dog_app_saved

September 15, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

        # 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(f'There are {len(human_files)} total human images.')
        print(f'There are {len(dog_files)} total dog images.')
```

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

Step 1: Detect Humans

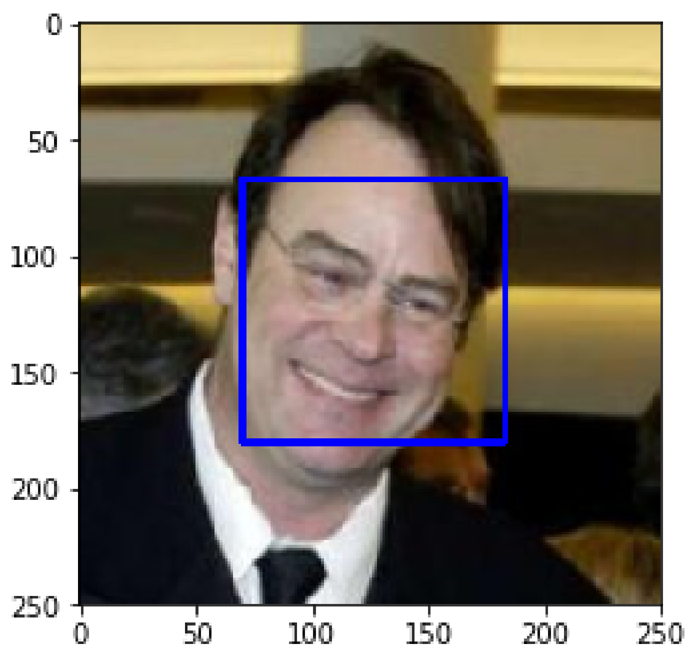
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

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

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

        # find faces in image
        faces = face_cascade.detectMultiScale(gray)

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

```python
# 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 [3]: # 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)
            human_faces = face_cascade.detectMultiScale(gray)
            return len(human_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)
98 % in human_files
17 % in dog_files

```
In [4]: from tqdm import tqdm

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        list_human = [face_detector(human_files_short[i]) for i in range (0, 100)]
        print (sum (list_human))

        list_dog = [face_detector(dog_files_short[i]) for i in range (0, 100)]
        print (sum (list_dog))
```

```
98
17
```

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

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

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 [6]: import torch
        import torchvision.models as models

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

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

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

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

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

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

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

```
In [7]: from PIL import Image
        import torchvision.transforms as transforms
        import torchvision
        from torchvision import datasets, models, transforms
        def VGG16_predict(img_path):
            '''
            Use pre-trained VGG-16 model to obtain index corresponding to
            predicted ImageNet class for image at specified path
```

```
    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path


    image = Image.open(img_path)

    data_transform = transforms.Compose([transforms.Resize(256),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         ])

    image_tensor = data_transform(image)
    image_tensor = image_tensor.unsqueeze(0)

    # move tensor to cuda
    if torch.cuda.is_available():
        image_tensor = image_tensor.cuda()



    ## Return the *index* of the predicted class for that image
    output = VGG16(image_tensor)

    if torch.cuda.is_available():
        output = output.cpu()

    class_index = output.data.numpy().argmax()

    return class_index # 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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            class_index = VGG16_predict(img_path)
            return (151 <= class_index <= 268)
```

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: 0 percentage of the images in human_files_short have a detected dog
90 percentage of the images in dog_files_short have a detected dog

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        list_human = [dog_detector(human_files_short[i]) for i in range (0, 100)]
        print (sum (list_human))

        list_dog = [dog_detector(dog_files_short[i]) for i in range (0, 100)]
        print (sum (list_dog))
```

```
0
90
```

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

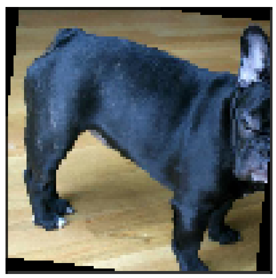
In [11]:
```
import os
import torch
from torchvision import datasets
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import torchvision.transforms as transforms

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

# define dataloader parameters
batch_size = 20
num_workers=0

# define training and test data directories
data_dir = '/data/dog_images/'
```

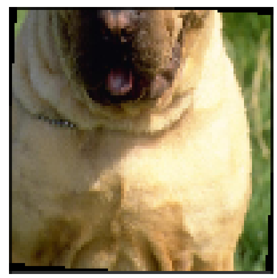
```
train_dir = os.path.join(data_dir, 'train/')
test_dir = os.path.join(data_dir, 'test/')
validation_dir = os.path.join(data_dir, 'valid/')


train_data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
transforms.RandomHorizontalFlip(), # randomly flip and rotate
transforms.RandomRotation(10),
transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

test_data_transform = transforms.Compose([transforms.Resize(256),
        transforms.CenterCrop(224),
transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])


# prepare data loaders
train_data = datasets.ImageFolder(train_dir, transform=train_data_transform)
validation_data = datasets.ImageFolder(validation_dir, transform=test_data_transform)
test_data = datasets.ImageFolder(test_dir, transform=test_data_transform)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
valid_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)

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

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

In [12]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image
```

```
In [13]: # obtain one batch of training images
         dataiter = iter(train_loader)
         images, labels = dataiter.next()
         images = images.numpy() # convert images to numpy for display

         # plot the images in the batch, along with the corresponding labels
         fig = plt.figure(figsize=(25, 4))
         # display 20 images
         for idx in np.arange(2):
             ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
             imshow(images[idx])
```





Answer: Images are resized using RandomResizedCrop. Size 224 was pciked for input tensor by looking at the published articles online. That seemed the most appropriate size to take. Yes, the dataset was augmented by the techniques mentioned in the videos using RandomHorizontalFlip, RandomRotation.Image file was finally converted to tensor and normalized.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [14]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 # convolutional layer (sees 224x224x3 image tensor)
                 self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
                 # convolutional layer (sees 112x112x16 tensor)
                 self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
                 # convolutional layer (sees 64x64x32 tensor)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
```

```
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (128 * 28 * 28 -> 500)
        self.fc1 = nn.Linear(128 * 28 * 28, 500)
        # linear layer (500 -> 133)
        self.fc2 = nn.Linear(500, 133)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)
    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, 128 * 28 * 28)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: 3 Convulational layer and 2 fully connected layer CNN were designed after reading articles and blogs. Relu activation function used with pooling of 2. 1st layer gets input image of 224 *224* 3 dimensions (224 as that was the size chosen for resizing images) and has kernel size of (3,3), and 32 filtered images as output. 224 as image was resized to this dimension. 1st conv layer - 3 as input_channels (as 3 RBG depth) and 32 as output_channels as acc to this article/blog, 32 as input gave best accuracy. https://github.com/gsurma/image_classifier#performance In these articles,they are increaing output_channels in every layer, and that is why for second and third layer, output_channels are chosen as 64 and 128. Kernel size has been remained constant at (3,3). Pooling layer has been added after every CNN layer to decrease the dimensions. ReLU is used as this activation function sppeds up training in CNN. Pooling would downsample initial dimensions from 224 *224* 3 to 112 * 112 * 32. 2nd layer would take 32 images, with output of 64. Relu function is used and pooling would downsize dim to 56 * 56 * 64. 3rd layer would take 64 as input and 128 as output, and relu function and pooling would downsize dimension to 28 * 28 * 128. This would be input for fully connected layer 1 with output as 500 classes. 2nd fully connected layer would taske these 500 as inputs and would give 133 classes as output using relu function. 133 classes as that are the number of classes in dog classification.

Dropout is applied 0.25 to prevent 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 [15]: import torch.optim as optim

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

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

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

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 [35]: 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 loss variables for training loss 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()
                     ## update the model parameters
                         optimizer.zero_grad()
                         output = model(data)
                         loss = criterion(output, target)
                         loss.backward()
```

```
                optimizer.step()


            ## average training loss
                train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_l

        # 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()
            output = model(data)
            # calculate loss and update average validation loss
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

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

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


# train the model
model_scratch = train(50, loaders_scratch, model_scratch, optimizer_scratch, criterion_

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

```
Epoch: 1         Training Loss: 4.883908         Validation Loss: 4.867745
Validation loss decreased (inf --> 4.8677449226379395).  Saving model ...
Epoch: 2         Training Loss: 4.857948         Validation Loss: 4.828182
Validation loss decreased (4.8677449226379395 --> 4.828181743621826).  Saving model ...
Epoch: 3         Training Loss: 4.776867         Validation Loss: 4.714515
Validation loss decreased (4.828181743621826 --> 4.714515209197998).  Saving model ...
Epoch: 4         Training Loss: 4.653889         Validation Loss: 4.627423
Validation loss decreased (4.714515209197998 --> 4.62742280960083).  Saving model ...
Epoch: 5         Training Loss: 4.603976         Validation Loss: 4.595919
```

```
Validation loss decreased (4.62742280960083 --> 4.595919132232666).  Saving model ...
Epoch: 6        Training Loss: 4.562114         Validation Loss: 4.592135
Validation loss decreased (4.595919132232666 --> 4.592134952545166).  Saving model ...
Epoch: 7        Training Loss: 4.527866         Validation Loss: 4.517825
Validation loss decreased (4.592134952545166 --> 4.517824649810791).  Saving model ...
Epoch: 8        Training Loss: 4.497367         Validation Loss: 4.512098
Validation loss decreased (4.517824649810791 --> 4.51209831237793).  Saving model ...
Epoch: 9        Training Loss: 4.473951         Validation Loss: 4.484348
Validation loss decreased (4.51209831237793 --> 4.484348297119141).  Saving model ...
Epoch: 10       Training Loss: 4.437525         Validation Loss: 4.456133
Validation loss decreased (4.484348297119141 --> 4.456132888793945).  Saving model ...
Epoch: 11       Training Loss: 4.391824         Validation Loss: 4.421599
Validation loss decreased (4.456132888793945 --> 4.4215989112854).  Saving model ...
Epoch: 12       Training Loss: 4.370867         Validation Loss: 4.465789
Epoch: 13       Training Loss: 4.345278         Validation Loss: 4.371666
Validation loss decreased (4.4215989112854 --> 4.371665954589844).  Saving model ...
Epoch: 14       Training Loss: 4.327038         Validation Loss: 4.431526
Epoch: 15       Training Loss: 4.314827         Validation Loss: 4.408340
Epoch: 16       Training Loss: 4.274291         Validation Loss: 4.339738
Validation loss decreased (4.371665954589844 --> 4.339737892150879).  Saving model ...
Epoch: 17       Training Loss: 4.255634         Validation Loss: 4.365775
Epoch: 18       Training Loss: 4.236125         Validation Loss: 4.359012
Epoch: 19       Training Loss: 4.199792         Validation Loss: 4.341966
Epoch: 20       Training Loss: 4.168004         Validation Loss: 4.377005
Epoch: 21       Training Loss: 4.148744         Validation Loss: 4.310865
Validation loss decreased (4.339737892150879 --> 4.3108649253845215).  Saving model ...
Epoch: 22       Training Loss: 4.126297         Validation Loss: 4.322583
Epoch: 23       Training Loss: 4.105670         Validation Loss: 4.290994
Validation loss decreased (4.3108649253845215 --> 4.290994167327881).  Saving model ...
Epoch: 24       Training Loss: 4.078447         Validation Loss: 4.200477
Validation loss decreased (4.290994167327881 --> 4.200477123260498).  Saving model ...
Epoch: 25       Training Loss: 4.049450         Validation Loss: 4.226094
Epoch: 26       Training Loss: 4.030415         Validation Loss: 4.183950
Validation loss decreased (4.200477123260498 --> 4.183949947357178).  Saving model ...
Epoch: 27       Training Loss: 4.002006         Validation Loss: 4.247792
Epoch: 28       Training Loss: 3.970855         Validation Loss: 4.154358
Validation loss decreased (4.183949947357178 --> 4.15435791015625).  Saving model ...
Epoch: 29       Training Loss: 3.944093         Validation Loss: 4.135512
Validation loss decreased (4.15435791015625 --> 4.135512351989746).  Saving model ...
Epoch: 30       Training Loss: 3.925878         Validation Loss: 4.113176
Validation loss decreased (4.135512351989746 --> 4.113176345825195).  Saving model ...
Epoch: 31       Training Loss: 3.911366         Validation Loss: 4.118761
Epoch: 32       Training Loss: 3.874173         Validation Loss: 4.112432
Validation loss decreased (4.113176345825195 --> 4.112431526184082).  Saving model ...
Epoch: 33       Training Loss: 3.822076         Validation Loss: 4.083956
Validation loss decreased (4.112431526184082 --> 4.083956241607666).  Saving model ...
Epoch: 34       Training Loss: 3.807226         Validation Loss: 4.134215
Epoch: 35       Training Loss: 3.796986         Validation Loss: 4.182295
```

```
Epoch: 36         Training Loss: 3.798250         Validation Loss: 4.070650
Validation loss decreased (4.083956241607666 --> 4.07064962387085).  Saving model ...
Epoch: 37         Training Loss: 3.748355         Validation Loss: 4.115151
Epoch: 38         Training Loss: 3.737085         Validation Loss: 4.053328
Validation loss decreased (4.07064962387085 --> 4.053328037261963).  Saving model ...
Epoch: 39         Training Loss: 3.716938         Validation Loss: 4.136744
Epoch: 40         Training Loss: 3.678439         Validation Loss: 4.066415
Epoch: 41         Training Loss: 3.663081         Validation Loss: 4.055721
Epoch: 42         Training Loss: 3.651802         Validation Loss: 4.059011
Epoch: 43         Training Loss: 3.607549         Validation Loss: 4.217405
Epoch: 44         Training Loss: 3.590004         Validation Loss: 4.035468
Validation loss decreased (4.053328037261963 --> 4.035468101501465).  Saving model ...
Epoch: 45         Training Loss: 3.592422         Validation Loss: 4.007452
Validation loss decreased (4.035468101501465 --> 4.00745153427124).  Saving model ...
Epoch: 46         Training Loss: 3.565088         Validation Loss: 4.053574
Epoch: 47         Training Loss: 3.537142         Validation Loss: 4.098477
Epoch: 48         Training Loss: 3.493073         Validation Loss: 4.099118
Epoch: 49         Training Loss: 3.471225         Validation Loss: 4.001259
Validation loss decreased (4.00745153427124 --> 4.0012593269348145).  Saving model ...
Epoch: 50         Training Loss: 3.496581         Validation Loss: 3.980536
Validation loss decreased (4.0012593269348145 --> 3.9805357456207275).  Saving model ...
```

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 [36]:
```
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(f'Test Loss: {test_loss}')

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

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

```
Test Loss: 3.9477438926696777

Test Accuracy: 11% (97/836)
```

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 [16]: #Specify data loaders
         loaders_transfer = loaders_scratch.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [17]: import torchvision.models as models
         import torch.nn as nn

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

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

         model_transfer.fc = nn.Linear(2048, 133, bias=True)
```

```
model_fc_parameters = model_transfer.fc.parameters()

for param in model_fc_parameters:
    param.requires_grad = True

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:09<00:00, 10339879.07it/s]
```

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:

ResNet is chosen as this model was the winner of ImageNet challenge in 2015. It is a good starting point for transfer learning. Final fully connected layer ouput is 133 as thats the total classes of dog we have.

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 [18]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(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 [40]: 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 loss variables for 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()

            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 paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()

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

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

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

    # return trained model
    return model

n_epochs = 50

# train the model
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
                       use_cuda, 'model_transfer.pt')
```

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

```
Epoch: 1        Training Loss: 4.825168         Validation Loss: 4.703339
Validation loss decreased (inf --> 4.703339).  Saving model ...
Epoch: 2        Training Loss: 4.629292         Validation Loss: 4.499588
Validation loss decreased (4.703339 --> 4.499588).  Saving model ...
Epoch: 3        Training Loss: 4.458992         Validation Loss: 4.306858
Validation loss decreased (4.499588 --> 4.306858).  Saving model ...
Epoch: 4        Training Loss: 4.290055         Validation Loss: 4.156352
Validation loss decreased (4.306858 --> 4.156352).  Saving model ...
Epoch: 5        Training Loss: 4.129441         Validation Loss: 3.987661
Validation loss decreased (4.156352 --> 3.987661).  Saving model ...
Epoch: 6        Training Loss: 3.979967         Validation Loss: 3.838576
Validation loss decreased (3.987661 --> 3.838576).  Saving model ...
Epoch: 7        Training Loss: 3.828594         Validation Loss: 3.672453
Validation loss decreased (3.838576 --> 3.672453).  Saving model ...
Epoch: 8        Training Loss: 3.711260         Validation Loss: 3.463255
Validation loss decreased (3.672453 --> 3.463255).  Saving model ...
Epoch: 9        Training Loss: 3.566027         Validation Loss: 3.395427
Validation loss decreased (3.463255 --> 3.395427).  Saving model ...
Epoch: 10        Training Loss: 3.437756         Validation Loss: 3.253336
Validation loss decreased (3.395427 --> 3.253336).  Saving model ...
Epoch: 11        Training Loss: 3.337365         Validation Loss: 3.119250
Validation loss decreased (3.253336 --> 3.119250).  Saving model ...
Epoch: 12        Training Loss: 3.217595         Validation Loss: 3.015644
Validation loss decreased (3.119250 --> 3.015644).  Saving model ...
Epoch: 13        Training Loss: 3.130946         Validation Loss: 2.875268
Validation loss decreased (3.015644 --> 2.875268).  Saving model ...
Epoch: 14        Training Loss: 3.003729         Validation Loss: 2.834037
Validation loss decreased (2.875268 --> 2.834037).  Saving model ...
Epoch: 15        Training Loss: 2.914529         Validation Loss: 2.729851
Validation loss decreased (2.834037 --> 2.729851).  Saving model ...
Epoch: 16        Training Loss: 2.849252         Validation Loss: 2.645480
Validation loss decreased (2.729851 --> 2.645480).  Saving model ...
Epoch: 17        Training Loss: 2.767242         Validation Loss: 2.594935
Validation loss decreased (2.645480 --> 2.594935).  Saving model ...
Epoch: 18        Training Loss: 2.692084         Validation Loss: 2.504748
Validation loss decreased (2.594935 --> 2.504748).  Saving model ...
Epoch: 19        Training Loss: 2.613697         Validation Loss: 2.431084
Validation loss decreased (2.504748 --> 2.431084).  Saving model ...
Epoch: 20        Training Loss: 2.542523         Validation Loss: 2.380444
Validation loss decreased (2.431084 --> 2.380444).  Saving model ...
Epoch: 21        Training Loss: 2.483855         Validation Loss: 2.281067
Validation loss decreased (2.380444 --> 2.281067).  Saving model ...
Epoch: 22        Training Loss: 2.425716         Validation Loss: 2.238971
Validation loss decreased (2.281067 --> 2.238971).  Saving model ...
```

```
Epoch: 23        Training Loss: 2.371985        Validation Loss: 2.116541
Validation loss decreased (2.238971 --> 2.116541).  Saving model ...
Epoch: 24        Training Loss: 2.325711        Validation Loss: 2.174445
Epoch: 25        Training Loss: 2.272822        Validation Loss: 2.090773
Validation loss decreased (2.116541 --> 2.090773).  Saving model ...
Epoch: 26        Training Loss: 2.222103        Validation Loss: 2.046500
Validation loss decreased (2.090773 --> 2.046500).  Saving model ...
Epoch: 27        Training Loss: 2.186184        Validation Loss: 1.997470
Validation loss decreased (2.046500 --> 1.997470).  Saving model ...
Epoch: 28        Training Loss: 2.143927        Validation Loss: 1.871257
Validation loss decreased (1.997470 --> 1.871257).  Saving model ...
Epoch: 29        Training Loss: 2.100025        Validation Loss: 1.967346
Epoch: 30        Training Loss: 2.066503        Validation Loss: 1.874444
Epoch: 31        Training Loss: 2.046357        Validation Loss: 1.835734
Validation loss decreased (1.871257 --> 1.835734).  Saving model ...
Epoch: 32        Training Loss: 2.016487        Validation Loss: 1.758526
Validation loss decreased (1.835734 --> 1.758526).  Saving model ...
Epoch: 33        Training Loss: 1.972798        Validation Loss: 1.775624
Epoch: 34        Training Loss: 1.920489        Validation Loss: 1.800687
Epoch: 35        Training Loss: 1.896557        Validation Loss: 1.716136
Validation loss decreased (1.758526 --> 1.716136).  Saving model ...
Epoch: 36        Training Loss: 1.872892        Validation Loss: 1.726189
Epoch: 37        Training Loss: 1.849016        Validation Loss: 1.662001
Validation loss decreased (1.716136 --> 1.662001).  Saving model ...
Epoch: 38        Training Loss: 1.848487        Validation Loss: 1.664742
Epoch: 39        Training Loss: 1.798081        Validation Loss: 1.596139
Validation loss decreased (1.662001 --> 1.596139).  Saving model ...
Epoch: 40        Training Loss: 1.780123        Validation Loss: 1.637440
Epoch: 41        Training Loss: 1.762377        Validation Loss: 1.602790
Epoch: 42        Training Loss: 1.726585        Validation Loss: 1.523098
Validation loss decreased (1.596139 --> 1.523098).  Saving model ...
Epoch: 43        Training Loss: 1.713249        Validation Loss: 1.517028
Validation loss decreased (1.523098 --> 1.517028).  Saving model ...
Epoch: 44        Training Loss: 1.726695        Validation Loss: 1.531536
Epoch: 45        Training Loss: 1.684299        Validation Loss: 1.506107
Validation loss decreased (1.517028 --> 1.506107).  Saving model ...
Epoch: 46        Training Loss: 1.645051        Validation Loss: 1.518529
Epoch: 47        Training Loss: 1.653809        Validation Loss: 1.527501
Epoch: 48        Training Loss: 1.630452        Validation Loss: 1.549457
Epoch: 49        Training Loss: 1.603473        Validation Loss: 1.483806
Validation loss decreased (1.506107 --> 1.483806).  Saving model ...
Epoch: 50        Training Loss: 1.599444        Validation Loss: 1.427736
Validation loss decreased (1.483806 --> 1.427736).  Saving model ...
```

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 [41]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

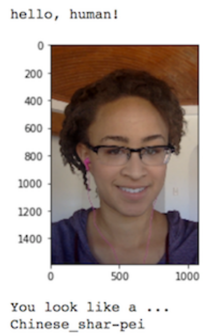
```
Test Loss: 1.4854999780654907


Test Accuracy: 69% (583/836)
```

In [19]:
```
# model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

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 [20]:
```
from PIL import Image
import torchvision.transforms as transforms

data_transfer = loaders_transfer.copy()

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].dataset.cl

def predict_breed_transfer(img_path):
    ### a function that takes a path to an image as input
    ### and returns the dog breed that is predicted by the model.

    global model_transfer
    global test_data_transform

    # load the image and return the predicted breed
    image = Image.open(img_path).convert('RGB')

    # Removing transparent, alpha
    image = test_data_transform(image)[:3,:,:].unsqueeze(0)

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

    model_transfer.eval()
    idx = torch.argmax(model_transfer(image))
    return class_names[idx]
```



Sample Human Output

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 [21]: def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             if face_detector(img_path) is True:
                 breed = predict_breed_transfer(img_path)
                 print('Human / resembling dog breed is ' + breed)
             elif dog_detector(img_path):
                 breed = predict_breed_transfer(img_path)
                 print('Dog / predicted dog breed is ' + breed)
             else:
                 print('Not Human, Neither Dog')
```

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.

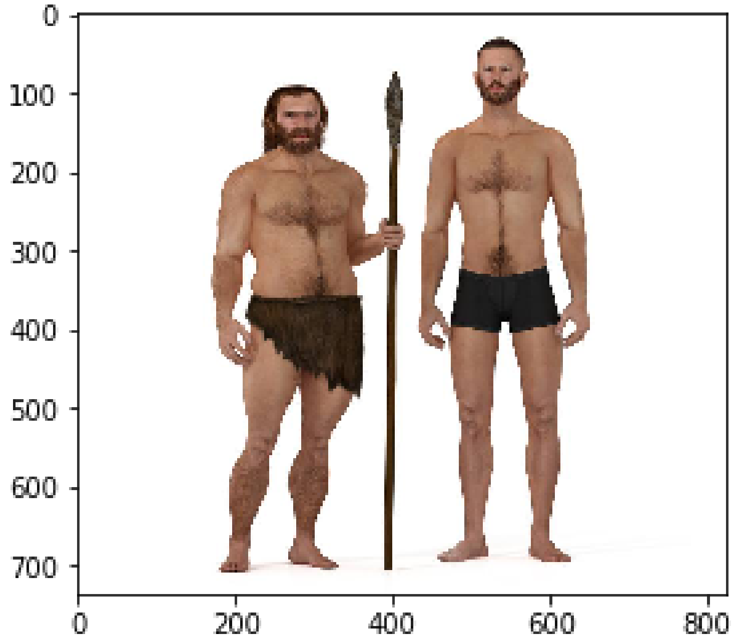
Answer:

The output is as I expected but definitely could be improved further. Cave man reembles bullmastiff and it predicted that right. Couldnt resemble a running lady but the face is not clear enough, so that might be the rerson for model to not work well. It also couldn't resemble an old man that might be but the face is little hairy that could be the reason. One pic with both monkey and human face is predicted right that it is neither human nor dog. I put 2 wolf pics and it predicted exactly right that they are not dogs. Out of 2 cat pics, it resembled that one is not dog or human and other cat looks like bulldog and so it predicted it wrong as a bulldog but resemblance was right. Other dog pics, it has predicted exactly right.

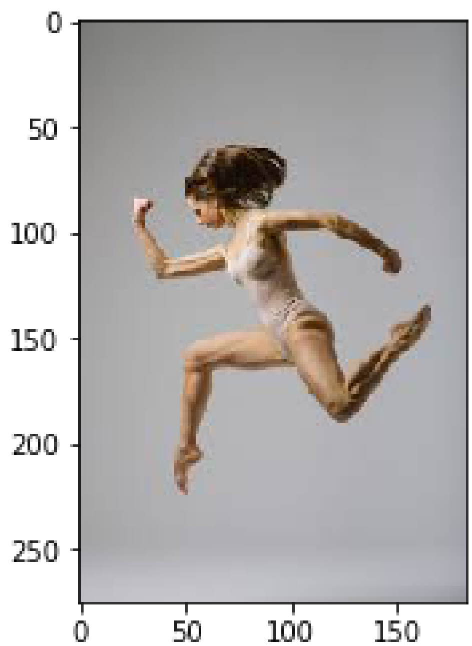
Three possible points for improvement: 1. More datapoints for each dog class would improve accuracy. 2. Better transfer learning models may be more layers 3. More epochs is increasing score defintely but compuationally expensive 4. Playing with different layers output

```
In [24]: ## 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.
         import os

         for img_file in os.listdir('human_direct/'):
             img_path = os.path.join('human_direct', img_file)
             run_app(img_path)
             print(img_path)
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
```

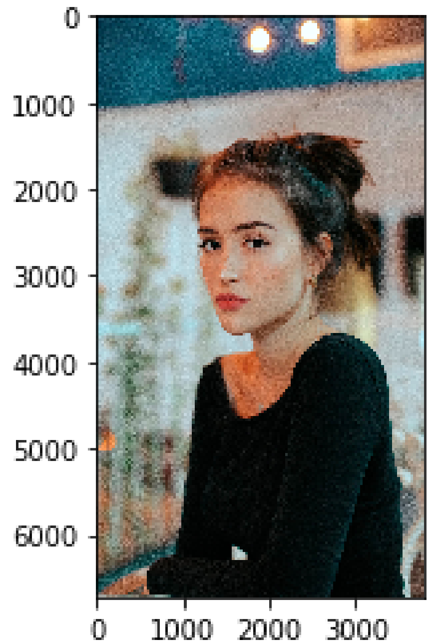
```
Human / resembling dog breed is Bullmastiff
human_direct/human3.jpg
```



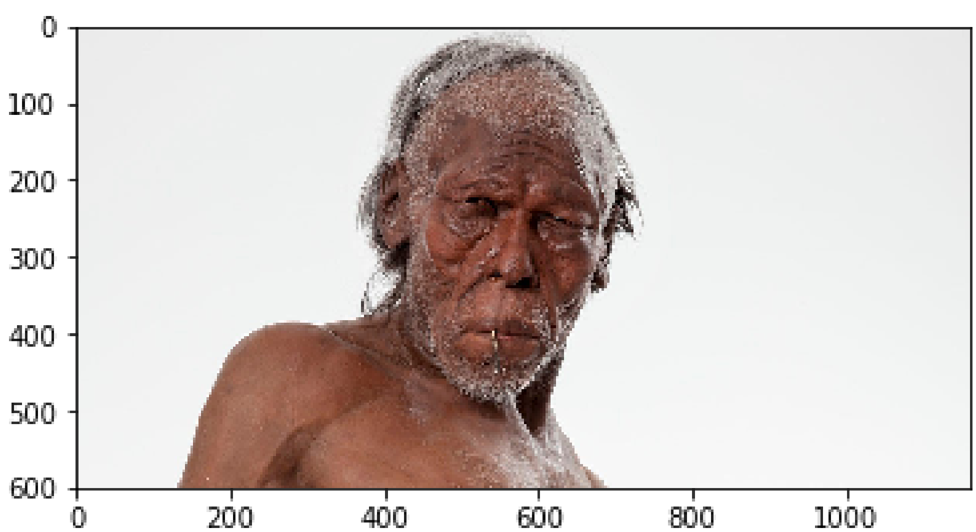
Not Human, Neither Dog
human_direct/human7.jpeg



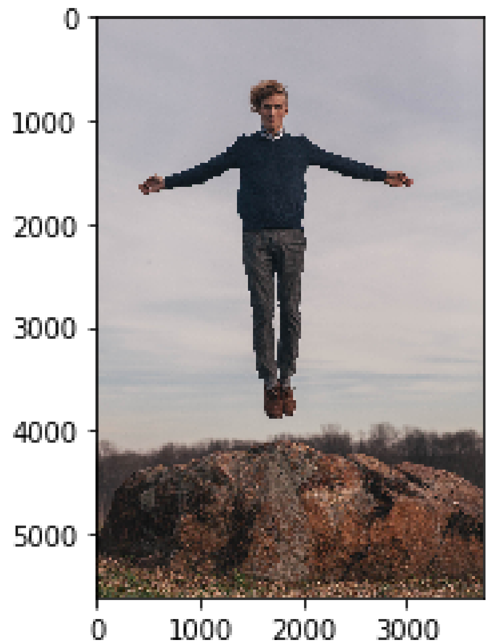
```
Human / resembling dog breed is Maltese
human_direct/human1.jpg
```



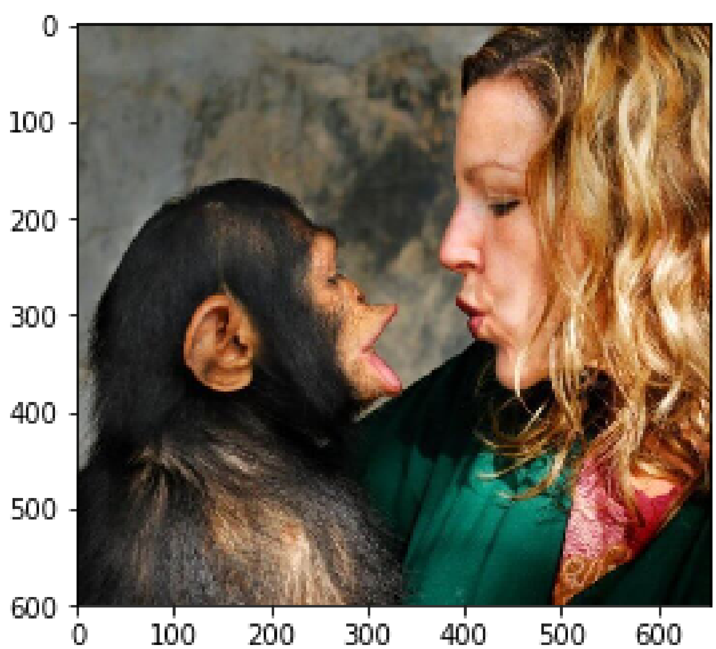

```
Not Human, Neither Dog
human_direct/human4.png
```



```
Human / resembling dog breed is Greyhound
human_direct/human2.jpg
```



```
Not Human, Neither Dog
human_direct/human6.jpg
```



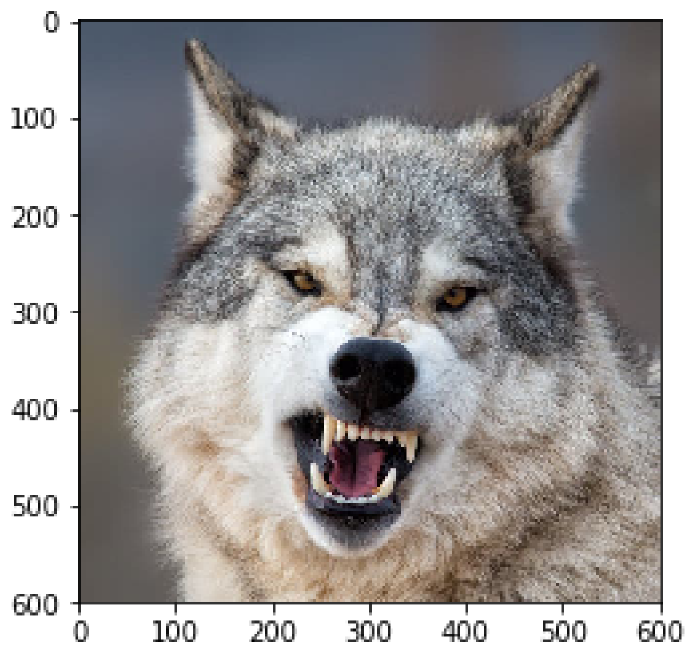
```
Human / resembling dog breed is Dachshund
human_direct/human8.jpeg
```

```
---------------------------------------------------------------------------

error                                     Traceback (most recent call last)

<ipython-input-24-cfcc2dba83cb> in <module>()
      6 for img_file in os.listdir('human_direct/'):
      7     img_path = os.path.join('human_direct', img_file)
----> 8     run_app(img_path)
      9     print(img_path)
     10     img = Image.open(img_path)


<ipython-input-21-b3970efa07fe> in run_app(img_path)
      2 def run_app(img_path):
      3     ## handle cases for a human face, dog, and neither
----> 4     if face_detector(img_path) is True:
      5         breed = predict_breed_transfer(img_path)
      6         print('Human / resembling dog breed is ' + breed)
```

```
<ipython-input-3-6b99dca2dc38> in face_detector(img_path)
  2 def face_detector(img_path):
  3     img = cv2.imread(img_path)
----> 4     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  5     human_faces = face_cascade.detectMultiScale(gray)
  6     return len(human_faces) > 0


error: /tmp/build/80754af9/opencv_1512491964794/work/modules/imgproc/src/color.cpp:11048
```

```
In [25]: for img_file in os.listdir('dog_direct/'):
             img_path = os.path.join('dog_direct', img_file)
             run_app(img_path)
             print(img_path)
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
```

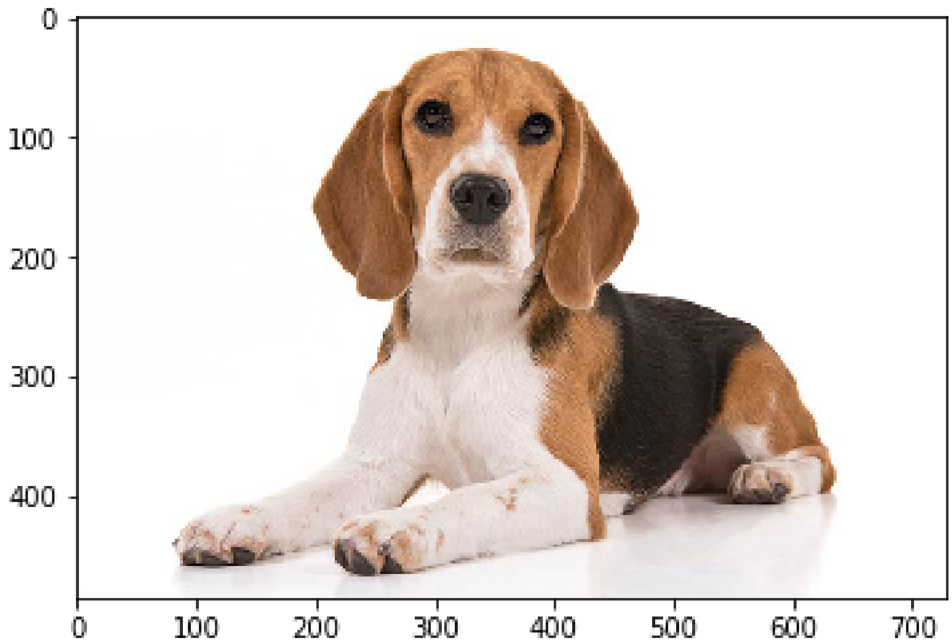
```
Not Human, Neither Dog
dog_direct/merlin_148630314_4eca699a-e353-4dd5-9607-2f56c0a61674-articleLarge.jpg
```



```
Dog / predicted dog breed is Norwich terrier
dog_direct/images (1).jpeg
```

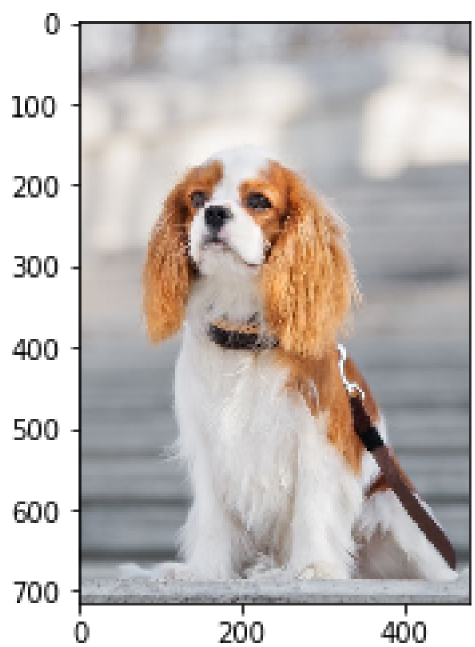

```
Dog / predicted dog breed is Beauceron
dog_direct/GettyImages-521536928-_1_.jpg
```

```
Dog / predicted dog breed is Beagle
dog_direct/Beagle-On-White-07.jpg
```

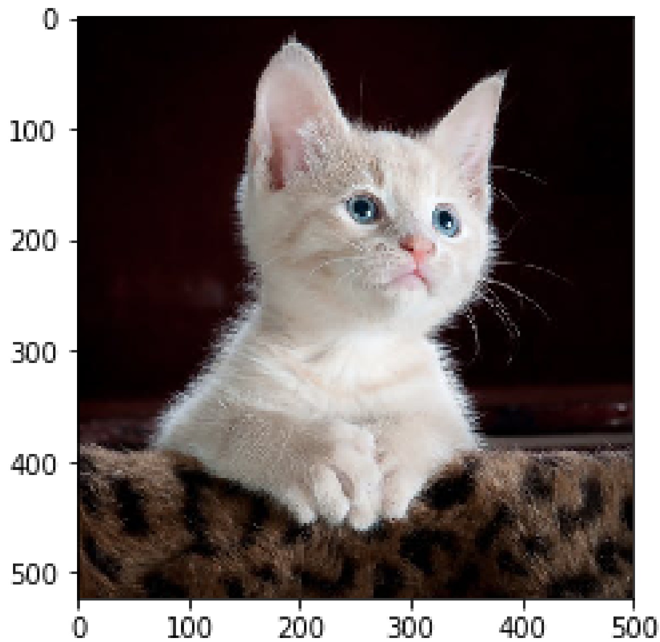



```
Human / resembling dog breed is French bulldog
dog_direct/02-cat-training-NationalGeographic_1484324.jpg
```

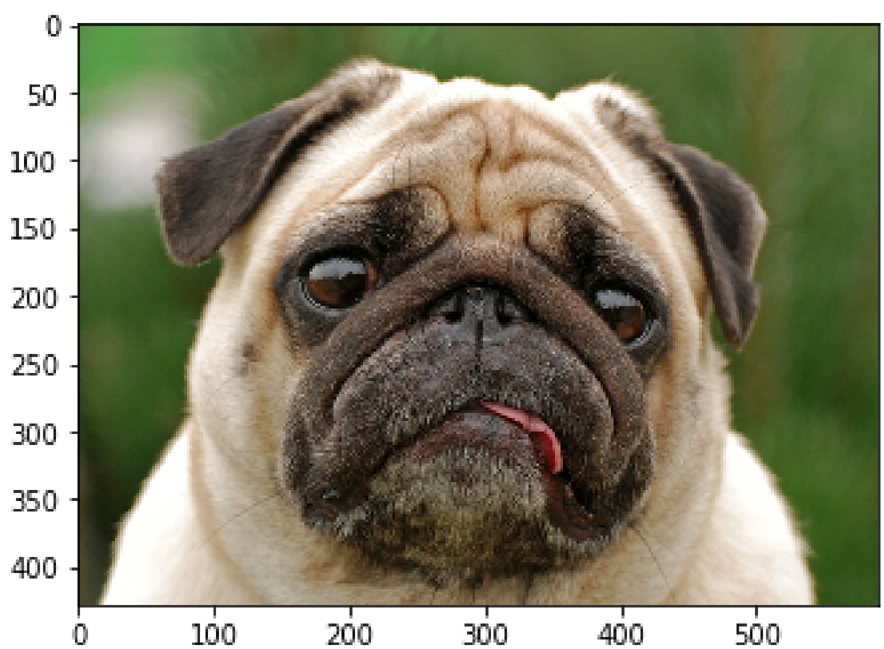

```
Dog / predicted dog breed is Cavalier king charles spaniel
dog_direct/adorable-cavalier-king-charles-spaniel-puppy-royalty-free-image-523255012-1565106446.
```



```
Not Human, Neither Dog
dog_direct/kitty-cat-kitten-pet-45201.jpeg
```



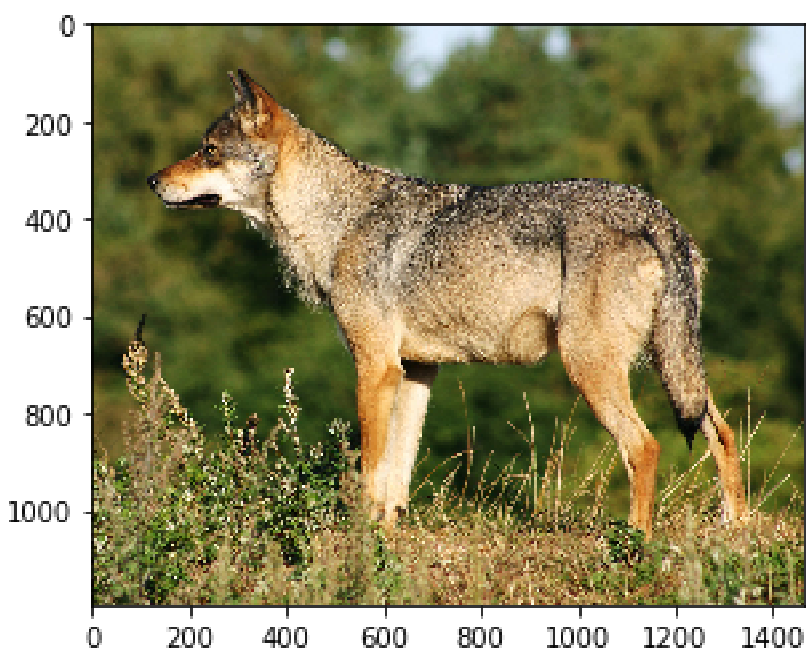

```
Dog / predicted dog breed is Bulldog
dog_direct/Acute-Dog-Diarrhea-47066074.jpg
```



```
Human / resembling dog breed is Labrador retriever
dog_direct/puppy-1903313__340.jpg
```


```
Dog / predicted dog breed is Great dane
dog_direct/images.jpeg
```


```
Not Human, Neither Dog
dog_direct/Scandinavian_grey_wolf_Canis_lupus_(cropped).jpg
```



In [ ]: