dog_app

April 13, 2020

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 [24]: 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

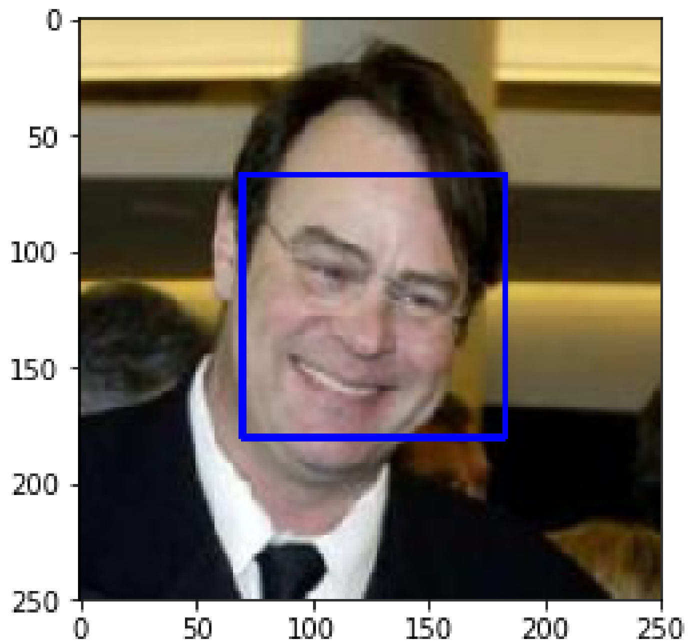
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [25]: 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 [26]: # 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 [4]: from tqdm import tqdm

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        human = 0
        dog = 0
        for i in range(0, len(human_files_short)):
            if face_detector(human_files_short[i]):
                human += 1
            if face_detector(dog_files_short[i]):
                dog += 1

        print(human/100)     #98%
        print(dog/100)       #17%
```

```
0.98
0.17
```

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

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

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [27]: 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 [28]: 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)
    img = transforms.Resize([224,224])(img)
    img = transforms.ToTensor()(img)
    img = transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))(img)
    img = img.unsqueeze(0)

    prediction = (VGG16(img.cuda())).cpu().data.numpy().argmax()

    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).

```
In [29]: ### 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)

             if (prediction >= 151 and prediction <= 268):
                 dog = True
             else:
                 dog = False
             return dog # 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:
10% humans were detected as dogs
100% dogs were detected as dogs

In [8]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

human = 0
dog = 0
for i in range(0, len(human_files_short)):
    if dog_detector(human_files_short[i]):
        human += 1
    if dog_detector(dog_files_short[i]):
        dog += 1

print(human/100)     #10%
print(dog/100)       #100%
```

```
0.01
1.0
```

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

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [4]: import os
        from torchvision import datasets
        from PIL import Image
        import torchvision.transforms as transforms
        import torch
        import torchvision.models as models
        import numpy as np
        from glob import glob

        use_cuda = torch.cuda.is_available()

        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        data_transform = transforms.Compose([transforms.Resize([28,28]), transforms.RandomHorizo
                                             transforms.RandomRotation(15), transforms.ToTensor(
                                             transforms.Normalize((0.485, 0.456, 0.406), (0.229,
```

```
batch_size = 64
num_workers = 0
shuffle = True

train_images = datasets.ImageFolder('/data/dog_images/train', transform=data_transform)
valid_images = datasets.ImageFolder('/data/dog_images/valid', transform=data_transform)
test_images = datasets.ImageFolder('/data/dog_images/test', transform=data_transform)

training_loader = torch.utils.data.DataLoader(train_images, batch_size=batch_size, num_w
validation_loader = torch.utils.data.DataLoader(valid_images, batch_size=batch_size, num
test_loader = torch.utils.data.DataLoader(test_images, batch_size=batch_size, num_worker
```

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

Answer: I decided to resize all the images to 28x28, turn them into tensors, and then regularize the tensor. I picked 28x28 for the image because I felt it would prevent overfitting to the training set and make for faster training.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [2]: 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, 4, 2)
                self.dropout = nn.Dropout(0.2)
                self.maxpooling = nn.MaxPool2d(2, 2)

                self.fcn1 = nn.Linear(16*6*6, 133)
                self.fcn2 = nn.Linear(133, 133)



            def forward(self, x):
                ## Define forward behavior
                x = F.relu(self.conv1(x))
                x = self.maxpooling(x)
                #print('Pooling: ' + str(x.shape))
```

```
        x = x.view(-1, 16*6*6)
        x = F.relu(self.fcn1(x))
        x = self.dropout(x)
        x = self.fcn2(x)
        #print('FCN2: ' + str(x.shape))
        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()

model_scratch

#dataiter = iter(training_loader)
#images, labels = dataiter.next()
```

```
Out[2]: Net(
  (conv1): Conv2d(3, 16, kernel_size=(4, 4), stride=(2, 2))
  (dropout): Dropout(p=0.2)
  (maxpooling): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
  (fcn1): Linear(in_features=576, out_features=133, bias=True)
  (fcn2): Linear(in_features=133, out_features=133, bias=True)
)
```

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

Answer:

Initially, I looked into some papers to determine the overall architecture of my model. However, I found that the complexity of those models suited much larger classification tasks and did not perform particularly well in validation. I reduced my model to a single convolutional layer outputting to one dense layer as a result. I still was not quite happy with the training rate, so I added one final dense layer with a dropout layer just before it. This really seemed to improve the training time and led to acceptable accuracy on the test set.

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 [3]: import torch.optim as optim

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

```
### TODO: select optimizer
optimizer_scratch = optim.Adam(model_scratch.parameters())
```

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 [4]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
            """returns trained model"""
            from PIL import ImageFile
            ImageFile.LOAD_TRUNCATED_IMAGES = True
            # 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[0]):
                    # move to GPU
                    if use_cuda:
                        data, target = data.cuda(), target.cuda()
                    ## find the loss and update the model parameters accordingly
                    optimizer.zero_grad()
                    output = model(data)
                    #print(output.shape, target.shape)
                    loss = criterion(output, target)
                    loss.backward()
                    optimizer.step()
                    ## record the average training loss, using something like
                    train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

                ######################
                # validate the model #
                ######################
                model.eval()
                for batch_idx, (data, target) in enumerate(loaders[1]):
                    # move to GPU
                    if use_cuda:
                        data, target = data.cuda(), target.cuda()
                    ## update the average validation loss
                    output = model(data)
```

```
                #print(output)
                loss = criterion(output, target)
                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 ...'.for
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), 'model_scratch.pt')
                valid_loss_min = valid_loss
        # return trained model
        return model


    # train the model
    #model_scratch = train(30, [training_loader, validation_loader], model_scratch, optimize
                          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: 4.797479         Validation Loss: 4.604949
Validation loss decreased (inf --> 4.604949).  Saving model ...
Epoch: 2         Training Loss: 4.488509         Validation Loss: 4.409910
Validation loss decreased (4.604949 --> 4.409910).  Saving model ...
Epoch: 3         Training Loss: 4.316478         Validation Loss: 4.302009
Validation loss decreased (4.409910 --> 4.302009).  Saving model ...
Epoch: 4         Training Loss: 4.220667         Validation Loss: 4.181058
Validation loss decreased (4.302009 --> 4.181058).  Saving model ...
Epoch: 5         Training Loss: 4.141889         Validation Loss: 4.212178
Epoch: 6         Training Loss: 4.076708         Validation Loss: 4.118093
Validation loss decreased (4.181058 --> 4.118093).  Saving model ...
Epoch: 7         Training Loss: 4.012421         Validation Loss: 4.218410
Epoch: 8         Training Loss: 3.967971         Validation Loss: 4.137615
Epoch: 9         Training Loss: 3.923424         Validation Loss: 4.094709
Validation loss decreased (4.118093 --> 4.094709).  Saving model ...
Epoch: 10         Training Loss: 3.885367         Validation Loss: 4.122522
Epoch: 11         Training Loss: 3.846921         Validation Loss: 4.114660
Epoch: 12         Training Loss: 3.804878         Validation Loss: 4.095655
Epoch: 13         Training Loss: 3.773431         Validation Loss: 4.108228
```

```
Epoch: 14         Training Loss: 3.753692        Validation Loss: 4.086092
Validation loss decreased (4.094709 --> 4.086092).  Saving model ...
Epoch: 15         Training Loss: 3.706149        Validation Loss: 4.039195
Validation loss decreased (4.086092 --> 4.039195).  Saving model ...
Epoch: 16         Training Loss: 3.699472        Validation Loss: 4.011334
Validation loss decreased (4.039195 --> 4.011334).  Saving model ...
Epoch: 17         Training Loss: 3.660452        Validation Loss: 4.053428
Epoch: 18         Training Loss: 3.614194        Validation Loss: 4.109522
Epoch: 19         Training Loss: 3.595212        Validation Loss: 4.107480
Epoch: 20         Training Loss: 3.581849        Validation Loss: 4.270388
Epoch: 21         Training Loss: 3.560525        Validation Loss: 3.987398
Validation loss decreased (4.011334 --> 3.987398).  Saving model ...
Epoch: 22         Training Loss: 3.536822        Validation Loss: 4.030657
Epoch: 23         Training Loss: 3.500296        Validation Loss: 4.100770
Epoch: 24         Training Loss: 3.499625        Validation Loss: 4.070855
Epoch: 25         Training Loss: 3.477958        Validation Loss: 4.129795
Epoch: 26         Training Loss: 3.469475        Validation Loss: 4.142934
Epoch: 27         Training Loss: 3.445681        Validation Loss: 4.109590
Epoch: 28         Training Loss: 3.425028        Validation Loss: 4.154293
Epoch: 29         Training Loss: 3.409617        Validation Loss: 4.116538
Epoch: 30         Training Loss: 3.375910        Validation Loss: 4.071336
```

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 [5]: 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):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the loss
                loss = criterion(output, target)
                # update average test loss
                test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
                # convert output probabilities to predicted class
                pred = output.data.max(1, keepdim=True)[1]
```

```
            # compare predictions to true label
            correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
            total += data.size(0)

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

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

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

```
Test Loss: 4.119383


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 [31]: import os
         from torchvision import datasets
         from PIL import Image
         import torchvision.transforms as transforms
         import torch
         import torchvision.models as models
         import numpy as np
         from glob import glob

         ## TODO: Specify data loaders
         data_transform = transforms.Compose([transforms.Resize([224, 224]), transforms.ToTensor
         batch_size = 64
         num_workers = 0
         shuffle = True

         train_images = datasets.ImageFolder('/data/dog_images/train', transform=data_transform)
         valid_images = datasets.ImageFolder('/data/dog_images/valid', transform=data_transform)
```

```
test_images = datasets.ImageFolder('/data/dog_images/test', transform=data_transform)

training_loader = torch.utils.data.DataLoader(train_images, batch_size=batch_size, num_
validation_loader = torch.utils.data.DataLoader(valid_images, batch_size=batch_size, nu
test_loader = torch.utils.data.DataLoader(test_images, batch_size=batch_size, num_worke
```

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 [8]: import torchvision.models as models
        import torch.nn as nn
        # define transfer learning model
        model_transfer = models.vgg16(pretrained=True)

        use_cuda = torch.cuda.is_available()
        ## TODO: Specify model architecture

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

        model_transfer.classifier.add_module("7", nn.ReLU(inplace=True))
        model_transfer.classifier.add_module("8", nn.Dropout(p=0.5))
        model_transfer.classifier.add_module("9", nn.Linear(in_features=1000, out_features=133,

        model_transfer.classifier

        # move model to GPU if CUDA is available
        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:

Because the features VGG16 has learned will still be good for this application, I froze all of those parameters. I added a ReLU, dropout layer, and new linear layer which outputs only the 133 classes we have for this classification problem.

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 [9]: import torch.optim as optim

        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.Adagrad(model_transfer.classifier.parameters())
```

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 [10]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    from PIL import ImageFile
    ImageFile.LOAD_TRUNCATED_IMAGES = True
    # 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[0]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            optimizer.zero_grad()
            output = model(data)
            #print(output.shape, target.shape)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            ## record the average training loss, using something like
            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders[1]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            #print(output)
            loss = criterion(output, target)
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))
```

```python
                # 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
                    valid_loss_min,
                    valid_loss))
                    torch.save(model.state_dict(), 'model_transfer.pt')
                    valid_loss_min = valid_loss
            # return trained model
            return model

        # train the model
        model_transfer = train(30, [training_loader, validation_loader], model_transfer, optimi

        # 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: 12.278503        Validation Loss: 2.599044
Validation loss decreased (inf --> 2.599044).  Saving model ...
Epoch: 2         Training Loss: 2.982768         Validation Loss: 1.915139
Validation loss decreased (2.599044 --> 1.915139).  Saving model ...
Epoch: 3         Training Loss: 2.184315         Validation Loss: 1.337732
Validation loss decreased (1.915139 --> 1.337732).  Saving model ...
Epoch: 4         Training Loss: 1.717597         Validation Loss: 1.199559
Validation loss decreased (1.337732 --> 1.199559).  Saving model ...
Epoch: 5         Training Loss: 1.386944         Validation Loss: 0.989490
Validation loss decreased (1.199559 --> 0.989490).  Saving model ...
Epoch: 6         Training Loss: 1.147627         Validation Loss: 0.920078
Validation loss decreased (0.989490 --> 0.920078).  Saving model ...
Epoch: 7         Training Loss: 0.982636         Validation Loss: 0.930177
Epoch: 8         Training Loss: 0.806476         Validation Loss: 0.785264
Validation loss decreased (0.920078 --> 0.785264).  Saving model ...
Epoch: 9         Training Loss: 0.727540         Validation Loss: 0.790246
Epoch: 10         Training Loss: 0.640851         Validation Loss: 0.795220
Epoch: 11         Training Loss: 0.540405         Validation Loss: 0.783335
Validation loss decreased (0.785264 --> 0.783335).  Saving model ...
Epoch: 12         Training Loss: 0.503524         Validation Loss: 0.775461
Validation loss decreased (0.783335 --> 0.775461).  Saving model ...
Epoch: 13         Training Loss: 0.420975         Validation Loss: 0.725142
Validation loss decreased (0.775461 --> 0.725142).  Saving model ...
Epoch: 14         Training Loss: 0.389162         Validation Loss: 0.786161
Epoch: 15         Training Loss: 0.348599         Validation Loss: 0.772676
```

```
Epoch: 16         Training Loss: 0.349917         Validation Loss: 0.813009
Epoch: 17         Training Loss: 0.318054         Validation Loss: 0.848432
Epoch: 18         Training Loss: 0.288135         Validation Loss: 0.761203
Epoch: 19         Training Loss: 0.265573         Validation Loss: 1.141502
Epoch: 20         Training Loss: 0.221672         Validation Loss: 0.798070
Epoch: 21         Training Loss: 0.210125         Validation Loss: 0.788010
Epoch: 22         Training Loss: 0.185576         Validation Loss: 0.782990
Epoch: 23         Training Loss: 0.209464         Validation Loss: 0.793830
Epoch: 24         Training Loss: 0.206058         Validation Loss: 0.762055
Epoch: 25         Training Loss: 0.181928         Validation Loss: 0.808751
Epoch: 26         Training Loss: 0.167169         Validation Loss: 0.761140
Epoch: 27         Training Loss: 0.160262         Validation Loss: 0.778332
Epoch: 28         Training Loss: 0.165464         Validation Loss: 0.767587
Epoch: 29         Training Loss: 0.162914         Validation Loss: 0.789072
Epoch: 30         Training Loss: 0.142589         Validation Loss: 0.805282
```

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

    FileNotFoundError                         Traceback (most recent call last)

    <ipython-input-10-f01e643e65aa> in <module>()
     64 
     65 # load the model that got the best validation accuracy (uncomment the line below)
---> 66 model_transfer.load_state_dict(torch.load('model_transfer.pt'))


    /opt/conda/lib/python3.6/site-packages/torch/serialization.py in load(f, map_location, p
    299             (sys.version_info[0] == 3 and isinstance(f, pathlib.Path)):
    300         new_fd = True
--> 301         f = open(f, 'rb')
    302     try:
    303         return _load(f, map_location, pickle_module)


    FileNotFoundError: [Errno 2] No such file or directory: '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 [11]: 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):
            # 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))

    model_transfer.load_state_dict(torch.load('model_transfer.pt'))
    test(test_loader, model_transfer, criterion_transfer, use_cuda)
```

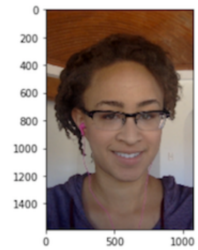
```
Test Loss: 0.815504


Test Accuracy: 76% (641/836)
```

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 [32]: ### 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_images.classes]

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

hello, human!



You look like a ...
Chinese_shar-pei

Sample Human Output

```
    img = transforms.Resize([224,224])(img)
    img = transforms.ToTensor()(img)
    img = transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))(img)
    img = img.unsqueeze(0)

    model_transfer = torch.load('model_transfer.pt')

    if use_cuda:
        model_transfer = model_transfer.cuda()

    prediction = (model_transfer(img.cuda())).cpu().data.numpy().argmax()
    return class_names[prediction]
```

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 [33]:
```
### 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
    #Check for human face
    if(face_detector(img_path)):
        print("Human face found! You look like a(n) " + str(predict_breed_transfer(img_
```

```
            #Check for dog face
            elif(dog_detector(img_path)):
                print("Dog found! It looks like it might be a(n) " + str(predict_breed_transfer
            
            #If we find neither, say so
            else:
                print("No face found :(")
```

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:

1. It might be cool if it could give multiple options for what breed it might be if several outcomes have similar probabi
2. It would be nice if the model showed a heatmap of where it is "seeing" the dog in the picture. What parts of the image are activating the classifier?
3. It would also be great if it would draw a box around the face it has detected. If there are multiple faces in an image, it would be exciting to label all of them rather than just picking one!

In [53]:
```
## 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.

human_files = np.array(glob("/home/workspace/dog_project/human_files/*"))
dog_files = np.array(glob("/home/workspace/dog_project/dog_files/*"))

## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```

```
Human face found! You look like a(n) Bloodhound
Human face found! You look like a(n) American staffordshire terrier
Human face found! You look like a(n) Chinese crested
```

```
Dog found! It looks like it might be a(n) Labrador retriever
Dog found! It looks like it might be a(n) Dandie dinmont terrier
Dog found! It looks like it might be a(n) Great pyrenees
```

In [43]:
```
human_files
```

Out[43]:
```
array(['/home/workspace/dog_project/human_files/3.jpeg',
       '/home/workspace/dog_project/human_files/1.jpeg',
       '/home/workspace/dog_project/human_files/2.jpeg'],
      dtype='<U46')
```

In [ ]: