dog_app

August 19, 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: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

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

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

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

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

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

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

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

1.2 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.

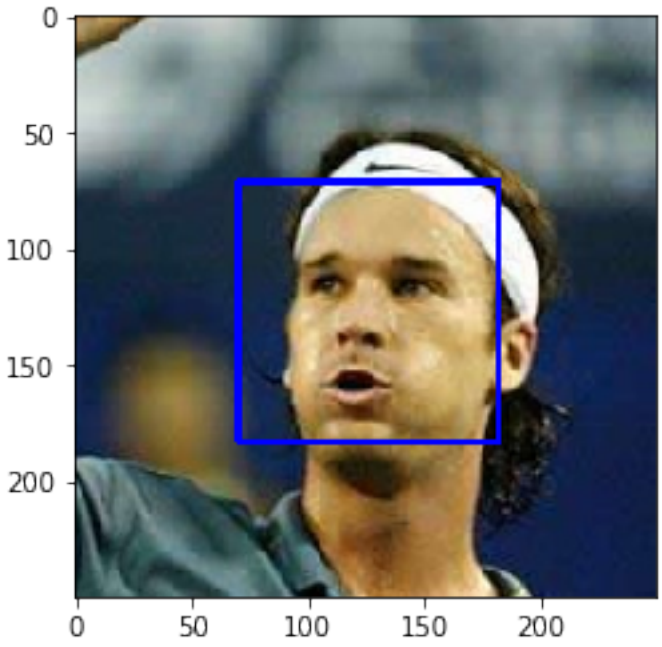
[8]:
```
! ls haarcascades/
```

```
haarcascade_frontalface_alt.xml
```

[59]:
```
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
```



[38]:
```
print(faces)
```

```
[[ 70  71 112 112]]
```

[37]:
```
# viewing the raw img converted to greyscale
```

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.

1.2.1 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.

[60]:
```
# 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
```

1.2.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell)

- Percentage True Positive: 1.0
- Percentage False Positive: 0.12 (Dogs with "human faces")

See analysis below

[5]:
```
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_perc = sum(face_detector(face) for face in human_files_short)/100
dog_perc = sum(face_detector(face) for face in dog_files_short)/100

print("Percentage True Positive:", "\t", human_perc)
print("Percentage False Positive:", "\t", dog_perc)
```

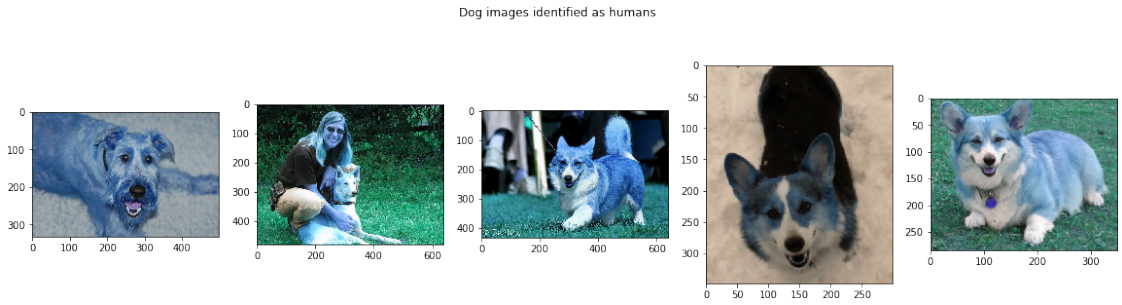
```
Percentage True Positive:       1.0
Percentage False Positive:      0.12
```

[62]:
```
# 12% of dogs were false classified as humans.
# Out of curiosity, lets see which dogs are seen as humans by the OpenCV alg

fp = []

for face in tqdm(dog_files_short):
    if face_detector(face):
        fp.append(face)
```

```
100%|| 100/100 [00:19<00:00,  5.09it/s]
```

[63]:
```
fig, axs = plt.subplots(1,5, figsize=(20,5))
fig.suptitle("Dog images identified as humans")

for i, path in enumerate(fp[:5]):
    axs[i].imshow(cv2.imread(path))

print("Apparently, humans might also be present in some dog pictures.")
```

```
Apparently, humans might also be present in some dog pictures.
```



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

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

Step 2: Detect Dogs

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

1.2.3 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.

[64]:
```
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()
```

[65]:
```
# Freeze model layers

for param in VGG16.parameters():
    param.requires_grad = False
```

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.

1.2.4 (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.

[68]:
```
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

    # Step 0. Load file
    img_raw = Image.open(img_path)
    img_np = np.asarray(img_raw).transpose(2,0,1) # reordering dimensions to␣
↪[channel, h, w]
    t = torch.tensor(img_np, dtype=torch.float32) # convert to torch tensor

    """ Step 1. Transform PIL image using following manipulations:
        - Resize to 256x256
        - Crop at center to 224x224
        - Normalize using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224,␣
↪0.225]
    """
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        )])

    t = transform(img_raw)

    t = t.unsqueeze(0) # setting tensor te required dimensions [batch, channel,␣
↪h, w]

    # Step 2. Obtaining predicted class
    pred = torch.argmax(VGG16(t))


    return pred
```

1.2.5 (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).

[66]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    idx = VGG16_predict(img_path)

    if idx in range(151, 269):
        return True
    else:
        return False
```

1.2.6 (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:

- 96% detected as dog (4% False Negatives)
- 1% falsely detected as dog on human images (False Positive)

See analysis below

[70]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

d = 0 # dog imgs counter
d_fn = [] # dog imgs false negatives

h = 0 # human imgs counter
h_fp = [] # dog false positives

for path in tqdm(dog_files_short, position=0):
    pred = dog_detector(path)

    if pred:
        d += 1
    else:
        d_fn.append(path)

for path in tqdm(human_files_short, position=0):
    pred = dog_detector(path)

    if pred:
        h_fp.append(path)
```

```
    else:
        h += 1

print("TP: ", d/100, "\nFP: ", h/100)
```

```
100%|| 100/100 [01:09<00:00,  1.44it/s]

TP:  0.96
FP:  0.99
```

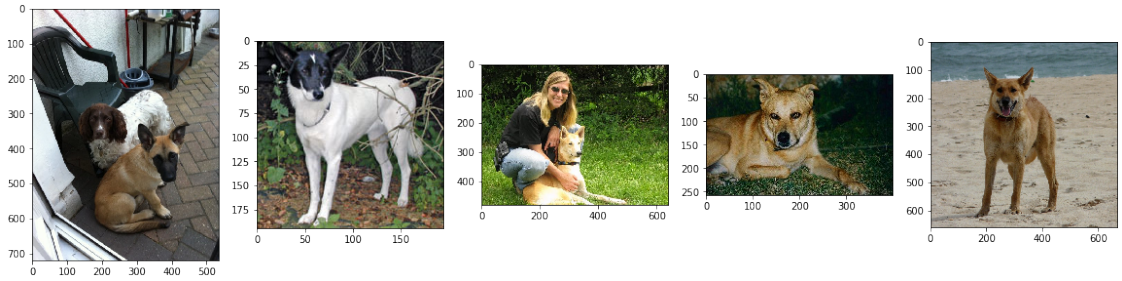
[71]:
```
# Lets check the missclassified images

# Starting with the FP on human ds
Image.open(h_fp[0])

# Clearly a mistake
```

[71]:

[165]:
```
# The FNs

fig, axs = plt.subplots(1,5, figsize=(20,5))

for i, img in enumerate(d_fn):
    axs[i].imshow(np.asarray(Image.open(img, 'r')))
    idx = VGG16_predict(img).item()
```

```
    print(i, ":\t", idx, " - ", img_dict[idx])
```

```
0 :      205  -  flat-coated retriever
1 :      158  -  toy terrier
2 :      273  -  dingo, warrigal, warragal, Canis dingo
3 :      273  -  dingo, warrigal, warragal, Canis dingo
4 :      273  -  dingo, warrigal, warragal, Canis dingo
```



Seems like VGG16 got the first two images correctly from the second try, and other dogs were classified as dingo, which is basically from the same animal family.

[ ]:

[158]:

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.
```

[196]:
```
# Load models

resnet152 = models.resnet152(pretrained=True)
squeezenet =  models.squeezenet1_1(pretrained=True)

for param in resnet152.parameters():
    param.requires_grad = False

for param in squeezenet.parameters():
    param.requires_grad = False
```

```
Downloading: "https://download.pytorch.org/models/squeezenet1_1-f364aa15.pth" to
/home/le-roy/.cache/torch/hub/checkpoints/squeezenet1_1-f364aa15.pth

HBox(children=(IntProgress(value=0, max=4966400), HTML(value='')))
```

[194]:
```
def predict(img_path, model):
    '''
    Use pre-trained model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image
        model:    pre-trained pytorch model instance

    Returns:
        Index corresponding to model's prediction
    '''

    # Step 0. Load file
    img_raw = Image.open(img_path)
    img_np = np.asarray(img_raw).transpose(2,0,1) # reordering dimensions to␣
↪[channel, h, w]
    t = torch.tensor(img_np, dtype=torch.float32) # convert to torch tensor

    """ Step 1. Transform PIL image using following manipulations:
        - Resize to 256x256
        - Crop at center to 224x224
        - Normalize using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224,␣
↪0.225]
    """
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        )])

    t = transform(img_raw)

    t = t.unsqueeze(0) # setting tensor te required dimensions [batch, channel,␣
↪h, w]

    # Step 2. Obtaining predicted class
    pred = torch.argmax(model(t))


    return pred
```

```
def dog_detector(img_path, model):
    idx = predict(img_path, model)

    if idx in range(151, 269):
        return True
    else:
        return False
```

[195]:
```
# ResNet152

h = np.sum(dog_detector(img, resnet152) for img in tqdm(human_files_short,␣
 ↪position=0, leave=True))
s = np.sum(dog_detector(img, resnet152) for img in tqdm(dog_files_short,␣
 ↪position=0, leave=True))

print("TP: ", d/100, "\nFP: ", h/100)
```

```
  0%|          | 0/100 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-
packages/ipykernel_launcher.py:1: DeprecationWarning: Calling np.sum(generator)
is deprecated, and in the future will give a different result. Use
np.sum(np.fromiter(generator)) or the python sum builtin instead.
  """Entry point for launching an IPython kernel.
100%|| 100/100 [01:39<00:00,  1.02it/s]



100%|| 100/100 [01:39<00:00,  1.00it/s]
  0%|          | 0/100 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-
packages/ipykernel_launcher.py:2: DeprecationWarning: Calling np.sum(generator)
is deprecated, and in the future will give a different result. Use
np.sum(np.fromiter(generator)) or the python sum builtin instead.

100%|| 100/100 [01:58<00:00,  1.19s/it]



TP:  0.95
FP:  0.0
```

[197]:
```
# squeezenet1_1

h = np.sum(dog_detector(img, squeezenet) for img in tqdm(human_files_short,␣
 ↪position=0, leave=True))
```

```
s = np.sum(dog_detector(img, squeezenet) for img in tqdm(dog_files_short,␣
 ↪position=0, leave=True))

print("TP: ", d/100, "\nFP: ", h/100)
```

```
  0%|          | 0/100 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-
packages/ipykernel_launcher.py:3: DeprecationWarning: Calling np.sum(generator)
is deprecated, and in the future will give a different result. Use
np.sum(np.fromiter(generator)) or the python sum builtin instead.
  This is separate from the ipykernel package so we can avoid doing imports
until
100%|| 100/100 [00:05<00:00, 15.68it/s]
100%|| 100/100 [00:05<00:00, 17.91it/s]
  0%|          | 0/100 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-
packages/ipykernel_launcher.py:4: DeprecationWarning: Calling np.sum(generator)
is deprecated, and in the future will give a different result. Use
np.sum(np.fromiter(generator)) or the python sum builtin instead.
  after removing the cwd from sys.path.

TP:  0.95
FP:  0.02
```

Since any significant performance improvements are observed, we will stick with the initial VGG16 model

---

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!

1.2.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

[7]:
```
import os
import torch
import torchvision

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision.transforms as transforms



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

transform = {

    "train" : transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomAffine(degrees=0.2),
    transforms.ToTensor(),
```

```
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )]),

    "test" : transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )])}

train_img = torchvision.datasets.ImageFolder("dogImages/train/",␣
 ↪transform=transform["train"])
test_img = torchvision.datasets.ImageFolder("dogImages/test/",  ␣
 ↪transform=transform["test"])
valid_img = torchvision.datasets.ImageFolder("dogImages/valid/",␣
 ↪transform=transform["test"])

train_data = DataLoader(train_img, batch_size=10, shuffle=True)
test_data  = DataLoader(test_img,  batch_size=10, shuffle=True)
valid_data = DataLoader(valid_img, batch_size=10, shuffle=True)

loaders_scratch = {"train" : train_data,
                   "test"  : test_data,
                   "valid" : valid_data}

classes = train_img.classes
```

[239]:
```
print(f"Total of {len(classes)} dog breeds")
```

```
Total of 133 dog breeds
```

[235]:
```
plt.figure(figsize=(20,10))

def imshow(img):
        img = img / 2 + 0.5 #denormalize
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1,2,0)))
        plt.show()

dataiter = iter(train_data)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images, nrow=5))
```

```
for i, dog in enumerate(labels):
    print(i, classes[dog.item()])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```


```
0 088.Irish_water_spaniel
1 021.Belgian_sheepdog
2 058.Dandie_dinmont_terrier
3 050.Chinese_shar-pei
4 054.Collie
5 126.Saint_bernard
6 031.Borzoi
7 049.Chinese_crested
8 038.Brussels_griffon
9 087.Irish_terrier
```

[ ]:

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:

- the input size is 224x224 like in the VGG16 (and majority of other pretrained models on pytorch website). Higher input resolution would also require more computational power, but since I am not using CUDA while training, 224x224 seemed to be an optimal size
- I have added random horizontal flip and affine shifts of up to 20%. Initially I have also added the random resize and crop, but for many samples heads were not in the picture anymore. Since face features are rather more distinctive than bodies, I have decided to remove this augmentation and simply go with standard center crop.

[238]:
```
nn.Conv2d(in_channels=3, out_channels=6, kernel_size=(3,3),␣
↪padding=1)(images[0].unsqueeze(0)).shape
```

[238]:
```
torch.Size([1, 6, 224, 224])
```

1.2.8 (IMPLEMENTATION) Model Architecture

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

[44]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN

        # Convolutional layers
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6,␣
↪kernel_size=(3,3), padding=1) # in 3x224x224 -> out 6x112x112
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16,␣
↪kernel_size=(3,3), padding=1)  # in 6x112x112 -> out 16x56x56
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,␣
↪kernel_size=(3,3), padding=1) # in 16x56x56 -> out 32x28x28
        self.conv4 = nn.Conv2d(in_channels=32, out_channels=64,␣
↪kernel_size=(3,3), padding=1) # in 32x28x28 -> out 64x14x14
        self.conv5 = nn.Conv2d(in_channels=64, out_channels=128,␣
↪kernel_size=(3,3), padding=1) # in 64x14x14 -> out 128x7x7

        # Batch norm
        self.bn1 = nn.BatchNorm2d(6)
        self.bn2 = nn.BatchNorm2d(16)
        self.bn3 = nn.BatchNorm2d(32)
        self.bn4 = nn.BatchNorm2d(64)


        # Fully connected layers
        self.fc1 = nn.Linear(in_features=128*7*7, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features= 512)
        self.out = nn.Linear(in_features=512, out_features=133)


    def forward(self, x):
        ## Define forward behavior

        # 1. Conv: in 3x224x224 -> out 6x112x112
```

```
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = self.bn1(x)

        # 2. Conv: in 6x112x112 -> out 16x56x56
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = self.bn2(x)

        # 3. Conv: in 16x56x56 -> out 32x28x28
        x = self.conv3(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = self.bn3(x)

        # 4. Conv: in 32x28x28 -> out 64x14x14
        x = self.conv4(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = self.bn4(x)

        # 5. Conv: in 64x14x14 -> out 128x7x7
        x = self.conv5(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        # Flatten
        x = x.reshape(-1, self.nfeatures(x))
        x = F.dropout(x, 0.2)


        # 6. FC1: 32*28*28 -> 512
        x = self.fc1(x)
        x = F.relu(x)
        x = F.dropout(x, 0.2)

        # 7. FC2: 512 -> 256
        x = self.fc2(x)
        x = F.relu(x)

        # 8. Output
        x = self.out(x)
        return x

    def nfeatures(self, x):
```

```
        n = 1
        for i in x.size()[1:]:
            n *= i

        return n

#-#-# 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:

1. Simple model with 3 Convolutional and 2 Linear layers. Val loss started spiking around 5th epoch -> overfitting
2. Increased complexity in the linear layers (3 layers with more features). This prevented val loss from going up, but model stopped improving at 5-7th epoch
3. Started experimenting with new Convolutional layers. Adding filters and increasing granularity has imroved performance, but overfitting was still an issue (increasing sparsity between train and valid loss)
4. Added dropout and data augmentation -> slight improvement in val loss
5. Added batch normalisation -> significant improvement in performance

As you will see below, model stopped improving at 15th epoch. There is still a lot of room for improvement. I would try further increasing model complexity and change the size and order of convolutional filters

1.2.9 (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.

[47]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001, momentum=0.
 ↪9)
```

1.2.10 (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'`.

[8]:
```
# the following import is required for training to be robust to truncated␣
 ↪images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

train_losses = []
valid_losses = []

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_loss))

            optimizer.zero_grad() # zero the parameter gradients

            # forward + backward + optimize
            pred = model(data)
            loss = criterion(pred, target)
            loss.backward()  # backpropagate errors - calculate gradients wrt␣
 ↪params
            optimizer.step() # update params

            train_loss = 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

            pred = model(data)
            loss = criterion(pred, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data -␣
↪valid_loss))

            valid_losses.append([epoch, batch_idx, valid_loss])

            #print(f"Epoch: {epoch} \t Train loss: {train_loss:.4f} \t Valid␣
↪loss: {valid_loss:.4f}")



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


        ## TODO: save the model if validation loss has decreased

        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model


epochs = 20
model_name = 'model_scratch.pt'
# train the model
```

```
model_scratch = train(epochs, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, model_name)

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

```
Epoch: 1        Training Loss: 4.848303         Validation Loss: 4.704661
Epoch: 2        Training Loss: 4.498919         Validation Loss: 4.329505
Epoch: 3        Training Loss: 4.237106         Validation Loss: 4.183437
Epoch: 4        Training Loss: 4.047360         Validation Loss: 4.090852
Epoch: 5        Training Loss: 3.910327         Validation Loss: 4.013178
Epoch: 6        Training Loss: 3.763604         Validation Loss: 3.877303
Epoch: 7        Training Loss: 3.636727         Validation Loss: 3.821249
Epoch: 8        Training Loss: 3.493447         Validation Loss: 3.703688
Epoch: 9        Training Loss: 3.346449         Validation Loss: 3.678420
Epoch: 10       Training Loss: 3.224846         Validation Loss: 3.870714
Epoch: 11       Training Loss: 3.080395         Validation Loss: 3.646093
Epoch: 12       Training Loss: 2.978658         Validation Loss: 3.517780
Epoch: 13       Training Loss: 2.831717         Validation Loss: 3.468354
Epoch: 14       Training Loss: 2.696723         Validation Loss: 3.567818
Epoch: 15       Training Loss: 2.539376         Validation Loss: 3.462093
Epoch: 16       Training Loss: 2.412947         Validation Loss: 3.441731
Epoch: 17       Training Loss: 2.274463         Validation Loss: 3.422442
Epoch: 18       Training Loss: 2.121807         Validation Loss: 3.412602
Epoch: 19       Training Loss: 2.008310         Validation Loss: 3.400733
Epoch: 20       Training Loss: 1.864391         Validation Loss: 3.464373
```

[8]:
```
<All keys matched successfully>
```

[45]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

[45]:
```
<All keys matched successfully>
```

1.2.11 (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%.

[42]:
```
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)
```

[48]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.374201


Test Accuracy: 20% (169/836)
```

This is just enough to pass, but it is clear that the net has clearly started to overfit already at the 15 epoch. If I was to develope a production model from scratch, I would start from from further data augmentation, dropouts, change model architecture etc.

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.

1.2.12 (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.

[29]:
```
## TODO: Specify data loaders

import os
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim import lr_scheduler

from torch.utils.data import DataLoader
import torchvision.transforms as transforms

data_transform = {

    "train" : transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomAffine(degrees=0.3),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )]),

    "test" : transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )]),
   "valid" : transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )])}

dog_img_path = 'dogImages/'

dog_datasets = {x: torchvision.datasets.ImageFolder(os.path.join(dog_img_path,␣
 ↪x),
```

```
                                                    transform=data_transform[x])
                                                    for x in ["train", "test",␣
 ↪"valid"]}

loaders_transfer = {x: DataLoader(dog_datasets[x], batch_size=10, shuffle=True,␣
 ↪num_workers=4)
                for x in ["train", "test", "valid"]}


classes = dog_datasets["train"].classes
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
dataset_sizes = {x: len(dog_datasets[x]) for x in ["train", "test", "valid"]}
```

1.2.13 (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`.

[30]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

# Using ResNeXt101 as fixed feature extractor
model_transfer = torchvision.models.resnet18(pretrained=True)

# Freezing pretrained model
for param in model_transfer.parameters():
    param.requires_grad = False

# Adding Fully Connected layer for dog breed classification
num_ftrs = model_transfer.fc.in_features
model_transfer.fc = nn.Linear(num_ftrs, 133)

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

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to
/home/le-roy/.cache/torch/hub/checkpoints/resnet18-5c106cde.pth

HBox(children=(IntProgress(value=0, max=46827520), HTML(value='')))
```

[ ]:
```

```

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:

Basically there are two options for transfer learning: fine-tune all layers of existing pretrained model given new data, or "freeze" params and add one Linear layer to predict classes in your data. Since we have a relatively small dataset, and the original ImageNet dataset contains a lot of different dog breeds already, picking the first technique could lead to overfitting.

In terms of the model selection, I went with ResNet18 due to its training time, since I do not have CUDA on my laptop:) Otherwise I would pick something more advanced like ResNext101 or DenseNet etc.

1.2.14 (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.

[31]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.parameters(), lr=0.001,␣
↪momentum=0.9)
```

1.2.15 (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'`.

[37]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
from tqdm import tqdm

train_losses = []
valid_losses = []

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 tqdm(enumerate(loaders['train']),␣
↪desc="Train", total=len(loaders['train']), position=0):
            # 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() # zero the parameter gradients

            # forward + backward + optimize
            pred = model(data)
            loss = criterion(pred, target)
            loss.backward()  # backpropagate errors - calculate gradients wrt␣
↪params
            optimizer.step() # update params

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


        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in tqdm(enumerate(loaders['valid']),␣
↪desc="Valid", total=len(loaders['valid']), position=0):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss

            pred = model(data)
            loss = criterion(pred, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data -␣
↪valid_loss))

            #print(f"Epoch: {epoch} \t Train loss: {train_loss:.4f} \t Valid␣
↪loss: {valid_loss:.4f}")



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

```
        train_losses.append(train_loss)
        valid_losses.append(valid_loss)


        ## TODO: save the model if validation loss has decreased

        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model
```

[38]:
```
n_epochs = 10

# train the model
model_transfer = train(n_epochs, 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'))
```

```
Train: 100%|| 668/668 [10:58<00:00,  1.01it/s]

Epoch: 1        Training Loss: 1.686759         Validation Loss: 1.140038


Train: 100%|| 668/668 [11:16<00:00,  1.01s/it]
Valid: 100%|| 84/84 [01:10<00:00,  1.19it/s]

Epoch: 2        Training Loss: 1.268201         Validation Loss: 0.906886

Train: 100%|| 668/668 [11:24<00:00,  1.03s/it]
Valid: 100%|| 84/84 [01:30<00:00,  1.08s/it]

Epoch: 3        Training Loss: 1.065474         Validation Loss: 0.800735

Train: 100%|| 668/668 [12:13<00:00,  1.10s/it]
Valid: 100%|| 84/84 [01:12<00:00,  1.16it/s]

Epoch: 4        Training Loss: 0.920021         Validation Loss: 0.734773

Train: 100%|| 668/668 [12:22<00:00,  1.11s/it]
Valid: 100%|| 84/84 [01:28<00:00,  1.05s/it]

Epoch: 5        Training Loss: 0.826467         Validation Loss: 0.685949

Train: 100%|| 668/668 [13:57<00:00,  1.25s/it]
Valid: 100%|| 84/84 [01:38<00:00,  1.17s/it]
```

```
Epoch: 6        Training Loss: 0.764538         Validation Loss: 0.659974

Train: 100%|| 668/668 [14:20<00:00,  1.29s/it]
Valid: 100%|| 84/84 [01:16<00:00,  1.10it/s]

Epoch: 7        Training Loss: 0.723437         Validation Loss: 0.656318

Train: 100%|| 668/668 [10:53<00:00,  1.02it/s]
Valid: 100%|| 84/84 [01:14<00:00,  1.13it/s]

Epoch: 8        Training Loss: 0.670126         Validation Loss: 0.632076

Train: 100%|| 668/668 [11:14<00:00,  1.01s/it]
Valid: 100%|| 84/84 [01:15<00:00,  1.11it/s]

Epoch: 9        Training Loss: 0.621858         Validation Loss: 0.610421

Train: 100%|| 668/668 [10:58<00:00,  1.01it/s]
Valid: 100%|| 84/84 [01:15<00:00,  1.11it/s]

Epoch: 10       Training Loss: 0.595384         Validation Loss: 0.593539
```

[39]:
```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

[39]:
```
<All keys matched successfully>
```

1.2.16 (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%.

[49]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.596198


Test Accuracy: 81% (684/836)
```

1.2.17 (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.

[120]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

from PIL import Image
import torchvision.transforms as transforms

# 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 dog_datasets['train'].
 ↪classes]

def predict_breed_transfer(img_path, model, class_names):
    """
        Returns:
            - breed name
            - class index
            - full class name
    """

    # load the image and return the predicted breed

       # Step 0. Load file
    img_raw = Image.open(img_path)
    img_np = np.asarray(img_raw).transpose(2,0,1) # reordering dimensions to␣
 ↪[channel, h, w]
    t = torch.tensor(img_np, dtype=torch.float32) # convert to torch tensor

    """ Step 1. Transform PIL image using following manipulations:
        - Resize to 256x256
        - Crop at center to 224x224
        - Normalize using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224,␣
 ↪0.225]
    """
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        )])

    t = transform(img_raw)

    t = t.unsqueeze(0) # setting tensor te required dimensions [batch, channel,␣
 ↪h, w]

    # Step 2. Obtaining predicted class index
    idx = torch.argmax(model(t))

    return class_names[idx], idx.item(), classes[idx]
```

[122]:
```
pred, *_ = predict_breed_transfer(dog_files_short[0], model_transfer,␣
 ↪class_names)
print(pred)
```

```
Image.open(dog_files_short[0])
```

```
Irish terrier
```

[122]:

[123]:
```
pred, *_ = predict_breed_transfer(human_files_short[0], model_transfer,␣
 ↪class_names)
print(pred)
Image.open(human_files_short[0])
```

```
American staffordshire terrier
```

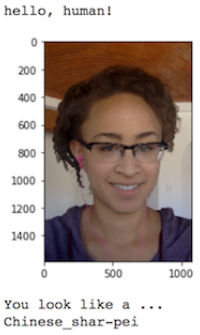
[123]:

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!



Sample Human Output

1.2.18 (IMPLEMENTATION) Write your Algorithm

[190]:
```
### 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

    whoami = None

    if dog_detector(img_path):
        whoami = "doggo"

    elif face_detector(img_path):
        whoami = "hooman"

    img = np.asarray(Image.open(img_path))

    if whoami:
        # predict breed
        breed, idx, breed_folder = predict_breed_transfer(img_path,␣
↪model_transfer, class_names)

        # construct subplot
        fig, axs = plt.subplots(1,2, figsize=(10,5))

        axs[0].set_title(f"Hello, {whoami}!", fontsize=12)
        axs[0].imshow(img)
        axs[0].set_xlabel(f"\nYou look like {breed}.", fontsize=12)

        # fetch image of resembling dog breed
        PATH = "dogImages/train/"
        breed_example = np.random.choice(os.listdir(PATH + breed_folder))
        ex_path = os.path.join(PATH, breed_folder, breed_example)
        axs[1].imshow(np.asarray(Image.open(ex_path)))

        #plt.savefig("images/"+img_path.split("/")[1])

    else:
        plt.suptitle("Sorry, do I know you?:)", fontsize=12)
        plt.imshow(img)

    plt.show()
```

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?

1.2.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

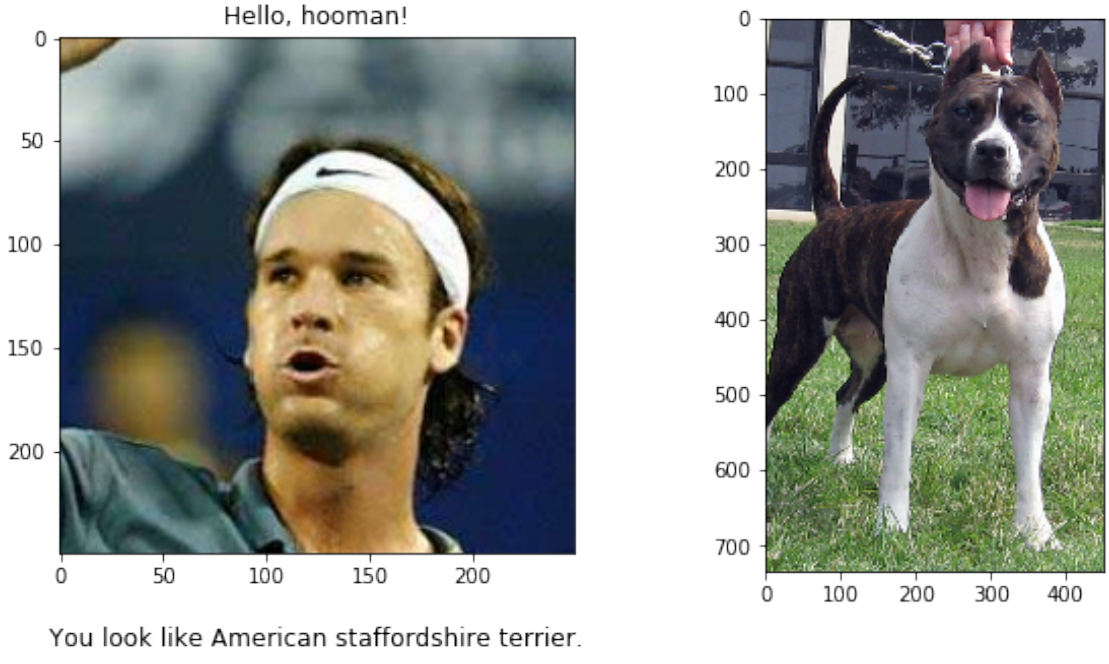
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

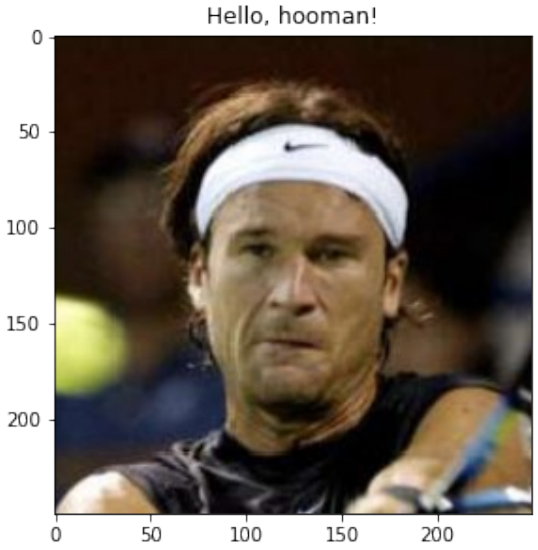
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

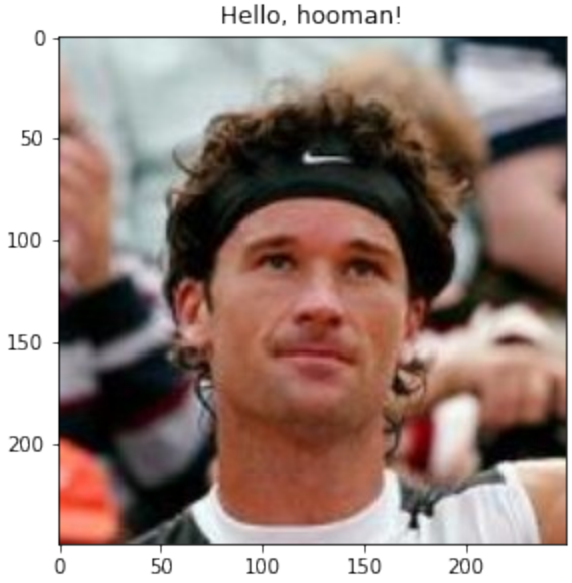
- Increase dataset size. 668 images per epoch is a small dataset for such a high class sparsity.
- Additionally apply further data augmentation techniques to the training data. For example, I have replaced the initially selected random crop with center crop, because for some images dog faces were not in the picture, which had negative effect on model performace.
- Select more advanced model for transfer learning. I went with ResNet18 mainly due to the smaller training time (10m per epoch). A more advanced base model like i.e. ResNeXT101 could potentially improve overall performance
- In terms of delivery, this algo seems to me like a fun android app material:)

[150]:
```
## 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)
```

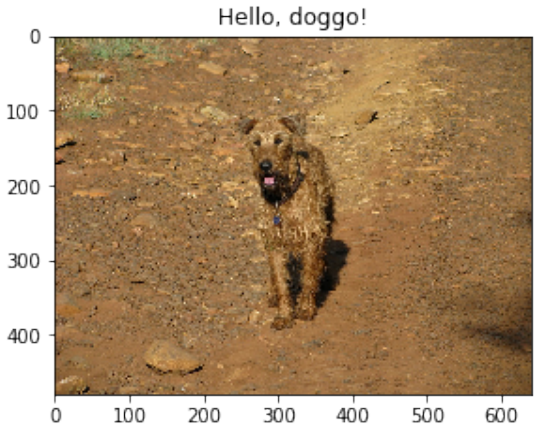


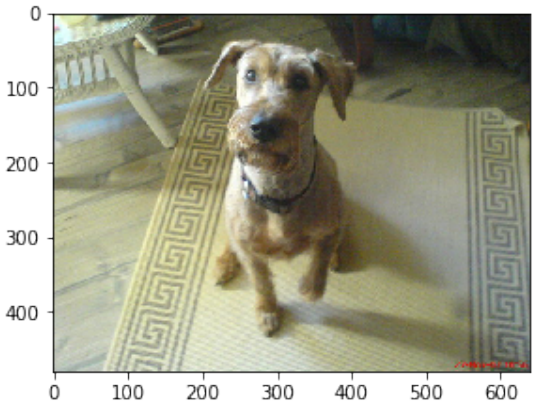


You look like American staffordshire terrier.

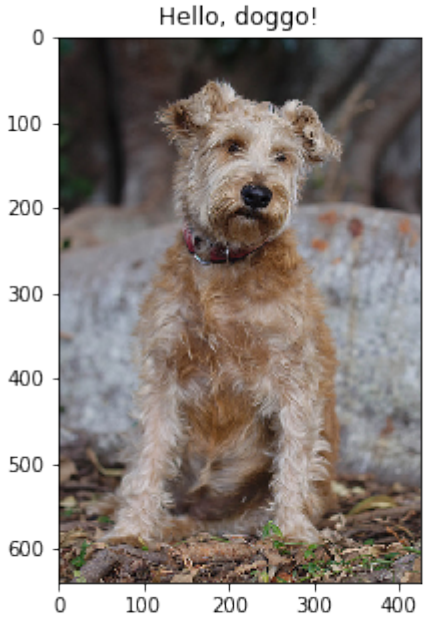


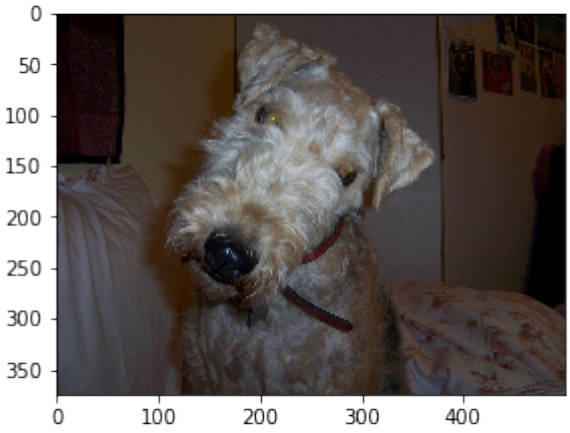
You look like Dogue de bordeaux.



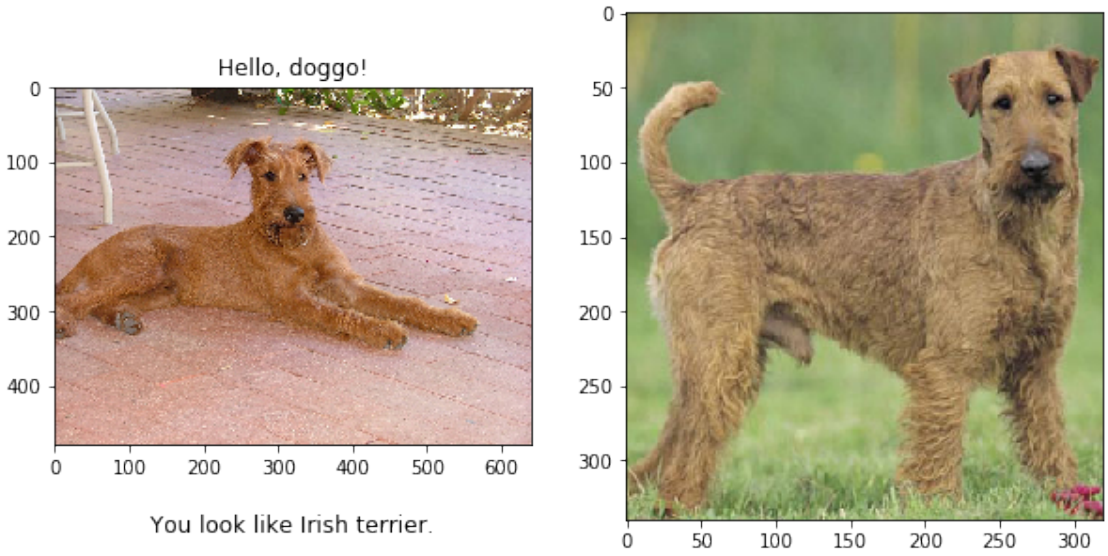


You look like Irish terrier.

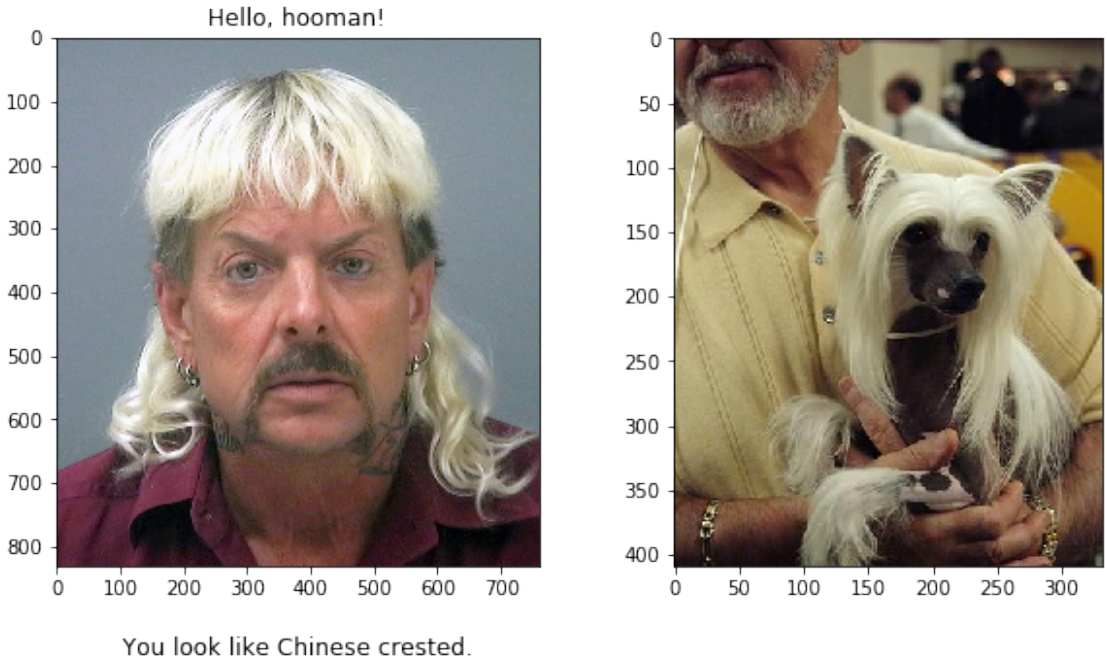


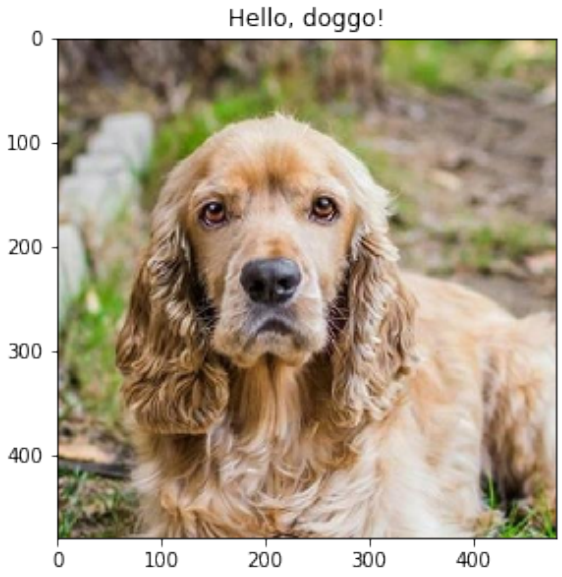


You look like Lakeland terrier.

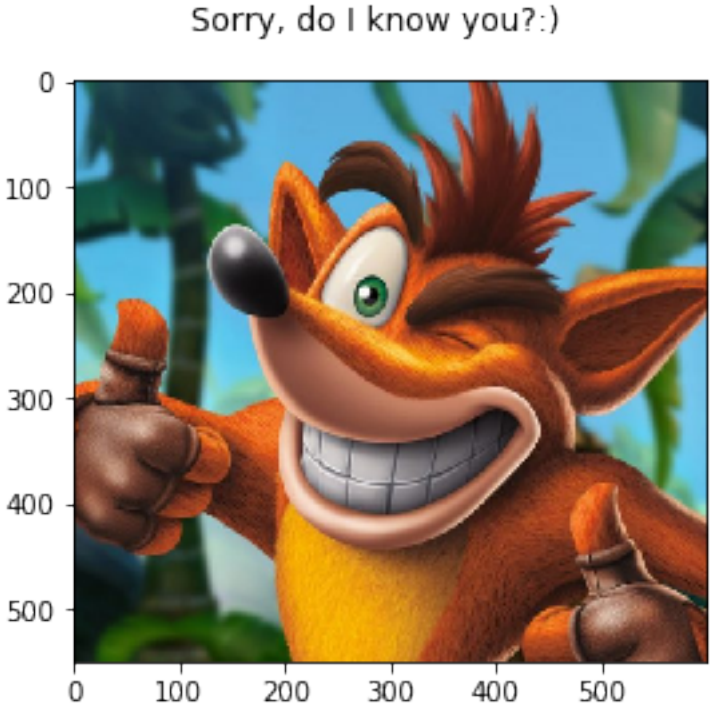


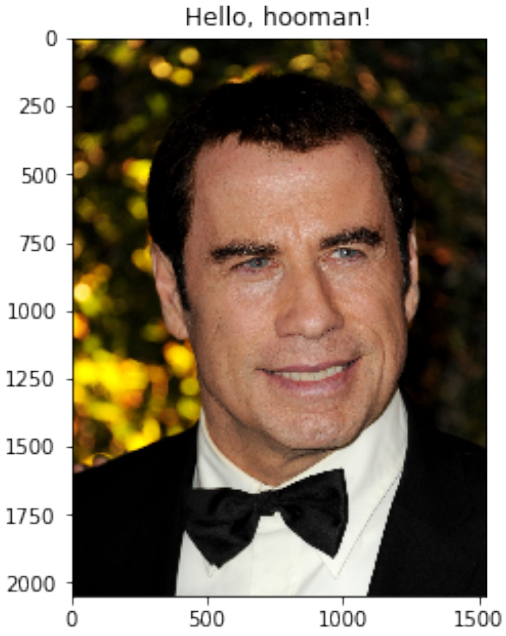
[194]:
```
img_test = os.listdir("img_test/")

for file in [os.path.join("img_test", img) for img in img_test]:
    run_app(file)
```



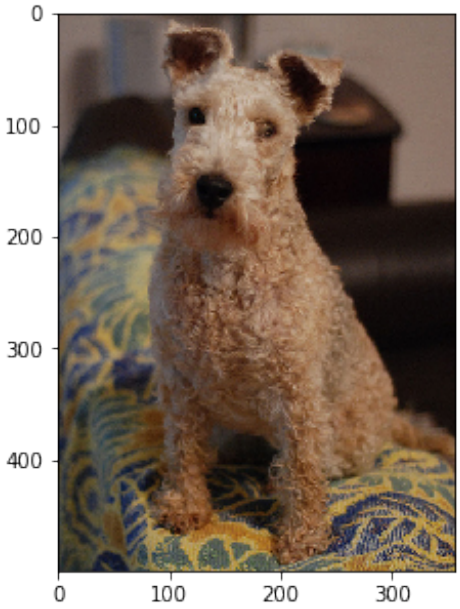


You look like Cocker spaniel.

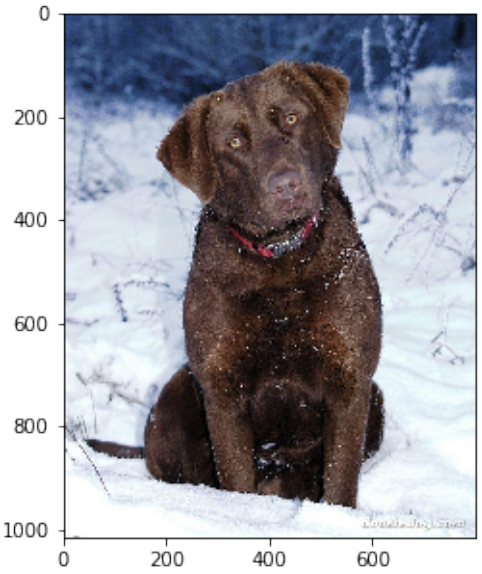


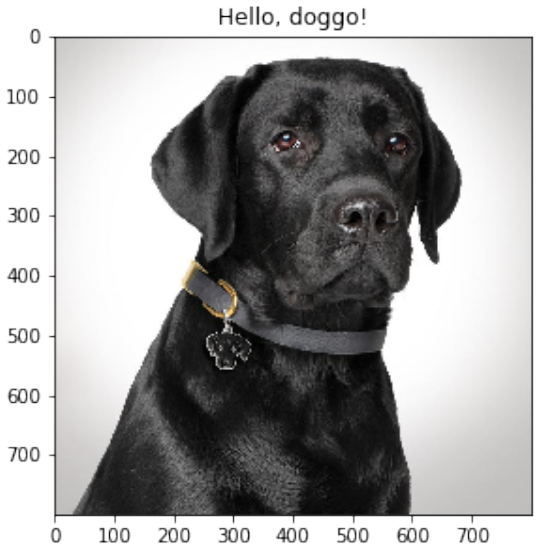


You look like Lakeland terrier.

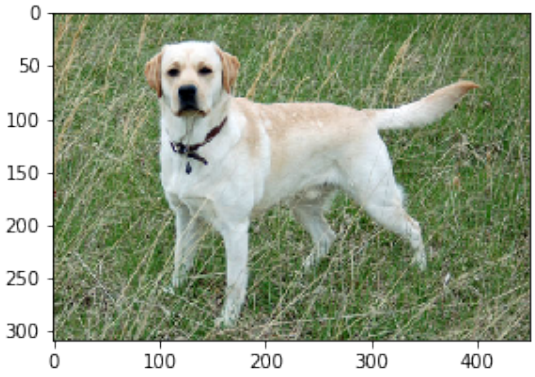


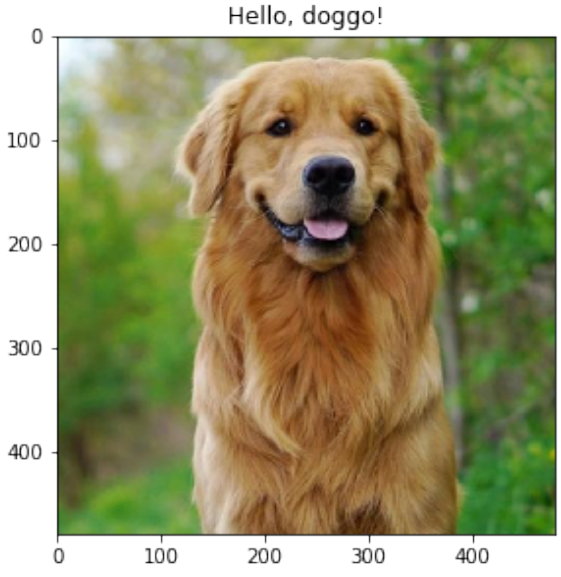
You look like Chesapeake bay retriever.



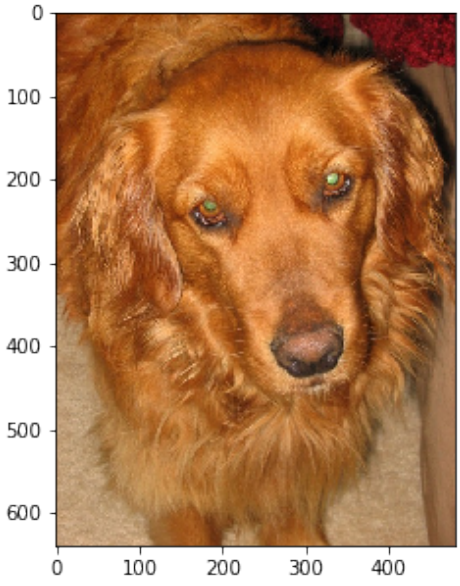


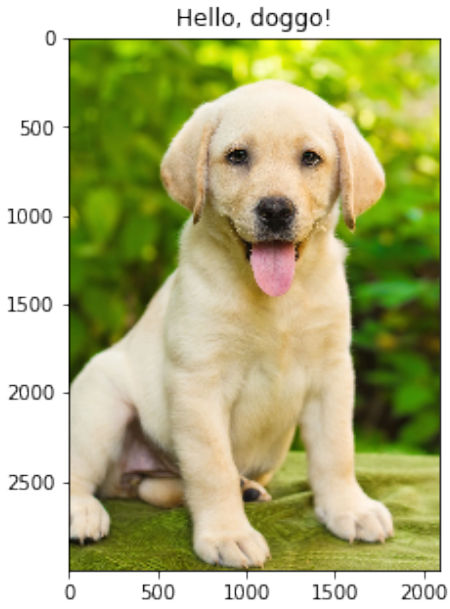
You look like Labrador retriever.

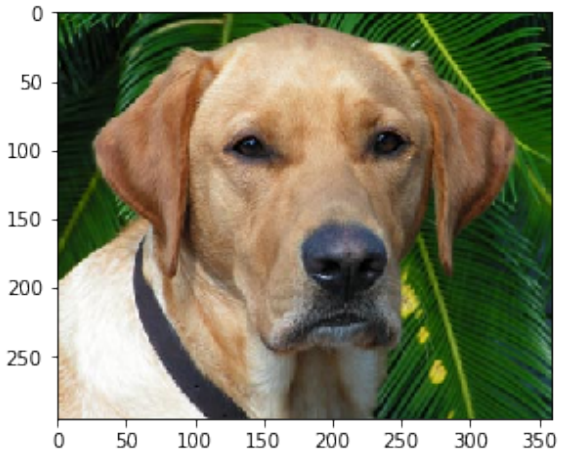




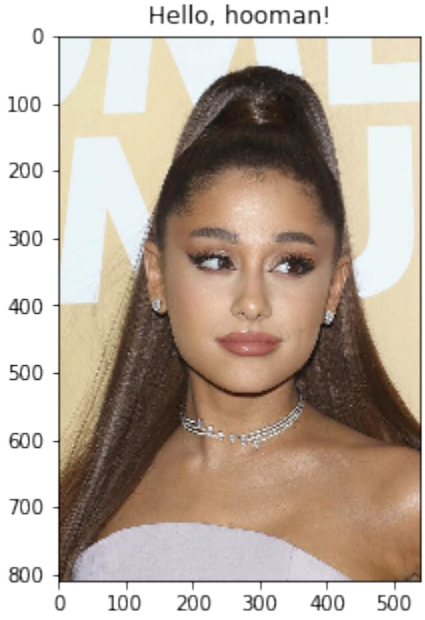
You look like Golden retriever.

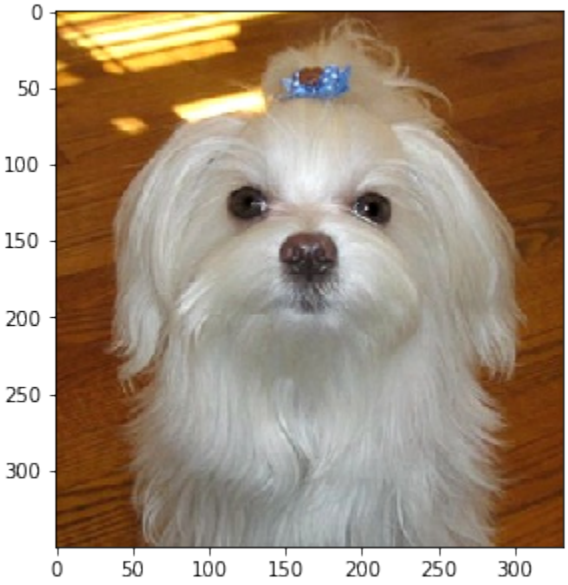






You look like Labrador retriever.





You look like Maltese.

[ ]:

[ ]:

[ ]:

[ ]: