dog_app

April 19, 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
import torch
# load filenames for human and dog images

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

human_files = np.array(glob("/data/lfw/*/*"))
dog_files = np.array(glob("/data/dog_images/*/*/*"))
print(dog_files)

# 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))
```

```
['/data/dog_images/train/103.Mastiff/Mastiff_06833.jpg'
 '/data/dog_images/train/103.Mastiff/Mastiff_06826.jpg'
 '/data/dog_images/train/103.Mastiff/Mastiff_06871.jpg' ...,
 '/data/dog_images/valid/100.Lowchen/Lowchen_06682.jpg'
 '/data/dog_images/valid/100.Lowchen/Lowchen_06708.jpg'
 '/data/dog_images/valid/100.Lowchen/Lowchen_06684.jpg']
```

Step 1: Detect Humans

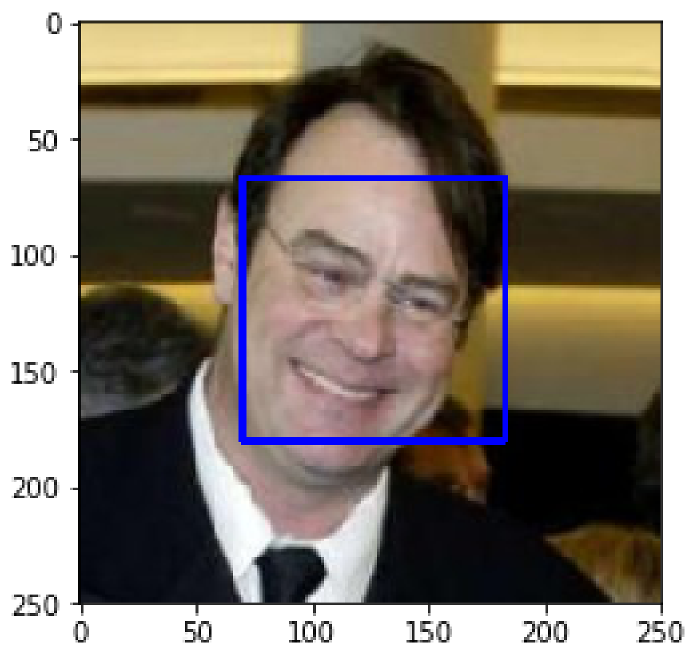
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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])
```

```python
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell) Out of the 100 dog images tested, 17 of them were identified as human. **17%** Out of the 100 human images tested, 98 of them were identified as human. **98%**

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        dogs = 0
        humans = 0
        for i in (human_files_short):
            if(face_detector(i)):
                humans += 1
        for i in (dog_files_short):
            if(face_detector(i)):
                dogs += 1
```

```
    print(str(dogs) + " dogs identified out of " + str(len(dog_files_short)) + " images")
    print(str(humans) + " humans identified out of " + str(len(human_files_short)) + " image
```

```
17 dogs identified out of 100 images
98 humans identified out of 100 images
```

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

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

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:11<00:00, 47480293.18it/s]


cuda
```

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

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

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

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

```
In [7]: from PIL import Image
        import torchvision.transforms as transforms

        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)
            #print(img)
            img = img.resize((224, 224))
            #print(img)

            transform = transforms.Compose([transforms.ToTensor()])
            img = transform(img)[:3, :, :].unsqueeze(0).to(device)
            #print(type(img))

            output = VGG16(img)
            #print(output)
            '''
            transformed = torch.nn.Sequential(
                tranforms.RandomAffine((-10, 10))
            )
            transformed = transformed.forward(img)
            transform_img = torch.jit.script(transformed)
            '''
            #predicted index
            index = torch.max(output, 1)[1].item()

            return index # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

In [8]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.

    if(VGG16_predict(img_path) >= 151 and VGG16_predict(img_path) <= 268):
        return True
    return False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: Out of the 100 dog images tested, 91 of then (**91%**) were identified as dogs. Out of the 100 human images tested, 0 of them (**0%**) were identified as dogs.

In [9]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_dogs = 0
dogs_dogs = 0
for human in human_files_short:
    if(dog_detector(human)):
        human_dogs += 1
for dog in dog_files_short:
    if(dog_detector(dog)):
        dogs_dogs += 1
print("Human Dogs detected: " + str(human_dogs))
print("Dogs detected: " + str(dogs_dogs))
```

```
Human Dogs detected: 0
Dogs detected: 93
```

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

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [11]: import os
         from torchvision import datasets

         ### TODO: Write data loaders for training, validation, and test sets
```

```
## Specify appropriate transforms, and batch_sizes

transform_m = transforms.Compose([transforms.Resize((224, 224)),
                                  transforms.RandomRotation((-10, 10)),
                                  transforms.RandomHorizontalFlip(p=0.2),
                                  transforms.RandomVerticalFlip(p=0.2),
                                  transforms.ToTensor(),
                                  transforms.Normalize([0.485, 0.456, 0.406],
                                              [0.229, 0.224, 0.225])
                                 ])
test_transform = transforms.Compose([transforms.Resize((224, 224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406],
                                              [0.229, 0.224, 0.225])])


training = datasets.ImageFolder("/data/dog_images/train", transform=transform_m)
training_loader = torch.utils.data.DataLoader(training,
                                              batch_size=10,
                                              shuffle=True,
                                              num_workers=0)

test = datasets.ImageFolder("/data/dog_images/test", transform= test_transform)
test_loader = torch.utils.data.DataLoader(test,
                                              batch_size=10,
                                              shuffle=False,
                                              num_workers=0)
validation = datasets.ImageFolder("/data/dog_images/valid", transform=test_transform)
valid_loader = torch.utils.data.DataLoader(validation,
                                              batch_size=10,
                                              shuffle=True,
                                              num_workers=0)
```

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: In term of my data loading, I didn't touch the testing data or validation data, but for training, I added some noise to the data to help with preventing overfitting and creating some variation in the data. I didn't go too crazy and just kept standard rotational variation as well as possible flips in the images. If I were to do this in the future, I would've also included some scaling as well as translation to add some more additional variation.

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
```

```python
# define the CNN architecture

class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        '''
        self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        '''
        self.conv1 = nn.Conv2d(3, 64, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)

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

        #self.dropout = nn.Dropout(0.1)
        self.dropout = nn.Dropout(0.3)

        self.fc1 = nn.Linear(7*7*256, 500)
        self.fc2 = nn.Linear(500, 133)

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


        x = x.view(-1, 7*7*256)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

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

# instantiate the CNN
model_scratch = Net()
```

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

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: It took me several tries to get my CNN architecture to work. I started with 3 layers because I thought that addeda. good amount of complexity but not too much to run into vanishing gradients. However, the nodes that I had in each layer originally was too little as overfitting was an issue I ran into several times. For this reason, I added a strong dropout layer (0.3) after trying 0.2, 0.5 which still was overfitting and underfitting respectively. For my learning rate, I started with 0.05 and changed finetuned it a couple times to 0.03 --> 0.01 until the results were promising. For the sake of time, I kept my epochs to 10 because it was enough to give the desirable results given the complexity of CNN. If I wanted a higher accuracy, I would increased the depth of my CNN while increasing my epochs.

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 = 0.03)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [14]: # the following import is required for training to be robust to truncated images
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

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

    ###################
    # train the model #
    ###################
    model.train()
    for batch_idx, (data, target) in enumerate(loaders['train']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## find the loss and update the model parameters accordingly
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
        '''
        '''
        optimizer.zero_grad()

        output = model(data)

        # calculate loss
        loss = criterion(output, target)

        # back prop
        loss.backward()

        # grad
        optimizer.step()

        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

        if batch_idx % 100 == 0:
            print('Epoch %d, Batch %d loss: %.6f' %
              (epoch, batch_idx + 1, train_loss))

        '''
        '''
    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['valid']):
        # move to GPU
```

```
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                '''
                '''
                output = model(data)
                loss = criterion(output, target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)
                '''
                '''
            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch,
                train_loss,
                valid_loss
                ))

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

```
In [15]: # train the model
         model_scratch = train(10, {"test": test_loader, "valid": valid_loader, "train": trainin
                               criterion_scratch, True, 'model_scratch.pt')

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

```
Epoch 1, Batch 1 loss: 4.884099
Epoch 1, Batch 101 loss: 4.893931
Epoch 1, Batch 201 loss: 4.892634
Epoch 1, Batch 301 loss: 4.888321
Epoch 1, Batch 401 loss: 4.881098
Epoch 1, Batch 501 loss: 4.872939
Epoch 1, Batch 601 loss: 4.856679
Epoch: 1 	Training Loss: 4.845883 	Validation Loss: 4.707011
Validation loss decreased (inf --> 4.707011).  Saving model ...
Epoch 2, Batch 1 loss: 4.717193
```

```
Epoch 2, Batch 101 loss: 4.686081
Epoch 2, Batch 201 loss: 4.677174
Epoch 2, Batch 301 loss: 4.652613
Epoch 2, Batch 401 loss: 4.631570
Epoch 2, Batch 501 loss: 4.611082
Epoch 2, Batch 601 loss: 4.590062
Epoch: 2        Training Loss: 4.581087         Validation Loss: 4.406184
Validation loss decreased (4.707011 --> 4.406184).  Saving model ...
Epoch 3, Batch 1 loss: 4.535175
Epoch 3, Batch 101 loss: 4.416432
Epoch 3, Batch 201 loss: 4.390098
Epoch 3, Batch 301 loss: 4.406672
Epoch 3, Batch 401 loss: 4.397237
Epoch 3, Batch 501 loss: 4.404197
Epoch 3, Batch 601 loss: 4.393214
Epoch: 3        Training Loss: 4.387294         Validation Loss: 4.391872
Validation loss decreased (4.406184 --> 4.391872).  Saving model ...
Epoch 4, Batch 1 loss: 4.368187
Epoch 4, Batch 101 loss: 4.318686
Epoch 4, Batch 201 loss: 4.300822
Epoch 4, Batch 301 loss: 4.280744
Epoch 4, Batch 401 loss: 4.271380
Epoch 4, Batch 501 loss: 4.264533
Epoch 4, Batch 601 loss: 4.255962
Epoch: 4        Training Loss: 4.251028         Validation Loss: 4.196798
Validation loss decreased (4.391872 --> 4.196798).  Saving model ...
Epoch 5, Batch 1 loss: 3.875624
Epoch 5, Batch 101 loss: 4.116337
Epoch 5, Batch 201 loss: 4.118879
Epoch 5, Batch 301 loss: 4.134486
Epoch 5, Batch 401 loss: 4.127290
Epoch 5, Batch 501 loss: 4.131657
Epoch 5, Batch 601 loss: 4.125459
Epoch: 5        Training Loss: 4.127630         Validation Loss: 4.073413
Validation loss decreased (4.196798 --> 4.073413).  Saving model ...
Epoch 6, Batch 1 loss: 3.506207
Epoch 6, Batch 101 loss: 3.978005
Epoch 6, Batch 201 loss: 4.016391
Epoch 6, Batch 301 loss: 4.022260
Epoch 6, Batch 401 loss: 4.027230
Epoch 6, Batch 501 loss: 4.021983
Epoch 6, Batch 601 loss: 4.021937
Epoch: 6        Training Loss: 4.018847         Validation Loss: 4.128045
Epoch 7, Batch 1 loss: 4.378740
Epoch 7, Batch 101 loss: 3.876582
Epoch 7, Batch 201 loss: 3.893513
Epoch 7, Batch 301 loss: 3.890886
Epoch 7, Batch 401 loss: 3.882089
```

```
Epoch 7, Batch 501 loss: 3.887866
Epoch 7, Batch 601 loss: 3.891506
Epoch: 7         Training Loss: 3.893743         Validation Loss: 4.027795
Validation loss decreased (4.073413 --> 4.027795).  Saving model ...
Epoch 8, Batch 1 loss: 4.093349
Epoch 8, Batch 101 loss: 3.696980
Epoch 8, Batch 201 loss: 3.761820
Epoch 8, Batch 301 loss: 3.768810
Epoch 8, Batch 401 loss: 3.777510
Epoch 8, Batch 501 loss: 3.782937
Epoch 8, Batch 601 loss: 3.781284
Epoch: 8         Training Loss: 3.785507         Validation Loss: 3.917286
Validation loss decreased (4.027795 --> 3.917286).  Saving model ...
Epoch 9, Batch 1 loss: 4.157474
Epoch 9, Batch 101 loss: 3.593159
Epoch 9, Batch 201 loss: 3.585684
Epoch 9, Batch 301 loss: 3.598347
Epoch 9, Batch 401 loss: 3.629140
Epoch 9, Batch 501 loss: 3.641550
Epoch 9, Batch 601 loss: 3.640603
Epoch: 9         Training Loss: 3.653280         Validation Loss: 3.813257
Validation loss decreased (3.917286 --> 3.813257).  Saving model ...
Epoch 10, Batch 1 loss: 3.343842
Epoch 10, Batch 101 loss: 3.431300
Epoch 10, Batch 201 loss: 3.481800
Epoch 10, Batch 301 loss: 3.478712
Epoch 10, Batch 401 loss: 3.497577
Epoch 10, Batch 501 loss: 3.514180
Epoch 10, Batch 601 loss: 3.520767
Epoch: 10         Training Loss: 3.528549         Validation Loss: 3.831519
```

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 [16]: 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, "valid": valid_loader, "train": training_loader}, model_scra
```

```
Test Loss: 3.795981


Test Accuracy: 12% (101/836)
```

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](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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 [33]: ## TODO: Specify data loaders

         #same as CNN from scratch



         transform_m = transforms.Compose([transforms.Resize((224, 224)),
                                           transforms.RandomRotation((-10, 10)),
                                           transforms.RandomHorizontalFlip(p=0.2),
```

```
                                            transforms.RandomVerticalFlip(p=0.2),
                                            transforms.ToTensor(),
                                            transforms.Normalize([0.485, 0.456, 0.406],
                                                                 [0.229, 0.224, 0.225])])
test_transform = transforms.Compose([transforms.Resize((224, 224)),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.406],
                                                          [0.229, 0.224, 0.225])])

training = datasets.ImageFolder("/data/dog_images/train", transform=transform_m)
training_loader = torch.utils.data.DataLoader(training,
                                              batch_size=10,
                                              shuffle=True,
                                              num_workers=0)

test = datasets.ImageFolder("/data/dog_images/test", transform=test_transform)
test_loader = torch.utils.data.DataLoader(test,
                                          batch_size=10,
                                          shuffle=False,
                                          num_workers=0)
validation = datasets.ImageFolder("/data/dog_images/valid", transform=test_transform)
valid_loader = torch.utils.data.DataLoader(validation,
                                           batch_size=10,
                                           shuffle=True,
                                           num_workers=0)
data_loader = {"test": test_loader, "valid": valid_loader, "train": training_loader}
```

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

         ## TODO: Specify model architecture
         model_transfer = models.resnet18(pretrained=True)

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

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root/.torch/models/
100%|| 46827520/46827520 [00:01<00:00, 32692015.23it/s]
```

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: My data loader structure was the exact same as before as the overall transforms did a good job of giving the data a bit of variance. For transferring learning, I decided to use resnet18 partly because it had a lower complexity than VGG16 to reduce runtime, but also had a deeper complexity so I was curious to see how it would see how it would compare to VGG. Overall, it turned out pretty well with a 73% accuracy. If I had more time, I would decrease the learning rate as the model converged faster than I expected in the transfer learning model.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [19]: criterion_transfer = nn.CrossEntropyLoss()

         optimizer_transfer = optim.SGD(model_transfer.parameters(), lr = 0.01)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [20]: # train the model
         model_transfer =  train(10, {"test": test_loader, "valid": valid_loader, "train": train

         # 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, Batch 1 loss: 9.837217
Epoch 1, Batch 101 loss: 6.998576
Epoch 1, Batch 201 loss: 5.813467
Epoch 1, Batch 301 loss: 5.079399
Epoch 1, Batch 401 loss: 4.566264
Epoch 1, Batch 501 loss: 4.180541
Epoch 1, Batch 601 loss: 3.896756
Epoch: 1 	Training Loss: 3.736050 	Validation Loss: 1.762045
Validation loss decreased (inf --> 1.762045).  Saving model ...
Epoch 2, Batch 1 loss: 1.349191
Epoch 2, Batch 101 loss: 1.843461
Epoch 2, Batch 201 loss: 1.810759
Epoch 2, Batch 301 loss: 1.771999
Epoch 2, Batch 401 loss: 1.760428
Epoch 2, Batch 501 loss: 1.731019
Epoch 2, Batch 601 loss: 1.693552
Epoch: 2 	Training Loss: 1.686150 	Validation Loss: 1.120143
Validation loss decreased (1.762045 --> 1.120143).  Saving model ...
Epoch 3, Batch 1 loss: 0.838437
Epoch 3, Batch 101 loss: 1.136410
Epoch 3, Batch 201 loss: 1.166230
Epoch 3, Batch 301 loss: 1.192348
Epoch 3, Batch 401 loss: 1.205835
Epoch 3, Batch 501 loss: 1.195056
```

```
Epoch 3, Batch 601 loss: 1.199719
Epoch: 3        Training Loss: 1.196599        Validation Loss: 1.067826
Validation loss decreased (1.120143 --> 1.067826).  Saving model ...
Epoch 4, Batch 1 loss: 0.804764
Epoch 4, Batch 101 loss: 0.994973
Epoch 4, Batch 201 loss: 0.955254
Epoch 4, Batch 301 loss: 0.952724
Epoch 4, Batch 401 loss: 0.938831
Epoch 4, Batch 501 loss: 0.933993
Epoch 4, Batch 601 loss: 0.929762
Epoch: 4        Training Loss: 0.932836        Validation Loss: 0.891682
Validation loss decreased (1.067826 --> 0.891682).  Saving model ...
Epoch 5, Batch 1 loss: 0.829798
Epoch 5, Batch 101 loss: 0.768718
Epoch 5, Batch 201 loss: 0.767983
Epoch 5, Batch 301 loss: 0.764099
Epoch 5, Batch 401 loss: 0.758138
Epoch 5, Batch 501 loss: 0.773875
Epoch 5, Batch 601 loss: 0.780611
Epoch: 5        Training Loss: 0.771187        Validation Loss: 0.875868
Validation loss decreased (0.891682 --> 0.875868).  Saving model ...
Epoch 6, Batch 1 loss: 0.607881
Epoch 6, Batch 101 loss: 0.702899
Epoch 6, Batch 201 loss: 0.678944
Epoch 6, Batch 301 loss: 0.664797
Epoch 6, Batch 401 loss: 0.652978
Epoch 6, Batch 501 loss: 0.642630
Epoch 6, Batch 601 loss: 0.629805
Epoch: 6        Training Loss: 0.632011        Validation Loss: 0.830197
Validation loss decreased (0.875868 --> 0.830197).  Saving model ...
Epoch 7, Batch 1 loss: 0.174211
Epoch 7, Batch 101 loss: 0.520344
Epoch 7, Batch 201 loss: 0.513485
Epoch 7, Batch 301 loss: 0.524491
Epoch 7, Batch 401 loss: 0.529810
Epoch 7, Batch 501 loss: 0.535215
Epoch 7, Batch 601 loss: 0.537675
Epoch: 7        Training Loss: 0.536807        Validation Loss: 0.888374
Epoch 8, Batch 1 loss: 0.095546
Epoch 8, Batch 101 loss: 0.458304
Epoch 8, Batch 201 loss: 0.451062
Epoch 8, Batch 301 loss: 0.457890
Epoch 8, Batch 401 loss: 0.446528
Epoch 8, Batch 501 loss: 0.452092
Epoch 8, Batch 601 loss: 0.458420
Epoch: 8        Training Loss: 0.462824        Validation Loss: 0.859769
Epoch 9, Batch 1 loss: 0.358968
Epoch 9, Batch 101 loss: 0.415242
```

```
Epoch 9, Batch 201 loss: 0.412756
Epoch 9, Batch 301 loss: 0.413836
Epoch 9, Batch 401 loss: 0.415706
Epoch 9, Batch 501 loss: 0.403042
Epoch 9, Batch 601 loss: 0.407078
Epoch: 9         Training Loss: 0.408167         Validation Loss: 0.813010
Validation loss decreased (0.830197 --> 0.813010).  Saving model ...
Epoch 10, Batch 1 loss: 0.402015
Epoch 10, Batch 101 loss: 0.314760
Epoch 10, Batch 201 loss: 0.313536
Epoch 10, Batch 301 loss: 0.318664
Epoch 10, Batch 401 loss: 0.325169
Epoch 10, Batch 501 loss: 0.322033
Epoch 10, Batch 601 loss: 0.335441
Epoch: 10        Training Loss: 0.345061         Validation Loss: 0.804711
Validation loss decreased (0.813010 --> 0.804711).  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%.

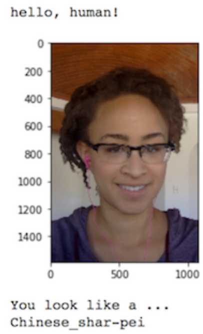
In [34]:
```
test(data_loader, model_transfer, criterion_transfer, use_cuda)

 #test({"test": test_loader, "valid": valid_loader, "train": training_loader}, model_scr
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-34-e29d1f292469> in <module>()
          1
    ----> 2 test(data_loader, model_transfer, criterion_transfer, use_cuda)
          3
          4 #test({"test": test_loader, "valid": valid_loader, "train": training_loader}, model_


        TypeError: 'ImageFolder' object is not callable
```

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



Sample Human Output

```
# list of class names by index, i.e. a name can be accessed like class_names[0]
dog_files = np.array(glob("/data/dog_images/*/*"))
#print(dog_files)
class_names = np.array(sorted(set([item[4:].replace("_", " ").split(".")[1] for item in
print(class_names)
def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = Image.open(img_path)
    img = img.resize((224, 224))
    transform = transforms.Compose([transforms.ToTensor()])

    img = transform(img)[:3, :, :].unsqueeze(0).to(device)

    out = model_transfer(img)

    _, preds_tensor = torch.max(out, 1)

    pred = np.squeeze(preds_tensor.cpu().numpy())

    return class_names[pred]
```

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 [ ]:
```
### 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
    msg = ""
    breed = predict_breed_transfer(img_path)
    if(face_detector(img_path)):
        #human
        msg = "Hi human, you look like a(n): " + str(breed)
    elif(dog_detector(img_path)):
        #dog
        msg = "Hi Dog, you look like you are a(n): " + str(breed)
    else:
        #neither
        msg = "Hi, ... whatever you are, you look like a(n): " + str(breed)
    plt.figure()
    plt.imshow(Image.open(img_path))
    plt.title(msg)
    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) One area of improvement would obviously to increase my test accuracy by modifying the model. I think I could've done a better job by lowering increasi

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