dog_app

April 20, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

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

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

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

Step 0: Import Datasets

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

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

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

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

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

In [25]:
```
import numpy as np
from glob import glob

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

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

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

Step 1: Detect Humans

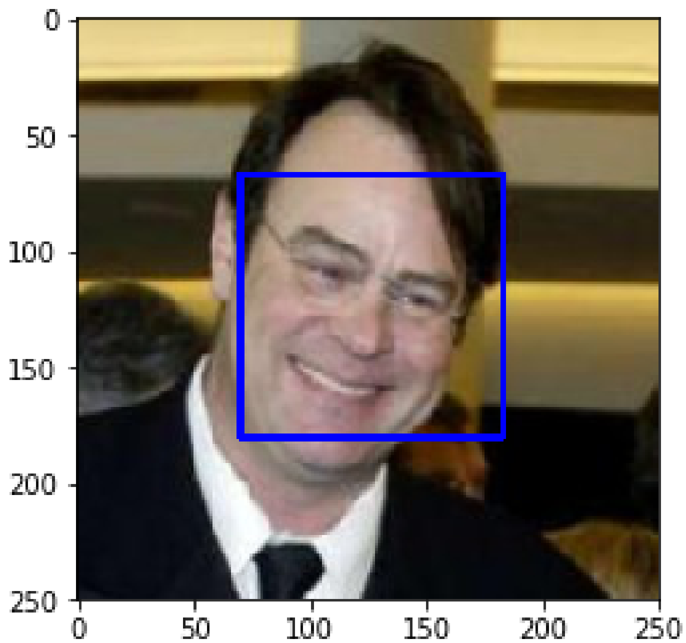
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [26]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

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

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

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

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

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

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

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



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [27]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: (You can print out your results and/or write your percentages in this cell) 98% and 17%

```
In [28]: from tqdm import tqdm

         human_files_short = human_files[:100]
         dog_files_short = dog_files[:100]

         #-#-# Do NOT modify the code above this line. #-#-#

         ## TODO: Test the performance of the face_detector algorithm
         ## on the images in human_files_short and dog_files_short.
         human_acc = np.zeros((100))
         dog_acc = np.zeros((100))
         for (i,human_f) in tqdm(enumerate(human_files_short)):
             human_acc[i] = int(face_detector(human_f))

         print(np.round(np.sum(human_acc)), "% of human images have a face detected")

         for (i,dog_f) in tqdm(enumerate(dog_files_short)):
             dog_acc[i] = int(face_detector(dog_f))

         print(np.round(np.sum(dog_acc)), "% of dog images have a face detected")
```

```
100it [00:02, 35.51it/s]
0it [00:00, ?it/s]
```

```
98.0 % of human images have a face detected
```

```
100it [00:29,  3.37it/s]
```

```
17.0 % of dog images have a face detected
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

In [29]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

In [30]:
```
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()
```

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

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

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

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

In [31]:
```
# Fix OSError: image file is truncated (150 bytes not processed)
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [32]:
```
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

    image = Image.open(img_path).convert('RGB')
    in_transform = transforms.Compose([
                        transforms.Resize(size=(244, 244)),
                        transforms.ToTensor()])
    img = in_transform(image)[:3,:,:].unsqueeze(0)

    """if use_cuda:
        img.cuda()"""

    """img = cv2.imread(img_path)
    img.resize(224*2,224*2, 3, 1)
    img = img.T
    #img = img.reshape(1,3,img.shape[1],img.shape[1])
    print(img.shape)
    img_t = torch.Tensor(img)
    with torch.no_grad():
        output = VGG16.forward(img_t)
```

```
        output = np.array(output.detach().numpy()).reshape(1000)
        print(output.shape, np.argmax(output), np.max(output))"""
        VGG16.cpu()
        VGG16.eval()
        output = VGG16(img)
        output = np.array(output.detach().numpy()).reshape(1000)
        return int(np.array(np.argmax(output))) # predicted class index
```

In [33]:
```
VGG16_predict(dog_files_short[10])
```

Out[33]:
```
243
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

In [34]:
```
### 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 <= 268) & (prediction >= 151)) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer: 0% and 99%

In [35]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_file = [dog_detector(file) for file in human_files_short]
dog_file = [dog_detector(file) for file in dog_files_short]

p_detected_dog_in_human_file = human_file.count(True) / len(human_file)
p_detected_dog_in_dog_file = dog_file.count(True) / len(dog_file)

print("Percentage of the images in human_files_short have a detected dog: {}%".format(p
print("Percentage of the images in dog_files_short have a detected dog {}%".format(p_de
```

```
Percentage of the images in human_files_short have a detected dog: 0.0%
Percentage of the images in dog_files_short have a detected dog 99.0%
```

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 [36]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

In [37]:
```
dog_files
```

Out[37]:
```
array(['/data/dog_images/train/103.Mastiff/Mastiff_06833.jpg',
       '/data/dog_images/train/103.Mastiff/Mastiff_06826.jpg',
       '/data/dog_images/train/103.Mastiff/Mastiff_06871.jpg', ...,
       '/data/dog_images/valid/100.Lowchen/Lowchen_06682.jpg',
       '/data/dog_images/valid/100.Lowchen/Lowchen_06708.jpg',
       '/data/dog_images/valid/100.Lowchen/Lowchen_06684.jpg'],
      dtype='<U106')
```

In [38]:
```
!ls ./images
```

```
American_water_spaniel_00648.jpg  Labrador_retriever_06457.jpg
Brittany_02625.jpg                  sample_cnn.png
Curly-coated_retriever_03896.jpg  sample_dog_output.png
Labrador_retriever_06449.jpg          sample_human_output.png
Labrador_retriever_06455.jpg          Welsh_springer_spaniel_08203.jpg
```

In [39]:
```
import os
from torchvision import datasets

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

data_dir = '/data/dog_images/' #'images/'
train_dir = data_dir+ 'train/'
valid_dir = data_dir+ 'valid/'
test_dir = data_dir+ 'test/'

"""All pre-trained models expect input images normalized in the same way, i.e. mini-bat
shape (3 x H x W), where H and W are expected to be at least 224. The images have to be
and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
#transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

load_transforms = {'train': transforms.Compose([transforms.Resize(size=(224,224)),
                                     transforms.RandomHorizontalFlip(), #vertical flip
                                     transforms.RandomRotation(60),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406], s
                   'valid': transforms.Compose([transforms.Resize(size=(224,224)),
```

```
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], s
                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], s
                  }

train_data = datasets.ImageFolder(train_dir, transform=load_transforms['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=load_transforms['valid'])
test_data = datasets.ImageFolder(test_dir, transform=load_transforms['test'])

batch_size = 16
num_workers = 0

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
}
```

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 used a resizing function which rescales and interpolates the images. The input tensor size is (3,224,224) since it is the size of images from ImageNet. - I decided to augment the training dataset by random horizontal flips and random rotations by (-60, 60) degrees.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [40]: VGG16.modules
```

```
Out[40]: <bound method Module.modules of VGG(
           (features): Sequential(
```

```
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)>
```

In [41]:
```
import torch.nn as nn
import torch.nn.functional as F

N_CLASSES = 133
b = 64 #base
```

```python
# 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, b, kernel_size=(3, 3), stride=1, padding=1)
        self.conv1a = nn.Conv2d(b, b, kernel_size=(3, 3), stride=1, padding=1)
        self.conv2 = nn.Conv2d(b, b*2, kernel_size=(3, 3), stride=1, padding=1)
        self.conv2a = nn.Conv2d(b*2, b*2, kernel_size=(3, 3), stride=1, padding=1)
        self.conv3 = nn.Conv2d(b*2, b*4, kernel_size=(3, 3), padding=1)
        self.conv4 = nn.Conv2d(b*4, b*8, kernel_size=(3, 3), padding=1)

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

        self.fc1 = nn.Linear(49*b*8*4, 256*4)
        self.fc2 = nn.Linear(256*4, N_CLASSES)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        #x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        #x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = F.relu(self.conv4(x))
        x = self.pool(x)

        x = x.view(-1, 4*49*b*8) #x.flatten()

        x = self.dropout(x)
        x = F.relu(self.fc1(x))# softmax to get probabilities

        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()
```

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

```
In [42]: model_scratch.modules
```

```
Out[42]: <bound method Module.modules of Net(
           (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv1a): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv2a): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv4): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
           (fc1): Linear(in_features=100352, out_features=1024, bias=True)
           (fc2): Linear(in_features=1024, out_features=133, bias=True)
           (dropout): Dropout(p=0.25)
         )>
```

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

Answer: I followed the common structure of CNN classifiers and VGG in particular: the first layers are convolutional layers, the number of features increases in higher layers. The feature extractor is followed by flattening of the feature tensor and the classifier: 2 dense layers with activation functions. The kernel size of (3, 3) is the most popular, having a number of features as a power of 2 is also a standart.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [43]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.05) #, momentum=0.9, dam
         #optim.Adam(model_scratch.parameters())
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

```
In [44]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
```

```
# initialize tracker for minimum validation loss
valid_loss_min = np.Inf

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

    ###################
    # train the model #
    ###################
    model.train()
    for batch_idx, (data, target) in enumerate(loaders['train']):
        # 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
        #print(data.shape,target.shape)
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(data)
        #print(data.shape,target.shape, outputs.shape)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
        # print statistics
        #running_loss += loss.item()
        if batch_idx % 200 == 0:    # print every 200 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch , batch_idx , train_loss))
            #running_loss = 0.0

#print('Finished Training')
    ######################
    # 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
```

```
                #optimizer.zero_grad() ?
                output = model(data)
                loss = criterion(output, target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch,
                train_loss,
                valid_loss
                ))

            ## TODO: save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                torch.save(model.state_dict(), save_path)
                print("valid_loss decreased from ",str(valid_loss_min)," to ", str(valid_lo
                valid_loss_min = valid_loss
        # return trained model
        return model
```

In [45]:
```
%%time

# train the model
model_scratch = train(20, 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'))
```

```
[1,     0] loss: 4.893
[1,   200] loss: 4.883
[1,   400] loss: 4.848
Epoch: 1         Training Loss: 4.842838         Validation Loss: 4.697783
valid_loss decreased from  inf  to  tensor(4.6978, device='cuda:0')
[2,     0] loss: 4.695
[2,   200] loss: 4.605
[2,   400] loss: 4.555
Epoch: 2         Training Loss: 4.552472         Validation Loss: 4.587571
valid_loss decreased from  tensor(4.6978, device='cuda:0')  to  tensor(4.5876, device='cuda:0')
[3,     0] loss: 4.332
[3,   200] loss: 4.408
[3,   400] loss: 4.391
Epoch: 3         Training Loss: 4.387720         Validation Loss: 4.473853
valid_loss decreased from  tensor(4.5876, device='cuda:0')  to  tensor(4.4739, device='cuda:0')
[4,     0] loss: 4.348
[4,   200] loss: 4.253
[4,   400] loss: 4.249
Epoch: 4         Training Loss: 4.242969         Validation Loss: 4.697742
```

```
[5,     0] loss: 4.643
[5,   200] loss: 4.127
[5,   400] loss: 4.114
Epoch: 5         Training Loss: 4.118827         Validation Loss: 4.375939
valid_loss decreased from  tensor(4.4739, device='cuda:0')  to  tensor(4.3759, device='cuda:0')
[6,     0] loss: 3.689
[6,   200] loss: 4.017
[6,   400] loss: 4.004
Epoch: 6         Training Loss: 3.996676         Validation Loss: 4.091784
valid_loss decreased from  tensor(4.3759, device='cuda:0')  to  tensor(4.0918, device='cuda:0')
[7,     0] loss: 4.239
[7,   200] loss: 3.900
[7,   400] loss: 3.880
Epoch: 7         Training Loss: 3.879401         Validation Loss: 3.944465
valid_loss decreased from  tensor(4.0918, device='cuda:0')  to  tensor(3.9445, device='cuda:0')
[8,     0] loss: 3.271
[8,   200] loss: 3.775
[8,   400] loss: 3.753
Epoch: 8         Training Loss: 3.745130         Validation Loss: 3.869298
valid_loss decreased from  tensor(3.9445, device='cuda:0')  to  tensor(3.8693, device='cuda:0')
[9,     0] loss: 3.878
[9,   200] loss: 3.640
[9,   400] loss: 3.641
Epoch: 9         Training Loss: 3.640849         Validation Loss: 4.053578
[10,     0] loss: 3.982
[10,   200] loss: 3.516
[10,   400] loss: 3.511
Epoch: 10         Training Loss: 3.510232         Validation Loss: 3.935733
[11,     0] loss: 3.548
[11,   200] loss: 3.359
[11,   400] loss: 3.387
Epoch: 11         Training Loss: 3.395982         Validation Loss: 3.739549
valid_loss decreased from  tensor(3.8693, device='cuda:0')  to  tensor(3.7395, device='cuda:0')
[12,     0] loss: 3.692
[12,   200] loss: 3.292
[12,   400] loss: 3.274
Epoch: 12         Training Loss: 3.270237         Validation Loss: 3.723879
valid_loss decreased from  tensor(3.7395, device='cuda:0')  to  tensor(3.7239, device='cuda:0')
[13,     0] loss: 3.150
[13,   200] loss: 3.131
[13,   400] loss: 3.156
Epoch: 13         Training Loss: 3.163692         Validation Loss: 3.642149
valid_loss decreased from  tensor(3.7239, device='cuda:0')  to  tensor(3.6421, device='cuda:0')
[14,     0] loss: 2.356
[14,   200] loss: 2.965
[14,   400] loss: 3.022
Epoch: 14         Training Loss: 3.022680         Validation Loss: 3.594607
valid_loss decreased from  tensor(3.6421, device='cuda:0')  to  tensor(3.5946, device='cuda:0')
```

```
[15,     0] loss: 3.043
[15,   200] loss: 2.872
[15,   400] loss: 2.891
Epoch: 15         Training Loss: 2.893548         Validation Loss: 3.611990
[16,     0] loss: 2.172
[16,   200] loss: 2.692
[16,   400] loss: 2.772
Epoch: 16         Training Loss: 2.778368         Validation Loss: 3.606719
[17,     0] loss: 2.178
[17,   200] loss: 2.544
[17,   400] loss: 2.596
Epoch: 17         Training Loss: 2.605014         Validation Loss: 3.733598
[18,     0] loss: 1.970
[18,   200] loss: 2.436
[18,   400] loss: 2.470
Epoch: 18         Training Loss: 2.471794         Validation Loss: 4.065575
[19,     0] loss: 2.406
[19,   200] loss: 2.263
[19,   400] loss: 2.350
Epoch: 19         Training Loss: 2.360877         Validation Loss: 3.736340
[20,     0] loss: 1.958
[20,   200] loss: 2.138
[20,   400] loss: 2.193
Epoch: 20         Training Loss: 2.199697         Validation Loss: 3.830315
CPU times: user 40min 27s, sys: 3min 23s, total: 43min 50s
Wall time: 38min 32s
```

3211264/b/4/16/4/7/7

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [46]: 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))
```

```
In [47]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.564770


Test Accuracy: 18% (154/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

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

```
In [15]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [16]: import torchvision.models as models
         import torch.nn as nn
```

```python
## TODO: Specify model architecture
model_transfer = models.vgg19_bn(pretrained=True) #resnext50_32x4d(pretrained=True, pro

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

if use_cuda:
    model_transfer = model_transfer.cuda()
model_transfer.modules
```

```
Downloading: "https://download.pytorch.org/models/vgg19_bn-c79401a0.pth" to /root/.torch/models/
100%|| 574769405/574769405 [00:08<00:00, 67001906.69it/s]
```

Out[16]:

```
<bound method Module.modules of VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU(inplace)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (5): ReLU(inplace)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
    (9): ReLU(inplace)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
    (12): ReLU(inplace)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
    (16): ReLU(inplace)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
    (19): ReLU(inplace)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
    (22): ReLU(inplace)
    (23): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
    (25): ReLU(inplace)
    (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (27): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
    (29): ReLU(inplace)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
      (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
      (32): ReLU(inplace)
      (33): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
      (35): ReLU(inplace)
      (36): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
      (38): ReLU(inplace)
      (39): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
      (42): ReLU(inplace)
      (43): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (44): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
      (45): ReLU(inplace)
      (46): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (47): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
      (48): ReLU(inplace)
      (49): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (50): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
      (51): ReLU(inplace)
      (52): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace)
      (2): Dropout(p=0.5)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace)
      (5): Dropout(p=0.5)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )>
```

```
In [22]: model_transfer.classifier[-1]
```

```
Out[22]: Linear(in_features=4096, out_features=1000, bias=True)
```

```
In [17]: model_transfer.classifier[-1] = nn.Linear(in_features=4096, out_features=N_CLASSES, bia
```

```
In [18]: for param in model_transfer.classifier[-1].parameters():
             param.requires_grad = True
         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 selected VGG19 with batchnorm based on the comparisson table of models pre-trained on ImageNet https://pytorch.org/docs/stable/torchvision/models.html#classification

(Top-1 error). It is suitable because it has already trained feature extractor that was proved to be good at classification of ImageNet images including classification of 133 dog breeds.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [19]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier[-1].parameters(), lr=0.04)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [20]: # train the model
         n_epochs = 30
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
                                criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
[1,     0] loss: 4.956
[1,   200] loss: 3.242
[1,   400] loss: 2.732
Epoch: 1         Training Loss: 2.708948         Validation Loss: 1.017700
valid_loss decreased from  inf  to  tensor(1.0177, device='cuda:0')
[2,     0] loss: 1.898
[2,   200] loss: 1.971
[2,   400] loss: 1.940
Epoch: 2         Training Loss: 1.932941         Validation Loss: 0.749977
valid_loss decreased from  tensor(1.0177, device='cuda:0')  to  tensor(0.7500, device='cuda:0')
[3,     0] loss: 1.769
[3,   200] loss: 1.859
[3,   400] loss: 1.866
Epoch: 3         Training Loss: 1.868006         Validation Loss: 0.691337
valid_loss decreased from  tensor(0.7500, device='cuda:0')  to  tensor(0.6913, device='cuda:0')
[4,     0] loss: 1.475
[4,   200] loss: 1.795
[4,   400] loss: 1.786
Epoch: 4         Training Loss: 1.779219         Validation Loss: 0.787941
[5,     0] loss: 2.455
[5,   200] loss: 1.804
[5,   400] loss: 1.773
Epoch: 5         Training Loss: 1.760853         Validation Loss: 0.729369
[6,     0] loss: 1.607
[6,   200] loss: 1.775
[6,   400] loss: 1.781
Epoch: 6         Training Loss: 1.777605         Validation Loss: 0.675390
valid_loss decreased from  tensor(0.6913, device='cuda:0')  to  tensor(0.6754, device='cuda:0')
```

```
[7,     0] loss: 1.634
[7,   200] loss: 1.782
[7,   400] loss: 1.779
Epoch: 7         Training Loss: 1.777059         Validation Loss: 0.661113
valid_loss decreased from  tensor(0.6754, device='cuda:0')  to  tensor(0.6611, device='cuda:0')
[8,     0] loss: 2.454
[8,   200] loss: 1.709
[8,   400] loss: 1.744
Epoch: 8         Training Loss: 1.745601         Validation Loss: 0.663266
[9,     0] loss: 1.142
[9,   200] loss: 1.725
[9,   400] loss: 1.743
Epoch: 9         Training Loss: 1.749842         Validation Loss: 0.630733
valid_loss decreased from  tensor(0.6611, device='cuda:0')  to  tensor(0.6307, device='cuda:0')
[10,     0] loss: 1.217
[10,   200] loss: 1.722
[10,   400] loss: 1.761
Epoch: 10         Training Loss: 1.761120         Validation Loss: 0.689626
[11,     0] loss: 1.368
[11,   200] loss: 1.734
[11,   400] loss: 1.741
Epoch: 11         Training Loss: 1.743922         Validation Loss: 0.632317
[12,     0] loss: 1.457
[12,   200] loss: 1.751
[12,   400] loss: 1.763
Epoch: 12         Training Loss: 1.764378         Validation Loss: 0.646411
[13,     0] loss: 2.032
[13,   200] loss: 1.734
[13,   400] loss: 1.740
Epoch: 13         Training Loss: 1.745529         Validation Loss: 0.591325
valid_loss decreased from  tensor(0.6307, device='cuda:0')  to  tensor(0.5913, device='cuda:0')
[14,     0] loss: 1.749
[14,   200] loss: 1.677
[14,   400] loss: 1.696
Epoch: 14         Training Loss: 1.700012         Validation Loss: 0.693056
[15,     0] loss: 1.529
[15,   200] loss: 1.721
[15,   400] loss: 1.740
Epoch: 15         Training Loss: 1.734405         Validation Loss: 0.663421
[16,     0] loss: 1.727
[16,   200] loss: 1.704
[16,   400] loss: 1.717
Epoch: 16         Training Loss: 1.720902         Validation Loss: 0.690628
[17,     0] loss: 1.390
[17,   200] loss: 1.826
[17,   400] loss: 1.799
Epoch: 17         Training Loss: 1.807306         Validation Loss: 0.757025
[18,     0] loss: 2.784
```

```
[18,   200] loss: 1.720
[18,   400] loss: 1.753
Epoch: 18         Training Loss: 1.744554         Validation Loss: 0.660554
[19,     0] loss: 2.342
[19,   200] loss: 1.735
[19,   400] loss: 1.761
Epoch: 19         Training Loss: 1.760177         Validation Loss: 0.684101
[20,     0] loss: 1.765
[20,   200] loss: 1.739
[20,   400] loss: 1.726
Epoch: 20         Training Loss: 1.731982         Validation Loss: 0.681321
[21,     0] loss: 2.320
[21,   200] loss: 1.775
[21,   400] loss: 1.796
Epoch: 21         Training Loss: 1.792711         Validation Loss: 0.646634
[22,     0] loss: 1.694
[22,   200] loss: 1.713
[22,   400] loss: 1.742
Epoch: 22         Training Loss: 1.746206         Validation Loss: 0.682373
[23,     0] loss: 1.605
[23,   200] loss: 1.702
[23,   400] loss: 1.680
Epoch: 23         Training Loss: 1.684169         Validation Loss: 0.659801
[24,     0] loss: 2.093
[24,   200] loss: 1.681
[24,   400] loss: 1.716
Epoch: 24         Training Loss: 1.718540         Validation Loss: 0.671668
[25,     0] loss: 2.064
[25,   200] loss: 1.703
[25,   400] loss: 1.694
Epoch: 25         Training Loss: 1.686543         Validation Loss: 0.652151
[26,     0] loss: 1.973
[26,   200] loss: 1.731
[26,   400] loss: 1.732
Epoch: 26         Training Loss: 1.731042         Validation Loss: 0.609297
[27,     0] loss: 1.253
[27,   200] loss: 1.765
[27,   400] loss: 1.788
Epoch: 27         Training Loss: 1.795531         Validation Loss: 0.659183
[28,     0] loss: 2.466
[28,   200] loss: 1.715
[28,   400] loss: 1.734
Epoch: 28         Training Loss: 1.735393         Validation Loss: 0.679942
[29,     0] loss: 2.204
[29,   200] loss: 1.746
[29,   400] loss: 1.740
Epoch: 29         Training Loss: 1.734465         Validation Loss: 0.601578
[30,     0] loss: 1.178
```

```
[30,   200] loss: 1.744
[30,   400] loss: 1.753
Epoch: 30         Training Loss: 1.755173         Validation Loss: 0.635635
```

In [21]:
```
# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

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

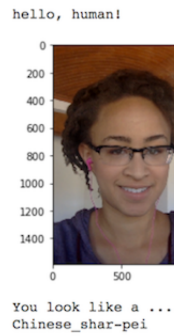
In [22]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.647432


Test Accuracy: 79% (663/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

In [33]:
```
### 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

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

    image = Image.open(img_path).convert('RGB')
    in_transform = transforms.Compose([
                        transforms.Resize(size=(244, 244)),
                        transforms.ToTensor()])
    img = in_transform(image)[:3,:,:].unsqueeze(0)

    """if use_cuda:
        img.cuda()
        model_transfer.cuda()"""
    model_transfer.cpu()
    model_transfer.eval()
    class_idx = torch.argmax(model_transfer(img).cpu())

    return class_names[class_idx]
```



Sample Human Output

```
In [37]: predict_breed_transfer(dog_files_short[6])

Out[37]: 'Glen of imaal terrier'
```

Step 5: Write your Algorithm

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

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

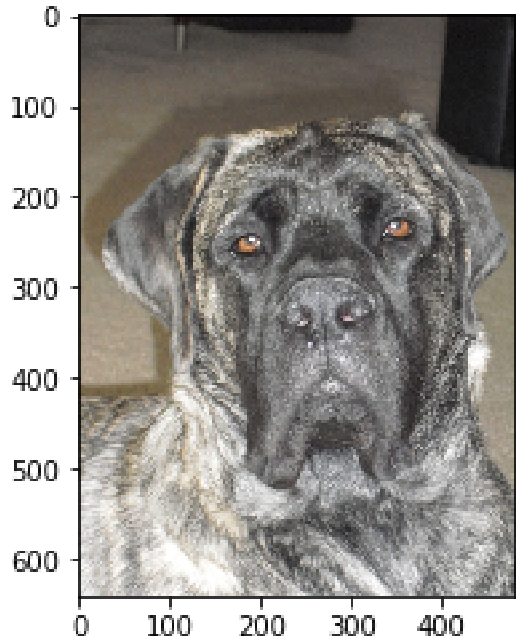
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [41]: ### 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):
                 breed = predict_breed_transfer(img_path)
                 print("Hi, dog \n Your predicted breed is: %s"%{breed})
             elif face_detector(img_path)>1:
                 print("Hey, people, there are too many of you..")
             elif face_detector(img_path)==1:
                 breed = predict_breed_transfer(img_path)
                 print("Hi, human \n You look like a %s"%{breed})
```

```
            else:
                print("No dogs or humans are detected.")
```

```
In [47]: run_app(dog_files_short[6])
```




```
Hi, dog
 Your predicted breed is: {'Glen of imaal terrier'}
```

Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

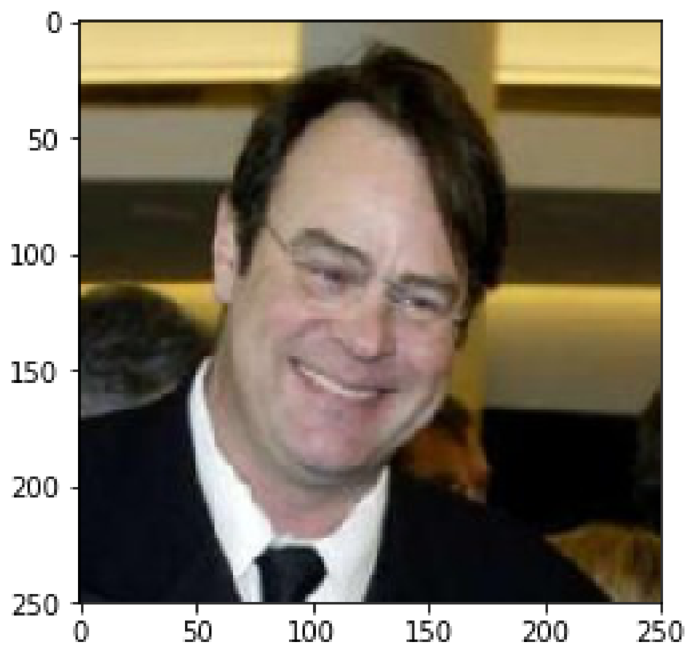
Answer: (Three possible points for improvement) - 1) Improve the model's accuracy by - a. exploring more network architectures - b. combining the training set with the dog images from Imagenet - c. enlarging the training set by augmentations - d. training the model for more epochs - e. fine tuning the optimazer's parameters - 2) Provide probabities of top-5 predicted breeds -

3) Explain the net's decision using interpretability techniques such as Saliency maps, Grad-CAM, Ablation-CAM, Occlusion. It would be insightful for classification of mix-breed dogs and funny for human "breed" predictions. - 4) Make a nice web app allowing to upload an image and get an instant result without running the notebook.

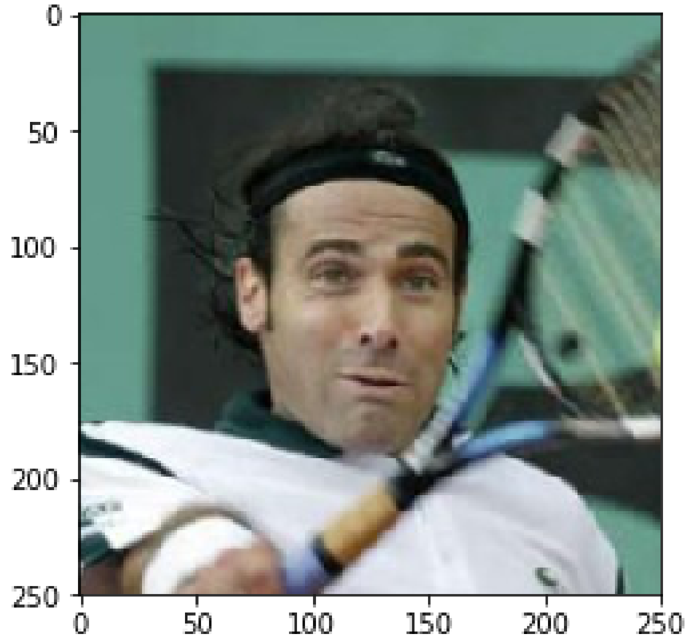
The output is worse than I expected

In [56]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[:3], dog_files[302:305])):
    print("file name: ",file)
    run_app(file)
```

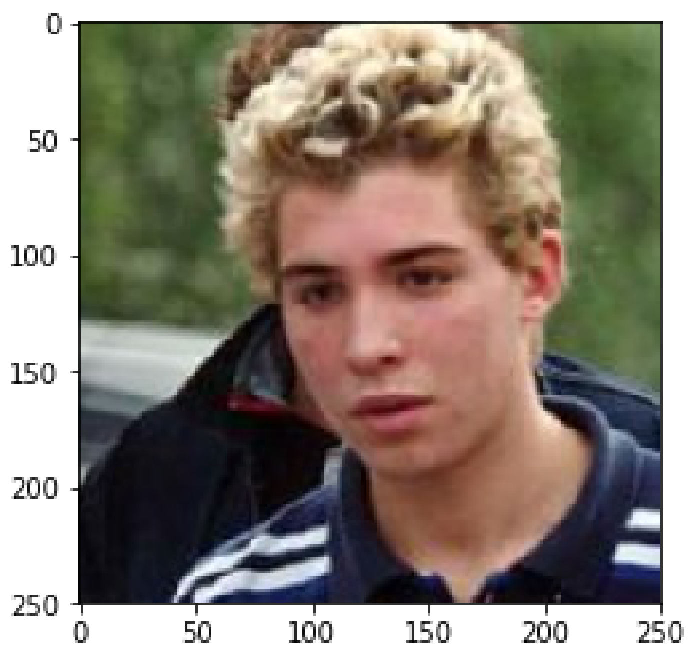
```
file name:  /data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
```




```
Hi, human
 You look like a {'American staffordshire terrier'}
file name:  /data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
```

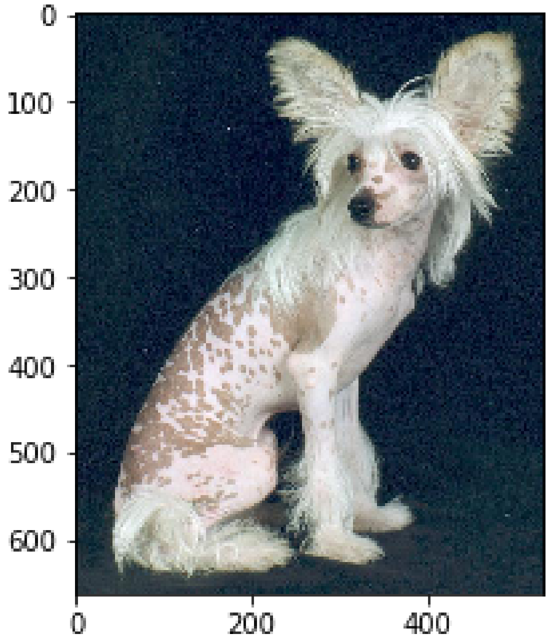


```
Hi, human
 You look like a {'Chesapeake bay retriever'}
file name:  /data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg
```

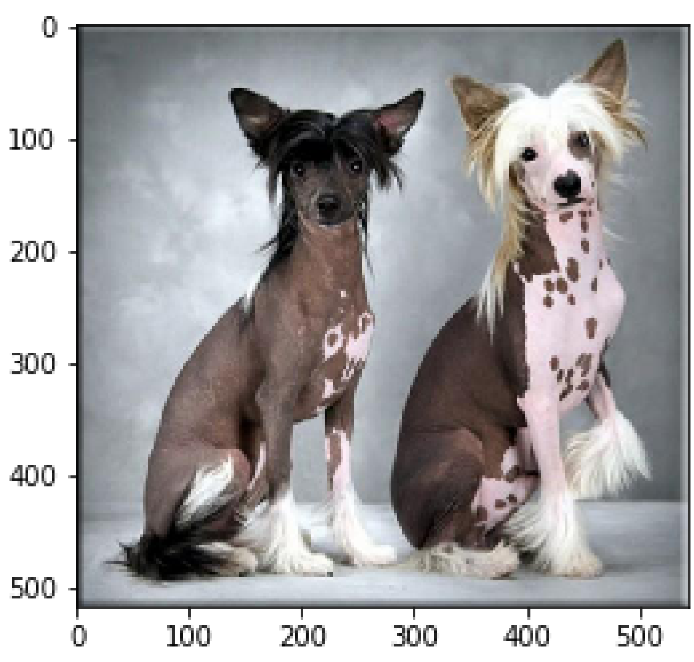


```
Hi, human
 You look like a {'Bichon frise'}
file name:  /data/dog_images/train/049.Chinese_crested/Chinese_crested_03469.jpg
```


```
Hi, dog
 Your predicted breed is: {'Chinese crested'}
file name:  /data/dog_images/train/049.Chinese_crested/Chinese_crested_03510.jpg
```




```
Hi, dog
 Your predicted breed is: {'Chinese crested'}
file name:  /data/dog_images/train/049.Chinese_crested/Chinese_crested_03522.jpg
```



```
Hi, dog
 Your predicted breed is: {'Xoloitzcuintli'}
```

In [ ]: