dog_app

June 2, 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

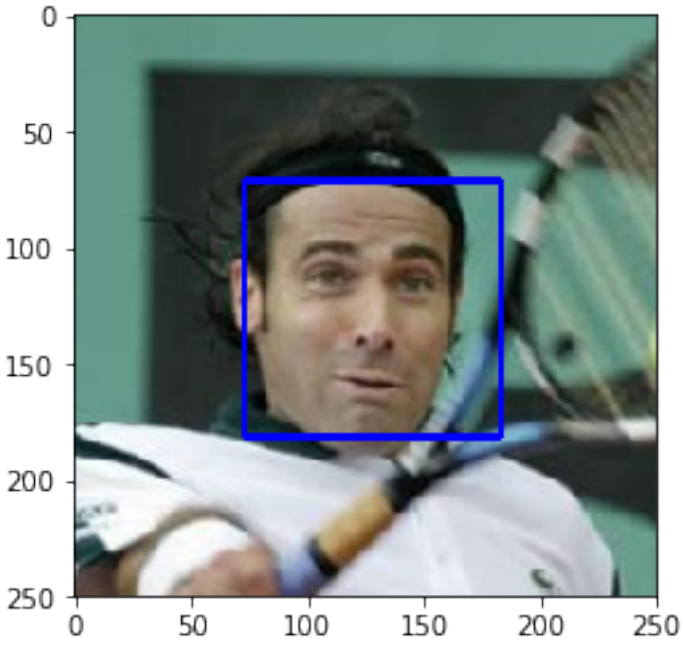
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell)
Performance on human dataset - 98 %
Performance on dog dataset - 17 %

```
In [15]: 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.
         human_per = 0.0
         dog_per = 0.0
         for human in human_files_short:
             if face_detector(human):
                 human_per +=1
         print("Performance on human dataset  - ", human_per, " % ")
         for dog in dog_files_short:
             if face_detector(dog):
                 dog_per +=1
         print("Performance on dog dataset  - ", dog_per, " % ")
```

```
Performance on human dataset  -  98.0  %
Performance on dog dataset  -  17.0  %
```

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 [16]: ### (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 [14]: import torch
         import torchvision.models as models

         # define VGG16 model
         VGG16 = models.vgg16(pretrained=True)

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

         # move model to GPU if CUDA is available
         if use_cuda:
             VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 95747433.24it/s]
```

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

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

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

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

```
In [14]: from PIL import Image
         import torchvision.transforms as transforms

         def VGG16_predict(img_path):
             '''
             Use pre-trained VGG-16 model to obtain index corresponding to
             predicted ImageNet class for image at specified path

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''

             ## TODO: Complete the function.
             ## Load and pre-process an image from the given img_path
             ## Return the *index* of the predicted class for that image
             img = Image.open(img_path).convert('RGB')
             transform = transforms.Compose([transforms.Resize(250),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize([0.485, 0.456, 0.406],[0.229,
             img = transform(img)[:3,:,:].unsqueeze(0)
             if use_cuda:
                 img = img.cuda()

             output = VGG16(img)
             _,pred = torch.max(output,1)
             pred = pred.cpu().numpy()
             return pred # 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 [15]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             output = VGG16_predict(img_path)
             if output>=151 & output <=268:
                 return True
```

```
    return False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: Performance on human dataset - 0.0 %
Performance on dog dataset - 100.0 %

```
In [20]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_files_short = human_files[:100]
         dog_files_short = dog_files[:100]
         human_per = 0.0
         dog_per = 0.0
         for human in human_files_short:
             if dog_detector(human)==False:
                 human_per +=1
         print("Performance on human dataset  - ", human_per, " % ")
         for dog in dog_files_short:
             if dog_detector(dog):
                 dog_per +=1
         print("Performance on dog dataset  - ", dog_per, " % ")
```

```
Performance on human dataset  -  0.0  %
Performance on dog dataset  -  100.0  %
```

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 [21]: ### (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 [4]: import os
        import torch
        import torchvision.models as models
        from torchvision import datasets
        from PIL import Image
        import torchvision.transforms as transforms

        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        train_dir = '/data/dog_images/train'
        valid_dir = '/data/dog_images/valid'
        test_dir = '/data/dog_images/test'
        batch_size = 64


        transform = transforms.Compose([transforms.Resize(250),
```

```
                                      transforms.CenterCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation(15),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224

testtransform = transforms.Compose([transforms.Resize(250),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224

traindata2 = datasets.ImageFolder(train_dir,transform=transform)
validdata2 = datasets.ImageFolder(valid_dir,transform=transform)
testdata2 = datasets.ImageFolder(test_dir,transform=testtransform)

train_loader2 = torch.utils.data.DataLoader(traindata2, batch_size=batch_size,shuffle= T
valid_loader2 = torch.utils.data.DataLoader(validdata2, batch_size=batch_size,shuffle =
test_loader2 = torch.utils.data.DataLoader(testdata2, batch_size=batch_size)
```

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: a- Resizing is done using transforms.resize() and finally transforms.CenterCrop() the size of input tensor is taken 224 x 224 pixel. Size is reduced to reduce the number of parameters and making training faster. this also mathces with input data size for pre-trained models so comparision can be made for same input information.

b- Yes data is augmented for training and validation dataset to increase the number of training samples by using transforms.RandomHorizontalFlip() (randomly flipping training samples along horizontal axis), transforms.RandomRotation(15) (randomly rotating images by 15 degrees) data augmentation increases number of samples and avoids overfitting

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [6]: import torch.nn as nn
        import torch.nn.functional as F

        # define the CNN architecture
        class Net(nn.Module):
            ### TODO: choose an architecture, and complete the class
            def __init__(self):
                super(Net, self).__init__()
                ## Define layers of a CNN
                self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                self.conv3 = nn.Conv2d(32, 32, 3, padding=1)
```

```
            self.conv4 = nn.Conv2d(32, 48, 3, padding=1)
            self.conv5 = nn.Conv2d(48, 64, 3, padding=1)
            self.pool = nn.MaxPool2d(2, 2)
            self.fc1 = nn.Linear(64*7*7, 1024)
            self.fc2 = nn.Linear(1024, 133)
            self.dropout = nn.Dropout(0.5)

        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.pool(F.relu(self.conv3(x)))
            x = self.pool(F.relu(self.conv4(x)))
            x = self.pool(F.relu(self.conv5(x)))
            x = x.view(-1,64*7*7)
            x = self.dropout(F.relu(self.fc1(x)))
            x = F.log_softmax(self.fc2(x), dim=1)

            return x


    use_cuda = torch.cuda.is_available()

    #-#-# 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:

We choose 5 cnn layers which increase depth from 3 to 48 (3 to 16, 16 to 32, 32, 32 to 48, 48 to 64).

The depth is increased as we go deeper in the network to increase number of filters so more fine details can be extracted.

Max pool layers of kernal size 2x2 is used after cnn layer 1 to 5 to reduce dimetionalality of output matrix of each hidden layer.

The output of final maxpool layer is flattened to 1-D array in size 64 * 7 * 7 . The size calculations are as follow -

Input image – 224x224

Passed as input with batch size of 64 – 64x224x224 = input

after passing 1st max pool layer = (224/2) * (224/2) = 112 * 112

after passing 2nd max pool layer = (112/2) * (112/2) = 56 * 56

after passing 3rd max pool layer = (56/2) * (56/2) = 28 * 28

after passing 4th max pool layer = (28/2) * (28/2) = 14 * 14
after passing 5th max pool layer = (14/2) * (14/2) = 7 * 7
thus,
Flattened image – 64 * 7 * 7 This can be also verified by printing the output size of last maxpool layer. final number of output - 133
dropout with probability of 0.5 is choosen to reduce chanaces of overfitting
ReLU activation is used to introduce non-linearity to output thus better perfromance
finally, log softmax output is used.

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 [7]: import torch.optim as optim
        from torch import nn

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

        ### TODO: select optimizer
        optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01, momentum = 0.7)
```

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 [8]: def train(n_epochs, trainloaders, validloaders, model, optimizer, criterion, use_cuda, s
            """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()
                try:
                    for batch_idx, (data, target) in enumerate(trainloaders):
                        # 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
            optimizer.zero_grad()
            logps = model(data)
            loss = criterion(logps, target)
            loss.backward()
            optimizer.step()
            train_loss += (1 / (batch_idx + 1)) * (loss.item() - train_loss)
    except OSError:
        pass

    ######################
    # validate the model #
    ######################
    valid_acc = 0.0
    model.eval()
    with torch.no_grad():
        try:
            for batch_idx, (data, target) in enumerate(validloaders):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                logps = model(data)
                batch_loss = criterion(logps, target)
                valid_loss +=(1 / (batch_idx + 1)) * (batch_loss.item() - valid_loss

        except OSError:
            pass

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

    ## TODO: save the model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Saving model ...')
        torch.save(model.state_dict(), save_path)

# return trained model
return model
```

```
In [9]: # train the model
        model_scratch = train(40, train_loader2,valid_loader2, 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: 4.890723         Validation Loss: 4.890548
Saving model ...
Epoch: 2         Training Loss: 4.888617         Validation Loss: 4.888326
Saving model ...
Epoch: 3         Training Loss: 0.000000         Validation Loss: 4.887942
Saving model ...
Epoch: 4         Training Loss: 4.886107         Validation Loss: 4.885780
Saving model ...
Epoch: 5         Training Loss: 4.881263         Validation Loss: 4.872320
Saving model ...
Epoch: 6         Training Loss: 4.872860         Validation Loss: 4.882848
Saving model ...
Epoch: 7         Training Loss: 4.863371         Validation Loss: 4.869353
Saving model ...
Epoch: 8         Training Loss: 4.854600         Validation Loss: 4.850084
Saving model ...
Epoch: 9         Training Loss: 4.875667         Validation Loss: 4.856214
Saving model ...
Epoch: 10         Training Loss: 4.846484         Validation Loss: 4.823047
Saving model ...
Epoch: 11         Training Loss: 4.798819         Validation Loss: 4.761028
Saving model ...
Epoch: 12         Training Loss: 4.748853         Validation Loss: 4.706467
Saving model ...
Epoch: 13         Training Loss: 4.704163         Validation Loss: 4.731790
Saving model ...
Epoch: 14         Training Loss: 4.708759         Validation Loss: 4.668263
Saving model ...
Epoch: 15         Training Loss: 4.678166         Validation Loss: 4.637478
Saving model ...
Epoch: 16         Training Loss: 4.640177         Validation Loss: 4.658593
Saving model ...
Epoch: 17         Training Loss: 4.608310         Validation Loss: 4.680147
Saving model ...
Epoch: 18         Training Loss: 4.584685         Validation Loss: 4.640967
Saving model ...
Epoch: 19         Training Loss: 4.564758         Validation Loss: 4.599651
Saving model ...
Epoch: 20         Training Loss: 4.500527         Validation Loss: 4.552146
Saving model ...
Epoch: 21         Training Loss: 4.500865         Validation Loss: 4.569289
```

```
Saving model ...
Epoch: 22         Training Loss: 4.461156         Validation Loss: 4.601456
Saving model ...
Epoch: 23         Training Loss: 4.402947         Validation Loss: 4.535856
Saving model ...
Epoch: 24         Training Loss: 4.417650         Validation Loss: 4.485701
Saving model ...
Epoch: 25         Training Loss: 4.380648         Validation Loss: 4.498749
Saving model ...
Epoch: 26         Training Loss: 4.377294         Validation Loss: 4.459432
Saving model ...
Epoch: 27         Training Loss: 4.325591         Validation Loss: 4.440990
Saving model ...
Epoch: 28         Training Loss: 4.295007         Validation Loss: 4.416767
Saving model ...
Epoch: 29         Training Loss: 4.245609         Validation Loss: 4.368398
Saving model ...
Epoch: 30         Training Loss: 4.196638         Validation Loss: 4.371539
Saving model ...
Epoch: 31         Training Loss: 4.149078         Validation Loss: 4.300864
Saving model ...
Epoch: 32         Training Loss: 4.123089         Validation Loss: 4.294948
Saving model ...
Epoch: 33         Training Loss: 4.093809         Validation Loss: 4.313311
Saving model ...
Epoch: 34         Training Loss: 4.021339         Validation Loss: 4.267560
Saving model ...
Epoch: 35         Training Loss: 3.997311         Validation Loss: 4.153843
Saving model ...
Epoch: 36         Training Loss: 3.931403         Validation Loss: 4.177800
Saving model ...
Epoch: 37         Training Loss: 3.877849         Validation Loss: 4.129204
Saving model ...
Epoch: 38         Training Loss: 3.797488         Validation Loss: 4.236309
Saving model ...
Epoch: 39         Training Loss: 3.740640         Validation Loss: 4.101164
Saving model ...
Epoch: 40         Training Loss: 3.605903         Validation Loss: 4.171843
Saving model ...
```

In [12]:
```
# load the model that got the best validation accuracy
#continuing model training for more epochs
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
model_scratch = train(20, train_loader2,valid_loader2, model_scratch, optimizer_scratch
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 3.556311         Validation Loss: 4.321260
Saving model ...
```

```
Epoch: 2         Training Loss: 3.455620         Validation Loss: 4.322265
Saving model ...
Epoch: 3         Training Loss: 3.609480         Validation Loss: 4.190783
Saving model ...
Epoch: 4         Training Loss: 3.545286         Validation Loss: 4.201549
Saving model ...
Epoch: 5         Training Loss: 3.568391         Validation Loss: 4.096832
Saving model ...
Epoch: 6         Training Loss: 3.458879         Validation Loss: 4.054095
Saving model ...
Epoch: 7         Training Loss: 3.483812         Validation Loss: 3.998593
Saving model ...
Epoch: 8         Training Loss: 3.368599         Validation Loss: 4.176605
Saving model ...
Epoch: 9         Training Loss: 3.388812         Validation Loss: 4.070485
Saving model ...
Epoch: 10         Training Loss: 3.132347         Validation Loss: 4.183546
Saving model ...
Epoch: 11         Training Loss: 3.231668         Validation Loss: 4.092609
Saving model ...
Epoch: 12         Training Loss: 0.000000         Validation Loss: 3.922336
Saving model ...
Epoch: 13         Training Loss: 3.194689         Validation Loss: 4.183305
Saving model ...
Epoch: 14         Training Loss: 3.041696         Validation Loss: 4.141242
Saving model ...
Epoch: 15         Training Loss: 2.982630         Validation Loss: 4.068111
Saving model ...
Epoch: 16         Training Loss: 2.858384         Validation Loss: 4.136570
Saving model ...
Epoch: 17         Training Loss: 2.881841         Validation Loss: 4.221618
Saving model ...
Epoch: 18         Training Loss: 2.736717         Validation Loss: 4.255808
Saving model ...
Epoch: 19         Training Loss: 2.612439         Validation Loss: 4.251598
Saving model ...
Epoch: 20         Training Loss: 2.343237         Validation Loss: 4.511215
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 [10]: def test(loaders, model, criterion, use_cuda):

             # monitor test loss and accuracy
```

```
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    try :
        for batch_idx, (data, target) in enumerate(loaders):
            # 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().nump
            total += data.size(0)
    except OSError:
        pass
    print('Test Loss: {:.6f}\n'.format(test_loss))

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

```
In [13]: # call test function
         test(test_loader2, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 4.316303


Test Accuracy: 12% (102/836)
```

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 [6]:
```
## TODO: Specify data loaders
import torch
import torchvision.models as models
from torchvision import datasets
from PIL import Image
import torchvision.transforms as transforms
train_dir = '/data/dog_images/train'
valid_dir = '/data/dog_images/valid'
test_dir = '/data/dog_images/test'
batch_size = 64

transform = transforms.Compose([transforms.Resize(250),
                                transforms.CenterCrop(224),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(15),
                                transforms.ToTensor(),
                                transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224

testtransform = transforms.Compose([transforms.Resize(250),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224

traindata = datasets.ImageFolder(train_dir,transform=transform)
validdata = datasets.ImageFolder(valid_dir,transform=transform)
testdata = datasets.ImageFolder(test_dir,transform=testtransform)

train_loader = torch.utils.data.DataLoader(traindata, batch_size=batch_size,shuffle= Tru
valid_loader = torch.utils.data.DataLoader(validdata, batch_size=batch_size,shuffle = Tr
test_loader = torch.utils.data.DataLoader(testdata, batch_size=batch_size)
```

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 [7]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

model_transfer = models.resnet50(pretrained = True)
for param in model_transfer.parameters():
    param.requires_grad = False

num_features = model_transfer.fc.in_features

model_transfer.fc = nn.Linear(num_features, 133)
```

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

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 95673444.40it/s]
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer:

Here i have used densenet121 pre-trained model for training the dog classifier. here the CNN layer weights are already fixed and only the classfier ie. dense layers weights are changed according to output required. The total output classes are 133 which are changed from actual 1000 classes for imagenet dataset.

The architecture is good for current problem because it helps to achieve high performance with such dataset.

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 [8]: import torch.optim as optim
        from torch import nn

        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.01, momentum =0.7)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [9]: # train the model

        n_epochs = 20
        model_transfer = train(n_epochs, train_loader,valid_loader, model_transfer, optimizer_tr

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

```
Epoch: 1         Training Loss: 4.404034         Validation Loss: 3.752018
Saving model ...
Epoch: 2         Training Loss: 3.785922         Validation Loss: 3.722273
Saving model ...
```

```
Epoch: 3         Training Loss: 3.718267         Validation Loss: 3.705332
Saving model ...
Epoch: 4         Training Loss: 3.533098         Validation Loss: 3.364309
Saving model ...
Epoch: 5         Training Loss: 2.921233         Validation Loss: 2.522898
Saving model ...
Epoch: 6         Training Loss: 2.341576         Validation Loss: 2.236548
Saving model ...
Epoch: 7         Training Loss: 2.085441         Validation Loss: 1.913002
Saving model ...
Epoch: 8         Training Loss: 1.704618         Validation Loss: 1.540742
Saving model ...
Epoch: 9         Training Loss: 1.429828         Validation Loss: 1.321867
Saving model ...
Epoch: 10         Training Loss: 1.264007         Validation Loss: 1.231520
Saving model ...
Epoch: 11         Training Loss: 1.135597         Validation Loss: 1.048040
Saving model ...
Epoch: 12         Training Loss: 0.967219         Validation Loss: 0.877013
Saving model ...
Epoch: 13         Training Loss: 0.861112         Validation Loss: 0.985093
Saving model ...
Epoch: 14         Training Loss: 0.838281         Validation Loss: 0.909854
Saving model ...
Epoch: 15         Training Loss: 0.813626         Validation Loss: 0.815347
Saving model ...
Epoch: 16         Training Loss: 0.752958         Validation Loss: 0.801201
Saving model ...
Epoch: 17         Training Loss: 0.722961         Validation Loss: 0.785450
Saving model ...
Epoch: 18         Training Loss: 0.654011         Validation Loss: 0.787167
Saving model ...
Epoch: 19         Training Loss: 0.673264         Validation Loss: 0.731974
Saving model ...
Epoch: 20         Training Loss: 0.656082         Validation Loss: 0.742425
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 [11]: test(test_loader, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.719785
```

```
Test Accuracy: 82% (691/836)
```

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.

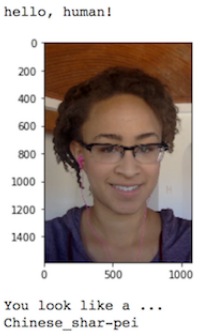
In [72]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# 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 traindata.class_to_idx ]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = Image.open(img_path).convert('RGB')
    transform = transforms.Compose([transforms.Resize(250),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406],[0.229,
    img = transform(img)[:3,:,:].unsqueeze(0)
    if use_cuda:
        img = img.cuda()

    output = model_transfer(img)
    _,pred = torch.max(output,1)
    pred = pred.cpu()
    return torch.tensor(pred,dtype=torch.int8)
```

In [68]:
```
class_names
```

Out[68]:
```
['Affenpinscher',
 'Afghan hound',
 'Airedale terrier',
 'Akita',
 'Alaskan malamute',
 'American eskimo dog',
 'American foxhound',
 'American staffordshire terrier',
 'American water spaniel',
 'Anatolian shepherd dog',
 'Australian cattle dog',
 'Australian shepherd',
 'Australian terrier',
 'Basenji',
 'Basset hound',
 'Beagle',
 'Bearded collie',
```

```
'Beauceron',
'Bedlington terrier',
'Belgian malinois',
'Belgian sheepdog',
'Belgian tervuren',
'Bernese mountain dog',
'Bichon frise',
'Black and tan coonhound',
'Black russian terrier',
'Bloodhound',
'Bluetick coonhound',
'Border collie',
'Border terrier',
'Borzoi',
'Boston terrier',
'Bouvier des flandres',
'Boxer',
'Boykin spaniel',
'Briard',
'Brittany',
'Brussels griffon',
'Bull terrier',
'Bulldog',
'Bullmastiff',
'Cairn terrier',
'Canaan dog',
'Cane corso',
'Cardigan welsh corgi',
'Cavalier king charles spaniel',
'Chesapeake bay retriever',
'Chihuahua',
'Chinese crested',
'Chinese shar-pei',
'Chow chow',
'Clumber spaniel',
'Cocker spaniel',
'Collie',
'Curly-coated retriever',
'Dachshund',
'Dalmatian',
'Dandie dinmont terrier',
'Doberman pinscher',
'Dogue de bordeaux',
'English cocker spaniel',
'English setter',
'English springer spaniel',
'English toy spaniel',
'Entlebucher mountain dog',
```

```
'Field spaniel',
'Finnish spitz',
'Flat-coated retriever',
'French bulldog',
'German pinscher',
'German shepherd dog',
'German shorthaired pointer',
'German wirehaired pointer',
'Giant schnauzer',
'Glen of imaal terrier',
'Golden retriever',
'Gordon setter',
'Great dane',
'Great pyrenees',
'Greater swiss mountain dog',
'Greyhound',
'Havanese',
'Ibizan hound',
'Icelandic sheepdog',
'Irish red and white setter',
'Irish setter',
'Irish terrier',
'Irish water spaniel',
'Irish wolfhound',
'Italian greyhound',
'Japanese chin',
'Keeshond',
'Kerry blue terrier',
'Komondor',
'Kuvasz',
'Labrador retriever',
'Lakeland terrier',
'Leonberger',
'Lhasa apso',
'Lowchen',
'Maltese',
'Manchester terrier',
'Mastiff',
'Miniature schnauzer',
'Neapolitan mastiff',
'Newfoundland',
'Norfolk terrier',
'Norwegian buhund',
'Norwegian elkhound',
'Norwegian lundehund',
'Norwich terrier',
'Nova scotia duck tolling retriever',
'Old english sheepdog',
```



Sample Human Output

```
'Otterhound',
'Papillon',
'Parson russell terrier',
'Pekingese',
'Pembroke welsh corgi',
'Petit basset griffon vendeen',
'Pharaoh hound',
'Plott',
'Pointer',
'Pomeranian',
'Poodle',
'Portuguese water dog',
'Saint bernard',
'Silky terrier',
'Smooth fox terrier',
'Tibetan mastiff',
'Welsh springer spaniel',
'Wirehaired pointing griffon',
'Xoloitzcuintli',
'Yorkshire terrier']
```

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

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             if face_detector(img_path):
                 print("HELLO HUMAN!")
             elif dog_detector(img_path):
                 print("HELLO DOG!")
             else :
                 print("YOU ARE NEITHER DOG NOR HUMAN! HELLO ALIEN")
                 return False
             breed = predict_breed_transfer(img_path)
             print("Your predicted breed class is ", class_names[breed])

             img = cv2.imread(img_path)
             RGB_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
             plt.imshow(RGB_img)
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement) 1. As we can see app can correctly classify many breeds of dog. It can clearly distinguish between human and dog. Improvement can be done increasing the dataset by adding more samples to each class as the number of classes is large compared to samples available for each class for dog dataset.

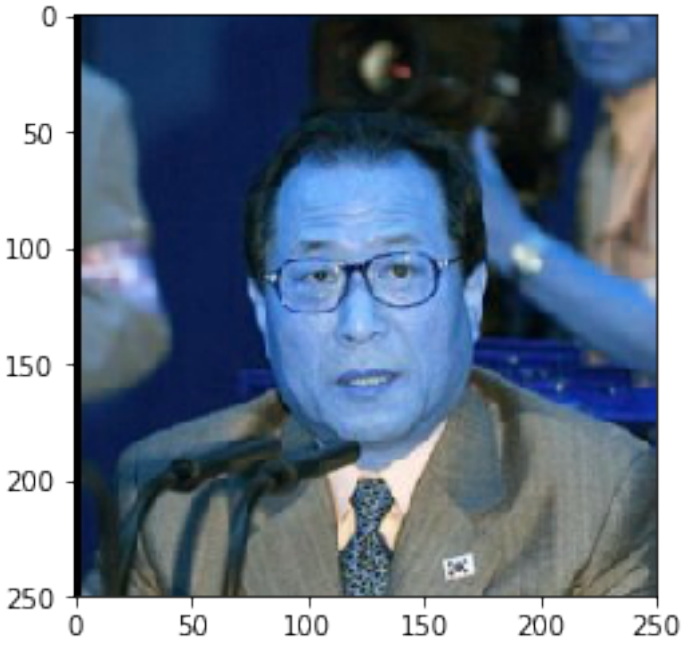
2. Hyperparameter tuning can also be optimised by studying more variations in learning rate, optimiser and loss criterion to make better predictions.

3. Image preprocessing using different opencv tools can be employed to reduce noise in data and imporve contrast and quality of images before feeding to the model.

```
In [80]: ## 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.
```

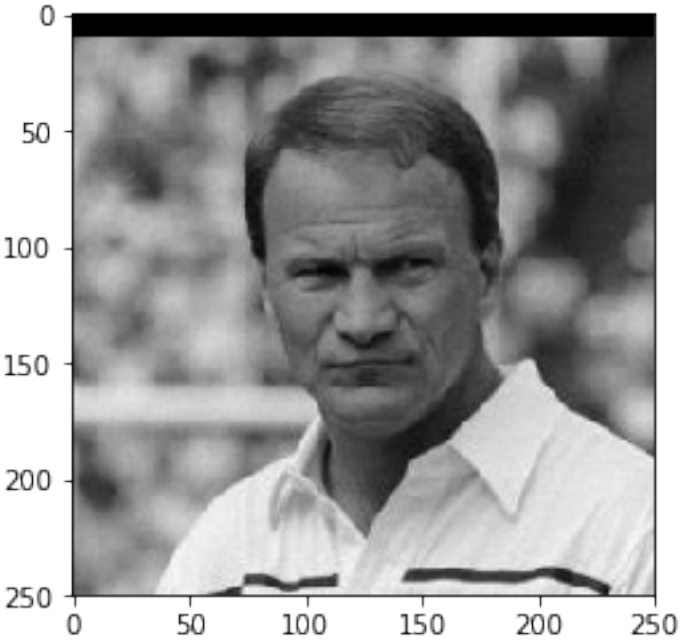
```
        ## suggested code, below
        for file in np.hstack((human_files[5:6])):
            run_app(file)
```

```
HELLO HUMAN!
Your predicted breed class is  Kerry blue terrier
```



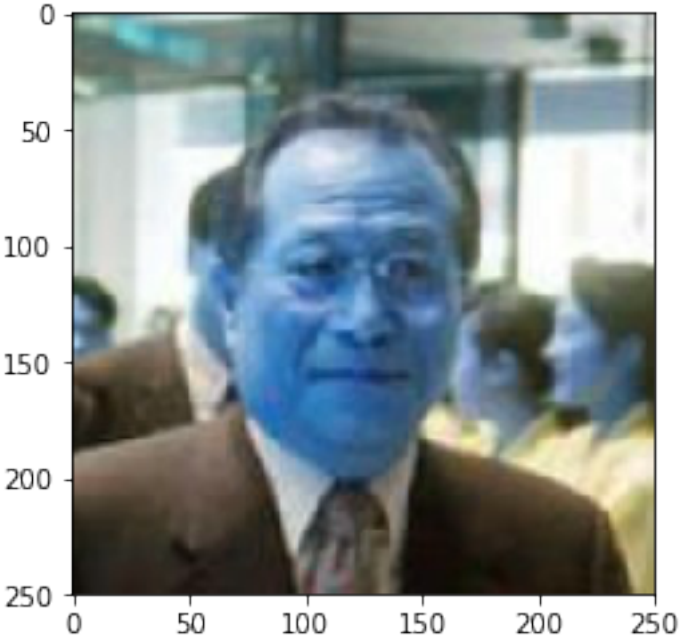

```
In [81]: for file in np.hstack((human_files[4:5])):
             run_app(file)
```

```
HELLO HUMAN!
Your predicted breed class is  Cane corso
```

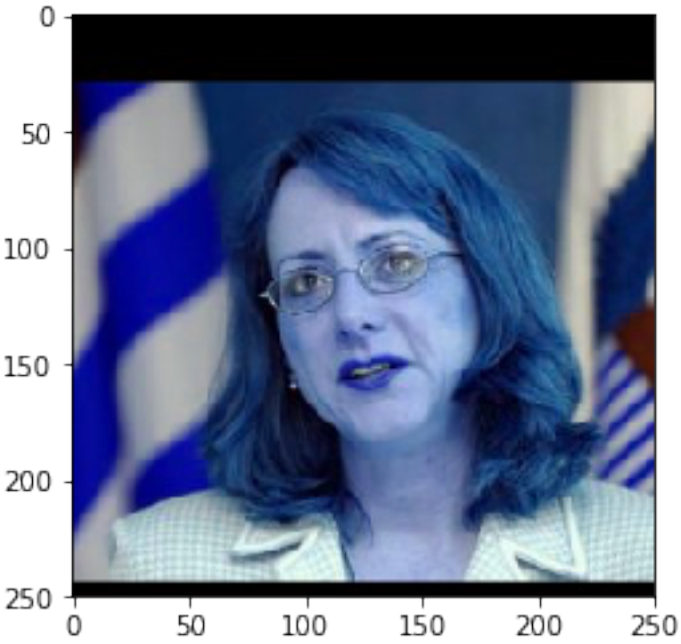



```
In [83]: for file in np.hstack((human_files[11:12])):
             run_app(file)
```

```
HELLO HUMAN!
Your predicted breed class is  Dachshund
```

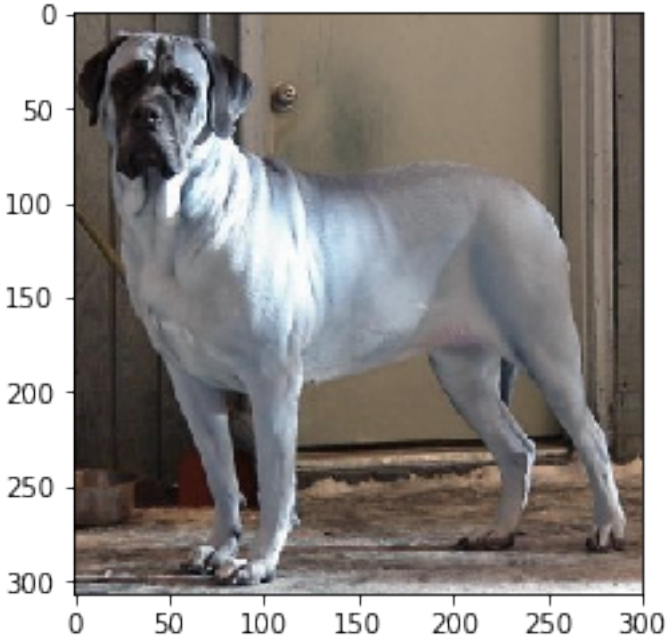


```
In [84]: for file in np.hstack((human_files[15:16])):
             run_app(file)

HELLO HUMAN!
Your predicted breed class is  Chinese crested
```

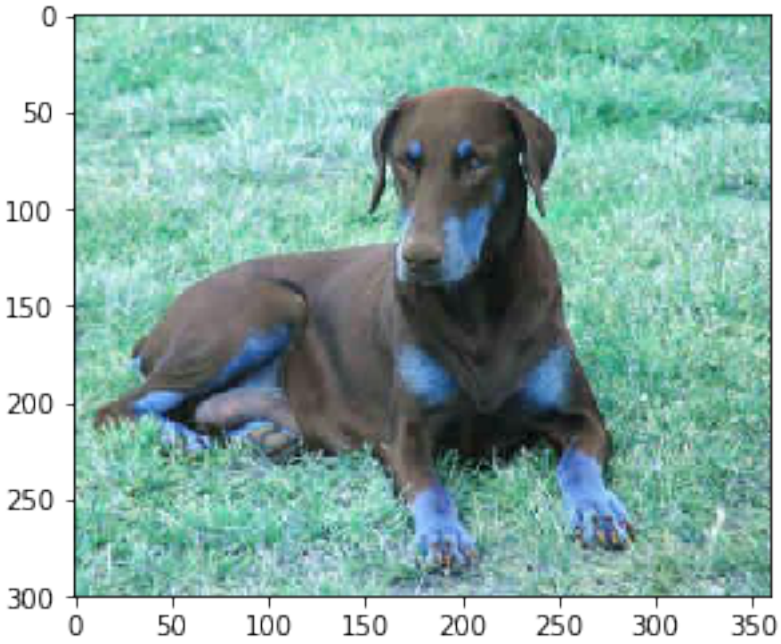



```
In [85]: for file in np.hstack((dog_files[5:6])):
             run_app(file)

HELLO DOG!
Your predicted breed class is  Mastiff
```

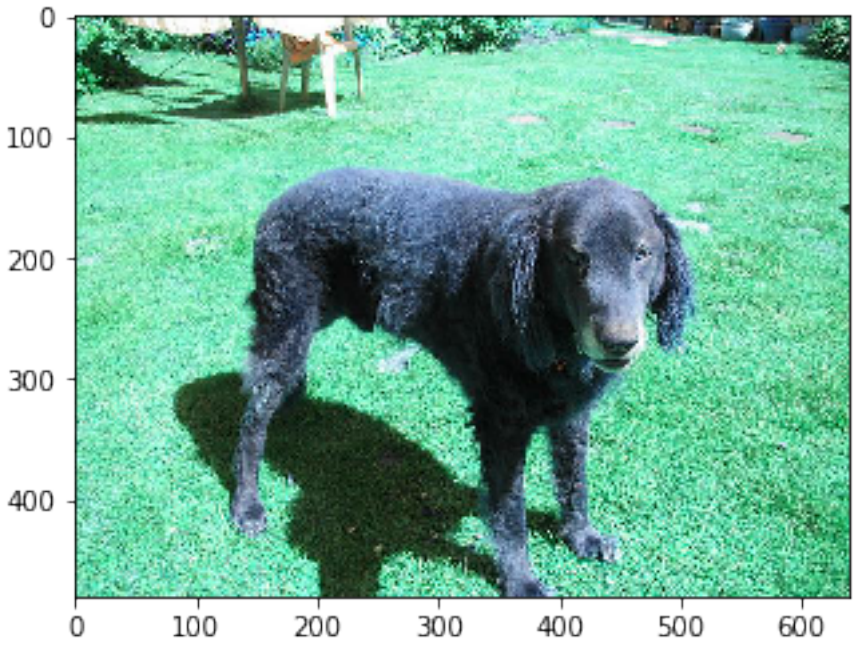

```
In [87]: for file in np.hstack((dog_files[1:2])):
             run_app(file)

HELLO DOG!
Your predicted breed class is  Mastiff
```



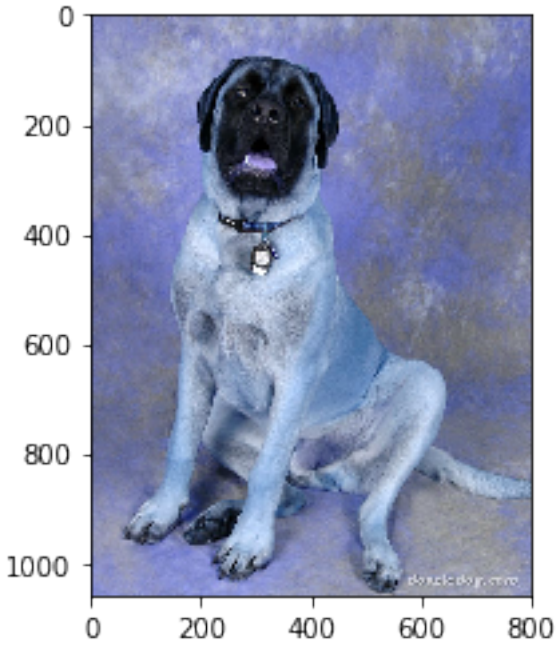
```
In [89]: for file in np.hstack((dog_files[102:103])):
             run_app(file)

HELLO DOG!
Your predicted breed class is  Doberman pinscher
```




```
In [90]: for file in np.hstack((dog_files[111:112])):
             run_app(file)

HELLO DOG!
Your predicted breed class is  Curly-coated retriever
```




```
In [91]: for file in np.hstack((dog_files[21:22])):
             run_app(file)

HELLO DOG!
Your predicted breed class is  Bullmastiff
```



```
In [ ]:
```