dog_app

May 5, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
In [1]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

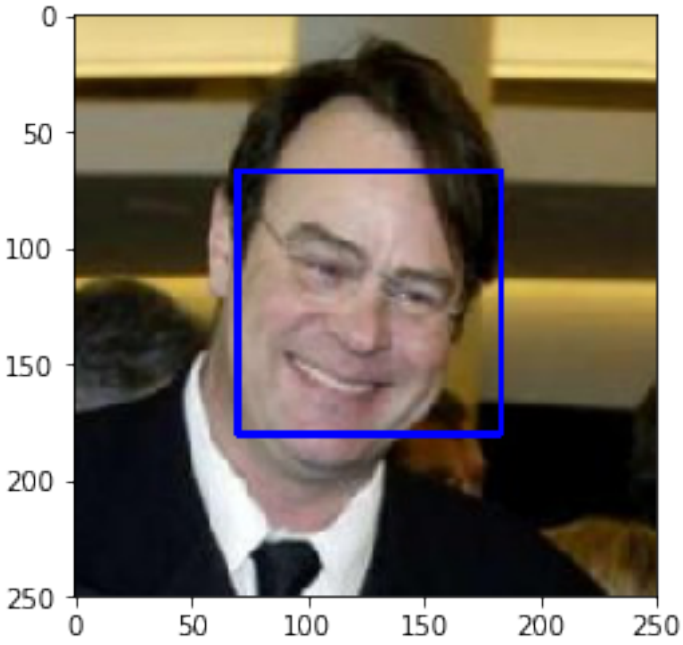
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

        # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # load color (BGR) image
        img = cv2.imread(human_files[0])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer:

Original implementation: The human face detector classified 98% of the Human Faces in the Human Face Dataset correctly as Human Face. The human face detector classified 17% of the Dogs in the Dogs Dataset incorrectly as Human Face.

Tweaked parameters implementation: The human face detector classified 93% of the Human Faces in the Human Face Dataset correctly as Human Face. The human face detector classified 5% of the Dogs in the Dogs Dataset incorrectly as Human Face.

```
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. #-#-#

        faces_in_human = 0
        faces_in_dog = 0

        for i in range(0,100):
            img_path_human = human_files_short[i]
            if face_detector(img_path_human):
                faces_in_human += 1
            img_path_dog = dog_files_short[i]
            if face_detector(img_path_dog):
                faces_in_dog += 1
```

```
        print("Percentage of faces detected in Human Face Dataset: ", faces_in_human, "%")
        print("Percentage of faces detected in Dog Face Dataset: ", faces_in_dog, "%")
```

```
Percentage of faces detected in Human Face Dataset:  98 %
Percentage of faces detected in Dog Face Dataset:  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]: ## We are tweaking the performance of the algorithm by tweaking the values of the detect
        ## The OpenCV Haar cascade seems to the state-of-the-art approach to face detection but
        ## throughout the internet such as "towardsdatascience" including a scale factor and min
        ## is suggested.

        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray, 1.15, 5) ### <----- Tweak here.
            return len(faces) > 0

        faces_in_human = 0
        faces_in_dog = 0

        for i in range(0,100):
            img_path_human = human_files_short[i]
            if face_detector(img_path_human):
                faces_in_human += 1
            img_path_dog = dog_files_short[i]
            if face_detector(img_path_dog):
                faces_in_dog += 1

        print("Percentage of faces detected in Human Face Dataset: ", faces_in_human, "%")
        print("Percentage of faces detected in Dog Face Dataset: ", faces_in_dog, "%")
```

```
Percentage of faces detected in Human Face Dataset:  93 %
Percentage of faces detected in Dog Face Dataset:  5 %
```

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()
```

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
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True
        import torchvision.transforms as transforms
        import torch.nn.functional as F

        def VGG16_predict(img_path):

            VGG16.eval()

            transform = transforms.Compose([transforms.Resize(size=(256)),
                                            transforms.CenterCrop(224),
                                            transforms.ToTensor(),
                                            transforms.Normalize((0.485, 0.456, 0.406), (0.229,

            img = Image.open(img_path)
            img_transform = transform(img)
```

```
    img_input = img_transform.unsqueeze(0)

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

    vgg_out = VGG16(img_input)

    if use_cuda:
        vgg_out = vgg_out.cpu()

    result = vgg_out.data.numpy().argmax()

    return result
```

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]: def dog_detector(img_path):

            pred = VGG16_predict(img_path)

            return ((pred >= 151) & (pred <= 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:

My implementation: The dogs detector classified 0% of the Human Faces in the Human Face Dataset incorrectly as Dogs. The dogs detector classified 100% of the Dogs in the Dogs Dataset correctly as Dogs.

```
In [9]: dogs_in_human = 0
        dogs_in_dog = 0

        for i in range(0,100):
            img_path_human = human_files_short[i]
            if dog_detector(img_path_human):
                dogs_in_human += 1
            img_path_dog = dog_files_short[i]
```

```
            if dog_detector(img_path_dog):
                dogs_in_dog += 1

        print("Percentage of dogs detected in Human Face Dataset: ", dogs_in_human, "%")
        print("Percentage of dogs detected in Dog Face Dataset: ", dogs_in_dog, "%")
```

```
Percentage of dogs detected in Human Face Dataset:  0 %
Percentage of dogs detected in Dog Face Dataset:  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]: print("No need to continue with another pre-trained network. The VGG16 performance seem
```

```
No need to continue with another pre-trained network. The VGG16 performance seems satisfactory.
```

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.

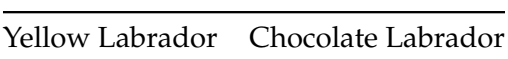
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

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

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
         from PIL import Image
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         transform_normal = transforms.Compose([transforms.Resize(size=(256)),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize((0.485, 0.456, 0.406), (0.2

         transform_augmented = transforms.Compose([transforms.Resize(size=(256)),
                                                   transforms.RandomResizedCrop(224),
                                                   transforms.RandomHorizontalFlip(), # randomly
                                                   transforms.RandomRotation(10),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize((0.485, 0.456, 0.406), (


         dog_train = datasets.ImageFolder("/data/dog_images/train/", transform=transform_augment
         dog_test = datasets.ImageFolder("/data/dog_images/test/", transform=transform_normal)
         dog_valid = datasets.ImageFolder("/data/dog_images/valid/", transform=transform_normal)

         batch_size = 20

         N_train = len(dog_train)
         N_test = len(dog_test)
         N_valid = len(dog_valid)

         train_loader = torch.utils.data.DataLoader(dog_train, batch_size=batch_size, shuffle=Tr
         test_loader = torch.utils.data.DataLoader(dog_test, batch_size=batch_size, shuffle=True
```

```
        valid_loader = torch.utils.data.DataLoader(dog_valid, batch_size=batch_size, shuffle=Fa

        print('There are %d Dog Trains Pics.' % N_train)
        print('There are %d Dog Test Pics.' % N_test)
        print('There are %d Dog Valid Pics.' % N_valid)

        N_classes = len(dog_train.classes)

        print("Number of Dog Classes: ", N_classes)
```

```
There are 6680 Dog Trains Pics.
There are 836 Dog Test Pics.
There are 835 Dog Valid Pics.
Number of Dog Classes:  133
```

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 implementation:

I encountered several difficulties when preprocessing the data, which I decided (and also while being inspired by the Jupyter notebooks during the Udacity video lectuers) to tackle as follows. Regarding the size of the images, I decided to follow the VGG16 definitions and crop a centerpiece of dimension 224x224 pixels from each image. But because randomly cropping from images of arbitrary size and aspect ratio makes no sense, I choose to first resize the arbitrarily inputted images to a size of 256x256 and crop from there within. For the normalisation of the images I again followed VGG16 guideline and choose mean and standard devitiation of the normalisation according to their suggestions.

For the augmentation I implemented a random horizontal flip (a vertical flip makes no sense... why would someone take pictures upside down???), and a small rotation of 10 degrees.

Again, this preprocesing is somewhat crude, as maybe a more manual approach to preprocessing of some images might give more sufficient results (e.g. resizing dramatically can change the aspect ratios of several dogs in the images, rendereing when non-natural and the network will learn on false data.)

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):
             def __init__(self):
                 super(Net, self).__init__()
```

```
                self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
                self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
                self.pool = nn.MaxPool2d(2, 2)

                self.fc1 = nn.Linear(128 * 14 * 14, 512)
                self.fc2 = nn.Linear(512, N_classes)
                self.dropout = nn.Dropout(1.0 / 3.0)
                self.batch1 = nn.BatchNorm2d(32)
                self.batch2 = nn.BatchNorm2d(64)
                self.batch3 = nn.BatchNorm2d(128)

            def forward(self, x):
                # size = 224
                x = self.batch1(self.pool(F.relu(self.conv1(x)))) #size = 56
                x = self.batch2(self.pool(F.relu(self.conv2(x)))) #size = 28
                x = self.batch3(self.pool(F.relu(self.conv3(x)))) #size = 14

                x = x.view(-1, 128 * 14 * 14)
                x = self.dropout(x)
                x = F.relu(self.fc1(x))
                x = self.dropout(x)
                x = self.fc2(x)
                return x

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

        # instantiate the CNN
        model_scratch = Net()
        print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=25088, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.3333333333333333)
  (batch1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

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

Answer:

I went abck to the Jupyter notebooks of the Video lectures and had a close took at the CIFAR10 example. I decided to follow the layout of the CNN with choosing 3 layers of incresing depths of 32, 64 and 128 for feature extraction. Afterwards it was common to use two fully connected layers for classification. I choose the stride=2 in the first layer, also to reduce size of data. I struggled a long time with the network architecture and, initially choose a completely oversized network of 4 layers with depth 512 and stride=1, which actually first lead to insufficient GPU memory and secondly to a stuckness of loss at 4.8 and also the training time was too long. From peer chat at Udacity I learned that 2d batch normalisation, is very useful and I implemented it after every CNN layer. So I did not decide the architecture upfront, it was more of a "look at training numbers and test error" and "go back to drawing board" approach.

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

         criterion_scratch = nn.CrossEntropyLoss()

         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)
```

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]: def train(n_epochs, train_loader, valid_loader, model, optimizer, criterion, use_cuda,
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(train_loader):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## 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

            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss += (1.0 / (batch_idx + 1.0)) * (loss.data - train_loss)

            if batch_idx % 100 == 0:
                print("Inter Epoch %d Step %d Train Loss %.4f: " % (epoch, batch_idx, t

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(valid_loader):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += (1.0 / (batch_idx + 1.0)) * (loss.data - valid_loss)

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

        # save 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(), 'model_scratch.pt')
            valid_loss_min = valid_loss

    # return trained model
    return model
```

```
        model_scratch = train(20, train_loader, valid_loader, model_scratch, optimizer_scratch,

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

```
Inter Epoch 1 Step 0 Train Loss 4.8288:
Inter Epoch 1 Step 100 Train Loss 4.8524:
Inter Epoch 1 Step 200 Train Loss 4.8011:
Inter Epoch 1 Step 300 Train Loss 4.7660:
Epoch: 1         Training Loss: 4.754153         Validation Loss: 4.483027
Validation loss decreased (inf --> 4.483027).  Saving model ...
Inter Epoch 2 Step 0 Train Loss 4.2316:
Inter Epoch 2 Step 100 Train Loss 4.5493:
Inter Epoch 2 Step 200 Train Loss 4.5401:
Inter Epoch 2 Step 300 Train Loss 4.5361:
Epoch: 2         Training Loss: 4.532821         Validation Loss: 4.315805
Validation loss decreased (4.483027 --> 4.315805).  Saving model ...
Inter Epoch 3 Step 0 Train Loss 4.1431:
Inter Epoch 3 Step 100 Train Loss 4.4148:
Inter Epoch 3 Step 200 Train Loss 4.4084:
Inter Epoch 3 Step 300 Train Loss 4.4004:
Epoch: 3         Training Loss: 4.394928         Validation Loss: 4.114999
Validation loss decreased (4.315805 --> 4.114999).  Saving model ...
Inter Epoch 4 Step 0 Train Loss 4.3136:
Inter Epoch 4 Step 100 Train Loss 4.2924:
Inter Epoch 4 Step 200 Train Loss 4.3074:
Inter Epoch 4 Step 300 Train Loss 4.2867:
Epoch: 4         Training Loss: 4.292993         Validation Loss: 4.049151
Validation loss decreased (4.114999 --> 4.049151).  Saving model ...
Inter Epoch 5 Step 0 Train Loss 4.3075:
Inter Epoch 5 Step 100 Train Loss 4.2039:
Inter Epoch 5 Step 200 Train Loss 4.1917:
Inter Epoch 5 Step 300 Train Loss 4.1945:
Epoch: 5         Training Loss: 4.196023         Validation Loss: 3.968646
Validation loss decreased (4.049151 --> 3.968646).  Saving model ...
Inter Epoch 6 Step 0 Train Loss 3.8298:
Inter Epoch 6 Step 100 Train Loss 4.1178:
Inter Epoch 6 Step 200 Train Loss 4.1058:
Inter Epoch 6 Step 300 Train Loss 4.1184:
Epoch: 6         Training Loss: 4.117053         Validation Loss: 3.982008
Inter Epoch 7 Step 0 Train Loss 3.6833:
Inter Epoch 7 Step 100 Train Loss 4.0100:
Inter Epoch 7 Step 200 Train Loss 4.0360:
Inter Epoch 7 Step 300 Train Loss 4.0237:
Epoch: 7         Training Loss: 4.028708         Validation Loss: 3.811349
Validation loss decreased (3.968646 --> 3.811349).  Saving model ...
```

```
Inter Epoch 8 Step 0 Train Loss 4.0944:
Inter Epoch 8 Step 100 Train Loss 3.9307:
Inter Epoch 8 Step 200 Train Loss 3.9746:
Inter Epoch 8 Step 300 Train Loss 3.9829:
Epoch: 8         Training Loss: 3.980670         Validation Loss: 3.786705
Validation loss decreased (3.811349 --> 3.786705).  Saving model ...
Inter Epoch 9 Step 0 Train Loss 4.1355:
Inter Epoch 9 Step 100 Train Loss 3.9510:
Inter Epoch 9 Step 200 Train Loss 3.9704:
Inter Epoch 9 Step 300 Train Loss 3.9534:
Epoch: 9         Training Loss: 3.946219         Validation Loss: 3.714442
Validation loss decreased (3.786705 --> 3.714442).  Saving model ...
Inter Epoch 10 Step 0 Train Loss 4.2070:
Inter Epoch 10 Step 100 Train Loss 3.8244:
Inter Epoch 10 Step 200 Train Loss 3.8655:
Inter Epoch 10 Step 300 Train Loss 3.8835:
Epoch: 10         Training Loss: 3.884375         Validation Loss: 3.681769
Validation loss decreased (3.714442 --> 3.681769).  Saving model ...
Inter Epoch 11 Step 0 Train Loss 3.7097:
Inter Epoch 11 Step 100 Train Loss 3.8267:
Inter Epoch 11 Step 200 Train Loss 3.8277:
Inter Epoch 11 Step 300 Train Loss 3.8265:
Epoch: 11         Training Loss: 3.821250         Validation Loss: 3.573455
Validation loss decreased (3.681769 --> 3.573455).  Saving model ...
Inter Epoch 12 Step 0 Train Loss 3.8467:
Inter Epoch 12 Step 100 Train Loss 3.7168:
Inter Epoch 12 Step 200 Train Loss 3.7744:
Inter Epoch 12 Step 300 Train Loss 3.7542:
Epoch: 12         Training Loss: 3.757131         Validation Loss: 3.570658
Validation loss decreased (3.573455 --> 3.570658).  Saving model ...
Inter Epoch 13 Step 0 Train Loss 3.7755:
Inter Epoch 13 Step 100 Train Loss 3.7237:
Inter Epoch 13 Step 200 Train Loss 3.7266:
Inter Epoch 13 Step 300 Train Loss 3.7432:
Epoch: 13         Training Loss: 3.743726         Validation Loss: 3.511258
Validation loss decreased (3.570658 --> 3.511258).  Saving model ...
Inter Epoch 14 Step 0 Train Loss 3.8873:
Inter Epoch 14 Step 100 Train Loss 3.6425:
Inter Epoch 14 Step 200 Train Loss 3.6540:
Inter Epoch 14 Step 300 Train Loss 3.6695:
Epoch: 14         Training Loss: 3.671977         Validation Loss: 3.608022
Inter Epoch 15 Step 0 Train Loss 3.7684:
Inter Epoch 15 Step 100 Train Loss 3.6808:
Inter Epoch 15 Step 200 Train Loss 3.6467:
Inter Epoch 15 Step 300 Train Loss 3.6430:
Epoch: 15         Training Loss: 3.639720         Validation Loss: 3.458668
Validation loss decreased (3.511258 --> 3.458668).  Saving model ...
Inter Epoch 16 Step 0 Train Loss 3.7706:
```

```
Inter Epoch 16 Step 100 Train Loss 3.5734:
Inter Epoch 16 Step 200 Train Loss 3.5908:
Inter Epoch 16 Step 300 Train Loss 3.6043:
Epoch: 16         Training Loss: 3.599753         Validation Loss: 3.534709
Inter Epoch 17 Step 0 Train Loss 3.6499:
Inter Epoch 17 Step 100 Train Loss 3.5767:
Inter Epoch 17 Step 200 Train Loss 3.5879:
Inter Epoch 17 Step 300 Train Loss 3.5865:
Epoch: 17         Training Loss: 3.591659         Validation Loss: 3.542568
Inter Epoch 18 Step 0 Train Loss 3.6249:
Inter Epoch 18 Step 100 Train Loss 3.4498:
Inter Epoch 18 Step 200 Train Loss 3.5071:
Inter Epoch 18 Step 300 Train Loss 3.5103:
Epoch: 18         Training Loss: 3.506181         Validation Loss: 3.516875
Inter Epoch 19 Step 0 Train Loss 3.4113:
Inter Epoch 19 Step 100 Train Loss 3.4714:
Inter Epoch 19 Step 200 Train Loss 3.5031:
Inter Epoch 19 Step 300 Train Loss 3.5096:
Epoch: 19         Training Loss: 3.517097         Validation Loss: 3.483115
Inter Epoch 20 Step 0 Train Loss 3.8282:
Inter Epoch 20 Step 100 Train Loss 3.4729:
Inter Epoch 20 Step 200 Train Loss 3.4722:
Inter Epoch 20 Step 300 Train Loss 3.4755:
Epoch: 20         Training Loss: 3.477715         Validation Loss: 3.456240
Validation loss decreased (3.458668 --> 3.456240).  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 [15]:
```python
def test(test_loader, model, criterion, use_cuda):

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

    model.eval()
    for batch_idx, (data, target) in enumerate(test_loader):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
```

```
            # update average test loss
            test_loss += (1 / (batch_idx + 1.0)) * (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(test_loader, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.395252


Test Accuracy: 18% (155/836)
```

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 [16]: import os
         import torch
         from torchvision import datasets, transforms
         from PIL import Image
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         transform_normal = transforms.Compose([transforms.Resize(size=(256)),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize((0.485, 0.456, 0.406), (0.2

         transform_augmented = transforms.Compose([transforms.Resize(size=(256)),
```

```
                                                transforms.RandomResizedCrop(224),
                                                transforms.RandomHorizontalFlip(), # randomly
                                                transforms.RandomRotation(10),
                                                transforms.ToTensor(),
                                                transforms.Normalize((0.485, 0.456, 0.406), (
```

```
dog_train = datasets.ImageFolder("/data/dog_images/train/", transform=transform_augment
dog_test = datasets.ImageFolder("/data/dog_images/test/", transform=transform_normal)
dog_valid = datasets.ImageFolder("/data/dog_images/valid/", transform=transform_normal)

batch_size = 20

N_train = len(dog_train)
N_test = len(dog_test)
N_valid = len(dog_valid)

train_loader = torch.utils.data.DataLoader(dog_train, batch_size=batch_size, shuffle=Tr
test_loader = torch.utils.data.DataLoader(dog_test, batch_size=batch_size, shuffle=True
valid_loader = torch.utils.data.DataLoader(dog_valid, batch_size=batch_size, shuffle=Fa

print('There are %d Dog Trains Pics.' % N_train)
print('There are %d Dog Test Pics.' % N_test)
print('There are %d Dog Valid Pics.' % N_valid)

N_classes = len(dog_train.classes)
print("Number of Dog Classes: ", N_classes)
```

```
There are 6680 Dog Trains Pics.
There are 836 Dog Test Pics.
There are 835 Dog Valid Pics.
Number of Dog Classes:  133
```

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 [17]: import torchvision.models as models
         import torch.nn as nn
         import torch.optim as optim
         import torch.nn.functional as F
         import numpy as np

         model_transfer = models.vgg16(pretrained=True)
         print(model_transfer.classifier)
```

```
        # Freeze training for all "features" layers
        for param in model_transfer.features.parameters():
            param.requires_grad = False

        myclassifier = nn.Sequential(nn.Linear(25088,4096), nn.ReLU(), nn.Dropout(0.5),
                                     nn.Linear(4096,768), nn.ReLU(), nn.Dropout(0.5),
                                     nn.Linear(768,N_classes))

        model_transfer.classifier = myclassifier

        print(model_transfer.classifier)

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

        if use_cuda:
            model_transfer = model_transfer.cuda()
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace)
  (2): Dropout(p=0.5)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace)
  (5): Dropout(p=0.5)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
)
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU()
  (2): Dropout(p=0.5)
  (3): Linear(in_features=4096, out_features=768, bias=True)
  (4): ReLU()
  (5): Dropout(p=0.5)
  (6): Linear(in_features=768, out_features=133, bias=True)
)
```

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:

My implementation:

I went back to the Transfer Learning Videos of this course. Then I printed the classifier of the pre-trained VGG16 to inspect the existing number of layers and nodes. Then I changed the number of output nodes from the total 1000 to the number of dog classes, which is 133. Then reduction in nodes of every layer in VGG is roughly 6x times, then 1x times and then 4 times. I decided to use an even spacing of a factor of 6x times and came up with 25088, 4096, 768 and 133 as numbers. This architecture should be powerful enough to extract features from the CNN layers and perform classification of dogs.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [18]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.01)
```

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 [19]: # train the model
         def train(n_epochs, train_loader, valid_loader, model, optimizer, criterion, use_cuda,
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(train_loader):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## 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

                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()
                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)
                     # calculate the batch loss
                     loss = criterion(output, target)
                     # backward pass: compute gradient of the loss with respect to model paramet
                     loss.backward()
                     # perform a single optimization step (parameter update)
                     optimizer.step()
                     # update training loss
                     train_loss += (1.0 / (batch_idx + 1.0)) * (loss.data - train_loss)
```

```
                if batch_idx % 100 == 0:
                    print("Inter Epoch %d Step %d Train Loss %.4f: " % (epoch, batch_idx, t

            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(valid_loader):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss += (1.0 / (batch_idx + 1.0)) * (loss.data - valid_loss)

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

            # save 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(), 'model_transfer.pt')
                valid_loss_min = valid_loss

        # return trained model
        return model


    model_transfer = train(8, train_loader, valid_loader, model_transfer, optimizer_transfe

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

```
Inter Epoch 1 Step 0 Train Loss 5.0063:
Inter Epoch 1 Step 100 Train Loss 4.7943:
Inter Epoch 1 Step 200 Train Loss 4.5771:
Inter Epoch 1 Step 300 Train Loss 4.2726:
Epoch: 1         Training Loss: 4.167273         Validation Loss: 2.113050
Validation loss decreased (inf --> 2.113050).  Saving model ...
Inter Epoch 2 Step 0 Train Loss 2.4986:
Inter Epoch 2 Step 100 Train Loss 2.9425:
Inter Epoch 2 Step 200 Train Loss 2.7882:
```

```
Inter Epoch 2 Step 300 Train Loss 2.6752:
Epoch: 2         Training Loss: 2.644864         Validation Loss: 1.174066
Validation loss decreased (2.113050 --> 1.174066).  Saving model ...
Inter Epoch 3 Step 0 Train Loss 2.3101:
Inter Epoch 3 Step 100 Train Loss 2.2004:
Inter Epoch 3 Step 200 Train Loss 2.1260:
Inter Epoch 3 Step 300 Train Loss 2.0698:
Epoch: 3         Training Loss: 2.045385         Validation Loss: 0.903130
Validation loss decreased (1.174066 --> 0.903130).  Saving model ...
Inter Epoch 4 Step 0 Train Loss 1.6484:
Inter Epoch 4 Step 100 Train Loss 1.8450:
Inter Epoch 4 Step 200 Train Loss 1.8523:
Inter Epoch 4 Step 300 Train Loss 1.8244:
Epoch: 4         Training Loss: 1.828307         Validation Loss: 0.766859
Validation loss decreased (0.903130 --> 0.766859).  Saving model ...
Inter Epoch 5 Step 0 Train Loss 2.1300:
Inter Epoch 5 Step 100 Train Loss 1.6949:
Inter Epoch 5 Step 200 Train Loss 1.6737:
Inter Epoch 5 Step 300 Train Loss 1.6748:
Epoch: 5         Training Loss: 1.668320         Validation Loss: 0.704765
Validation loss decreased (0.766859 --> 0.704765).  Saving model ...
Inter Epoch 6 Step 0 Train Loss 1.2866:
Inter Epoch 6 Step 100 Train Loss 1.6232:
Inter Epoch 6 Step 200 Train Loss 1.6180:
Inter Epoch 6 Step 300 Train Loss 1.6186:
Epoch: 6         Training Loss: 1.621836         Validation Loss: 0.696546
Validation loss decreased (0.704765 --> 0.696546).  Saving model ...
Inter Epoch 7 Step 0 Train Loss 1.4697:
Inter Epoch 7 Step 100 Train Loss 1.5173:
Inter Epoch 7 Step 200 Train Loss 1.5331:
Inter Epoch 7 Step 300 Train Loss 1.5063:
Epoch: 7         Training Loss: 1.512454         Validation Loss: 0.658958
Validation loss decreased (0.696546 --> 0.658958).  Saving model ...
Inter Epoch 8 Step 0 Train Loss 2.0102:
Inter Epoch 8 Step 100 Train Loss 1.4429:
Inter Epoch 8 Step 200 Train Loss 1.4576:
Inter Epoch 8 Step 300 Train Loss 1.4790:
Epoch: 8         Training Loss: 1.461810         Validation Loss: 0.634943
Validation loss decreased (0.658958 --> 0.634943).  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 [20]: def test(test_loader, model, criterion, use_cuda):
```

```
            # monitor test loss and accuracy
            test_loss = 0.0
            correct = 0.0
            total = 0.0

            model.eval()
            for batch_idx, (data, target) in enumerate(test_loader):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the loss
                loss = criterion(output, target)
                # update average test loss
                test_loss += (1 / (batch_idx + 1.0)) * (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))

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

```
Test Loss: 0.684134


Test Accuracy: 79% (665/836)
```

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 [21]: import heapq

        def predict_breed_transfer(img_path):
            # load the image and return the predicted breed
            # will return a list of 4 values
            # First value: Most likely breed name
            # Second value: Model prediction value for most likely breed
            # Third value: Second most likely breed name
```

```
    # Fourth value: Model prediciton value for second most likely breed

    model_transfer.eval()

    img = Image.open(img_path)
    img_transform = transform_normal(img)
    img_input = img_transform.unsqueeze(0)

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

    predict = model_transfer(img_input)

    if use_cuda:
        predict = predict.cpu()

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

    prob = np.squeeze(np.asarray(predict.data.numpy()))

    breeds = heapq.nlargest(2, range(len(prob)), key=prob.__getitem__)
    probs = heapq.nlargest(2,prob)

    result = [class_names[breeds[0]], probs[0], class_names[breeds[1]], probs[1]]

    return result
```

Step 5: Write your Algorithm

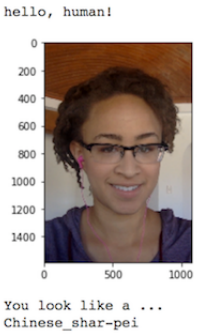
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is 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 [22]: def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    if face_detector(img_path):
        print("The following picture contains a Human:")
        plt.imshow(Image.open(img_path))
```



Sample Human Output

```
    plt.show()
elif dog_detector(img_path):
    dog_breed = predict_breed_transfer(img_path)
    print("First breed prediction: ", dog_breed[0], " with a value of ", dog_breed[
    print("Second breed prediction: ", dog_breed[2], " with a value of ", dog_breed
    print("If both values are close to each other, it might aswell be a mixed dog b
    plt.imshow(Image.open(img_path))
    plt.show()
else:
    print("I am clueless, Scotty beam me up.")
    plt.imshow(Image.open(img_path))
    plt.show()
print("-------------------------------------------------")
```

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:

First improvmenent:

The preprocessing of the images could be improved. At the moment the algorithm resizes and crops images of arbitrary size and aspect ratio into squares of 224x224 pixels. Here inseat of cropping at the center, the cropping could be performed at the center of the dog bounding box, which would need to be outputted by the "dog_detector".

Second improvmenent:

The Fédération Cynologique Internationale recognizes 344 individual dog classes worldwide. Our algorithm only takes into account 133 dog classes, so it is by construction bound to misclassify.

Therefore, the model should be extended to include all dog classes and the training should be performed on purebred dog data. Currently, the dog images data set contains also mixed dog breed data and the algorithm is bound to fail on that.

Third improvmenent:

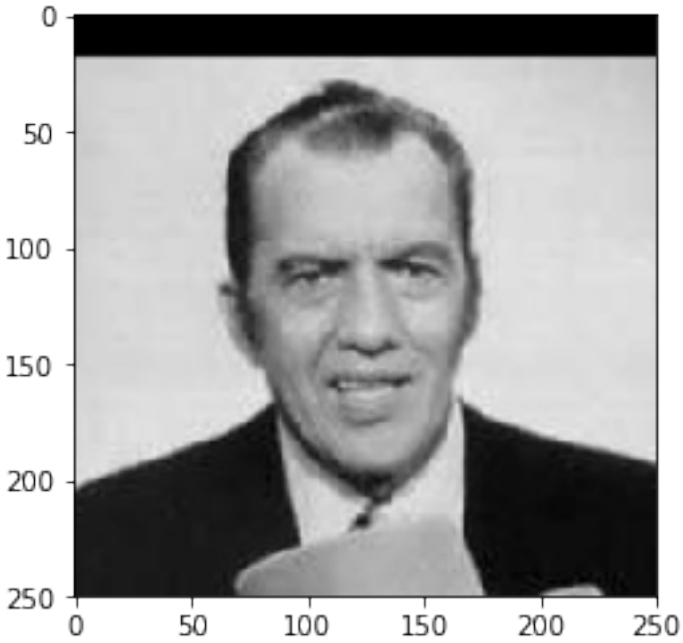
Some pictures contain two dogs or one dog and one human. The algorithm will then have difficulties distinguishing the human and dog features within the pictures. Therefore, I propose an alternative pre-processing step, where all images that get recognized with two or more faces... or with three or more eyes will be manually preprocessed and cleaned. The algorithm can only be produce high quality, if the data is of high quality.

Fourth improvment:

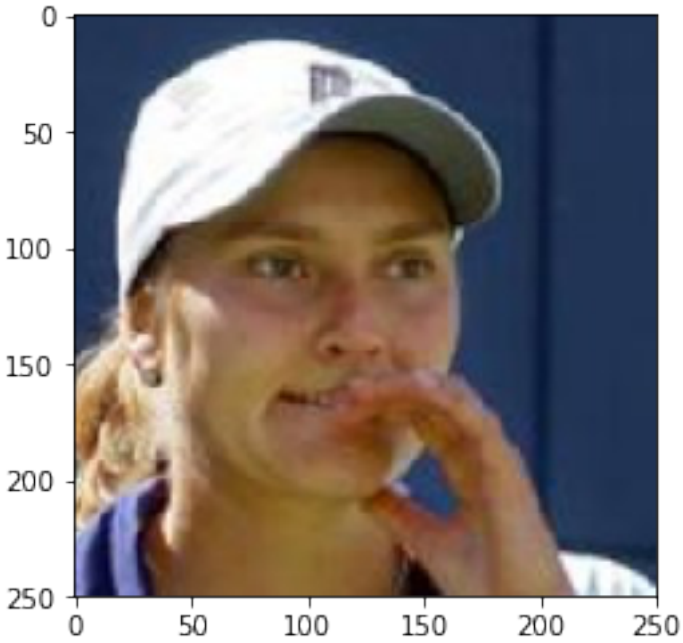
The number of CNN layers and the depth of each layer certainly hold room for further investigations. So the depth of the CNN could be increased up to 512, the stride decreased to 1 or one or two more fully connected layers. However, then training time will increase greatly.

```
In [29]: from numpy import random

         print("TESTING ON HUMAN FACE DATASET:")
         print("------------------------------")
         for file in np.hstack(random.choice(human_files,3)):
             run_app(file)
```

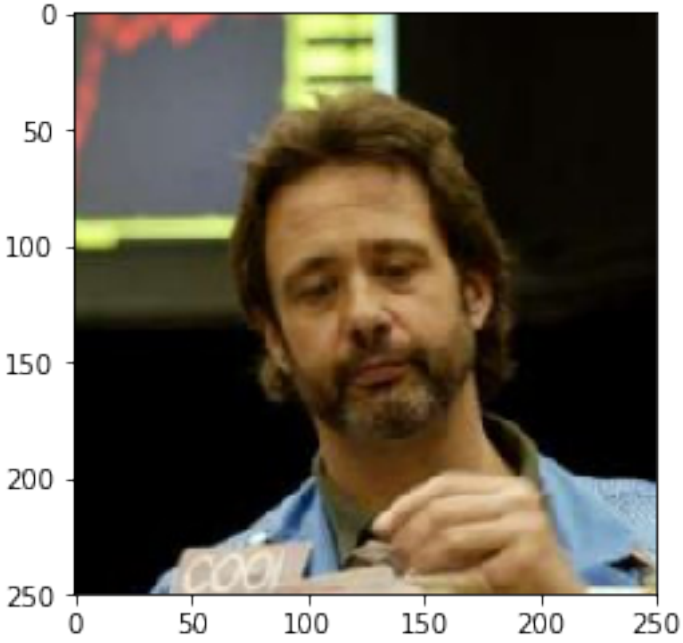
```
TESTING ON HUMAN FACE DATASET:
------------------------------
The following picture contains a Human:
```



```
-------------------------------------------------
The following picture contains a Human:
```



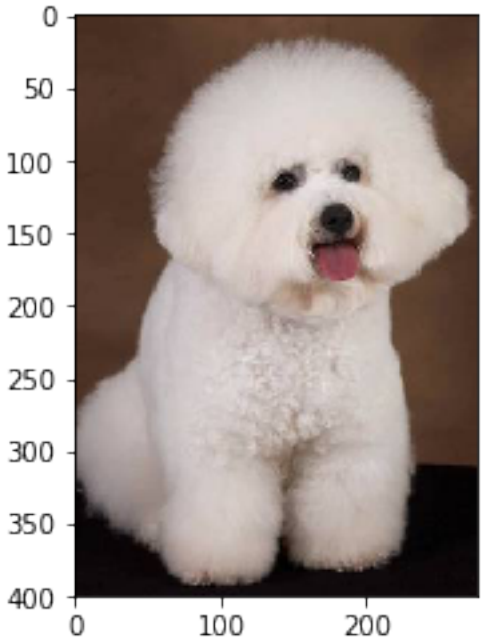

```
-------------------------------------------------
The following picture contains a Human:
```



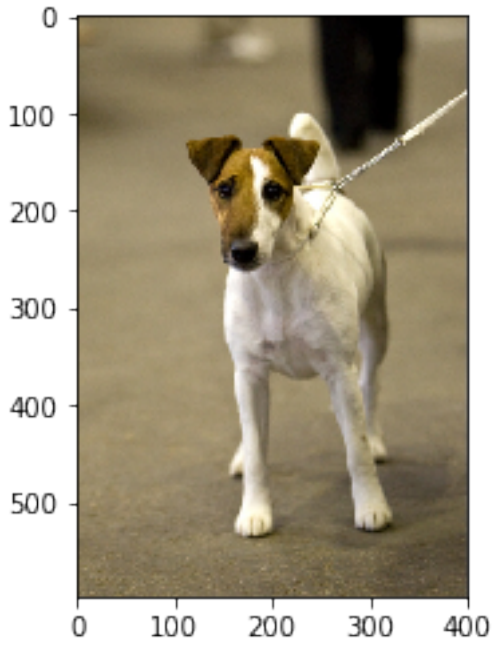
```
----------------------------------------------
```

```
In [30]: print("TESTING ON DOGS DATASET:")
         print("------------------------------")
         for file in np.hstack(random.choice(dog_files,3)):
             run_app(file)
```

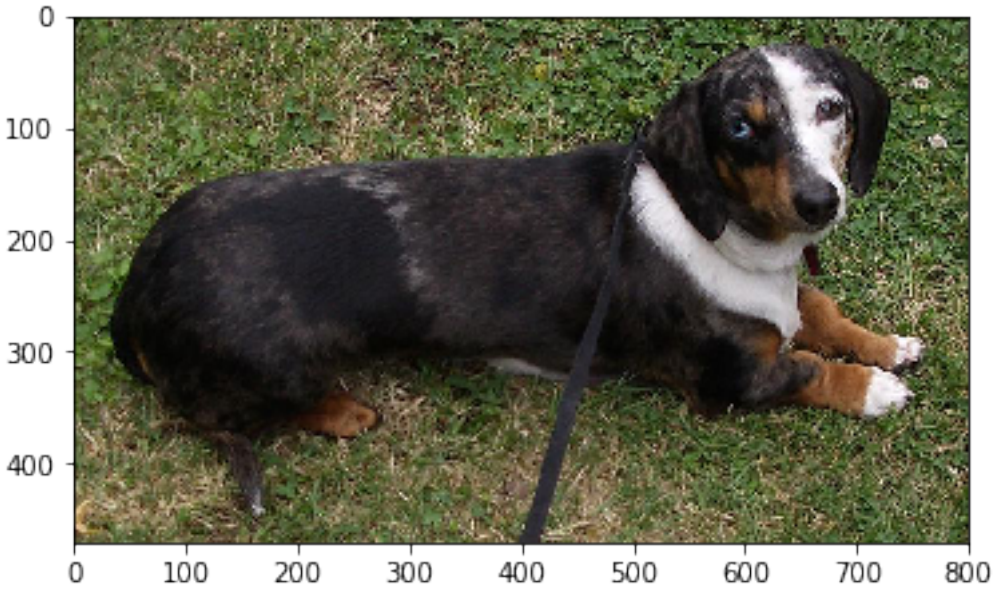
```
TESTING ON DOGS DATASET:
------------------------------
First breed prediction:  Bichon frise  with a value of  15.6028
Second breed prediction:  Maltese  with a value of  11.8769
If both values are close to each other, it might aswell be a mixed dog breed!
```




```
----------------------------------------------
First breed prediction:  Bull terrier  with a value of  12.7244
Second breed prediction:  Smooth fox terrier  with a value of  11.4444
If both values are close to each other, it might aswell be a mixed dog breed!
```




```
------------------------------------------------
First breed prediction:  Dachshund  with a value of  4.43927
Second breed prediction:  Australian shepherd  with a value of  4.33339
If both values are close to each other, it might aswell be a mixed dog breed!
```



```
-------------------------------------------------
```

In [ ]: