dog_app

April 7, 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

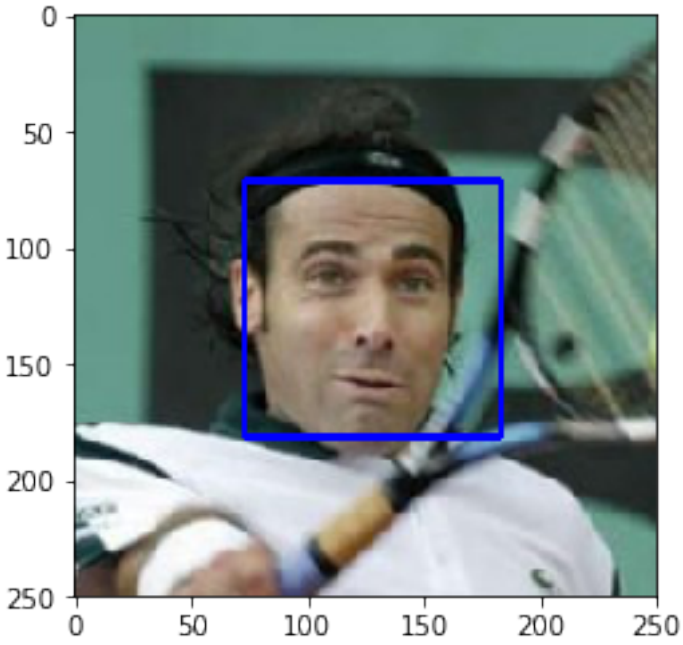
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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[1])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
```

```
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: (You can print out your results and/or write your percentages in this cell) Please see the implemented code below. The python built-in function is used to apply function (face_detector) on all images.

```
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.

        # for faces in human_files_short:
        #     human_face_detected = face_detector(faces)
        match1= sum(map(face_detector, human_files_short))
        match2= sum(map(face_detector, dog_files_short))
        print('Percentage of humans in 100 human images is: {:.0%}'.format(match1/ 100))
        print('Percentage of humans in 100 dog images is: {:.0%}'.format(match2/ 100))
```

```
Percentage of humans in 100 human images is: 98%
Percentage of humans in 100 dog images is: 17%
```

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()


        print(VGG16.classifier[6].in_features)
        print(VGG16.classifier[6].out_features)
```

```
4096
1000
```

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
        from torch.autograd import Variable
```

```
#img_path= Image.open("data/dog_project/dogImages/train/001.Affenpinscher/Affenpinscher_

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
    image= Image.open(img_path)

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

    transformed_image= transform(image).unsqueeze(0)
    if use_cuda:
      transformed_image = Variable(transformed_image.cuda())
    else:
      transformed_image = Variable(transformed_image)

    predicted_value= torch.argmax(VGG16(transformed_image)).item()

    return predicted_value # predicted class index

print(VGG16_predict(dog_files[0]))
```

```
243
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            if (VGG16_predict(img_path)) in range(151,269):
                return True
            else:
                return False # 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: Please see the code implemented below. The python built-in function 'map' is used to apply the function on images.

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        match1= sum(map(dog_detector, human_files_short))
        match2= sum(map(dog_detector, dog_files_short))
        print('Percentage of dogs in humans images is: {:.0%}'.format(match1/ 100))
        print('Percentage of dogs in dogs images is: {:.0%}'.format(match2/ 100))
```

```
Percentage of dogs in humans images is: 2%
Percentage of dogs in dogs images is: 100%
```

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.
```

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
         from torchvision import datasets

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

         data_dir = 'dogImages'
         train_dir = data_dir + '/train'
         valid_dir = data_dir + '/valid'
         test_dir = data_dir + '/test'

         batch_size = 16
         image_transforms = {'train': transforms.Compose([transforms.RandomRotation(15),
                                                          transforms.RandomHorizontalFlip(),
                                                          transforms.Resize(256),
                                                          transforms.CenterCrop(224),
```

```
                                       transforms.ToTensor(),
                                       transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                            std=[0.229, 0.224, 0.225])]),
                        'validtest': 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])])}
loaders_scratch = {'train': torch.utils.data.DataLoader(datasets.ImageFolder(train_dir,
                   'valid': torch.utils.data.DataLoader(datasets.ImageFolder(valid_dir, t
                   'test': torch.utils.data.DataLoader(datasets.ImageFolder(test_dir, tra
```

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

Answer: I used the torchvision transforms like Resize and CenterCrop to feed to the network. I choose the tensor size of 224 as almost all the networks except the Inception Network need input images of size 224*224.

Yes, I used the augmentation technique to increse the dataset size so that model can generalize the images well. The horizonal flip and rotation of images with a certain degree is used to accomplish that.

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

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 32, 3)
                 self.conv2 = nn.Conv2d(32, 64, 3)
                 self.conv3 = nn.Conv2d(64, 128, 3)
                 self.conv4 = nn.Conv2d(128, 256, 3)
                 self.pool = nn.MaxPool2d(2, 2)
                 self.fc1 = nn.Linear(256 * 12 * 12, 2304)
                 self.fc2 = nn.Linear(2304, 1021)
                 self.fc3 = nn.Linear(1021, 133)

             def forward(self, x):
                 ## Define forward behavior
                 x = self.pool(F.relu(self.conv1(x)))
```

```
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = x.view(-1, 256 * 12 * 12)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(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: My CNN architecture includes four convolution layers and three fully connected layers. As we are feeding the RGB images to the network the input channels for the first convolution layer is set to 3. The output channels is taken as some random small number and it indicates the number of filters used during convolution. I have choosen the filter kernel size to be 3 for all convolutional layers.

To control the size of convolved output the maxpooling is applied after applying the relu activation function.

The number of input features fed to the first fully connected layer are calulated from the result of maxpooling the fourth convolutional layer output.

I received more than 10% test accuracy with this simple network but the batch normalization and dropout are some regularisation techniques we can use to further increase the accuracy.

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.SGD(model_scratch.parameters(), lr=0.001, momentum=0.9)
```

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 [14]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [15]:
```
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_scratch['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            # zero the parameter gradients
            optimizer.zero_grad()
            ## find the loss and update the model parameters accordingly
            with torch.set_grad_enabled(True):
                outputs = model(data)
                loss = criterion(outputs, target)
                loss.backward()
                optimizer.step()
            ## record the average training loss, using something like

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders_scratch['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            with torch.set_grad_enabled(False):
                outputs = model(data)  # batch_size x 133
                loss = criterion(outputs, target)  # Average loss value over batch.

            valid_loss = 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 ...'.form
                    torch.save(model.state_dict(), save_path)
                    valid_loss_min = valid_loss
            return model
```

In [16]:
```
# train the model
model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.888273         Validation Loss: 4.884447
Validation loss decreased (inf --> 4.884447).Saving model ...
Epoch: 2         Training Loss: 4.877956         Validation Loss: 4.866100
Validation loss decreased (4.884447 --> 4.866100).Saving model ...
Epoch: 3         Training Loss: 4.857882         Validation Loss: 4.831257
Validation loss decreased (4.866100 --> 4.831257).Saving model ...
Epoch: 4         Training Loss: 4.797168         Validation Loss: 4.714963
Validation loss decreased (4.831257 --> 4.714963).Saving model ...
Epoch: 5         Training Loss: 4.743843         Validation Loss: 4.658160
Validation loss decreased (4.714963 --> 4.658160).Saving model ...
Epoch: 6         Training Loss: 4.715777         Validation Loss: 4.629999
Validation loss decreased (4.658160 --> 4.629999).Saving model ...
Epoch: 7         Training Loss: 4.690953         Validation Loss: 4.593922
Validation loss decreased (4.629999 --> 4.593922).Saving model ...
Epoch: 8         Training Loss: 4.665226         Validation Loss: 4.571082
Validation loss decreased (4.593922 --> 4.571082).Saving model ...
Epoch: 9         Training Loss: 4.647754         Validation Loss: 4.546420
Validation loss decreased (4.571082 --> 4.546420).Saving model ...
Epoch: 10         Training Loss: 4.578623         Validation Loss: 4.412492
Validation loss decreased (4.546420 --> 4.412492).Saving model ...
Epoch: 11         Training Loss: 4.515930         Validation Loss: 4.379284
Validation loss decreased (4.412492 --> 4.379284).Saving model ...
Epoch: 12         Training Loss: 4.481694         Validation Loss: 4.380676
Epoch: 13         Training Loss: 4.444018         Validation Loss: 4.318460
Validation loss decreased (4.379284 --> 4.318460).Saving model ...
Epoch: 14         Training Loss: 4.422425         Validation Loss: 4.362939
Epoch: 15         Training Loss: 4.378257         Validation Loss: 4.211885
```

```
Validation loss decreased (4.318460 --> 4.211885).Saving model ...
Epoch: 16         Training Loss: 4.331741         Validation Loss: 4.226194
Epoch: 17         Training Loss: 4.307666         Validation Loss: 4.161197
Validation loss decreased (4.211885 --> 4.161197).Saving model ...
Epoch: 18         Training Loss: 4.286213         Validation Loss: 4.203600
Epoch: 19         Training Loss: 4.233843         Validation Loss: 4.124864
Validation loss decreased (4.161197 --> 4.124864).Saving model ...
Epoch: 20         Training Loss: 4.209561         Validation Loss: 4.070109
Validation loss decreased (4.124864 --> 4.070109).Saving model ...
Epoch: 21         Training Loss: 4.189383         Validation Loss: 4.068805
Validation loss decreased (4.070109 --> 4.068805).Saving model ...
Epoch: 22         Training Loss: 4.139936         Validation Loss: 4.033444
Validation loss decreased (4.068805 --> 4.033444).Saving model ...
Epoch: 23         Training Loss: 4.124140         Validation Loss: 4.006100
Validation loss decreased (4.033444 --> 4.006100).Saving model ...
Epoch: 24         Training Loss: 4.063908         Validation Loss: 4.033412
Epoch: 25         Training Loss: 4.030656         Validation Loss: 3.928234
Validation loss decreased (4.006100 --> 3.928234).Saving model ...
Epoch: 26         Training Loss: 3.998524         Validation Loss: 3.903857
Validation loss decreased (3.928234 --> 3.903857).Saving model ...
Epoch: 27         Training Loss: 3.959373         Validation Loss: 3.957678
Epoch: 28         Training Loss: 3.918110         Validation Loss: 3.807583
Validation loss decreased (3.903857 --> 3.807583).Saving model ...
Epoch: 29         Training Loss: 3.878995         Validation Loss: 3.872515
Epoch: 30         Training Loss: 3.832620         Validation Loss: 3.796522
Validation loss decreased (3.807583 --> 3.796522).Saving model ...
```

1.1.11 (IMPLEMENTATION) Test the Model

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

In [17]:
```
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: 3.804701


Test Accuracy: 12% (103/836)
```

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 [18]: ## TODO: Specify data loaders
         tbatch_size = 32
         image_transforms = {'train': transforms.Compose([transforms.RandomRotation(15),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.RandomResizedCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                     std=[0.229, 0.224, 0.225])]),
                             'validtest': transforms.Compose([transforms.Resize(224),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize(mean=[0.485, 0.456, 0.406],
```

```
                                                                    std=[0.229, 0.224, 0.225])])}
loaders_transfer = {'train': torch.utils.data.DataLoader(datasets.ImageFolder(train_dir
                    'valid': torch.utils.data.DataLoader(datasets.ImageFolder(valid_dir, t
                    'test': torch.utils.data.DataLoader(datasets.ImageFolder(test_dir, tra
```

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 [19]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=True)
         for param in model_transfer.parameters():
             param.requires_grad = False

         num_ftrs = model_transfer.fc.in_features
         model_transfer.fc = nn.Linear(num_ftrs, 133)

         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 choose the resnet50 pre-trained architecture as it performs really well on almost all types of classification tasks. The main objective behind using the ResNet architecture is that it avoids the vanishing gradient problem in larger networks because of the presence of residual connections.

I used the horizontal filp and rotation to augment the data to feed to the network. The batch size of 32 is choosen. I used the adam optimizer to optimize the network performance by minimizing the loss calulated using CrossEntropy.

As the network is pre-trained on ImageNet dataset so to achieve considerable accuracy I trained the network only for 10 epochs and succeeded in getting 85% accuracy on test set in my first attempt.

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 [20]: criterion_transfer =  nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(filter(lambda p: p.requires_grad, model_transfer.parame
         #optimizer.SGD(filter(lambda p: p.requires_grad, model.parameters()), lr=1e-3)
```

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 [21]: # train the model
         transfer_epochs= 10
         model_transfer = train(transfer_epochs, loaders_transfer, model_transfer, optimizer_tra

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

```
Epoch: 1         Training Loss: 4.169981         Validation Loss: 3.041902
Validation loss decreased (inf --> 3.041902).Saving model ...
Epoch: 2         Training Loss: 3.001199         Validation Loss: 2.007290
Validation loss decreased (3.041902 --> 2.007290).Saving model ...
Epoch: 3         Training Loss: 2.319520         Validation Loss: 1.454026
Validation loss decreased (2.007290 --> 1.454026).Saving model ...
Epoch: 4         Training Loss: 1.927833         Validation Loss: 1.117106
Validation loss decreased (1.454026 --> 1.117106).Saving model ...
Epoch: 5         Training Loss: 1.679690         Validation Loss: 0.933273
Validation loss decreased (1.117106 --> 0.933273).Saving model ...
Epoch: 6         Training Loss: 1.504469         Validation Loss: 0.811842
Validation loss decreased (0.933273 --> 0.811842).Saving model ...
Epoch: 7         Training Loss: 1.380139         Validation Loss: 0.733539
Validation loss decreased (0.811842 --> 0.733539).Saving model ...
Epoch: 8         Training Loss: 1.288976         Validation Loss: 0.660213
Validation loss decreased (0.733539 --> 0.660213).Saving model ...
Epoch: 9         Training Loss: 1.199841         Validation Loss: 0.619560
Validation loss decreased (0.660213 --> 0.619560).Saving model ...
Epoch: 10         Training Loss: 1.154247         Validation Loss: 0.582813
Validation loss decreased (0.619560 --> 0.582813).Saving model ...
```

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

```
Test Loss: 0.598777


Test Accuracy: 85% (712/836)
```

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 [24]: ### 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]
        data_transfer = {'train' : datasets.ImageFolder(train_dir)}
        class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].classes]
```

```
In [26]: class_names[:10]
```

```
Out[26]: ['Affenpinscher',
          'Afghan hound',
          'Airedale terrier',
          'Akita',
          'Alaskan malamute',
          'American eskimo dog',
          'American foxhound',
          'American staffordshire terrier',
          'American water spaniel',
          'Anatolian shepherd dog']
```

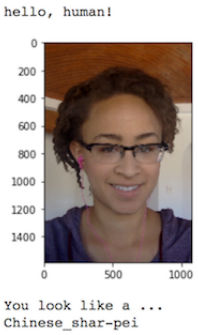
```
In [36]: def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             images_dogs= Image.open(img_path).convert('RGB')
             transfer_transforms = transforms.Compose([transforms.Resize(256),
                                                       transforms.CenterCrop(224),
                                                       transforms.ToTensor(),
                                                       transforms.Normalize(mean=[0.485, 0.456, 0
                                                                            std=[0.229, 0.224, 0.
             tranformed_dogs= transfer_transforms(images_dogs).unsqueeze(0)

             if use_cuda:
                 tranformed_dogs = tranformed_dogs.cuda()
             predicted_breed = model_transfer(tranformed_dogs)
             breed = torch.argmax(predicted_breed)
             return class_names[breed]
```

Step 5: Write your Algorithm

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

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

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



Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [45]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         import matplotlib.pyplot as plt
         def run_app(img_path):
             print(img_path)
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
             ## handle cases for a human face, dog, and neither
             if dog_detector(img_path):
                 print('The image consist dog')
                 breed = predict_breed_transfer(img_path)
                 print('The detected dog breed is:'+breed)
             elif face_detector(img_path):
                 print('The image cosists human')
                 breed = predict_breed_transfer(img_path)
                 print("The human in the image looks like:"+breed)
             else:
                 print('Neither dog nor human found')
             return
```

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.

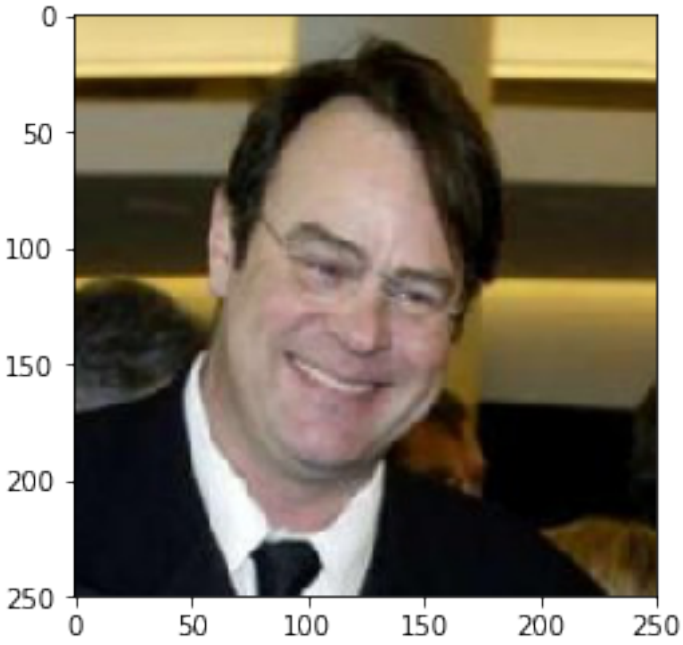
Answer: (Three possible points for improvement)

The output is same as I expected for the given dataset and network. There is a lot of scope of improvement in test results by using more augmentation techniques and fine-tuning the network by trial and error. The use of more deep network architectures like ResNet152 and DenseNet can also be tried.

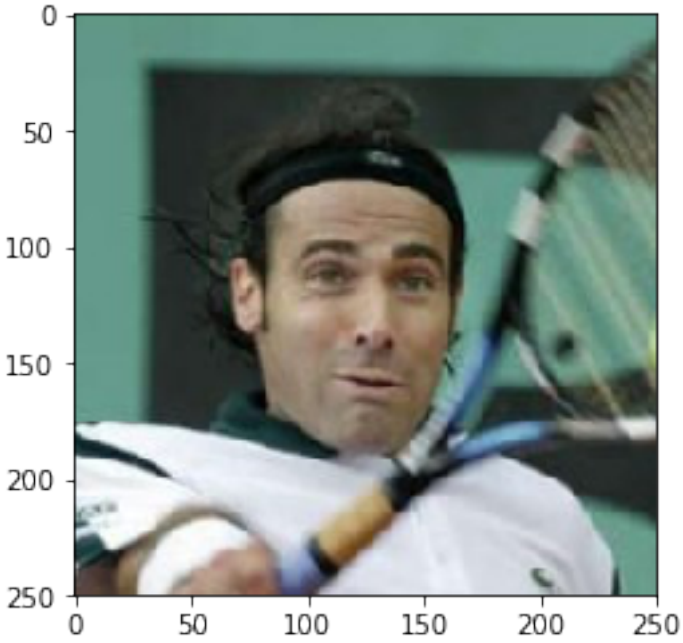
The three possible points of improvement for algorithm are: 1. Use of more data augmentation techniques. 2. Use of other networks (Resnet152, DenseNet etc.) 3. Fine-tuning the network

```
In [46]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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

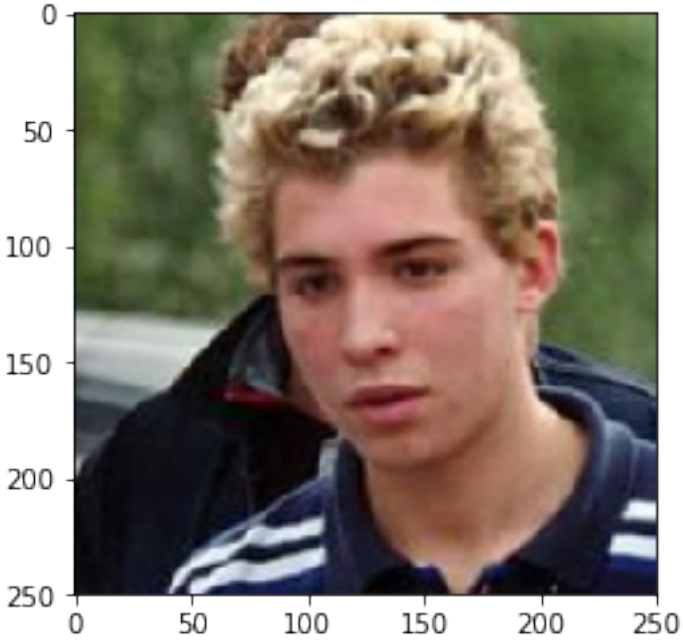
```
/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
```



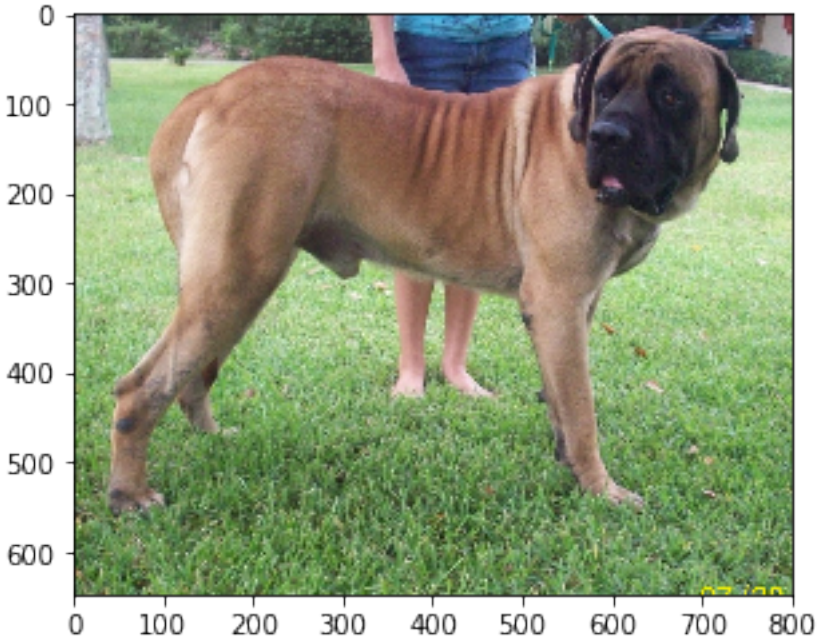

```
The image cosists human
The human in the image looks like:Chihuahua
/data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
```



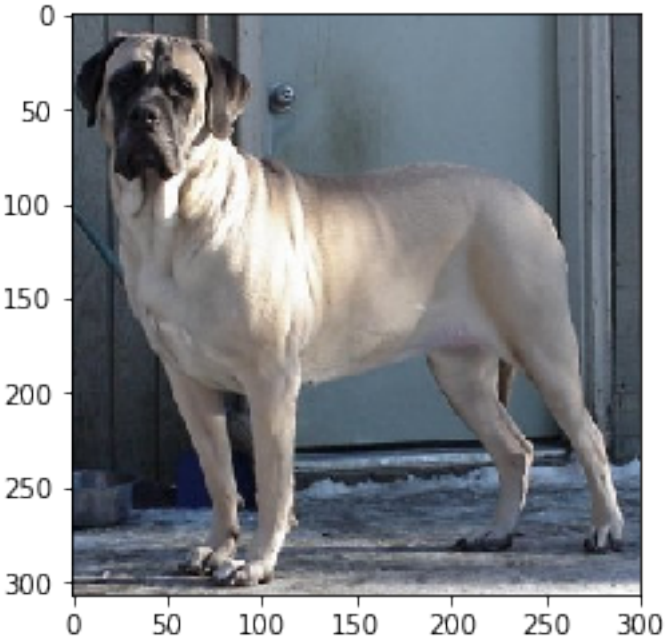

```
The image cosists human
The human in the image looks like:Doberman pinscher
/data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg
```



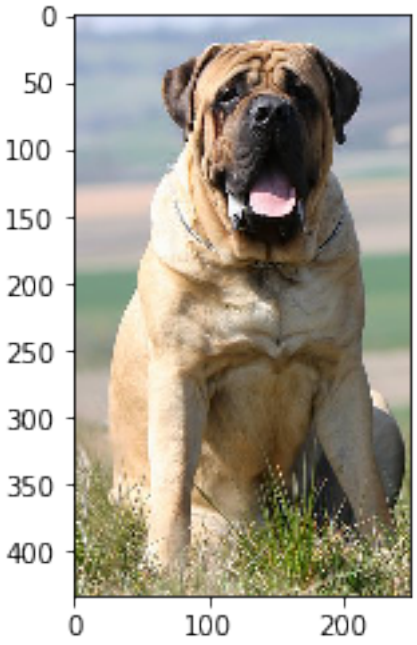
```
The image cosists human
The human in the image looks like:American water spaniel
/data/dog_images/train/103.Mastiff/Mastiff_06833.jpg
```




```
The image consist dog
The detected dog breed is:Mastiff
/data/dog_images/train/103.Mastiff/Mastiff_06826.jpg
```




```
The image consist dog
The detected dog breed is:Mastiff
/data/dog_images/train/103.Mastiff/Mastiff_06871.jpg
```



```
The image consist dog
The detected dog breed is:Chinese shar-pei
```

In [ ]: