dog_app

June 16, 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

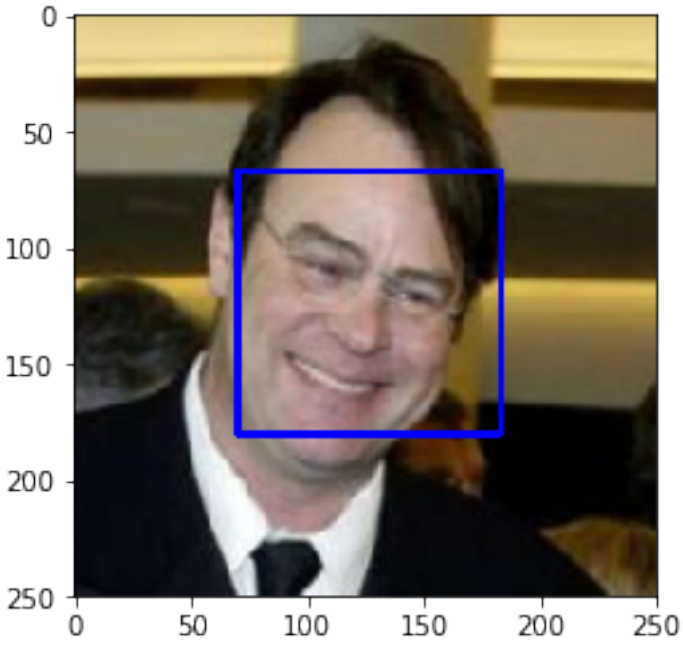
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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: (You can print out your results and/or write your percentages in this cell)

```
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 = 0
        dog_faces = 0
        for human_image in human_files_short:
            if face_detector(human_image):
                human_faces += 1

        for dog_image in dog_files_short:
            if face_detector(dog_image):
                dog_faces += 1

        print (f"Percentage of detected human faces in human_files: {human_faces}")
        print (f"Percentage of detected human faces in dog_files: {dog_faces}")
```

```
Percentage of detected human faces in human_files: 98
Percentage of detected human faces in dog_files: 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 [ ]: ### (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 [5]: 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, 98623457.40it/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 [6]: from PIL import Image
        import torchvision.transforms as transforms
        from torchvision import datasets, models, 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
            # VGG-16 Takes 224x224 images as input, so we resize all of them
            data_transform = transforms.Compose([transforms.Resize(256),transforms.CenterCrop(22
            image = Image.open(img_path)
            image = data_transform(image).float()
            image = image.unsqueeze(0)
            if use_cuda:
                image = image.cuda()

            ## Return the *index* of the predicted class for that image
            output = VGG16(image)
            _, preds_tensor = torch.max(output, 1)
            preds = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso

            return preds # 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 [7]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index = VGG16_predict(img_path)
```

```
    if (index > 150) & (index < 269):
        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:
0% detected dog in human_files_short
92% detected dogs in dog_files_short

In [8]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_faces = 0
dog_faces = 0

for human_image in human_files_short:
    if dog_detector(human_image):
        human_faces += 1

for dog_image in dog_files_short:
    if dog_detector(dog_image):
        dog_faces += 1

print (f"Percentage of detected dogs in human_files: {human_faces}")
print (f"Percentage of detected dogs in dog_files: {dog_faces}")
```

```
Percentage of detected dogs in human_files: 0
Percentage of detected dogs in dog_files: 92
```

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 [ ]:
```
### (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 [9]: import os
        from torchvision import datasets
        from PIL import Image, ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True
        import torchvision.transforms as transforms
        from torchvision import datasets, models, transforms
        import torch
        import torchvision.models as models

        # check if CUDA is available
```

```
use_cuda = torch.cuda.is_available()

### TODO: Write data loaders for training, validation, and test sets
data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

## Specify appropriate transforms, and batch_sizes
data_transform = transforms.Compose([transforms.Resize(80),transforms.CenterCrop(64), tr
                                     transforms.RandomRotation(20, resample=Image.BILINE
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.
test_transform = transforms.Compose([transforms.Resize(80),transforms.CenterCrop(64), tr
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.
train_data = datasets.ImageFolder(train_dir, transform=data_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=test_transform)
test_data = datasets.ImageFolder(test_dir, transform=test_transform)

# print out some data stats
print('Num training images: ', len(train_data))
print('Num valid images: ', len(valid_data))
print('Num test images: ', len(test_data))

batch_size = 20
num_workers=0

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=False)

test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=False)

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

```
Num training images:  6680
Num valid images:  835
Num test images:  836
```

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:

First, I resized the images to 80 and then applied a center crop in order to obtain a better image assuming that the object is in the center. I used a 64x64x3 image as an input tensor to help speed up the classification process.

I decide augment the dataset using random horizontal flip and random rotation, this helped my CNN better classify the images.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [10]: import torch.nn as nn
         import torch.nn.functional as F
         import numpy as np

         classes = len(train_data.classes)

         # 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
                 # convolutional layer (sees 64x64x3 image tensor)
                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                 # convolutional layer (sees 32x32x16 tensor)
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                 # convolutional layer (sees 16x16x32 tensor)
                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)
                 # linear layer (64 * 8 * 8 -> 500)
                 self.fc1 = nn.Linear(64 * 8 * 8, 500)
                 # linear layer (500 -> 133)
                 self.fc2 = nn.Linear(500, classes)
                 # dropout layer (p=0.4)
                 self.dropout = nn.Dropout(0.4)

             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)))
                 # flatten image input
                 x = x.view(-1, 64 * 8 * 8)
                 # add dropout layer
                 x = self.dropout(x)
                 # add 1st hidden layer, with relu activation function
```

```
            x = F.relu(self.fc1(x))
            # add dropout layer
            x = self.dropout(x)
            # add 2nd hidden layer, with relu activation function
            x = self.fc2(x)

            return x

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

    # instantiate the CNN
    model_scratch = Net()

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

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

Answer:

First, I searched the state of the art in order to see how they solved the problem. Then, I took one of the CNN created in past lessons and start to train and saw the results. After that I fined tune the parameters using the advices that gave us in previous lessons and in the mentor help. I chaged the sizes of the architecture to adapt it to my size image. Also, I changed the dropout because using 0.25 as dropout lead to overfitting, and for this reason I increased it to 0.4. Finally, I use an Adam optimazer because it had the best behaviour for my CNN in order to reach a test accuracy greater than 10%.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [11]: 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 [12]: import numpy as np

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


    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['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 += loss.item() * data.size(0)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch,
        train_loss,
        valid_loss / (len(loaders['valid']) * loaders['valid'].batch_size)
        ))

    ## 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
```

In [13]:
```
# train the model
model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1         Training Loss: 4.791976         Validation Loss: 4.492617
Validation loss decreased (inf --> 3773.798134).  Saving model ...
Epoch: 2         Training Loss: 4.428594         Validation Loss: 4.266817
Validation loss decreased (3773.798134 --> 3584.126623).  Saving model ...
Epoch: 3         Training Loss: 4.232344         Validation Loss: 4.148394
Validation loss decreased (3584.126623 --> 3484.650562).  Saving model ...
Epoch: 4         Training Loss: 4.102221         Validation Loss: 4.065092
Validation loss decreased (3484.650562 --> 3414.677372).  Saving model ...
Epoch: 5         Training Loss: 3.985348         Validation Loss: 4.099727
Epoch: 6         Training Loss: 3.892124         Validation Loss: 3.926777
Validation loss decreased (3414.677372 --> 3298.492987).  Saving model ...
Epoch: 7         Training Loss: 3.807601         Validation Loss: 3.966232
Epoch: 8         Training Loss: 3.705197         Validation Loss: 3.911214
Validation loss decreased (3298.492987 --> 3285.419478).  Saving model ...
Epoch: 9         Training Loss: 3.650980         Validation Loss: 3.842532
Validation loss decreased (3285.419478 --> 3227.726686).  Saving model ...
Epoch: 10         Training Loss: 3.571817         Validation Loss: 3.836846
Validation loss decreased (3227.726686 --> 3222.950361).  Saving model ...
Epoch: 11         Training Loss: 3.486632         Validation Loss: 3.859051
Epoch: 12         Training Loss: 3.422288         Validation Loss: 3.827978
Validation loss decreased (3222.950361 --> 3215.501893).  Saving model ...
Epoch: 13         Training Loss: 3.389552         Validation Loss: 3.766969
Validation loss decreased (3215.501893 --> 3164.253867).  Saving model ...
Epoch: 14         Training Loss: 3.283254         Validation Loss: 3.792568
Epoch: 15         Training Loss: 3.246365         Validation Loss: 3.733484
Validation loss decreased (3164.253867 --> 3136.126776).  Saving model ...
Epoch: 16         Training Loss: 3.178700         Validation Loss: 3.785893
Epoch: 17         Training Loss: 3.155690         Validation Loss: 3.801850
Epoch: 18         Training Loss: 3.074292         Validation Loss: 3.793414
Epoch: 19         Training Loss: 3.040052         Validation Loss: 3.851699
Epoch: 20         Training Loss: 2.982417         Validation Loss: 3.762050
Epoch: 21         Training Loss: 2.917672         Validation Loss: 3.827458
```

```
Epoch: 22         Training Loss: 2.871650         Validation Loss: 3.903660
Epoch: 23         Training Loss: 2.842509         Validation Loss: 3.744475
Epoch: 24         Training Loss: 2.781149         Validation Loss: 3.899990
Epoch: 25         Training Loss: 2.750448         Validation Loss: 3.902382
Epoch: 26         Training Loss: 2.722361         Validation Loss: 3.842859
Epoch: 27         Training Loss: 2.676223         Validation Loss: 3.874791
Epoch: 28         Training Loss: 2.652164         Validation Loss: 3.942229
Epoch: 29         Training Loss: 2.594910         Validation Loss: 3.861647
Epoch: 30         Training Loss: 2.555207         Validation Loss: 4.007850
```

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 [14]:
```
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))
```

In [15]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.724851


Test Accuracy: 15% (127/836)
```

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 [26]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets
from PIL import Image, ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import torchvision.transforms as transforms
from torchvision import datasets, models, transforms
import torch
import torchvision.models as models

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

### TODO: Write data loaders for training, validation, and test sets
data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

## Specify appropriate transforms, and batch_sizes
data_transform = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.RandomRotation(20, resample=Image.BILIN
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.485, 0.456, 0.406),(0.229,
test_transform = transforms.Compose([transforms.Resize(256),transforms.CenterCrop(224),
                                     transforms.Normalize((0.485, 0.456, 0.406),(0.229,
train_data = datasets.ImageFolder(train_dir, transform=data_transform)
```

```
        valid_data = datasets.ImageFolder(valid_dir, transform=test_transform)
        test_data = datasets.ImageFolder(test_dir, transform=test_transform)

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

        # print out some data stats
        print('Num training images: ', len(train_data))
        print('Num valid images: ', len(valid_data))
        print('Num test images: ', len(test_data))

        batch_size = 20
        num_workers=0

        # prepare data loaders
        train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)

        valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=False)

        test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                  num_workers=num_workers, shuffle=False)

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

```
Num training images:  6680
Num valid images:  835
Num test images:  836
```

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

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

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

```
classes = len(train_data.classes)
n_inputs = model_transfer.classifier[6].in_features
last_layer = nn.Linear(n_inputs, classes)

model_transfer.classifier[6] = last_layer

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:

First, I searched for the state of the art and how they solve the problem; I found some interesting articles and I started to try some of the proposed solutions. I started using Inception V3, then VGG 16 and VGG 19 but I chose VGG 16 because gave me better results compared to Inception and it took less time in training than VGG 19. Finally, I used the Adam and SGD optimizers and chose the SGD because gave me better accuracy. Also I tried several learning rates until I found a good one.

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 [22]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()

         optimizer_transfer = optim.SGD(model_transfer.classifier.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 [27]: # train the model
         model_transfer = train(30, 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: 1.759079         Validation Loss: 0.963104
Validation loss decreased (inf --> 809.007212).  Saving model ...
Epoch: 2         Training Loss: 1.103476         Validation Loss: 0.679120
Validation loss decreased (809.007212 --> 570.460452).  Saving model ...
Epoch: 3         Training Loss: 0.871027         Validation Loss: 0.572723
Validation loss decreased (570.460452 --> 481.086939).  Saving model ...
Epoch: 4         Training Loss: 0.749651         Validation Loss: 0.505812
Validation loss decreased (481.086939 --> 424.881910).  Saving model ...
```

```
Epoch: 5         Training Loss: 0.667468         Validation Loss: 0.478986
Validation loss decreased (424.881910 --> 402.347857).  Saving model ...
Epoch: 6         Training Loss: 0.606993         Validation Loss: 0.451820
Validation loss decreased (402.347857 --> 379.529004).  Saving model ...
Epoch: 7         Training Loss: 0.562885         Validation Loss: 0.431370
Validation loss decreased (379.529004 --> 362.350829).  Saving model ...
Epoch: 8         Training Loss: 0.521892         Validation Loss: 0.419612
Validation loss decreased (362.350829 --> 352.474057).  Saving model ...
Epoch: 9         Training Loss: 0.509750         Validation Loss: 0.411053
Validation loss decreased (352.474057 --> 345.284480).  Saving model ...
Epoch: 10         Training Loss: 0.468550         Validation Loss: 0.404172
Validation loss decreased (345.284480 --> 339.504837).  Saving model ...
Epoch: 11         Training Loss: 0.456559         Validation Loss: 0.384618
Validation loss decreased (339.504837 --> 323.078971).  Saving model ...
Epoch: 12         Training Loss: 0.428383         Validation Loss: 0.382744
Validation loss decreased (323.078971 --> 321.504557).  Saving model ...
Epoch: 13         Training Loss: 0.401197         Validation Loss: 0.386708
Epoch: 14         Training Loss: 0.395327         Validation Loss: 0.373147
Validation loss decreased (321.504557 --> 313.443309).  Saving model ...
Epoch: 15         Training Loss: 0.371768         Validation Loss: 0.374158
Epoch: 16         Training Loss: 0.363710         Validation Loss: 0.372894
Validation loss decreased (313.443309 --> 313.230920).  Saving model ...
Epoch: 17         Training Loss: 0.359995         Validation Loss: 0.360743
Validation loss decreased (313.230920 --> 303.024019).  Saving model ...
Epoch: 18         Training Loss: 0.340795         Validation Loss: 0.369961
Epoch: 19         Training Loss: 0.331257         Validation Loss: 0.364752
Epoch: 20         Training Loss: 0.312738         Validation Loss: 0.359874
Validation loss decreased (303.024019 --> 302.294111).  Saving model ...
Epoch: 21         Training Loss: 0.311819         Validation Loss: 0.363275
Epoch: 22         Training Loss: 0.308615         Validation Loss: 0.365275
Epoch: 23         Training Loss: 0.283849         Validation Loss: 0.364687
Epoch: 24         Training Loss: 0.296332         Validation Loss: 0.361010
Epoch: 25         Training Loss: 0.285077         Validation Loss: 0.357727
Validation loss decreased (302.294111 --> 300.490753).  Saving model ...
Epoch: 26         Training Loss: 0.272077         Validation Loss: 0.365402
Epoch: 27         Training Loss: 0.256202         Validation Loss: 0.358226
Epoch: 28         Training Loss: 0.263867         Validation Loss: 0.346841
Validation loss decreased (300.490753 --> 291.346631).  Saving model ...
Epoch: 29         Training Loss: 0.250766         Validation Loss: 0.350075
Epoch: 30         Training Loss: 0.243701         Validation Loss: 0.341956
Validation loss decreased (291.346631 --> 287.242659).  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 [28]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.407096


Test Accuracy: 87% (730/836)
```

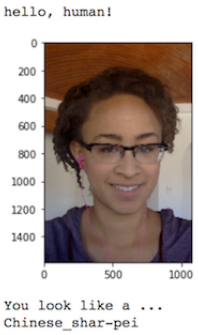
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [29]: ### 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_transfer['train'].classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             ## Load and pre-process an image from the given img_path
             data_transform = transforms.Compose([transforms.Resize(256),transforms.CenterCrop(2
                                                  transforms.Normalize((0.485, 0.456, 0.406),(0.
             if "http" in img_path:
                 response = requests.get(img_path)
                 image = Image.open(BytesIO(response.content)).convert('RGB')
             else:
                 image = Image.open(img_path)

             #image = Image.open(img_path)
             image = data_transform(image).float()
             image = image.unsqueeze(0)
             if use_cuda:
                 image = image.cuda()
             ## Return the *class name* of the predicted class for that image
             model_transfer.eval()
             output = model_transfer(image)
             _, preds_tensor = torch.max(output, 1)
             preds = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tens

             return class_names[preds]
```

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



Sample Human Output

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 [30]:
```
import matplotlib.pyplot as plt
%matplotlib inline

def print_image(img_path):
    image = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()
```

In [31]:
```
### 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
    print("------------------------------------------------------------")
    #Dog detected
    if dog_detector(img_path):
        print("You are DOG")
        print_image(img_path)
        print("Your breed is ...")
        print(predict_breed_transfer(img_path))
    #Human detected
    elif face_detector(img_path):
        print("You are HUMAN")
        print_image(img_path)
        print("Your inner puppy is ...")
```

```
        print(predict_breed_transfer(img_path))
    #Neither human nor dog
    else:
        print ("ERROR: You are neither dog nor human")
    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.

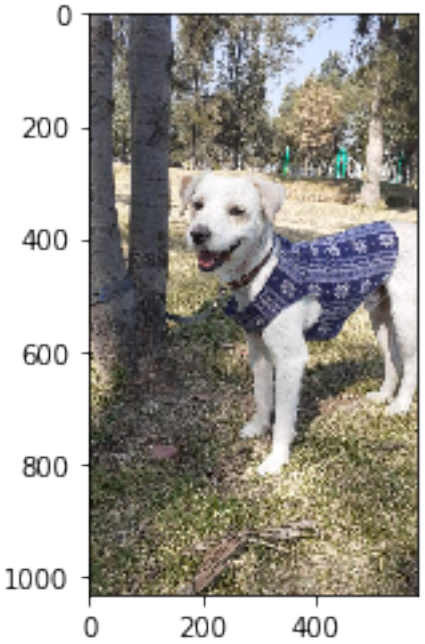
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

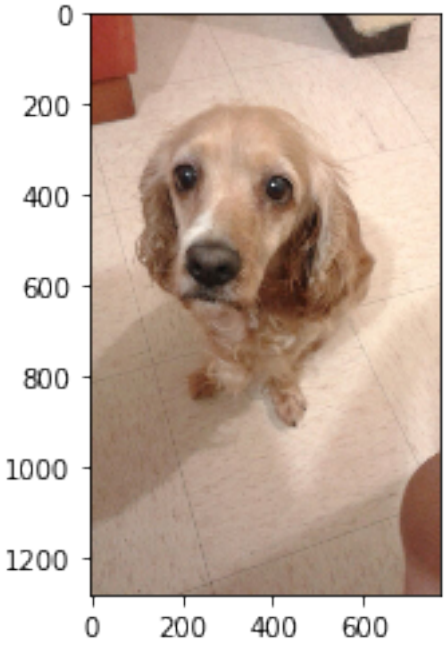
1. I think I can improve the speed of the CNN's training using another architecture
2. I can try another transforms for data augmentation
3. I can try another optimizer

In [34]:
```
## 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.


import numpy as np
from glob import glob

# load filenames for human and dog images
my_files = np.array(glob("/home/workspace/dog_project/MyImages/*"))

# print number of images in each dataset
print('There are %d total images.' % len(my_files))

## suggested code, below
for file in np.hstack((my_files)):
    run_app(file)


#run_app("images/cocker.jpg")
#run_app("images/woman.jpg")
#run_app("images/cat.jpg")
```

```
There are 7 total images.
---------------------------------------------------------
You are DOG
```

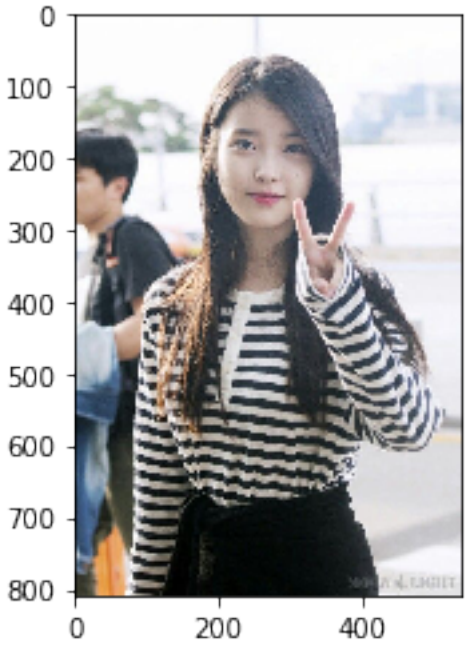



```
Your breed is ...
Parson russell terrier

---------------------------------------------------------
You are DOG
```

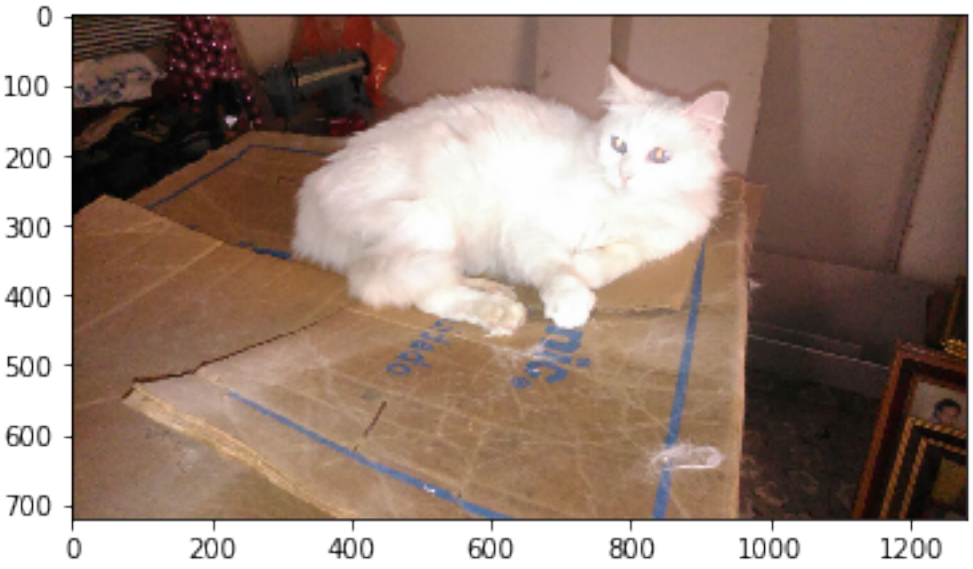



```
Your breed is ...
Cocker spaniel

-------------------------------------------------------
You are HUMAN
```

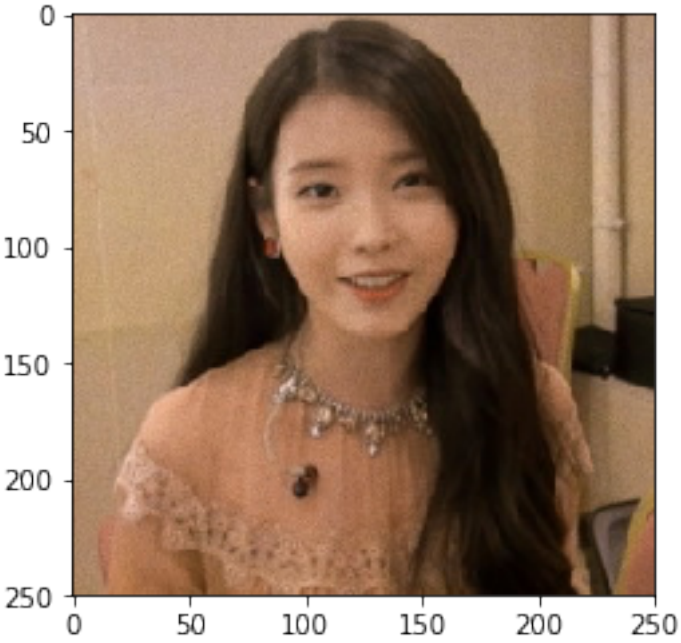
```
Your inner puppy is ...
Welsh springer spaniel

-------------------------------------------------------
You are HUMAN
```



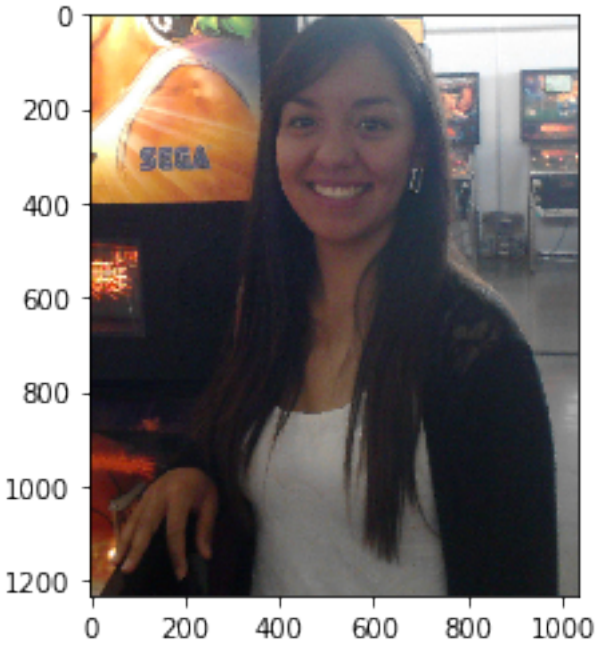

```
Your inner puppy is ...
Dalmatian

-------------------------------------------------------
You are DOG
```




```
Your breed is ...
American eskimo dog

-------------------------------------------------------------
You are HUMAN
```



```
Your inner puppy is ...
American water spaniel

---------------------------------------------------------
You are HUMAN
```




```
Your inner puppy is ...
Labrador retriever


In [ ]:
```