dog_app

March 23, 2021

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))
```

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

Step 1: Detect Humans

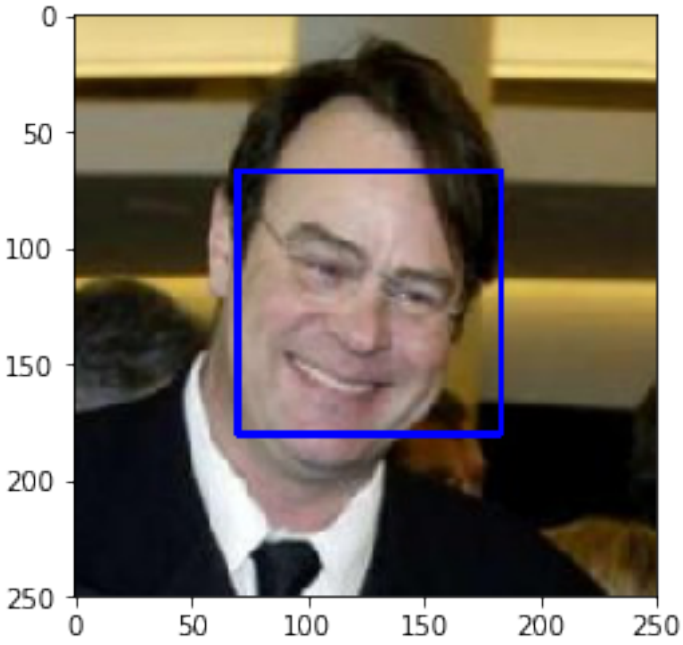
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [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.
        def face_detector_performance(file_paths, file_type):
            file_cnt = len(file_paths)
            face_detection_cnt=0
            for img_path in file_paths:
                if face_detector(img_path):
                    face_detection_cnt+=1
            print(f'{face_detection_cnt} faces detected on {file_cnt} {file_type} files')

        face_detector_performance(human_files_short, 'human')
        face_detector_performance(dog_files_short, 'dog')
```

```
98 faces detected on 100 human files
17 faces detected on 100 dog files
```

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()
```

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

        # Set PIL to be tolerant of image files that are truncated.
```

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    global use_cuda
    img_transforms = transforms.Compose([transforms.CenterCrop(224),
                                         transforms.ToTensor()])
    img = img_transforms(Image.open(img_path))[None , :]
    if use_cuda:
        img = img.cuda()
    VGG16.eval()
    output = VGG16(img)
    _, predictions = torch.max(output, 1)
    if use_cuda:
        predictions = predictions.cpu()
    return predictions.numpy()[0] # predicted class index
```

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.
    predicted_class = VGG16_predict(img_path)
    return predicted_class>=151 and predicted_class<=268 # 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:

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        def dog_detector_performance(file_paths, file_type):
            file_cnt = len(file_paths)
            dog_detection_cnt=0
            for img_path in file_paths:
                if dog_detector(img_path):
                    dog_detection_cnt+=1
            print(f'{dog_detection_cnt} dogs detected on {file_cnt} {file_type} files')

        dog_detector_performance(human_files_short, 'human')
        dog_detector_performance(dog_files_short, 'dog')

0 dogs detected on 100 human files
59 dogs detected on 100 dog files
```

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

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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 [11]: import os
         from torchvision import datasets

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         batch_size = 64
         num_workers=0

         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         val_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')

         train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                                transforms.RandomResizedCrop(224),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485, 0.456, 0.406],
                                                                     [0.229, 0.224, 0.225])])
```

```
        test_transforms = transforms.Compose([transforms.Resize(255),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              transforms.Normalize([0.485, 0.456, 0.406],
                                                                   [0.229, 0.224, 0.225])])

        train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
        val_data = datasets.ImageFolder(val_dir, transform=train_transforms)
        test_data = datasets.ImageFolder(test_dir, transform=test_transforms)

        print('Num training images: ', len(train_data))
        print('Num validation images: ', len(val_data))
        print('Num test images: ', len(test_data))

        train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
        val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,
                                                 num_workers=num_workers, shuffle=True)
        test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                  num_workers=num_workers, shuffle=False)

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

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

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:

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
```

```
            super(Net, self).__init__()
            ## Define layers of a CNN
            self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
            self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
            self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
            self.pool = nn.MaxPool2d(2, 2)
            self.fc1 = nn.Linear(64 * 28 * 28, 512)
            self.fc2 = nn.Linear(512, 133)
            self.dropout = nn.Dropout(p=0.4)

        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.pool(F.relu(self.conv3(x)))
            x = self.dropout(x)
            x = x.view(-1, 64 * 28 * 28)
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = F.log_softmax(self.fc2(x), dim=1)
            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()
```

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

Answer:

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 [13]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.Adam(params=model_scratch.parameters(), lr=0.001)
```

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 [14]:
```
# 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
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output,target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()*data.size(0)

        ######################
        # 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 += loss.item()*data.size(0)
```

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

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

            # return trained model
            return model


        # train the model
        model_scratch = train(100, loaders_scratch, model_scratch, optimizer_scratch,
                              criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 32665.783737         Validation Loss: 4037.933428
Validation loss decreased (inf --> 4037.933428).  Saving model ...
Epoch: 2         Training Loss: 32089.726273         Validation Loss: 3957.459234
Validation loss decreased (4037.933428 --> 3957.459234).  Saving model ...
Epoch: 3         Training Loss: 31555.548836         Validation Loss: 3902.700102
Validation loss decreased (3957.459234 --> 3902.700102).  Saving model ...
Epoch: 4         Training Loss: 30891.651997         Validation Loss: 3802.503546
Validation loss decreased (3902.700102 --> 3802.503546).  Saving model ...
Epoch: 5         Training Loss: 30297.944782         Validation Loss: 3772.458672
Validation loss decreased (3802.503546 --> 3772.458672).  Saving model ...
Epoch: 6         Training Loss: 29848.480602         Validation Loss: 3673.347998
Validation loss decreased (3772.458672 --> 3673.347998).  Saving model ...
Epoch: 7         Training Loss: 29354.959995         Validation Loss: 3624.491729
Validation loss decreased (3673.347998 --> 3624.491729).  Saving model ...
Epoch: 8         Training Loss: 28977.849863         Validation Loss: 3654.774834
Epoch: 9         Training Loss: 28547.366497         Validation Loss: 3571.834757
Validation loss decreased (3624.491729 --> 3571.834757).  Saving model ...
Epoch: 10         Training Loss: 28374.819271         Validation Loss: 3587.990577
Epoch: 11         Training Loss: 28093.393032         Validation Loss: 3521.398690
Validation loss decreased (3571.834757 --> 3521.398690).  Saving model ...
Epoch: 12         Training Loss: 27618.801067         Validation Loss: 3518.567867
```

```
Validation loss decreased (3521.398690 --> 3518.567867).  Saving model ...
Epoch: 13         Training Loss: 27252.258690         Validation Loss: 3473.153795
Validation loss decreased (3518.567867 --> 3473.153795).  Saving model ...
Epoch: 14         Training Loss: 27013.796013         Validation Loss: 3436.287169
Validation loss decreased (3473.153795 --> 3436.287169).  Saving model ...
Epoch: 15         Training Loss: 26730.043320         Validation Loss: 3436.945453
Epoch: 16         Training Loss: 26466.895580         Validation Loss: 3350.259587
Validation loss decreased (3436.287169 --> 3350.259587).  Saving model ...
Epoch: 17         Training Loss: 26213.759516         Validation Loss: 3345.234124
Validation loss decreased (3350.259587 --> 3345.234124).  Saving model ...
Epoch: 18         Training Loss: 26000.501402         Validation Loss: 3367.954536
Epoch: 19         Training Loss: 25805.931822         Validation Loss: 3310.864699
Validation loss decreased (3345.234124 --> 3310.864699).  Saving model ...
Epoch: 20         Training Loss: 25578.754313         Validation Loss: 3280.007258
Validation loss decreased (3310.864699 --> 3280.007258).  Saving model ...
Epoch: 21         Training Loss: 25272.414061         Validation Loss: 3290.825582
Epoch: 22         Training Loss: 25112.457870         Validation Loss: 3243.486123
Validation loss decreased (3280.007258 --> 3243.486123).  Saving model ...
Epoch: 23         Training Loss: 24864.545490         Validation Loss: 3313.352450
Epoch: 24         Training Loss: 24764.712263         Validation Loss: 3220.566557
Validation loss decreased (3243.486123 --> 3220.566557).  Saving model ...
Epoch: 25         Training Loss: 24655.249678         Validation Loss: 3198.291999
Validation loss decreased (3220.566557 --> 3198.291999).  Saving model ...
Epoch: 26         Training Loss: 24380.150955         Validation Loss: 3231.044840
Epoch: 27         Training Loss: 24380.745178         Validation Loss: 3191.010493
Validation loss decreased (3198.291999 --> 3191.010493).  Saving model ...
Epoch: 28         Training Loss: 24126.151073         Validation Loss: 3180.585790
Validation loss decreased (3191.010493 --> 3180.585790).  Saving model ...
Epoch: 29         Training Loss: 23977.849913         Validation Loss: 3127.479195
Validation loss decreased (3180.585790 --> 3127.479195).  Saving model ...
Epoch: 30         Training Loss: 23794.722321         Validation Loss: 3165.239299
Epoch: 31         Training Loss: 23589.558510         Validation Loss: 3155.863513
Epoch: 32         Training Loss: 23583.108032         Validation Loss: 3138.336584
Epoch: 33         Training Loss: 23516.678263         Validation Loss: 3126.913801
Validation loss decreased (3127.479195 --> 3126.913801).  Saving model ...
Epoch: 34         Training Loss: 23398.673006         Validation Loss: 3080.702184
Validation loss decreased (3126.913801 --> 3080.702184).  Saving model ...
Epoch: 35         Training Loss: 23178.363541         Validation Loss: 3125.836486
Epoch: 36         Training Loss: 23142.463160         Validation Loss: 3044.569588
Validation loss decreased (3080.702184 --> 3044.569588).  Saving model ...
Epoch: 37         Training Loss: 23004.568529         Validation Loss: 3145.755581
Epoch: 38         Training Loss: 22929.087694         Validation Loss: 3063.363769
Epoch: 39         Training Loss: 23012.551085         Validation Loss: 3078.228834
Epoch: 40         Training Loss: 22839.377729         Validation Loss: 3183.908964
Epoch: 41         Training Loss: 22658.717493         Validation Loss: 3064.254043
Epoch: 42         Training Loss: 22522.403227         Validation Loss: 3099.272696
Epoch: 43         Training Loss: 22415.490133         Validation Loss: 3077.060434
Epoch: 44         Training Loss: 22347.833025         Validation Loss: 3041.242470
```

```
Validation loss decreased (3044.569588 --> 3041.242470).  Saving model ...
Epoch: 45         Training Loss: 22189.515165         Validation Loss: 3071.109537
Epoch: 46         Training Loss: 22319.362722         Validation Loss: 3033.820153
Validation loss decreased (3041.242470 --> 3033.820153).  Saving model ...
Epoch: 47         Training Loss: 22099.030455         Validation Loss: 3063.670548
Epoch: 48         Training Loss: 22158.322716         Validation Loss: 3072.830530
Epoch: 49         Training Loss: 22100.989305         Validation Loss: 3093.891111
Epoch: 50         Training Loss: 22091.829264         Validation Loss: 3040.878851
Epoch: 51         Training Loss: 21858.424683         Validation Loss: 3075.979146
Epoch: 52         Training Loss: 21684.235123         Validation Loss: 3079.606749
Epoch: 53         Training Loss: 21698.420914         Validation Loss: 2965.841858
Validation loss decreased (3033.820153 --> 2965.841858).  Saving model ...
Epoch: 54         Training Loss: 21723.724800         Validation Loss: 3051.214343
Epoch: 55         Training Loss: 21549.986639         Validation Loss: 3038.621895
Epoch: 56         Training Loss: 21676.935736         Validation Loss: 3050.677788
Epoch: 57         Training Loss: 21639.472746         Validation Loss: 3062.349828
Epoch: 58         Training Loss: 21543.637079         Validation Loss: 3004.441653
Epoch: 59         Training Loss: 21447.679659         Validation Loss: 3004.701239
Epoch: 60         Training Loss: 21437.150011         Validation Loss: 3080.856846
Epoch: 61         Training Loss: 21330.964983         Validation Loss: 3033.418592
Epoch: 62         Training Loss: 21113.080011         Validation Loss: 2978.909515
Epoch: 63         Training Loss: 21144.878290         Validation Loss: 2994.751176
Epoch: 64         Training Loss: 21115.797396         Validation Loss: 2995.054447
Epoch: 65         Training Loss: 20903.090433         Validation Loss: 3021.964839
Epoch: 66         Training Loss: 20790.640650         Validation Loss: 2955.417559
Validation loss decreased (2965.841858 --> 2955.417559).  Saving model ...
Epoch: 67         Training Loss: 21086.158587         Validation Loss: 2938.667968
Validation loss decreased (2955.417559 --> 2938.667968).  Saving model ...
Epoch: 68         Training Loss: 21203.707731         Validation Loss: 3003.142344
Epoch: 69         Training Loss: 20706.605692         Validation Loss: 2972.496949
Epoch: 70         Training Loss: 20868.151945         Validation Loss: 3008.639179
Epoch: 71         Training Loss: 20778.761482         Validation Loss: 2981.310283
Epoch: 72         Training Loss: 20647.031693         Validation Loss: 3004.550169
Epoch: 73         Training Loss: 20755.801233         Validation Loss: 2980.906195
Epoch: 74         Training Loss: 20714.029976         Validation Loss: 3019.542224
Epoch: 75         Training Loss: 20565.967081         Validation Loss: 2958.852547
Epoch: 76         Training Loss: 20572.659163         Validation Loss: 3028.009973
Epoch: 77         Training Loss: 20548.505564         Validation Loss: 2969.037051
Epoch: 78         Training Loss: 20553.591980         Validation Loss: 3027.388981
Epoch: 79         Training Loss: 20360.236969         Validation Loss: 3032.649041
Epoch: 80         Training Loss: 20642.667784         Validation Loss: 2958.595060
Epoch: 81         Training Loss: 20439.923605         Validation Loss: 2996.018471
Epoch: 82         Training Loss: 20680.465082         Validation Loss: 2977.188937
Epoch: 83         Training Loss: 20472.580353         Validation Loss: 2984.049016
Epoch: 84         Training Loss: 20329.540327         Validation Loss: 3054.066691
Epoch: 85         Training Loss: 20133.196009         Validation Loss: 2982.265831
Epoch: 86         Training Loss: 20237.047197         Validation Loss: 2967.265299
Epoch: 87         Training Loss: 20175.491928         Validation Loss: 2905.791218
```

```
Validation loss decreased (2938.667968 --> 2905.791218).  Saving model ...
Epoch: 88         Training Loss: 20155.836905         Validation Loss: 3032.615486
Epoch: 89         Training Loss: 20089.574411         Validation Loss: 3032.834261
Epoch: 90         Training Loss: 20129.983482         Validation Loss: 2996.616970
Epoch: 91         Training Loss: 19955.138084         Validation Loss: 3084.078001
Epoch: 92         Training Loss: 20025.586874         Validation Loss: 3023.987530
Epoch: 93         Training Loss: 19933.159750         Validation Loss: 2953.268861
Epoch: 94         Training Loss: 19781.025873         Validation Loss: 2989.435800
Epoch: 95         Training Loss: 19815.003105         Validation Loss: 3046.475769
Epoch: 96         Training Loss: 19928.712975         Validation Loss: 2962.288950
Epoch: 97         Training Loss: 19992.299942         Validation Loss: 2968.157297
Epoch: 98         Training Loss: 19744.063339         Validation Loss: 2888.367768
Validation loss decreased (2905.791218 --> 2888.367768).  Saving model ...
Epoch: 99         Training Loss: 19755.890188         Validation Loss: 3039.314175
Epoch: 100         Training Loss: 19571.530609         Validation Loss: 3023.433103
```

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 [15]:
```python
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: 3.006184


Test Accuracy: 26% (223/836)
```

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 [16]:
```
## TODO: Specify data loaders
batch_size = 64
num_workers=0

data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
val_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
```

```
        val_data = datasets.ImageFolder(val_dir, transform=train_transforms)
        test_data = datasets.ImageFolder(test_dir, transform=test_transforms)

        print('Num training images: ', len(train_data))
        print('Num validation images: ', len(val_data))
        print('Num test images: ', len(test_data))

        train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
        val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,
                                                 num_workers=num_workers, shuffle=True)
        test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                  num_workers=num_workers, shuffle=False)

        loaders_transfer = {
            'train' : train_loader,
            'valid' : val_loader,
            'test'  : test_loader
        }
```

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

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 [17]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.vgg16_bn(pretrained=True)
         for param in model_transfer.features.parameters():
             param.requires_grad = False

         model_transfer.classifier = nn.Sequential(
                                                 nn.Linear(25088, 512),
                                                 nn.ReLU(),
                                                 nn.Linear(512, 133),
                                                 nn.LogSoftmax(dim=1)
                                                 )

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

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

Answer:

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 [18]: criterion_transfer = nn.NLLLoss()
         optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.001)
```

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 [19]: # train the model
         n_epochs=30
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

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

```
Epoch: 1         Training Loss: 20334.892336         Validation Loss: 1548.240222
Validation loss decreased (inf --> 1548.240222).  Saving model ...
Epoch: 2         Training Loss: 10333.306893         Validation Loss: 1320.631229
Validation loss decreased (1548.240222 --> 1320.631229).  Saving model ...
Epoch: 3         Training Loss: 8755.716475         Validation Loss: 1172.539727
Validation loss decreased (1320.631229 --> 1172.539727).  Saving model ...
Epoch: 4         Training Loss: 7757.450219         Validation Loss: 1171.405518
Validation loss decreased (1172.539727 --> 1171.405518).  Saving model ...
Epoch: 5         Training Loss: 7599.778422         Validation Loss: 1060.975441
Validation loss decreased (1171.405518 --> 1060.975441).  Saving model ...
Epoch: 6         Training Loss: 7151.188336         Validation Loss: 1050.933993
Validation loss decreased (1060.975441 --> 1050.933993).  Saving model ...
Epoch: 7         Training Loss: 6993.750770         Validation Loss: 946.889775
Validation loss decreased (1050.933993 --> 946.889775).  Saving model ...
Epoch: 8         Training Loss: 6715.933586         Validation Loss: 1004.966621
Epoch: 9         Training Loss: 6541.341844         Validation Loss: 1027.831962
Epoch: 10         Training Loss: 6248.899484         Validation Loss: 1055.743507
Epoch: 11         Training Loss: 6275.004569         Validation Loss: 997.029740
Epoch: 12         Training Loss: 6248.199096         Validation Loss: 1066.131750
Epoch: 13         Training Loss: 6081.487301         Validation Loss: 1009.653577
Epoch: 14         Training Loss: 5896.003774         Validation Loss: 1034.638857
Epoch: 15         Training Loss: 5617.956768         Validation Loss: 1001.601515
Epoch: 16         Training Loss: 5703.763981         Validation Loss: 1061.112535
Epoch: 17         Training Loss: 5624.552036         Validation Loss: 1057.808902
Epoch: 18         Training Loss: 5712.915356         Validation Loss: 1125.362635
```

```
Epoch: 19         Training Loss: 5626.562125         Validation Loss: 1095.288045
Epoch: 20         Training Loss: 5592.141645         Validation Loss: 1075.383369
Epoch: 21         Training Loss: 5464.224342         Validation Loss: 1114.737560
Epoch: 22         Training Loss: 5477.952904         Validation Loss: 1077.751617
Epoch: 23         Training Loss: 5354.770203         Validation Loss: 1179.778366
Epoch: 24         Training Loss: 5369.847132         Validation Loss: 1022.311079
Epoch: 25         Training Loss: 5378.731007         Validation Loss: 1168.831202
Epoch: 26         Training Loss: 5353.515193         Validation Loss: 1110.529977
Epoch: 27         Training Loss: 5277.749853         Validation Loss: 1160.979753
Epoch: 28         Training Loss: 5327.770922         Validation Loss: 1128.546820
Epoch: 29         Training Loss: 5360.081553         Validation Loss: 1100.921136
Epoch: 30         Training Loss: 5357.426623         Validation Loss: 1053.116259
```

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%.

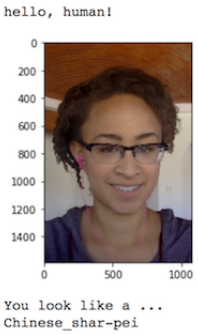
```
In [20]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.636312


Test Accuracy: 80% (675/836)
```

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

         data_transfer = {
             'train' : datasets.ImageFolder(train_dir),
             'valid' : datasets.ImageFolder(val_dir),
             'test'  : datasets.ImageFolder(test_dir)
         }

         # 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 data_transfer['train'].classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             global use_cuda
             img_transforms = transforms.Compose([transforms.CenterCrop(224),
                                                  transforms.ToTensor()])
```



Sample Human Output

```
img = img_transforms(Image.open(img_path))[None , :]
if use_cuda:
    img = img.cuda()
model_transfer.eval()
output = model_transfer(img)
_, predictions = torch.max(output, 1)
if use_cuda:
    predictions = predictions.cpu()
return class_names[predictions.numpy()[0]]
```

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 [22]:
```
### 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)
    if dog_detector(img_path):
        print(f"hello, dog!")
        imgplot = plt.imshow(img)
        plt.show()
        print("You look like a ...")
        print(predict_breed_transfer(img_path))
```

```
    elif face_detector(img_path):
        print(f"hello, human!")
        imgplot = plt.imshow(img)
        plt.show()
        print("You look like a ...")
        print(predict_breed_transfer(img_path))
    else:
        print("An error has occurred, please contact support")
```

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!

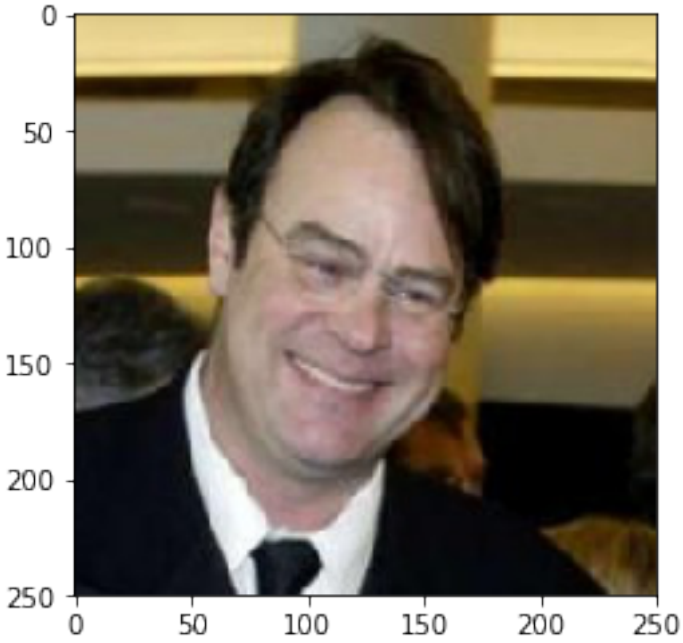
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

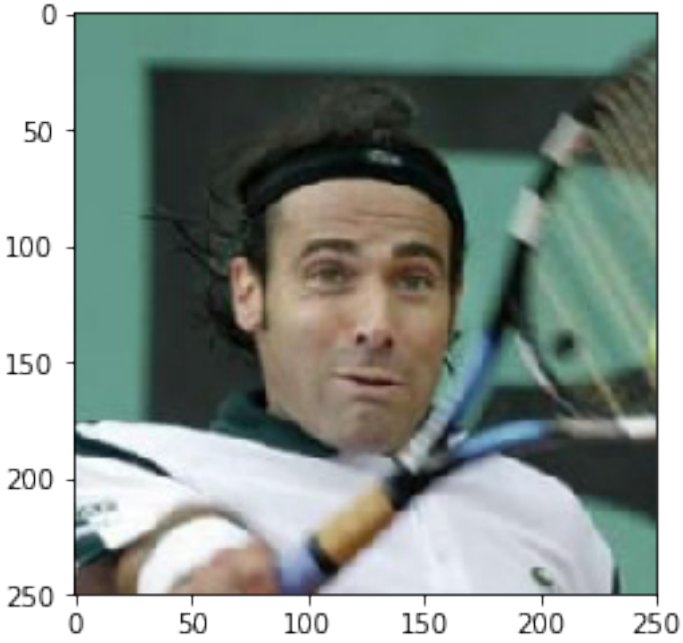
Answer: (Three possible points for improvement)

In [23]:
```
## 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)
```

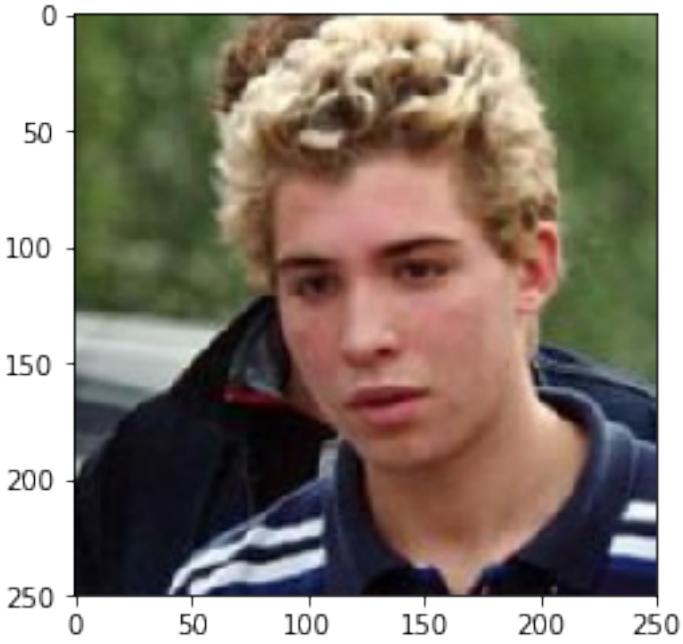
```
hello, human!
```



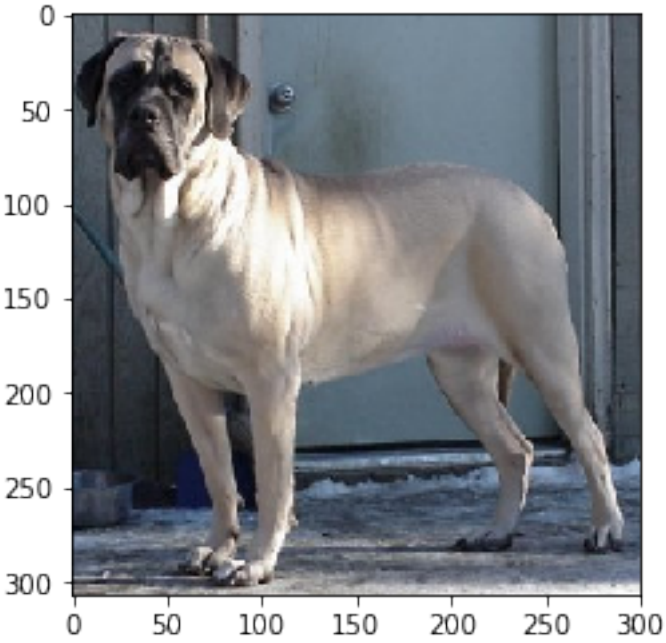

```
You look like a ...
Chinese crested
hello, human!
```



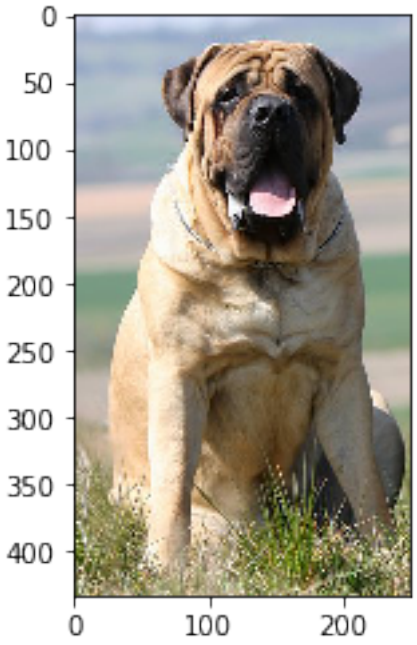
```
You look like a ...
Chinese crested
hello, human!
```




```
You look like a ...
Chinese crested
An error has occurred, please contact support
hello, dog!
```




```
You look like a ...
Bull terrier
hello, dog!
```



```
You look like a ...
Bull terrier
```

In [ ]: