dog_app

May 6, 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
```

```
In [2]: # 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

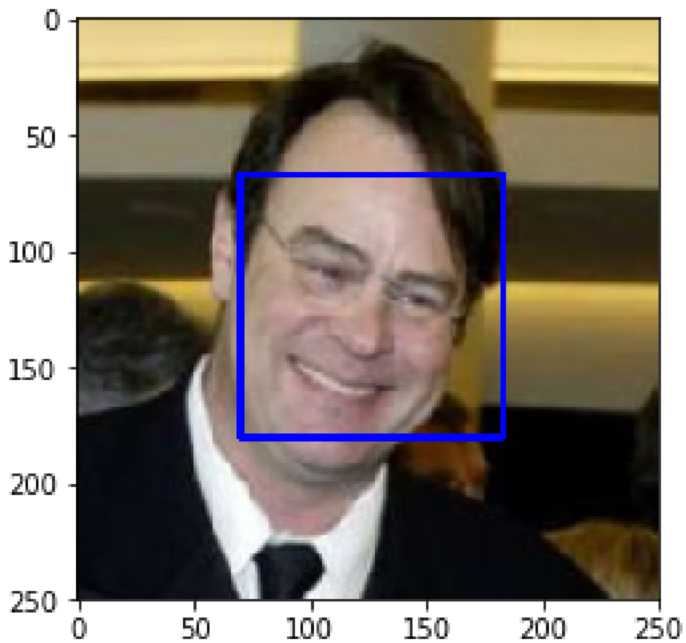
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [3]: 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))
```

```python
# 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 [4]: # 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:

- Human Detected: 0.98%
- Others Detected: 0.83%

```
In [5]: 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.
        detected_humans = 0
        for human_file in human_files_short:
            if(face_detector(human_file)):
                detected_humans += 1

        detected_other = 0
        for dog_file in dog_files_short:
            if(face_detector(dog_file) == False):
                detected_other += 1

        print("Human Detected: {}%".format(detected_humans/len(human_files_short)))
        print("Others Detected: {}%".format(detected_other/len(dog_files_short)))
```

```
Human Detected: 0.98%
Others Detected: 0.83%
```

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 [23]:
```
### (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 [6]:
```
import torch
import torchvision.models as models
```

In [7]:
```
# 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 [00:09<00:00, 59798980.31it/s]
```

In [8]:
```
import urllib
import pickle as pickle
vgg16_classes = pickle.load(urllib.request.urlopen('https://gist.githubusercontent.com/y
```

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.

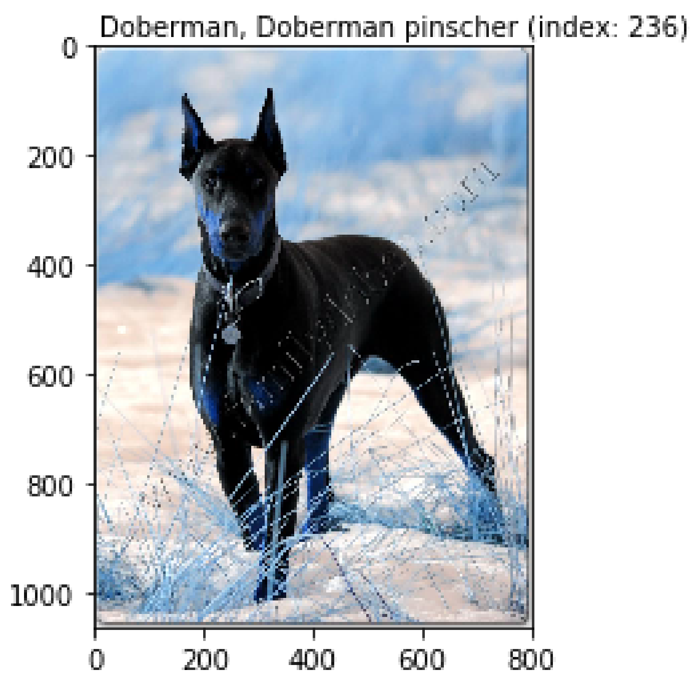
Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

In [62]:
```
from PIL import Image
import torchvision.transforms as T
import torch.nn  as nn

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
    transform = T.Compose([
        T.Resize(256),
        T.CenterCrop(224),
        T.ToTensor(),
        T.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
    ])
    sigmoid = nn.Sigmoid()

    #Applying the transformation
    image = transform(Image.open(img_path).convert('RGB'))
    if use_cuda:
        image = image.cuda()

    #Inserting batch dimension at the begibbing
    image.unsqueeze_(0)
    output = VGG16(image)
    _,pred = torch.max(sigmoid(output), 1)

    return pred.item() # predicted class index
```

In [78]:
```
# Detecting object
```

```
file_name = dog_files[87]
print('Detecting objects from: {}'.format(file_name))
output = VGG16_predict(file_name)

# Displaying the result
img = cv2.imread(file_name)
plt.imshow(img)
plt.text(10, -20, "{} (index: {})".format(vgg16_classes[output], output))
plt.show()
```

```
Detecting objects from: /data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher_04170.jpg
```



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 [37]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
```

```
        ## TODO: Complete the function.
        prediction = VGG16_predict(img_path)

        return prediction >= 151 and prediction <= 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:

- Dogs detected from human photos: 0.01
- Dogs detected from dog photos: 0.98

```
In [38]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         from_human = 0
         for human in human_files_short:
             if(dog_detector(human)):
                 from_human += 1
         print("Dogs detected from human photos: {}".format(from_human/len(human_files_short)))

         from_dog = 0
         for dog in dog_files_short:
             if(dog_detector(dog)):
                 from_dog += 1
         print("Dogs detected from dog photos: {}".format(from_dog/len(dog_files_short)))
```

```
Dogs detected from human photos: 0.0
Dogs detected from dog photos: 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 [ ]: ### (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	Black 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 [39]:
```
import os
import torch
from torchvision import datasets
import torchvision.transforms as T

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

batch_size = 30
num_workers = 0
# normalize = T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
resize = T.Resize(224)
```

```
center_crop = T.CenterCrop(224)

trainning_transform = T.Compose([
        T.RandomRotation(45),
        T.RandomHorizontalFlip(),
        resize,
        center_crop,
        T.ToTensor()
])

validation_transform = T.Compose([
        T.RandomHorizontalFlip(),
        resize,
        center_crop,
        T.ToTensor(),
])

test_transform = T.Compose([
        resize,
        center_crop,
        T.ToTensor(),
])

training_dogs = datasets.ImageFolder("/data/dog_images/train", transform=trainning_tran
validation_dogs = datasets.ImageFolder("/data/dog_images/valid", transform=validation_t
test_dogs = datasets.ImageFolder("/data/dog_images/test", transform=test_transform)

loaders_scratch = {}
loaders_scratch["train"] = torch.utils.data.DataLoader(training_dogs, batch_size=batch_
loaders_scratch["valid"] = torch.utils.data.DataLoader(validation_dogs, batch_size=batc
loaders_scratch["test"] = torch.utils.data.DataLoader(test_dogs, batch_size=batch_size,
```

In [41]:
```
if use_cuda:
    print("GPU Enabled!")
else:
    print("just cpu")
```

```
GPU Enabled!
```

In [42]:
```
(image, label) = next(iter(loaders_scratch["train"]))
print("Train: Tensor: {}, Label: {}".format(image.shape, label.shape))
(image, label) = next(iter(loaders_scratch["test"]))
print("Test: Tensor: {}, Label: {}".format(image.shape, label.shape))
(image, label) = next(iter(loaders_scratch["valid"]))
print("Validation: Tensor: {}, Label: {}".format(image.shape, label.shape))
```

```
Train: Tensor: torch.Size([30, 3, 224, 224]), Label: torch.Size([30])
Test: Tensor: torch.Size([30, 3, 224, 224]), Label: torch.Size([30])
```

```
Validation: Tensor: torch.Size([30, 3, 224, 224]), Label: torch.Size([30])
```

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:

1. I am doing `Resize(224)` and `CenterCrop(224)`. I use the same value for both because I wanted to include the maximum portion of the image. Using centerl crop to make it a square.

2. For training dataset, I used RandomRotation and Horizontal flip to create more variation of the input training dataset.

   For validation, I only used Horizontal Flip.

   For test, data augmentation is not used at all.

   I stopped using normalization after realizing the pixel values are already in float and between 0 to 1.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [44]: 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
                 # [(WF+2P)/S]+1

                 # Input: 224
                 self.conv1_1 = nn.Conv2d(3, 32, 3, padding = 1)
                 # Output: 112
                 self.batch1_conv = nn.BatchNorm2d(32)

                 self.conv2_1 = nn.Conv2d(32, 64, 3, padding = 1)
                 # Output: 56
                 self.batch2_conv = nn.BatchNorm2d(64)

                 self.conv3_1 = nn.Conv2d(64, 128, 3, padding = 1)
                 # Output: 28
                 self.batch3_conv = nn.BatchNorm2d(128)
```

```python
        self.conv4_1 = nn.Conv2d(128, 128, 3, padding = 1)
        # Output: 14
        self.batch4_conv = nn.BatchNorm2d(128)

        # Reducing by half
        self.pool = nn.MaxPool2d(2, 2)

        # 2d dropout
        # self.dropout2d = nn.Dropout2d(0.25)

        # Dense layer
        self.fc1 = nn.Linear(128 * 14 * 14, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 133)

        self.dropout = nn.Dropout(0.5)


    def forward(self, x):
        ## Define forward behavior
        # x = self.dropout2d(x)
        x = F.relu(self.conv1_1(x))
        x = self.pool(x)

        x = F.relu(self.batch2_conv(self.conv2_1(x)))
        x = self.pool(x)

        x = F.relu(self.batch3_conv(self.conv3_1(x)))
        x = self.pool(x)

        x = F.relu(self.batch4_conv(self.conv4_1(x)))
        x = self.pool(x)

        # Flattening
        x = x.view(-1, 128 * 14 * 14)

        #Input -> hidden 1
        x = self.dropout(x)
        x = F.relu(self.fc1(x))

        # hidden 1 -> hidden 2
        x = self.dropout(x)
        x = F.relu(self.fc2(x))

        # hidden 2 -> output
        x = self.dropout(x)
        x = self.fc3(x)
```

```
        return x

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

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:

1.1.9 First Step

- **Convolutional Layer:** 16 32,32 64,64,64 128,128,128
- **FC Layer:** input 512 133
- **Activation**: Use ReLU in all layers as activation function.
- **Regularization**: Used DropOut(0.25) and MaxPool2d(2,2).
- **Reasoning:** Classifying dog breeds inherently looked like a very complicated task so I started with a pretty complicated conv layers with a hope that it will identify the patterns better. But the validation loss was consistently around 4.8 and test accuracy was 1%.

1.1.10 All Interemdiate Steps

- Tried to find out a way to reduce the training loss. Played around with data augmentation, changing nework architectures, using SGD optimizer, varying learning rates without any luck. I started documenting my steps at some point. Those steps are listed at the end of this cell.

1.1.11 Final version

As recommended by the mentors, I used batch normalization between the convolution layers. That magically decreased the training loss and validation loss. I got 24% test accuracy.

- **Convolutional Layer**: 32 64 128 128. 4 blocks, 1 layer in each. Block 3 and 4 has same depth. I did not want increase depth more than 128 but wanted to do another convolution operation expecting that the network will be able to capture more pattern.
- **Activation**: Use ReLU in all layers as activation function. *Same as the initial*
- **FC Layer**: input 512 256 133. Again added two hidden layers assuming it will increase the learning rate.
- **Regularization**: I DropOut(0.5) and used MaxPool2d(2,2).
- **Reasoning**: The first one that worked after numerous failures.

1.1.12 All Trials and Errors

At some point train loss seemed to be constant around 4.87 no matter what I do. The test accuracy was 1%. Following the areas that I have experimented with,

```
1. Using different weight initialization in Linear layer
   * Tried with xavier_uniform_ => same train loss
   * By default it uses uniform_ distribution => same train loss
   * kaiming_normal_ => same train loss
2. Change loss function.
   * CrossEntropyLoss seems to be the best fit for this problem.
3. Use low number of convolution blocks.
   *
4. Use higher number of convolution blocks.
   * 16 -> 32,32 -> 64,64 -> 128,128,128  => same train loss
   * 16 -> 32 -> 64 -> 128 => same train loss
   * 16 -> 32,32 -> 64,64 -> 128 => same train loss
5. Change Dense layer architecture
   * input -> 512 -> 256 -> 133  => same train loss
   * input -> 512 -> 133 => same train loss
   * input -> 5000
 6. Using Dropout in convolutional layer
   *
 7. Use learning rate 0.001
 8. Use momentum
```

1.1.13 (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 [45]:
```
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)
```

1.1.14 (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 [46]:
```
import numpy as np
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import time
from workspaceutils import active_session
```

In [7]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
```

```python
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
    ts1 = ts2 = ts3 = time.time()


    ###################
    # train the model #
    ###################
    model.train()
    print("Running Epoch: {}".format(epoch), end='\r')
    for batch_idx, (data, target) in enumerate(loaders['train']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()

        # Reset optimizer
        optimizer.zero_grad()

        # Feed forward
        output = model(data)

        # Calculate loss
        loss = criterion(output, target)

        # Backprop the loss
        loss.backward()

        # Optimization step
        optimizer.step()

        # Calculate train loss
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.item() - train_loss
        if(batch_idx % 10 == 0):
            print('Intra batch trainning loss: {:.6f}'.format(train_loss), end='\r')

    ts2 = time.time()
    print("Training time: {:3.2f}s".format(ts2-ts1), end='\r')


    ######################
    # validate the model #
    ######################
    model.eval()
    with torch.no_grad():
```

```
            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.item() - valid_

        print("Validation time: {:3.2f}s".format(ts3-ts2), end='\r')


        # print training/validation statistics
        print('Epoch: {} ({:3.2f}s) \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.f
            epoch,
            time.time() - ts1,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if(valid_loss_min > valid_loss):
            print('Validation loss reduced ({} -> {}). Storing...'.format(valid_loss_min
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)

    # return trained model
    return model


# train the model
with active_session():
    model_scratch = train(100, 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 (151.14s)          Training Loss: 4.941184         Validation Loss: 4.882150
Validation loss reduced (inf -> 4.882150326456341). Storing...
Epoch: 2 (132.60s)          Training Loss: 4.836387         Validation Loss: 4.795066
Validation loss reduced (4.882150326456341 -> 4.795066475868225). Storing...
Epoch: 3 (133.17s)          Training Loss: 4.792559         Validation Loss: 4.742783
Validation loss reduced (4.795066475868225 -> 4.742783325059073). Storing...
Epoch: 4 (133.45s)          Training Loss: 4.743351         Validation Loss: 4.715377
Validation loss reduced (4.742783325059073 -> 4.715376513344901). Storing...
```

```
Epoch: 5 (132.26s)        Training Loss: 4.705534        Validation Loss: 4.683668
Validation loss reduced (4.715376513344901 -> 4.683668000357492). Storing...
Epoch: 6 (132.98s)        Training Loss: 4.679967        Validation Loss: 4.670808
Validation loss reduced (4.683668000357492 -> 4.670807940619333). Storing...
Epoch: 7 (133.22s)        Training Loss: 4.659315        Validation Loss: 4.737132
Epoch: 8 (134.77s)        Training Loss: 4.645207        Validation Loss: 4.619360
Validation loss reduced (4.670807940619333 -> 4.619359833853588). Storing...
Epoch: 9 (133.12s)        Training Loss: 4.629600        Validation Loss: 4.707429
Epoch: 10 (132.55s)        Training Loss: 4.606914        Validation Loss: 4.808571
Epoch: 11 (133.97s)        Training Loss: 4.586500        Validation Loss: 4.617708
Validation loss reduced (4.619359833853588 -> 4.61770762716021). Storing...
Epoch: 12 (135.66s)        Training Loss: 4.577278        Validation Loss: 4.561105
Validation loss reduced (4.61770762716021 -> 4.561105472700937). Storing...
Epoch: 13 (134.28s)        Training Loss: 4.557544        Validation Loss: 4.556974
Validation loss reduced (4.561105472700937 -> 4.556973951203482). Storing...
Epoch: 14 (132.44s)        Training Loss: 4.531638        Validation Loss: 4.555874
Validation loss reduced (4.556973951203482 -> 4.555873802730015). Storing...
Epoch: 15 (133.45s)        Training Loss: 4.500557        Validation Loss: 4.455764
Validation loss reduced (4.555873802730015 -> 4.455764174461364). Storing...
Epoch: 16 (134.44s)        Training Loss: 4.486760        Validation Loss: 4.599395
Epoch: 17 (133.47s)        Training Loss: 4.457210        Validation Loss: 4.511585
Epoch: 18 (132.43s)        Training Loss: 4.433437        Validation Loss: 4.535558
Epoch: 19 (133.16s)        Training Loss: 4.408112        Validation Loss: 4.493400
Epoch: 20 (133.59s)        Training Loss: 4.388364        Validation Loss: 5.096106
Epoch: 21 (131.67s)        Training Loss: 4.354593        Validation Loss: 4.536571
Epoch: 22 (131.24s)        Training Loss: 4.325920        Validation Loss: 4.883980
Epoch: 23 (131.77s)        Training Loss: 4.319724        Validation Loss: 4.310263
Validation loss reduced (4.455764174461364 -> 4.310263429369246). Storing...
Epoch: 24 (133.04s)        Training Loss: 4.281604        Validation Loss: 4.323860
Epoch: 25 (131.40s)        Training Loss: 4.243190        Validation Loss: 4.264284
Validation loss reduced (4.310263429369246 -> 4.264283520834787). Storing...
Epoch: 26 (132.90s)        Training Loss: 4.221059        Validation Loss: 4.301149
Epoch: 27 (133.16s)        Training Loss: 4.180931        Validation Loss: 4.379992
Epoch: 28 (133.47s)        Training Loss: 4.161054        Validation Loss: 4.432032
Epoch: 29 (133.67s)        Training Loss: 4.124996        Validation Loss: 4.414614
Epoch: 30 (132.72s)        Training Loss: 4.055329        Validation Loss: 3.939291
Validation loss reduced (4.264283520834787 -> 3.939291085515703). Storing...
Epoch: 31 (132.22s)        Training Loss: 4.026610        Validation Loss: 4.185588
Epoch: 32 (132.68s)        Training Loss: 4.002832        Validation Loss: 3.925028
Validation loss reduced (3.939291085515703 -> 3.9250284263065884). Storing...
Epoch: 33 (131.56s)        Training Loss: 3.972093        Validation Loss: 4.610380
Epoch: 34 (131.25s)        Training Loss: 3.933669        Validation Loss: 3.840933
Validation loss reduced (3.9250284263065884 -> 3.84093280349459). Storing...
Epoch: 35 (132.50s)        Training Loss: 3.923729        Validation Loss: 3.916146
Epoch: 36 (132.78s)        Training Loss: 3.888939        Validation Loss: 4.408803
Epoch: 37 (132.38s)        Training Loss: 3.860575        Validation Loss: 3.801412
Validation loss reduced (3.84093280349459 -> 3.8014121055603023). Storing...
Epoch: 38 (132.10s)        Training Loss: 3.844556        Validation Loss: 3.824283
```

```
Epoch: 39 (133.10s)        Training Loss: 3.811409        Validation Loss: 3.783419
Validation loss reduced (3.8014121055603023 -> 3.7834188001496454). Storing...
Epoch: 40 (133.23s)        Training Loss: 3.783521        Validation Loss: 3.843208
Epoch: 41 (132.42s)        Training Loss: 3.756266        Validation Loss: 4.520872
Epoch: 42 (132.52s)        Training Loss: 3.746385        Validation Loss: 3.954605
Epoch: 43 (132.47s)        Training Loss: 3.702447        Validation Loss: 3.716342
Validation loss reduced (3.7834188001496454 -> 3.716341827596937). Storing...
Epoch: 44 (132.92s)        Training Loss: 3.668956        Validation Loss: 4.109519
Epoch: 45 (131.27s)        Training Loss: 3.665893        Validation Loss: 3.664719
Validation loss reduced (3.716341827596937 -> 3.6647187641688754). Storing...
Epoch: 46 (131.64s)        Training Loss: 3.659790        Validation Loss: 3.908310
Epoch: 47 (132.43s)        Training Loss: 3.616818        Validation Loss: 4.011522
Epoch: 48 (133.98s)        Training Loss: 3.595306        Validation Loss: 3.636843
Validation loss reduced (3.6647187641688754 -> 3.636842804295676). Storing...
Epoch: 49 (132.67s)        Training Loss: 3.571726        Validation Loss: 3.566130
Validation loss reduced (3.636842804295676 -> 3.566130306039537). Storing...
Epoch: 50 (132.28s)        Training Loss: 3.563968        Validation Loss: 4.086761
Epoch: 51 (133.42s)        Training Loss: 3.538390        Validation Loss: 3.647918
Epoch: 52 (133.62s)        Training Loss: 3.526606        Validation Loss: 3.589744
Epoch: 53 (132.02s)        Training Loss: 3.491499        Validation Loss: 3.418389
Validation loss reduced (3.566130306039537 -> 3.418389073440007). Storing...
Epoch: 54 (130.90s)        Training Loss: 3.481420        Validation Loss: 3.436902
Epoch: 55 (132.40s)        Training Loss: 3.465790        Validation Loss: 3.614922
Epoch: 56 (133.57s)        Training Loss: 3.421379        Validation Loss: 3.554297
Epoch: 57 (131.76s)        Training Loss: 3.447459        Validation Loss: 3.449073
Epoch: 58 (131.58s)        Training Loss: 3.392082        Validation Loss: 3.805948
Epoch: 59 (132.29s)        Training Loss: 3.402286        Validation Loss: 3.610820
Epoch: 60 (132.95s)        Training Loss: 3.360239        Validation Loss: 3.939695
Epoch: 61 (131.92s)        Training Loss: 3.374275        Validation Loss: 3.405398
Validation loss reduced (3.418389073440007 -> 3.4053976195199147). Storing...
Epoch: 62 (132.06s)        Training Loss: 3.322160        Validation Loss: 3.269413
Validation loss reduced (3.4053976195199147 -> 3.269413479736873). Storing...
Epoch: 63 (132.75s)        Training Loss: 3.320666        Validation Loss: 3.308740
Epoch: 64 (133.48s)        Training Loss: 3.300586        Validation Loss: 3.297499
Epoch: 65 (130.98s)        Training Loss: 3.277824        Validation Loss: 3.296666
Epoch: 66 (130.66s)        Training Loss: 3.286770        Validation Loss: 3.401966
Epoch: 67 (131.79s)        Training Loss: 3.252364        Validation Loss: 3.233327
Validation loss reduced (3.269413479736873 -> 3.2333272950989858). Storing...
Epoch: 68 (132.76s)        Training Loss: 3.254062        Validation Loss: 3.297263
Epoch: 69 (131.53s)        Training Loss: 3.207307        Validation Loss: 3.373700
Epoch: 70 (129.80s)        Training Loss: 3.226955        Validation Loss: 3.251211
Epoch: 71 (129.86s)        Training Loss: 3.181867        Validation Loss: 3.196840
Validation loss reduced (3.2333272950989858 -> 3.196839792387826). Storing...
Epoch: 72 (130.03s)        Training Loss: 3.184173        Validation Loss: 3.192720
Validation loss reduced (3.196839792387826 -> 3.192720259938921). Storing...
Epoch: 73 (129.61s)        Training Loss: 3.165915        Validation Loss: 3.266239
Epoch: 74 (129.49s)        Training Loss: 3.157217        Validation Loss: 3.157430
Validation loss reduced (3.192720259938921 -> 3.1574299761227196). Storing...
```

```
Epoch: 75 (130.42s)         Training Loss: 3.131554         Validation Loss: 3.158868
Epoch: 76 (131.13s)         Training Loss: 3.130880         Validation Loss: 3.248961
Epoch: 77 (128.96s)         Training Loss: 3.116803         Validation Loss: 3.175594
Epoch: 78 (129.52s)         Training Loss: 3.080968         Validation Loss: 3.110154
Validation loss reduced (3.1574299761227196 -> 3.110154066767011). Storing...
Epoch: 79 (130.42s)         Training Loss: 3.104496         Validation Loss: 3.450674
Epoch: 80 (131.31s)         Training Loss: 3.067707         Validation Loss: 3.236796
Epoch: 81 (129.45s)         Training Loss: 3.051567         Validation Loss: 3.089866
Validation loss reduced (3.110154066767011 -> 3.0898663231304715). Storing...
Epoch: 82 (130.23s)         Training Loss: 3.029843         Validation Loss: 3.088256
Validation loss reduced (3.0898663231304715 -> 3.088255916322981). Storing...
Epoch: 83 (130.65s)         Training Loss: 3.039706         Validation Loss: 3.030453
Validation loss reduced (3.088255916322981 -> 3.0304531114442006). Storing...
Epoch: 84 (131.31s)         Training Loss: 3.046815         Validation Loss: 3.177742
Epoch: 85 (129.70s)         Training Loss: 2.978537         Validation Loss: 3.045809
Epoch: 86 (129.69s)         Training Loss: 2.987076         Validation Loss: 3.075882
Epoch: 87 (130.86s)         Training Loss: 2.982430         Validation Loss: 3.086472
Epoch: 88 (130.69s)         Training Loss: 2.953657         Validation Loss: 3.048207
Epoch: 89 (130.83s)         Training Loss: 2.957855         Validation Loss: 2.980615
Validation loss reduced (3.0304531114442006 -> 2.980614721775055). Storing...
Epoch: 90 (131.20s)         Training Loss: 2.934363         Validation Loss: 3.009865
Epoch: 91 (131.69s)         Training Loss: 2.931088         Validation Loss: 3.191010
Epoch: 92 (134.05s)         Training Loss: 2.935002         Validation Loss: 2.927245
Validation loss reduced (2.980614721775055 -> 2.927244816507612). Storing...
Epoch: 93 (133.99s)         Training Loss: 2.922792         Validation Loss: 2.942953
Epoch: 94 (134.60s)         Training Loss: 2.894114         Validation Loss: 3.021676
Epoch: 95 (134.89s)         Training Loss: 2.884322         Validation Loss: 3.124343
Epoch: 96 (134.64s)         Training Loss: 2.872069         Validation Loss: 2.942584
Epoch: 97 (135.96s)         Training Loss: 2.833519         Validation Loss: 3.104710
Epoch: 98 (136.09s)         Training Loss: 2.827288         Validation Loss: 2.951065
Epoch: 99 (136.64s)         Training Loss: 2.835089         Validation Loss: 2.925249
Validation loss reduced (2.927244816507612 -> 2.9252489549773086). Storing...
Epoch: 100 (138.79s)         Training Loss: 2.834788         Validation Loss: 2.973323
```

1.1.15 (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 [8]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
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: 2.967315


Test Accuracy: 24% (207/836)
```

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.

1.1.16 (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 [47]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch
```

1.1.17 (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 [48]: import torch.nn as nn

In [49]: import torchvision.models as models

         ## TODO: Specify model architecture
         model_transfer = models.densenet201(pretrained=True)

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

         model_transfer.classifier = nn.Sequential(
             nn.Linear(1920, 730),
             nn.ReLU(),
             nn.Linear(730, 133)
         )

         for fc_param in model_transfer.classifier.parameters():
             fc_param.requires_grad = True

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

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/densenet.p
Downloading: "https://download.pytorch.org/models/densenet201-c1103571.pth" to /root/.torch/mode
100%|| 81131730/81131730 [00:01<00:00, 64994479.18it/s]
```

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

1.1.18 Choosing Network

- First choice was ResNet because I saw a chart of ILSVRC error rates in different years where ResNet was the most accurate one with 3.57% error rate.
- While looking at the pytorch models section, I noticed DensNet has some lower error rates for some cases.
- Based on the paper, https://arxiv.org/pdf/1608.06993.pdf in Figure 3, it looks like DensNet201 is performing better than ResNet50 for same number of parameters with same numbr FLOPs. Did not want to go beyond ResNet50 or DensNet201 only because of their higher FLOP requirements.

1.1.19 Deciding Classifier

- Size of input vector to classifier is 1920 and size of output vector is 133. I included a hidden layer of 730 nodes. This is a number around the point of 2/3rd of the difference between input size and output size.

1.1.20 (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 [50]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.01)
```

1.1.21 (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 [51]: import numpy as np
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         import time
         from workspaceutils import active_session

In [15]: # train the model
         def train(epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             valid_loss_min = np.Inf

             for epoch in range(1, n_epochs+1):
                 train_loss = 0.0
                 valid_loss = 0.0
                 ts1 = ts2 = time.time()

                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     optimizer.zero_grad()
                     output = model(data)
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()
                     train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.item() - train_los
                     if(batch_idx % 10 == 0):
                         print('Trainning loss: {:.6f}...'.format(train_loss), end='\r')
```

```
            ts2 = time.time()
            print("Training time: {:3.2f}s".format(ts2-ts1), end='\r')

            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                output = model(data)
                loss = criterion(output, target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.item() - valid_los

            # print training/validation statistics
            print('Epoch: {} ({:3.2f}s) \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.
                epoch,
                time.time() - ts1,
                train_loss,
                valid_loss
                ))

            if(valid_loss_min > valid_loss):
                print('Validation loss reduced ({} -> {}). Storing...'.format(valid_loss_mi
                valid_loss_min = valid_loss
                torch.save(model.state_dict(), save_path)


    n_epochs = 30
    with active_session():
        model_transfer = train(n_epochs, loaders_transfer, model_transfer,
                               optimizer_transfer, criterion_transfer, use_cuda, 'model_transfe
```

```
Epoch: 1 (197.35s)          Training Loss: 1.530174         Validation Loss: 1.285070
Validation loss reduced (inf -> 1.2850697061845235). Storing...
Epoch: 2 (197.71s)          Training Loss: 1.516531         Validation Loss: 1.317766
Epoch: 3 (197.51s)          Training Loss: 1.539513         Validation Loss: 0.979705
Validation loss reduced (1.2850697061845235 -> 0.9797047227621077). Storing...
Epoch: 4 (197.65s)          Training Loss: 1.471492         Validation Loss: 1.242100
Epoch: 5 (197.38s)          Training Loss: 1.444955         Validation Loss: 1.229433
Epoch: 6 (197.08s)          Training Loss: 1.449855         Validation Loss: 1.109189
Epoch: 7 (197.35s)          Training Loss: 1.399830         Validation Loss: 1.083606
Epoch: 8 (197.63s)          Training Loss: 1.512026         Validation Loss: 1.373642
Epoch: 9 (197.52s)          Training Loss: 1.494645         Validation Loss: 1.205852
Epoch: 10 (198.01s)          Training Loss: 1.422205         Validation Loss: 1.243280
Epoch: 11 (198.18s)          Training Loss: 1.472805         Validation Loss: 1.059874
Epoch: 12 (197.88s)          Training Loss: 1.367011         Validation Loss: 1.199502
Epoch: 13 (198.63s)          Training Loss: 1.391085         Validation Loss: 1.218227
Epoch: 14 (199.14s)          Training Loss: 1.425874         Validation Loss: 1.180437
Epoch: 15 (198.38s)          Training Loss: 1.456709         Validation Loss: 1.179387
```

```
Epoch: 16 (199.31s)        Training Loss: 1.460287        Validation Loss: 1.203491
Epoch: 17 (197.99s)        Training Loss: 1.412267        Validation Loss: 1.151448
Epoch: 18 (197.53s)        Training Loss: 1.427482        Validation Loss: 1.258504
Epoch: 19 (198.08s)        Training Loss: 1.336290        Validation Loss: 1.127632
Epoch: 20 (198.72s)        Training Loss: 1.415259        Validation Loss: 1.060623
Epoch: 21 (198.15s)        Training Loss: 1.398887        Validation Loss: 1.050416
Epoch: 22 (198.11s)        Training Loss: 1.368552        Validation Loss: 1.085698
Epoch: 23 (198.40s)        Training Loss: 1.437224        Validation Loss: 1.138688
Epoch: 24 (199.57s)        Training Loss: 1.497325        Validation Loss: 1.116832
Epoch: 25 (199.75s)        Training Loss: 1.477076        Validation Loss: 1.049507
Epoch: 26 (198.81s)        Training Loss: 1.459051        Validation Loss: 1.235004
Epoch: 27 (198.22s)        Training Loss: 1.432079        Validation Loss: 1.084422
Epoch: 28 (198.16s)        Training Loss: 1.452912        Validation Loss: 1.199832
Epoch: 29 (197.98s)        Training Loss: 1.401630        Validation Loss: 1.384973
Epoch: 30 (198.57s)        Training Loss: 1.414215        Validation Loss: 1.225356
```

```
In [52]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.22 (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 [53]: 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))

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

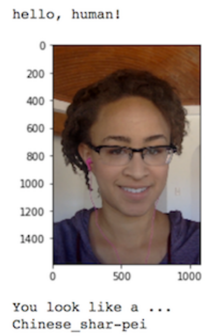
```
Test Loss: 1.123980


Test Accuracy: 67% (567/836)
```

1.1.23 (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 [67]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
from PIL import Image

# 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 training_dogs.classes]
def get_image_tensor(img_path):
    transformer = T.Compose([
        T.Resize(224),
        T.CenterCrop(224),
        T.ToTensor()
    ])
    image = transformer(Image.open(img_path))[:3,:,:].unsqueeze_(0)
#     image.unsqueeze_(0)
    if use_cuda:
        image = image.cuda()
    return image

def predict_breed_transfer(img_path):
    image_tensor = get_image_tensor(img_path)
    output = model_transfer(image_tensor)
    pred = output.data.max(1, keepdim=True)[1]

    return class_names[pred]

file_name = '/data/dog_images/valid/051.Chow_chow/Chow_chow_03657.jpg'
predict_breed_transfer(file_name)
```

Out[67]: 'Chow chow'



Sample Human Output

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.24 (IMPLEMENTATION) Write your Algorithm

In [103]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def app_output(img_path, text):
    img = Image.open(img_path)
    plt.imshow(img)
    plt.text(10, -20, text)
    plt.show()

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    print(img_path)
    is_dog = dog_detector(img_path)
    if is_dog == True:
        breed = predict_breed_transfer(img_path)
        app_output(img_path, 'Hello Dog!! You look like a {}'.format(breed))
    else:
        is_human = face_detector(img_path)
        if is_human == True:
            breed = predict_breed_transfer(img_path)
            app_output(img_path, 'Hello Human! You look like a {}'.format(breed))
        else:
```

```
            app_output(img_path, 'Neither a dog nor a 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?

1.1.25 (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.

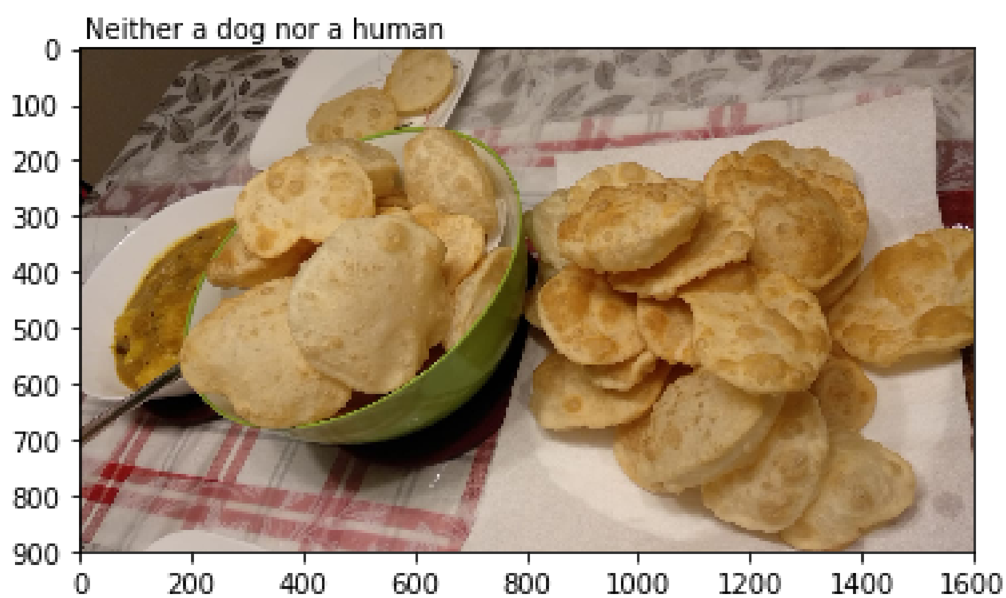
Answer: The output is better than I expected.

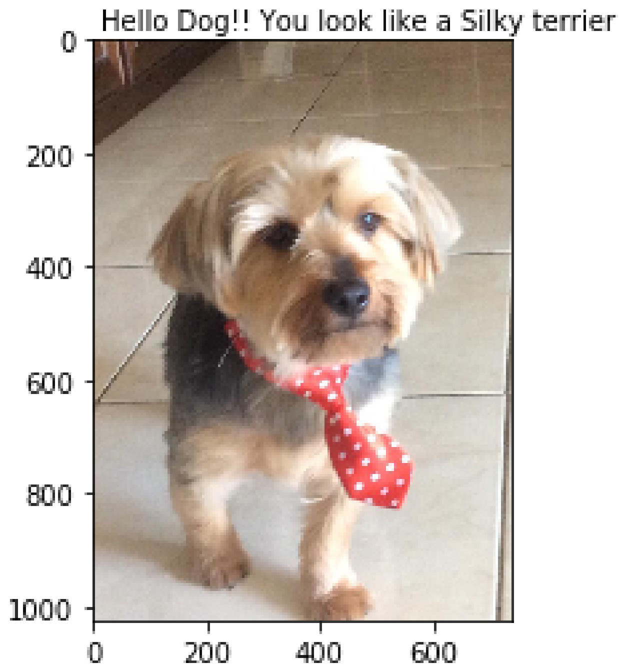
Scopes of improvements:

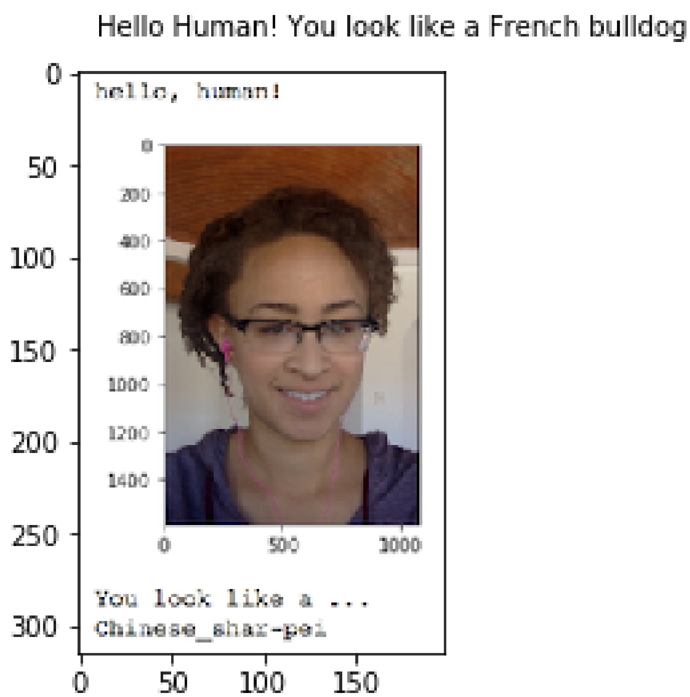
- Use data augmentation to increase the number of input images.
- Use DropOut2d in convolutional network as a regularization technique.
- Increasing the convolution stride size an filter size.

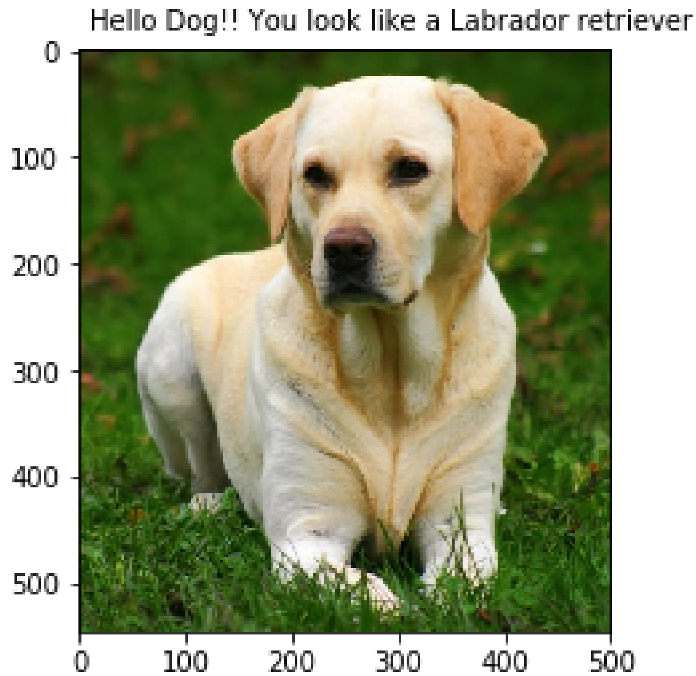
```
In [109]: ## TODO: Execute your algorithm from Step 6 on
          ## at least 6 images on your computer.
          ## Feel free to use as many code cells as needed.

          ## suggested code, below
          app_inputs = np.array(glob("images/*"))

          for file in app_inputs:
              run_app(file)
```

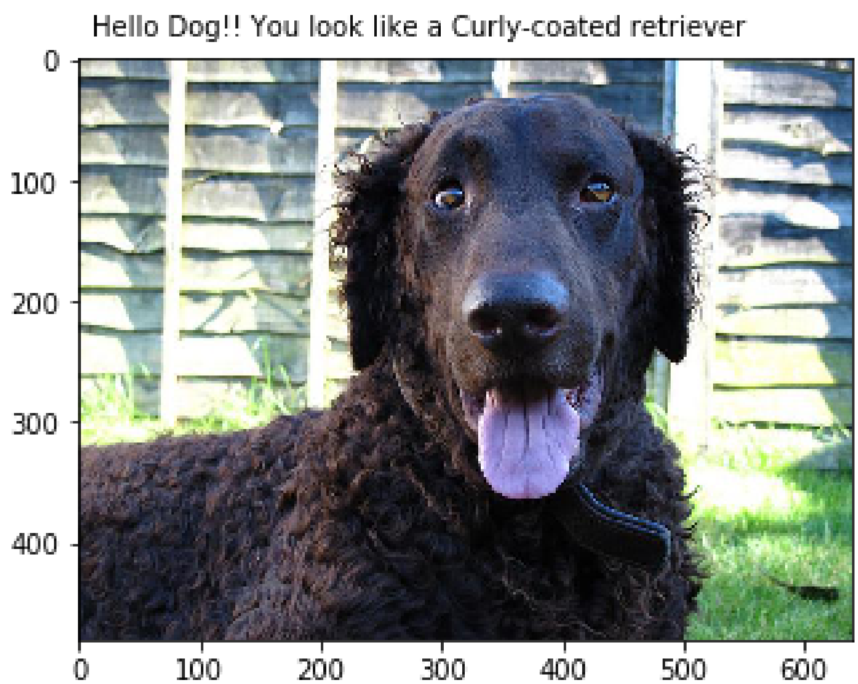

Hello Dog!! You look like a Welsh springer spaniel
0
50
100
150
200
250
300
350
JustDogBreeds.com
0
100
200
300
400
500

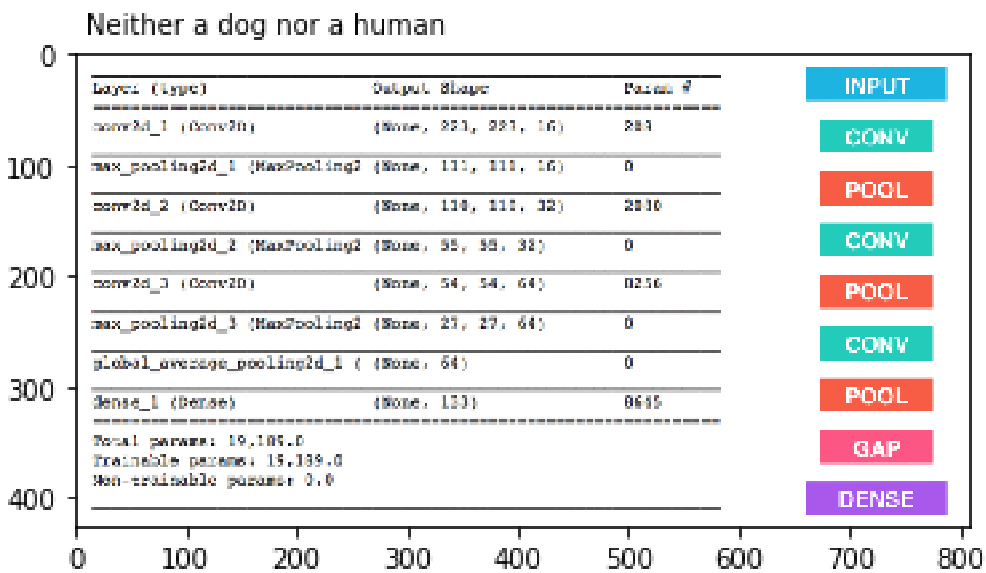

Neither a dog nor a human
0
100
200
300
400
500
600
700
800
900
0
200
400
600
800
1000
1200
1400
1600

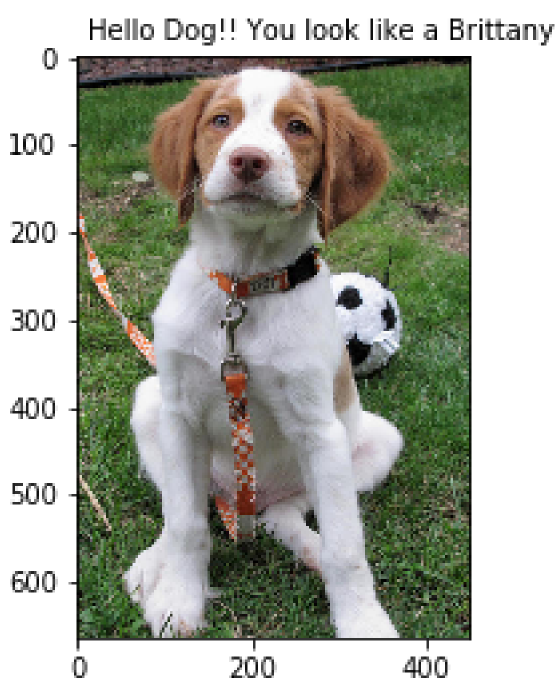

Hello Dog!! You look like a Silky terrier
0
200
400
600
800
1000
0
200
400
600


Hello Human! You look like a French bulldog
hello, human!
You look like a ...
Chinese_shar-pei
0
50
100
150
200
250
300
0
50
100
150


Hello Dog!! You look like a Labrador retriever
0
100
200
300
400
500
0
100
200
300
400
500

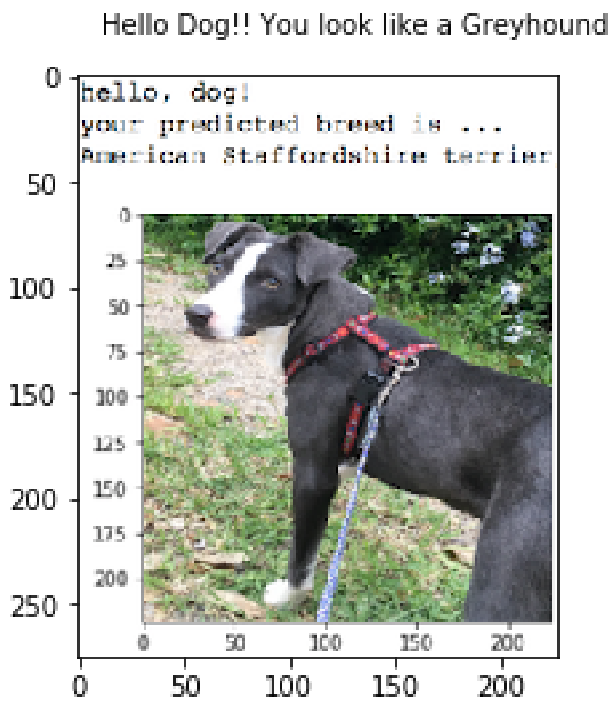

Hello Dog!! You look like a Curly-coated retriever
0
100
200
300
400
0
100
200
300
400
500
600

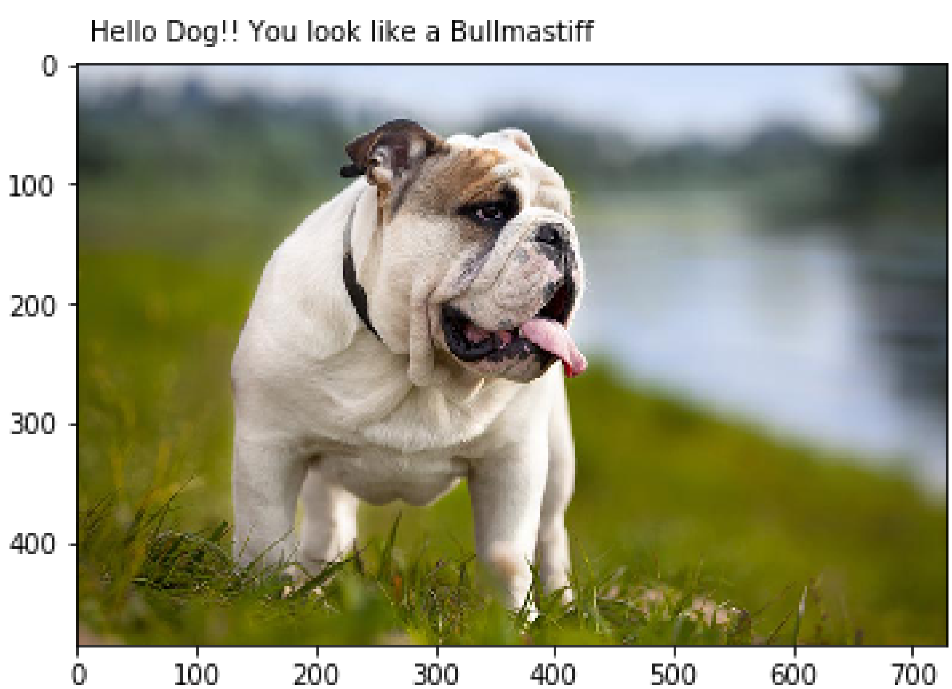

Neither a dog nor a human
INPUT
CONV
POOL
CONV
POOL
CONV
POOL
GAP
DENSE


Hello Dog!! You look like a Brittany

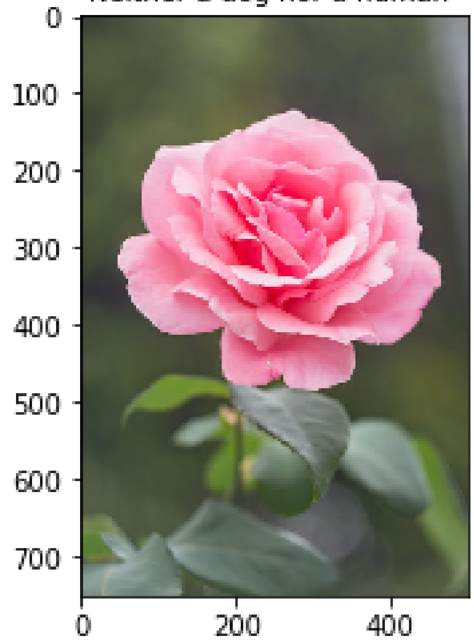
Hello Dog!! You look like a Plott

Hello Dog!! You look like a American water spaniel

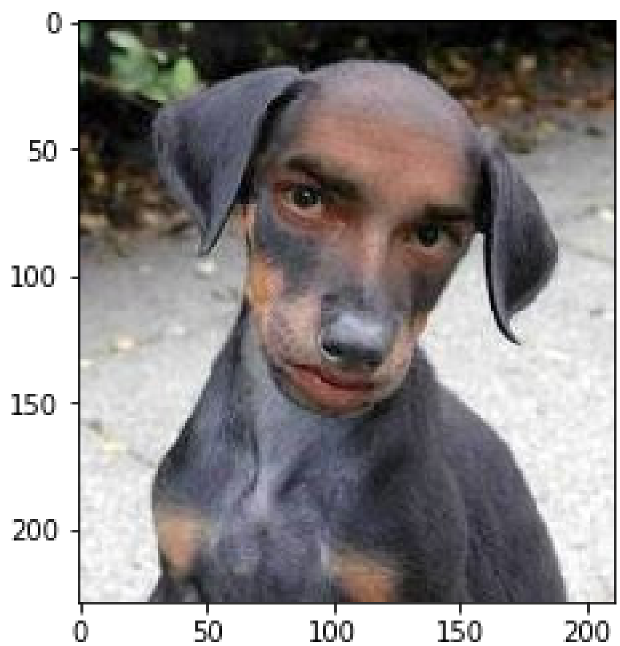

Hello Dog!! You look like a Greyhound
hello, dog!
your predicted breed is ...
American Staffordshire terrier

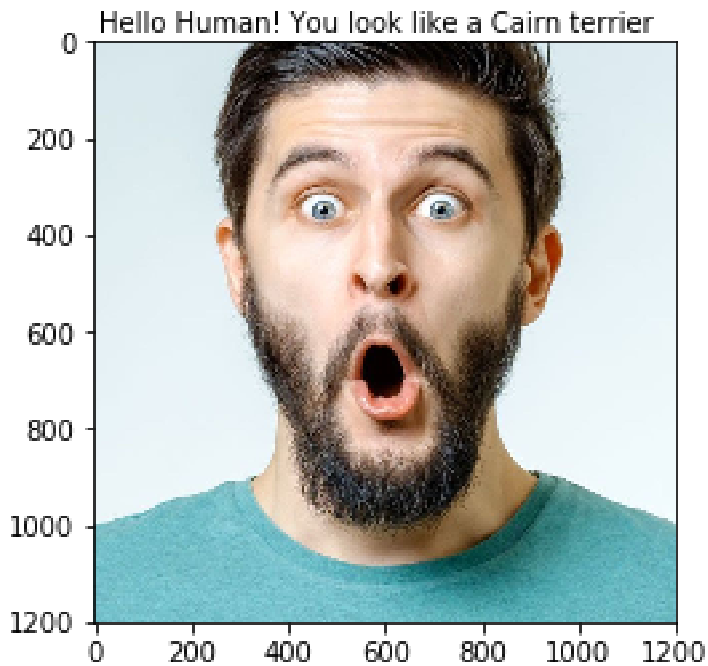

Hello Dog!! You look like a Bullmastiff

Neither a dog nor a human



Hello Dog!! You look like a Doberman pinscher




Hello Human! You look like a Cairn terrier
0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200


Hello Dog!! You look like a Chesapeake bay retriever
0
100
200
300
400
0
100
200
300
400
500
600
700