dog_app

May 16, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

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

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

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

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

Step 1: Detect Humans

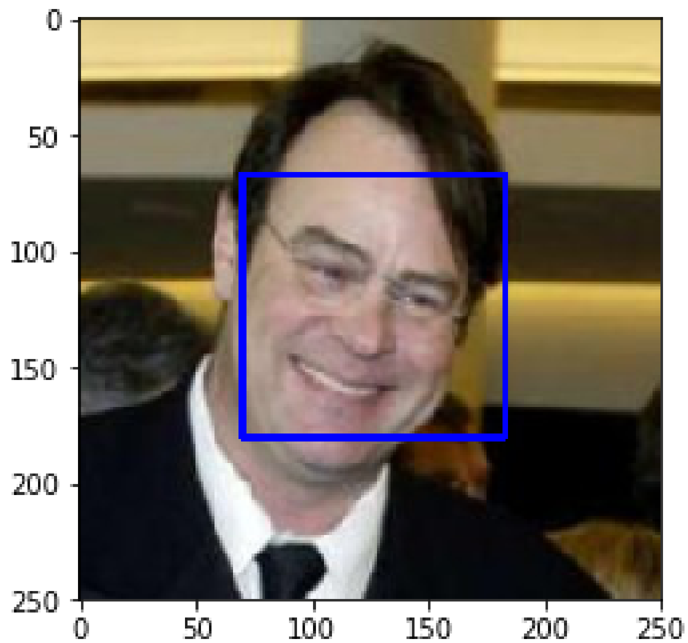
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

# load color (BGR) image
img = cv2.imread(human_files[0])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))
```

```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 [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: 98% of the first 100 images of the human files are detected as humans, while 17% of the first 100 images of the dogs are detected as humans.

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

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

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

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

```
In [11]: human_test = 0
         dog_test = 0

         for file in human_files_short:
             human_test += face_detector(file)

         for file in dog_files_short:
             dog_test += face_detector(file)

         print('The human test accuracy is: %s' % (human_test/len(human_files_short)))
         print('The dog test accuracy is: %s' % (dog_test/len(dog_files_short)))
```

```
The human test accuracy is: 0.98
The dog test accuracy is: 0.17
```

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

```
In [6]: ### (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 [00:08<00:00, 66418722.95it/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
# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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
    '''

    # Load the Image as an RGB
    image = Image.open(img_path).convert('RGB')

    # Normalize Data, transforming according to the RGB mean and std
    transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                         std=[0.229, 0.224, 0.225])])

    # Apply Transformation
    image = transform(image)

    # Adjusting the tensor dimensions to (1, length)
    image = image.unsqueeze(0)

    if use_cuda:
        image = image.cuda()

    # Returns class with the highest predcition probability of VGG16
    prediction = VGG16(image)
    prediction = prediction.data.argmax()

    return prediction
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [7]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            # evaluates whether we are in one of the dog classes
            class_index = VGG16_predict(img_path)
            if class_index >150 and class_index < 269:
                return True
            else:
                return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: In 1% of the Human images dogs were detected, whilst 99% of the Dog images were in fact detected as dogs.

```
In [10]: human_detected = 0
         dog_detected = 0

         for i in human_files_short:
             if dog_detector(i) == True:
                 human_detected += 1

         for i in dog_files_short:
             if dog_detector(i) == True:
                 dog_detected += 1

         print('Percentage that humans were detected as dogs: %s' % (human_detected/len(human_fi
         print('Percentage that dogs were detected as dogs: %s' % (dog_detected/len(dog_files_sh
```

```
Percentage that humans were detected as dogs: 0.01
Percentage that dogs were detected as dogs: 0.98
```

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

```
In [11]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador	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 [51]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms

         # Load for Validation Sampler
         from torch.utils.data.sampler import SubsetRandomSampler
```

```
# number of subprocesses to use for data loading
num_workers = 2
# how many samples per batch to load
batch_size = 16

# Transform images and then convert data to a normalized torch.FloatTensor, one set wit
transform = transforms.Compose([transforms.Resize(224),
                                transforms.CenterCrop((224,224)),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(10),
                                transforms.RandomResizedCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                     std=[0.229, 0.224, 0.225])])

# load the datasets
train_ds = datasets.ImageFolder('/data/dog_images/train', transform = transform)
valid_ds = datasets.ImageFolder('/data/dog_images/valid', transform = transform)
test_ds = datasets.ImageFolder('/data/dog_images/test', transform = transform)

# prepare data loaders from the respective dataset
train_data = torch.utils.data.DataLoader(train_ds, batch_size=batch_size, shuffle=True,
                                         num_workers=num_workers)
valid_data = torch.utils.data.DataLoader(valid_ds, batch_size=batch_size, num_workers=n
test_data = torch.utils.data.DataLoader(test_ds, batch_size=batch_size, num_workers=num
```

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 transform code resizes the images to 224x224. Then it augments the data by random flips, rotation and additional crops. Additionally data gets shuffled in the train data set.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [52]: import torch.nn as nn
         import torch.nn.functional as F

         # change droprate for alternative results
         droprate = 0.25

         # define the CNN architecture
         class Net(nn.Module):
             def __init__(self):
                 super(Net, self).__init__()
```

```
        # Convolutional Layers
        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)

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

        # linear layers (64 * 28 * 28) to 500 to 133
        self.fc1 = nn.Linear(64*28*28, 500)
        self.fc2 = nn.Linear(500, 133)

        # dropout layer with rate to be defined at the top
        self.dropout = nn.Dropout(droprate)

        # Batch Normalization has been proven to be a valuable technique in order to sp
        self.batch = nn.BatchNorm1d(500)

    def forward(self, x):
        # Forward behavior Convolutional Layer + Activation + Max pool + Dropout, altog
        x = self.pool(F.relu(self.conv1(x)))
        x = self.dropout(x)
        x = self.pool(F.relu(self.conv2(x)))
        x = self.dropout(x)
        x = self.pool(F.relu(self.conv3(x)))
        x = self.dropout(x)

        # Adapting the Output to th FCN, flatten the input tensor
        x = x.view(-1,64*28*28)

        # add 1st hidden layer, with relu activation function
        x = F.relu(self.batch(self.fc1(x)))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, without activation function in order to apply the Cross
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()
print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
  (batch): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

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

Answer: I used the techniques describes in the previous lessons. I did additional research online to understand other peoples' approaches.

The result are three convolutional layers, which expand from 3 (RGB depth) to eventually 64 depth. Additionally Max Pooling is applied after each Convolutional Layer as well as dropouts.

The Fully connected network comprises only of two layers, the first one to scale down and the second one to match the 133 potential classes.

Size we only focus on dogs, a simple, straightforward solution can be used as we can focus on shapes, forms and colors with those 3 CVLs.

After multiple trials, I eventually trained it for 45 epochs.

The result is a solid 11% accuracy.

P.S. Along the working on this notebook, I adjusted the valid_loss_min multiple times to fit the learnings - I always trained for some epochs and then reloaded.

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 [53]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)
```

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

```
# define the training epochs
n_epochs = 15

def train(n_epochs, train_data, valid_data, model, optimizer, criterion, use_cuda, save
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = 4.058242 #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(train_data):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()

            # clear gradients of the optimized variables
            optimizer.zero_grad()
            # forward pass
            output = model(data)
            # calculate batch loss
            loss = criterion(output, target)
            # backward pass
            loss.backward()
            # update weights
            optimizer.step()
            # update training loss
            train_loss += loss.item()*data.size(0)

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(valid_data):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss += loss.item() * data.size(0)
```

```
                # calculate average losses
                train_loss = train_loss/len(train_data.dataset)
                valid_loss = valid_loss/len(valid_data.dataset)

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

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

            # return trained model
            return model
```

In [55]:
```
# train the model
model_scratch = train(n_epochs, train_data, valid_data, model_scratch,
                      optimizer_scratch, criterion_scratch, use_cuda, 'model_scratch.pt'
```

```
Epoch: 1         Training Loss: 4.801039         Validation Loss: 4.859694
Validation loss decreased (inf --> 4.859694).  Saving model ...
Epoch: 2         Training Loss: 4.623906         Validation Loss: 4.789952
Validation loss decreased (4.859694 --> 4.789952).  Saving model ...
Epoch: 3         Training Loss: 4.534249         Validation Loss: 4.788925
Validation loss decreased (4.789952 --> 4.788925).  Saving model ...
Epoch: 4         Training Loss: 4.476868         Validation Loss: 4.823198
Epoch: 5         Training Loss: 4.421346         Validation Loss: 4.752300
Validation loss decreased (4.788925 --> 4.752300).  Saving model ...
Epoch: 6         Training Loss: 4.378138         Validation Loss: 4.662072
Validation loss decreased (4.752300 --> 4.662072).  Saving model ...
Epoch: 7         Training Loss: 4.347228         Validation Loss: 4.690437
Epoch: 8         Training Loss: 4.296660         Validation Loss: 4.706581
Epoch: 9         Training Loss: 4.272903         Validation Loss: 4.623295
Validation loss decreased (4.662072 --> 4.623295).  Saving model ...
Epoch: 10         Training Loss: 4.239881         Validation Loss: 4.567139
Validation loss decreased (4.623295 --> 4.567139).  Saving model ...
Epoch: 11         Training Loss: 4.213519         Validation Loss: 4.462849
Validation loss decreased (4.567139 --> 4.462849).  Saving model ...
Epoch: 12         Training Loss: 4.174609         Validation Loss: 4.594954
Epoch: 13         Training Loss: 4.152221         Validation Loss: 4.431690
Validation loss decreased (4.462849 --> 4.431690).  Saving model ...
Epoch: 14         Training Loss: 4.103482         Validation Loss: 4.566440
Epoch: 15         Training Loss: 4.076334         Validation Loss: 4.389106
Validation loss decreased (4.431690 --> 4.389106).  Saving model ...
```

In [57]:
```
# load the model and train the model again
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
model_scratch = train(n_epochs, train_data, valid_data, model_scratch,
                      optimizer_scratch, criterion_scratch, use_cuda, 'model_scratch.pt'
```

```
Epoch: 1        Training Loss: 4.046828         Validation Loss: 4.349996
Validation loss decreased (4.389106 --> 4.349996).  Saving model ...
Epoch: 2        Training Loss: 4.012627         Validation Loss: 4.272949
Validation loss decreased (4.349996 --> 4.272949).  Saving model ...
Epoch: 3        Training Loss: 3.992673         Validation Loss: 4.408520
Epoch: 4        Training Loss: 3.927700         Validation Loss: 4.316044
Epoch: 5        Training Loss: 3.921840         Validation Loss: 4.324195
Epoch: 6        Training Loss: 3.864018         Validation Loss: 4.292810
Epoch: 7        Training Loss: 3.861463         Validation Loss: 4.247597
Validation loss decreased (4.272949 --> 4.247597).  Saving model ...
Epoch: 8        Training Loss: 3.820595         Validation Loss: 4.295908
Epoch: 9        Training Loss: 3.814387         Validation Loss: 4.287608
Epoch: 10        Training Loss: 3.768156         Validation Loss: 4.137185
Validation loss decreased (4.247597 --> 4.137185).  Saving model ...
Epoch: 11        Training Loss: 3.753731         Validation Loss: 4.358924
Epoch: 12        Training Loss: 3.750687         Validation Loss: 4.148303
Epoch: 13        Training Loss: 3.710351         Validation Loss: 4.080996
Validation loss decreased (4.137185 --> 4.080996).  Saving model ...
Epoch: 14        Training Loss: 3.696744         Validation Loss: 4.062928
Validation loss decreased (4.080996 --> 4.062928).  Saving model ...
Epoch: 15        Training Loss: 3.658099         Validation Loss: 4.058242
Validation loss decreased (4.062928 --> 4.058242).  Saving model ...
```

In [62]:
```
# load the model and train the model again
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
model_scratch = train(n_epochs, train_data, valid_data, model_scratch,
                      optimizer_scratch, criterion_scratch, use_cuda, 'model_scratch.pt'
```

```
Epoch: 1        Training Loss: 3.643860         Validation Loss: 4.106158
Epoch: 2        Training Loss: 3.642086         Validation Loss: 4.066299
Epoch: 3        Training Loss: 3.617544         Validation Loss: 4.107358
Epoch: 4        Training Loss: 3.594912         Validation Loss: 3.953319
Validation loss decreased (4.058242 --> 3.953319).  Saving model ...
Epoch: 5        Training Loss: 3.565135         Validation Loss: 4.049259
Epoch: 6        Training Loss: 3.562608         Validation Loss: 4.020075
Epoch: 7        Training Loss: 3.522806         Validation Loss: 4.232254
Epoch: 8        Training Loss: 3.514754         Validation Loss: 4.007078
Epoch: 9        Training Loss: 3.509571         Validation Loss: 4.214652
Epoch: 10        Training Loss: 3.496914         Validation Loss: 3.963474
Epoch: 11        Training Loss: 3.461641         Validation Loss: 3.906111
Validation loss decreased (3.953319 --> 3.906111).  Saving model ...
Epoch: 12        Training Loss: 3.443540         Validation Loss: 3.935689
```

```
Epoch: 13         Training Loss: 3.446780        Validation Loss: 3.940138
Epoch: 14         Training Loss: 3.399461        Validation Loss: 3.871397
Validation loss decreased (3.906111 --> 3.871397).  Saving model ...
Epoch: 15         Training Loss: 3.396348        Validation Loss: 4.136131
```

```
In [63]: # Load the best model in order to test it
         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 [14]: def test(loaders, model, criterion, use_cuda):

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

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders):
                 # 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 [64]: # call test function
         test(test_data, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.867385
```

```
Test Accuracy: 11% (99/836)
```

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 [45]: # The same data loaders as before with a slightly adjusted transforms step
         import os
         from torchvision import datasets
         import torchvision.transforms as transforms

         # Load for Validation Sampler
         from torch.utils.data.sampler import SubsetRandomSampler

         # number of subprocesses to use for data loading
         num_workers = 0
         # how many samples per batch to load
         batch_size = 20

         # Transform images and then convert data to a normalized torch.FloatTensor, one set wit
         transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                         transforms.RandomHorizontalFlip(),
                                         transforms.RandomRotation(30),
                                         transforms.ToTensor(),
                                         transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                              std=[0.229, 0.224, 0.225])])

         test_transform = transforms.Compose([transforms.Resize((224,224)),
                                              transforms.ToTensor(),
                                              transforms.Normalize((0.485, 0.456, 0.406), (0.229

         # load the datasets
         train_ds = datasets.ImageFolder('/data/dog_images/train', transform = transform)
         valid_ds = datasets.ImageFolder('/data/dog_images/valid', transform = test_transform)
         test_ds = datasets.ImageFolder('/data/dog_images/test', transform = test_transform)

         # prepare data loaders from the respective dataset
         train_data = torch.utils.data.DataLoader(train_ds, batch_size=batch_size, shuffle=True,
```

```
                                                   num_workers=num_workers)
valid_data = torch.utils.data.DataLoader(valid_ds, batch_size=batch_size, num_workers=n
test_data = torch.utils.data.DataLoader(test_ds, batch_size=batch_size, num_workers=num
```

```
train_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                      transforms.RandomRotation(30),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.485, 0.456, 0.406), (0.22
```

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 [46]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.vgg19(pretrained=True)

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

In [17]:
```
print(model_transfer)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

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

Answer: I am loading in the VGG19 Architecture - I looked at the torchvision.models page to see which kind of model might make sense and has low errors. Additionally, I asked in the Udaycity, as I could not get it above 60%.

Eventually, I adapted to an approach which adapts the final layer, and selects the weights for the entire classifier section instead of the final layer, and this helped me to get across to an astonishing 81%.

In [47]:
```python
# Keeping the Weights of the network
for param in model_transfer.features.parameters():
    param.requires_grad = False

# Replacing the fully conneced layer at the end with one which only goes to 133 classes
model_transfer.classifier[6]=nn.Linear(4096,133)

# Using the GPU
```

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

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 [48]: import torch.optim as optim
         criterion_transfer = nn.CrossEntropyLoss()
         # Applying the optimzer to the entire classifier section, after trial and error this le
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [49]: # train the model for 10 epochs (less would have worked well as well)
         model_transfer = train(10, train_data, valid_data, model_transfer,
                                optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.
```

```
Epoch: 1        Training Loss: 4.445924        Validation Loss: 3.308066
Validation loss decreased (inf --> 3.308066).  Saving model ...
Epoch: 2        Training Loss: 3.190241        Validation Loss: 1.742554
Validation loss decreased (3.308066 --> 1.742554).  Saving model ...
Epoch: 3        Training Loss: 2.369977        Validation Loss: 1.086239
Validation loss decreased (1.742554 --> 1.086239).  Saving model ...
Epoch: 4        Training Loss: 1.997928        Validation Loss: 0.833174
Validation loss decreased (1.086239 --> 0.833174).  Saving model ...
Epoch: 5        Training Loss: 1.783708        Validation Loss: 0.710707
Validation loss decreased (0.833174 --> 0.710707).  Saving model ...
Epoch: 6        Training Loss: 1.657915        Validation Loss: 0.635891
Validation loss decreased (0.710707 --> 0.635891).  Saving model ...
Epoch: 7        Training Loss: 1.572780        Validation Loss: 0.593238
Validation loss decreased (0.635891 --> 0.593238).  Saving model ...
Epoch: 8        Training Loss: 1.513424        Validation Loss: 0.575640
Validation loss decreased (0.593238 --> 0.575640).  Saving model ...
Epoch: 9        Training Loss: 1.470306        Validation Loss: 0.550254
Validation loss decreased (0.575640 --> 0.550254).  Saving model ...
Epoch: 10        Training Loss: 1.444315        Validation Loss: 0.529602
Validation loss decreased (0.550254 --> 0.529602).  Saving model ...
```

1.1.16 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

```
In [50]: # load the model and test
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         test(test_data, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.559400


Test Accuracy: 81% (684/836)
```

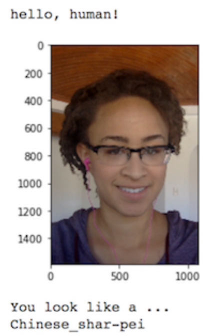
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [65]: # 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 train_ds.classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image = Image.open(img_path)
             #some simple transformation to fit the network structure
             transformations = transforms.Compose([transforms.CenterCrop(size=224),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize(mean=[0.485, 0.456, 0.40
                                                                        std=[0.229, 0.224, 0.225
             image_tensor = transformations(image)[:3,:,:].unsqueeze(0)

             # move model inputs to cuda, if GPU available
             if use_cuda:
                 image_tensor = image_tensor.cuda()

             # get sample outputs
             output = model_transfer(image_tensor)

             # convert output probabilities to predicted class
             _, preds_tensor = torch.max(output, 1)
             pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso

             return class_names[pred]
```

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.



Sample Human Output

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 [66]: # This algorithm uses the detetors from above
         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             dog = dog_detector(img_path)
             human = face_detector(img_path)
             # check if it is a dog
             if(dog==True):
                 print("Hi, dog!")
                 breed = predict_breed_transfer(img_path)
                 print(f"You look like a...{breed}")
                 image = Image.open(img_path)
                 plt.imshow(image)
                 plt.show()
             # or a human
             elif(human==True):
                 print("Hi, human!")
                 image = Image.open(img_path)
                 plt.imshow(image)
                 plt.show()
             # or neither - a cat maybe (jk)
             else:
                 print("You are neither human nor dog - chances are you are just a cat")
                 image = Image.open(img_path)
                 plt.imshow(image)
                 plt.show()
```

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

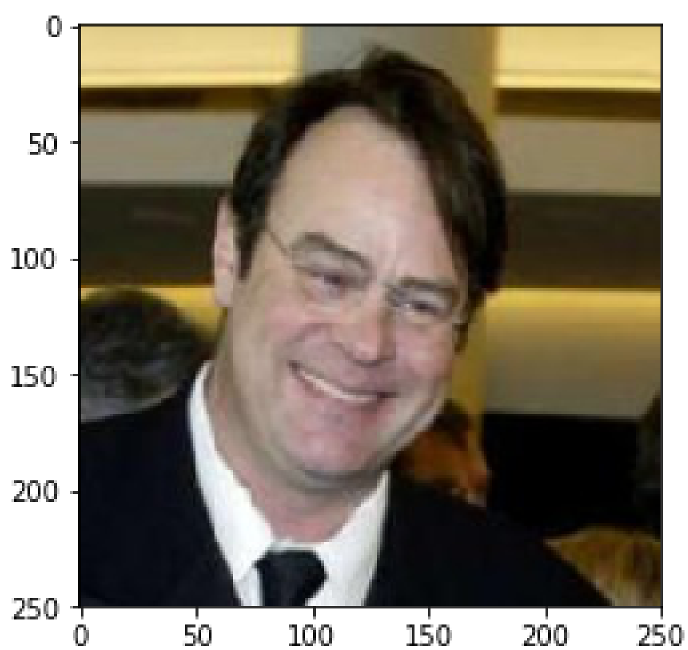
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

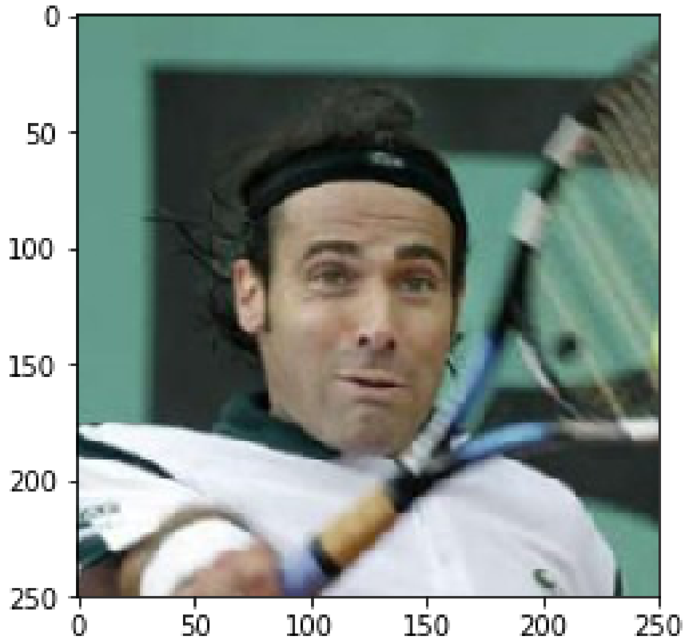
Answer: The algorithm works well - some pictures are not correctly identifiyed. Yet, I am positively surprised. Areas for impovement: - There could be a different way of preprocessing. I am not convinced that the crops I used are sufficient - The model could be trained on an even bigger data set and for longer in order to increase accuracy - We could also use the other classes VGG is trained on to classiy other objects and reduce the discrepancy

```
In [67]: ## 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
         for file in np.hstack((human_files[:3], dog_files[:3])):
             run_app(file)
```

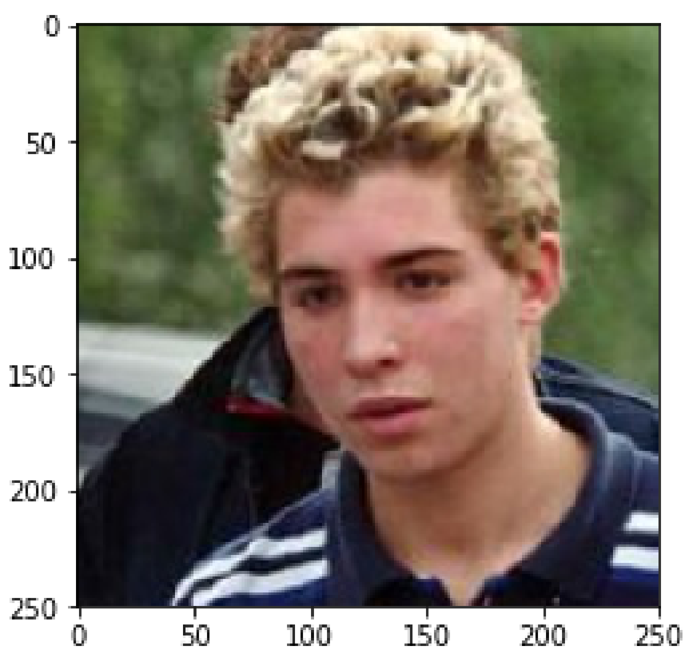
```
Hi, human!
```



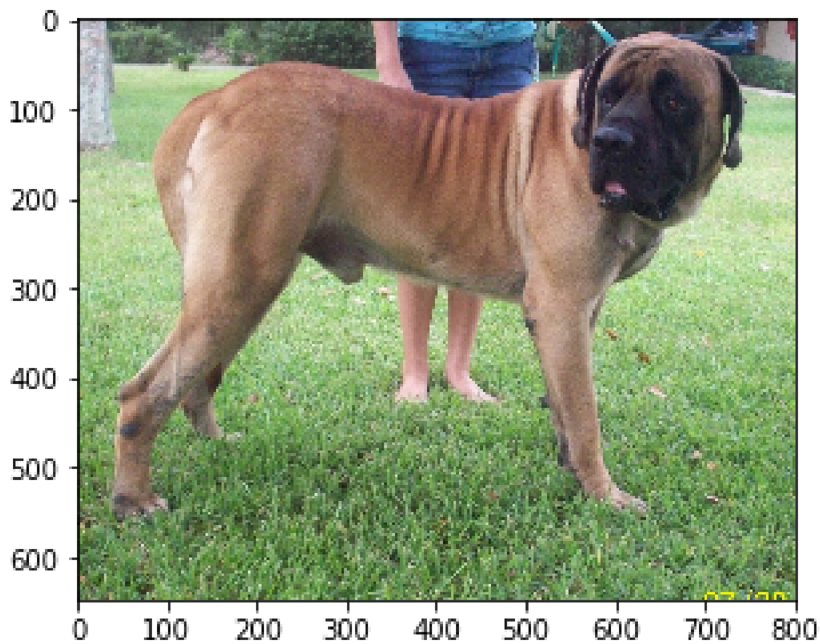
```
Hi, human!
```



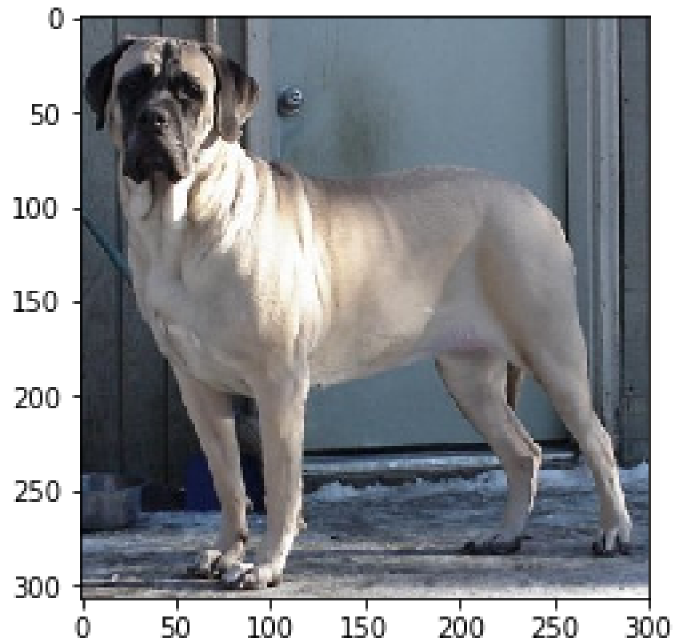

```
Hi, dog!
You look like a...Curly-coated retriever
```



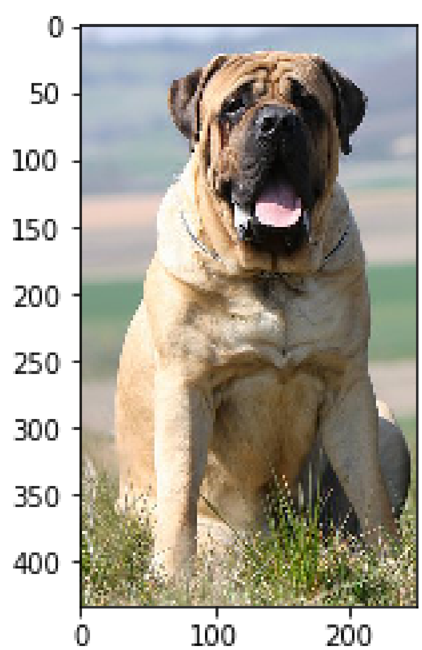
```
Hi, dog!
You look like a...Belgian malinois
```




```
Hi, dog!
You look like a...Mastiff
```

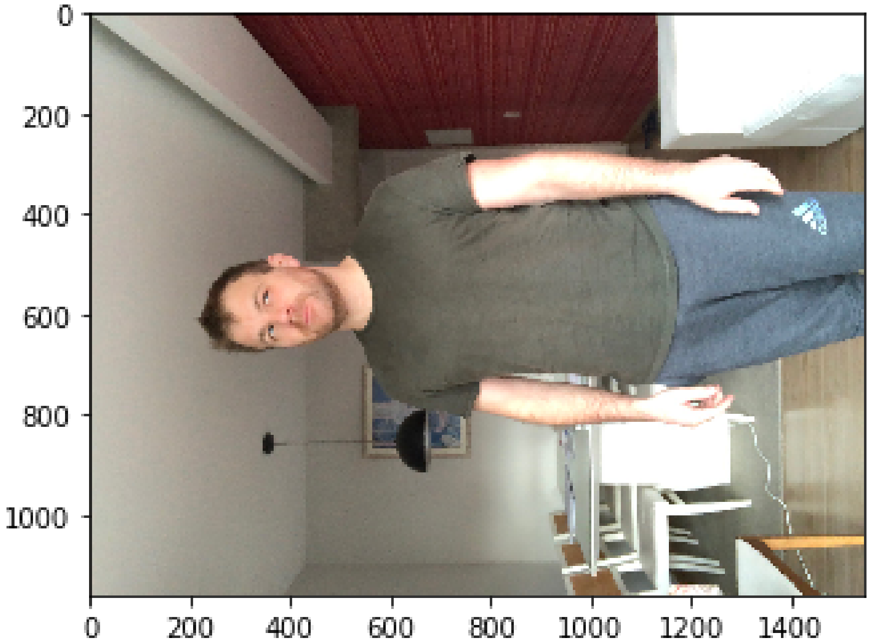



```
Hi, dog!
You look like a...Mastiff
```



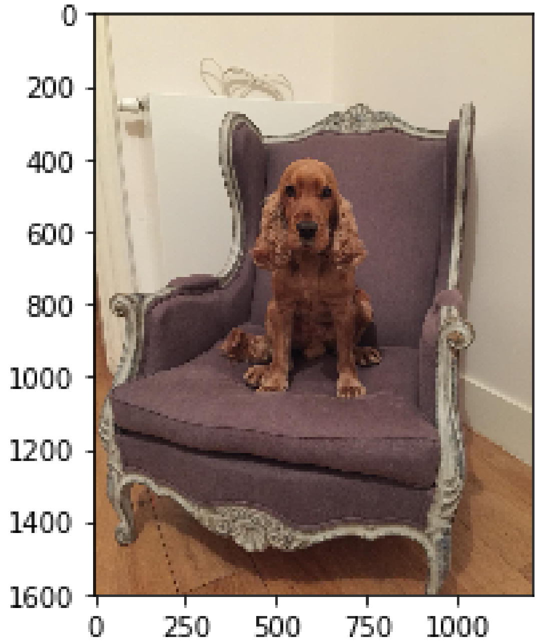
```
In [75]: run_app('lars.jpg')
```

```
Hi, human!
```



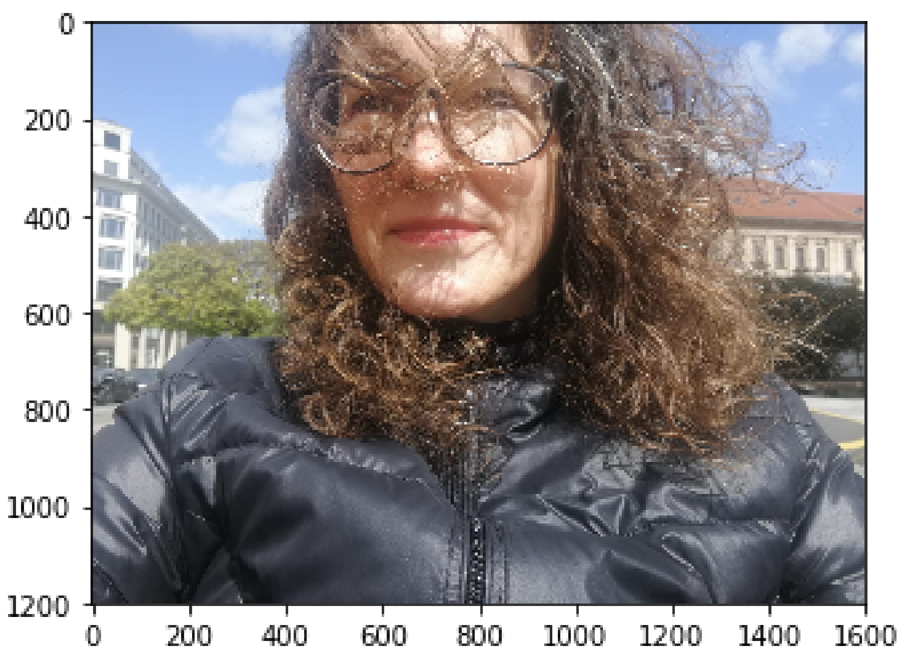

```
In [70]: run_app('monty.jpg')
```

```
Hi, dog!
You look like a...Irish setter
```



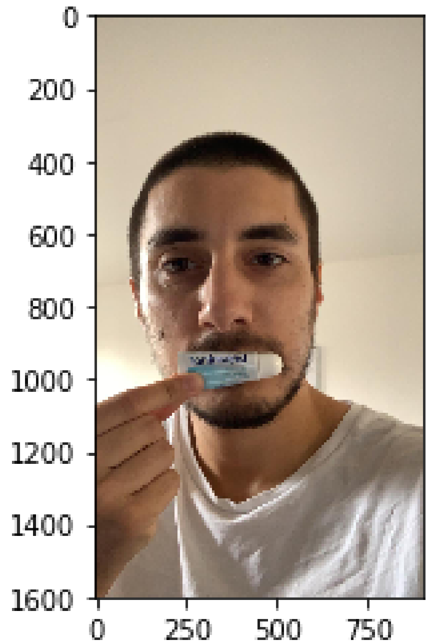

```
In [71]: run_app('anke.jpg')
```

```
You are neither human nor dog - chances are you are just a cat
```



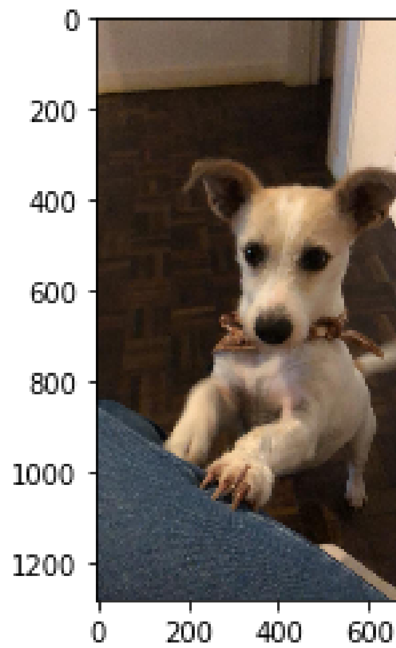
```
In [72]: run_app('aaron.jpg')
```

```
Hi, human!
```



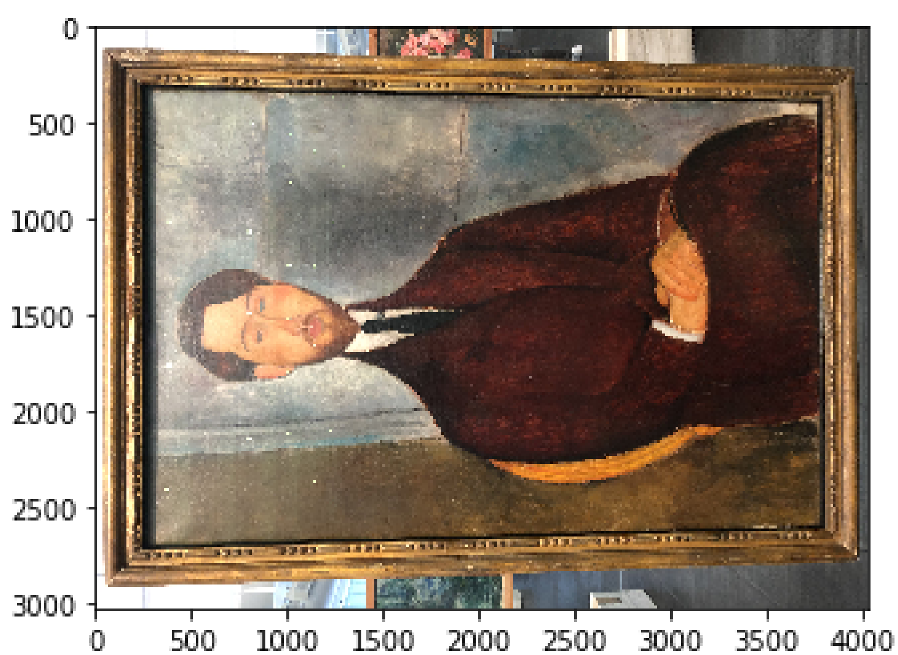
```
In [73]: run_app('hund.jpg')
```

```
Hi, dog!
You look like a...Old english sheepdog
```




```
In [74]: run_app('bild.jpg')
```

```
Hi, human!
```



```
In [ ]:
```