dog_app

May 3, 2021

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

```
In [1]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

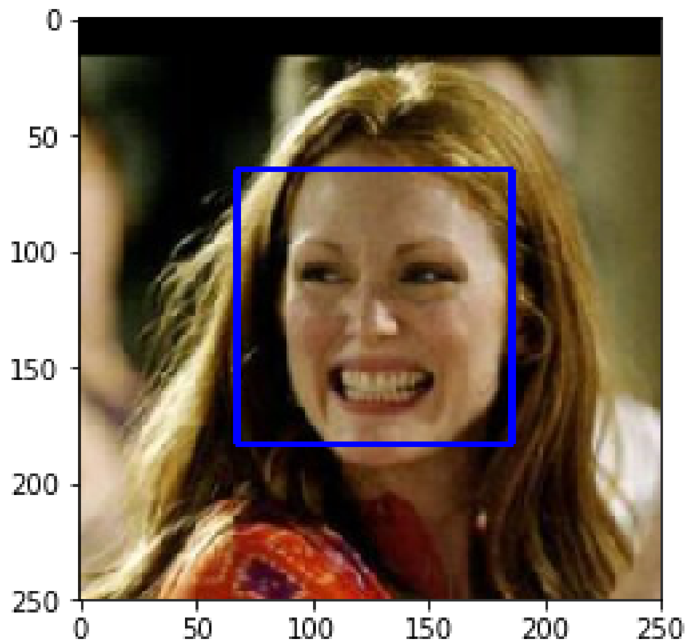
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

        # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # load color (BGR) image
        img = cv2.imread(human_files[44])
        # 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:

Corrected identified humans percentage is: 98.0%

Misidentified humans in dogs dataset percentage is: 17.0%

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

        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.



        def calculate_percetages(detector):

            size = len(human_files_short)
            human_percetage = 0.
            dog_percentage = 0.

            step = 1 * 100 / len(human_files_short)

            for human_path, dog_path in tqdm(zip(human_files_short, dog_files_short), total=size
                if detector(human_path):
                    human_percetage += step
```

```
            if detector(dog_path):
                dog_percentage += step

        return human_percetage, dog_percentage
```

```
In [5]: humans, dogs = calculate_percetages(face_detector)
```

```
100%|| 100/100 [00:32<00:00,  3.06it/s]
```

```
In [6]: print(f'Corrected identified humans percentage is: {humans}%')
        print(f'Misidentified humans in dogs dataset percentage is: {dogs}%')
```

```
Corrected identified humans percentage is: 98.0%
Misidentified humans in dogs dataset percentage is: 17.0%
```

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

```
In [7]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [5]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:41<00:00, 13186779.12it/s]
```

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

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

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

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

In [6]:
```
from PIL import Image
import torchvision.transforms as transforms

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

    # Load Image
    img = Image.open(img_path)

    # Apply transforms
    test_transforms = transforms.Compose([transforms.Resize([224, 224]),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406],
                                                               [0.229, 0.224, 0.225])])
    transformed = test_transforms(img).unsqueeze(0)

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

```
        output = VGG16.forward(transformed)

        # ps = torch.exp(output)

        _, pred = torch.max(output, 1)

        correct = np.squeeze(pred)

        return correct # predicted class index
```

In [10]:
```
VGG16_predict(dog_files_short[0])
```

Out[10]:
```
tensor(243, device='cuda:0')
```

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 [25]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    dog_detected = False

    output = VGG16_predict(img_path)
    if output in range(151, 269):
        dog_detected = True

    return dog_detected
```

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:
Humans identified as dogs percentage is: 0.0%
Dogs correct identified percentage is: 100.0%

In [12]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

humans, dogs = calculate_percetages(dog_detector)
```

```
100%|| 100/100 [00:08<00:00, 13.55it/s]
```

```
In [13]: print(f'Humans identified as dogs percentage is: {humans}%')
         print(f'Dogs correct identified percentage is: {dogs}%')
```

```
Humans identified as dogs percentage is: 0.0%
Dogs correct identified percentage is: 100.0%
```

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

```
In [14]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

In [7]:
```
import os
import torch
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import RandomSampler

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

# number of subprocesses to use for data loading
num_workers = 0

# how many samples per batch to load
batch_size = 64

# dog classes in dataset
dog_classes = 133

# Apply transforms

normalization =  transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

transform = transforms.Compose([transforms.Resize([224, 224]),
                                transforms.ToTensor(),
                                normalization])

train_transform = transforms.Compose([transforms.Resize([224, 224]),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation(10),
                                      transforms.ToTensor(),
                                      normalization])
```

```
train_data = datasets.ImageFolder('/data/dog_images/train', transform=train_transform)
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=transform)
test_data = datasets.ImageFolder('/data/dog_images/test', transform=transform)

# define samplers for obtaining training and validation batches
train_sampler = RandomSampler(train_data)
valid_sampler = RandomSampler(valid_data)

# prepare data loaders (combine dataset and sampler)

loaders_scratch = {
    'train': torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=trai
    'valid': torch.utils.data.DataLoader(valid_data, batch_size=batch_size, sampler=vali
    'test':  torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False
}
```

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:

The first transformation applied in the images was to resize to [224, 224] pixels. These values were taken from the VGG16 inputs since this network has good performance.

The reason for choosing the resize operation instead of cropping the image was not to lose possible essential features from the image.

The augmentation operations were based on the previous examples from Udacity lectures. Random horizontal flip and random rotation were chosen to augment the dataset.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [8]: import torch.nn as nn
        import torch.nn.functional as F

        # define the CNN architecture
        class Net(nn.Module):
            ### TODO: choose an architecture, and complete the class
            def __init__(self):
                super(Net, self).__init__()
                ## Define layers of a CNN
                self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
                self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
```

```
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (256 * 7 * 7 -> 500)
        self.fc1 = nn.Linear(256 * 7 * 7, 500)
        # linear layer (500 -> 133)
        self.fc2 = nn.Linear(500, 133)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        ## Define forward behavior
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))
        # flatten image input
        x = x.view(-1, 256 * 7 * 7)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

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

# 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:

- Step 1: The starting point to build the architecture was based on the lecture notebook were the CIFAR-10 database was explored. This configuration was not robust and did not show good accuracy results above 10%.

- Step 2: Two more convolutional layers were added to the architecture hopping to help the filters catch more details from the image. Still, the test accuracy did not perform well.
- Step 3: The optimizer was swapped from SGD to Adam.

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 [9]: import torch.optim as optim

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [10]: 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

                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()
                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)
                     # calculate the batch loss
                     loss = criterion(output, target)
```

```
                # backward pass: compute gradient of the loss with respect to model paramet
                loss.backward()
                # perform a single optimization step (parameter update)
                optimizer.step()

                ## record the average training loss, using something like
                ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                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: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average 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
            # save model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss


        # return trained model
        return model
```

```
In [19]: # train the model
         model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
```

```
                                    criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.883981         Validation Loss: 4.854794
Validation loss decreased (inf --> 4.854794).  Saving model ...
Epoch: 2         Training Loss: 4.719866         Validation Loss: 4.484887
Validation loss decreased (4.854794 --> 4.484887).  Saving model ...
Epoch: 3         Training Loss: 4.479741         Validation Loss: 4.459697
Validation loss decreased (4.484887 --> 4.459697).  Saving model ...
Epoch: 4         Training Loss: 4.307276         Validation Loss: 4.335955
Validation loss decreased (4.459697 --> 4.335955).  Saving model ...
Epoch: 5         Training Loss: 4.163141         Validation Loss: 4.185121
Validation loss decreased (4.335955 --> 4.185121).  Saving model ...
Epoch: 6         Training Loss: 3.996287         Validation Loss: 4.154917
Validation loss decreased (4.185121 --> 4.154917).  Saving model ...
Epoch: 7         Training Loss: 3.789764         Validation Loss: 3.956946
Validation loss decreased (4.154917 --> 3.956946).  Saving model ...
Epoch: 8         Training Loss: 3.653620         Validation Loss: 3.810956
Validation loss decreased (3.956946 --> 3.810956).  Saving model ...
Epoch: 9         Training Loss: 3.498921         Validation Loss: 3.848896
Epoch: 10         Training Loss: 3.343244         Validation Loss: 3.789235
Validation loss decreased (3.810956 --> 3.789235).  Saving model ...
Epoch: 11         Training Loss: 3.214835         Validation Loss: 3.830738
Epoch: 12         Training Loss: 3.070912         Validation Loss: 4.111822
Epoch: 13         Training Loss: 2.899936         Validation Loss: 3.891617
Epoch: 14         Training Loss: 2.799925         Validation Loss: 3.907792
Epoch: 15         Training Loss: 2.677295         Validation Loss: 3.859073
Epoch: 16         Training Loss: 2.525458         Validation Loss: 3.827398
Epoch: 17         Training Loss: 2.384200         Validation Loss: 3.963200
Epoch: 18         Training Loss: 2.279096         Validation Loss: 3.954926
Epoch: 19         Training Loss: 2.170022         Validation Loss: 3.923759
Epoch: 20         Training Loss: 2.077986         Validation Loss: 4.408227
```

```
In [11]: # 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 [12]: def test(loaders, model, criterion, use_cuda):

             # monitor test loss and accuracy
             test_loss = 0.
             correct = 0.
             total = 0.

             model.eval()
```

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

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

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

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

```
Test Loss: 3.794767


Test Accuracy: 11% (100/836)
```

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 [14]:
```
## TODO: Specify data loaders
import torch

loaders_transfer = {
    'train': torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=tra
```

```
            'valid': torch.utils.data.DataLoader(valid_data, batch_size=batch_size, sampler=val
            'test':  torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=Fals
        }
```

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

         ## TODO: Specify model architecture

         ## Let's try the resnet50 architecture
         model_transfer = models.resnet50(pretrained=True)

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

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:03<00:00, 30092949.02it/s]
```

```
In [16]: # print out the model structure
         print(model_transfer)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
```

```
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
```

```
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

In [17]:
```python
# We observe the last part of the classifier has 2048 in features and 1000 out
print(model_transfer.fc.in_features)
print(model_transfer.fc.out_features)
```

```
2048
1000
```

In [18]:
```python
# Let's rebuild the out part to have the 133 dog classes

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

n_inputs = model_transfer.fc.in_features

# add last linear layer (n_inputs -> 133 dog classes)
# new layers automatically have requires_grad = True
last_layer = nn.Linear(n_inputs, dog_classes)

model_transfer.fc = last_layer

# if GPU is available, move the model to GPU
if use_cuda:
    model_transfer.cuda()

# check to see that your last layer produces the expected number of outputs
print(model_transfer.fc.out_features)
```

```
133
```

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:

- Step 1: The starting point to build the architecture was based on VGG16. This architecture was previous introduced in the lecture notebooks. This configuration was ok and did show accuracy results of around 70%. Nevertheless, these results could be improved, so the investigation continued.
- Step 2: A good architecture for classification was found doing research around the internet. It performed pretty well in the classic datasets, such as CIFAR10. The architecture is the Resnet50, so the new transfer learning architecture was based on this one.
- Step 3: The last layer of the Resnet50 has 2048 in and 1000 out features. This last layer had to be changed to classify out 133 dog breed contained in the dataset.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [19]: import torch.optim as optim

         # For multiclass classification (categorical cross-entropy)
         criterion_transfer = nn.CrossEntropyLoss()

         # Use Adam optimizer
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [26]: # train the model
         model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer,
                                criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1         Training Loss: 2.442321        Validation Loss: 1.065222
Validation loss decreased (inf --> 1.065222).  Saving model ...
Epoch: 2         Training Loss: 0.809652        Validation Loss: 0.726737
Validation loss decreased (1.065222 --> 0.726737).  Saving model ...
Epoch: 3         Training Loss: 0.570087        Validation Loss: 0.647549
Validation loss decreased (0.726737 --> 0.647549).  Saving model ...
Epoch: 4         Training Loss: 0.454914        Validation Loss: 0.611771
Validation loss decreased (0.647549 --> 0.611771).  Saving model ...
Epoch: 5         Training Loss: 0.401838        Validation Loss: 0.598601
Validation loss decreased (0.611771 --> 0.598601).  Saving model ...
Epoch: 6         Training Loss: 0.339169        Validation Loss: 0.639760
Epoch: 7         Training Loss: 0.315644        Validation Loss: 0.538174
Validation loss decreased (0.598601 --> 0.538174).  Saving model ...
Epoch: 8         Training Loss: 0.277626        Validation Loss: 0.549074
Epoch: 9         Training Loss: 0.243422        Validation Loss: 0.550934
```

```
Epoch: 10 	Training Loss: 0.219789 	Validation Loss: 0.491398
Validation loss decreased (0.538174 --> 0.491398).  Saving model ...
Epoch: 11 	Training Loss: 0.205040 	Validation Loss: 0.562686
Epoch: 12 	Training Loss: 0.201552 	Validation Loss: 0.462077
Validation loss decreased (0.491398 --> 0.462077).  Saving model ...
Epoch: 13 	Training Loss: 0.185581 	Validation Loss: 0.503030
Epoch: 14 	Training Loss: 0.166988 	Validation Loss: 0.495990
Epoch: 15 	Training Loss: 0.161160 	Validation Loss: 0.523461
Epoch: 16 	Training Loss: 0.154318 	Validation Loss: 0.517130
Epoch: 17 	Training Loss: 0.146912 	Validation Loss: 0.486600
Epoch: 18 	Training Loss: 0.142056 	Validation Loss: 0.612026
Epoch: 19 	Training Loss: 0.131549 	Validation Loss: 0.531115
Epoch: 20 	Training Loss: 0.131594 	Validation Loss: 0.498021
```

```
In [20]: # 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 [21]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.512411


Test Accuracy: 83% (698/836)
```

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 [80]: ### 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 loaders_transfer['train'].dataset

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

             # Apply transforms
             transform = transforms.Compose([transforms.Resize([224, 224]),
                                             transforms.ToTensor(),
```

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

        transformed = transform(img).unsqueeze(0)

        if use_cuda:
            transformed = transformed.cuda()

        output = model_transfer(transformed)

        # Calculate accuracy
        _, preds_tensor = torch.max(output, 1)
        preds = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tens
        predicted_class = class_names[preds]

        # Dog mutt estimation
        ps = F.softmax(output, dim=1)
        ps = ps if not use_cuda else ps.cpu()
        top_p, top_class = ps.topk(2, dim=1)

        top_p = top_p.detach().numpy().squeeze() * 100
        top_class = top_class.detach().numpy().squeeze()

        class_and_percetage = []

        for index, percentage in zip(top_class, top_p):
            predicted_class = class_names[index]
            class_and_percetage.append([predicted_class, percentage])

        return class_and_percetage
```

Step 5: Write your Algorithm

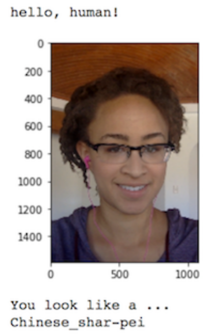
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is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

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [93]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
```



Sample Human Output

```python
def run_app(img_path):
    ## handle cases for a human face, dog, and neither

    if dog_detector(img_path):
        print("\nHello Dog!")
    elif face_detector(img_path):
        print("\nHello Human!")
    else:
        print("\nHello Stranger!")

    # Print image
    img = cv2.imread(img_path)
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.show()

    results = predict_breed_transfer(img_path)

    print("You look like a...\n")

    for result in results:
        percentage = "{:.2f}".format(result[1])
        print(f"{result[0]} with probability of {percentage}%")
```

Step 6: Test Your Algorithm
In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

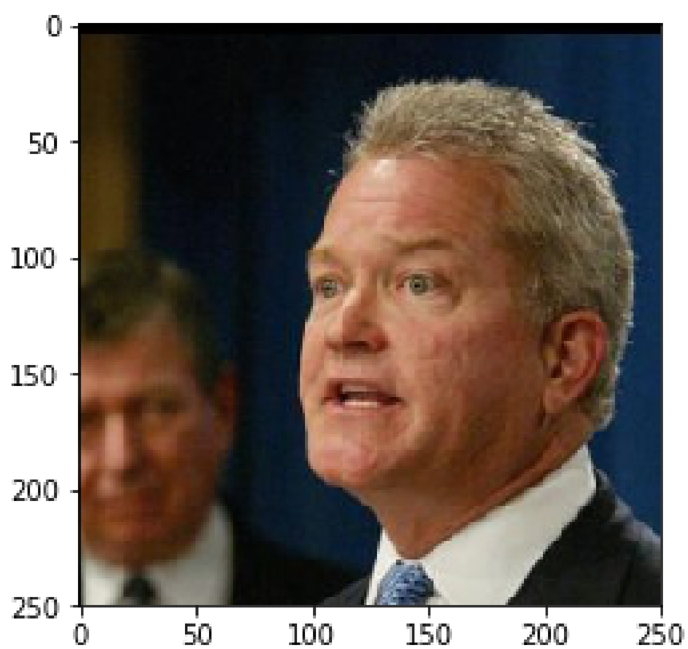
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

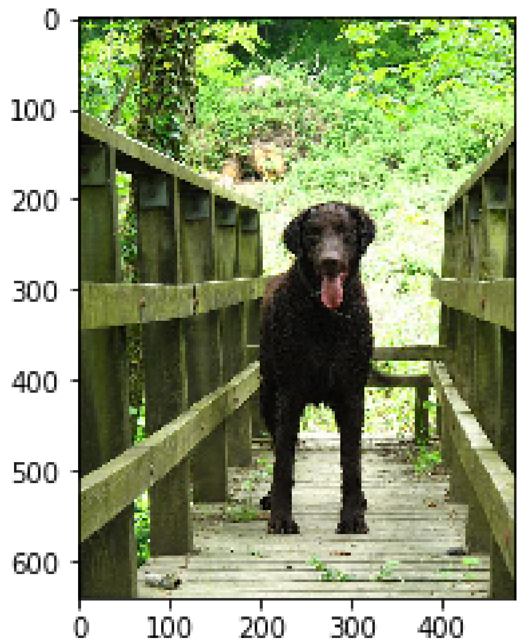
- Point 1: The Test Accuracy of 83% is OK. However, improve the accuracy to reach around 90% would be better. This can be achieved by either gathering new images to improve the dataset or to define better augmentation operations.
- Point 2: There was no test regarding images with a dog and a human in the same scene. The algorithm could fail badly in cases like this. A point of improvement would be to test out this kind of inputs and propose a segmentation network to crop humans and dogs in the scene.
- Point 3: Make it a live app using the webcam to get image input.

In [119]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
import random

humans = random.sample(list(human_files), 5)
dogs = random.sample(list(dog_files), 5)

## suggested code, below
for h, d in zip(humans, dogs):
    run_app(h)
    run_app(d)
```

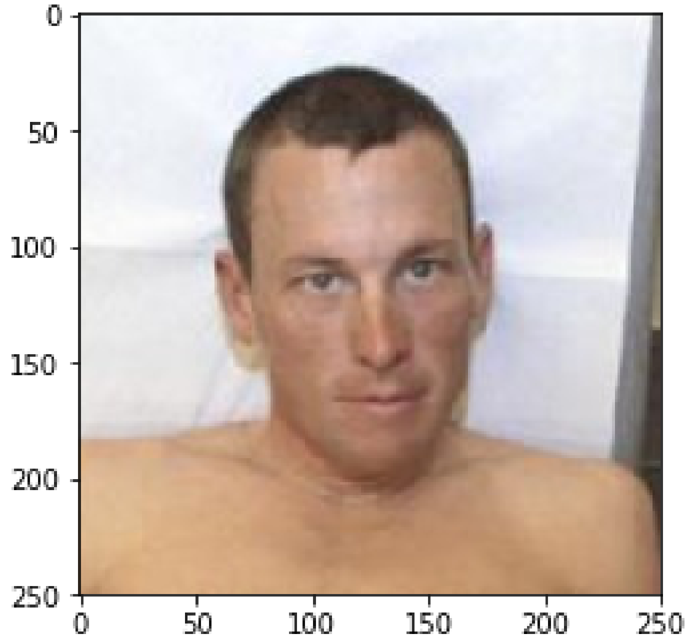
```
Hello Human!
```



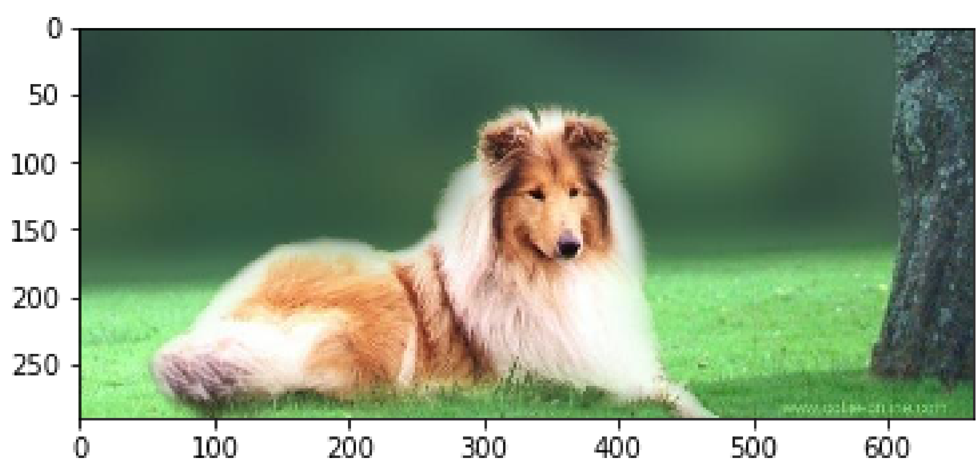
```
You look like a...

Portuguese water dog with probability of 19.98%
Old english sheepdog with probability of 15.27%

Hello Dog!
```



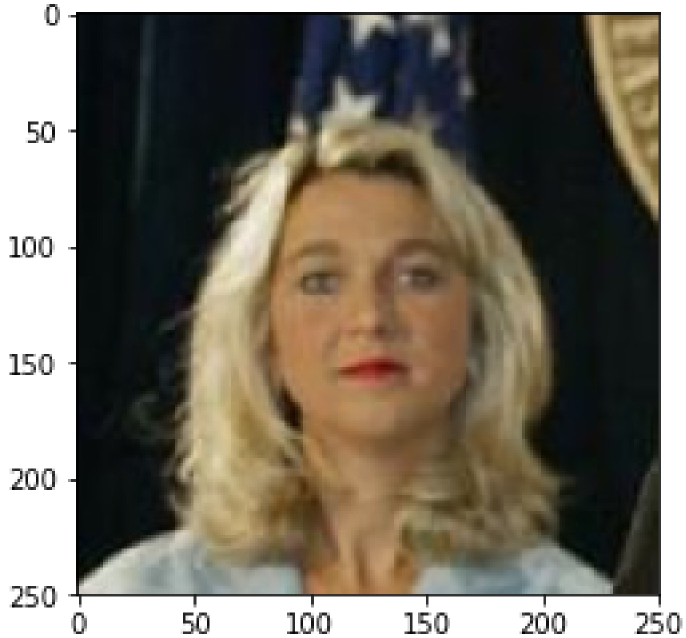

```
You look like a...

Curly-coated retriever with probability of 99.94%
Chesapeake bay retriever with probability of 0.02%

Hello Human!
```



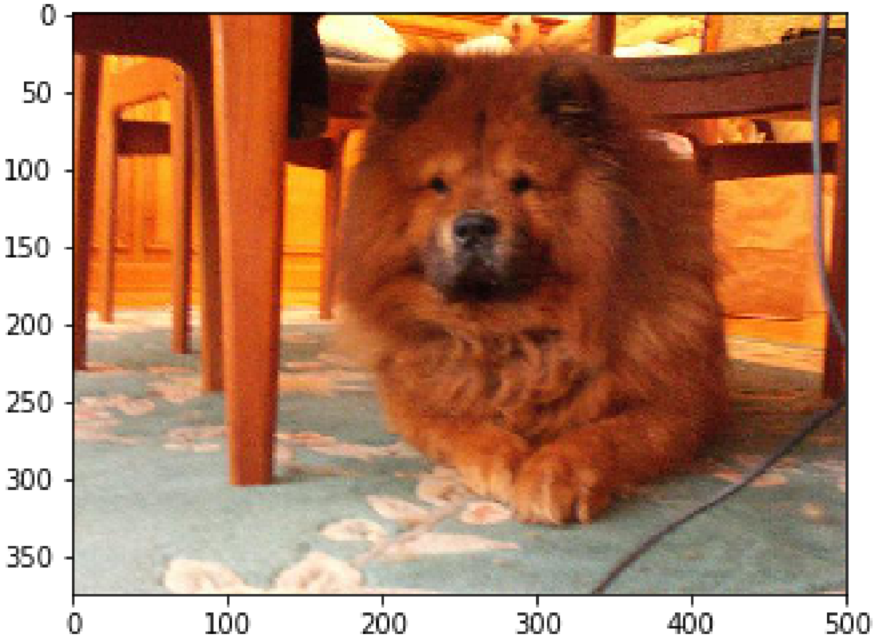

```
You look like a...

Dogue de bordeaux with probability of 31.33%
French bulldog with probability of 7.91%

Hello Dog!
```



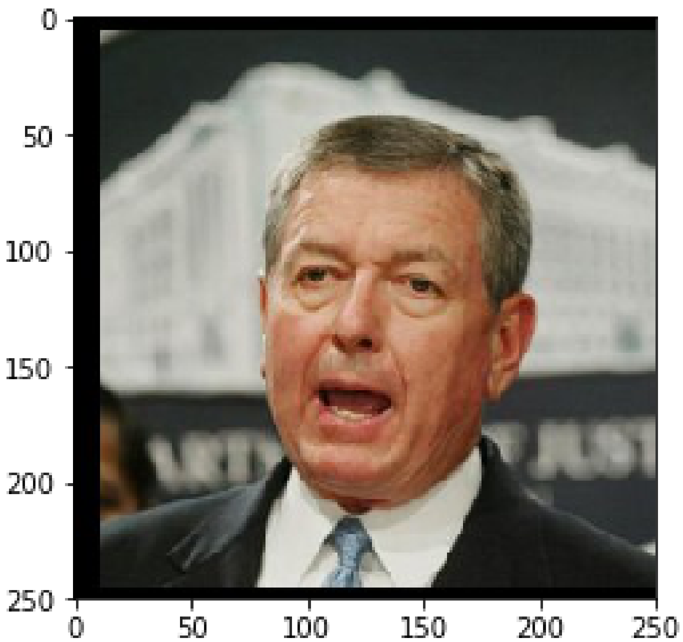
```
You look like a...

Collie with probability of 99.66%
Bearded collie with probability of 0.09%

Hello Human!
```



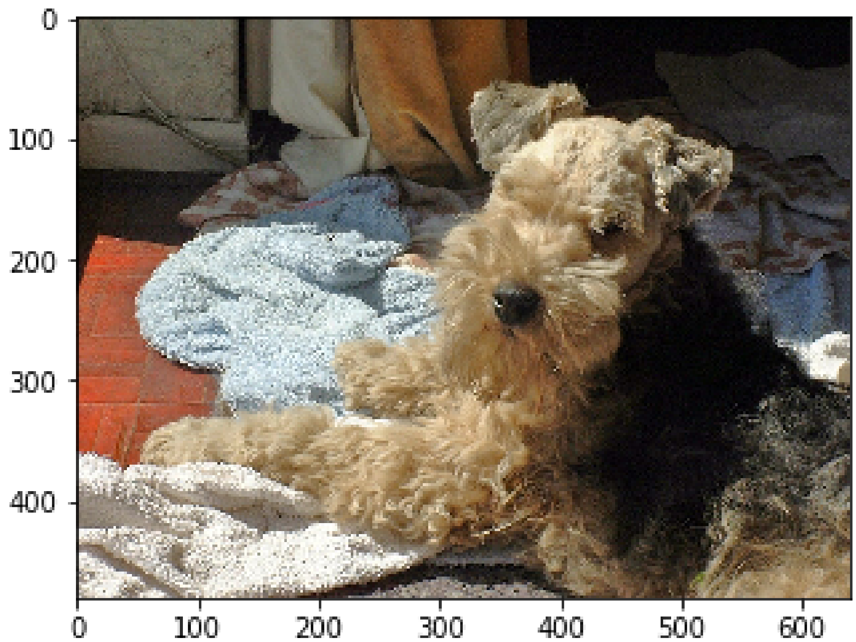

```
You look like a...

Dachshund with probability of 30.03%
American water spaniel with probability of 6.92%

Hello Dog!
```



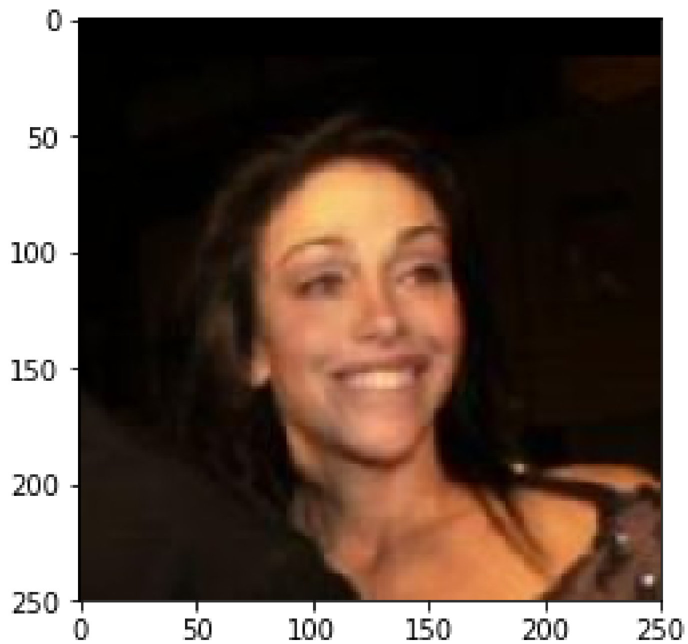

```
You look like a...

Chow chow with probability of 100.00%
Finnish spitz with probability of 0.00%

Hello Human!
```



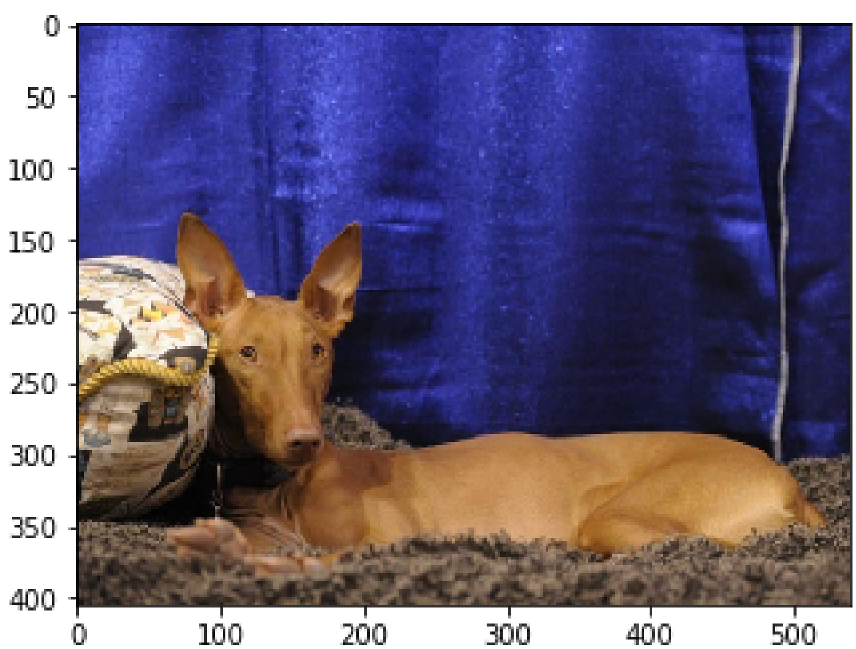
```
You look like a...

Dachshund with probability of 15.93%
Lowchen with probability of 13.64%

Hello Dog!
```



```
You look like a...

Lakeland terrier with probability of 99.84%
Airedale terrier with probability of 0.10%

Hello Human!
```



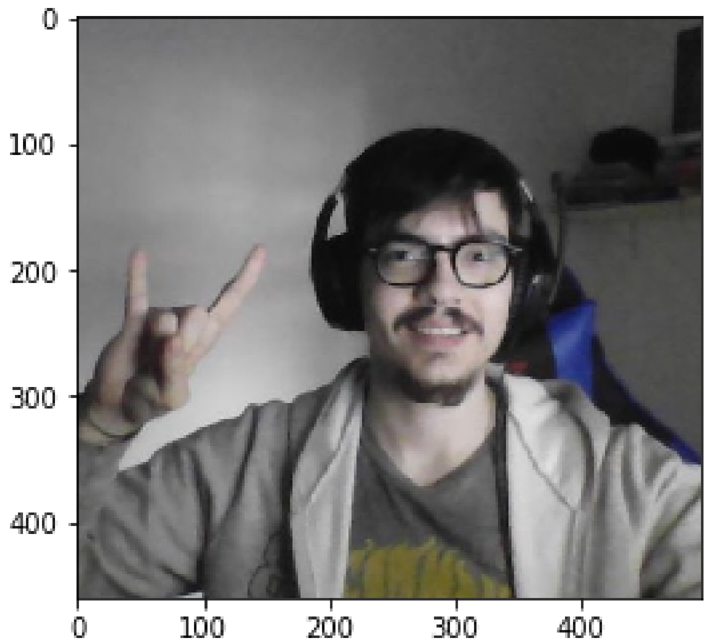

```
You look like a...

Irish water spaniel with probability of 21.29%
Dachshund with probability of 12.11%

Hello Dog!
```



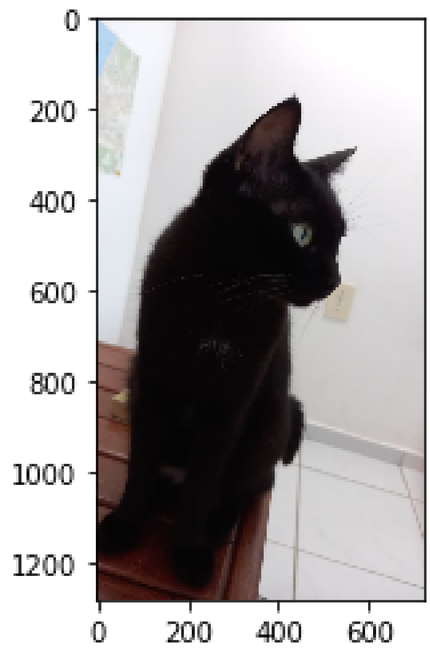
```
You look like a...

Pharaoh hound with probability of 91.07%
Australian cattle dog with probability of 1.87%
```

```
In [120]: # Lets test other pictures
          import glob

          for file in np.hstack(glob.glob('custom_images/*.*')):
              run_app(file)
```

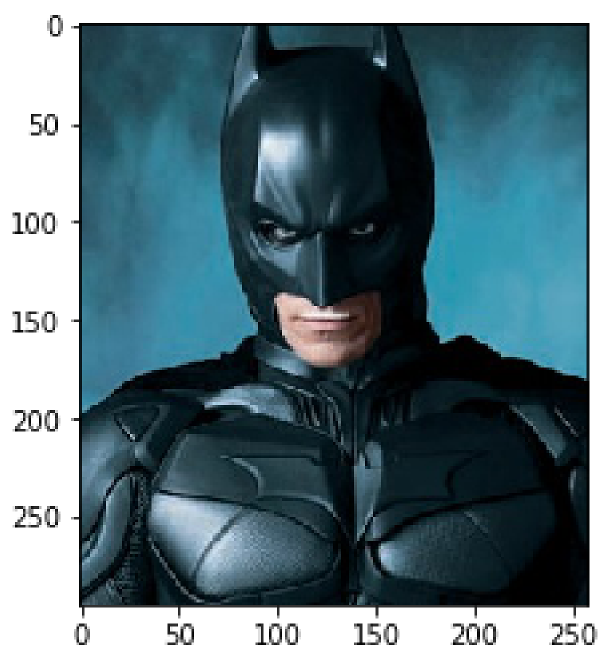
```
Hello Human!
```



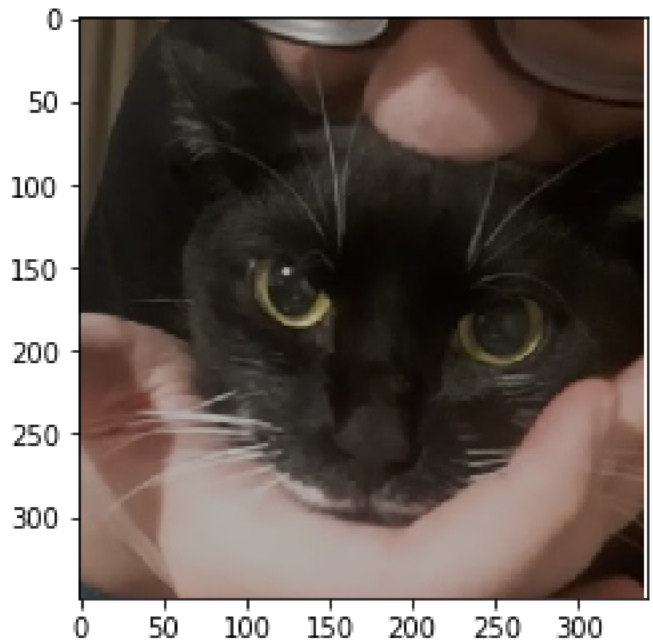

```
You look like a...

Dachshund with probability of 47.93%
English toy spaniel with probability of 13.73%

Hello Stranger!
```



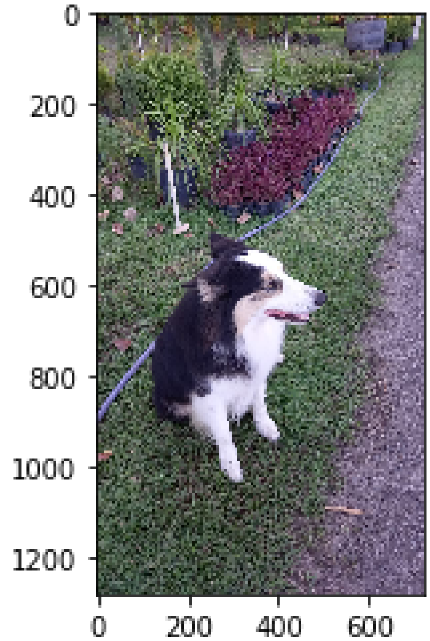

```
You look like a...

Dachshund with probability of 18.60%
Cardigan welsh corgi with probability of 11.73%

Hello Stranger!
```



```
You look like a...

Irish water spaniel with probability of 27.82%
Akita with probability of 16.74%

Hello Stranger!
```



```
You look like a...

Dachshund with probability of 40.09%
Affenpinscher with probability of 20.73%

Hello Dog!
```




```
You look like a...

Collie with probability of 56.52%
Border collie with probability of 15.58%
```

In [ ]: