Convolutional Neural Networks

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.

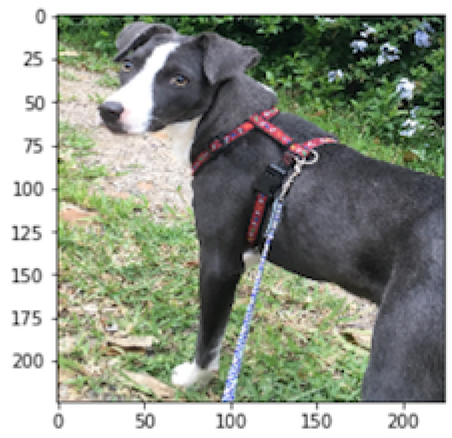
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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.

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```



In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

- Step 0: Import Datasets
- Step 1: Detect Humans
- Step 2: Detect Dogs
- Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
- Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 5: Write your Algorithm
- Step 6: Test Your Algorithm

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`.

```
# Download the dog dataset
!wget -c https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
```

```
!unzip dogImages.zip


# Download the humans dataset
!wget -c https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
!unzip lfw.zip


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.

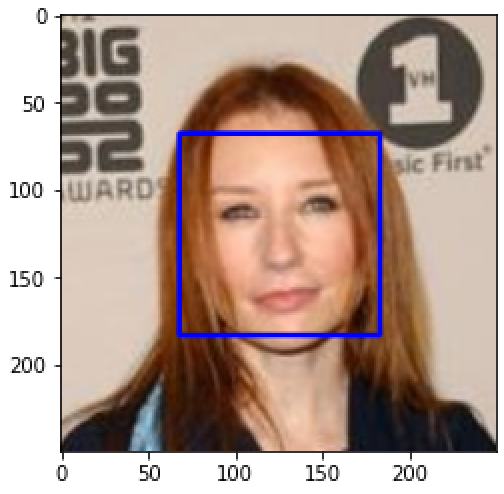
```
# Make haarcascades folder in colab
%mkdir haarcascades
# Download haarcascade file
!wget -O haarcascades/haarcascade_frontalface_alt.xml https://raw.githubusercontent.com/opencv/opencv/m
```

```
--2020-07-26 05:04:45--  https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.133,
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... connect
HTTP request sent, awaiting response... 200 OK
Length: 676709 (661K) [text/plain]
Saving to: 'haarcascades/haarcascade_frontalface_alt.xml'

haarcascades/haarca 100%[===================>] 660.85K  --.-KB/s    in 0.03s

2020-07-26 05:04:46 (20.8 MB/s) - 'haarcascades/haarcascade_frontalface_alt.xml' saved [676709/67
```

```
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
    img = 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.

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.

```
# 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
```

(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:

1. Percentage of detected human face in the first 100 images in human_files is 99%.

2. Percentage of detected human face in the first 100 images in dog_files is 9%.

```
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.
total_faces_human = 0
total_faces_dog = 0

for human,dog in zip(human_files_short, dog_files_short):
  # find human faces in human_files_short
  faces_human = face_detector(human)
  # find human faces in dog_files_short
  faces_dog = face_detector(dog)
  # calculating total human faces in human_files_short
  total_faces_human += faces_human
  # calculating total human faces in dog_files_short
  total_faces_dog += faces_dog

# print percentage of detected faces in human_files and dog_files
print(f"Percentage of detected faces in human_files is {(total_faces_human / 100) * 100} %")
print(f"Percentage of detected faces in dog_files is {(total_faces_dog / 100) * 100} %")
```

```
Percentage of detected faces in human_files is 99.0 %
Percentage of detected faces in dog_files is 9.0 %
```

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`.

```
### (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.

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.

```
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.

(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.

```
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
    ## Return the *index* of the predicted class for that image

    # read image
    image = Image.open(img_path)

    # define transforms
    transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485,0.456,0.406],
                                                          [0.229,0.224,0.225])])
    # apply transformation on image
    image = transform(image)
    image = image.unsqueeze(0)

    # move image to GPU if CUDA available
    if use_cuda:
      image = image.cuda()

    # calculate predicted class index from output of forward pass
    output = VGG16(image)
    _, pred = torch.max(output, 1)




    return pred # predicted class index
```

(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).

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    pred_idx = VGG16_predict(img_path)
    if pred_idx >=151 and pred_idx <= 268:
      return True
    else:
      return False
```

(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:

1. Percentage of detected dog faces in the first 100 images in human_files is 1.0 %
2. Percentage of detected dog faces in the first 100 images in dog_files is 97.0 %

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
total_faces_human = 0
total_faces_dog = 0

for human,dog in zip(human_files_short, dog_files_short):
  # find dog faces in human files short
  # find dog faces in human_files_short
  faces_human = dog_detector(human)
  # find dog faces in dog_files_short
  faces_dog = dog_detector(dog)
  # calculating total dog faces in human_files_short
  total_faces_human += faces_human
  # calculating total dog faces in dog_files_short
  total_faces_dog += faces_dog

# print percentage of detected dog faces in human_files and dog_files
print(f"Percentage of detected dog faces in human_files is {(total_faces_human / 100) * 100} %")
print(f"Percentage of detected dog faces in dog_files is {(total_faces_dog / 100) * 100} %")
```

```
Percentage of detected dog faces in human_files is 1.0 %
Percentage of detected dog faces in dog_files is 97.0 %
```

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`.

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

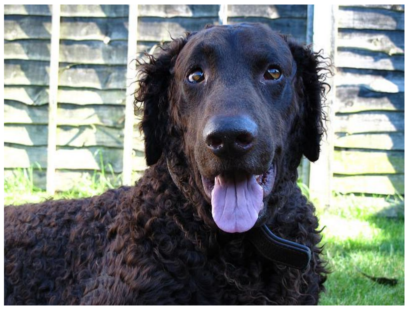
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.

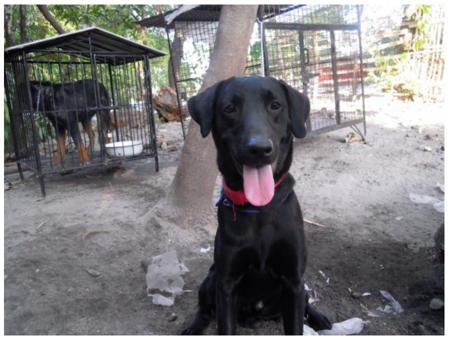
Brittany	Welsh Springer Spaniel

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 | Black 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!

(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!

```
import os
import torch
from torchvision import datasets
import torchvision.transforms as transforms

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

```
# define transforms
transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485,0.456,0.406],
                                                      [0.229,0.224,0.225])])

# define train, valid and test directory
data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

# define train, valid and test datasets
train_data = datasets.ImageFolder(train_dir, transform=transform)
valid_data = datasets.ImageFolder(valid_dir, transform=transform)
test_data = datasets.ImageFolder(test_dir, transform=transform)

# define train, valid and test loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=64, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=64, shuffle=True)

# define dictionary for loaders
loaders_scratch = {"train":train_loader, "valid":valid_loader, "test":test_loader}
```

Question 3: Describe your chosen procedure for preprocessing the data.

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

Answer:

Resizing Images

Images are cropped to size of 224 * 224 as these were the dimensions on which VGG16 model was originally trained.

Data Augmentation

I trained the model without applying data augmentation. The reason behind for not applying data augmentation is that I could achieve a test accuracy of greater than 10% as required. Data augmentation prevents overfitting of model and helps in generalizing and improves the performance on test dataset.

(IMPLEMENTATION) Model Architecture

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

```
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 layer
        self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding = 1)

        # pooling layer
        self.pool1 = nn.MaxPool2d(2,2)

        # fully connected layers
        self.fc1 = nn.Linear(28 * 28 * 64, 512)
        self.fc2 = nn.Linear(512, 133)

        # dropout layer
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):

        ## Define forward behavior
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool1(F.relu(self.conv2(x)))
        x = self.pool1(F.relu(self.conv3(x)))
        x = x.view(-1, 28 * 28 * 64)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer:

Steps for final CNN architecture

I first tried with five convolutional layers and three fully connected layers. The model seems to be overfitting even by fine tuning the hyperparameters like learning rate, . So I tried to reduce the complexity of the model by taking three convolutional layers and two fully connected layers as described in the Network class above and the model seems to be a good fit as I saw the training loss decreasing till 20 epochs. I increased the number of epochs from 20 to 40 and changing the learning rate from 0.003 to 0.001 to achieve the desired accuracy of 10%.

Explanation of CNN architecture

Three convolutional layers with kernel size of 3 and stride of 1 are used. In the first convolutional layer, colored input image has a depth of 3 and 16 filtered images are produced. These 16 filtered images serve as input to second convolutional layer which produces a layer with depth of 32. Now these 32 filtered images serve as input to third convolutional layer which produces a layer with depth of 64. Maxpooling layer will the downsize the dimensions of image by a factor of 2. The parameters left when the image passes through the convolutional layers followed by the relu activation and pooling layers will be 28 * 28 * 64 which will be input to the fully connected layers.The number of hidden neurons for the first fully connected layer is 512 which is decided taking in the consideration of 133 output classes in the second fully connected layer. A dropout of 0.5 is used to avoid overfitting and applied to the first fully connected layer.

(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.

```
import torch.optim as optim

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

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

(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'`.

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

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
```

```
valid_loss_min = np.inf

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

    ###################
    # train the model #
    ###################
    model.train()
    for batch_idx, (data, target) in enumerate(loaders['train']):
    # for data, target in 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))
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output,target)
        loss.backward()
        optimizer.step()
        train_loss += ((1 / (batch_idx +1)) * (loss.data - train_loss))

    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['valid']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## update the average validation loss
        output = model(data)
        loss = criterion(output, target)
        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:
      print('Validation loss is decreasing from {:.6f} to {:.6f}'.format(
          valid_loss_min,
          valid_loss
          ))
      torch.save(model.state_dict(), save_path)
```

```
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
model_scratch = train(40, 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: 4.898773         Validation Loss: 4.818474
Validation loss is decreasing from inf to 4.818474
Epoch: 2        Training Loss: 4.780053         Validation Loss: 4.675214
Validation loss is decreasing from 4.818474 to 4.675214
Epoch: 3        Training Loss: 4.607555         Validation Loss: 4.586164
Validation loss is decreasing from 4.675214 to 4.586164
Epoch: 4        Training Loss: 4.515732         Validation Loss: 4.518387
Validation loss is decreasing from 4.586164 to 4.518387
Epoch: 5        Training Loss: 4.452729         Validation Loss: 4.410076
Validation loss is decreasing from 4.518387 to 4.410076
Epoch: 6        Training Loss: 4.381900         Validation Loss: 4.366341
Validation loss is decreasing from 4.410076 to 4.366341
Epoch: 7        Training Loss: 4.340976         Validation Loss: 4.301746
Validation loss is decreasing from 4.366341 to 4.301746
Epoch: 8        Training Loss: 4.274848         Validation Loss: 4.255956
Validation loss is decreasing from 4.301746 to 4.255956
Epoch: 9        Training Loss: 4.227596         Validation Loss: 4.177423
Validation loss is decreasing from 4.255956 to 4.177423
Epoch: 10       Training Loss: 4.150698         Validation Loss: 4.092969
Validation loss is decreasing from 4.177423 to 4.092969
Epoch: 11       Training Loss: 4.106104         Validation Loss: 4.105625
```

(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%.

```
Validation loss is decreasing from 4.043239 to 3.984571
```

```
def test(loaders, model, criterion, use_cuda):

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

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

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

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

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

```
Test Loss: 3.648138


Test Accuracy: 13% (116/836)
```

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.

(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.

```
import os
import torch
from torchvision import datasets
import torchvision.transforms as transforms
import torch.optim as optim

## TODO: Specify data loaders

# Define training and validation transforms
train_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                    transforms.RandomRotation(10),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485,0.456,0.406],
                                                      [0.229,0.224,0.225])])

val_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485,0.456,0.406],
                                                     [0.229,0.224,0.225])])

test_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485,0.456,0.406],
                                                     [0.229,0.224,0.225])])

# Define directoru for training, validation and test data
data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train/')
```

```
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

# Make datasets with the help of directories
train_data = datasets.ImageFolder(train_dir, transform=train_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=val_transform)
test_data = datasets.ImageFolder(test_dir, transform=test_transform)
data_transfer = {"train":train_data, "valid":valid_data, "test":test_data}

# Define data loaders for training, validation and test data
train_loader = torch.utils.data.DataLoader(train_data, batch_size=64, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=64, shuffle=True)
loaders_transfer = {"train":train_loader, "valid":valid_loader, "test":test_loader}
```

(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`.

```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.vgg19(pretrained = True)
print(model_transfer)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.cache/torch/check
100%                                        548M/548M [1:34:38<00:00, 101kB/s]

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

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

# Replace last linear layer of classifier
model_transfer.classifier[6].out_features = 133

# Check if GPU is available
use_cuda = torch.cuda.is_available()

# Move the model to GPU if GPU is available
if use_cuda:
    model_transfer = model_transfer.cuda()
```

```
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
```

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.

```
    (2): Dropout(p=0.5, inplace=False)
```

Answer:

I used VGG19 pretrained model as it is used for classification and transfer learning.VGG-19 is a convolutional neural network trained on million number of images like animals, vehicles, household objects etc from ImageNet database. Here the dataset used is smaller and similar to the ImageNet database so I changed the output of the last fully connected layer to 133, freezed all the weights of feature layers and updated weights by training the classifier layers.

(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.

```
# Define loss function
criterion_transfer = nn.CrossEntropyLoss()

# Define optimizer
optimizer_transfer = optim.Adam(model_transfer.parameters(), lr=0.001)
```

(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'`.

```
# Copying the train function
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
        # for data, target in 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))
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output,target)
            loss.backward()
            optimizer.step()
            train_loss += ((1 / (batch_idx +1)) * (loss.data - train_loss))
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            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:
          print('Validation loss is decreasing from {:.6f} to {:.6f}'.format(
              valid_loss_min,
              valid_loss
              ))
          torch.save(model.state_dict(), save_path)
          valid_loss_min = valid_loss

    # return trained model
    return model



# train the model
n_epochs = 40
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transf

# 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: 2.536671         Validation Loss: 1.560666
Validation loss is decreasing from inf to 1.560666
Epoch: 2        Training Loss: 2.475343         Validation Loss: 1.401648
Validation loss is decreasing from 1.560666 to 1.401648
Epoch: 3        Training Loss: 2.488996         Validation Loss: 1.401282
Validation loss is decreasing from 1.401648 to 1.401282
Epoch: 4        Training Loss: 2.373926         Validation Loss: 1.420215
Epoch: 5        Training Loss: 2.311528         Validation Loss: 1.376946
Validation loss is decreasing from 1.401282 to 1.376946
Epoch: 6        Training Loss: 2.385478         Validation Loss: 1.327383
Validation loss is decreasing from 1.376946 to 1.327383
Epoch: 7        Training Loss: 2.413022         Validation Loss: 1.458666
Epoch: 8        Training Loss: 2.404610         Validation Loss: 1.613971
Epoch: 9        Training Loss: 2.409312         Validation Loss: 1.372239
Epoch: 10       Training Loss: 2.419741         Validation Loss: 1.689238
Epoch: 11       Training Loss: 2.341285         Validation Loss: 1.494080
Epoch: 12       Training Loss: 2.335911         Validation Loss: 1.445717
Epoch: 13       Training Loss: 2.400214         Validation Loss: 1.465824
Epoch: 14       Training Loss: 2.359194         Validation Loss: 1.491992
Epoch: 15       Training Loss: 2.339470         Validation Loss: 1.321142
Validation loss is decreasing from 1.327383 to 1.321142
Epoch: 16       Training Loss: 2.268065         Validation Loss: 1.406668
Epoch: 17       Training Loss: 2.381537         Validation Loss: 1.392948
Epoch: 18       Training Loss: 2.336616         Validation Loss: 1.286697
Validation loss is decreasing from 1.321142 to 1.286697
Epoch: 19       Training Loss: 2.340639         Validation Loss: 1.233015
Validation loss is decreasing from 1.286697 to 1.233015
Epoch: 20       Training Loss: 2.330301         Validation Loss: 1.489987
Epoch: 21       Training Loss: 2.289911         Validation Loss: 1.368890
Epoch: 22       Training Loss: 2.255482         Validation Loss: 1.388444
Epoch: 23       Training Loss: 2.237954         Validation Loss: 1.509675
Epoch: 24       Training Loss: 2.381832         Validation Loss: 1.292680
Epoch: 25       Training Loss: 2.227059         Validation Loss: 1.364152
Epoch: 26       Training Loss: 2.277735         Validation Loss: 1.389796
Epoch: 27       Training Loss: 2.254157         Validation Loss: 1.396768
Epoch: 28       Training Loss: 2.294016         Validation Loss: 1.362260
Epoch: 29       Training Loss: 2.298254         Validation Loss: 1.338849
```

(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%.

```
Epoch: 36       Training Loss: 2.274654         Validation Loss: 1.523675
```

```
# Copying the test function
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
        # 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))
```

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.365663


Test Accuracy: 62% (524/836)
```

(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.

```
### 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 data_transfer['train'].classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed

    # Read image
    image = Image.open(img_path)

    # Define transforms
    transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485,0.456,0.406],
                                                         [0.229,0.224,0.225])])

    # Applying necessary transformation to image
    image = transform(image)
    image = image.unsqueeze(0)
    # Move the image to GPU if it is available
    if use_cuda:
      image = image.cuda()

    # Pass the image to model to get the predictions
    output = model_transfer(image)
    _, pred = torch.max(output, 1)

    return class_names[pred]
```

Step 5: Write your Algorithm

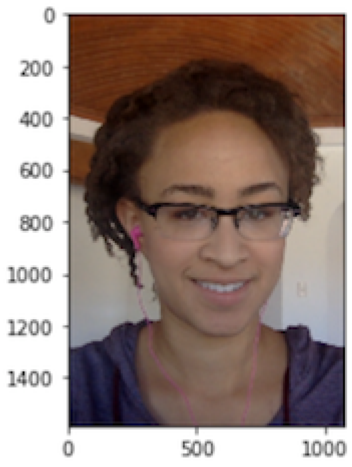
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

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

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `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!

```
hello, human!
```



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

(IMPLEMENTATION) Write your Algorithm

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

    Args:
        img_path: path to an image

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

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

    # Read image
    image = Image.open(img_path)

    # Define transforms
    transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485,0.456,0.406],
                                                         [0.229,0.224,0.225])])
    # Apply transformationto the image
    image = transform(image)
    image = image.unsqueeze(0)

    # Move model to GPU if it is available
    if use_cuda:
      image = image.cuda()

    # Pass image to model to get predictions
    output = model_transfer(image)
    _, pred = torch.max(output, 1)

    # Return predicted class index
    return pred.item()


### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    pred_idx = model_transfer_predict(img_path)
    if pred_idx >=0:
      return True
    else:
      return False


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

```
def run_app(img_path):

    # Read and display image
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    # Handle cases for a human face, dog, and neither
    human_face = face_detector(img_path)
    if human_face > 0:
      print("Hello human")
      dog_breed = predict_breed_transfer(img_path)
      print(f"You look like {dog_breed}")
    else:
      dog_face = dog_detector(img_path)
      if dog_face == True:
        print("Hello dog")
        dog_breed = predict_breed_transfer(img_path)
        print(f"Dog is {dog_breed}")
      else:
        print("Error: Neither human nor dog is 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?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

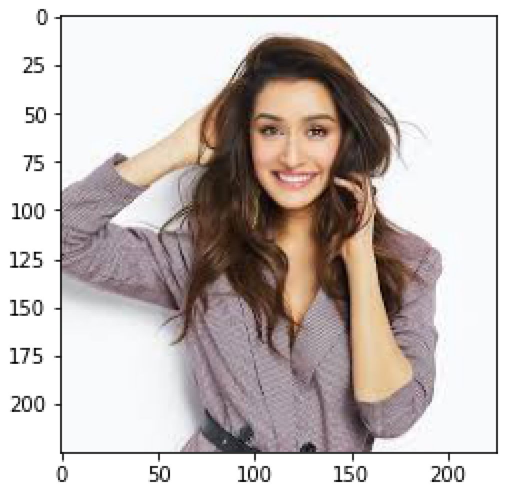
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

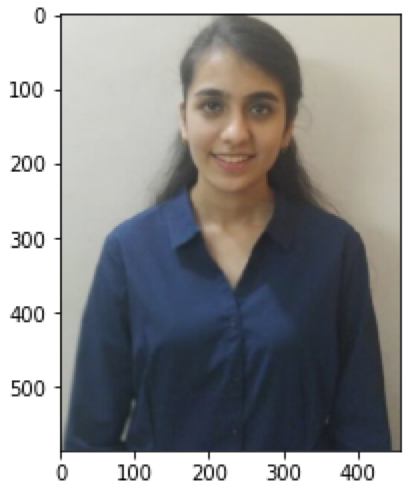
Answer: (Three possible points for improvement)

1. Applying more data augmentations
2. Fine Tuning hyperparameters
3. Choosing other pretrained models

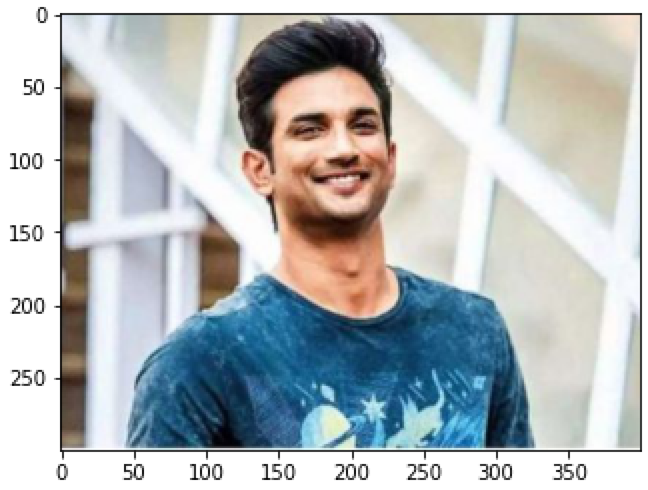
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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



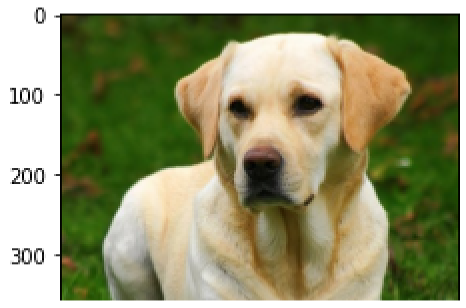
```
Hello human
You look like Bearded collie
```

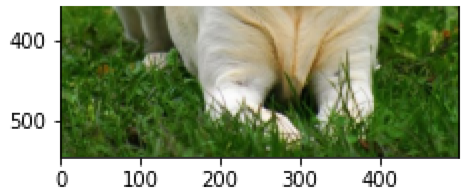


```
Hello human
You look like German wirehaired pointer
```



```
Hello human
You look like Finnish spitz
```





```
Hello dog
Dog is Labrador retriever
```