dog_app

March 1, 2021

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
        import pandas as pd

        # 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

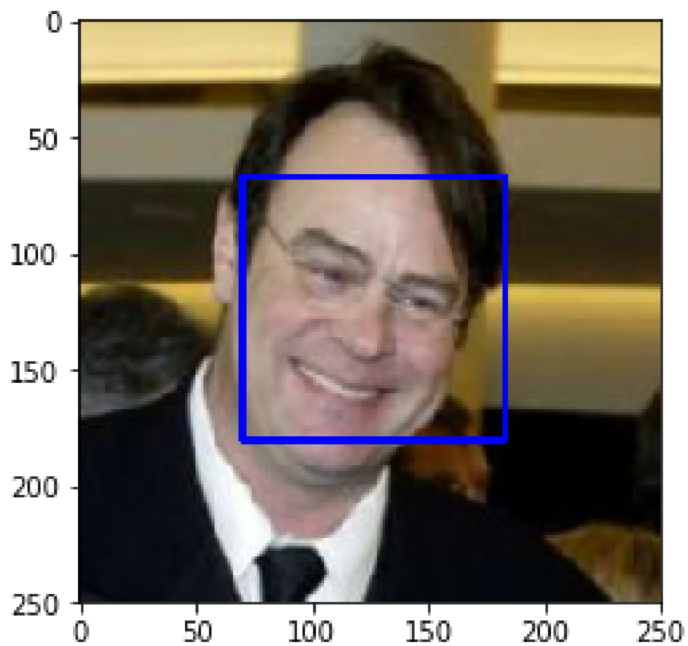
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [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
```

```python
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 [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)
            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)

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        human_count = 0
        dog_count = 0

        for img in human_files_short:
            if face_detector(img): human_count += 1

        for img in dog_files_short:
            if face_detector(img): dog_count += 1

        print("Human faces detected in human_files_short:",human_count)
        print("Humans faces detected in dog_files_short:",dog_count)
```

```
Human faces detected in human_files_short: 98
Humans faces detected in dog_files_short: 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.

def face_detector_ext(img_path):
    '''
    Using CascaseClassifier from cv2 to detect human face in an image
    Classifier works with haarcascade-file 'haarcascade_frontalface_alt2.xml'
    Args:
        img_path: path of an image
    Returns:
        True, if a human face is present
        False, otherwise
    '''
    # extract pre-trained face detector using haarcascade_frontalface_alt2.xml
    face_cascade_ex = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt2.x

    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade_ex.detectMultiScale(gray)
    return len(faces) > 0
```

In [6]:
```
human_count_ext = 0
dog_count_ext = 0

for img in human_files_short:
    if face_detector_ext(img): human_count_ext += 1

for img in dog_files_short:
    if face_detector_ext(img): dog_count_ext += 1

print("Human faces detected in human_files_short:", human_count_ext)
print("Humans faces detected in dog_files_short:", dog_count_ext)
```

```
Human faces detected in human_files_short: 100
Humans faces detected in dog_files_short: 21
```

Performance on human_files_short and dog_files_short for haarcascade_frontalface_alt.xml and haarcascade_frontalface_alt2.xml is as follows:

Using haarcascade_frontalface_alt.xml: Percentage of the detected human face in human_files_short: 98 Percentage of the detected human face in dog_files_short: 17

Using haarcascade_frontalface_alt2.xml: Human faces detected in human_files_short: 100 Humans faces detected in dog_files_short: 21

As evident from results, haarcascade_frontalface_alt2 performs better with 100% accuracy on human_files_short for human face detection and detected more human faces in dog_files_short.

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:09<00:00, 58626205.05it/s]
```

```
In [8]: # Check if cuda is available and get device
        print('Cuda available:', use_cuda)
```

```
Cuda available: True
```

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 [9]: from PIL import Image
        import torchvision.transforms as 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
            ## Return the *index* of the predicted class for that image

            #Define the image transformations
            transform = transforms.Compose([transforms.Resize((224,224)), # VGG expects 224x224
                                            transforms.ToTensor(), #image to Tensor data type co
                                            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                 std=[0.229, 0.224, 0.225]) #ima
            ])

            #Load image
            img = Image.open(img_path)
            #Transform image
            img = transform(img)
            #Flatten the tensor
            img = img.unsqueeze(0)

            #If cuda then convert to cuda data type
            if use_cuda:
                img = img.cuda()
            #Get the prediction
            prediction = VGG16(img)

            #Get maximum value and its index from prediction metrix
            _,ind = torch.max(prediction,1)

            return ind.item() # predicted class index

In [10]: #Test VGG16_predict
         predict = VGG16_predict(dog_files_short[0])
         print(predict)

243
```

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 [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             '''
             This function is used to predict if dog is present in an image or not
             Args:
                 img_path: path to an image
             Returns: True if a dog is detected, False otherwise
             '''
             ind = VGG16_predict(img_path)
             return ind >=151 and ind <=268 # true/false
```

```
In [12]: #Test dog_detector function with first image from dog_files_short and human_files_short
         print("Is first image from dog_files_short a dog image?:",dog_detector(dog_files_short[
         print("Is first image from human_files_short a dog image?",dog_detector(human_files_sho
```

```
Is first image from dog_files_short a dog image?: True
Is first image from human_files_short a dog image? 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:

```
In [13]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_count = 0
         dog_count = 0

         for img in human_files_short:
             if dog_detector(img): human_count += 1

         for img in dog_files_short:
             if dog_detector(img): dog_count += 1
```

```
        print("Number of dogs detected in human_files_short (VGG16):",human_count)
        print("Number of dogs detected in dog_files_short (VGG16):",dog_count)
```

```
Number of dogs detected in human_files_short (VGG16): 0
Number of dogs detected in dog_files_short (VGG16): 100
```

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

         #Pretrained Inception-v3 model

         # define inception v3 model
         inception = models.inception_v3(pretrained=True)

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

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

         inception.eval()
```

```
Downloading: "https://download.pytorch.org/models/inception_v3_google-1a9a5a14.pth" to /root/.to
100%|| 108857766/108857766 [00:01<00:00, 62106075.32it/s]
```

```
Out[14]: Inception3(
           (Conv2d_1a_3x3): BasicConv2d(
             (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), bias=False)
             (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=Tru
           )
           (Conv2d_2a_3x3): BasicConv2d(
             (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
             (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=Tru
           )
           (Conv2d_2b_3x3): BasicConv2d(
             (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fals
             (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=Tru
           )
           (Conv2d_3b_1x1): BasicConv2d(
```

```
    (conv): Conv2d(64, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(80, eps=0.001, momentum=0.1, affine=True, track_running_stats=Tru
  )
  (Conv2d_4a_3x3): BasicConv2d(
    (conv): Conv2d(80, 192, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=Tr
  )
  (Mixed_5b): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=Fa
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(192, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
  )
  (Mixed_5c): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=Fa
```

```
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
  )
  (Mixed_5d): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(288, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=Fa
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
    )
```

```
)
(Mixed_6a): InceptionB(
  (branch3x3): BasicConv2d(
    (conv): Conv2d(288, 384, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch3x3dbl_1): BasicConv2d(
    (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
  )
  (branch3x3dbl_2): BasicConv2d(
    (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
    (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
  )
  (branch3x3dbl_3): BasicConv2d(
    (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=T
  )
)
(Mixed_6b): InceptionC(
  (branch1x1): BasicConv2d(
    (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7_1): BasicConv2d(
    (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7_2): BasicConv2d(
    (conv): Conv2d(128, 128, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7_3): BasicConv2d(
    (conv): Conv2d(128, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_1): BasicConv2d(
    (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_2): BasicConv2d(
    (conv): Conv2d(128, 128, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_3): BasicConv2d(
    (conv): Conv2d(128, 128, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
```

```
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(128, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (Mixed_6c): InceptionC(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7_3): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_3): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
```

```
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (Mixed_6d): InceptionC(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7_3): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_3): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
```

```
)
(Mixed_6e): InceptionC(
  (branch1x1): BasicConv2d(
    (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7_1): BasicConv2d(
    (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7_2): BasicConv2d(
    (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7_3): BasicConv2d(
    (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_1): BasicConv2d(
    (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_2): BasicConv2d(
    (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_3): BasicConv2d(
    (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_4): BasicConv2d(
    (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch7x7dbl_5): BasicConv2d(
    (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
  (branch_pool): BasicConv2d(
    (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
)
(AuxLogits): InceptionAux(
  (conv0): BasicConv2d(
    (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=
  )
```

```
    (conv1): BasicConv2d(
      (conv): Conv2d(128, 768, kernel_size=(5, 5), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(768, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (fc): Linear(in_features=768, out_features=1000, bias=True)
  )
  (Mixed_7a): InceptionD(
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3_2): BasicConv2d(
      (conv): Conv2d(192, 320, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7x3_1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7x3_2): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7x3_3): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch7x7x3_4): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (Mixed_7b): InceptionE(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(1280, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(1280, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3_2a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3_2b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
```

```
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(1280, 448, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(448, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(448, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3dbl_3a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3dbl_3b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(1280, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (Mixed_7c): InceptionE(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(2048, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(2048, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3_2a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3_2b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(2048, 448, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(448, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(448, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
    )
    (branch3x3dbl_3a): BasicConv2d(
```

```
          (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=
          (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
        )
        (branch3x3dbl_3b): BasicConv2d(
          (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=
          (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=
        )
        (branch_pool): BasicConv2d(
          (conv): Conv2d(2048, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=
        )
      )
      (fc): Linear(in_features=2048, out_features=1000, bias=True)
    )
```

In [15]:
```
# Check if cuda is available and get device
print('Cuda available:', torch.cuda.is_available())
```

```
Cuda available: True
```

In [16]:
```
def INCEPTION_predict(img_path):
    '''
    Use pre-trained Inception v3 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 Inception v3 model's prediction
    '''

    # Normalizing the image with specific mean and standard deviation
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])
    # In contrast to the other models the inception_v3 expects tensors with a size of N

    preprocess = transforms.Compose([
        transforms.Resize(299),
        transforms.CenterCrop(299),
        transforms.ToTensor(),
        normalize,
    ])

    input_image = Image.open(img_path) # Load image
    input_tensor = preprocess(input_image) # Transform image
    input_batch = input_tensor.unsqueeze(0) # create a mini-batch as expected by the mo
```

```
    # move the input and model to GPU for speed if available
    if torch.cuda.is_available():
        input_batch = input_batch.to('cuda')

    with torch.no_grad():
        output = inception(input_batch) # Get the prediction of the model


    # Tensor of shape 1000, with confidence scores over Imagenet's 1000 classes
    #print(output[0])

    # The output has unnormalized scores. To get probabilities, you can run a softmax o
    #predictions = torch.nn.functional.softmax(output[0], dim=0)

    # Get the max-Value of the Tensor-matrix and return as integer
    _, index = torch.max(output, 1)

    return index.item() # predicted class index
```

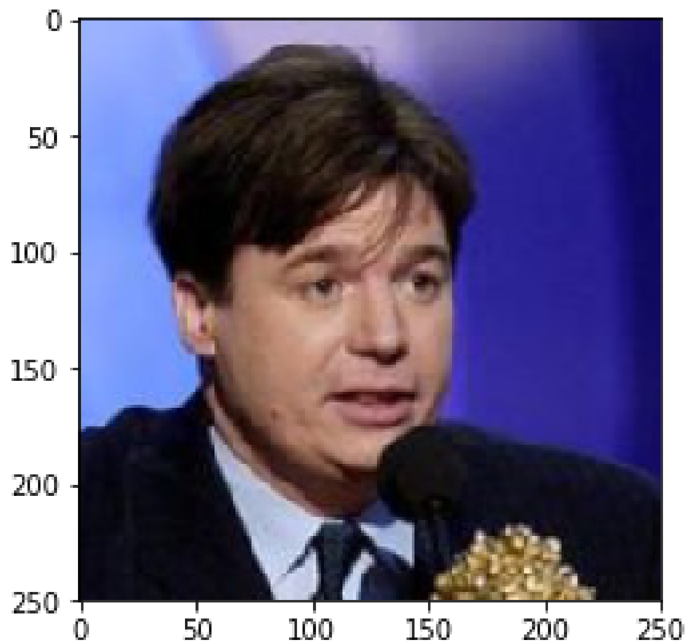
In [17]:
```
#Test INCEPTION_predict
predict = INCEPTION_predict(dog_files_short[0])
print(predict)
```

243

In [18]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector_inc(img_path):
    ## TODO: Complete the function.
    '''
    This function is used to predict if dog is present in an image or not
    Args:
        img_path: path to an image
    Returns: True if a dog is detected, False otherwise
    '''
    ind = INCEPTION_predict(img_path)
    return ind >=151 and ind <=268 # true/false
```

In [19]:
```
### Test the performance of the dog_detector function using Inception v3 model
### on the images in human_files_short and dog_files_short.

import matplotlib.image as mpimg


human_count = 0
dog_count = 0

for img in human_files_short:
```

```
        if dog_detector_inc(img):
            human_count += 1
            img = mpimg.imread(img)
            plt.imshow(img)
            plt.show()

    for img in dog_files_short:
        if dog_detector_inc(img): dog_count += 1

    print("Number of dogs detected in human_files_short (Inception v3):",human_count)
    print("Number of dogs detected in dog_files_short (Inception v3):",dog_count)
```




```
Number of dogs detected in human_files_short (Inception v3): 1
Number of dogs detected in dog_files_short (Inception v3): 100
```

Performance on human_files_short and dog_files_short for VGG16 and Inception V3 is as follows:

Using VGG16: Number of dogs detected in human_files_short (VGG16): 0 Number of dogs detected in dog_files_short (VGG16): 100

Using Inception V3: Number of dogs detected in human_files_short (Inception v3): 1 Number of dogs detected in dog_files_short (Inception v3): 100

As evident from results, both the models VGG16 and Inception V3 performs similar on dog_files_short with 100% accuracy whereas Inception V3 had a false positive and detected a dog in human_files_short.

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 [20]:
```
import os
from torchvision import datasets

from PIL import Image
```

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

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir,'train/')
valid_dir = os.path.join(data_dir,'valid/')
test_dir = os.path.join(data_dir,'test/')
```

1.1.8 Define Data Augementation using transforms

In [21]:
```
import torchvision.transforms as transforms
# Normalizing the image with specific mean and standard deviation
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

train_transform = transforms.Compose([transforms.Resize(256),           # Resize the i
                                      transforms.RandomResizedCrop(224), # Crop the ima
                                      transforms.RandomHorizontalFlip(), # Horizontally
                                      transforms.RandomRotation(10),     # Rotate the i
                                      transforms.ToTensor(),             # Convert the
                                      normalize])                        # Normalize us
valid_transform = transforms.Compose([transforms.Resize(256),           # Resize the i
                                      transforms.CenterCrop(224),       # Crop the ima
                                      transforms.ToTensor(),             # Convert the
                                      normalize])                        # Normalize us
test_transform = transforms.Compose([transforms.Resize(256),            # Resize the i
                                      transforms.CenterCrop(224),       # Crop the ima
                                      transforms.ToTensor(),             # Convert the
                                      normalize])                        # Normalize us
```

1.1.9 Define Data Loaders

In [22]:
```
# Set Batch size and number of workers
batch_size = 20
num_workers = 0

# Instantiate generic data loaders where the images are arranged in a certain way
train_data = datasets.ImageFolder(train_dir, transform=train_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=valid_transform)
test_data = datasets.ImageFolder(test_dir, transform=test_transform)

# Set data loaders for training, validation and testing
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
test_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=Tr
```

```
# Put data loaders to a dictionary
loaders_scratch = {"train" : train_loader, "valid" : valid_loader, "test" : test_loader
```

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

Answer: Image resizing is done using transforms such as Resize (256), RandomResizeCrop (224), and RandomHorizontalFlip. RandomResizeCrop (224), RandomHorizontalFlip and RandomRotation (10 degree) are only applied on the train_data to improve model performance using data augmentation and prevent overfitting.

On validation and test data sets, I have only applied Resize of 256 and CenterCrop to covert image to 224x224 size. 224x224 pixels size is selected to compare model performance against ResNet50 model which expects input size to be 224 x 224 x 3.

Data augmentation is not applied on validation and test data set as these will be used to validate and test our model performance.

1.1.10 (IMPLEMENTATION) Model Architecture

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

In [23]:
```
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
        self.conv1 = nn.Conv2d(3,32,3,stride=1,padding=1)
        self.conv2 = nn.Conv2d(32,64,3,stride=1,padding=1)
        self.conv3 = nn.Conv2d(64,128,3,stride=1,padding=1)
        self.conv4 = nn.Conv2d(128,128,3,stride=1,padding=1)

        #Pooling
        self.pool = nn.MaxPool2d(2,2)

        #Full connected layers
        self.fc1 = nn.Linear(14*14*128,4096)
        self.fc2 = nn.Linear(4096,133)

        #drop-out layer
        self.dropout = nn.Dropout(0.25)

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

```
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = F.relu(self.conv4(x))
        x = self.pool(x)

        #flatten
        x = x.view(-1,14*14*128)

        #drop out
        x = self.dropout(x)

        #linear
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

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

# instantiate the CNN
model_scratch = Net()
print(model_scratch)
# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 128, 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=25088, out_features=4096, bias=True)
  (fc2): Linear(in_features=4096, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

Answer: High level final CNN architecture is as follows:

For CNN model from scratch, I followed standard CNN architecture having convolution layer,pooling layer, fully connected layers and drop out layers. I defined the forward behavior with four convolution layers, pooling layer after each convolution layer to reduce features by half and finally added two linear layers and dropout layers with probabality of 25%.

Step by step forward pass for the model is as follows: 1. First Convolutional Layer: 3 inputs, 32 outputs, 3x3 kernel - 224 x 224 x 3: 224x224 for image size, 3 for RGB 2. Relu Activation function: - for non-linearity and better performance over tanh or sigmoid 3. Pooling layer: 2x2 kernel - reduces the dimensionality of each map and retaining important information 4. Second Convolutional Layer: 32 inputs, 64 outputs,3x3 kernel 5. Relu Activation function 6. Pooling layer: 2x2 kernel 7. Third Convolutional Layer: 64 inputs, 128 outputs, 3x3 kernel 8. Relu Activation function 9. Pooling layer: 2x2 kernel 10. Fourth Convolutional Layer: 128 inputs, 128 outputs, 3x3 kernel 11. Relu Activation function 12. Pooling layer: 2x2 kernel 13. Flatten layer: 25088 length single vector - fully connected layer expects vector inputs 14. Dropout layer: 25% probability - prevent overfitting 15. First Fully connected Linear Layer: 25088 inputs, 4096 outputs 16. Relu Activation function 17. Dropout layer: 25% probability 18. Second Fully connected Linear Layer: 4096 inputs and 133 outputs (number of dog breeds)

1.1.11 (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 [24]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(),lr=0.02)
```

1.1.12 (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 [25]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True #truncated image file handling

         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             '''
             This function is responsible for training the model

             Args:
                 n_epochs:    Number of epochs to train
                 loaders:     Dictionary of the definded dataloaders
                 model:       Model for training
                 optimizer:   Optimizer
                 criterion:   Loss Function
                 use_cuda:    True if GPU is used, False if CPU is used
                 save_path:   Saving path to save the model
             Returns:
```

```
    Returns trained model and pandas dataframe with train and validation losses
'''
# initialize tracker for minimum validation loss
valid_loss_min = np.Inf


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

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

        # clearing the Gradients of the model parameters
        optimizer.zero_grad()

        # prediction for training set
        output = model(data)

        # computing the training loss
        loss_train = criterion(output, target)

        # Backward pass to compute gradients for the model parameters
        loss_train.backward()

        #optmization step to update parameters
        optimizer.step()

        train_loss += ((1/(batch_idx+1))* (loss_train.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

                # prediction for validation set
                output = model(data)

                # computing the validation loss
                loss_valid = criterion(output, target)

                valid_loss += ((1/(batch_idx+1))* (loss_valid.data - valid_loss))


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



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

        # return trained model
        return model
```

```
In [12]: # train the model
         model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1        Training Loss: 4.873483         Validation Loss: 4.823107
Validation loss has decreased (inf --> 4.823107).  Saving model.
Epoch: 2        Training Loss: 4.796203         Validation Loss: 4.721483
Validation loss has decreased (4.823107 --> 4.721483).  Saving model.
Epoch: 3        Training Loss: 4.725484         Validation Loss: 4.719444
Validation loss has decreased (4.721483 --> 4.719444).  Saving model.
Epoch: 4        Training Loss: 4.626395         Validation Loss: 4.526433
Validation loss has decreased (4.719444 --> 4.526433).  Saving model.
Epoch: 5        Training Loss: 4.550735         Validation Loss: 4.473731
Validation loss has decreased (4.526433 --> 4.473731).  Saving model.
Epoch: 6        Training Loss: 4.522107         Validation Loss: 4.453767
Validation loss has decreased (4.473731 --> 4.453767).  Saving model.
Epoch: 7        Training Loss: 4.477876         Validation Loss: 4.397085
Validation loss has decreased (4.453767 --> 4.397085).  Saving model.
Epoch: 8        Training Loss: 4.420782         Validation Loss: 4.338241
```

```
Validation loss has decreased (4.397085 --> 4.338241).  Saving model.
Epoch: 9        Training Loss: 4.373080         Validation Loss: 4.294878
Validation loss has decreased (4.338241 --> 4.294878).  Saving model.
Epoch: 10       Training Loss: 4.350490         Validation Loss: 4.241675
Validation loss has decreased (4.294878 --> 4.241675).  Saving model.
Epoch: 11       Training Loss: 4.287039         Validation Loss: 4.217596
Validation loss has decreased (4.241675 --> 4.217596).  Saving model.
Epoch: 12       Training Loss: 4.233805         Validation Loss: 4.120996
Validation loss has decreased (4.217596 --> 4.120996).  Saving model.
Epoch: 13       Training Loss: 4.208917         Validation Loss: 4.155601
Epoch: 14       Training Loss: 4.153142         Validation Loss: 4.072165
Validation loss has decreased (4.120996 --> 4.072165).  Saving model.
Epoch: 15       Training Loss: 4.106969         Validation Loss: 3.978069
Validation loss has decreased (4.072165 --> 3.978069).  Saving model.
Epoch: 16       Training Loss: 4.038295         Validation Loss: 3.910873
Validation loss has decreased (3.978069 --> 3.910873).  Saving model.
Epoch: 17       Training Loss: 4.004927         Validation Loss: 3.833880
Validation loss has decreased (3.910873 --> 3.833880).  Saving model.
Epoch: 18       Training Loss: 3.943060         Validation Loss: 3.882541
Epoch: 19       Training Loss: 3.883121         Validation Loss: 3.736700
Validation loss has decreased (3.833880 --> 3.736700).  Saving model.
Epoch: 20       Training Loss: 3.846575         Validation Loss: 3.705761
Validation loss has decreased (3.736700 --> 3.705761).  Saving model.
```

In [26]:
```python
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.1.13 (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 [27]:
```python
def test(loaders, model, criterion, use_cuda):

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

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

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

        print('Test Loss: {:.6f}\n'.format(test_loss))
        print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
            100. * correct / total, correct, total))
```

In [28]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.736864


Test Accuracy: 14% (974/6680)
```

1.1.14 Evaluate the model

In [29]:
```
from sklearn.metrics import classification_report, confusion_matrix,precision_recall_fs

def evaluate_model(loaders, model, criterion, use_cuda):
    '''
    This function will calculate various evaluation metrics to get the overall accuracy
    error, loss, precision, recall, and F1 score

    Args:
        loaders:   dataloader with test dataset
        model:     model for prediction
        criterion: Loss Function
        use_cuda:  True if GPU is used, False if CPU is used
    Returns:
        Returns a classification_report from sklearn
    '''
    # Initialize the prediction and label lists(tensors)
    predlist=torch.zeros(0,dtype=torch.long, device='cpu')
    lbllist=torch.zeros(0,dtype=torch.long, device='cpu')


    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)

        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]

        _, preds = torch.max(output, 1)

        # Append batch prediction results for later calculating the f1-score
        predlist=torch.cat([predlist,preds.view(-1).cpu()])
        lbllist=torch.cat([lbllist,target.view(-1).cpu()])

    precision, recall, fscore, support = precision_recall_fscore_support(lbllist.numpy(

    print('Test Precision score: {:.4f}\n'.format(precision))
    print('Test Recall score: {:.4f}\n'.format(recall))
    print('Test F1 score: {:.4f}\n'.format(fscore))
```

```
In [30]: # Evaluate model with test data
         evaluate_model(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Precision score: 0.2170

Test Recall score: 0.1429

Test F1 score: 0.1262
```

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

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.

1.1.16 (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.

1.1.17 Import the required modules

```
In [31]: ## TODO: Specify data loaders
         import os
```

```
from torchvision import datasets

from PIL import Image
import torch
# check if CUDA is available
use_cuda = torch.cuda.is_available()

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

1.1.18 Define Data Augmentation using Transforms

In [32]:
```
import torchvision.transforms as transforms
# Normalizing the image with specific mean and standard deviation
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

train_transform = transforms.Compose([transforms.Resize(256),           # Resize the i
                                      transforms.RandomResizedCrop(224), # Crop the ima
                                      transforms.RandomHorizontalFlip(), # Horizontally
                                      transforms.RandomRotation(10),     # Rotate the i
                                      transforms.ToTensor(),             # Convert the
                                      normalize])                        # Normalize us
valid_transform = transforms.Compose([transforms.Resize(256),           # Resize the i
                                      transforms.CenterCrop(224),       # Crop the ima
                                      transforms.ToTensor(),             # Convert the
                                      normalize])                        # Normalize us
test_transform = transforms.Compose([transforms.Resize(256),            # Resize the i
                                      transforms.CenterCrop(224),       # Crop the ima
                                      transforms.ToTensor(),             # Convert the
                                      normalize])                        # Normalize us
```

1.1.19 Define Data loaders

In [33]:
```
# Set Batch size and number of workers
batch_size = 20
num_workers = 0

# Instantiate generic data loaders where the images are arranged in a certain way
train_data = datasets.ImageFolder(train_dir, transform=train_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=valid_transform)
test_data = datasets.ImageFolder(test_dir, transform=test_transform)

# Set data loaders for training, validation and testing
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
```

```
test_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=Tr

# Put data loaders to a dictionary
loaders_transfer = {"train" : train_loader, "valid" : valid_loader, "test" : test_loade
```

1.1.20 (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 [34]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         # Load the pretrained ResNet50 Model from pytorch
         model_transfer = models.resnet50(pretrained=True)

         # Add a Dropout layer
         model_transfer.add_module('drop', nn.Dropout(0.25))

         # Add a fully-connected layer
         model_transfer.add_module('fc1', nn.Linear(in_features=1000, out_features=133, bias=Tru

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

         #Replace the last layer
         model_transfer.fc = nn.Linear(2048, 1000, bias=True)
         #add a dropout layer
         model_transfer.drop = nn.Dropout(0.25)
         #add a fully connected layer
         model_transfer.fc1 = nn.Linear(in_features=1000, out_features=133, bias=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 [00:01<00:00, 61607563.60it/s]
```

```
In [35]: print(model_transfer)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
```

```
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace)
      )
      (2): Bottleneck(
        (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace)
      )
      (3): Bottleneck(
        (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace)
      )
    )
    (layer3): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace)
        (downsample): Sequential(
          (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace)
```

```
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
```

```
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
  (drop): Dropout(p=0.25)
  (fc1): Linear(in_features=1000, out_features=133, bias=True)
)
```

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

Answer: Resnet50 is one of the pre-trained models on ImageNet similar to other models such as VGG16 or Inception V3. ResNet50 is one of the models with lower Top-1 error rate. I have chosen the ResNet50 model as described her Transfer Learning using ResNet50 in PyTorch.

I have frozen all the parameters from pre-trained model and added a dropout and fully connected layer at the end to fine tune the final classifier.

Step by step process is as follows: 1. Load the pre-trained model 2. Freeze all the model parameters 3. Replace the last fully connected layer 4. Add a dropout layer to reduce overfitting 5. Add a new fully connected layer to classify 133 dog breeds

Reference: https://keras.io/api/applications/

1.1.21 (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 [36]: import torch.optim as optim
```

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001,momentum=0.9)
```

I chose SGD over adam as SGD + momentum can converge better with longer training time compared to adam.

1.1.22 (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 [56]: # Number of epochs
         n_epochs = 10

         # train the model
         model_transfer =  train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
```

```
Epoch: 1         Training Loss: 4.741766         Validation Loss: 3.453007
Validation loss has decreased (inf --> 3.453007).  Saving model.
Epoch: 2         Training Loss: 3.149532         Validation Loss: 2.432256
Validation loss has decreased (3.453007 --> 2.432256).  Saving model.
Epoch: 3         Training Loss: 2.472412         Validation Loss: 1.933016
Validation loss has decreased (2.432256 --> 1.933016).  Saving model.
Epoch: 4         Training Loss: 2.059152         Validation Loss: 1.684849
Validation loss has decreased (1.933016 --> 1.684849).  Saving model.
Epoch: 5         Training Loss: 1.838440         Validation Loss: 1.497692
Validation loss has decreased (1.684849 --> 1.497692).  Saving model.
Epoch: 6         Training Loss: 1.663197         Validation Loss: 1.401120
Validation loss has decreased (1.497692 --> 1.401120).  Saving model.
Epoch: 7         Training Loss: 1.554517         Validation Loss: 1.254564
Validation loss has decreased (1.401120 --> 1.254564).  Saving model.
Epoch: 8         Training Loss: 1.455464         Validation Loss: 1.214825
Validation loss has decreased (1.254564 --> 1.214825).  Saving model.
Epoch: 9         Training Loss: 1.379295         Validation Loss: 1.132802
Validation loss has decreased (1.214825 --> 1.132802).  Saving model.
Epoch: 10         Training Loss: 1.339136         Validation Loss: 1.110001
Validation loss has decreased (1.132802 --> 1.110001).  Saving model.
```

```
In [37]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.23 (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 [38]: # Test the model with test data
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.122238


Test Accuracy: 74% (4945/6680)
```

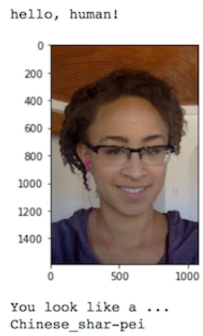
```
In [39]: # Evaluate model with test data
         evaluate_model(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Precision score: 0.7641

Test Recall score: 0.7221

Test F1 score: 0.7260
```

1.1.24 (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 [40]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         import torch
         import torchvision.transforms as transforms
         from PIL import Image

         # 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['test'].dataset.


         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             # Normalizing the image with specific mean and standard deviation
             normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                              std=[0.229, 0.224, 0.225])

             transform = transforms.Compose([transforms.Resize(256),             # Resize the im
                                             transforms.CenterCrop(224),         # Crop the ima
                                             transforms.ToTensor(),              # Convert the
                                             normalize])                         # Normalize us

             img = Image.open(img_path).convert('RGB') # Load Image
             img = transform(img).unsqueeze(0) # Transform Image and convert to one-dimensional
             if use_cuda:
                 img = img.cuda()  # transform img to CUDA-Datatype otherwise use CPU-Datatype
             out = model_transfer(img) # get prediction
             _, prediction = torch.max(out, 1) # Get the indexes for maximum values
```



Sample Human Output

```
        pred = np.squeeze(prediction.cpu().numpy()) # convert to one-dimensional tensor

        return class_names[pred] # return the predicted class name
```

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!

1.1.25 (IMPLEMENTATION) Write your Algorithm

```
In [41]: ### 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_ext(img_path):
                 print("Human detected!")
                 predicted_breed = predict_breed_transfer(img_path)
                 image = Image.open(img_path)
                 plt.imshow(image)
                 plt.show()
                 print("You look like a ", predicted_breed)
                 print()

             elif dog_detector(img_path):
                 print("Dog detected!")
```

```
            predicted_breed = predict_breed_transfer(img_path)
            image = Image.open(img_path)
            plt.imshow(image)
            plt.show()
            print("Detected breed is: ", predicted_breed)
            print()

        else:
            print("Error! Please try again.")
            image = Image.open(img_path)
            plt.imshow(image)
            plt.show()
            print('\n')
```

Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.26 (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.

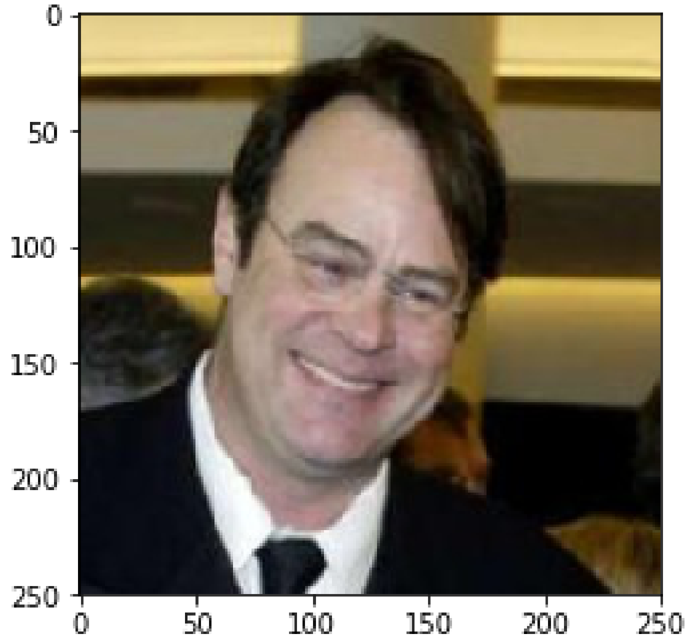
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

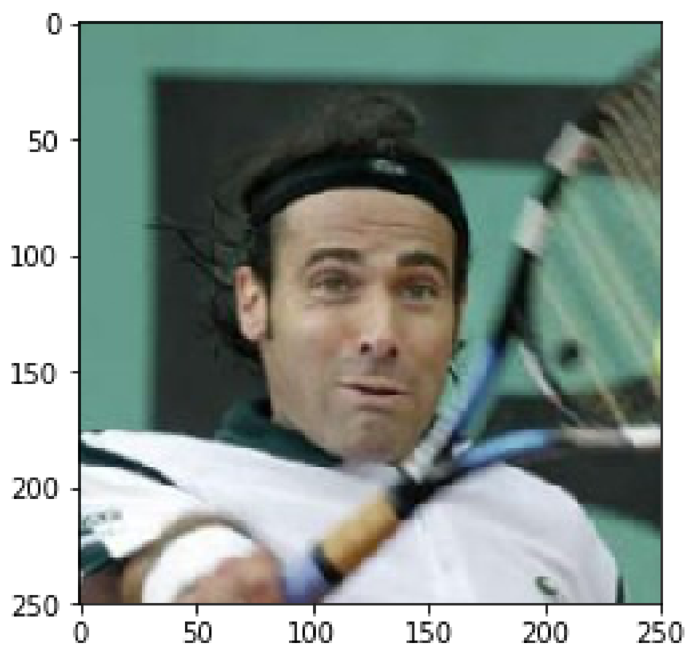
I think there is enough scope for improvment in the model: 1. Hyperparameter tuning (weights, learning rate, dropouts, batch size, etc.) can help in model performance improvement 2. Additional data with more dog breeds and/or more data augmentation can certaily help in improving model performance 3. Model can be made available to end users via web application

```
In [42]: ## 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)
```

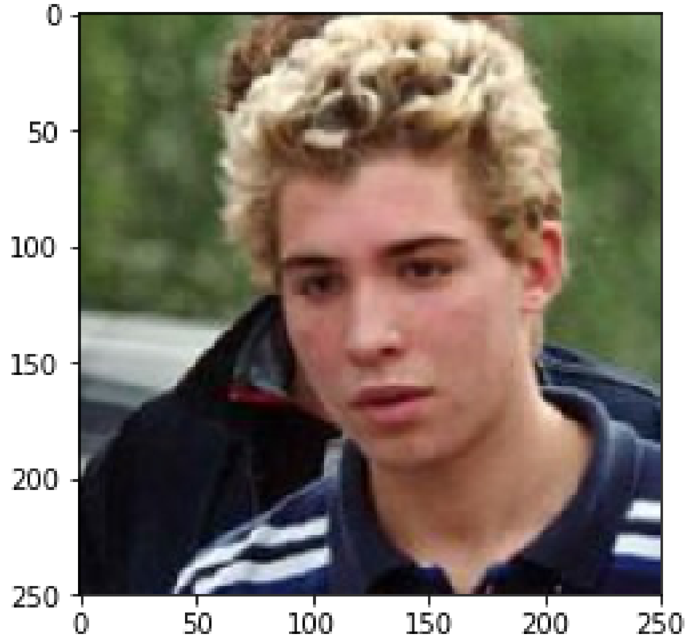
```
Human detected!
```



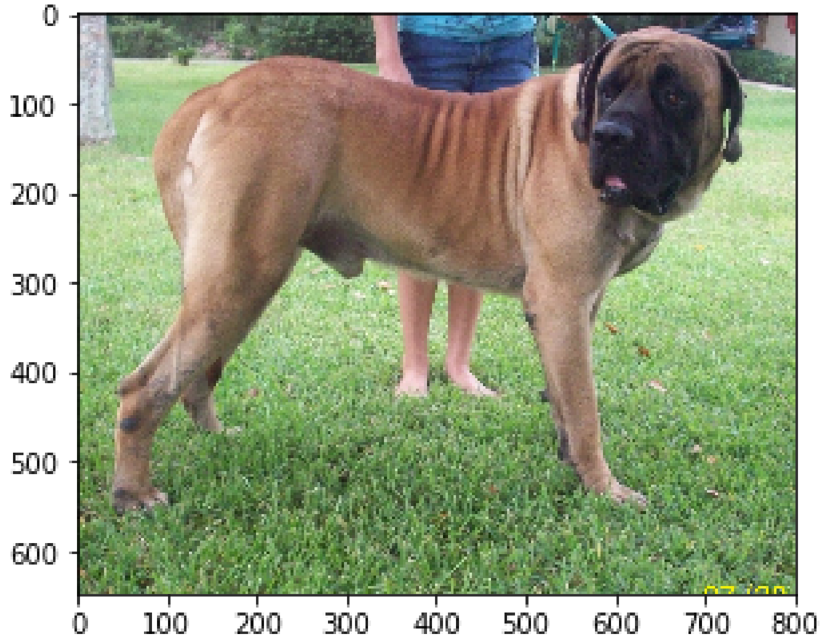

```
You look like a  Chihuahua

Human detected!
```



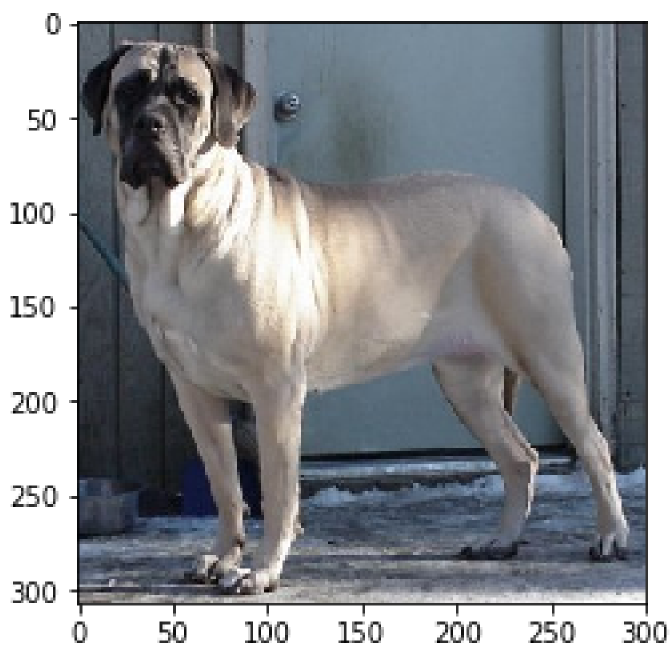
```
You look like a  Bull terrier

Human detected!
```



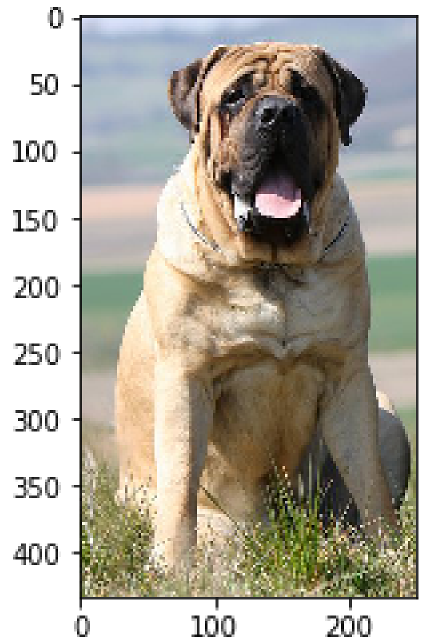

```
You look like a  American water spaniel

Dog detected!
```




```
Detected breed is:  Bullmastiff

Dog detected!
```



```
Detected breed is:  Bullmastiff

Dog detected!
```




```
Detected breed is:  Bullmastiff
```

In [ ]: