dog_app

March 2, 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: * 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 [28]:
```
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('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

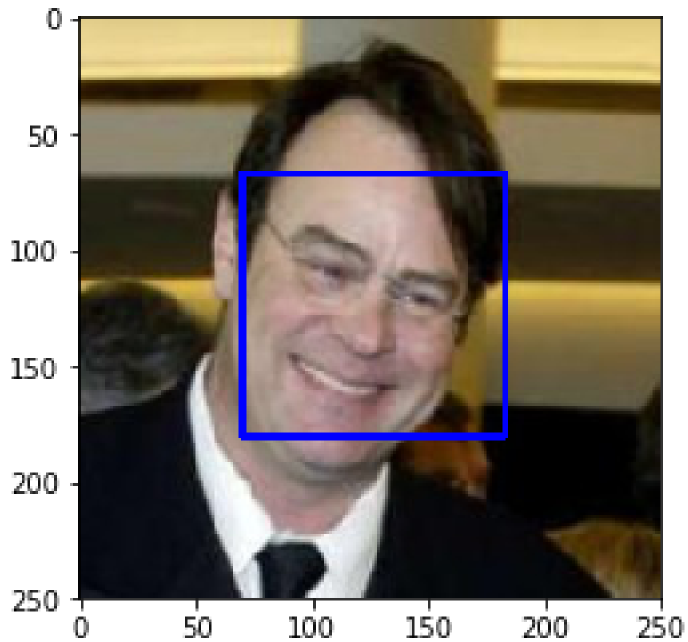
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [29]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [30]: # 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:

What percentage of the first 100 images in human_files have a detected human face? > 98%

What percentage of the first 100 images in dog_files have a detected human face? > 17%

```
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.
```

```
In [41]: #print(human_files_short[0])
         #print(face_detector(human_files_short[0]))
         human_face_found =0
         for h_imgpath in human_files_short:
             if face_detector(h_imgpath):
                 human_face_found +=1

         print("{}% human faces have been detected correctly from the human_files images.".forma

         dog_face_found =0
         for d_imgpath in dog_files_short:
             if face_detector(d_imgpath):
                 dog_face_found +=1

         print("{}% human faces have been detected incorrectly in the dog_files images.".format(
```

```
98.0% human faces have been detected correctly from the human_files images.
17.0% human faces have been detected incorrectly in the dog_files images.
```

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 [ ]: ### (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 [31]: import torch
         import torchvision.models as models

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

```
In [32]: # define VGG16 model
         VGG16 = models.vgg16(pretrained=True)



         # 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.

```
In [17]: print(VGG16)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

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 [33]: 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

             im = Image.open(img_path)

             #printing the image for a test
             #plt.imshow(im)
             #plt.show()


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

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

             getImage = transforms.ToPILImage()

             im_preproc =  preprocessimg(im)


             #printing the image for a test
             #plt.imshow(getImage(im_preproc))
             #plt.show()


             im_preproc.unsqueeze_(0)
             if use_cuda:
```

```
            im_preproc = im_preproc.cuda()

        output = VGG16(im_preproc)

        if use_cuda:
            output = output.cpu()


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

        #max index
        #print(predicted_Index)

        return predicted_Index # predicted class index

    #VGG16_predict(human_files[1])
```

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.
    pred_Index = VGG16_predict(img_path)
    #print(pred_Index)
    if pred_Index >=151 and pred_Index <=268:

        return True
    else:
        return False


    #return None # true/false

print("This is a Human. Dog ditector:{}".format(dog_detector(human_files[1])))
print("This is a dog. Dog ditector:{}".format(dog_detector(dog_files[1])))
print("This is a Human. Wrongly classified.  Dog ditector:{}".format(dog_detector('/data
```

```
This is a Human. Dog ditector:False
This is a dog. Dog ditector:True
This is a Human. Wrongly classified.  Dog ditector: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?

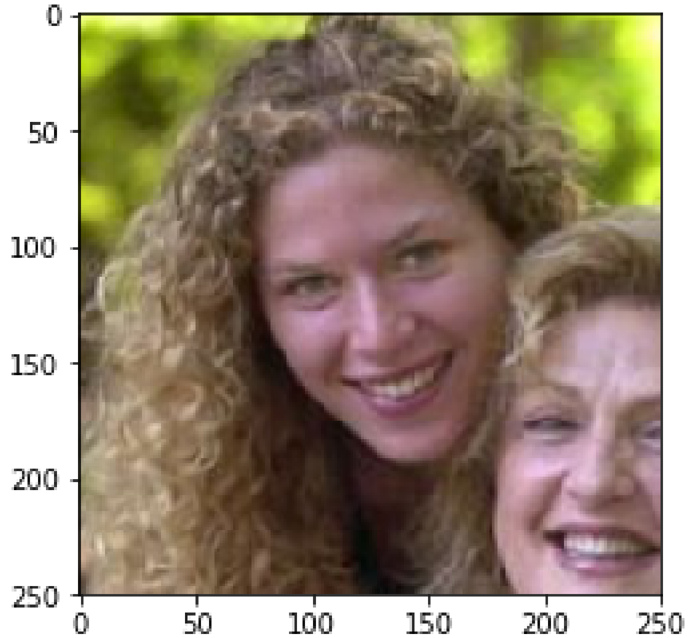
Answer:
What percentage of the images in human_files_short have a detected dog? > 1%
What percentage of the images in dog_files_short have a detected dog? > 100%

In [42]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

human_face_found =0
for h_imgpath in human_files_short:
    if dog_detector(h_imgpath):
        human_face_found +=1

print("{}% dog faces have been detected from the human_files images.".format(human_face

dog_face_found =0
for d_imgpath in dog_files_short:
    if dog_detector(d_imgpath):
        dog_face_found +=1

print("{}% dog faces have been detected  in the dog_files images.".format(dog_face_foun
```

```
1.0% dog faces have been detected from the human_files images.
100.0% dog faces have been detected  in the dog_files images.
```

In [44]:
```
#printing the error ditectioin image for a test

for h_imgpath in human_files_short:
    if dog_detector(h_imgpath):
        print(h_imgpath)
        im = Image.open(h_imgpath)
        plt.imshow(im)
        plt.show()
        break
```

```
/data/lfw/Perri_Shaw/Perri_Shaw_0001.jpg
```



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 [ ]: ### (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 [34]:
```
import os
from torchvision import datasets

#otherwise throwing error during converions (training)
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

batch_size = 20
test_batch_size = 15

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

train_transforms_process = transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        normalize
    ])
```

```
transforms_process = transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        normalize
    ])

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


train_data_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuf
valid_data_loader = torch.utils.data.DataLoader(valid_data, batch_size=test_batch_size)
test_data_loader = torch.utils.data.DataLoader(test_data, batch_size=test_batch_size)

loaders_scratch ={'train': train_data_loader,
                  'valid': valid_data_loader,
                  'test': test_data_loader}
```

In [35]:
```
print(" train data: {}".format(len(train_data)))
print(" valid data: {}".format(len(valid_data)))
print(" test data: {}".format(len(test_data)))
```

```
 train data: 6680
 valid data: 835
 test data: 836
```

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: >For the training dataset, I wanted to create 224 * 224 * 3 tensors. So I randomly cropped the image in 224 * 224 dimension. This was big enough image size to work on the current problem.

> Yes, the image was cropped in the above size, rotated 10 degree randomly and horizentally flipped randomly. This was done to avoid overfitting of data and also to make the model more efficient for cases when it gets slight distorted images.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [36]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
```

```python
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, 16, 3, padding=1) # 224 * 224 * 3  >  112 * 112 * 16
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1) # 112 * 112 * 16 > 56 * 56 * 64
        self.conv3= nn.Conv2d(32, 64, 3, padding=1) # 56 * 56 * 32 > 28 * 28 * 64
        self.conv4= nn.Conv2d(64, 128, 3, padding=1) # 28 * 28 * 64> 14 * 14 * 128

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

        self.fc1 = nn.Linear(128*14*14, 500)
        self.fc2 = nn.Linear(500,133)  #output size 133

        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)))
        x = self.pool(F.relu(self.conv4(x)))

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

        x= self.dropout(x)
        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, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 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=500, bias=True)
  (fc2): Linear(in_features=500, 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:

After input image processing, I decide the input layer to be 224 * 224 * 3 tensor. So in the model, the first convulation layer takes 224 * 224 * 3 as input and outputs 112 * 112 * 16. It reduces the dimension by half and increase the depth by 2 times. Likewise all convolution layer gradually increasing the depth and reduces the dimention, this helps the model to extract more features form the input image.

I started with only 3 convulation layers but it did not produce desired accuracy. So I increase the convulation layer and with 4 layers I was able to reach the target.

After convulation layer, MaxPool2d is used to down-sample so that eventualy we can flatten the layers. I have used 2 linear layers to flatten the features and narrowed it down. The network is trying to categorise the images into 133 classes so the output layer is one of 133. I have also used dropout (with probability of .25) to avoid overfitting and used Relu in the linear layer as it has been very efficient.

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 [37]: import torch.optim as optim

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


         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters())
```

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 [38]: 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
```

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

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

        output = model(data)

        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss = 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))



    train_loss = train_loss/len(train_data_loader.dataset)
    valid_loss = valid_loss/len(valid_data_loader.dataset)

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

```
                valid_loss
                ))

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

            if valid_loss <= valid_loss_min:
                print("Saving Model. Valid_loss {} is less than valid_loss_min {} ".format(
                torch.save(model.state_dict(),save_path)
                valid_loss_min = valid_loss


        # return trained model
        return model
```

In [39]:
```
# train the model
#100
model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1 	Training Loss: 0.000727 	Validation Loss: 0.005694
Saving Model. Valid_loss 0.005694288294762373 is less than valid_loss_min inf
Epoch: 2 	Training Loss: 0.000702 	Validation Loss: 0.005456
Saving Model. Valid_loss 0.005455581936985254 is less than valid_loss_min 0.005694288294762373
Epoch: 3 	Training Loss: 0.000680 	Validation Loss: 0.005286
Saving Model. Valid_loss 0.005286138970404863 is less than valid_loss_min 0.005455581936985254
Epoch: 4 	Training Loss: 0.000663 	Validation Loss: 0.005142
Saving Model. Valid_loss 0.005141913890838623 is less than valid_loss_min 0.005286138970404863
Epoch: 5 	Training Loss: 0.000654 	Validation Loss: 0.005155
Epoch: 6 	Training Loss: 0.000642 	Validation Loss: 0.005033
Saving Model. Valid_loss 0.005032563582062721 is less than valid_loss_min 0.005141913890838623
Epoch: 7 	Training Loss: 0.000632 	Validation Loss: 0.004924
Saving Model. Valid_loss 0.004923985805362463 is less than valid_loss_min 0.005032563582062721
Epoch: 8 	Training Loss: 0.000623 	Validation Loss: 0.004870
Saving Model. Valid_loss 0.004870202392339706 is less than valid_loss_min 0.004923985805362463
Epoch: 9 	Training Loss: 0.000618 	Validation Loss: 0.004891
Epoch: 10 	Training Loss: 0.000612 	Validation Loss: 0.004781
Saving Model. Valid_loss 0.004781119525432587 is less than valid_loss_min 0.004870202392339706
Epoch: 11 	Training Loss: 0.000607 	Validation Loss: 0.004767
Saving Model. Valid_loss 0.004766898695379496 is less than valid_loss_min 0.004781119525432587
Epoch: 12 	Training Loss: 0.000600 	Validation Loss: 0.004770
Epoch: 13 	Training Loss: 0.000592 	Validation Loss: 0.004680
Saving Model. Valid_loss 0.0046796840615570545 is less than valid_loss_min 0.004766898695379496
Epoch: 14 	Training Loss: 0.000586 	Validation Loss: 0.004750
Epoch: 15 	Training Loss: 0.000581 	Validation Loss: 0.004612
Saving Model. Valid_loss 0.0046124448999762535 is less than valid_loss_min 0.004679684061557054
```

```
Epoch: 16 	Training Loss: 0.000577 	Validation Loss: 0.004595
Saving Model. Valid_loss 0.004594812635332346 is less than valid_loss_min 0.004612444899762535
Epoch: 17 	Training Loss: 0.000573 	Validation Loss: 0.004756
Epoch: 18 	Training Loss: 0.000563 	Validation Loss: 0.004541
Saving Model. Valid_loss 0.004540877416729927 is less than valid_loss_min 0.004594812635332346
Epoch: 19 	Training Loss: 0.000560 	Validation Loss: 0.004505
Saving Model. Valid_loss 0.004504651762545109 is less than valid_loss_min 0.004540877416729927
Epoch: 20 	Training Loss: 0.000553 	Validation Loss: 0.004505
Epoch: 21 	Training Loss: 0.000551 	Validation Loss: 0.004459
Saving Model. Valid_loss 0.004459007643163204 is less than valid_loss_min 0.004504651762545109
Epoch: 22 	Training Loss: 0.000549 	Validation Loss: 0.004509
Epoch: 23 	Training Loss: 0.000549 	Validation Loss: 0.004463
Epoch: 24 	Training Loss: 0.000541 	Validation Loss: 0.004493
Epoch: 25 	Training Loss: 0.000541 	Validation Loss: 0.004432
Saving Model. Valid_loss 0.004432493820786476 is less than valid_loss_min 0.004459007643163204
Epoch: 26 	Training Loss: 0.000535 	Validation Loss: 0.004479
Epoch: 27 	Training Loss: 0.000534 	Validation Loss: 0.004459
Epoch: 28 	Training Loss: 0.000527 	Validation Loss: 0.004427
Saving Model. Valid_loss 0.004427027422934771 is less than valid_loss_min 0.004432493820786476
Epoch: 29 	Training Loss: 0.000526 	Validation Loss: 0.004352
Saving Model. Valid_loss 0.004351572133600712 is less than valid_loss_min 0.004427027422934771
Epoch: 30 	Training Loss: 0.000527 	Validation Loss: 0.004265
Saving Model. Valid_loss 0.00426473980769515 is less than valid_loss_min 0.004351572133600712
```

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 [10]:
```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 [14]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.640179


Test Accuracy: 18% (153/836)
```

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 [11]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets
import torchvision.transforms as transforms

#otherwise throwing error during converions (training)
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

batch_size = 20
test_batch_size = 15
```

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

train_transforms_process = transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        #transforms.RandomVerticalFlip(),
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        normalize
    ])

transforms_process = transforms.Compose([
       transforms.Resize((224,224)),
       transforms.ToTensor(),
       normalize
    ])

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


train_data_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuf
valid_data_loader = torch.utils.data.DataLoader(valid_data, batch_size=test_batch_size)
test_data_loader = torch.utils.data.DataLoader(test_data, batch_size=test_batch_size)

loaders_transfer ={'train': train_data_loader,
                'valid': valid_data_loader,
                'test': test_data_loader}
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

In [12]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.resnet18(pretrained=True)

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

print(model_transfer)
model_transfer.fc = nn.Linear(512, 133, bias=True)
```

```python
#for param in model_transfer.fc.parameters():
#    param.required_grad = True

print(model_transfer)

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

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root/.torch/models/
100%|| 46827520/46827520 [00:01<00:00, 25211542.06it/s]


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

```
      (relu): ReLU(inplace)
      (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)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
      (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): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

```
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): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
```

```
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
      (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): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (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)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=512, 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: Any pretrained torchvision models probably would have been good enough for the given problem. I have choosen ResNet because, base on some online search, I found ResNet has better performance than some of the other models. It even won the ImageNet prize on 2015.

I am using the pretrained model as is, apart from the last linear layer. The model had 1000 output feature but in this problem, we need only 133 output features. So I replaced the last linear layer with a new one. We also had to set the required_grad to false so that the model does not back propagate and start update the weights of all other layers for the new training set. Our intention is to train only the last year of the model.

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 [13]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()

         optimizer_transfer = optim.Adam(model_transfer.fc.parameters())
```

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 [17]: # train the model
         #model_transfer = # train(n_epochs, loaders_transfer, model_transfer, optimizer_transfe
         n_epochs = 10

         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
```

```
Epoch: 1         Training Loss: 0.000193         Validation Loss: 0.000946
Saving Model. Valid_loss 0.0009458689601160586 is less than valid_loss_min inf
Epoch: 2         Training Loss: 0.000177         Validation Loss: 0.000906
Saving Model. Valid_loss 0.0009062682511284947 is less than valid_loss_min 0.0009458689601160586
Epoch: 3         Training Loss: 0.000170         Validation Loss: 0.000908
Epoch: 4         Training Loss: 0.000164         Validation Loss: 0.000839
Saving Model. Valid_loss 0.0008389318827539682 is less than valid_loss_min 0.0009062682511284947
Epoch: 5         Training Loss: 0.000166         Validation Loss: 0.000859
Epoch: 6         Training Loss: 0.000161         Validation Loss: 0.000848
Epoch: 7         Training Loss: 0.000161         Validation Loss: 0.000916
Epoch: 8         Training Loss: 0.000161         Validation Loss: 0.000877
Epoch: 9         Training Loss: 0.000154         Validation Loss: 0.000897
Epoch: 10         Training Loss: 0.000153         Validation Loss: 0.000854
```

```
In [14]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

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

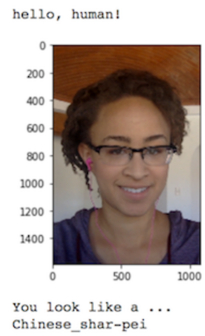
```
In [18]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.740331


Test Accuracy: 78% (653/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

In [23]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
data_transfer ={'train': train_data,
                'valid': valid_data,
                'test': test_data}


# 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'].classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    im = Image.open(img_path)

    #printing the image for a test
    #plt.imshow(im)
    #plt.show()


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

    preprocessimg = transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        normalize
    ])

    #getImage = transforms.ToPILImage()

    im_preproc =  preprocessimg(im)
    im_preproc.unsqueeze_(0)

    if use_cuda:
        im_preproc = im_preproc.cuda()

    output = model_transfer(im_preproc)

    #if use_cuda:
     #   output = output.cpu()

    #print(output)
    #print(torch.max(output.data,1))
```



Sample Human Output

```
    _, predicted_tensor = torch.max(output,1)

    if use_cuda:
        predicted_Index = np.squeeze(predicted_tensor.cpu().numpy())
    else:
        predicted_Index = np.squeeze(predicted_tensor.numpy())

    print("Predicted Index {}".format(predicted_Index))
    dog_breed =  class_names[predicted_Index]
    return dog_breed
```

```
In [24]: #print(dog_files_short[0])
         #print(human_files_short[0])

         predict_breed_transfer(dog_files_short[0])
```

```
Predicted Index 40
```

```
Out[24]: 'Bullmastiff'
```

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 [25]: import matplotlib.pyplot as plt
         from PIL import Image

         ### 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

             im = Image.open(img_path)
             #printing the image for a test
             plt.imshow(im)
             plt.show()

             if face_detector(img_path) == True:
                 prediction = predict_breed_transfer(img_path)
                 print("Human detected. Looks like: {}".format(prediction))
             elif dog_detector(img_path) == True:
                 prediction = predict_breed_transfer(img_path)
                 print("Dog detected. Looks like: {}".format(prediction))
             else:
                 print("Could not detect any dog or humnan in the picture.")
```

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.

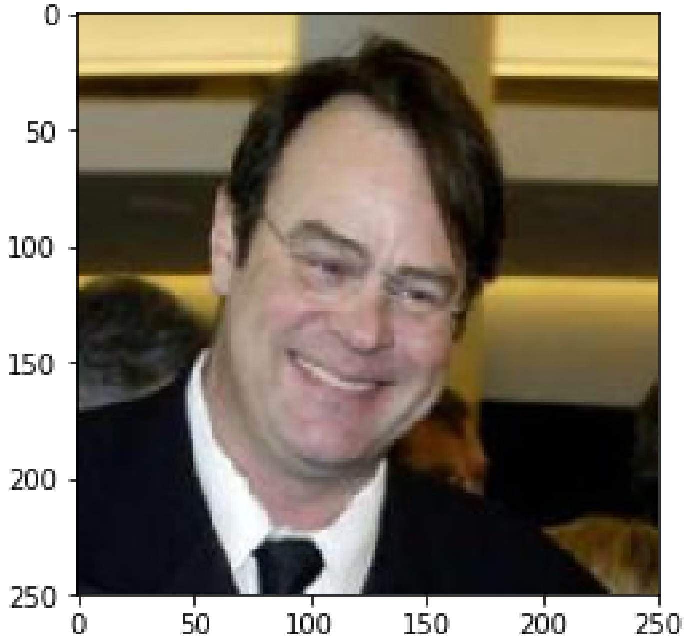
Answer:

I was fascinated that with such little training, the model can detect if there is no human/dog in the image and the breed of dogs. With only 45 Mb memory it can detect dog breeds more accurately than me!

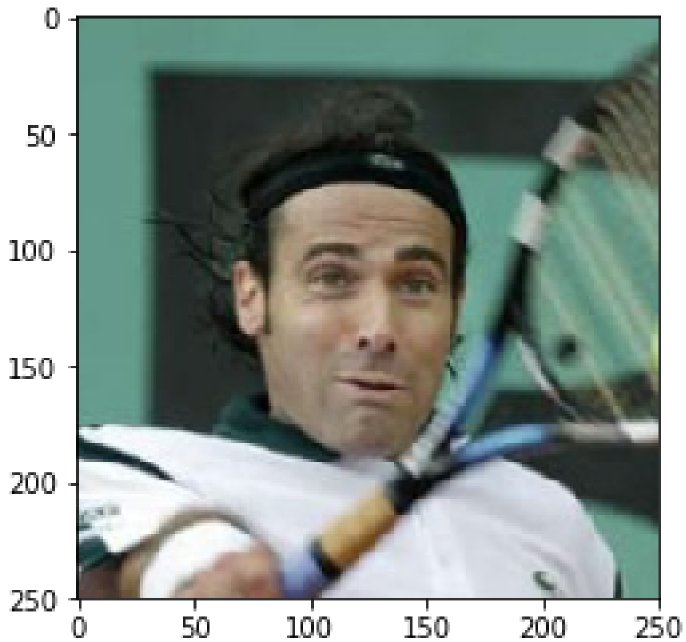
There are several things can be done to improve the performance. Here are three: * Use more varieties of training dataset to train the model * Use some other models and compare the result. Pick the model that performs the best * Increase the number of epoch and training time when we can keep reducing the validation loss.

```
In [26]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
```

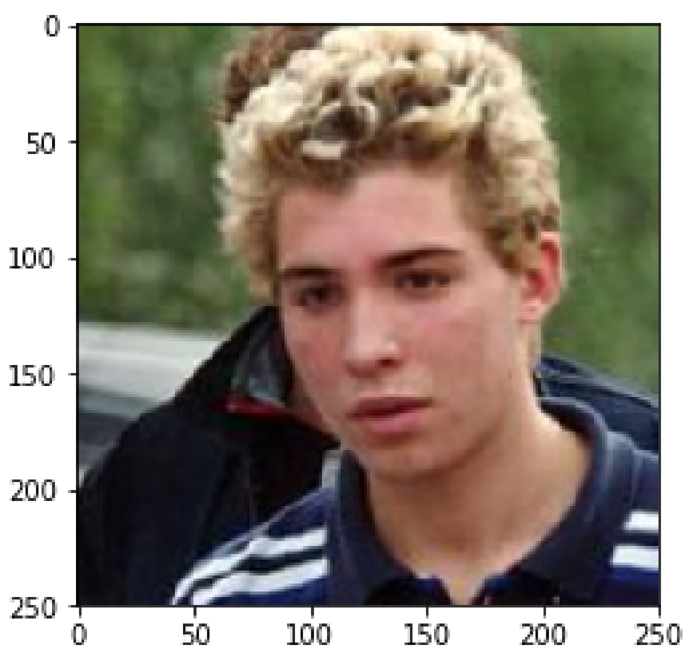
```python
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[:6], dog_files[:6])):
    run_app(file)
```



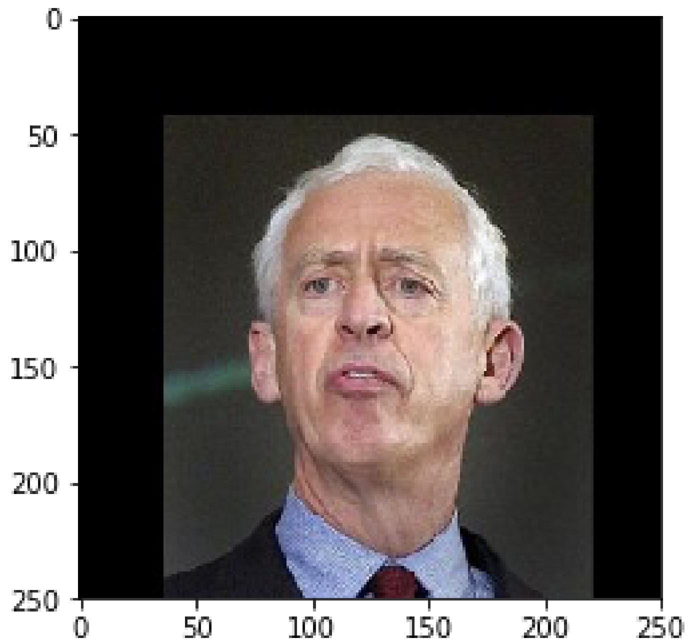

```
Predicted Index 64
Human detected. Looks like: Entlebucher mountain dog
```



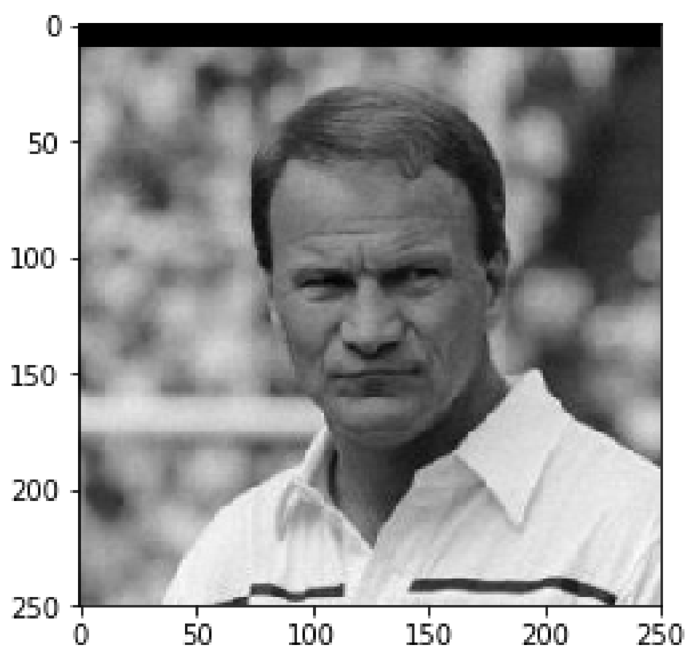

```
Predicted Index 55
Human detected. Looks like: Dachshund
```



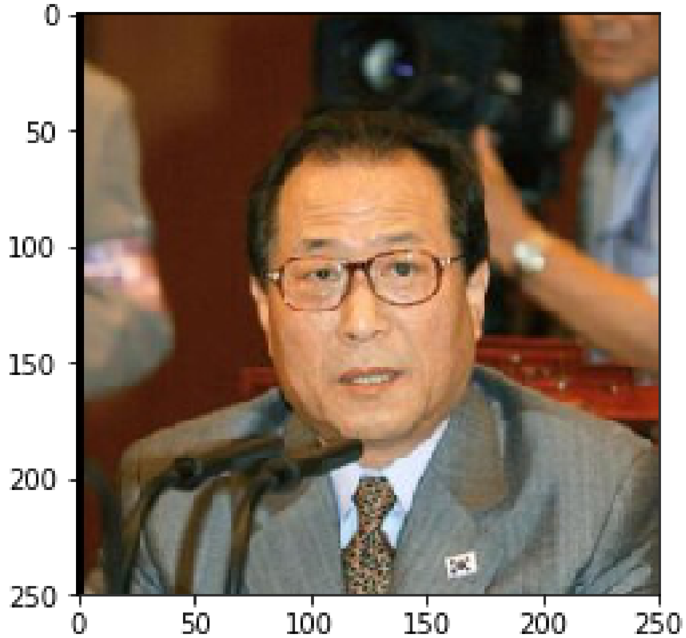
```
Predicted Index 8
Human detected. Looks like: American water spaniel
```



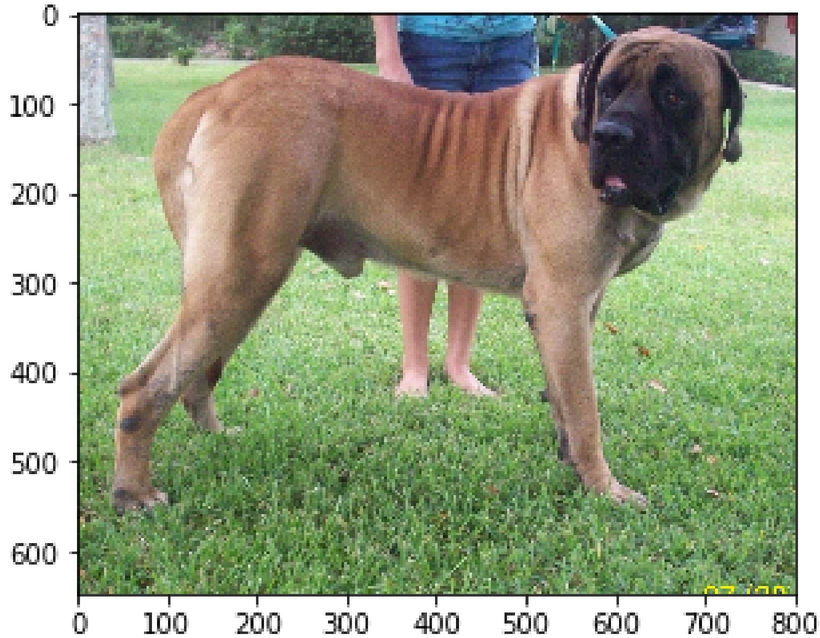

```
Predicted Index 96
Human detected. Looks like: Lakeland terrier
```



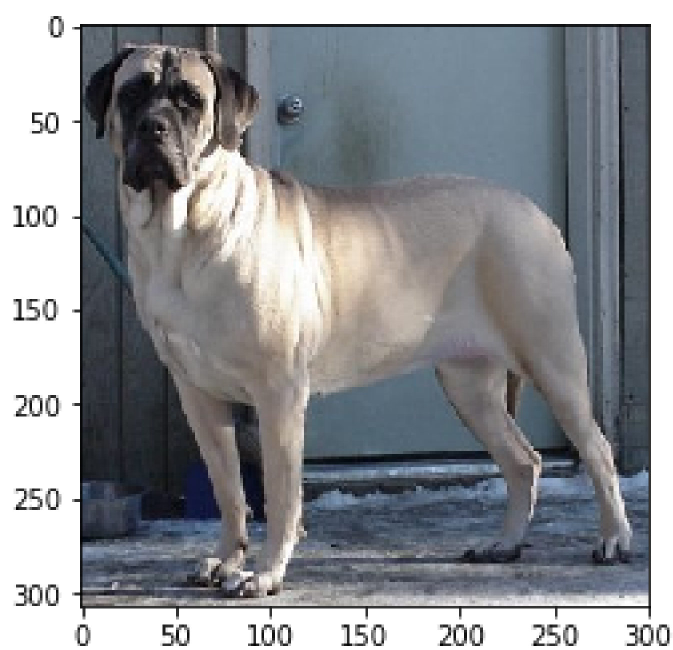
```
Predicted Index 123
Human detected. Looks like: Poodle
```



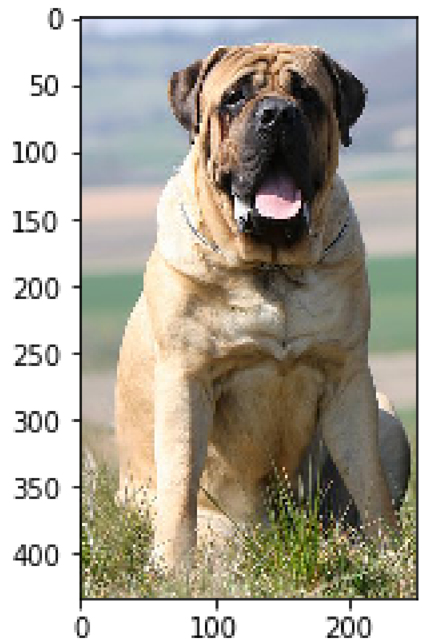

```
Predicted Index 13
Human detected. Looks like: Basenji
```



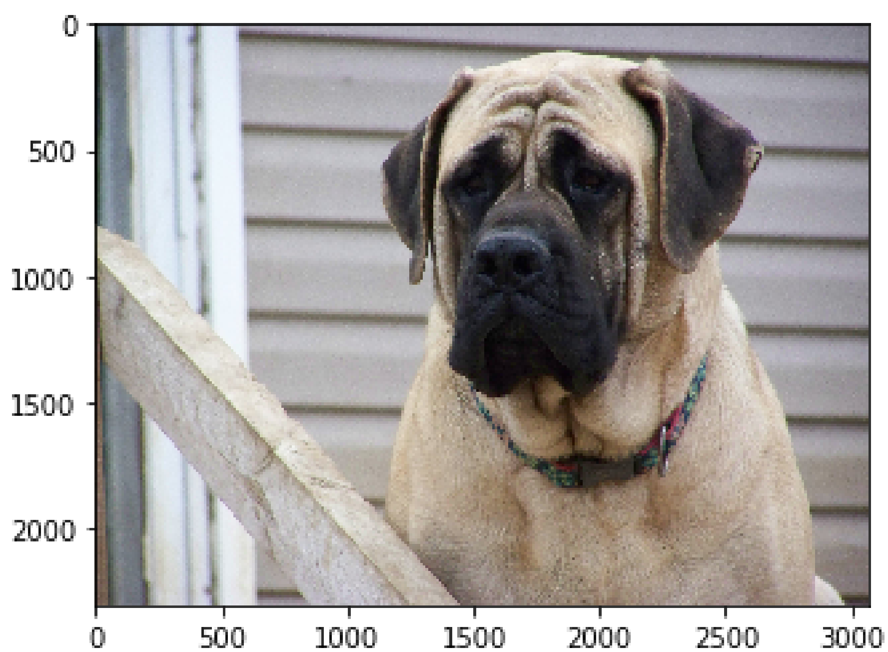
Predicted Index 40
Dog detected. Looks like: Bullmastiff



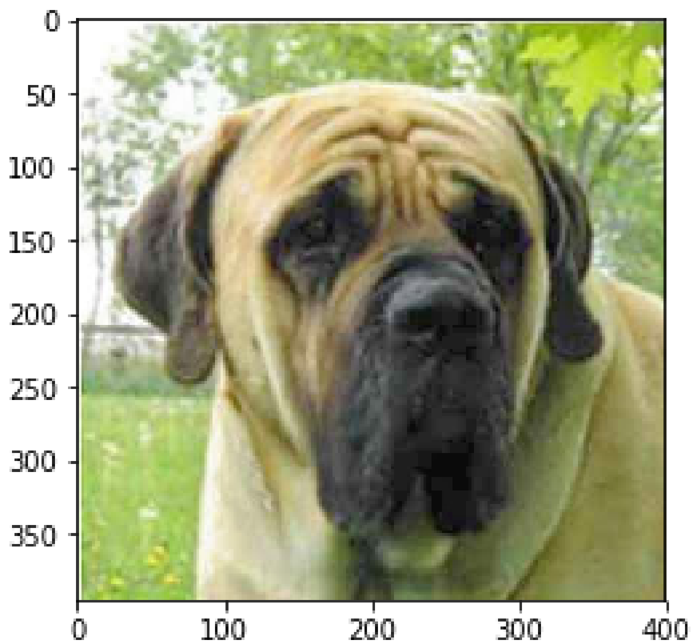
```
Predicted Index 9
Dog detected. Looks like: Anatolian shepherd dog
```



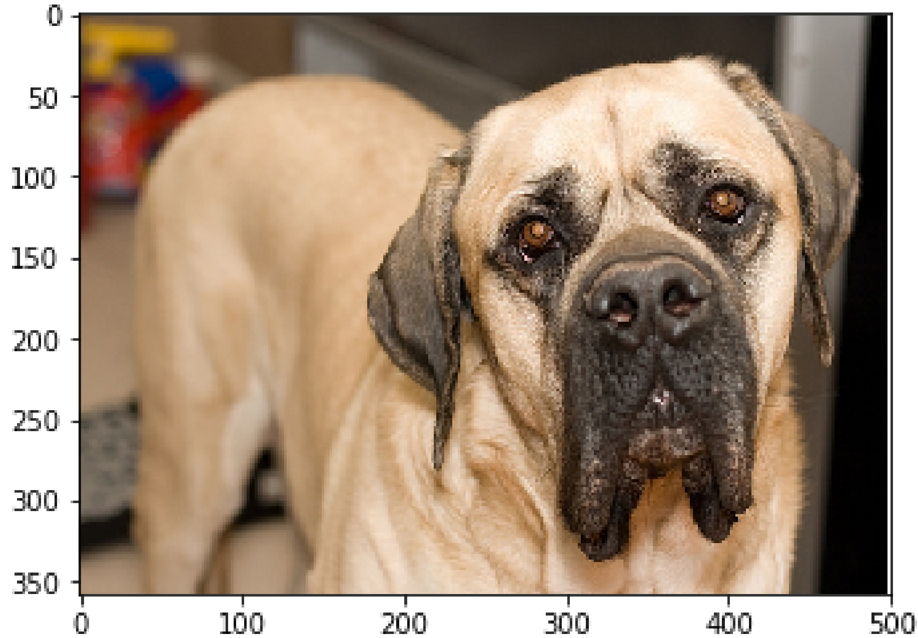

```
Predicted Index 40
Dog detected. Looks like: Bullmastiff
```



```
Predicted Index 102
Dog detected. Looks like: Mastiff
```



```
Predicted Index 40
Dog detected. Looks like: Bullmastiff
```

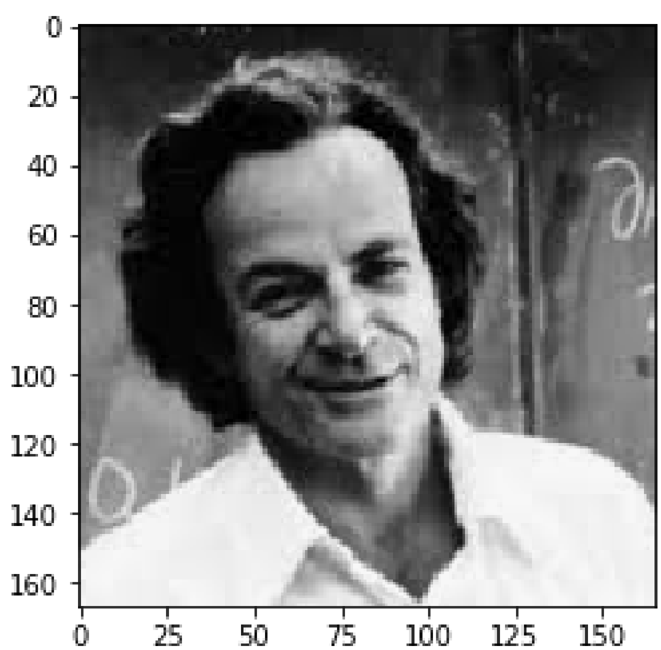


```
Predicted Index 102
Dog detected. Looks like: Mastiff
```

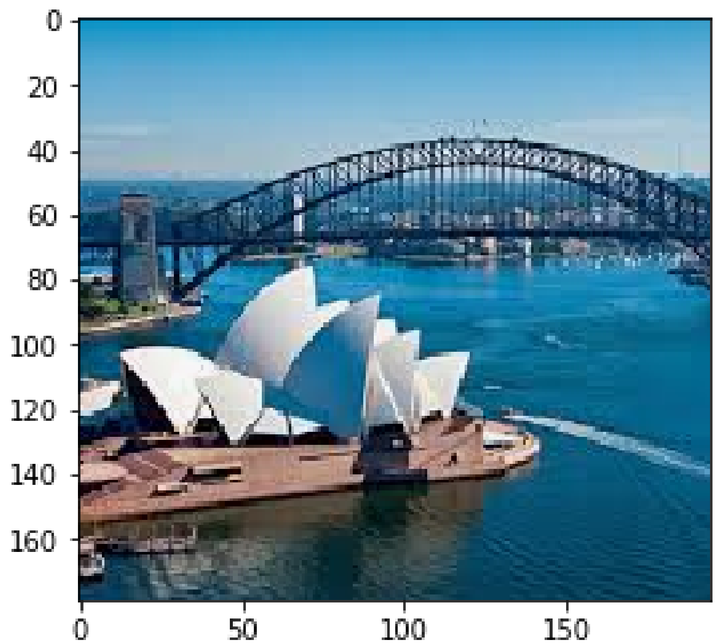
```
In [27]: import numpy as np
         from glob import glob
         import matplotlib.pyplot as plt
         from PIL import Image

         # load filenames for human and dog images
         test_files = np.array(glob('test_images/*'))

         # print number of images in each dataset
         print('There are %d total test images.' % len(test_files))
         for img_path in test_files:
             run_app(img_path)
             #im = Image.open(img_path)
             #printing the image for a test
             #plt.imshow(im)
             #plt.show()
```

```
There are 7 total test images.
```

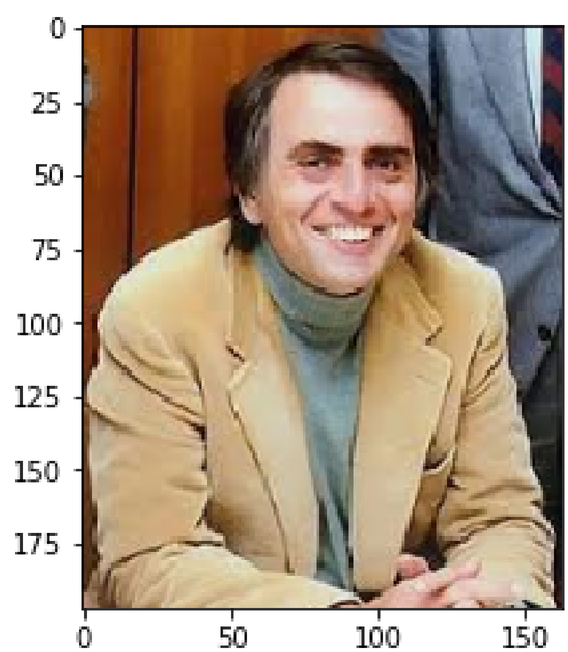

```
Predicted Index 19
Dog detected. Looks like: Belgian malinois
```



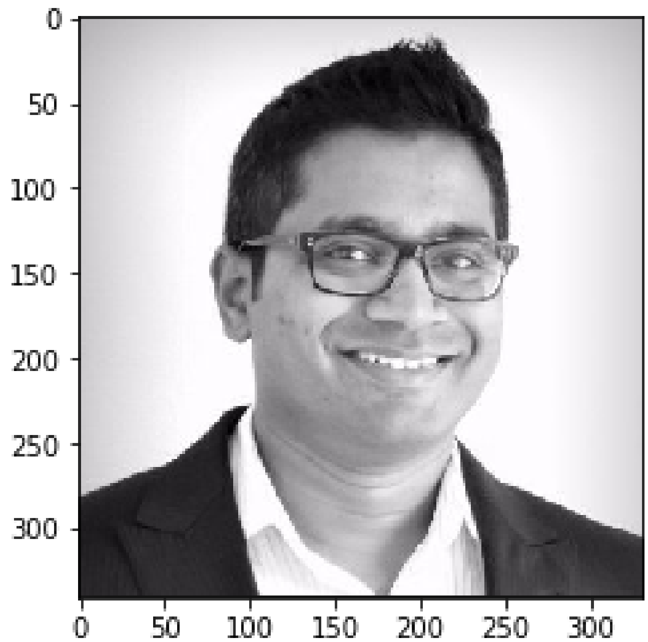
```
Predicted Index 55
Human detected. Looks like: Dachshund
```



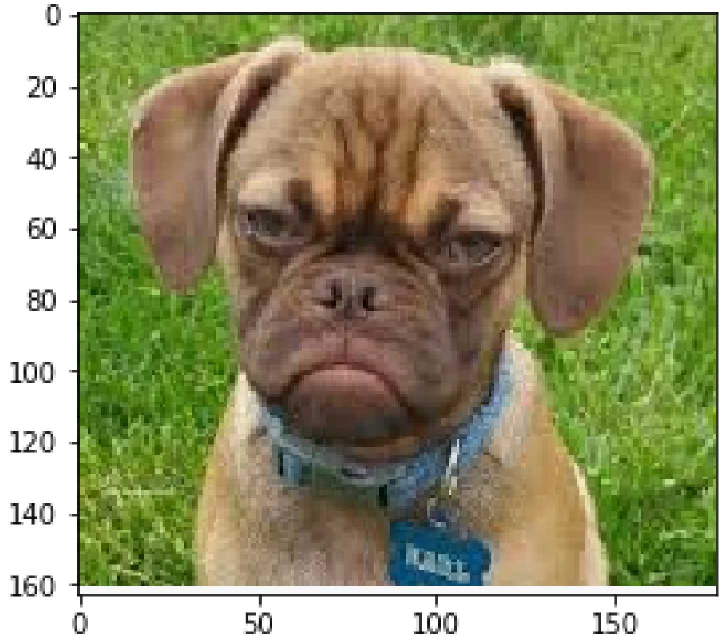
```
Could not detect any dog or humnan in the picture.
```



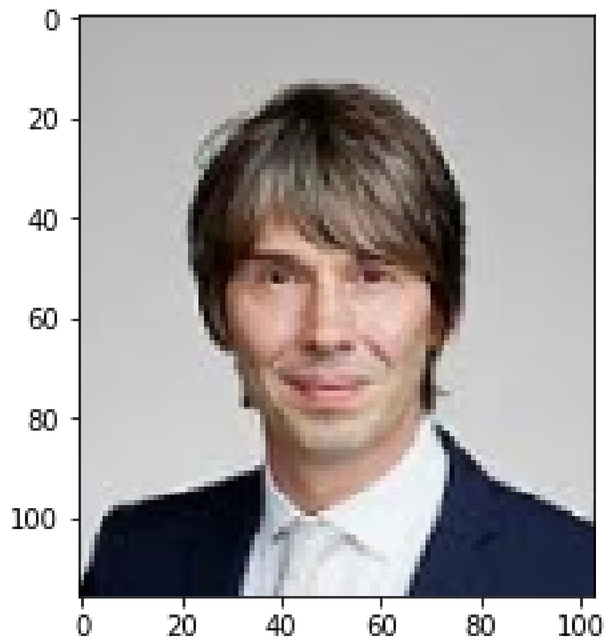
```
Predicted Index 59
Human detected. Looks like: Dogue de bordeaux
```




```
Predicted Index 123
Human detected. Looks like: Poodle
```




```
Predicted Index 59
Dog detected. Looks like: Dogue de bordeaux
```



```
Predicted Index 51
Human detected. Looks like: Clumber spaniel
```

```
In [ ]:
```