dog_app

September 5, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

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

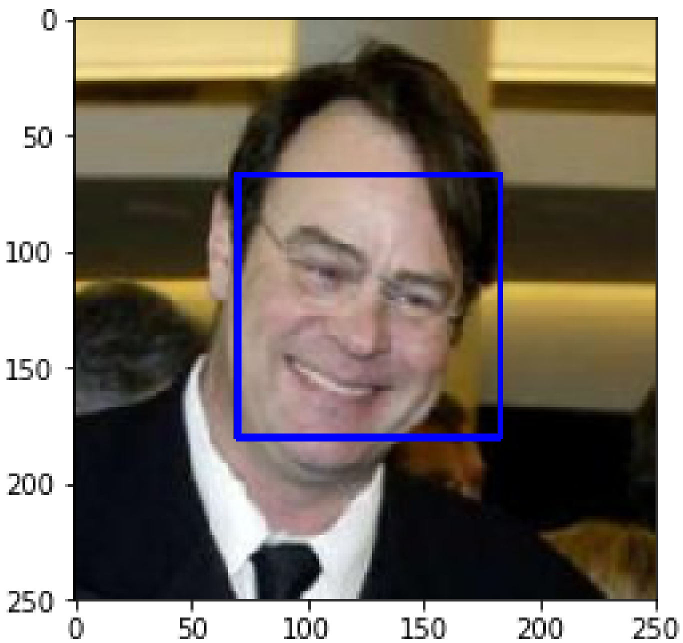
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: 98% and 83%

```
In [4]: #from tqdm import tqdm
        import time

        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.
        print ("Starting loop to detect first 100 Human Images")
        count_humanFaces=0
        for img in human_files_short:
            if face_detector(img):
                #print ('Human face is detected = ',face_detector(img))
                count_humanFaces+=1

        print ("Overall performance in Human files: ", 100*count_humanFaces/len(human_files_shor
```

```
Starting loop to detect first 100 Human Images
Overall performance in Human files:  98.0
```

```
In [5]: print ("\n\nStarting loop to detect first 100 dog Images")
        count_humanFaces=0
```

```
for img in dog_files_short:
    if face_detector(img):
        #print ('Human face is detected = ',face_detector(img))
        count_humanFaces+=1

print ("Overall performance in dog_files: ", 100*(len(dog_files_short)-count_humanFaces)
```

```
Starting loop to detect first 100 dog Images
Overall performance in dog_files:  83.0
```

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()

        if not use_cuda:
            print('CUDA is not available.  Training on CPU ...')
        else:
            print('CUDA is available!  Training on GPU ...')

        if use_cuda:
            VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:42<00:00, 12907664.56it/s]


CUDA is available!  Training on GPU ...
```

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 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 color (BGR) image
            bgr_img = Image.open(img_path)



            # convert to image to tensor
            transform = transforms.Compose([
            transforms.Resize(size=(244, 244)),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
            ])

            img_tensor=transform(bgr_img)
```

```
        index_pred=VGG16(img_tensor.unsqueeze(0).cuda()).cpu()
        index_pred=index_pred.data.numpy().argmax()
        #print (index_pred)

        ## Load and pre-process an image from the given img_path
        ## Return the *index* of the predicted class for that image

        return index_pred # predicted class index
```

```
In [9]: print (VGG16_predict(human_files[0]))
```

```
906
```

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 [10]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.

             index=VGG16_predict(img_path)
             if index >=151 and index<=268:
                 return True
             else:
                 return False
```

```
In [11]: print(dog_detector(dog_files_short[1]))
```

```
True
```

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

```
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: 2% detected dogs in human_files_short, 100% dogs detected in dog_files_short.

```
In [21]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         print ("Starting loop to detect first 100 Human Images")
         count_dogs=0
         for img in human_files_short:
             if dog_detector(img):
                 #print ('Human face is detected = ',face_detector(img))
                 count_dogs+=1

         print ("Overall percentage of dogs detected in Human files: ", 100*count_dogs/len(human
```

```
Starting loop to detect first 100 Human Images
Overall percentage of dogs detected in Human files:  2.0
```

```
In [22]: print ("\n\nStarting loop to detect first 100 dog Images")
         count_dogs=0
         for img in dog_files_short:
             if dog_detector(img):
                 count_dogs+=1

         print ("Overall percentage of dogs detected in dog_files: ", 100*count_dogs/len(dog_fil
```

```
Starting loop to detect first 100 dog Images
Overall percentage of dogs detected in dog_files:  100.0
```

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 [23]: ### (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 [24]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms
         ### TODO: Write data loaders for training, validation, and test sets

         # number of subprocesses
         num_workers=0

         batch_size=20
         num_classes=133
         data_dir = '/data/dog_images/'
         Image_database={x: datasets.ImageFolder(os.path.join(data_dir, x), transforms_dogImages
                           for x in ['train', 'valid', 'test']}
         class_names = Image_database['train'].classes
         num_classes = len(class_names)
```

```
# Create data loaders (train, valid, test)
data_loaders = {x: torch.utils.data.DataLoader(Image_database[x], batch_size=batch_size
                                               shuffle=True, num_workers=num_workers)
               for x in ['train', 'valid', 'test']}
```

In [25]:
```
print("Number of classes:", num_classes)
print("\nClass names: \n\n", class_names)
```

```
Number of classes: 133

Class names:

 ['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mal
```

In [26]:
```
data_dir = '/data/dog_images/'
train = os.path.join(data_dir, 'train/')
valid = os.path.join(data_dir, 'valid/')
test = os.path.join(data_dir, 'test/')
print(train)
print(valid)
print(test)
```

```
/data/dog_images/train/
/data/dog_images/valid/
/data/dog_images/test/
```

In [27]:
```
## Specify appropriate transforms, and batch_sizes
transforms_dogImages = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0
                   'valid': transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0
                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0
                  }

train_data = datasets.ImageFolder(train, transform=transforms_dogImages['train'])
valid_data = datasets.ImageFolder(valid, transform=transforms_dogImages['valid'])
test_data = datasets.ImageFolder(test, transform=transforms_dogImages['test'])

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worke
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
```

```
loaders_from_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:Applied RandomResizedCrop and RandomHorizontalFlip to training data. This should prevent overfitting gdue to randomness. Further, resized to 256 and convert/crop in center to make 224x224. Kept the Normalize mean and std to 0.5 values to begin with, may tune these values later on.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [28]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        ## Adding first layer
        self.conv1=nn.Conv2d(3, 32, 3, stride = 2, padding = 1)
        ## Adding second layer
        self.conv2=nn.Conv2d(32, 64, 3, stride = 2, padding = 1)
        ## Adding third layer
        self.conv3=nn.Conv2d(64, 128, 3, padding = 1)

        # Max pooling layers:
        self.pool=nn.MaxPool2d(2,2)

        ## Now lets create a fully Connected layer
        ## Adding fully connected layer 1
        self.fc1 = nn.Linear(7*7*128, 500)
        ## Adding fully connected layer 2
        self.fc2 = nn.Linear(500, num_classes)

        # drop-out layer definition
        self.dropout = nn.Dropout(0.2)
```

```
    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)

        ## Now lets flatten the images

        x=x.view(-1, 7*7*128)

        x=self.dropout(x)

        x=F.relu(self.fc1(x))
        x=self.dropout(x)
        x=self.fc2(x)

        return x

#-#-# You do 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=0.2)
)
```

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

Answer: Added 3 convolutional layers with Max pool layer after Relu for each convolutional layer. Added two fully connected layers and a dropout of 20% after each fully connected layer. Applied Relu to one fully connected layer and then dropout layer. Final connected layer output is passed to finalize the output.

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 [29]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.03)
```

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 [30]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

        ###################
        # 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


            # optimized gradient, setting to zero
            optimizer.zero_grad()

            ## Model output
```

```
        output=model(data)

        ## Loss calculation
        loss=criterion(output, target)

        ## Backward propagation for the loss

        loss.backward()

        ## Gradient
        optimizer.step()

        ## Training Loss calculation

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

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

        ## Model output

        output=model(data)

        ## Loss calculation
        loss=criterion(output, target)

        ## Validation Loss calculation

        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

    ## Checking if current validation loss is smaller than previous validation loss
```

```
            ## initial value set to infinity so this loop will execute for first time
            if valid_loss < valid_loss_min:
                torch.save(model.state_dict(), save_path)
                print("Previous Validation Loss: {:.3f}".format(valid_loss_min))
                print("Current Validation Loss: {:.3f}".format(valid_loss))

                ## Settign the current validation loss to the minimum, next iteration will
                valid_loss_min=valid_loss


        # return trained model
        return model
```

In [31]:
```
# train the model
model_scratch = train(100, loaders_from_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1        Training Loss: 4.883040         Validation Loss: 4.865408
Previous Validation Loss: inf
Current Validation Loss: 4.865
Epoch: 2        Training Loss: 4.857053         Validation Loss: 4.810315
Previous Validation Loss: 4.865
Current Validation Loss: 4.810
Epoch: 3        Training Loss: 4.790599         Validation Loss: 4.680253
Previous Validation Loss: 4.810
Current Validation Loss: 4.680
Epoch: 4        Training Loss: 4.697958         Validation Loss: 4.535346
Previous Validation Loss: 4.680
Current Validation Loss: 4.535
Epoch: 5        Training Loss: 4.603085         Validation Loss: 4.457339
Previous Validation Loss: 4.535
Current Validation Loss: 4.457
Epoch: 6        Training Loss: 4.549881         Validation Loss: 4.402323
Previous Validation Loss: 4.457
Current Validation Loss: 4.402
Epoch: 7        Training Loss: 4.510945         Validation Loss: 4.346046
Previous Validation Loss: 4.402
Current Validation Loss: 4.346
Epoch: 8        Training Loss: 4.468969         Validation Loss: 4.291792
Previous Validation Loss: 4.346
Current Validation Loss: 4.292
Epoch: 9        Training Loss: 4.415800         Validation Loss: 4.255110
Previous Validation Loss: 4.292
Current Validation Loss: 4.255
Epoch: 10         Training Loss: 4.364923         Validation Loss: 4.225342
```

```
Previous Validation Loss: 4.255
Current Validation Loss: 4.225
Epoch: 11         Training Loss: 4.319691         Validation Loss: 4.130605
Previous Validation Loss: 4.225
Current Validation Loss: 4.131
Epoch: 12         Training Loss: 4.284359         Validation Loss: 4.103869
Previous Validation Loss: 4.131
Current Validation Loss: 4.104
Epoch: 13         Training Loss: 4.245298         Validation Loss: 4.096952
Previous Validation Loss: 4.104
Current Validation Loss: 4.097
Epoch: 14         Training Loss: 4.205200         Validation Loss: 4.031079
Previous Validation Loss: 4.097
Current Validation Loss: 4.031
Epoch: 15         Training Loss: 4.187174         Validation Loss: 4.003117
Previous Validation Loss: 4.031
Current Validation Loss: 4.003
Epoch: 16         Training Loss: 4.133278         Validation Loss: 3.976982
Previous Validation Loss: 4.003
Current Validation Loss: 3.977
Epoch: 17         Training Loss: 4.078130         Validation Loss: 3.940412
Previous Validation Loss: 3.977
Current Validation Loss: 3.940
Epoch: 18         Training Loss: 4.061856         Validation Loss: 3.870653
Previous Validation Loss: 3.940
Current Validation Loss: 3.871
Epoch: 19         Training Loss: 3.993440         Validation Loss: 3.875874
Epoch: 20         Training Loss: 3.951913         Validation Loss: 3.896111
```

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

        KeyboardInterrupt                         Traceback (most recent call last)

        <ipython-input-31-7378572b6810> in <module>()
          1 # train the model
          2 model_scratch = train(100, loaders_from_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-30-edcae5de29b4> in train(n_epochs, loaders, model, optimizer, criterion,
         17         ###################
         18         model.train()
    ---> 19         for batch_idx, (data, target) in enumerate(loaders['train']):
         20             # move to GPU
```

```
     21             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/
     99         """
    100         path, target = self.samples[index]
--> 101         sample = self.loader(path)
    102         if self.transform is not None:
    103             sample = self.transform(sample)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    145         return accimage_loader(path)
    146     else:
--> 147         return pil_loader(path)
    148
    149


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    128     with open(path, 'rb') as f:
    129         img = Image.open(f)
--> 130         return img.convert('RGB')
    131
    132


    /opt/conda/lib/python3.6/site-packages/PIL/Image.py in convert(self, mode, matrix, dithe
    890         """
    891
--> 892         self.load()
    893
```

```
    894         if not mode and self.mode == "P":


    /opt/conda/lib/python3.6/site-packages/PIL/ImageFile.py in load(self)
    233
    234                             b = b + s
--> 235                             n, err_code = decoder.decode(b)
    236                             if n < 0:
    237                                 break


    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 [ ]: 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(loaders_from_scratch, model_scratch, criterion_scratch, use_cuda)
```

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 [13]: ## TODO: Specify data loaders
         import torch.nn as nn
         import torch.nn.functional as F
         import os
         from torchvision import datasets
         import torchvision.transforms as transforms

         # number of subprocesses
         num_workers=0

         batch_size=20
         num_classes=133

         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/')
         print(train_dir)
         print(valid_dir)
         print(test_dir)




         ## Specify appropriate transforms, and batch_sizes


         transforms_dogImages = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                             transforms.RandomHorizontalFlip(),
                                             transforms.ToTensor(),
                                             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0
                       'valid': transforms.Compose([transforms.Resize(256),
```

```
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0
                  'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0
                 }

Image_database={x: datasets.ImageFolder(os.path.join(data_dir, x), transforms_dogImages
                 for x in ['train', 'valid', 'test']}
class_names = Image_database['train'].classes
num_classes = len(class_names)

train_data = datasets.ImageFolder(train_dir, transform=transforms_dogImages['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=transforms_dogImages['valid'])
test_data = datasets.ImageFolder(test_dir, transform=transforms_dogImages['test'])

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worke
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

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

```
/data/dog_images/train/
/data/dog_images/valid/
/data/dog_images/test/
```

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

         ## TODO: Specify model architecture

         model_transfer = models.vgg16(pretrained=True)


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

In [15]:
```python
## input from sixth layer

n_inputs=model_transfer.classifier[6].in_features

# Last layer
last_layer = nn.Linear(n_inputs, len(class_names))

model_transfer.classifier[6] = last_layer

if use_cuda:
    model_transfer.cuda()

# See if last layer producing expected number of outputs
print (model_transfer.classifier[6].out_features)
print (model_transfer)
```

```
133
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, 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.

In [ ]:

Answer: Chose the model VGG16 for the CNN architecture for pretrained network. This should be helpful for the current problem due to the large dataset. Going to use the pre trained weights (froze weights) and initializing the weight for Fully connected layers.

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 [16]: import torch.optim as optim
         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.01)
```

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 [17]: # the following import is required for training to be robust to truncated images
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

```
train_loss = 0.0
valid_loss = 0.0

###################
# 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


    # optimized gradient, setting to zero
    optimizer.zero_grad()

    ## Model output

    output=model(data)

    ## Loss calculation
    loss=criterion(output, target)

    ## Backward propagation for the loss

    loss.backward()

    ## Gradient
    optimizer.step()

    ## Training Loss calculation

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

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

    ## Model output
```

```
                output=model(data)

                ## Loss calculation
                loss=criterion(output, target)

                ## Validation Loss calculation

                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

            ## Checking if current validation loss is smaller than previous validation loss
            ## initial value set to infinity so this loop will execute for first time
            if valid_loss < valid_loss_min:
                torch.save(model.state_dict(), save_path)
                print("Previous Validation Loss: {:.3f}".format(valid_loss_min))
                print("Current Validation Loss: {:.3f}".format(valid_loss))

                ## Settign the current validation loss to the minimum, next iteration will
                valid_loss_min=valid_loss


        # return trained model
        return model
```

In [18]:
```
# train the model
n_epochs=20
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

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

```
Epoch: 1         Training Loss: 2.251441         Validation Loss: 0.721650
Previous Validation Loss: inf
Current Validation Loss: 0.722
Epoch: 2         Training Loss: 1.313978         Validation Loss: 0.626662
Previous Validation Loss: 0.722
Current Validation Loss: 0.627
Epoch: 3         Training Loss: 1.155656         Validation Loss: 0.564881
Previous Validation Loss: 0.627
```

```
Current Validation Loss: 0.565
Epoch: 4        Training Loss: 1.076866         Validation Loss: 0.544374
Previous Validation Loss: 0.565
Current Validation Loss: 0.544
Epoch: 5        Training Loss: 1.015363         Validation Loss: 0.526041
Previous Validation Loss: 0.544
Current Validation Loss: 0.526
Epoch: 6        Training Loss: 0.996722         Validation Loss: 0.525631
Previous Validation Loss: 0.526
Current Validation Loss: 0.526
Epoch: 7        Training Loss: 0.944383         Validation Loss: 0.506900
Previous Validation Loss: 0.526
Current Validation Loss: 0.507
Epoch: 8        Training Loss: 0.898475         Validation Loss: 0.521455
Epoch: 9        Training Loss: 0.893969         Validation Loss: 0.488433
Previous Validation Loss: 0.507
Current Validation Loss: 0.488
Epoch: 10        Training Loss: 0.850616         Validation Loss: 0.500852
Epoch: 11        Training Loss: 0.840560         Validation Loss: 0.482078
Previous Validation Loss: 0.488
Current Validation Loss: 0.482
Epoch: 12        Training Loss: 0.815499         Validation Loss: 0.466697
Previous Validation Loss: 0.482
Current Validation Loss: 0.467
Epoch: 13        Training Loss: 0.813820         Validation Loss: 0.478459
Epoch: 14        Training Loss: 0.778120         Validation Loss: 0.474840
Epoch: 15        Training Loss: 0.760961         Validation Loss: 0.483650
Epoch: 16        Training Loss: 0.773595         Validation Loss: 0.486199
Epoch: 17        Training Loss: 0.747635         Validation Loss: 0.468335
Epoch: 18        Training Loss: 0.754907         Validation Loss: 0.499111
Epoch: 19        Training Loss: 0.739344         Validation Loss: 0.473178
Epoch: 20        Training Loss: 0.725280         Validation Loss: 0.429346
Previous Validation Loss: 0.467
Current Validation Loss: 0.429
```

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 [26]: 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 [27]:
```
# call test function
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.581146


Test Accuracy: 83% (699/836)
```

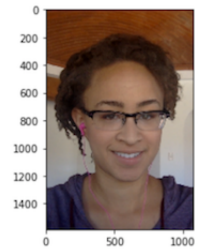
In [ ]:

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 [28]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.



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

    transform_predict = transforms.Compose([transforms.Resize(256),
                                            transforms.CenterCrop(224),
```

```
hello, human!
```




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

Sample Human Output

```
                                            transforms.ToTensor(),
                                            transforms.Normalize((0.5, 0.5, 0.5), (0.5,

        ## Convert image to Tensor

        img_tensor=transform_predict(bgr_img_input)

        index_pred=model_transfer(img_tensor.unsqueeze(0).cuda()).cpu()

        _, image_pred_tensor = torch.max(index_pred, 1)
        predicted_image = np.squeeze(image_pred_tensor.numpy())

        return class_names[predicted_image]
```

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 [29]: def run_app(img_path):
             ## handle cases for a human face, dog, and neither

             img=Image.open(img_path)
             ## Show Input Image
             plt.imshow(img)
             plt.show()
```

```
    if face_detector(img_path):
        print ("Human detected in the Image")
        print ("Matching dog breed for this Image is: ")
        print (predict_breed_transfer(img_path))

    elif dog_detector(img_path):
        print ("Dog detected in the Image")
        print ("Predicted dog breed for this Image is: ")
        print (predict_breed_transfer(img_path))

    else:
        print ("Could not predict if its a dog or human in the Image")
```

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?

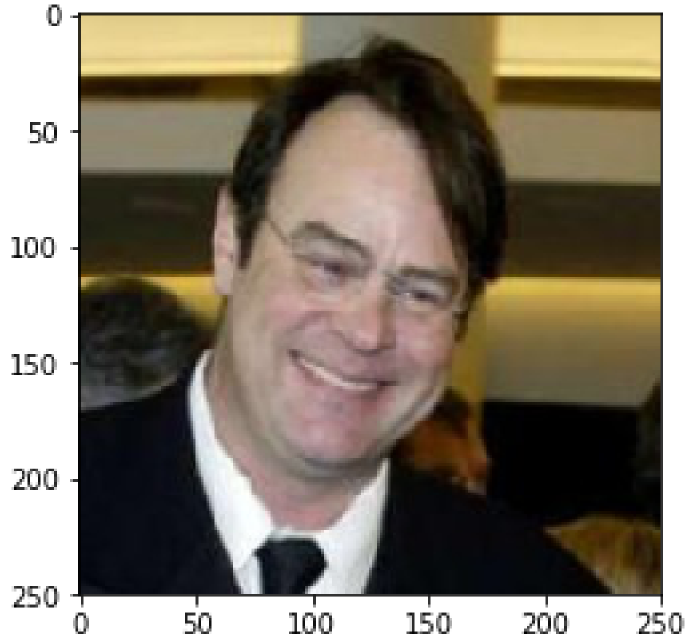
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

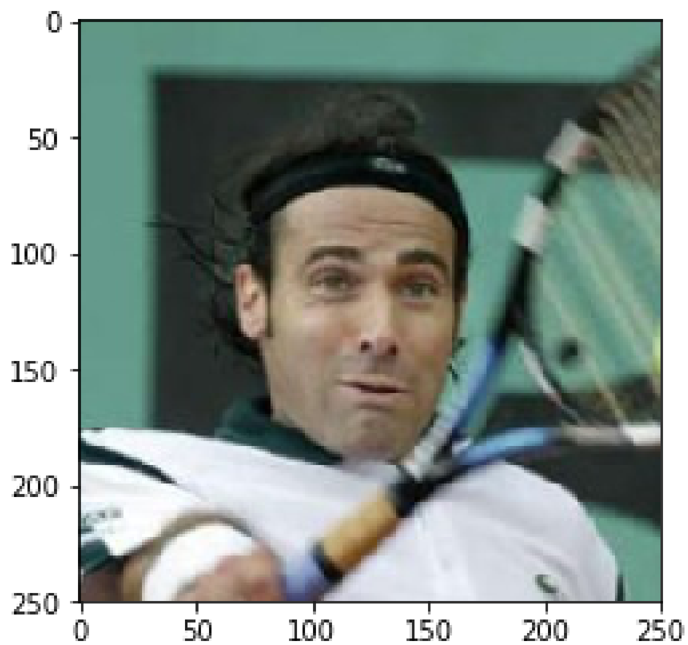
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: Model seems to be working perfectly

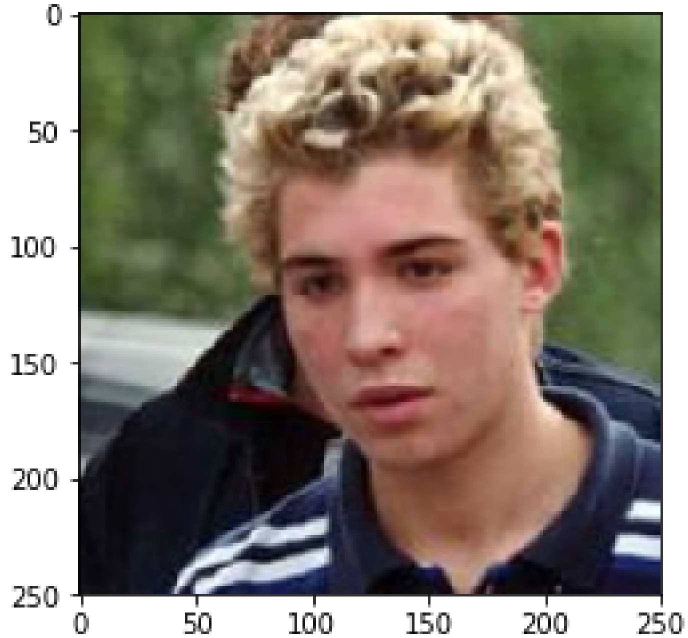
```
In [30]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[:10], dog_files[95:110])):
             run_app(file)
```



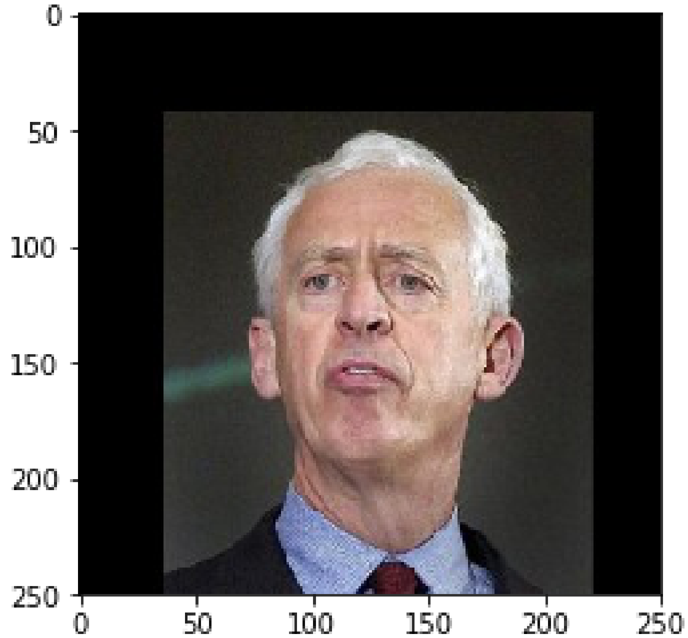

```
Human detected in the Image
Matching dog breed for this Image is:
016.Beagle
```



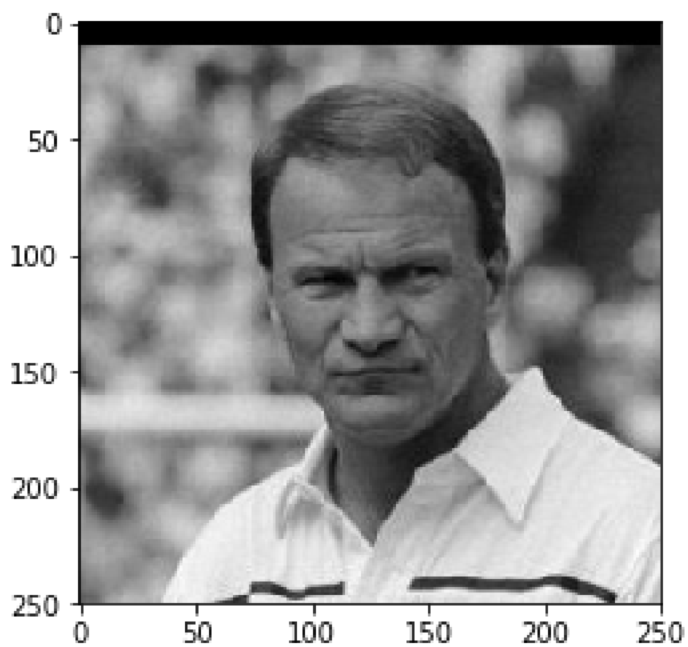
```
Human detected in the Image
Matching dog breed for this Image is:
056.Dachshund
```



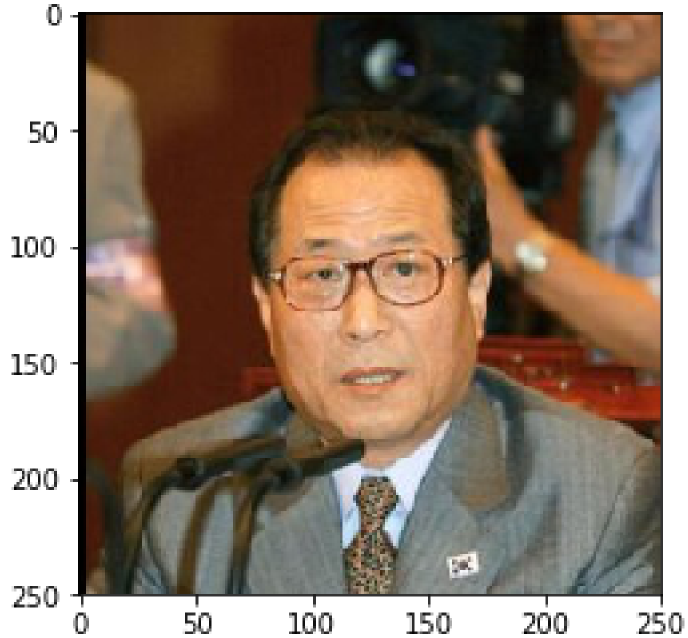
```
Human detected in the Image
Matching dog breed for this Image is:
130.Welsh_springer_spaniel
```



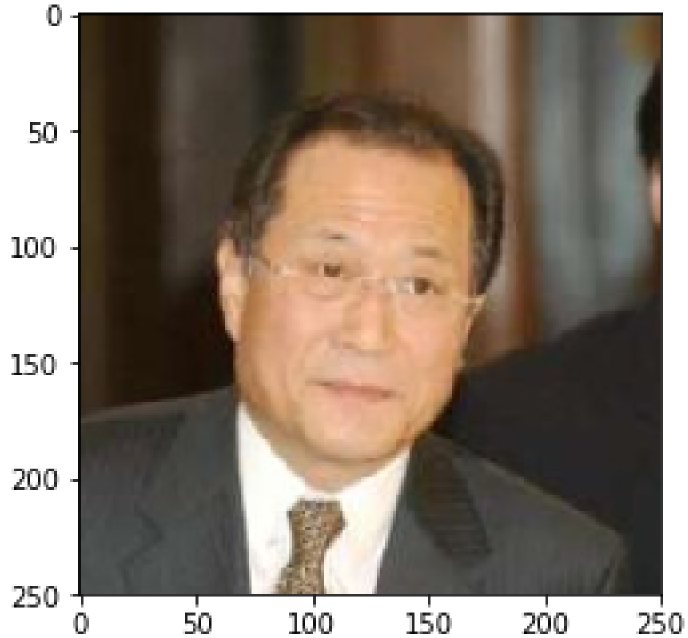

```
Human detected in the Image
Matching dog breed for this Image is:
130.Welsh_springer_spaniel
```



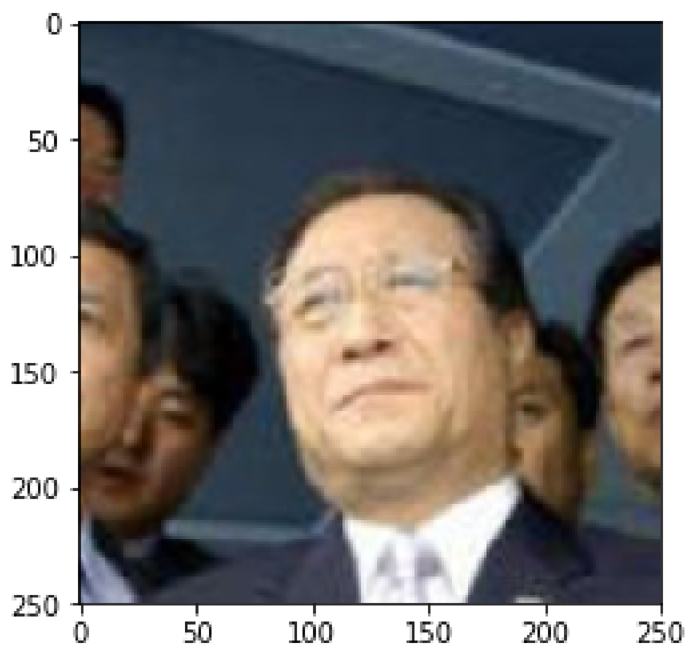
```
Human detected in the Image
Matching dog breed for this Image is:
130.Welsh_springer_spaniel
```



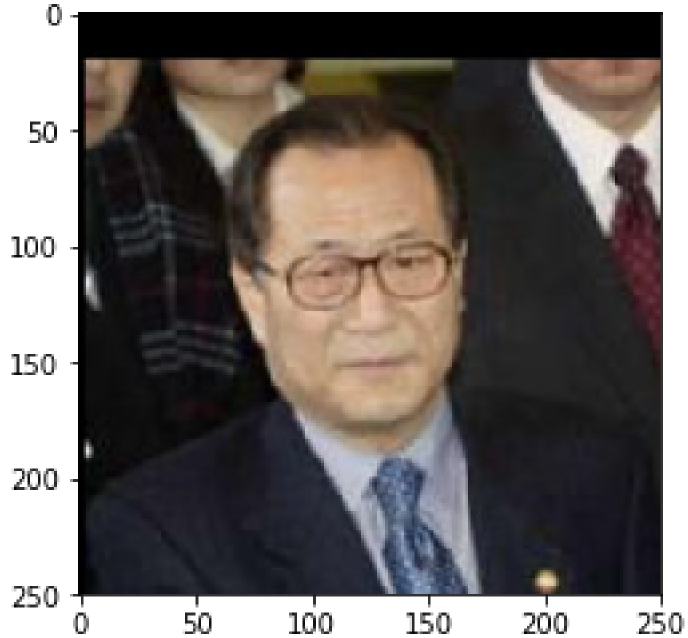

```
Human detected in the Image
Matching dog breed for this Image is:
130.Welsh_springer_spaniel
```



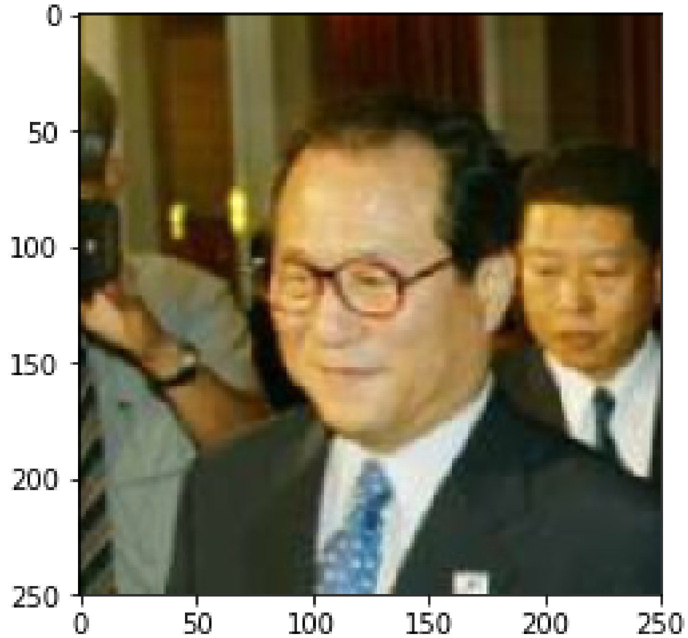
```
Human detected in the Image
Matching dog breed for this Image is:
063.English_springer_spaniel
```



```
Human detected in the Image
Matching dog breed for this Image is:
007.American_foxhound
```

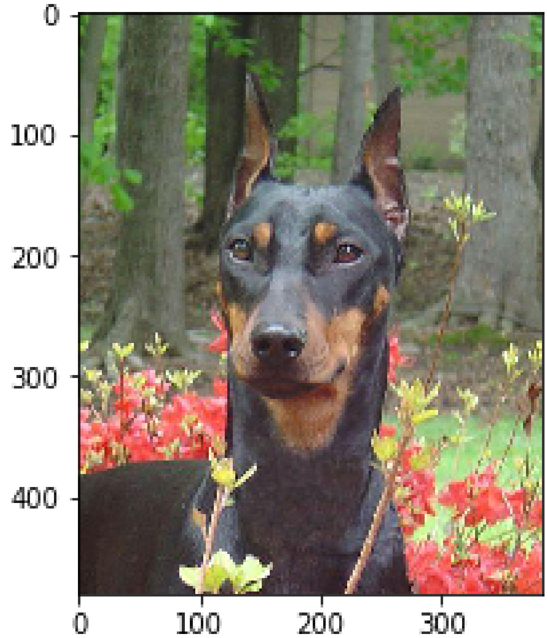


```
Human detected in the Image
Matching dog breed for this Image is:
014.Basenji
```

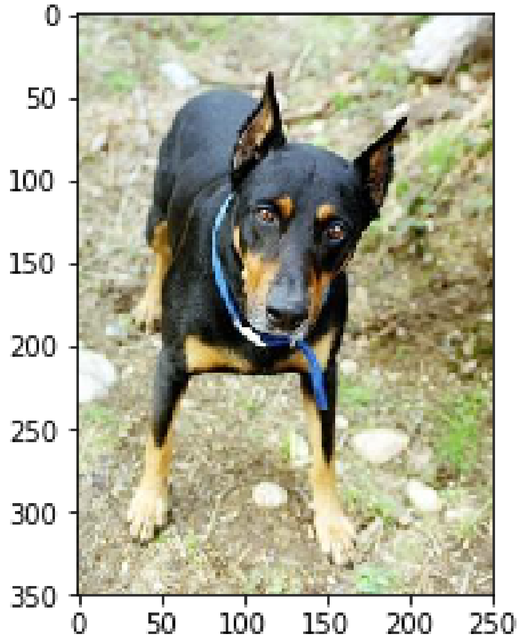



```
Human detected in the Image
Matching dog breed for this Image is:
005.Alaskan_malamute
```

```
Human detected in the Image
Matching dog breed for this Image is:
059.Doberman_pinscher
```

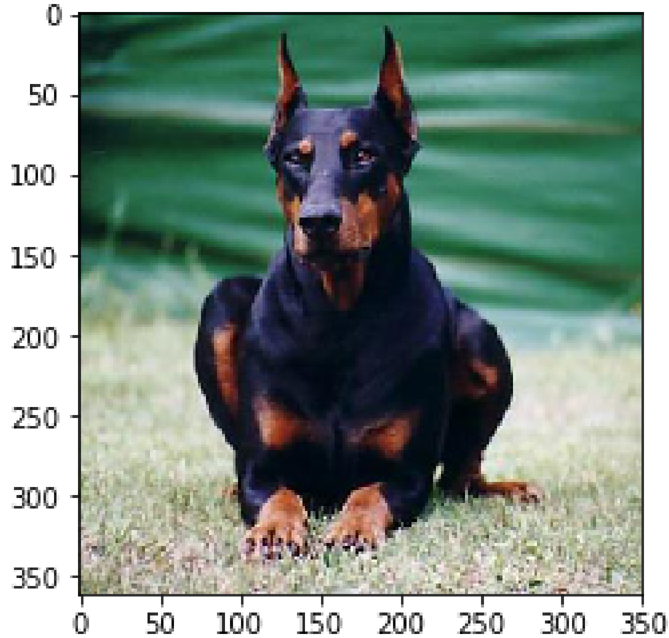



```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```

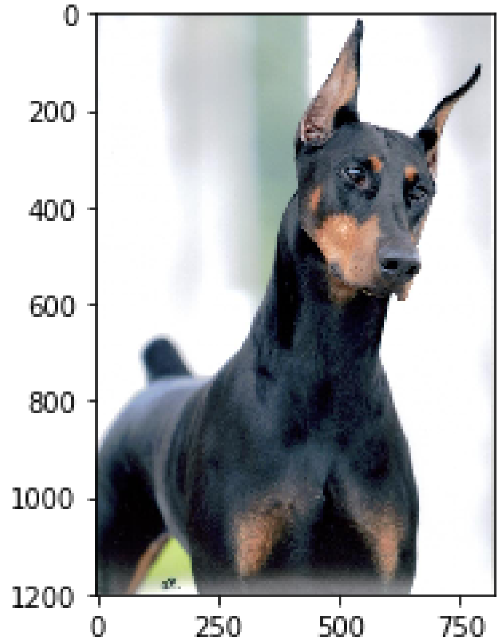



```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```

```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```

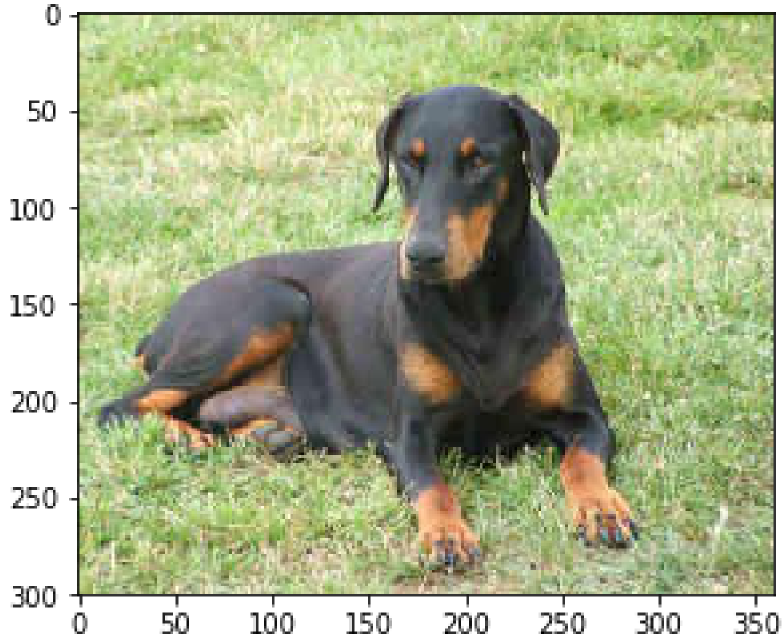



```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```



```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```

```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```




```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```

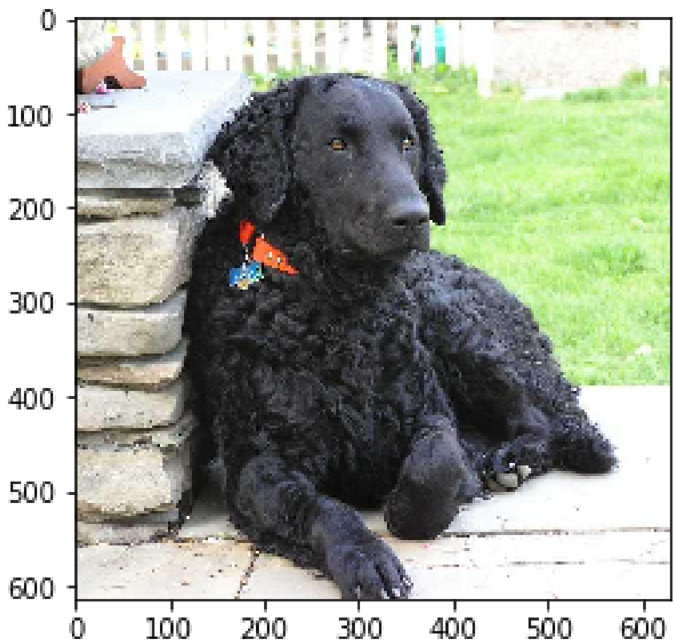

```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```

```
Dog detected in the Image
Predicted dog breed for this Image is:
059.Doberman_pinscher
```

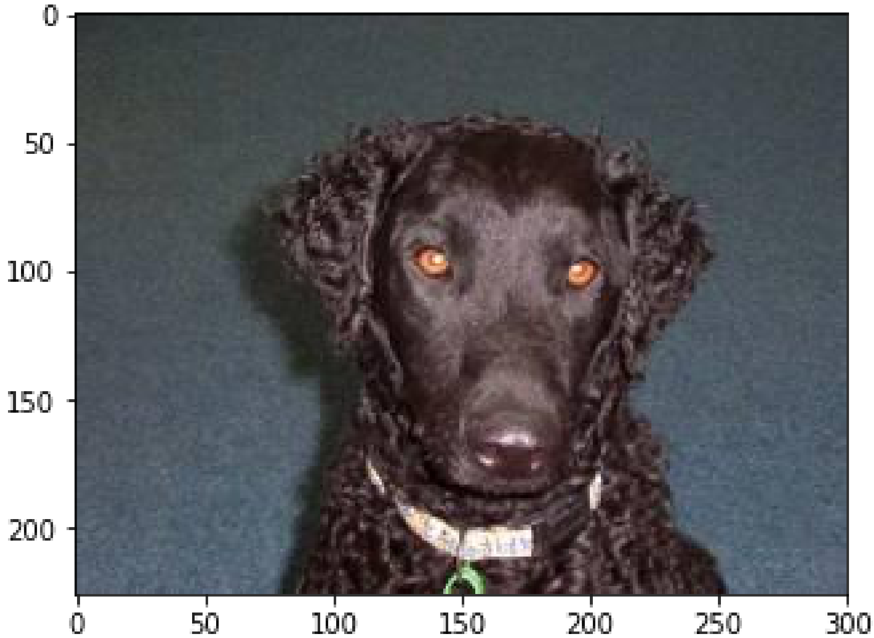

```
Dog detected in the Image
Predicted dog breed for this Image is:
055.Curly-coated_retriever
```

```
Dog detected in the Image
Predicted dog breed for this Image is:
055.Curly-coated_retriever
```

```
Dog detected in the Image
Predicted dog breed for this Image is:
055.Curly-coated_retriever
```




```
Dog detected in the Image
Predicted dog breed for this Image is:
055.Curly-coated_retriever
```




```
Dog detected in the Image
Predicted dog breed for this Image is:
055.Curly-coated_retriever
```

In [ ]: