dog_app

November 14, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

[1]:
```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

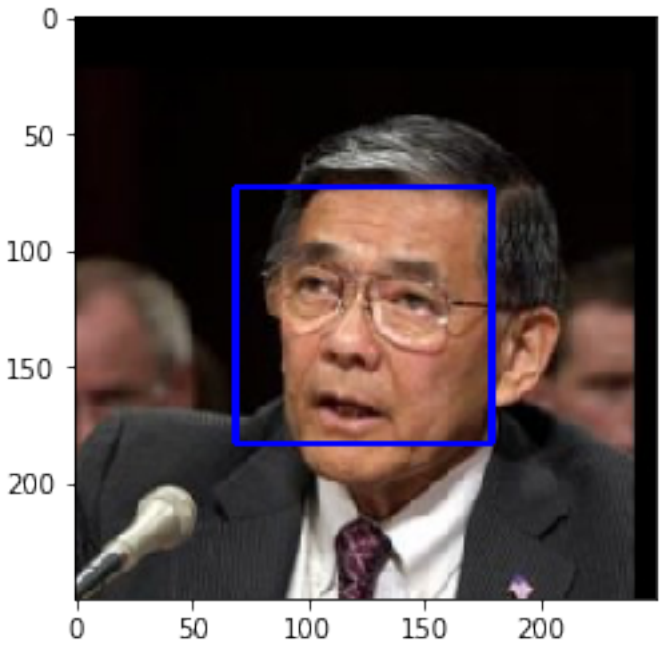
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

[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.

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

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)

[4]:
```
from tqdm import tqdm

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

#-#-# Do NOT modify the code above this line. #-#-#

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
print(human_files_short[1].dtype)

#test_face_detector_performance(dog_files_short)
def test_face_detector_performance(images):
    accurate_face_count = 0
    for i in images:
        if (face_detector(i) == True):
            accurate_face_count += 1
    percentage = (accurate_face_count/len(images)) * 100
    return percentage

print('Percentage of Face detection is: {}% in human_files'.
 ↪format(test_face_detector_performance(human_files_short)))
print('Percentage of Face detection is: {}% in dog_files'.
 ↪format(test_face_detector_performance(dog_files[:100])))
```

```
<U40
Percentage of Face detection is: 99.0% in human_files
Percentage of Face detection is: 8.0% in dog_files
```

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`.

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

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

[5]:
```
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()
```

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.

[6]:
```
from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    image = Image.open(img_path).convert('RGB')
    # resize image
    transformed_image = transforms.Compose([transforms.Resize(size=(244, 244)),
                                            transforms.ToTensor()])

    # discard alpha channel to add batch dimension
    image = transformed_image(image)[:3,:,:].unsqueeze(0)
    ## Return the *index* of the predicted class for that image

    if use_cuda:
        image = image.cuda()
    resu = VGG16(image)
    return torch.max(resu,1)[1].item() # predicted class index

print(VGG16_predict(human_files_short[5]))
```

```
903
```

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).

[10]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    image = VGG16_predict(img_path)
    image_res = image >= 151 and image <= 268
    return image_res  # true/false

print(dog_detector(dog_files_short[5]))
print(dog_detector(human_files_short[5]))
```

```
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:

[9]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
def test_dog_detector_performance(images):
    accurate_dog_count = 0
    for i in images:
        if (dog_detector(i) == True):
            accurate_dog_count += 1
    percentage = (accurate_dog_count/len(images)) * 100
    return percentage

print('Percentage of Dog detection is: {}% in human_files'.
 ↪format(test_dog_detector_performance(human_files_short)))
print('Percentage of Dog detection is: {}% in dog_files'.
 ↪format(test_dog_detector_performance(dog_files_short)))
```

```
Percentage of Dog detection is: 0.0% in human_files
Percentage of Dog detection is: 91.0% in dog_files
```

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`.

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

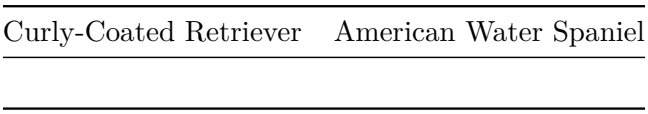
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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

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

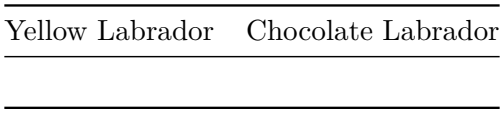
Brittany Welsh Springer Spaniel

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



Curly-Coated Retriever American Water Spaniel

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



Yellow Labrador Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

[11]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

image_path = 'dogImages/dogImages'
training_data = os.path.join(image_path, 'train/')
testing_data = os.path.join(image_path, 'test/')
validation_data = os.path.join(image_path, 'valid/')

#image resing
image_transform = {'train': transforms.Compose([transforms.
 ↪RandomResizedCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
                                                           std=[0.229, 0.224, 0.225])]),
                   'test': transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
                                                           std=[0.229, 0.224, 0.225])]),
                   'valid': transforms.Compose([transforms.
 ↪Resize(size=(224,224)),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
                                                           std=[0.229, 0.224, 0.225])])}


dog_train_data = datasets.ImageFolder(training_data,␣
 ↪transform=image_transform['train'])
train_data_loader = torch.utils.data.DataLoader(dog_train_data,
                                                batch_size=20,
                                                shuffle=True,
                                                num_workers=0)
```

```
dog_test_data = datasets.ImageFolder(testing_data,␣
 ↪transform=image_transform['test'])
test_data_loader = torch.utils.data.DataLoader(dog_test_data,
                                               batch_size=20,
                                               shuffle=True,
                                               num_workers=0)

dog_valid_data = datasets.ImageFolder(validation_data,␣
 ↪transform=image_transform['valid'])
valid_data_loader = torch.utils.data.DataLoader(dog_valid_data,
                                               batch_size=20,
                                               shuffle=True,
                                               num_workers=0)
data_loaders = {
    'train': train_data_loader,
    'valid': valid_data_loader,
    'test': test_data_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: I have considered image resizing and augmentation on the train data to support randomness, prevent overfitting and improve the model performance during predictions through flips. To achieve that I've used the RandomResizedCrop and RandomHorizontalFlip functions in tranforms. For validation data, I have applied image cropping and resizing using CenterCrop and Resize respectively. Then, I have only considered image resizing on the testing data.

I didn't consider image augmentation on both testing and validation data since it will be used for just testing and validation checks respectively.

1.1.8 (IMPLEMENTATION) Model Architecture

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

[12]:
```
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
```

```
        #convolutional layers
        self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

        #connected layers
        self.conn1 = nn.Linear(7*7*128, 500)
        #total number of dog breed classes is 133
        self.conn2 = nn.Linear(500, 133)

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

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)

        x = F.relu(self.conv2(x))
        x = self.pool(x)

        x = F.relu(self.conv3(x))
        x = self.pool(x)


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

        x = self.dropout(x)
        x = F.relu(self.conn1(x))

        x = self.dropout(x)
        x = self.conn2(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)
  (conn1): Linear(in_features=6272, out_features=500, bias=True)
  (conn2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
)
```

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

Answer: In my CNN model architecture, I have: - - 3 convolutional layers of which the first 2 have a kernel size of 3 with a stride of 2 and the third convolutional layer is consist of kernel_size of 3 with stride 1. - 1 maxpooling layers with strides of 4 and 2 respectively - 2 fully connected layers - 1 dropout layer

For the first 2 convolutional layers, 1 maxpooling layer with stride 2 is placed after each convolutional layer to downsize the input image by 2. The third convolutional layer has a stride of 1 and this will not reduce input image. After the final maxpooling of stride 2, the total output of the image size is downsized by factor of 32 and the final depth will be 128. I used the dropout layer to prevent overfitting, then placed the first fully connected layer and then the second one which is intended to produce the final output size which predicts the dog breed classes.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

[13]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.05)
```

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'`.

[14]:
```
# 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
    last_validation_loss = None
    if last_validation_loss is not None:
        valid_loss_min = last_validation_loss
    else:
        valid_loss_min = np.Inf

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

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

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

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
```

```
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data -␣
↪valid_loss))


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

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model␣
↪...'.format(
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
model_scratch = train(25, data_loaders, 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, Batch 1 loss: 4.882099
Epoch 1, Batch 101 loss: 4.883565
Epoch 1, Batch 201 loss: 4.880141
Epoch 1, Batch 301 loss: 4.875488
Epoch: 1        Training Loss: 4.871869         Validation Loss: 4.833367
Validation loss decreased (inf --> 4.833367).  Saving model …
Epoch 2, Batch 1 loss: 4.681248
Epoch 2, Batch 101 loss: 4.810892
Epoch 2, Batch 201 loss: 4.795093
Epoch 2, Batch 301 loss: 4.771942
Epoch: 2        Training Loss: 4.760646         Validation Loss: 4.619383
```

```
Validation loss decreased (4.833367 --> 4.619383).  Saving model …
Epoch 3, Batch 1 loss: 4.454704
Epoch 3, Batch 101 loss: 4.633994
Epoch 3, Batch 201 loss: 4.637691
Epoch 3, Batch 301 loss: 4.629588
Epoch: 3        Training Loss: 4.627565         Validation Loss: 4.513686
Validation loss decreased (4.619383 --> 4.513686).  Saving model …
Epoch 4, Batch 1 loss: 4.403468
Epoch 4, Batch 101 loss: 4.571610
Epoch 4, Batch 201 loss: 4.572260
Epoch 4, Batch 301 loss: 4.564627
Epoch: 4        Training Loss: 4.565605         Validation Loss: 4.403347
Validation loss decreased (4.513686 --> 4.403347).  Saving model …
Epoch 5, Batch 1 loss: 4.595936
Epoch 5, Batch 101 loss: 4.512516
Epoch 5, Batch 201 loss: 4.500206
Epoch 5, Batch 301 loss: 4.496191
Epoch: 5        Training Loss: 4.494160         Validation Loss: 4.345341
Validation loss decreased (4.403347 --> 4.345341).  Saving model …
Epoch 6, Batch 1 loss: 4.650065
Epoch 6, Batch 101 loss: 4.423903
Epoch 6, Batch 201 loss: 4.435905
Epoch 6, Batch 301 loss: 4.429854
Epoch: 6        Training Loss: 4.431696         Validation Loss: 4.337811
Validation loss decreased (4.345341 --> 4.337811).  Saving model …
Epoch 7, Batch 1 loss: 4.403378
Epoch 7, Batch 101 loss: 4.361153
Epoch 7, Batch 201 loss: 4.372077
Epoch 7, Batch 301 loss: 4.373088
Epoch: 7        Training Loss: 4.371491         Validation Loss: 4.220118
Validation loss decreased (4.337811 --> 4.220118).  Saving model …
Epoch 8, Batch 1 loss: 4.328329
Epoch 8, Batch 101 loss: 4.355619
Epoch 8, Batch 201 loss: 4.358055
Epoch 8, Batch 301 loss: 4.329449
Epoch: 8        Training Loss: 4.326734         Validation Loss: 4.166078
Validation loss decreased (4.220118 --> 4.166078).  Saving model …
Epoch 9, Batch 1 loss: 4.519702
Epoch 9, Batch 101 loss: 4.245837
Epoch 9, Batch 201 loss: 4.243743
Epoch 9, Batch 301 loss: 4.253542
Epoch: 9        Training Loss: 4.252599         Validation Loss: 4.074879
Validation loss decreased (4.166078 --> 4.074879).  Saving model …
Epoch 10, Batch 1 loss: 4.437534
Epoch 10, Batch 101 loss: 4.178147
Epoch 10, Batch 201 loss: 4.172981
Epoch 10, Batch 301 loss: 4.190989
Epoch: 10       Training Loss: 4.189558         Validation Loss: 4.017558
```

```
Validation loss decreased (4.074879 --> 4.017558).  Saving model …
Epoch 11, Batch 1 loss: 4.165414
Epoch 11, Batch 101 loss: 4.112181
Epoch 11, Batch 201 loss: 4.150693
Epoch 11, Batch 301 loss: 4.149960
Epoch: 11       Training Loss: 4.152700         Validation Loss: 4.065785
Epoch 12, Batch 1 loss: 4.070299
Epoch 12, Batch 101 loss: 4.065803
Epoch 12, Batch 201 loss: 4.095875
Epoch 12, Batch 301 loss: 4.096495
Epoch: 12       Training Loss: 4.094399         Validation Loss: 3.927583
Validation loss decreased (4.017558 --> 3.927583).  Saving model …
Epoch 13, Batch 1 loss: 4.191236
Epoch 13, Batch 101 loss: 4.070046
Epoch 13, Batch 201 loss: 4.053849
Epoch 13, Batch 301 loss: 4.042639
Epoch: 13       Training Loss: 4.040476         Validation Loss: 4.083091
Epoch 14, Batch 1 loss: 4.164064
Epoch 14, Batch 101 loss: 3.994609
Epoch 14, Batch 201 loss: 4.000149
Epoch 14, Batch 301 loss: 3.999689
Epoch: 14       Training Loss: 3.999815         Validation Loss: 3.943221
Epoch 15, Batch 1 loss: 3.339029
Epoch 15, Batch 101 loss: 3.948883
Epoch 15, Batch 201 loss: 3.966433
Epoch 15, Batch 301 loss: 3.974664
Epoch: 15       Training Loss: 3.964453         Validation Loss: 3.894125
Validation loss decreased (3.927583 --> 3.894125).  Saving model …
Epoch 16, Batch 1 loss: 3.660712
Epoch 16, Batch 101 loss: 3.869851
Epoch 16, Batch 201 loss: 3.910177
Epoch 16, Batch 301 loss: 3.909070
Epoch: 16       Training Loss: 3.902665         Validation Loss: 3.933489
Epoch 17, Batch 1 loss: 4.124616
Epoch 17, Batch 101 loss: 3.866318
Epoch 17, Batch 201 loss: 3.872093
Epoch 17, Batch 301 loss: 3.868011
Epoch: 17       Training Loss: 3.870832         Validation Loss: 3.914896
Epoch 18, Batch 1 loss: 3.928917
Epoch 18, Batch 101 loss: 3.814684
Epoch 18, Batch 201 loss: 3.807485
Epoch 18, Batch 301 loss: 3.808164
Epoch: 18       Training Loss: 3.806307         Validation Loss: 3.819235
Validation loss decreased (3.894125 --> 3.819235).  Saving model …
Epoch 19, Batch 1 loss: 3.776705
Epoch 19, Batch 101 loss: 3.703864
Epoch 19, Batch 201 loss: 3.749259
Epoch 19, Batch 301 loss: 3.768237
```

```
Epoch: 19       Training Loss: 3.767354         Validation Loss: 3.876869
Epoch 20, Batch 1 loss: 4.222556
Epoch 20, Batch 101 loss: 3.707685
Epoch 20, Batch 201 loss: 3.723547
Epoch 20, Batch 301 loss: 3.737077
Epoch: 20       Training Loss: 3.743930         Validation Loss: 3.694498
Validation loss decreased (3.819235 --> 3.694498).  Saving model …
Epoch 21, Batch 1 loss: 3.825597
Epoch 21, Batch 101 loss: 3.646901
Epoch 21, Batch 201 loss: 3.653035
Epoch 21, Batch 301 loss: 3.678960
Epoch: 21       Training Loss: 3.691600         Validation Loss: 3.768610
Epoch 22, Batch 1 loss: 3.476888
Epoch 22, Batch 101 loss: 3.627496
Epoch 22, Batch 201 loss: 3.621227
Epoch 22, Batch 301 loss: 3.638665
Epoch: 22       Training Loss: 3.634136         Validation Loss: 3.726614
Epoch 23, Batch 1 loss: 3.867521
Epoch 23, Batch 101 loss: 3.559690
Epoch 23, Batch 201 loss: 3.595249
Epoch 23, Batch 301 loss: 3.605464
Epoch: 23       Training Loss: 3.610316         Validation Loss: 3.761326
Epoch 24, Batch 1 loss: 3.825171
Epoch 24, Batch 101 loss: 3.532915
Epoch 24, Batch 201 loss: 3.575519
Epoch 24, Batch 301 loss: 3.583894
Epoch: 24       Training Loss: 3.583815         Validation Loss: 3.633060
Validation loss decreased (3.694498 --> 3.633060).  Saving model …
Epoch 25, Batch 1 loss: 3.774716
Epoch 25, Batch 101 loss: 3.549665
Epoch 25, Batch 201 loss: 3.567695
Epoch 25, Batch 301 loss: 3.567471
Epoch: 25       Training Loss: 3.568363         Validation Loss: 3.669594
```

[14]:
```
<All keys matched successfully>
```

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%.

[15]:
```python
def test(loaders, model, criterion, use_cuda):

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

```
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data -␣
↪test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().
↪numpy())
        total += data.size(0)

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

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

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

```
Test Loss: 3.400714


Test Accuracy: 18% (157/836)
```

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.

[27]:
```
## TODO: Specify data loaders
transfer_loaders = data_loaders.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`.

[21]:
```
import torchvision.models as models
import torch.nn as nn

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

for i in transfer_model.parameters():
    i.requires_grad = False

transfer_model.fc = nn.Linear(2048, 133, bias=True)
parameters_fc = transfer_model.fc.parameters()

for j in parameters_fc:
    j.requires_grad = True

transfer_model


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

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

Answer: I used the ResNet archtiecture to create my transfer model because it's good at Image Classification.I've modified the final Fully-connected layer and replaced with a Fully-connected layer with an output of 133 which represents the number of dog breed 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.

[23]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(transfer_model.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'`.

[24]:
```
# train the model

transfer_model = train(25, transfer_loaders, transfer_model,␣
↪optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')


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

```
Epoch 1, Batch 1 loss: 5.099311
Epoch 1, Batch 101 loss: 4.925706
Epoch 1, Batch 201 loss: 4.885988
Epoch 1, Batch 301 loss: 4.846992
Epoch: 1        Training Loss: 4.833614         Validation Loss: 4.665434
Validation loss decreased (inf --> 4.665434).  Saving model …
Epoch 2, Batch 1 loss: 4.806275
Epoch 2, Batch 101 loss: 4.676220
Epoch 2, Batch 201 loss: 4.646351
Epoch 2, Batch 301 loss: 4.618983
Epoch: 2        Training Loss: 4.608865         Validation Loss: 4.422648
Validation loss decreased (4.665434 --> 4.422648).  Saving model …
Epoch 3, Batch 1 loss: 4.386357
Epoch 3, Batch 101 loss: 4.471227
Epoch 3, Batch 201 loss: 4.445383
Epoch 3, Batch 301 loss: 4.423930
Epoch: 3        Training Loss: 4.415401         Validation Loss: 4.197909
Validation loss decreased (4.422648 --> 4.197909).  Saving model …
Epoch 4, Batch 1 loss: 4.130036
Epoch 4, Batch 101 loss: 4.286703
Epoch 4, Batch 201 loss: 4.270967
Epoch 4, Batch 301 loss: 4.240128
Epoch: 4        Training Loss: 4.230893         Validation Loss: 3.977608
Validation loss decreased (4.197909 --> 3.977608).  Saving model …
Epoch 5, Batch 1 loss: 4.224308
Epoch 5, Batch 101 loss: 4.109378
Epoch 5, Batch 201 loss: 4.078003
Epoch 5, Batch 301 loss: 4.062603
Epoch: 5        Training Loss: 4.052932         Validation Loss: 3.758017
Validation loss decreased (3.977608 --> 3.758017).  Saving model …
Epoch 6, Batch 1 loss: 3.964343
Epoch 6, Batch 101 loss: 3.924499
Epoch 6, Batch 201 loss: 3.902728
```

```
Epoch 6, Batch 301 loss: 3.888518
Epoch: 6        Training Loss: 3.887339         Validation Loss: 3.558235
Validation loss decreased (3.758017 --> 3.558235).  Saving model …
Epoch 7, Batch 1 loss: 3.769019
Epoch 7, Batch 101 loss: 3.766655
Epoch 7, Batch 201 loss: 3.752321
Epoch 7, Batch 301 loss: 3.737102
Epoch: 7        Training Loss: 3.729060         Validation Loss: 3.383438
Validation loss decreased (3.558235 --> 3.383438).  Saving model …
Epoch 8, Batch 1 loss: 3.606258
Epoch 8, Batch 101 loss: 3.601114
Epoch 8, Batch 201 loss: 3.601942
Epoch 8, Batch 301 loss: 3.580110
Epoch: 8        Training Loss: 3.574330         Validation Loss: 3.217206
Validation loss decreased (3.383438 --> 3.217206).  Saving model …
Epoch 9, Batch 1 loss: 3.706635
Epoch 9, Batch 101 loss: 3.459916
Epoch 9, Batch 201 loss: 3.458233
Epoch 9, Batch 301 loss: 3.434215
Epoch: 9        Training Loss: 3.428900         Validation Loss: 3.030002
Validation loss decreased (3.217206 --> 3.030002).  Saving model …
Epoch 10, Batch 1 loss: 3.685843
Epoch 10, Batch 101 loss: 3.326163
Epoch 10, Batch 201 loss: 3.312447
Epoch 10, Batch 301 loss: 3.302534
Epoch: 10       Training Loss: 3.296596         Validation Loss: 2.886613
Validation loss decreased (3.030002 --> 2.886613).  Saving model …
Epoch 11, Batch 1 loss: 2.784620
Epoch 11, Batch 101 loss: 3.194579
Epoch 11, Batch 201 loss: 3.188428
Epoch 11, Batch 301 loss: 3.177142
Epoch: 11       Training Loss: 3.172974         Validation Loss: 2.734970
Validation loss decreased (2.886613 --> 2.734970).  Saving model …
Epoch 12, Batch 1 loss: 3.109624
Epoch 12, Batch 101 loss: 3.110062
Epoch 12, Batch 201 loss: 3.080580
Epoch 12, Batch 301 loss: 3.058492
Epoch: 12       Training Loss: 3.057135         Validation Loss: 2.598874
Validation loss decreased (2.734970 --> 2.598874).  Saving model …
Epoch 13, Batch 1 loss: 2.634591
Epoch 13, Batch 101 loss: 2.970141
Epoch 13, Batch 201 loss: 2.955913
Epoch 13, Batch 301 loss: 2.950608
Epoch: 13       Training Loss: 2.945253         Validation Loss: 2.490359
Validation loss decreased (2.598874 --> 2.490359).  Saving model …
Epoch 14, Batch 1 loss: 2.621831
Epoch 14, Batch 101 loss: 2.858073
Epoch 14, Batch 201 loss: 2.869595
```

```
Epoch 14, Batch 301 loss: 2.848950
Epoch: 14       Training Loss: 2.842190         Validation Loss: 2.359793
Validation loss decreased (2.490359 --> 2.359793).  Saving model …
Epoch 15, Batch 1 loss: 2.377192
Epoch 15, Batch 101 loss: 2.771111
Epoch 15, Batch 201 loss: 2.772430
Epoch 15, Batch 301 loss: 2.761443
Epoch: 15       Training Loss: 2.754136         Validation Loss: 2.279238
Validation loss decreased (2.359793 --> 2.279238).  Saving model …
Epoch 16, Batch 1 loss: 2.629310
Epoch 16, Batch 101 loss: 2.677885
Epoch 16, Batch 201 loss: 2.648466
Epoch 16, Batch 301 loss: 2.649792
Epoch: 16       Training Loss: 2.641090         Validation Loss: 2.189802
Validation loss decreased (2.279238 --> 2.189802).  Saving model …
Epoch 17, Batch 1 loss: 2.732138
Epoch 17, Batch 101 loss: 2.599465
Epoch 17, Batch 201 loss: 2.593549
Epoch 17, Batch 301 loss: 2.580875
Epoch: 17       Training Loss: 2.574208         Validation Loss: 2.057765
Validation loss decreased (2.189802 --> 2.057765).  Saving model …
Epoch 18, Batch 1 loss: 2.417754
Epoch 18, Batch 101 loss: 2.490932
Epoch 18, Batch 201 loss: 2.487750
Epoch 18, Batch 301 loss: 2.490162
Epoch: 18       Training Loss: 2.490320         Validation Loss: 1.993343
Validation loss decreased (2.057765 --> 1.993343).  Saving model …
Epoch 19, Batch 1 loss: 2.666246
Epoch 19, Batch 101 loss: 2.455063
Epoch 19, Batch 201 loss: 2.440416
Epoch 19, Batch 301 loss: 2.429405
Epoch: 19       Training Loss: 2.423704         Validation Loss: 1.931031
Validation loss decreased (1.993343 --> 1.931031).  Saving model …
Epoch 20, Batch 1 loss: 2.028595
Epoch 20, Batch 101 loss: 2.374100
Epoch 20, Batch 201 loss: 2.383795
Epoch 20, Batch 301 loss: 2.363088
Epoch: 20       Training Loss: 2.356963         Validation Loss: 1.844818
Validation loss decreased (1.931031 --> 1.844818).  Saving model …
Epoch 21, Batch 1 loss: 2.643274
Epoch 21, Batch 101 loss: 2.317666
Epoch 21, Batch 201 loss: 2.308668
Epoch 21, Batch 301 loss: 2.292697
Epoch: 21       Training Loss: 2.291370         Validation Loss: 1.785658
Validation loss decreased (1.844818 --> 1.785658).  Saving model …
Epoch 22, Batch 1 loss: 2.081243
Epoch 22, Batch 101 loss: 2.244416
Epoch 22, Batch 201 loss: 2.240983
```

```
Epoch 22, Batch 301 loss: 2.233562
Epoch: 22       Training Loss: 2.235034         Validation Loss: 1.711865
Validation loss decreased (1.785658 --> 1.711865).  Saving model …
Epoch 23, Batch 1 loss: 1.629980
Epoch 23, Batch 101 loss: 2.179592
Epoch 23, Batch 201 loss: 2.184750
Epoch 23, Batch 301 loss: 2.185357
Epoch: 23       Training Loss: 2.179716         Validation Loss: 1.662499
Validation loss decreased (1.711865 --> 1.662499).  Saving model …
Epoch 24, Batch 1 loss: 2.786542
Epoch 24, Batch 101 loss: 2.165208
Epoch 24, Batch 201 loss: 2.155458
Epoch 24, Batch 301 loss: 2.143380
Epoch: 24       Training Loss: 2.133512         Validation Loss: 1.605393
Validation loss decreased (1.662499 --> 1.605393).  Saving model …
Epoch 25, Batch 1 loss: 2.129339
Epoch 25, Batch 101 loss: 2.090601
Epoch 25, Batch 201 loss: 2.066656
Epoch 25, Batch 301 loss: 2.068623
Epoch: 25       Training Loss: 2.069548         Validation Loss: 1.564308
Validation loss decreased (1.605393 --> 1.564308).  Saving model …
```

[24]:
```
<All keys matched successfully>
```

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%.

[44]:
```
test(transfer_loaders, transfer_model, criterion_transfer, use_cuda)
```

```
Test Loss: 1.442572


Test Accuracy: 76% (643/836)
```

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.

[33]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# list of class names by index, i.e. a name can be accessed like class_names[0]
```

```
class_names = [item[4:].replace("_", " ") for item in transfer_loaders['train'].
↪dataset.classes]

def predict_breed_transfer(model, class_names,img_path):
    # load the image and return the predicted breed
    image = Image.open(img_path).convert('RGB')
    transform_prediction = transforms.Compose([transforms.Resize(size=(224,␣
↪224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.
↪406],
                                                         std=[0.229, 0.224, 0.225])])

    # discard the transparent, alpha channel (that's the :3) and add the batch␣
↪dimension
    image = transform_prediction(image)[:3,:,:].unsqueeze(0)
    model = model.cpu()
    model.eval()
    idx = torch.argmax(model(image))
    return class_names[idx]
```

[34]:
```
for i in os.listdir('./images'):
    image_path = os.path.join('./images', i)
    predition = predict_breed_transfer(transfer_model, class_names, image_path)
    print("image_file_name: {0}, \t predition breed: {1}".format(image_path,␣
↪predition))
```

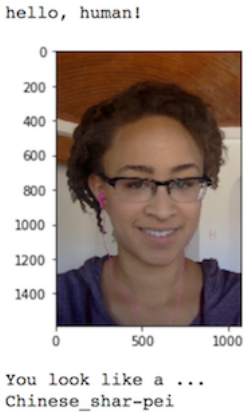
```
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,      predition
breed: Welsh springer spaniel
image_file_name: ./images/sample_dog_output.png,         predition breed:
Italian greyhound
image_file_name: ./images/Curly-coated_retriever_03896.jpg,      predition
breed: Curly-coated retriever
image_file_name: ./images/sample_cnn.png,        predition breed: American
eskimo dog
image_file_name: ./images/Labrador_retriever_06449.jpg,          predition
breed: Flat-coated retriever
image_file_name: ./images/sample_human_output.png,       predition breed:
Bulldog
image_file_name: ./images/American_water_spaniel_00648.jpg,      predition
breed: Curly-coated retriever
image_file_name: ./images/Labrador_retriever_06457.jpg,          predition
breed: Labrador retriever
image_file_name: ./images/Labrador_retriever_06455.jpg,          predition
breed: Chesapeake bay retriever
image_file_name: ./images/Brittany_02625.jpg,    predition breed: Brittany
```

Step 5: Write your Algorithm

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

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

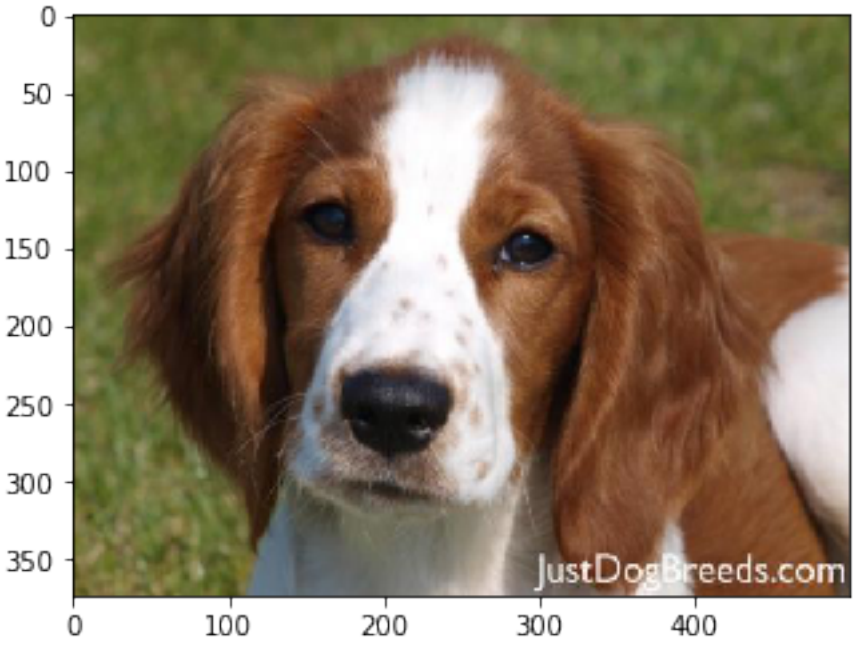
Some sample output for our algorithm is provided below, but feel free to design your own user experience!



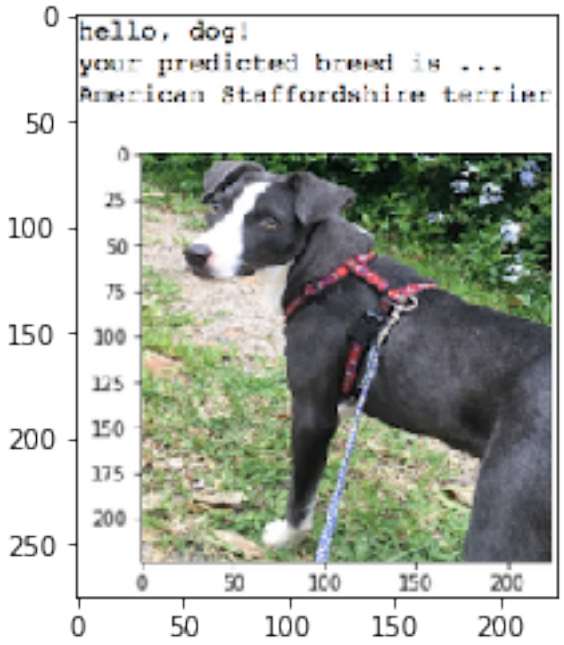
1.1.18 (IMPLEMENTATION) Write your Algorithm

[37]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    image = Image.open(img_path)
    plt.imshow(image)
    plt.show()
    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(transfer_model, class_names,␣
↪img_path)
        print("Dog Detected!\n Probably of dog bread {0}".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(transfer_model, class_names,␣
↪img_path)
        print("Hello, human!\You look like a {0}".format(prediction))
    else:
        print("Oops! Can't recognise anything..")
```

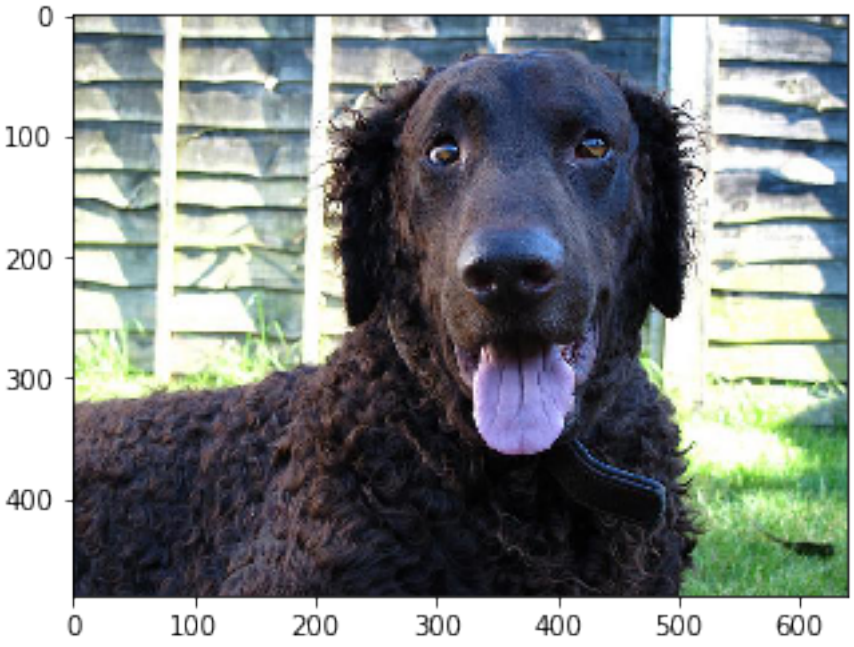
```
for i in os.listdir('./images'):
    image_path = os.path.join('./images', i)
    run_app(image_path)
```




```
Dog Detected!
 Probably of dog bread Welsh springer spaniel
```

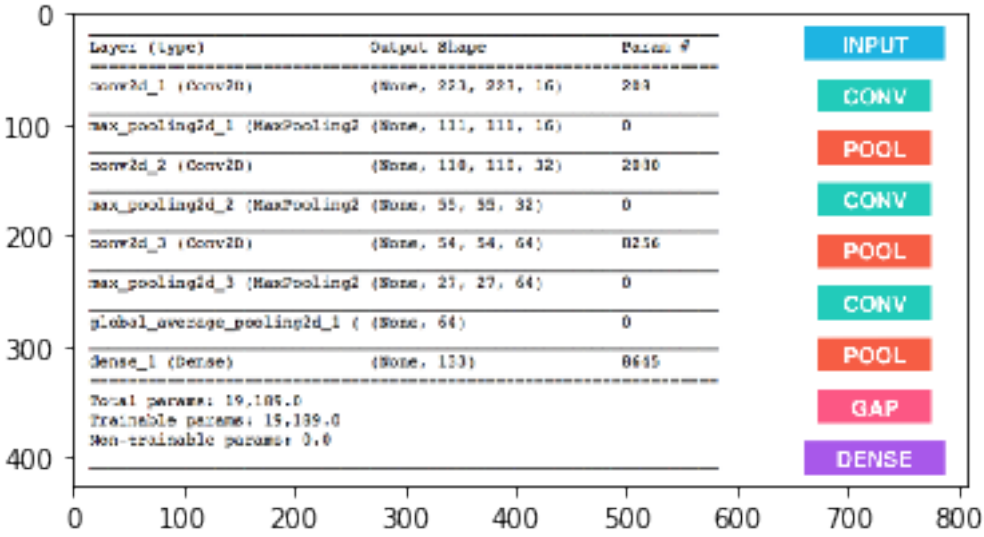



```
Dog Detected!
 Probably of dog bread Italian greyhound
```



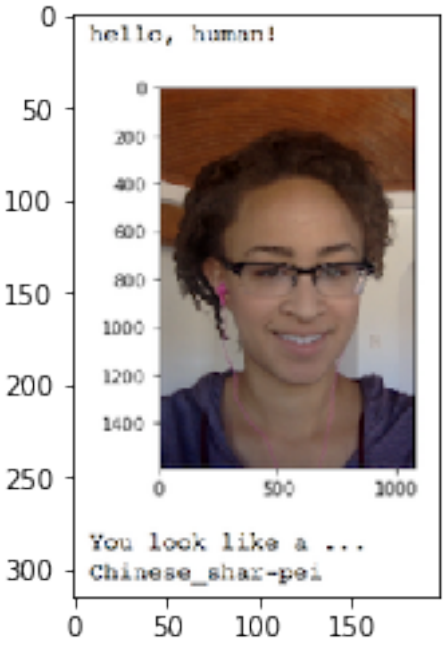
```
Dog Detected!
```

Probably of dog bread Curly-coated retriever



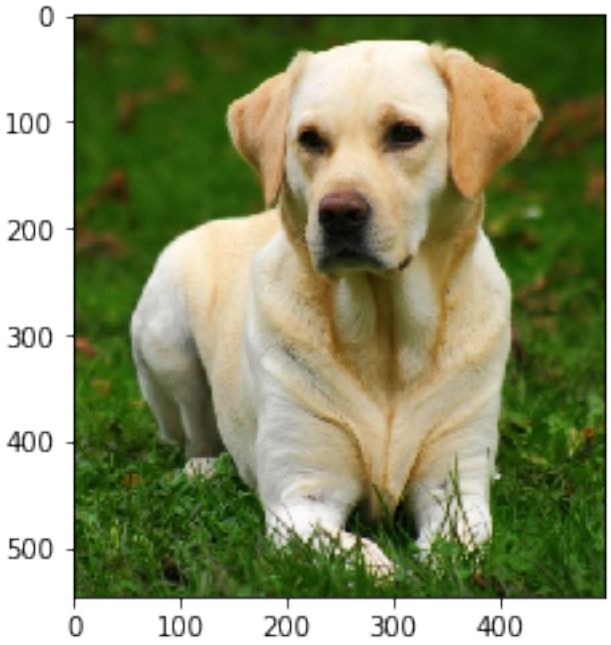
Oops! Can't recognise anything..

Dog Detected!
Probably of dog bread Flat-coated retriever

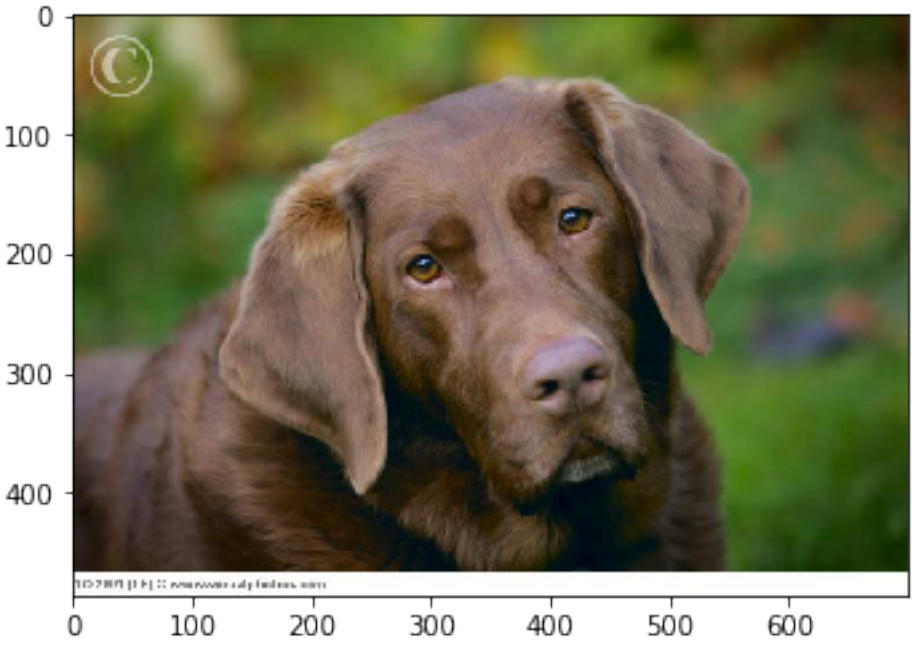



```
Hello, human!\You look like a Bulldog
```

```
Dog Detected!
 Probably of dog bread Curly-coated retriever
```

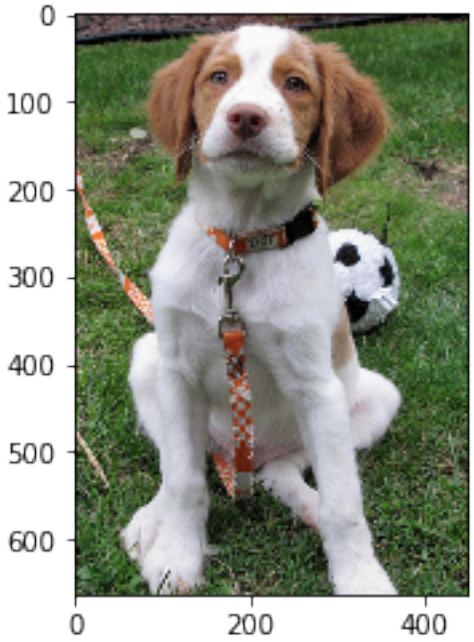



```
Dog Detected!
 Probably of dog bread Labrador retriever
```




```
Dog Detected!
```

```
Probably of dog bread Chesapeake bay retriever
```




```
Dog Detected!
 Probably of dog bread Brittany
```

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.

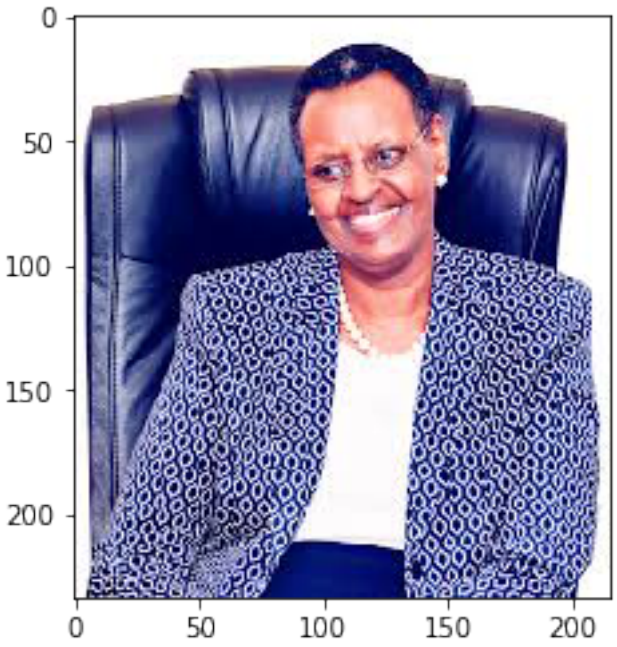
Answer: (Three possible points for improvement)

[43]:
```
## 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.
new_human_files = np.array(glob("sample_images/humans/*"))
new_dog_files = np.array(glob("sample_images/dogs/*"))
```

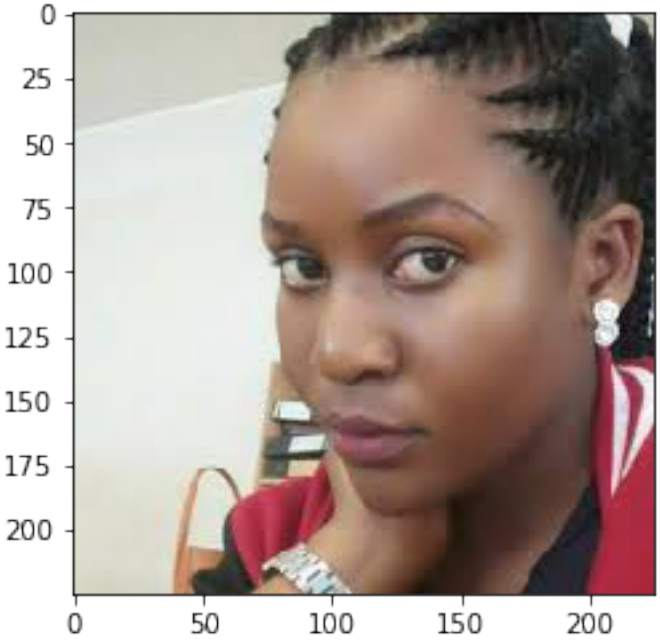
```
## suggested code, below
for file in np.hstack((new_human_files, new_dog_files)):
    run_app(file)
```


```
Hello, human!\You look like a Irish water spaniel
```

```
Hello, human!\You look like a Dachshund
```

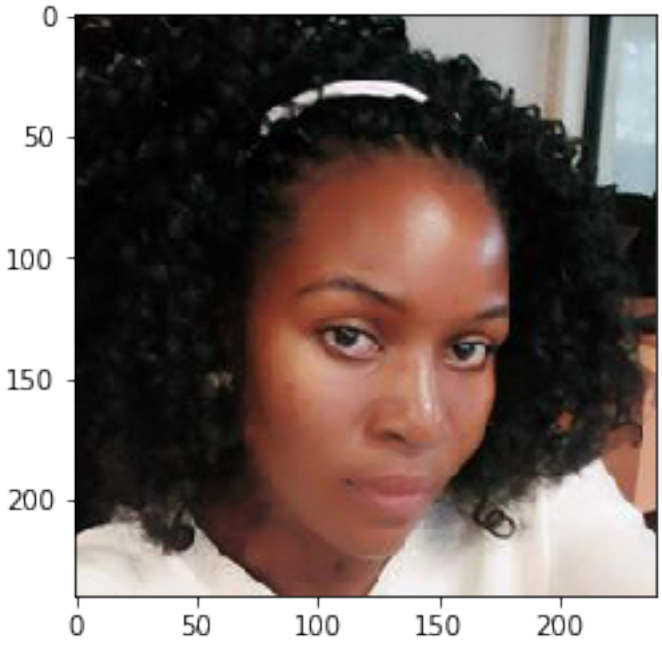



```
Hello, human!\You look like a Dalmatian
```

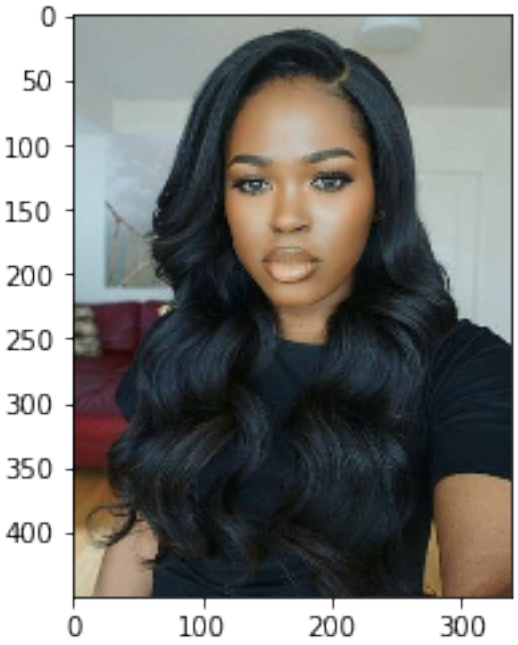


```
Hello, human!\You look like a Chinese crested
```


```
Hello, human!\You look like a Bull terrier
```

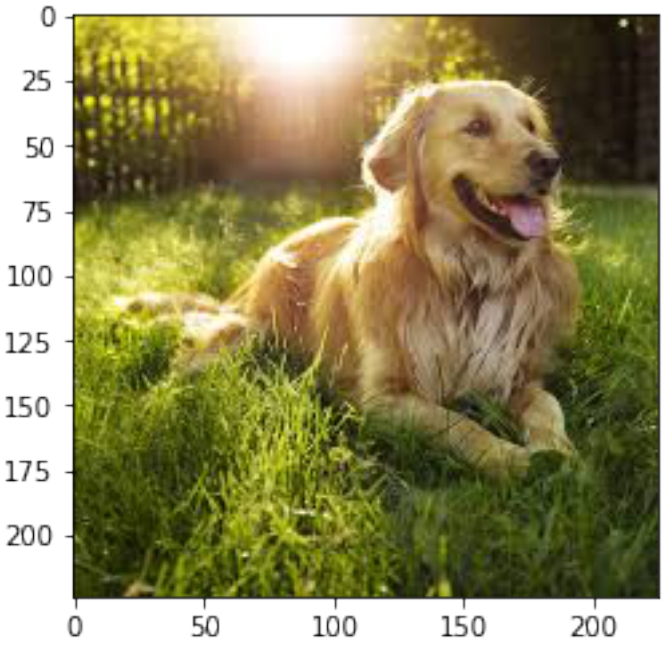



```
Hello, human!\You look like a Poodle
```

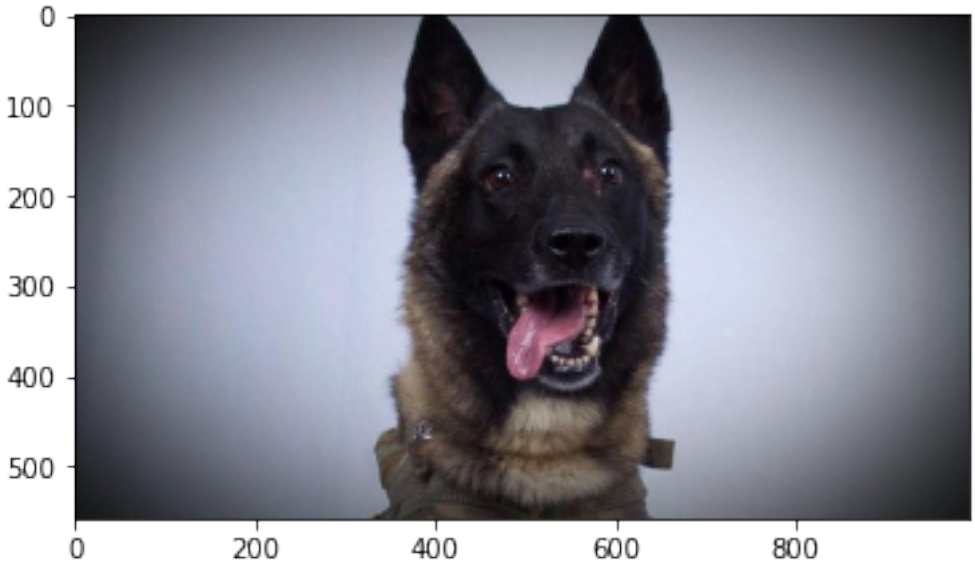



```
Hello, human!\You look like a Havanese
```


```
Dog Detected!
 Probably of dog bread Alaskan malamute
```

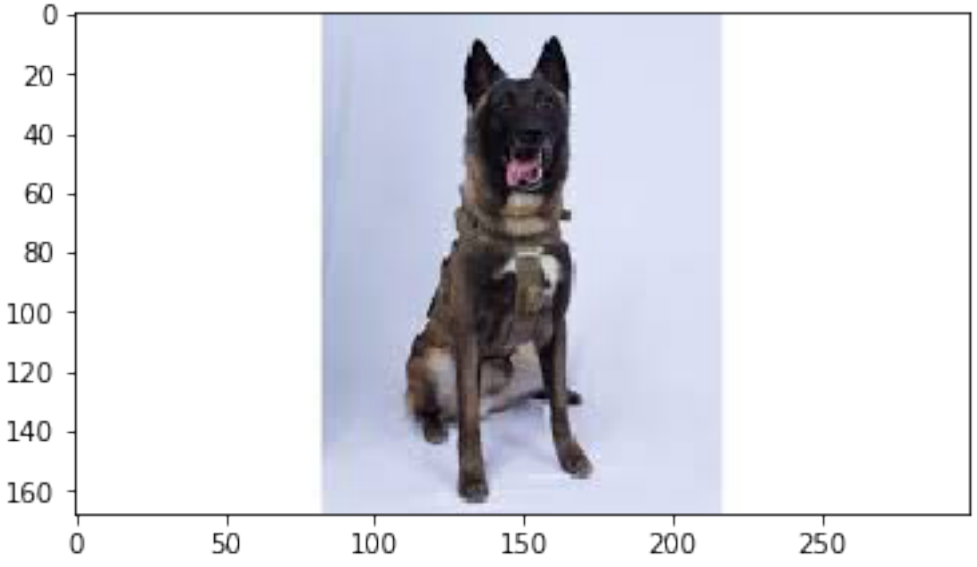



```
Dog Detected!
 Probably of dog bread Golden retriever
```

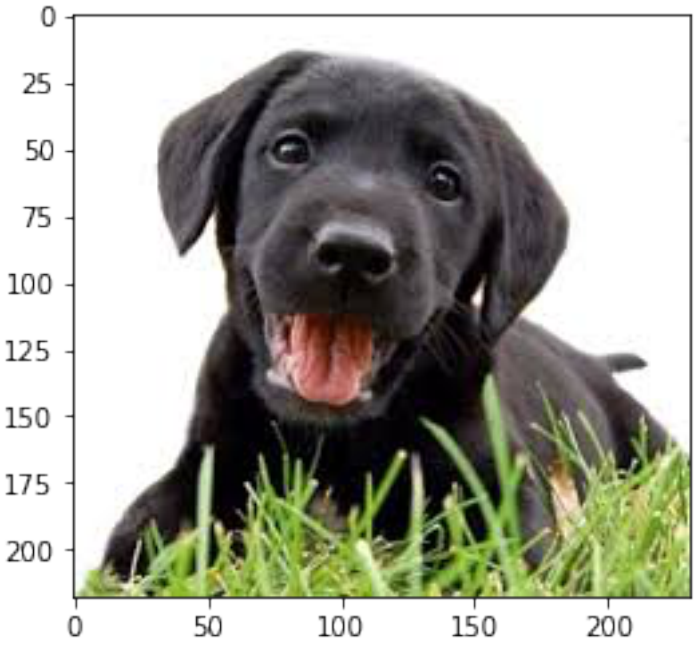



```
Dog Detected!
 Probably of dog bread German shepherd dog
```

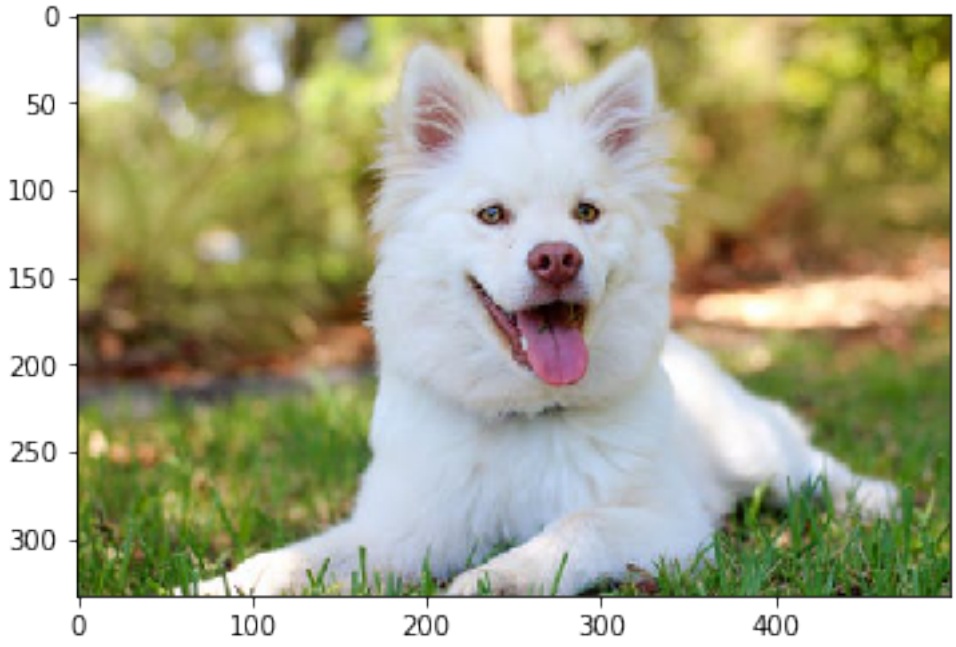

```
Dog Detected!
 Probably of dog bread Dachshund
```




```
Oops! Can't recognise anything..
```

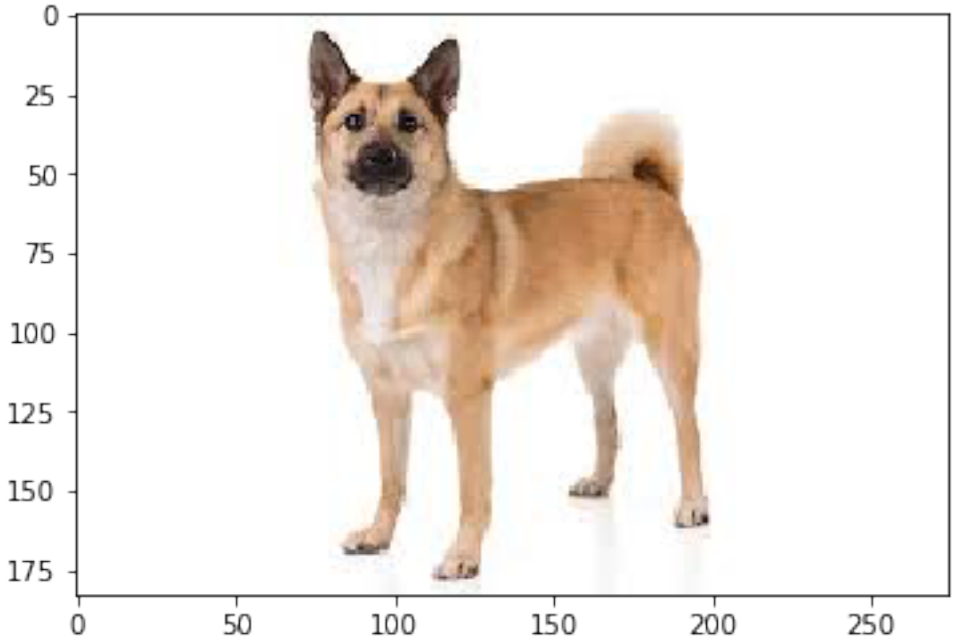



```
Dog Detected!
 Probably of dog bread Flat-coated retriever
```




```
Dog Detected!
```

```
Probably of dog bread American eskimo dog
```




```
Dog Detected!
 Probably of dog bread Akita
```

[ ]: