# dog_app

February 17, 2019

# 1 Convolutional Neural Networks

## 1.1 Project: Write an Algorithm for a Dog Identification App

---

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.
## Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/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

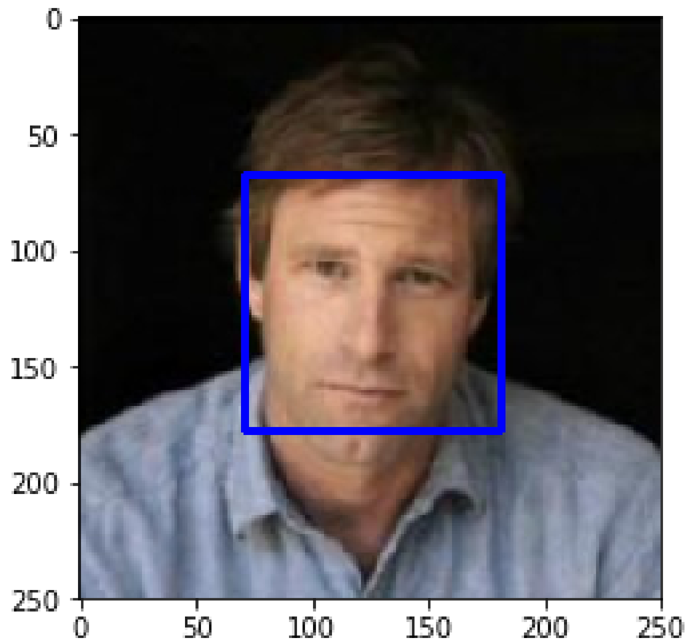
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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:**

```
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_files_face_detections = np.sum([face_detector(i) for i in human_files_short])
        dog_files_face_detections = np.sum([face_detector(i) for i in dog_files_short])

        print("The percentage of human faces detected in the human images: {}%".format(human_fil
        print("The percentage of human faces detected in the dog images: {}%".format(dog_files_f
```

```
The percentage of human faces detected in the human images: 96%
The percentage of human faces detected in the dog images: 18%
```

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

def preprocess_image(img_path):

    # open and convert to RGB
    image = Image.open(img_path).convert('RGB')

    # transform
    image_transforms = transforms.Compose([
        transforms.Resize(224),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
```

```
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225])])

        # remove alpha channel
        image = image_transforms(image)[:3,:,:].unsqueeze(0)

        # move to GPU if available
        if use_cuda:
            image = image.cuda()

        return image


    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

        # pre-process the image
        image = preprocess_image(img_path)

        # get the network output
        output = VGG16(image.cuda())

        return output.data.cpu().numpy().argmax() # 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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.

            detection = VGG16_predict(img_path)

            return detection >= 151 and detection <= 268
```

#### 1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.

        human_files_dog_detections = np.sum([dog_detector(i) for i in human_files_short])
        dog_files_dog_detections = np.sum([dog_detector(i) for i in dog_files_short])

        print("The percentage of dogs detected in the human images: {}%".format(human_files_dog_
        print("The percentage of dogs detected in the dog images: {}%".format(dog_files_dog_dete

The percentage of dogs detected in the human images: 0%
The percentage of dogs detected in the dog images: 94%
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [10]: ### (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 [11]: import sys
         !{sys.executable} -m pip install Augmentor
```

```
Requirement already satisfied: Augmentor in c:\users\stempbar\appdata\local\continuum\anaconda3\
Requirement already satisfied: numpy>=1.11.0 in c:\users\stempbar\appdata\local\continuum\anacon
Requirement already satisfied: Pillow>=4.0.0 in c:\users\stempbar\appdata\local\continuum\anacon
Requirement already satisfied: future>=0.16.0 in c:\users\stempbar\appdata\local\continuum\anaco
Requirement already satisfied: tqdm>=4.9.0 in c:\users\stempbar\appdata\local\continuum\anaconda
```

```
In [12]: import os
         from torchvision import datasets
         import Augmentor

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

```
# augmentor
p = Augmentor.Pipeline()
p.resize(probability=1.0, width=500, height=500, resample_filter='BICUBIC')
p.rotate(probability=0.75, max_left_rotation=25, max_right_rotation=25)
p.flip_left_right(probability=0.5)
p.random_distortion(probability=0.5, grid_width=8, grid_height=8, magnitude=4)
p.random_brightness(probability=0.75, min_factor=0.8, max_factor=1.2)
p.random_contrast(probability=0.75, min_factor=0.8, max_factor=1.2)
p.zoom(probability=0.75, min_factor=1.0, max_factor=1.1)

data_transforms = {
    'train': transforms.Compose([p.torch_transform(),
                                 transforms.RandomResizedCrop(224, scale=(0.1, 1.0)),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485, 0.456, 0.406],
                                                      [0.229, 0.224, 0.225])]),
    'valid': transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485, 0.456, 0.406],
                                                      [0.229, 0.224, 0.225])]),
    'test': transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485, 0.456, 0.406],
                                                      [0.229, 0.224, 0.225])])
}

# Load the datasets with ImageFolder
data_dir = 'dogImages'

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'valid', 'test']}

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 32

# prepare data loaders for the training, test and validation datasets
loaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                             shuffle=True, num_workers=num_workers)
              for x in ['train', 'valid', 'test']}

# print data sizes for each dataset
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid', 'test']}
```

```
        print ("Dasaset Size: "+ str(dataset_sizes) + "\n")
```

```
Dasaset Size: {'train': 6680, 'valid': 835, 'test': 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**:

I decided to use Augmentor, an image augmentation library in Python for machine learning. In my pipeline, I am resizing, rotating, right/left flipping, distorting, brightening, changing contrast and zooming in/out images at different probability rates. I apply these operations to the training dataset in hope to increase model accuracy. I do not apply them to test and valid datasets. I crop the images to 224 x 224 pixels. Then, I transform to tensors, which gives me 3 x 224 x 224 tensors. As the last step, I normalize tensors.

#### 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

         # 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)
                 self.conv2 = nn.Conv2d(16, 32, 3)
                 self.conv3 = nn.Conv2d(32, 64, 3)
                 self.conv4 = nn.Conv2d(64, 128, 3)
                 self.conv5 = nn.Conv2d(128, 256, 3)

                 # batch normalize
                 self.bn1 = nn.BatchNorm2d(16)
                 self.bn2 = nn.BatchNorm2d(32)
                 self.bn3 = nn.BatchNorm2d(64)
                 self.bn4 = nn.BatchNorm2d(128)
                 self.bn5 = nn.BatchNorm2d(256)

                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)

                 # linear layer
```

```
        self.fc1 = nn.Linear(256 * 5 * 5, 500)
        self.fc2 = nn.Linear(500, 133)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(self.bn1(F.relu(self.conv1(x))))
        x = self.pool(self.bn2(F.relu(self.conv2(x))))
        x = self.pool(self.bn3(F.relu(self.conv3(x))))
        x = self.pool(self.bn4(F.relu(self.conv4(x))))
        x = self.pool(self.bn5(F.relu(self.conv5(x))))

        # flatten image input
        x = x.view(-1, 5 * 5 * 256)

        # 1st hidden layer
        x = F.relu(self.fc1(x))

        # 2nd hidden layer
        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 decided to have 5 convolutional layers with kernel size 3, each followed by batch normalization layer and the maxpooling layer of kernel size 2 and stride 2 (which reduces the size by 50%). Then, I have two linear layers.

I decided to not use dropout, and use batch normalization layers, after I read this article on Towards Data Science and this discussion on Reddit.

#### 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=1e-5)
```

### 1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [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

                     # zero the parameter gradients
                     optimizer.zero_grad()

                     # forward pass
                     output = model(data)

                     # batch loss
                     loss = criterion(output, target)

                     # backward pass
                     loss.backward()

                     # parameter update
```

```
                optimizer.step()

                # update training loss
                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

                # forward pass
                output = model(data)

                # batch loss
                loss = criterion(output, target)

                # update validation loss
                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
                torch.save(model_scratch.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model
```

```
In [17]: # train the model
         model_scratch = train(20, loaders, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1        Training Loss: 4.111157         Validation Loss: 4.099396
Validation loss decreased (inf --> 4.099396).  Saving model ...
Epoch: 2        Training Loss: 4.084867         Validation Loss: 4.008404
Validation loss decreased (4.099396 --> 4.008404).  Saving model ...
```

```
Epoch: 3         Training Loss: 4.061570         Validation Loss: 3.978844
Validation loss decreased (4.008404 --> 3.978844).  Saving model ...
Epoch: 4         Training Loss: 4.037389         Validation Loss: 3.960296
Validation loss decreased (3.978844 --> 3.960296).  Saving model ...
Epoch: 5         Training Loss: 4.023890         Validation Loss: 3.964105
Epoch: 6         Training Loss: 4.004258         Validation Loss: 3.939120
Validation loss decreased (3.960296 --> 3.939120).  Saving model ...
Epoch: 7         Training Loss: 3.990747         Validation Loss: 3.899761
Validation loss decreased (3.939120 --> 3.899761).  Saving model ...
Epoch: 8         Training Loss: 3.954503         Validation Loss: 3.878145
Validation loss decreased (3.899761 --> 3.878145).  Saving model ...
Epoch: 9         Training Loss: 3.964483         Validation Loss: 3.907765
Epoch: 10         Training Loss: 3.945692         Validation Loss: 3.853640
Validation loss decreased (3.878145 --> 3.853640).  Saving model ...
Epoch: 11         Training Loss: 3.942500         Validation Loss: 3.856295
Epoch: 12         Training Loss: 3.927703         Validation Loss: 3.856947
Epoch: 13         Training Loss: 3.889638         Validation Loss: 3.857355
Epoch: 14         Training Loss: 3.887756         Validation Loss: 3.894609
Epoch: 15         Training Loss: 3.870874         Validation Loss: 3.784280
Validation loss decreased (3.853640 --> 3.784280).  Saving model ...
Epoch: 16         Training Loss: 3.856426         Validation Loss: 3.751899
Validation loss decreased (3.784280 --> 3.751899).  Saving model ...
Epoch: 17         Training Loss: 3.847791         Validation Loss: 3.831577
Epoch: 18         Training Loss: 3.818767         Validation Loss: 3.753702
Epoch: 19         Training Loss: 3.817476         Validation Loss: 3.742204
Validation loss decreased (3.751899 --> 3.742204).  Saving model ...
Epoch: 20         Training Loss: 3.806627         Validation Loss: 3.783620
```

In [16]:
```python
# load the model that got the best validation accuracy
# model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

#### 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 [18]:
```python
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, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.702221


Test Accuracy: 15% (130/836)
```

---

## 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 [19]: ## 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 [20]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture

         # load a pre-trained network
         model_transfer = models.resnet152(pretrained=True)

         # Freeze training for all "features" layers
         for _, param in model_transfer.named_parameters():
             param.requires_grad = False

         # Build a feed-forward network
         num_features = model_transfer.fc.in_features

         model_transfer.fc = nn.Sequential(nn.BatchNorm1d(num_features),
                                  nn.Linear(num_features,133))

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

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

**Answer:** I decided to use resnet152 pretrained model, at is has many layers and therefore will be more accurate for photos which might be only slightly different. I load the pretrained model, freeze parameters, and then I add batch normalization layer to increase model accuracy and performance. After that, I have one linear later to receive predictions for 133 categories.

#### 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 [21]: ### TODO: select loss function
         criterion_transfer = nn.CrossEntropyLoss()

         ### TODO: select optimizer
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=1e-5)
```

#### 1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [22]: # train the model
         model_transfer = train(20, loaders, model_transfer, optimizer_transfer,
                                criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1         Training Loss: 4.831743        Validation Loss: 4.645917
Validation loss decreased (inf --> 4.645917).  Saving model ...
Epoch: 2         Training Loss: 4.615231        Validation Loss: 4.387256
Validation loss decreased (4.645917 --> 4.387256).  Saving model ...
Epoch: 3         Training Loss: 4.409658        Validation Loss: 4.137614
Validation loss decreased (4.387256 --> 4.137614).  Saving model ...
Epoch: 4         Training Loss: 4.213600        Validation Loss: 3.907495
Validation loss decreased (4.137614 --> 3.907495).  Saving model ...
Epoch: 5         Training Loss: 4.026468        Validation Loss: 3.655552
Validation loss decreased (3.907495 --> 3.655552).  Saving model ...
Epoch: 6         Training Loss: 3.842345        Validation Loss: 3.451309
Validation loss decreased (3.655552 --> 3.451309).  Saving model ...
Epoch: 7         Training Loss: 3.668385        Validation Loss: 3.247590
Validation loss decreased (3.451309 --> 3.247590).  Saving model ...
Epoch: 8         Training Loss: 3.501289        Validation Loss: 3.011819
Validation loss decreased (3.247590 --> 3.011819).  Saving model ...
Epoch: 9         Training Loss: 3.345718        Validation Loss: 2.847074
Validation loss decreased (3.011819 --> 2.847074).  Saving model ...
Epoch: 10         Training Loss: 3.210324        Validation Loss: 2.661612
Validation loss decreased (2.847074 --> 2.661612).  Saving model ...
Epoch: 11         Training Loss: 3.066343        Validation Loss: 2.516430
Validation loss decreased (2.661612 --> 2.516430).  Saving model ...
Epoch: 12         Training Loss: 2.941160        Validation Loss: 2.338651
Validation loss decreased (2.516430 --> 2.338651).  Saving model ...
Epoch: 13         Training Loss: 2.810794        Validation Loss: 2.199334
Validation loss decreased (2.338651 --> 2.199334).  Saving model ...
Epoch: 14         Training Loss: 2.688991        Validation Loss: 2.073131
Validation loss decreased (2.199334 --> 2.073131).  Saving model ...
Epoch: 15         Training Loss: 2.597221        Validation Loss: 1.955527
Validation loss decreased (2.073131 --> 1.955527).  Saving model ...
Epoch: 16         Training Loss: 2.493003        Validation Loss: 1.849462
Validation loss decreased (1.955527 --> 1.849462).  Saving model ...
Epoch: 17         Training Loss: 2.395825        Validation Loss: 1.731358
Validation loss decreased (1.849462 --> 1.731358).  Saving model ...
Epoch: 18         Training Loss: 2.328654        Validation Loss: 1.647751
Validation loss decreased (1.731358 --> 1.647751).  Saving model ...
Epoch: 19         Training Loss: 2.223225        Validation Loss: 1.554958
Validation loss decreased (1.647751 --> 1.554958).  Saving model ...
Epoch: 20         Training Loss: 2.152429        Validation Loss: 1.501510
Validation loss decreased (1.554958 --> 1.501510).  Saving model ...
```

In [39]:
```
# load the model that got the best validation accuracy (uncomment the line below)
# model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

### 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, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.453572


Test Accuracy: 79% (667/836)
```

### 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 image_datasets['train'].classes]

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

                 image = preprocess_image(img_path)

                 output = model_transfer(image.cuda())

                 return class_names[output.data.cpu().numpy().argmax()]

         print(predict_breed_transfer('dogImages/train/069.French_bulldog/French_bulldog_04789.j
```

```
French bulldog
```

---

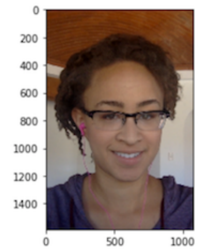
## Step 5: Write your Algorithm

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 `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!

```
hello, human!
```



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

Sample Human Output

#### 1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [30]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):

             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()

             ## handle cases for a human face, dog, and neither
             if dog_detector(img_path) or face_detector(img_path):
                 print("You look like a ...", predict_breed_transfer(img_path))
             else:
                 print("This does not look like a dog or human!")
```

---

## 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!

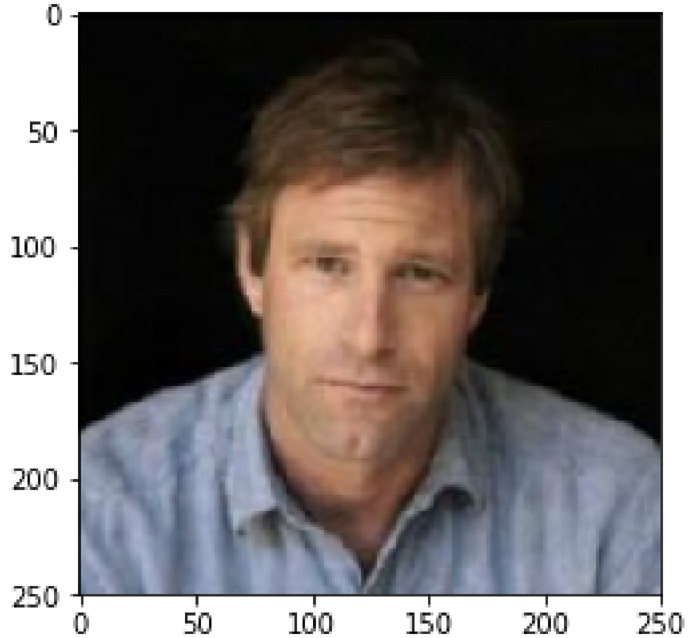
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

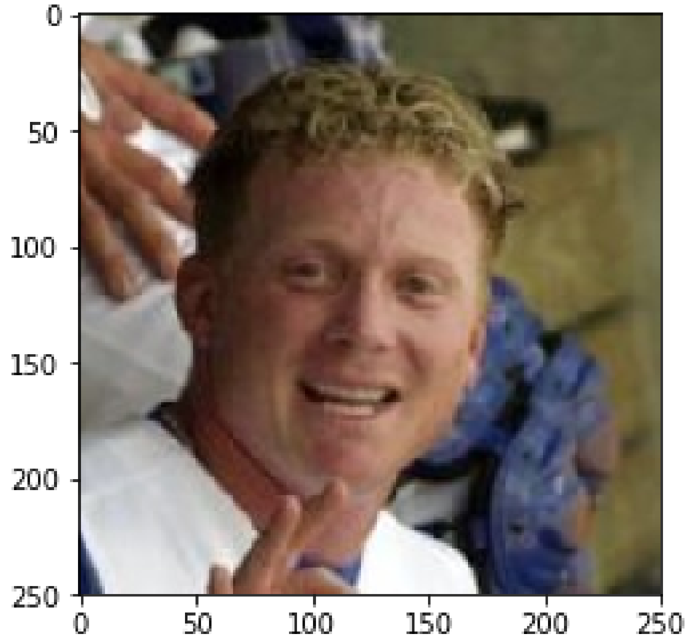
**Answer:** I expected a bit better output, however, I did not train my model extensively. I only run 20 epochs, which is a low number. Accuracy would be gladly improved by more extensive training.

```
In [33]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
```

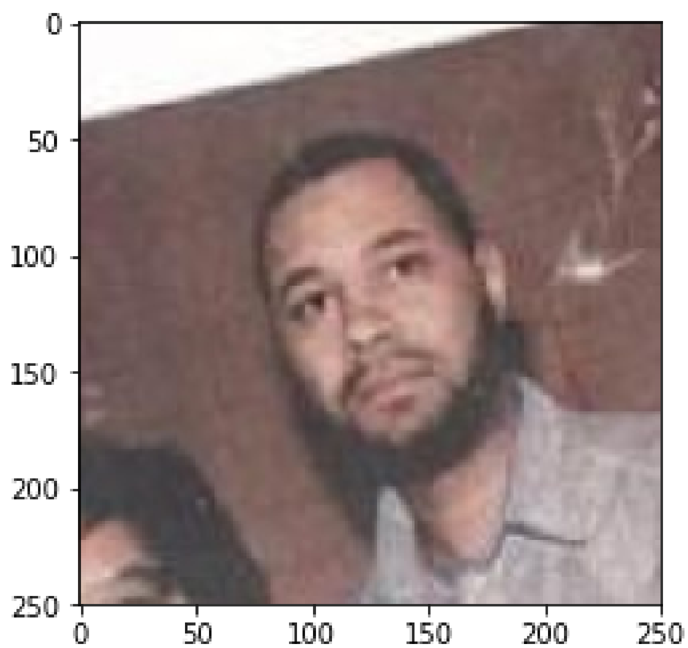
```python
## Feel free to use as many code cells as needed.

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



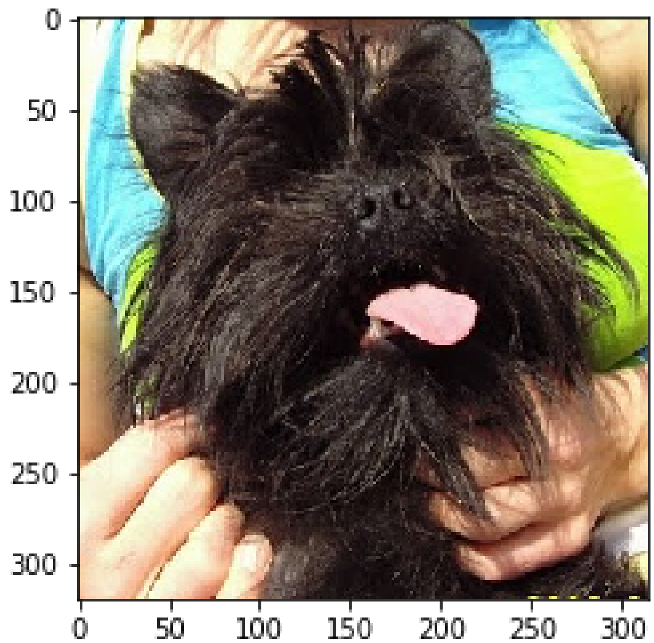

```
You look like a ... Bearded collie
```



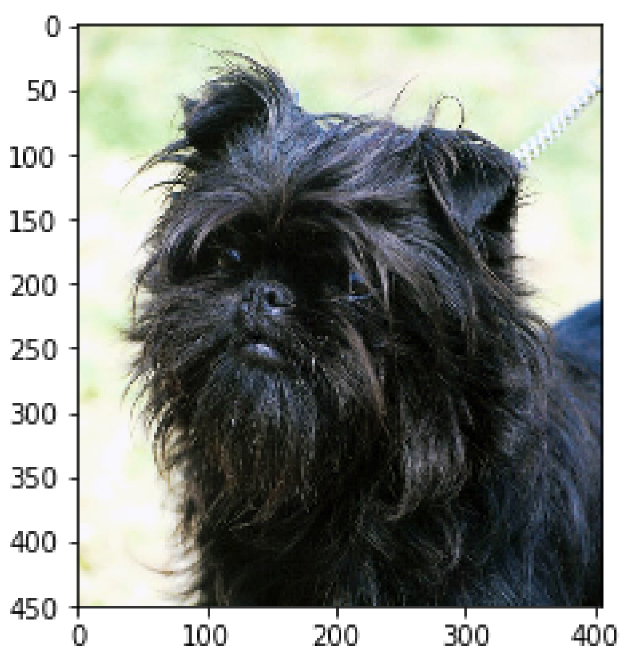

You look like a ... Bull terrier



This does not look like a dog or human!




You look like a ... Affenpinscher



```
You look like a ... Affenpinscher
```


```
You look like a ... Affenpinscher
```

```
In [ ]:
```