dog_app

August 23, 2020

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
        import numpy as np
        from glob import glob
        from PIL import Image
        import torchvision.transforms as transforms
        import torch
        import torchvision.models as models
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        import os
        from torchvision import datasets
        from collections import OrderedDict
        import json
        from PIL import ImageFile
        import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

        # 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.
```

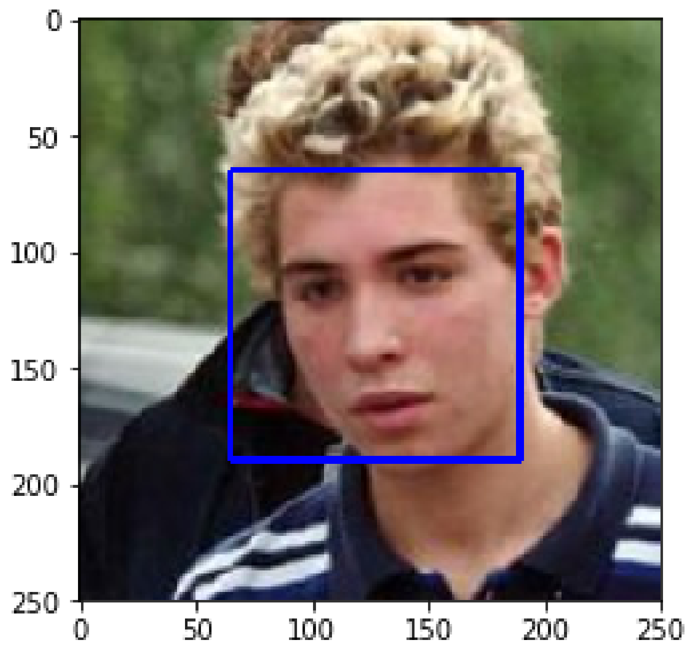
Step 1: Detect Humans

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]:
```python
#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[2])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

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

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

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

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

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



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.
H_human_detected = [face_detector(file) for file in human_files_short]
D_human_detected = [face_detector(file) for file in dog_files_short]
print(f'{100*sum(H_human_detected)/len(human_files_short):.1f}% of faces were detected i
print(f'{100*sum(D_human_detected)/len(human_files_short):.1f}% of faces were detected i
```

```
98.0% of faces were detected in 'human_files_short'.
17.0% of faces were detected in 'dog_files_short'.
```

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 [5]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

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 [6]:
```
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, 84322681.57it/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.

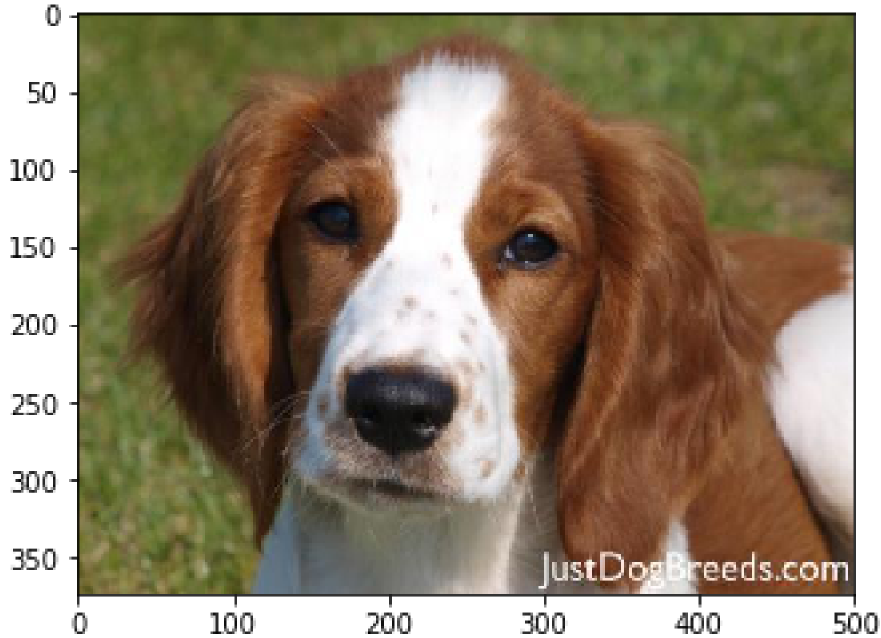
1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

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 [8]: from PIL import Image
        import torchvision.transforms as transforms

        def load_image(img_path):
            image = Image.open(img_path).convert('RGB')
            # resize to (244, 244) because VGG16 accept this shape
            in_transform = transforms.Compose([
                                transforms.Resize(size=(244, 244)),
                                transforms.ToTensor()]) # normalizaiton parameters from pytorch

            # discard the transparent, alpha channel (that's the :3) and add the batch dimension
            image = in_transform(image)[:3,:,:].unsqueeze(0)
            return image

        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
    '''

    # Load image with PIL package.
    im = Image.open(img_path).convert('RGB')

    # Define transforms needed for VGG16.
    transform_im = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])])

    # Transform image and reshape tensor.
    im_t = transform_im(im)
    im_t = im_t.view(-1,3,224,224)

    # Move image to cuda.
    if use_cuda:
        im_t = im_t.cuda()

    # Pass image through VGG network.
    VGG16.eval()
    output = VGG16(im_t)

    # Get the class with highest predicted probability.
    prob, pred = torch.topk(output,1)

    # Exctract and return the predicted class.
    pred_class = pred.item()
    return pred_class


# Display test image.
im_path = "images/Welsh_springer_spaniel_08203.jpg"
im = Image.open(im_path).convert('RGB')
plt.imshow(im)
plt.show()
```



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 [9]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            idx = VGG16_predict(img_path)
            return idx >= 151 and idx <= 268

        print(dog_detector(dog_files_short[0]))
        print(dog_detector(human_files_short[0]))
```

```
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 [10]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         def dog_detector_test(files):
             count = 0
             for i in files:
                 if (dog_detector(i) == True):
                     count += 1
             return ( '{:.2%} dog face detection'.format( count/100))

         print (dog_detector_test(human_files_short) + ' in human_files_short')
         print (dog_detector_test(dog_files_short) + ' in dog_files_short')
```

```
0.00% dog face detection in human_files_short
100.00% dog face detection in dog_files_short
```

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 [11]: ### (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

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!

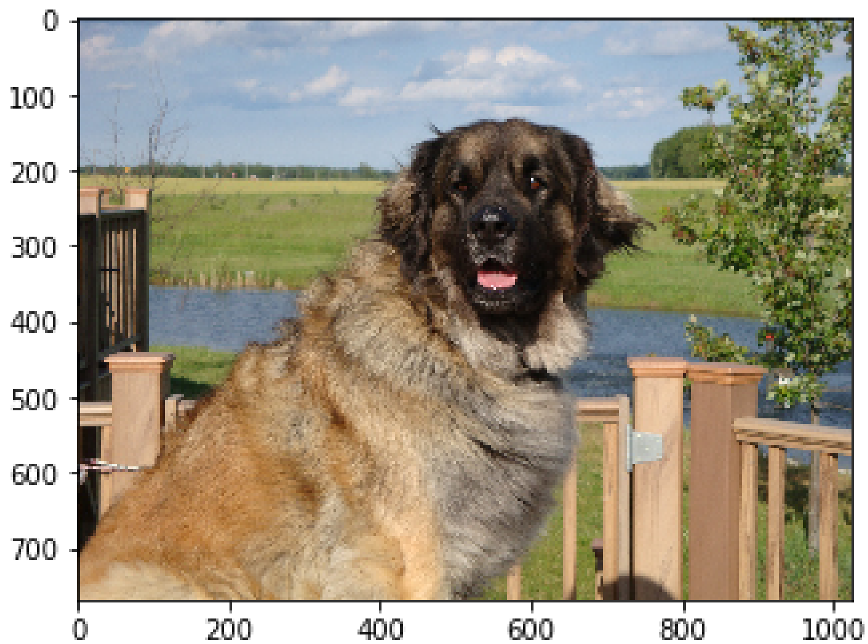
```
In [14]: import os
         from torchvision import datasets
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes

         # Test if all images from dataset can be succesfully loaded.
         for index, file in enumerate(np.array(glob("/data/dog_images/train/*/*"))):
             try:
                 # Try loading image.
                 Image.open(file).convert('RGB')
                 print(f'Tested up to index: {index}', end='\r')
             except Exception as e:
                 print(f'Problematic file: {index}.')
                 print(f'Reason for failure: {e}!!!')
```

```
Tested up to index: 6679
```

```
In [15]: truncated_file = np.array(glob("/data/dog_images/*/*/*"))[2271]
         print(f'Corrupted F path: \'{truncated_file}\'.')
```

```
ImageFile.LOAD_TRUNCATED_IMAGES = True
print('dog of the training dataset')
im = Image.open(truncated_file).convert('RGB')
plt.imshow(im)
plt.show()
```

```
Corrupted F path: '/data/dog_images/train/098.Leonberger/Leonberger_06571.jpg'.
dog of the training dataset
```




```
In [16]: # Paths to train  valid test sets.
         root_dir = '/data/dog_images/'
         train_dir = root_dir + 'train/'
         valid_dir = root_dir + 'valid/'
         test_dir = root_dir + 'test/'

         calculate_trans = transforms.Compose([
             transforms.Resize(256),
             transforms.CenterCrop(224),
             transforms.ToTensor()])

         calculate_data = datasets.ImageFolder(train_dir, transform = calculate_trans)
         calculate_loader = torch.utils.data.DataLoader(calculate_data, batch_size=1, shuffle=Fa

         means = list()
         stds = list()
```

```
for batch, (image, label) in enumerate(calculate_loader):
    arr = np.array(image)
    mean = np.mean(arr, axis = (0,2,3))
    std = np.std(arr, axis = (0,2,3))
    means.append(mean)
    stds.append(std)
print(f'The mean of training dataset: {np.mean(means, axis = 0)}')
print(f'The std of training dataset: {np.mean(stds, axis = 0)}')
```

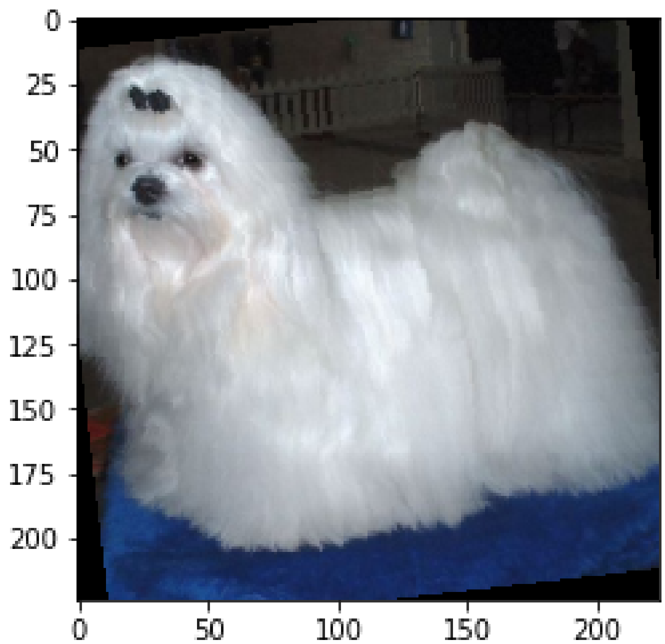
```
The mean of training dataset: [ 0.48640606  0.45601946  0.39183509]
The std of training dataset: [ 0.23047173  0.22563872  0.22351009]
```

```
In [17]: train_trans_sc = transforms.Compose([
             transforms.Resize(256),
             transforms.CenterCrop(224),
             transforms.RandomRotation(15),
             transforms.RandomHorizontalFlip(),
             transforms.ToTensor(),
             transforms.Normalize([0.5, 0.5, 0.5],[0.5, 0.5, 0.5])
             #transforms.Normalize([0.48640606,  0.45601946,  0.39183509],[ 0.23047173,  0.22563
             #transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
         ])

         test_valid_trans_sc = transforms.Compose([
             transforms.Resize(256),
             transforms.CenterCrop(224),
             transforms.ToTensor(),
             transforms.Normalize([0.5, 0.5, 0.5],[0.5, 0.5, 0.5])
             ])
         train_data_sc = datasets.ImageFolder(train_dir, transform = train_trans_sc)
         valid_data_sc = datasets.ImageFolder(valid_dir, transform = test_valid_trans_sc)
         test_data_sc = datasets.ImageFolder(test_dir, transform = test_valid_trans_sc)

         collect_data = [train_data_sc, valid_data_sc, test_data_sc]
         train_loader_sc = torch.utils.data.DataLoader(train_data_sc, batch_size=100, shuffle=Tr
         valid_loader_sc = torch.utils.data.DataLoader(valid_data_sc, batch_size=100, shuffle=Tr
         test_loader_sc = torch.utils.data.DataLoader(test_data_sc, batch_size=100, shuffle=Fals

         loaders_scratch = {'train':train_loader_sc,'valid':valid_loader_sc,'test':test_loader_s
         print(f'Number of classes: {len(np.array(glob(train_dir + "*")))}.')
         print(f'Number of examples: \nTrain set: {len(train_data_sc)}, \nValidation set: {len(v
```

```
Number of classes: 133.
Number of examples:
Train set: 6680,
Validation set: 835,
```

Test set: 836.

```
In [18]: practice_loader = torch.utils.data.DataLoader(train_data_sc, batch_size=1, shuffle=True

         def show_transformations():
             data, target = next(iter(practice_loader))
             arr = np.array(data)
             image = arr.reshape(3,224,224).transpose(1,2,0)
             image = image*np.array([0.5, 0.5, 0.5]) + np.array([0.5, 0.5, 0.5])
             image[image < 0] = 0
             image[image > 1] = 1
             plt.imshow(image)
             plt.show()

         show_transformations()
```



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: here we have resizing picture to 256 pixels on the shorter side and cropping 224x224 pixels from the center of the image. Along with two transformations to increase the accuracy. After transforming image to a tensor, the values must be normalized with mean and standard deviation for each color channel. Then normalizing the values with mean = 0.5 and std = 0.5 for all color channels that transforms values to be within the range [-1,1].

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [19]: 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
                 self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
                 # convolutional layer
                 self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
                 # convolutional layer
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
                 # convolutional layer
                 self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
                 self.conv5 = nn.Conv2d(256, 512, 3, padding = 1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)

                 # linear layer
                 self.fc1 = nn.Linear(512*7*7, 500)
                 # linear layer
                 self.fc2 = nn.Linear(500, 500)
                 # linear layer
                 self.fc3 = nn.Linear(500, 133)
                 # dropout layer (p=0.5)
                 self.dropout = nn.Dropout(0.5)

             def forward(self, x):
                 ## Define forward behavior
                 # add sequence of convolutional and max pooling layers
                 x = self.pool(F.relu(self.conv1(x)))
                 x = self.pool(F.relu(self.conv2(x)))
                 x = self.pool(F.relu(self.conv3(x)))
                 x = self.pool(F.relu(self.conv4(x)))
                 x = self.pool(F.relu(self.conv5(x)))

                 # flatten image input, size is depth of image in previous maxpool*depth of conv
                 x = x.view(-1, 512*7*7)


                 x = self.dropout(F.relu(self.fc1(x)))
```

```
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.fc3(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, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(256, 512, 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=25088, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=500, bias=True)
  (fc3): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.5)
)
```

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

Answer: the architecture has five convolutional layers. starts witht 32 filters and finish with 512. Each filter has size = 3, stride = 1 and padding = 1. Then a ReLU activation function applied after that max pooling layer with kernel size = 2 and stride = 2, to decreases the size of image to half.

Moreover, three fully connected layers that start with 25088 input and finish with 133 output to match the number of dog breed classes. after the first two layers comes the ReLU activation function along with dropout with probability p = 0.5. The last layer has no activation function.

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 [20]: import torch.optim as optim

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

```
### TODO: select optimizer
optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.0001)
```

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 [21]: 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
             #print_current_time()

             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()
                 print(f'\nTraining epoch {epoch}...', end='\r')
                 for batch_idx, (data, target) in enumerate(loaders['train']):

                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     # Set gradients to 0
                     optimizer.zero_grad()
                     # Forward pass
                     output = model(data)
                     # Calculate loss
                     loss = criterion(output,target)
                     # Backward pass
                     loss.backward()
                     # take optimizer step
                     optimizer.step()
                     # Get average training loss
                     train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)


                 print(f'Evaluating epoch {epoch}...', end='\r')
```

```
            correct = 0.
            total = 0.


            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                # Forward pass
                with torch.no_grad():
                    output = model(data)
                # Calculate loss
                loss = criterion(output,target)
                # Get average validation loss
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_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().nump
                total += data.size(0)


            # print training/validation statistics
            print(f'Epoch: {epoch}.. Train Loss: {train_loss:.3f}... Valid Loss: {valid_los
            #print(f'\nTraining Accuracy: {100.*train_accuracy/ train_total:.2f}% ({train_a
            #print(f'\n')

            ## TODO: save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                print(f'...In epoch {epoch} the validation loss decreased from {valid_loss_
                # Save current state dict
                torch.save(model.state_dict(),save_path)
                # Update min. validation loss
                valid_loss_min = valid_loss


        # return trained model
        return model

    # Train the model.
    model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch, criterion_
```

```
Epoch: 1.. Train Loss: 4.889... Valid Loss: 4.882... Valid Accuracy: 1.08% (9/835)...
```

```
...In epoch 1 the validation loss decreased from inf to 4.881991. Model saved...

Epoch: 2.. Train Loss: 4.874... Valid Loss: 4.852... Valid Accuracy: 1.20% (10/835)...
...In epoch 2 the validation loss decreased from 4.881991 to 4.851633. Model saved...

Epoch: 3.. Train Loss: 4.830... Valid Loss: 4.765... Valid Accuracy: 2.28% (19/835)...
...In epoch 3 the validation loss decreased from 4.851633 to 4.764626. Model saved...

Epoch: 4.. Train Loss: 4.745... Valid Loss: 4.686... Valid Accuracy: 2.63% (22/835)...
...In epoch 4 the validation loss decreased from 4.764626 to 4.685556. Model saved...

Epoch: 5.. Train Loss: 4.680... Valid Loss: 4.599... Valid Accuracy: 3.47% (29/835)...
...In epoch 5 the validation loss decreased from 4.685556 to 4.599264. Model saved...

Epoch: 6.. Train Loss: 4.602... Valid Loss: 4.518... Valid Accuracy: 3.47% (29/835)...
...In epoch 6 the validation loss decreased from 4.599264 to 4.518139. Model saved...

Epoch: 7.. Train Loss: 4.518... Valid Loss: 4.453... Valid Accuracy: 3.71% (31/835)...
...In epoch 7 the validation loss decreased from 4.518139 to 4.453176. Model saved...

Epoch: 8.. Train Loss: 4.439... Valid Loss: 4.356... Valid Accuracy: 4.91% (41/835)...
...In epoch 8 the validation loss decreased from 4.453176 to 4.356094. Model saved...

Epoch: 9.. Train Loss: 4.375... Valid Loss: 4.289... Valid Accuracy: 6.95% (58/835)...
...In epoch 9 the validation loss decreased from 4.356094 to 4.288790. Model saved...

Epoch: 10.. Train Loss: 4.343... Valid Loss: 4.259... Valid Accuracy: 5.39% (45/835)...
...In epoch 10 the validation loss decreased from 4.288790 to 4.258676. Model saved...

Epoch: 11.. Train Loss: 4.289... Valid Loss: 4.181... Valid Accuracy: 5.51% (46/835)...
...In epoch 11 the validation loss decreased from 4.258676 to 4.180696. Model saved...

Epoch: 12.. Train Loss: 4.239... Valid Loss: 4.169... Valid Accuracy: 6.71% (56/835)...
...In epoch 12 the validation loss decreased from 4.180696 to 4.169080. Model saved...

Epoch: 13.. Train Loss: 4.199... Valid Loss: 4.145... Valid Accuracy: 6.59% (55/835)...
...In epoch 13 the validation loss decreased from 4.169080 to 4.144951. Model saved...

Epoch: 14.. Train Loss: 4.186... Valid Loss: 4.111... Valid Accuracy: 6.95% (58/835)...
...In epoch 14 the validation loss decreased from 4.144951 to 4.111095. Model saved...

Epoch: 15.. Train Loss: 4.140... Valid Loss: 4.081... Valid Accuracy: 6.23% (52/835)...
...In epoch 15 the validation loss decreased from 4.111095 to 4.081482. Model saved...

Epoch: 16.. Train Loss: 4.093... Valid Loss: 4.023... Valid Accuracy: 6.71% (56/835)...
...In epoch 16 the validation loss decreased from 4.081482 to 4.022654. Model saved...

Epoch: 17.. Train Loss: 4.072... Valid Loss: 4.003... Valid Accuracy: 7.43% (62/835)...
```

```
...In epoch 17 the validation loss decreased from 4.022654 to 4.002845. Model saved...

Epoch: 18.. Train Loss: 4.053... Valid Loss: 3.981... Valid Accuracy: 8.02% (67/835)...
...In epoch 18 the validation loss decreased from 4.002845 to 3.980629. Model saved...

Epoch: 19.. Train Loss: 4.008... Valid Loss: 3.963... Valid Accuracy: 6.95% (58/835)...
...In epoch 19 the validation loss decreased from 3.980629 to 3.963425. Model saved...

Epoch: 20.. Train Loss: 4.004... Valid Loss: 3.937... Valid Accuracy: 7.78% (65/835)...
...In epoch 20 the validation loss decreased from 3.963425 to 3.936803. Model saved...

Epoch: 21.. Train Loss: 3.971... Valid Loss: 3.956... Valid Accuracy: 7.66% (64/835)...

Epoch: 22.. Train Loss: 3.933... Valid Loss: 3.897... Valid Accuracy: 8.74% (73/835)...
...In epoch 22 the validation loss decreased from 3.936803 to 3.896650. Model saved...

Epoch: 23.. Train Loss: 3.901... Valid Loss: 3.838... Valid Accuracy: 9.70% (81/835)...
...In epoch 23 the validation loss decreased from 3.896650 to 3.837609. Model saved...

Epoch: 24.. Train Loss: 3.870... Valid Loss: 3.843... Valid Accuracy: 8.74% (73/835)...

Epoch: 25.. Train Loss: 3.844... Valid Loss: 3.871... Valid Accuracy: 9.94% (83/835)...

Epoch: 26.. Train Loss: 3.808... Valid Loss: 3.880... Valid Accuracy: 9.70% (81/835)...

Epoch: 27.. Train Loss: 3.810... Valid Loss: 3.770... Valid Accuracy: 9.46% (79/835)...
...In epoch 27 the validation loss decreased from 3.837609 to 3.769869. Model saved...

Epoch: 28.. Train Loss: 3.773... Valid Loss: 3.761... Valid Accuracy: 11.14% (93/835)...
...In epoch 28 the validation loss decreased from 3.769869 to 3.760702. Model saved...

Epoch: 29.. Train Loss: 3.746... Valid Loss: 3.765... Valid Accuracy: 11.50% (96/835)...

Epoch: 30.. Train Loss: 3.721... Valid Loss: 3.738... Valid Accuracy: 11.50% (96/835)...
...In epoch 30 the validation loss decreased from 3.760702 to 3.737525. Model saved...
```

In [22]:
```python
# 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 [23]:
```python
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()
            data, target = data.cuda(), target.cuda()
        # Forward pass
        with torch.no_grad():
            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(f'\nTest Accuracy: {100.*correct/ total:.2f}% ({correct:.0f}/{total:.0f})')

test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Accuracy: 13.04% (109/836)
```

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 [24]: ## TODO: Specify data loaders
         train_trans_tr = transforms.Compose([
             transforms.Resize(256),
             transforms.CenterCrop(224),
             transforms.RandomRotation(15),
```

```
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])])

test_valid_trans_tr = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])])
train_data_tr = datasets.ImageFolder(train_dir, transform = train_trans_tr)
valid_data_tr = datasets.ImageFolder(valid_dir, transform = test_valid_trans_tr)
test_data_tr = datasets.ImageFolder(test_dir, transform = test_valid_trans_tr)

data_transfer = {'train':train_data_tr, 'valid':valid_data_tr, 'test':test_data_tr}
train_loader_tr = torch.utils.data.DataLoader(train_data_tr, batch_size=100, shuffle=Tr
valid_loader_tr = torch.utils.data.DataLoader(valid_data_tr, batch_size=100, shuffle=Tr
test_loader_tr = torch.utils.data.DataLoader(test_data_tr, batch_size=100, shuffle=Fals

loaders_transfer = {'train':train_loader_tr,'valid':valid_loader_tr,'test':test_loader_
```

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 [25]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.vgg16(pretrained=True)

for param in model_transfer.parameters():
    param.requires_grad = False
model_transfer.classifier
```

Out[25]:
```
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)
)
```

In [26]:
```
dog_breed_classifier = nn.Sequential(OrderedDict([
    ('0', nn.Linear(25088,4096)),
    ('1', nn.ReLU()),
```

```
        ('2', nn.Dropout(0.5)),
        ('3', nn.Linear(4096,4096)),
        ('4', nn.ReLU()),
        ('5', nn.Dropout(0.5)),
        ('6', nn.Linear(4096,133))
            ]))

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

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: VGG16 is one of most used models for transfer learning. Itt was trained on ImageNet to identify 1000 class of objects including dog breeds. Herein, I update the number of outputs tto match the number of classes resulted in Valid Accuracy of 70.78%

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 [27]: # Select loss function.
         criterion_transfer = nn.CrossEntropyLoss()

         # Select optimizer, only optimize parameters of the classifier.
         optimizer_transfer = optim.Adam(model_transfer.classifier.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 [28]: # train the model
         model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfer, criter

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

```
Epoch: 1.. Train Loss: 3.797... Valid Loss: 1.548... Valid Accuracy: 54.01% (451/835)...
...In epoch 1 the validation loss decreased from inf to 1.547915. Model saved...

Epoch: 2.. Train Loss: 2.150... Valid Loss: 1.253... Valid Accuracy: 61.80% (516/835)...
...In epoch 2 the validation loss decreased from 1.547915 to 1.252501. Model saved...

Epoch: 3.. Train Loss: 1.927... Valid Loss: 1.155... Valid Accuracy: 65.27% (545/835)...
...In epoch 3 the validation loss decreased from 1.252501 to 1.154505. Model saved...
```

```
Epoch: 4.. Train Loss: 1.755... Valid Loss: 1.266... Valid Accuracy: 66.11% (552/835)...

Epoch: 5.. Train Loss: 1.650... Valid Loss: 1.117... Valid Accuracy: 67.31% (562/835)...
...In epoch 5 the validation loss decreased from 1.154505 to 1.116535. Model saved...

Epoch: 6.. Train Loss: 1.651... Valid Loss: 1.111... Valid Accuracy: 68.62% (573/835)...
...In epoch 6 the validation loss decreased from 1.116535 to 1.110811. Model saved...

Epoch: 7.. Train Loss: 1.604... Valid Loss: 1.121... Valid Accuracy: 68.62% (573/835)...

Epoch: 8.. Train Loss: 1.594... Valid Loss: 1.092... Valid Accuracy: 71.38% (596/835)...
...In epoch 8 the validation loss decreased from 1.110811 to 1.092129. Model saved...

Epoch: 9.. Train Loss: 1.505... Valid Loss: 0.988... Valid Accuracy: 73.17% (611/835)...
...In epoch 9 the validation loss decreased from 1.092129 to 0.988107. Model saved...

Epoch: 10.. Train Loss: 1.507... Valid Loss: 1.070... Valid Accuracy: 70.42% (588/835)...
```

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

```
Test Accuracy: 74.76% (625/836)
```

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 [30]: ### 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 data_transfer['train'].classes]

         def predict_breed_transfer(img_path):
             im = Image.open(img_path).convert('RGB')

             # Define transforms needed for VGG16.
             transform_im = transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
```

```
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])])

        # Transform image and reshape tensor.
        im_t = transform_im(im)
        im_t = im_t.view(-1,3,224,224)

        if use_cuda:
            im_t = im_t.cuda()

        # Pass image through VGG network.
        model_transfer.eval()
        with torch.no_grad():
            output = model_transfer(im_t)

            # Get the class with highest predicted probability.
            prob, pred = torch.topk(output,1)

        # Exctract and return the predicted class.
        pred_class = pred.item()
        dog = class_names[pred_class]

        return pred_class, dog
```

Step 5: Write your Algorithm

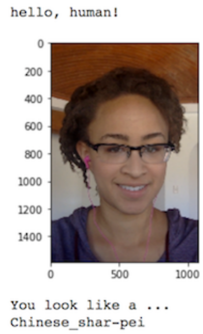
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is 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 [31]:
```
### 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
    if dog_detector(img_path):
        print('\nI definitely see a dog in this picture.')

        # Let's take a look at the picture
```



Sample Human Output

```
        im = Image.open(img_path).convert('RGB')
        plt.imshow(im)
        plt.show()

        # Predict dog breed.
        index, breed_class = predict_breed_transfer(img_path)
        print(f'I think it is a {breed_class}.')
        if breed_class == 'Australian shepherd':
            print('!!!The best dog ever!!!')


    elif face_detector(img_path):
        print('\nThis must be a human.')

        # Predict dog breed.
        index, breed_class = predict_breed_transfer(img_path)

        # Get image of predicted breed
        folder = data_transfer['train'].classes[index]
        dog = np.array(glob(f"/data/dog_images/train/{folder}/*"))[0]

        # Let's take a look at the picture
        im, (ax1, ax2) = plt.subplots(1, 2, figsize = (10,10))
        im1 = Image.open(img_path).convert('RGB')
        ax1.imshow(im1)
        im2 = Image.open(dog).convert('RGB')
        ax2.imshow(im2)
        plt.show()

        print(f'...Who looks like a {breed_class}. Do they look alike?')


    else:
        print('\nI don\'t see any dogs or humans here!!!')
        # Let's take a look at the picture
```

```
im = Image.open(img_path).convert('RGB')
plt.imshow(im)
plt.show()

# Predict VGG16 category
ImageNet_class = VGG16_predict(img_path)
category = imagenet[str(ImageNet_class)]

print(f'This looks more like a {category}...')
```

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!

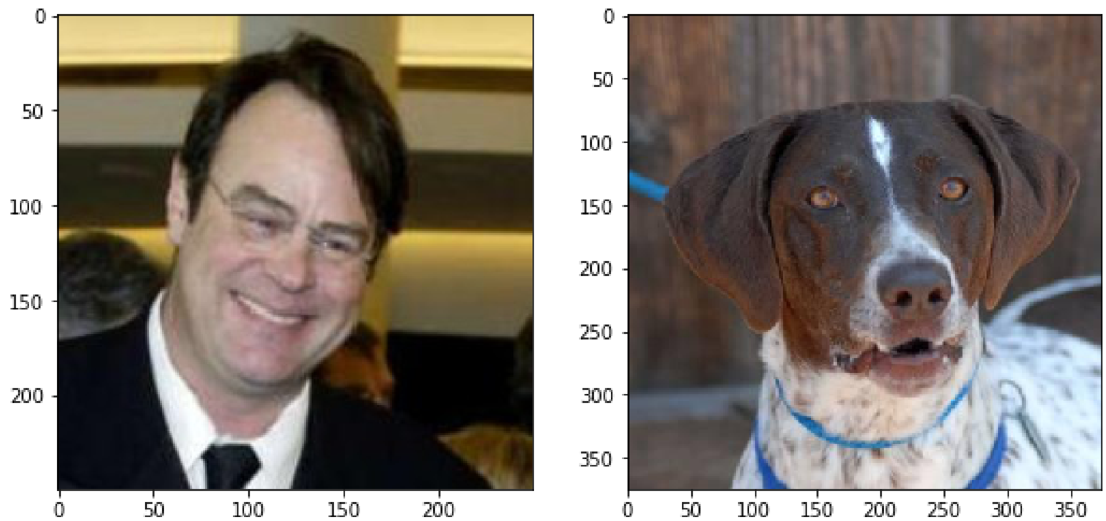
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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 model is just doing fine in classifying dog breeds. Ways to improve: - Try the data augmentation. - Enriching the datasett with more dog breeeds. - Adjusting the learning rate. - Increasing the number of hidden layers.

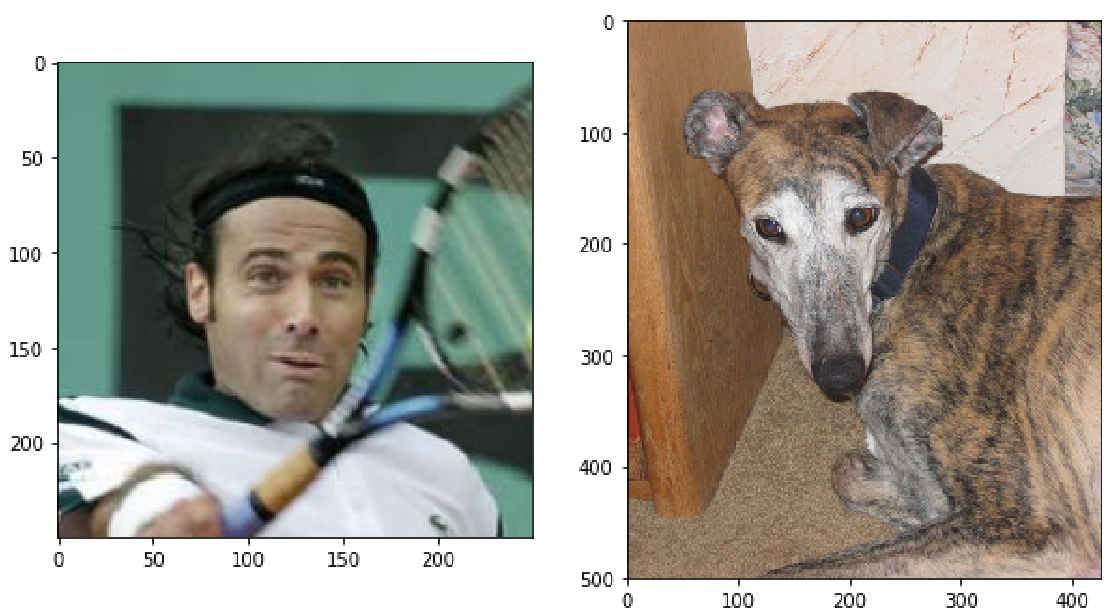
```
In [32]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[:3], dog_files[:3])):
             run_app(file)
         run_app('../Rosa.png')
         run_app('../Cheddar.png')
         run_app('../Jane.png')
         run_app('../Amy_Jake.png')
         run_app('../Terry.png')
         run_app('../num5.png')
```

```
This must be a human.
```



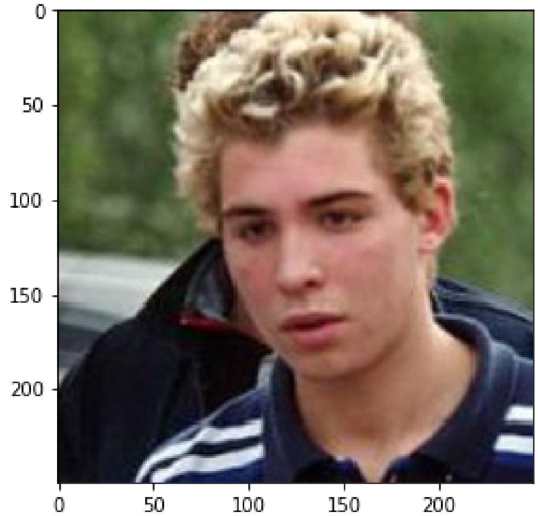
...Who looks like a German shorthaired pointer. Do they look alike?

This must be a human.



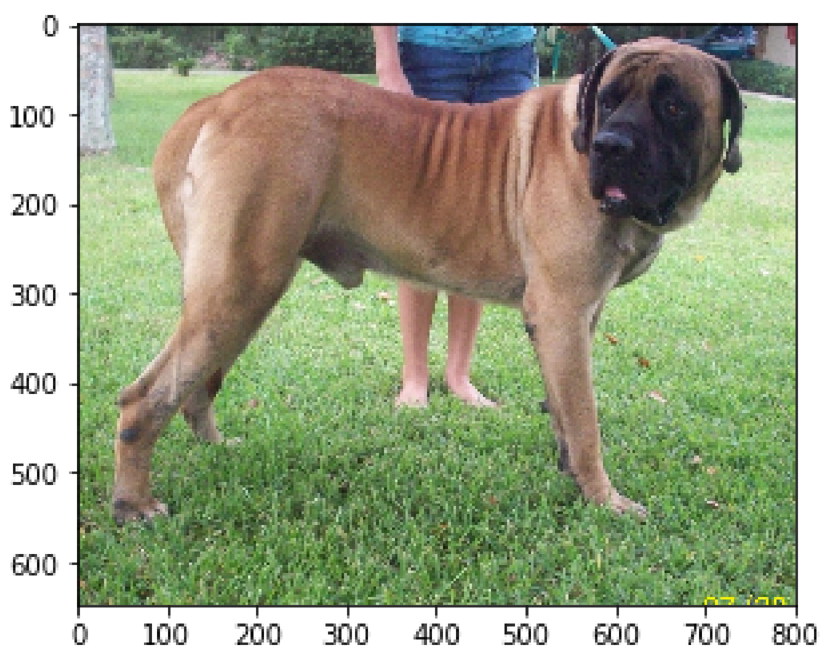
...Who looks like a Greyhound. Do they look alike?

This must be a human.

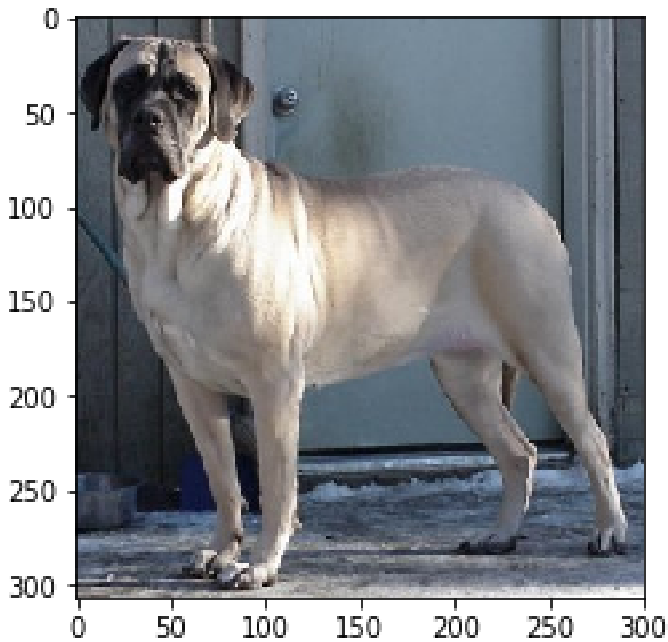


...Who looks like a Airedale terrier. Do they look alike?

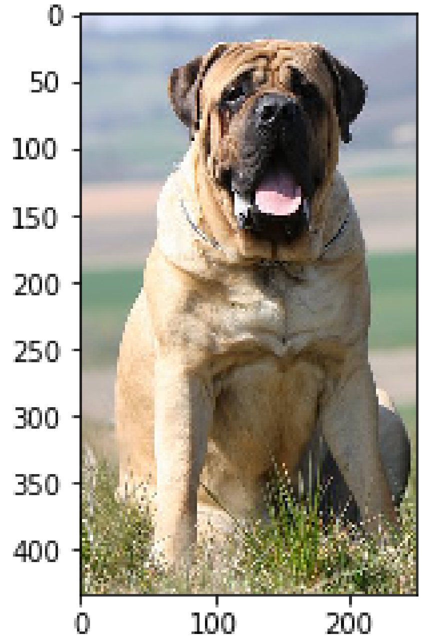
I definitely see a dog in this picture.



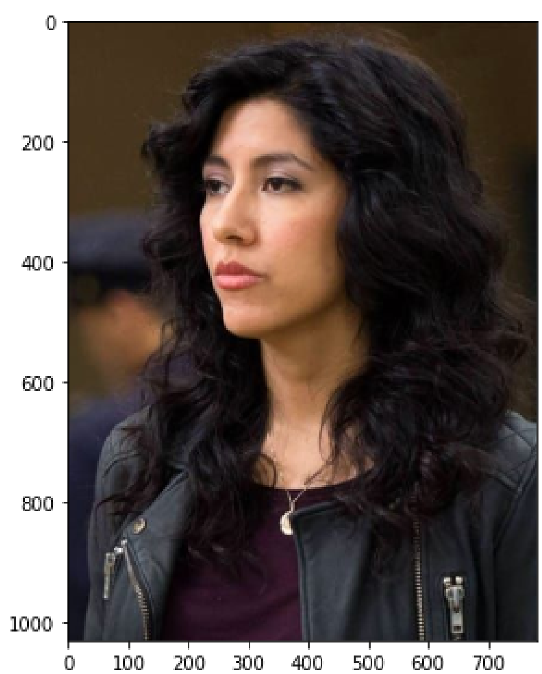
```
I think it is a Mastiff.

I definitely see a dog in this picture.
```




```
I think it is a Mastiff.

I definitely see a dog in this picture.
```

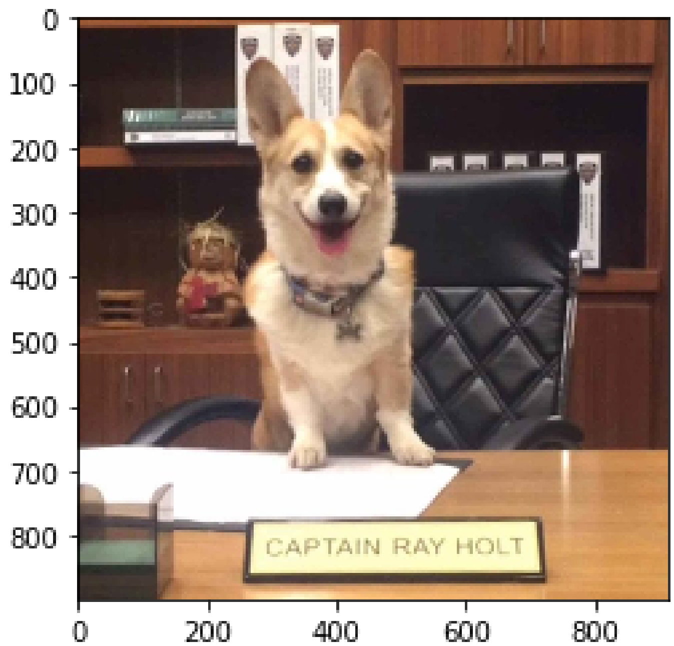



```
I think it is a Mastiff.

This must be a human.
```



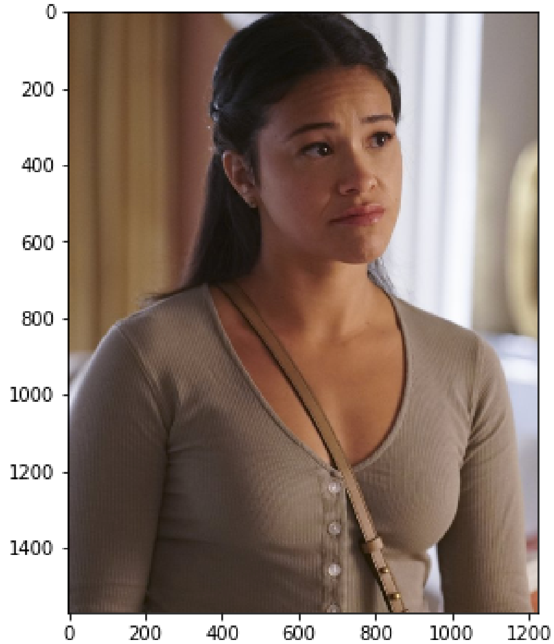
...Who looks like a English toy spaniel. Do they look alike?

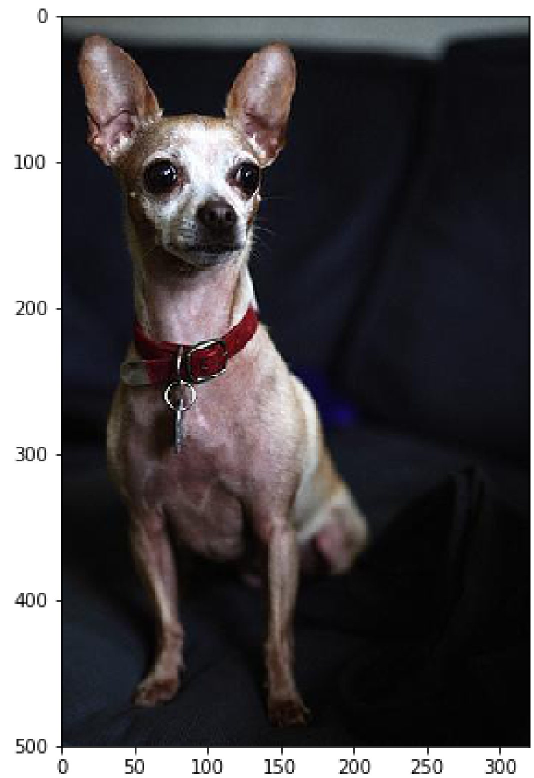
I definitely see a dog in this picture.



I think it is a Cardigan welsh corgi.

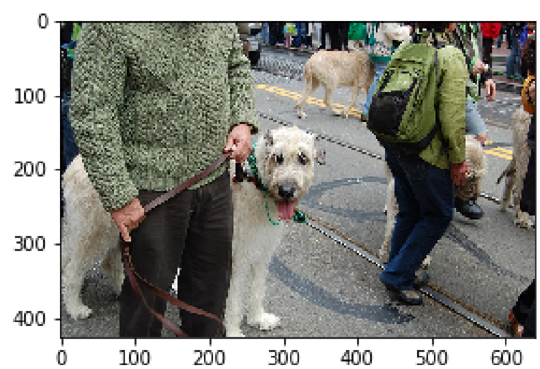
This must be a human.





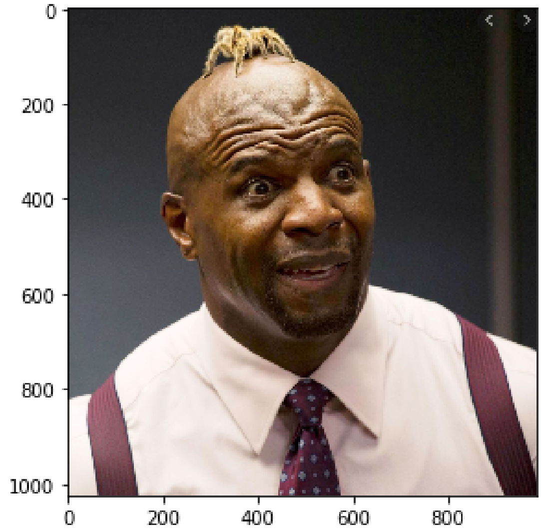
...Who looks like a Chihuahua. Do they look alike?

This must be a human.



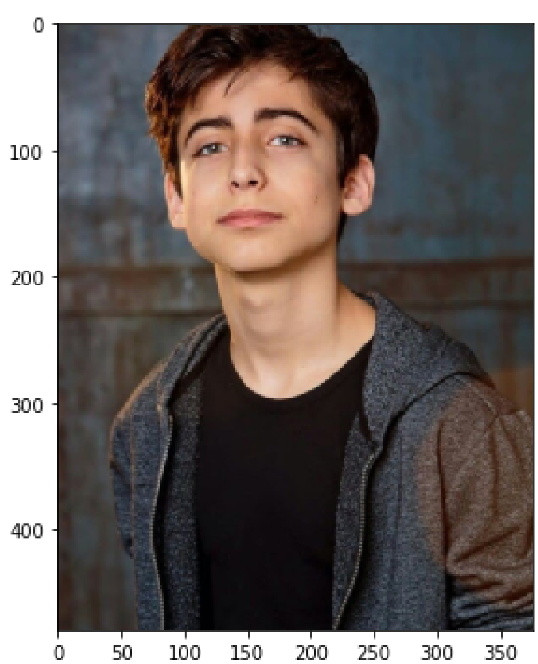
...Who looks like a Irish wolfhound. Do they look alike?

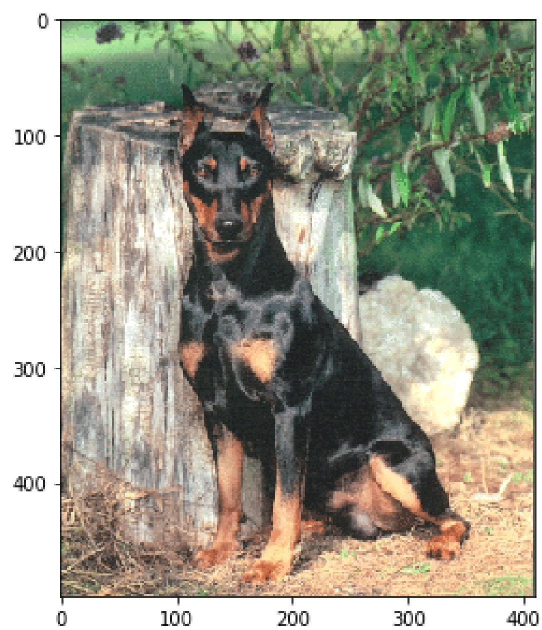
This must be a human.



...Who looks like a Pharaoh hound. Do they look alike?

This must be a human.





...Who looks like a German pinscher. Do they look alike?

```
In [ ]:
```