dog_app

December 28, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location /dog_images.
- Download the human dataset. Unzip the folder and place it in the home directory, at location /lfw.

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

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

```
In [2]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        if len(human_files) == 0:
            human_files = np.array(glob("data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))
        if len(dog_files) == 0:
            dog_files = np.array(glob("data/dog_images/*/*/*"))

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

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

Step 1: Detect Humans

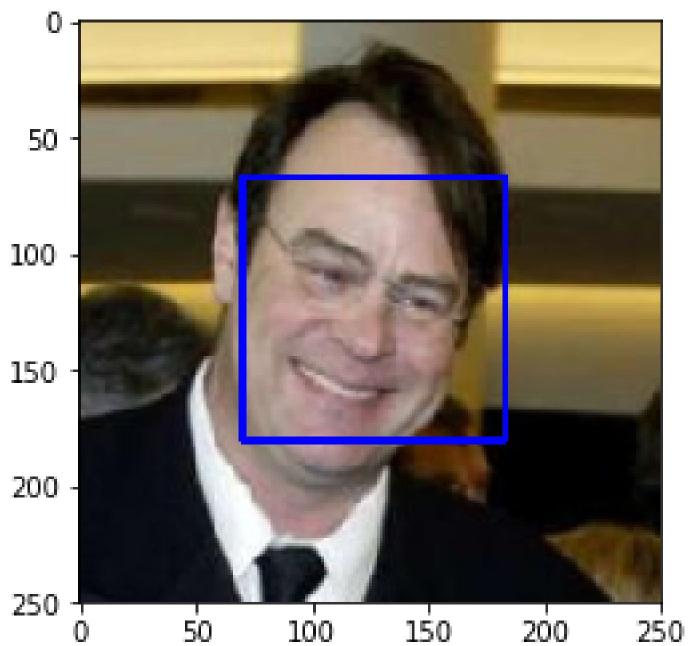
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

```
In [44]: import cv2
         import matplotlib.pyplot as plt
         %matplotlib inline

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

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

         # find faces in image
```

```
faces = face_cascade.detectMultiScale(gray)

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

In [36]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer:

Percentage of detected human faces in human_files_short 99.0%

Percentage of detected human faces in dog_files_short 18.0%

In [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.

human_files_short_human_face_count = 0
dog_files_short_human_face_count = 0

for human_img in human_files_short:
    if face_detector(human_img):
        human_files_short_human_face_count += 1

for dog_img in dog_files_short:
    if face_detector(dog_img):
        dog_files_short_human_face_count += 1

print(f"Percentage of detected human faces in human_files_short "
      f"{human_files_short_human_face_count/len(human_files_short) * 100}% \n"
```

```
          f"Percentage of detected human faces in dog_files_short "
          f"{dog_files_short_human_face_count/len(dog_files_short) * 100}%")
```

```
Percentage of detected human faces in human_files_short 99.0%
Percentage of detected human faces in dog_files_short 18.0%
```

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

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

face_cascade_default = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_defau

# returns "True" if face is detected in image stored at img_path
def face_detector_default(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade_default.detectMultiScale(gray)
    return len(faces) > 0

human_files_short_human_face_count_default = 0
dog_files_short_human_face_count_default = 0

for human_img in human_files_short:
    if face_detector_default(human_img):
        human_files_short_human_face_count_default += 1

for dog_img in dog_files_short:
    if face_detector_default(dog_img):
        dog_files_short_human_face_count_default += 1

print(f"Percentage of detected human faces in human_files_short with haarcascade_frontal
      f"{human_files_short_human_face_count_default/len(human_files_short) * 100}% \n"
      f"Percentage of detected human faces in dog_files_short with haarcascade_frontalfa
      f"{dog_files_short_human_face_count_default/len(dog_files_short) * 100}%")
```

```
Percentage of detected human faces in human_files_short with haarcascade_frontalface_default 100
Percentage of detected human faces in dog_files_short with haarcascade_frontalface_default 51.0%
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [35]: import torch
         import torchvision.models as models

         # define VGG16 model
         VGG16 = models.vgg16(pretrained=True)

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

         # move model to GPU if CUDA is available
         if use_cuda:
             VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:20<00:00, 26545950.13it/s]
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [40]: from PIL import Image, ImageFile
         import torchvision.transforms as transforms
         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
    img_transforms = transforms.Compose([transforms.Resize(255),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.485, 0.456, 0.406],
                                                              [0.229, 0.224, 0.225])])
    img = Image.open(img_path)
    img = img_transforms(img).float()
    img = img.requires_grad_(True)
    img = img.unsqueeze(0)
    if use_cuda:
        img = img.cuda()

    output = VGG16(img)
    if use_cuda:
        output = output.cpu()

    return np.argmax(output.detach().numpy()) # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [33]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             pred_index = VGG16_predict(img_path)
             return pred_index >= 151 and pred_index <= 268 # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:
Percentage of detected dog faces in human_files_short 0.0%
Percentage of detected dog faces in dog_files_short 96.0%

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

        human_files_short_dog_face_count = 0
        dog_files_short_dog_face_count = 0

        for human_img in human_files_short:
            if dog_detector(human_img):
                human_files_short_dog_face_count += 1

        for dog_img in dog_files_short:
            if dog_detector(dog_img):
                dog_files_short_dog_face_count += 1

        print(f"Percentage of detected dog faces in human_files_short "
              f"{human_files_short_dog_face_count/len(human_files_short) * 100}% \n"
              f"Percentage of detected dog faces in dog_files_short "
              f"{dog_files_short_dog_face_count/len(dog_files_short) * 100}%")
```

```
Percentage of detected dog faces in human_files_short 0.0%
Percentage of detected dog faces in dog_files_short 96.0%
```

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

```
In [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.

         ShuffleNet = models.shufflenet_v2_x1_0(pretrained=True)

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

```
In [11]: def shufflenet_predict(img_path):
             '''
             Use pre-trained ShuffleNet 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 ShuffleNet model's prediction
             '''
```

```
        img_transforms = transforms.Compose([transforms.Resize(255),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              transforms.Normalize([0.485, 0.456, 0.406],
                                                                   [0.229, 0.224, 0.225])])
        img = Image.open(img_path)
        img = img_transforms(img).float()
        img = img.requires_grad_(True)
        img = img.unsqueeze(0)

        output = ShuffleNet(img)

        return np.argmax(output.detach().numpy())
```

In [12]:
```
def dog_detector_shuffle(img_path):
    pred_index = shufflenet_predict(img_path)
    return pred_index >= 151 and pred_index <= 268
```

In [13]:
```
human_files_short_dog_face_count_shuffle = 0
dog_files_short_dog_face_count_shuffle = 0

for human_img in human_files_short:
    if dog_detector_shuffle(human_img):
        human_files_short_dog_face_count_shuffle += 1

for dog_img in dog_files_short:
    if dog_detector_shuffle(dog_img):
        dog_files_short_dog_face_count_shuffle += 1

print(f"Percentage of detected dog faces in human_files_short with shufflenet "
      f"{human_files_short_dog_face_count_shuffle/len(human_files_short) * 100}% \n"
      f"Percentage of detected dog faces in dog_files_short with shufflenet "
      f"{dog_files_short_dog_face_count_shuffle/len(dog_files_short) * 100}%")
```

```
Percentage of detected dog faces in human_files_short with shufflenet 14.000000000000002%
Percentage of detected dog faces in dog_files_short with shufflenet 21.0%
```

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [3]: import torch
        import torchvision.models as models
        import torchvision.transforms as transforms
        from PIL import Image, ImageFile

        ImageFile.LOAD_TRUNCATED_IMAGES = True
        use_cuda = torch.cuda.is_available()
```

```
In [4]: import os
        from torchvision import datasets

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

        train_transforms = transforms.Compose([transforms.RandomRotation(60),
                                               transforms.RandomResizedCrop(224),
```

```
                                          transforms.RandomHorizontalFlip(),
                                          transforms.RandomVerticalFlip(),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406],
                                                               [0.229, 0.224, 0.225])])

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

image_path = 'data/dog_images'
if not os.path.isdir(image_path):
    image_path = '/' + image_path

train_path = os.path.join(image_path, 'train')
val_path = os.path.join(image_path, 'valid')
test_path = os.path.join(image_path, 'test')

train_dataset = datasets.ImageFolder(train_path, train_transforms)
val_dataset = datasets.ImageFolder(val_path, train_transforms)
test_dataset = datasets.ImageFolder(test_path, test_transforms)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=True)

loaders_scratch = {'train': train_loader, 'valid': val_loader, 'test': test_loader}
```

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:

I scaled images to a square (255, 255), then cropped them to 224x224px. I picked these sizes because many pretrained models also use these sizes. Then I transformed the images to tensors.

I normalized the train, validation, and test data again following the pattern of many pretrained models. However I augmented the train and validation datasets by randomizing the resize and crop. I also added random horizontal and vertical flipping.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [5]: import torch.nn as nn
        import torch.nn.functional as F
```

```
def calc_w_conv_out(conv, pool_stride = 1):
    return (((conv["W"] - conv["F"] + (2*conv["P"])) / conv["S"]) + 1) / pool_stride

conv1_w_in = 224
conv1 = {"W": conv1_w_in, "D": 3, "K": 16, "F": 3, "P": 1, "S": 1}
conv1_w_out = calc_w_conv_out(conv1)

conv2 = {"W": conv1_w_out, "D": conv1["K"], "K": 24, "F": 3, "P": 1, "S": 1}
conv2_w_out = calc_w_conv_out(conv2, 7)

conv3 = {"W": conv2_w_out, "D": conv2["K"], "K": 32, "F": 3, "P": 1, "S": 1}
conv3_w_out = calc_w_conv_out(conv3)

conv4 = {"W": conv3_w_out, "D": conv3["K"], "K": 48, "F": 3, "P": 1, "S": 1}
conv4_w_out = calc_w_conv_out(conv4, 4)

conv5 = {"W": conv4_w_out, "D": conv4["K"], "K": 56, "F": 3, "P": 1, "S": 1}
conv5_w_out = calc_w_conv_out(conv5)

conv6 = {"W": conv5_w_out, "D": conv5["K"], "K": 64, "F": 3, "P": 1, "S": 1}
conv6_w_out = calc_w_conv_out(conv6, 4)

conv7 = {"W": conv6_w_out, "D": conv6["K"], "K": 176, "F": 3, "P": 1, "S": 1}
conv7_w_out = calc_w_conv_out(conv7)

conv8 = {"W": conv7_w_out, "D": conv7["K"], "K": 192, "F": 3, "P": 1, "S": 1}
conv8_w_out = calc_w_conv_out(conv8, 2)

conv9 = {"W": conv8_w_out, "D": conv8["K"], "K": 208, "F": 3, "P": 1, "S": 1}
conv9_w_out = calc_w_conv_out(conv9)

conv10 = {"W": conv9_w_out, "D": conv9["K"], "K": 224, "F": 3, "P": 1, "S": 1}
conv10_w_out = calc_w_conv_out(conv10, 2)


conv_features_out = conv6_w_out**2 * conv6["K"]

#print(conv1_w_out, conv2_w_out, conv3_w_out, conv4_w_out, conv5_w_out,
#       conv6_w_out, conv7_w_out, conv8_w_out, conv9_w_out, conv10_w_out, conv_features_c

print(conv1_w_out, conv2_w_out, conv3_w_out, conv4_w_out, conv_features_out)

def make_nn_conv(conv):
    return nn.Conv2d(conv["D"], conv["K"], conv["F"], padding=conv["P"], stride=conv["S"


# define the CNN architecture
class Net(nn.Module):
```

```python
### TODO: choose an architecture, and complete the class
def __init__(self):
    super(Net, self).__init__()
    ## Define layers of a CNN
    ## Layer 1
    self.conv1 = make_nn_conv(conv1)
    self.conv2 = make_nn_conv(conv2)
    ## Layer 2
    self.conv3 = make_nn_conv(conv3)
    self.conv4 = make_nn_conv(conv4)
    ## Layer 3
    self.conv5 = make_nn_conv(conv5)
    self.conv6 = make_nn_conv(conv6)
    ## Layer 4
    #self.conv7 = make_nn_conv(conv7)
    #self.conv8 = make_nn_conv(conv8)
    ## Layer 5
    #self.conv9 = make_nn_conv(conv9)
    #self.conv10 = make_nn_conv(conv10)

    ## Layer 6
    self.fc1 = nn.Linear(int(conv_features_out), 133)
    ## Layer 7
    #self.fc2 = nn.Linear(4096, 256)
    ## Layer 8
    #self.fc3 = nn.Linear(256, 133)

def forward(self, x):
    ## Define forward behavior
    batch_size = x.size()[0]

    # layer 1
    x = F.dropout(F.relu(self.conv1(x)), 0.2)
    x = F.dropout(F.max_pool2d(F.relu(self.conv2(x)), 7, 7), 0.2)
    # layer 2
    x = F.dropout(F.relu(self.conv3(x)), 0.2)
    x = F.dropout(F.max_pool2d(F.relu(self.conv4(x)), 4, 4), 0.2)
    # layer 3
    x = F.dropout(F.relu(self.conv5(x)), 0.2)
    x = F.dropout(F.max_pool2d(F.relu(self.conv6(x)), 4, 4), 0.2)
    # layer 4
    #x = F.dropout(F.relu(self.conv7(x)), 0.2)
    #x = F.dropout(F.max_pool2d(F.relu(self.conv8(x)), 2, 2), 0.2)
    # layer 5
    #x = F.dropout(F.relu(self.conv9(x)), 0.2)
    #x = F.dropout(F.max_pool2d(F.relu(self.conv10(x)), 2, 2), 0.2)

    x = x.view(batch_size, -1)
```

```
            #x = F.dropout(F.relu(self.fc1(x)), 0.2)
            #x = F.dropout(F.relu(self.fc2(x)), 0.2)
            #x = F.log_softmax(self.fc3(x))
            x = self.fc1(x)

            return x

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

    # instantiate the CNN
    model_scratch = Net()

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

```
224.0 32.0 32.0 8.0 256.0
```

In [11]:
```
model_scratch
```

Out[11]:
```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(24, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(32, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(48, 56, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv6): Conv2d(56, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=256, out_features=133, bias=True)
)
```

In [12]:
```
#!nvidia-smi
#!sudo fuser -v /dev/nvidia*
#!sudo kill -9 85
```

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

Answer:

My architecture is a take on the VGG architecture. Each of the convolutional layers consists of 2 conv steps and a max pool step. The classification layer set consists of 1 fully connected layer. I tried to maximize depth without exhausting my resources. I went through many iterations (commented above), for instance I have tested batch sizes from 4 to 128, conv layers of 1 to 5, and fully connected layers from 1 to 3. In the end I settled on 1 fc layer, and 3 conv layers. I also tested different methods of spatial dimentionality reduction, larger and smaller pool and conv strides for instance. The above model and batch size demonstrated good performance while not exhausting my GPU memory.

As I step through each feature extraction layer, I reduce the spatial dimentionality and increase the depth. Since we have many complex features to learn, I opted for a deeper network.

The classification layers simply reduces the features output by the feature extraction layers into 133 prediction outputs. Much of the detail is output above.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [6]: import torch.optim as optim

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

        ### TODO: select optimizer
        optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01, momentum=0.9)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [12]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

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

                     optimizer.zero_grad()

                     output = model(data)

                     loss = criterion(output, target)
                     loss.backward()
```

```
                optimizer.step()

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

            train_loss = train_loss/len(loaders['train'].sampler)

            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                output = model(data)
                loss = criterion(output, target)
                valid_loss += loss.item()*data.size(0)

            valid_loss = valid_loss/len(loaders['valid'].sampler)

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

            ## TODO: save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                print(f'Saved model, validation decresed: {valid_loss_min} => {valid_loss}'
                valid_loss_min = valid_loss
                torch.save(model.state_dict(), save_path)

        # return trained model
        return model
```

```
In [14]: # train the model
         model_scratch = train(50, 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: 4.886349         Validation Loss: 4.874956
Saved model_scratch.pt, validation decresed: inf => 4.874955803762653
Epoch: 2        Training Loss: 4.871458         Validation Loss: 4.856403
```

```
Saved model_scratch.pt, validation decresed: 4.874955803762653 => 4.856402617585873
Epoch: 3         Training Loss: 4.844694         Validation Loss: 4.824011
Saved model_scratch.pt, validation decresed: 4.856402617585873 => 4.824011480023048
Epoch: 4         Training Loss: 4.817443         Validation Loss: 4.797559
Saved model_scratch.pt, validation decresed: 4.824011480023048 => 4.797558997491163
Epoch: 5         Training Loss: 4.774604         Validation Loss: 4.759784
Saved model_scratch.pt, validation decresed: 4.797558997491163 => 4.7597839144175635
Epoch: 6         Training Loss: 4.745781         Validation Loss: 4.738268
Saved model_scratch.pt, validation decresed: 4.7597839144175635 => 4.738268052175373
Epoch: 7         Training Loss: 4.716604         Validation Loss: 4.710740
Saved model_scratch.pt, validation decresed: 4.738268052175373 => 4.7107401162564395
Epoch: 8         Training Loss: 4.676164         Validation Loss: 4.681243
Saved model_scratch.pt, validation decresed: 4.7107401162564395 => 4.681243339127409
Epoch: 9         Training Loss: 4.617986         Validation Loss: 4.667093
Saved model_scratch.pt, validation decresed: 4.681243339127409 => 4.667092974885495
Epoch: 10        Training Loss: 4.573935         Validation Loss: 4.585720
Saved model_scratch.pt, validation decresed: 4.667092974885495 => 4.585719639098573
Epoch: 11        Training Loss: 4.531621         Validation Loss: 4.488469
Saved model_scratch.pt, validation decresed: 4.585719639098573 => 4.488468882851972
Epoch: 12        Training Loss: 4.491057         Validation Loss: 4.527221
Epoch: 13        Training Loss: 4.433595         Validation Loss: 4.532133
Epoch: 14        Training Loss: 4.378784         Validation Loss: 4.420561
Saved model_scratch.pt, validation decresed: 4.488468882851972 => 4.4205613890093955
Epoch: 15        Training Loss: 4.353498         Validation Loss: 4.308159
Saved model_scratch.pt, validation decresed: 4.4205613890093955 => 4.3081587277486655
Epoch: 16        Training Loss: 4.313212         Validation Loss: 4.312609
Epoch: 17        Training Loss: 4.259010         Validation Loss: 4.365662
Epoch: 18        Training Loss: 4.238743         Validation Loss: 4.269065
Saved model_scratch.pt, validation decresed: 4.3081587277486655 => 4.269064992916085
Epoch: 19        Training Loss: 4.183247         Validation Loss: 4.250778
Saved model_scratch.pt, validation decresed: 4.269064992916085 => 4.250778046054041
Epoch: 20        Training Loss: 4.166183         Validation Loss: 4.319319
Epoch: 21        Training Loss: 4.127019         Validation Loss: 4.151792
Saved model_scratch.pt, validation decresed: 4.250778046054041 => 4.15179211450908
Epoch: 22        Training Loss: 4.082067         Validation Loss: 4.211116
Epoch: 23        Training Loss: 4.042719         Validation Loss: 4.105904
Saved model_scratch.pt, validation decresed: 4.15179211450908 => 4.105903582087534
Epoch: 24        Training Loss: 4.034509         Validation Loss: 4.173188
Epoch: 25        Training Loss: 4.022800         Validation Loss: 4.071308
Saved model_scratch.pt, validation decresed: 4.105903582087534 => 4.07130839125125
Epoch: 26        Training Loss: 3.964795         Validation Loss: 4.074127
Epoch: 27        Training Loss: 3.919819         Validation Loss: 3.950540
Saved model_scratch.pt, validation decresed: 4.07130839125125 => 3.9505399475554506
Epoch: 28        Training Loss: 3.889702         Validation Loss: 3.909950
Saved model_scratch.pt, validation decresed: 3.9505399475554506 => 3.909949774370936
Epoch: 29        Training Loss: 3.870314         Validation Loss: 3.959555
Epoch: 30        Training Loss: 3.859976         Validation Loss: 3.932009
Epoch: 31        Training Loss: 3.813756         Validation Loss: 3.988577
```

```
Epoch: 32 	Training Loss: 3.795176 	Validation Loss: 4.013737
Epoch: 33 	Training Loss: 3.770210 	Validation Loss: 3.841450
Saved model_scratch.pt, validation decresed: 3.909949774370936 => 3.8414497872312627
Epoch: 34 	Training Loss: 3.755823 	Validation Loss: 4.000259
Epoch: 35 	Training Loss: 3.718831 	Validation Loss: 4.004603
Epoch: 36 	Training Loss: 3.703429 	Validation Loss: 3.893162
Epoch: 37 	Training Loss: 3.712983 	Validation Loss: 3.844040
Epoch: 38 	Training Loss: 3.675443 	Validation Loss: 3.784656
Saved model_scratch.pt, validation decresed: 3.8414497872312627 => 3.7846560572435757
Epoch: 39 	Training Loss: 3.646301 	Validation Loss: 3.848840
Epoch: 40 	Training Loss: 3.646254 	Validation Loss: 3.844699
Epoch: 41 	Training Loss: 3.611488 	Validation Loss: 3.800444
Epoch: 42 	Training Loss: 3.611060 	Validation Loss: 3.873859
Epoch: 43 	Training Loss: 3.623455 	Validation Loss: 3.868454
Epoch: 44 	Training Loss: 3.579911 	Validation Loss: 3.828724
Epoch: 45 	Training Loss: 3.613435 	Validation Loss: 3.867723
Epoch: 46 	Training Loss: 3.603369 	Validation Loss: 3.780542
Saved model_scratch.pt, validation decresed: 3.7846560572435757 => 3.780542138665022
Epoch: 47 	Training Loss: 3.597731 	Validation Loss: 3.822047
Epoch: 48 	Training Loss: 3.554749 	Validation Loss: 3.780447
Saved model_scratch.pt, validation decresed: 3.780542138665022 => 3.780446781512506
Epoch: 49 	Training Loss: 3.529535 	Validation Loss: 3.700819
Saved model_scratch.pt, validation decresed: 3.780446781512506 => 3.700818885586219
Epoch: 50 	Training Loss: 3.586106 	Validation Loss: 3.778319
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [18]: def test(loaders, model, criterion, use_cuda):

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

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders['test']):
                 # move to GPU
                 if use_cuda:
                     data, target = data.cuda(), target.cuda()
                 # forward pass: compute predicted outputs by passing inputs to the model
                 output = model(data)
                 # calculate the loss
                 loss = criterion(output, target)
                 # update average test loss
```

```
            test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
            # convert output probabilities to predicted class
            pred = output.data.max(1, keepdim=True)[1]
            # compare predictions to true label
            correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
            total += data.size(0)

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

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

In [15]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))

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

```
Test Loss: 3.344630


Test Accuracy: 18% (158/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

In [7]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

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

1.1.13 (IMPLEMENTATION) Model Architecture

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

In [8]:
```
import torchvision.models as models
import torch.nn as nn
```

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

for param in model_transfer.features.parameters():
    param.requires_grad_(False)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.torch/models/vgg
100%|| 574673361/574673361 [00:07<00:00, 77889513.18it/s]
```

```
Out[8]: VGG(
          (features): Sequential(
            (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): ReLU(inplace)
            (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (3): ReLU(inplace)
            (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (6): ReLU(inplace)
            (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (8): ReLU(inplace)
            (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (11): ReLU(inplace)
            (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (13): ReLU(inplace)
            (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (15): ReLU(inplace)
            (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (17): ReLU(inplace)
            (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)
            (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (22): ReLU(inplace)
            (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (24): ReLU(inplace)
            (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (26): ReLU(inplace)
            (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (29): ReLU(inplace)
            (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (31): ReLU(inplace)
            (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (33): ReLU(inplace)
            (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
      (35): ReLU(inplace)
      (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace)
      (2): Dropout(p=0.5)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace)
      (5): Dropout(p=0.5)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
```

In [9]:
```
in_features = model_transfer.classifier[6].in_features

output_fc_layer = nn.Linear(in_features, 133)

model_transfer.classifier[6] = output_fc_layer

if use_cuda:
    model_transfer = model_transfer.cuda()

model_transfer
```

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

```
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace)
      (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (24): ReLU(inplace)
      (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (26): ReLU(inplace)
      (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace)
      (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (31): ReLU(inplace)
      (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (33): ReLU(inplace)
      (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (35): ReLU(inplace)
      (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace)
      (2): Dropout(p=0.5)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace)
      (5): Dropout(p=0.5)
      (6): Linear(in_features=4096, out_features=133, bias=True)
    )
  )
```

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

Answer:

Similar to the reasoning in my scratch model, I opted for a deeper network to maximize feature extraction. My only addition was the custom output layer to respect the number of outputs required to solve our problem.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [22]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001, moment
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [23]: # train the model
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))

         model_transfer = train(50, loaders_transfer, model_transfer, optimizer_transfer,
                                criterion_transfer, use_cuda, 'model_transfer.pt')

         # 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: 2.440045         Validation Loss: 1.933388
Saved model, validation decresed: inf => 1.9333875885980571
Epoch: 2         Training Loss: 2.101694         Validation Loss: 1.831920
Saved model, validation decresed: 1.9333875885980571 => 1.831920090692486
Epoch: 3         Training Loss: 1.994816         Validation Loss: 1.812513
Saved model, validation decresed: 1.831920090692486 => 1.812513408546676
Epoch: 4         Training Loss: 1.916495         Validation Loss: 1.672239
Saved model, validation decresed: 1.812513408546676 => 1.672239061886679
Epoch: 5         Training Loss: 1.848945         Validation Loss: 1.646405
Saved model, validation decresed: 1.672239061886679 => 1.6464046883012007
Epoch: 6         Training Loss: 1.804890         Validation Loss: 1.731276
Epoch: 7         Training Loss: 1.772146         Validation Loss: 1.678199
Epoch: 8         Training Loss: 1.742338         Validation Loss: 1.593539
Saved model, validation decresed: 1.6464046883012007 => 1.5935393957320803
Epoch: 9         Training Loss: 1.687611         Validation Loss: 1.603921
Epoch: 10         Training Loss: 1.654356         Validation Loss: 1.626794
Epoch: 11         Training Loss: 1.629365         Validation Loss: 1.550525
Saved model, validation decresed: 1.5935393957320803 => 1.5505245928992768
Epoch: 12         Training Loss: 1.598305         Validation Loss: 1.564545
Epoch: 13         Training Loss: 1.586382         Validation Loss: 1.505549
Saved model, validation decresed: 1.5505245928992768 => 1.5055486672652696
Epoch: 14         Training Loss: 1.561946         Validation Loss: 1.543798
Epoch: 15         Training Loss: 1.578767         Validation Loss: 1.535505
Epoch: 16         Training Loss: 1.515793         Validation Loss: 1.462840
Saved model, validation decresed: 1.5055486672652696 => 1.4628396194138213
Epoch: 17         Training Loss: 1.553354         Validation Loss: 1.505737
Epoch: 18         Training Loss: 1.529435         Validation Loss: 1.459117
Saved model, validation decresed: 1.4628396194138213 => 1.4591167906801144
Epoch: 19         Training Loss: 1.480919         Validation Loss: 1.586827
Epoch: 20         Training Loss: 1.464608         Validation Loss: 1.445554
Saved model, validation decresed: 1.4591167906801144 => 1.4455540545686276
Epoch: 21         Training Loss: 1.468181         Validation Loss: 1.550692
Epoch: 22         Training Loss: 1.462875         Validation Loss: 1.575371
Epoch: 23         Training Loss: 1.440352         Validation Loss: 1.436228
Saved model, validation decresed: 1.4455540545686276 => 1.4362275693231
Epoch: 24         Training Loss: 1.413724         Validation Loss: 1.554304
Epoch: 25         Training Loss: 1.411300         Validation Loss: 1.442014
Epoch: 26         Training Loss: 1.408343         Validation Loss: 1.504286
Epoch: 27         Training Loss: 1.391556         Validation Loss: 1.453332
```

```
Epoch: 28         Training Loss: 1.371108        Validation Loss: 1.390759
Saved model, validation decresed: 1.4362275693231 => 1.3907590637306968
Epoch: 29         Training Loss: 1.374468        Validation Loss: 1.491788
Epoch: 30         Training Loss: 1.348547        Validation Loss: 1.452440
Epoch: 31         Training Loss: 1.352558        Validation Loss: 1.452513
Epoch: 32         Training Loss: 1.333859        Validation Loss: 1.461971
Epoch: 33         Training Loss: 1.334603        Validation Loss: 1.488666
Epoch: 34         Training Loss: 1.328539        Validation Loss: 1.442902
Epoch: 35         Training Loss: 1.325614        Validation Loss: 1.461992
Epoch: 36         Training Loss: 1.302224        Validation Loss: 1.376617
Saved model, validation decresed: 1.3907590637306968 => 1.3766168343092866
Epoch: 37         Training Loss: 1.317673        Validation Loss: 1.488859
Epoch: 38         Training Loss: 1.273219        Validation Loss: 1.398614
Epoch: 39         Training Loss: 1.290325        Validation Loss: 1.392804
Epoch: 40         Training Loss: 1.280754        Validation Loss: 1.366158
Saved model, validation decresed: 1.3766168343092866 => 1.366158004292471
Epoch: 41         Training Loss: 1.227731        Validation Loss: 1.350499
Saved model, validation decresed: 1.366158004292471 => 1.3504994914917174
Epoch: 42         Training Loss: 1.238393        Validation Loss: 1.453760
Epoch: 43         Training Loss: 1.257587        Validation Loss: 1.417194
Epoch: 44         Training Loss: 1.258255        Validation Loss: 1.449541
Epoch: 45         Training Loss: 1.252617        Validation Loss: 1.459572
Epoch: 46         Training Loss: 1.222234        Validation Loss: 1.436493
Epoch: 47         Training Loss: 1.225767        Validation Loss: 1.543634
Epoch: 48         Training Loss: 1.194729        Validation Loss: 1.399152
Epoch: 49         Training Loss: 1.224239        Validation Loss: 1.392043
Epoch: 50         Training Loss: 1.252902        Validation Loss: 1.415003
```

1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [24]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.503427


Test Accuracy: 85% (711/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [49]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))

         # 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 test_dataset.classes]

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

             img = Image.open(img_path)
             img = test_transforms(img).float()
             img = img.requires_grad_(True)
             img = img.unsqueeze(0)
             if use_cuda:
                 img = img.cuda()

             output = model_transfer(img)
             if use_cuda:
                 output = output.cpu()

             return class_names[np.argmax(output.detach().numpy())]
```

Step 5: Write your Algorithm

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

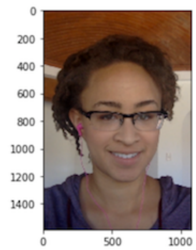
You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [70]: ### 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
             found_dog = dog_detector(img_path)
             found_human = face_detector(img_path)
             img = cv2.imread(img_path)
             cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

hello, human!



You look like a ...
Chinese_shar-pei

Sample Human Output

```
plt.imshow(cv_rgb)
if not found_dog and not found_human:
    print("Hmmm, hold on a minute...")
    plt.show()
    print("I don't know what I'm looking at...! Can you try again?\n\n")
else:
    pred_breed = predict_breed_transfer(img_path)
    print(f"Hello, {'dog' if found_dog else 'human'}!")
    plt.show()
    print(f"You {'are' if found_dog else 'look like'} a...")
    print(f"{pred_breed}\n")
```

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)

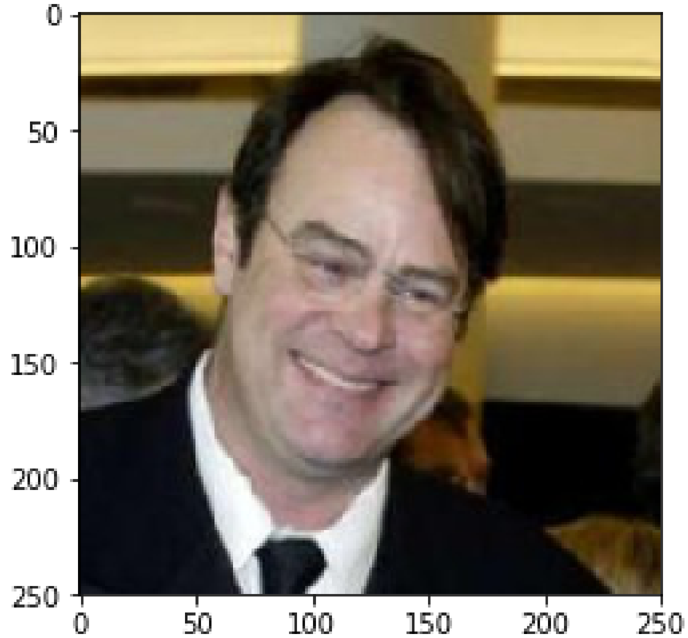
The model performed better than I expected. There are however points where it could be improved:

- The algorithm only supports single subject predictions, I would have liked to predict more than just one subject per image. For instance, it could be nice to draw a square around each human/dog face in the image and plot the appropriate prediction as a label.
- The algorithm is too dull for human images, it would be better to provide a reference image for human predictions, i.e "You look like THIS Pharoah hound!" It would be even better if the selected refrence image was one that closly matched the weights of the human image.

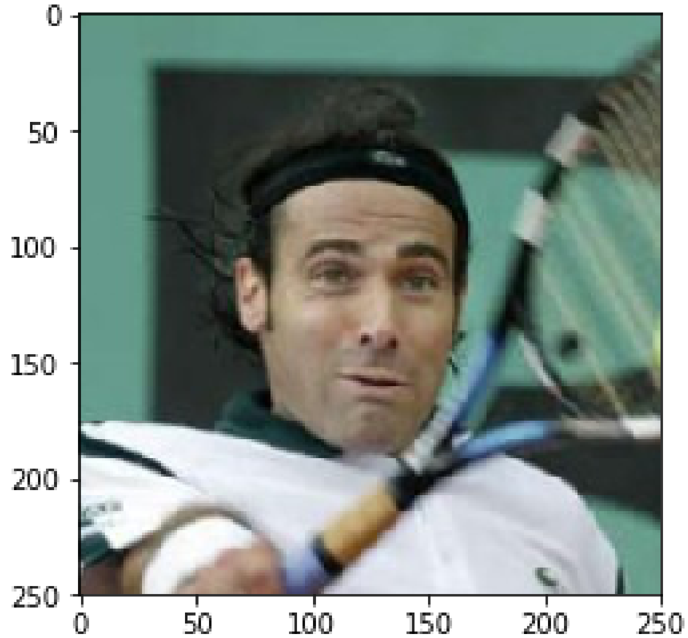
- The algorithm is limited to dogs, I would prefer an algorithm that could match many species of animals.

In [66]:
```
## 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)
```

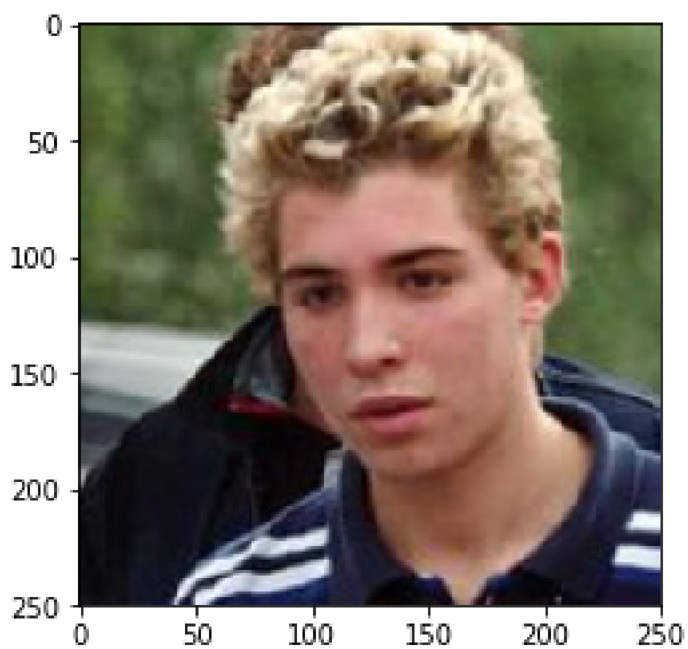
```
Hello, human!
```



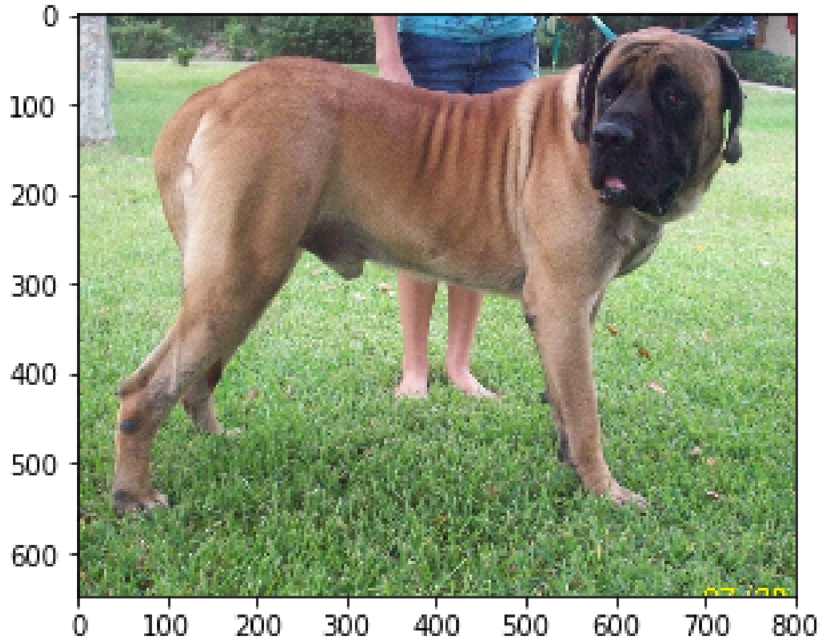
```
You look like a...
Pharaoh hound

Hello, human!
```



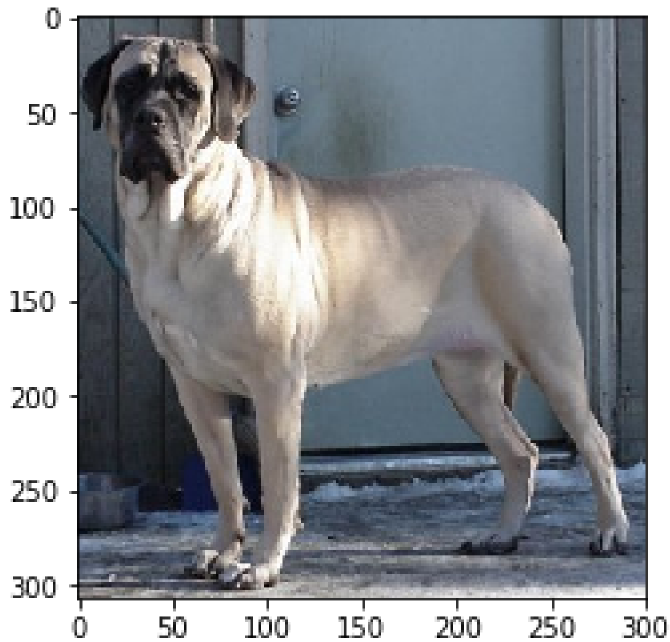

```
You look like a...
Dogue de bordeaux

Hello, human!
```



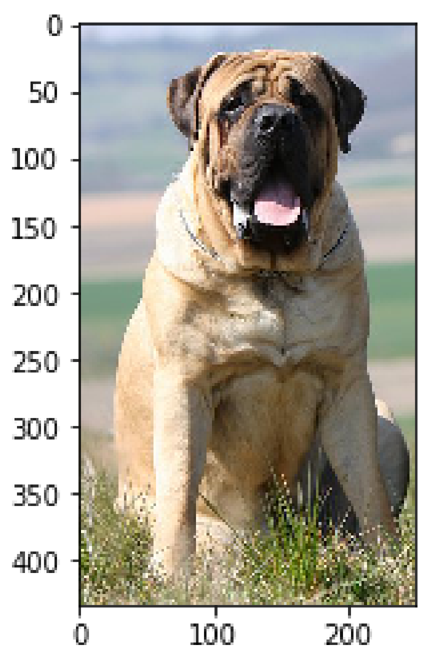
```
You look like a...
Dogue de bordeaux

Hello, dog!
```



```
You are a...
Mastiff

Hello, dog!
```



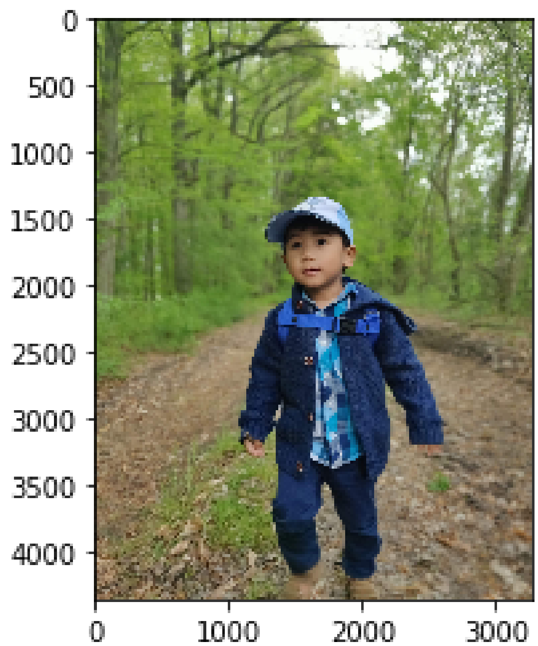
```
You are a...
Mastiff

Hello, dog!
```



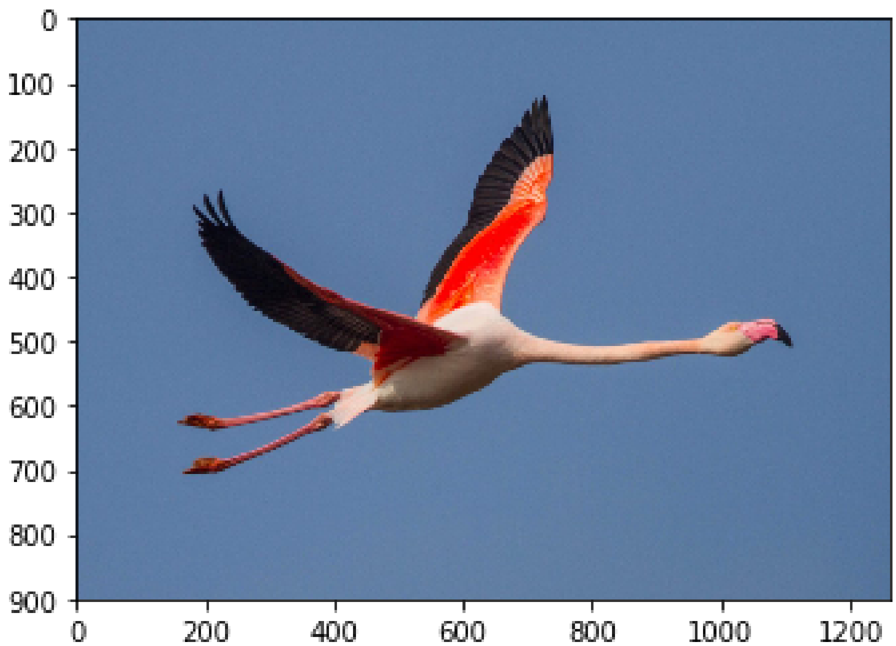
```
You are a...
Mastiff
```

```
In [71]: custom_files = np.array(glob("custom_images/*"))

         for file in custom_files:
             run_app(file)
```

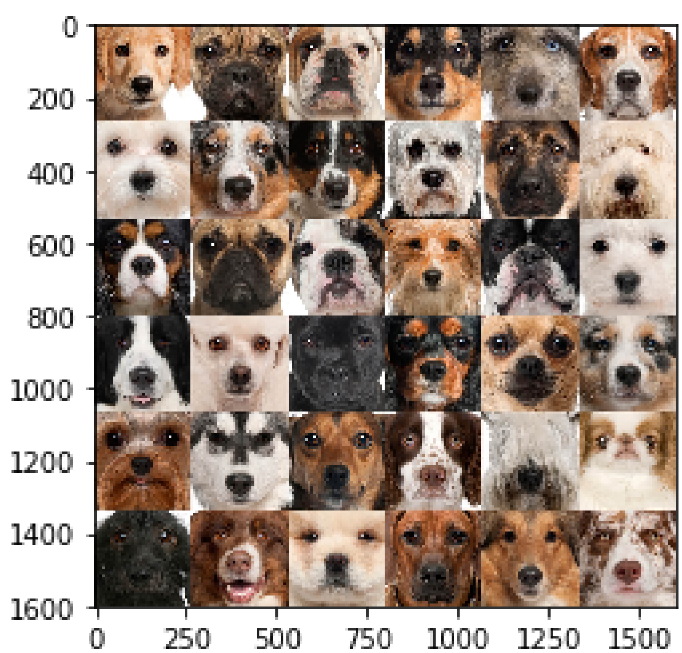
```
Hello, human!
```



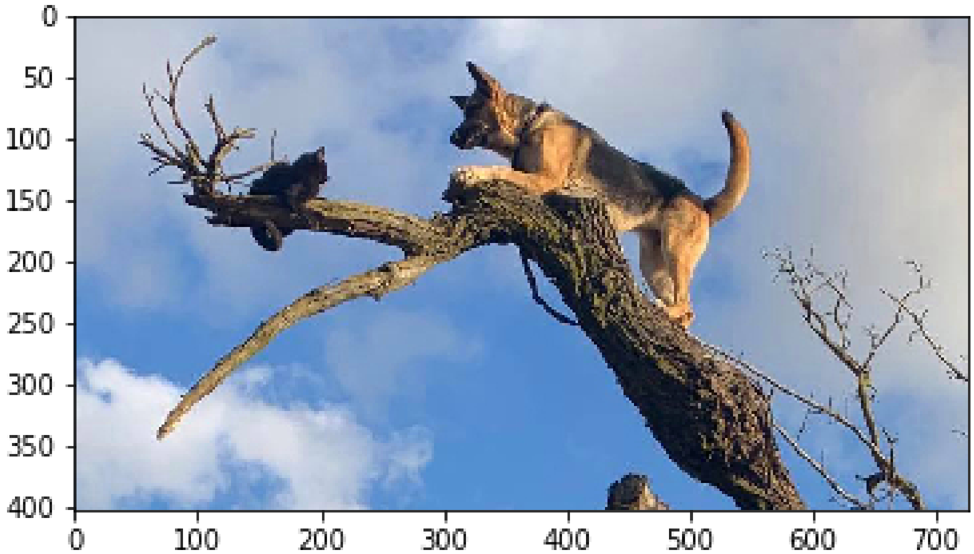

```
You look like a...
Labrador retriever

Hmmm, hold on a minute...
```



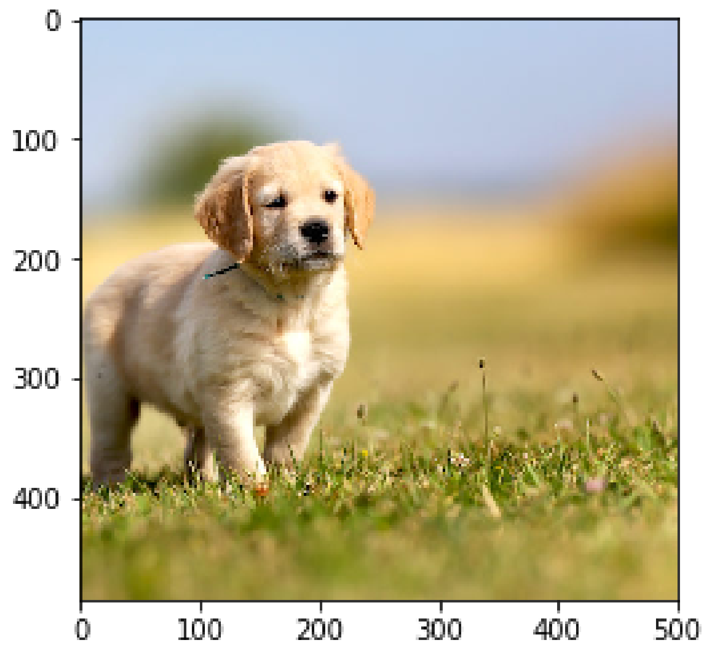

```
I don't know what I'm looking at...! Can you try again?


Hello, human!
```



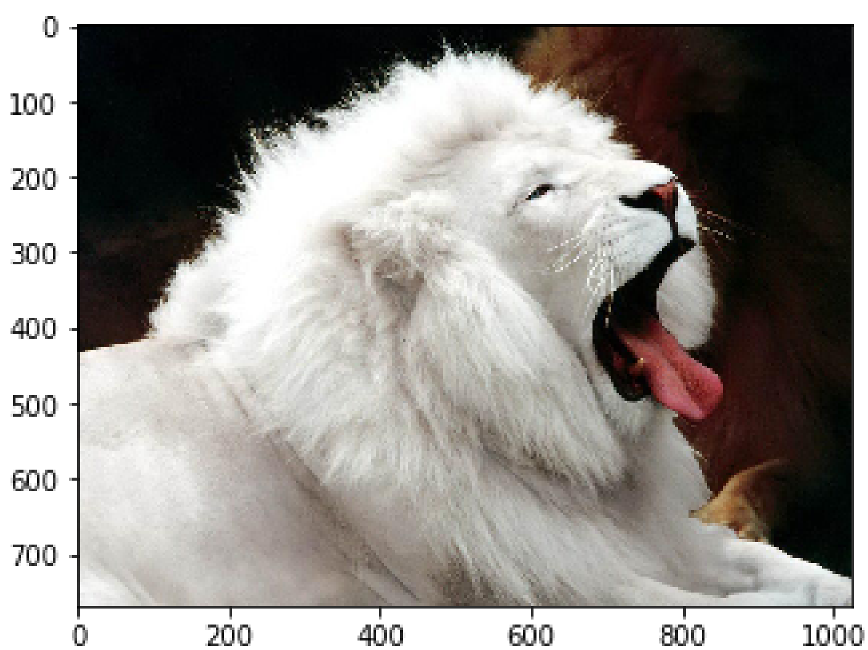
```
You look like a...
Smooth fox terrier

Hello, dog!
```




```
You are a...
German shepherd dog

Hello, dog!
```




```
You are a...
Golden retriever

Hello, dog!
```



```
You are a...
American eskimo dog
```

```
In [ ]:
```