# dog_app

April 27, 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(human_files[0])
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

```
/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
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)
```

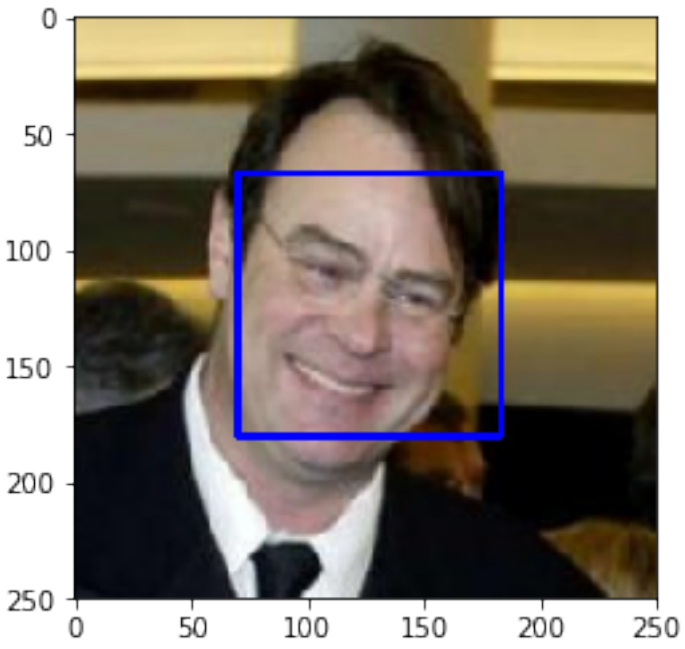
```
        # print number of faces detected in the image
        print('Number of faces detected:', len(faces))

        # 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:**

The percentage of human faces are detected by the model is 98.00%

The percentage of dog faces are detected by the model is 17.00%

```
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. #-#-#

        predicted_human_face = [1 if (face_detector(file_name)) else 0 for file_name in human_fi
        predicted_dog_face = [1 if (face_detector(file_name)) else 0 for file_name in dog_files_
        print("Percentage of human faces detected {:.2f}%".format((sum(predicted_human_face) / l
        print("Percentage of dog faces detected {:.2f}%".format((sum(predicted_dog_face)/ len(pr
```

```
Percentage of human faces detected 98.00%
Percentage of dog faces detected 17.00%
```

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 [00:23<00:00, 23331657.50it/s]
```

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

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

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

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

```
In [7]: from PIL import Image
        import torchvision.transforms as transforms
        VGG16.eval()
        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
    '''
    transforms_vgg = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
    ])
    img = Image.open(img_path)
    preprocessed_img = transforms_vgg(img)
    if(use_cuda):
        preprocessed_img = preprocessed_img.cuda()
    input_batch = preprocessed_img.unsqueeze(0)
    output=VGG16(input_batch)
    index_predicted = torch.max(torch.nn.functional.softmax(output[0], dim=0), 0)
    return index_predicted[1] # predicted class index
```

In [8]:
```
# input = torch.rand(1,3, 2, 2)
# print(input)
# print(input.unsqueeze(2))
output = VGG16_predict(human_files_short[0])
print(output)
```

```
tensor(906, device='cuda:0')
```

### 1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

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

### 1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

**Answer:**
The percentage of human faces are detected by the model is 0.00%
The percentage of dog faces are detected by the model is 100.00%

```
In [10]: predicted_human_face = [1 if (dog_detector(file_name)) else 0 for file_name in human_fi
         predicted_dog_face = [1 if (dog_detector(file_name)) else 0 for file_name in dog_files_
         print("Percentage of human faces detected {:.2f}%".format((sum(predicted_human_face) /
         print("Percentage of dog faces detected {:.2f}%".format((sum(predicted_dog_face)/ len(p
```

```
Percentage of human faces detected 0.00%
Percentage of dog faces detected 100.00%
```

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

---

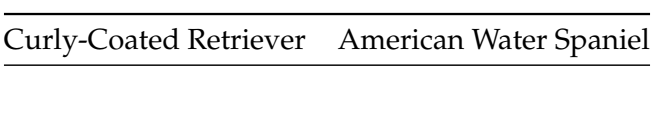
## Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).



Curly-Coated Retriever American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

| Yellow Labrador | Chocolate Labrador |
|---|---|

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

#### 1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

```
In [28]: import os
         from torchvision import datasets
         from torchvision.transforms import transforms
         from torch.utils.data import Dataset
         import os
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         transform_scratch = {
         'train' : transforms.Compose([
             transforms.Resize(224),
             transforms.CenterCrop(224),
             transforms.RandomHorizontalFlip(0.5),
             transforms.ToTensor(),
             transforms.Normalize(mean=[0.5, 0.5, 0.5],std=[0.5, 0.5, 0.5])
         ]),
         'test' : transforms.Compose([
             transforms.Resize(224),
             transforms.CenterCrop(224),
         #     transforms.RandomHorizontalFlip(0.5),
             transforms.ToTensor(),
             transforms.Normalize(mean=[0.5, 0.5, 0.5],std=[0.5, 0.5, 0.5])
         ])
         }
         batch_size = 20
         num_workers = 0
         # print(os.listdir('/data/dog_images/train/'))
         train_datasets = datasets.ImageFolder(root ='/data/dog_images/train/', transform = tran
```

```
test_datasets = datasets.ImageFolder(root ='/data/dog_images/test/', transform = transf
valid_datasets = datasets.ImageFolder(root ='/data/dog_images/valid/', transform = tran

trainDataLoader = torch.utils.data.DataLoader(train_datasets, shuffle=True, batch_size=
testDataLoader = torch.utils.data.DataLoader(test_datasets, shuffle=False, batch_size=b
validDataLoader = torch.utils.data.DataLoader(valid_datasets, shuffle=False, batch_size
loaders_scratch={'train' : trainDataLoader, 'valid' : validDataLoader , 'test' : testDa
```

```
In [22]: from glob import glob
         classes = set(glob('/data/dog_images/train/*'))
         len(classes)
```

```
Out[22]: 133
```

```
In [23]: img = cv2.imread(dog_files[1006])
         img.shape
```

```
Out[23]: (500, 375, 3)
```

**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**:

***1.*** I selected transforms.Resize() and transforms.CenterCrop() methods for cropping the images. For my model, I used the 224 * 224 image size because in our training data sets the dog images are different in sizes The reason for selecting the 224*224 image size because the vgg19 model uses it. In my next transfer learning model, I used vgg19 and the test accuracy is 70%

**2** I used only transforms.RandomHorizontalFlip()'s augmentation method, which flips the image horizontally. I could have used transforms.RandomVerticalFlip() method as well. But right now my only goal to achieve 10%+ accuracy.

### 1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [24]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         #224(3,32)->112(3,64)->56(3,128)->28(3,256)->14(3)->7(3)
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 self.conv_0 = nn.Conv2d(3, 32, 3, padding= 1)#224->112
                 self.batch_0 = nn.BatchNorm2d(32)
                 self.conv_1 = nn.Conv2d(32, 64, 3, padding= 1)#112->56
                 self.batch_1 = nn.BatchNorm2d(64)
```

```
            self.conv_2 = nn.Conv2d(64, 128, 3, padding= 1)#56->28
            self.batch_2 = nn.BatchNorm2d(128)
#           self.conv_3 = nn.Conv2d(128, 256, 3, padding= 1)
#           self.conv_4 = nn.Conv2d(256, 256, 3, padding= 1)
#           self.linear1 = nn.Linear(7*7*256, 500)
            self.linear1 = nn.Linear(28*28*128, 1000)
            self.linear2 = nn.Linear(1000, 500)
            self.linear3 = nn.Linear(500, 133)
            self.pool = nn.MaxPool2d(2, 2)
            self.dropout= nn.Dropout(0.5)
        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.conv_0(x)))
            x = self.batch_0(x)
            x = self.pool(F.relu(self.conv_1(x)))
            x = self.batch_1(x)
            x = self.pool(F.relu(self.conv_2(x)))
            x = self.batch_2(x)
#           x = self.pool(F.relu(self.conv_3(x)))
#           x = self.pool(F.relu(self.conv_4(x)))
#           x = x.view(-1, 256*7*7)
            x = x.view(-1, 128*28*28)
            x = self.linear1(x)
            x = self.dropout(x)
            x = self.linear2(x)
            x = self.dropout(x)
            x = self.linear3(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()
```

In [25]:
```
# img = cv2.imread(human_files[0])
# img.shape
import torch.nn as nn
input = torch.rand(1,3, 224, 224)
# conv = nn.Conv2d(3, 32, 3, padding = 1)
if(use_cuda):
    input = input.cuda()
output = model_scratch(input)
output.shape
```

Out[25]:
```
torch.Size([1, 133])
```

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

**Answer:**

I have used three Convolutions layers with three Batch normalization layers for learning the features, and three Linear layers used for the classification.

I have used three Convolutions layers with three Batch normalization layers for learning the features, and three Linear layers used for the classification.

1. First, the Convolution layer takes the input of 224 * 224 images with 3 channels and convert to 32 channels.
2. Next layer after the Convolution layer, I used a Batch normalization layer to avoid any shift variance.
3. Also, I used Relu for activation function in each layer to avoid the gradient vanishing problem.
4. After applying activation to the Convolution layer I used the Maxpool layer(2,2).

The current model architecture used 3 times of all the steps mentioned.
Classification:

1. I used three full connected Linear layers for classification with dropout probabilities 0.5 to avoid the overfitting.

I used only 30 epochs in training times because the training the model was taking the time. However, during the training time, the validation loss kept going down. I am confident with the increase of epoch size validation size will be less.

### 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 [26]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = torch.optim.SGD(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 [29]: 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()
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - trai
        ## find the loss and update the model parameters accordingly
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['valid']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## update the average validation loss
        output = model(data)
        loss = criterion(output, target)
        valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - vali
    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch,
        train_loss,
        valid_loss
        ))

    ## TODO: save the model if validation loss has decreased
    if valid_loss < valid_loss_min:
        print("Saving model valid_loss_min {:.6f} and new valid_loss {:.6f}".format
        torch.save(model.state_dict(), save_path)
        valid_loss_min = valid_loss
```

```
            # return trained model
        #     model.load_state_dict(torch.load('model_cifar.pt'))
            return model


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

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

```
Epoch: 1         Training Loss: 4.809955         Validation Loss: 4.681983
Saving model valid_loss_min inf and new valid_loss 4.681983
Epoch: 2         Training Loss: 4.547645         Validation Loss: 4.494184
Saving model valid_loss_min 4.681983 and new valid_loss 4.494184
Epoch: 3         Training Loss: 4.352602         Validation Loss: 4.363928
Saving model valid_loss_min 4.494184 and new valid_loss 4.363928
Epoch: 4         Training Loss: 4.169559         Validation Loss: 4.270957
Saving model valid_loss_min 4.363928 and new valid_loss 4.270957
Epoch: 5         Training Loss: 4.043731         Validation Loss: 4.193615
Saving model valid_loss_min 4.270957 and new valid_loss 4.193615
Epoch: 6         Training Loss: 3.910028         Validation Loss: 4.136693
Saving model valid_loss_min 4.193615 and new valid_loss 4.136693
Epoch: 7         Training Loss: 3.772280         Validation Loss: 4.090240
Saving model valid_loss_min 4.136693 and new valid_loss 4.090240
Epoch: 8         Training Loss: 3.659722         Validation Loss: 4.045811
Saving model valid_loss_min 4.090240 and new valid_loss 4.045811
Epoch: 9         Training Loss: 3.550312         Validation Loss: 4.010273
Saving model valid_loss_min 4.045811 and new valid_loss 4.010273
Epoch: 10         Training Loss: 3.423713         Validation Loss: 3.976034
Saving model valid_loss_min 4.010273 and new valid_loss 3.976034
Epoch: 11         Training Loss: 3.305895         Validation Loss: 3.937198
Saving model valid_loss_min 3.976034 and new valid_loss 3.937198
Epoch: 12         Training Loss: 3.183761         Validation Loss: 3.916758
Saving model valid_loss_min 3.937198 and new valid_loss 3.916758
Epoch: 13         Training Loss: 3.068322         Validation Loss: 3.889495
Saving model valid_loss_min 3.916758 and new valid_loss 3.889495
Epoch: 14         Training Loss: 2.940068         Validation Loss: 3.863577
Saving model valid_loss_min 3.889495 and new valid_loss 3.863577
Epoch: 15         Training Loss: 2.821360         Validation Loss: 3.847912
Saving model valid_loss_min 3.863577 and new valid_loss 3.847912
Epoch: 16         Training Loss: 2.703960         Validation Loss: 3.824378
Saving model valid_loss_min 3.847912 and new valid_loss 3.824378
Epoch: 17         Training Loss: 2.564787         Validation Loss: 3.821624
Saving model valid_loss_min 3.824378 and new valid_loss 3.821624
Epoch: 18         Training Loss: 2.440749         Validation Loss: 3.805866
Saving model valid_loss_min 3.821624 and new valid_loss 3.805866
```

```
Epoch: 19         Training Loss: 2.320993         Validation Loss: 3.797641
Saving model valid_loss_min 3.805866 and new valid_loss 3.797641
Epoch: 20         Training Loss: 2.175330         Validation Loss: 3.786885
Saving model valid_loss_min 3.797641 and new valid_loss 3.786885
Epoch: 21         Training Loss: 2.058232         Validation Loss: 3.752358
Saving model valid_loss_min 3.786885 and new valid_loss 3.752358
Epoch: 22         Training Loss: 1.931340         Validation Loss: 3.748323
Saving model valid_loss_min 3.752358 and new valid_loss 3.748323
Epoch: 23         Training Loss: 1.786296         Validation Loss: 3.739765
Saving model valid_loss_min 3.748323 and new valid_loss 3.739765
Epoch: 24         Training Loss: 1.679717         Validation Loss: 3.757197
Epoch: 25         Training Loss: 1.554001         Validation Loss: 3.749390
Epoch: 26         Training Loss: 1.445666         Validation Loss: 3.742570
Epoch: 27         Training Loss: 1.312996         Validation Loss: 3.729371
Saving model valid_loss_min 3.739765 and new valid_loss 3.729371
Epoch: 28         Training Loss: 1.207430         Validation Loss: 3.757889
Epoch: 29         Training Loss: 1.095725         Validation Loss: 3.730099
Epoch: 30         Training Loss: 1.000091         Validation Loss: 3.727132
Saving model valid_loss_min 3.729371 and new valid_loss 3.727132
```

```
In [30]: model_scratch
```

```
Out[30]: Net(
           (conv_0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (batch_0): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           (conv_1): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (batch_1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           (conv_2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (batch_2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
           (linear1): Linear(in_features=100352, out_features=1000, bias=True)
           (linear2): Linear(in_features=1000, out_features=500, bias=True)
           (linear3): Linear(in_features=500, out_features=133, bias=True)
           (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
           (dropout): Dropout(p=0.5)
         )
```

#### 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 [33]: def test(loaders, model, criterion, use_cuda):

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

```
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['test']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the loss
                loss = criterion(output, target)
                # update average test loss
                test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
                # convert output probabilities to predicted class
                pred = output.data.max(1, keepdim=True)[1]
                # compare predictions to true label
                correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
                total += data.size(0)

            print('Test Loss: {:.6f}\n'.format(test_loss))

            print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
                100. * correct / total, correct, total))

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

```
Test Loss: 3.649839


Test Accuracy: 16% (139/836)
```

---

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

```
from torch.utils.data import Dataset
import os
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
transform_transfer = {
'train' : transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop(224),
    transforms.RandomHorizontalFlip(0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5],std=[0.5, 0.5, 0.5])
]),
'test' : transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop(224),
#     transforms.RandomHorizontalFlip(0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5],std=[0.5, 0.5, 0.5])
])
}
batch_size = 20
num_workers = 0
# print(os.listdir('/data/dog_images/train/'))
train_datasets_transfer = datasets.ImageFolder(root ='/data/dog_images/train/', transfo
test_datasets_transfer = datasets.ImageFolder(root ='/data/dog_images/test/', transform
valid_datasets_transfer = datasets.ImageFolder(root ='/data/dog_images/valid/', transfo

trainDataLoader_transfer = torch.utils.data.DataLoader(train_datasets_transfer, shuffle
testDataLoader_transfer = torch.utils.data.DataLoader(test_datasets_transfer, shuffle=F
validDataLoader_transfer = torch.utils.data.DataLoader(valid_datasets_transfer, shuffle
loaders_transfer={'train' : trainDataLoader_transfer, 'valid' : validDataLoader_transfe
```

#### 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 [35]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture

         model_transfer = models.vgg19(pretrained=True)
         for param in model_transfer.features.parameters():
           param.requires_grad = False
```

```
        model_transfer.classifier[6] = nn.Linear(4096, 133)
        if use_cuda:
            model_transfer = model_transfer.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.torch/models/vgg
100%|| 574673361/574673361 [00:08<00:00, 68552880.16it/s]
```

**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:**

For this current model, I used the vgg19 transfer learning model. To predict the dog breeds by this model I changed the last layer of the vgg19 model from 1000 classes to 133.

The model layers divided into two parts the features and classifier. The layer features are not needed to be trained so set the requires_grad= False, and for the layer classifier is trainable. However, for the final layer "Linear" of classifier layer replaced by the nn.Linear(4096, 133)

I feel the transfer learning model is suitable for this classification as it achieves 70% without training the conv2d layers. As our utilizing the already trained weights and need not run again.

### 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 [36]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = torch.optim.SGD(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 [37]: # train the model
         import numpy as np
         n_epochs = 10
         model_transfer = train(n_epochs, loaders_transfer, 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'))
```

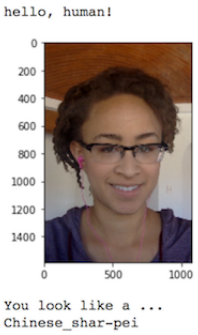
### 1.1.16 (IMPLEMENTATION) Test the Model

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

```
Test Loss: 0.559212


Test Accuracy: 82% (688/836)
```




Sample Human Output

### 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 [39]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         # from PIL import Image
         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in train_datasets_transfer.classes]
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         model_transfer.eval()
         def predict_breed_transfer(img_path):
             img = Image.open(img_path)
             transformed_img = transform_transfer['test'](img)
             if use_cuda:
                 transformed_img = transformed_img.cuda()
             transformed_img = transformed_img.unsqueeze(0)
             output = model_transfer(transformed_img)
             pred = output.data.max(1, keepdim=True)[1]
             return class_names[pred]
```

---

## Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

#### 1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [40]: def imshow(file):
             img = cv2.imread(file)
             img = cv2.cvtColor(img, cv2.COLOR_RGB2BGRA)
             plt.imshow(img)

In [42]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         import matplotlib.pyplot as plt
         %matplotlib inline

         def run_app(img_path):
             if(dog_detector(img_path)):
                 return "dog", predict_breed_transfer(img_path)

             elif(face_detector(img_path)):
                 return "human", predict_breed_transfer(img_path)
             ## handle cases for a human face, dog, and neither
             else:
                 return None, None
```

---

## Step 6: Test Your Algorithm

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

#### 1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

**Answer:** (Three possible points for improvement)

1. Include more data for training by an augmentation technique(like Vertical image flipping)
2. Hidden Layer: Make the model more deeper to extract features by creating more hidden layers as of now only 4 layers of Convolution and 3 layers of Linear. Also, I would try to find some other architecture of a neural network from web, which works for this kind of problem and implements here.
3. Finally, I would like to do some more experements like:

a. Try different weight intilization methods (like weight initlization based on a number of neurons in the network, etc.)

b. Also, want to try different learning rates with different momentum values in the Adam optimizer.

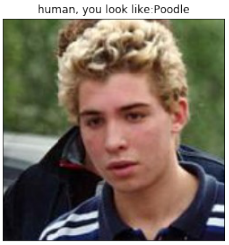
c. Different activation function like ElU, etc.

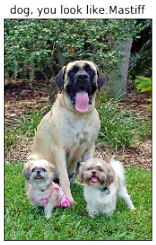
In [45]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
## suggested code, below

dog_files = np.array(glob("/data/dog_images/test/*/*"))

fig = plt.figure(figsize = (25, 10))
for index, file in enumerate(np.hstack((human_files[:3], dog_files[:3]))):
    ax = fig.add_subplot(2, 3, index+1, xticks=[], yticks=[])
    type_detector, output = run_app(file)
    if(type_detector is not None):
        imshow(file)
        ax.set_title(type_detector + ", you look like:" + output)
    else:
        print("Error in detecting type of an image")
```








In [ ]: