dog_app

June 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`.

In [1]:
```
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

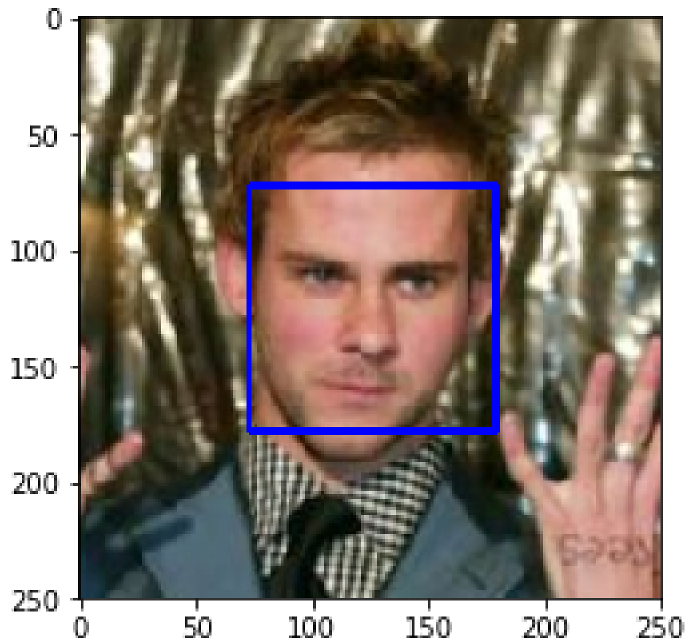
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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.

In [2]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# load color (BGR) image
img = cv2.imread(human_files[0])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
```

```
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
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. #-#-#

        human_matches = sum(map(lambda x: 1 if face_detector(x) else 0, human_files_short))
        dog_matches = sum(map(lambda x: 1 if face_detector(x) else 0, dog_files_short))

        print(human_matches, dog_matches)

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
```

```
100 9
```

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /home/ubuntu/.cache/tor
100%|| 553433881/553433881 [00:04<00:00, 125222955.14it/s]


CUDA baby!
```

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

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

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

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

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

        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
    '''
    img = Image.open(img_path)
    normalizer = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                      std=[0.229, 0.224, 0.225])
    preprocess_chain = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      normalizer])
    tensor = preprocess_chain(img).float()

    tensor.unsqueeze_(0)

    tensor = Variable(tensor) #The input to the network needs to be an autograd Variable
    if use_cuda:
        tensor = Variable(tensor.cuda())
    VGG16.eval()
    return VGG16(tensor).cpu().data.numpy().argmax()
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            prediction = VGG16_predict(img_path)
            return 151<= prediction and prediction <= 268
```

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: See below

```
In [9]: human_matches = sum(map(lambda x: 1 if dog_detector(x) else 0, human_files_short))
        dog_matches = sum(map(lambda x: 1 if dog_detector(x) else 0, dog_files_short))
```

```
In [32]: print(f"Dog matches in human file: {human_matches}%")
         print(f"Dog matches in dog files: {dog_matches}%")
```

```
Dog matches in human file: 1%
Dog matches in dog files: 100%
```

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

```
In [11]: ### (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!

In [21]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
from torchvision import datasets, models


base_dir = "dogImages"
train_data = datasets.ImageFolder(f"{base_dir}/train",
                                  transforms.Compose([
                                      transforms.RandomResizedCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation(10),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                           std=[0.229, 0.224, 0.225])
                                  ]))
valid_data = datasets.ImageFolder(f"{base_dir}/valid",
                                 transforms.Compose([
                                     transforms.Resize(224),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                          std=[0.229, 0.224, 0.225])
                                 ]))
test_data = datasets.ImageFolder(f"{base_dir}/test",
                                transforms.Compose([
                                    transforms.Resize(224),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                         std=[0.229, 0.224, 0.225])
                                ]))
batch_size = 20
num_workers = 0
loaders= {
    'train': torch.utils.data.DataLoader(train_data,
                                        batch_size=batch_size,
```

```
                                                  num_workers=num_workers,
                                                  shuffle=True),
           'valid': torch.utils.data.DataLoader(valid_data,
                                                  batch_size=batch_size,
                                                  num_workers=num_workers),
           'test': torch.utils.data.DataLoader(test_data,
                                                 batch_size=batch_size,
                                                 num_workers=num_workers)
       }
```

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 resized all images to 224x224 pixels, as this is the expected size for the trained models we are using. I augmented the dataset by doing random rotations and horizontal flips.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [13]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    def __init__(self,num_classes):
        super(Net, self).__init__()
        # convolutional layer (sees 224x224x3 image tensor)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 112x112x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        # convolutional layer (sees 56x56x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # convolutional layer (sees 28x28x64 tensor)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        # convolutional layer (sees 14x14x128 tensor)
        self.conv5 = nn.Conv2d(128, 256, 3, padding=1)

        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (256 * 7 * 7 -> 500)
        self.fc1 = nn.Linear(256 * 7 * 7, 500)
        # linear layer (500 -> 133)
        self.fc2 = nn.Linear(500, num_classes)
        # dropout layer (p=0.2)
        self.dropout = nn.Dropout(0.2)
```

```python
    def forward(self, x):
        # 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)))
        x = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))
        # flatten image input
        x = x.view(-1, 256 * 7 * 7)
        # 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

# create a complete CNN
model_scratch = Net(num_classes=len(test_data.classes))
print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(128, 256, 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=12544, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.2)
)
```

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

Answer: The task was very similar to the CIFAR classification problem we did during class, so I decided to reuse this architecture. I decided to add more CNN layers via trial and error. After 100 train iterations, the test accuracy was 26% (above the required 10%).

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 [14]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001, 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 [64]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     optimizer.zero_grad()
                     output = model(data)
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()
                     ## record the average training loss, using something like
                     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
                output = model(data)
                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
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model
```

```
In [ ]: # train the model
        loaders_scratch = loaders
        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: 4.889784         Validation Loss: 4.886490
Validation loss decreased (inf --> 4.886490).  Saving model ...
Epoch: 2          Training Loss: 4.883115         Validation Loss: 4.877176
Validation loss decreased (4.886490 --> 4.877176).  Saving model ...
Epoch: 3          Training Loss: 4.872932         Validation Loss: 4.867966
Validation loss decreased (4.877176 --> 4.867966).  Saving model ...
Epoch: 4          Training Loss: 4.865955         Validation Loss: 4.862979
Validation loss decreased (4.867966 --> 4.862979).  Saving model ...
Epoch: 5          Training Loss: 4.856029         Validation Loss: 4.847419
```

```
Validation loss decreased (4.862979 --> 4.847419).  Saving model ...
Epoch: 6        Training Loss: 4.822382         Validation Loss: 4.780346
Validation loss decreased (4.847419 --> 4.780346).  Saving model ...
Epoch: 7        Training Loss: 4.774634         Validation Loss: 4.709701
Validation loss decreased (4.780346 --> 4.709701).  Saving model ...
Epoch: 8        Training Loss: 4.754444         Validation Loss: 4.678847
Validation loss decreased (4.709701 --> 4.678847).  Saving model ...
Epoch: 9        Training Loss: 4.725448         Validation Loss: 4.644566
Validation loss decreased (4.678847 --> 4.644566).  Saving model ...
Epoch: 10        Training Loss: 4.704302         Validation Loss: 4.615551
Validation loss decreased (4.644566 --> 4.615551).  Saving model ...
Epoch: 11        Training Loss: 4.674893         Validation Loss: 4.582921
Validation loss decreased (4.615551 --> 4.582921).  Saving model ...
Epoch: 12        Training Loss: 4.659141         Validation Loss: 4.562819
Validation loss decreased (4.582921 --> 4.562819).  Saving model ...
Epoch: 13        Training Loss: 4.611168         Validation Loss: 4.470247
Validation loss decreased (4.562819 --> 4.470247).  Saving model ...
Epoch: 14        Training Loss: 4.542099         Validation Loss: 4.417639
Validation loss decreased (4.470247 --> 4.417639).  Saving model ...
Epoch: 15        Training Loss: 4.517997         Validation Loss: 4.379624
Validation loss decreased (4.417639 --> 4.379624).  Saving model ...
Epoch: 16        Training Loss: 4.488869         Validation Loss: 4.370356
Validation loss decreased (4.379624 --> 4.370356).  Saving model ...
Epoch: 17        Training Loss: 4.452358         Validation Loss: 4.319997
Validation loss decreased (4.370356 --> 4.319997).  Saving model ...
Epoch: 18        Training Loss: 4.441359         Validation Loss: 4.296715
Validation loss decreased (4.319997 --> 4.296715).  Saving model ...
Epoch: 19        Training Loss: 4.404367         Validation Loss: 4.243629
Validation loss decreased (4.296715 --> 4.243629).  Saving model ...
Epoch: 20        Training Loss: 4.372698         Validation Loss: 4.269446
Epoch: 21        Training Loss: 4.356566         Validation Loss: 4.204774
Validation loss decreased (4.243629 --> 4.204774).  Saving model ...
Epoch: 22        Training Loss: 4.326534         Validation Loss: 4.153225
Validation loss decreased (4.204774 --> 4.153225).  Saving model ...
Epoch: 23        Training Loss: 4.291367         Validation Loss: 4.137485
Validation loss decreased (4.153225 --> 4.137485).  Saving model ...
Epoch: 24        Training Loss: 4.261029         Validation Loss: 4.149223
Epoch: 25        Training Loss: 4.229192         Validation Loss: 4.114480
Validation loss decreased (4.137485 --> 4.114480).  Saving model ...
Epoch: 26        Training Loss: 4.204386         Validation Loss: 4.129507
Epoch: 27        Training Loss: 4.201059         Validation Loss: 4.095520
Validation loss decreased (4.114480 --> 4.095520).  Saving model ...
Epoch: 28        Training Loss: 4.148169         Validation Loss: 4.049977
Validation loss decreased (4.095520 --> 4.049977).  Saving model ...
Epoch: 29        Training Loss: 4.113026         Validation Loss: 4.006947
Validation loss decreased (4.049977 --> 4.006947).  Saving model ...
Epoch: 30        Training Loss: 4.067419         Validation Loss: 3.964527
Validation loss decreased (4.006947 --> 3.964527).  Saving model ...
```

```
Epoch: 31         Training Loss: 4.056256        Validation Loss: 3.910993
Validation loss decreased (3.964527 --> 3.910993).  Saving model ...
Epoch: 32         Training Loss: 4.024541        Validation Loss: 3.923293
Epoch: 33         Training Loss: 3.990745        Validation Loss: 3.912482
Epoch: 34         Training Loss: 3.949804        Validation Loss: 3.944984
Epoch: 35         Training Loss: 3.919040        Validation Loss: 3.808733
Validation loss decreased (3.910993 --> 3.808733).  Saving model ...
Epoch: 36         Training Loss: 3.861351        Validation Loss: 3.768476
Validation loss decreased (3.808733 --> 3.768476).  Saving model ...
Epoch: 37         Training Loss: 3.846671        Validation Loss: 3.841857
Epoch: 38         Training Loss: 3.829388        Validation Loss: 3.772561
Epoch: 39         Training Loss: 3.783464        Validation Loss: 3.716734
Validation loss decreased (3.768476 --> 3.716734).  Saving model ...
Epoch: 40         Training Loss: 3.776481        Validation Loss: 3.743884
Epoch: 41         Training Loss: 3.715876        Validation Loss: 3.744565
Epoch: 42         Training Loss: 3.678153        Validation Loss: 3.673555
Validation loss decreased (3.716734 --> 3.673555).  Saving model ...
Epoch: 43         Training Loss: 3.669951        Validation Loss: 3.692069
Epoch: 44         Training Loss: 3.647253        Validation Loss: 3.647503
Validation loss decreased (3.673555 --> 3.647503).  Saving model ...
Epoch: 45         Training Loss: 3.602177        Validation Loss: 3.655503
Epoch: 46         Training Loss: 3.574227        Validation Loss: 3.643983
Validation loss decreased (3.647503 --> 3.643983).  Saving model ...
Epoch: 47         Training Loss: 3.531542        Validation Loss: 3.854998
Epoch: 48         Training Loss: 3.532554        Validation Loss: 3.713891
Epoch: 49         Training Loss: 3.487007        Validation Loss: 3.611365
Validation loss decreased (3.643983 --> 3.611365).  Saving model ...
Epoch: 50         Training Loss: 3.472543        Validation Loss: 3.597763
Validation loss decreased (3.611365 --> 3.597763).  Saving model ...
Epoch: 51         Training Loss: 3.415751        Validation Loss: 3.544697
Validation loss decreased (3.597763 --> 3.544697).  Saving model ...
Epoch: 52         Training Loss: 3.382478        Validation Loss: 3.567870
Epoch: 53         Training Loss: 3.345677        Validation Loss: 3.600057
Epoch: 54         Training Loss: 3.358352        Validation Loss: 3.684749
Epoch: 55         Training Loss: 3.295126        Validation Loss: 3.460619
Validation loss decreased (3.544697 --> 3.460619).  Saving model ...
Epoch: 56         Training Loss: 3.270388        Validation Loss: 3.509361
Epoch: 57         Training Loss: 3.239504        Validation Loss: 3.524297
Epoch: 58         Training Loss: 3.210621        Validation Loss: 3.518078
Epoch: 59         Training Loss: 3.194969        Validation Loss: 3.498194
Epoch: 60         Training Loss: 3.169324        Validation Loss: 3.461069
Epoch: 61         Training Loss: 3.145607        Validation Loss: 3.515675
Epoch: 62         Training Loss: 3.119119        Validation Loss: 3.447179
Validation loss decreased (3.460619 --> 3.447179).  Saving model ...
Epoch: 63         Training Loss: 3.069993        Validation Loss: 3.462328
Epoch: 64         Training Loss: 3.034256        Validation Loss: 3.480963
Epoch: 65         Training Loss: 2.991189        Validation Loss: 3.383351
Validation loss decreased (3.447179 --> 3.383351).  Saving model ...
```

```
Epoch: 66         Training Loss: 2.990716         Validation Loss: 3.430193
Epoch: 67         Training Loss: 2.930177         Validation Loss: 3.476462
Epoch: 68         Training Loss: 2.917440         Validation Loss: 3.482674
Epoch: 69         Training Loss: 2.927912         Validation Loss: 3.357332
Validation loss decreased (3.383351 --> 3.357332).  Saving model ...
Epoch: 70         Training Loss: 2.876815         Validation Loss: 3.429902
Epoch: 71         Training Loss: 2.842196         Validation Loss: 3.425013
Epoch: 72         Training Loss: 2.830063         Validation Loss: 3.471098
Epoch: 73         Training Loss: 2.796131         Validation Loss: 3.398894
Epoch: 74         Training Loss: 2.774165         Validation Loss: 3.370842
Epoch: 75         Training Loss: 2.755107         Validation Loss: 3.382489
Epoch: 76         Training Loss: 2.730317         Validation Loss: 3.335224
Validation loss decreased (3.357332 --> 3.335224).  Saving model ...
Epoch: 77         Training Loss: 2.686940         Validation Loss: 3.422957
Epoch: 78         Training Loss: 2.714158         Validation Loss: 3.377885
Epoch: 79         Training Loss: 2.651335         Validation Loss: 3.386372
Epoch: 80         Training Loss: 2.620216         Validation Loss: 3.348988
Epoch: 81         Training Loss: 2.608140         Validation Loss: 3.439491
Epoch: 82         Training Loss: 2.575302         Validation Loss: 3.491554
Epoch: 83         Training Loss: 2.552158         Validation Loss: 3.373610
Epoch: 84         Training Loss: 2.519066         Validation Loss: 3.495273
```

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 [17]:
```python
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))

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

```
Test Loss: 3.200159


Test Accuracy: 26% (223/836)
```

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

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


         def transfer_model_for(model_generator, use_cuda):
             model = model_generator(pretrained=True)
             last_classifier_idx  = len(model.classifier) -1
             n_inputs = model.classifier[last_classifier_idx].in_features
             model.classifier[last_classifier_idx] = nn.Linear(n_inputs, len(test_data.classes))
```

```
        # Freeze training for all "features" layers
        for param in model.features.parameters():
            param.requires_grad = False

        if use_cuda:
            model = model.cuda()
        return model
```

In [73]:
```
# Load the pretrained model from pytorch
mf_vgg19 = transfer_model_for(models.vgg19, use_cuda)
```

In [74]:
```
# Load the pretrained model from pytorch
mf_vgg16 = transfer_model_for(models.vgg16, use_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:

I used vgg16, which was suggested by the project outline and also used during lecture. To adapt it to this classification task, I replaced the last classifier of the trained model with a fully connected linear layer with 133 output probabilities. This was enough to achieve an accuracy above the 60% called for by the task.

I also tried vgg19, but the results where very close and vgg16 is a simpler model, so I decided to stick with it.

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 [75]:
```
# specify loss function (categorical cross-entropy)
criterion_transfer = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.001
optimizer_transfer_vgg16 = optim.SGD(mf_vgg16.classifier.parameters(), lr=0.001)
```

In [76]:
```
# specify loss function (categorical cross-entropy)
criterion_transfer = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.001
optimizer_transfer_vgg19 = optim.SGD(mf_vgg19.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'`.

1.1.16 VGG19 Training

```
In [77]: loaders_transfer = loaders
         mf_vgg19 = train(10, loaders_transfer, mf_vgg19, optimizer_transfer_vgg19, criterion_tr
```

```
Epoch: 1 	Training Loss: 4.282469 	Validation Loss: 2.921531
Validation loss decreased (inf --> 2.921531).  Saving model ...
Epoch: 2 	Training Loss: 2.802637 	Validation Loss: 1.368044
Validation loss decreased (2.921531 --> 1.368044).  Saving model ...
Epoch: 3 	Training Loss: 1.907629 	Validation Loss: 0.832766
Validation loss decreased (1.368044 --> 0.832766).  Saving model ...
Epoch: 4 	Training Loss: 1.601625 	Validation Loss: 0.669369
Validation loss decreased (0.832766 --> 0.669369).  Saving model ...
Epoch: 5 	Training Loss: 1.428332 	Validation Loss: 0.585610
Validation loss decreased (0.669369 --> 0.585610).  Saving model ...
Epoch: 6 	Training Loss: 1.329207 	Validation Loss: 0.517709
Validation loss decreased (0.585610 --> 0.517709).  Saving model ...
Epoch: 7 	Training Loss: 1.274927 	Validation Loss: 0.480587
Validation loss decreased (0.517709 --> 0.480587).  Saving model ...
Epoch: 8 	Training Loss: 1.198537 	Validation Loss: 0.457677
Validation loss decreased (0.480587 --> 0.457677).  Saving model ...
Epoch: 9 	Training Loss: 1.192268 	Validation Loss: 0.450701
Validation loss decreased (0.457677 --> 0.450701).  Saving model ...
Epoch: 10 	Training Loss: 1.129528 	Validation Loss: 0.425760
Validation loss decreased (0.450701 --> 0.425760).  Saving model ...
```

1.1.17 VGG16 Training

```
In [79]: # train the model
         loaders_transfer = loaders
         mf_vgg16 = train(10, loaders_transfer, mf_vgg16, optimizer_transfer_vgg16, criterion_tr

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

```
Epoch: 1 	Training Loss: 4.301332 	Validation Loss: 2.932971
Validation loss decreased (inf --> 2.932971).  Saving model ...
Epoch: 2 	Training Loss: 2.844489 	Validation Loss: 1.347385
Validation loss decreased (2.932971 --> 1.347385).  Saving model ...
Epoch: 3 	Training Loss: 1.982736 	Validation Loss: 0.836266
Validation loss decreased (1.347385 --> 0.836266).  Saving model ...
Epoch: 4 	Training Loss: 1.622594 	Validation Loss: 0.673663
Validation loss decreased (0.836266 --> 0.673663).  Saving model ...
Epoch: 5 	Training Loss: 1.470130 	Validation Loss: 0.573100
Validation loss decreased (0.673663 --> 0.573100).  Saving model ...
Epoch: 6 	Training Loss: 1.333560 	Validation Loss: 0.528198
Validation loss decreased (0.573100 --> 0.528198).  Saving model ...
Epoch: 7 	Training Loss: 1.294317 	Validation Loss: 0.499459
```

```
Validation loss decreased (0.528198 --> 0.499459).  Saving model ...
Epoch: 8         Training Loss: 1.229500         Validation Loss: 0.474340
Validation loss decreased (0.499459 --> 0.474340).  Saving model ...
Epoch: 9         Training Loss: 1.193651         Validation Loss: 0.464403
Validation loss decreased (0.474340 --> 0.464403).  Saving model ...
Epoch: 10         Training Loss: 1.164508         Validation Loss: 0.449790
Validation loss decreased (0.464403 --> 0.449790).  Saving model ...
```

1.1.18 (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%.

1.1.19 VGG16 Results

```
In [83]: test(loaders_transfer, mf_vgg16, criterion_transfer, use_cuda)
```

```
Test Loss: 0.484016


Test Accuracy: 85% (712/836)
```

1.1.20 VGG19 Results

```
In [84]: test(loaders_transfer, mf_vgg19, criterion_transfer, use_cuda)
```

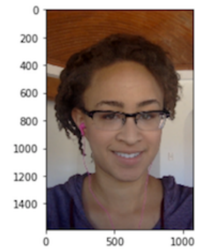
```
Test Loss: 0.473940


Test Accuracy: 85% (713/836)
```

1.1.21 (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 [85]: from PIL import Image
         import torchvision.transforms as transforms
         from torch.autograd import Variable

         def transfer_predict(model, img_path):
             img = Image.open(img_path)
             normalizer = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                               std=[0.229, 0.224, 0.225])
             preprocess_chain = transforms.Compose([transforms.Resize(256),
                                               transforms.CenterCrop(224),
```

```
hello, human!
```



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

Sample Human Output

```
                                                transforms.ToTensor(),
                                                normalizer])
    tensor = preprocess_chain(img).float()

    tensor.unsqueeze_(0)

    tensor = Variable(tensor) #The input to the network needs to be an autograd Variabl
    if use_cuda:
        tensor = Variable(tensor.cuda())
    return model(tensor).cpu().data.numpy().argmax()
```

In [91]:
```
### 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, model):
    index = transfer_predict(model,img_path)
    return class_names[index], train_data.classes[index]
```

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!

1.1.22 (IMPLEMENTATION) Write your Algorithm

In [98]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def handle_human():
    pass

def handle_dog():
    pass

def run_app(img_path, model):
    is_dog = dog_detector(img_path)
    breed, name = predict_breed_transfer(img_path,model)

    # display test image
    fig = plt.figure(figsize=(16,4))

    if(face_detector(img_path)):
        print("Hello, human")
        ax = fig.add_subplot(1,2,1)
        cv_rgb = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)

        # display the image, along with bounding box
        ax.imshow(cv_rgb)
        plt.axis('off')

        print("You look like a ...")
        print(name)
        plt.show()
        return

    if(dog_detector(img_path)):
        print("Hello, dog")
        ax = fig.add_subplot(1,2,1)
        cv_rgb = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)
        ax.imshow(cv_rgb)
        plt.axis('off')
        print("You look like ... " + breed)
        plt.show()
        return

    ax = fig.add_subplot(1,2,1)
    img = cv2.imread(img_path)
    ax.imshow(img)
    plt.axis('off')
    plt.show()
    print('Hello, stranger. What are you?')
```

```
        return
```

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?

1.1.23 (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:

The algorithm is worse than I though. I was expecting higher (>95%) accuracy. I think I could spend more time tweeking the following elements of my model:

1. The optimizer used
2. The number of layers added to the tranfer model--I only replaced the last. Could I have added more?
3. The data augmentation mechanisms used.

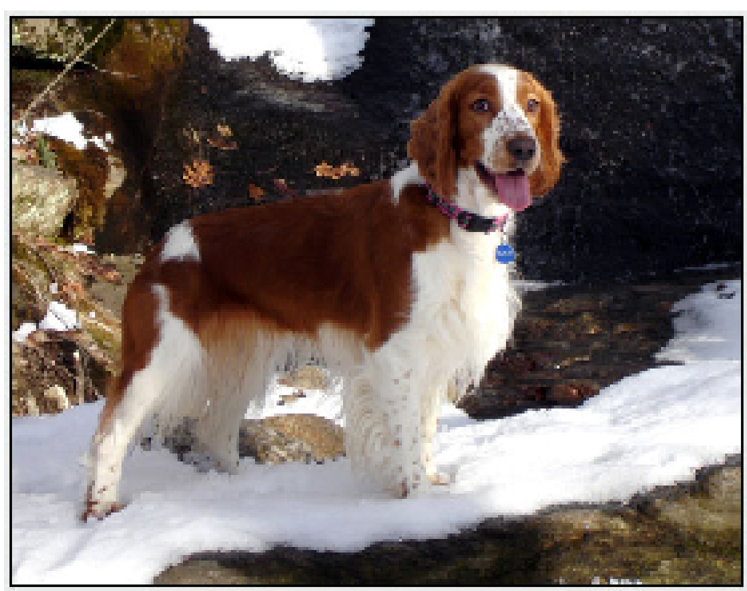
```
In [99]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[:2], dog_files[:2])):
             run_app(file, mf_vgg16)
```

```
Hello, human
You look like a ...
009.American_water_spaniel
```

```
Hello, human
You look like a ...
120.Pharaoh_hound
```

```
Hello, dog
You look like ... Welsh springer spaniel
```


```
Hello, dog
You look like ... Welsh springer spaniel
```

```
In [ ]:
```

```
In [ ]:
```