dog_app-Copy1

January 8, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
In [20]: 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

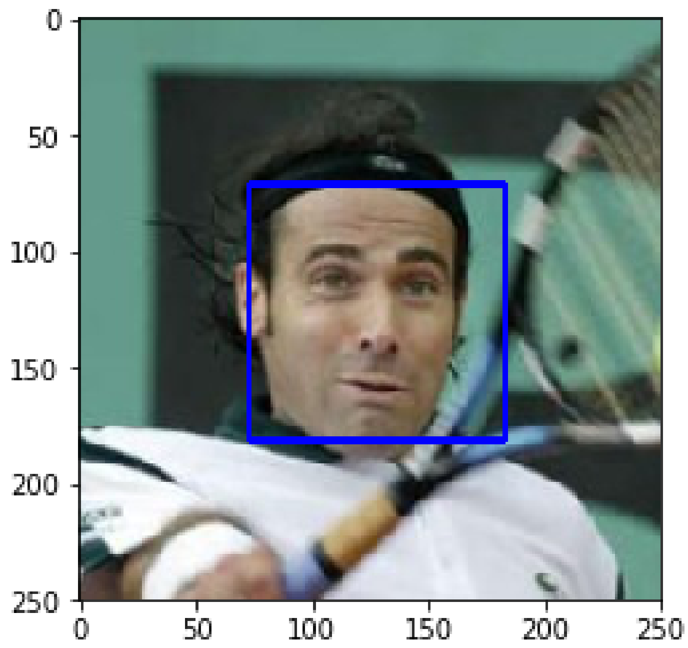
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [25]: 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[1])
         # 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 [22]: # 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. #-#-#
        humans = 0
        dogs = 0
        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.

        for hf in human_files_short:
            if face_detector(hf): humans+=1

        for df in dog_files_short:
            if face_detector(df): dogs+=1

        print('There are %d%% of faces in human images.' % int (100 * humans/len(human_files_sho
        print('There are %d%% of faces in dogs images.' % int (100 * dogs/len(dog_files_short)))

There are 98% of faces in human images.
There are 17% of faces in dogs images.
```

```
In [ ]:
```

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

```
In [ ]:
```

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 [28]: 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 [00:08<00:00, 66369532.17it/s]
```

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

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 [30]:
```
from PIL import Image
import torchvision.transforms as transforms

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

    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    img = Image.open(img_path).convert('RGB')
    transform = transforms.Compose([
     transforms.RandomResizedCrop(224),
     transforms.ToTensor(),
     transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
     ])
    #print(transform(img).shape)
    img = transform(img)[:3,:,:].unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    result = VGG16(img)
    _, preds_tensor = torch.max(result, 1)
    pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso

    return int(pred)
#     return None # predicted class index
VGG16_predict('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jpg') #252
#VGG16_predict('/data/dog_images/train/103.Mastiff/Mastiff_06861.jpg') #243
#VGG16_predict('/data/dog_images/train/103.Mastiff/Mastiff_06829.jpg') #243
#VGG16_predict('/data/dog_images/train/103.Mastiff/Mastiff_06835.jpg') #286
#VGG16_predict('/data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher_04167.jp
#VGG16_predict('/data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher_04191.jp
```

Out[30]: 252

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

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 [13]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         dogsinhumans = 0
         dogsindogs = 0
         for hf in human_files_short:
             if dog_detector(hf): dogsinhumans+=1

         for df in dog_files_short:
             if dog_detector(df):
                 dogsindogs+=1
             else:
                 print(df)
                 plt.imshow(Image.open(df))
                 plt.show()

         print('There are %d dogs in human_files_short images.' % (dogsinhumans))
         print('There are %d dogs in dog_files_short images.' % (dogsindogs))
```

```
/data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher_04195.jpg
```

```
There are 0 dogs in human_files_short images.
There are 99 dogs in dog_files_short images.
```

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 [ ]: ### (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 [2]: import os
        import numpy as np
        import torch
        import torchvision.models as models
        from torchvision import datasets
        import torchvision.transforms as transforms
        from torch.utils.data.sampler import SubsetRandomSampler

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

        # number of subprocesses to use for data loading
        num_workers = 0
        # how many samples per batch to load
        batch_size = 16
```

```
transform = transforms.Compose([transforms.Resize(size=224),
                                transforms.CenterCrop((224,224)),
                                transforms.RandomHorizontalFlip(), # randomly flip and r
                                transforms.RandomRotation(10),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.

data_dir = '/data/dog_images/'

train_data = datasets.ImageFolder(os.path.join(data_dir, 'train'), transform)
valid_data = datasets.ImageFolder(os.path.join(data_dir, 'valid'), transform)
test_data = datasets.ImageFolder(os.path.join(data_dir, 'test'), transform)

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

classNames = train_data.classes

print("Number of classes:", len(classNames))
print("\nClass names: \n\n", classNames)
```

```
Number of classes: 133

Class names:

 ['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mal
```

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:

All images have been resized to 224 because most of the pretrained models require the input to be 224x224 images (224,224,3), the dataset is been augmentated by flipping and rotating randomly.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [19]: # Install a pip package in the current Jupyter kernel
         #import sys
         #!{sys.executable} -m pip install torchsummary
```

```
Collecting torchsummary
  Downloading https://files.pythonhosted.org/packages/7d/18/1474d06f721b86e6a9b9d7392ad68bed711a
Installing collected packages: torchsummary
Successfully installed torchsummary-1.5.1
```

In [7]:
```
import torch.nn as nn
import torch.nn.functional as F
#from torchsummary import summary

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        # convolutional layer (sees 224x224x3 image tensor)
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 28 * 28, 1024)
        self.fc2 = nn.Linear(1024, 1024)
        self.fc3 = nn.Linear(1024, 133)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.dropout(x)
        x = self.pool(F.relu(self.conv2(x)))
        x = self.dropout(x)
        x = self.pool(F.relu(self.conv3(x)))
        x = self.dropout(x)
        x = x.view(-1, 128 * 28 * 28)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

print(model_scratch)
use_cuda = torch.cuda.is_available()

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
    #summary(model_scratch, (3, 224, 224))
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=100352, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=1024, bias=True)
  (fc3): Linear(in_features=1024, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

```
In [ ]:
```

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

Answer:

I have built a CNN inspired in the exercises we have made in the course. Fisrt I had included 3 convolutional layers with 16, 32 and 64 filters and 2 lineal layers but I could not get an accuracy more than 8%.

I finished with 4 convolutional and 3 linear layers increasing the parameters numbers in each layer to grap better the images details.

Every convolutional output is being pooled with a (2,2) factor and using dropout to avoid overfitting.

The last linear layer try to reduce the output to the 133 dog breeds classes.

I have found that the trainning process seems not to be not very efficient. I have tried adding additional layers, changing optimizers like Adam one and learning rates but I did not get a substantial improvement.

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 [8]: 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)
```

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 [9]:
```python
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

   #if os.path.exists(save_path):
   #    model.load_state_dict(torch.load(save_path))

    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_los
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            # update training loss
            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)
```

```python
            # calculate average losses

            train_loss = train_loss/len(loaders['train'].dataset)
            valid_loss = valid_loss/len(loaders['valid'].dataset)

            # 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 ...'.for
                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(16, {'train': train_loader, 'valid': valid_loader}, model_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: 4.882348        Validation Loss: 4.874929
Validation loss decreased (inf --> 4.874929).  Saving model ...
Epoch: 2        Training Loss: 4.813109        Validation Loss: 4.741261
Validation loss decreased (4.874929 --> 4.741261).  Saving model ...
Epoch: 3        Training Loss: 4.640878        Validation Loss: 4.610670
Validation loss decreased (4.741261 --> 4.610670).  Saving model ...
Epoch: 4        Training Loss: 4.511118        Validation Loss: 4.467227
Validation loss decreased (4.610670 --> 4.467227).  Saving model ...
Epoch: 5        Training Loss: 4.358467        Validation Loss: 4.388524
Validation loss decreased (4.467227 --> 4.388524).  Saving model ...
Epoch: 6        Training Loss: 4.262843        Validation Loss: 4.333995
Validation loss decreased (4.388524 --> 4.333995).  Saving model ...
Epoch: 7        Training Loss: 4.192284        Validation Loss: 4.265019
Validation loss decreased (4.333995 --> 4.265019).  Saving model ...
Epoch: 8        Training Loss: 4.117994        Validation Loss: 4.232421
Validation loss decreased (4.265019 --> 4.232421).  Saving model ...
```

```
Epoch: 9        Training Loss: 4.054137        Validation Loss: 4.199164
Validation loss decreased (4.232421 --> 4.199164).  Saving model ...
Epoch: 10       Training Loss: 3.976802        Validation Loss: 4.161303
Validation loss decreased (4.199164 --> 4.161303).  Saving model ...
Epoch: 11       Training Loss: 3.917630        Validation Loss: 4.094585
Validation loss decreased (4.161303 --> 4.094585).  Saving model ...
Epoch: 12       Training Loss: 3.836421        Validation Loss: 4.148582
Epoch: 13       Training Loss: 3.759663        Validation Loss: 4.073022
Validation loss decreased (4.094585 --> 4.073022).  Saving model ...
Epoch: 14       Training Loss: 3.690651        Validation Loss: 4.065976
Validation loss decreased (4.073022 --> 4.065976).  Saving model ...
Epoch: 15       Training Loss: 3.601172        Validation Loss: 4.079483
Epoch: 16       Training Loss: 3.517645        Validation Loss: 4.025002
Validation loss decreased (4.065976 --> 4.025002).  Saving model ...
```

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 [10]: 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({'test': test_loader}, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 4.030829


Test Accuracy: 11% (96/836)
```

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 [11]: ## TODO: Specify data loaders
         import os
         from torchvision import datasets
         import torchvision.transforms as transforms

         transform = transforms.Compose([transforms.Resize(size=224),
                                         transforms.CenterCrop((224,224)),
                                         transforms.RandomHorizontalFlip(), # randomly flip and
                                         transforms.RandomRotation(10),
                                         transforms.ToTensor(),
                                         transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0

         data_dir = '/data/dog_images/'

         train_data = datasets.ImageFolder(os.path.join(data_dir, 'train'), transform)
         valid_data = datasets.ImageFolder(os.path.join(data_dir, 'valid'), transform)
         test_data = datasets.ImageFolder(os.path.join(data_dir, 'test'), transform)

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

         classNames = train_data.classes
```

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

         ## TODO: Specify model architecture
         model_transfer = models.vgg19(pretrained=True)
         if use_cuda:
             model_transfer = model_transfer.cuda()

         model_transfer
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.torch/models/vgg
100%|| 574673361/574673361 [00:31<00:00, 18055062.48it/s]
```

```
Out[12]: 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): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (17): ReLU(inplace)
             (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (19): Conv2d(256, 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): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (24): ReLU(inplace)
             (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (26): ReLU(inplace)
             (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (31): ReLU(inplace)
        (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (33): ReLU(inplace)
        (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (35): ReLU(inplace)
        (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
      (classifier): Sequential(
        (0): Linear(in_features=25088, out_features=4096, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.5)
        (3): Linear(in_features=4096, out_features=4096, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.5)
        (6): Linear(in_features=4096, out_features=1000, bias=True)
      )
    )
```

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

Answer:

- I have choosen to use the vgg19 model to apply for this problem, because it has been trained on more than a million images from the ImageNet database.
- I have only changed the out feature of the lasts linears layers and re-training the whole classifier section of the network.
- It seems to have a good performance.

```
In [13]: for param in model_transfer.parameters():
             param.requires_grad = False


         fc1 = nn.Linear(4096, 1024,bias= True)
         fc2 = nn.Linear(1024, 133,bias= True)
         model_transfer.classifier[3] = fc1
         model_transfer.classifier[6] = fc2

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

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 [14]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = torch.optim.SGD(filter(lambda p: p.requires_grad,model_transfer.pa
```

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 [15]:
```
# train the model
model_transfer = train(16, {'train': train_loader, 'valid': valid_loader}, model_transf

# 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: 4.554513         Validation Loss: 3.817998
Validation loss decreased (inf --> 3.817998).  Saving model ...
Epoch: 2         Training Loss: 3.481737         Validation Loss: 2.676933
Validation loss decreased (3.817998 --> 2.676933).  Saving model ...
Epoch: 3         Training Loss: 2.527849         Validation Loss: 1.865136
Validation loss decreased (2.676933 --> 1.865136).  Saving model ...
Epoch: 4         Training Loss: 1.911984         Validation Loss: 1.401593
Validation loss decreased (1.865136 --> 1.401593).  Saving model ...
Epoch: 5         Training Loss: 1.548569         Validation Loss: 1.138372
Validation loss decreased (1.401593 --> 1.138372).  Saving model ...
Epoch: 6         Training Loss: 1.300849         Validation Loss: 0.976759
Validation loss decreased (1.138372 --> 0.976759).  Saving model ...
Epoch: 7         Training Loss: 1.147129         Validation Loss: 0.881237
Validation loss decreased (0.976759 --> 0.881237).  Saving model ...
Epoch: 8         Training Loss: 1.033445         Validation Loss: 0.786961
Validation loss decreased (0.881237 --> 0.786961).  Saving model ...
Epoch: 9         Training Loss: 0.930085         Validation Loss: 0.720966
Validation loss decreased (0.786961 --> 0.720966).  Saving model ...
Epoch: 10         Training Loss: 0.869556         Validation Loss: 0.669597
Validation loss decreased (0.720966 --> 0.669597).  Saving model ...
Epoch: 11         Training Loss: 0.817162         Validation Loss: 0.649684
Validation loss decreased (0.669597 --> 0.649684).  Saving model ...
Epoch: 12         Training Loss: 0.774572         Validation Loss: 0.599896
Validation loss decreased (0.649684 --> 0.599896).  Saving model ...
Epoch: 13         Training Loss: 0.750737         Validation Loss: 0.588656
Validation loss decreased (0.599896 --> 0.588656).  Saving model ...
Epoch: 14         Training Loss: 0.698400         Validation Loss: 0.567796
Validation loss decreased (0.588656 --> 0.567796).  Saving model ...
Epoch: 15         Training Loss: 0.677543         Validation Loss: 0.550209
Validation loss decreased (0.567796 --> 0.550209).  Saving model ...
Epoch: 16         Training Loss: 0.632281         Validation Loss: 0.528468
Validation loss decreased (0.550209 --> 0.528468).  Saving model ...
```

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%.

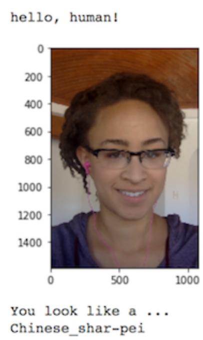
```
In [16]: test({'test': test_loader}, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.553483


Test Accuracy: 85% (717/836)
```

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

         from PIL import Image
         import torchvision.transforms as transforms

         model_transfer.load_state_dict(torch.load('model_transfer.pt'))

         # 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 train_data.classes]

         def predict_breed_transfer(img_path):

             img = Image.open(img_path).convert('RGB')
             #plt.imshow(img)
             #plt.show()
             transform = transforms.Compose([
              transforms.RandomResizedCrop(224),
              transforms.ToTensor(),
              transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
              ])
             #print(transform(img).shape)
             img = transform(img)[:3,:,:].unsqueeze(0)
             #print(img.shape)
             img = img.cuda()
             pred = model_transfer(img)
             a,b = torch.max(pred,1)
             b = np.squeeze(b.cpu().numpy())
             #print(class_names[b])
             return class_names[b]

         predict_breed_transfer('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jp

         #predict_breed_transfer('/data/dog_images/train/103.Mastiff/Mastiff_06861.jpg') #243
         #predict_breed_transfer('/data/dog_images/train/103.Mastiff/Mastiff_06829.jpg') #243
```



Sample Human Output

```
#predict_breed_transfer('/data/dog_images/train/103.Mastiff/Mastiff_06835.jpg') #286
#predict_breed_transfer('/data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher
#predict_breed_transfer('/data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher
```

Out[17]: 'Affenpinscher'

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 [34]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    image = Image.open(img_path)
    if(face_detector(img_path)):
        breed = predict_breed_transfer(img_path)
        plt.imshow(image)
        plt.show()
        print(f"you are a human than look like a: {breed}")

    elif(dog_detector(img_path)):
        breed = predict_breed_transfer(img_path)
        plt.imshow(image)
        plt.show()
```

```python
            print(f"this is a {breed}")

        else:
            print('nor a human or a dog')
            plt.imshow(image)
            plt.show()
```

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!

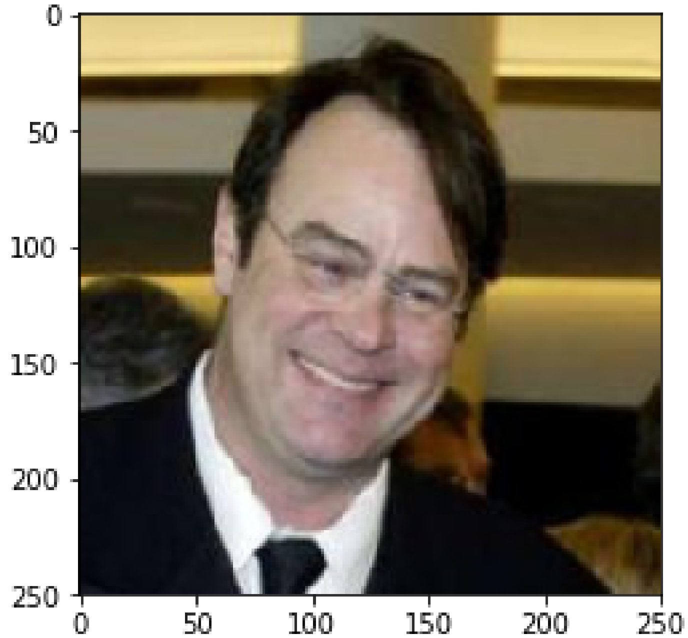
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

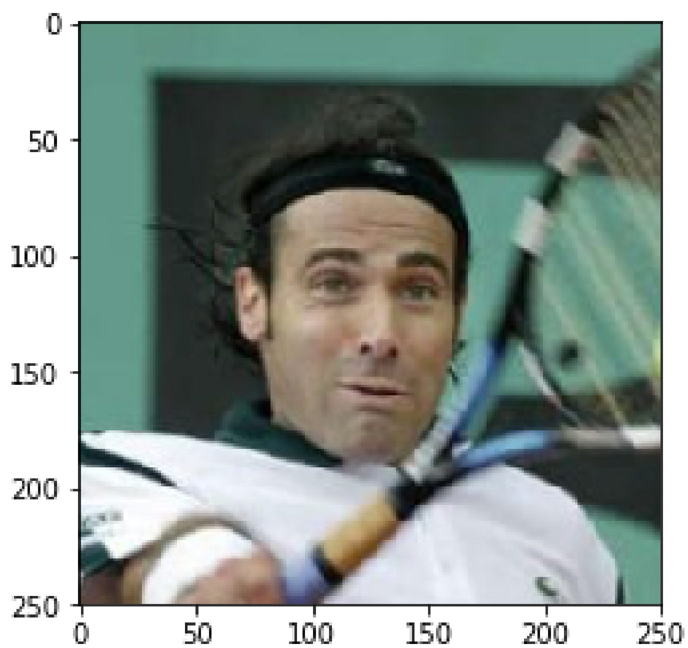
Answer: (Three possible points for improvement)

- Improve accuracy benchmarking different options like optimizers, data transformation and different models
- Show others breed predictions probabilities.
- Deploy the model as an API with Flask hosting at AWS.

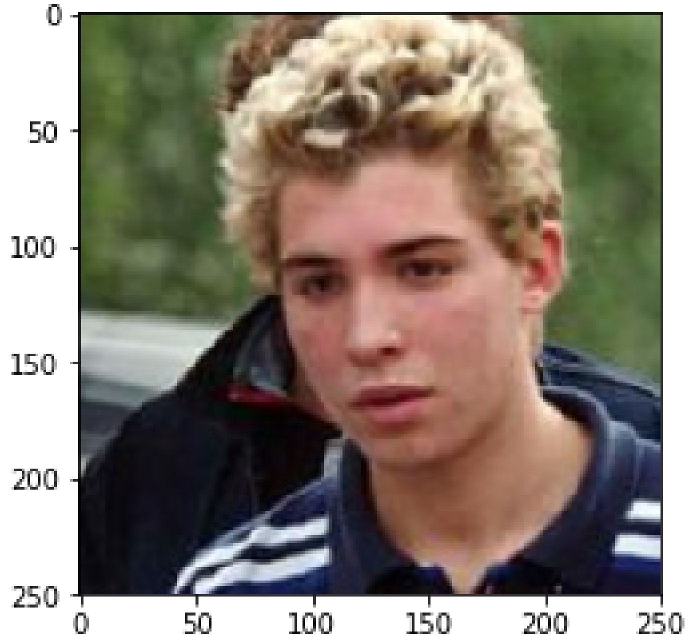
In [35]:
```python
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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



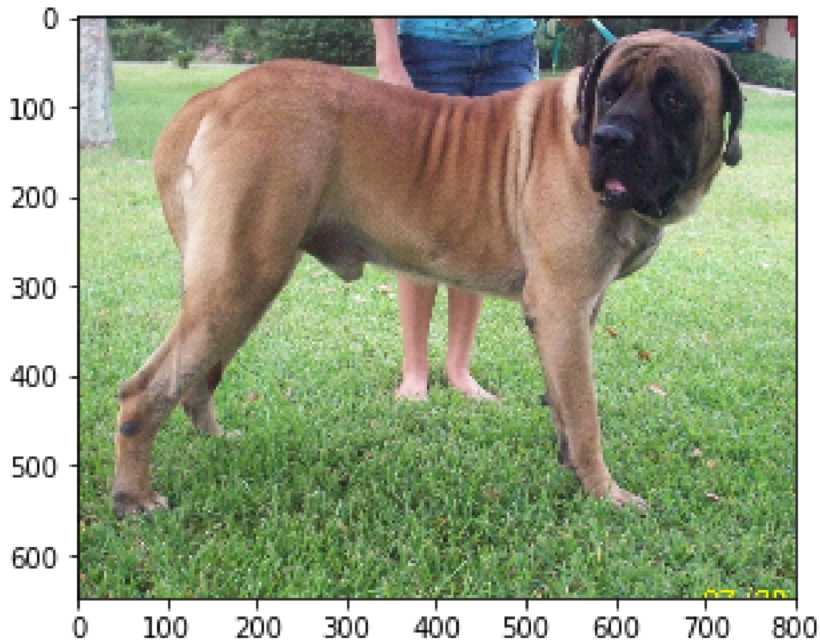
you are a human than look like a: Xoloitzcuintli



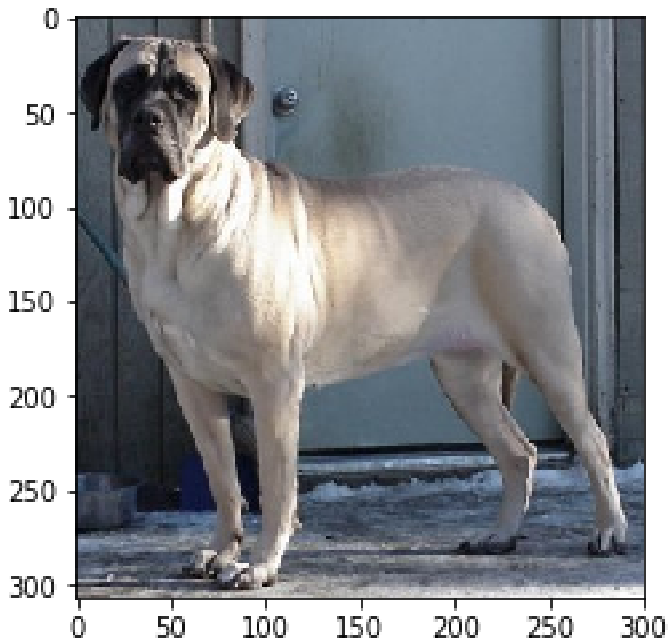
```
you are a human than look like a: Chinese crested
```



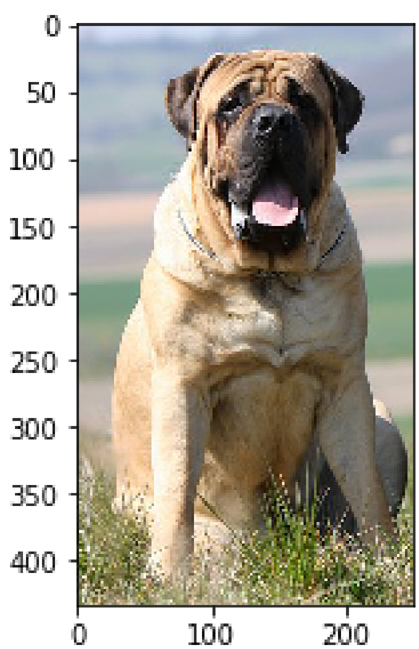

```
you are a human than look like a: Chesapeake bay retriever
```



this is a Chinese shar-pei



this is a Mastiff



```
this is a Mastiff
```