dog_app

April 1, 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: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- 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("lfw/*/*"))
        dog_files = np.array(glob("dogImages/*/*/*"))

        # 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.
```

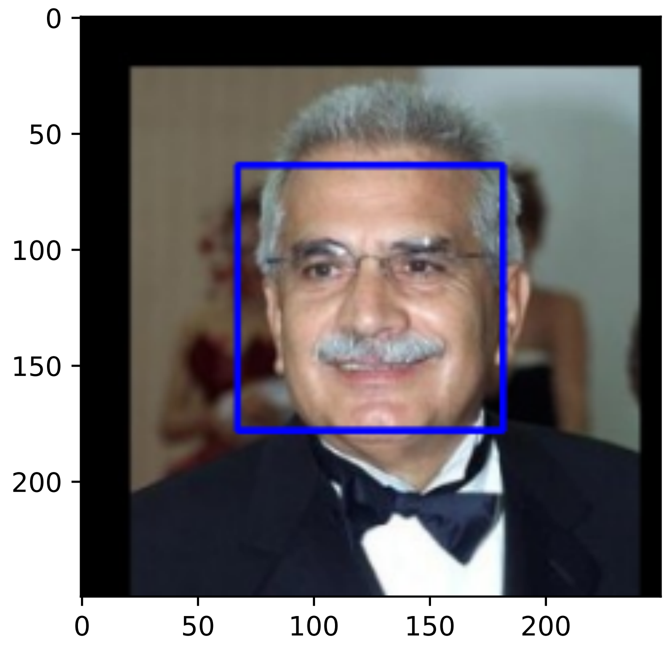
## Step 1: Detect Humans

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:
98% of faces have been detected in human_files
5% of dog images have been detected wrong as faces.

```
In [4]: from tqdm import tqdm

        human_files_short = human_files[:100]
        dog_files_short = dog_files[:100]

        #-#-# Do NOT modify the code above this line. #-#-#

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.

        human_faces_detected = 0
        human_faces_in_dogs = 0

        for image in human_files_short:
            if face_detector(image): human_faces_detected += 1

        for image in dog_files_short:
            if face_detector(image): human_faces_in_dogs += 1

        print(human_faces_detected, human_faces_in_dogs)
```

```
98 5
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [5]: ### (Optional)
        ### TODO: Test performance of another face detection algorithm.
        ### Feel free to use as many code cells as needed.
```

## Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

```
In [6]: import torch
        import torchvision.models as models

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

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

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

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

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

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

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

```
In [7]: from PIL import Image
        import torchvision.transforms as transforms

        # Set PIL to be tolerant of image files that are truncated.
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

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


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

            Args:
                img_path: path to an image

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

            ## TODO: Complete the function.
            ## Load and pre-process an image from the given img_path
            ## Return the *index* of the predicted class for that image


            image = Image.open(img_path)
            # plt.imshow(image)
            image = transformer(image).unsqueeze(0)
            image = image.cuda()

            prediction = VGG16(image)

            _, preds_tensor = torch.max(prediction, 1)

            predicted_class = preds_tensor.cpu().data.numpy().argmax()

            return preds_tensor.cpu().numpy()[0] # predicted class index
```

```
In [8]: VGG16_predict(dog_files_short[26])
```

```
Out[8]: 245
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [9]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            image_class = VGG16_predict(img_path)

            if  151 <= image_class <= 268:
                return True
            else:
                return False

            return None # 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:
0 % of images in human_files_short have been detected as dogs
99% of dogs have been detected in dog_files_short

```
In [28]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         dogs_in_humans_detected = 0
         dogs_detected = 0


         for image in human_files_short:
             if dog_detector(image): dogs_in_humans_detected += 1

         for image in dog_files_short:
             if dog_detector(image): dogs_detected += 1

         print(dogs_in_humans_detected, dogs_detected)
```

```
0 99
```

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 [11]: ### (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 `dogImages/train`, `dogImages/valid`, and `dogImages/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 [12]: import os
         from torchvision import datasets

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

```
batch_size =  50


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

augment_transformer = transforms.Compose([transforms.RandomResizedCrop(224),
                                          transforms.RandomRotation(15),
                                          # transforms.RandomPerspective(0.5),
                                          transforms.ToTensor(),
                                          transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                               std=[0.229, 0.224, 0.225])])

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


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

data_scratch = {'train'  : train_data,
                    'test'   : test_data,
                    'valid'  : valid_data}

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

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

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

Answer:

I resized all imges to 224 x 224 because thats the default size for many CNNs. Randomly resized croped the train images as well as augmented it by rotating it by -15 to 15 Degrees. For the test and validation set I only resized the images to 224.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [13]: import torch.nn as nn
         import torch.nn.functional as F

         number_of_classes = 133

         # 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(3,64, kernel_size=3, padding=1)
                 self.conv2 = nn.Conv2d(64,128, kernel_size=3, padding=1)
                 self.conv3 = nn.Conv2d(128,256, kernel_size=3, padding=1)
                 self.conv4 = nn.Conv2d(256,512, kernel_size=3, padding=1)

                 self.pool = nn.MaxPool2d(2, 2)
                 self.size_after_conv = (14 * 14 * 512)
                 self.fc1 = nn.Linear(self.size_after_conv, 512)
                 self.fc2 = nn.Linear(512, number_of_classes)

                 self.dropout = nn.Dropout(0.25)

             def forward(self, x):
                 ## Define forward behavior
                 x = self.pool(F.relu(self.conv1(x)))
                 x = self.pool(F.relu(self.conv2(x)))
                 x = self.pool(F.relu(self.conv3(x)))
                 x = self.pool(F.relu(self.conv4(x)))

                 x = x.view(-1, self.size_after_conv)
                 x = self.dropout(F.relu(self.fc1(x)))
                 x = self.fc2(x)

                 return x

         #-#-# You do 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: I used a similar CNN design like VGG, but for performance resons I used one less convolutional layer as well as one less Linear layer. The conv Layers have a kernel size of 3x3 and a padding of 1 and they increase the feature maps from layer to layer. The maxPool step after each conv layer decreses the size by 2 to a final size of 14x14 and 512 maps.

So the first fully connected Linear Layer has a input size of 100352 and output of 512. the second and final layer as a output size of 133, so the dog bread classes that need to be detected.

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 [14]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [15]: # the following import is required for training to be robust to truncated images
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
```

```
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

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

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
        valid_loss_min,
        valid_loss))
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.877331        Validation Loss: 4.781959
Validation loss decreased (inf --> 4.781959).  Saving model ...
Epoch: 2         Training Loss: 4.678349        Validation Loss: 4.498584
Validation loss decreased (4.781959 --> 4.498584).  Saving model ...
Epoch: 3         Training Loss: 4.518485        Validation Loss: 4.365222
Validation loss decreased (4.498584 --> 4.365222).  Saving model ...
Epoch: 4         Training Loss: 4.413215        Validation Loss: 4.362777
Validation loss decreased (4.365222 --> 4.362777).  Saving model ...
Epoch: 5         Training Loss: 4.329422        Validation Loss: 4.199742
Validation loss decreased (4.362777 --> 4.199742).  Saving model ...
Epoch: 6         Training Loss: 4.241622        Validation Loss: 4.161664
Validation loss decreased (4.199742 --> 4.161664).  Saving model ...
Epoch: 7         Training Loss: 4.157160        Validation Loss: 4.126622
Validation loss decreased (4.161664 --> 4.126622).  Saving model ...
Epoch: 8         Training Loss: 4.114581        Validation Loss: 4.127338
Epoch: 9         Training Loss: 4.051382        Validation Loss: 4.015965
Validation loss decreased (4.126622 --> 4.015965).  Saving model ...
Epoch: 10         Training Loss: 4.022501        Validation Loss: 3.992149
Validation loss decreased (4.015965 --> 3.992149).  Saving model ...
Epoch: 11         Training Loss: 3.961408        Validation Loss: 3.928814
Validation loss decreased (3.992149 --> 3.928814).  Saving model ...
Epoch: 12         Training Loss: 3.879439        Validation Loss: 3.947771
Epoch: 13         Training Loss: 3.847980        Validation Loss: 3.900843
Validation loss decreased (3.928814 --> 3.900843).  Saving model ...
Epoch: 14         Training Loss: 3.818995        Validation Loss: 3.800278
Validation loss decreased (3.900843 --> 3.800278).  Saving model ...
Epoch: 15         Training Loss: 3.751255        Validation Loss: 3.832453
Epoch: 16         Training Loss: 3.742434        Validation Loss: 3.750625
Validation loss decreased (3.800278 --> 3.750625).  Saving model ...
Epoch: 17         Training Loss: 3.691217        Validation Loss: 3.662983
Validation loss decreased (3.750625 --> 3.662983).  Saving model ...
Epoch: 18         Training Loss: 3.661873        Validation Loss: 3.752754
Epoch: 19         Training Loss: 3.606929        Validation Loss: 3.796502
Epoch: 20         Training Loss: 3.541917        Validation Loss: 3.689314
```

```
Out[15]: <All keys matched successfully>
```

```
In [16]: model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Out[16]: <All keys matched successfully>
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [17]: def test(loaders, model, criterion, use_cuda):
    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

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

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

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

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

```
Test Loss: 3.636119


Test Accuracy: 14% (125/836)
```

## Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [18]: ## TODO: Specify data loaders
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

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

         ## TODO: Specify model architecture
         vgg16 = models.vgg16(pretrained=True)
         for param in vgg16.features.parameters():
             param.requires_grad = False

         n_inputs = vgg16.classifier[6].in_features
         last_layer = nn.Linear(n_inputs, number_of_classes)

         vgg16.classifier[6] = last_layer

         model_transfer = vgg16


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

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

Answer:

I took a pretrained Vgg16 model and froze the parameters so Backprop will not change the weights. Then I replaced the last linear layer with a new linear layer for final classification. Input of the layer are the same as the original Layer, output length is the number of dog breads to identify.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [20]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [21]: # train the model
         loaders_transfer = loaders_scratch
```

```
        model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criter

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

```
Epoch: 1        Training Loss: 3.854121        Validation Loss: 1.738747
Validation loss decreased (inf --> 1.738747).  Saving model ...
Epoch: 2        Training Loss: 3.255345        Validation Loss: 1.599171
Validation loss decreased (1.738747 --> 1.599171).  Saving model ...
Epoch: 3        Training Loss: 3.109434        Validation Loss: 1.509523
Validation loss decreased (1.599171 --> 1.509523).  Saving model ...
Epoch: 4        Training Loss: 3.032688        Validation Loss: 1.370433
Validation loss decreased (1.509523 --> 1.370433).  Saving model ...
Epoch: 5        Training Loss: 2.921536        Validation Loss: 1.264932
Validation loss decreased (1.370433 --> 1.264932).  Saving model ...
Epoch: 6        Training Loss: 2.881172        Validation Loss: 1.268693
Epoch: 7        Training Loss: 2.846045        Validation Loss: 1.352395
Epoch: 8        Training Loss: 2.844887        Validation Loss: 1.256096
Validation loss decreased (1.264932 --> 1.256096).  Saving model ...
Epoch: 9        Training Loss: 2.853865        Validation Loss: 1.352045
Epoch: 10        Training Loss: 2.779168        Validation Loss: 1.207472
Validation loss decreased (1.256096 --> 1.207472).  Saving model ...
Epoch: 11        Training Loss: 2.813471        Validation Loss: 1.269254
Epoch: 12        Training Loss: 2.736855        Validation Loss: 1.130445
Validation loss decreased (1.207472 --> 1.130445).  Saving model ...
Epoch: 13        Training Loss: 2.729744        Validation Loss: 1.120103
Validation loss decreased (1.130445 --> 1.120103).  Saving model ...
Epoch: 14        Training Loss: 2.782887        Validation Loss: 1.201438
Epoch: 15        Training Loss: 2.805633        Validation Loss: 1.267789
Epoch: 16        Training Loss: 2.722189        Validation Loss: 1.103398
Validation loss decreased (1.120103 --> 1.103398).  Saving model ...
Epoch: 17        Training Loss: 2.791021        Validation Loss: 1.162205
Epoch: 18        Training Loss: 2.685872        Validation Loss: 1.170260
Epoch: 19        Training Loss: 2.683661        Validation Loss: 1.091353
Validation loss decreased (1.103398 --> 1.091353).  Saving model ...
Epoch: 20        Training Loss: 2.645587        Validation Loss: 1.117688
```

```
Out[21]: <All keys matched successfully>
```

```
In [22]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Out[22]: <All keys matched successfully>
```

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 [23]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 1.194254


Test Accuracy: 63% (533/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

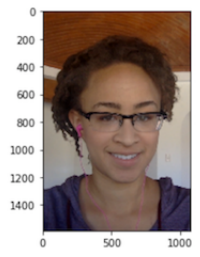
```
In [24]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

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


         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img = Image.open(img_path)

             image = transformer(img).unsqueeze_(0)
             if use_cuda:
                 image = image.cuda()

             model_transfer.eval()
             with torch.no_grad():
                 prediction = model_transfer(image)
                 prediction = torch.argmax(prediction).item()

             model_transfer.train()

             return class_names[prediction]

In [25]: for image in dog_files_short:
             breed = predict_breed_transfer(image)
             print(breed)

American staffordshire terrier
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
```

```
Mastiff
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Mastiff
Bulldog
Bulldog
Boxer
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Boxer
Bulldog
Bulldog
French bulldog
French bulldog
Bulldog
Bulldog
Bulldog
Mastiff
Bulldog
French bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Boxer
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Bulldog
Gordon setter
English cocker spaniel
English cocker spaniel
```

```
Welsh springer spaniel
English cocker spaniel
English cocker spaniel
Irish setter
English cocker spaniel
Welsh springer spaniel
English cocker spaniel
Boykin spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
Irish setter
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
Irish setter
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
Irish setter
English cocker spaniel
English cocker spaniel
Irish setter
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
English cocker spaniel
Irish setter
English cocker spaniel
Irish setter
English cocker spaniel
Welsh springer spaniel
English cocker spaniel
```

## Step 5: Write your Algorithm

```
hello, human!
```



```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

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 `dog_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 [26]:
```
### 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
    predicted_dog_breed = predict_breed_transfer(img_path)
    if face_detector(img_path):
        print('a human was detected that looks like a {}'.format(predicted_dog_breed))
    elif dog_detector(img_path):
        print('this is DOG, a {}'.format(predicted_dog_breed))
    else:
        print('Neither dog nor human please use different neural net')

    plt.imshow(Image.open(img_path))
    plt.show()
```

## Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement)

The output is about there where I expected it to be.

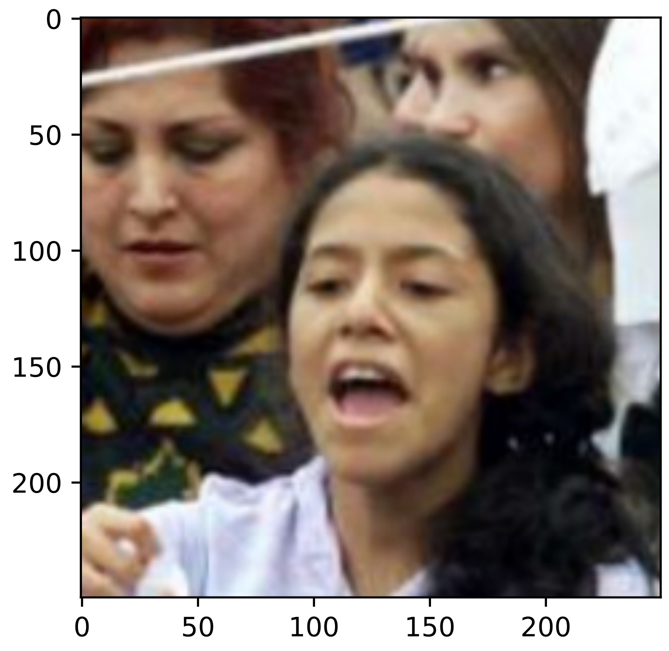
Things to improve could be longer training, probably with different learning rates.

Another thing would be to test another pretrained network like Inceptionv3 or ResNet.

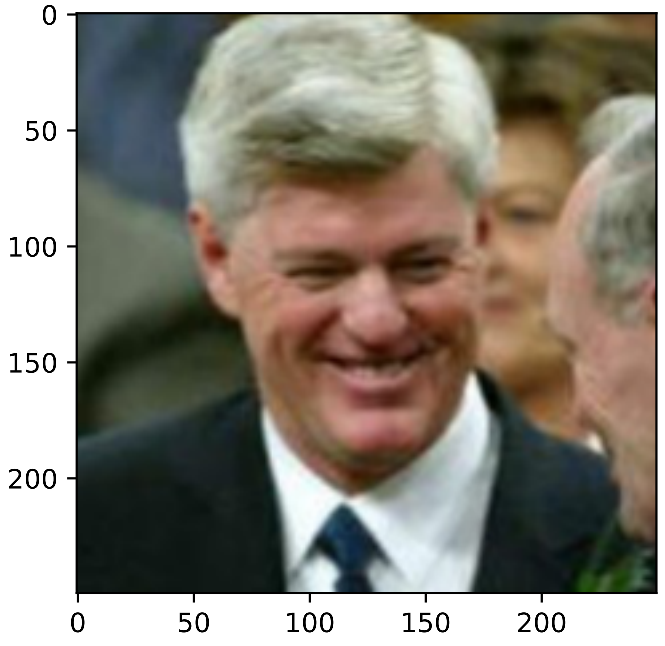
But maybe the first thing to do is to change the classification Layer. In my implementation I just replaced the last LinearLayer with a new one with the right number of classes as output. Maybe another design with different number of nodes will achieve better results.

```
In [29]: ## 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
         np.random.shuffle(human_files)
         np.random.shuffle(dog_files)

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

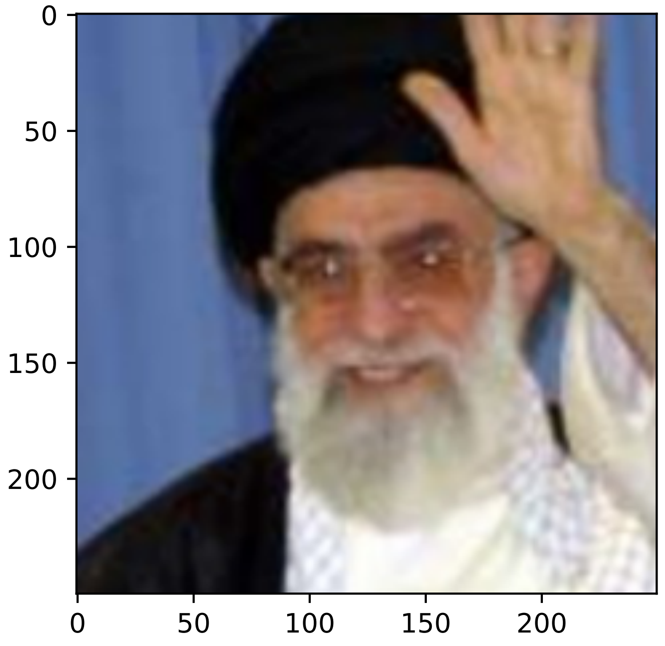
```
a human was detected that looks like a Dachshund
```



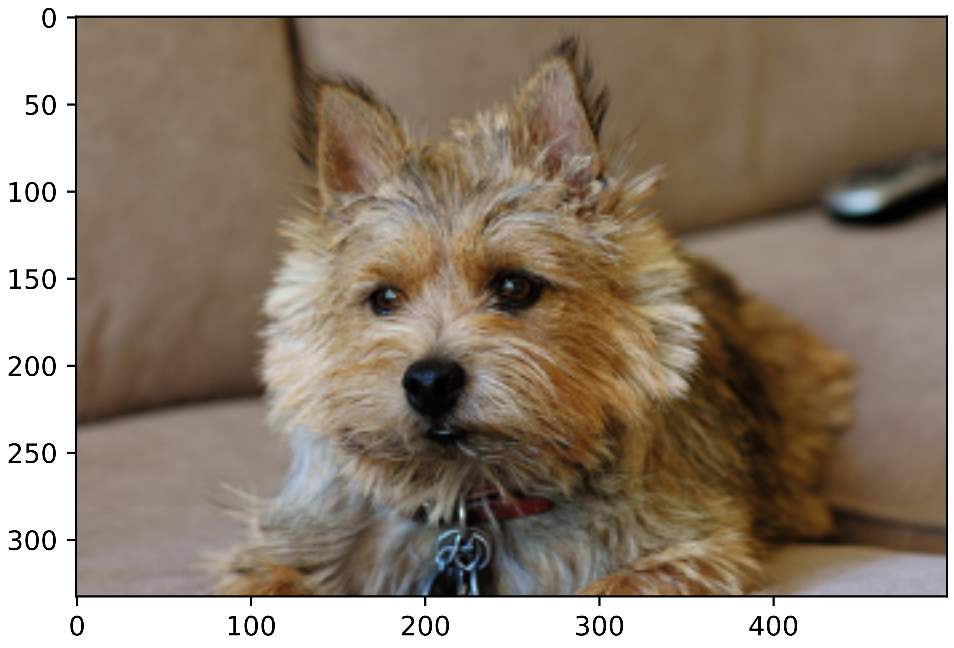
```
a human was detected that looks like a American staffordshire terrier
```



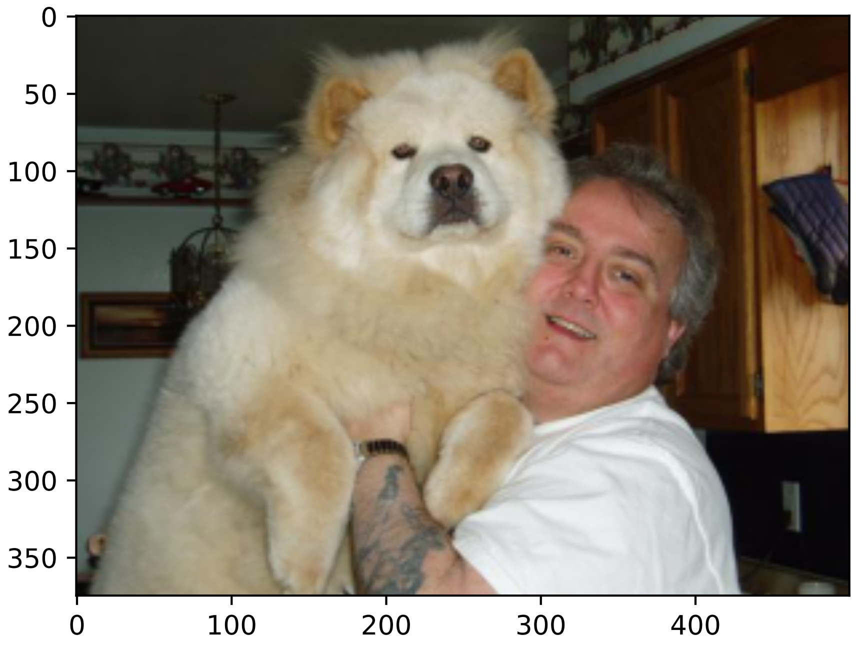
```
a human was detected that looks like a Dachshund
```



```
this is DOG, a Norfolk terrier
```



```
this is DOG, a Chow chow
```



```
this is DOG, a Entlebucher mountain dog
```