dog_app

February 25, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.
## Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.

- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

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

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

```
In [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.
```

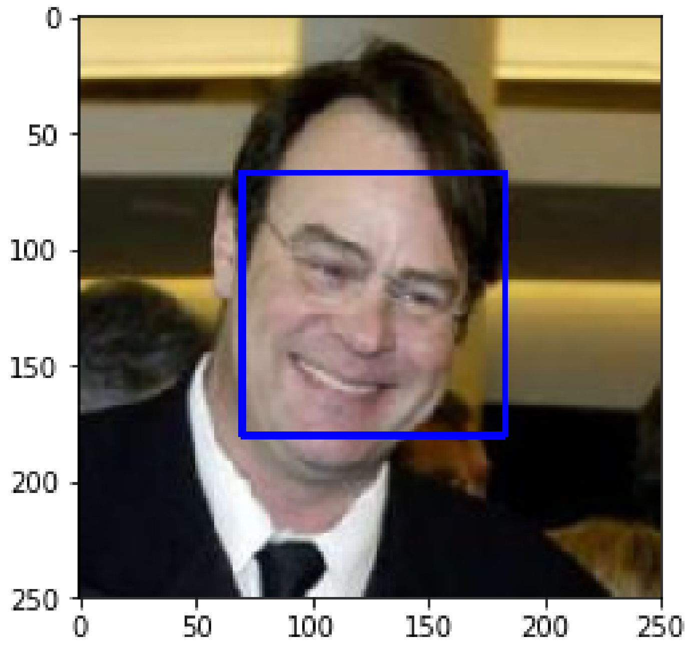
## Step 1: Detect Humans

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: Face in human_files_short: 98 / 100 Face in dog_files_short: 17 / 100

```
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.
        def test_performance_face_detector(files):
            count = 0
            total = len(files)
            for file in files:
                count += face_detector(file)
            return count/total
```

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.
        percent = 0
        percent = test_performance_face_detector(human_files_short)
        print("Face in human_files_short: {}%".format(percent * 100))
        percent = test_performance_face_detector(dog_files_short)
        print("Face in dog_files_short: {}%".format(percent * 100))
```

```
Face in human_files_short: 98.0%
Face in dog_files_short: 17.0%
```

## 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 [00:04<00:00, 118877603.33it/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]: 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

    #Loading and processing the image
    image = Image.open(img_path).convert('RGB')
    in_transform = transforms.Compose([
                                transforms.Resize(size = (224, 224)),
                                transforms.ToTensor()])

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

    #Image index generation
    if use_cuda:
        image = image.cuda()
    returnIndex = VGG16(image)

    returnIndex = torch.max(returnIndex, 1)[1].item()

    return returnIndex # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            dog_index = VGG16_predict(img_path)
            detected = dog_index > 150 and dog_index < 269
            return detected
```

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: Dog in human_files_short: 0 / 100 Dog in dog_files_short: 92 / 100

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        def test_performance_dog_detector(files):
            count = 0
            total = len(files)
            for file in files:
                count += dog_detector(file)
            return count/total
```

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

```
In [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
         percent = 0
         percent = test_performance_dog_detector(human_files_short)
         print("Dog in human_files_short: {}%".format(percent * 100))
         percent = test_performance_dog_detector(dog_files_short)
         print("Dog in dog_files_short: {}%".format(percent * 100))
```

```
Dog in human_files_short: 0.0%
Dog in dog_files_short: 94.0%
```

## 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 [11]: import os
         import torch
         from torchvision import datasets, transforms

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

         data_dir = '/data/dog_images'

         train_transforms = transforms.Compose([transforms.Resize(size=258),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomRotation(10),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.5, 0.5, 0.5],
                                                                    [0.5, 0.5, 0.5])])

         valid_transforms = transforms.Compose([transforms.Resize(size=258),
                                                   transforms.CenterCrop(224),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize([0.5, 0.5, 0.5],
                                                                        [0.5, 0.5, 0.5])])
```

```
test_transforms = transforms.Compose([transforms.Resize(size=258),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize([0.5, 0.5, 0.5],
                                                                [0.5, 0.5, 0.5])])

train_dataset = datasets.ImageFolder(os.path.join(data_dir, 'train'), transform=train_t
valid_dataset = datasets.ImageFolder(os.path.join(data_dir, 'valid'), transform=valid_t
test_dataset = datasets.ImageFolder(os.path.join(data_dir, 'test'), transform=test_tran

trainLoader = torch.utils.data.DataLoader(train_dataset,
                                          batch_size=128,
                                          shuffle=True,
                                          num_workers=0)

validLoader = torch.utils.data.DataLoader(valid_dataset,
                                          batch_size=128,
                                          shuffle=True,
                                          num_workers=0)

testLoader = torch.utils.data.DataLoader(test_dataset,
                                         batch_size=64,
                                         shuffle=False,
                                         num_workers=0)
```

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:

First of all I set up the directory information, in order to access all the images, and then created specific directory variables for each of the train, valid and test data.

Next I defined the data transforms, which transform the data into various forms, as specified. First of all for the training data, the images are resized into a single size, as the original images may vary in many different sizes. Secondly as the input image may be in any form, we need to train our model with all type of images. So we horizontally flip the images and also rotate them randomly by 10 degrees. This generalises our model and also increases the amount of dataset for our model as the same image is presented as 3 images. Finally the image is converted to a tensor and then normalised with a mean and standard deviation of 0.5.

Then I created the data loaders, which are made by first making ImageFolders and tehn accessing them via the DataLoader. Batch size is kept to be 128 for training and validation data and 64 for the test data. For the training data, the images are shuffled from the dtaset, so that the model generalises well.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: import torch.nn as nn
         import torch.nn.functional as F

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

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## CNN layers
                 self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
                 self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
                 self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
                 self.conv4 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
                 self.conv5 = nn.Conv2d(512, 512, kernel_size=3, padding=1)

                 #Max Pool layer
                 self.pool = nn.MaxPool2d(2, 2)

                 #Fully connected layers
                 self.fc1 = nn.Linear(25088, 512)
                 self.fc2 = nn.Linear(512, 512)
                 self.fc3 = nn.Linear(512, 133)

                 #Dropout layer
                 self.dropout = nn.Dropout(0.5)

             def forward(self, x):
                 ## Define forward behavior
                 x = F.relu(self.conv1(x))
                 x = self.pool(x)
                 x = F.relu(self.conv2(x))
                 x = self.pool(x)
                 x = F.relu(self.conv3(x))
                 x = self.pool(x)
                 x = F.relu(self.conv4(x))
                 x = self.pool(x)
                 x = F.relu(self.conv5(x))
                 x = self.pool(x)

                 #Flatteing the images
                 x = x.view(-1, 25088)

                 #Fully connected layers
                 x = self.fc1(x)
                 x = F.relu(x)
```

```
            x = self.dropout(x)
            x = self.fc2(x)
            x = F.relu(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()
```

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

Answer:

In this model, I decided to keep 5 convolutional layers , a kernel size of 3x3, and a padding of 1. After every convolutional layer, there is a maxpool layer using a 2x2 kernel and stride equal to 2, that reduces the size of each feature map by the factor of 0.5. After the processing of the 5 convolutional layers and maxpool layers after each Conv2d layer, we now have 512 7x7 feature maps.

Feature maps are then flattened to a vector and passed into the fully connected (FC) layers for classification. I took 3 Fully Connected layers, the first one acting as the input layer, the second onw acting as the hidden layer, and the last one acts as the output layer, which consists of our final classification result. Since there are 133 types of dog species, the output layer consists of 133 nodes.

Further I will be using CrossEntropyLoss, which is combination of NLLLoss and the log_softmax() funtion. As a result, I dont have to apply the softmax function or any other activation function to the output layer

Also, I will be using Adam Optimiser with a learning rate of 0.001

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 [13]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
```

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 [9]: from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        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 gradients of all optimized variables
                    optimizer.zero_grad()

                    # forwarward pass
                    output = model(data)

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

                    # backward pass
                    loss.backward()

                    # perform optimization step
                    optimizer.step()

                    # update training 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
                with torch.no_grad():
                    output = model(data)
                loss = criterion(output, target)
                valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

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

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

        # return trained model
        return model


    # define loaders_scratch
    loaders_scratch = {'train': trainLoader,
                       'valid': validLoader,
                       'test': testLoader}

    # train the model
    model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.884393         Validation Loss: 4.869245
Validation loss decreased (inf --> 4.869245). Saving model...
Epoch: 2         Training Loss: 4.854156         Validation Loss: 4.810519
Validation loss decreased (4.869245 --> 4.810519). Saving model...
Epoch: 3         Training Loss: 4.721883         Validation Loss: 4.658133
Validation loss decreased (4.810519 --> 4.658133). Saving model...
```

```
Epoch: 4        Training Loss: 4.614090        Validation Loss: 4.566344
Validation loss decreased (4.658133 --> 4.566344). Saving model...
Epoch: 5        Training Loss: 4.562675        Validation Loss: 4.530954
Validation loss decreased (4.566344 --> 4.530954). Saving model...
Epoch: 6        Training Loss: 4.500739        Validation Loss: 4.491057
Validation loss decreased (4.530954 --> 4.491057). Saving model...
Epoch: 7        Training Loss: 4.439700        Validation Loss: 4.413995
Validation loss decreased (4.491057 --> 4.413995). Saving model...
Epoch: 8        Training Loss: 4.408467        Validation Loss: 4.410840
Validation loss decreased (4.413995 --> 4.410840). Saving model...
Epoch: 9        Training Loss: 4.349395        Validation Loss: 4.338177
Validation loss decreased (4.410840 --> 4.338177). Saving model...
Epoch: 10        Training Loss: 4.292171        Validation Loss: 4.403984
Epoch: 11        Training Loss: 4.230017        Validation Loss: 4.242815
Validation loss decreased (4.338177 --> 4.242815). Saving model...
Epoch: 12        Training Loss: 4.169528        Validation Loss: 4.157911
Validation loss decreased (4.242815 --> 4.157911). Saving model...
Epoch: 13        Training Loss: 4.097095        Validation Loss: 4.229656
Epoch: 14        Training Loss: 4.016327        Validation Loss: 4.070597
Validation loss decreased (4.157911 --> 4.070597). Saving model...
Epoch: 15        Training Loss: 3.951644        Validation Loss: 4.008198
Validation loss decreased (4.070597 --> 4.008198). Saving model...
Epoch: 16        Training Loss: 3.887952        Validation Loss: 3.947999
Validation loss decreased (4.008198 --> 3.947999). Saving model...
Epoch: 17        Training Loss: 3.790622        Validation Loss: 3.841542
Validation loss decreased (3.947999 --> 3.841542). Saving model...
Epoch: 18        Training Loss: 3.731617        Validation Loss: 3.834623
Validation loss decreased (3.841542 --> 3.834623). Saving model...
Epoch: 19        Training Loss: 3.650102        Validation Loss: 3.764687
Validation loss decreased (3.834623 --> 3.764687). Saving model...
Epoch: 20        Training Loss: 3.596928        Validation Loss: 3.759065
Validation loss decreased (3.764687 --> 3.759065). Saving model...
Epoch: 21        Training Loss: 3.535342        Validation Loss: 3.741863
Validation loss decreased (3.759065 --> 3.741863). Saving model...
Epoch: 22        Training Loss: 3.493925        Validation Loss: 3.756743
Epoch: 23        Training Loss: 3.454011        Validation Loss: 3.773289
Epoch: 24        Training Loss: 3.389601        Validation Loss: 3.652301
Validation loss decreased (3.741863 --> 3.652301). Saving model...
Epoch: 25        Training Loss: 3.294478        Validation Loss: 3.612253
Validation loss decreased (3.652301 --> 3.612253). Saving model...
```

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

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]: 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
                 output = F.softmax(output, dim=1)
                 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: 3.561043


Test Accuracy: 13% (117/836)
```

## 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]: # Importing all necessary modules for step 4 of the project

         import os
         import numpy as np
         from PIL import Image
         import torch
         import torch.nn as nn
         import torch.nn.functional as F
         import torch.optim as optim
         from torchvision import datasets, transforms, models

         # TODO: Specify data loaders

         data_dir = '/data/dog_images'

         train_transforms = transforms.Compose([transforms.Resize(size=258),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomRotation(10),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.406],
                                                                    [0.229, 0.224, 0.225])])

         valid_transforms = transforms.Compose([transforms.Resize(size=258),
                                                   transforms.CenterCrop(224),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize([0.485, 0.456, 0.406],
                                                                    [0.229, 0.224, 0.225])])

         test_transforms = transforms.Compose([transforms.Resize(size=258),
                                                   transforms.CenterCrop(224),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize([0.485, 0.456, 0.406],
                                                                    [0.229, 0.224, 0.225])])

         train_dataset = datasets.ImageFolder(os.path.join(data_dir, 'train'), transform=train_t
         valid_dataset = datasets.ImageFolder(os.path.join(data_dir, 'valid'), transform=valid_t
test_dataset = datasets.ImageFolder(os.path.join(data_dir, 'test'), transform=test_tran

trainLoader = torch.utils.data.DataLoader(train_dataset,
                                          batch_size=128,
                                          shuffle=True,
                                          num_workers=0)

validLoader = torch.utils.data.DataLoader(valid_dataset,
                                          batch_size=128,
                                          shuffle=True,
                                          num_workers=0)

testLoader = torch.utils.data.DataLoader(test_dataset,
                                          batch_size=16,
                                          shuffle=False,
                                          num_workers=0)
```

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]: ## TODO: Specify model architecture

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

         # download VGG16 pretrained model
         model_transfer = models.vgg16(pretrained=True)

         # Freeze parameters of the model to avoid brackpropagation
         for param in model_transfer.parameters():
             param.requires_grad = False

         # get the number of dog classes from the train_dataset
         number_of_dog_classes = 133

         # Define dog breed classifier part of model_transfer
         classifier = nn.Sequential(nn.Linear(7*7*512, 4096),
                                    nn.ReLU(),
                                    nn.Dropout(0.3),
                                    nn.Linear(4096, 512),
                                    nn.ReLU(),
                                    nn.Dropout(0.3),
                                    nn.Linear(512, number_of_dog_classes))

         # Rplace the original classifier with the dog breed classifier from above
         model_transfer.classifier = classifier
```

```
        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:

I have used the VGG16 pretrained model for the transfer learning part of this notebook. As the VGG16 network is pretraiend trained on dogs, it'll help us predict the results better as it would contain much information priorly.

This is the reason for using the entire feature extraction portion from the VGG16 pretrained model with keeping the weights constant, so that I can avoid overfitting when training on a small dataset.

The drog breed classifier is modeled along the original VGG16 calssifier with 3 fully connected layers 2 dropout layers to reduce the number of parameters and then 2 ReLU activations.

The number of nodes per fully connected layer was changed to resemble the 133 dog species

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 [17]: criterion_transfer = nn.CrossEntropyLoss()

         # only train the classifier! -> model_transfer.classifier.parameters()
         optimizer_transfer = optim.Adam(model_transfer.classifier.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 [9]: from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):

            valid_loss_min = np.Inf

            print(f"Batch Size: {loaders['train'].batch_size}\n")

            for epoch in range(1, n_epochs+1):
                train_loss = 0.0
                valid_loss = 0.0

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

```
                    data, target = data.cuda(), target.cuda()

                optimizer.zero_grad()
                output = model(data)
                loss = criterion(output, target)
                loss.backward()
                optimizer.step()
                train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

                if (batch_idx + 1) % 5 == 0:
                    print(f'Epoch:{epoch}/{n_epochs} \tBatch:{batch_idx + 1}')
                    print(f'Train Loss: {train_loss}\n')

            # validate the model
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                with torch.no_grad():
                    output = model(data)
                loss = criterion(output, target)
                valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch,
                train_loss,
                valid_loss
                ))
              # save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model...'.forma
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model


    # define loaders_transfer
    loaders_transfer = {'train': trainLoader,
                        'valid': validLoader,
                        'test': testLoader}

    model_transfer = train(15, loaders_transfer, model_transfer, optimizer_transfer,
                           criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Batch Size: 128
```

```
Epoch:1/15        Batch:5
Train Loss: 1.427089810371399

Epoch:1/15        Batch:10
Train Loss: 1.4388024806976318

Epoch:1/15        Batch:15
Train Loss: 1.4217954874038696

Epoch:1/15        Batch:20
Train Loss: 1.4295933246612549

Epoch:1/15        Batch:25
Train Loss: 1.3971494436264038

Epoch:1/15        Batch:30
Train Loss: 1.3964941501617432

Epoch:1/15        Batch:35
Train Loss: 1.3696553707122803

Epoch:1/15        Batch:40
Train Loss: 1.355819821357727

Epoch:1/15        Batch:45
Train Loss: 1.338710904121399

Epoch:1/15        Batch:50
Train Loss: 1.3256032466888428

Epoch: 1        Training Loss: 1.309060        Validation Loss: 0.916991
Validation loss decreased (inf --> 0.916991). Saving model...
Epoch:2/15        Batch:5
Train Loss: 0.8108029961585999

Epoch:2/15        Batch:10
Train Loss: 0.8211743235588074

Epoch:2/15        Batch:15
Train Loss: 0.8621042370796204

Epoch:2/15        Batch:20
Train Loss: 0.8507435321807861

Epoch:2/15        Batch:25
Train Loss: 0.844939112663269
```

```
Epoch:2/15         Batch:30
Train Loss: 0.8461205959320068

Epoch:2/15         Batch:35
Train Loss: 0.8281208872795105

Epoch:2/15         Batch:40
Train Loss: 0.8303547501564026

Epoch:2/15         Batch:45
Train Loss: 0.829035222530365

Epoch:2/15         Batch:50
Train Loss: 0.8249326348304749

Epoch: 2         Training Loss: 0.817975         Validation Loss: 0.843375
Validation loss decreased (0.916991 --> 0.843375). Saving model...
Epoch:3/15         Batch:5
Train Loss: 0.6093114614486694

Epoch:3/15         Batch:10
Train Loss: 0.6090707182884216

Epoch:3/15         Batch:15
Train Loss: 0.6005129218101501

Epoch:3/15         Batch:20
Train Loss: 0.6081052422523499

Epoch:3/15         Batch:25
Train Loss: 0.5947156548500061

Epoch:3/15         Batch:30
Train Loss: 0.5981037020683289

Epoch:3/15         Batch:35
Train Loss: 0.6228029131889343

Epoch:3/15         Batch:40
Train Loss: 0.6243066787719727

Epoch:3/15         Batch:45
Train Loss: 0.6290121078491211

Epoch:3/15         Batch:50
Train Loss: 0.6321786642074585

Epoch: 3         Training Loss: 0.632280         Validation Loss: 0.764235
```

```
Validation loss decreased (0.843375 --> 0.764235). Saving model...
Epoch:4/15         Batch:5
Train Loss: 0.4999653995037079

Epoch:4/15         Batch:10
Train Loss: 0.47690126299858093

Epoch:4/15         Batch:15
Train Loss: 0.49732503294944763

Epoch:4/15         Batch:20
Train Loss: 0.5050666928291321

Epoch:4/15         Batch:25
Train Loss: 0.5006044507026672

Epoch:4/15         Batch:30
Train Loss: 0.49558529257774353

Epoch:4/15         Batch:35
Train Loss: 0.5012351870536804

Epoch:4/15         Batch:40
Train Loss: 0.4987010359764099

Epoch:4/15         Batch:45
Train Loss: 0.4921268820762634

Epoch:4/15         Batch:50
Train Loss: 0.49801674485206604

Epoch: 4         Training Loss: 0.508509         Validation Loss: 0.684560
Validation loss decreased (0.764235 --> 0.684560). Saving model...
Epoch:5/15         Batch:5
Train Loss: 0.341793030500412

Epoch:5/15         Batch:10
Train Loss: 0.41444146633148193

Epoch:5/15         Batch:15
Train Loss: 0.42749208211898804

Epoch:5/15         Batch:20
Train Loss: 0.4260594844818115

Epoch:5/15         Batch:25
Train Loss: 0.4291095435619354
```

```
Epoch:5/15        Batch:30
Train Loss: 0.4338321387767792

Epoch:5/15        Batch:35
Train Loss: 0.4401853680610657

Epoch:5/15        Batch:40
Train Loss: 0.42639443278312683

Epoch:5/15        Batch:45
Train Loss: 0.4288542568683624

Epoch:5/15        Batch:50
Train Loss: 0.42528268694877625

Epoch: 5        Training Loss: 0.423645         Validation Loss: 0.700608
Epoch:6/15        Batch:5
Train Loss: 0.37737783789634705

Epoch:6/15        Batch:10
Train Loss: 0.3513084053993225

Epoch:6/15        Batch:15
Train Loss: 0.3473784327507019

Epoch:6/15        Batch:20
Train Loss: 0.3432712256908417

Epoch:6/15        Batch:25
Train Loss: 0.34934282302856445

Epoch:6/15        Batch:30
Train Loss: 0.35548582673072815

Epoch:6/15        Batch:35
Train Loss: 0.35447555780410767

Epoch:6/15        Batch:40
Train Loss: 0.3520931303501129

Epoch:6/15        Batch:45
Train Loss: 0.3650243878364563

Epoch:6/15        Batch:50
Train Loss: 0.3723621666431427

Epoch: 6        Training Loss: 0.368664         Validation Loss: 0.771488
Epoch:7/15        Batch:5
```

```
Train Loss: 0.29471156001091003

Epoch:7/15         Batch:10
Train Loss: 0.2890653908252716

Epoch:7/15         Batch:15
Train Loss: 0.29841628670692444

Epoch:7/15         Batch:20
Train Loss: 0.29804515838623047

Epoch:7/15         Batch:25
Train Loss: 0.30704325437545776

Epoch:7/15         Batch:30
Train Loss: 0.31478095054626465

Epoch:7/15         Batch:35
Train Loss: 0.32330355048179626

Epoch:7/15         Batch:40
Train Loss: 0.3229244649410248

Epoch:7/15         Batch:45
Train Loss: 0.3309136927127838

Epoch:7/15         Batch:50
Train Loss: 0.3383764922618866

Epoch: 7         Training Loss: 0.341507         Validation Loss: 0.760447
Epoch:8/15         Batch:5
Train Loss: 0.2854400873184204

Epoch:8/15         Batch:10
Train Loss: 0.2627279758453369

Epoch:8/15         Batch:15
Train Loss: 0.2876936197280884

Epoch:8/15         Batch:20
Train Loss: 0.29456326365470886

Epoch:8/15         Batch:25
Train Loss: 0.3120196759700775

Epoch:8/15         Batch:30
Train Loss: 0.31597745418548584
```

```
Epoch:8/15         Batch:35
Train Loss: 0.3167196214199066

Epoch:8/15         Batch:40
Train Loss: 0.3316594064235687

Epoch:8/15         Batch:45
Train Loss: 0.33807218074798584

Epoch:8/15         Batch:50
Train Loss: 0.33550941944122314

Epoch: 8         Training Loss: 0.344680         Validation Loss: 0.771171
Epoch:9/15         Batch:5
Train Loss: 0.2997615337371826

Epoch:9/15         Batch:10
Train Loss: 0.3409773111343384

Epoch:9/15         Batch:15
Train Loss: 0.32357853651046753

Epoch:9/15         Batch:20
Train Loss: 0.329850971698761

Epoch:9/15         Batch:25
Train Loss: 0.3145599961280823

Epoch:9/15         Batch:30
Train Loss: 0.31554967164993286

Epoch:9/15         Batch:35
Train Loss: 0.31129583716392517

Epoch:9/15         Batch:40
Train Loss: 0.3091791272163391

Epoch:9/15         Batch:45
Train Loss: 0.30754128098487854

Epoch:9/15         Batch:50
Train Loss: 0.304308146238327

Epoch: 9         Training Loss: 0.309999         Validation Loss: 0.893910
Epoch:10/15         Batch:5
Train Loss: 0.29776814579963684

Epoch:10/15         Batch:10
```

```
Train Loss: 0.28959333896636963

Epoch:10/15         Batch:15
Train Loss: 0.3425484299659729

Epoch:10/15         Batch:20
Train Loss: 0.32780060172080994

Epoch:10/15         Batch:25
Train Loss: 0.32683685421943665

Epoch:10/15         Batch:30
Train Loss: 0.31664764881134033

Epoch:10/15         Batch:35
Train Loss: 0.32025086879730225

Epoch:10/15         Batch:40
Train Loss: 0.31423965096473694

Epoch:10/15         Batch:45
Train Loss: 0.3104628026485443

Epoch:10/15         Batch:50
Train Loss: 0.3172435462474823

Epoch: 10         Training Loss: 0.317747         Validation Loss: 0.914560
Epoch:11/15         Batch:5
Train Loss: 0.35662612318992615

Epoch:11/15         Batch:10
Train Loss: 0.3417188227176666

Epoch:11/15         Batch:15
Train Loss: 0.3307883143424988

Epoch:11/15         Batch:20
Train Loss: 0.33194321393966675

Epoch:11/15         Batch:25
Train Loss: 0.327163428068161

Epoch:11/15         Batch:30
Train Loss: 0.3203119933605194

Epoch:11/15         Batch:35
Train Loss: 0.3165564239025116
```

```
Epoch:11/15         Batch:40
Train Loss: 0.3197851777076721

Epoch:11/15         Batch:45
Train Loss: 0.3196573555469513

Epoch:11/15         Batch:50
Train Loss: 0.31562626361846924

Epoch: 11         Training Loss: 0.315537         Validation Loss: 0.924801
Epoch:12/15         Batch:5
Train Loss: 0.21247601509094238

Epoch:12/15         Batch:10
Train Loss: 0.24719597399234772

Epoch:12/15         Batch:15
Train Loss: 0.2608592212200165

Epoch:12/15         Batch:20
Train Loss: 0.2612208425998688

Epoch:12/15         Batch:25
Train Loss: 0.27872195839881897

Epoch:12/15         Batch:30
Train Loss: 0.27672216296195984

Epoch:12/15         Batch:35
Train Loss: 0.26648226380348206

Epoch:12/15         Batch:40
Train Loss: 0.26870331168174744

Epoch:12/15         Batch:45
Train Loss: 0.26490309834480286

Epoch:12/15         Batch:50
Train Loss: 0.2603883147239685

Epoch: 12         Training Loss: 0.255896         Validation Loss: 0.888187
Epoch:13/15         Batch:5
Train Loss: 0.27572953701019287

Epoch:13/15         Batch:10
Train Loss: 0.24823884665966034

Epoch:13/15         Batch:15
```

```
Train Loss: 0.24372050166130066

Epoch:13/15        Batch:20
Train Loss: 0.2296670526266098

Epoch:13/15        Batch:25
Train Loss: 0.22917471826076508

Epoch:13/15        Batch:30
Train Loss: 0.23111800849437714

Epoch:13/15        Batch:35
Train Loss: 0.22807221114635468

Epoch:13/15        Batch:40
Train Loss: 0.22437942028045654

Epoch:13/15        Batch:45
Train Loss: 0.2277379035949707

Epoch:13/15        Batch:50
Train Loss: 0.23620644211769104

Epoch: 13          Training Loss: 0.234066          Validation Loss: 0.942900
Epoch:14/15        Batch:5
Train Loss: 0.23195558786392212

Epoch:14/15        Batch:10
Train Loss: 0.24961140751838684

Epoch:14/15        Batch:15
Train Loss: 0.2596309781074524

Epoch:14/15        Batch:20
Train Loss: 0.25601211190223694

Epoch:14/15        Batch:25
Train Loss: 0.2617609202861786

Epoch:14/15        Batch:30
Train Loss: 0.2574203610420227

Epoch:14/15        Batch:35
Train Loss: 0.24787579476833344

Epoch:14/15        Batch:40
Train Loss: 0.2545822858810425
```

```
Epoch:14/15         Batch:45
Train Loss: 0.25325971841812134

Epoch:14/15         Batch:50
Train Loss: 0.2562991678714752

Epoch: 14         Training Loss: 0.252466         Validation Loss: 0.951581
Epoch:15/15         Batch:5
Train Loss: 0.28966909646987915

Epoch:15/15         Batch:10
Train Loss: 0.32727906107902527

Epoch:15/15         Batch:15
Train Loss: 0.29566165804862976

Epoch:15/15         Batch:20
Train Loss: 0.2738187611103058

Epoch:15/15         Batch:25
Train Loss: 0.2636953890323639

Epoch:15/15         Batch:30
Train Loss: 0.26391249895095825

Epoch:15/15         Batch:35
Train Loss: 0.2583701014518738

Epoch:15/15         Batch:40
Train Loss: 0.2423083484172821

Epoch:15/15         Batch:45
Train Loss: 0.24164380133152008

Epoch:15/15         Batch:50
Train Loss: 0.24058763682842255

Epoch: 15         Training Loss: 0.246865         Validation Loss: 0.898243
```

```
In [18]: # load the model that got the best validation accuracy
         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 [10]: 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
                 output = F.softmax(output, dim=1)
                 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)

         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.785375


Test Accuracy: 77% (651/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

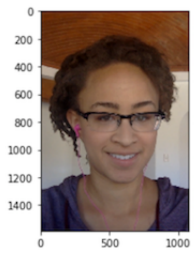
Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [19]: ### 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]
```

```python
class_names = [item[4:].replace("_", " ") for item in train_dataset.classes]

model_transfer.load_state_dict(torch.load('model_transfer.pt'))

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

    # Define normalization step for image
    normalize = transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                     std=(0.229, 0.224, 0.225))

    # Define transformations of image
    preprocess = transforms.Compose([transforms.Resize(224),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     normalize])

    # Preprocess image to 4D Tensor (.unsqueeze(0) adds a dimension)
    img_tensor = preprocess(img).unsqueeze_(0)

    # Move tensor to GPU if available
    if use_cuda:
        img_tensor = img_tensor.cuda()

    ## Inference
    # Turn on evaluation mode
    model_transfer.eval()

    # Get predicted category for image
    with torch.no_grad():
        output = model_transfer(img_tensor)
        prediction = torch.argmax(output).item()

    # Turn off evaluation mode
    model_transfer.train()

    # Use prediction to get dog breed
    breed = class_names[prediction]

    return 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.

```
hello, human!
```



```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

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

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             if face_detector(img_path):
                 print('Showing the tested image')
                 plt.imshow(Image.open(img_path))
                 plt.show()
                 print('Human detected')
                 print(f'The human in the image resembles a - {predict_breed_transfer(img_path)}
             elif dog_detector(img_path):
                 print('Showing the tested image')
                 plt.imshow(Image.open(img_path))
                 plt.show()
                 print('Dog detected')
                 print(f'This dog breed is called -  {predict_breed_transfer(img_path)}\n\n\n')
             else:
                 plt.imshow(Image.open(img_path))
                 plt.show()
                 print('Could not find anything (human or dog) in this image\n\n\n')
```

## Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer:

I expected no such results. I just aimed for a good accuracy, wherein 77% seems quite good. So yes I am happy with what I got. But definetely I will try to improve upon this accuracy.

1. I could have tried more combinations for hyperparameters, but due to shortage of time, I chose to go on whatever result I was getting
2. Choosing more epochs. It may or ma not have helped me increase the accuracy of my project, but I chose to continue with 25 epochs because I felt it was a pretty decent number to train my model on
3. I could have tried different pretrained models like the resnet152 model or any of the availabe models to check if they give me a better accuracy. It may have helped me out ar may not have.

```
In [30]: pwd
```

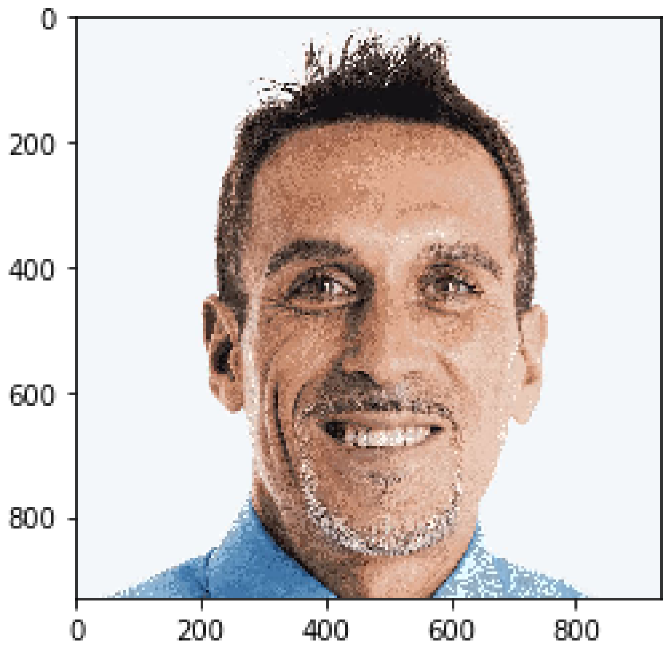
```
Out[30]: '/home/workspace/dog_project'
```

```
In [44]: ## 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_human_files = ['./human1.jpg', './human2.jpg', './human3.jpg' ]
         my_dog_files = ['./boxer.jpg', './chihuahua.jpg', './german shepard.jpg']

         ## suggested code, below
         for file in np.hstack((my_human_files, my_dog_files)):
             run_app(file)
```

```
Showing the tested image
```

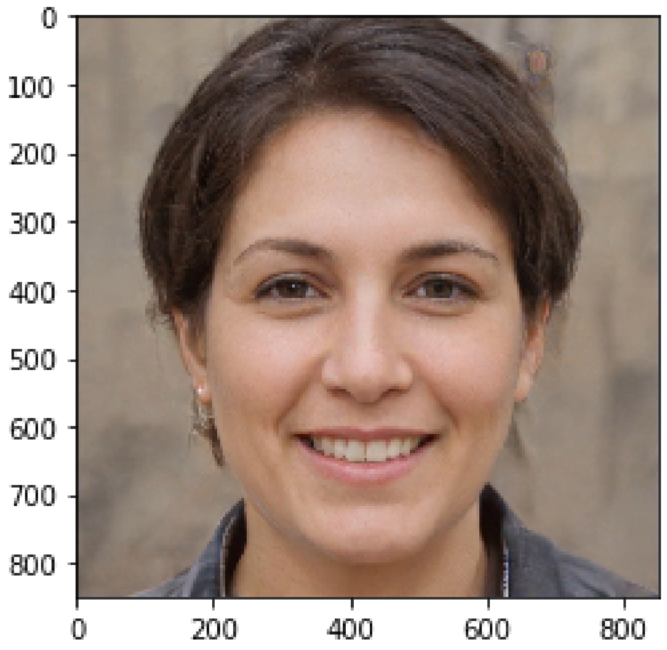
```
Human detected
The human in the image resembles a - Chinese crested
```

```
Showing the tested image
```



```
Human detected
The human in the image resembles a - Dogue de bordeaux
```

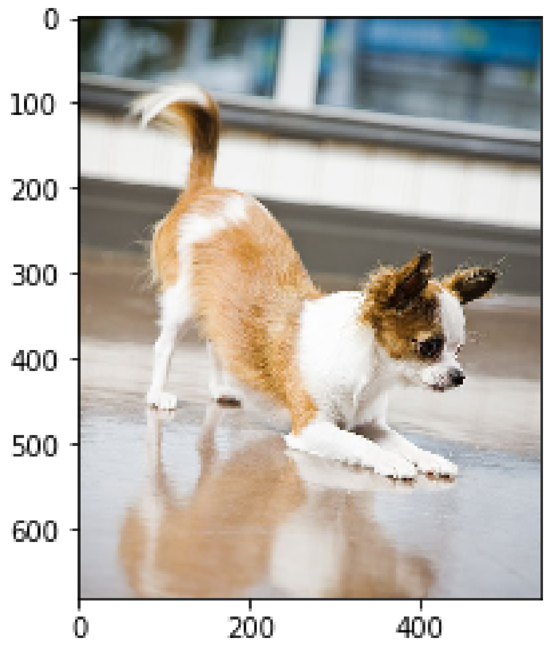
```
Showing the tested image
```



```
Human detected
The human in the image resembles a - Nova scotia duck tolling retriever
```

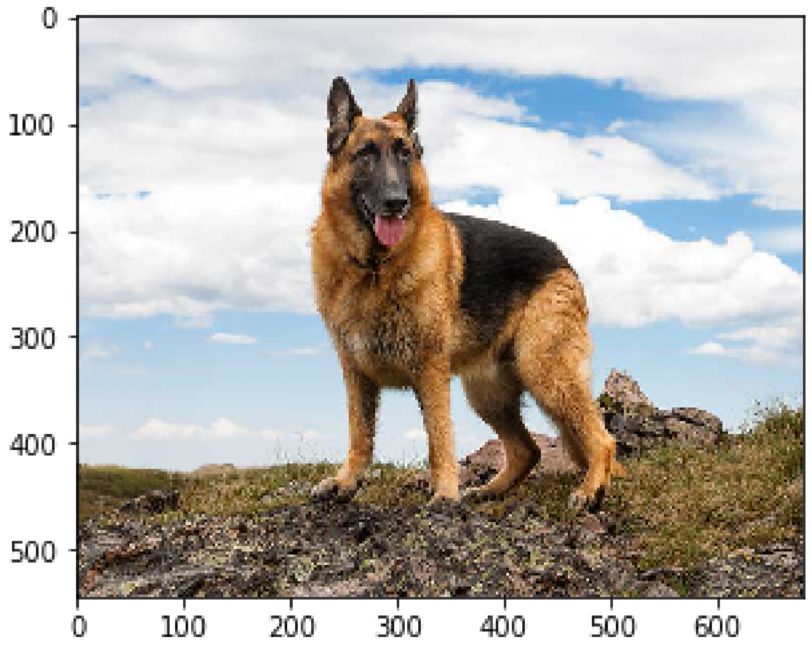
```
Showing the tested image
```

```
Dog detected
This dog breed is called -  Boxer



Showing the tested image
```



```
Dog detected
This dog breed is called -  Papillon
```

```
Showing the tested image
```



```
Dog detected
This dog breed is called -  German shepherd dog
```

```
In [ ]:
```