dog_app

December 5, 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 [29]: import numpy as np
         from glob import glob

         # load filenames for human and dog images
         human_files = np.array(glob("/data/lfw/*/*"))
         dog_files = np.array(glob("/data/dog_images/*/*/*"))

         # print number of images in each dataset
         print('There are %d total human images.' % len(human_files))
         print('There are %d total dog images.' % len(dog_files))
```

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

Step 1: Detect Humans

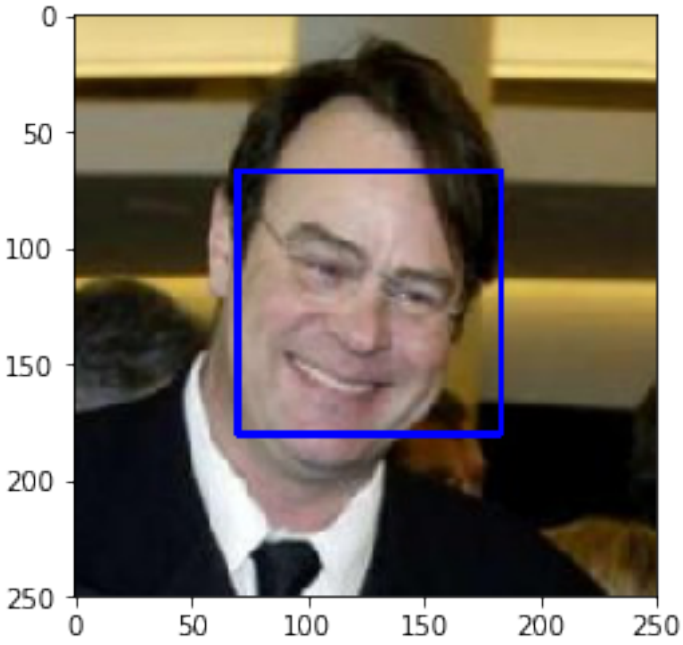
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

```
In [30]: 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 [31]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
In [32]: 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.


         human_count = 0
         dog_count = 0

         for img in human_files_short:
             if face_detector(img):
                 human_count += 1

         for img in dog_files_short:
             if face_detector(img):
                 dog_count += 1

         print("Humans in human_files_short:", human_count)
         print("Humans in dog_files_short:", dog_count)
```

```
Humans in human_files_short: 98
Humans in dog_files_short: 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 [33]: ### (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 [34]: import torch
         import torchvision.models as models

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

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

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

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

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

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

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

In [35]:
```
from PIL import Image
import torchvision.transforms as transforms
from torch.autograd import Variable

#PIL for truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True


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

    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    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])])

    # Load image
    img = Image.open(img_path)

    # Transform image
    img = transform(img)

    # convert Tensor to one-dimensional one
    img = img.unsqueeze(0)

    # img to a var.
    img = Variable(img)

    if use_cuda:
        img = img.cuda()

    # prediction model
    prediction = VGG16(img)
```

```
    # max Value of Tensor matrix and return as int
    _, index = torch.max(prediction, 1)


    return index.item() # 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 [36]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.

             predicted_index = VGG16_predict(img_path)

             result = predicted_index >=151 and predicted_index <=268

             return result # 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:

- Dogs in human_files_short: 1%
- Dogs in dog_files_short: 100%

```
In [37]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         human_count = 0
         dog_count = 0

         for img in human_files_short:
             if dog_detector(img):
                 human_count += 1

         for img in dog_files_short:
             if dog_detector(img):
                 dog_count += 1
```

```
        print("Dogs in human_files_short:", human_count)
        print("Dogs in dog_files_short:", dog_count)
```

```
Dogs in human_files_short: 1
Dogs in dog_files_short: 100
```

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

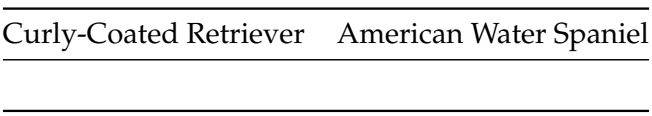
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.

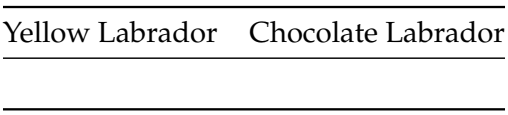
Brittany Welsh Springer Spaniel

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 [39]: import os
         from torchvision import datasets

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


         data_dir = '/data/dog_images/'

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


         # std transormations normalize

         normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406],         # Normalize im
                                               std=[0.229, 0.224, 0.225])

         train_transform = transforms.Compose([transforms.Resize(256),            # Resize image
                                               transforms.RandomResizedCrop(224), # image crop t
                                               transforms.RandomHorizontalFlip(), # flip image h
                                               transforms.RandomRotation(10),     # image rotate
                                               transforms.ToTensor(),             # Convert to P
                                               normalization])


         test_transform  = transforms.Compose([transforms.Resize(256),            # Resize image
                                               transforms.CenterCrop(224),        # image crop t
                                               transforms.ToTensor(),             # Convert to P
                                               normalization])


         valid_transform = transforms.Compose([transforms.Resize(256),            # Resize image
                                               transforms.CenterCrop(224),        # image crop t
```

```
                                        transforms.ToTensor(),              # Convert to P
                                        normalization])


batch_size = 20
num_workers = 0


train_data = datasets.ImageFolder(train_dir, transform=train_transform)
test_data = datasets.ImageFolder(test_dir, transform=test_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=valid_transform)



train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
test_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=Tr
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T



loaders_scratch = {"train" : train_loader, "valid" : valid_loader, "test" : test_loader
```

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:

- To prevent overfitting the model with test or training data, I used image augmentation. Random resize crop to 224, random flipping, and random rotation were utilised as transforms.
- Only image scaling was done for validation and test data.
- Each of the three datasets has been normalised.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [40]: import torch.nn as nn
         import torch.nn.functional as F

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

```
            self.conv1 = nn.Conv2d(3, 36, 3, padding=1)

            self.conv2 = nn.Conv2d(36, 64, 3, padding=1)

            self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

            self.fc1 = nn.Linear(28*28*128, 512)

            self.fc2 = nn.Linear(512, 133)

            self.dropout = nn.Dropout(0.25)

            self.batch_norm = nn.BatchNorm1d(512)


        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 = x.view(-1, 28*28*128)

            x = F.relu(self.batch_norm(self.fc1(x)))

            x = self.dropout(x)

            x = F.relu(self.fc2(x))

            return x

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

    # instantiate the CNN
    model_scratch = Net()

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

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

Answer:

- There are three convolutional layers in the model.
- The kernal size of all convolutional layers is 3 and the stride is 1.
- In channels = 3 in the first conv layer (conv1), and output size = 128 in the last conv layer (conv3).
- Here, the ReLU activation function is used. The pooling layer (2,2) is used, resulting in a 2x reduction in input size.
- We have two layers that are totally coupled and provide a 133-dimensional output.
- To avoid overfitting, a 0.25 dropout is included.

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 [41]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.02)
```

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 [42]: 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

    ######################
    # 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
        optimizer.zero_grad()

        output = model(data)

        loss = criterion(output, target)

        loss.backward()

        optimizer.step()

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


    ##Model Evaluation

    model.eval()

    for batch_idx, (data, target) in enumerate(loaders['valid']):

        if use_cuda:
            data, target = data.cuda(), target.cuda()

        # avg validation loss update
        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:
                    print('Validation loss decreased to ({:.5f} -> {:.5f}). Saving your model..

                    torch.save(model.state_dict(), save_path)

                    valid_loss_min = valid_loss

            # return trained model
            return model


        # train the model
        model_scratch = train(50, loaders_scratch, model_scratch, optimizer_scratch,
                              criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.885409         Validation Loss: 4.867907
Validation loss decreased to (inf -> 4.86791). Saving your model...
Epoch: 2         Training Loss: 4.844233         Validation Loss: 4.800344
Validation loss decreased to (4.86791 -> 4.80034). Saving your model...
Epoch: 3         Training Loss: 4.758378         Validation Loss: 4.720137
Validation loss decreased to (4.80034 -> 4.72014). Saving your model...
Epoch: 4         Training Loss: 4.701898         Validation Loss: 4.689781
Validation loss decreased to (4.72014 -> 4.68978). Saving your model...
Epoch: 5         Training Loss: 4.674685         Validation Loss: 4.645593
Validation loss decreased to (4.68978 -> 4.64559). Saving your model...
Epoch: 6         Training Loss: 4.652097         Validation Loss: 4.613872
Validation loss decreased to (4.64559 -> 4.61387). Saving your model...
Epoch: 7         Training Loss: 4.628933         Validation Loss: 4.592705
Validation loss decreased to (4.61387 -> 4.59270). Saving your model...
Epoch: 8         Training Loss: 4.600497         Validation Loss: 4.547919
Validation loss decreased to (4.59270 -> 4.54792). Saving your model...
Epoch: 9         Training Loss: 4.581365         Validation Loss: 4.528256
Validation loss decreased to (4.54792 -> 4.52826). Saving your model...
Epoch: 10         Training Loss: 4.517785         Validation Loss: 4.456105
Validation loss decreased to (4.52826 -> 4.45611). Saving your model...
Epoch: 11         Training Loss: 4.485442         Validation Loss: 4.511076
Epoch: 12         Training Loss: 4.447281         Validation Loss: 4.386565
Validation loss decreased to (4.45611 -> 4.38656). Saving your model...
Epoch: 13         Training Loss: 4.419789         Validation Loss: 4.382131
Validation loss decreased to (4.38656 -> 4.38213). Saving your model...
Epoch: 14         Training Loss: 4.385170         Validation Loss: 4.299922
Validation loss decreased to (4.38213 -> 4.29992). Saving your model...
```

```
Epoch: 15         Training Loss: 4.341622         Validation Loss: 4.272478
Validation loss decreased to (4.29992 -> 4.27248). Saving your model...
Epoch: 16         Training Loss: 4.303837         Validation Loss: 4.261592
Validation loss decreased to (4.27248 -> 4.26159). Saving your model...
Epoch: 17         Training Loss: 4.284104         Validation Loss: 4.175511
Validation loss decreased to (4.26159 -> 4.17551). Saving your model...
Epoch: 18         Training Loss: 4.224945         Validation Loss: 4.150139
Validation loss decreased to (4.17551 -> 4.15014). Saving your model...
Epoch: 19         Training Loss: 4.209164         Validation Loss: 4.113162
Validation loss decreased to (4.15014 -> 4.11316). Saving your model...
Epoch: 20         Training Loss: 4.162547         Validation Loss: 4.037256
Validation loss decreased to (4.11316 -> 4.03726). Saving your model...
Epoch: 21         Training Loss: 4.106855         Validation Loss: 3.988164
Validation loss decreased to (4.03726 -> 3.98816). Saving your model...
Epoch: 22         Training Loss: 4.062446         Validation Loss: 4.014199
Epoch: 23         Training Loss: 4.041790         Validation Loss: 3.927135
Validation loss decreased to (3.98816 -> 3.92713). Saving your model...
Epoch: 24         Training Loss: 4.012832         Validation Loss: 4.000094
Epoch: 25         Training Loss: 3.965011         Validation Loss: 3.911045
Validation loss decreased to (3.92713 -> 3.91105). Saving your model...
Epoch: 26         Training Loss: 3.926896         Validation Loss: 3.815401
Validation loss decreased to (3.91105 -> 3.81540). Saving your model...
Epoch: 27         Training Loss: 3.867234         Validation Loss: 3.766513
Validation loss decreased to (3.81540 -> 3.76651). Saving your model...
Epoch: 28         Training Loss: 3.831679         Validation Loss: 3.674906
Validation loss decreased to (3.76651 -> 3.67491). Saving your model...
Epoch: 29         Training Loss: 3.816123         Validation Loss: 3.705954
Epoch: 30         Training Loss: 3.746735         Validation Loss: 3.697977
Epoch: 31         Training Loss: 3.710623         Validation Loss: 3.539691
Validation loss decreased to (3.67491 -> 3.53969). Saving your model...
Epoch: 32         Training Loss: 3.656083         Validation Loss: 3.566325
Epoch: 33         Training Loss: 3.611207         Validation Loss: 3.475034
Validation loss decreased to (3.53969 -> 3.47503). Saving your model...
Epoch: 34         Training Loss: 3.554877         Validation Loss: 3.454915
Validation loss decreased to (3.47503 -> 3.45491). Saving your model...
Epoch: 35         Training Loss: 3.512004         Validation Loss: 3.408947
Validation loss decreased to (3.45491 -> 3.40895). Saving your model...
Epoch: 36         Training Loss: 3.495479         Validation Loss: 3.288110
Validation loss decreased to (3.40895 -> 3.28811). Saving your model...
Epoch: 37         Training Loss: 3.459836         Validation Loss: 3.337583
Epoch: 38         Training Loss: 3.408551         Validation Loss: 3.345372
Epoch: 39         Training Loss: 3.369099         Validation Loss: 3.281963
Validation loss decreased to (3.28811 -> 3.28196). Saving your model...
Epoch: 40         Training Loss: 3.347204         Validation Loss: 3.444664
Epoch: 41         Training Loss: 3.334260         Validation Loss: 3.187074
Validation loss decreased to (3.28196 -> 3.18707). Saving your model...
Epoch: 42         Training Loss: 3.279957         Validation Loss: 3.165447
Validation loss decreased to (3.18707 -> 3.16545). Saving your model...
```

```
Epoch: 43         Training Loss: 3.238782         Validation Loss: 3.128869
Validation loss decreased to (3.16545 -> 3.12887). Saving your model...
Epoch: 44         Training Loss: 3.215044         Validation Loss: 3.137818
Epoch: 45         Training Loss: 3.164517         Validation Loss: 3.064783
Validation loss decreased to (3.12887 -> 3.06478). Saving your model...
Epoch: 46         Training Loss: 3.121461         Validation Loss: 3.091384
Epoch: 47         Training Loss: 3.076018         Validation Loss: 2.946315
Validation loss decreased to (3.06478 -> 2.94631). Saving your model...
Epoch: 48         Training Loss: 3.066149         Validation Loss: 3.002539
Epoch: 49         Training Loss: 3.051628         Validation Loss: 2.877955
Validation loss decreased to (2.94631 -> 2.87795). Saving your model...
Epoch: 50         Training Loss: 3.004128         Validation Loss: 2.832122
Validation loss decreased to (2.87795 -> 2.83212). Saving your model...
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [43]: def test(loaders, model, criterion, use_cuda):

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

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders['test']):
                 # move to GPU
                 if use_cuda:
                     data, target = data.cuda(), target.cuda()
                 # forward pass: compute predicted outputs by passing inputs to the model
                 output = model(data)
                 # calculate the loss
                 loss = criterion(output, target)
                 # update average test loss
                 test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
                 # convert output probabilities to predicted class
                 pred = output.data.max(1, keepdim=True)[1]
                 # compare predictions to true label
                 correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
                 total += data.size(0)

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

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

```
        # call test function
        test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.810720


Test Accuracy: 31% (2112/6680)
```

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 [49]:
```
## TODO: Specify data loaders

loaders_transfer = loaders_scratch.copy()
```

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 [50]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

model_transfer = models.resnet101(pretrained=True)

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

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

Answer:

Here, I chose Transfer Learning as one of the best available options for creating a model for a dog breed classification system. For this method, I used the ResNext-101 model, which can be found at facebookresearch - ResNeXt. To fine-tune the classifier, I've frozen all of the model's

parameters and replaced the fully-connected layer (fc). I've added a dropout (drop) and a fully-connected (fc) layer to the mix. The Top-1 error rate of the ResNext-101 model is lower than that of the ResNet-50 model. As a result, I believe it will perform better. The following procedures are followed:

- loaded the pretrained resnext101 32x8d model,
- frozen all of the model's parameters,
- replaced the fully-connected layer (fc) to reset the classifier for fine-tuning,
- added a dropout layer (drop) to reduce overfitting,
- and added a fully-connected (fc1) layer with an output-size of 134 neurons.

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 [51]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.02)
```

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 [52]: # train the model
         model_transfer = train(6, loaders_transfer, model_transfer, optimizer_transfer, criteri

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

```
Epoch: 1         Training Loss: 4.835162         Validation Loss: 2.204572
Validation loss decreased to (inf -> 2.20457). Saving your model...
Epoch: 2         Training Loss: 1.781053         Validation Loss: 1.445770
Validation loss decreased to (2.20457 -> 1.44577). Saving your model...
Epoch: 3         Training Loss: 1.312442         Validation Loss: 1.193947
Validation loss decreased to (1.44577 -> 1.19395). Saving your model...
Epoch: 4         Training Loss: 1.180456         Validation Loss: 1.031374
Validation loss decreased to (1.19395 -> 1.03137). Saving your model...
Epoch: 5         Training Loss: 1.044462         Validation Loss: 0.965804
Validation loss decreased to (1.03137 -> 0.96580). Saving your model...
Epoch: 6         Training Loss: 0.970093         Validation Loss: 0.920613
Validation loss decreased to (0.96580 -> 0.92061). Saving your 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 [53]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

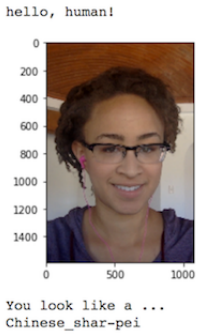
```
Test Loss: 0.909920


Test Accuracy: 76% (5085/6680)
```

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 [62]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         import os
         from torchvision import datasets

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

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

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed

             image = Image.open(img_path).convert('RGB')

             normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.22

             transformations = transforms.Compose([transforms.Resize(size=(224, 224)),transforms

             trans_image = transformations(image)[:3,:,:].unsqueeze(0)

             if use_cuda:
                 trans_image = trans_image.cuda()

             output = model_transfer(trans_image)

             pred_index = torch.max(output,1)[1].item()

             return class_names[pred_index]
```

Step 5: Write your Algorithm



Sample Human Output

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 [68]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def image_load(img_path):
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

def run_app(img_path):
    ## handle cases for a human face, dog, and neither


    if face_detector(img_path):
        print ("Hi Human!")

        predicted_breed = predict_breed_transfer(img_path)
        image_load(img_path)
        print("Predicted : ",predicted_breed,'\n')

    elif dog_detector(img_path):

        print ("It's a Dog!")
```

```
            predicted_breed = predict_breed_transfer(img_path)
            image_load(img_path)
            print("Predicted : ",predicted_breed,'\n')

        else:
            print ("Cannot Identify")
```

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.

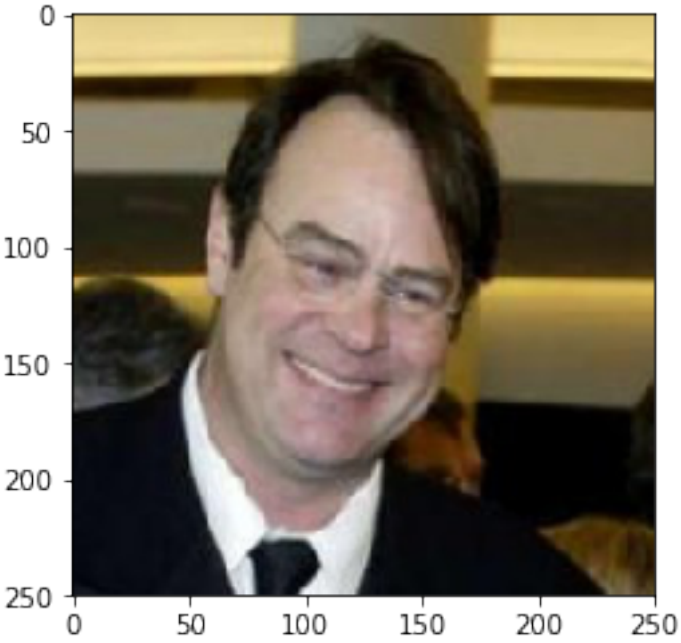
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

- Model improvement will be aided by more training data.
- Hyper parameter adjustment will also aid in performance enhancement.
- To make the algorithm more accessible to individuals, it might be made available as a web application.
- If an image contains both a dog and a human, an additional function might be added to generate a prediction for both.

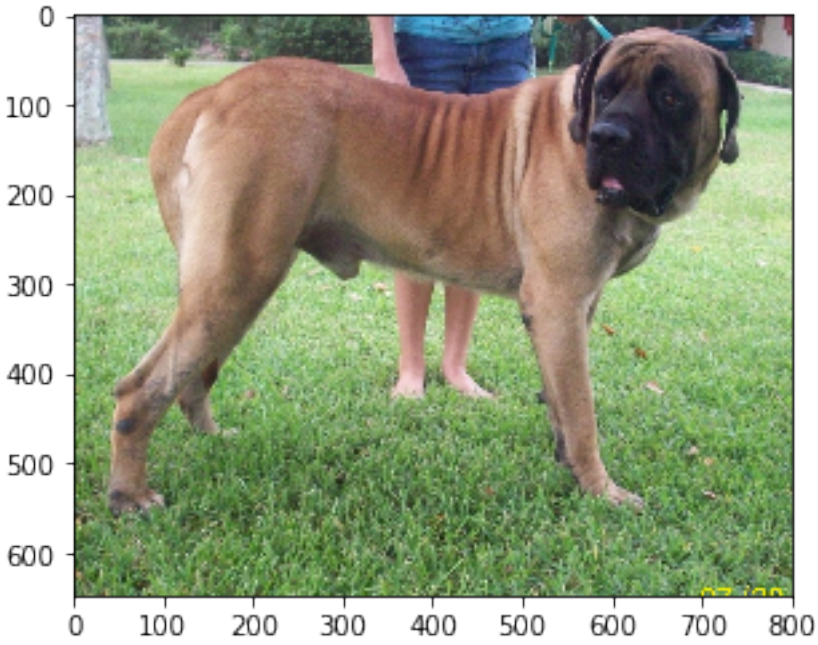
```
In [71]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[:1], dog_files[:1])) :
             run_app(file)
```

```
Hi Human!
```




```
Predicted :  Petit basset griffon vendeen
```

```
It's a Dog!
```



```
Predicted :  Bullmastiff
```

1.2 OPTIONAL: Question for the reviewer

If you have any question about the starter code or your own implementation, please add it in the cell below.

For example, if you want to know why a piece of code is written the way it is, or its function, or alternative ways of implementing the same functionality, or if you want to get feedback on a specific part of your code or get feedback on things you tried but did not work.

Please keep your questions succinct and clear to help the reviewer answer them satisfactorily.

In [ ]: