dog_app

February 17, 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

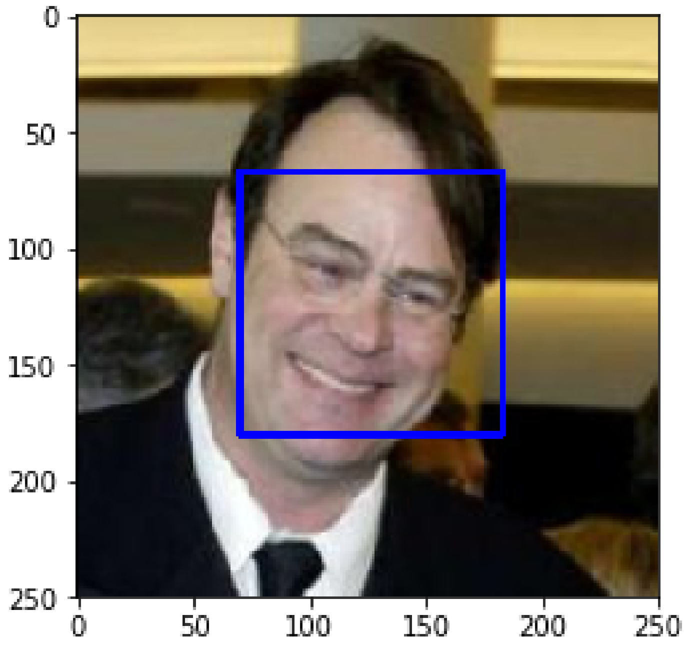
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [30]: 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 [2]: # 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 [8]: 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=0
        for i in human_files_short:
            if face_detector(i) is True:
                count+=1
        print(count/100)

        count=0
        for i in dog_files_short:
            if face_detector(i) is True:
                count+=1
        print(count/100)
```

```
0.98
0.17
```

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

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

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [3]: 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 [00:22<00:00, 24614759.04it/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 [20]: from PIL import Image
         import torchvision.transforms as transforms

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

             Args:
                 img_path: path to an image

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

             ## TODO: Complete the function.
             ## Load and pre-process an image from the given img_path
             ## Return the *index* of the predicted class for that image
             img = Image.open(img_path)


             trans= transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

             im_tensor=trans(img).unsqueeze_(0)
             im_tensor.requires_grad_(False)
             im_tensor=im_tensor.to(device='cuda')

             VGG16.eval()

             output = VGG16(im_tensor)

             pred = output.data.cpu().numpy().argmax()


             return pred  # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [5]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            VGG16_predict(img_path) in range(152,269)

            return VGG16_predict(img_path) in range(151,269) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:
percentage of dogs detected in human_files_short:0
percentage of dogs detected in dog_files_short:100

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.


        count=0
        for i in human_files_short:
            if dog_detector(i) is True:
                count+=1
        print(count/100)

        count=0
        for i in dog_files_short:
            if dog_detector(i) is True:
                count+=1
        print(count/100)
```

```
0.0
1.0
```

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

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

## Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [11]: import torch
         import os
         from torchvision import datasets
         import torchvision.transforms as transforms

         ### TODO: Write data loaders for training, validation, and test sets
```

```
## Specify appropriate transforms, and batch_sizes
traindir='/data/dog_images/train'
validdir='/data/dog_images/valid'
testdir='/data/dog_images/test'

trans_train=transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

trans=transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

batch_size=64

train_loader=torch.utils.data.DataLoader(datasets.ImageFolder(traindir,transform=trans_
val_loader=torch.utils.data.DataLoader(datasets.ImageFolder(validdir,transform=trans),b
test_loader=torch.utils.data.DataLoader(datasets.ImageFolder(testdir,transform=trans),b

loaders_scratch={'train':train_loader,'valid':val_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: Since we intend to use the same dataloaders in both training from scratch and transfer learning, as the pretrained model in transfer learning expects the input to be mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224, the images are resized accordingly. The size of the input tensor will be (64 x 3 x 224 x 224). As we can train on a GPU i chose a larger batch size for the input.

The train dataloader has been augmented with horizontal flipping and rotating of images. This process could allow the model to learn to distinguish between similar looking classes.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [6]: ## import torch.nn as nn
        import torch.nn.functional as F
        import torch.nn as nn
        # define the CNN architecture
        class Net(nn.Module):
```

```
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # max pooling layer
        self.conv4 = nn.Conv2d(64, 128,3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (64 * 4 * 4 -> 500)
        self.fc1 = nn.Linear(128 * 14 * 14, 500)
        # linear layer (500 -> 10)
        self.fc2 = nn.Linear(500, 133)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)




    def forward(self, x):
        ## Define forward behavior
        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)))
        # flatten image input

        x = x.view(-1, 128 * 14 * 14)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)

        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()
```

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

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

Answer: I chose a 4 layer convolutional neural network, with non-linearity and max pooling operation applied after each convolutional layer and a classifier at the end with two fully connected layers. The first conv. layer has 16 filters with 3 input channels, second 32 filters with 16 input channels, third 64 filters with 32 input channels and the fourth 128 filters with 64 input channels. In all the conv. layers kernel size of 3, zero padding and stride of 1 are applied. As the depth in the network increases the kernels learn better representations which contribute to the network's accuracy. A dropout layer is also applied before and after the first fully connected layer which helps the model regularize better.

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 [7]: import torch.optim as optim

In [8]: ### TODO: select loss function
        criterion_scratch =torch.nn.CrossEntropyLoss()

        params=model_scratch.parameters()
        ### TODO: select optimizer
        optimizer_scratch = optim.Adam(params,lr=0.003,weight_decay=0.01)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

```
In [22]: n_epochs=20
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

In [10]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

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

        train_loss += loss.item()*data.size(0)
        ######################
        # 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 += loss.item()*data.size(0)

        train_loss = train_loss/len(train_loader.sampler)
        valid_loss = valid_loss/len(val_loader.sampler)


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

        ## TODO: save the model if validation loss has decreased

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




        # return trained model
        return model
```

```
In [11]: import numpy as np
```

```
In [52]: # train the model
         model_scratch = train(100, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 0.017284         Validation Loss: 4.643530
Validation loss decreased (inf --> 4.643530).  Saving model ...
Epoch: 2         Training Loss: 0.016554         Validation Loss: 4.578369
Validation loss decreased (4.643530 --> 4.578369).  Saving model ...
Epoch: 3         Training Loss: 0.015642         Validation Loss: 4.408116
Validation loss decreased (4.578369 --> 4.408116).  Saving model ...
Epoch: 4         Training Loss: 0.016524         Validation Loss: 4.362533
Validation loss decreased (4.408116 --> 4.362533).  Saving model ...
Epoch: 5         Training Loss: 0.015332         Validation Loss: 4.365221
Epoch: 6         Training Loss: 0.016272         Validation Loss: 4.348300
Validation loss decreased (4.362533 --> 4.348300).  Saving model ...
Epoch: 7         Training Loss: 0.013888         Validation Loss: 4.269045
Validation loss decreased (4.348300 --> 4.269045).  Saving model ...
Epoch: 8         Training Loss: 0.014718         Validation Loss: 4.290411
Epoch: 9         Training Loss: 0.015941         Validation Loss: 4.262902
Validation loss decreased (4.269045 --> 4.262902).  Saving model ...
Epoch: 10         Training Loss: 0.014270         Validation Loss: 4.297939
Epoch: 11         Training Loss: 0.014354         Validation Loss: 4.252883
Validation loss decreased (4.262902 --> 4.252883).  Saving model ...
Epoch: 12         Training Loss: 0.015065         Validation Loss: 4.235670
Validation loss decreased (4.252883 --> 4.235670).  Saving model ...
Epoch: 13         Training Loss: 0.014091         Validation Loss: 4.292613
Epoch: 14         Training Loss: 0.015037         Validation Loss: 4.238232
Epoch: 15         Training Loss: 0.015187         Validation Loss: 4.296842
Epoch: 16         Training Loss: 0.015789         Validation Loss: 4.250259
Epoch: 17         Training Loss: 0.014584         Validation Loss: 4.253049
Epoch: 18         Training Loss: 0.014924         Validation Loss: 4.225910
Validation loss decreased (4.235670 --> 4.225910).  Saving model ...
Epoch: 19         Training Loss: 0.015125         Validation Loss: 4.193740
```

```
Validation loss decreased (4.225910 --> 4.193740).  Saving model ...
Epoch: 20         Training Loss: 0.014742         Validation Loss: 4.205581
Epoch: 21         Training Loss: 0.013656         Validation Loss: 4.182762
Validation loss decreased (4.193740 --> 4.182762).  Saving model ...
Epoch: 22         Training Loss: 0.015314         Validation Loss: 4.234131
Epoch: 23         Training Loss: 0.015912         Validation Loss: 4.212915
Epoch: 24         Training Loss: 0.015620         Validation Loss: 4.178705
Validation loss decreased (4.182762 --> 4.178705).  Saving model ...
Epoch: 25         Training Loss: 0.014455         Validation Loss: 4.173918
Validation loss decreased (4.178705 --> 4.173918).  Saving model ...
Epoch: 26         Training Loss: 0.014336         Validation Loss: 4.235647
Epoch: 27         Training Loss: 0.016123         Validation Loss: 4.170332
Validation loss decreased (4.173918 --> 4.170332).  Saving model ...
Epoch: 28         Training Loss: 0.015424         Validation Loss: 4.200591
Epoch: 29         Training Loss: 0.014624         Validation Loss: 4.210564
Epoch: 30         Training Loss: 0.013825         Validation Loss: 4.137906
Validation loss decreased (4.170332 --> 4.137906).  Saving model ...
Epoch: 31         Training Loss: 0.014143         Validation Loss: 4.177898
Epoch: 32         Training Loss: 0.015762         Validation Loss: 4.198606
Epoch: 33         Training Loss: 0.013473         Validation Loss: 4.130986
Validation loss decreased (4.137906 --> 4.130986).  Saving model ...
Epoch: 34         Training Loss: 0.015309         Validation Loss: 4.163159
Epoch: 35         Training Loss: 0.014578         Validation Loss: 4.139673
Epoch: 36         Training Loss: 0.013786         Validation Loss: 4.151301
Epoch: 37         Training Loss: 0.013036         Validation Loss: 4.154505
Epoch: 38         Training Loss: 0.016295         Validation Loss: 4.159282
Epoch: 39         Training Loss: 0.014761         Validation Loss: 4.139404
Epoch: 40         Training Loss: 0.014239         Validation Loss: 4.120108
Validation loss decreased (4.130986 --> 4.120108).  Saving model ...
Epoch: 41         Training Loss: 0.014879         Validation Loss: 4.118592
Validation loss decreased (4.120108 --> 4.118592).  Saving model ...
Epoch: 42         Training Loss: 0.014359         Validation Loss: 4.118618
Epoch: 43         Training Loss: 0.013229         Validation Loss: 4.097235
Validation loss decreased (4.118592 --> 4.097235).  Saving model ...
Epoch: 44         Training Loss: 0.014073         Validation Loss: 4.119397
Epoch: 45         Training Loss: 0.016774         Validation Loss: 4.091030
Validation loss decreased (4.097235 --> 4.091030).  Saving model ...
Epoch: 46         Training Loss: 0.015344         Validation Loss: 4.157911
Epoch: 47         Training Loss: 0.014265         Validation Loss: 4.151067
Epoch: 48         Training Loss: 0.014178         Validation Loss: 4.092861
Epoch: 49         Training Loss: 0.014639         Validation Loss: 4.075852
Validation loss decreased (4.091030 --> 4.075852).  Saving model ...
Epoch: 50         Training Loss: 0.012905         Validation Loss: 4.058304
Validation loss decreased (4.075852 --> 4.058304).  Saving model ...
Epoch: 51         Training Loss: 0.015258         Validation Loss: 4.073922
Epoch: 52         Training Loss: 0.014745         Validation Loss: 4.110152
Epoch: 53         Training Loss: 0.016130         Validation Loss: 4.071425
Epoch: 54         Training Loss: 0.014282         Validation Loss: 4.076036
```

```
Epoch: 55        Training Loss: 0.015519        Validation Loss: 4.057760
Validation loss decreased (4.058304 --> 4.057760).  Saving model ...
Epoch: 56        Training Loss: 0.014589        Validation Loss: 4.057196
Validation loss decreased (4.057760 --> 4.057196).  Saving model ...
Epoch: 57        Training Loss: 0.013825        Validation Loss: 4.067527
Epoch: 58        Training Loss: 0.014534        Validation Loss: 4.126654
Epoch: 59        Training Loss: 0.013362        Validation Loss: 4.083846
Epoch: 60        Training Loss: 0.014301        Validation Loss: 4.120347
Epoch: 61        Training Loss: 0.012919        Validation Loss: 4.005120
Validation loss decreased (4.057196 --> 4.005120).  Saving model ...
Epoch: 62        Training Loss: 0.013452        Validation Loss: 3.996721
Validation loss decreased (4.005120 --> 3.996721).  Saving model ...
Epoch: 63        Training Loss: 0.013923        Validation Loss: 4.030571
```

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

        KeyboardInterrupt                         Traceback (most recent call last)

        <ipython-input-52-c0d3078ebe65> in <module>()
          1 # train the model
          2 model_scratch = train(100, loaders_scratch, model_scratch, optimizer_scratch,
    ----> 3                       criterion_scratch, use_cuda, 'model_scratch.pt')
          4
          5 # load the model that got the best validation accuracy


        <ipython-input-48-601b5d3609f0> in train(n_epochs, loaders, model, optimizer, criterion,
         13         ###################
         14         model.train()
    ---> 15         for batch_idx, (data, target) in enumerate(loaders['train']):
         16             # move to GPU
         17             if use_cuda:


        /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in __next__(self)
        262         if self.num_workers == 0:  # same-process loading
        263             indices = next(self.sample_iter)  # may raise StopIteration
    --> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
        265             if self.pin_memory:
        266                 batch = pin_memory_batch(batch)


        /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in <listcomp>(.0)
        262         if self.num_workers == 0:  # same-process loading
        263             indices = next(self.sample_iter)  # may raise StopIteration
    --> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    101         sample = self.loader(path)
    102         if self.transform is not None:
--> 103             sample = self.transform(sample)
    104         if self.target_transform is not None:
    105             target = self.target_transform(target)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transform
     47     def __call__(self, img):
     48         for t in self.transforms:
---> 49             img = t(img)
     50         return img
     51


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transform
    173             PIL Image: Rescaled image.
    174         """
--> 175         return F.resize(img, self.size, self.interpolation)
    176
    177     def __repr__(self):


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transform
    198             ow = size
    199             oh = int(size * h / w)
--> 200             return img.resize((ow, oh), interpolation)
    201         else:
    202             oh = size


    /opt/conda/lib/python3.6/site-packages/PIL/Image.py in resize(self, size, resample, box)
   1763         self.load()
   1764
-> 1765         return self._new(self.im.resize(size, resample, box))
   1766
   1767     def rotate(self, angle, resample=NEAREST, expand=0, center=None,


    KeyboardInterrupt:
```

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 [12]: 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))
```

```
In [54]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 4.019897



Test Accuracy: 10% (87/836)
```

## 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 [12]: datasets.ImageFolder(traindir,transform=trans).classes
```

```
Out[12]: ['001.Affenpinscher',
          '002.Afghan_hound',
          '003.Airedale_terrier',
          '004.Akita',
          '005.Alaskan_malamute',
          '006.American_eskimo_dog',
          '007.American_foxhound',
          '008.American_staffordshire_terrier',
          '009.American_water_spaniel',
          '010.Anatolian_shepherd_dog',
          '011.Australian_cattle_dog',
          '012.Australian_shepherd',
          '013.Australian_terrier',
          '014.Basenji',
          '015.Basset_hound',
          '016.Beagle',
          '017.Bearded_collie',
          '018.Beauceron',
          '019.Bedlington_terrier',
          '020.Belgian_malinois',
          '021.Belgian_sheepdog',
          '022.Belgian_tervuren',
          '023.Bernese_mountain_dog',
          '024.Bichon_frise',
          '025.Black_and_tan_coonhound',
          '026.Black_russian_terrier',
          '027.Bloodhound',
          '028.Bluetick_coonhound',
          '029.Border_collie',
          '030.Border_terrier',
          '031.Borzoi',
          '032.Boston_terrier',
          '033.Bouvier_des_flandres',
          '034.Boxer',
          '035.Boykin_spaniel',
          '036.Briard',
          '037.Brittany',
          '038.Brussels_griffon',
          '039.Bull_terrier',
'040.Bulldog',
'041.Bullmastiff',
'042.Cairn_terrier',
'043.Canaan_dog',
'044.Cane_corso',
'045.Cardigan_welsh_corgi',
'046.Cavalier_king_charles_spaniel',
'047.Chesapeake_bay_retriever',
'048.Chihuahua',
'049.Chinese_crested',
'050.Chinese_shar-pei',
'051.Chow_chow',
'052.Clumber_spaniel',
'053.Cocker_spaniel',
'054.Collie',
'055.Curly-coated_retriever',
'056.Dachshund',
'057.Dalmatian',
'058.Dandie_dinmont_terrier',
'059.Doberman_pinscher',
'060.Dogue_de_bordeaux',
'061.English_cocker_spaniel',
'062.English_setter',
'063.English_springer_spaniel',
'064.English_toy_spaniel',
'065.Entlebucher_mountain_dog',
'066.Field_spaniel',
'067.Finnish_spitz',
'068.Flat-coated_retriever',
'069.French_bulldog',
'070.German_pinscher',
'071.German_shepherd_dog',
'072.German_shorthaired_pointer',
'073.German_wirehaired_pointer',
'074.Giant_schnauzer',
'075.Glen_of_imaal_terrier',
'076.Golden_retriever',
'077.Gordon_setter',
'078.Great_dane',
'079.Great_pyrenees',
'080.Greater_swiss_mountain_dog',
'081.Greyhound',
'082.Havanese',
'083.Ibizan_hound',
'084.Icelandic_sheepdog',
'085.Irish_red_and_white_setter',
'086.Irish_setter',
'087.Irish_terrier',
          '088.Irish_water_spaniel',
          '089.Irish_wolfhound',
          '090.Italian_greyhound',
          '091.Japanese_chin',
          '092.Keeshond',
          '093.Kerry_blue_terrier',
          '094.Komondor',
          '095.Kuvasz',
          '096.Labrador_retriever',
          '097.Lakeland_terrier',
          '098.Leonberger',
          '099.Lhasa_apso',
          '100.Lowchen',
          '101.Maltese',
          '102.Manchester_terrier',
          '103.Mastiff',
          '104.Miniature_schnauzer',
          '105.Neapolitan_mastiff',
          '106.Newfoundland',
          '107.Norfolk_terrier',
          '108.Norwegian_buhund',
          '109.Norwegian_elkhound',
          '110.Norwegian_lundehund',
          '111.Norwich_terrier',
          '112.Nova_scotia_duck_tolling_retriever',
          '113.Old_english_sheepdog',
          '114.Otterhound',
          '115.Papillon',
          '116.Parson_russell_terrier',
          '117.Pekingese',
          '118.Pembroke_welsh_corgi',
          '119.Petit_basset_griffon_vendeen',
          '120.Pharaoh_hound',
          '121.Plott',
          '122.Pointer',
          '123.Pomeranian',
          '124.Poodle',
          '125.Portuguese_water_dog',
          '126.Saint_bernard',
          '127.Silky_terrier',
          '128.Smooth_fox_terrier',
          '129.Tibetan_mastiff',
          '130.Welsh_springer_spaniel',
          '131.Wirehaired_pointing_griffon',
          '132.Xoloitzcuintli',
          '133.Yorkshire_terrier']
```

```
In [14]: ## TODO: Specify data loaders
```

```
         batch_size=64

         train_loader=torch.utils.data.DataLoader(datasets.ImageFolder(traindir,transform=trans_
         val_loader=torch.utils.data.DataLoader(datasets.ImageFolder(validdir,transform=trans),b
         test_loader=torch.utils.data.DataLoader(datasets.ImageFolder(testdir,transform=trans),b
```

```
In [15]: loaders_transfer={'train':train_loader,'valid':val_loader,'test':test_loader}
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [6]: import torchvision.models as models
        import torch
        import torch.nn as nn

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

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

        # move model to GPU if CUDA is available
```

```
In [17]: model_transfer.classifier[6].in_features
```

```
Out[17]: 4096
```

```
In [9]: for param in model_transfer.features.parameters():
            param.requires_grad = False
```

```
In [7]: n_inputs = model_transfer.classifier[6].in_features
        last_layer = nn.Linear(n_inputs, 133)


        model_transfer.classifier[6] = last_layer

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

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: A pretrained VGG16 model is chosen for finetuning to our dataset. We set all the parameters of the model unchanged. A final layer is added in place of the pretrained model's final layer which needs to be finetuned. Final layer's outputs are set equal to the total number of classes in our dataset. As the pretrained model has already learnt the basic representations it's enough if we train the final few layers. This architecture thus can perform well on the current dataset.

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 [21]: params=model_transfer.classifier.parameters()

         import torch.optim as optim
         criterion_transfer = torch.nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(params,lr=0.001,weight_decay=0.01)
```

```
In [6]: pwd
```

```
Out[6]: '/home/workspace/dog_project'
```

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 [38]: # train the model
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
```

```
Epoch: 1         Training Loss: 0.009629        Validation Loss: 1.283092
Validation loss decreased (inf --> 1.283092).  Saving model ...
Epoch: 2         Training Loss: 0.005606        Validation Loss: 1.170218
Validation loss decreased (1.283092 --> 1.170218).  Saving model ...
Epoch: 3         Training Loss: 0.004945        Validation Loss: 1.364561
Epoch: 4         Training Loss: 0.005267        Validation Loss: 1.087081
Validation loss decreased (1.170218 --> 1.087081).  Saving model ...
Epoch: 5         Training Loss: 0.005359        Validation Loss: 1.127590
Epoch: 6         Training Loss: 0.006592        Validation Loss: 0.989095
Validation loss decreased (1.087081 --> 0.989095).  Saving model ...
Epoch: 7         Training Loss: 0.004175        Validation Loss: 0.912134
Validation loss decreased (0.989095 --> 0.912134).  Saving model ...
Epoch: 8         Training Loss: 0.006723        Validation Loss: 1.101097
Epoch: 9         Training Loss: 0.004964        Validation Loss: 0.976401
Epoch: 10         Training Loss: 0.005434        Validation Loss: 0.932008
Epoch: 11         Training Loss: 0.003914        Validation Loss: 0.947224
Epoch: 12         Training Loss: 0.006072        Validation Loss: 0.900483
Validation loss decreased (0.912134 --> 0.900483).  Saving model ...
Epoch: 13         Training Loss: 0.005199        Validation Loss: 0.878920
Validation loss decreased (0.900483 --> 0.878920).  Saving model ...
Epoch: 14         Training Loss: 0.005058        Validation Loss: 0.984050
Epoch: 15         Training Loss: 0.006189        Validation Loss: 0.957044
Epoch: 16         Training Loss: 0.004444        Validation Loss: 0.940076
Epoch: 17         Training Loss: 0.003323        Validation Loss: 0.870056
Validation loss decreased (0.878920 --> 0.870056).  Saving model ...
Epoch: 18         Training Loss: 0.006195        Validation Loss: 0.982277
```

```
Epoch: 19        Training Loss: 0.004738        Validation Loss: 1.057192
Epoch: 20        Training Loss: 0.005052        Validation Loss: 1.036487
```

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

        FileNotFoundError                         Traceback (most recent call last)

        <ipython-input-38-205e5b541c40> in <module>()
          3
          4 # load the model that got the best validation accuracy (uncomment the line below)
    ----> 5 model_transfer.load_state_dict(torch.load('model_transfer.pt'))


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


        FileNotFoundError: [Errno 2] No such file or directory: 'model_transfer.pt'
```

```
In [8]: # load the model that got the best validation accuracy (uncomment the line below)
        model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [40]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.802813


Test Accuracy: 75% (630/836)
```

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 [25]: ### 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 datasets.ImageFolder(traindir,tra

def predict_breed_transfer(img_path):

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

    trans= transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

    im_tensor=trans(img).unsqueeze_(0)
    im_tensor.requires_grad_(False)

    im_tensor=im_tensor.to(device='cuda')

    model_transfer.eval()

    output = model_transfer(im_tensor)

    pred = output.data.cpu().numpy().argmax()

    return class_names[pred]
```

## Step 5: Write your Algorithm
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

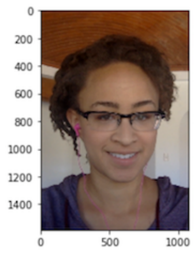
You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [27]: import matplotlib.pyplot as plt
         def imshow(img_path, title="", normalize=True):
             fig, ax = plt.subplots()
             ax.imshow(plt.imread(img_path))
             plt.title(title)
             plt.show()
```

```
hello, human!
```



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

Sample Human Output

```
In [34]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):

             if dog_detector(img_path):
                 breed = predict_breed_transfer(img_path)
                 imshow(img_path, "Hello, dog! \n your predicted breed is ...\n{0} ".format(bree

             elif face_detector(img_path):
                 breed = predict_breed_transfer(img_path)
                 imshow(img_path, "Hello, human \n you look like a...\n{0}".format(breed))

             else:
                 imshow(img_path, "Neither dog nor human were detected")



             ## handle cases for a human face, dog, and neither
```

## 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.

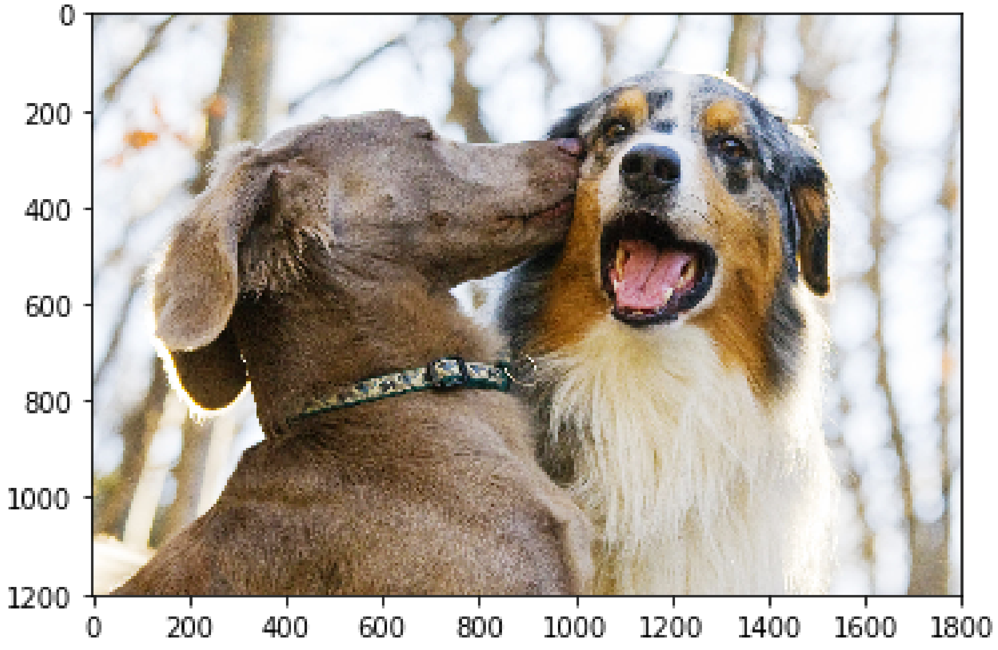
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

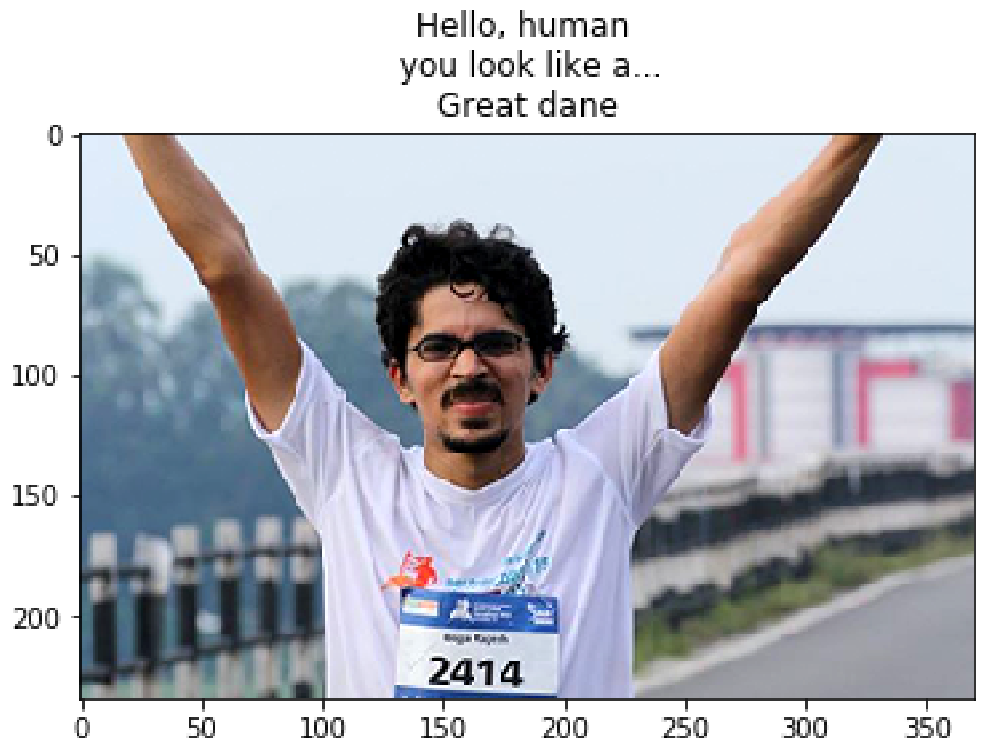
Answer: The model does decently well on my input images.

1. We can use a different pretrained model which has a better accuracy on ImageNet dataset.
2. We can train for more number of epochs.
3. We can try using more augmentations on the dataset.

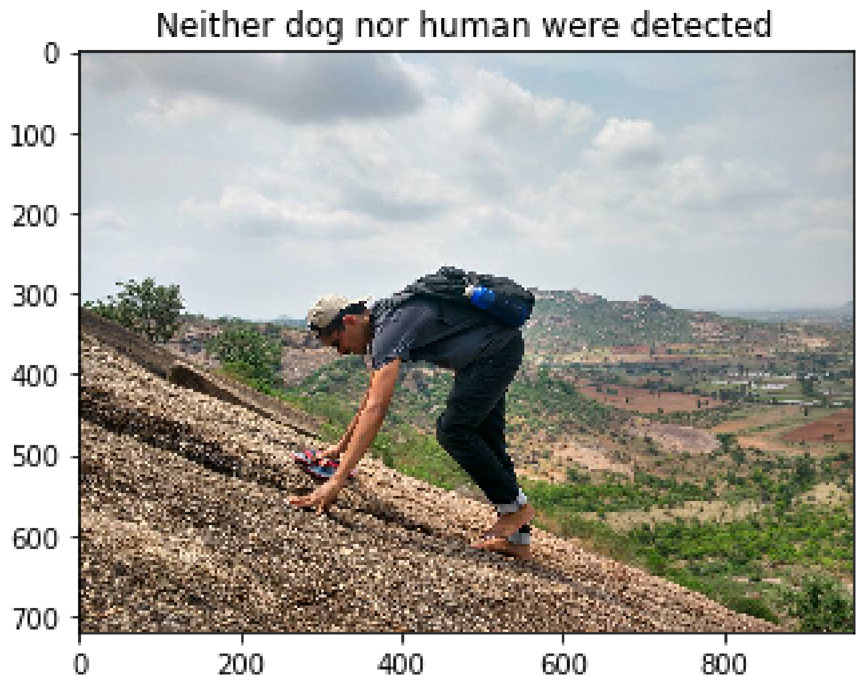
```
In [35]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         # for file in np.hstack((human_files[:3], dog_files[:3])):
         run_app('my_pics/1800x1200_surprises_about_dogs_and_cats_slideshow.jpg')
         run_app('my_pics/Official_Photos_2015_0014_SCM_Gallery_11.png')
         run_app('my_pics/19146277_1555555251142638_6379125548404873215_n.jpg')
         run_app('my_pics/Image0349.jpg')
         run_app('my_pics/download (6).jpg')
         run_app('my_pics/maxresdefault.jpg')
```

Hello, dog!
your predicted breed is ...
Smooth fox terrier



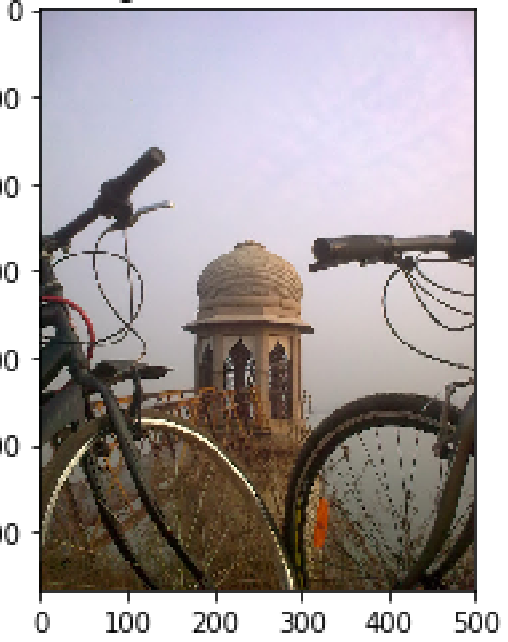


Hello, human
you look like a...
Great dane



Neither dog nor human were detected

Neither dog nor human were detected



Neither dog nor human were detected

Hello, dog!
your predicted breed is ...
American eskimo dog
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200