dog_app

May 13, 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(type(dog_files))
        print(dog_files.shape)
        print(dog_files[:5])

        print(dog_files[0].split('/'))

        # 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))
```

```
<class 'numpy.ndarray'>
(8351,)
['/data/dog_images/train/103.Mastiff/Mastiff_06833.jpg'
 '/data/dog_images/train/103.Mastiff/Mastiff_06826.jpg'
 '/data/dog_images/train/103.Mastiff/Mastiff_06871.jpg'
 '/data/dog_images/train/103.Mastiff/Mastiff_06812.jpg'
 '/data/dog_images/train/103.Mastiff/Mastiff_06831.jpg']
['', 'data', 'dog_images', 'train', '103.Mastiff', 'Mastiff_06833.jpg']
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

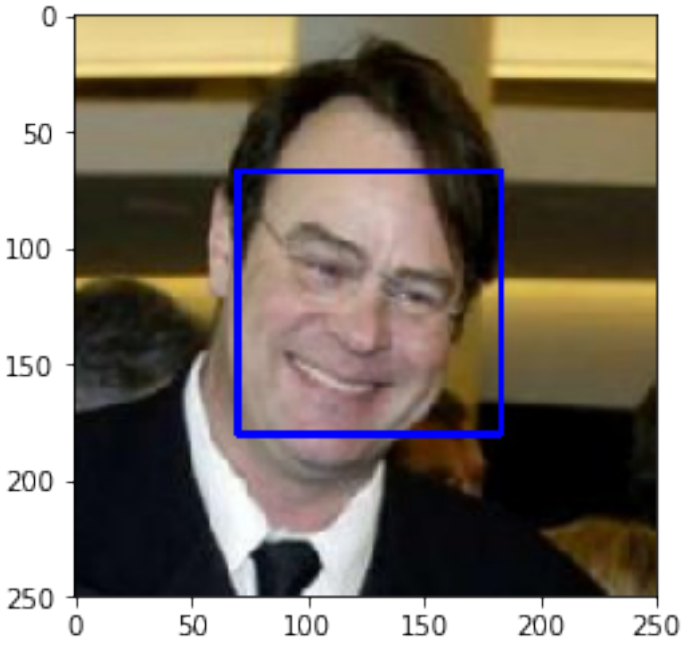
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [4]: # 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)

Percentage of detected human face from human files: 0.98

Percentage of detected human face from dog files : 0.17

```
In [5]: 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.
        hf_from_human_files = 0
        hf_from_dog_files = 0
```

```
        for i in range(100):
            if face_detector(human_files_short[i]):
                hf_from_human_files += 1
            if face_detector(dog_files_short[i]):
                hf_from_dog_files += 1

        print("Percentage of detected human face from human files: "
              + str(hf_from_human_files/100.0))
        print("Percentage of detected human face from dog files  : "
              + str(hf_from_dog_files/100.0))
```

```
Percentage of detected human face from human files: 0.98
Percentage of detected human face from dog files  : 0.17
```

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 [ ]: ### (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()
        print(use_cuda)

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:13<00:00, 40737395.20it/s]
```

```
True
```

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 [8]:
```
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
    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std =[0.229, 0.224, 0.225])])
    img = Image.open(img_path)
    img_t = transform(img)
    # has to add the batch dim
    img_t = img_t.view(1, img_t.shape[0], img_t.shape[1], img_t.shape[2])
    # Ligang: necessary, or 'RuntimeError: Expected object of type torch.FloatTensor
    # but found type torch.cuda.FloatTensor for argument #2 weight'
    img_t = img_t.to('cuda')
```

```
    VGG16.eval()
    output = VGG16(img_t)
    value, idx = torch.max(output, 1)

    return idx # 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 [9]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            predicted_idx = VGG16_predict(img_path)
            if 151<=predicted_idx and predicted_idx <=268:
                return True
            else:
                return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: Percentage of detected dogs from human files: 0.0
Percentage of detected dogs from dog files : 1.0

```
In [10]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         dog_from_human_files = 0
         dog_from_dog_files = 0
         for i in range(100):
             if dog_detector(human_files_short[i]):
                 dog_from_human_files += 1
             if dog_detector(dog_files_short[i]):
                 dog_from_dog_files += 1

         print("Percentage of detected dogs from human files: "
               + str(dog_from_human_files/100.0))
         print("Percentage of detected dogs from dog files  : "
               + str(dog_from_dog_files/100.0))
```

```
Percentage of detected dogs from human files: 0.0
Percentage of detected dogs from dog files  : 1.0
```

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

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

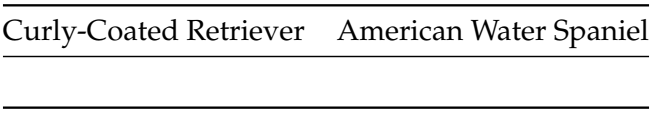
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.

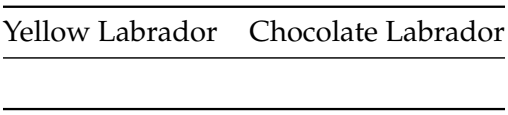
Brittany Welsh Springer Spaniel

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
         import glob
         from PIL import Image
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         import torchvision.transforms as transforms
         import torch

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         num_workers = 0
         batch_size = 20
         transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                     transforms.RandomRotation(10),
                     transforms.Resize((224, 224)), transforms.ToTensor(),
                     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                          std =[0.229, 0.224, 0.225])])

         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         valid_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')
         train_data = datasets.ImageFolder(train_dir, transform=transform)
         valid_data = datasets.ImageFolder(valid_dir, transform=transform)
         test_data  = datasets.ImageFolder(test_dir , transform=transform)

         print('Num train images: ', len(train_data))
         print('Num valid images: ', len(valid_data))
         print('Num test  images: ', len(test_data ))

         # train_load, valid_loader, test_loader
         loaders_scratch = {}
         loaders_scratch['train'] = torch.utils.data.DataLoader(train_data,
             batch_size=batch_size, num_workers=num_workers, shuffle=True)
         loaders_scratch['valid'] = torch.utils.data.DataLoader(valid_data,
             batch_size=batch_size, num_workers=num_workers, shuffle=True)
         loaders_scratch['test']  = torch.utils.data.DataLoader(test_data ,
             batch_size=batch_size, num_workers=num_workers, shuffle=True)
```

```
        print(type(loaders_scratch['train']))
        print(loaders_scratch['train'])
```

```
Num train images:  6680
Num valid images:  835
Num test  images:  836
<class 'torch.utils.data.dataloader.DataLoader'>
<torch.utils.data.dataloader.DataLoader object at 0x7f65d02f8c18>
```

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer: I resized all the images to 224x224 by stretching. This size was chosen is because most of the dog picturs are about this size, and it's what the VGG model expects. It will allow us to extract much features as needed. I also augmented the dataset through transforms.RandomHorizontalFlip() and transforms.RandomRotation(10), which seems to help improve the test accuracy.

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
                 # input: 224x224x3
                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                 # input: 112x112x16
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                 # input: 56x56x32
                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                 # input : 28x28x64
                 self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
                 # input : 14x14x128; output: 7x7x256;
                 self.conv5 = nn.Conv2d(128, 256, 3, padding=1)

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

                 self.fc1 = nn.Linear(7*7*256, 6000)
                 self.fc2 = nn.Linear(6000, 1000)
                 self.fc3 = nn.Linear(1000, 133)
```

```
            self.dropout = nn.Dropout(0.25)

        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.pool(F.relu(self.conv4(x)))
            x = self.pool(F.relu(self.conv5(x)))
            # flatten image input
            x = x.view(-1, 256*7*7)
            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 = F.relu(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)

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

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(128, 256, 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=12544, out_features=6000, bias=True)
  (fc2): Linear(in_features=6000, out_features=1000, bias=True)
  (fc3): Linear(in_features=1000, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

Answer: I want to have relatively large depth so I defined 5 convolutional layers followed by the MaxPooling layer to reduce the x-y size. At last I would like to define 3 fully-connected

layers instead of 2, to give me more trainable parameters. I added the dropout after each of the fully-connected layer except the last one, to prevent the model from overfitting.

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.CrossEntropyLoss()

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

```
In [17]: 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):
                 print('In train(), epoch = ', epoch)
                 # 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']):
                     # print("In loaders['train'], batch_idx = ", batch_idx)
                     # print(data.shape)
                     # print(target.shape)

                     # 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_loss))
                     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)

        train_loss = train_loss/len(train_data)
        valid_loss = valid_loss/len(valid_data)

        # 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 ...'.format(valid_loss_min, valid_loss))
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model
```

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 [ ]: # train the model, initially 100,
        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'))
```

```
In train(), epoch =  1
Epoch: 1         Training Loss: 4.890002         Validation Loss: 4.889220
Validation loss decreased (inf --> 4.889220). Saving model ...
In train(), epoch =  2
Epoch: 2         Training Loss: 4.888827         Validation Loss: 4.888337
Validation loss decreased (4.889220 --> 4.888337). Saving model ...
In train(), epoch =  3
Epoch: 3         Training Loss: 4.887986         Validation Loss: 4.887459
Validation loss decreased (4.888337 --> 4.887459). Saving model ...
In train(), epoch =  4
Epoch: 4         Training Loss: 4.886916         Validation Loss: 4.886391
Validation loss decreased (4.887459 --> 4.886391). Saving model ...
In train(), epoch =  5
Epoch: 5         Training Loss: 4.885547         Validation Loss: 4.884685
Validation loss decreased (4.886391 --> 4.884685). Saving model ...
In train(), epoch =  6
Epoch: 6         Training Loss: 4.883195         Validation Loss: 4.881447
Validation loss decreased (4.884685 --> 4.881447). Saving model ...
In train(), epoch =  7
Epoch: 7         Training Loss: 4.878226         Validation Loss: 4.875055
Validation loss decreased (4.881447 --> 4.875055). Saving model ...
In train(), epoch =  8
Epoch: 8         Training Loss: 4.873455         Validation Loss: 4.869237
Validation loss decreased (4.875055 --> 4.869237). Saving model ...
In train(), epoch =  9
Epoch: 9         Training Loss: 4.865153         Validation Loss: 4.850404
Validation loss decreased (4.869237 --> 4.850404). Saving model ...
In train(), epoch =  10
Epoch: 10         Training Loss: 4.827634         Validation Loss: 4.809048
Validation loss decreased (4.850404 --> 4.809048). Saving model ...
In train(), epoch =  11
Epoch: 11         Training Loss: 4.777888         Validation Loss: 4.773305
Validation loss decreased (4.809048 --> 4.773305). Saving model ...
In train(), epoch =  12
Epoch: 12         Training Loss: 4.736513         Validation Loss: 4.732460
Validation loss decreased (4.773305 --> 4.732460). Saving model ...
In train(), epoch =  13
Epoch: 13         Training Loss: 4.678268         Validation Loss: 4.684638
Validation loss decreased (4.732460 --> 4.684638). Saving model ...
In train(), epoch =  14
Epoch: 14         Training Loss: 4.630718         Validation Loss: 4.646945
```

```
Validation loss decreased (4.684638 --> 4.646945). Saving model ...
In train(), epoch =  15
Epoch: 15         Training Loss: 4.589535         Validation Loss: 4.602025
Validation loss decreased (4.646945 --> 4.602025). Saving model ...
In train(), epoch =  16
Epoch: 16         Training Loss: 4.538712         Validation Loss: 4.549599
Validation loss decreased (4.602025 --> 4.549599). Saving model ...
In train(), epoch =  17
Epoch: 17         Training Loss: 4.485884         Validation Loss: 4.536003
Validation loss decreased (4.549599 --> 4.536003). Saving model ...
In train(), epoch =  18
Epoch: 18         Training Loss: 4.434803         Validation Loss: 4.480768
Validation loss decreased (4.536003 --> 4.480768). Saving model ...
In train(), epoch =  19
Epoch: 19         Training Loss: 4.397144         Validation Loss: 4.483602
In train(), epoch =  20
Epoch: 20         Training Loss: 4.345626         Validation Loss: 4.430904
Validation loss decreased (4.480768 --> 4.430904). Saving model ...
In train(), epoch =  21
Epoch: 21         Training Loss: 4.301154         Validation Loss: 4.390102
Validation loss decreased (4.430904 --> 4.390102). Saving model ...
In train(), epoch =  22
Epoch: 22         Training Loss: 4.257663         Validation Loss: 4.355863
Validation loss decreased (4.390102 --> 4.355863). Saving model ...
In train(), epoch =  23
Epoch: 23         Training Loss: 4.189283         Validation Loss: 4.351224
Validation loss decreased (4.355863 --> 4.351224). Saving model ...
In train(), epoch =  24
Epoch: 24         Training Loss: 4.117589         Validation Loss: 4.294880
Validation loss decreased (4.351224 --> 4.294880). Saving model ...
In train(), epoch =  25
Epoch: 25         Training Loss: 4.050408         Validation Loss: 4.262041
Validation loss decreased (4.294880 --> 4.262041). Saving model ...
In train(), epoch =  26
Epoch: 26         Training Loss: 3.965056         Validation Loss: 4.202362
Validation loss decreased (4.262041 --> 4.202362). Saving model ...
In train(), epoch =  27
Epoch: 27         Training Loss: 3.884754         Validation Loss: 4.154329
Validation loss decreased (4.202362 --> 4.154329). Saving model ...
In train(), epoch =  28
Epoch: 28         Training Loss: 3.783228         Validation Loss: 4.158983
In train(), epoch =  29
Epoch: 29         Training Loss: 3.689750         Validation Loss: 4.148835
Validation loss decreased (4.154329 --> 4.148835). Saving model ...
In train(), epoch =  30
Epoch: 30         Training Loss: 3.594836         Validation Loss: 4.063939
Validation loss decreased (4.148835 --> 4.063939). Saving model ...
In train(), epoch =  31
```

```
Epoch: 31         Training Loss: 3.475420         Validation Loss: 4.104453
In train(), epoch =  32
Epoch: 32         Training Loss: 3.348982         Validation Loss: 4.148038
In train(), epoch =  33
Epoch: 33         Training Loss: 3.225604         Validation Loss: 4.162480
In train(), epoch =  34
Epoch: 34         Training Loss: 3.068622         Validation Loss: 4.182389
In train(), epoch =  35
Epoch: 35         Training Loss: 2.893786         Validation Loss: 4.152201
In train(), epoch =  36
Epoch: 36         Training Loss: 2.695937         Validation Loss: 4.438045
In train(), epoch =  37
Epoch: 37         Training Loss: 2.520827         Validation Loss: 4.323316
In train(), epoch =  38
Epoch: 38         Training Loss: 2.310770         Validation Loss: 4.445394
In train(), epoch =  39
Epoch: 39         Training Loss: 2.106821         Validation Loss: 4.560288
In train(), epoch =  40
Epoch: 40         Training Loss: 1.864195         Validation Loss: 4.945055
In train(), epoch =  41
```

In [ ]:

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 [19]:
```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
```

```
            pred = output.data.max(1, keepdim=True)[1]
            # compare predictions to true label
            correct += np.sum(np.squeeze(pred.eq(
                target.data.view_as(pred))).cpu().numpy())
            total += data.size(0)

        print('Test Loss: {:.6f}\n'.format(test_loss))

        print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
            100. * correct / total, correct, total))
```

```
In [23]: model_scratch.load_state_dict(torch.load('model_scratch.pt'))
         # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 4.058991


Test Accuracy: 10% (88/836)
```

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 [24]: ## TODO: Specify data loaders
         import os
         from torchvision import datasets
         import glob
         from PIL import Image
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         import torchvision.transforms as transforms
         import torch

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         num_workers = 0
         batch_size = 20
```

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

data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')
train_data = datasets.ImageFolder(train_dir, transform=transform)
valid_data = datasets.ImageFolder(valid_dir, transform=transform)
test_data  = datasets.ImageFolder(test_dir , transform=transform)

print('Num train images: ', len(train_data))
print('Num valid images: ', len(valid_data))
print('Num test  images: ', len(test_data ))

# added only for task "Predict Dog Breed with the Model"
data_transfer = {}
data_transfer['train'] = train_data
data_transfer['valid'] = valid_data
data_transfer['test']  = test_data

# train_load, valid_loader, test_loader
loaders_transfer = {}
loaders_transfer['train'] = torch.utils.data.DataLoader(train_data,
    batch_size=batch_size, num_workers=num_workers, shuffle=True)
loaders_transfer['valid'] = torch.utils.data.DataLoader(valid_data,
    batch_size=batch_size, num_workers=num_workers, shuffle=True)
loaders_transfer['test']  = torch.utils.data.DataLoader(test_data ,
    batch_size=batch_size, num_workers=num_workers, shuffle=True)

print(type(loaders_transfer['train']))
print(loaders_transfer['train'])
```

```
Num train images:  6680
Num valid images:  835
Num test  images:  836
<class 'torch.utils.data.dataloader.DataLoader'>
<torch.utils.data.dataloader.DataLoader object at 0x7f65c8b0ea58>
```

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 [25]: import torchvision.models as models
         import torch.nn as nn
```

```
        ## TODO: Specify model architecture
        # define VGG16 model
        # VGG16 = models.vgg16(pretrained=True)
        model_transfer = models.vgg16(pretrained=True)

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

        # replace last layer
        n_inputs = model_transfer.classifier[6].in_features

        last_layer = nn.Linear(n_inputs, 133)
        model_transfer.classifier[6] = last_layer
        print(model_transfer.classifier[6].out_features)

        # check if CUDA is available
        use_cuda = torch.cuda.is_available()
        # move model to GPU if CUDA is available
        if use_cuda:
            model_transfer = model_transfer.cuda()
        print(model_transfer)
```

```
133
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

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

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

Answer: I have chosen the vgg16 model which is because the dog breed pictures here is smilar to those in imageNet, with smaller size. I only need to replace the final classifier Linear layer with out_features of 133, which is the number of dog breeds, and only train for the parameters for classifier. The test accuracy of 87% is pretty impressive.

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 [26]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(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 [ ]: # train the model
        model_transfer = train(100, loaders_transfer, model_transfer,
            optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
```

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

```
In train(), epoch =  1
Epoch: 1         Training Loss: 3.799182         Validation Loss: 2.217105
Validation loss decreased (inf --> 2.217105). Saving model ...
In train(), epoch =  2
Epoch: 2         Training Loss: 1.634577         Validation Loss: 0.944097
Validation loss decreased (2.217105 --> 0.944097). Saving model ...
In train(), epoch =  3
Epoch: 3         Training Loss: 0.964395         Validation Loss: 0.666475
Validation loss decreased (0.944097 --> 0.666475). Saving model ...
In train(), epoch =  4
Epoch: 4         Training Loss: 0.737628         Validation Loss: 0.567165
Validation loss decreased (0.666475 --> 0.567165). Saving model ...
In train(), epoch =  5
Epoch: 5         Training Loss: 0.619807         Validation Loss: 0.512390
Validation loss decreased (0.567165 --> 0.512390). Saving model ...
In train(), epoch =  6
Epoch: 6         Training Loss: 0.538072         Validation Loss: 0.485841
Validation loss decreased (0.512390 --> 0.485841). Saving model ...
In train(), epoch =  7
Epoch: 7         Training Loss: 0.493562         Validation Loss: 0.457464
Validation loss decreased (0.485841 --> 0.457464). Saving model ...
In train(), epoch =  8
Epoch: 8         Training Loss: 0.445005         Validation Loss: 0.440538
Validation loss decreased (0.457464 --> 0.440538). Saving model ...
In train(), epoch =  9
Epoch: 9         Training Loss: 0.402290         Validation Loss: 0.426545
Validation loss decreased (0.440538 --> 0.426545). Saving model ...
In train(), epoch =  10
Epoch: 10         Training Loss: 0.370772         Validation Loss: 0.422863
Validation loss decreased (0.426545 --> 0.422863). Saving model ...
In train(), epoch =  11
Epoch: 11         Training Loss: 0.339550         Validation Loss: 0.411922
Validation loss decreased (0.422863 --> 0.411922). Saving model ...
In train(), epoch =  12
Epoch: 12         Training Loss: 0.323670         Validation Loss: 0.407340
Validation loss decreased (0.411922 --> 0.407340). Saving model ...
In train(), epoch =  13
Epoch: 13         Training Loss: 0.297414         Validation Loss: 0.397381
Validation loss decreased (0.407340 --> 0.397381). Saving model ...
In train(), epoch =  14
Epoch: 14         Training Loss: 0.285797         Validation Loss: 0.400193
In train(), epoch =  15
Epoch: 15         Training Loss: 0.263606         Validation Loss: 0.389932
Validation loss decreased (0.397381 --> 0.389932). Saving model ...
In train(), epoch =  16
```

```
Epoch: 16         Training Loss: 0.247330         Validation Loss: 0.388546
Validation loss decreased (0.389932 --> 0.388546). Saving model ...
In train(), epoch =  17
Epoch: 17         Training Loss: 0.233965         Validation Loss: 0.389626
In train(), epoch =  18
Epoch: 18         Training Loss: 0.230284         Validation Loss: 0.390986
In train(), epoch =  19
Epoch: 19         Training Loss: 0.214810         Validation Loss: 0.394008
In train(), epoch =  20
Epoch: 20         Training Loss: 0.204551         Validation Loss: 0.391241
In train(), epoch =  21
Epoch: 21         Training Loss: 0.193930         Validation Loss: 0.389065
In train(), epoch =  22
Epoch: 22         Training Loss: 0.180788         Validation Loss: 0.383748
Validation loss decreased (0.388546 --> 0.383748). Saving model ...
In train(), epoch =  23
Epoch: 23         Training Loss: 0.181958         Validation Loss: 0.380217
Validation loss decreased (0.383748 --> 0.380217). Saving model ...
In train(), epoch =  24
Epoch: 24         Training Loss: 0.163827         Validation Loss: 0.369671
Validation loss decreased (0.380217 --> 0.369671). Saving model ...
In train(), epoch =  25
Epoch: 25         Training Loss: 0.157742         Validation Loss: 0.373034
In train(), epoch =  26
Epoch: 26         Training Loss: 0.145520         Validation Loss: 0.381735
In train(), epoch =  27
Epoch: 27         Training Loss: 0.141760         Validation Loss: 0.383525
In train(), epoch =  28
Epoch: 28         Training Loss: 0.144963         Validation Loss: 0.378544
In train(), epoch =  29
Epoch: 29         Training Loss: 0.124519         Validation Loss: 0.376772
In train(), epoch =  30
Epoch: 30         Training Loss: 0.126002         Validation Loss: 0.388463
In train(), epoch =  31
Epoch: 31         Training Loss: 0.119398         Validation Loss: 0.382014
In train(), epoch =  32
Epoch: 32         Training Loss: 0.115739         Validation Loss: 0.385213
In train(), epoch =  33
Epoch: 33         Training Loss: 0.110891         Validation Loss: 0.379254
In train(), epoch =  34
Epoch: 34         Training Loss: 0.104787         Validation Loss: 0.386373
In train(), epoch =  35
Epoch: 35         Training Loss: 0.108917         Validation Loss: 0.378296
In train(), epoch =  36
Epoch: 36         Training Loss: 0.100663         Validation Loss: 0.382243
In train(), epoch =  37
```

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%.

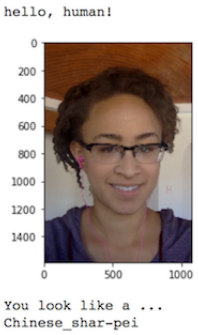
In [27]:
```
# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))

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

```
Test Loss: 0.434784


Test Accuracy: 87% (729/836)
```

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 [28]:
```
from PIL import Image
import torchvision.transforms as transforms

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

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ")
               for item in data_transfer['train'].classes]
# print(class_names)

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    transform = transforms.Compose([transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std =[0.229, 0.224, 0.225])])

    img = Image.open(img_path)
    img_t = transform(img)
    img_t = img_t.view(1, img_t.shape[0], img_t.shape[1], img_t.shape[2])
    # Ligang: necessary or RuntimeError: Expected object of type
    # torch.FloatTensor but found type torch.cuda.FloatTensor for
    # argument #2 weight
    img_t = img_t.to('cuda')

    model_transfer.eval()
    output = model_transfer(img_t)
```



Sample Human Output

```
    value, idx = torch.max(output, 1)

    return class_names[idx]
```

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 [29]:
```
### 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
    if dog_detector(img_path):
        dog_breed = predict_breed_transfer(img_path)
        print('Predicted dog breed: ', dog_breed)
        return dog_breed
    elif face_detector(img_path):
        resembling_dog_breed = predict_breed_transfer(img_path)
        print('Resembing dog breed: ', resembling_dog_breed)
        return resembling_dog_breed
    else:
        print('In function run_app(), ERROR: neither dog \
            nor human faces detected!')
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement) 1. It seems like transfer learning is really effective for this kind of task, one possible point for improvement is to train to enhance the classifier of the transferred model (here it's VGG16) to detect the dog breed; 2. the human face detection algorithm using OpenCV can be improved as the percentage of detected human face from dog files is 17% which is somehow high and unacceptable; 3. We may use CNN to detect human face, which is expected to have higher accuracy.

In [30]:
```
## 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])):
    print('file: ', file)
    run_app(file)
```

```
file:  /data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
Resembing dog breed:  Brittany
file:  /data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
Resembing dog breed:  Dachshund
file:  /data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg
Resembing dog breed:  Irish water spaniel
file:  /data/dog_images/train/103.Mastiff/Mastiff_06833.jpg
Predicted dog breed:  Mastiff
file:  /data/dog_images/train/103.Mastiff/Mastiff_06826.jpg
Predicted dog breed:  Mastiff
file:  /data/dog_images/train/103.Mastiff/Mastiff_06871.jpg
Predicted dog breed:  Mastiff
```

In [ ]: