dog_app

April 11,2020

1 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 **'(IMPLEMENTA- TION)'** 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 export- ing 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 us- ing the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

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

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

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

Step 0: Import Datasets

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

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

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

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

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

In [2]:
```
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.
```

In [3]:
```
# !rm -r dogImages/
# !rm -r lfw/
```

In [4]:
```
# !rm dogImages.zip
# !rm lfw.zip
```

In [5]:
```
# !wget -c https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
# !wget -c https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
```

In [6]:
```
# !unzip dogImages.zip
```

In [7]:
```
# !unzip lfw.zip
```

Step 1: Detect Humans

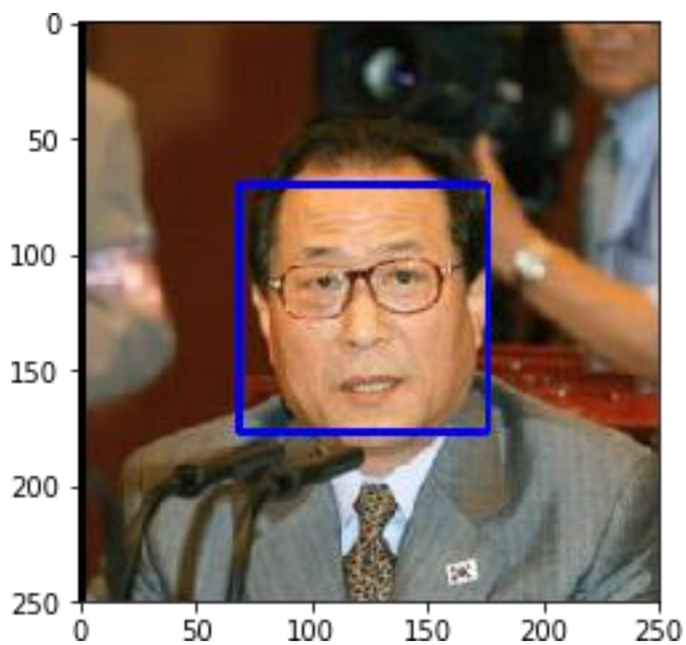
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [8]:
```
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[5])
# 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 [9]:
```
# 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) Detected human face in human images: 98 / 100 Detected dog in dog images: 17 / 100

In [10]:
```
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.
humans_in_human_files = 0
humans_in_dog_files = 0

for i in human_files_short:
    if face_detector(i):
        humans_in_human_files += 1
```

```
for i in dog_files_short:
    if face_detector(i):
        humans_in_dog_files += 1

print("Detected human face in human images: {} / {}".format(humans_in_human_files, 100)
print("Detected dog in dog images: {} / {}".format(humans_in_dog_files, 100))
```

```
Detected human face in human images: 98 / 100
Detected dog in dog images: 17 / 100
```

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 [11]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

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

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 [12]:
```
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()

# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 94733609.17it/s]
```

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

(IMPLEMENTATION) Making Predictions with a Pre-trained Model

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

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

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

def load_image(img_path):
    image = Image.open(img_path).convert('RGB')

    in_transform = transforms.Compose([
        transforms.Resize(size=(244, 244)),
        transforms.ToTensor()
    ])

    image = in_transform(image)[:3, :, :].unsqueeze(0)

    return image
```

In [14]:
```
from PIL import Image
import torchvision.transforms as transforms

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

    img = load_image(img_path)
```

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

        ret = VGG16(img)

        return torch.max(ret, 1)[1].item()
```

(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 [15]:
```
### 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 (prediction>=151 and prediction<=268) # true/false
```

(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:

Detected human face in dog images: 0 / 100 Detected dog in dog images: 99 / 100

In [16]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
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.
dogs_in_human_files =  0
dogs_in_dog_files =  0

for i in human_files_short:
    if dog_detector(i):
        dogs_in_human_files +=  1
```

```
for i in dog_files_short:
    if dog_detector(i):
        dogs_in_dog_files += 1

print("Detected human face in dog images: {} / {}".format(dogs_in_human_files, 100))
print("Detected dog in dog images: {} / {}".format(dogs_in_dog_files, 100))
```

```
Detected human face in dog images: 0 / 100
Detected dog in dog images: 99 / 100
```

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 [17]:
```
### (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 algo- rithm 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!

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

In [18]:
```
import os
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

img_short_side_resize = 256
img_input_size = 224
num_workers = 0
batch_size = 20
valid_size = 0.2

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/')
```

In [ ]:

In [19]:
```
standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                              std=[0.229, 0.224, 0.225])

data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    standard_normalization]),
                   'val': transforms.Compose([transforms.Resize(img_short_side_resize),
```

```
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     standard_normalization]),
                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                     transforms.ToTensor(),
                                     standard_normalization])
                  }
```

In [20]:
```
train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['val'])
test_data = datasets.ImageFolder(test_dir, transform=data_transforms['test'])
```

In [21]:
```
# train_data = datasets.ImageFolder(glob("/data/dog_images/*/*/"), transform=data_trans
# valid_data = datasets.ImageFolder(glob("/data/dog_images/*/*/", transform=data_transf
# test_data = datasets.ImageFolder(glob("/data/dog_images/*/*/", transform=data_transfo
```

In [22]:
```
train_loader = torch.utils.data.DataLoader(train_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=False)
test_loader = torch.utils.data.DataLoader(test_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=False)
loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

In [ ]:

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:

Transforms - - Random crop - Random horizontal flip - Convert to tensor - Normalization Yes, I augmented the dataset by including randomly cropped images and including randomly flipped images. This is necessary for the generalized training of the images.

Insights - Random crop used to include generality among images - Horizontal flips possible as dogs can face either left or right - Normalize the values to maintain consistency among images

(IMPLEMENTATION) Model Architecture

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

```
In [23]: import torch.nn as nn
         import torch.nn.functional as F
         num_classes = 133 # total classes of dog breeds

         # 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, 32, 3, stride=2, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

                 #  fully-connected
                 self.fc1 = nn.Linear(7*7*128, 500)
                 self.fc2 = nn.Linear(500, num_classes)

                 # drop-out
                 self.dropout = nn.Dropout(0.3)


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

                 # flatten
                 x = x.view(-1, 7*7*128)

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

                 x = self.dropout(x)
                 x = self.fc2(x)

                 return x

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

         # instantiate the CNN
```

```
model_scratch = Net()
print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
)
```

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

Answer:

Start with some convolutional layers and then add some pooling layers. We did this for three times. Then we flattened the image tensor to a column vector. Then we added 2 fully connected layers to train the network on the features as extracted from the images.

Conv 1 224*224*3 => 112*112*32
Pool 112*112*32 => 56*56*32
Conv 2 56*56*32 => 28*28*64
Pool 28*28*64 => 14*14*64
Conv 3 14*14*64 => 14*14*128
Pool 14*14*128 => 7*7*128
Flatten x to a column vector x = [................] dimension = 1*6272 (as 7*7*128 = 6272)
Fully connected layer 1 6272 => 500
Fully connected layer 2 500 => 133

(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 [24]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.1)
```

(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 [25]:
```
import time

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
    time_start = time.time()

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

        ###################
        # 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 - train_lo

            # initialize weights to zero
            optimizer.zero_grad()

            output = model(data)

            # calculate  loss
            loss = criterion(output, target)

            # back prop
            loss.backward()

            # grad
            optimizer.step()

            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
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


        # Print epoch statistics
        print('Epoch {} done in {:.2f} seconds. \tTraining Loss: {:.3f} \tValidation Lo
            epoch,
            time.time() - time_start_epoch,
            train_loss,
            valid_loss
            ))



        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # Show final statistics
    print(f"{n_epochs} epochs ready in {(time.time() - time_start):.3f} seconds. Minimu

    # return trained model
    return model
```

In [26]:
```
# train the model

n_epochs = 20
model_scratch = train(n_epochs, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch 1 done in 84.75 seconds.        Training Loss: 4.838        Validation Loss: 4.675
Epoch 2 done in 84.17 seconds.        Training Loss: 4.688        Validation Loss: 4.472
Epoch 3 done in 83.83 seconds.        Training Loss: 4.602        Validation Loss: 4.415
Epoch 4 done in 83.65 seconds.        Training Loss: 4.505        Validation Loss: 4.289
Epoch 5 done in 83.79 seconds.        Training Loss: 4.459        Validation Loss: 4.257
Epoch 6 done in 84.15 seconds.        Training Loss: 4.355        Validation Loss: 4.128
Epoch 7 done in 83.56 seconds.        Training Loss: 4.302        Validation Loss: 4.073
Epoch 8 done in 84.35 seconds.        Training Loss: 4.244        Validation Loss: 4.023
Epoch 9 done in 84.05 seconds.        Training Loss: 4.168        Validation Loss: 4.022
```

```
Epoch 10 done in 84.05 seconds.     Training Loss: 4.103     Validation Loss: 3.837
Epoch 11 done in 84.00 seconds.     Training Loss: 4.080     Validation Loss: 3.892
Epoch 12 done in 84.23 seconds.     Training Loss: 4.045     Validation Loss: 3.840
Epoch 13 done in 83.80 seconds.     Training Loss: 3.992     Validation Loss: 3.797
Epoch 14 done in 84.18 seconds.     Training Loss: 3.908     Validation Loss: 3.691
Epoch 15 done in 83.86 seconds.     Training Loss: 3.895     Validation Loss: 3.692
Epoch 16 done in 83.97 seconds.     Training Loss: 3.875     Validation Loss: 3.679
Epoch 17 done in 83.98 seconds.     Training Loss: 3.820     Validation Loss: 3.698
Epoch 18 done in 83.63 seconds.     Training Loss: 3.794     Validation Loss: 3.698
Epoch 19 done in 83.91 seconds.     Training Loss: 3.756     Validation Loss: 3.647
Epoch 20 done in 83.76 seconds.     Training Loss: 3.729     Validation Loss: 3.633
20 epochs ready in 1680.245 seconds. Minimum validation loss: 3.633
```

In [27]:
```
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

(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 [28]:
```
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: 3.730881


Test Accuracy: 15% (131/836)
```

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 for the training, validation, and test datasets of dog images (located at dogImages/train, dogImages/valid, and dogImages/test, re- spectively).

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

In [29]:
```
## TODO: Specify data  loaders
loaders_transfer = loaders_scratch.copy()
```

(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 [30]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture model_transfer

= models.resnet50(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 94991041.99it/s]
```

In [31]:
```
for param in model_transfer.parameters():
    param.requires_grad = False
```

In [32]:
```
model_transfer.fc = nn.Linear(2048, 133, bias=True)
```

```
In [33]: fc_parameters = model_transfer.fc.parameters()

In [34]: for param in fc_parameters:
             param.requires_grad = True

In [35]: model_transfer

Out[35]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
           (layer1): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
               (downsample): Sequential(
                 (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
                 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               )
             )
             (1): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
             )
             (2): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
             )
           )
           (layer2): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

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

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

```
        (relu): ReLU(inplace)
      )
    )
    (layer4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
        (downsample): Sequential(
          (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
    )
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
    (fc): Linear(in_features=2048, out_features=133, bias=True)
  )
```

In [36]:
```
if use_cuda:
    model_transfer = model_transfer.cuda()
```

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

Answer:

Resnet is containing 1000s of images and they are trained on millions of images to classify images with a high accuracy. It is useful for classifying the dog images and here we have 133 different categories of dogs. We can use the trained model and the parameters to classify the dog images based on our needs.

(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 [40]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.01)
```

(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 [41]:
```
import time

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
    time_start = time.time()

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

        ###################
        # 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 - train_lo

            # initialize weights to zero
            optimizer.zero_grad()

            output = model(data)

            # calculate  loss
            loss = criterion(output, target)
```

```
            # back prop
            loss.backward()

            # grad
            optimizer.step()

            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
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


        # Print epoch statistics
        print('Epoch {} done in {:.2f} seconds. \tTraining Loss: {:.3f} \tValidation Lo
            epoch,
            time.time() - time_start_epoch,
            train_loss,
            valid_loss
            ))



        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # Show final statistics
    print(f"{n_epochs} epochs ready in {(time.time() - time_start):.3f} seconds. Minimu

    # return trained model
    return model
```

In [42]:
```
# train the model

n_epochs = 20
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
```

```
Epoch 1 done in 89.41 seconds.     Training Loss: 1.277     Validation Loss: 0.686
Epoch 2 done in 89.39 seconds.     Training Loss: 1.174     Validation Loss: 0.638
Epoch 3 done in 89.19 seconds.     Training Loss: 1.117     Validation Loss: 0.592
Epoch 4 done in 88.78 seconds.     Training Loss: 1.091     Validation Loss: 0.568
Epoch 5 done in 89.12 seconds.     Training Loss: 1.041     Validation Loss: 0.554
Epoch 6 done in 89.12 seconds.     Training Loss: 0.993     Validation Loss: 0.528
Epoch 7 done in 90.14 seconds.     Training Loss: 0.982     Validation Loss: 0.516
Epoch 8 done in 89.52 seconds.     Training Loss: 0.970     Validation Loss: 0.495
Epoch 9 done in 89.17 seconds.     Training Loss: 0.928     Validation Loss: 0.478
Epoch 10 done in 88.99 seconds.     Training Loss: 0.915     Validation Loss: 0.472
Epoch 11 done in 88.81 seconds.     Training Loss: 0.903     Validation Loss: 0.467
Epoch 12 done in 89.30 seconds.     Training Loss: 0.902     Validation Loss: 0.447
Epoch 13 done in 88.96 seconds.     Training Loss: 0.875     Validation Loss: 0.441
Epoch 14 done in 89.29 seconds.     Training Loss: 0.853     Validation Loss: 0.442
Epoch 15 done in 88.90 seconds.     Training Loss: 0.826     Validation Loss: 0.441
Epoch 16 done in 88.90 seconds.     Training Loss: 0.834     Validation Loss: 0.430
Epoch 17 done in 89.88 seconds.     Training Loss: 0.836     Validation Loss: 0.438
Epoch 18 done in 90.13 seconds.     Training Loss: 0.838     Validation Loss: 0.422
Epoch 19 done in 89.87 seconds.     Training Loss: 0.791     Validation Loss: 0.418
Epoch 20 done in 89.80 seconds.     Training Loss: 0.805     Validation Loss: 0.415
20 epochs ready in 1791.424 seconds. Minimum validation loss: 0.415
```

In [43]:
```
# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

(IMPLEMENTATION) Test the Model

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

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

```
Test Loss: 0.472685


Test Accuracy: 86% (724/836)
```

(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 [45]:
```
### 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'].dataset
```

```
In [46]: from PIL import Image
         import torchvision.transforms as transforms

         def load_input_image(img_path):
             image = Image.open(img_path).convert('RGB')
             prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                              transforms.ToTensor(),
                                              standard_normalization])

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

In [47]: def predict_breed_transfer(model, class_names, img_path):
             # load the image and return the predicted breed
             img = load_input_image(img_path)
             model = model.cpu()
             model.eval()
             idx = torch.argmax(model(img))
             return class_names[idx]

In [48]: for img_file in os.listdir('./images'):
             img_path = os.path.join('./images', img_file)
             predition = predict_breed_transfer(model_transfer, class_names, img_path)
             print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))
```

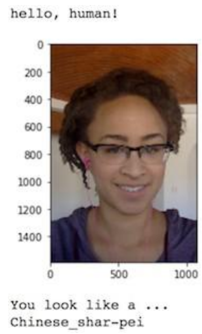
```
image_file_name: ./images/Labrador_retriever_06449.jpg,      predition breed: Labrador retri
image_file_name: ./images/sample_dog_output.png,     predition breed: Great dane
image_file_name: ./images/Brittany_02625.jpg,    predition breed: Brittany
image_file_name: ./images/sample_human_output.png,       predition breed: Brussels griffon
image_file_name: ./images/American_water_spaniel_00648.jpg,      predition breed: Curly-coat
image_file_name: ./images/Curly-coated_retriever_03896.jpg,      predition breed: Curly-coat
image_file_name: ./images/Labrador_retriever_06457.jpg,      predition breed: Labrador retri
image_file_name: ./images/Labrador_retriever_06455.jpg,      predition breed: Labrador retri
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,      predition breed: Welsh spri
image_file_name: ./images/sample_cnn.png,    predition breed: American eskimo dog
```

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the face_detector and human_detector functions developed above. You are **re- quired** 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!



Sample Human Output

(IMPLEMENTATION) Write your Algorithm

```
In [49]: ### TODO:  Write your algorithm.
         ### Feel free to use as many code cells as needed.


         def run_app(img_path):
             ## handle cases for a human face,  dog,  and  neither
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
             if dog_detector(img_path) is True:
                 prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                 print("Dogs Detected!\nIt looks like a {0}".format(prediction))
             elif face_detector(img_path) > 0:
                 prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                 print("Hello, human!\nIf you were a dog, you may look like a {0}".format(predic
             else:
                 print("Error! Can't detect anything..")
```

Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algo- rithm 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.

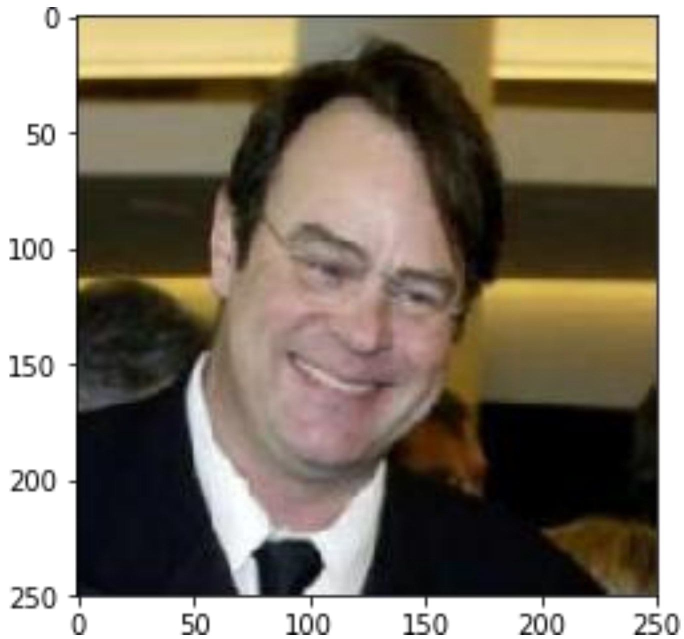
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

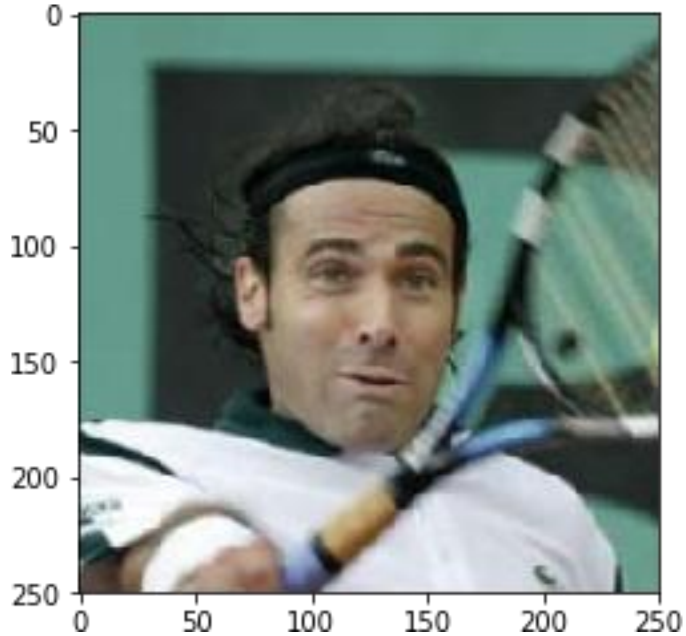
Improvement scope - - Model not able to detect humans and dogs in their respective images - Images with different imputs can be given to the model and the prediction can be done - Learning

rates can be adjusted for the optimum and the no of epochs can be increased - Ensemble learning will give better results and performance will be improved - Add more data to improve accuracy
- Accuracy can be increased by increasing the depth of the neural network - Detect and predict multiple dogs and humans in the same image - Hyper parameter tuning can improve the perfor- mance

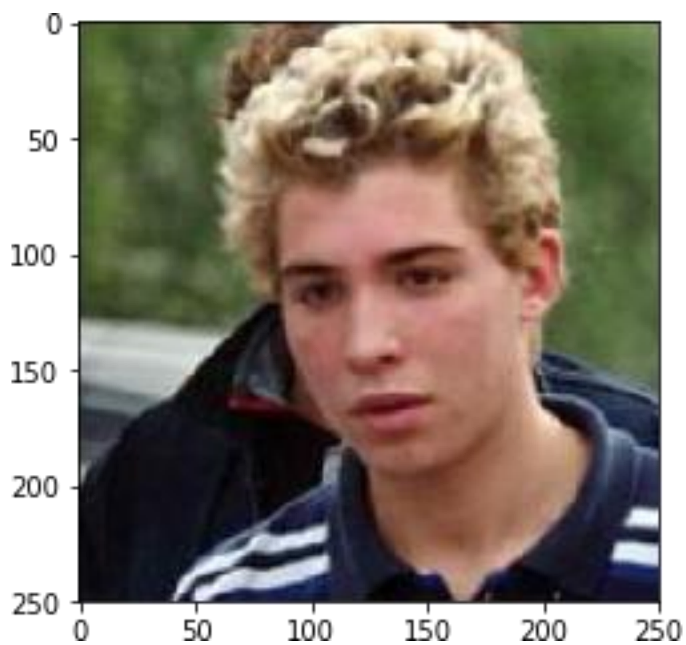
In [50]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[:5], dog_files[:10])):
    run_app(file)
```



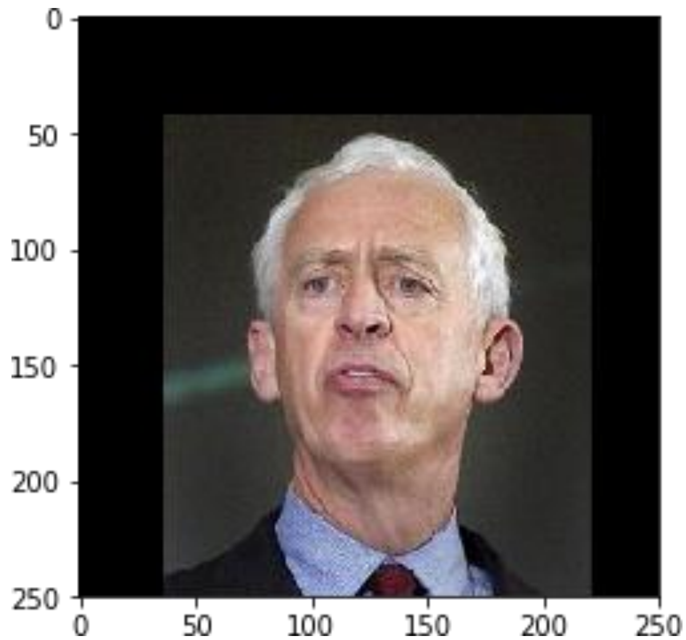
Hello, human!
If you were a dog, you may look like a Chihuahua



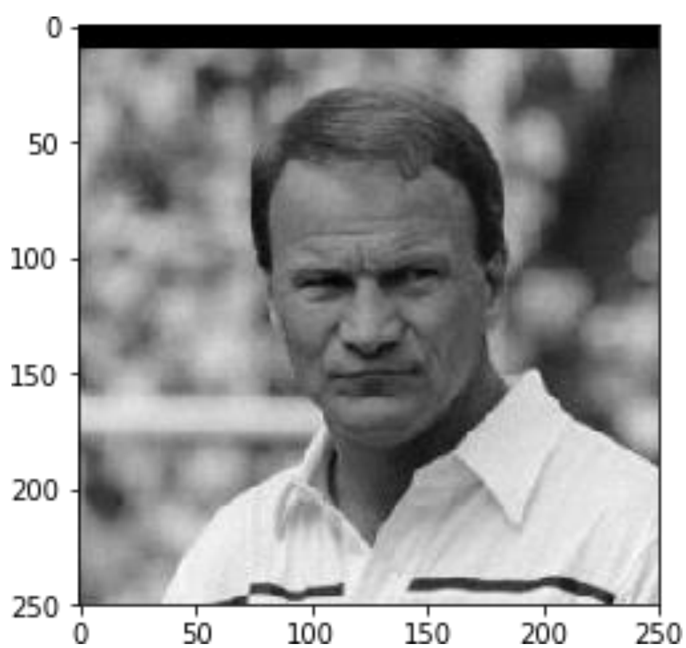
Hello, human!
If you were a dog, you may look like a Ibizan hound



Hello, human!
If you were a dog, you may look like a American water spaniel

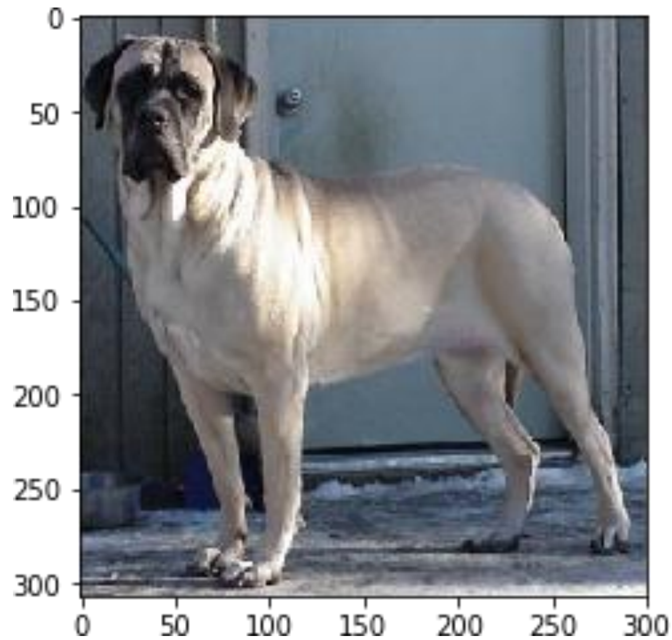


Hello, human!
If you were a dog, you may look like a Ibizan hound

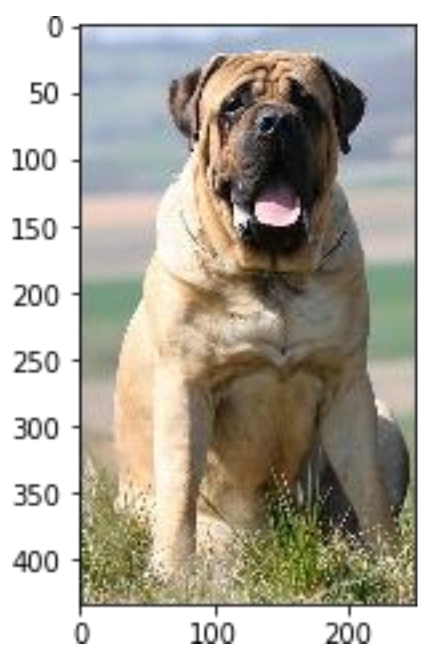


Hello, human!
If you were a dog, you may look like a German pinscher

Dogs Detected!
It looks like a Bullmastiff

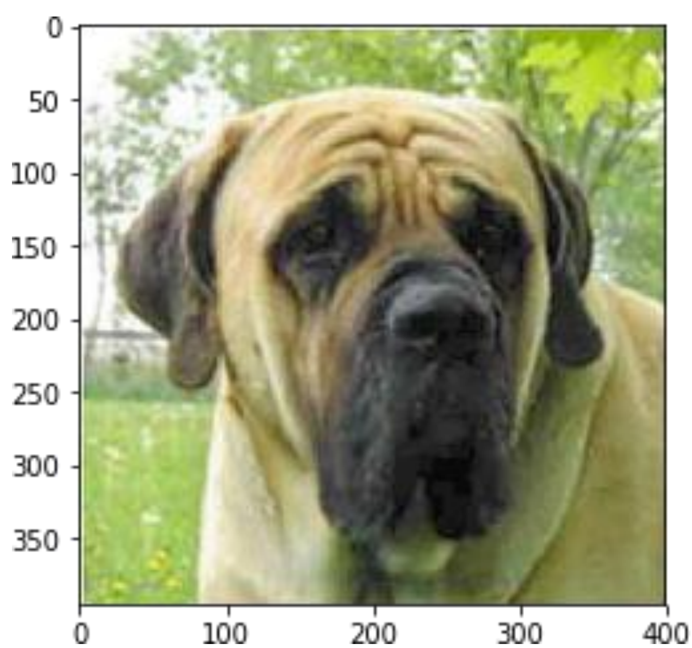


Dogs Detected!
It looks like a Bullmastiff



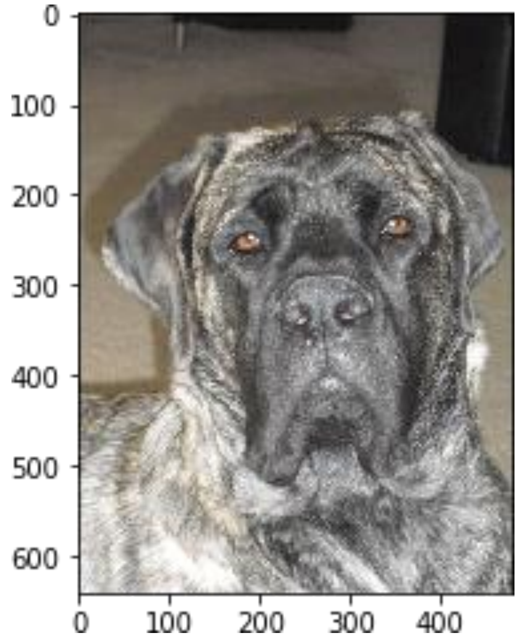
Dogs Detected!
It looks like a Bullmastiff

Dogs Detected!
It looks like a Mastiff



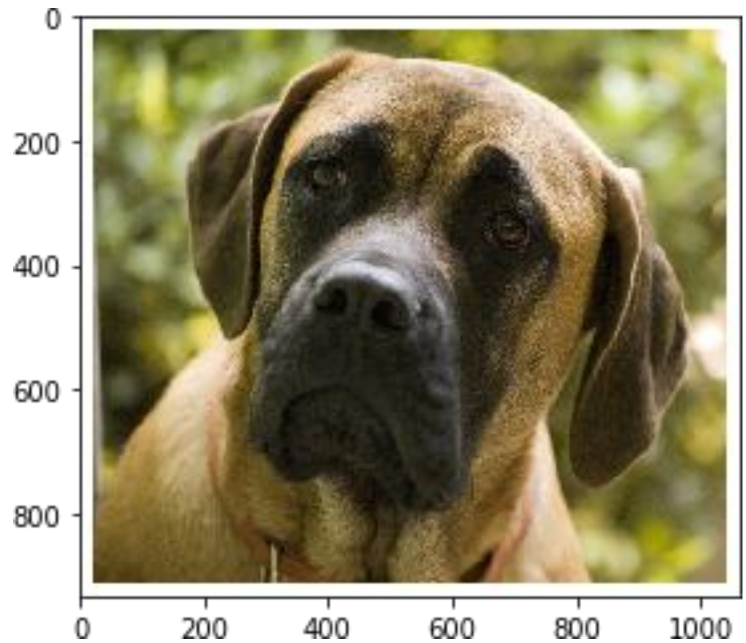
Dogs Detected!
It looks like a Mastiff

Dogs Detected!
It looks like a Mastiff

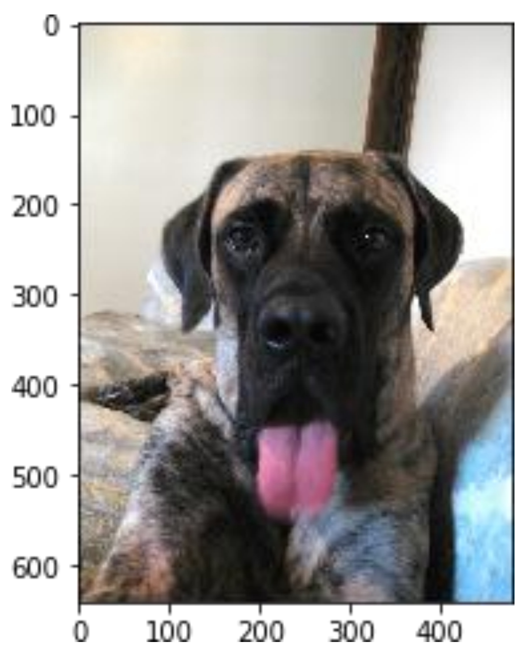


Dogs Detected!
It looks like a Mastiff

Dogs Detected!
It looks like a Mastiff



Dogs Detected!
It looks like a Mastiff



```
Dogs  Detected!
It looks like a Mastiff
```

In [ ]:

In [ ]:

In [ ]: