dog_app

December 8, 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

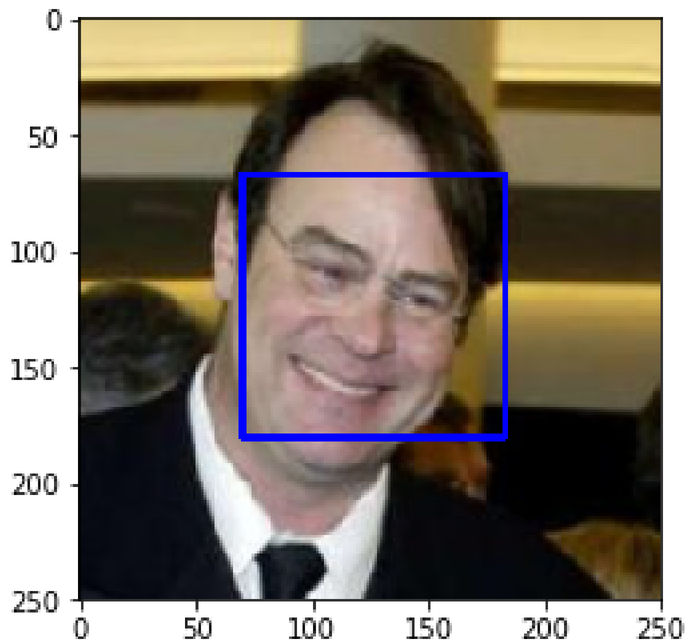
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [23]: # 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 [20]: from tqdm import tqdm

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

         #-#-# Do NOT modify the code above this line. #-#-#
         human_with_human_face = 0
         dogs_with_human_face = 0

         for h in human_files_short:
             if face_detector(h):
                 human_with_human_face+=1

         for d in dog_files_short:
             if face_detector(d):
                 dogs_with_human_face+=1
         ## TODO: Test the performance of the face_detector algorithm
         ## on the images in human_files_short and dog_files_short.
         h = (human_with_human_face*100)/len(human_files_short)
         d = (dogs_with_human_face*100)/len(dog_files_short)
         print("Percentage of human face detected in human files: ",h,'%')
         print("Percentage of human face detected in dog files: ",d,'%')
```

```
Percentage of human face detected in human files:  98.0 %
Percentage of human face detected in dog files:  17.0 %
```

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 [ ]: ### (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 [21]: 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 [00:06<00:00, 79248237.14it/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 [19]:
```python
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
    '''
    img = Image.open(img_path)
    ## 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
    resize_img = 256
    transform = transforms.Compose([transforms.Resize(resize_img),
                               transforms.CenterCrop(size=224),
                               transforms.ToTensor(),
                               transforms.Normalize(mean=[0.485, 0.456, 0.406],
                              std=[0.229, 0.224, 0.225])])

    img_t = transform(img)
#     print(img_t.size())
    batch_t = img_t.unsqueeze(0)
#     print(batch_t.size())

    if torch.cuda.is_available():
#         print("its here")
        img_t = img_t.cuda()
        batch_t = batch_t.cuda()

    #evaluate
    VGG16.eval()
    out = VGG16(batch_t)
    out = out.data.argmax().item()
#     print(out.shape)

    return out # predicted class index
```

In [23]:
```python
VGG16_predict(dog_files_short[0])
```

Out[23]: 243

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

In [17]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    if VGG16_predict(img_path) in range(151,269):
        return True
    else:
        return False # 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:

In [25]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_with_human_face = 0
dogs_with_human_face = 0

for h in human_files_short:
    if dog_detector(h):
        human_with_human_face+=1

for d in dog_files_short:
    if dog_detector(d):
        dogs_with_human_face+=1
## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
h = (human_with_human_face*100)/len(human_files_short)
d = (dogs_with_human_face*100)/len(dog_files_short)
print("Percentage of human face detected in human files: ",h,'%')
print("Percentage of human face detected in dog files: ",d,'%')
```

```
Percentage of human face detected in human files:  0.0 %
Percentage of human face 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 [26]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
         alexnet = models.alexnet(pretrained=True)
         use_cuda = torch.cuda.is_available()
         if use_cuda:
             alexnet = alexnet.cuda()
```

```
Downloading: "https://download.pytorch.org/models/alexnet-owt-4df8aa71.pth" to /root/.torch/mode
100%|| 244418560/244418560 [00:11<00:00, 20408267.86it/s]
```

```
In [27]: def alexnet_predict(img_path):
             '''
             Use pre-trained VGG-16 model to obtain index corresponding to
             predicted ImageNet class for image at specified path

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''
             img = Image.open(img_path)
             ## 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
             resize_img = 256
             transform = transforms.Compose([transforms.Resize(resize_img),
                                             transforms.CenterCrop(size=224),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                           std=[0.229, 0.224, 0.225])])


             img_t = transform(img)
         #     print(img_t.size())
             batch_t = img_t.unsqueeze(0)
         #     print(batch_t.size())

             if torch.cuda.is_available():
         #         print("its here")
                 img_t = img_t.cuda()
                 batch_t = batch_t.cuda()

             #evaluate
```

```
        alexnet.eval()
        out = VGG16(batch_t)
        out = out.data.argmax().item()
#       print(out.shape)

        return out # predicted class index
```

```
In [28]: def dog_detector_with_alexnet(img_path):
             ## TODO: Complete the function.
             if alexnet_predict(img_path) in range(151,269):
                 return True
             else:
                 return False # true/false
```

```
In [29]: human_with_human_face = 0
         dogs_with_human_face = 0

         for h in human_files_short:
             if dog_detector_with_alexnet(h):
                 human_with_human_face+=1

         for d in dog_files_short:
             if dog_detector_with_alexnet(d):
                 dogs_with_human_face+=1
         ## TODO: Test the performance of the face_detector algorithm
         ## on the images in human_files_short and dog_files_short.
         h = (human_with_human_face*100)/len(human_files_short)
         d = (dogs_with_human_face*100)/len(dog_files_short)
         print("Percentage of human face detected in human files: ",h,'%')
         print("Percentage of human face detected in dog files: ",d,'%')
```

```
Percentage of human face detected in human files:  0.0 %
Percentage of human face detected in dog files:  100.0 %
```

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 [2]: import os
        import torch
        from torchvision import datasets,transforms

        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        data_dir = os.path.join('/data/dog_images')
        train_transform = transforms.Compose([transforms.Resize(255),
                                              transforms.CenterCrop(224),
                                              transforms.RandomHorizontalFlip(),
                                                     transforms.RandomRotation(10),
                                                     transforms.ToTensor(),
                                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                    std=[0.229, 0.224, 0.225])])
        valid_transform = transforms.Compose([transforms.Resize(255),
                                              transforms.CenterCrop(224),
```

```
                                 transforms.ToTensor(),
                                       transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                            std=[0.229, 0.224, 0.225])])
test_transform = transforms.Compose([transforms.Resize(255),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                       transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                            std=[0.229, 0.224, 0.225])])

batch_size = 32
num_workers = 0

train_data = datasets.ImageFolder(data_dir+'/train',transform=train_transform)
valid_data = datasets.ImageFolder(data_dir+'/valid',transform=valid_transform)
test_data = datasets.ImageFolder(data_dir+'/test',transform=test_transform)

trainloader = torch.utils.data.DataLoader(train_data,batch_size=batch_size,shuffle=True,
validloader = torch.utils.data.DataLoader(valid_data,batch_size=batch_size,shuffle=True,
testloader = torch.utils.data.DataLoader(test_data,batch_size=batch_size,shuffle=True,nu
loaders_scratch = {'train':trainloader,'valid':validloader,'test':testloader}
num_classes=len(valid_data.classes)
num_classes
```

Out[2]: 133

In [3]:
```
len(loaders_scratch['train'].dataset)
# for batch_idx, (data, target) in enumerate(loaders_scratch['train']):
#     print()
```

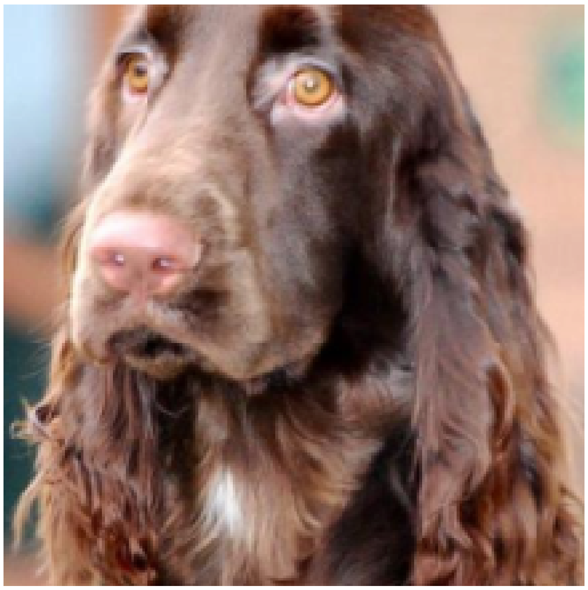
Out[3]: 6680

In [5]:
```
def imshow(image, ax=None, title=None, normalize=True):
    """Imshow for Tensor."""
    if ax is None:
        fig, ax = plt.subplots()
    image = image.numpy().transpose((1, 2, 0))

    if normalize:
        mean = np.array([0.485, 0.456, 0.406])
        std = np.array([0.229, 0.224, 0.225])
        image = std * image + mean
        image = np.clip(image, 0, 1)

    ax.imshow(image)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['left'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
    ax.tick_params(axis='both', length=0)
```

```
        ax.set_xticklabels('')
        ax.set_yticklabels('')

        return ax
```

In [7]:
```
import matplotlib.pyplot as plt
%matplotlib inline
images,labels = next(iter(testloader))
imshow(images[0],normalize=True)
# print(labels)
```

Out[7]: `<matplotlib.axes._subplots.AxesSubplot at 0x7f66a457e4a8>`



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?

The dataset is augmented by horizontal filp, as it is essential to get better accuracy. Due to flips, the model learns rotated images as well.

Answer: All the images have resized 255 pixels and centercrop to 224. The reference taken from vgg16

The dataset is augmented by horizontal filp, as it is essential to get better accuracy. Due to flips, the model learns rotated images as well.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [2]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
#     def __init__(self,constant_weight = None):
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        '''The formula to compute the output size of convolution layer is [(WK+2P)/S]+1

    W is the input size (width , height) in our case starting value is 224
    K is the Kernel size - you are using 3 as filter size
    P is the padding
    S is the stride '''
        self.conv1 = nn.Conv2d(3,32,3,stride=1,padding=1) #(224-3+(2*1))/1 = 223
        self.conv2 = nn.Conv2d(32,64,3,stride=1,padding=1) #(223 - 3+(2*1))/1 = 222
        self.conv3 = nn.Conv2d(64,128,3,stride=1,padding=1) #(222 - 3 + (2*1))/1 = 221
        self.conv4 = nn.Conv2d(128,256,3,stride=1,padding=1) # (221 - 3 + (2*1))/1  = 22
        self.conv5 = nn.Conv2d(256,512,3,stride=1,padding=1) # (220-3+(2*1))/1 = 219

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

        #fully connected
        self.fc1 = nn.Linear(512*7*7,512)
        self.fc2 = nn.Linear(512,num_classes)

        #dropout
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        ## Define forward behavior
        shape = 224

        x = F.relu(self.conv1(x))
        x = self.pool(x) # shape 112
        x = F.relu(self.conv2(x))
        x = self.pool(x) # shape 56
        x = F.relu(self.conv3(x))
        x = self.pool(x) # shape 28
        x = F.relu(self.conv4(x))
        x = self.pool(x) # shape 14
        x = F.relu(self.conv5(x))
```

```python
        x = self.pool(x)  # shape 7

#       print(x.size())

        # flatten the image
        x = x.view(-1,512*7*7)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x


# instantiate the CNN
model_scratch = Net()
# model_scratch.apply(weights_init_normal)
print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(256, 512, 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=25088, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

Answer: The input image size is 3 channel that is RGB. The total 5 convolutional layers have been used. The The formula to compute the output size of convolution layer is [(WK+2P)/S]+1

```
W is the input size (width , height) in our case starting value is 224
K is the Kernel size - you are using 3 as filter size
P is the padding
S is the stride
The sizes of the convolutional layer is given below as per the formula
self.conv1 = nn.Conv2d(3,32,3,stride=1,padding=1) #(224-3+(2*1))/1 = 223
    self.conv2 = nn.Conv2d(32,64,3,stride=1,padding=1) #(223 - 3+(2*1))/1 = 222
```

```
self.conv3 = nn.Conv2d(64,128,3,stride=1,padding=1) #(222 - 3 + (2*1))/1 = 221
self.conv4 = nn.Conv2d(128,256,3,stride=1,padding=1) # (221 - 3 + (2*1))/1  = 220
self.conv5 = nn.Conv2d(256,512,3,stride=1,padding=1) # (220-3+(2*1))/1 = 219

We have used maxpool layer of stride 2 and kernel size 2, which reduces the x-y dimension by

x = F.relu(self.conv1(x))
x = self.pool(x) # shape 112
x = F.relu(self.conv2(x))
x = self.pool(x) # shape 56
x = F.relu(self.conv3(x))
x = self.pool(x) # shape 28
x = F.relu(self.conv4(x))
x = self.pool(x) # shape 14
x = F.relu(self.conv5(x))
x = self.pool(x)  # shape 7
```

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [3]: import torch.optim as optim

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

        ### TODO: select optimizer
        # optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001)
        optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [4]: from PIL import ImageFile
        import numpy as np
        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_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)

#calculate average losses
train_loss = train_loss/len(loaders['train'].dataset)
valid_loss = valid_loss/len(loaders['valid'].dataset)

# 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 ...."
    torch.save(model.state_dict(),save_path)
    valid_loss_min = valid_loss
```

```
        # return trained model
        return model
```

```
In [5]: # train the model
        model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch,
                              criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.843733         Validation Loss: 4.712962
Validation loss decreased (inf --> 4.712962).     Saving model ...
Epoch: 2         Training Loss: 4.589357         Validation Loss: 4.495395
Validation loss decreased (4.712962 --> 4.495395).     Saving model ...
Epoch: 3         Training Loss: 4.319837         Validation Loss: 4.227104
Validation loss decreased (4.495395 --> 4.227104).     Saving model ...
Epoch: 4         Training Loss: 4.075535         Validation Loss: 4.054444
Validation loss decreased (4.227104 --> 4.054444).     Saving model ...
Epoch: 5         Training Loss: 3.908346         Validation Loss: 4.029185
Validation loss decreased (4.054444 --> 4.029185).     Saving model ...
Epoch: 6         Training Loss: 3.718034         Validation Loss: 3.896463
Validation loss decreased (4.029185 --> 3.896463).     Saving model ...
Epoch: 7         Training Loss: 3.552626         Validation Loss: 3.898020
Epoch: 8         Training Loss: 3.339921         Validation Loss: 3.785672
Validation loss decreased (3.896463 --> 3.785672).     Saving model ...
Epoch: 9         Training Loss: 3.165722         Validation Loss: 3.747928
Validation loss decreased (3.785672 --> 3.747928).     Saving model ...
Epoch: 10         Training Loss: 2.967329         Validation Loss: 3.778125
```

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 [6]: 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 [7]: # call test function
        test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.644566


Test Accuracy: 12% (103/836)
```

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 [8]: ## TODO: Specify data loaders
        transfer_loaders=loaders_scratch.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [27]: import torchvision.models as models
         import torch.nn as nn
```

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

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

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


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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 94906891.69it/s]
```

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: The resnet50 model is already trained on wide dataset with various different objects. The subsequent layers of this model has already learnt patterns, so these layers freezed. The learnings of these layers can be useful for us to classify our own dataset. Only the last layer is changed that is fully connected layer to predict 133 classes.

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 [10]: import torch.optim as optim
         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [11]: # train the model
         model_transfer = train(10, transfer_loaders, model_transfer, optimizer_transfer, criter

         # 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: 2.147254         Validation Loss: 0.839101
Validation loss decreased (inf --> 0.839101).     Saving model ...
Epoch: 2         Training Loss: 0.717141         Validation Loss: 0.628409
Validation loss decreased (0.839101 --> 0.628409).     Saving model ...
Epoch: 3         Training Loss: 0.550836         Validation Loss: 0.587890
Validation loss decreased (0.628409 --> 0.587890).     Saving model ...
```

```
Epoch: 4         Training Loss: 0.447321         Validation Loss: 0.529380
Validation loss decreased (0.587890 --> 0.529380).     Saving model ...
Epoch: 5         Training Loss: 0.401545         Validation Loss: 0.537478
Epoch: 6         Training Loss: 0.337873         Validation Loss: 0.471468
Validation loss decreased (0.529380 --> 0.471468).     Saving model ...
Epoch: 7         Training Loss: 0.295551         Validation Loss: 0.559612
Epoch: 8         Training Loss: 0.268737         Validation Loss: 0.482832
Epoch: 9         Training Loss: 0.259422         Validation Loss: 0.519624
Epoch: 10         Training Loss: 0.250651         Validation Loss: 0.506176
```

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 [12]: test(transfer_loaders, model_transfer, criterion_transfer, use_cuda)
```

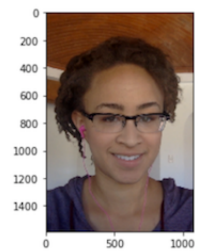
```
Test Loss: 0.507343


Test Accuracy: 84% (708/836)
```

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 [12]: ### 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]
         all_class_dir = os.listdir(data_dir + '/train/')
         # class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].classes]
         class_names = [item[4:].replace("_", " ") for item in all_class_dir]
         # print(class_names)

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img_transforms = transforms.Compose([transforms.Resize(size=(224, 224)),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize(mean=[0.485, 0.456,
                                                                       std=[0.229, 0.224,

             img = Image.open(img_path)
             img_tensor = img_transforms(img)[None,:]
             if use_cuda:
                 img_tensor = img_tensor.cuda()
             model_transfer.eval()
```

```
hello, human!
```



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

Sample Human Output

```
    output = model_transfer(img_tensor)
    _,preds_tensor = torch.max(output,1)
    return class_names[preds_tensor.cpu().numpy()[0]]
```

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 [15]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
from PIL import Image
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path) is True:
        print(predict_breed_transfer(img_path))
        print('Dog detected')
    elif face_detector(img_path):
        print(predict_breed_transfer(img_path))
        print('Human detected')
```

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!

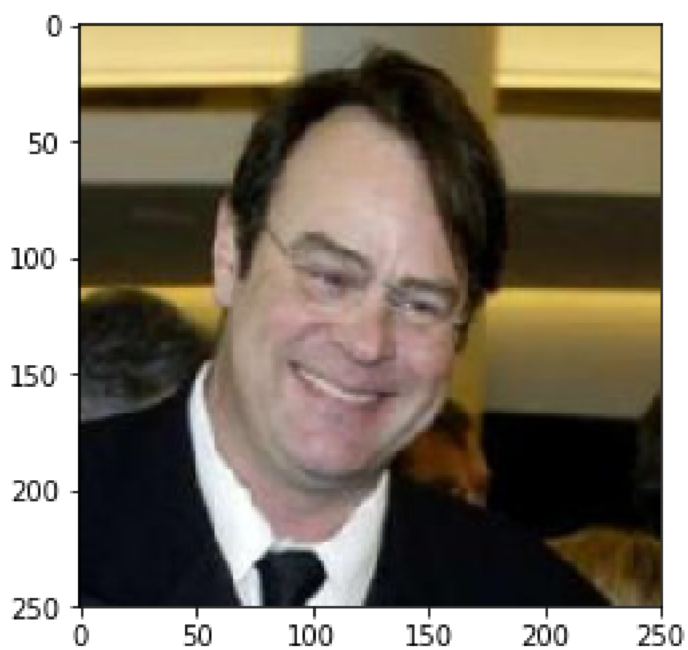
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

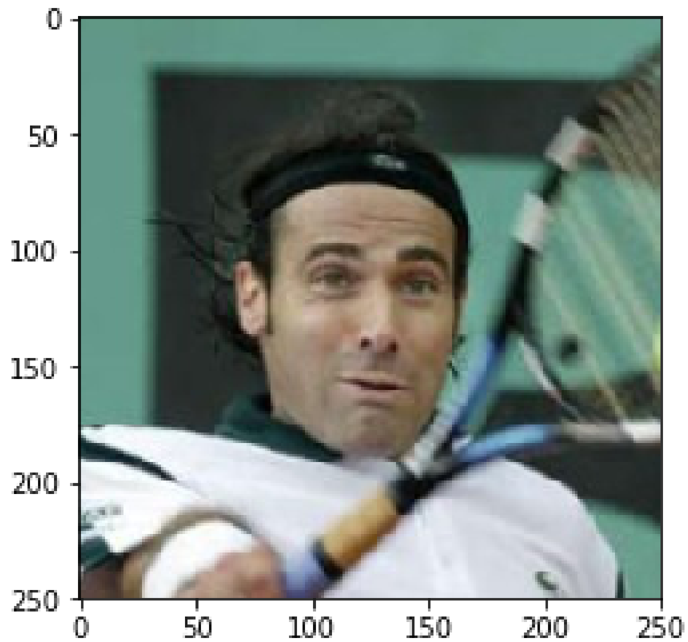
Answer: (Three possible points for improvement) 1. The app is working fine for the test dataset. Scope for the improvemnt 1. It does not work well with gray scale images. 2. It does not classify the object which are neither dogs not humans 3. It does not classify the multiple objects in the images such dog and human together in the same image.

How it can be improved. 1. Make it multiclass classification problem to identify the dogs and humans together. 2. Improve and increase train and test datasets.

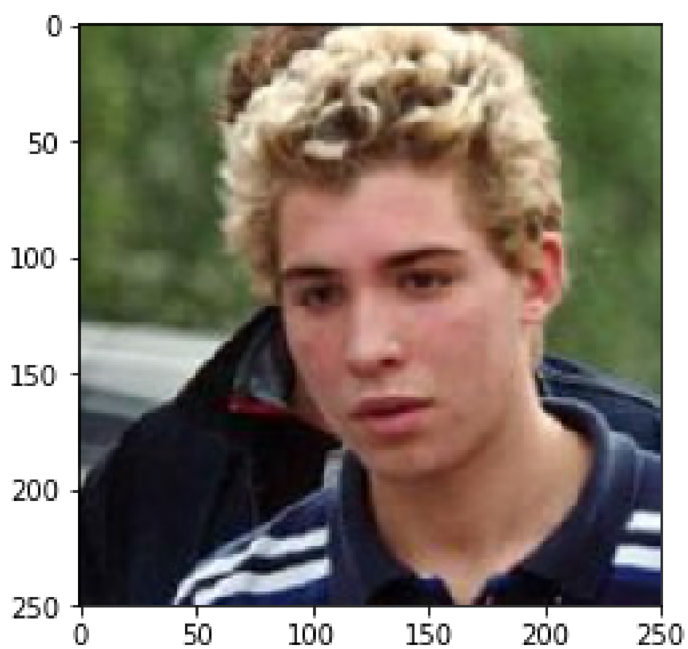
```
In [32]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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



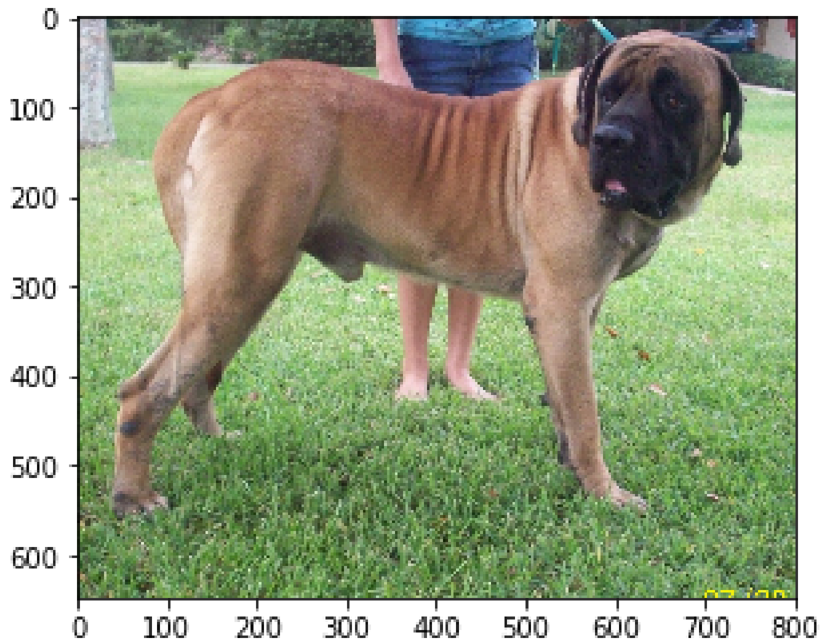
```
Norwegian elkhound
Human detected
```



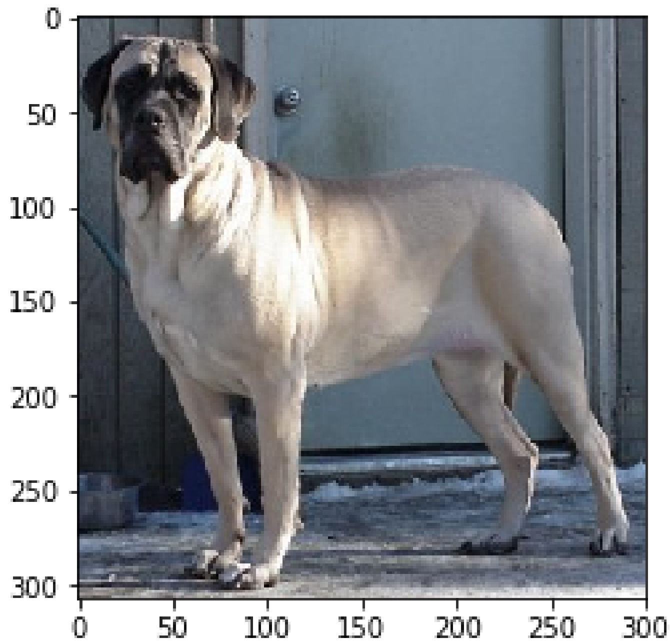

```
Norwegian elkhound
Human detected
```



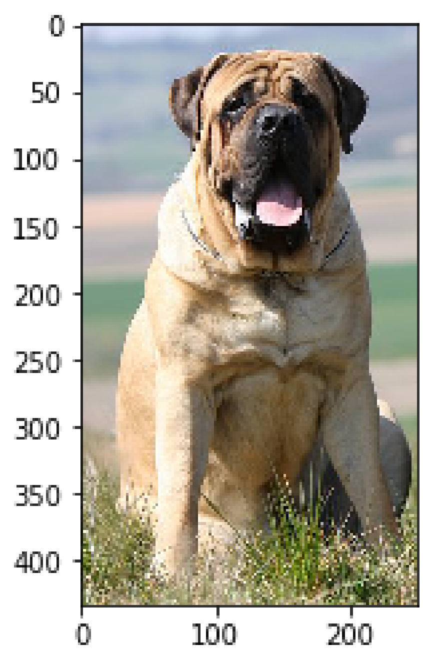
Bullmastiff
Human detected



Plott
Dog detected



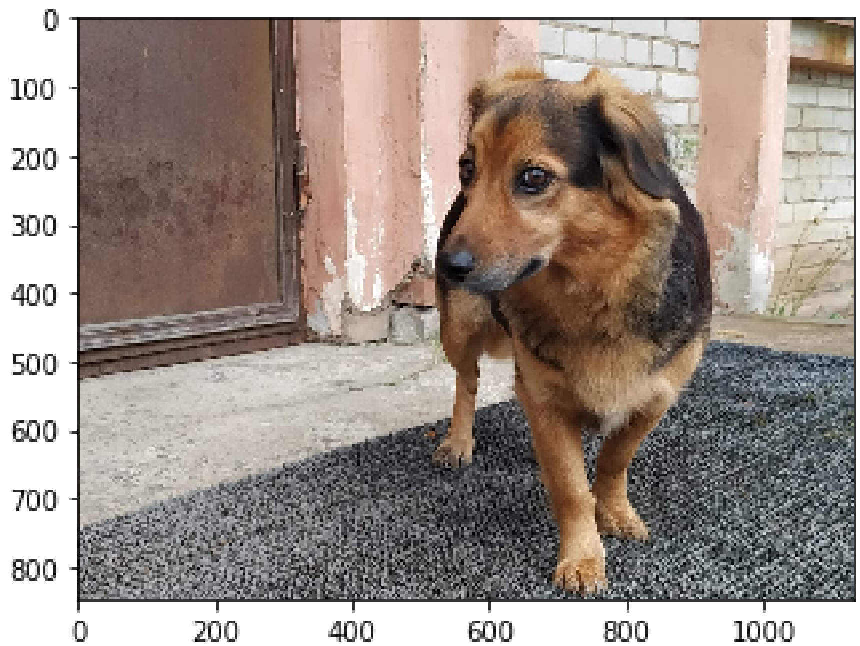
```
Plott
Dog detected
```



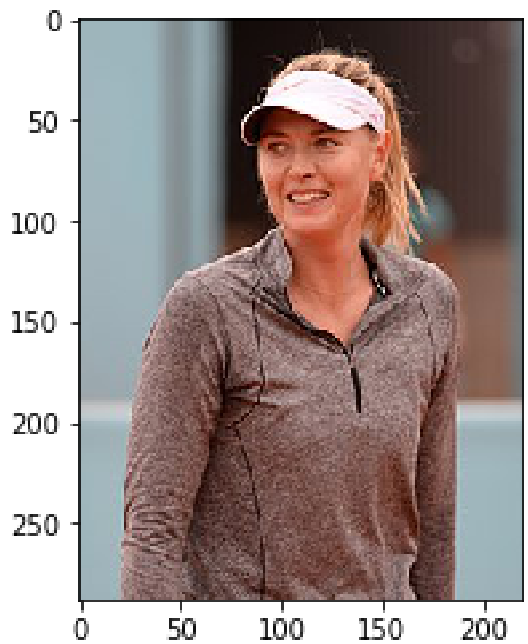
```
Bluetick coonhound
Dog detected
```

```
In [33]: print("Let's see some of the images")
         test_images = np.array(glob("./test_images/*"))
         for test_image in test_images:
             run_app(test_image)
```

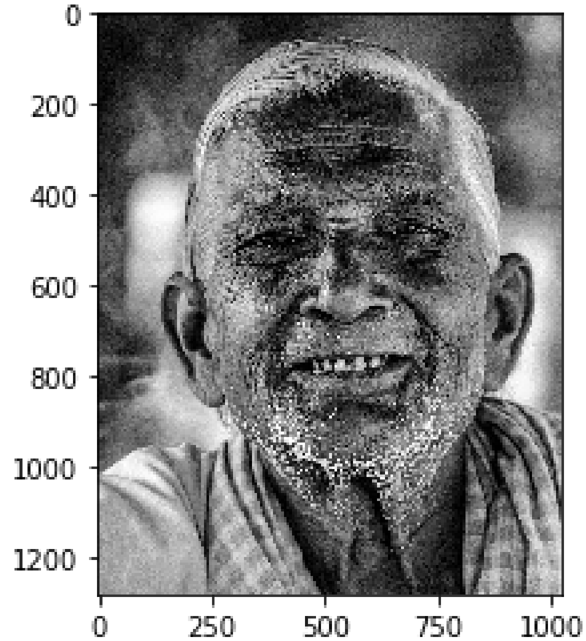
```
Let's see some of the images
```

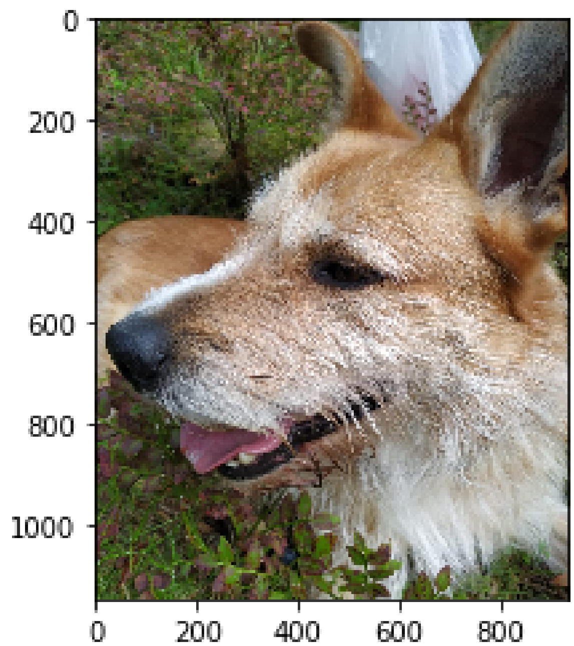



```
Saint bernard
Dog detected
```

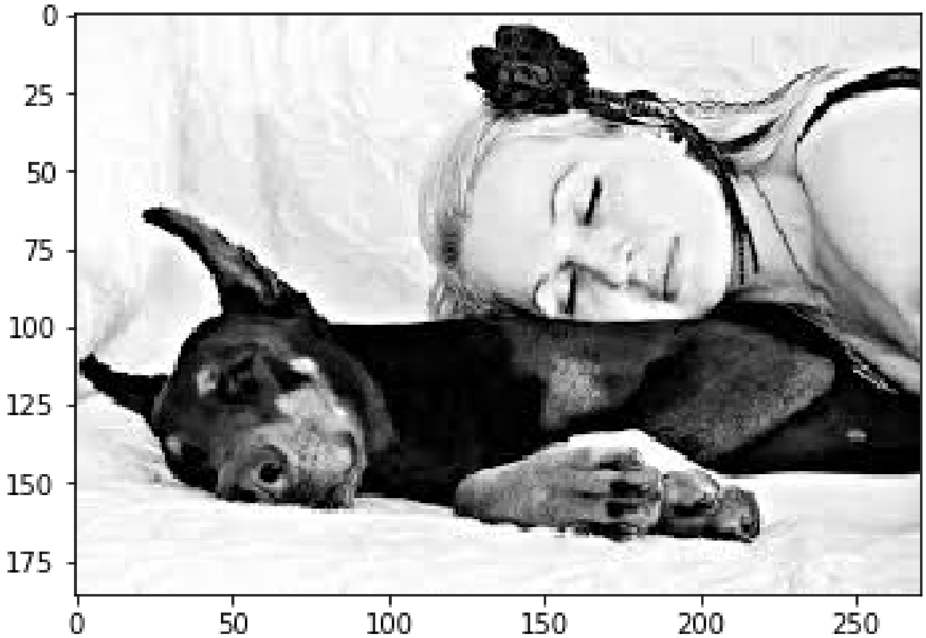


```
Bullmastiff
Human detected
```






```
Bullmastiff
Dog detected
```




```
Norwegian elkhound
Dog detected
```

In [ ]:

In [ ]: