dog_app

April 6, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

```
In [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

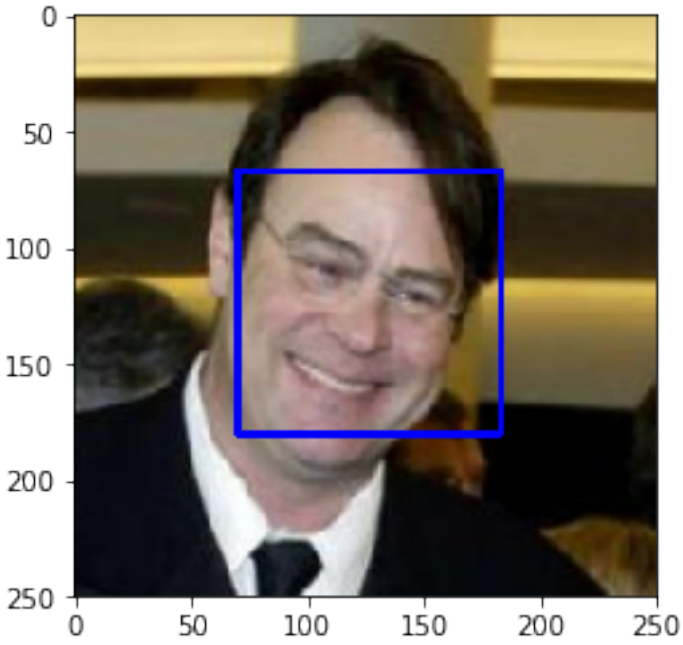
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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: Using the face_detector function, 98 human faces were detected in the first 100 images of human_files (98%) and 1 human face was detected in the first 100 images of dog_files (1%).

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

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

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

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

        for i in tqdm(range(len(human_files_short)), desc='Calculating'):
            if face_detector(human_files[i]):
                human_faces +=1
            elif face_detector(dog_files[i]):
                human_faces_dogs +=1
            sleep(0.01)
        print('The accuracy of guessing human faces from human images is %d' % (human_faces)+'%'
        print('The accuracy of guessing human faces from dog images is %d' % (human_faces_dogs)+

Calculating: 100%|| 100/100 [00:04<00:00, 21.43it/s]
```

```
The accuracy of guessing human faces from human images is 98%
The accuracy of guessing human faces from dog images is 1%
```

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

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

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 100879739.42it/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 [7]:
```
import requests

labels_url = 'https://s3.amazonaws.com/outcome-blog/imagenet/labels.json'
response = requests.get(labels_url)
labels = {}

for key, value in response.json().items():
    labels[int(key)] = value
```

In [8]:
```
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

    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])])

    image = transform(img)
    image = image.unsqueeze(0)
```

```
        output_1 = VGG16(image) #this is used to get the tensor of predictions
        output_2 = output_1.data.numpy().argmax() #this is used to determine the index of th

        return output_2
        #return labels[output_2] <-- is used to get the value from the predicted index
    print(VGG16_predict(dog_files[0]))
```

```
243
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [9]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            if VGG16_predict(img_path)>=151 and VGG16_predict(img_path)<=268:
                return True
            else:
                return False

        dog_detector(dog_files[0])
```

```
Out[9]: True
```

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 the human_files_short (or the first 100 images of human_files) no dog faces were detected --> 0%. In the dog_files_short (or the first 100 images of dog_files) 100 dog faces were detected --> 100%.

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

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

         dog_faces = 0
```

```
        dog_faces_humans = 0

        for i in tqdm(range(len(human_files_short)), desc='Calculating'):
            if dog_detector(dog_files[i]):
                dog_faces +=1
            elif face_detector(human_files[i]):
                dog_faces_humans +=1
            sleep(0.01)
        print('The accuracy of guessing dog faces from dog images is %d' % (dog_faces)+'%')
        print('The accuracy of guessing dog faces from human images is %d' % (dog_faces_humans)
```

```
Calculating: 100%|| 100/100 [02:49<00:00,  1.66s/it]

The accuracy of guessing dog faces from dog images is 100%
The accuracy of guessing dog faces from human images is 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.
```

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.

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 [12]: import os
         from torchvision import datasets

         train_transform = transforms.Compose([transforms.RandomRotation(40),
                                               transforms.RandomResizedCrop(224),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                    std=[0.229, 0.224, 0.225])]

         valid_transform = transforms.Compose([transforms.Resize(255),
                                               transforms.CenterCrop((224, 224)),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                    std=[0.229, 0.224, 0.225])]

         test_transform = transforms.Compose([transforms.Resize(255),
                                              transforms.CenterCrop((224, 224)),
                                              transforms.ToTensor(),
                                              transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                   std=[0.229, 0.224, 0.225])]

         ### TODO: Write data loaders for training, validation, and test sets
```

```
## Specify appropriate transforms, and batch_sizes
train_data = datasets.ImageFolder('/data/dog_images/train', transform=train_transform)
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=valid_transform)
test_data = datasets.ImageFolder('/data/dog_images/test', transform=test_transform)

trainloader = torch.utils.data.DataLoader(train_data, batch_size=20, shuffle=True)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=20, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=20, shuffle=True)
```

In [ ]:

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

Answer: The code resizes the images by cropping them. For validation and test the images are cropped to to squares of dimension 224. For training, the images are being cropped at random. I chose the dimension of 224 because I have seen in the lectures that this is a good starting point and the images of the dogs are already about that size. I included the normalization for each data set which is known to be good for RGB images.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [ ]:

In [13]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv1_bn = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv2_bn = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv3_bn = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
        self.conv4_bn = nn.BatchNorm2d(256)
        self.conv5 = nn.Conv2d(256, 512, 3, padding=1)

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

        self.fc1 = nn.Linear(512*7*7, 1000)
```

```
        self.fc2 = nn.Linear(1000, 133)
        #self.fc3 = nn.Linear(700, 133)

        self.dropout = nn.Dropout(p=0.6)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1_bn(self.conv1(x))))
        #x = self.dropout(x)
        x = self.pool(F.relu(self.conv2_bn(self.conv2(x))))
        #x = self.dropout(x)
        x = self.pool(F.relu(self.conv3_bn(self.conv3(x))))
        #x = self.dropout(x)
        x = self.pool(F.relu(self.conv4_bn(self.conv4(x))))
        #x = self.dropout(x)
        x = self.pool(F.relu(self.conv5(x)))
        #x = self.dropout(x)
        x = x.view(x.shape[0], -1)
        x = self.dropout(F.relu(self.fc1(x)))
        #x = self.dropout(F.relu(self.fc2(x)))
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: I included 5 convolutional layers, as I thought that the network would benefit from being able to extract more complex features from the dog images. Each convolution layer was followed by a maxpool layer. As I saw in other convolutional models the fully connected layers were not that long, therefore I stuck to 2 layers. The output of the last alayer is 133 to match the amount of dog breeds there are in this dataset. As this model is rather simple and the dataset is not that big in comparison to the dataset for the vgg16 model I decided to keep the dropout rather high to at 60% to make sure that overfitting was not an issue. BatchNorm2d was used as per recommendation to speed up the process of learning.

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 [14]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.04)
```

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 [ ]:
```
```

In [17]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
def train(n_epochs, train_loader, valid_loader, model, optimizer, criterion, use_cuda,
    """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(train_loader):
                # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()

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

            ## 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_loss)
            #train_loss += loss.item()*data.size(0)
            if batch_idx % 100 == 0:
                print('Epoch: %d     Batch: %d     Training Loss: %.6f' % (epoch, batch
```

```
            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(valid_loader):
                # 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)
                #valid_loss += loss.item * data.size(0)

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

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

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


# train the model
model_scratch = train(40, trainloader, validloader, 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     Batch: 0     Training Loss: 4.966285
Epoch: 1     Batch: 100     Training Loss: 4.878249
Epoch: 1     Batch: 200     Training Loss: 4.852528
Epoch: 1     Batch: 300     Training Loss: 4.823517
Epoch: 1         Training Loss: 4.811834         Validation Loss: 4.655429
```

```
Validation loss decreased (inf --> 4.655429).  Saving model ...
Epoch: 2     Batch: 0     Training Loss: 4.609734
Epoch: 2     Batch: 100     Training Loss: 4.672195
Epoch: 2     Batch: 200     Training Loss: 4.653183
Epoch: 2     Batch: 300     Training Loss: 4.649123
Epoch: 2         Training Loss: 4.644660         Validation Loss: 4.436154
Validation loss decreased (4.655429 --> 4.436154).  Saving model ...
Epoch: 3     Batch: 0     Training Loss: 4.492978
Epoch: 3     Batch: 100     Training Loss: 4.531199
Epoch: 3     Batch: 200     Training Loss: 4.545135
Epoch: 3     Batch: 300     Training Loss: 4.542447
Epoch: 3         Training Loss: 4.543982         Validation Loss: 4.323753
Validation loss decreased (4.436154 --> 4.323753).  Saving model ...
Epoch: 4     Batch: 0     Training Loss: 4.471242
Epoch: 4     Batch: 100     Training Loss: 4.479846
Epoch: 4     Batch: 200     Training Loss: 4.459786
Epoch: 4     Batch: 300     Training Loss: 4.460681
Epoch: 4         Training Loss: 4.455568         Validation Loss: 4.298209
Validation loss decreased (4.323753 --> 4.298209).  Saving model ...
Epoch: 5     Batch: 0     Training Loss: 4.265311
Epoch: 5     Batch: 100     Training Loss: 4.387385
Epoch: 5     Batch: 200     Training Loss: 4.378591
Epoch: 5     Batch: 300     Training Loss: 4.386944
Epoch: 5         Training Loss: 4.382299         Validation Loss: 4.548362
Epoch: 6     Batch: 0     Training Loss: 4.232441
Epoch: 6     Batch: 100     Training Loss: 4.303070
Epoch: 6     Batch: 200     Training Loss: 4.317791
Epoch: 6     Batch: 300     Training Loss: 4.319761
Epoch: 6         Training Loss: 4.319384         Validation Loss: 4.487078
Epoch: 7     Batch: 0     Training Loss: 4.284685
Epoch: 7     Batch: 100     Training Loss: 4.222863
Epoch: 7     Batch: 200     Training Loss: 4.233783
Epoch: 7     Batch: 300     Training Loss: 4.235152
Epoch: 7         Training Loss: 4.225164         Validation Loss: 4.317381
Epoch: 8     Batch: 0     Training Loss: 4.523171
Epoch: 8     Batch: 100     Training Loss: 4.161835
Epoch: 8     Batch: 200     Training Loss: 4.169507
Epoch: 8     Batch: 300     Training Loss: 4.172083
Epoch: 8         Training Loss: 4.163577         Validation Loss: 3.843909
Validation loss decreased (4.298209 --> 3.843909).  Saving model ...
Epoch: 9     Batch: 0     Training Loss: 4.121362
Epoch: 9     Batch: 100     Training Loss: 4.097316
Epoch: 9     Batch: 200     Training Loss: 4.088156
Epoch: 9     Batch: 300     Training Loss: 4.099228
Epoch: 9         Training Loss: 4.103813         Validation Loss: 3.981568
Epoch: 10     Batch: 0     Training Loss: 4.067137
Epoch: 10     Batch: 100     Training Loss: 4.022371
Epoch: 10     Batch: 200     Training Loss: 4.015722
```

```
Epoch: 10     Batch: 300     Training Loss: 4.015148
Epoch: 10         Training Loss: 4.015239         Validation Loss: 3.700208
Validation loss decreased (3.843909 --> 3.700208).  Saving model ...
Epoch: 11     Batch: 0     Training Loss: 4.269770
Epoch: 11     Batch: 100     Training Loss: 3.992631
Epoch: 11     Batch: 200     Training Loss: 3.989216
Epoch: 11     Batch: 300     Training Loss: 3.988339
Epoch: 11         Training Loss: 3.985214         Validation Loss: 3.682343
Validation loss decreased (3.700208 --> 3.682343).  Saving model ...
Epoch: 12     Batch: 0     Training Loss: 3.562852
Epoch: 12     Batch: 100     Training Loss: 3.944786
Epoch: 12     Batch: 200     Training Loss: 3.934417
Epoch: 12     Batch: 300     Training Loss: 3.911368
Epoch: 12         Training Loss: 3.906815         Validation Loss: 3.574053
Validation loss decreased (3.682343 --> 3.574053).  Saving model ...
Epoch: 13     Batch: 0     Training Loss: 4.093870
Epoch: 13     Batch: 100     Training Loss: 3.834355
Epoch: 13     Batch: 200     Training Loss: 3.850061
Epoch: 13     Batch: 300     Training Loss: 3.845371
Epoch: 13         Training Loss: 3.846137         Validation Loss: 3.554516
Validation loss decreased (3.574053 --> 3.554516).  Saving model ...
Epoch: 14     Batch: 0     Training Loss: 3.313786
Epoch: 14     Batch: 100     Training Loss: 3.793172
Epoch: 14     Batch: 200     Training Loss: 3.791937
Epoch: 14     Batch: 300     Training Loss: 3.795569
Epoch: 14         Training Loss: 3.795514         Validation Loss: 3.451510
Validation loss decreased (3.554516 --> 3.451510).  Saving model ...
Epoch: 15     Batch: 0     Training Loss: 3.586854
Epoch: 15     Batch: 100     Training Loss: 3.730677
Epoch: 15     Batch: 200     Training Loss: 3.756616
Epoch: 15     Batch: 300     Training Loss: 3.745268
Epoch: 15         Training Loss: 3.743247         Validation Loss: 3.816637
Epoch: 16     Batch: 0     Training Loss: 3.575707
Epoch: 16     Batch: 100     Training Loss: 3.697533
Epoch: 16     Batch: 200     Training Loss: 3.715468
Epoch: 16     Batch: 300     Training Loss: 3.712311
Epoch: 16         Training Loss: 3.699737         Validation Loss: 3.399422
Validation loss decreased (3.451510 --> 3.399422).  Saving model ...
Epoch: 17     Batch: 0     Training Loss: 3.037588
Epoch: 17     Batch: 100     Training Loss: 3.590598
Epoch: 17     Batch: 200     Training Loss: 3.626235
Epoch: 17     Batch: 300     Training Loss: 3.624125
Epoch: 17         Training Loss: 3.627720         Validation Loss: 3.374807
Validation loss decreased (3.399422 --> 3.374807).  Saving model ...
Epoch: 18     Batch: 0     Training Loss: 3.824654
Epoch: 18     Batch: 100     Training Loss: 3.557801
Epoch: 18     Batch: 200     Training Loss: 3.585120
Epoch: 18     Batch: 300     Training Loss: 3.603530
```

```
Epoch: 18         Training Loss: 3.600673         Validation Loss: 3.343167
Validation loss decreased (3.374807 --> 3.343167).  Saving model ...
Epoch: 19     Batch: 0     Training Loss: 3.592627
Epoch: 19     Batch: 100     Training Loss: 3.544240
Epoch: 19     Batch: 200     Training Loss: 3.542642
Epoch: 19     Batch: 300     Training Loss: 3.568863
Epoch: 19         Training Loss: 3.561457         Validation Loss: 3.313464
Validation loss decreased (3.343167 --> 3.313464).  Saving model ...
Epoch: 20     Batch: 0     Training Loss: 3.286080
Epoch: 20     Batch: 100     Training Loss: 3.508519
Epoch: 20     Batch: 200     Training Loss: 3.534178
Epoch: 20     Batch: 300     Training Loss: 3.536244
Epoch: 20         Training Loss: 3.525450         Validation Loss: 3.388129
Epoch: 21     Batch: 0     Training Loss: 3.318735
Epoch: 21     Batch: 100     Training Loss: 3.391604
Epoch: 21     Batch: 200     Training Loss: 3.440175
Epoch: 21     Batch: 300     Training Loss: 3.458564
Epoch: 21         Training Loss: 3.453221         Validation Loss: 3.028087
Validation loss decreased (3.313464 --> 3.028087).  Saving model ...
Epoch: 22     Batch: 0     Training Loss: 3.411294
Epoch: 22     Batch: 100     Training Loss: 3.426133
Epoch: 22     Batch: 200     Training Loss: 3.410539
Epoch: 22     Batch: 300     Training Loss: 3.411329
Epoch: 22         Training Loss: 3.411856         Validation Loss: 3.301156
Epoch: 23     Batch: 0     Training Loss: 3.908358
Epoch: 23     Batch: 100     Training Loss: 3.281500
Epoch: 23     Batch: 200     Training Loss: 3.321533
Epoch: 23     Batch: 300     Training Loss: 3.337882
Epoch: 23         Training Loss: 3.343824         Validation Loss: 2.958780
Validation loss decreased (3.028087 --> 2.958780).  Saving model ...
Epoch: 24     Batch: 0     Training Loss: 3.278238
Epoch: 24     Batch: 100     Training Loss: 3.367344
Epoch: 24     Batch: 200     Training Loss: 3.345165
Epoch: 24     Batch: 300     Training Loss: 3.322013
Epoch: 24         Training Loss: 3.314184         Validation Loss: 2.926057
Validation loss decreased (2.958780 --> 2.926057).  Saving model ...
Epoch: 25     Batch: 0     Training Loss: 3.326211
Epoch: 25     Batch: 100     Training Loss: 3.306420
Epoch: 25     Batch: 200     Training Loss: 3.278814
Epoch: 25     Batch: 300     Training Loss: 3.291080
Epoch: 25         Training Loss: 3.298996         Validation Loss: 2.910838
Validation loss decreased (2.926057 --> 2.910838).  Saving model ...
Epoch: 26     Batch: 0     Training Loss: 3.975695
Epoch: 26     Batch: 100     Training Loss: 3.183485
Epoch: 26     Batch: 200     Training Loss: 3.233075
Epoch: 26     Batch: 300     Training Loss: 3.243652
Epoch: 26         Training Loss: 3.245974         Validation Loss: 2.967634
Epoch: 27     Batch: 0     Training Loss: 3.251649
```

```
Epoch: 27     Batch: 100     Training Loss: 3.169667
Epoch: 27     Batch: 200     Training Loss: 3.185458
Epoch: 27     Batch: 300     Training Loss: 3.181858
Epoch: 27         Training Loss: 3.177668         Validation Loss: 2.716637
Validation loss decreased (2.910838 --> 2.716637).  Saving model ...
Epoch: 28     Batch: 0     Training Loss: 2.907328
Epoch: 28     Batch: 100     Training Loss: 3.137556
Epoch: 28     Batch: 200     Training Loss: 3.120748
Epoch: 28     Batch: 300     Training Loss: 3.148235
Epoch: 28         Training Loss: 3.145641         Validation Loss: 2.856136
Epoch: 29     Batch: 0     Training Loss: 3.033786
Epoch: 29     Batch: 100     Training Loss: 3.075925
Epoch: 29     Batch: 200     Training Loss: 3.075857
Epoch: 29     Batch: 300     Training Loss: 3.083342
Epoch: 29         Training Loss: 3.095368         Validation Loss: 2.660051
Validation loss decreased (2.716637 --> 2.660051).  Saving model ...
Epoch: 30     Batch: 0     Training Loss: 3.274188
Epoch: 30     Batch: 100     Training Loss: 2.998957
Epoch: 30     Batch: 200     Training Loss: 3.047685
Epoch: 30     Batch: 300     Training Loss: 3.058614
Epoch: 30         Training Loss: 3.062048         Validation Loss: 2.733025
Epoch: 31     Batch: 0     Training Loss: 3.060052
Epoch: 31     Batch: 100     Training Loss: 2.993610
Epoch: 31     Batch: 200     Training Loss: 3.014415
Epoch: 31     Batch: 300     Training Loss: 3.016309
Epoch: 31         Training Loss: 3.025643         Validation Loss: 2.916820
Epoch: 32     Batch: 0     Training Loss: 2.999474
Epoch: 32     Batch: 100     Training Loss: 2.949792
Epoch: 32     Batch: 200     Training Loss: 2.979971
Epoch: 32     Batch: 300     Training Loss: 2.970576
Epoch: 32         Training Loss: 2.969559         Validation Loss: 2.788739
Epoch: 33     Batch: 0     Training Loss: 3.350197
Epoch: 33     Batch: 100     Training Loss: 2.914400
Epoch: 33     Batch: 200     Training Loss: 2.936454
Epoch: 33     Batch: 300     Training Loss: 2.937541
Epoch: 33         Training Loss: 2.942057         Validation Loss: 2.749985
Epoch: 34     Batch: 0     Training Loss: 3.224390
Epoch: 34     Batch: 100     Training Loss: 2.866598
Epoch: 34     Batch: 200     Training Loss: 2.879781
Epoch: 34     Batch: 300     Training Loss: 2.902619
Epoch: 34         Training Loss: 2.910102         Validation Loss: 2.502789
Validation loss decreased (2.660051 --> 2.502789).  Saving model ...
Epoch: 35     Batch: 0     Training Loss: 2.386970
Epoch: 35     Batch: 100     Training Loss: 2.819673
Epoch: 35     Batch: 200     Training Loss: 2.880972
Epoch: 35     Batch: 300     Training Loss: 2.893286
Epoch: 35         Training Loss: 2.893918         Validation Loss: 2.516446
Epoch: 36     Batch: 0     Training Loss: 2.683756
```

```
Epoch: 36     Batch: 100     Training Loss: 2.854438
Epoch: 36     Batch: 200     Training Loss: 2.841537
Epoch: 36     Batch: 300     Training Loss: 2.857388
Epoch: 36         Training Loss: 2.863426         Validation Loss: 2.461687
Validation loss decreased (2.502789 --> 2.461687).  Saving model ...
Epoch: 37     Batch: 0     Training Loss: 2.850345
Epoch: 37     Batch: 100     Training Loss: 2.819192
Epoch: 37     Batch: 200     Training Loss: 2.842704
Epoch: 37     Batch: 300     Training Loss: 2.809844
Epoch: 37         Training Loss: 2.799310         Validation Loss: 2.976945
Epoch: 38     Batch: 0     Training Loss: 3.220532
Epoch: 38     Batch: 100     Training Loss: 2.840192
Epoch: 38     Batch: 200     Training Loss: 2.805707
Epoch: 38     Batch: 300     Training Loss: 2.795623
Epoch: 38         Training Loss: 2.785834         Validation Loss: 2.378316
Validation loss decreased (2.461687 --> 2.378316).  Saving model ...
Epoch: 39     Batch: 0     Training Loss: 2.174507
Epoch: 39     Batch: 100     Training Loss: 2.771411
Epoch: 39     Batch: 200     Training Loss: 2.755577
Epoch: 39     Batch: 300     Training Loss: 2.756057
Epoch: 39         Training Loss: 2.752630         Validation Loss: 2.413177
Epoch: 40     Batch: 0     Training Loss: 3.116024
Epoch: 40     Batch: 100     Training Loss: 2.690379
Epoch: 40     Batch: 200     Training Loss: 2.688097
Epoch: 40     Batch: 300     Training Loss: 2.692736
Epoch: 40         Training Loss: 2.698986         Validation Loss: 2.270658
Validation loss decreased (2.378316 --> 2.270658).  Saving model ...
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [17]: model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
In [14]: def test(test_loader, 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(test_loader):
                 # 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(testloader, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.249493


Test Accuracy: 41% (343/836)
```

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 [15]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets

train_transform = transforms.Compose([transforms.RandomRotation(40),
                                      transforms.RandomResizedCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.406],
```

```
                                                                 std=[0.229, 0.224, 0.225])]

        valid_transform = transforms.Compose([transforms.Resize(255),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                   std=[0.229, 0.224, 0.225])]

        test_transform = transforms.Compose([transforms.Resize(255),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                  std=[0.229, 0.224, 0.225])]

        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        train_data = datasets.ImageFolder('/data/dog_images/train', transform=train_transform)
        valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=valid_transform)
        test_data = datasets.ImageFolder('/data/dog_images/test', transform=test_transform)

        trainloader = torch.utils.data.DataLoader(train_data, batch_size=20, shuffle=True)
        validloader = torch.utils.data.DataLoader(valid_data, batch_size=20, shuffle=True)
        testloader = torch.utils.data.DataLoader(test_data, batch_size=20, shuffle=True)
```

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 [16]:
```
        import torchvision.models as models
        import torch.nn as nn

        ## TODO: Specify model architecture
        model_transfer = models.vgg16(pretrained=True)

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

        final_layer = nn.Linear(model_transfer.classifier[6].in_features, 133)

        model_transfer.classifier[6] = final_layer


        if use_cuda:
            model_transfer = model_transfer.cuda()

        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=133, bias=True)
  )
)
```

In [ ]:

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

Answer: I froze the weights and 'grad' of the vgg16 model to lessen the computational effort that the network would have to make during training. I froze them because I wanted to take benefit of the already trained vgg16 model and therefore I would only need to train the model on the final linear layer. I replaced the final linear layer's output to match the 133 dog breed classes.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [18]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.003)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [24]: # train the model
         model_transfer = train(10, trainloader, validloader, model_transfer, optimizer_transfer

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

```
Epoch: 1     Batch: 0     Training Loss: 1.385076
Epoch: 1     Batch: 100     Training Loss: 1.349122
Epoch: 1     Batch: 200     Training Loss: 1.355116
Epoch: 1     Batch: 300     Training Loss: 1.330011
Epoch: 1         Training Loss: 1.326728         Validation Loss: 0.515757
Validation loss decreased (inf --> 0.515757).  Saving model ...
Epoch: 2     Batch: 0     Training Loss: 0.744087
Epoch: 2     Batch: 100     Training Loss: 1.206233
Epoch: 2     Batch: 200     Training Loss: 1.173480
Epoch: 2     Batch: 300     Training Loss: 1.186587
Epoch: 2         Training Loss: 1.196120         Validation Loss: 0.461714
Validation loss decreased (0.515757 --> 0.461714).  Saving model ...
Epoch: 3     Batch: 0     Training Loss: 1.531485
Epoch: 3     Batch: 100     Training Loss: 1.113699
Epoch: 3     Batch: 200     Training Loss: 1.143330
Epoch: 3     Batch: 300     Training Loss: 1.150393
Epoch: 3         Training Loss: 1.136868         Validation Loss: 0.436017
Validation loss decreased (0.461714 --> 0.436017).  Saving model ...
Epoch: 4     Batch: 0     Training Loss: 0.895819
Epoch: 4     Batch: 100     Training Loss: 1.102379
Epoch: 4     Batch: 200     Training Loss: 1.119950
Epoch: 4     Batch: 300     Training Loss: 1.126592
Epoch: 4         Training Loss: 1.117489         Validation Loss: 0.406498
Validation loss decreased (0.436017 --> 0.406498).  Saving model ...
Epoch: 5     Batch: 0     Training Loss: 0.853308
```

```
Epoch: 5     Batch: 100     Training Loss: 1.061306
Epoch: 5     Batch: 200     Training Loss: 1.062114
Epoch: 5     Batch: 300     Training Loss: 1.057459
Epoch: 5         Training Loss: 1.063095         Validation Loss: 0.416118
Epoch: 6     Batch: 0     Training Loss: 0.997930
Epoch: 6     Batch: 100     Training Loss: 1.054250
Epoch: 6     Batch: 200     Training Loss: 1.045308
Epoch: 6     Batch: 300     Training Loss: 1.047019
Epoch: 6         Training Loss: 1.045469         Validation Loss: 0.384031
Validation loss decreased (0.406498 --> 0.384031).  Saving model ...
Epoch: 7     Batch: 0     Training Loss: 1.144775
Epoch: 7     Batch: 100     Training Loss: 1.006497
Epoch: 7     Batch: 200     Training Loss: 1.008925
Epoch: 7     Batch: 300     Training Loss: 1.019615
Epoch: 7         Training Loss: 1.024048         Validation Loss: 0.380825
Validation loss decreased (0.384031 --> 0.380825).  Saving model ...
Epoch: 8     Batch: 0     Training Loss: 0.356323
Epoch: 8     Batch: 100     Training Loss: 0.933451
Epoch: 8     Batch: 200     Training Loss: 0.938369
Epoch: 8     Batch: 300     Training Loss: 0.963836
Epoch: 8         Training Loss: 0.961101         Validation Loss: 0.395749
Epoch: 9     Batch: 0     Training Loss: 1.217564
Epoch: 9     Batch: 100     Training Loss: 0.936789
Epoch: 9     Batch: 200     Training Loss: 0.937767
Epoch: 9     Batch: 300     Training Loss: 0.946258
Epoch: 9         Training Loss: 0.949490         Validation Loss: 0.373203
Validation loss decreased (0.380825 --> 0.373203).  Saving model ...
Epoch: 10     Batch: 0     Training Loss: 0.991447
Epoch: 10     Batch: 100     Training Loss: 0.941984
Epoch: 10     Batch: 200     Training Loss: 0.908232
Epoch: 10     Batch: 300     Training Loss: 0.918101
Epoch: 10         Training Loss: 0.918873         Validation Loss: 0.378162
```

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

```
Test Loss: 0.428837


Test Accuracy: 86% (722/836)
```

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.

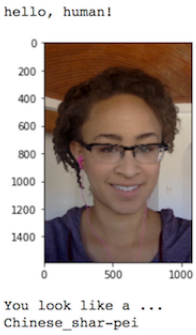
```
In [20]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

         # 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 train_data.classes]

         def predict_breed_transfer(img_path):
             img = Image.open(img_path).convert('RGB')

             transform = transforms.Compose([
                 transforms.Resize(255),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                      std=[0.229, 0.224, 0.225])])


             image = transform(img)[:3,:,:].unsqueeze(0)

             if use_cuda:
                 image = image.cuda()

             model_transfer.eval()

             with torch.no_grad():
                 output = model_transfer(image)
                 _, prediction = torch.max(output, 1)

             model_transfer.train()

             return class_names[prediction.item()] #<-- is used to get the value from the predic
```

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!



Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [21]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             if dog_detector(img_path):
                 plt.imshow(Image.open(img_path))
                 dog_breed = predict_breed_transfer(img_path)
                 print('That looks like a dog...')
                 plt.show()
                 print(f'This dog looks like a {dog_breed}')

             elif face_detector(img_path) and not dog_detector(img_path):
                 plt.imshow(Image.open(img_path))
                 dog_breed = predict_breed_transfer(img_path)
                 print('That looks like a person...')
                 plt.show()
                 print(f'This person resembles a {dog_breed}')

             #elif not dog_detector(img_path) and not face_detector(img_path):
             else:
                 print('It seems like you put an image that\'s neither a dog or a human! Try aga
```

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.

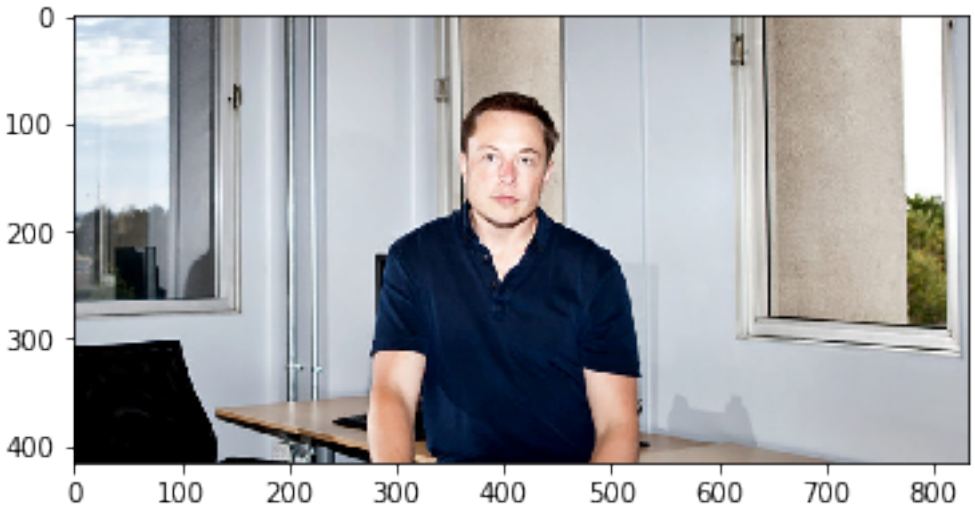
Answer: This is exactly what I expected. To be honest I'm not a dog expert and from running a search on the dog breeds that the model outputs to the given images, everything looks correct!

But for sure I would still like to improve on the following points: 1. hyperparameter tuning 2. probably a larger amount of images to train/validate/test with 3. change architecture to improve accuracy

```
In [29]: ## 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.

         my_pics = np.array(glob("./images/MyFolder/*"))


         ## suggested code, below
         for file in my_pics:
             run_app(file)
```

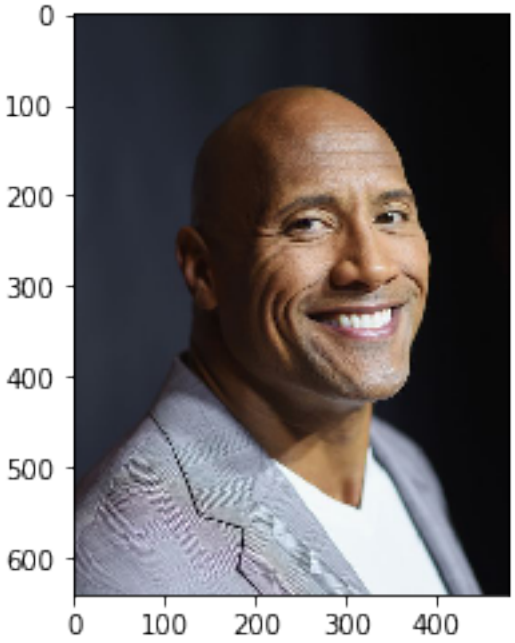
```
That looks like a dog...
```


```
This dog looks like a German shepherd dog
That looks like a person...
```

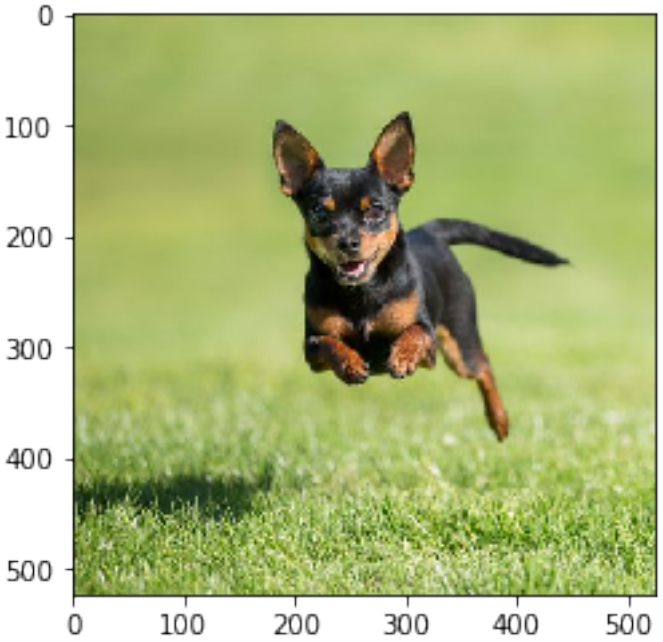



```
This person resembles a Dachshund
That looks like a person...
```

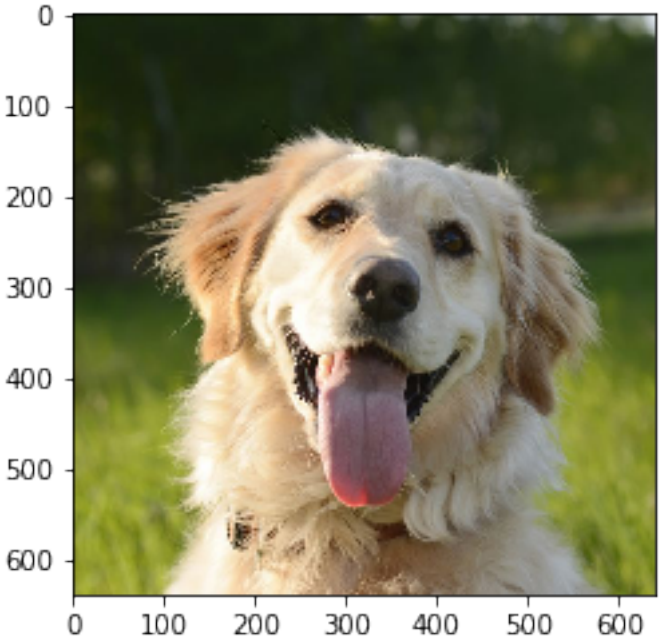

```
This person resembles a Pharaoh hound
That looks like a person...
```




```
This person resembles a Dogue de bordeaux
That looks like a dog...
```



```
This dog looks like a Chihuahua
That looks like a dog...
```




```
This dog looks like a Golden retriever
```

In [ ]: