Convolutional Neural Networks

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.

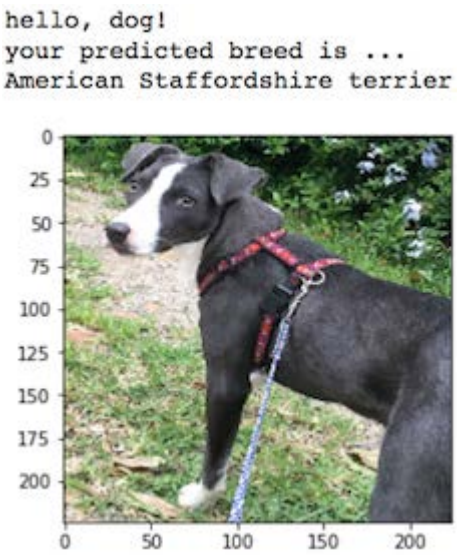
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes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

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).



In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

Step 0: Import Datasets

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

- Download the dog dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip). Unzip the folder and place it in this project's home directory, at the location `/dogImages`.
- Download the human dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip). 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 (http://www.7-zip.org/) 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 [2]:

```
# !wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
# !unzip dogImages.zip
```

In [3]:

```
# !wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
# !unzip lfw.zip
```

In [4]:

```
import numpy as np
from glob import glob

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

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

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

Step 1: Detect Humans

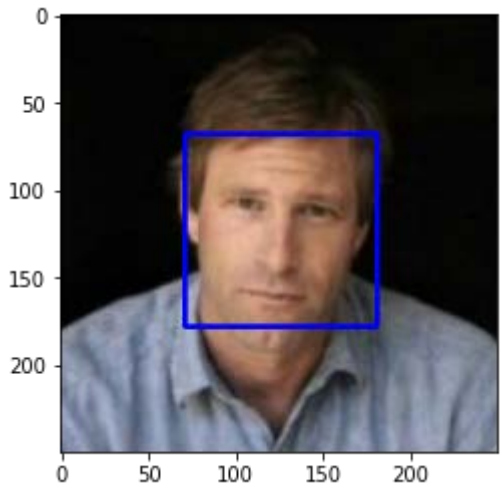
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). 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 [5]:

```
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.

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 [6]:

```
# 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
```

(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 [7]:

```
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.

positive_in_human_files_short = np.sum([face_detector(f) for f in human_files_short]
print(f"Human faces are detected in {positive_in_human_files_short}% of human_files_

positive_in_dog_files_short = np.sum([face_detector(f) for f in dog_files_short])
print(f"Human faces are detected in {positive_in_dog_files_short}% of dog_files_shor
```

```
Human faces are detected in 96% of human_files_short.
Human faces are detected in 18% of 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 [8]:

```
### (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 (http://pytorch.org/docs/master/torchvision/models.html) to detect dogs in images.

Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet (http://www.image-net.org/), 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 (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [9]:

```
import torch
from torchvision import 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()
```

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.

(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 (http://pytorch.org/docs/stable/torchvision/models.html).

In [10]:

```
from PIL import Image
from torchvision import transforms as T

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

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
    pil_img = Image.open(img_path)
    transform = T.Compose([T.Resize(256), T.CenterCrop(224), T.ToTensor()])
    tensor_img = transform(pil_img).unsqueeze_(0)
    if use_cuda:
        tensor_img = tensor_img.cuda()
    output = VGG16(tensor_img)
    _, pred = torch.max(output, 1)
    return pred.item()
```

(IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), 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 [11]:

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    class_idx = VGG16_predict(img_path)
    return 151 <= class_idx <= 268
```

(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 [12]:

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

positive_in_human_files_short = np.sum([dog_detector(f) for f in human_files_short])
print(f"Dogs are detected in {positive_in_human_files_short}% of human_files_short."

positive_in_dog_files_short = np.sum([dog_detector(f) for f in dog_files_short])
print(f"Dogs are detected in {positive_in_dog_files_short}% of dog_files_short.")
```

```
Dogs are detected in 0% of human_files_short.
Dogs are detected in 92% of 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 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3), ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3), 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`.

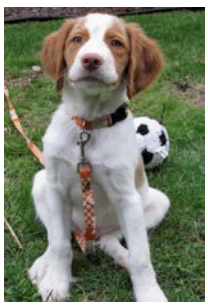
In [13]:

```
### (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 | Black 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!

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader)](http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) 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 (http://pytorch.org/docs/stable/torchvision/datasets.html)](http://pytorch.org/docs/stable/torchvision/datasets.html) to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of [transforms (http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)](http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)!

In [156]:

```
import os
from torchvision import datasets

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

img_size = 224
num_workers = 0
batch_size = 20

# define training and test data directories
data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

# VGG-16 Takes 224x224 images as input, so we resize all of them
aug_transform = T.Compose([T.RandomHorizontalFlip(),
                           T.RandomRotation(15),
                          #T.RandomPerspective(),
                          #T.Resize((224, 224)),
                           T.RandomResizedCrop(img_size, scale=(0.8, 1.2), ratio=(0
                           T.ColorJitter(brightness=0.1, contrast=0.1, saturation=0
                           T.ToTensor()])

basic_transform = T.Compose([T.Resize(img_size), T.CenterCrop(img_size), T.ToTensor(

train_data = datasets.ImageFolder(train_dir, transform=aug_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=basic_transform)
test_data = datasets.ImageFolder(test_dir, transform=basic_transform)

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

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffl
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffl
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=

loaders_scratch = {'train': train_loader, 'valid': valid_loader, 'test': test_loader
```

```
Num training images: 6680
Num validation images: 835
Num test images: 836
```

In [157]:

```
# obtain one batch of training images
dataiter = iter(test_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
# display 20 images
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.transpose(images[idx], (1,2,0)))
    ax.set_title(labels[idx].item())
```

In [16]:

```
# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
# display 20 images
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.transpose(images[idx], (1,2,0)))
    ax.set_title(labels[idx].item())
```

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 the image to 224x224, which is the same size as the VGG input. This will help to compare the result between the two models easily. Also, the dataset size of a breed is about only 60, and this amount of image is too small to train the model distinguishing a dog among 118 breeds. To increase the size of datasets, I used the transforms which torchvision provides; RandomHorizontalFlip, RandomRotation. I avoided using cropping or translations because, in some cases, they generated an image with just a part of the dog. Later, I added standard normalization according to the requirements of input of the VGG network.

(IMPLEMENTATION) Model Architecture

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

In [96]:

```
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__()
        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)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.bn2 = nn.BatchNorm2d(32)
        self.bn3 = nn.BatchNorm2d(64)
        self.bn4 = nn.BatchNorm2d(128)
        self.maxpool = nn.MaxPool2d(2)
        self.dropout = nn.Dropout(0.1)
        self.dropout2d = nn.Dropout2d(0.1)
        self.fc1 = nn.Linear(14*14*128, 8192)
        self.fc2 = nn.Linear(8192, 512)
        self.fc3 = nn.Linear(512, 133)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)
        x = self.dropout2d(x)
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.maxpool(x)
        x = self.dropout2d(x)
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.maxpool(x)
        x = self.dropout2d(x)
        x = F.relu(self.bn4(self.conv4(x)))
        x = self.maxpool(x)
        x = self.dropout2d(x)
        x = x.view(-1, 14*14*128)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(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()
```

In [97]:

```
model_scratch
```

Out[97]:

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
  (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
  (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
  (bn4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (dropout): Dropout(p=0.1, inplace=False)
  (dropout2d): Dropout2d(p=0.1, inplace=False)
  (fc1): Linear(in_features=25088, out_features=8192, bias=True)
  (fc2): Linear(in_features=8192, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=133, bias=True)
)
```

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

Answer: I had failed to reach the accuracy of 10% with a small network consists of a couple of convolutional layers and a couple of dense layers. I tried to add more layers and more nodes on each layer. Finally, I can get better than 10% accuracy on training, but the network is undertaken overfitting.

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
 1))
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (fc1): Linear(in_features=25088, out_features=8192, bias=True)
  (fc2): Linear(in_features=8192, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=133, bias=True)
)
```

I added dropout layers after each convolutional and dense layers. It seems to work, but the learning speed was too small to reach 10% of accuracy. The bigger learning rate of more than 1e-2 led to overshoot and made it worse. According to https://arxiv.org/abs/1502.03167 (https://arxiv.org/abs/1502.03167), I added the BatchNorm2d, and I could reach 11% of accuracy with the faster learning speed without increasing the learning rate. Finally, even a smaller learning rate, 1e-3 with 100 epochs, results in 26% accuracy on the test dataset.

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a [loss function (http://pytorch.org/docs/stable/nn.html#loss-functions)](http://pytorch.org/docs/stable/nn.html#loss-functions) and [optimizer (http://pytorch.org/docs/stable/optim.html)](http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [98]:

```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=1e-3)
```

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. [Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html)](http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_scratch.pt'`.

In [99]:

```
# 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
        valid_loss = 0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - trair
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            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
            output = model(data)
            loss = criterion(output, target)
            valid_loss += 1 / (batch_idx + 1) * (loss.data - valid_loss)

        # print training/validation statistics
        print(f"Epoch: {epoch} \tTraining Loss: {train_loss:.6f} \tValidation Loss:

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            print(f"Validation loss decreased",
                  f"{valid_loss_min:.6f} --> {valid_loss:.6f}).",
                  f"Saving model ...")
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model
```

```
# train the model
model_scratch = train(100, 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: 5.385852         Validation Loss: 4.66
7729
Validation loss decreased inf --> 4.667729). Saving model ...
Epoch: 2        Training Loss: 4.603566         Validation Loss: 4.46
4767
Validation loss decreased 4.667729 --> 4.464767). Saving model ...
Epoch: 3        Training Loss: 4.491615         Validation Loss: 4.48
0530
Epoch: 4        Training Loss: 4.425432         Validation Loss: 4.36
5483
Validation loss decreased 4.464767 --> 4.365483). Saving model ...
Epoch: 5        Training Loss: 4.351926         Validation Loss: 4.37
3513
Epoch: 6        Training Loss: 4.328231         Validation Loss: 4.30
6231
Validation loss decreased 4.365483 --> 4.306231). Saving model ...
Epoch: 7        Training Loss: 4.292843         Validation Loss: 4.23
4534
Validation loss decreased 4.306231 --> 4.234534). Saving model ...
```

(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 [153]:

```
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 += 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"Test Loss: {test_loss:.6f}\n")

    print(f"\nTest Accuracy: {int(100 * correct / total):2d}% ({correct:2d}/{total:2

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

```
Test Loss: 3.117711


Test Accuracy: 24% (207/836)
```

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.

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader)](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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 [158]:

```
## TODO: Specify data loaders

# VGG-16 Takes 224x224 images as input, so we resize all of them
aug_transform = T.Compose([T.RandomHorizontalFlip(),
                           T.RandomRotation(15),
                          #T.RandomPerspective(),
                          #T.Resize((224, 224)),
                           T.RandomResizedCrop(img_size, scale=(0.8, 1.2), ratio=(0
                           T.ColorJitter(brightness=0.1, contrast=0.1, saturation=0
                           T.ToTensor(),
                           T.Normalize(mean = [0.485, 0.456, 0.406],
                                       std = [0.229, 0.224, 0.225])])

basic_transform = T.Compose([T.Resize(img_size), T.CenterCrop(img_size), T.ToTensor(
                            T.Normalize(mean = [0.485, 0.456, 0.406],
                                       std = [0.229, 0.224, 0.225])])

train_data = datasets.ImageFolder(train_dir, transform=aug_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=basic_transform)
test_data = datasets.ImageFolder(test_dir, transform=basic_transform)

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

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffl
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffl
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=

loaders_transfer = {'train': train_loader, 'valid': valid_loader, 'test': test_loade
```

```
Num training images: 6680
Num validation images: 835
Num test images: 836
```

(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 [92]:

```
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.features.parameters():
    param.requires_grad = False
model_transfer.classifier[6] = nn.Linear(model_transfer.classifier[6].in_features, 1

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: I chose the VGG model for the pre-trained feature extraction model because VGG is trained to classify 1000 classes. Also, the minimum size of the input image is 224 is somewhat similar to the size of the given images.

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/master/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

In [93]:

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.Adam(model_transfer.parameters(), lr=1e-4)
```

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'`.

In [94]:

```
# train the model
model_transfer = train(100, loaders_transfer, model_transfer, optimizer_transfer, cr

# 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: 2.548605         Validation Loss: 1.23
9957
Validation loss decreased inf --> 1.239957). Saving model ...
Epoch: 2        Training Loss: 1.344366         Validation Loss: 1.08
2291
Validation loss decreased 1.239957 --> 1.082291). Saving model ...
Epoch: 3        Training Loss: 1.006615         Validation Loss: 0.99
2048
Validation loss decreased 1.082291 --> 0.992048). Saving model ...
Epoch: 4        Training Loss: 0.863290         Validation Loss: 0.91
4890
Validation loss decreased 0.992048 --> 0.914890). Saving model ...
Epoch: 5        Training Loss: 0.678144         Validation Loss: 0.95
6535
Epoch: 6        Training Loss: 0.587663         Validation Loss: 0.92
4383
Epoch: 7        Training Loss: 0.502243         Validation Loss: 1.00
8521
Epoch: 8        Training Loss: 0.461544         Validation Loss: 1.00
```

(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 [159]:

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.232821


Test Accuracy: 70% (587/836)
```

(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 [163]:

```
### 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 loaders_transfer['train'].data

img_size = 224
basic_transform = T.Compose([T.Resize(img_size), T.CenterCrop(img_size), T.ToTensor(
                             T.Normalize(mean = [0.485, 0.456, 0.406],
                                         std = [0.229, 0.224, 0.225])])

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    pil_img = Image.open(img_path)
    tensor_img = basic_transform(pil_img).unsqueeze_(0)
    if use_cuda:
        tensor_img = tensor_img.cuda()
    output = model_transfer(tensor_img)
    _, pred = torch.max(output, 1)
    return class_names[pred.item()]
```

Step 5: Write your Algorithm

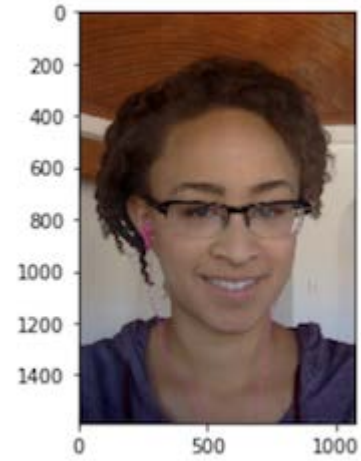
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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!

```
hello, human!
```




```
You look like a ...
Chinese_shar-pei
```

(IMPLEMENTATION) Write your Algorithm

In [184]:

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

def plot_img(img_path):
    img = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    human_positive = face_detector(img_path)
    dog_positive = dog_detector(img_path)

    if dog_positive:
        print("Hello, dog!")
        print("Your predicted breed is ...")
        print(predict_breed_transfer(img_path))
        plot_img(img_path)

    elif human_positive:
        print('Hello, human!')
        print("You look like a ...")
        print(predict_breed_transfer(img_path))
        plot_img(img_path)

    else:
        print("Hello, something!")
        print("You are not recognized ...")
        plot_img(img_path)
```

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?

(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.

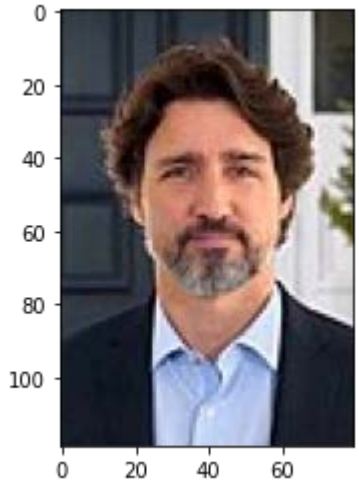
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

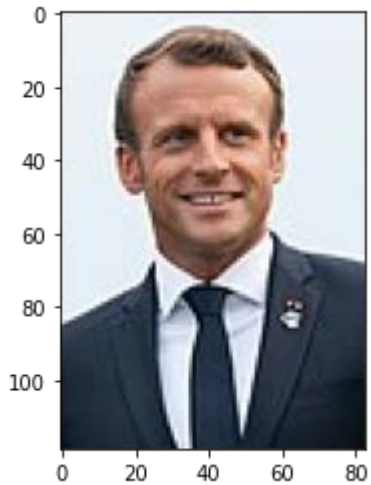
In [185]:

```
## 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.

human_files = np.array(glob("human_files/*"))
dog_files = np.array(glob("dog_files/*"))

## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:4])):
    run_app(file)
    print("")
```

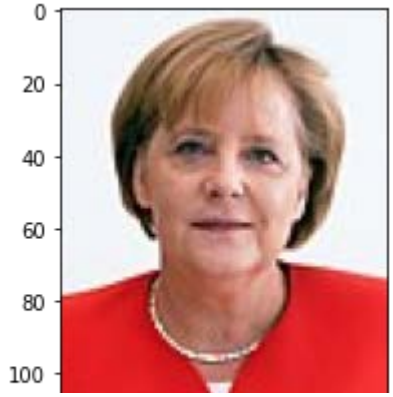
```
Hello, human!
You look like a ...
Nova scotia duck tolling retriever
```



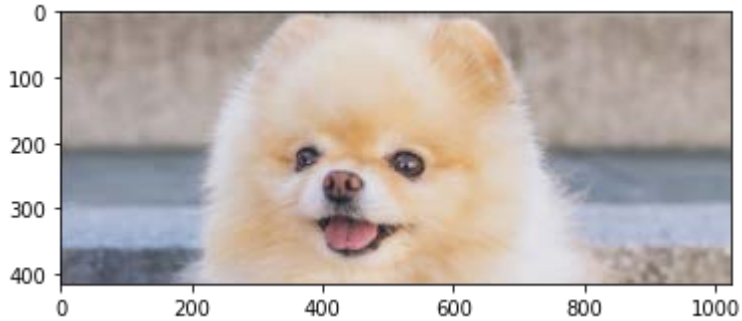

```
Hello, human!
You look like a ...
German pinscher
```



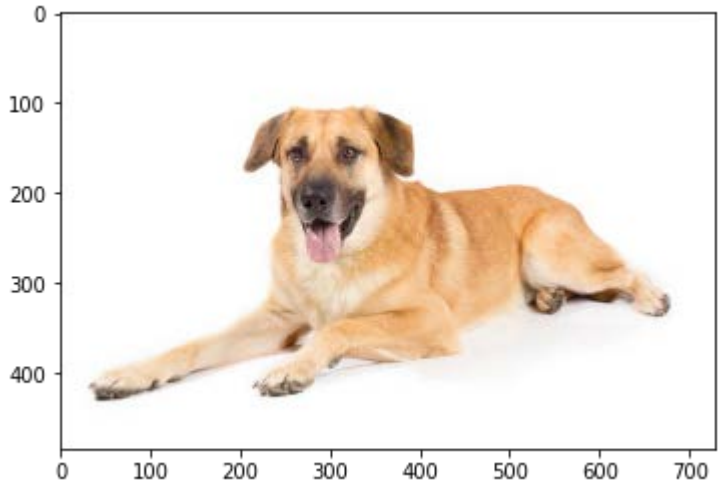

```
Hello, human!
You look like a ...
Dachshund
```




```
Hello, dog!
Your predicted breed is ...
Pomeranian
```

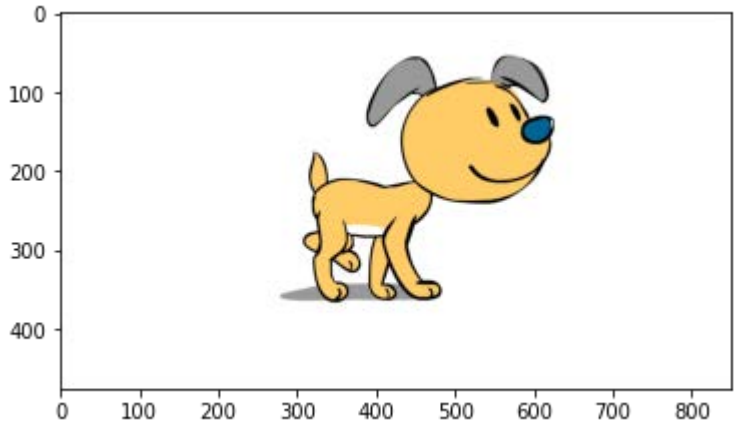



```
Hello, dog!
Your predicted breed is ...
Golden retriever
```




```
Hello, something!
You are not recognized ...
```


```
Hello, something!
You are not recognized ...
```



In [ ]: