dog_app

October 28, 2019

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 [15]: 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

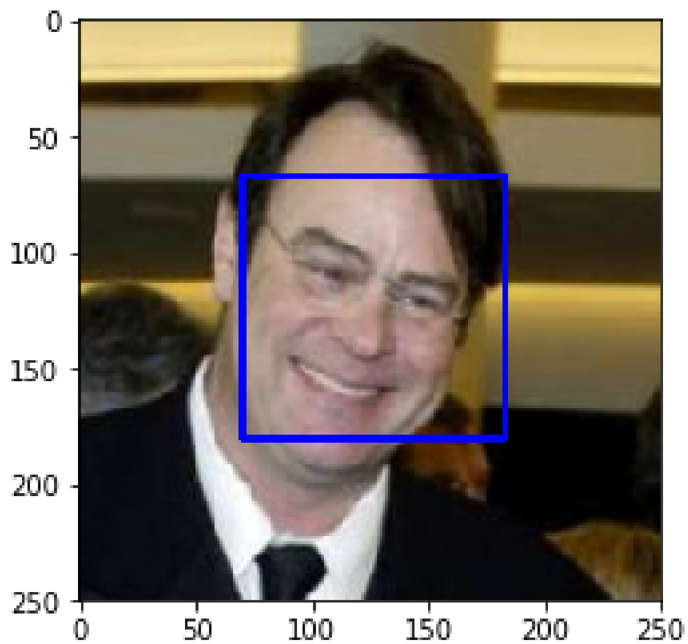
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [16]: 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))
```

```python
# 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 [52]: # 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) - The percentage of the human images that include a detected, human face: 100.0%. - The percentage of the dog images that include a detected, human face: 9.0%.

```
In [7]: from tqdm import tqdm

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        def percentageOfDetectingFaces(detector, img_paths):
            count = 0
            for img_path in img_paths:
                if detector(img_path):
                    count += 1
            return count / len(img_paths) * 100

        print("The percentage of the human images that include a detected, human face: {}%.".for
        print("The percentage of the dog images that include a detected, human face: {}%.".forma
```

```
The percentage of the human images that include a detected, human face: 100.0%.
The percentage of the dog images that include a detected, human face: 9.0%.
```

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 [0]: ### (Optional)
        ### TODO: Test performance of another face 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 [20]: import torch
         import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:21<00:00, 26102828.02it/s]
```

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

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

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

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

In [21]:
```
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)

    # Define the preprocessing transform.
    transform = transforms.Compose([transforms.Resize(255),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor()])

    # Reference: https://mlpipes.com/pytorch-quick-start-classifying-an-image/
    # Preprocess the image.
    img_tensor = transform(img)
    img_tensor.unsqueeze_(0)

    if use_cuda:
        img_tensor = img_tensor.cuda()

    prediction = VGG16(img_tensor)

    if use_cuda:
        prediction = prediction.cpu()

    return prediction.data.numpy().argmax() # 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 [12]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             index = VGG16_predict(img_path)

             return index >= 151 and index <= 268 # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: - The percentage of the human images that include a detected, dog face: 0.0%. - The percentage of the dog images that include a detected, dog face: 94.0%.

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         print("The percentage of the human images that include a detected, dog face: {}%.".form
         print("The percentage of the dog images that include a detected, dog face: {}%.".format

The percentage of the human images that include a detected, dog face: 0.0%.
The percentage of the dog images that include a detected, dog face: 94.0%.
```

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 [0]: ### (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 [25]: 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/')
         valid_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')

         # VGG-16 Takes 224x224 images as input, so we resize all of them
         data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                              transforms.RandomHorizontalFlip(), # randomly flip
                                              transforms.RandomRotation(10),
                                              transforms.ToTensor(),
                                              transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

         train_data = datasets.ImageFolder(train_dir, transform=data_transform)
         valid_data = datasets.ImageFolder(valid_dir, transform=data_transform)
         test_data = datasets.ImageFolder(test_dir, transform=data_transform)
```

```
# print out some data stats
print('Num training images: ', len(train_data))
print('Num valid images: ', len(valid_data))
print('Num test images: ', len(test_data))
```

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

```
In [26]: # define dataloader parameters
         batch_size = 20
         num_workers=0

         # prepare data loaders
         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                    num_workers=num_workers, shuffle=True)
         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                                    num_workers=num_workers, shuffle=True)
         test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
```

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: - My code resizes the images to the scale of 224x224 because VGG-16 takes 224x224 images as input. - Yes, I flip and rotate the images randomly to let the program recognize dogs in different positions and angles of the images.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [20]: 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
                 # convolutional layer
                 self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
                 # max pooling layer
```

```
            self.pool = nn.MaxPool2d(2, 2)
            # linear layer
            self.fc1 = nn.Linear(128 * 28 * 28, 500)
            # linear layer
            self.fc2 = nn.Linear(500, 133)
            # dropout layer (p=0.2)
            self.dropout = nn.Dropout(0.2)

        def forward(self, x):
            ## Define forward behavior
            # add sequence of convolutional and max pooling layers
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.pool(F.relu(self.conv3(x)))
            # flatten image input
            x = x.view(-1, 128 * 28 * 28)
            # add dropout layer
            x = self.dropout(x)
            # add 1st hidden layer, with relu activation function
            x = F.relu(self.fc1(x))
            # add dropout layer
            x = self.dropout(x)
            # add 2nd hidden layer, with relu activation function
            x = self.fc2(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, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=100352, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

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

Answer: - The architecture has three convolutional layers, each of them followed by max pooling. - The initial depth of 3 is scaled up to 128 and the initial image size is scaled down to 28x28 after convolutional layers and max pools. - The dropout is set to 0.2 to pervent overfitting. - The output nodes is 133 because there are totally 133 different classes of dogs.

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 [0]: 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, momentum=0.9)
```

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 [22]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True # to prevent the error image file is truncated

         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

                     # clear the gradients of all optimized variables
```

```
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - 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
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            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 ({:.6f} --> {:.6f}).  Saving model ...'.fo
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model

loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
```

```
        'test': test_loader
    }

    # train the model
    model_scratch = train(20, 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: 4.845941         Validation Loss: 4.712556
Validation loss decreased (inf --> 4.712556).  Saving model ...
Epoch: 2        Training Loss: 4.658334         Validation Loss: 4.591525
Validation loss decreased (4.712556 --> 4.591525).  Saving model ...
Epoch: 3        Training Loss: 4.587118         Validation Loss: 4.542990
Validation loss decreased (4.591525 --> 4.542990).  Saving model ...
Epoch: 4        Training Loss: 4.517776         Validation Loss: 4.528581
Validation loss decreased (4.542990 --> 4.528581).  Saving model ...
Epoch: 5        Training Loss: 4.480244         Validation Loss: 4.460931
Validation loss decreased (4.528581 --> 4.460931).  Saving model ...
Epoch: 6        Training Loss: 4.409455         Validation Loss: 4.415310
Validation loss decreased (4.460931 --> 4.415310).  Saving model ...
Epoch: 7        Training Loss: 4.358947         Validation Loss: 4.332190
Validation loss decreased (4.415310 --> 4.332190).  Saving model ...
Epoch: 8        Training Loss: 4.323617         Validation Loss: 4.312619
Validation loss decreased (4.332190 --> 4.312619).  Saving model ...
Epoch: 9        Training Loss: 4.249129         Validation Loss: 4.361903
Epoch: 10        Training Loss: 4.225628         Validation Loss: 4.333164
Epoch: 11        Training Loss: 4.173563         Validation Loss: 4.238185
Validation loss decreased (4.312619 --> 4.238185).  Saving model ...
Epoch: 13        Training Loss: 4.069853         Validation Loss: 4.148793
Validation loss decreased (4.238185 --> 4.148793).  Saving model ...
Epoch: 14        Training Loss: 4.043428         Validation Loss: 4.167723
Epoch: 15        Training Loss: 3.997738         Validation Loss: 4.118558
Validation loss decreased (4.148793 --> 4.118558).  Saving model ...
Epoch: 16        Training Loss: 3.946996         Validation Loss: 4.003539
Validation loss decreased (4.118558 --> 4.003539).  Saving model ...
Epoch: 17        Training Loss: 3.903003         Validation Loss: 3.966622
Validation loss decreased (4.003539 --> 3.966622).  Saving model ...
Epoch: 18        Training Loss: 3.887954         Validation Loss: 4.109065
Epoch: 19        Training Loss: 3.889887         Validation Loss: 3.974712
Epoch: 20        Training Loss: 3.797853         Validation Loss: 4.057388
```

```
Out[22]: <All keys matched successfully>
```

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 [23]: 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: 4.047194


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 [0]: ## TODO: Specify data loaders
        loaders_transfer = loaders_scratch
```

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

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

         if use_cuda:
             model_transfer = model_transfer.cuda()

In [31]: # Freeze training for all "features" layers
         for param in model_transfer.features.parameters():
             param.requires_grad = False

In [32]: n_inputs = model_transfer.classifier[6].in_features # 4096

         last_layer = nn.Linear(n_inputs, 133) # (4096, 133)

         model_transfer.classifier[6] = last_layer

         # after completing your model, if GPU is available, move the model to GPU
         if use_cuda:
             model_transfer.cuda()

         print(model_transfer.classifier[6].out_features)
```

```
133
```

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: - VGGNet is great because it's simple and has great performance, winning the ImageNet competition. We keep all the convolutional layers, but replace the final fully-connected layer with output of 133 (total calsses of dog).

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 [35]: 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 [76]: # train the model
         model_transfer = train(70, loaders_transfer, model_transfer, optimizer_transfer, criter
```

```
Epoch: 1        Training Loss: 4.457830         Validation Loss: 3.756705
Validation loss decreased (inf --> 3.756705).  Saving model ...
Epoch: 2        Training Loss: 3.264982         Validation Loss: 2.570255
Validation loss decreased (3.756705 --> 2.570255).  Saving model ...
Epoch: 3        Training Loss: 2.394022         Validation Loss: 1.923572
Validation loss decreased (2.570255 --> 1.923572).  Saving model ...
Epoch: 4        Training Loss: 1.989586         Validation Loss: 1.593057
Validation loss decreased (1.923572 --> 1.593057).  Saving model ...
Epoch: 5        Training Loss: 1.747360         Validation Loss: 1.510814
Validation loss decreased (1.593057 --> 1.510814).  Saving model ...
Epoch: 6        Training Loss: 1.668579         Validation Loss: 1.384804
Validation loss decreased (1.510814 --> 1.384804).  Saving model ...
Epoch: 7        Training Loss: 1.520721         Validation Loss: 1.328442
Validation loss decreased (1.384804 --> 1.328442).  Saving model ...
Epoch: 8        Training Loss: 1.480053         Validation Loss: 1.343120
Epoch: 9        Training Loss: 1.412881         Validation Loss: 1.283560
Validation loss decreased (1.328442 --> 1.283560).  Saving model ...
Epoch: 10        Training Loss: 1.413544         Validation Loss: 1.236705
Validation loss decreased (1.283560 --> 1.236705).  Saving model ...
Epoch: 11        Training Loss: 1.365724         Validation Loss: 1.276508
Epoch: 12        Training Loss: 1.339597         Validation Loss: 1.281874
Epoch: 13        Training Loss: 1.279702         Validation Loss: 1.216250
Validation loss decreased (1.236705 --> 1.216250).  Saving model ...
Epoch: 14        Training Loss: 1.285909         Validation Loss: 1.173033
Validation loss decreased (1.216250 --> 1.173033).  Saving model ...
Epoch: 15        Training Loss: 1.234109         Validation Loss: 1.142331
Validation loss decreased (1.173033 --> 1.142331).  Saving model ...
Epoch: 16        Training Loss: 1.238246         Validation Loss: 1.140303
Validation loss decreased (1.142331 --> 1.140303).  Saving model ...
Epoch: 17        Training Loss: 1.212854         Validation Loss: 1.217446
Epoch: 18        Training Loss: 1.202437         Validation Loss: 1.167387
Epoch: 19        Training Loss: 1.182448         Validation Loss: 1.200487
Epoch: 20        Training Loss: 1.147347         Validation Loss: 1.105770
```

```
Validation loss decreased (1.140303 --> 1.105770).  Saving model ...
Epoch: 21         Training Loss: 1.176210         Validation Loss: 1.148812
Epoch: 22         Training Loss: 1.131747         Validation Loss: 1.066216
Validation loss decreased (1.105770 --> 1.066216).  Saving model ...
Epoch: 23         Training Loss: 1.123849         Validation Loss: 1.062447
Validation loss decreased (1.066216 --> 1.062447).  Saving model ...
Epoch: 24         Training Loss: 1.146539         Validation Loss: 1.112465
Epoch: 25         Training Loss: 1.106221         Validation Loss: 1.153991
Epoch: 26         Training Loss: 1.079287         Validation Loss: 1.160262
Epoch: 27         Training Loss: 1.105784         Validation Loss: 1.138088
Epoch: 28         Training Loss: 1.087885         Validation Loss: 1.163047
Epoch: 29         Training Loss: 1.065370         Validation Loss: 1.060704
Validation loss decreased (1.062447 --> 1.060704).  Saving model ...
Epoch: 30         Training Loss: 1.068958         Validation Loss: 1.102930
Epoch: 31         Training Loss: 1.059951         Validation Loss: 1.115527
Epoch: 32         Training Loss: 1.041136         Validation Loss: 1.002694
Validation loss decreased (1.060704 --> 1.002694).  Saving model ...
Epoch: 33         Training Loss: 1.049507         Validation Loss: 1.093351
Epoch: 34         Training Loss: 1.022958         Validation Loss: 1.126156
Epoch: 35         Training Loss: 1.043649         Validation Loss: 1.079873
Epoch: 36         Training Loss: 1.018728         Validation Loss: 1.044626
Epoch: 37         Training Loss: 0.984133         Validation Loss: 1.070367
Epoch: 38         Training Loss: 1.015402         Validation Loss: 1.117371
Epoch: 39         Training Loss: 0.989166         Validation Loss: 1.067131
Epoch: 40         Training Loss: 0.988948         Validation Loss: 1.081849
Epoch: 41         Training Loss: 1.000261         Validation Loss: 1.189765
Epoch: 42         Training Loss: 0.993784         Validation Loss: 1.107110
Epoch: 43         Training Loss: 0.996839         Validation Loss: 1.065609
Epoch: 44         Training Loss: 1.003506         Validation Loss: 1.118948
Epoch: 45         Training Loss: 0.984958         Validation Loss: 1.176740
Epoch: 46         Training Loss: 0.975827         Validation Loss: 1.082198
Epoch: 47         Training Loss: 0.985061         Validation Loss: 1.116801
Epoch: 48         Training Loss: 0.959913         Validation Loss: 1.033677
Epoch: 49         Training Loss: 0.959081         Validation Loss: 1.176982
Epoch: 50         Training Loss: 0.968246         Validation Loss: 0.924098
Validation loss decreased (1.002694 --> 0.924098).  Saving model ...
Epoch: 51         Training Loss: 0.952530         Validation Loss: 1.058980
Epoch: 52         Training Loss: 0.958747         Validation Loss: 1.095847
Epoch: 53         Training Loss: 0.924452         Validation Loss: 1.080862
Epoch: 54         Training Loss: 0.960513         Validation Loss: 1.028885
Epoch: 55         Training Loss: 0.940214         Validation Loss: 1.070900
Epoch: 56         Training Loss: 0.916038         Validation Loss: 1.002515
Epoch: 57         Training Loss: 0.945493         Validation Loss: 1.073759
Epoch: 58         Training Loss: 0.912153         Validation Loss: 1.098408
Epoch: 59         Training Loss: 0.924112         Validation Loss: 1.057835
Epoch: 60         Training Loss: 0.911634         Validation Loss: 1.045067
Epoch: 61         Training Loss: 0.922258         Validation Loss: 1.005181
Epoch: 62         Training Loss: 0.917000         Validation Loss: 1.088855
```

```
Epoch: 63       Training Loss: 0.895475       Validation Loss: 1.094395
Epoch: 64       Training Loss: 0.902495       Validation Loss: 1.001868
Epoch: 65       Training Loss: 0.909647       Validation Loss: 0.994231
Epoch: 66       Training Loss: 0.881228       Validation Loss: 1.002413
Epoch: 67       Training Loss: 0.900459       Validation Loss: 1.004029
Epoch: 68       Training Loss: 0.907303       Validation Loss: 1.075352
Epoch: 69       Training Loss: 0.893335       Validation Loss: 1.112376
Epoch: 70       Training Loss: 0.915204       Validation Loss: 1.014869
```

```
In [39]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

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

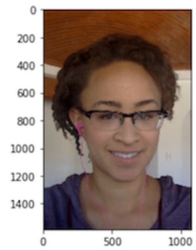
```
Test Loss: 1.003106


Test Accuracy: 71% (599/836)
```

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 [40]: ### 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 train_data.classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img = Image.open(img_path)

             transform = transforms.Compose([transforms.Resize(255),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor()])

             img_tensor = transform(img)
             img_tensor.unsqueeze_(0)

             if use_cuda:
```

```
hello, human!
```




```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

```
        img_tensor = img_tensor.cuda()

    prediction = model_transfer(img_tensor)

    if use_cuda:
        prediction = prediction.cpu()

    return prediction.data.numpy().argmax()
```

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 [43]: ### 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
             face_detectd = face_detector(img_path)
             dog_detected = dog_detector(img_path)
             pred = predict_breed_transfer(img_path)

             if dog_detected and not face_detectd:
                 title = "Hello, dog\nYour predicted breed is...\n{}!".format(class_names[pred])
             elif not dog_detected and face_detectd:
                 title = "Hello, human\nYou look like a {}!".format(class_names[pred])
```

```
    elif dog_detected and face_detectd:
        title = "Hello human, that's a nice {} you've got there!".format(class_names[pr
    else:
        title = "I can't find any dogs or humans here..."

    img = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)
    plt.imshow(img)
    plt.gca().set_title(title)
    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?

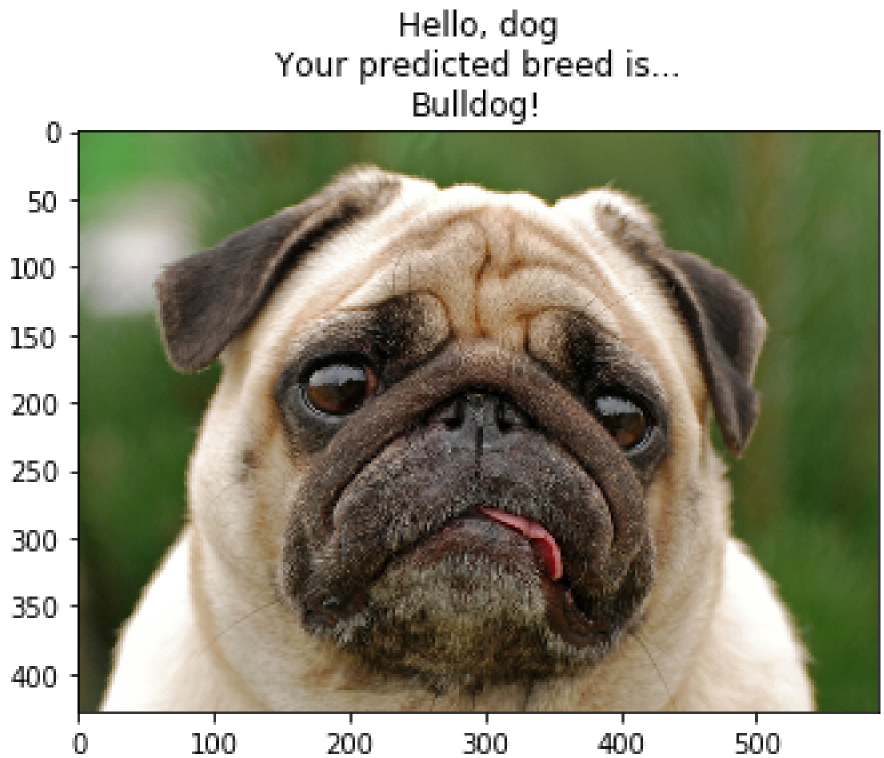
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

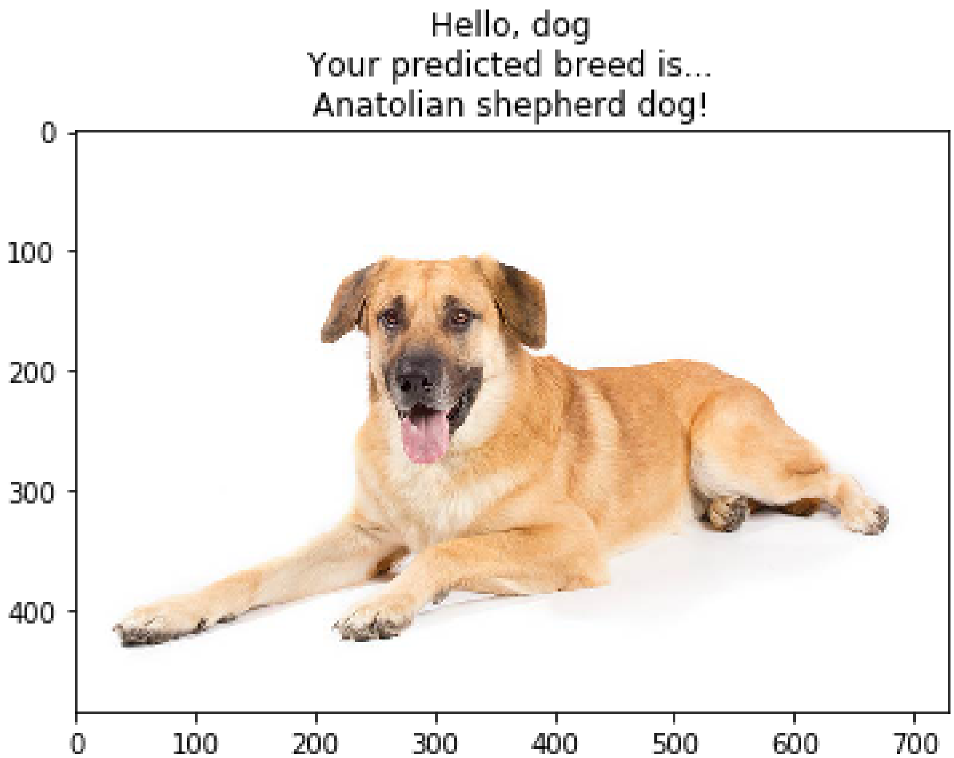
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement) - More dog images for each class - The model trained with more epochs and maybe with a smarter learning rate might help. - Increase the numbers of the output nodes of the convolutional layers.

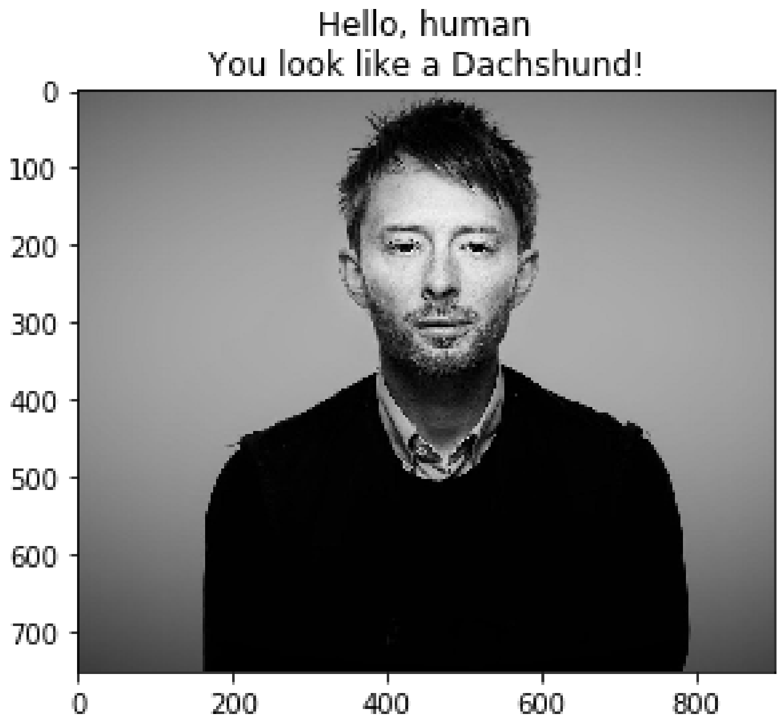
```
In [72]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         for file in np.array(glob("customized_files/*")):
             run_app(file)
```


Hello, dog
Your predicted breed is...
Bulldog!
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500


I can't find any dogs or humans here...
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600


Hello, dog
Your predicted breed is...
Anatolian shepherd dog!


Hello human, that's a nice American eskimo dog you've got there!


Hello, human
You look like a Dachshund!
0
100
200
300
400
500
600
700
0
200
400
600
800


Hello, human
You look like a Xoloitzcuintli!
0
200
400
600
800
1000
1200
0
250
500
750
1000
1250
1500
1750

Image Credits: - https://www.nme.com/news/music/mac-demarco-record-label-solo-tour-2364518 - https://www.thefamouspeople.com/profiles/thomas-edward-yorke-3003.php - https://www.npr.org/sections/thetwo-way/2011/12/08/143346081/pusuke-worlds-oldest-dog-dies-at-age-26-or-125-in-human-years - https://www.akc.org/dog-breeds/chinook/ - https://www.petmd.com/dog/conditions/digestive/c_dg_diarrhea_acute - https://www.livescience.com/55223-capybara-facts.html