dog_app

February 22, 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 [21]: 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

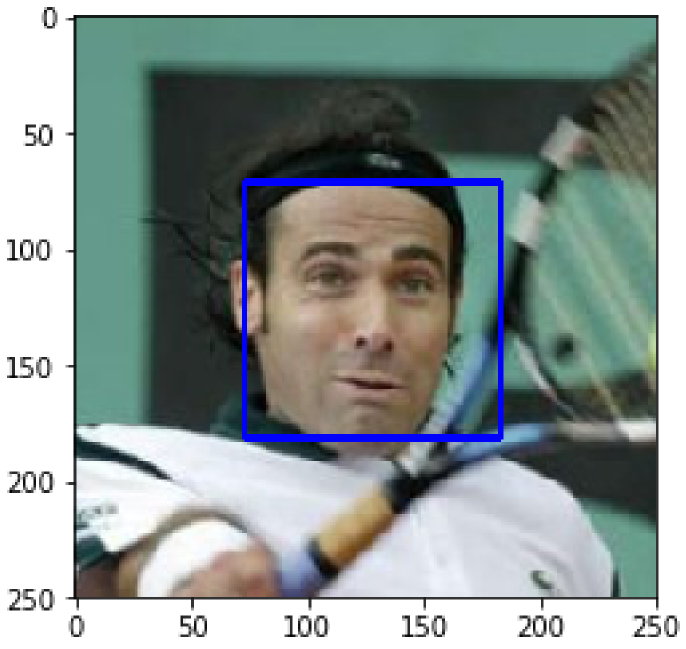
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [22]: 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 [23]: # 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:

```
In [24]: 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.
         humans = 0
         dogs = 0

         for human in human_files_short:
             if(face_detector(human)):
                 humans += 1
         for dog in dog_files_short:
             if(face_detector(dog)):
                 dogs += 1

         print("Humans: "+str(humans)+"%")
         print("Dogs: "+str(dogs)+"%")
```

```
Humans: 98%
Dogs: 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 [25]: ### (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 [26]: 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()
```

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 [27]: from PIL import Image
         import torchvision.transforms as transforms

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         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)

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

    image = in_transform(image).unsqueeze_(0)

    if use_cuda:
        image = image.cuda()

    output = VGG16(image)

    _, x = torch.max(output, 1)

    if x >= 0 and x <= 999:
        return x
    return None # 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 [28]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
```

```
    x = VGG16_predict(img_path)
    if x >= 151 and x <= 268:
        return True
    else:
        return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

```
In [29]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         humans = 0
         dogs = 0

         for human in human_files_short:
             if(dog_detector(human)):
                 humans += 1
         for dog in dog_files_short:
             if(dog_detector(dog)):
                 dogs += 1

         print("Humans: "+str(humans)+"%")
         print("Dogs: "+str(dogs)+"%")
```

```
Humans: 0%
Dogs: 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 [30]: ### (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 [31]: import zipfile


         #with zipfile.ZipFile("lfw.zip","r") as zip_ref:
         #    zip_ref.extractall()
```

```
In [32]: import os
         from torchvision import datasets

         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

data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')
valid_dir = os.path.join(data_dir, 'valid')

transform = transforms.Compose([
transforms.Resize(256),
transforms.CenterCrop(224),
transforms.ToTensor(),
transforms.Normalize(mean=[0.5, 0.5, 0.5],
std=[0.5, 0.5, 0.5])
])

train_data = datasets.ImageFolder(train_dir, transform=transform)
test_data = datasets.ImageFolder(test_dir, transform=transform)
valid_data = datasets.ImageFolder(valid_dir, transform=transform)

batch_size = 20
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=Tru
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffle=T

loaders_scratch = {
'train': train_loader,
'valid': valid_loader,
'test': test_loader
}
```

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:

My code resizes the images to a 256 size, and then crops the image to a 224 size. This is done in order to avoid cropping a big amount of the image, in case of it being too large. The size of 224 * 224 images was chosen after some research over the internet.

I decided not to augment the dataset, because I wanted to have the same translations applied to all the datasets. Augmenting the dataset would be an improvement of this project

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [33]: import torch.nn as nn
         import torch.nn.functional as F
```

```python
# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.conv4 = nn.Conv2d(64, 128, 3)
        self.conv5 = nn.Conv2d(128, 256, 3)

        self.pool = nn.MaxPool2d(2, 2)
        self.batch1 = nn.BatchNorm2d(16)
        self.batch2 = nn.BatchNorm2d(32)
        self.batch3 = nn.BatchNorm2d(64)
        self.batch4 = nn.BatchNorm2d(128)
        self.batch5 = nn.BatchNorm2d(256)

        self.fc1 = nn.Linear(256 * 5 * 5, 512)
        self.fc2 = nn.Linear(512, 133)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(self.batch1(F.relu(self.conv1(x))))
        x = self.pool(self.batch2(F.relu(self.conv2(x))))
        x = self.pool(self.batch3(F.relu(self.conv3(x))))
        x = self.pool(self.batch4(F.relu(self.conv4(x))))
        x = self.pool(self.batch5(F.relu(self.conv5(x))))

        x = x.view(-1, 5 * 5 * 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer:

At first, I started with the same model used in the CIFAR example, in the CNN lesson, in order to obtain a point of reference about how far I was from the desired loss target (10%).

After some hypermarameter tuning, I decided to change the optimizer to Adam, remove the momemtum and lower the learning rate, and the model accuracy improved until 5%.

Then, based on what was seen in the previous lessons, I decided the model would probably need more layers, so I kept adding more layers and checking the performance. This improved the results until a 6-7%.

Finally, after being stuck for some time adding more layers, I decided to do some research, and found that I should try to add Batch Normalization. It allowed me to finally have a validation accuracy over 10%.

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 [34]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=1e-5)
```

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

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders["train"]):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
```

```
                ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                optimizer.zero_grad()
                output = model(data)
                loss = criterion(output, target)
                loss.backward()
                optimizer.step()
                train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders["valid"]):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                output = model(data)
                loss = criterion(output, target)
                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 ...'.fo
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss
        # return trained model
        return model

    # train the model
    model_scratch = train(20, 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: 4.840322         Validation Loss: 4.759336
Validation loss decreased (inf --> 4.759336).  Saving model ...
Epoch: 2         Training Loss: 4.610448         Validation Loss: 4.593342
Validation loss decreased (4.759336 --> 4.593342).  Saving model ...
```

```
Epoch: 3         Training Loss: 4.377165         Validation Loss: 4.456581
Validation loss decreased (4.593342 --> 4.456581).  Saving model ...
Epoch: 4         Training Loss: 4.170690         Validation Loss: 4.357640
Validation loss decreased (4.456581 --> 4.357640).  Saving model ...
Epoch: 5         Training Loss: 3.981411         Validation Loss: 4.270709
Validation loss decreased (4.357640 --> 4.270709).  Saving model ...
Epoch: 6         Training Loss: 3.808099         Validation Loss: 4.213124
Validation loss decreased (4.270709 --> 4.213124).  Saving model ...
Epoch: 7         Training Loss: 3.630831         Validation Loss: 4.153650
Validation loss decreased (4.213124 --> 4.153650).  Saving model ...
Epoch: 8         Training Loss: 3.467814         Validation Loss: 4.103078
Validation loss decreased (4.153650 --> 4.103078).  Saving model ...
Epoch: 9         Training Loss: 3.301696         Validation Loss: 4.060882
Validation loss decreased (4.103078 --> 4.060882).  Saving model ...
Epoch: 10         Training Loss: 3.140786         Validation Loss: 4.028663
Validation loss decreased (4.060882 --> 4.028663).  Saving model ...
Epoch: 11         Training Loss: 2.984076         Validation Loss: 4.000325
Validation loss decreased (4.028663 --> 4.000325).  Saving model ...
Epoch: 12         Training Loss: 2.817748         Validation Loss: 3.973617
Validation loss decreased (4.000325 --> 3.973617).  Saving model ...
Epoch: 13         Training Loss: 2.662048         Validation Loss: 3.952837
Validation loss decreased (3.973617 --> 3.952837).  Saving model ...
Epoch: 14         Training Loss: 2.512618         Validation Loss: 3.935135
Validation loss decreased (3.952837 --> 3.935135).  Saving model ...
Epoch: 15         Training Loss: 2.354098         Validation Loss: 3.917017
Validation loss decreased (3.935135 --> 3.917017).  Saving model ...
Epoch: 16         Training Loss: 2.209976         Validation Loss: 3.904017
Validation loss decreased (3.917017 --> 3.904017).  Saving model ...
Epoch: 17         Training Loss: 2.056247         Validation Loss: 3.887494
Validation loss decreased (3.904017 --> 3.887494).  Saving model ...
Epoch: 18         Training Loss: 1.915071         Validation Loss: 3.880890
Validation loss decreased (3.887494 --> 3.880890).  Saving model ...
Epoch: 19         Training Loss: 1.772144         Validation Loss: 3.874207
Validation loss decreased (3.880890 --> 3.874207).  Saving model ...
Epoch: 20         Training Loss: 1.636775         Validation Loss: 3.867334
Validation loss decreased (3.874207 --> 3.867334).  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 [36]: 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: 3.792592


Test Accuracy: 14% (118/836)
```

## 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 [37]: ## TODO: Specify data loaders
```

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

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

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

         model_transfer.classifier[6] = torch.nn.Linear(4096, 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:
Based on the Transfer Learning lesson, I decided to do the same as in that notebook, since then it worked pretty well, and started with the VGG16 model, modifiying the last layer to fit the current problem. Fortunately, that model ended up working pretty well on the current problem at the first try.
Based on what was seen in that same lesson, and in the Style Transfer lesson, about the VGG16 architecture, I think this model is suitable for the current problem because the dataset it was trained with already contained images about dogs, and differenciated between dog breeds, as we saw in the step 2 of this same notebook. Because of it, the convolutional part of the model, which remains unchanged, already knew of to obtain dog attributes and classify them.

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 [39]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=1e-5)
```

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 [40]: # train the model
         model_transfer = train(20, loaders_scratch, model_transfer, optimizer_transfer,
                                criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
        # 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: 3.856641 	Validation Loss: 1.782950
Validation loss decreased (inf --> 1.782950).  Saving model ...
Epoch: 2 	Training Loss: 1.567901 	Validation Loss: 0.894707
Validation loss decreased (1.782950 --> 0.894707).  Saving model ...
Epoch: 3 	Training Loss: 0.988500 	Validation Loss: 0.718225
Validation loss decreased (0.894707 --> 0.718225).  Saving model ...
Epoch: 4 	Training Loss: 0.733581 	Validation Loss: 0.625941
Validation loss decreased (0.718225 --> 0.625941).  Saving model ...
Epoch: 5 	Training Loss: 0.584139 	Validation Loss: 0.577500
Validation loss decreased (0.625941 --> 0.577500).  Saving model ...
Epoch: 6 	Training Loss: 0.443392 	Validation Loss: 0.539133
Validation loss decreased (0.577500 --> 0.539133).  Saving model ...
Epoch: 7 	Training Loss: 0.376721 	Validation Loss: 0.521288
Validation loss decreased (0.539133 --> 0.521288).  Saving model ...
Epoch: 8 	Training Loss: 0.310989 	Validation Loss: 0.498052
Validation loss decreased (0.521288 --> 0.498052).  Saving model ...
Epoch: 9 	Training Loss: 0.254171 	Validation Loss: 0.483259
Validation loss decreased (0.498052 --> 0.483259).  Saving model ...
Epoch: 10 	Training Loss: 0.193377 	Validation Loss: 0.488287
Epoch: 11 	Training Loss: 0.167109 	Validation Loss: 0.478775
Validation loss decreased (0.483259 --> 0.478775).  Saving model ...
Epoch: 12 	Training Loss: 0.144361 	Validation Loss: 0.468428
Validation loss decreased (0.478775 --> 0.468428).  Saving model ...
Epoch: 13 	Training Loss: 0.116582 	Validation Loss: 0.458774
Validation loss decreased (0.468428 --> 0.458774).  Saving model ...
Epoch: 14 	Training Loss: 0.102623 	Validation Loss: 0.461350
Epoch: 15 	Training Loss: 0.082290 	Validation Loss: 0.459981
Epoch: 16 	Training Loss: 0.069121 	Validation Loss: 0.453337
Validation loss decreased (0.458774 --> 0.453337).  Saving model ...
Epoch: 17 	Training Loss: 0.064904 	Validation Loss: 0.458249
Epoch: 18 	Training Loss: 0.052464 	Validation Loss: 0.476337
Epoch: 19 	Training Loss: 0.043151 	Validation Loss: 0.473436
Epoch: 20 	Training Loss: 0.041855 	Validation Loss: 0.453314
Validation loss decreased (0.453337 --> 0.453314).  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 [41]: test(loaders_scratch, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.547798
Test Accuracy: 83% (697/836)
```

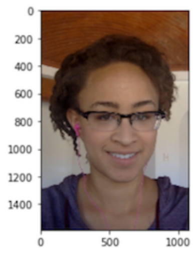
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

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

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             model_transfer.eval()

             image = Image.open(img_path).convert('RGB')

             image_transforms = 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])])

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

             if use_cuda:
                 image = image.cuda()
             output = model_transfer(image)

             if use_cuda:
                 return class_names[output.data.cpu().numpy().argmax()]
             else:
                 return class_names[output.data.numpy().argmax()]
```

## 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 [43]: ### 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
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()

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

             image = in_transform(img).unsqueeze_(0)

             if use_cuda:
                 image = image.cuda()

             output = VGG16(image)

             _, x = torch.max(output, 1)

             img2 = cv2.imread(img_path)
             # convert BGR image to grayscale
```

```
        gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

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

        if len(faces) > 0:
            print("Hello Human! Your lovely face looks like a... ", predict_breed_transfer(
        elif x >= 151 and x <= 268:
            print("Hello dog! I guess your breed is... ", predict_breed_transfer(img_path))
        else:
            print("I can't see any human or dogs out there!")
```

## 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.

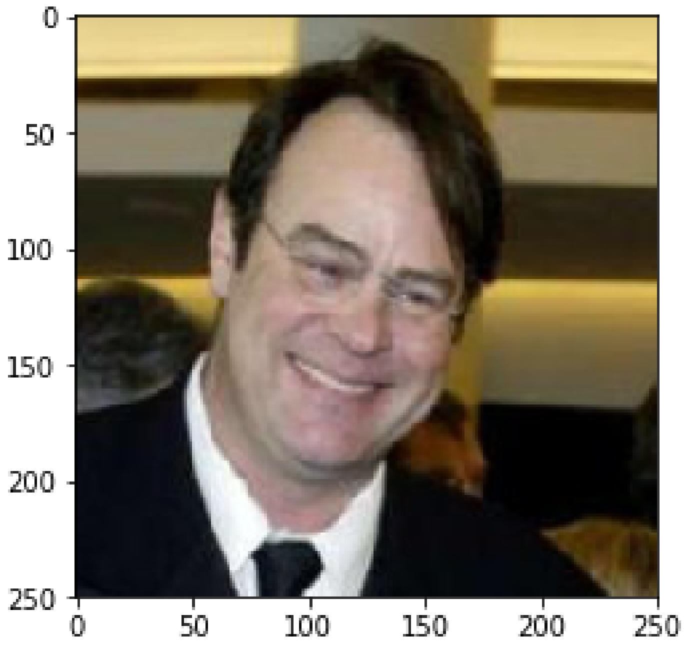
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

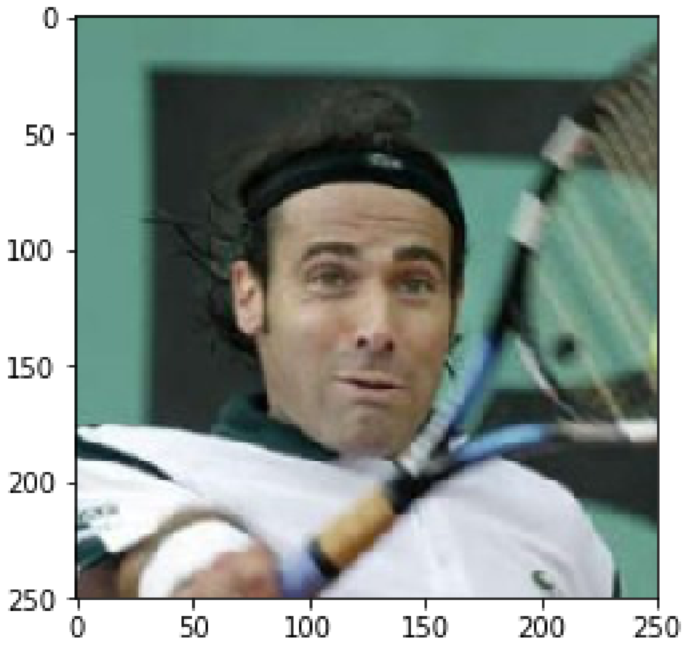
The output is better than I expected, specially looking at the accuracy of the pre-trained model. Some possible improvements are:

- Augmenting the training dataset, adding some random flip or crop
- Training for a longer time. 20 steps is fine for the current accuracy threshold, but a longer number of steps may obtain better results
- Adding Dropout to avoid overfitting, specially in the scratch model
- Trying different pretrained models. Since only the VGG16 was tested, other models may obtain better performance than it.
- Increase the accuracy of the face detection by using other face detector.
- Increase the dataset, specially in the breeds that we know the model struggles to predict properly
- Hyperparameter tuning may increase the performance.

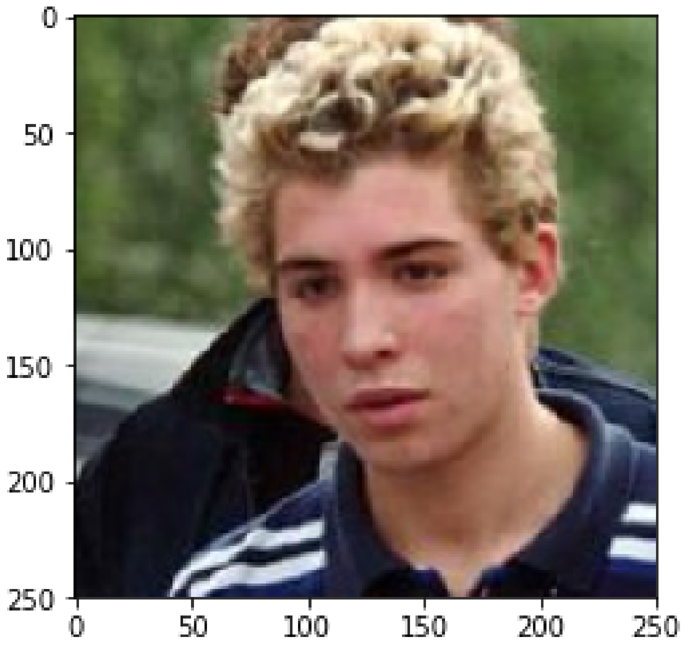
```
In [44]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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



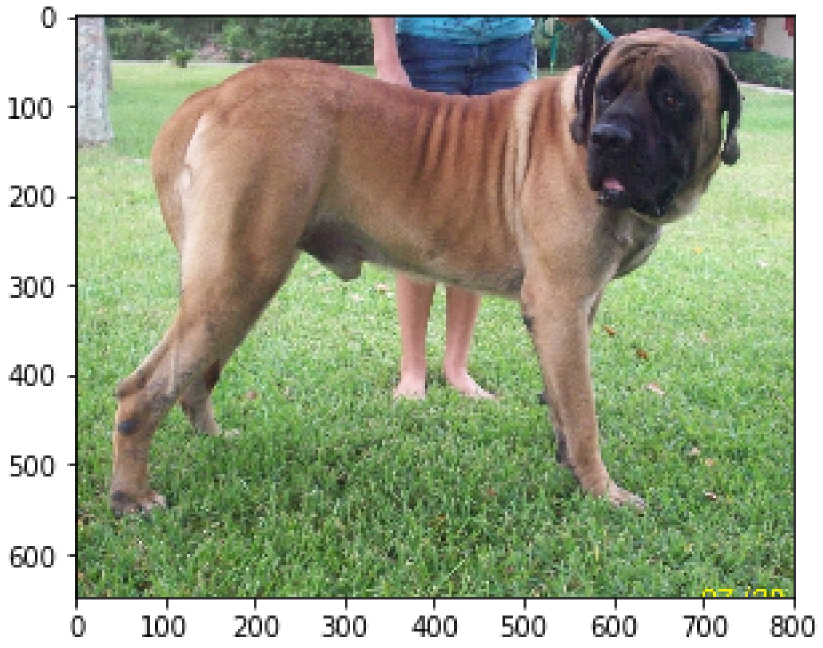
```
Hello Human! Your lovely face looks like a...  Welsh springer spaniel
```



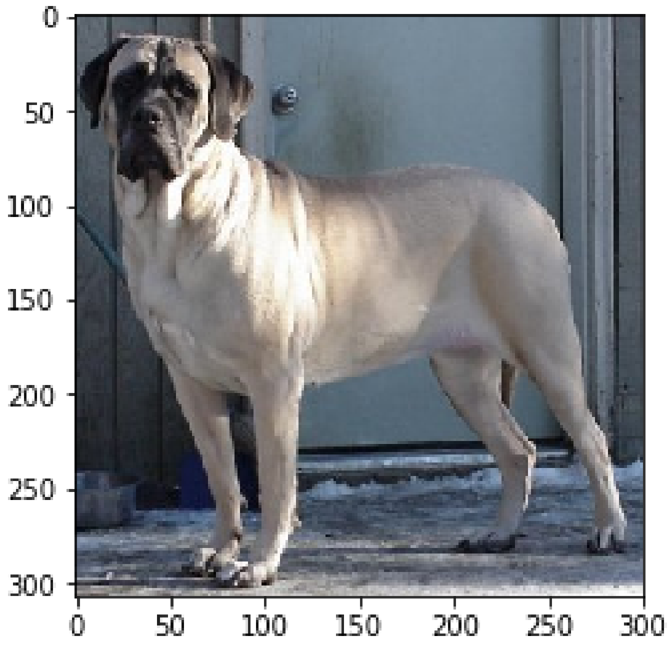
```
Hello Human! Your lovely face looks like a...  Chinese crested
```



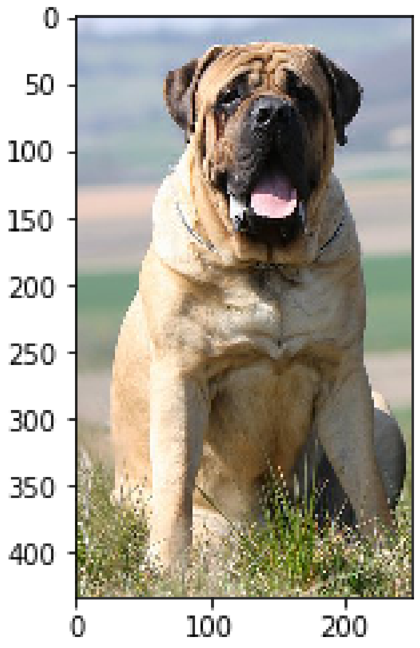
```
Hello Human! Your lovely face looks like a...  Irish water spaniel
```



```
Hello dog! I guess your breed is...  Bullmastiff
```



```
Hello dog! I guess your breed is...  Bullmastiff
```



```
Hello dog! I guess your breed is...  Mastiff
```

```
In [ ]:
```