dog_app

April 19, 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 [1]: 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

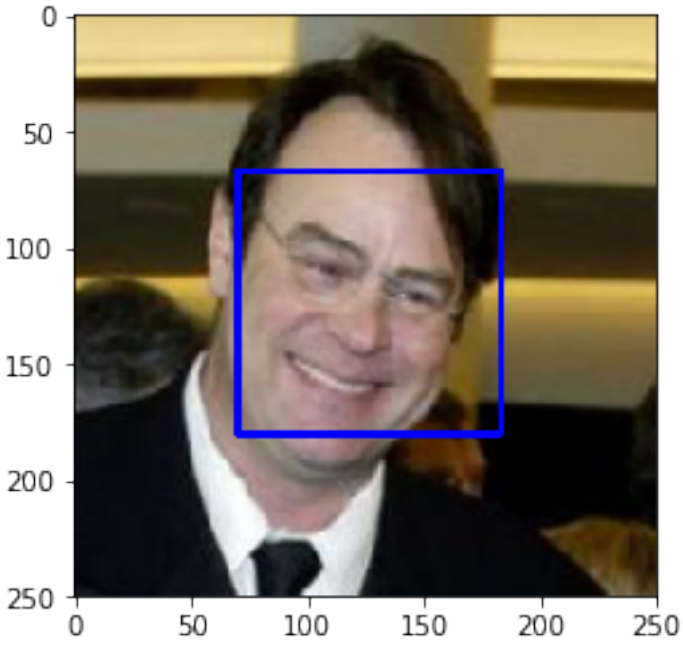
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [3]: 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 [4]: # 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 [5]: 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.

        count_humans = 0
        count_dogs = 0

        for file in human_files_short:
            if face_detector(file) == True:
                count_humans +=1

        for file in dog_files_short:
            if face_detector(file) == True:
                count_dogs +=1
        print('%.1f%% images of the first 100 human_files were detected as human face.' % count_
        print('%.1f%% images of the first 100 dog_files were detected as human face.' % count_do
```

```
98.0% images of the first 100 human_files were detected as human face.
17.0% images of the first 100 dog_files were detected as human face.
```

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 [6]: ### (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 [7]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 98324211.51it/s]
```

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

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

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

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

```
In [8]: from PIL import Image
        import torchvision.transforms as transforms

        def load_image(img_path):
            image = Image.open(img_path).convert('RGB')
            # resize the input image into 224x224 because VGG16 takes only 224x224 pixel image
            in_transform = transforms.Compose([transforms.Resize(size=(224,224)),
                                               transforms.ToTensor()
                                               ])
            image = in_transform(image)[:3,:,:].unsqueeze(0)
            return image
```

```
In [9]: from PIL import Image
        import torchvision.transforms as transforms

        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

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

            Args:
                img_path: path to an image

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

            ## TODO: Complete the function.
            ## Load and pre-process an image from the given img_path
            ## Return the *index* of the predicted class for that image
            img = load_image(img_path)
            if use_cuda:
                img = img.cuda()
            ret = VGG16(img)

            return torch.max(ret,1)[1].item() # predicted class index
```

```
In [10]: VGG16_predict(dog_files_short[0])
```

```
Out[10]: 243
```

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

In [12]: print(dog_detector(dog_files_short[0]))
         print(dog_detector(human_files_short[0]))

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: 0% of dogs in human_files_short and 93% of dogs in dog_files_short

```
In [13]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         def dog_detector_test(files):
             detection_cnt = 0;
             total_cnt = len(files)
             for file in files:
                 detection_cnt += dog_detector(file)
             return detection_cnt, total_cnt

In [14]: print("detect a dog in human_files: {} / {}".format(dog_detector_test(human_files_short
         print("detect a dog in dog_files: {} / {}".format(dog_detector_test(dog_files_short)[0]

detect a dog in human_files: 0 / 100
detect a dog in dog_files: 93 / 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 [15]: ### (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 [16]: import os
         from torchvision import datasets

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
batch_size = 20
num_workers = 0 # default is 0

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


standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0

data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                                transforms.RandomHorizontalFlip(),
                                                 transforms.ToTensor(),
                                                 standard_normalization
                                                ]),

                   'val': transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                standard_normalization]),

                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                                transforms.ToTensor(),
                                                standard_normalization])

                  }
```

In [17]:
```
train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['val'])
test_data = datasets.ImageFolder(test_dir, transform = data_transforms['test'])

train_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size, num_worker
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_worke
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers

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

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

Answer:As resizing the image data is one of the important part of preprocessing because different images have different pixel sizes,so i resize my images dataset using RandomResizedCrop and i set the resize value to be 224. Augmenting the data also helps when we have imbalanced dataset problem. So i applied RandomHorizontalClip function to our training data only because we will train our data on that train dataset only rest for valid and test dataset we will not require to do augmentation.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [18]:
```
num_classes = 133
```

In [19]:
```
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

# 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, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

        # fully-connected
        self.fc1 = nn.Linear(7*7*128, 500)
        self.fc2 = nn.Linear(500, num_classes)

        # drop-out
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        (_, C, H, W) = x.data.size()

        # flatten
        x = x.view( -1 , C * H * W)

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

        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

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

        # instantiate the CNN
        model_scratch = Net()
        print(model_scratch)
        # move tensors to GPU if CUDA is available
        if use_cuda:
            model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
)
```

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

Answer: I have used 3 conv2d layer for architecture with kernel size = 3,stride = 2, padding = 1. The first 2 layer implemented with a kernel size of 3 and stride 2 and padding of 1. The 3rd layer is implemented with kernal size of 3 and padding of 1 but i have not implemented stride operation on 3rd layer.After each layer i implemented the maxpooling layer and at last i have used dropout regularization with value 0.3.Dropout prevents our algorithm from overfitting. I have used RELU activation function at the hidden layers of our CNN architecture. RELU activation function gives values between 0 and 1.I have used 2 fully connected layers. Last fully connected layer gives the predictions from one of the 133 classes.

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 [20]: import torch.optim as optim

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

        ### TODO: select optimizer
        optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.0001)
```

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 [21]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path, last_val
             """returns trained model"""
             # initialize tracker for minimum validation loss
             if last_validation_loss is not None:
                 valid_loss_min = last_validation_loss
             else:
                 valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

                     # initialize weights to zero
                     optimizer.zero_grad()

                     output = model(data)

                     # calculate loss
                     loss = criterion(output, target)

                     # back prop
                     loss.backward()

                     # grad
                     optimizer.step()

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

                     if batch_idx % 100 == 0:
                         print('Epoch %d, Batch %d loss: %.6f' %
                           (epoch, batch_idx + 1, 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)
                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:
                torch.save(model.state_dict(), save_path)
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                valid_loss_min,
                valid_loss))
                valid_loss_min = valid_loss

        # return trained model
        return model
```

```
In [22]: model_scratch = train(40, loader_scratch, model_scratch, optimizer_scratch,  criterion_
```

```
Epoch 1, Batch 1 loss: 4.871049
Epoch 1, Batch 101 loss: 4.893119
Epoch 1, Batch 201 loss: 4.885406
Epoch 1, Batch 301 loss: 4.867074
Epoch: 1         Training Loss: 4.864035         Validation Loss: 4.786391
Validation loss decreased (inf --> 4.786391).  Saving model ...
Epoch 2, Batch 1 loss: 4.781516
Epoch 2, Batch 101 loss: 4.751566
Epoch 2, Batch 201 loss: 4.721057
Epoch 2, Batch 301 loss: 4.705526
Epoch: 2         Training Loss: 4.704655         Validation Loss: 4.559371
Validation loss decreased (4.786391 --> 4.559371).  Saving model ...
Epoch 3, Batch 1 loss: 4.849067
Epoch 3, Batch 101 loss: 4.630446
Epoch 3, Batch 201 loss: 4.603781
Epoch 3, Batch 301 loss: 4.596972
Epoch: 3         Training Loss: 4.597475         Validation Loss: 4.470085
```

```
Validation loss decreased (4.559371 --> 4.470085).  Saving model ...
Epoch 4, Batch 1 loss: 4.419483
Epoch 4, Batch 101 loss: 4.535429
Epoch 4, Batch 201 loss: 4.543274
Epoch 4, Batch 301 loss: 4.535500
Epoch: 4         Training Loss: 4.531193         Validation Loss: 4.384349
Validation loss decreased (4.470085 --> 4.384349).  Saving model ...
Epoch 5, Batch 1 loss: 4.461885
Epoch 5, Batch 101 loss: 4.467141
Epoch 5, Batch 201 loss: 4.457199
Epoch 5, Batch 301 loss: 4.452735
Epoch: 5         Training Loss: 4.450023         Validation Loss: 4.286588
Validation loss decreased (4.384349 --> 4.286588).  Saving model ...
Epoch 6, Batch 1 loss: 4.386744
Epoch 6, Batch 101 loss: 4.389340
Epoch 6, Batch 201 loss: 4.402273
Epoch 6, Batch 301 loss: 4.414482
Epoch: 6         Training Loss: 4.413668         Validation Loss: 4.272698
Validation loss decreased (4.286588 --> 4.272698).  Saving model ...
Epoch 7, Batch 1 loss: 4.475667
Epoch 7, Batch 101 loss: 4.334605
Epoch 7, Batch 201 loss: 4.358491
Epoch 7, Batch 301 loss: 4.357951
Epoch: 7         Training Loss: 4.356931         Validation Loss: 4.196947
Validation loss decreased (4.272698 --> 4.196947).  Saving model ...
Epoch 8, Batch 1 loss: 4.300153
Epoch 8, Batch 101 loss: 4.298705
Epoch 8, Batch 201 loss: 4.298914
Epoch 8, Batch 301 loss: 4.295869
Epoch: 8         Training Loss: 4.297266         Validation Loss: 4.116701
Validation loss decreased (4.196947 --> 4.116701).  Saving model ...
Epoch 9, Batch 1 loss: 4.138509
Epoch 9, Batch 101 loss: 4.234140
Epoch 9, Batch 201 loss: 4.247304
Epoch 9, Batch 301 loss: 4.245508
Epoch: 9         Training Loss: 4.244715         Validation Loss: 4.052224
Validation loss decreased (4.116701 --> 4.052224).  Saving model ...
Epoch 10, Batch 1 loss: 4.305787
Epoch 10, Batch 101 loss: 4.179132
Epoch 10, Batch 201 loss: 4.197581
Epoch 10, Batch 301 loss: 4.203821
Epoch: 10         Training Loss: 4.202878         Validation Loss: 4.025463
Validation loss decreased (4.052224 --> 4.025463).  Saving model ...
Epoch 11, Batch 1 loss: 3.869030
Epoch 11, Batch 101 loss: 4.157185
Epoch 11, Batch 201 loss: 4.169479
Epoch 11, Batch 301 loss: 4.175796
Epoch: 11         Training Loss: 4.169549         Validation Loss: 3.970160
```

```
Validation loss decreased (4.025463 --> 3.970160).  Saving model ...
Epoch 12, Batch 1 loss: 3.899272
Epoch 12, Batch 101 loss: 4.119015
Epoch 12, Batch 201 loss: 4.103691
Epoch 12, Batch 301 loss: 4.101757
Epoch: 12         Training Loss: 4.104415         Validation Loss: 3.932025
Validation loss decreased (3.970160 --> 3.932025).  Saving model ...
Epoch 13, Batch 1 loss: 3.868602
Epoch 13, Batch 101 loss: 4.068631
Epoch 13, Batch 201 loss: 4.070665
Epoch 13, Batch 301 loss: 4.061615
Epoch: 13         Training Loss: 4.063232         Validation Loss: 3.899366
Validation loss decreased (3.932025 --> 3.899366).  Saving model ...
Epoch 14, Batch 1 loss: 4.004123
Epoch 14, Batch 101 loss: 4.018809
Epoch 14, Batch 201 loss: 4.028363
Epoch 14, Batch 301 loss: 4.042857
Epoch: 14         Training Loss: 4.038366         Validation Loss: 3.867612
Validation loss decreased (3.899366 --> 3.867612).  Saving model ...
Epoch 15, Batch 1 loss: 3.664016
Epoch 15, Batch 101 loss: 3.969246
Epoch 15, Batch 201 loss: 3.970093
Epoch 15, Batch 301 loss: 3.990197
Epoch: 15         Training Loss: 3.990407         Validation Loss: 3.834262
Validation loss decreased (3.867612 --> 3.834262).  Saving model ...
Epoch 16, Batch 1 loss: 3.723940
Epoch 16, Batch 101 loss: 3.971588
Epoch 16, Batch 201 loss: 3.976279
Epoch 16, Batch 301 loss: 3.962242
Epoch: 16         Training Loss: 3.968453         Validation Loss: 3.805078
Validation loss decreased (3.834262 --> 3.805078).  Saving model ...
Epoch 17, Batch 1 loss: 3.696424
Epoch 17, Batch 101 loss: 3.919991
Epoch 17, Batch 201 loss: 3.914757
Epoch 17, Batch 301 loss: 3.931229
Epoch: 17         Training Loss: 3.927326         Validation Loss: 3.753586
Validation loss decreased (3.805078 --> 3.753586).  Saving model ...
Epoch 18, Batch 1 loss: 3.783653
Epoch 18, Batch 101 loss: 3.846440
Epoch 18, Batch 201 loss: 3.889733
Epoch 18, Batch 301 loss: 3.898976
Epoch: 18         Training Loss: 3.905682         Validation Loss: 3.753460
Validation loss decreased (3.753586 --> 3.753460).  Saving model ...
Epoch 19, Batch 1 loss: 3.314863
Epoch 19, Batch 101 loss: 3.896262
Epoch 19, Batch 201 loss: 3.902553
Epoch 19, Batch 301 loss: 3.881989
Epoch: 19         Training Loss: 3.875132         Validation Loss: 3.711045
```

```
Validation loss decreased (3.753460 --> 3.711045).  Saving model ...
Epoch 20, Batch 1 loss: 3.673454
Epoch 20, Batch 101 loss: 3.822494
Epoch 20, Batch 201 loss: 3.841267
Epoch 20, Batch 301 loss: 3.837610
Epoch: 20         Training Loss: 3.844916         Validation Loss: 3.709469
Validation loss decreased (3.711045 --> 3.709469).  Saving model ...
Epoch 21, Batch 1 loss: 3.816968
Epoch 21, Batch 101 loss: 3.781257
Epoch 21, Batch 201 loss: 3.804330
Epoch 21, Batch 301 loss: 3.802641
Epoch: 21         Training Loss: 3.805174         Validation Loss: 3.669756
Validation loss decreased (3.709469 --> 3.669756).  Saving model ...
Epoch 22, Batch 1 loss: 3.867445
Epoch 22, Batch 101 loss: 3.813908
Epoch 22, Batch 201 loss: 3.803663
Epoch 22, Batch 301 loss: 3.786221
Epoch: 22         Training Loss: 3.785575         Validation Loss: 3.676198
Epoch 23, Batch 1 loss: 3.669495
Epoch 23, Batch 101 loss: 3.757560
Epoch 23, Batch 201 loss: 3.744747
Epoch 23, Batch 301 loss: 3.748357
Epoch: 23         Training Loss: 3.749282         Validation Loss: 3.664670
Validation loss decreased (3.669756 --> 3.664670).  Saving model ...
Epoch 24, Batch 1 loss: 3.860677
Epoch 24, Batch 101 loss: 3.674917
Epoch 24, Batch 201 loss: 3.699544
Epoch 24, Batch 301 loss: 3.726098
Epoch: 24         Training Loss: 3.718968         Validation Loss: 3.636617
Validation loss decreased (3.664670 --> 3.636617).  Saving model ...
Epoch 25, Batch 1 loss: 3.915054
Epoch 25, Batch 101 loss: 3.700215
Epoch 25, Batch 201 loss: 3.732625
Epoch 25, Batch 301 loss: 3.728964
Epoch: 25         Training Loss: 3.727643         Validation Loss: 3.599848
Validation loss decreased (3.636617 --> 3.599848).  Saving model ...
Epoch 26, Batch 1 loss: 3.663528
Epoch 26, Batch 101 loss: 3.650700
Epoch 26, Batch 201 loss: 3.672613
Epoch 26, Batch 301 loss: 3.660950
Epoch: 26         Training Loss: 3.665729         Validation Loss: 3.588993
Validation loss decreased (3.599848 --> 3.588993).  Saving model ...
Epoch 27, Batch 1 loss: 4.046014
Epoch 27, Batch 101 loss: 3.658147
Epoch 27, Batch 201 loss: 3.658832
Epoch 27, Batch 301 loss: 3.632631
Epoch: 27         Training Loss: 3.648492         Validation Loss: 3.617462
Epoch 28, Batch 1 loss: 3.133695
```

```
Epoch 28, Batch 101 loss: 3.617220
Epoch 28, Batch 201 loss: 3.628532
Epoch 28, Batch 301 loss: 3.626976
Epoch: 28         Training Loss: 3.627260         Validation Loss: 3.578496
Validation loss decreased (3.588993 --> 3.578496).  Saving model ...
Epoch 29, Batch 1 loss: 3.431058
Epoch 29, Batch 101 loss: 3.524776
Epoch 29, Batch 201 loss: 3.563466
Epoch 29, Batch 301 loss: 3.565335
Epoch: 29         Training Loss: 3.567729         Validation Loss: 3.627954
Epoch 30, Batch 1 loss: 3.966942
Epoch 30, Batch 101 loss: 3.619689
Epoch 30, Batch 201 loss: 3.611555
Epoch 30, Batch 301 loss: 3.592018
Epoch: 30         Training Loss: 3.603190         Validation Loss: 3.556231
Validation loss decreased (3.578496 --> 3.556231).  Saving model ...
Epoch 31, Batch 1 loss: 3.824828
Epoch 31, Batch 101 loss: 3.538270
Epoch 31, Batch 201 loss: 3.564185
Epoch 31, Batch 301 loss: 3.559955
Epoch: 31         Training Loss: 3.562279         Validation Loss: 3.517561
Validation loss decreased (3.556231 --> 3.517561).  Saving model ...
Epoch 32, Batch 1 loss: 3.483950
Epoch 32, Batch 101 loss: 3.516318
Epoch 32, Batch 201 loss: 3.515965
Epoch 32, Batch 301 loss: 3.521655
Epoch: 32         Training Loss: 3.534929         Validation Loss: 3.519052
Epoch 33, Batch 1 loss: 3.179828
Epoch 33, Batch 101 loss: 3.532397
Epoch 33, Batch 201 loss: 3.533552
Epoch 33, Batch 301 loss: 3.532106
Epoch: 33         Training Loss: 3.532175         Validation Loss: 3.486520
Validation loss decreased (3.517561 --> 3.486520).  Saving model ...
Epoch 34, Batch 1 loss: 3.470614
Epoch 34, Batch 101 loss: 3.464051
Epoch 34, Batch 201 loss: 3.472976
Epoch 34, Batch 301 loss: 3.476699
Epoch: 34         Training Loss: 3.488072         Validation Loss: 3.505313
Epoch 35, Batch 1 loss: 3.099227
Epoch 35, Batch 101 loss: 3.453094
Epoch 35, Batch 201 loss: 3.491020
Epoch 35, Batch 301 loss: 3.491463
Epoch: 35         Training Loss: 3.489741         Validation Loss: 3.476178
Validation loss decreased (3.486520 --> 3.476178).  Saving model ...
Epoch 36, Batch 1 loss: 3.070490
Epoch 36, Batch 101 loss: 3.439601
Epoch 36, Batch 201 loss: 3.468182
Epoch 36, Batch 301 loss: 3.465155
```

```
Epoch: 36         Training Loss: 3.462722         Validation Loss: 3.455202
Validation loss decreased (3.476178 --> 3.455202).  Saving model ...
Epoch 37, Batch 1 loss: 3.788702
Epoch 37, Batch 101 loss: 3.420576
Epoch 37, Batch 201 loss: 3.460891
Epoch 37, Batch 301 loss: 3.454637
Epoch: 37         Training Loss: 3.456223         Validation Loss: 3.464204
Epoch 38, Batch 1 loss: 3.067639
Epoch 38, Batch 101 loss: 3.419470
Epoch 38, Batch 201 loss: 3.436675
Epoch 38, Batch 301 loss: 3.440780
Epoch: 38         Training Loss: 3.448012         Validation Loss: 3.466990
Epoch 39, Batch 1 loss: 3.400626
Epoch 39, Batch 101 loss: 3.445652
Epoch 39, Batch 201 loss: 3.425551
Epoch 39, Batch 301 loss: 3.432493
Epoch: 39         Training Loss: 3.437361         Validation Loss: 3.438279
Validation loss decreased (3.455202 --> 3.438279).  Saving model ...
Epoch 40, Batch 1 loss: 3.243506
Epoch 40, Batch 101 loss: 3.434815
Epoch 40, Batch 201 loss: 3.412618
Epoch 40, Batch 301 loss: 3.404395
Epoch: 40         Training Loss: 3.410262         Validation Loss: 3.423297
Validation loss decreased (3.438279 --> 3.423297).  Saving model ...
```

```
In [23]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratches.pt'))
```

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 [24]: def test(loaders, model, criterion, use_cuda):

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

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders['test']):
                 # move to GPU
                 if use_cuda:
                     data, target = data.cuda(), target.cuda()
                 # forward pass: compute predicted outputs by passing inputs to the model
                 output = model(data)
```

```
            # calculate the loss
            loss = criterion(output, target)
            # update average test loss
            test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
            # convert output probabilities to predicted class
            pred = output.data.max(1, keepdim=True)[1]
            # compare predictions to true label
            correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
            total += data.size(0)

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

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

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

```
Test Loss: 3.666922


Test Accuracy: 16% (138/836)
```

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 [25]: ## TODO: Specify data loaders
         loader_transfer = loader_scratch.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [26]: import torchvision.models as models
         import torch.nn as nn
```

```
        ## TODO: Specify model architecture
        model_transfer = models.resnet50(pretrained=True)

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

        model_transfer.fc = nn.Linear(2048,133,bias=True)

        fc_parameters = model_transfer.fc.parameters()

        for param in fc_parameters:
            param.requires_grad = True

        model_transfer

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 85765821.99it/s]
```

```
In [27]: model_transfer
```

```
Out[27]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
           (layer1): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
               (downsample): Sequential(
                 (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
                 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               )
             )
             (1): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
```

```
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
```

```
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
)
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
```

```
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

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: After trying using VGG16 pretrained model for our custom CNN i get less accuracy but when i used Transfer Learning with resnet50 pretrained model even after less epochs i get great accuracy as compare to our custom CNN.In our custom CNN even after 40 epochs i get accuracy of 16% and when i used transfer learning even with epoch=20 i get accuracy of 85%. reset also prevents overfitting problems so i think it is the best possible solution to our 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 [28]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(),lr = 0.0001)
```

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 [29]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""

             valid_loss_min = np.Inf

             for epoch in range(1, n_epochs+1):

                 train_loss = 0.0
                 valid_loss = 0.0


                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):

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


                     optimizer.zero_grad()
```

```
            output = model(data)

            loss = criterion(output, target)

            loss.backward()

            optimizer.step()

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

            if batch_idx % 100 == 0:
                print('Epoch %d, Batch %d loss: %.6f' %
                  (epoch, batch_idx + 1, train_loss))


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

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

            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)



        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))


        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss

    # return trained model
    return model
```

```
In [30]: # train the model
         model_transfer = train(20, loader_transfer, model_transfer, optimizer_transfer, criteri

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

```
Epoch 1, Batch 1 loss: 4.946682
Epoch 1, Batch 101 loss: 4.721754
Epoch 1, Batch 201 loss: 4.472858
Epoch 1, Batch 301 loss: 4.254427
Epoch: 1         Training Loss: 4.191656         Validation Loss: 3.134689
Validation loss decreased (inf --> 3.134689).  Saving model ...
Epoch 2, Batch 1 loss: 3.408025
Epoch 2, Batch 101 loss: 3.316442
Epoch 2, Batch 201 loss: 3.180854
Epoch 2, Batch 301 loss: 3.051431
Epoch: 2         Training Loss: 3.014789         Validation Loss: 2.058029
Validation loss decreased (3.134689 --> 2.058029).  Saving model ...
Epoch 3, Batch 1 loss: 2.375204
Epoch 3, Batch 101 loss: 2.491588
Epoch 3, Batch 201 loss: 2.393599
Epoch 3, Batch 301 loss: 2.319348
Epoch: 3         Training Loss: 2.296501         Validation Loss: 1.485861
Validation loss decreased (2.058029 --> 1.485861).  Saving model ...
Epoch 4, Batch 1 loss: 2.178491
Epoch 4, Batch 101 loss: 1.930932
Epoch 4, Batch 201 loss: 1.916455
Epoch 4, Batch 301 loss: 1.891082
Epoch: 4         Training Loss: 1.876997         Validation Loss: 1.141196
Validation loss decreased (1.485861 --> 1.141196).  Saving model ...
Epoch 5, Batch 1 loss: 1.679326
Epoch 5, Batch 101 loss: 1.684159
Epoch 5, Batch 201 loss: 1.654349
Epoch 5, Batch 301 loss: 1.634781
Epoch: 5         Training Loss: 1.625985         Validation Loss: 0.957861
Validation loss decreased (1.141196 --> 0.957861).  Saving model ...
Epoch 6, Batch 1 loss: 1.200581
Epoch 6, Batch 101 loss: 1.489776
Epoch 6, Batch 201 loss: 1.475439
Epoch 6, Batch 301 loss: 1.459616
Epoch: 6         Training Loss: 1.446528         Validation Loss: 0.834507
Validation loss decreased (0.957861 --> 0.834507).  Saving model ...
Epoch 7, Batch 1 loss: 1.191185
Epoch 7, Batch 101 loss: 1.313285
Epoch 7, Batch 201 loss: 1.326967
Epoch 7, Batch 301 loss: 1.318720
Epoch: 7         Training Loss: 1.313999         Validation Loss: 0.749984
Validation loss decreased (0.834507 --> 0.749984).  Saving model ...
```

```
Epoch 8, Batch 1 loss: 1.310118
Epoch 8, Batch 101 loss: 1.295108
Epoch 8, Batch 201 loss: 1.261436
Epoch 8, Batch 301 loss: 1.236604
Epoch: 8         Training Loss: 1.230660         Validation Loss: 0.688011
Validation loss decreased (0.749984 --> 0.688011).  Saving model ...
Epoch 9, Batch 1 loss: 1.020746
Epoch 9, Batch 101 loss: 1.138199
Epoch 9, Batch 201 loss: 1.144033
Epoch 9, Batch 301 loss: 1.144463
Epoch: 9         Training Loss: 1.142856         Validation Loss: 0.639458
Validation loss decreased (0.688011 --> 0.639458).  Saving model ...
Epoch 10, Batch 1 loss: 0.947936
Epoch 10, Batch 101 loss: 1.100933
Epoch 10, Batch 201 loss: 1.108604
Epoch 10, Batch 301 loss: 1.102136
Epoch: 10         Training Loss: 1.102283         Validation Loss: 0.591206
Validation loss decreased (0.639458 --> 0.591206).  Saving model ...
Epoch 11, Batch 1 loss: 1.030619
Epoch 11, Batch 101 loss: 1.039137
Epoch 11, Batch 201 loss: 1.043089
Epoch 11, Batch 301 loss: 1.037969
Epoch: 11         Training Loss: 1.035946         Validation Loss: 0.554545
Validation loss decreased (0.591206 --> 0.554545).  Saving model ...
Epoch 12, Batch 1 loss: 1.012159
Epoch 12, Batch 101 loss: 0.941199
Epoch 12, Batch 201 loss: 0.980108
Epoch 12, Batch 301 loss: 0.987987
Epoch: 12         Training Loss: 0.991515         Validation Loss: 0.533829
Validation loss decreased (0.554545 --> 0.533829).  Saving model ...
Epoch 13, Batch 1 loss: 1.028037
Epoch 13, Batch 101 loss: 0.953201
Epoch 13, Batch 201 loss: 0.954135
Epoch 13, Batch 301 loss: 0.957232
Epoch: 13         Training Loss: 0.958578         Validation Loss: 0.525996
Validation loss decreased (0.533829 --> 0.525996).  Saving model ...
Epoch 14, Batch 1 loss: 1.021285
Epoch 14, Batch 101 loss: 0.971218
Epoch 14, Batch 201 loss: 0.954885
Epoch 14, Batch 301 loss: 0.935948
Epoch: 14         Training Loss: 0.943676         Validation Loss: 0.498315
Validation loss decreased (0.525996 --> 0.498315).  Saving model ...
Epoch 15, Batch 1 loss: 1.019009
Epoch 15, Batch 101 loss: 0.908760
Epoch 15, Batch 201 loss: 0.900598
Epoch 15, Batch 301 loss: 0.898875
Epoch: 15         Training Loss: 0.905018         Validation Loss: 0.492271
Validation loss decreased (0.498315 --> 0.492271).  Saving model ...
```

```
Epoch 16, Batch 1 loss: 1.040815
Epoch 16, Batch 101 loss: 0.893491
Epoch 16, Batch 201 loss: 0.880899
Epoch 16, Batch 301 loss: 0.881799
Epoch: 16         Training Loss: 0.878515         Validation Loss: 0.467434
Validation loss decreased (0.492271 --> 0.467434).  Saving model ...
Epoch 17, Batch 1 loss: 1.129126
Epoch 17, Batch 101 loss: 0.870486
Epoch 17, Batch 201 loss: 0.854885
Epoch 17, Batch 301 loss: 0.848426
Epoch: 17         Training Loss: 0.848242         Validation Loss: 0.460464
Validation loss decreased (0.467434 --> 0.460464).  Saving model ...
Epoch 18, Batch 1 loss: 0.946332
Epoch 18, Batch 101 loss: 0.871542
Epoch 18, Batch 201 loss: 0.843692
Epoch 18, Batch 301 loss: 0.837989
Epoch: 18         Training Loss: 0.839341         Validation Loss: 0.454109
Validation loss decreased (0.460464 --> 0.454109).  Saving model ...
Epoch 19, Batch 1 loss: 1.096290
Epoch 19, Batch 101 loss: 0.810670
Epoch 19, Batch 201 loss: 0.826903
Epoch 19, Batch 301 loss: 0.823902
Epoch: 19         Training Loss: 0.823866         Validation Loss: 0.431243
Validation loss decreased (0.454109 --> 0.431243).  Saving model ...
Epoch 20, Batch 1 loss: 0.912636
Epoch 20, Batch 101 loss: 0.822987
Epoch 20, Batch 201 loss: 0.825596
Epoch 20, Batch 301 loss: 0.812630
Epoch: 20         Training Loss: 0.814144         Validation Loss: 0.421388
Validation loss decreased (0.431243 --> 0.421388).  Saving model ...
```

1.1.16 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

```
In [31]: test(loader_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.493131


Test Accuracy: 85% (718/836)
```

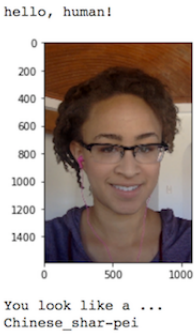
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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]: from PIL import Image
         import torchvision.transforms as transforms

         def load_image(img_path):
             image = Image.open(img_path).convert('RGB')
             # resize the input image into 224x224 because VGG16 takes only 224x224 pixel image
             prediction_transform = transforms.Compose([transforms.Resize(size=(224,224)),
                                                        transforms.ToTensor()
                                                        ])
             image = prediction_transform(image)[:3,:,:].unsqueeze(0)
             return image

In [33]: ### 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 loader_transfer['train'].dataset.

         def predict_breed_transfer(model,class_names,img_path):
             # load the image and return the predicted breed
             img = load_image(img_path)
             model = model.cpu()
             model.eval()
             idx = torch.argmax(model(img))
             return class_names[idx]

In [34]: for img_file in os.listdir('./images'):
             img_path = os.path.join('./images', img_file)
             predition = predict_breed_transfer(model_transfer, class_names, img_path)
             print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))

image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,          predition breed: Basset hou
image_file_name: ./images/sample_human_output.png,          predition breed: Dachshund
image_file_name: ./images/Labrador_retriever_06457.jpg,          predition breed: Labrador retri
image_file_name: ./images/Curly-coated_retriever_03896.jpg,          predition breed: Curly-coat
image_file_name: ./images/sample_cnn.png,          predition breed: Nova scotia duck tolling ret
image_file_name: ./images/Brittany_02625.jpg,          predition breed: Brittany
image_file_name: ./images/Labrador_retriever_06449.jpg,          predition breed: Plott
image_file_name: ./images/American_water_spaniel_00648.jpg,          predition breed: Cocker spa
image_file_name: ./images/sample_dog_output.png,          predition breed: Great dane
image_file_name: ./images/Labrador_retriever_06455.jpg,          predition breed: Neapolitan mas
```

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



Sample Human Output

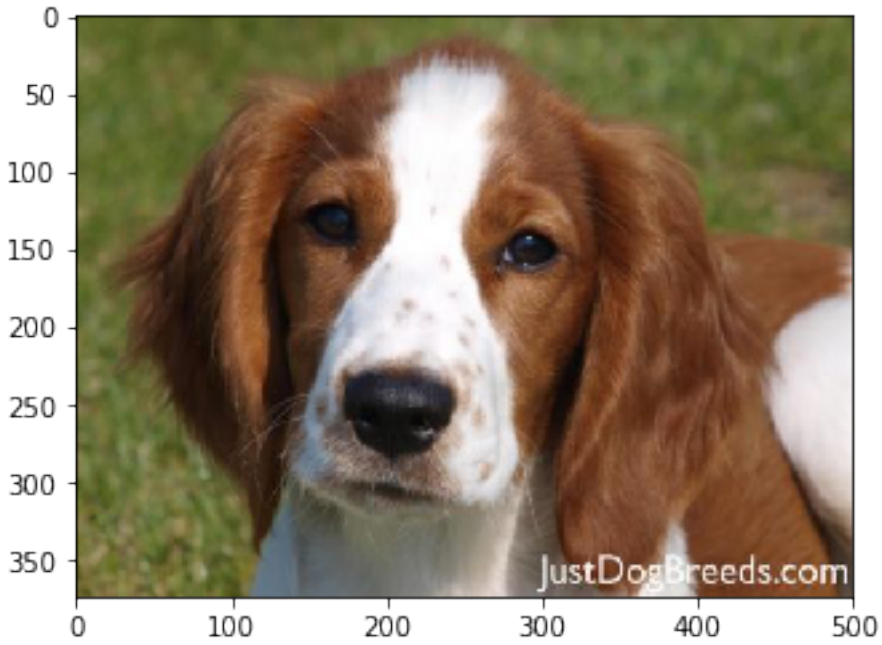
breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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.

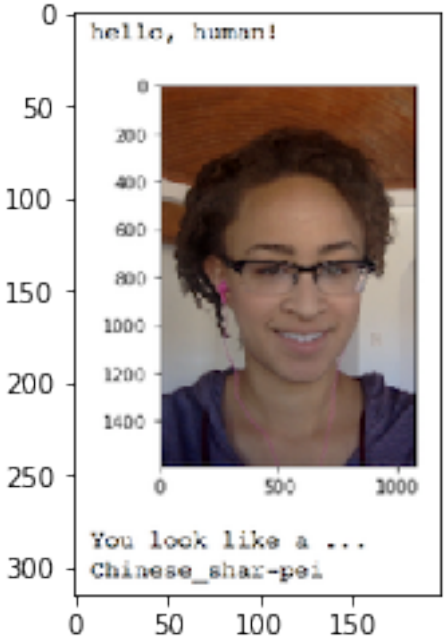
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

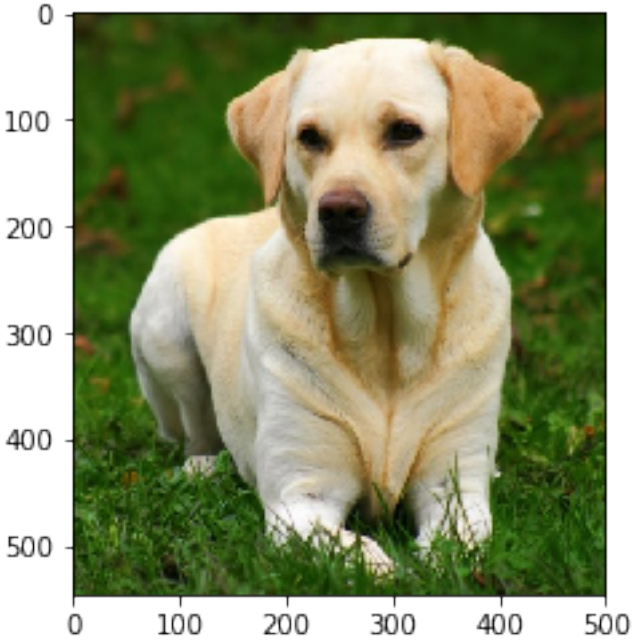
```
In [35]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         import matplotlib.pyplot as plt
         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()

             if dog_detector(img_path) is True:
                 prediction = predict_breed_transfer(model_transfer, class_names,img_path)
                 print('Dog is detected !!!!!! Ohh i think it looks {0}'.format(prediction))
             elif face_detector(img_path) > 0:
                 prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                 print("Ohh Human is detected!!!!! Human looks like {0}". format(prediction))
             else:
                 print("Error!!!! Neither Dogs are found nor Human in the given supplied image")


         for img_file in os.listdir('./images'):
             img_path = os.path.join('./images', img_file)
             run_app(img_path)
```



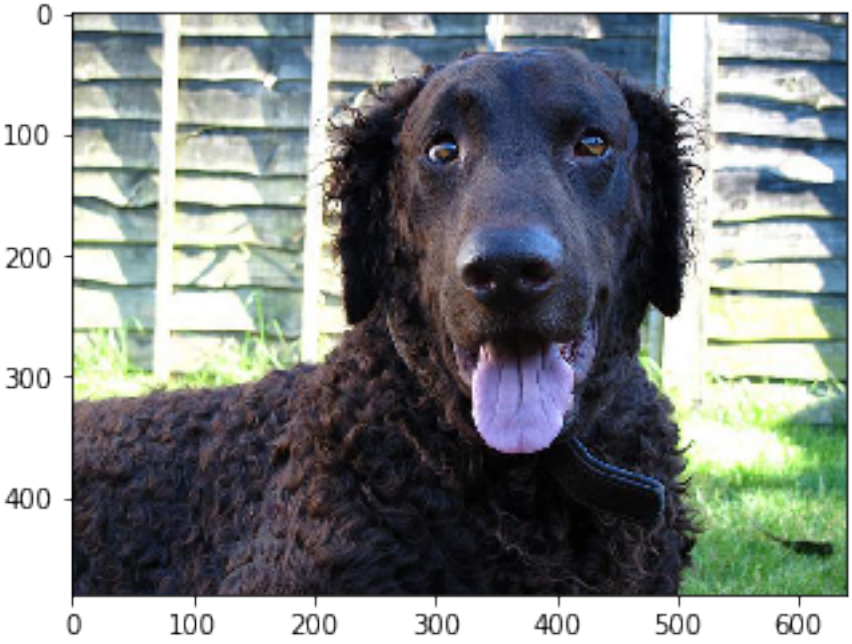

```
Dog is detected !!!!!! Ohh i think it looks Basset hound
```



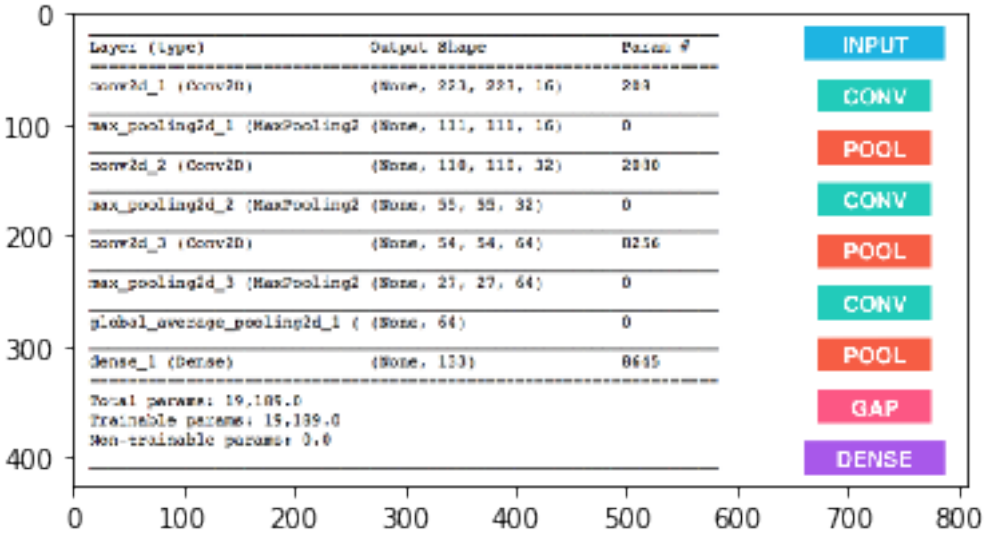
Ohh Human is detected!!!!! Human looks like Dachshund



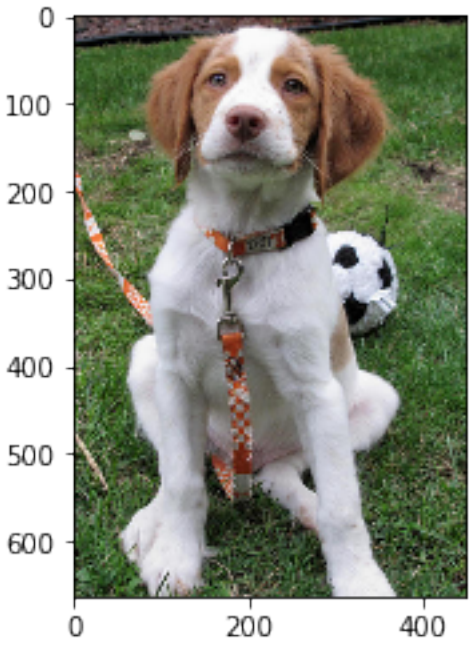
Dog is detected !!!!!! Ohh i think it looks Labrador retriever



Dog is detected !!!!!! Ohh i think it looks Curly-coated retriever



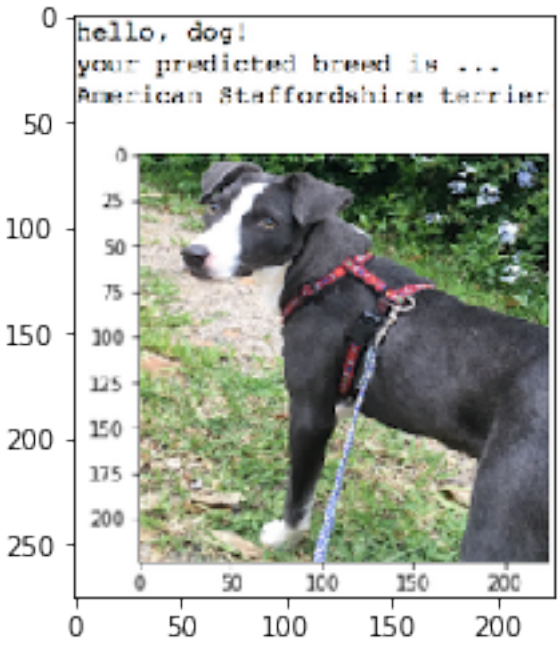
Error!!!! Neither Dogs are found nor Human in the given supplied image



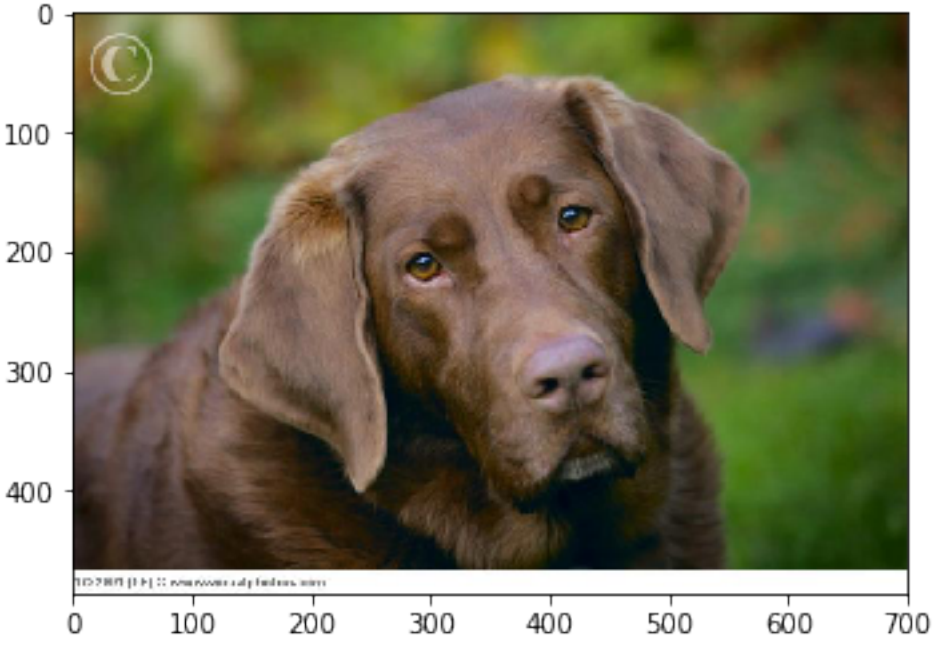
```
Dog is detected !!!!!! Ohh i think it looks Brittany
```


```
Dog is detected !!!!!! Ohh i think it looks Plott
```

Dog is detected !!!!!! Ohh i think it looks Cocker spaniel



Dog is detected !!!!!! Ohh i think it looks Great dane



```
Dog is detected !!!!!! Ohh i think it looks Neapolitan mastiff
```

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!

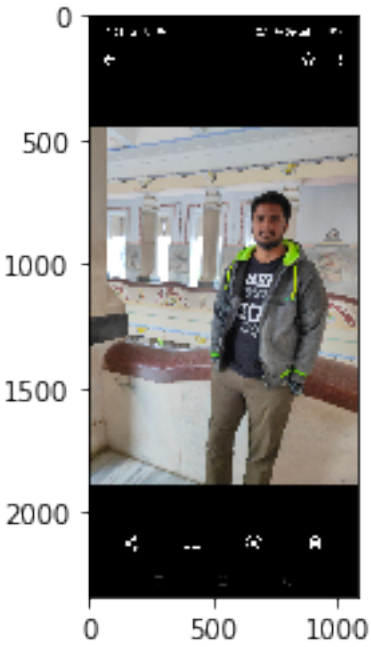
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

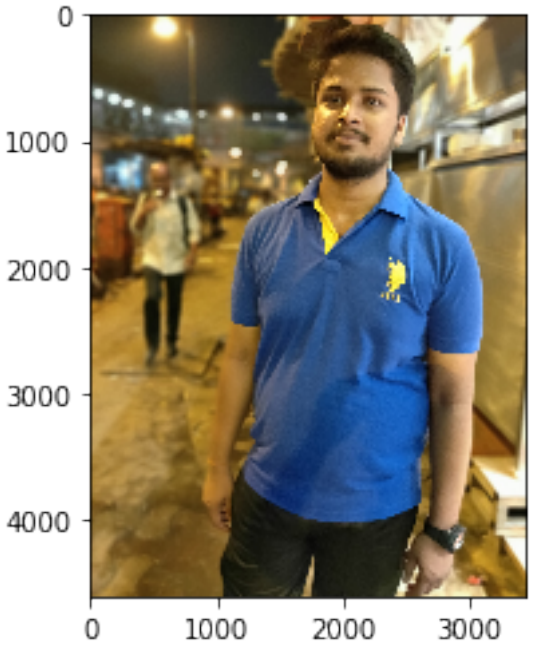
Answer: I tested my algorithm on atleast 7 images out of which 3 images is of human and 2 images is of cat and 2 is of dog. I got the results as expected. My algorithm accurately predicting the dogs breed as well as predicting the human's breed. For further testing i uploaded 2 images of cats in my dog folder. And my algorithm accurately predicts of that given image.It shows its niether human nor dogs. 1).For improving our model we can do hyperparameters changes to get better and more accurate model. 2).We can do more epochs to get even better accuracy for most complex images of dog's breed.

```
In [36]: human_files = ['./test_images/dp.jpg','./test_images/test_images_1.jpg','./test_images/
         dog_files = ['./test_images/dog_1.jpg','./test_images/dog_2.jpg','./test_images/cat_1.j
```

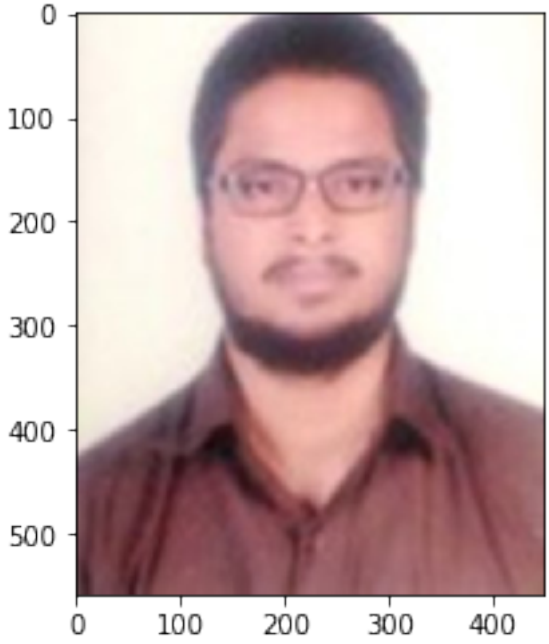
```
In [37]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[:3], dog_files[:4])):
             run_app(file)
```




```
Ohh Human is detected!!!!! Human looks like American staffordshire terrier
```

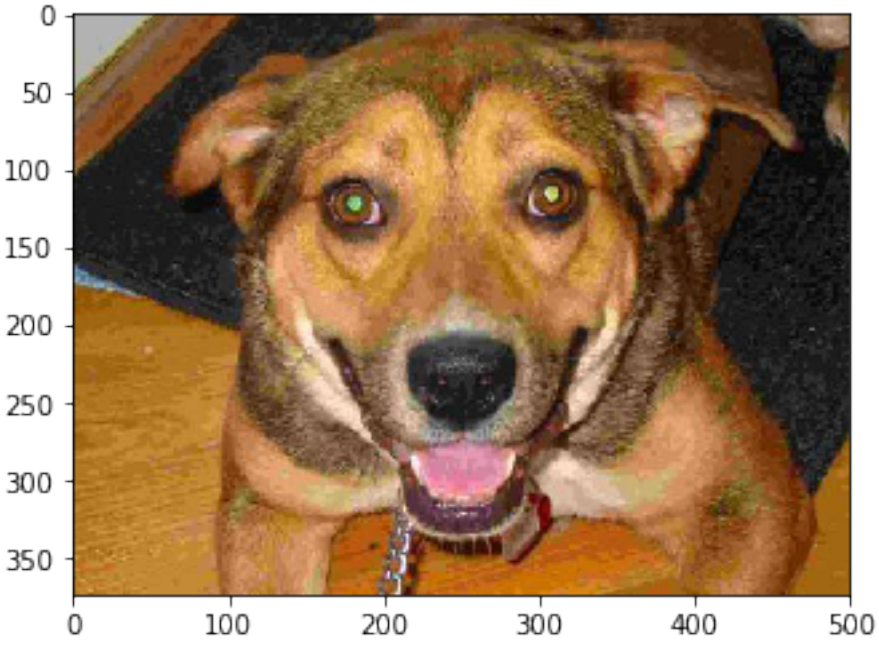


```
Ohh Human is detected!!!!! Human looks like Bull terrier
```

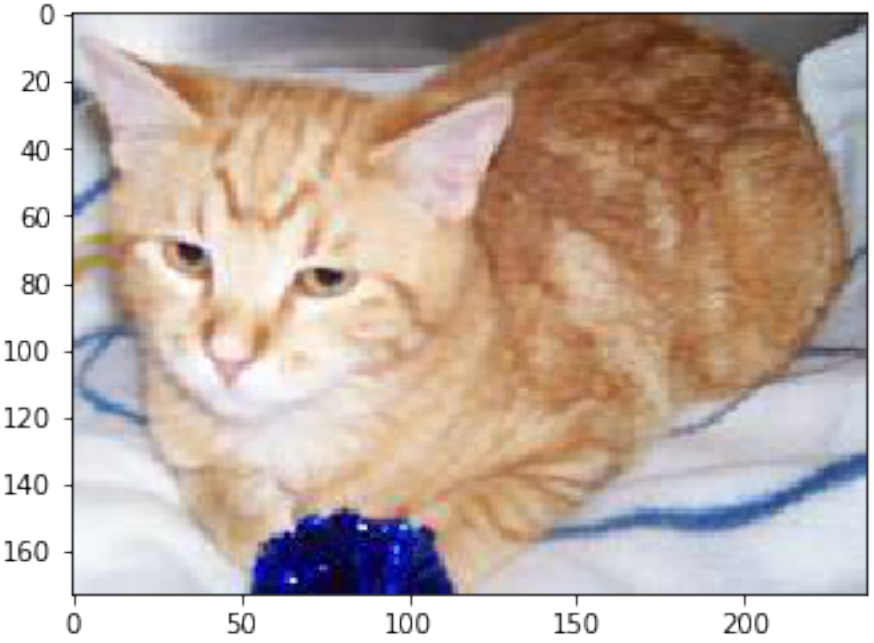



```
Ohh Human is detected!!!!! Human looks like Bichon frise
```

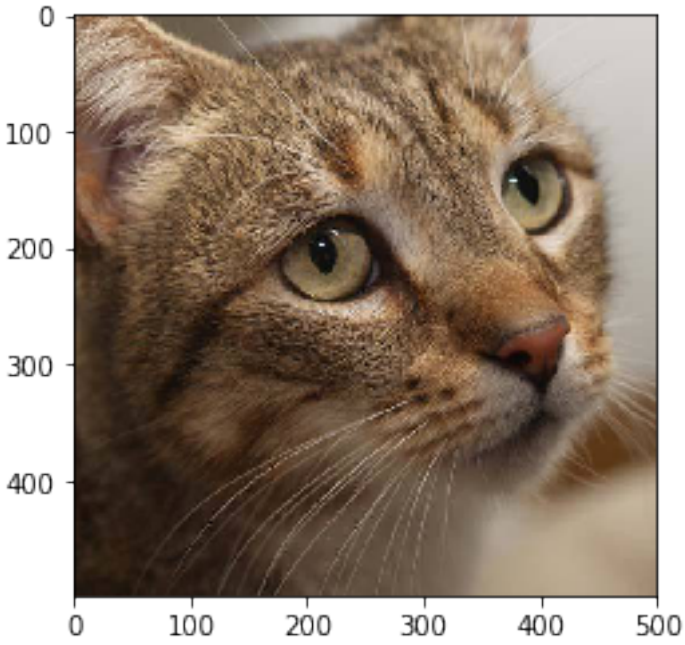
```
Dog is detected !!!!!! Ohh i think it looks Black and tan coonhound
```




```
Dog is detected !!!!!! Ohh i think it looks Plott
```



```
Error!!!! Neither Dogs are found nor Human in the given supplied image
```




```
Error!!!! Neither Dogs are found nor Human in the given supplied image
```

```
In [ ]:
```