dog_app

February 13, 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

        # 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

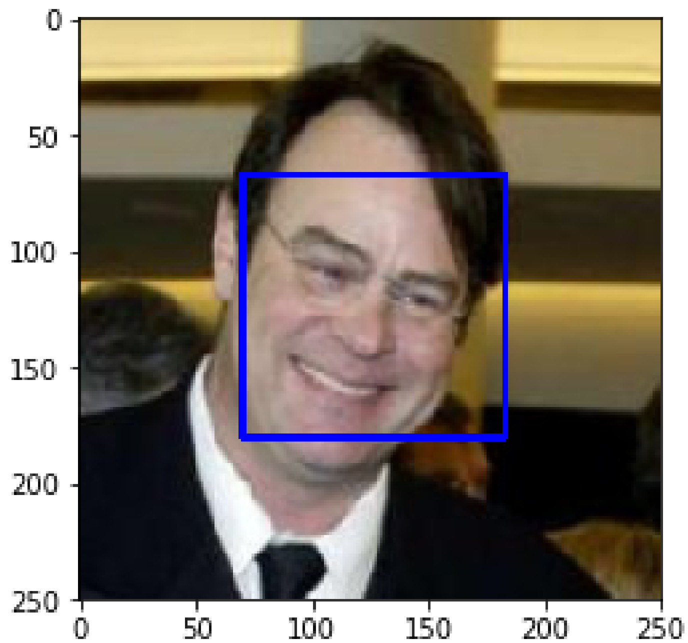
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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
        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.
        tp_counts = 0
        fp_counts = 0
        for human_file in human_files_short:
            if (face_detector(human_file) == True):
                tp_counts += 1
        tp_counts = (tp_counts / len(human_files_short)) * 100.0

        for dog_file in dog_files_short:
            if (face_detector(dog_file) == True):
                fp_counts += 1
        fp_counts = (fp_counts / len(dog_files_short)) * 100.0

        print(f"Percentage of detected human faces in human files is {tp_counts}%")
        print(f"Percentage of detected human faces in dog files is {fp_counts}%")
```

```
Percentage of detected human faces in human files is 98.0%
Percentage of detected human faces in dog files is 17.0%
```

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

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

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

In [4]:
```
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 [00:05<00:00, 94889242.50it/s]
```

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

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

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

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

In [5]:
```
from PIL import Image
import torchvision.transforms as transforms

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

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

    # load the image
    img = Image.open(img_path)

    # perform image preprocessing before feeding it to the network
    # takecare that the sequence of transform matters!
    transformToTensor = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])

    img = transformToTensor(img) # torch.Size([3, 224, 224])

    # create a batch of 1 image before passing to the network
    batch_img = img.unsqueeze(0) # torch.Size([1, 3, 224, 224])

    # Turn-off dropouts by turning the model to evaluation mode
```

```
    VGG16.eval()

    if use_cuda == True:
        batch_img = batch_img.to('cuda')

    # forward propagation
    output = VGG16(batch_img)

    # take the index of the highest score
    output = output.argmax()

    return output # predicted class index
```

In [6]:
```
VGG16_predict(dog_files_short[0])
```

Out[6]:
```
tensor(243, device='cuda:0')
```

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 [7]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    prediction = VGG16_predict(img_path)
    return (prediction >= 151 and prediction <= 268) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

In [10]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

fp_counts_2 = 0
for human_file in human_files_short:
    if (dog_detector(human_file) == True):
        fp_counts_2 += 1
fp_counts_2 = (fp_counts_2 * len(human_files_short)) / 100.0
```

```
tp_counts_2 = 0
for dog_file in dog_files_short:
    if (dog_detector(dog_file) == True):
        tp_counts_2 += 1
tp_counts_2 = (tp_counts_2 * len(dog_files_short)) / 100.0

print(f'Percentage of detected dogs in human images is {fp_counts_2}%')
print(f'Percentage of detected dogs in dog images is {tp_counts_2}%')
```

```
Percentage of detected dogs in human images is 0.0%
Percentage of detected dogs in dog images is 100.0%
```

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

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

         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()
```

```
Downloading: "https://download.pytorch.org/models/inception_v3_google-1a9a5a14.pth" to /root/.to
100%|| 108857766/108857766 [00:01<00:00, 105257706.43it/s]
```

```
In [12]: def inception_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

    # load the image
    img = Image.open(img_path)

    # perform image preprocessing before feeding it to the network
    # takecare that the sequence of transform matters!
    transformToTensor = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])

    img = transformToTensor(img) # torch.Size([3, 224, 224])

    # create a batch of 1 image before passing to the network
    batch_img = img.unsqueeze(0) # torch.Size([1, 3, 224, 224])

    # Turn-off dropouts by turning the model to evaluation mode
    inception.eval()

    # forward propagation
    output = inception.forward(batch_img)

    # take the index of the highest score
    output = output.data.numpy().argmax()

    return output # predicted class index

### returns "True" if a dog is detected in the image stored at img_path
def dog_detector_2(img_path):
    ## TODO: Complete the function.
    prediction = inception_predict(img_path)
    return (prediction >= 151 and prediction <= 268) # true/false
```

In [13]:
```
fp_counts_3 = 0
for human_file in human_files_short:
    if (dog_detector(human_file) == True):
        fp_counts_3 += 1
fp_counts_3 = (fp_counts_3 * len(human_files_short)) / 100.0


tp_counts_3 = 0
for dog_file in dog_files_short:
    if (dog_detector(dog_file) == True):
        tp_counts_3 += 1
tp_counts_3 = (tp_counts_3 * len(dog_files_short)) / 100.0
```

```
print(f'Percentage of detected dogs in human images is {fp_counts_3}%')
print(f'Percentage of detected dogs in dog images is {tp_counts_3}%')
```

```
Percentage of detected dogs in human images is 0.0%
Percentage of detected dogs in dog images is 100.0%
```

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 [1]: import numpy as np
        import torch
        import torch.nn as nn
        import os
        import torchvision
        from torchvision import datasets
        import torchvision.transforms as transforms
        from PIL import ImageFile
        import matplotlib.pyplot as plt

        ImageFile.LOAD_TRUNCATED_IMAGES = True

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

        transform = {
            'train' : transforms.Compose([
                                            transforms.Resize(256),
                                            transforms.CenterCrop(224),
                                            transforms.RandomHorizontalFlip(),
                                            transforms.RandomRotation(10),
                                            transforms.ToTensor(),
                                            transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                                                 std=(0.229, 0.224, 0.225))
                                         ]),
            'valid' : transforms.Compose([
                                        transforms.Resize(256),
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor(),
                                        transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                                             std=(0.229, 0.224, 0.225))
                                    ]),
        }

        # number of subprocesses to use for data loading
        num_workers = 0
        # how many samples per batch to load
        batch_size = 64

        train_data = datasets.ImageFolder(root='/data/dog_images/train/', transform=transform['t
        valid_data = datasets.ImageFolder(root='/data/dog_images/valid/', transform=transform['v
        tests_data = datasets.ImageFolder(root='/data/dog_images/test/', transform=transform['va

        n_classes = len(train_data.classes)
```

```
print(n_classes)

loaders_scratch = {
    'train' : torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_workers
    'valid' : torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_workers
    'test'  : torch.utils.data.DataLoader(tests_data, batch_size=batch_size, num_workers
}

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

```
133
```

We want visualize sample of the augmented training data

```
In [2]: def VisualizeImageTensor(tensor_img, title=None):
            """Imshow for Tensor."""
            tensor_img = tensor_img.numpy().transpose((1, 2, 0))
            mean = np.array([0.485, 0.456, 0.406])
            stdv = np.array([0.229, 0.224, 0.225])

            tensor_img = stdv * tensor_img + mean
            tensor_img = np.clip(tensor_img, 0, 1)

            figure = plt.figure(figsize=(60, 25))
            plt.axis('off')
            plt.imshow(tensor_img)

            if title is not None:
                plt.title(title)
            plt.pause(0.001)  # pause a bit so that plots are updated


        # Get a batch of training data
        # inputs contains 4 images because batch_size=4 for the dataloaders
        inputs, classes = next(iter(loaders_scratch['train']))

        # Make a grid from batch
        out = torchvision.utils.make_grid(inputs, nrow=5)

        #VisualizeImageTensor(out, title=[train_data.classes[x] for x in classes])
        VisualizeImageTensor(out)
```

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: - I've picked the image size of 224x224 to be similar to the input size of VGG network by croping from the center of the image by the 224 in both vertical and horizontal directions.

- Yes, I've augmented the data for the training datasets by adding extra manipulated images. I've performed random rotations, and flipping to increase the variations in the datasets, and allow the model to generalize during training and be rotation and translation invariant.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [8]: 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__()

                # Output size of convolutional layer is:
                # W_out = ((W_in - F + 2P) / S) + 1
                # W_out : size of the output image
                # W_in  : size of the input image
                # F     : filter size
                # P     : padding size
                # S     : stride size

                ## Define layers of a CNN
                # Conv-layer-1 (take   : 224 x 224 x 3)
                #              (output : 222 x 222 x 16)
                #              (pool   : 111 x 111 x 16)
                self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=0)

                # Conv-layer-2 (take   : 111 x 111 x 16)
                #              (output : 109 x 109 x 32)
                #              (pool   : 54 x 54 x 32)
                self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=0)

                # Conv-layer-3 (take   : 54 x 54 x 32)
                #              (output : 52 x 52 x 64)
                #              (pool   : 26 x 26 x 64)
```

```python
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=0)

        # Conv-layer-4 (take   : 26 x 26 x 64)
        #              (output : 24 x 24 x 128)
        #              (pool   : 12 x 12 x 128)
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=0)

        # Conv-layer-5 (take   : 12 x 12 x 128)
        #              (output : 10 x 10 x 256)
        #              (pool   : 5 x 5 x 256)
        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=0)

        # max pooling layer to reduce the dimensions to half size
        self.pool = nn.MaxPool2d(2, 2)

        # linear layer (28 * 28 * 256 -> 512)
        self.fc1 = nn.Linear(5 * 5 * 256, 500)

        # linear layer (512 -> 500)
        self.fc2 = nn.Linear(500, 500)

        # linear layer (512 -> n_classes)
        self.fc3 = nn.Linear(500, n_classes)

        # dropout layer with 25% probability that a node is turned-off
        self.dropout = nn.Dropout(p=0.2)

    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 = self.pool(F.relu(self.conv5(x)))
        x = x.view(-1 , 5 * 5 * 256) # flatten
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

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

# instantiate the CNN
model_scratch = Net()
```

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

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

NameError                                 Traceback (most recent call last)

<ipython-input-8-0e27c2ca2edf> in <module>()
     76
     77 # instantiate the CNN
---> 78 model_scratch = Net()
     79
     80 # move tensors to GPU if CUDA is available


<ipython-input-8-0e27c2ca2edf> in __init__(self)
     52
     53         # linear layer (512 -> n_classes)
---> 54         self.fc3 = nn.Linear(500, n_classes)
     55
     56         # dropout layer with 25% probability that a node is turned-off


NameError: name 'n_classes' is not defined
```

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

- I've designed the network to consist of five convolutional layers.
- The size of the output of a convolutional layer can be calculated by the formula:

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1$$

Where, W_{in} is the input image size to the conv-layer, W_{out} is the output image size from the conv-layer, F is the filter/kernel size, P is the padding amount, and S is the stride amount.

- The output of each convolutional layer passes to a ReLU activation function to add non-linearity.
- Each convolution layer is followed by a max-pooling layer to reduce the dimensionality of the output from the previous layer and pass only the most activated pixels.
- I've added 16 kernels/filters to the first layer to detect the basic features like edges and colors.
- The second layer consists of 32 kernels which are responsible to detect features like curves, circles, rectangles, etc. These kernels detect the combined patterns from the basic features from the first layer.

- The third layer consists of 64 kernels to detect more complex shapes like faces which are formed by combined patterns of the features from the second layer.
- The fourth and fifth convolutional layers detects more complex patterns.
- The output from the fifth convolutional layer is then flattened, so that it will be passed to a classifier network which consists of three fully-connected layers to classify the object from which class it belongs to.
- The final fully-connected layer output is not passed to softmax activation function, because later in the next code section a 'nn.CrossEntropyLoss()' is used as our criterion which internally perform the softmax on the output of the network and use the negative log likely-hood loss 'nn.NLLLoss()'.
- Dropout layers have been added with a 25% that each node in the fully-connected layers could be turned-off while training to increase the generalization of the trained model and avoid overfitting.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [4]: 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 [6]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
            """returns trained model"""
            # initialize tracker for minimum validation loss
            valid_loss_min = np.Inf

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

                ###################
                # train the model #
                ###################
                model.train()
                for batch_idx, (data, target) in enumerate(loaders['train']):
                    # move to GPU
                    if use_cuda:
```

```
        data, target = data.cuda(), target.cuda()
    ## find the loss and update the model parameters accordingly
    ## record the average training loss, using something like
    ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_los

    # clear the gradients of all the optimized variables
    optimizer.zero_grad()

    # forward pass
    output = model(data)

    # calculate batch loss
    loss = criterion(output, target)

    # backward pass, calculate the gradient of the loss w.r.t. model parameters
    loss.backward()

    # perform a single optimization step and update the weights
    optimizer.step()

    # update the training loss
    #train_loss += loss.item() * data.size(0)
    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))


######################
# validate the model #
######################
model.eval()
for batch_idx, (data, target) in enumerate(loaders['valid']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## update the average validation loss

    # forward pass
    output = model(data)

    # calculate batch validation loss
    loss = criterion(output, target)

    # update the validation loss
    #valid_loss += loss.item() * data.size(0)
    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

train_loss /= len(loaders['train'].dataset)
valid_loss /= len(loaders['valid'].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(f'Validation model decreased ({valid_loss_min} -> {valid_loss}) \t Sav
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    n_epochs = 30
    model_scratch = train(n_epochs, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 0.000621         Validation Loss: 0.004938
Validation model decreased (inf -> 0.004937856923788786)         Saving Model ...
Epoch: 2         Training Loss: 0.000599         Validation Loss: 0.004806
Validation model decreased (0.004937856923788786 -> 0.004805851727724075)         Saving Model
Epoch: 3         Training Loss: 0.000576         Validation Loss: 0.004733
Validation model decreased (0.004805851727724075 -> 0.004732580855488777)         Saving Model
Epoch: 4         Training Loss: 0.000557         Validation Loss: 0.004631
Validation model decreased (0.004732580855488777 -> 0.004630543291568756)         Saving Model
Epoch: 5         Training Loss: 0.000533         Validation Loss: 0.004519
Validation model decreased (0.004630543291568756 -> 0.004518547095358372)         Saving Model
Epoch: 6         Training Loss: 0.000517         Validation Loss: 0.004369
Validation model decreased (0.004518547095358372 -> 0.0043694316409528255)         Saving Model
Epoch: 7         Training Loss: 0.000497         Validation Loss: 0.004382
Epoch: 8         Training Loss: 0.000479         Validation Loss: 0.004266
Validation model decreased (0.0043694316409528255 -> 0.004265594761818647)         Saving Model
Epoch: 9         Training Loss: 0.000455         Validation Loss: 0.004163
Validation model decreased (0.004265594761818647 -> 0.00416330574080348)         Saving Model .
Epoch: 10         Training Loss: 0.000436         Validation Loss: 0.004218
Epoch: 11         Training Loss: 0.000416         Validation Loss: 0.004306
Epoch: 12         Training Loss: 0.000399         Validation Loss: 0.004302
Epoch: 13         Training Loss: 0.000378         Validation Loss: 0.004125
Validation model decreased (0.00416330574080348 -> 0.004124581813812256)         Saving Model .
Epoch: 14         Training Loss: 0.000368         Validation Loss: 0.004236
Epoch: 15         Training Loss: 0.000340         Validation Loss: 0.004189
Epoch: 16         Training Loss: 0.000337         Validation Loss: 0.004330
Epoch: 17         Training Loss: 0.000314         Validation Loss: 0.004117
```

```
Validation model decreased (0.004124581813812256 -> 0.004116714000701904)         Saving Model
Epoch: 18         Training Loss: 0.000297         Validation Loss: 0.004475
Epoch: 19         Training Loss: 0.000285         Validation Loss: 0.004511
Epoch: 20         Training Loss: 0.000268         Validation Loss: 0.004335
Epoch: 21         Training Loss: 0.000259         Validation Loss: 0.004566
Epoch: 22         Training Loss: 0.000245         Validation Loss: 0.004487
Epoch: 23         Training Loss: 0.000235         Validation Loss: 0.004666
Epoch: 24         Training Loss: 0.000219         Validation Loss: 0.004718
Epoch: 25         Training Loss: 0.000209         Validation Loss: 0.004799
Epoch: 26         Training Loss: 0.000203         Validation Loss: 0.005161
Epoch: 27         Training Loss: 0.000195         Validation Loss: 0.004811
Epoch: 28         Training Loss: 0.000181         Validation Loss: 0.004899
Epoch: 29         Training Loss: 0.000176         Validation Loss: 0.005109
Epoch: 30         Training Loss: 0.000167         Validation Loss: 0.004823
```

In [10]:
```python
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
model_scratch.eval()
```

Out[10]:
```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6400, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=500, bias=True)
  (fc3): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.2)
)
```

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 [11]:
```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))

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

```
Test Loss: 3.263612


Test Accuracy: 24% (201/836)
```

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 torch
         import torch.nn as nn
         import matplotlib.pyplot as plt
         import numpy as np
         import os
         import torchvision
         from torchvision import datasets
```

```python
import torchvision.transforms as transforms
from PIL import ImageFile
from PIL import Image

ImageFile.LOAD_TRUNCATED_IMAGES = True

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

transform = {
    'train' : transforms.Compose([
                                    transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.RandomRotation(10),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                                         std=(0.229, 0.224, 0.225))
                                 ]),
    'valid' : transforms.Compose([
                                    transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                                         std=(0.229, 0.224, 0.225))
                                 ]),
}

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 64

train_data = datasets.ImageFolder(root='/data/dog_images/train/', transform=transform['
valid_data = datasets.ImageFolder(root='/data/dog_images/valid/', transform=transform['
tests_data = datasets.ImageFolder(root='/data/dog_images/test/', transform=transform['v

n_classes = len(train_data.classes)

loaders_transfer = {
    'train' : torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worker
    'valid' : torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_worker
    'test'  : torch.utils.data.DataLoader(tests_data, batch_size=batch_size, num_worker
}
```

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

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

## TODO: Specify model architecture
# Load the pretrained model from pytorch
model_transfer = models.vgg16(pretrained=True)

# print out the model structure
print(model_transfer)
```

```
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)
  )
)
```

In [13]:
```
# Freeze training for all "features" layers
for param in model_transfer.features.parameters():
    param.requires_grad = False

## TODO: add a last linear layer  that maps n_inputs -> 5 flower classes
## new layers automatically have requires_grad = True
model_transfer.classifier[6] = torch.nn.Linear(model_transfer.classifier[6].in_features

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

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:

The selected model is the VGG. VGG is a convolutional neural network introduced by K. Simonyan and A. Zisserman from the University of Oxford. The model is trained for weeks on the ImageNet dataset which contains over 14 million images belonging to 1000 classes. The model achieved accuracy of 92.7% which is considered as top 5 test accuracy in ImageNet dataset.

The pretained model is suitable for our problem since our problem is to classify dog breeds and the network is trained on 14 million images of 1000 classes including dog images. Besides, the early layers in the pretrained model already learned common and general features which is also contributing in the images of the dataset of our problem of classifying dog breeds. Which means there is no need to train these convolutional layer from scratch again.

Basically, the VGG net is taken as it is and only the last fully-connected layer in the classifier is replaced with a new fully-connected layer that output our 133 classes of dog breeds instead of 1000 classes from the ImageNet.

The optimizer is then configured to train and update only the weights of the new fully-connected layer and leave the rest of the trained model as it is.

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

         # specify loss function (categorical cross-entropy)
         criterion_transfer = nn.CrossEntropyLoss()

         # specify optimizer (stochastic gradient descent) and learning rate = 0.001
         optimizer_transfer = optim.Adam(model_transfer.classifier[6].parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [14]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

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

                     # clear the gradients of all the optimized variables
                     optimizer.zero_grad()

                     # forward pass
                     output = model(data)

                     # calculate batch loss
                     loss = criterion(output, target)

                     # backward pass, calculate the gradient of the loss w.r.t. model parameters
                     loss.backward()

                     # perform a single optimization step and update the weights
```

```
                optimizer.step()

                # update the training loss
                #train_loss += loss.item() * data.size(0)
                train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)


            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss

                # forward pass
                output = model(data)

                # calculate batch validation loss
                loss = criterion(output, target)

                # update the validation loss
                #valid_loss += loss.item() * data.size(0)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

            train_loss /= len(loaders['train'].dataset)
            valid_loss /= len(loaders['valid'].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(f'Validation model decreased ({valid_loss_min} -> {valid_loss}) \t Sa
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model
```

```
In [15]: # train the model
         n_epochs = 10
```

```
model_transfer = train(n_epochs, loaders_transfer, model_transfer,
                       optimizer_transfer, criterion_transfer, use_cuda, 'model_transfe
```

```
Epoch: 1         Training Loss: 0.000198         Validation Loss: 0.000616
Validation model decreased (inf -> 0.0006156761664897203)         Saving Model ...
Epoch: 2         Training Loss: 0.000076         Validation Loss: 0.000523
Validation model decreased (0.0006156761664897203 -> 0.0005232847179286182)         Saving Mode
Epoch: 3         Training Loss: 0.000065         Validation Loss: 0.000488
Validation model decreased (0.0005232847179286182 -> 0.0004875543818343431)         Saving Mode
Epoch: 4         Training Loss: 0.000056         Validation Loss: 0.000472
Validation model decreased (0.0004875543818343431 -> 0.0004717969277407974)         Saving Mode
Epoch: 5         Training Loss: 0.000053         Validation Loss: 0.000483
Epoch: 6         Training Loss: 0.000046         Validation Loss: 0.000454
Validation model decreased (0.0004717969277407974 -> 0.00045415456406772137)         Saving Mod
Epoch: 7         Training Loss: 0.000044         Validation Loss: 0.000501
Epoch: 8         Training Loss: 0.000042         Validation Loss: 0.000509
Epoch: 9         Training Loss: 0.000040         Validation Loss: 0.000567
Epoch: 10         Training Loss: 0.000039         Validation Loss: 0.000500
```

```
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]: 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 [19]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.442764


Test Accuracy: 84% (708/836)
```

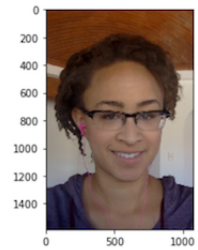
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [15]:
```
## TODO: Specify data loaders
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
import os
import torchvision
from torchvision import datasets
import torchvision.transforms as transforms
from PIL import ImageFile
from PIL import Image

ImageFile.LOAD_TRUNCATED_IMAGES = True

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

transform = {
    'train' : transforms.Compose([
                                    transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.RandomRotation(10),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                                         std=(0.229, 0.224, 0.225))
                                 ]),
    'valid' : transforms.Compose([
```

```
                                    transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                                         std=(0.229, 0.224, 0.225))
                                ]),
}

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 64

data_transfer = {
    'train' : datasets.ImageFolder(root='/data/dog_images/train/', transform=transform[
    'valid' : datasets.ImageFolder(root='/data/dog_images/valid/', transform=transform[
    'test'  : datasets.ImageFolder(root='/data/dog_images/test/', transform=transform['
}

n_classes = len(data_transfer['train'].classes)

loaders_transfer = {
    'train' : torch.utils.data.DataLoader(data_transfer['train'], batch_size=batch_size
    'valid' : torch.utils.data.DataLoader(data_transfer['valid'], batch_size=batch_size
    'test'  : torch.utils.data.DataLoader(data_transfer['test'], batch_size=batch_size,
}
```

In [16]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

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

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

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

    img = transform(img)
```



Sample Human Output

```
    img = img.unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    output = model_transfer(img)
    prediction = torch.argmax(output.data, 1)
    dog_breed = class_names[prediction - 1]

    return dog_breed
```

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 [17]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

from PIL import Image

def run_app(img_path):
    ## handle cases for a human face, dog, and neither

    img = Image.open(img_path)
    plt.imshow(img)
```

```
    plt.show()

    if (dog_detector(img_path) == True) or (face_detector(img_path) == True):
        breed = predict_breed_transfer(img_path)
        if face_detector(img_path) == True:
            print(f"He/she looks like {breed}")
        else:
            print(f"This dog is {breed}")
    else:
        print("Image doesn't contain dogs or humans")
```

Step 6: Test Your Algorithm
In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement)

1) The learning can be improved with using the Leaky-ReLU instead of ReLU which prevents from the "dying ReLU" problem when the ReLU always have values under zero (negative values). When the inputs to the ReLU is negative then the output will be zero which blocks the learning completely because of zero gradients. In Leaky-ReLU the negative part has a small slope which gives small gradient, while for the positive part the gradient is always equal to one.

2) I can try different modern architectures than the VGG-16 like Inception, ResNet, or DenseNet and compare the outputs to the VGG-16 architecture that I've used the solution.

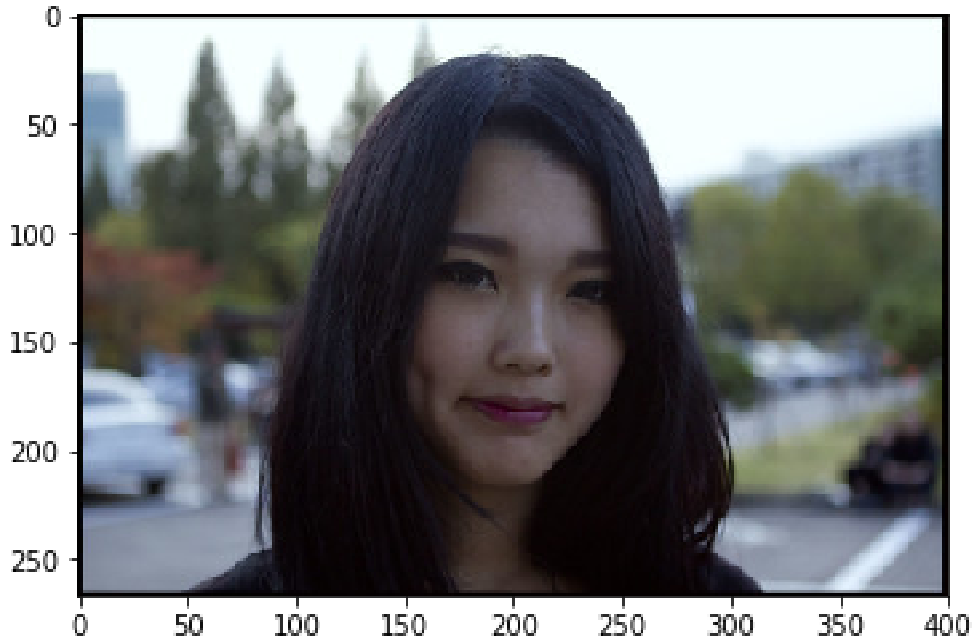
3) Adaptive learning rate optimizer can be used like ADADELTA optimizer.

In [18]:
```
human_files = np.array(glob("ext_images/humans/*"))
dog_files = np.array(glob("ext_images/dogs/*"))

print(f"There are {len(human_files)} human files")
print(f"There are {len(dog_files)} dog files")
```

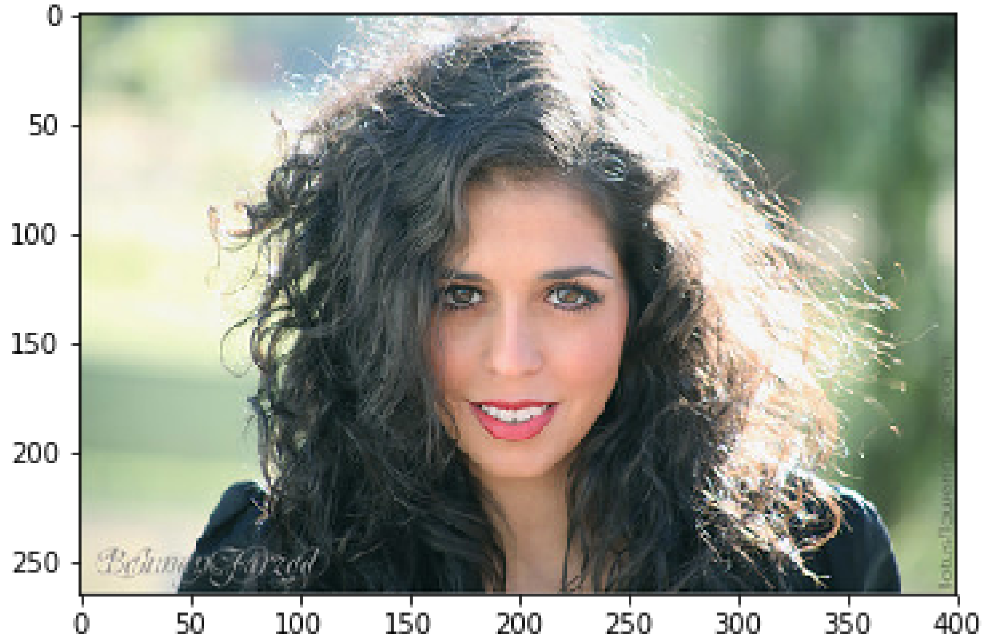
```
There are 5 human files
There are 3 dog files
```

In [19]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
```

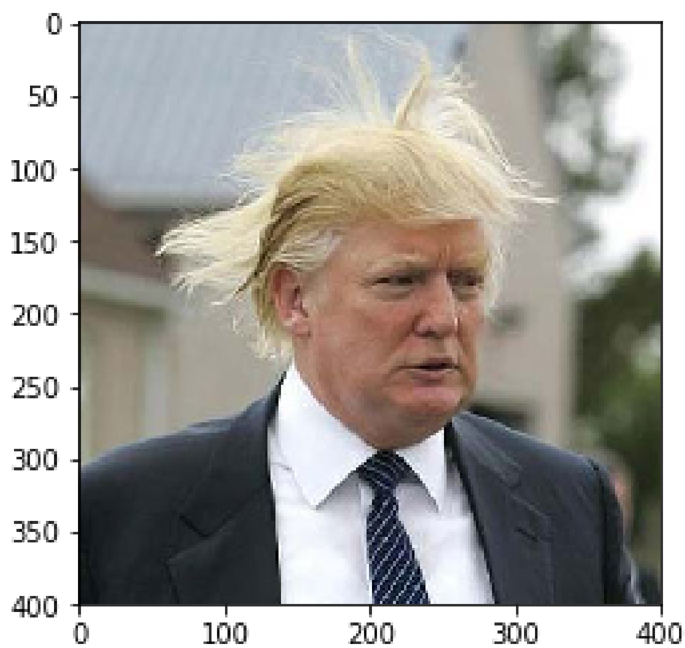
```
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```



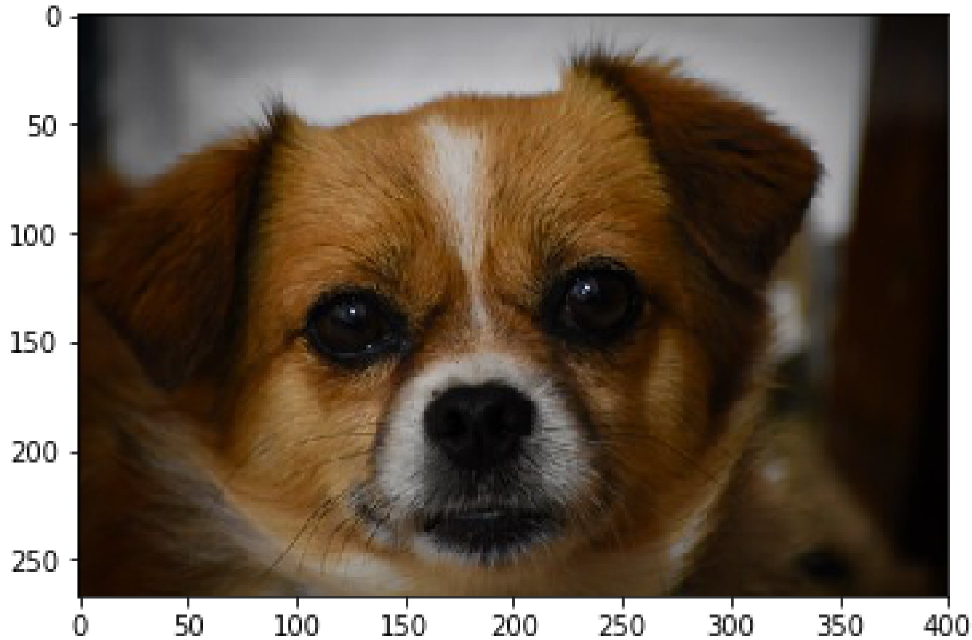
```
He/she looks like Greyhound
```



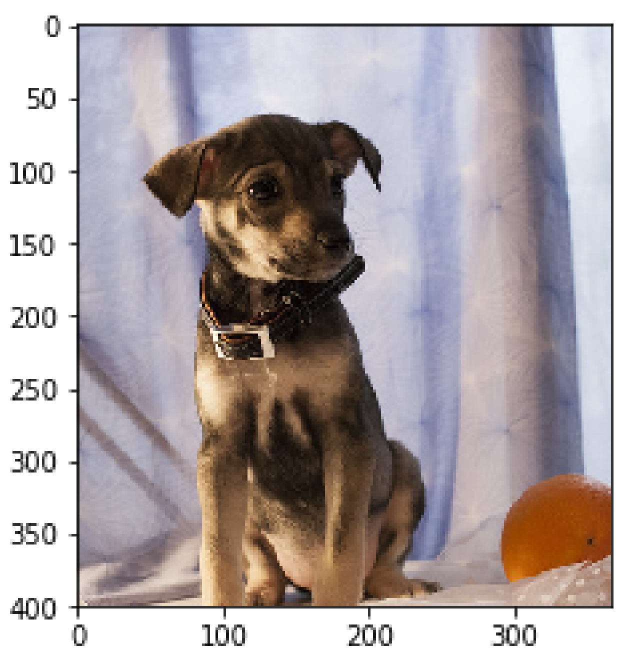
He/she looks like Affenpinscher



He/she looks like Tibetan mastiff



This dog is Norwegian elkhound



This dog is Chesapeake bay retriever

This dog is Basenji

In [ ]: