dog_app

February 21, 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: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- 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 [9]: 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

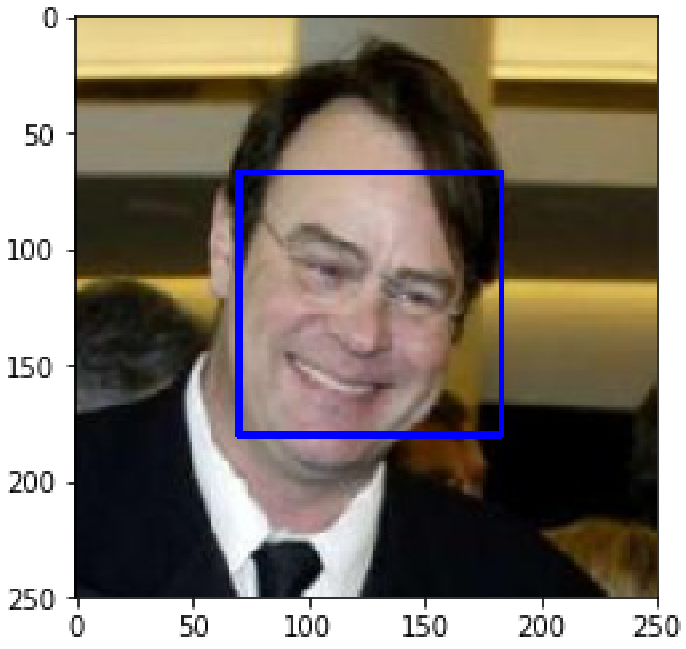
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [10]: import cv2
         import matplotlib.pyplot as plt
         %matplotlib inline

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

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

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

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

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

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

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

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



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 [11]: # 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 [12]: 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.
         human_faces_recognized = [1 for i in human_files_short if face_detector(i)]
         dog_faces_recognized = [1 for i in dog_files_short if face_detector(i)]
         print('%{} of human faces detected.'.format(len(human_faces_recognized)/len(human_files
         print('%{} of dog faces detected.'.format(len(dog_faces_recognized)/len(dog_files_short
```

```
%98.0 of human faces detected.
%17.0 of dog faces detected.
```

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 [13]: ### (Optional)
         ### TODO: Test performance of another face 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 [14]: 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()

         #ME
         print(VGG16)
         print(VGG16.classifier[6].in_features)
         print(VGG16.classifier[6].out_features)
```

```
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)
  )
)
4096
1000
```

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

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

         def process_image(img_path):
             ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
                 returns an Numpy array
             '''
```

```
    pil_im = Image.open(img_path, 'r')

    pil_im.thumbnail((256,256))
    pil_im=pil_im.crop((16,16,240,240))


    #ish(np.asarray(pil_im))
    np_image = np.array(pil_im)

    #couldn't make it the other way
    transformations = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0
                                                                                    (0.229, 0.224, 0.225))])
    np_image = transformations(np_image).float()
    return np_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
    '''

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

    if use_cuda:
        VGG16.cuda()

    VGG16.eval()

    with torch.no_grad():

        image = process_image(img_path)
        if use_cuda:
            image = image.type(torch.FloatTensor).cuda()
        #I do not understand why we shold do this why not 3 dimensions are not sufficen
        #print(image.shape)
        image = image.unsqueeze(0)
        #print(image.shape)

        output = VGG16.forward(image)
```

```
            #ps = torch.exp(logps)
            _,cls_idx=torch.max(output, 1)
            #probs_tensor, classes_tensor = ps.topk(1,dim=1)




        #return probs_tensor, classes_tensor
        return cls_idx.item()

    #imaj=process_image("a.jpg")
    #imaj.shape
    predict_that=VGG16_predict("/data/dog_images/test/022.Belgian_tervuren/Belgian_tervuren
    print(predict_that)
```

```
224
```

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 [32]: ### 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)

             return not (prediction<151 or prediction>268)
         dog_detector("/data/dog_images/test/022.Belgian_tervuren/Belgian_tervuren_01561.jpg")
```

```
Out[32]: True
```

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:

- %0.0 of human faces detected.
- %100.0 of dog faces detected.

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        humano_faces_recognized = [1 for i in human_files_short if dog_detector(i)]
        dogo_faces_recognized = [1 for i in dog_files_short if dog_detector(i)]
        print('%{} of human faces detected.'.format(len(humano_faces_recognized)/len(human_files
        print('%{} of dog faces detected.'.format(len(dogo_faces_recognized)/len(dog_files_short
```

```
%0.0 of human faces detected.
%100.0 of dog faces detected.
```

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 [10]: ### (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

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 `dogImages/train`, `dogImages/valid`, and `dogImages/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 [16]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms
         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         valid_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')
         train_transform = transforms.Compose([transforms.Resize(130),transforms.CenterCrop(128)
                                               transforms.RandomResizedCrop(128),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomRotation(50),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                          std=[0.229, 0.224, 0.225])])


         data_transform = transforms.Compose([transforms.Resize(130),transforms.CenterCrop(128),
                                              transforms.ToTensor(),
                                              transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                          std=[0.229, 0.224, 0.225])])




         train_data = datasets.ImageFolder(train_dir, transform=train_transform)
         valid_data = datasets.ImageFolder(valid_dir, transform=data_transform)
         test_data = datasets.ImageFolder(test_dir, transform=data_transform)

         #----I've Just added this to understand what is going on------------
```

```
batch_size = 20
num_workers=0

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)
loaders_scratch={}
loaders_scratch["train"]=train_loader
loaders_scratch["valid"]=valid_loader
loaders_scratch["test"]=test_loader

dataiter = iter(train_loader)
images, labels = dataiter.next()


#unnormalizing images 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


fig = plt.figure(figsize=(25, 4))
# plot the images in the batch, along with the corresponding labels
for idx in np.arange(20):
    c_image=im_convert(images[idx])
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    plt.imshow(c_image)
#----I've Just added this to understand what is going on------------
```

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

Answer: - I resized images on the dataset via torchvision.transform tool bu cropping. there were various sizes of images in the dataset that some of them are very big so 224 seemed reasonable at this time. - Yes, again with the help of transform tool.I made them(only training portion) flipped,cropped,rotated.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [17]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN



                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)

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

                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)

                 #my image size is 128*128, so after third pool layer it becomes 16*16
                 self.fc1 = nn.Linear(64 * 16 * 16, 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
                 x = x.view(-1, 64 * 16 * 16)
```

```
            # 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 do NOT have to modify the code below this line. #-#-#

    # instantiate the CNN
    model_scratch = Net()

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

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

Answer: - I start with 128*128 image as an input to keep model size a bit lower compare to VGG16, - create three Convolutional layer sequentially as feature extractors that double the feature maps each step, - then applied relu function for adding nonlinearity - finally applied pooling operation to lower the number of paramaters and discard the spatial information. - last two layers for classification. Just before first one, I flatten out all the information come from feature maps. - For last layer I have 133 output for each dog breeds - To prevent overfitting add droppout layer for FC layers

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [19]: 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 [20]: # the following import is required for training to be robust to truncated images
         from PIL import ImageFile
```

```
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

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


            optimizer.zero_grad()

            output = model(data)

            loss = criterion(output, target)

            loss.backward()
            optimizer.step()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass: 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)
            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch,
                train_loss,
                valid_loss
                ))

            ## TODO: save the model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.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


    # train the model
    model_scratch = train(40, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1        Training Loss: 4.883419         Validation Loss: 4066.713603
Validation loss decreased (inf --> 4066.713603).  Saving model ...
Epoch: 2        Training Loss: 4.858892         Validation Loss: 4040.409513
Validation loss decreased (4066.713603 --> 4040.409513).  Saving model ...
Epoch: 3        Training Loss: 4.792527         Validation Loss: 3951.604486
Validation loss decreased (4040.409513 --> 3951.604486).  Saving model ...
Epoch: 4        Training Loss: 4.664647         Validation Loss: 3824.049020
Validation loss decreased (3951.604486 --> 3824.049020).  Saving model ...
Epoch: 5        Training Loss: 4.579844         Validation Loss: 3768.377526
Validation loss decreased (3824.049020 --> 3768.377526).  Saving model ...
Epoch: 6        Training Loss: 4.538202         Validation Loss: 3743.325334
Validation loss decreased (3768.377526 --> 3743.325334).  Saving model ...
Epoch: 7        Training Loss: 4.515414         Validation Loss: 3720.292847
Validation loss decreased (3743.325334 --> 3720.292847).  Saving model ...
Epoch: 8        Training Loss: 4.495499         Validation Loss: 3686.268785
Validation loss decreased (3720.292847 --> 3686.268785).  Saving model ...
Epoch: 9        Training Loss: 4.473136         Validation Loss: 3694.879448
Epoch: 10        Training Loss: 4.450355         Validation Loss: 3669.316256
Validation loss decreased (3686.268785 --> 3669.316256).  Saving model ...
Epoch: 11        Training Loss: 4.432211         Validation Loss: 3711.634200
Epoch: 12        Training Loss: 4.404918         Validation Loss: 3625.468256
```

```
Validation loss decreased (3669.316256 --> 3625.468256).  Saving model ...
Epoch: 13        Training Loss: 4.380626        Validation Loss: 3631.659245
Epoch: 14        Training Loss: 4.364997        Validation Loss: 3588.462188
Validation loss decreased (3625.468256 --> 3588.462188).  Saving model ...
Epoch: 15        Training Loss: 4.341889        Validation Loss: 3588.553038
Epoch: 16        Training Loss: 4.334906        Validation Loss: 3573.654652
Validation loss decreased (3588.462188 --> 3573.654652).  Saving model ...
Epoch: 17        Training Loss: 4.301415        Validation Loss: 3582.538617
Epoch: 18        Training Loss: 4.271464        Validation Loss: 3536.978762
Validation loss decreased (3573.654652 --> 3536.978762).  Saving model ...
Epoch: 19        Training Loss: 4.274193        Validation Loss: 3503.114505
Validation loss decreased (3536.978762 --> 3503.114505).  Saving model ...
Epoch: 20        Training Loss: 4.240935        Validation Loss: 3666.617088
Epoch: 21        Training Loss: 4.234035        Validation Loss: 3542.543993
Epoch: 22        Training Loss: 4.200685        Validation Loss: 3597.661912
Epoch: 23        Training Loss: 4.207452        Validation Loss: 3439.505157
Validation loss decreased (3503.114505 --> 3439.505157).  Saving model ...
Epoch: 24        Training Loss: 4.184228        Validation Loss: 3460.990825
Epoch: 25        Training Loss: 4.151198        Validation Loss: 3449.332516
Epoch: 26        Training Loss: 4.126995        Validation Loss: 3432.598786
Validation loss decreased (3439.505157 --> 3432.598786).  Saving model ...
Epoch: 27        Training Loss: 4.103266        Validation Loss: 3453.349491
Epoch: 28        Training Loss: 4.088383        Validation Loss: 3406.095228
Validation loss decreased (3432.598786 --> 3406.095228).  Saving model ...
Epoch: 29        Training Loss: 4.077822        Validation Loss: 3380.612673
Validation loss decreased (3406.095228 --> 3380.612673).  Saving model ...
Epoch: 30        Training Loss: 4.043926        Validation Loss: 3453.760935
Epoch: 31        Training Loss: 4.049820        Validation Loss: 3361.085839
Validation loss decreased (3380.612673 --> 3361.085839).  Saving model ...
Epoch: 32        Training Loss: 4.017241        Validation Loss: 3361.049440
Validation loss decreased (3361.085839 --> 3361.049440).  Saving model ...
Epoch: 33        Training Loss: 3.985492        Validation Loss: 3322.024257
Validation loss decreased (3361.049440 --> 3322.024257).  Saving model ...
Epoch: 34        Training Loss: 3.969734        Validation Loss: 3447.742255
Epoch: 35        Training Loss: 3.942060        Validation Loss: 3301.541464
Validation loss decreased (3322.024257 --> 3301.541464).  Saving model ...
Epoch: 36        Training Loss: 3.960695        Validation Loss: 3296.066978
Validation loss decreased (3301.541464 --> 3296.066978).  Saving model ...
Epoch: 37        Training Loss: 3.918153        Validation Loss: 3333.022668
Epoch: 38        Training Loss: 3.902590        Validation Loss: 3269.852604
Validation loss decreased (3296.066978 --> 3269.852604).  Saving model ...
Epoch: 39        Training Loss: 3.891379        Validation Loss: 3322.333426
Epoch: 40        Training Loss: 3.887197        Validation Loss: 3228.919644
Validation loss decreased (3269.852604 --> 3228.919644).  Saving model ...
```

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 [21]: def test(loaders, model, criterion, use_cuda):

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

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

             print('Test Loss: {:.6f}\n'.format(test_loss))

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

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

```
Test Loss: 3.900254


Test Accuracy: 12% (104/836)
```

## 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 [22]: ## TODO: Specify data loaders
         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         valid_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')
         train_transform = transforms.Compose([transforms.Resize(250),transforms.CenterCrop(224)
                                               transforms.RandomResizedCrop(224),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomRotation(50),
                                               transforms.ToTensor()])
         """,transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))"""

         data_transform = transforms.Compose([transforms.Resize(250),transforms.CenterCrop(224),
                                              transforms.ToTensor()])
         """,transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))"""


         train_data = datasets.ImageFolder(train_dir, transform=train_transform)
         valid_data = datasets.ImageFolder(valid_dir, transform=data_transform)
         test_data = datasets.ImageFolder(test_dir, transform=data_transform)

         #----I've Just added this to understand what is going on------------

         batch_size = 20
         num_workers=0

         # prepare data loaders
         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                    num_workers=num_workers, shuffle=True)
         test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                                    num_workers=num_workers, shuffle=True)

         loaders_transfer={}
         loaders_transfer["train"]=train_loader
         loaders_transfer["valid"]=valid_loader
         loaders_transfer["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 [23]: import torchvision.models as models
         import torch.nn as nn



         ## TODO: Specify model architecture
         model_transfer = models.vgg16(pretrained=True)
         print(model_transfer)
         for param in model_transfer.features.parameters():
             param.requires_grad = False
         #number of dog classes in the dataset
         dog_classes=133
         n_inputs = model_transfer.classifier[6].in_features


         # new layers automatically have requires_grad = True
         last_layer = nn.Linear(n_inputs, dog_classes)
         model_transfer.classifier[6] = last_layer
         print(model_transfer)
         # check if CUDA is available
         use_cuda = torch.cuda.is_available()

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

```
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)
  )
)
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=133, bias=True)
  )
)
```

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

Answer: - I start with a pre-trained model VGG16 as a feature extractor. - to use its pre trained weights freeze them before training. - I replaced current last fully connected layer with new one that is more suitable with my 133 dog classes. - This VGG16 model usable for my needs because it trained on a huge dataset that also includes most of my dog breeds.So features it find would be compatible with my dataset.Only additional step is training fully connected layers.

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 [24]: import torch.optim as optim
         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.01)
```

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 [25]: # train the model
         n_epochs=20
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
         # 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 	Training Loss: 4.206079 	Validation Loss: 1788.600531
Validation loss decreased (inf --> 1788.600531).  Saving model ...
Epoch: 2 	Training Loss: 3.237751 	Validation Loss: 1261.074567
Validation loss decreased (1788.600531 --> 1261.074567).  Saving model ...
Epoch: 3 	Training Loss: 2.935025 	Validation Loss: 1150.476137
Validation loss decreased (1261.074567 --> 1150.476137).  Saving model ...
Epoch: 4 	Training Loss: 2.747016 	Validation Loss: 1010.027809
Validation loss decreased (1150.476137 --> 1010.027809).  Saving model ...
Epoch: 5 	Training Loss: 2.651780 	Validation Loss: 946.398115
Validation loss decreased (1010.027809 --> 946.398115).  Saving model ...
Epoch: 6 	Training Loss: 2.591071 	Validation Loss: 975.882382
Epoch: 7 	Training Loss: 2.508532 	Validation Loss: 924.886099
Validation loss decreased (946.398115 --> 924.886099).  Saving model ...
Epoch: 8 	Training Loss: 2.458711 	Validation Loss: 909.383541
Validation loss decreased (924.886099 --> 909.383541).  Saving model ...
Epoch: 9 	Training Loss: 2.426657 	Validation Loss: 903.601698
Validation loss decreased (909.383541 --> 903.601698).  Saving model ...
Epoch: 10 	Training Loss: 2.393345 	Validation Loss: 873.990644
Validation loss decreased (903.601698 --> 873.990644).  Saving model ...
Epoch: 11 	Training Loss: 2.326861 	Validation Loss: 852.366130
Validation loss decreased (873.990644 --> 852.366130).  Saving model ...
Epoch: 12 	Training Loss: 2.313775 	Validation Loss: 841.157330
Validation loss decreased (852.366130 --> 841.157330).  Saving model ...
Epoch: 13 	Training Loss: 2.281017 	Validation Loss: 851.368233
Epoch: 14 	Training Loss: 2.203907 	Validation Loss: 871.683766
Epoch: 15 	Training Loss: 2.238973 	Validation Loss: 851.991068
Epoch: 16 	Training Loss: 2.200655 	Validation Loss: 822.150648
Validation loss decreased (841.157330 --> 822.150648).  Saving model ...
Epoch: 17 	Training Loss: 2.219464 	Validation Loss: 806.875275
Validation loss decreased (822.150648 --> 806.875275).  Saving model ...
Epoch: 18 	Training Loss: 2.158723 	Validation Loss: 779.548876
Validation loss decreased (806.875275 --> 779.548876).  Saving model ...
Epoch: 19 	Training Loss: 2.165520 	Validation Loss: 809.705009
Epoch: 20 	Training Loss: 2.144737 	Validation Loss: 856.258096
```

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%.

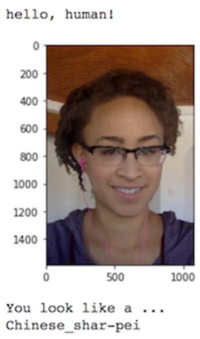
```
In [26]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.938510


Test Accuracy: 71% (600/836)
```

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 [29]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         #loaders_transfer["train"]





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

         def predict_breed_transfer(img_path):

             if use_cuda:
                 model_transfer.cuda()

             model_transfer.eval()

             with torch.no_grad():

                 image = process_image(img_path)
                 if use_cuda:
                     image = image.type(torch.FloatTensor).cuda()

                 image = image.unsqueeze(0)
                 #print(image.shape)

                 output = model_transfer.forward(image)

                 #ps = torch.exp(logps)
                 _,cls_idx=torch.max(output, 1)
                 #probs_tensor, classes_tensor = ps.topk(1,dim=1)
```



Sample Human Output

```
    #return probs_tensor, classes_tensor
    return class_names[cls_idx.item()]
predict_that=predict_breed_transfer("/data/dog_images/test/022.Belgian_tervuren/Belgian
#predict_that=predict_breed_transfer("/data/dog_images/test/001.Affenpinscher/Affenpins
print(predict_that)
```

```
Belgian tervuren
```

## 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 `dog_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 [30]: ### TODO: (Write your algorithm.
         ### Feel free to use as many code cells as needed.




         def run_app(img_path):
             print("\n\n")
             print("file name: {}".format(img_path))
             print("----------------------------------------------------------")
             if face_detector(img_path):
                 print("This human is look like:")
```

```
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            # find faces in image
            faces = face_cascade.detectMultiScale(gray)
            cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            plt.imshow(cv_rgb)
            plt.show()
            dog_like=predict_breed_transfer(img_path)
            return print(dog_like)
        if dog_detector(img_path):
            d_breed=predict_breed_transfer(img_path)
            print("This good dog's breed:")
            im=process_image(img_path)
            im=im_convert(im)
            plt.imshow(im)
            plt.show()
            return print(d_breed)
        else:
            print("no humans or dogs at this time")




        ## handle cases for a human face, dog, and neither
    #aa=run_app("/data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg")
```

## 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.

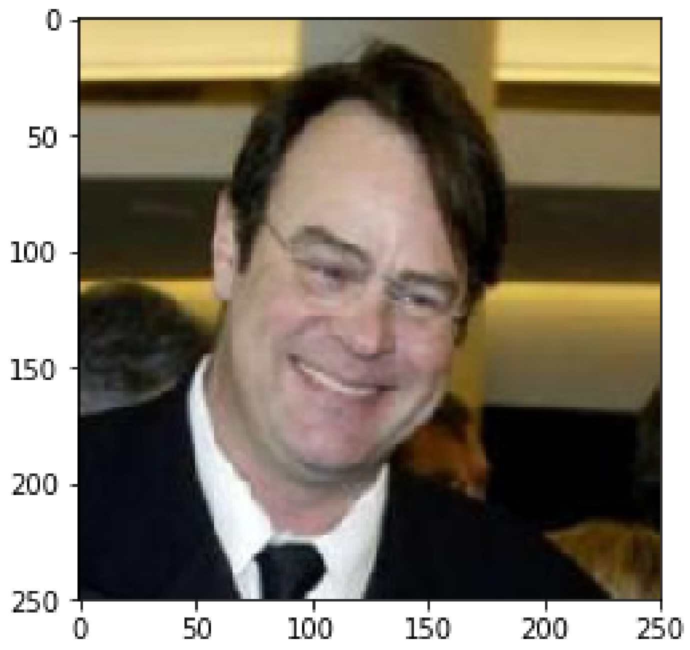
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: - My algorithm doesn't check for more then one person or different dog breeds in the image. - Normalizing the input image could be helpful. - Tuning VGG16 for not 1000 categories but only dog categories could be helpful. - There could be a option for use of Cuda or not. - My images are slowing the app. because they aren't resized.

```
In [33]: ## 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.
         my_human_files = np.array(glob("my_human/*"))
         my_dog_files = np.array(glob("my_dog/*"))
```

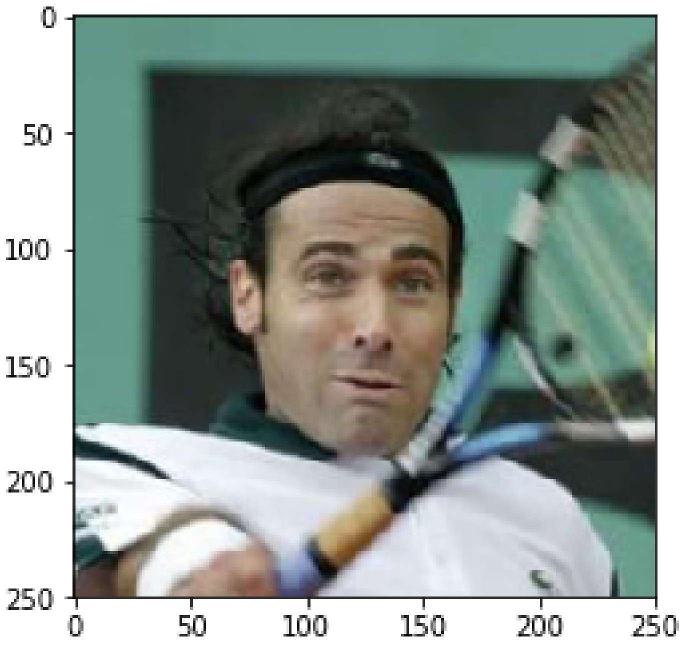
```
## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)

for my_file in np.hstack((my_human_files,my_dog_files)):
    run_app(my_file)
print('There are %d total human images.' % len(my_human_files))
print('There are %d total dog images.' % len(my_dog_files))
```

```
file name: /data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
---------------------------------------------------------
This human is look like:
```

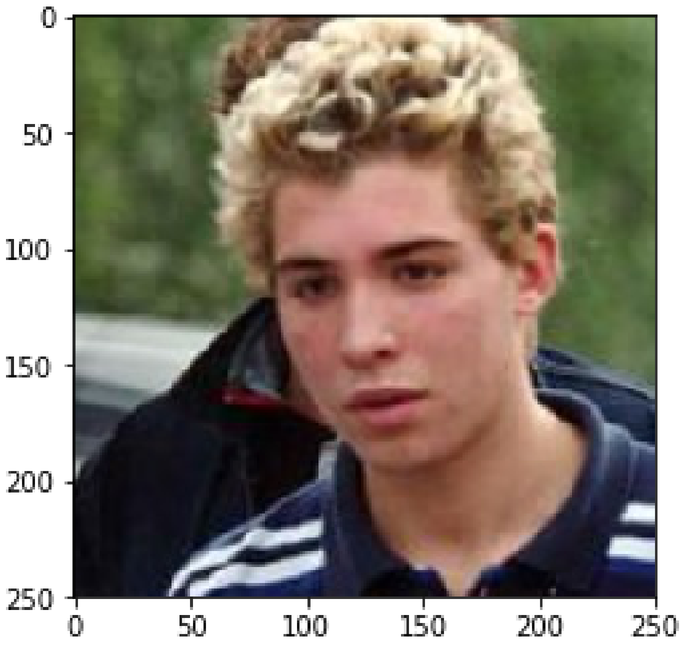


```
Ibizan hound
```

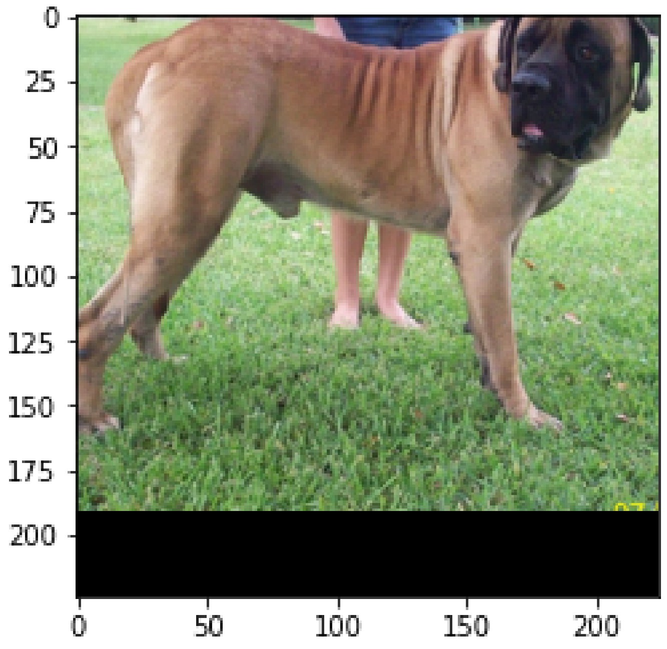
```
file name: /data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
---------------------------------------------------------
This human is look like:
```



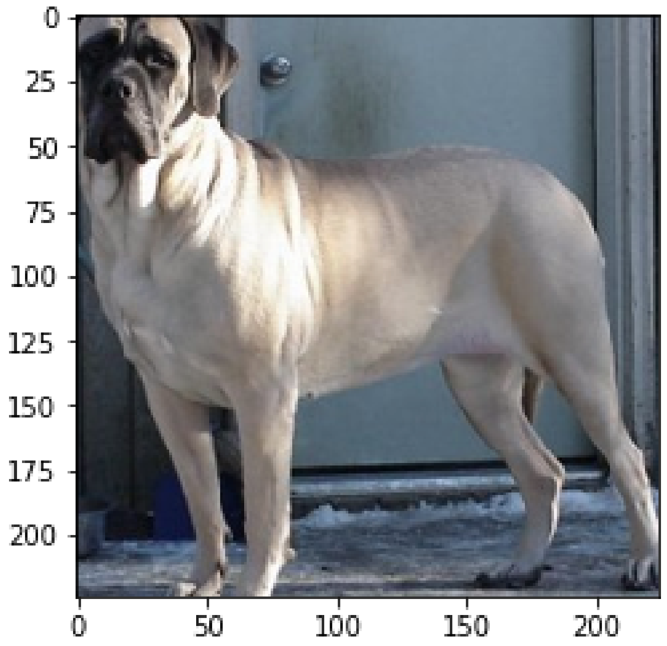
```
Bearded collie



file name: /data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg
---------------------------------------------------------
This human is look like:
```



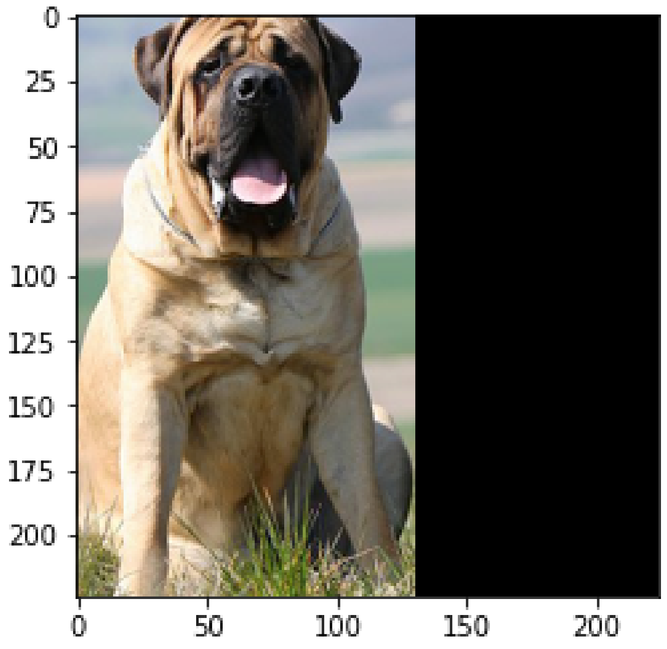
```
German pinscher



file name: /data/dog_images/train/103.Mastiff/Mastiff_06833.jpg
----------------------------------------------------------
This good dog's breed:
```



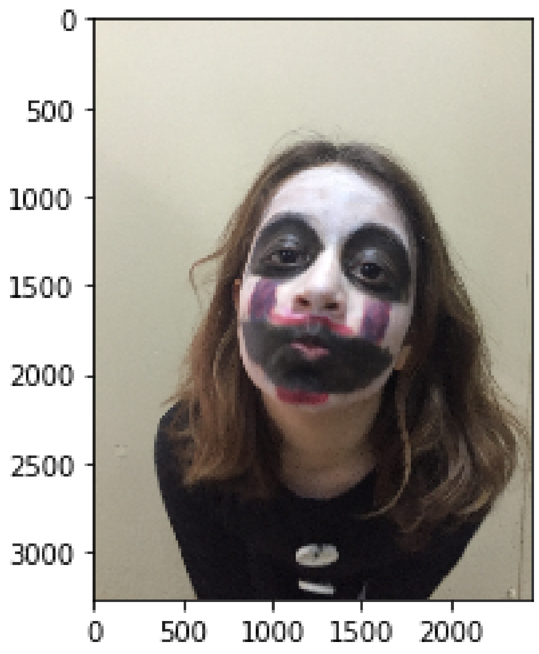
```
Bullmastiff



file name: /data/dog_images/train/103.Mastiff/Mastiff_06826.jpg
--------------------------------------------------------
This good dog's breed:
```



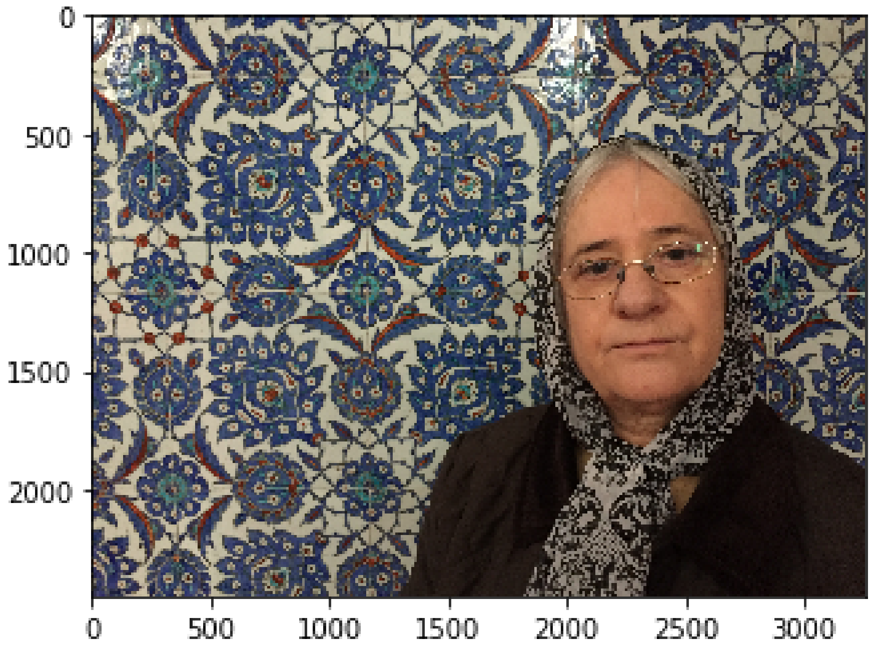
```
Bullmastiff



file name: /data/dog_images/train/103.Mastiff/Mastiff_06871.jpg
--------------------------------------------------------
This good dog's breed:
```



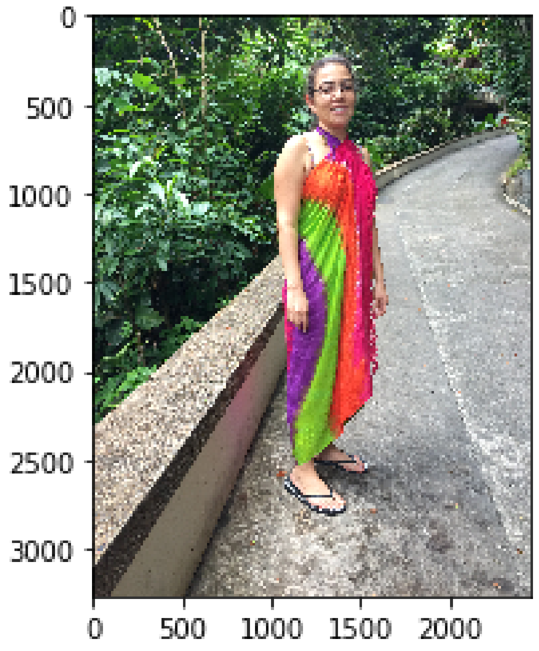
```
Bullmastiff



file name: my_human/2016-01-05 13.00.48.jpg
-------------------------------------------------------
This human is look like:
```



```
English toy spaniel



file name: my_human/2015-11-13 13.41.54.jpg
-------------------------------------------------------
This human is look like:
```

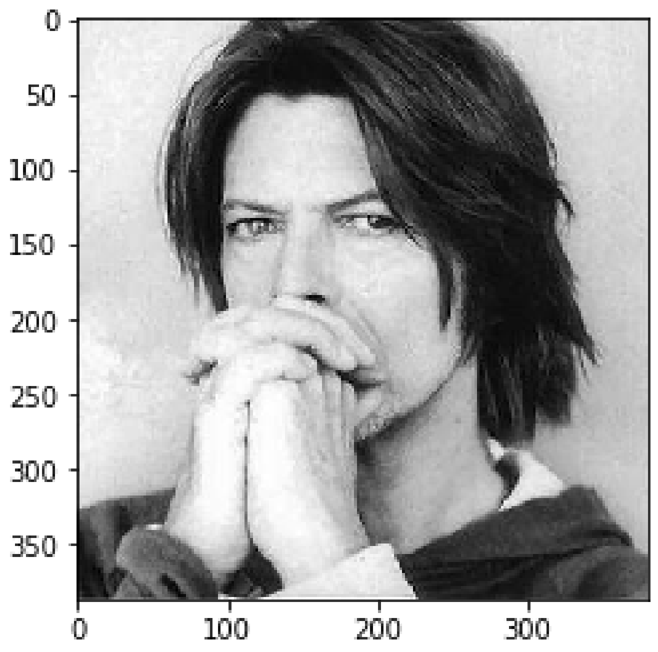


```
Dalmatian



file name: my_human/2015-12-31 14.08.45.jpg
----------------------------------------------------------
no humans or dogs at this time



file name: my_human/2015-11-08 14.31.54.jpg
----------------------------------------------------------
This human is look like:
```

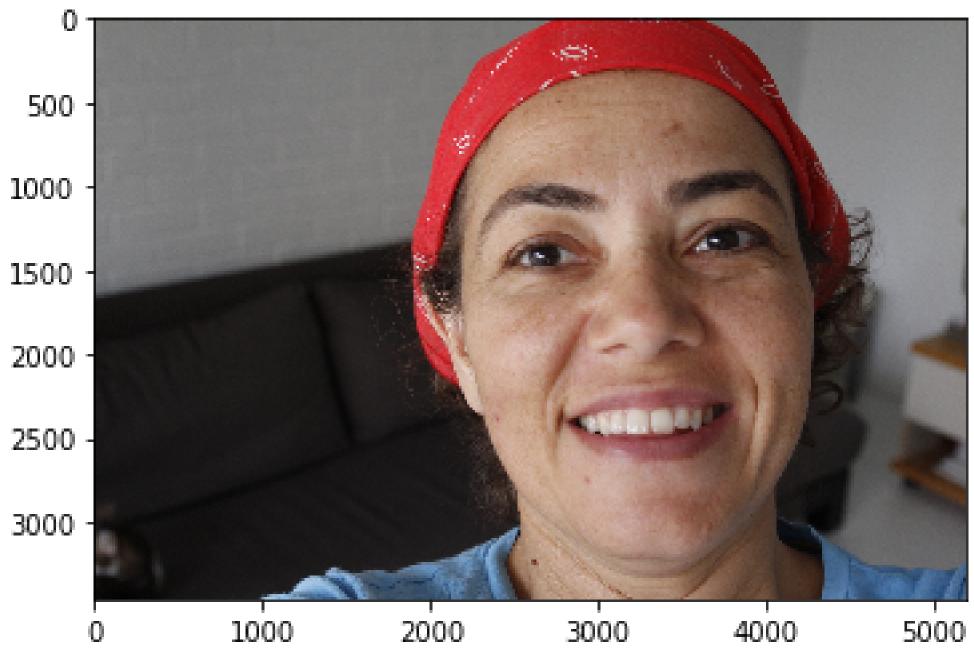


```
Afghan hound
```

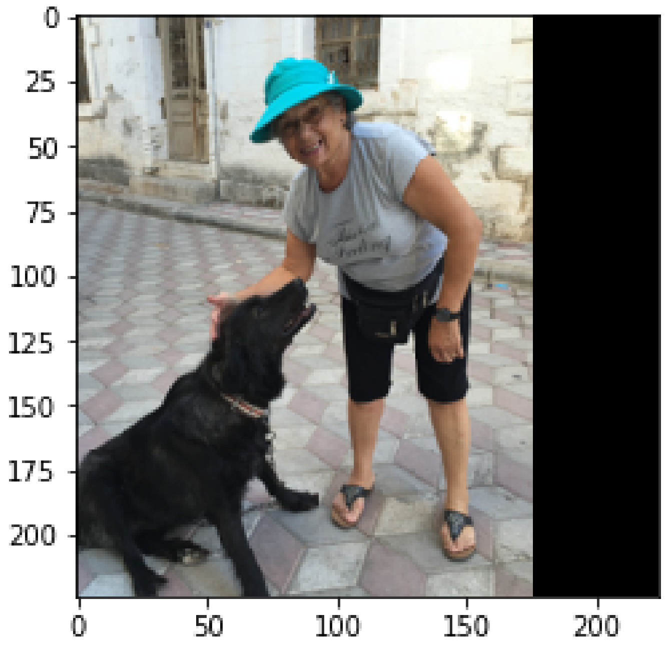
```
file name: my_human/2016-01-12 03.10.32.jpg
-------------------------------------------------------
This human is look like:
```



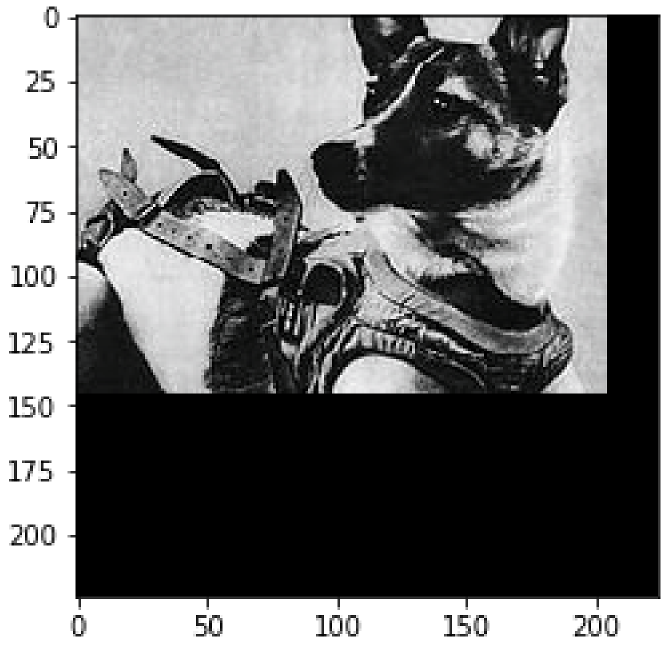
```
Poodle



file name: my_human/_MG_9755.JPG
-------------------------------------------------------
This human is look like:
```



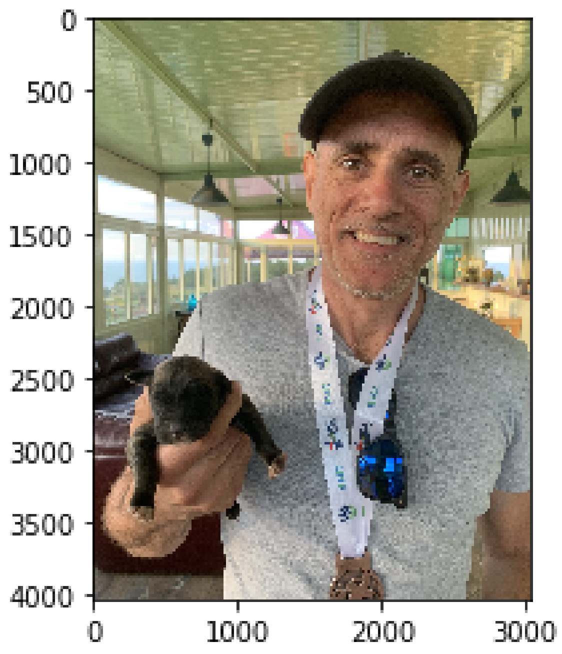
```
Dogue de bordeaux



file name: my_dog/2016-02-02 11.47.53.jpg
---------------------------------------------------------
no humans or dogs at this time



file name: my_dog/2015-09-01 16.58.55.jpg
---------------------------------------------------------
This good dog's breed:
```



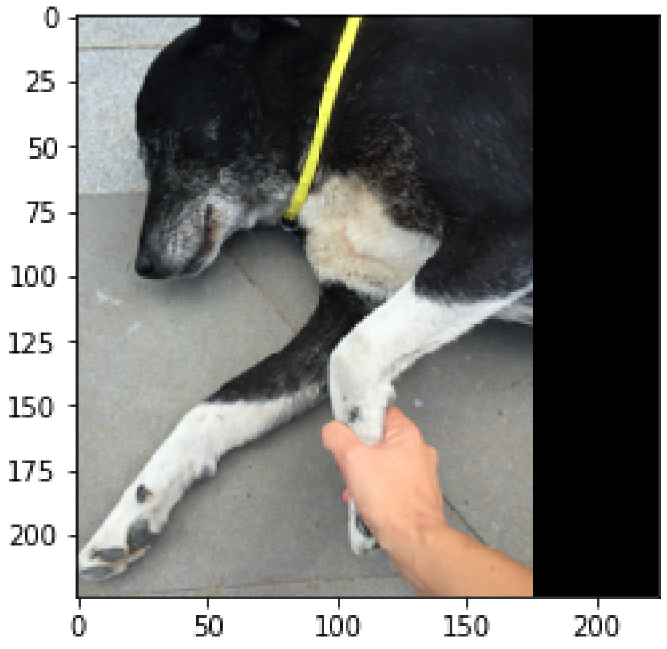
```
Giant schnauzer



file name: my_dog/2015-12-14 12.39.01.jpg
--------------------------------------------------------
This good dog's breed:
```



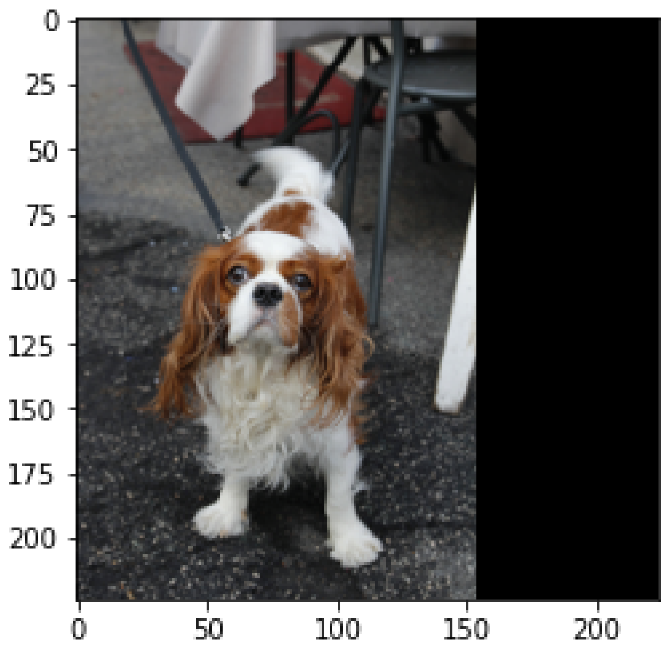
```
Norwegian elkhound



file name: my_dog/2016-02-02 11.50.09.jpg
--------------------------------------------------------
no humans or dogs at this time



file name: my_dog/20190526_163123269_iOS.jpg
--------------------------------------------------------
This human is look like:
```



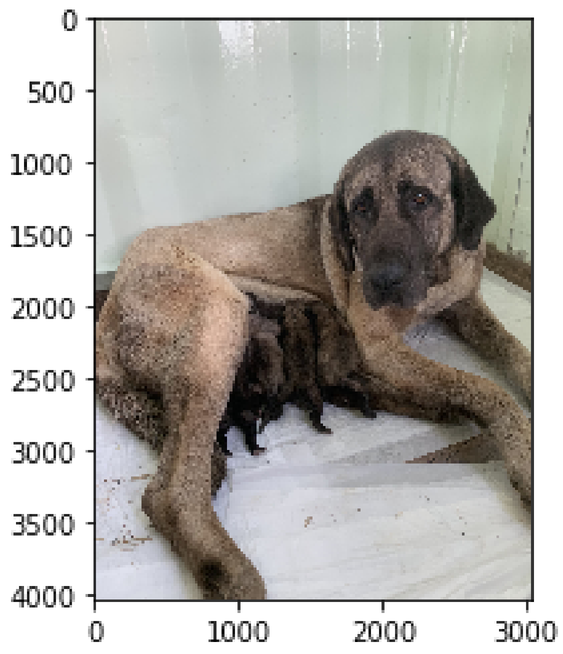
```
Parson russell terrier



file name: my_dog/2015-09-15 15.26.25.jpg
---------------------------------------------------------
This good dog's breed:
```



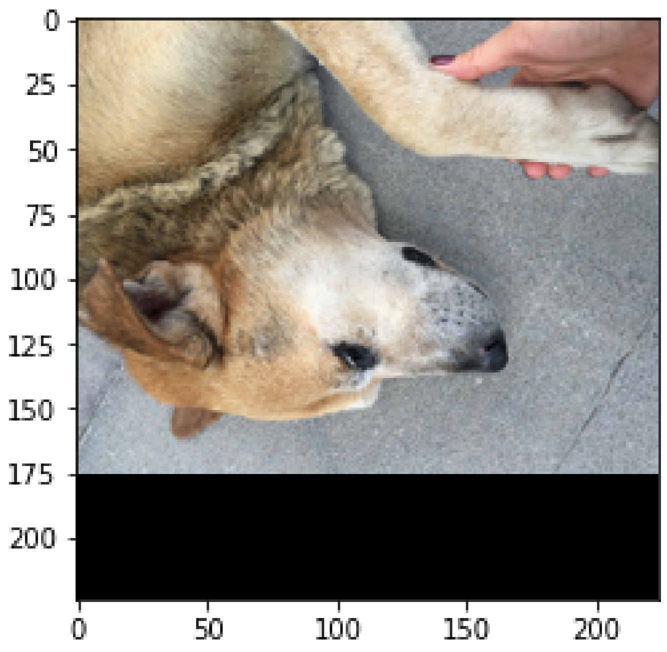
```
Australian cattle dog



file name: my_dog/2016-02-02 11.51.15.jpg
------------------------------------------------------
no humans or dogs at this time



file name: my_dog/2016-02-01 10.45.02.jpg
------------------------------------------------------
This good dog's breed:
```



```
Cavalier king charles spaniel



file name: my_dog/20190526_163620251_iOS.jpg
--------------------------------------------------------
This human is look like:
```



```
Belgian malinois



file name: my_dog/2015-09-15 15.26.00.jpg
-------------------------------------------------------
This good dog's breed:
```



```
Border terrier



file name: my_dog/20190508_164639998_iOS.jpg
----------------------------------------------------------
no humans or dogs at this time
There are 6 total human images.
There are 11 total dog images.
```

```
In [ ]:
```