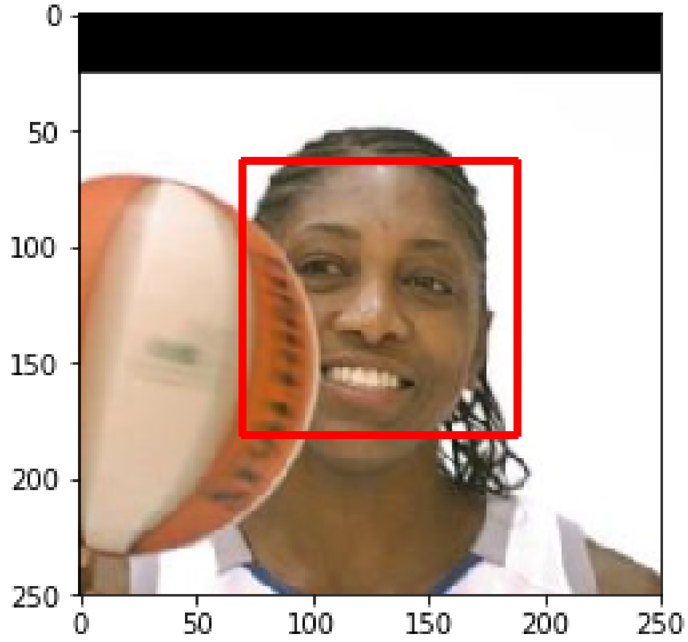
In [7]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.

#Approach II: OPENCV LBP CASCADE CLASSIFIER
#load cascade classifier training file for lbpcascade
lbp_face_cascade = cv2.CascadeClassifier('lbpcascade/lbpcascade_frontalface_improved.xml

def lbp_face_detector(img_path):
    img = cv2.imread(img_path)
    gray = convertToRGB(img)
    faces = lbp_face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

In [8]:
```
detected_faces_in_humans = 0
detected_faces_in_dogs = 0

for ii in range(100):
    if lbp_face_detector(human_files_short[ii]):
        detected_faces_in_humans += 1
    if lbp_face_detector(dog_files_short[ii]):
        detected_faces_in_dogs +=1

print (f"Detected human faces (LBP): {detected_faces_in_humans}%")
print (f"Detected faces in dogs(LBP): {detected_faces_in_dogs}%")
```

```
Detected human faces (LBP): 82%
Detected faces in dogs(LBP): 7%
```

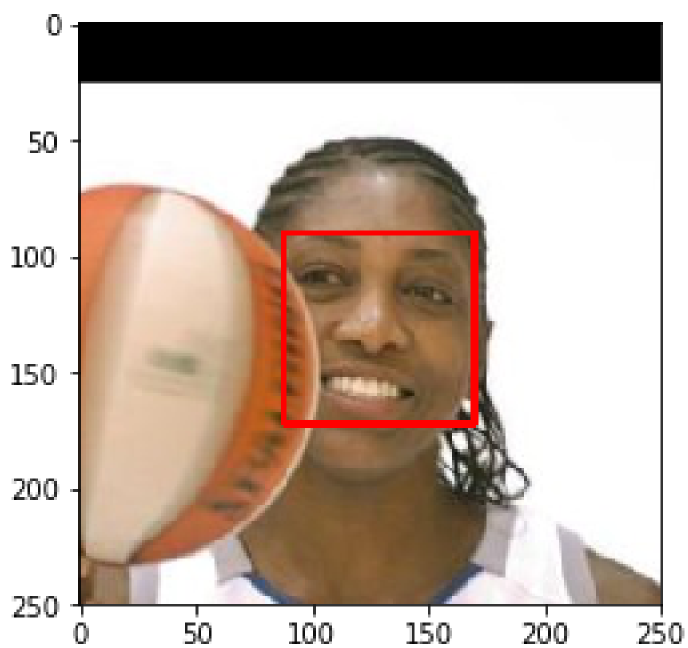
In [9]:
```
img = cv2.imread(human_files[110])
test_face_detector_on_image(img, haar_face_cascade)
```

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



```
In [10]: img = cv2.imread(human_files[110])
         test_face_detector_on_image(img, lbp_face_cascade)
```

```
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 [11]:
```
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 [00:06<00:00, 91275352.19it/s]
```

In [12]:
```
VGG16
```

Out[12]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
      (classifier): Sequential(
        (0): Linear(in_features=25088, out_features=4096, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.5)
        (3): Linear(in_features=4096, out_features=4096, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.5)
        (6): Linear(in_features=4096, out_features=1000, bias=True)
      )
    )
```

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.

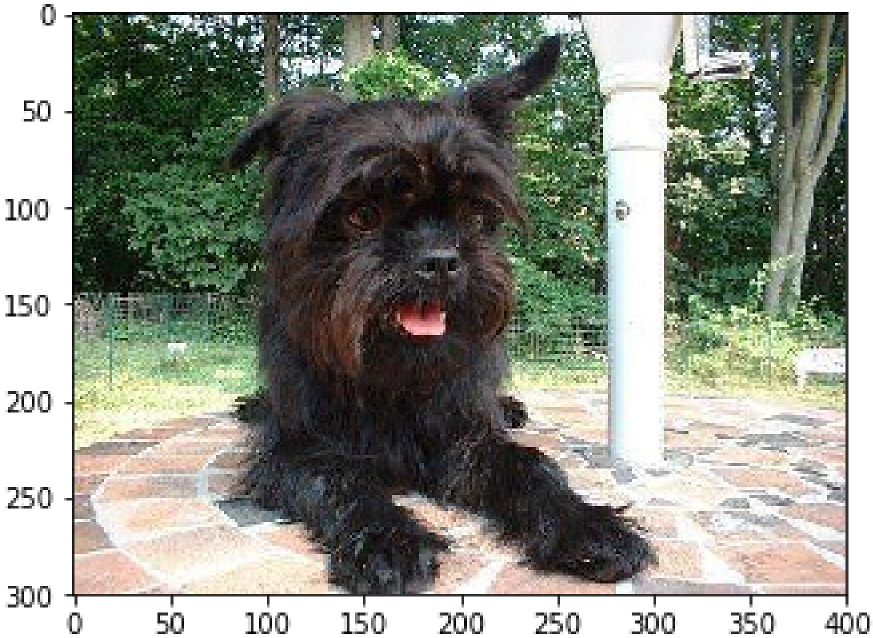
Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [13]: from PIL import Image
         import torchvision.transforms as transforms
```

```
In [14]: def image_to_tensor(img_path):
             '''
             As per Pytorch documentations: All pre-trained models expect input images normalize
             i.e. mini-batches of 3-channel RGB images
```

```
    of shape (3 x H x W), where H and W are expected to be at least 224.
    The images have to be loaded in to a range of [0, 1] and
    then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
    You can use the following transform to normalize:
    '''
    img = Image.open(img_path).convert('RGB')
    transformations = 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_tensor = transformations(img)[:3,:,:].unsqueeze(0)
    return image_tensor


# helper function for un-normalizing an image  - from STYLE TRANSFER exercise
# and converting it from a Tensor image to a NumPy image for display
def im_convert(tensor):
    """ Display a tensor as an image. """

    image = tensor.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1)

    return image
```

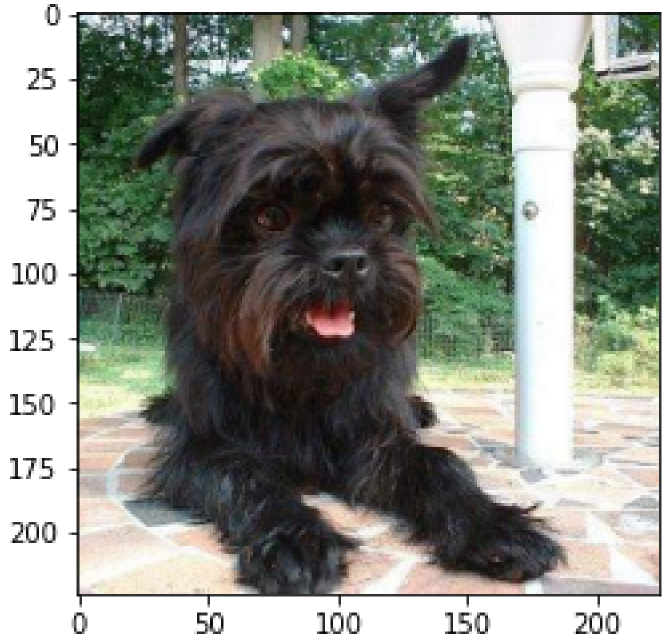
```
In [15]: dog_image = Image.open('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jp
         plt.imshow(dog_image)
         plt.show()
```




```
In [16]: test_tensor = image_to_tensor('/data/dog_images/train/001.Affenpinscher/Affenpinscher_0
         # print(test_tensor)
         print(test_tensor.shape)
         plt.imshow(im_convert(test_tensor))
```

```
torch.Size([1, 3, 224, 224])
```

```
Out[16]: <matplotlib.image.AxesImage at 0x7f7b8395a710>
```



In [17]:
```
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
    '''

    image_tensor = image_to_tensor(img_path)

    # move model inputs to cuda, if GPU available
    if use_cuda:
        image_tensor = image_tensor.cuda()

    # get sample outputs
    output = VGG16(image_tensor)
    # convert output probabilities to predicted class
    _, preds_tensor = torch.max(output, 1)
    pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso

    return int(pred)
```

```
In [18]: import requests
         import ast

         LABELS_MAP_URL = "https://gist.githubusercontent.com/yrevar/942d3a0ac09ec9e5eb3a/raw/c2

         def get_human_readable_label_for_class_id(class_id):
             labels = ast.literal_eval(requests.get(LABELS_MAP_URL).text)
             print(f"Label:{labels[class_id]}")
             return labels[class_id]


         test_prediction = VGG16_predict('/data/dog_images/train/001.Affenpinscher/Affenpinscher
         pred_class = int(test_prediction)

         print(f"Predicted class id: {pred_class}")
         class_description = get_human_readable_label_for_class_id(pred_class)
         print(f"Predicted class for image is *** {class_description.upper()} ***")
```

```
Predicted class id: 252
Label:affenpinscher, monkey pinscher, monkey dog
Predicted class for image is *** AFFENPINSCHER, MONKEY PINSCHER, MONKEY DOG ***
```

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

             prediction = VGG16_predict(img_path)
         #     print(prediction)
             return ((prediction >= 151) & (prediction <=268))
```

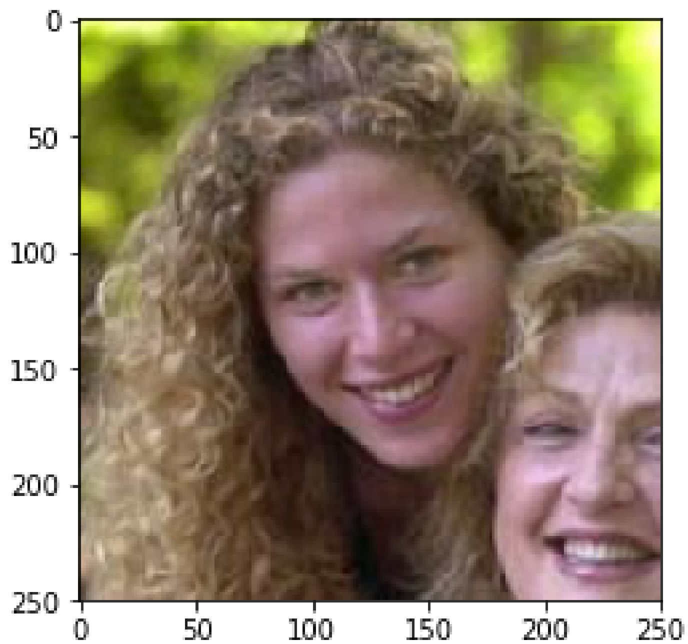
1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

         detected_dogs_in_humans = 0
         detected_dogs_in_dogs = 0

         for ii in range(100):
             if dog_detector(human_files_short[ii]):
                 detected_dogs_in_humans += 1
                 print(f"This human ({ii}) looks like a dog")
                 human_dog_image = Image.open(human_files_short[ii])
                 plt.imshow(human_dog_image)
                 plt.show()
             if dog_detector(dog_files_short[ii]):
                 detected_dogs_in_dogs +=1

         print (f"Percentage of the images in human_files_short that have a detected dog: {detec
         print (f"Percentage of the images in dog_files_short that have a detected dog: {detecte
```

```
This human (88) looks like a dog
```




```
Percentage of the images in human_files_short that have a detected dog: 1%
Percentage of the images in dog_files_short that have a detected dog: 99%
```

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 [21]: ### (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 [22]:
```
import os
from torchvision import datasets

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


# how many samples per batch to load
batch_size = 4

# number of subprocesses to use for data loading
num_workers = 4

# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([transforms.Resize(size=224),
                                transforms.CenterCrop((224,224)),
                                transforms.RandomHorizontalFlip(), # randomly flip and
                                transforms.RandomRotation(10),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0

# define training, test and validation data directories
data_dir = '/data/dog_images/'

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), transform)
                  for x in ['train', 'valid', 'test']}
loaders_scratch = {
    x: torch.utils.data.DataLoader(image_datasets[x], shuffle=True, batch_size=batch_si
    for x in ['train', 'valid', 'test']}
```

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?

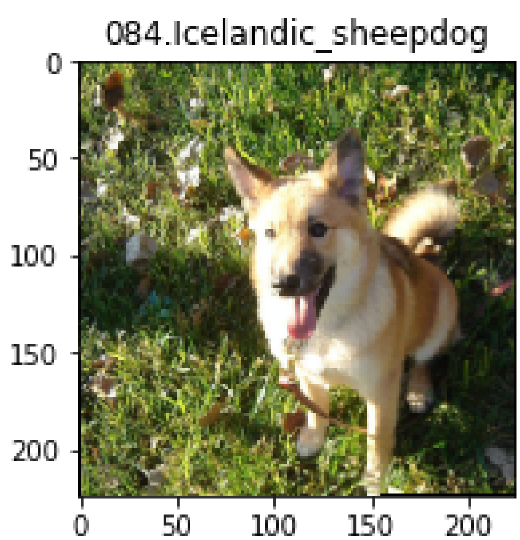
In [23]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

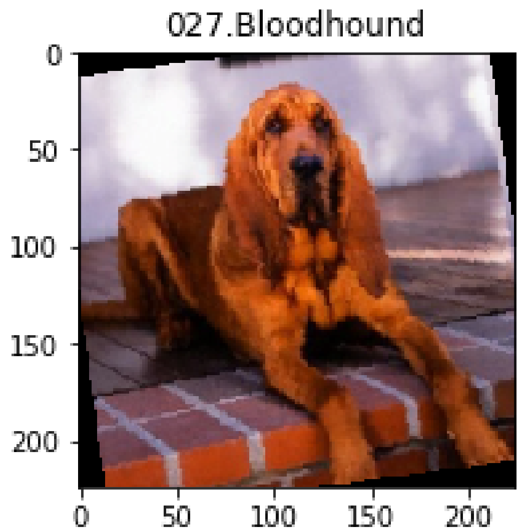
In [24]:
```
# # print out some data stats
# print('Num training images: ', len(train_data))
# print('Num test images: ', len(test_data))
# print('Num validation images: ', len(valid_data))
```

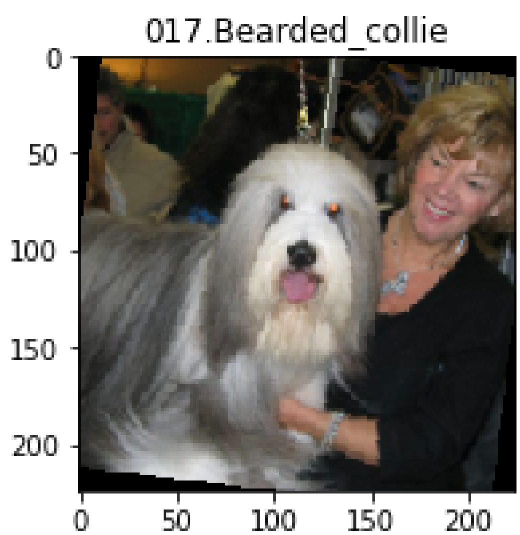
```
In [25]: class_names = image_datasets['train'].classes
         nb_classes = len(class_names)

         print("Number of classes:", nb_classes)
         print("\nClass names: \n\n", class_names)
```

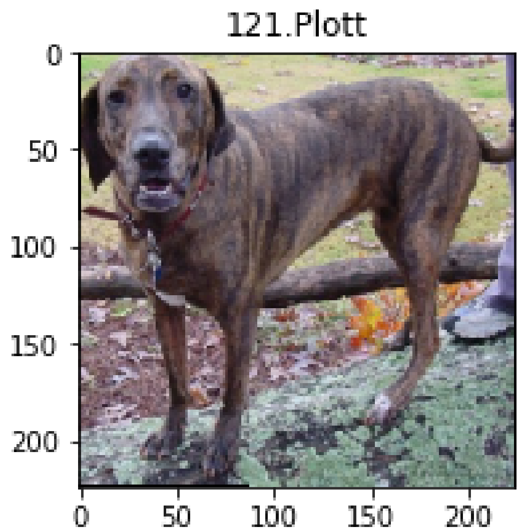
```
Number of classes: 133

Class names:

 ['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mal
```

```
In [26]: import torchvision
         # Get a batch of training data
         inputs, classes = next(iter(loaders_scratch['train']))

         for image, label in zip(inputs, classes):
             image = image.to("cpu").clone().detach()
             image = image.numpy().squeeze()
             image = image.transpose(1,2,0)
             image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
             image = image.clip(0, 1)

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




027.Bloodhound
0
50
100
150
200
0
50
100
150
200


017.Bearded_collie
0
50
100
150
200
0
50
100
150
200



Answer:

I loaded in the training, test and validation data, then created DataLoaders for each of these sets of data.

I resized all image to 224, center cropped and added some simple data augmentation by randomly flipping and rotating the given image data.

I approached the problem iteratively, starting with the examples from the previous labs, especially Cifar and MMNIST examples.

We are working with (224, 224, 3) images in this project so the inputs are significantly bigger than the labs (28, 28, 1) for Mnist and (32x32x3) for CIFAR.

Most of the pretrained models require the input to be 224x224 images. Also, we'll need to match the normalization used when the models were trained. Each color channel was normalized separately, the means are [0.485, 0.456, 0.406] and the standard deviations are [0.229, 0.224, 0.225].

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [27]: 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
                 # convolutional layer (sees 224x224x3 image tensor)
                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                 # convolutional layer (sees 16x16x16 tensor)
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
```

```
            # convolutional layer (sees 8x8x32 tensor)
            self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
            # max pooling layer
            self.pool = nn.MaxPool2d(2, 2)
            # linear layer (64 * 28 * 28 -> 500)
            self.fc1 = nn.Linear(64 * 28 * 28, 500)
            # linear layer (500 -> 133)
            self.fc2 = nn.Linear(500, 133)
            # dropout layer (p=0.25)
            self.dropout = nn.Dropout(0.25)

        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.pool(F.relu(self.conv3(x)))
            # flatten image input
            # 64 * 28 * 28
            x = x.view(-1, 64 * 28 * 28)
            # add dropout layer
            x = self.dropout(x)
            # add 1st hidden layer, with relu activation function
            x = F.relu(self.fc1(x))
            # add dropout layer
            x = self.dropout(x)
            # add 2nd hidden layer, with relu activation function
            x = self.fc2(x)
            return x

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

    # instantiate the CNN
    model_scratch = Net()
    print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

Answer:

First layer has input shape of (224, 224, 3) and last layer should output 133 classes.

I started adding Convolutional layers (stack of filtered images) and Maxpooling layers(reduce the x-y size of an input, keeping only the most active pixels from the previous layer) as well as the usual Linear + Dropout layers to avoid overfitting and produce a 133-dim output.

MaxPooling2D seems to be a common choice to downsample in these type of classification problems and that is why I chose it.

The more convolutional layers we include, the more complex patterns in color and shape a model can detect.

The first layer in my CNN is a convolutional layer that takes (224, 224, 3) inpute shap.

I'd like my new layer to have 16 filters, each with a height and width of 3. When performing the convolution, I'd like the filter to jump 1 pixel at a time.

nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)

I want this layer to have the same width and height as the input layer, so I will pad accordingly; Then, to construct this convolutional layer, I would use the following line of code: self.conv2 = nn.Conv2d(3, 32, 3, padding=1)

I am adding a pool layer that takes in a kernel_size and a stride after every convolution layer. This will down-sample the input's x-y dimensions, by a factor of 2:

self.pool = nn.MaxPool2d(2,2)

I am adding a fully connected Linear Layer to produce a 133-dim output. As well as a Dropout layer to avoid overfitting.

Forward pass would give:

```
# torch.Size([4, 3, 224, 224])
# torch.Size([4, 16, 112, 112])
# torch.Size([4, 32, 56, 56])
# torch.Size([4, 64, 28, 28])
# torch.Size([4, 50176])
# torch.Size([4, 50176])
```

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 [28]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)
```

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 [29]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf #last: 4.166247

    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 #
        ###################
        # load your previously trained model:
#          model.load_state_dict(torch.load(save_path))
        model.train()
        for data, target in loaders['train']:
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()

            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss += loss.item()*data.size(0)

        ######################
        # validate the model #
        ######################
        model.eval()
        for data, target in loaders['valid']:
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
```

```
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss += loss.item()*data.size(0)

            # calculate average losses
            train_loss = train_loss/len(loaders['train'].dataset)
            valid_loss = valid_loss/len(loaders['valid'].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
                 # save model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss
        # return trained model
        return model
```

```
In [30]: # train the model
         model_scratch = train(12, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.796418         Validation Loss: 4.605913
Validation loss decreased (inf --> 4.605913).  Saving model ...
Epoch: 2         Training Loss: 4.458308         Validation Loss: 4.496463
Validation loss decreased (4.605913 --> 4.496463).  Saving model ...
Epoch: 3         Training Loss: 4.280283         Validation Loss: 4.284789
Validation loss decreased (4.496463 --> 4.284789).  Saving model ...
Epoch: 4         Training Loss: 4.146587         Validation Loss: 4.137463
Validation loss decreased (4.284789 --> 4.137463).  Saving model ...
Epoch: 5         Training Loss: 3.998347         Validation Loss: 4.135734
Validation loss decreased (4.137463 --> 4.135734).  Saving model ...
Epoch: 6         Training Loss: 3.833039         Validation Loss: 3.989093
Validation loss decreased (4.135734 --> 3.989093).  Saving model ...
Epoch: 7         Training Loss: 3.656575         Validation Loss: 3.967157
Validation loss decreased (3.989093 --> 3.967157).  Saving model ...
Epoch: 8         Training Loss: 3.465782         Validation Loss: 3.987541
```

```
Epoch: 9        Training Loss: 3.277553        Validation Loss: 3.966600
Validation loss decreased (3.967157 --> 3.966600).  Saving model ...
Epoch: 10       Training Loss: 3.044261        Validation Loss: 4.222845
Epoch: 11       Training Loss: 2.839388        Validation Loss: 3.871563
Validation loss decreased (3.966600 --> 3.871563).  Saving model ...
Epoch: 12       Training Loss: 2.574108        Validation Loss: 4.133825
```

```
In [33]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

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 [34]: 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 += loss.item()*data.size(0)
                 # 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 testing statistics

             # calculate average loss
             test_loss = test_loss/len(loaders['test'].dataset)

             # print test statistics
             print('Testing Loss Average: {:.6f} '.format(test_loss))
```

```
        print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
            100. * correct / total, correct, total))
```

```
In [35]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Testing Loss Average: 3.807470

Test Accuracy: 14% (123/836)
```

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 [36]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch
         print(loaders_transfer)
```

```
{'train': <torch.utils.data.dataloader.DataLoader object at 0x7f7bfc4b2d68>, 'valid': <torch.uti
```

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 [37]: 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 [00:04<00:00, 24853315.78it/s]
```

```
In [38]: model_transfer

Out[38]: 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:

It is very efficient to use pre-trained networks to solve challenging problems in computer vision.

Once trained, these models work very well as feature detectors for images they weren't trained on. Here we'll use transfer learning to train a network that can classify our dog photos.

For this task speciffically, I'll use `resnet50` trained on ImageNet available from torchvision.

The classifier part of the model is a single fully-connected layer:

`(fc): Linear(in_features=2048, out_features=1000, bias=True)`

This layer was trained on the ImageNet dataset, so it won't work for the dog classification specific problem.

That means we need to replace the classifier (133 classes), but the features will work perfectly on their own.

Choice of criterion: `nn.CrossEntropyLoss()` This criterion combines :func:`nn.LogSoftmax` and :func:`nn.NLLLoss` in one single class. It is useful when training a classification problem with `C` classes.

```
In [39]: # Freeze parameters so we don't backprop through them
         for param in model_transfer.parameters():
             param.requires_grad = False
         # Replace the last fully connected layer with a Linnear layer with 133 out features
         model_transfer.fc = nn.Linear(2048, 133)
         if use_cuda:
             model_transfer = model_transfer.cuda()
```

```
In [40]: import time
```

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 [41]: 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 [55]: # train the model
         model_transfer =  train(12, loaders_transfer, model_transfer, optimizer_transfer, crite
```

```
Epoch: 1         Training Loss: 2.001459         Validation Loss: 1.153466
Validation loss decreased (inf --> 1.153466).  Saving model ...
Epoch: 2         Training Loss: 1.726210         Validation Loss: 1.256094
Epoch: 3         Training Loss: 1.600156         Validation Loss: 1.069591
Validation loss decreased (1.153466 --> 1.069591).  Saving model ...
Epoch: 4         Training Loss: 1.654100         Validation Loss: 0.916355
Validation loss decreased (1.069591 --> 0.916355).  Saving model ...
Epoch: 5         Training Loss: 1.473094         Validation Loss: 1.021815
Epoch: 6         Training Loss: 1.522992         Validation Loss: 1.124330
Epoch: 7         Training Loss: 1.443252         Validation Loss: 1.135807
Epoch: 8         Training Loss: 1.466599         Validation Loss: 1.334824
Epoch: 9         Training Loss: 1.406350         Validation Loss: 1.068416
Epoch: 10         Training Loss: 1.296084         Validation Loss: 1.460796
Epoch: 11         Training Loss: 1.372668         Validation Loss: 1.293226
Epoch: 12         Training Loss: 1.350619         Validation Loss: 1.199614
```

```
In [42]: # 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 [44]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Testing Loss Average: 1.051463


Test Accuracy: 80% (670/836)
```

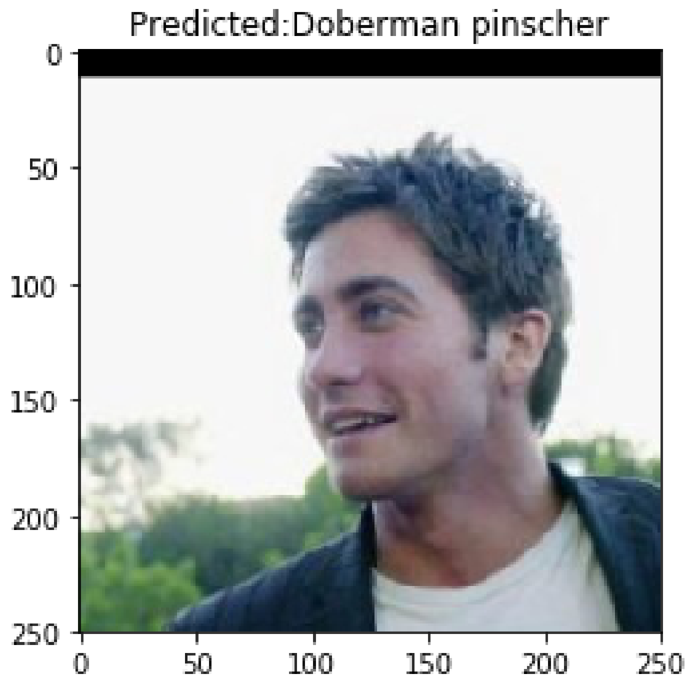
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

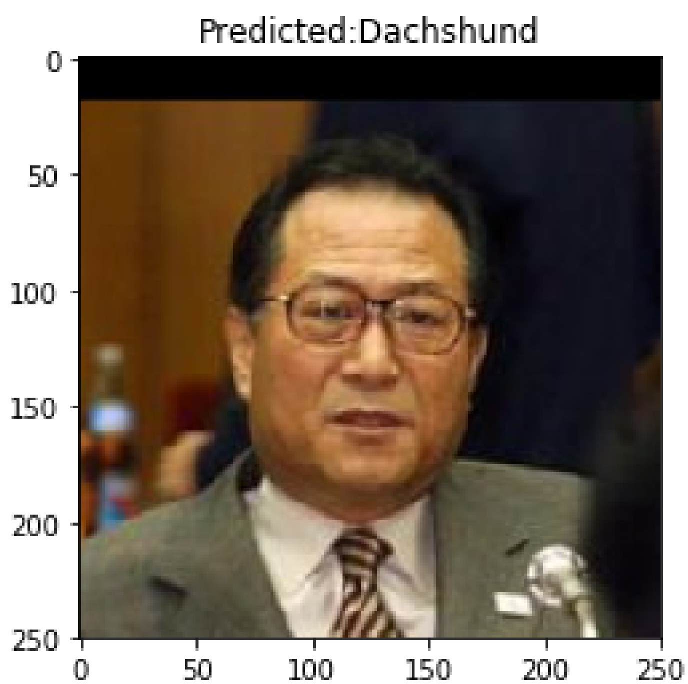
Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

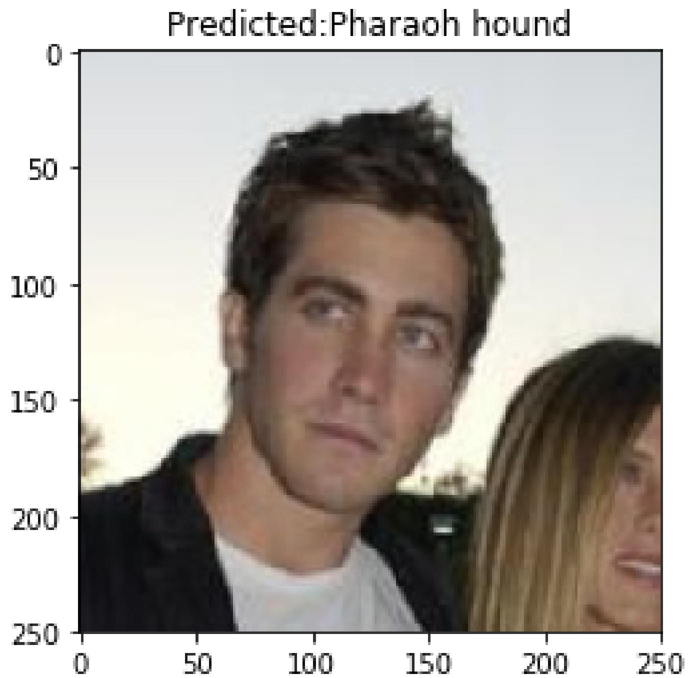
```
In [60]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

         # 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  image_datasets['train'].classes]

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

             # move model inputs to cuda, if GPU available
             if use_cuda:
                 image_tensor = image_tensor.cuda()

             # get sample outputs
             output = model_transfer(image_tensor)
             # convert output probabilities to predicted class
             _, preds_tensor = torch.max(output, 1)
             pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso

             return class_names[pred]
```

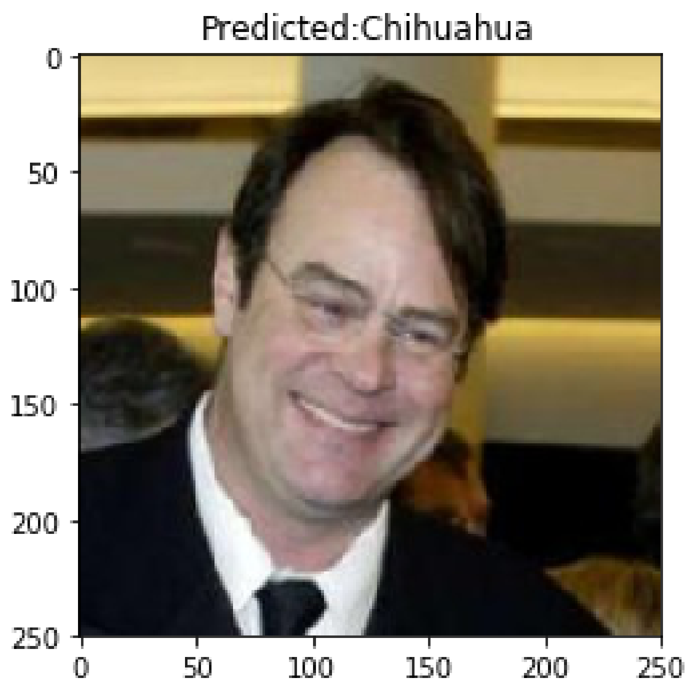
```
In [66]: def display_image(img_path, title="Title"):
             image = Image.open(img_path)
             plt.title(title)
             plt.imshow(image)
             plt.show()
```

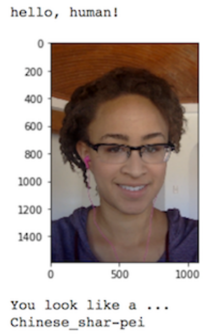
```
In [67]: import random

         # Try out the function
         for image in random.sample(list(human_files_short), 4):
             predicted_breed = predict_breed_transfer(image)
             display_image(image, title=f"Predicted:{predicted_breed}")
```


Predicted:Doberman pinscher
0
50
100
150
200
250
0
50
100
150
200
250


Predicted:Dachshund
0
50
100
150
200
250
0
50
100
150
200
250


Predicted:Pharaoh hound
0
50
100
150
200
250
0
50
100
150
200
250


Predicted:Chihuahua
0
50
100
150
200
250
0
50
100
150
200
250



Sample Human Output

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 [103]: ### TODO: Write your algorithm.
          ### Feel free to use as many code cells as needed.

          def run_app(img_path):
              # check if image has juman faces:
              if (haar_face_detector(img_path)):
                  print("Hello Human!")
                  predicted_breed = predict_breed_transfer(img_path)
                  display_image(img_path, title=f"Predicted:{predicted_breed}")

                  print("You look like a ...")
                  print(predicted_breed.upper())
              # check if image has dogs:
              elif dog_detector(img_path):
                  print("Hello Doggie!")
                  predicted_breed = predict_breed_transfer(img_path)
                  display_image(img_path, title=f"Predicted:{predicted_breed}")

                  print("Your breed is most likley ...")
                  print(predicted_breed.upper())
              else:
                  print("Oh, we're sorry! We couldn't detect any dog or human face in the image.
```

```
        display_image(img_path, title="...")
        print("Try another!")
    print("\n")
```

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.

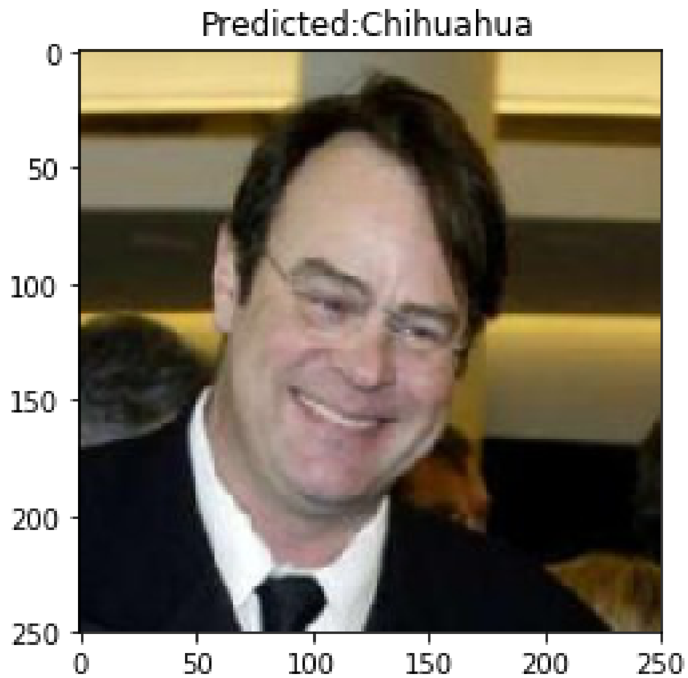
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

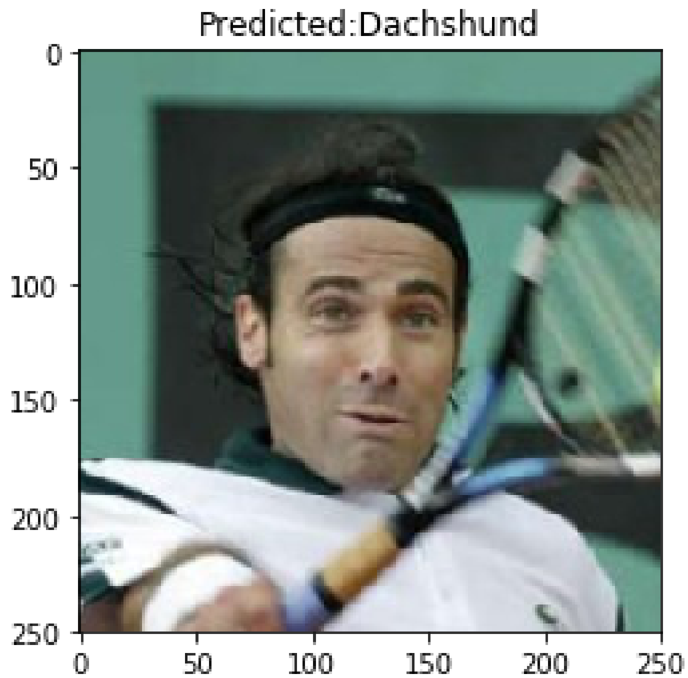
Some improvements: - Fine tune the model to give a better accuracy - Return the top N predicted classes and their probabilities, not just one class - Serve this function as an API (Flask, AWS, etc.) - Handle better the case when there are multiple dogs/humans or dogs and humans in an image - Benckmark different models, opimizers and loss functions, as well as different input image sizes. - Clean up code and make it more modular

```
In [104]: ## 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)
```

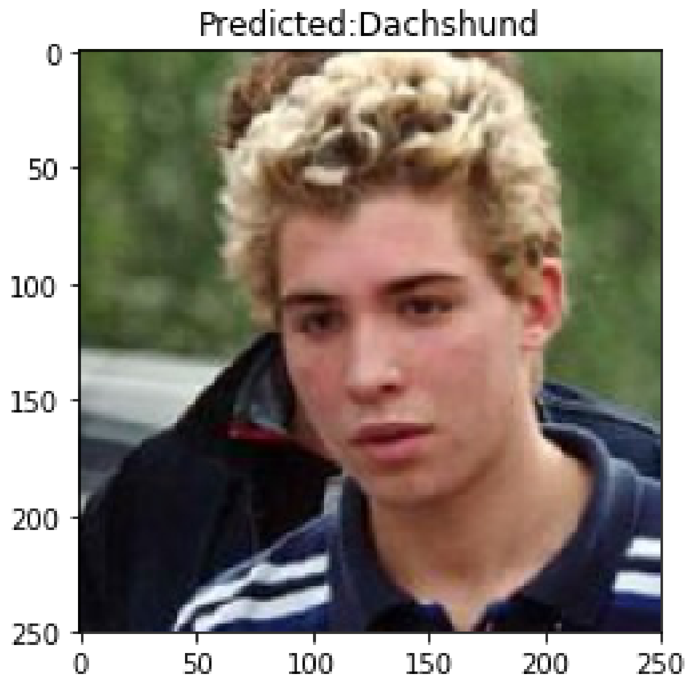
```
Hello Human!
```



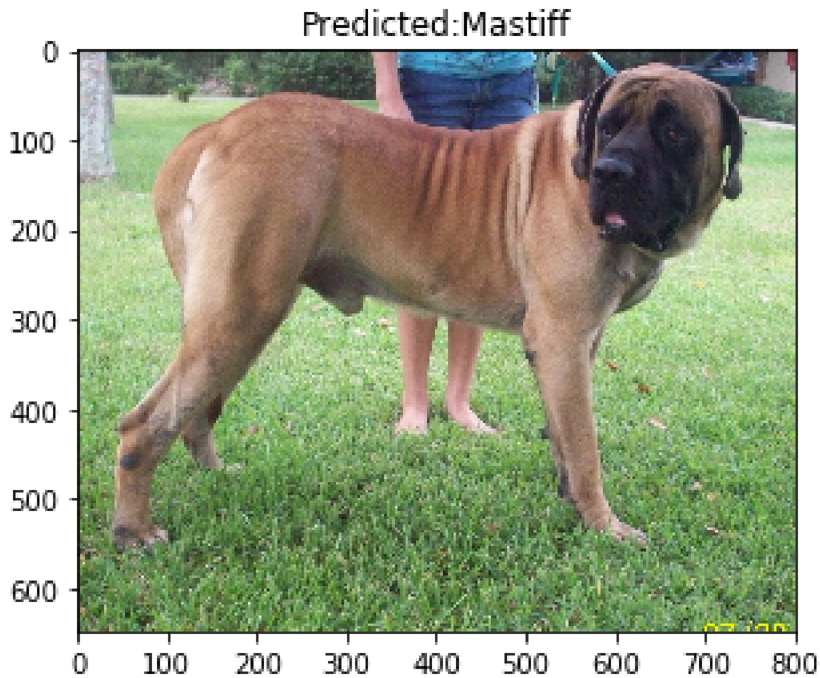

```
You look like a ...
CHIHUAHUA


Hello Human!
```



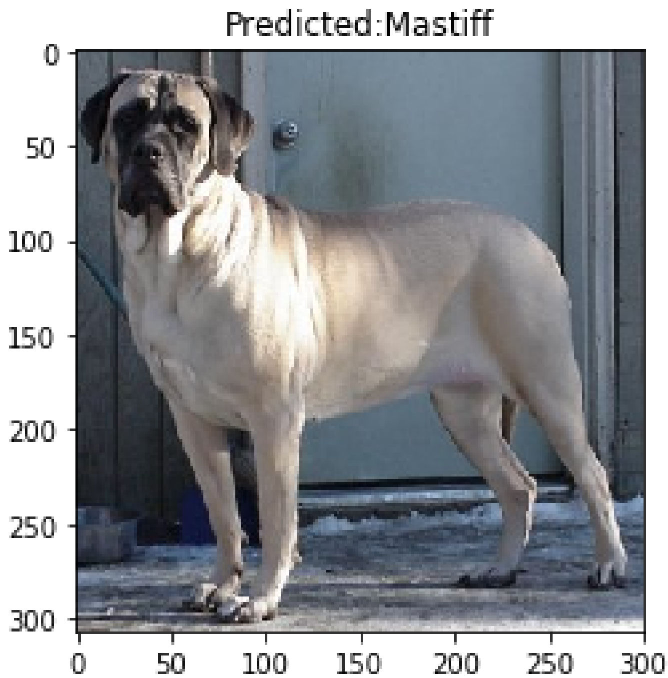

```
You look like a ...
DACHSHUND


Hello Human!
```



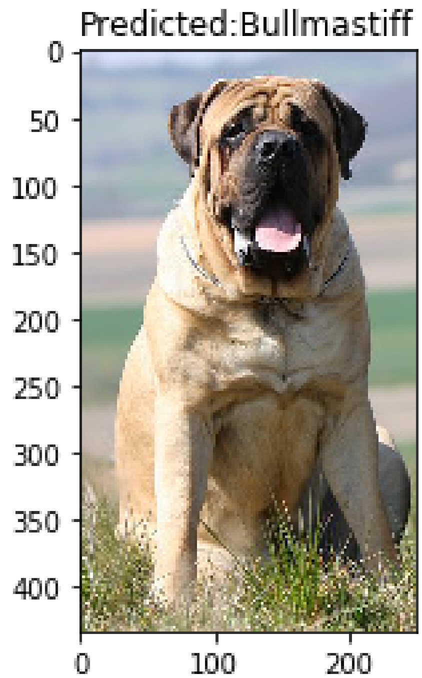

```
You look like a ...
DACHSHUND


Hello Doggie!
```




```
Your breed is most likley ...
MASTIFF


Hello Doggie!
```



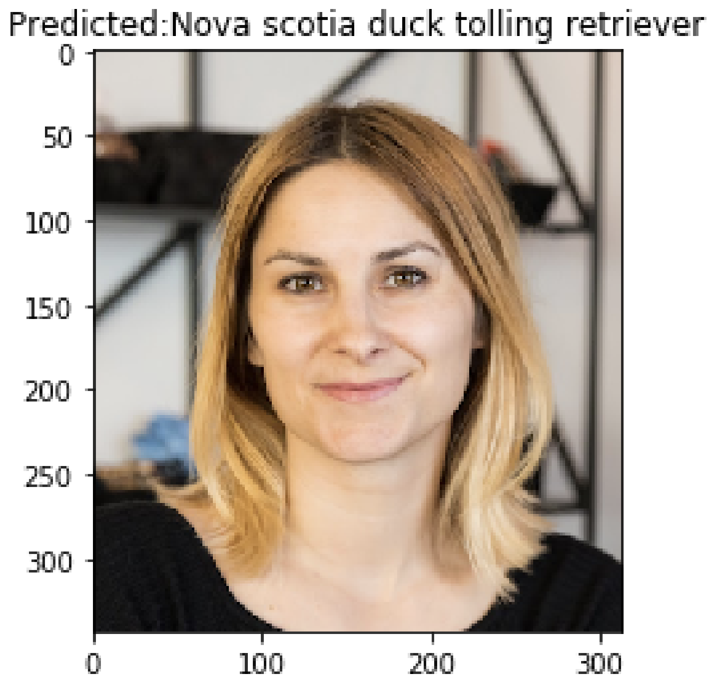

```
Your breed is most likley ...
MASTIFF


Hello Doggie!
```




```
Your breed is most likley ...
BULLMASTIFF
```

```
In [105]: run_app("moi.png")
```

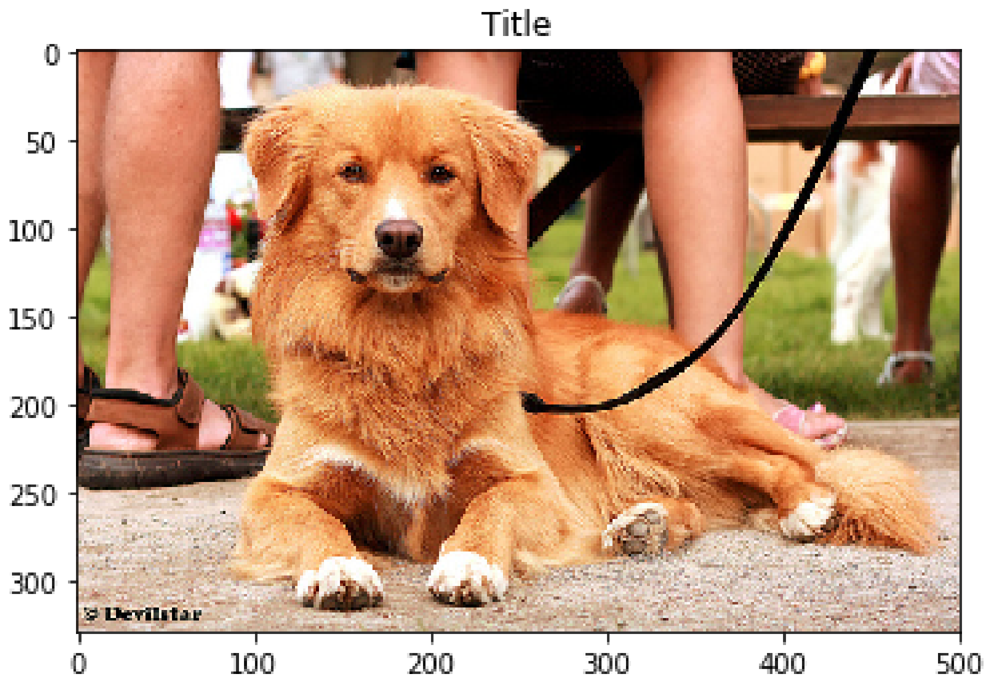
```
Hello Human!
```




```
You look like a ...
NOVA SCOTIA DUCK TOLLING RETRIEVER
```

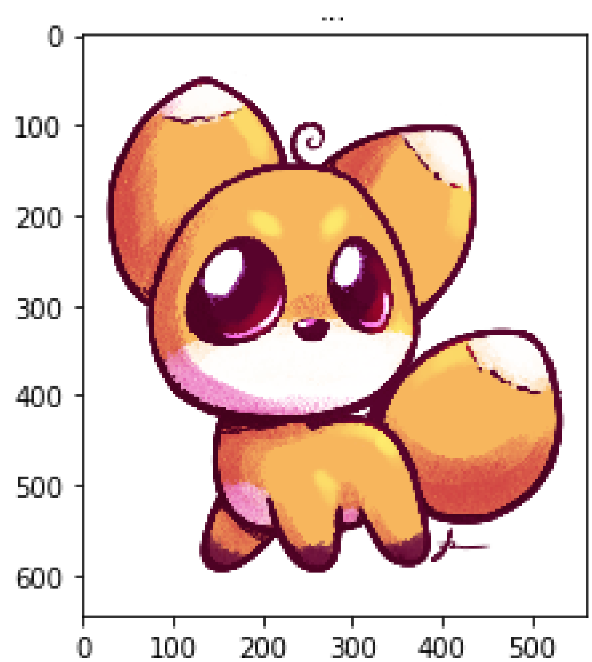
```
In [106]: import os, random
          dog_dir = "/data/dog_images/train/112.Nova_scotia_duck_tolling_retriever"
          nova_scotia_retriever_sample = os.path.join(dog_dir, random.choice(os.listdir(dog_dir)
          print(nova_scotia_retriever_sample)
          display_image(nova_scotia_retriever_sample)
```

```
/data/dog_images/train/112.Nova_scotia_duck_tolling_retriever/Nova_scotia_duck_tolling_retriever
```



In [107]: run_app("drawing.png")

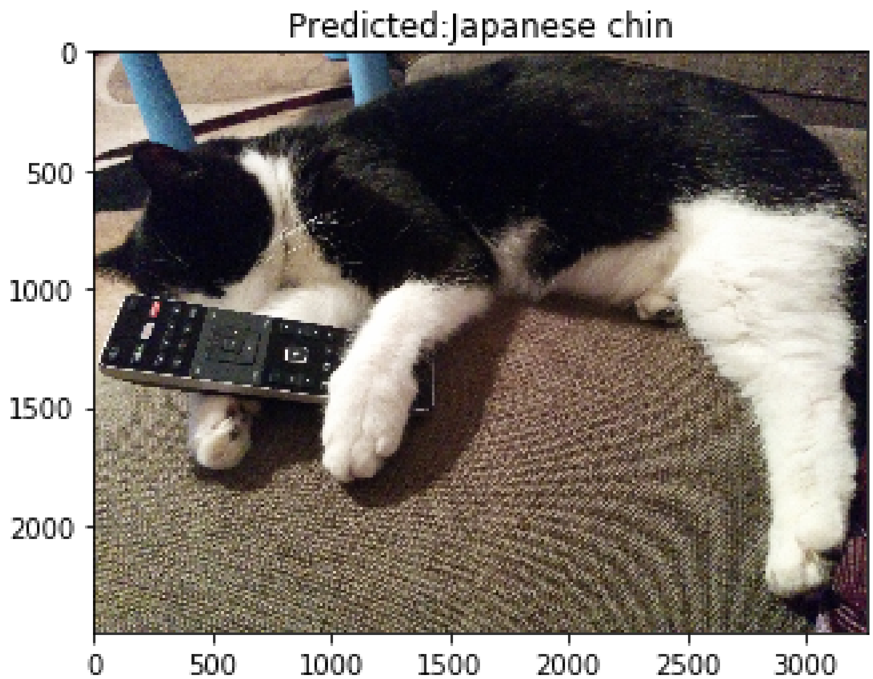
```
Oh, we're sorry! We couldn't detect any dog or human face in the image.
```



```
Try another!
```

```
In [108]: run_app("luna.jpg")
```

```
Hello Human!
```




```
You look like a ...
JAPANESE CHIN
```

```
In [109]: dog_dir = "/data/dog_images/train/091.Japanese_chin"
          sample = os.path.join(dog_dir, random.choice(os.listdir(dog_dir)))
          print(sample)
          display_image(sample)
```

```
/data/dog_images/train/091.Japanese_chin/Japanese_chin_06178.jpg
```


```
In [ ]:
```