

dog_app

November 12, 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 [15]: import numpy as np
         from glob import glob
         import cv2
         import matplotlib.pyplot as plt
         from tqdm import tqdm
         import torch
         import torchvision.models as models
         from PIL import Image, ImageFile
         import torchvision.transforms as transforms
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         import os
         from torchvision import datasets
         from torch.utils.data.sampler import SubsetRandomSampler
         import torch.nn as nn
         import torch.nn.functional as F
         import torch.optim as optim

In [16]: # check if CUDA is available
         use_cuda = torch.cuda.is_available()

In [31]: # load filenames for human and dog images
         human_files = np.array(glob("/data/lfw/*/"))
         dog_files = np.array(glob("/data/dog_images/*/"))

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

There are 13233 total human images.

There are 8351 total dog images.

Step 1: Detect Humans

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

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

```

In [35]: %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))

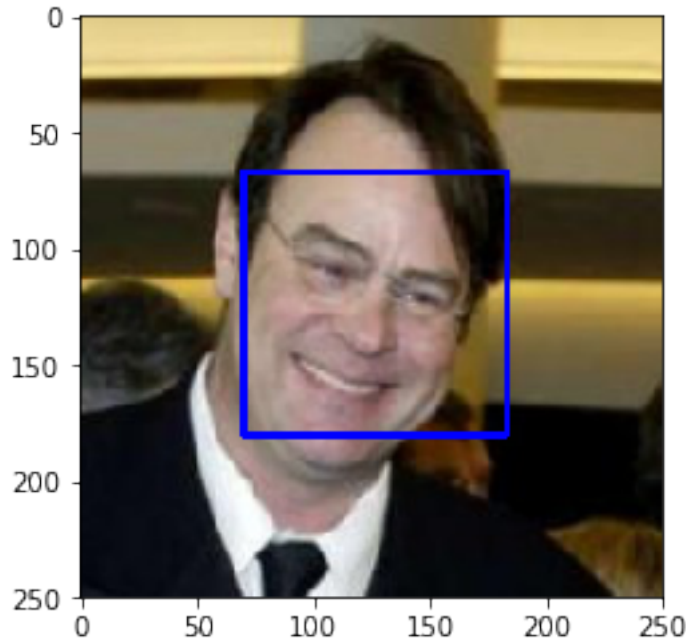
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img, (x,y), (x+w,y+h), (255,0,0), 2)

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.

- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

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

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

```
In [5]: 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.
        humanfacelist = []
        dogfacelist = []
        for i in range(100):
            hum = face_detector(human_files_short[i])
            dog = face_detector(dog_files_short[i])
            humanfacelist.append(hum)
            dogfacelist.append(dog)
        print(str(sum(humanfacelist))+"%"+" of the first 100 images in human_files have a detected face")
        print(str(sum(dogfacelist))+"%"+" of the first 100 images in dog_files have a detected face")
```

98% of the first 100 images in human_files have a detected human face

17% of the first 100 images in dog_files have a detected human face

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

Step 2: Detect Dogs

In this section, we use a [pre-trained model](#) to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [43]: # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

        # 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/vgg16-397923af.pth
 100%|| 553433881/553433881 [00:19<00:00, 28327223.18it/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 [41]: 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

        # pre-processing: mini-batches of 3-channel RGB images of shape (3 x H x W),
        # where H and W are expected to be at least 224. The images have to be loaded
        # in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406]
        # and std = [0.229, 0.224, 0.225]
        img = Image.open(img_path).convert('RGB')
        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])
    ])
    img = transform(img).unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    # get output
    output = VGG16(img)
    # convert output probabilities to predicted class
    _, preds_tensor = torch.max(output, 1)
    preds = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tensor)

    return preds # 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 [28]: ### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    pred = VGG16_predict(img_path)

    return ((pred>=151) and (pred<=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:

Printed below

```

In [10]: ### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
humans_detected_dogs = []
dogs_detected_dogs = []
for i in range(100):
    hum = dog_detector(human_files_short[i])
    dog = dog_detector(dog_files_short[i])
    humans_detected_dogs.append(hum)
    dogs_detected_dogs.append(dog)

```

```
print(str(sum(humans_detected_dogs))+"%"+" of the images in human_files_short have a detected dog")
print(str(sum(dogs_detected_dogs))+"%"+" of the images in dog_files_short have a detected dog")
```

```
1% of the images in human_files_short have a detected dog
100% of the images in dog_files_short have a detected dog
```

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

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

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

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

Brittany	Welsh Springer Spaniel
----------	------------------------

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

Curly-Coated Retriever	American Water Spaniel
------------------------	------------------------

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

Yellow Labrador	Chocolate Labrador
-----------------	--------------------

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imbalanced, 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 [17]: ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         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])
         ])

         transform_augm = transforms.Compose([
             transforms.RandomRotation(45),
             transforms.Resize(256),
             transforms.CenterCrop(224),
             transforms.RandomHorizontalFlip(),
             transforms.ToTensor(),
             transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
         ])

         train_data = datasets.ImageFolder('/data/dog_images/train', transform=transform_augm)
         valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=transform)
         test_data = datasets.ImageFolder('/data/dog_images/test', transform=transform)

         num_workers = 0
         batch_size = 20

         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_workers=
         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_workers=
         test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers=
```

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 code resizes the image to 256x256. Then, it crops the image in the center to 224x224 which

are the dimensions specified in pytorch documentation. Then, I normalize the image using values from pytoch documentation. - I applied augmentation to the training data to make the model generalize better and to prevent overfitting. Applied augmentations are random horizontal filping (with default probability 0.5) and random 10 degrees rotations.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [18]: # define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        # convolutional layer input is (224, 224, 3)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer 224/2/2/2 = 28 (bc 3 max pooling layers)
        self.fc1 = nn.Linear(64 * 28 * 28, 1000)
        # linear layer
        self.fc2 = nn.Linear(1000, 1000)
        self.fc3 = nn.Linear(1000, 133) # 133 classes
        # dropout layer (p=0.2)
        self.dropout = nn.Dropout(0.2)
        self.batch_norm1D = nn.BatchNorm1d(num_features=1000)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        # flatten image input
        x = x.view(-1, 64 * 28 * 28)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = self.dropout(F.relu(self.batch_norm1D(self.fc1(x))))
        x = self.dropout(F.relu(self.batch_norm1D(self.fc2(x))))
        # add 2nd hidden layer, with relu activation function
        x = self.fc3(x)
        return x

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

```

# instantiate the CNN
model_scratch = Net()

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

```

Net(

```

(conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(fc1): Linear(in_features=50176, out_features=1000, bias=True)
(fc2): Linear(in_features=1000, out_features=1000, bias=True)
(fc3): Linear(in_features=1000, out_features=133, bias=True)
(dropout): Dropout(p=0.2)
(batch_norm1d): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)

```

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

Answer:

- The input of the first layer is (224, 224, 3) RGB image and since the kernel is 3x3 and the stripe is 1, we add a padding of 1 to the convolution layers. - Then, in each convolutional layer, I increase the channels but to reduce complexity, I add a max pooling layer in between. The max pooling layers reduce the x and y dimentions of the channels by a factor that I set to 2. Since the input image is 224x224 and I have 3 max pooling layers, the input to the fully connected layer is 28x28. - Lastly, I add fully connected layers to produce the probabilities for the 133 classes. I add dropout layers with probability of 25% between the fully connected layers to avoid overfitting.

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

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

```

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 [19]: 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
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss += loss.item()*data.size(0)
            # 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()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            ## update the average validation loss
            valid_loss += loss.item()*data.size(0)
            # valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

```

```

# calculate average losses
train_loss = train_loss/len(train_loader.dataset)
valid_loss = valid_loss/len(valid_loader.dataset)

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

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

```

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

In [32]: # train the model

```
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
    criterion_scratch, use_cuda, 'model_scratch.pt')
```

```

Epoch: 1      Training Loss: 4.722793      Validation Loss: 4.660092
Validation loss decreased (inf --> 4.660092). Saving model ...
Epoch: 2      Training Loss: 4.433974      Validation Loss: 4.559115
Validation loss decreased (4.660092 --> 4.559115). Saving model ...
Epoch: 3      Training Loss: 4.269698      Validation Loss: 4.524389
Validation loss decreased (4.559115 --> 4.524389). Saving model ...
Epoch: 4      Training Loss: 4.147884      Validation Loss: 4.456146
Validation loss decreased (4.524389 --> 4.456146). Saving model ...
Epoch: 5      Training Loss: 4.026188      Validation Loss: 4.407316
Validation loss decreased (4.456146 --> 4.407316). Saving model ...
Epoch: 6      Training Loss: 3.927227      Validation Loss: 4.348983
Validation loss decreased (4.407316 --> 4.348983). Saving model ...
Epoch: 7      Training Loss: 3.834426      Validation Loss: 4.338407
Validation loss decreased (4.348983 --> 4.338407). Saving model ...
Epoch: 8      Training Loss: 3.746452      Validation Loss: 4.228581
Validation loss decreased (4.338407 --> 4.228581). Saving model ...
Epoch: 9      Training Loss: 3.645689      Validation Loss: 4.190355
Validation loss decreased (4.228581 --> 4.190355). Saving model ...
Epoch: 10     Training Loss: 3.552885      Validation Loss: 4.106801
Validation loss decreased (4.190355 --> 4.106801). Saving model ...

```

```

Epoch: 11          Training Loss: 3.486883          Validation Loss: 4.055947
Validation loss decreased (4.106801 --> 4.055947). Saving model ...
Epoch: 12          Training Loss: 3.412898          Validation Loss: 4.031553
Validation loss decreased (4.055947 --> 4.031553). Saving model ...
Epoch: 13          Training Loss: 3.329382          Validation Loss: 3.946697
Validation loss decreased (4.031553 --> 3.946697). Saving model ...
Epoch: 14          Training Loss: 3.260785          Validation Loss: 3.900336
Validation loss decreased (3.946697 --> 3.900336). Saving model ...
Epoch: 15          Training Loss: 3.172871          Validation Loss: 3.905441
Epoch: 16          Training Loss: 3.115170          Validation Loss: 3.781980
Validation loss decreased (3.900336 --> 3.781980). Saving model ...
Epoch: 17          Training Loss: 3.034173          Validation Loss: 3.774131
Validation loss decreased (3.781980 --> 3.774131). Saving model ...
Epoch: 18          Training Loss: 2.943292          Validation Loss: 3.679574
Validation loss decreased (3.774131 --> 3.679574). Saving model ...
Epoch: 19          Training Loss: 2.888461          Validation Loss: 3.849481
Epoch: 20          Training Loss: 2.806600          Validation Loss: 3.779245

```

```

In [8]: # load the model that got the best validation accuracy
        model_scratch.load_state_dict(torch.load('model_scratch.pt'))

```

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 [46]: 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}'.format(test_loss))

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

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

Test Loss: 4.890855
Test Accuracy:  1% (12/836)

```

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 [20]: ## TODO: Specify data loaders
         loaders_transfer = {'train': train_loader, 'valid': valid_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 [21]: ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=True)
         print(model_transfer.fc)

```

```

Linear(in_features=2048, out_features=1000, 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:

I used the resnet50 model as suggested above.

By looking at the structure of the model, we can see that the last layer is a fully connected layer that outputs 1000 features. For the dog breed classification problem, we only need to classify 133 outputs. Therefore, I will change the fully connected layer.

```
In [22]: model_transfer.fc = nn.Linear(2048, 133)
        print(model_transfer.fc)
        if use_cuda:
            model_transfer = model_transfer.cuda()

Linear(in_features=2048, out_features=133, bias=True)
```

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

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [24]: n_epochs = 15
        # train the model
        model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
                               criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1      Training Loss: 2.258341      Validation Loss: 0.852894
Validation loss decreased (inf --> 0.852894). Saving model ...
Epoch: 2      Training Loss: 0.904291      Validation Loss: 0.643187
Validation loss decreased (0.852894 --> 0.643187). Saving model ...
Epoch: 3      Training Loss: 0.728750      Validation Loss: 0.617624
Validation loss decreased (0.643187 --> 0.617624). Saving model ...
Epoch: 4      Training Loss: 0.631235      Validation Loss: 0.517142
Validation loss decreased (0.617624 --> 0.517142). Saving model ...
Epoch: 5      Training Loss: 0.575184      Validation Loss: 0.597033
Epoch: 6      Training Loss: 0.538891      Validation Loss: 0.534641
Epoch: 7      Training Loss: 0.485832      Validation Loss: 0.491377
Validation loss decreased (0.517142 --> 0.491377). Saving model ...
Epoch: 8      Training Loss: 0.479076      Validation Loss: 0.504963
Epoch: 9      Training Loss: 0.444034      Validation Loss: 0.639487
Epoch: 10     Training Loss: 0.434617      Validation Loss: 0.575493
Epoch: 11     Training Loss: 0.417515      Validation Loss: 0.605072
Epoch: 12     Training Loss: 0.419660      Validation Loss: 0.547591
Epoch: 13     Training Loss: 0.376053      Validation Loss: 0.634738
Epoch: 14     Training Loss: 0.397752      Validation Loss: 0.576669
Epoch: 15     Training Loss: 0.392880      Validation Loss: 0.586920
```

```
In [25]: # load the model that got the best validation accuracy (uncomment the line below)
        model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```


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 [27]: 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}'.format(test_loss))

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

test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.480169
Test Accuracy: 86% (719/836)
```

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

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

```

class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].dataset]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = Image.open(img_path).convert('RGB')
    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])
    ])
    img = transform(img).unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    # get output
    output = model_transfer(img)
    # convert output probabilities to predicted class
    _, preds_tensor = torch.max(output, 1)
    # predicted class index
    preds = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tensor)

    return class_names[preds]

```

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

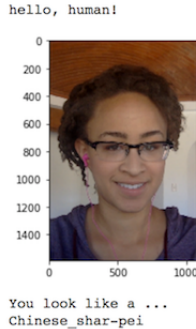
In [62]: *### TODO: Write your algorithm.*

Feel free to use as many code cells as needed.

```

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    if face_detector(img_path):
        predicted_breed = predict_breed_transfer(img_path)

```



Sample Human Output

```
plt.imshow(Image.open(img_path))
plt.show()
print('Hello, Human!')
print('You look like a ...\n' + predicted_breed + '\n\n')
elif dog_detector(img_path):
    predicted_breed = predict_breed_transfer(img_path)
    plt.imshow(Image.open(img_path))
    plt.show()
    print('dog!')
    print('Detected dog looks like a ...\n' + predicted_breed + '\n\n')
else:
    predicted_breed = predict_breed_transfer(img_path)
    plt.imshow(Image.open(img_path))
    plt.show()
    print('Error! You are neither a human nor a dog.\n But, you look like a ' + pre
```

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.

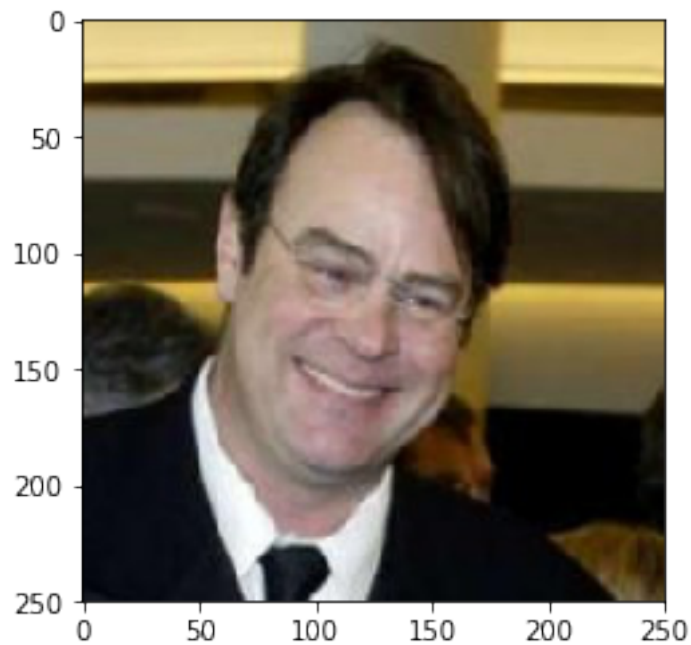
Answer: (Three possible points for improvement)

- Instead of center-cropping, cropping the images around the face of the dog or human can improve the detection of new images that do not have the face centered.

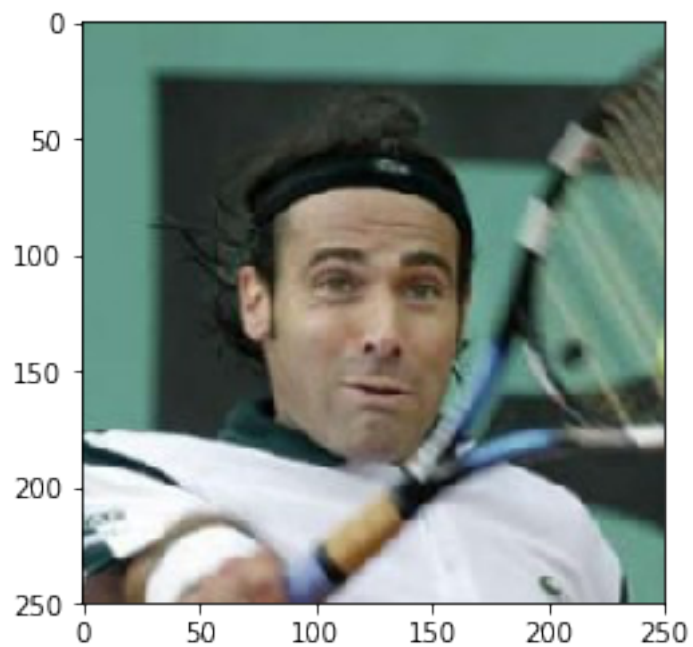
- Having more data is always better. The dataset has 8351 dog images, roughly 62 images per dog breed.

- Having a more complex architecture for the network can improve the results but would take longer time to train.

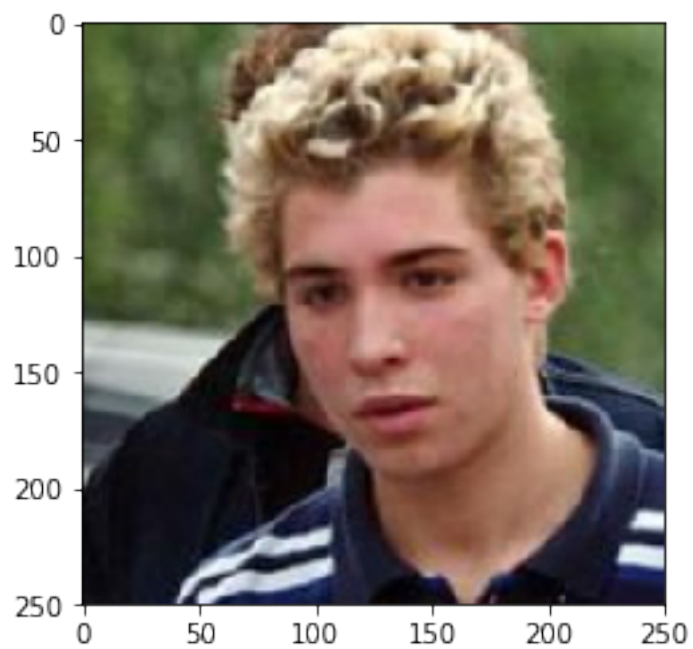
```
In [65]: ## TODO: Execute your algorithm from Step 6 on  
## at least 6 images on your computer.  
## Feel free to use as many code cells as needed.  
  
## suggested code, below  
for file in np.hstack((human_files[:3], dog_files[:3])):  
    run_app(file)
```



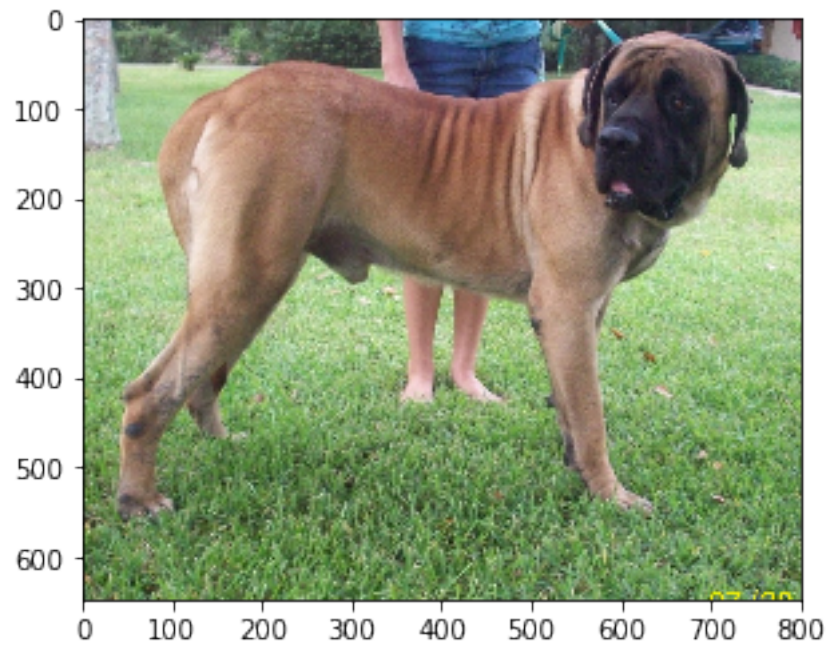
```
Hello, Human!  
You look like a ...  
Norfolk terrier
```



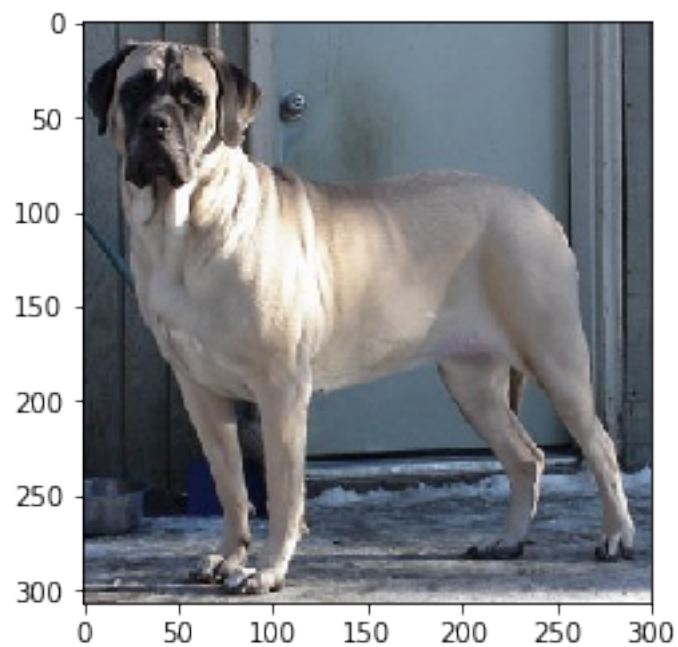
Hello, Human!
You look like a ...
Chihuahua



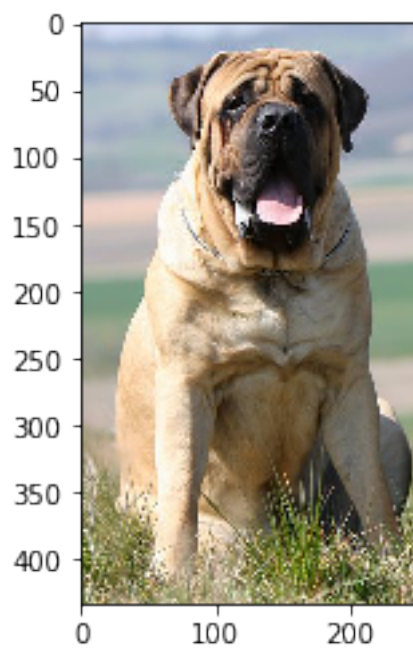
Hello, Human!
You look like a ...
American water spaniel



dog!
Detected dog looks like a ...
Mastiff



dog!
Detected dog looks like a ...
Mastiff

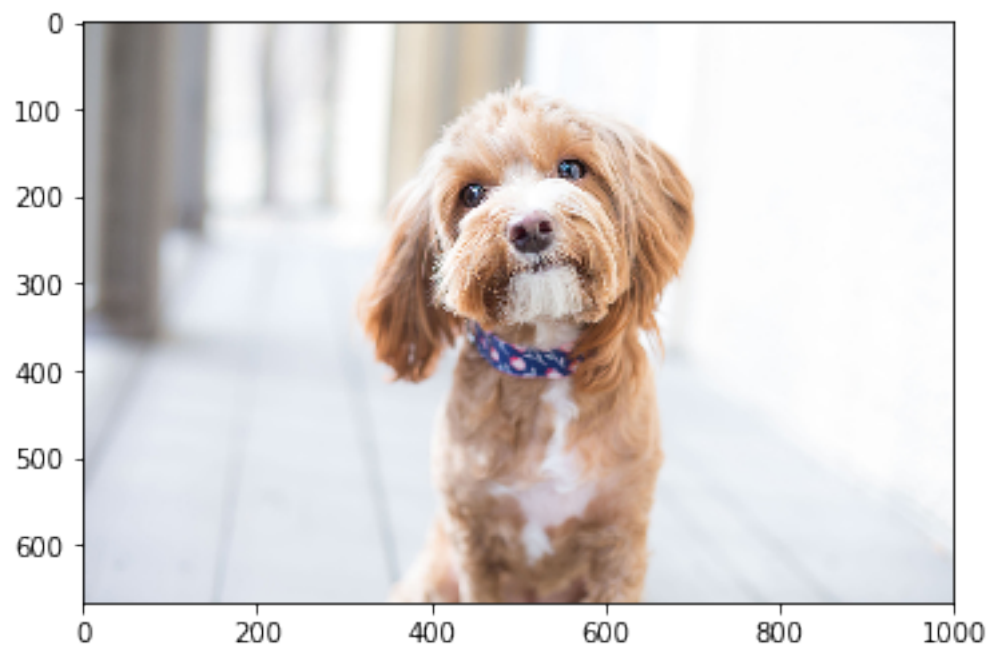


```
dog!  
Detected dog looks like a ...  
Bullmastiff
```

```
In [64]: pics = [os.path.join('./test_pics/', pic) for pic in os.listdir('./test_pics/')]  
         for pic in pics:  
             run_app(pic)
```



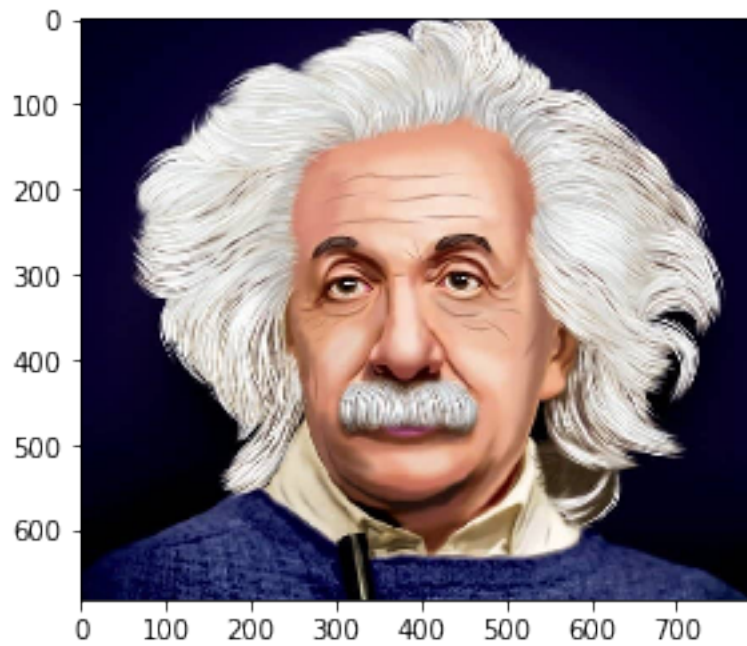
```
Error! You are neither a human nor a dog.  
But, you look like a Nova scotia duck tolling retriever
```

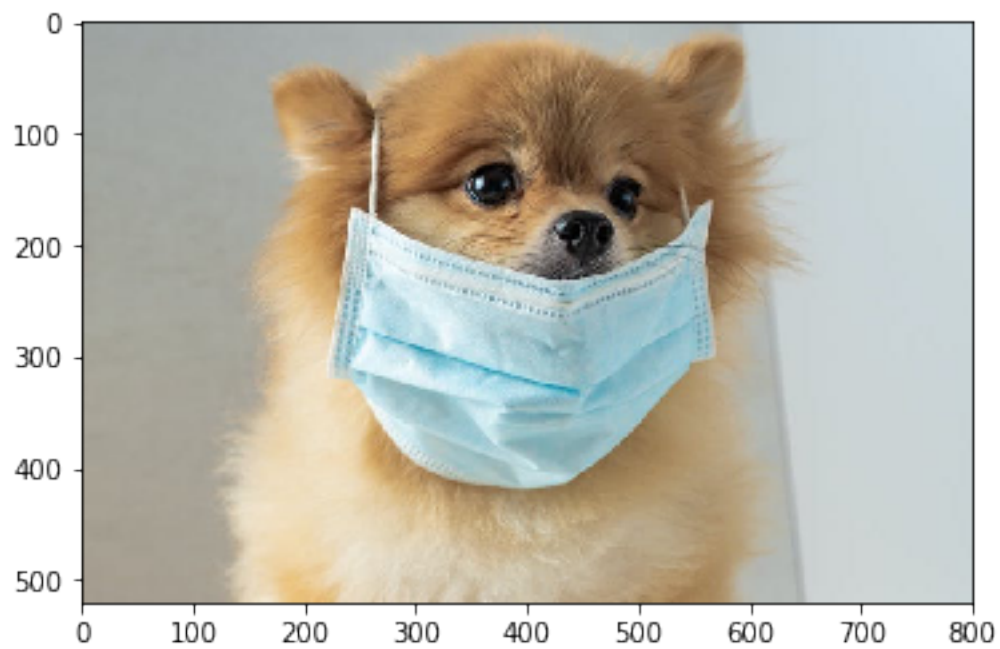
dog!
Detected dog looks like a ...
Silky terrier



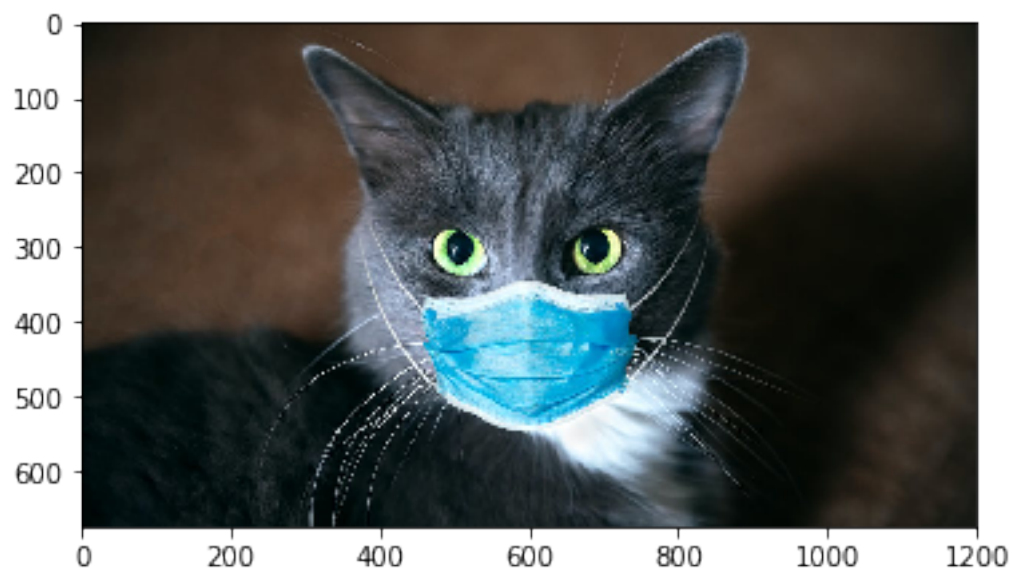
Hello, Human!
You look like a ...
Maltese



Hello, Human!
You look like a ...
Chinese crested



dog!
Detected dog looks like a ...
Pomeranian



```
Error! You are neither a human nor a dog.  
But, you look like a Pomeranian
```

```
In [ ]:
```