dog_app

June 16, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

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

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

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

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

Step 1: Detect Humans

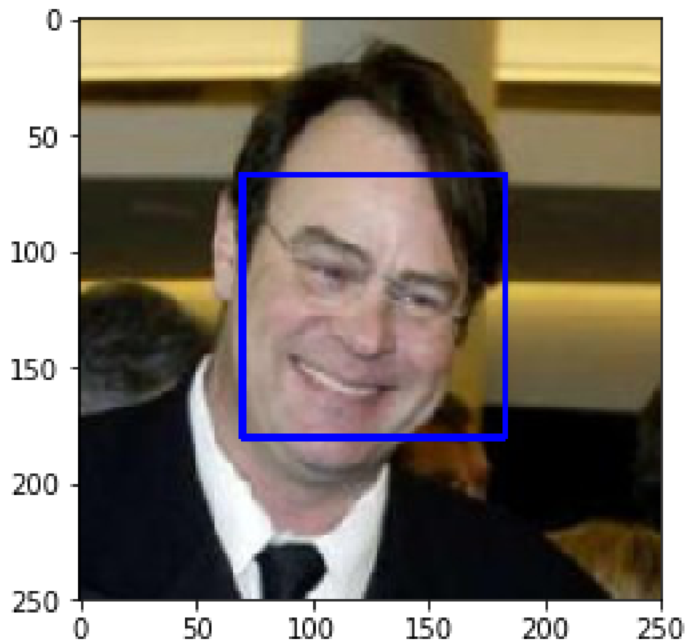
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

```
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

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

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

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

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

        human_files_as_human = np.average([face_detector(img) for img in tqdm(human_files_short)
        dog_files_as_human = np.average([face_detector(img) for img in tqdm(dog_files_short)])

        print('Accuracy human Avg : {}'.format(human_files_as_human))
        print('Dog Avg is detected as human : {}'.format(dog_files_as_human))
```

```
100%|| 100/100 [00:02<00:00, 35.95it/s]
100%|| 100/100 [00:29<00:00,  7.34it/s]

Accuracy human Avg : 0.98
Dog Avg is detected as human : 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 [25]: ### (Optional)
         ### TODO: Test performance of anotherface detection algorithm.
         ### Feel free to use as many code cells as needed.
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 100650407.03it/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



    # Open the image
    img = Image.open(img_path)

    # convert imge into Tensor usind (toTensor = transforms.ToTensor())

    # human face width=250 (jpg file), dog jpg file size is different>> resize into 250

    transform_pipeline = transforms.Compose([transforms.RandomResizedCrop(250),
                                             transforms.ToTensor()])

    img_tensor = transform_pipeline(img)
    img_tensor = img_tensor.unsqueeze(0)

    # from tensor to cuda
    if torch.cuda.is_available():
        img_tensor = img_tensor.cuda()

    prediction = VGG16(img_tensor)

    # from tensor to cpu (for cpu processing)
    if torch.cuda.is_available():
        prediction = prediction.cpu()

    index = prediction.data.numpy().argmax()

    return index # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

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

            # print index between 151 and 268 (inclusive)
            return (151 <= index and index <= 268) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

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

        human_files_as_human = np.average([dog_detector(img) for img in human_files_short])
        dog_files_as_human = np.average([dog_detector(img) for img in dog_files_short])

        print('Human Avg is detected as Dog : {}'.format(human_files_as_human))
        print('Accuracy Dog Avg : {}'.format(dog_files_as_human))
```

```
Human Avg is detected as Dog : 0.01
Accuracy Dog Avg : 0.77
```

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [9]: import os
        from torchvision import datasets
        from PIL import ImageFile
        import torchvision.transforms as transforms

        ImageFile.LOAD_TRUNCATED_IMAGES = True
```

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



# Pre-process train dataset with augmentation

train_transform = transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])



# Pre-process vaild/test dataset without augmentation
transform_pipline = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])


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

batch_size = 10
num_workers = 0

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

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

test_loader = torch.utils.data.DataLoader(test_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=False)
loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
```

```
        'test': test_loader
    }
```

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:

- **How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?**

  ```
  I have use (RandomResizedCrop) with scaled size into 224
  ```

- **Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?**

  ```
  Yes, by Convert image file to tensor
  ```

1.1.8 (IMPLEMENTATION) Model Architecture

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

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

         # define the CNN architecture

         dog_classes = 133 # total dog class

         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, 32, 3, padding=1)
                 self.norm2d1 = nn.BatchNorm2d(32)
                 self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

                 size_linear_layer = 500

                 # linear layer (128 * 28 * 28 ==> 500)
                 self.fc1 = nn.Linear(128 * 28 * 28, size_linear_layer)
                 self.fc2 = nn.Linear(size_linear_layer, dog_classes)
```

```
    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.norm2d1(self.conv1(x))))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        # image input
        x = x.view(-1, 128 * 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

#-#-# You so 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 = model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (norm2d1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, 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=100352, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
)
```

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

Answer:

1. Conv 1: layer, activation, pooling

   -(conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
   -Activation Fun: relu
   -(pooling): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

2. Conv 2: layer, activation, pooling

   -(conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
   -Activation Fun:relu
   -(pooling): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

3. Conv 3: layer, activation, pooling

-(conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
-Activation Fun:relu
-(pooling): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

4. (fc1): Linear(in_features=61504, out_features=500, bias=True)

5. (fc2): Linear(in_features=500, out_features=133, bias=True)

There are three conv layers, and maxpooling
I have add (Maxpool) when image passed cov layer to avoid iverfitting in each cov layer and to decrease the featrue mape.
in each cov layer, width and height are decreased as below:
(250,250) = input image size
-> (125,125)
-> (62,62)
-> (28,28)
The depth = 64 ==> i.e: the fully-connected layer size is 64 * 28 * 28.
Also, It is connected to last fully connected layer and the size=133 which is similar to the total classes of dog

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [11]: import torch.optim as optim

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

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

         if use_cuda:
             criterion_scratch.cuda()
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [12]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

             for epoch in range(1, n_epochs+1):
```

```
# initialize variables to monitor training and validation loss
train_loss = 0.0
valid_loss = 0.0

###################
# train the model #
###################
model.train()
for batch_idx, (data, target) in enumerate(loaders['train']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## find the loss and update the model parameters accordingly
    ## record the average training loss, using something like
    ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

    optimizer.zero_grad()

    # forward pass
    output = model(data)

    # calculate batch loss
    loss = criterion(output, target)

    # backward pass
    loss.backward()

    # parameter update
    optimizer.step()

    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)


######################
# validate the model #
######################
model.eval()
for batch_idx, (data, target) in enumerate(loaders['valid']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## update the average validation loss
    output = model(data)
    loss = criterion(output, target)
    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


# print training/validation statistics
```

```python
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch,
                train_loss,
                valid_loss
                ))

            ## TODO: save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                torch.save(model.state_dict(), save_path)
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'
                      .format(valid_loss_min, valid_loss))
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    model_scratch = train(35, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

    # load the model that got the best validation accuracy
    model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1 	Training Loss: 4.874071 	Validation Loss: 4.799191
Validation loss decreased (inf --> 4.799191).  Saving model ...
Epoch: 2 	Training Loss: 4.690374 	Validation Loss: 4.483826
Validation loss decreased (4.799191 --> 4.483826).  Saving model ...
Epoch: 3 	Training Loss: 4.579238 	Validation Loss: 4.438984
Validation loss decreased (4.483826 --> 4.438984).  Saving model ...
Epoch: 4 	Training Loss: 4.518841 	Validation Loss: 4.358006
Validation loss decreased (4.438984 --> 4.358006).  Saving model ...
Epoch: 5 	Training Loss: 4.451295 	Validation Loss: 4.277254
Validation loss decreased (4.358006 --> 4.277254).  Saving model ...
Epoch: 6 	Training Loss: 4.382032 	Validation Loss: 4.201300
Validation loss decreased (4.277254 --> 4.201300).  Saving model ...
Epoch: 7 	Training Loss: 4.324695 	Validation Loss: 4.123540
Validation loss decreased (4.201300 --> 4.123540).  Saving model ...
Epoch: 8 	Training Loss: 4.287301 	Validation Loss: 4.111712
Validation loss decreased (4.123540 --> 4.111712).  Saving model ...
Epoch: 9 	Training Loss: 4.219664 	Validation Loss: 4.080149
Validation loss decreased (4.111712 --> 4.080149).  Saving model ...
Epoch: 10 	Training Loss: 4.164010 	Validation Loss: 4.054220
Validation loss decreased (4.080149 --> 4.054220).  Saving model ...
Epoch: 11 	Training Loss: 4.142148 	Validation Loss: 3.987947
Validation loss decreased (4.054220 --> 3.987947).  Saving model ...
Epoch: 12 	Training Loss: 4.076387 	Validation Loss: 3.985136
Validation loss decreased (3.987947 --> 3.985136).  Saving model ...
```

```
Epoch: 13         Training Loss: 4.042631         Validation Loss: 3.866470
Validation loss decreased (3.985136 --> 3.866470).  Saving model ...
Epoch: 14         Training Loss: 3.996782         Validation Loss: 3.832814
Validation loss decreased (3.866470 --> 3.832814).  Saving model ...
Epoch: 15         Training Loss: 3.936854         Validation Loss: 3.856555
Epoch: 16         Training Loss: 3.895902         Validation Loss: 3.760493
Validation loss decreased (3.832814 --> 3.760493).  Saving model ...
Epoch: 17         Training Loss: 3.855852         Validation Loss: 3.739038
Validation loss decreased (3.760493 --> 3.739038).  Saving model ...
Epoch: 18         Training Loss: 3.793676         Validation Loss: 3.703358
Validation loss decreased (3.739038 --> 3.703358).  Saving model ...
Epoch: 19         Training Loss: 3.752026         Validation Loss: 3.742903
Epoch: 20         Training Loss: 3.721624         Validation Loss: 3.705671
Epoch: 21         Training Loss: 3.673020         Validation Loss: 3.700200
Validation loss decreased (3.703358 --> 3.700200).  Saving model ...
Epoch: 22         Training Loss: 3.646251         Validation Loss: 3.659706
Validation loss decreased (3.700200 --> 3.659706).  Saving model ...
Epoch: 23         Training Loss: 3.586033         Validation Loss: 3.621545
Validation loss decreased (3.659706 --> 3.621545).  Saving model ...
Epoch: 24         Training Loss: 3.528124         Validation Loss: 3.607066
Validation loss decreased (3.621545 --> 3.607066).  Saving model ...
Epoch: 25         Training Loss: 3.510861         Validation Loss: 3.567182
Validation loss decreased (3.607066 --> 3.567182).  Saving model ...
Epoch: 26         Training Loss: 3.444341         Validation Loss: 3.677048
Epoch: 27         Training Loss: 3.416143         Validation Loss: 3.571202
Epoch: 28         Training Loss: 3.351775         Validation Loss: 3.588123
Epoch: 29         Training Loss: 3.313012         Validation Loss: 3.512544
Validation loss decreased (3.567182 --> 3.512544).  Saving model ...
Epoch: 30         Training Loss: 3.256693         Validation Loss: 3.561528
Epoch: 31         Training Loss: 3.239824         Validation Loss: 3.580557
Epoch: 32         Training Loss: 3.190357         Validation Loss: 3.514172
Epoch: 33         Training Loss: 3.140689         Validation Loss: 3.451639
Validation loss decreased (3.512544 --> 3.451639).  Saving model ...
Epoch: 34         Training Loss: 3.098761         Validation Loss: 3.553817
Epoch: 35         Training Loss: 3.083761         Validation Loss: 3.499618
```

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 [13]: def test(loaders, model, criterion, use_cuda):

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

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

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

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

    # call test function
    test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.387951


Test Accuracy: 20% (171/836)
```

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
         loaders_transfer = loaders_scratch.copy()
```

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
         model_transfer = models.resnet50(pretrained=True)

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

         model_transfer.fc = nn.Linear(2048, 133, bias=True)

         fc_parameters = model_transfer.fc.parameters()

         for param in fc_parameters:
             param.requires_grad = True


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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 57652152.55it/s]
```

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

Answer:

For good image classification (ResNet) was chosen.

At the end, fully-connected layer is add pluse fully-connected layer with output of 133 (which represent the total dog calsses).

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 [16]: criterion_transfer =  nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(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 [17]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""

             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

             for epoch in range(1, n_epochs+1):

                 # initialize variables to monitor training and validation loss
                 train_loss = 0.0
                 valid_loss = 0.0

                 ###################
                 # train the model #
                 ###################

                 model.train()

                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

                     optimizer.zero_grad()

                     # forward pass
                     output = model(data)

                     # calculate batch loss
                     loss = criterion(output, target)

                     # backward pass
                     loss.backward()

                     # parameter update
                     optimizer.step()

                     train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)


                 ######################
                 # validate the model #
                 ######################
                 model.eval()
                 for batch_idx, (data, target) in enumerate(loaders['valid']):
```

```
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                output = model(data)
                loss = criterion(output, target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

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

            ## TODO: save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                torch.save(model.state_dict(), save_path)
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'
                      .format(valid_loss_min, valid_loss))
                valid_loss_min = valid_loss

        # return trained model
        return model

    n_epochs = 20


    # train the model
    # train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
    #                                            criterion_transfer, use_cuda, 'model_trans
    model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
                           use_cuda, 'model_transfer.pt')


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

```
Epoch: 1        Training Loss: 4.745024        Validation Loss: 4.443148
Validation loss decreased (inf --> 4.443148).  Saving model ...
Epoch: 2        Training Loss: 4.429244        Validation Loss: 4.044027
Validation loss decreased (4.443148 --> 4.044027).  Saving model ...
Epoch: 3        Training Loss: 4.148744        Validation Loss: 3.653620
Validation loss decreased (4.044027 --> 3.653620).  Saving model ...
Epoch: 4        Training Loss: 3.881971        Validation Loss: 3.302384
Validation loss decreased (3.653620 --> 3.302384).  Saving model ...
Epoch: 5        Training Loss: 3.641710        Validation Loss: 2.999373
Validation loss decreased (3.302384 --> 2.999373).  Saving model ...
```

```
Epoch: 6        Training Loss: 3.422979        Validation Loss: 2.757210
Validation loss decreased (2.999373 --> 2.757210).  Saving model ...
Epoch: 7        Training Loss: 3.236698        Validation Loss: 2.506831
Validation loss decreased (2.757210 --> 2.506831).  Saving model ...
Epoch: 8        Training Loss: 3.040642        Validation Loss: 2.304289
Validation loss decreased (2.506831 --> 2.304289).  Saving model ...
Epoch: 9        Training Loss: 2.893299        Validation Loss: 2.098266
Validation loss decreased (2.304289 --> 2.098266).  Saving model ...
Epoch: 10        Training Loss: 2.757908        Validation Loss: 1.949198
Validation loss decreased (2.098266 --> 1.949198).  Saving model ...
Epoch: 11        Training Loss: 2.626062        Validation Loss: 1.819421
Validation loss decreased (1.949198 --> 1.819421).  Saving model ...
Epoch: 12        Training Loss: 2.517177        Validation Loss: 1.724380
Validation loss decreased (1.819421 --> 1.724380).  Saving model ...
Epoch: 13        Training Loss: 2.420096        Validation Loss: 1.606547
Validation loss decreased (1.724380 --> 1.606547).  Saving model ...
Epoch: 14        Training Loss: 2.320333        Validation Loss: 1.463492
Validation loss decreased (1.606547 --> 1.463492).  Saving model ...
Epoch: 15        Training Loss: 2.243379        Validation Loss: 1.379712
Validation loss decreased (1.463492 --> 1.379712).  Saving model ...
Epoch: 16        Training Loss: 2.146862        Validation Loss: 1.328910
Validation loss decreased (1.379712 --> 1.328910).  Saving model ...
Epoch: 17        Training Loss: 2.077968        Validation Loss: 1.260870
Validation loss decreased (1.328910 --> 1.260870).  Saving model ...
Epoch: 18        Training Loss: 2.020319        Validation Loss: 1.200652
Validation loss decreased (1.260870 --> 1.200652).  Saving model ...
Epoch: 19        Training Loss: 1.956258        Validation Loss: 1.124372
Validation loss decreased (1.200652 --> 1.124372).  Saving model ...
Epoch: 20        Training Loss: 1.932868        Validation Loss: 1.108843
Validation loss decreased (1.124372 --> 1.108843).  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 [18]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.084870


Test Accuracy: 80% (673/836)
```

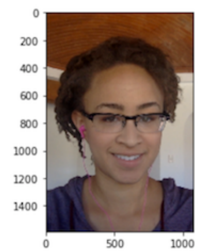
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [21]:
```python
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

from PIL import Image
import torchvision.transforms as transforms

data_transfer = loaders_transfer.copy()

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].dataset.cl

def predict_breed_transfer(img_path):
    global model_transfer
    global train_transform



# load the image and return the predicted breed
    image = Image.open(img_path).convert('RGB')




# Removing transparent, alpha
    image = train_transform(image)[:3,:,:].unsqueeze(0)

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

    model_transfer.eval()
    idx = torch.argmax(model_transfer(image))
    return class_names[idx]
```

In [22]:
```python
# For test predict_breed_transfer

for img_file in os.listdir("/data/dog_images/test/001.Affenpinscher/"):
    img_path = os.path.join('/data/dog_images/test/001.Affenpinscher/', img_file)
    predition = predict_breed_transfer(img_path)
    print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))
```

```
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00036.jpg,          predi
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00047.jpg,          predi
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00071.jpg,          predi
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00023.jpg,          predi
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00048.jpg,          predi
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00058.jpg,          predi
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00078.jpg,          predi
```



Sample Human Output

```
image_file_name: /data/dog_images/test/001.Affenpinscher/Affenpinscher_00003.jpg,          predi
```

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 [23]:
```python
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):

    ## handle cases for a human face, dog, and neither
    if face_detector(img_path) > 0:
        breed = predict_breed_transfer(img_path)
        print('Human / similar to dog breed is ' + breed)
    elif dog_detector(img_path):
        breed = predict_breed_transfer(img_path)
        print('Dog / dog breed is ' + breed)
    else:
        print('Not Dog, Not Human')
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

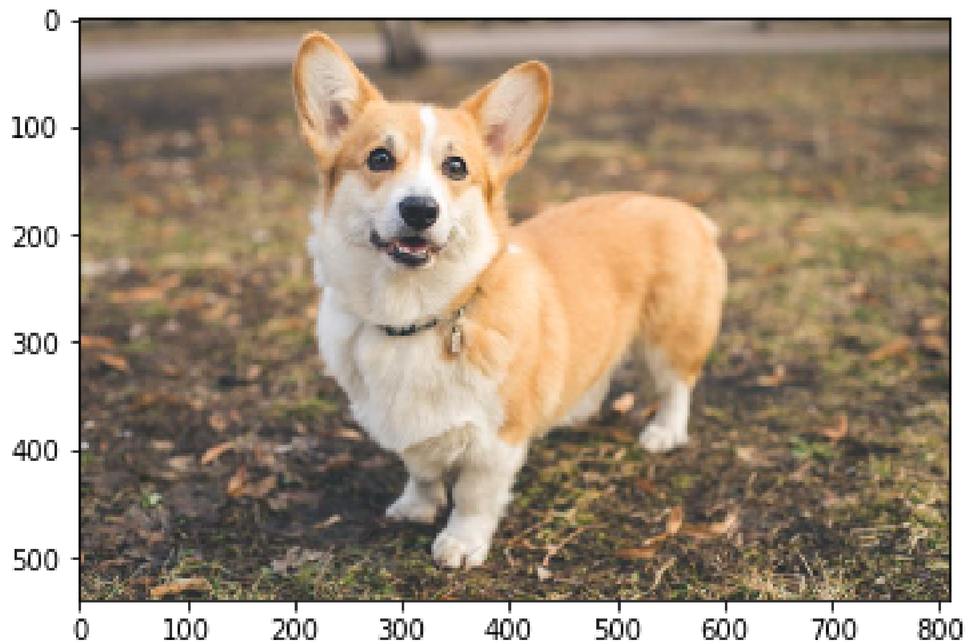
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

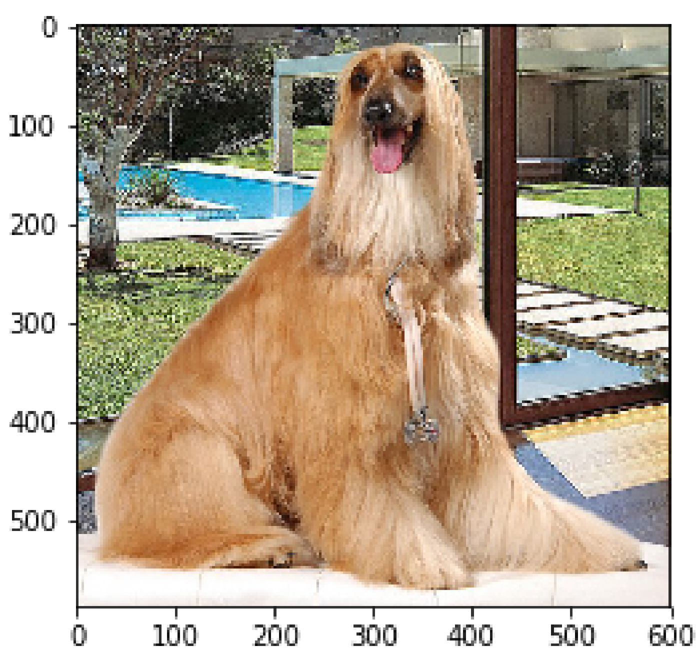
- Adding more breeds of dogs and/or increase human pictures
- Increase number of epochs (if we have great resource will help to discover)
- Playing with some parameters could help too, such as inputs and output values (maybe will affect)

```
In [24]: import os
         from PIL import Image


         ## 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.


         for img_file in os.listdir('/home/workspace/dog_project/new_images/'):
             img_path = os.path.join('/home/workspace/dog_project/new_images/', img_file)
             run_app(img_path)
             print(img_path)
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
```

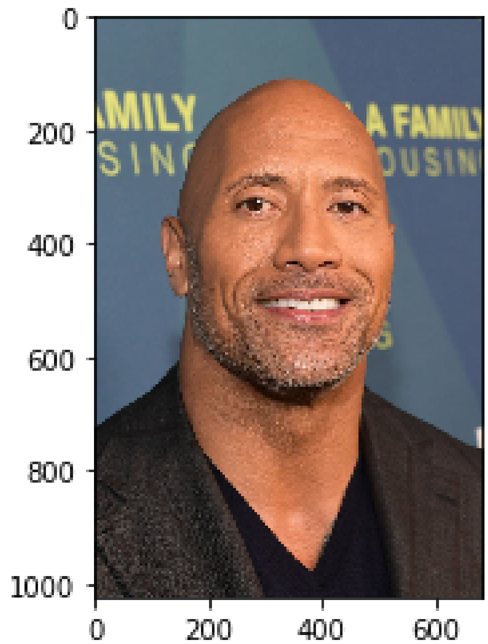
```
Dog / dog breed is Akita
/home/workspace/dog_project/new_images/ThinkstockPhotos-471884456.jpg
```



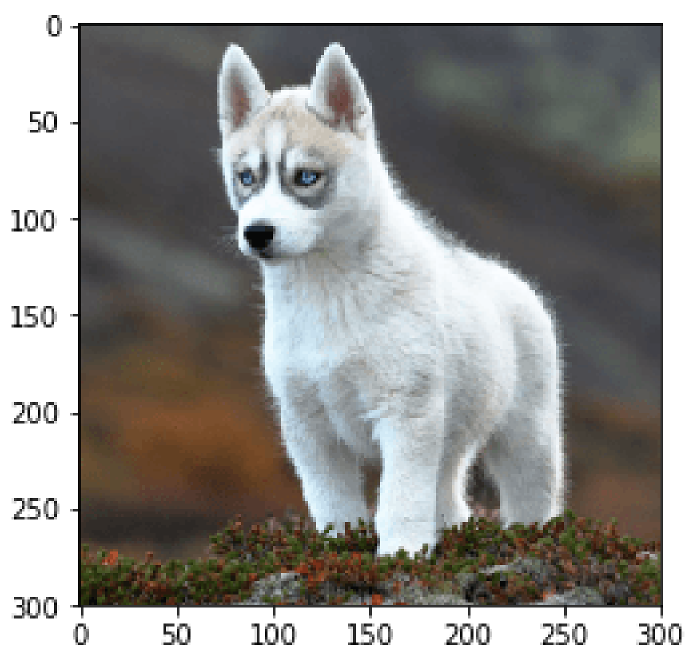
Dog / dog breed is Afghan hound
/home/workspace/dog_project/new_images/2. Afghan Hound.jpg



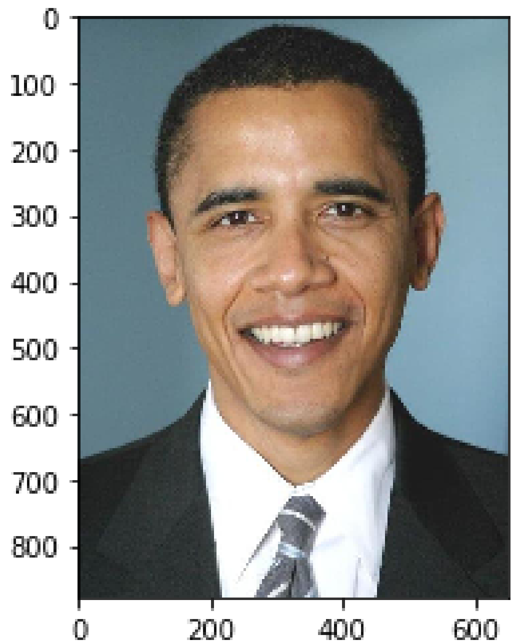
```
Human / similar to dog breed is Italian greyhound
/home/workspace/dog_project/new_images/sub-buzz-1096-1579879662-3.jpg
```



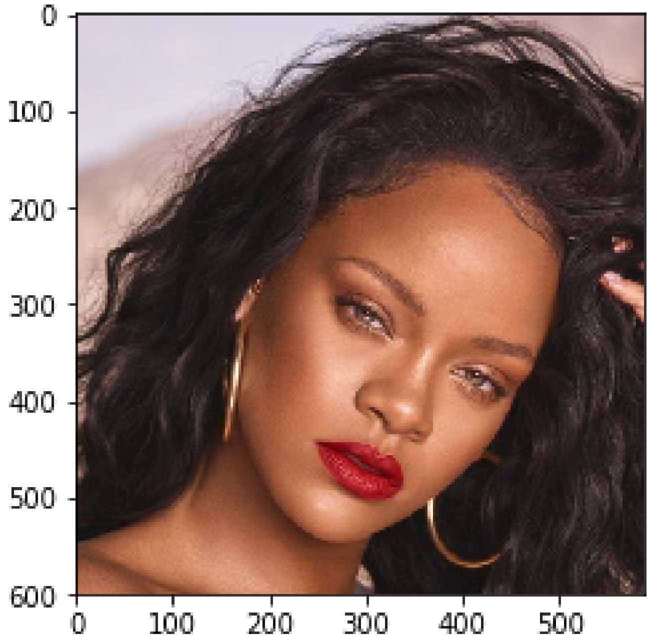

```
Dog / dog breed is Alaskan malamute
/home/workspace/dog_project/new_images/0_EK8EubWpGkk72RIT.png
```



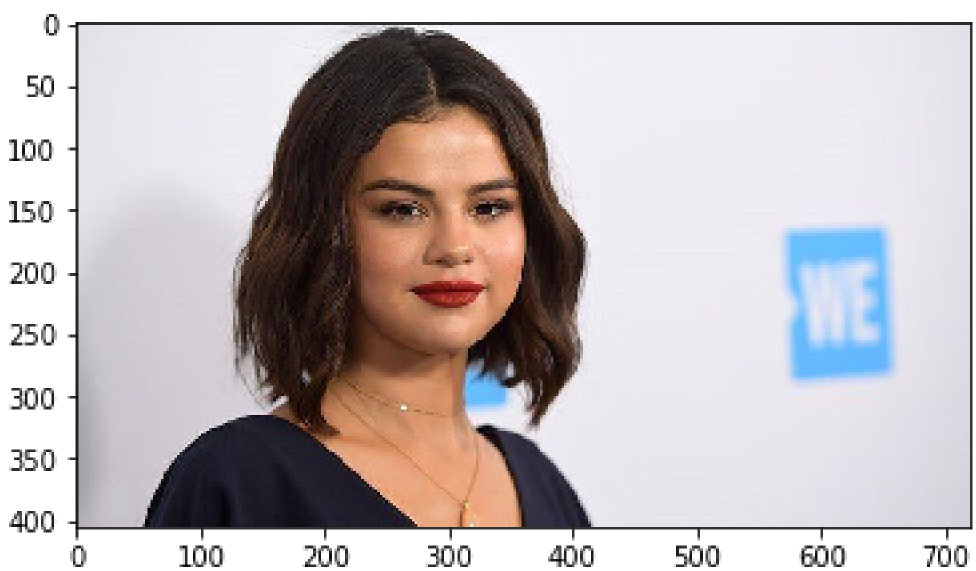
```
Human / similar to dog breed is Dogue de bordeaux
/home/workspace/dog_project/new_images/barack-obama-writers-photo-1.jpeg
```



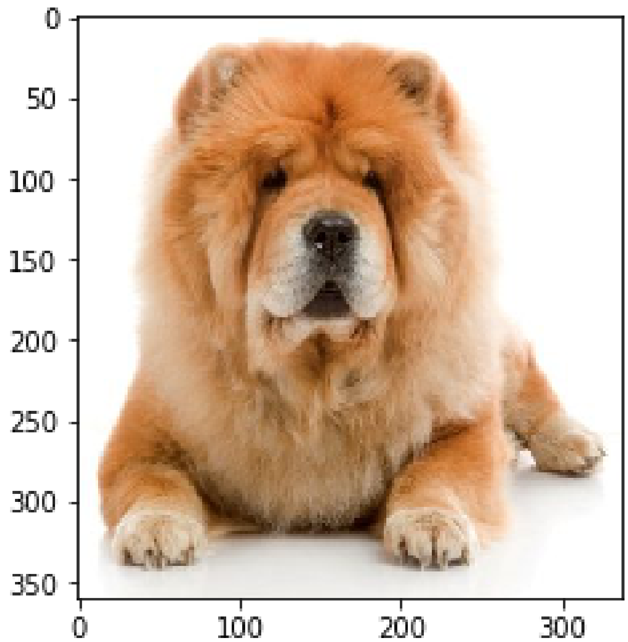
```
Dog / dog breed is Chinese crested
/home/workspace/dog_project/new_images/real-names-of-famous-people-1.jpg
```




```
Human / similar to dog breed is Maltese
/home/workspace/dog_project/new_images/famous-people-with-anxiety-rm-selena-gomez-722x406.jpg
```




```
Dog / dog breed is Chow chow
/home/workspace/dog_project/new_images/f75335ae6328ae3556cb3e8c5046bfc9.jpg
```



In [ ]: