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: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- 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`.

```
[311]: import numpy as np
from glob import glob

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

# 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

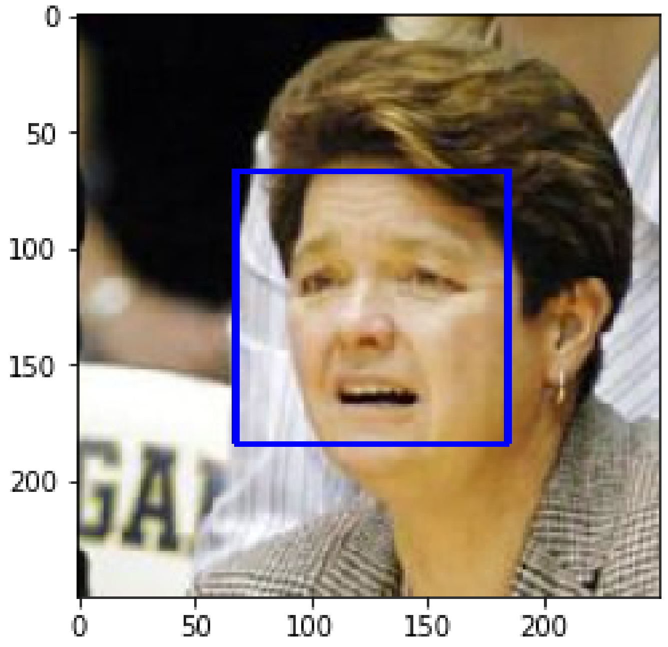
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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.

```
[312]: 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[100])
# 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.

[313]:
```
# 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)

[301]:
```
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_detected = 0
dog_detected = 0
total_files = len(dog_files_short)

for i in range(total_files):
    if face_detector(dog_files_short[i]):
        dog_detected += 1
    if face_detector(human_files_short[i]):
        human_detected += 1

print(f'Human Detected    {((human_detected) * 100) / total_humans:.6f}%')

print(f'Dog Detected      {((dog_detected) * 100) / total_dogs:.6f}%')
```

```
Human Detected    99.000000%
Dog Detected      6.000000%
```

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`.

[233]:
```
### (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.

[314]:
```
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.

[315]:
```
from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
```

```
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
    img = Image.open(img_path)
    transform = transforms.Compose([
        transforms.CenterCrop(256),
        transforms.ToTensor(),
    ])
    img = transform(img)
    img = torch.unsqueeze(img, 0)
    prediction = VGG16(img)
    _, prediction = torch.max(prediction, 1)
    prediction = np.squeeze(prediction.cpu().numpy())

    return prediction # 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).

[316]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    prediction = VGG16_predict(img_path)
    return prediction in range(151, 269) # 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:

[302]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_detected = 0
dog_detected = 0
total_files = len(dog_files_short)

for i in range(total_files):
    if dog_detector(human_files_short[i]):
        human_detected += 1
    if dog_detector(dog_files_short[i]):
        dog_detected += 1

print(f'Human Detected    {((human_detected) * 100) / total_files:.6f}%')

print(f'Dog Detected      {((dog_detected) * 100) / total_files:.6f}%')
```

```
Human Detected    0.000000%
Dog Detected      88.000000%
```

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`.

[ ]:
```
### (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 `dogImages/train`, `dogImages/valid`, and `dogImages/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!

[317]:
```python
import os
import numpy as np
import torch

import torchvision
from torchvision import datasets, models, transforms

import matplotlib.pyplot as plt

%matplotlib inline

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

```

```
batch_size = 20

transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(20),
        transforms.Resize(size=(224,224)),
        transforms.CenterCrop(size=224),
        transforms.ColorJitter(),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ]),
    'val': transforms.Compose([
        transforms.Resize(size=(224,224)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ]),
}

train_data = datasets.ImageFolder('dogImages/train',␣
 ↪transform=transforms['train'])
valid_data = datasets.ImageFolder('dogImages/valid',␣
 ↪transform=transforms['val'])
test_data = datasets.ImageFolder('dogImages/test', transform=transforms['val'])

train_loader = torch.utils.data.DataLoader(
    dataset=train_data, batch_size=batch_size, shuffle=True
)

valid_loader = torch.utils.data.DataLoader(
    dataset=valid_data, batch_size=batch_size
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_data, batch_size=batch_size
)
```

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:

1. I used a 224 resize and I chose this such that the image is clear enough for dog features to be well detected. My choice was also based on the ImageNet standards
2. Yes, for my training dataset, I used a RandomResizeCrop of size 224 with a RandomHori-

zontalFlip and a RandomRotation of 20degrees

1.1.8 (IMPLEMENTATION) Model Architecture

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

[318]:
```
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, 16, 5)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 5)
        self.dropout = nn.Dropout(0.2)
        self.fc1 = nn.Linear(32*53*53, 256)
        self.fc2 = nn.Linear(256, 84)
        self.fc3 = nn.Linear(84, 133)
        self.softmax = nn.LogSoftmax(dim=1)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.dropout(x)
        x = x.view(-1, 32 * 53 * 53)
        x = F.relu(self.fc1(x))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.softmax(self.fc3(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
use_cuda = False
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:

- Started off by using architectures used in previous lectures as blueprint. I tuned the convolutional layers based on the performance of my model.
- I had a couple of trial and errors for setting an appropriate linear layer architecture. I noticed adding more layers caused overfitting and cramming as my training loss was decreasing while the validation loss was increasing
- I used softmax for my final activation to likelihood probabilities of each dog breed class

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.

[319]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.003, 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'`.

[122]:
```
# 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
        #i = 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_loss))
            # clear the gradients of all optimized variables
            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))
        #print(f'train - output: {output}    target: {target}')
        ######################
        # 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(f'valid - output: {output}    target: {target}')
        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.
↪format(
            epoch,
            train_loss,
            valid_loss
        ))

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

```
    # return trained model
    return model



loaders_scratch = {
    'train': train_loader, 'valid': valid_loader, 'test': test_loader
}

# train the model
model_scratch = train(100, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1        Training Loss: 4.887654         Validation Loss: 4.881164
Validation loss decreased (inf --> 4.881164).  Saving model …
Epoch: 2        Training Loss: 4.876915         Validation Loss: 4.858784
Validation loss decreased (4.881164 --> 4.858784).  Saving model …
Epoch: 3        Training Loss: 4.850556         Validation Loss: 4.809115
Validation loss decreased (4.858784 --> 4.809115).  Saving model …
Epoch: 4        Training Loss: 4.796139         Validation Loss: 4.733317
Validation loss decreased (4.809115 --> 4.733317).  Saving model …
Epoch: 5        Training Loss: 4.734840         Validation Loss: 4.613291
Validation loss decreased (4.733317 --> 4.613291).  Saving model …
Epoch: 6        Training Loss: 4.644067         Validation Loss: 4.525655
Validation loss decreased (4.613291 --> 4.525655).  Saving model …
Epoch: 7        Training Loss: 4.611328         Validation Loss: 4.486628
Validation loss decreased (4.525655 --> 4.486628).  Saving model …
Epoch: 8        Training Loss: 4.572527         Validation Loss: 4.454288
Validation loss decreased (4.486628 --> 4.454288).  Saving model …
Epoch: 9        Training Loss: 4.563535         Validation Loss: 4.485192
Epoch: 10       Training Loss: 4.534442         Validation Loss: 4.425680
Validation loss decreased (4.454288 --> 4.425680).  Saving model …
Epoch: 11       Training Loss: 4.516336         Validation Loss: 4.391881
Validation loss decreased (4.425680 --> 4.391881).  Saving model …
Epoch: 12       Training Loss: 4.508131         Validation Loss: 4.404052
Epoch: 13       Training Loss: 4.483377         Validation Loss: 4.403751
Epoch: 14       Training Loss: 4.475268         Validation Loss: 4.371981
Validation loss decreased (4.391881 --> 4.371981).  Saving model …
Epoch: 15       Training Loss: 4.449717         Validation Loss: 4.420759
Epoch: 16       Training Loss: 4.420536         Validation Loss: 4.272496
Validation loss decreased (4.371981 --> 4.272496).  Saving model …
Epoch: 17       Training Loss: 4.430001         Validation Loss: 4.337231
Epoch: 18       Training Loss: 4.387584         Validation Loss: 4.309460
```

```
Epoch: 19       Training Loss: 4.389237         Validation Loss: 4.272539
Epoch: 20       Training Loss: 4.366737         Validation Loss: 4.225631
Validation loss decreased (4.272496 --> 4.225631).  Saving model …
Epoch: 21       Training Loss: 4.343097         Validation Loss: 4.268722
Epoch: 22       Training Loss: 4.326871         Validation Loss: 4.224926
Validation loss decreased (4.225631 --> 4.224926).  Saving model …
Epoch: 23       Training Loss: 4.313392         Validation Loss: 4.274879
Epoch: 24       Training Loss: 4.290093         Validation Loss: 4.239108
Epoch: 25       Training Loss: 4.287838         Validation Loss: 4.221360
Validation loss decreased (4.224926 --> 4.221360).  Saving model …
Epoch: 26       Training Loss: 4.231579         Validation Loss: 4.195645
Validation loss decreased (4.221360 --> 4.195645).  Saving model …
Epoch: 27       Training Loss: 4.217978         Validation Loss: 4.167116
Validation loss decreased (4.195645 --> 4.167116).  Saving model …
Epoch: 28       Training Loss: 4.208602         Validation Loss: 4.123720
Validation loss decreased (4.167116 --> 4.123720).  Saving model …
Epoch: 29       Training Loss: 4.225681         Validation Loss: 4.253713
Epoch: 30       Training Loss: 4.181579         Validation Loss: 4.173272
Epoch: 31       Training Loss: 4.195097         Validation Loss: 4.116863
Validation loss decreased (4.123720 --> 4.116863).  Saving model …
Epoch: 32       Training Loss: 4.151564         Validation Loss: 4.126726
Epoch: 33       Training Loss: 4.131686         Validation Loss: 4.198118
Epoch: 34       Training Loss: 4.130297         Validation Loss: 4.084854
Validation loss decreased (4.116863 --> 4.084854).  Saving model …
Epoch: 35       Training Loss: 4.123418         Validation Loss: 4.082998
Validation loss decreased (4.084854 --> 4.082998).  Saving model …
Epoch: 36       Training Loss: 4.100706         Validation Loss: 4.010163
Validation loss decreased (4.082998 --> 4.010163).  Saving model …
Epoch: 37       Training Loss: 4.070792         Validation Loss: 4.069144
Epoch: 38       Training Loss: 4.077219         Validation Loss: 4.080659
Epoch: 39       Training Loss: 4.051897         Validation Loss: 4.144297
Epoch: 40       Training Loss: 4.039759         Validation Loss: 4.032405
Epoch: 41       Training Loss: 4.025325         Validation Loss: 4.022763
Epoch: 42       Training Loss: 4.030695         Validation Loss: 4.083611
Epoch: 43       Training Loss: 4.014375         Validation Loss: 3.973122
Validation loss decreased (4.010163 --> 3.973122).  Saving model …
Epoch: 44       Training Loss: 3.997461         Validation Loss: 4.014820
Epoch: 45       Training Loss: 3.996006         Validation Loss: 4.035382
Epoch: 46       Training Loss: 4.012661         Validation Loss: 4.018641
Epoch: 47       Training Loss: 3.977873         Validation Loss: 4.061254
Epoch: 48       Training Loss: 3.949306         Validation Loss: 3.993841
Epoch: 49       Training Loss: 3.946636         Validation Loss: 4.067809
Epoch: 50       Training Loss: 3.933068         Validation Loss: 4.005712
Epoch: 51       Training Loss: 3.928800         Validation Loss: 4.030672
Epoch: 52       Training Loss: 3.897200         Validation Loss: 3.968748
Validation loss decreased (3.973122 --> 3.968748).  Saving model …
Epoch: 53       Training Loss: 3.900960         Validation Loss: 3.947146
Validation loss decreased (3.968748 --> 3.947146).  Saving model …
```

```
Epoch: 54       Training Loss: 3.875229         Validation Loss: 4.013808
Epoch: 55       Training Loss: 3.894972         Validation Loss: 3.921855
Validation loss decreased (3.947146 --> 3.921855).  Saving model …
Epoch: 56       Training Loss: 3.865209         Validation Loss: 4.011038
Epoch: 57       Training Loss: 3.867589         Validation Loss: 4.030294
Epoch: 58       Training Loss: 3.859959         Validation Loss: 4.125837
Epoch: 59       Training Loss: 3.859476         Validation Loss: 4.116861
Epoch: 60       Training Loss: 3.829945         Validation Loss: 3.986689
Epoch: 61       Training Loss: 3.844543         Validation Loss: 3.954662
Epoch: 62       Training Loss: 3.834918         Validation Loss: 3.880362
Validation loss decreased (3.921855 --> 3.880362).  Saving model …
Epoch: 63       Training Loss: 3.792737         Validation Loss: 3.931297
Epoch: 64       Training Loss: 3.805804         Validation Loss: 3.924830
Epoch: 65       Training Loss: 3.773263         Validation Loss: 3.933840
Epoch: 66       Training Loss: 3.763914         Validation Loss: 3.986971
Epoch: 67       Training Loss: 3.778186         Validation Loss: 3.982902
Epoch: 68       Training Loss: 3.790726         Validation Loss: 3.920581
Epoch: 69       Training Loss: 3.774585         Validation Loss: 3.977839
Epoch: 70       Training Loss: 3.759367         Validation Loss: 3.951402
Epoch: 71       Training Loss: 3.756902         Validation Loss: 3.920926
Epoch: 72       Training Loss: 3.746154         Validation Loss: 4.030005
Epoch: 73       Training Loss: 3.728976         Validation Loss: 4.017392
Epoch: 74       Training Loss: 3.723669         Validation Loss: 3.964202
Epoch: 75       Training Loss: 3.723467         Validation Loss: 3.932775
Epoch: 76       Training Loss: 3.699989         Validation Loss: 3.920182
Epoch: 77       Training Loss: 3.668531         Validation Loss: 3.929796
Epoch: 78       Training Loss: 3.725981         Validation Loss: 3.940018
Epoch: 79       Training Loss: 3.711438         Validation Loss: 3.991723
Epoch: 80       Training Loss: 3.667674         Validation Loss: 3.988170
Epoch: 81       Training Loss: 3.682300         Validation Loss: 3.897781
Epoch: 82       Training Loss: 3.678222         Validation Loss: 3.957871
Epoch: 83       Training Loss: 3.640646         Validation Loss: 3.974771
Epoch: 84       Training Loss: 3.652612         Validation Loss: 3.844697
Validation loss decreased (3.880362 --> 3.844697).  Saving model …
Epoch: 85       Training Loss: 3.657524         Validation Loss: 3.981571
Epoch: 86       Training Loss: 3.655191         Validation Loss: 3.849064
Epoch: 87       Training Loss: 3.642025         Validation Loss: 3.962432
Epoch: 88       Training Loss: 3.637341         Validation Loss: 3.850223
Epoch: 89       Training Loss: 3.634640         Validation Loss: 3.958183
Epoch: 90       Training Loss: 3.641489         Validation Loss: 3.817956
Validation loss decreased (3.844697 --> 3.817956).  Saving model …
Epoch: 91       Training Loss: 3.597867         Validation Loss: 3.984128
Epoch: 92       Training Loss: 3.636848         Validation Loss: 3.933499
Epoch: 93       Training Loss: 3.592646         Validation Loss: 4.023183
Epoch: 94       Training Loss: 3.591207         Validation Loss: 3.926426
Epoch: 95       Training Loss: 3.576216         Validation Loss: 3.849892
Epoch: 96       Training Loss: 3.582966         Validation Loss: 3.887123
Epoch: 97       Training Loss: 3.572348         Validation Loss: 4.065247
```

```
Epoch: 98       Training Loss: 3.560251         Validation Loss: 3.883852
Epoch: 99       Training Loss: 3.594180         Validation Loss: 3.929646
Epoch: 100      Training Loss: 3.566508         Validation Loss: 3.913562
```

```
[122]: <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%.

```
[335]: 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
       model_scratch.load_state_dict(torch.load('model_scratch.pt'))
       test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.834505
Test Accuracy: 12% (103/836)
```

## 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.

[337]:
```python
## TODO: Specify data loaders

import os
import numpy as np
import torch

import torchvision
from torchvision import datasets, models, transforms

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

batch_size = 20

transfer_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(20),
        transforms.Resize(size=(224,224)),
        transforms.CenterCrop(size=224),
        transforms.ColorJitter(),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ]),
    'val': transforms.Compose([
        transforms.Resize(size=(224,224)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ]),
}
```

```
train_data = datasets.ImageFolder('dogImages/train',␣
↪transform=transfer_transforms['train'])
valid_data = datasets.ImageFolder('dogImages/valid',␣
↪transform=transfer_transforms['val'])
test_data = datasets.ImageFolder('dogImages/test',␣
↪transform=transfer_transforms['val'])

train_loader = torch.utils.data.DataLoader(
    dataset=train_data, batch_size=batch_size, shuffle=True
)

valid_loader = torch.utils.data.DataLoader(
    dataset=valid_data, batch_size=batch_size
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_data, batch_size=batch_size
)

data_transfer = {'train': train_data, 'valid': valid_data, 'test': test_data}

loaders_transfer = {'train': train_loader, 'valid': valid_loader, 'test':␣
↪test_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`.

[338]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

model_transfer = VGG16

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

n_inputs = model_transfer.classifier[6].in_features
n_outputs = model_transfer.classifier[6].out_features

inputs, classes = next(iter(test_loader))
last_layer = nn.Linear(n_inputs, 133)

model_transfer.classifier[6] = last_layer
```

```
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:

- My transfer learning architecture was quite simplistic, I just changed the number of output features of the VGG16 to match the classes of dog breeds I have.
- This was based on the fact the idea that the VGG16 was doiing quites well on detecting human and dogs, so I basically just had to mod it to be able to classify my dog breeds

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.

[324]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.parameters(), lr=0.003,␣
 ↪momentum=0.9)
```

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'`.

[138]:
```
# train the model
model_transfer = train(10, loaders_transfer, model_transfer,␣
 ↪optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')

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

```
Epoch: 1        Training Loss: 2.588766         Validation Loss: 1.194412
Validation loss decreased (inf --> 1.194412).  Saving model …
Epoch: 2        Training Loss: 1.940535         Validation Loss: 1.027533
Validation loss decreased (1.194412 --> 1.027533).  Saving model …
Epoch: 3        Training Loss: 1.858806         Validation Loss: 1.194097
Epoch: 4        Training Loss: 1.843201         Validation Loss: 0.852349
Validation loss decreased (1.027533 --> 0.852349).  Saving model …
Epoch: 5        Training Loss: 1.753623         Validation Loss: 1.008644
Epoch: 6        Training Loss: 1.755470         Validation Loss: 1.033488
Epoch: 7        Training Loss: 1.737064         Validation Loss: 0.944934
Epoch: 8        Training Loss: 1.723327         Validation Loss: 0.933020
```

```
Epoch: 9        Training Loss: 1.753200         Validation Loss: 0.924208
Epoch: 10       Training Loss: 1.704314         Validation Loss: 0.965395
```

[325]:
```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

[325]:
```
<All keys matched successfully>
```

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%.

[339]:
```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.927759


Test Accuracy: 74% (619/836)
```

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.

[210]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

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

from torchvision import datasets, models, transforms
def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = Image.open(img_path)
    transform = transforms.Compose([
        transforms.Resize(size=(224,224)),
        transforms.ToTensor(),
    ])

    img = transform(img)
    img = torch.unsqueeze(img, 0)
    output = model_transfer(img)
    _, pred = torch.max(output, 1)
    pred_index = np.squeeze(pred.numpy()) if not use_cuda else np.squeeze(pred.
 ↪cpu().numpy())
```

```python
    return class_names[pred_index] # predicted class
```

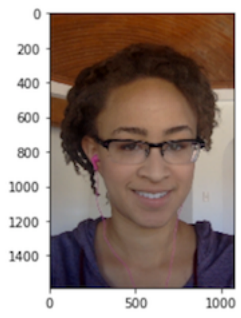
## Step 5: Write your Algorithm

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 `dog_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!

```
hello, human!
```



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

1.1.18 (IMPLEMENTATION) Write your Algorithm

[240]:
```python
### 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
    identity = "Hello Human!"

    if dog_detector(img_path):
        identity = "Hey Doggy!"

    print(identity)
    class_name = predict_breed_transfer(img_path)

    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(img)
    print(f"You look like a ...\n{class_name}\n\n")
```

## 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.

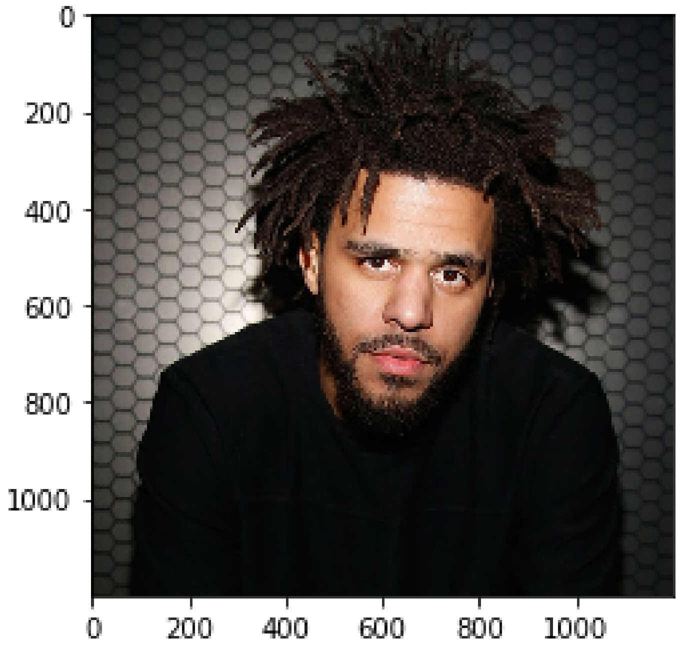
Answer: (Three possible points for improvement)

[242]:
```
## 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)
run_app(human_files[5])
```

```
Hello Human!
You look like a …
Dogue de bordeaux
```

[284]:
```
test_files = human_files = np.array(glob("test_me/*"))
run_app(human_files[0])
```

```
Hello Human!
You look like a …
Bearded collie
```



```
[ ]:
```