# dog_app

March 24, 2022

## 1 Convolutional Neural Networks

### 1.1 Project: Write an Algorithm for a Dog Identification App

---

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

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

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

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

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

## Step 0: Import Datasets

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

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

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

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

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

```
In [1]: import numpy as np
        from glob import glob

        # 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))


        print(len(np.array(glob("/data/dog_images/valid/*/*"))))
```

```
There are 13233 total human images.
There are 8351 total dog images.
835
```

## Step 1: Detect Humans

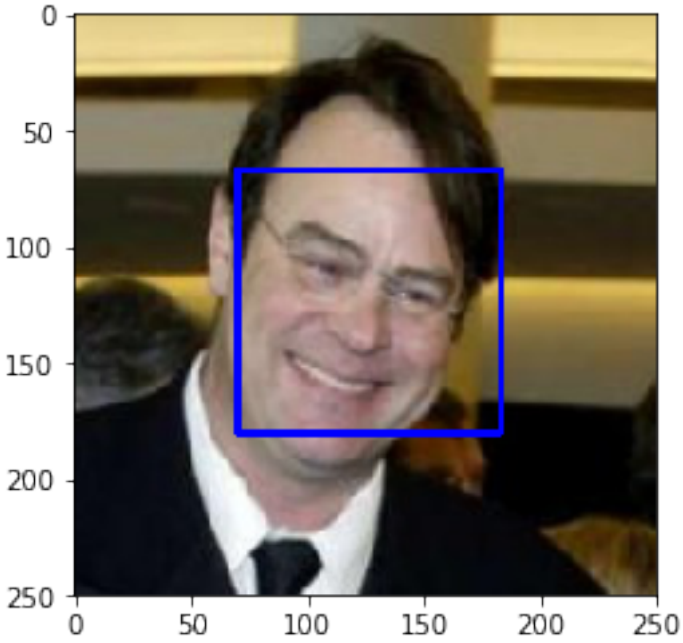
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

```
In [2]: 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 [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

### 1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
In [4]: from tqdm import tqdm

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        count_FN = 0
        count_FP = 0

        #Count the number of FN
        for file in human_files_short:
            face_detected = face_detector(file)
            if face_detected == False:
                count_FN += 1

        #Count the number of FP
        for file in dog_files_short:
            face_detected = face_detector(file)
            if face_detected == True:
                count_FP += 1
```

```
        print('there is {}% false negatives'.format(count_FN))
        print('there is {}% false positives'.format(count_FP))
```

```
there is 2% false negatives
there is 17% false positives
```

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

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

---

## Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

### 1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [6]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 109163100.94it/s]
```

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

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

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

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

```
In [7]: 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
            #expected model input:
            #- input size: 224 x 224
            #- tensor with shape: [b, 3, h, w]
            #- normalize (values found in documentation)

            transform = transforms.Compose([
                transforms.Resize((224, 224)),
                transforms.ToTensor(),
                transforms.Normalize((0.485, 0.456, 0.406),
                                     (0.229, 0.224, 0.225))])

            image = Image.open(img_path)
            image = transform(image).float()
            image = image.unsqueeze(0)

            if use_cuda:
                image = image.cuda()

            log_probabilities = VGG16(image) #model evaluation
            probabilities = torch.exp(log_probabilities) #convert model output to probabilities
            prediction = torch.max(probabilities, dim = 1) #get class with highest probability
            ## Return the *index* of the predicted class for that image
            return prediction[1].item() # return predicted class as integer
```

### 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 [8]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    dog_keys = range(151, 268)
    if VGG16_predict(img_path) in dog_keys:
        return True
    else:
        return False
```

### 1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

**Answer:**

In [9]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
count_FN = 0
count_FP = 0

#Count the number of FP
for file in human_files_short:
    dog_detected = dog_detector(file)
    if dog_detected == True:
        count_FP += 1

#Count the number of FN
for file in dog_files_short:
    dog_detected = dog_detector(file)
    if dog_detected == False:
        count_FN += 1

print('there is {}% false negatives'.format(count_FN))
print('there is {}% false positives'.format(count_FP))
```

```
there is 0% false negatives
there is 0% false positives
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

In [10]:
```
### (Optional)
### TODO: Report the performance of another pre-trained network.
### Feel free to use as many code cells as needed.
import torch
import torchvision.models as models

# define VGG16 model
model_resnet = models.resnet50(pretrained=True)

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:00<00:00, 103122169.97it/s]
```

In [11]:
```
from PIL import Image
import torchvision.transforms as transforms

def resnet18_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
    #expected model input:
    #- input size: 224 x 224
    #- tensor with shape: [b, 3, h, w]
    #- normalize (values found in documentation)

    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406),
```

```
                                    (0.229, 0.224, 0.225))])

        image = Image.open(img_path)
        image = transform(image).float()
        image = image.unsqueeze(0)

        if use_cuda:
            image = image.cuda()

        log_probabilities = model_resnet(image) #model evaluation
        probabilities = torch.exp(log_probabilities) #convert model output to probabilities
        prediction = torch.max(probabilities, dim = 1) #get class with highest probability
        ## Return the *index* of the predicted class for that image
        return prediction[1].item() # return predicted class as integer

    ### returns "True" if a dog is detected in the image stored at img_path
    def dog_detector_resnet(img_path):
        ## TODO: Complete the function.
        dog_keys = range(151, 268)
        if resnet18_predict(img_path) in dog_keys:
            return True
        else:
            return False
```

```
In [12]: count_FN = 0
         count_FP = 0

         #Count the number of FP
         for file in human_files_short:
             dog_detected = dog_detector_resnet(file)
             if dog_detected == True:
                 count_FP += 1

         #Count the number of FN
         for file in dog_files_short:
             dog_detected = dog_detector_resnet(file)
             if dog_detected == False:
                 count_FN += 1

         print('there is {}% false negatives'.format(count_FN))
         print('there is {}% false positives'.format(count_FP))
```

```
there is 100% false negatives
there is 0% false positives
```

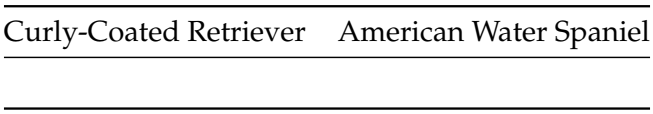
---

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

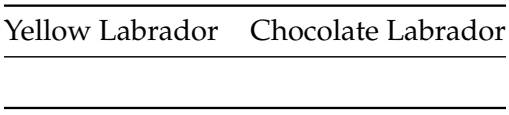
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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


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




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.




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 [13]: import os
         from torchvision import datasets
         from torchvision import transforms
         from torch.utils.data import DataLoader

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

```
#Specifiy train, test and validation folders
traindir = "/data/dog_images/train"
testdir = "/data/dog_images/test"
valdir = "/data/dog_images/valid"

#specify batch size
batch_size = 32

#Specifiy transformations for train, test and validaiton sets
image_transforms = {

    'train':
    transforms.Compose([
        transforms.RandomRotation(degrees=15),
        transforms.RandomHorizontalFlip(),
        transforms.Resize(size=[224, 224]),  #VGG16 standards
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])  # VGG16 standards
        ]),

    'test':
    transforms.Compose([
        transforms.Resize(size=[224, 224]),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
        ]),

    'valid':
    transforms.Compose([
        transforms.Resize(size=[224, 224]),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
        ]),
    }

# Load the datasets
data = {
    'train':
    datasets.ImageFolder(root=traindir, transform=image_transforms['train']),
    'test':
    datasets.ImageFolder(root=testdir, transform=image_transforms['test']),
    'valid':
    datasets.ImageFolder(root=valdir, transform=image_transforms['valid']),
    }
```

```
# Dataloader iterators
loaders_scratch = {
    'train': DataLoader(data['train'],
                        shuffle = True,
                        batch_size = batch_size,
                        drop_last = True),
    'test': DataLoader(data['test'],
                       shuffle = True,
                       batch_size = batch_size,
                       drop_last = True),
    'valid': DataLoader(data['valid'],
                        shuffle = True,
                        batch_size = batch_size,
                        drop_last = True)
    }
```

**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**: Pytorch expects the input to be of size: 224x225xbatch_size. Therfore, we resize each image to this size and split our dataset into batches. Furthermore, the images need to be normalized using the values [0.485, 0.456, 0.406], [0.229, 0.224, 0.225], as is described in the documentation linked above. We also transform the data into Tensors.

The dataset augmented using random rotation and random flip, to addd some randomness to the dataset. This helps to prevent overfitting.

### 1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [14]: import torch.nn as nn
         import torch.nn.functional as F
         import math

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
                 self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
                 self.conv3 = nn.Conv2d(32, 64, kernel_size = 3, padding=1)

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

```
        self.fc1 = nn.Linear(64 * 28 * 28, 128)
        self.fc2 = nn.Linear(128, 133)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        ## Define forward propagation
        #convulutional layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.dropout(x)

        #flatten
        x = x.view(-1, 64 * 28 * 28)

        #linear layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))

        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

**Answer:**

First, I created a first outline of the CNN architecture, choose the training parameters and trained and tested the model. Then I went back and played around with the architecture and training parameters untill I reached a satisfying accuracy.

The final CNN architecture contains three layers, with each layer containing a convolutional, non-linearity and pooling step. Then there is a dropout step, followed by two linear layer with another dropout step in between.

### 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 [15]: import torch.optim as optim
```

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(),lr = 0.01, momentum = 0.9)
```

#### 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 [16]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

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

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

            ## find the loss and update the model parameters accordingly
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            ## record the average training loss, using something like
            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 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("Validation loss decreased; model saved.")
                min_valid_loss = valid_loss
        # return trained model
        return model

    # 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'))
```

```
Epoch: 1         Training Loss: 4.891045         Validation Loss: 4.882836
Validation loss decreased; model saved.
Epoch: 2         Training Loss: 4.859623         Validation Loss: 4.772453
Validation loss decreased; model saved.
Epoch: 3         Training Loss: 4.774552         Validation Loss: 4.680654
Validation loss decreased; model saved.
Epoch: 4         Training Loss: 4.662278         Validation Loss: 4.509542
Validation loss decreased; model saved.
Epoch: 5         Training Loss: 4.570396         Validation Loss: 4.494501
Validation loss decreased; model saved.
Epoch: 6         Training Loss: 4.504544         Validation Loss: 4.403435
Validation loss decreased; model saved.
Epoch: 7         Training Loss: 4.435193         Validation Loss: 4.332157
Validation loss decreased; model saved.
Epoch: 8         Training Loss: 4.368533         Validation Loss: 4.287293
Validation loss decreased; model saved.
```

```
Epoch: 9         Training Loss: 4.288458         Validation Loss: 4.127836
Validation loss decreased; model saved.
Epoch: 10         Training Loss: 4.238190         Validation Loss: 4.137369
Validation loss decreased; model saved.
Epoch: 11         Training Loss: 4.168552         Validation Loss: 4.190521
Validation loss decreased; model saved.
Epoch: 12         Training Loss: 4.143503         Validation Loss: 4.111053
Validation loss decreased; model saved.
Epoch: 13         Training Loss: 4.085019         Validation Loss: 3.964564
Validation loss decreased; model saved.
Epoch: 14         Training Loss: 4.041239         Validation Loss: 3.971485
Validation loss decreased; model saved.
Epoch: 15         Training Loss: 3.988674         Validation Loss: 4.030461
Validation loss decreased; model saved.
Epoch: 16         Training Loss: 3.940645         Validation Loss: 3.909655
Validation loss decreased; model saved.
Epoch: 17         Training Loss: 3.888275         Validation Loss: 3.904326
Validation loss decreased; model saved.
Epoch: 18         Training Loss: 3.826378         Validation Loss: 3.882841
Validation loss decreased; model saved.
Epoch: 19         Training Loss: 3.794665         Validation Loss: 3.824771
Validation loss decreased; model saved.
Epoch: 20         Training Loss: 3.785203         Validation Loss: 3.894913
Validation loss decreased; model saved.
```

#### 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 [17]:
```
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: 3.876513


Test Accuracy: 10% (86/832)
```

---

## 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 [18]: ## TODO: Specify data loaders
         ##Using the same dataloaders as before.
         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 [19]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
```

```
        #get pretrained model
        model_transfer = models.vgg16(pretrained = True)

        # Freeze training for all layers
        for param in model_transfer.features.parameters():
            param.require_grad = False

        # Newly created modules have require_grad=True by default
        num_features = model_transfer.classifier[6].in_features
        features = list(model_transfer.classifier.children())[:-1] # Remove last layer
        features.extend([nn.Linear(num_features, 133)]) # Add our layer with 4 outputs
        model_transfer.classifier = nn.Sequential(*features) # Replace the model classifier
        print(model_transfer)

        #move to GPU
        if use_cuda:
            model_transfer = model_transfer.cuda()
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
```

```
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
```

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

**Answer:** First, I implemented VGG16, as advised in the assignment. Next, I tried ResNet18, but this achieved a lower accuracy, so I went back to VGG16. The model is the same as in the original model, except for the final linear layer, which was adapted to differentiate between 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 [20]: import torch.optim as optim
         from torch.optim import lr_scheduler

         criterion_transfer = nn.CrossEntropyLoss() #nn.sparsemax()?
         optimizer_transfer = optim.SGD(model_transfer.parameters(), lr = 0.01, momentum = 0.9,
```

#### 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 [21]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             '''returns trained model'''

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

             for epoch in range(1, n_epochs + 1):
```

```
#initialize variables
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()

    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    #scheduler.step()

    #record the average training loss
    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

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

##Reduce the learning rate according to the scheduler


## Save the model if validation loss has decreased
if valid_loss < valid_loss_min:
    torch.save(model.state_dict(), save_path)
    print("Validation loss decreased; model saved.")
    min_valid_loss = valid_loss
```

```
        # return trained model
        return model


# train the model
model_transfer = train(20,
                       loaders_transfer,
                       model_transfer,
                       optimizer_transfer,
                       criterion_transfer,
                       use_cuda,
                       'model_transfer.pt')

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

```
Epoch: 1         Training Loss: 2.695596         Validation Loss: 1.744397
Validation loss decreased; model saved.
Epoch: 2         Training Loss: 1.521296         Validation Loss: 1.305221
Validation loss decreased; model saved.
Epoch: 3         Training Loss: 1.156904         Validation Loss: 1.032776
Validation loss decreased; model saved.
Epoch: 4         Training Loss: 0.960985         Validation Loss: 1.088657
Validation loss decreased; model saved.
Epoch: 5         Training Loss: 0.853646         Validation Loss: 0.958716
Validation loss decreased; model saved.
Epoch: 6         Training Loss: 0.783738         Validation Loss: 1.016468
Validation loss decreased; model saved.
Epoch: 7         Training Loss: 0.658246         Validation Loss: 0.965463
Validation loss decreased; model saved.
Epoch: 8         Training Loss: 0.601631         Validation Loss: 1.033512
Validation loss decreased; model saved.
Epoch: 9         Training Loss: 0.514729         Validation Loss: 1.043511
Validation loss decreased; model saved.
Epoch: 10         Training Loss: 0.503706         Validation Loss: 1.030100
Validation loss decreased; model saved.
Epoch: 11         Training Loss: 0.505275         Validation Loss: 1.063623
Validation loss decreased; model saved.
Epoch: 12         Training Loss: 0.425153         Validation Loss: 0.916429
Validation loss decreased; model saved.
Epoch: 13         Training Loss: 0.376692         Validation Loss: 1.141220
Validation loss decreased; model saved.
Epoch: 14         Training Loss: 0.342494         Validation Loss: 1.027357
Validation loss decreased; model saved.
Epoch: 15         Training Loss: 0.343409         Validation Loss: 1.035364
Validation loss decreased; model saved.
Epoch: 16         Training Loss: 0.340821         Validation Loss: 1.119737
Validation loss decreased; model saved.
```

```
Epoch: 17         Training Loss: 0.340197         Validation Loss: 1.090501
Validation loss decreased; model saved.
Epoch: 18         Training Loss: 0.269612         Validation Loss: 1.093122
Validation loss decreased; model saved.
Epoch: 19         Training Loss: 0.269013         Validation Loss: 1.088967
Validation loss decreased; model saved.
Epoch: 20         Training Loss: 0.241796         Validation Loss: 1.008078
Validation loss decreased; model saved.
```

#### 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]: 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))


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

```
Test Loss: 1.220313
```

```
Test Accuracy: 71% (599/832)
```

### 1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

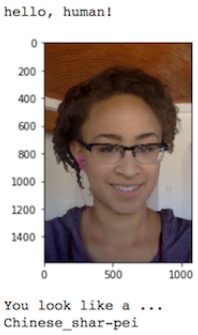
Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [44]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         import torch
         import torchvision.transforms as transforms
         from PIL import Image

         # 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['test'].dataset.

         def predict_breed_transfer(img_path):
             #same image transformations as before
             transform = transforms.Compose([
                 transforms.Resize(size=[224, 224]),
                 transforms.ToTensor(),
                 transforms.Normalize([0.485, 0.456, 0.406],
                                      [0.229, 0.224, 0.225])
                 ])

             #load image
             img = Image.open(img_path)
             img = transform(img)
             img.unsqueeze_(0) #because not in batches this time

             if use_cuda:
                 img = img.cuda()

             #predict
             output = model_transfer(img)
             _, prediction = torch.max(output, 1)

             return class_names[np.squeeze(prediction.cpu().numpy())]
```

---

## 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.




Sample Human Output

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 [46]:
```
### 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
    contains_dog = dog_detector(img_path)
    contains_human = face_detector(img_path)
    dog_breed = predict_breed_transfer(img_path)

    if contains_dog:
        print('The image contains a dog!')
        plt.imshow(cv2.imread(img_path))
        plt.show()
        print('It is a {}'.format(dog_breed))

    elif contains_human:
        print('The image contains a human!')
        plt.imshow(cv2.imread(img_path))
        plt.show()
        print('This human resembles a {}'.format(dog_breed))

    else:
        plt.imshow(cv2.imread(img_path))
        plt.show()
        print('No dogs or humans detected.')
```

---

## 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!

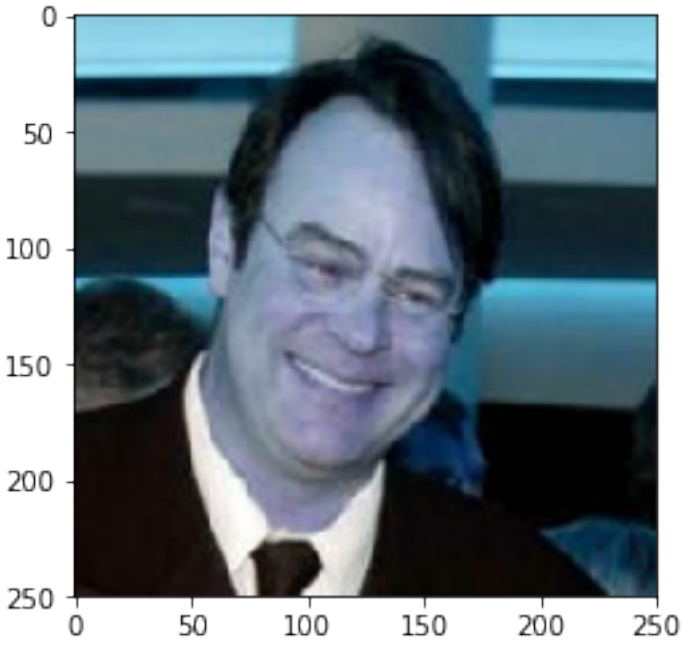
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

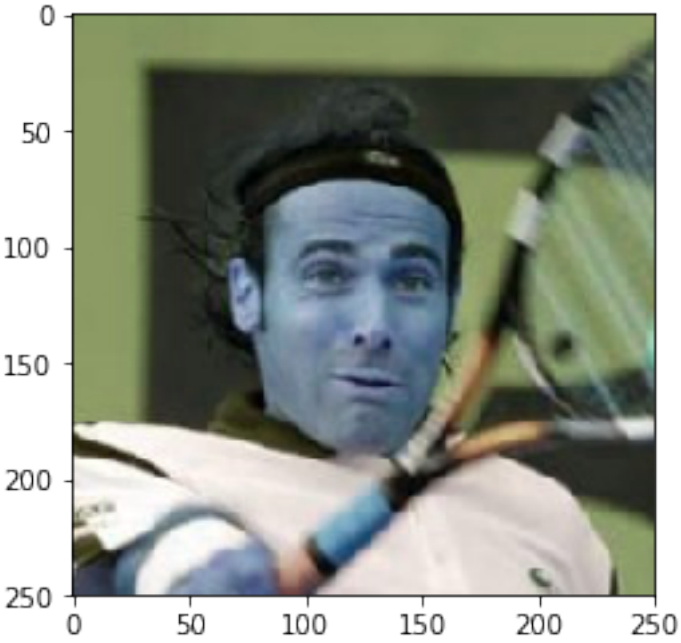
**Answer:** (Three possible points for improvement)

In [47]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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

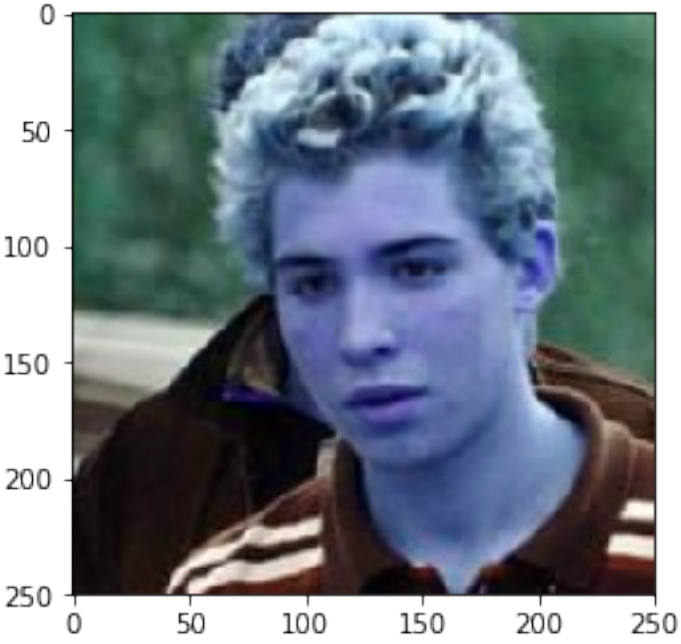
```
The image contains a human!
```



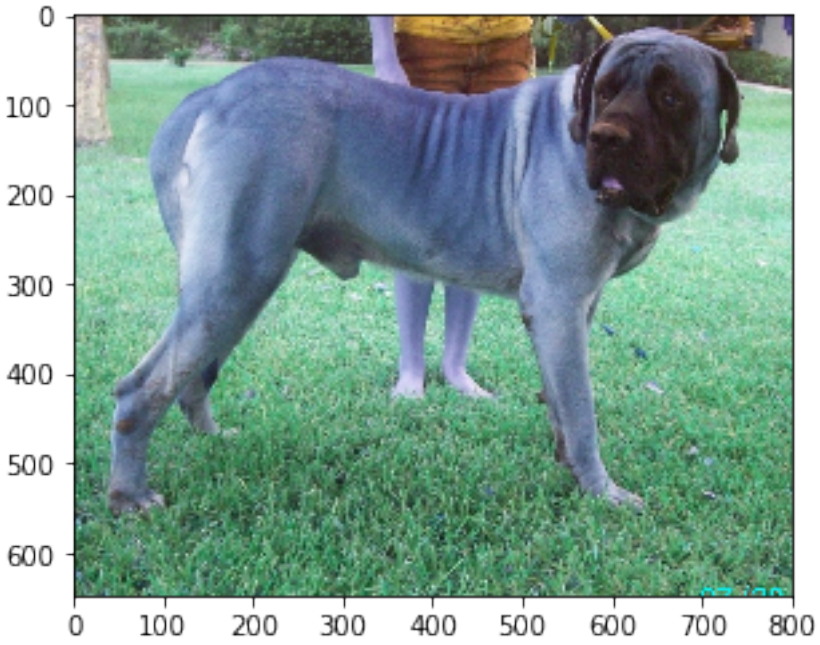

```
This human resembles a French bulldog
The image contains a human!
```



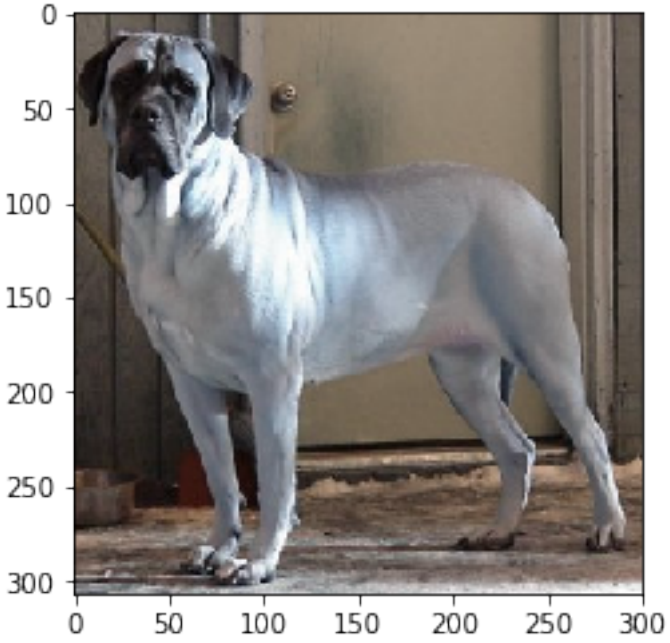

```
This human resembles a Bulldog
The image contains a human!
```



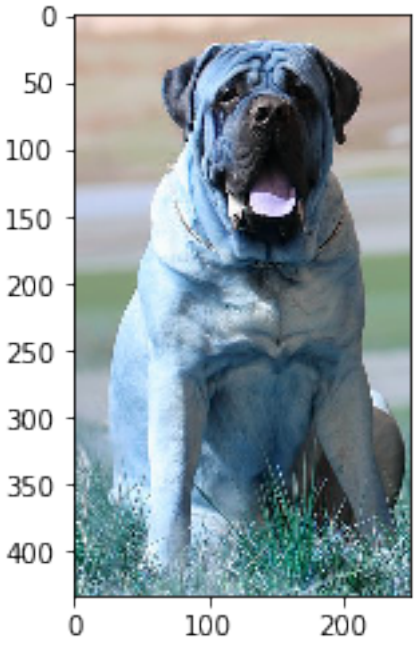
```
This human resembles a Cocker spaniel
The image contains a dog!
```




```
It is a Mastiff
The image contains a dog!
```



```
It is a Mastiff
The image contains a dog!
```




```
It is a Mastiff
```