dog_app

November 6, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

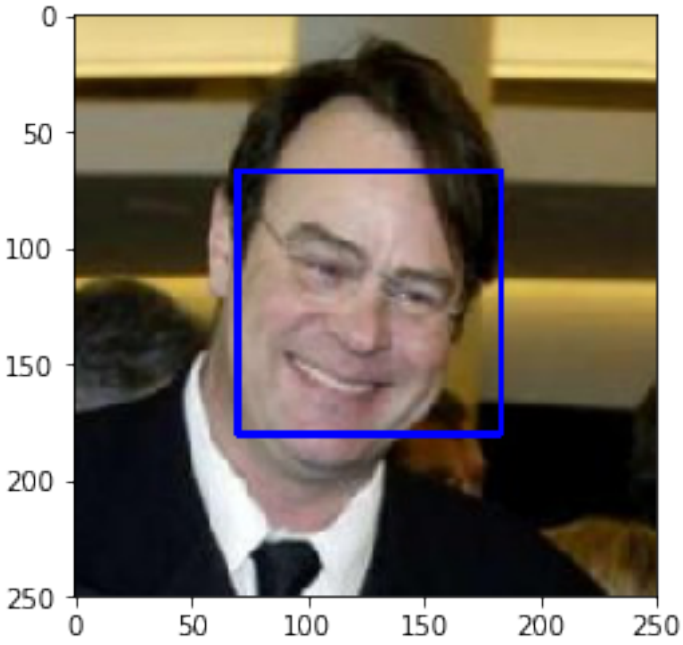
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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: 98% of the first 100 images in `human_files` have a detected human face and 17% of the first 100 images in `dog_files`.

```
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.

        faces_h =0
        faces_d = 0

        for path in human_files_short:
            if face_detector(path):
                faces_h += 1

        for path in dog_files_short:
            if face_detector(path):
                faces_d += 1

        print(faces_h)
        print(faces_d)
```

```
98
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 [5]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [6]: import torch
        import torchvision.models as models

        # 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, 102878569.31it/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 [7]: 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
            '''
            image = Image.open(img_path).convert('RGB')
            ## TODO: Complete the function.
            ## Load and pre-process an image from the given img_path
            in_transform = transforms.Compose([
                                transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406),
                                                     (0.229, 0.224, 0.225))])
            image = in_transform(image)
            image = image.unsqueeze(0)
            if use_cuda:
                image = image.cuda()
            ## Return the *index* of the predicted class for that image
            prediction = VGG16(image)
            _,indices = torch.max(prediction,1)

            return indices
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index = VGG16_predict(img_path)
            dog = False
            if (index >= 151) and (index <= 268):
                dog = True
            return dog # 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: 0% of the images in `human_files_short` have a detected dog and 100% of the images in `dog_files_short`.

```
In [10]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         dogs_h =0
         dogs_d = 0

         for path in human_files_short:
             if dog_detector(path):
                 dogs_h += 1

         for path in dog_files_short:
             if dog_detector(path):
                 dogs_d += 1

         print(dogs_h)
         print(dogs_d)
```

```
0
100
```

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

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 [12]: import os
         from torchvision import datasets

         from PIL import ImageFile
```

```
ImageFile.LOAD_TRUNCATED_IMAGES = True


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

# number of subprocesses to use for data loading
num_workers = 2
# how many samples per batch to load
batch_size = 20

# location of data
train_data_loc = '/data/dog_images/train'
test_data_loc = '/data/dog_images/test'
valid_data_loc = '/data/dog_images/valid'

transform_train = transforms.Compose([
                        transforms.Resize(256),
                        transforms.CenterCrop(224),
                        transforms.RandomHorizontalFlip(),
                        transforms.RandomRotation(20),
                        transforms.ToTensor(),
                        transforms.Normalize((0.485, 0.456, 0.406),
                                             (0.229, 0.224, 0.225))])

transform_test = transforms.Compose([
                        transforms.Resize(256),
                        transforms.CenterCrop(224),
                        transforms.ToTensor(),
                        transforms.Normalize((0.485, 0.456, 0.406),
                                             (0.229, 0.224, 0.225))])

train_data = datasets.ImageFolder(root=train_data_loc, transform=transform_train)
test_data = datasets.ImageFolder(root=test_data_loc, transform=transform_test)
valid_data = datasets.ImageFolder(root=valid_data_loc, transform=transform_test)

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=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers, shuffle=True)

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:

The images are first resized to 256 pixels and then croped around the center to 224 pixels. This is the same size as the VGG16 model takes as an input and it is not too large, so that computation time will hopefully not be too long.

To make the data more rotational and translational invariant, I applied a random horizontal flip and a random rotation by 20ř to the training data.

After the images are converted to tensors, the data is normalized.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [13]:
```
# get number of dog breeds to classify:
import os
len(os.listdir("/data/dog_images/train"))
```

Out[13]: 133

In [14]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 16, 5, padding=2)
        self.conv2 = nn.Conv2d(16, 32, 5, padding=2)
        self.conv3 = nn.Conv2d(32, 64, 5, padding=2)
        self.conv4 = nn.Conv2d(64, 128, 5, padding=2)
        self.pool = nn.MaxPool2d(2,2)
        self.fc1 = nn.Linear(14*14*128, 2000)
        self.fc2 = nn.Linear(2000,500)
        self.fc3 = nn.Linear(500,133)
        self.dropout = nn.Dropout(p=0.2)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = x.view(-1, 14*14*128)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.fc3(x)
```

```
            return x

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

        # instantiate the CNN
        model_scratch = Net()

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

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

Answer: After researching similar dog breed classifyers, I found that 3-4 convolutional layers are mostly combined with 2 to 3 fully connected layers. The more convolutional layers we include, the more complex patterns in color and shape a model can detect. After trying out simpler models, this gave me the best results: - 4 convolultions layers going from 3 (input) to 16 (first), to 32 (second), to 64 (third), and finally to 128 (fourth) filters; - the filters are applied to learn about high level features and increase the depth of the input tensor; - each convolutional layer is activated by a relu function; - after each convolutional layer a max pooling (2,2) is applied; - the max pooling is added to reduce the dimensionality of the input tensor and keeping only the most active pixels from the previous layer; - after 4 convolutional layers, the tensor has a shape of 14x14x128; - this needs to be reshaped for the fully connected linear layers; - this first fully connected layer has 25088 inputs and 2000 outputs, the second 500 outputs and the third 133 (the numbere of dog breeds to classify); - the first two fully connected layers are activated by a relu function and dropout (with p=0.2) is applied to advoid overfitting

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 [15]: import torch.optim as optim

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

        ### TODO: select optimizer
        optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.02)
```

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 [16]: 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()

        output = model(data)

        loss = criterion(output, target) #batch loss

        loss.backward()

        optimizer.step()

        train_loss += loss.item()*data.size(0) # update training 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()

        output = model(data)

        loss = criterion(output, target) # batch loss

        valid_loss += loss.item()*data.size(0) # update validation loss

    # calculate average losses
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)
```

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

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

            # return trained model
            return model
```

In [17]:
```
# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch, criterion_

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

```
Epoch: 1         Training Loss: 4.888194         Validation Loss: 4.881909
Validation loss decreased (inf --> 4.881909).  Saving model ...
Epoch: 2         Training Loss: 4.872764         Validation Loss: 4.846431
Validation loss decreased (4.881909 --> 4.846431).  Saving model ...
Epoch: 3         Training Loss: 4.743691         Validation Loss: 4.671318
Validation loss decreased (4.846431 --> 4.671318).  Saving model ...
Epoch: 4         Training Loss: 4.621735         Validation Loss: 4.595361
Validation loss decreased (4.671318 --> 4.595361).  Saving model ...
Epoch: 5         Training Loss: 4.551602         Validation Loss: 4.601108
Epoch: 6         Training Loss: 4.432635         Validation Loss: 4.441287
Validation loss decreased (4.595361 --> 4.441287).  Saving model ...
Epoch: 7         Training Loss: 4.318696         Validation Loss: 4.316684
Validation loss decreased (4.441287 --> 4.316684).  Saving model ...
Epoch: 8         Training Loss: 4.235289         Validation Loss: 4.285427
Validation loss decreased (4.316684 --> 4.285427).  Saving model ...
Epoch: 9         Training Loss: 4.150300         Validation Loss: 4.259720
Validation loss decreased (4.285427 --> 4.259720).  Saving model ...
Epoch: 10         Training Loss: 4.074110         Validation Loss: 4.171267
Validation loss decreased (4.259720 --> 4.171267).  Saving model ...
Epoch: 11         Training Loss: 3.987644         Validation Loss: 4.092035
Validation loss decreased (4.171267 --> 4.092035).  Saving model ...
Epoch: 12         Training Loss: 3.887328         Validation Loss: 4.248550
```

```
Epoch: 13         Training Loss: 3.788372         Validation Loss: 4.177383
Epoch: 14         Training Loss: 3.679230         Validation Loss: 3.983554
Validation loss decreased (4.092035 --> 3.983554).  Saving model ...
Epoch: 15         Training Loss: 3.541892         Validation Loss: 4.207381
Epoch: 16         Training Loss: 3.409439         Validation Loss: 3.961992
Validation loss decreased (3.983554 --> 3.961992).  Saving model ...
Epoch: 17         Training Loss: 3.272539         Validation Loss: 3.882238
Validation loss decreased (3.961992 --> 3.882238).  Saving model ...
Epoch: 18         Training Loss: 3.106074         Validation Loss: 3.905772
Epoch: 19         Training Loss: 2.910587         Validation Loss: 4.102928
Epoch: 20         Training Loss: 2.743450         Validation Loss: 4.066293
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

In [18]:
```
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: 3.893184


Test Accuracy: 11% (94/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [19]: ## TODO: Specify data loaders
         # same as before
         loaders_transfer = {'train': train_loader, 'valid' : valid_loader, 'test': test_loader}
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [20]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture

         # vgg16 with all parameters frozen, remove last layer and create new fully connected la

         # Load the pretrained model from pytorch
         vgg16 = models.vgg16(pretrained=True)

         # Freeze training for all "features" layers
         for param in vgg16.features.parameters():
             param.requires_grad = False

         n_inputs = vgg16.classifier[6].in_features
         # add last linear layer (n_inputs -> 133 dog breeds)
         # new layers automatically have requires_grad = True
         last_layer = nn.Linear(n_inputs, 133)
```

```
vgg16.classifier[6] = last_layer

model_transfer = vgg16

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

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

Answer:

For the transfer learning model, I use the VGG16 model again because it worked well for the dog detector. The data set is not very large and the model was already trained on similar data. Therefore I leave the weights fixed and just remove the last fully connected layer. I created a new one with 133 output nodes (the number of dog breeds to classify).

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [21]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.01)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [22]: # train the model
         model_transfer =  train(5, loaders_transfer, model_transfer, optimizer_transfer, criter

         # 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: 1.755274         Validation Loss: 0.732974
Validation loss decreased (inf --> 0.732974).  Saving model ...
Epoch: 2         Training Loss: 0.830977         Validation Loss: 0.545527
Validation loss decreased (0.732974 --> 0.545527).  Saving model ...
Epoch: 3         Training Loss: 0.635406         Validation Loss: 0.525728
Validation loss decreased (0.545527 --> 0.525728).  Saving model ...
Epoch: 4         Training Loss: 0.535331         Validation Loss: 0.500559
Validation loss decreased (0.525728 --> 0.500559).  Saving model ...
Epoch: 5         Training Loss: 0.476685         Validation Loss: 0.501582
```

1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [23]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.553545


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

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

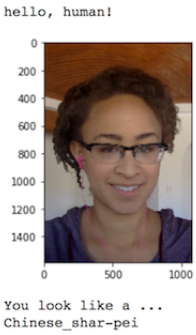
Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [24]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         data_transfer = {'train': train_data, 'valid' : valid_data, 'test': test_data}

         # 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'].classes]
         #print(class_names[0])

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image = Image.open(img_path).convert('RGB')
             in_transform = transforms.Compose([
                                 transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.485, 0.456, 0.406),
                                                      (0.229, 0.224, 0.225))])
             image = in_transform(image)
             image = image.unsqueeze(0)
             if use_cuda:
                 image = image.cuda()
             prediction = model_transfer(image)
             _,ind = torch.max(prediction,1)

             return class_names[ind]
```

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.



Sample Human Output

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 [25]: ### 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

             image = Image.open(img_path).convert('RGB')
             in_transform = transforms.Compose([
                                transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406),
                                                     (0.229, 0.224, 0.225))])
             image = in_transform(image)
             image = image.unsqueeze(0)
             if use_cuda:
                 image = image.cuda()
             prediction = model_transfer(image)
             _,ind = torch.max(prediction,1)

             if face_detector(img_path):
                 text = 'The picture shows a human who looks like a ... {}!'.format(predict_bree
             elif dog_detector(img_path):
                 text = 'The picture shows a dog, more specifically a ...{}!'.format(predict_bre
             else:
                 text = 'The picture shows neither a human nor a dog!'

             # load color (BGR) image
             img = cv2.imread(img_path)
```

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

    # display the image, along with text
    fig = plt.figure(figsize=(5, 5))
    plt.imshow(cv_rgb)
    plt.title(text)
    plt.show()

    return text
```

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.

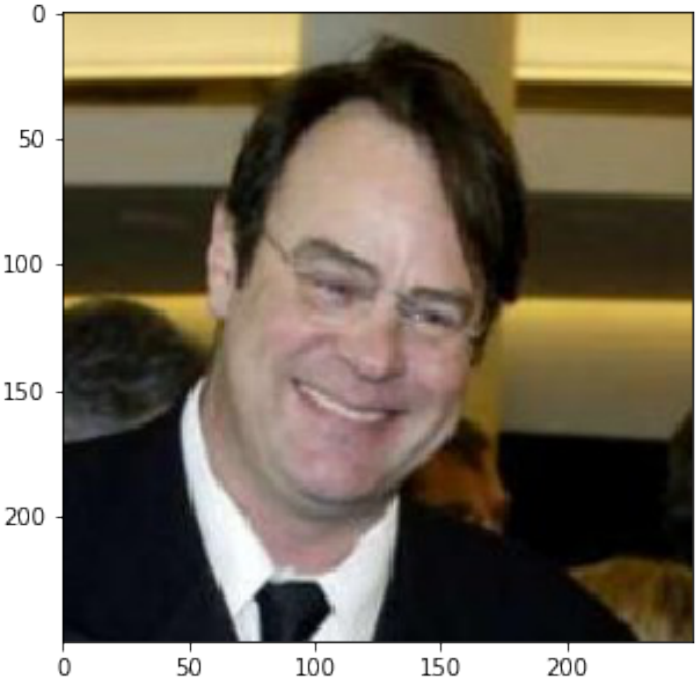
Answer: (Three possible points for improvement)

My algorithm could still be improved further. The accuracy of 83% from the transfer learning model for detecting dog breeds is not bad, but could be better. This could have been reached by longer training (more epochs). I have also just exchanged the last linear layer and left the convolutional layers unchanged. I could have fine-tuned them and just used the pretrained weights as starting points. This might have led to better results.

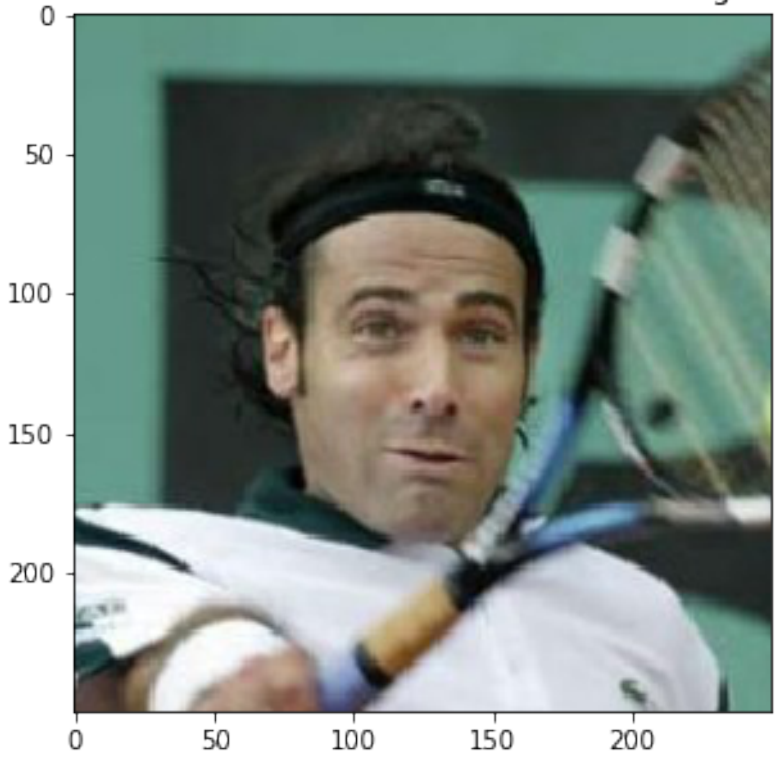
Also the test results for the dog_detector are really good (but do not work on all my test cases), but worse for the face_detector. I could have tried a similar approach for the face_detector as for the dog_detector (using a pretrained model to detect human faces) to get better results.

```
In [26]: ## 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)
```

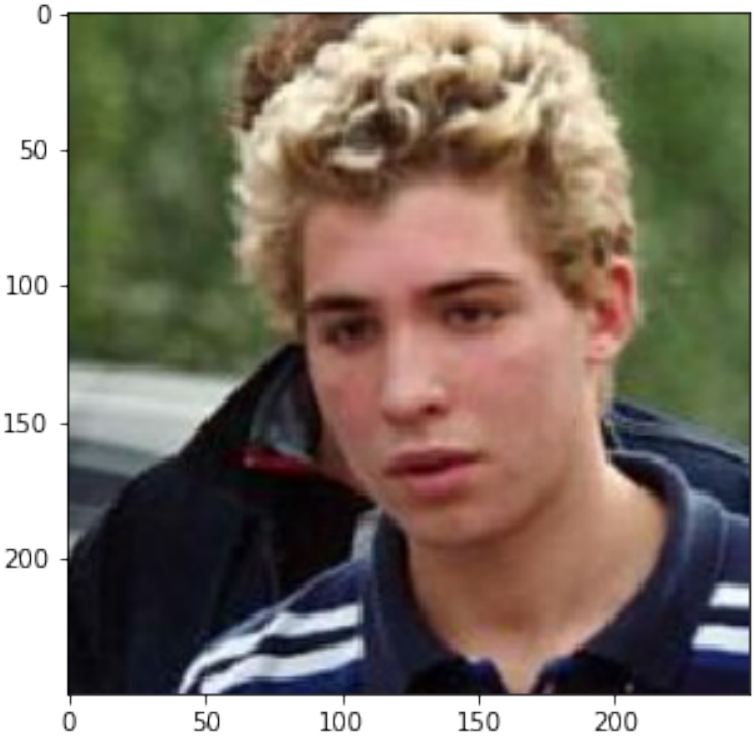
The picture shows a human who looks like a ... American staffordshire terrier!



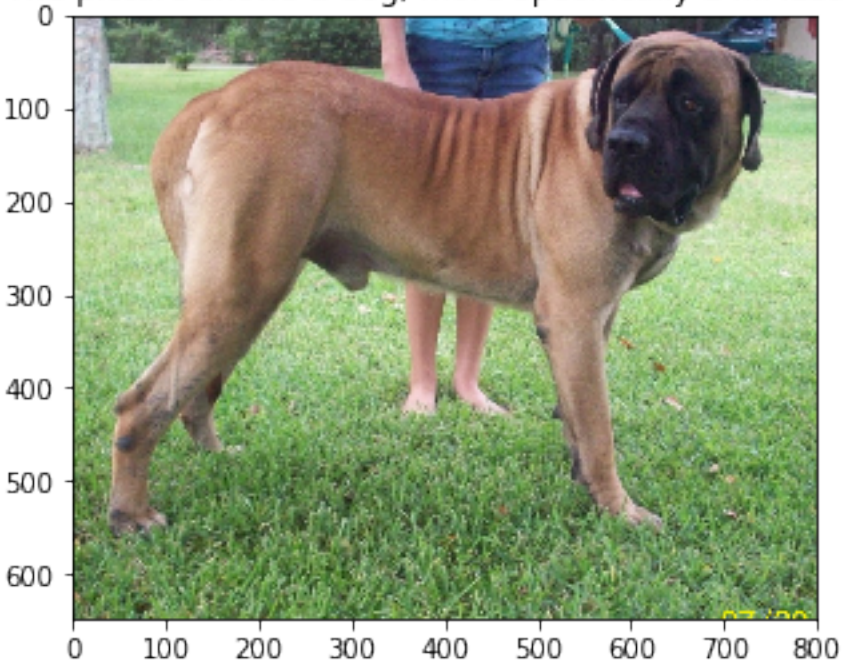
The picture shows a human who looks like a ... Norwegian lundehund!

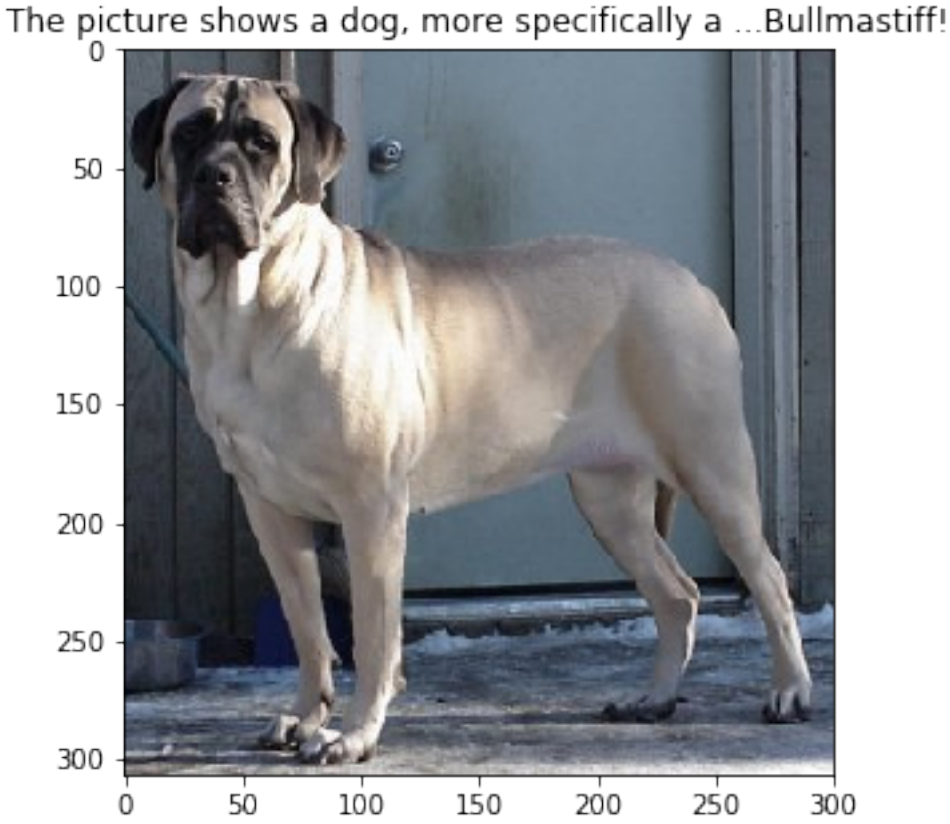


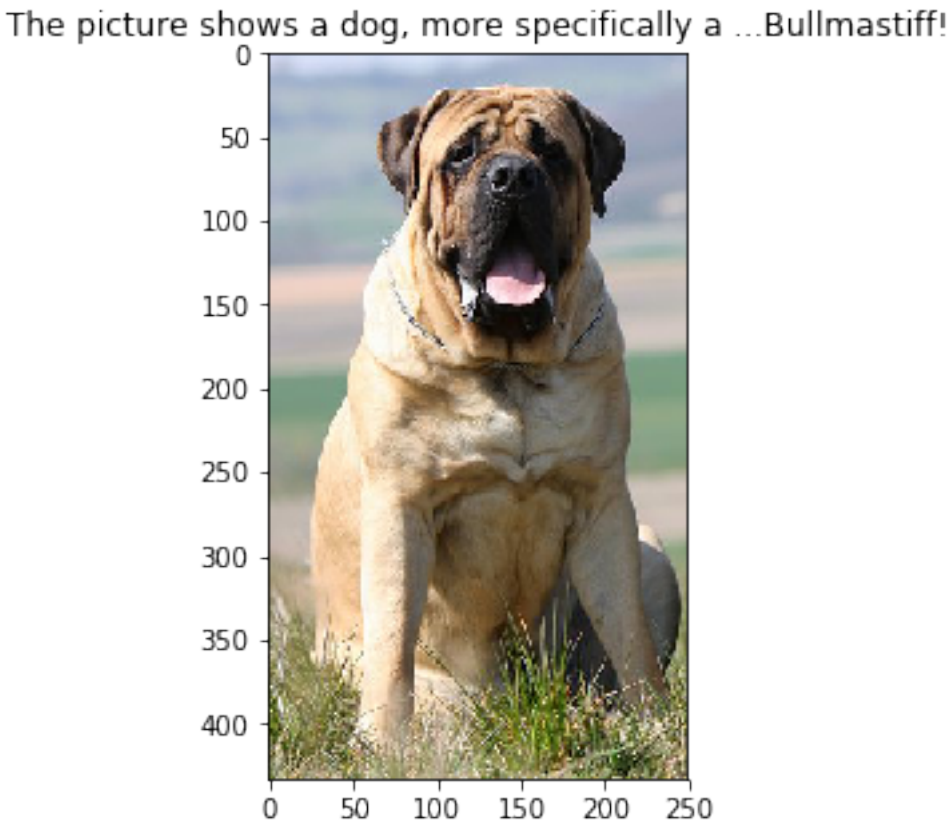
The picture shows a human who looks like a ... American water spaniel!



The picture shows a dog, more specifically a ...Mastiff!

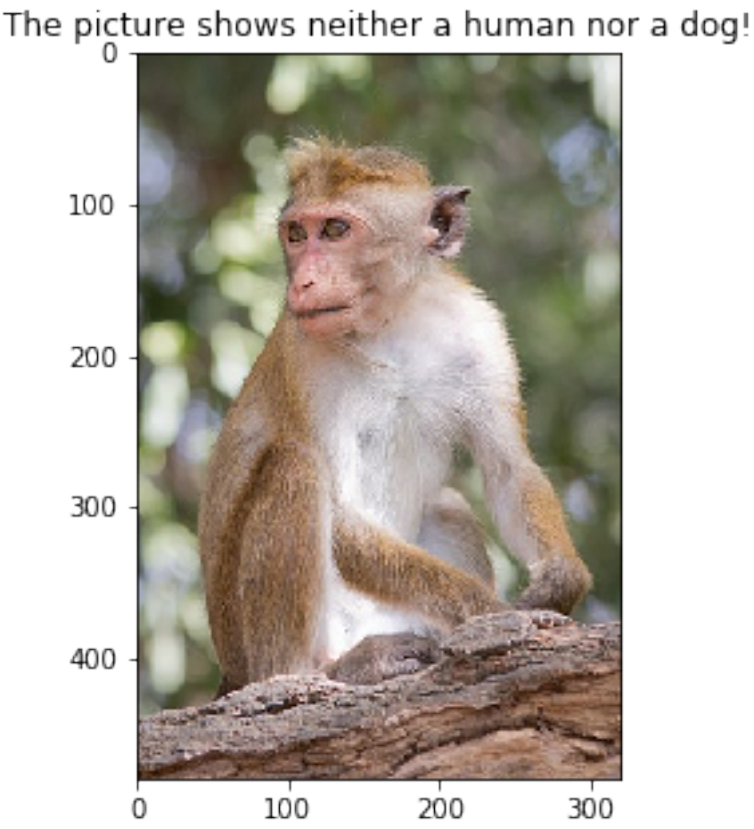



The picture shows a dog, more specifically a ...Bullmastiff!
0
50
100
150
200
250
300
0
50
100
150
200
250
300

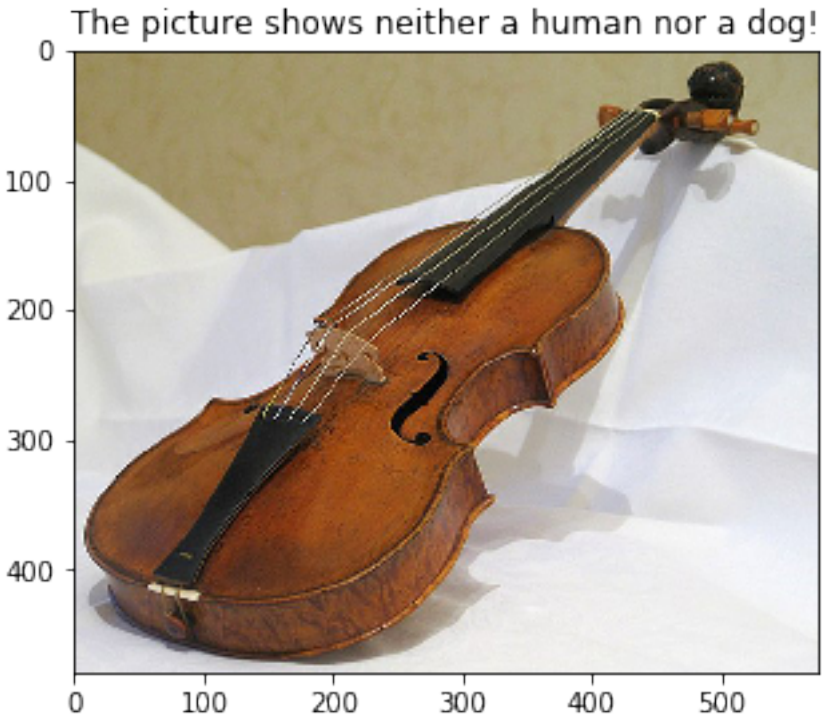



```
In [27]: own_files = np.array(glob("./test_pics/*"))
         for file in own_files:
             print(file)
             run_app(file)
```

```
./test_pics/neither2.jpg
```



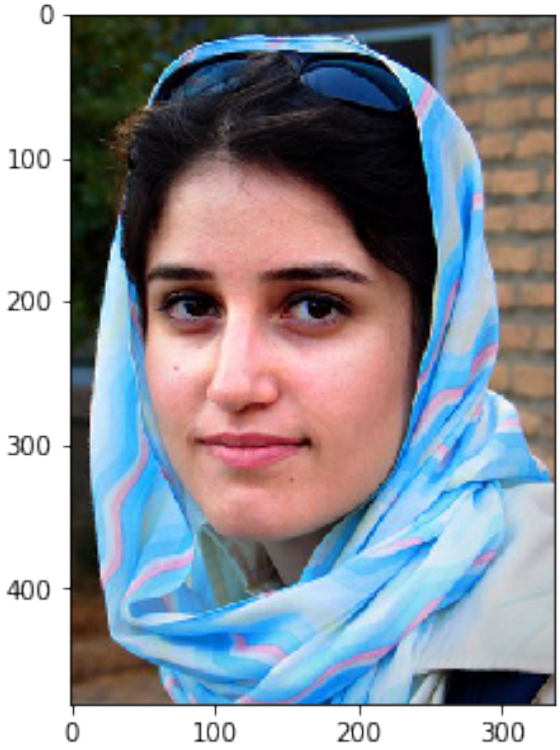

```
./test_pics/neither3.jpg
```



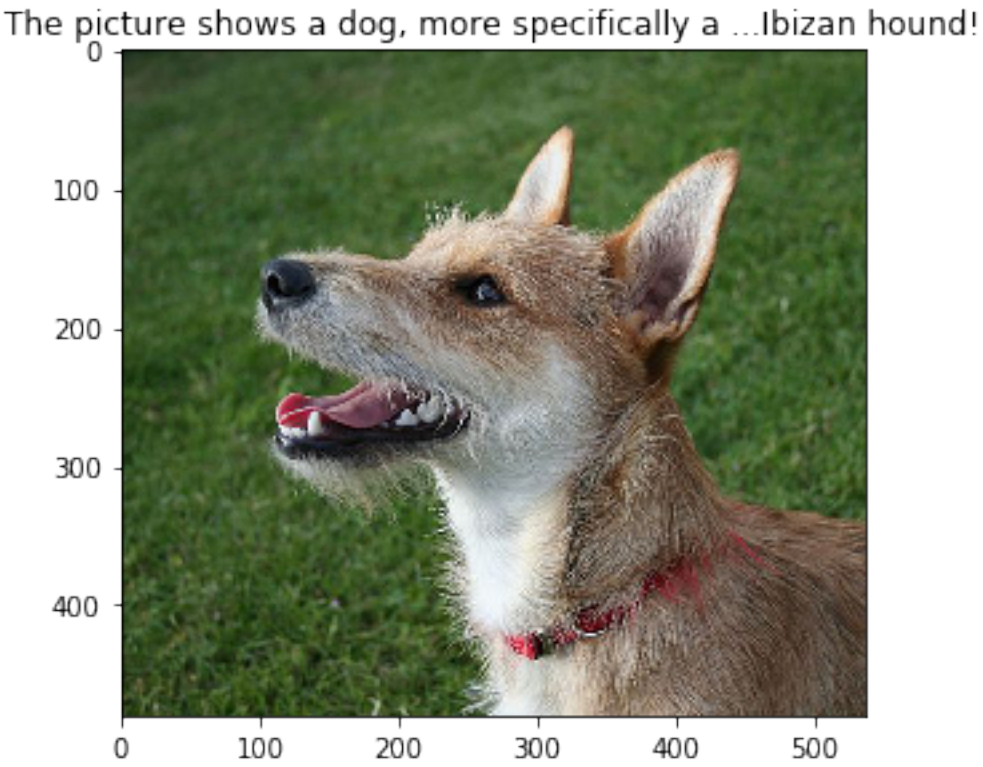
./test_pics/human3.jpg


```
./test_pics/human1.jpg
```

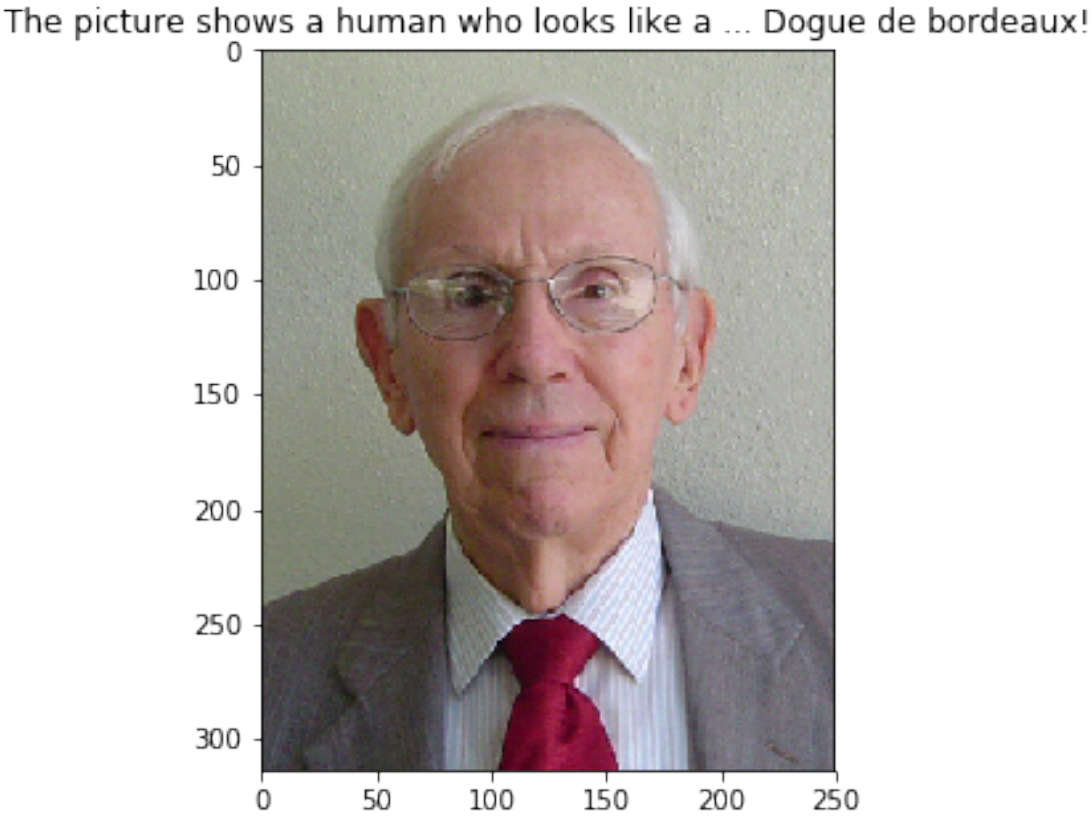
The picture shows a human who looks like a ... Welsh springer spaniel!



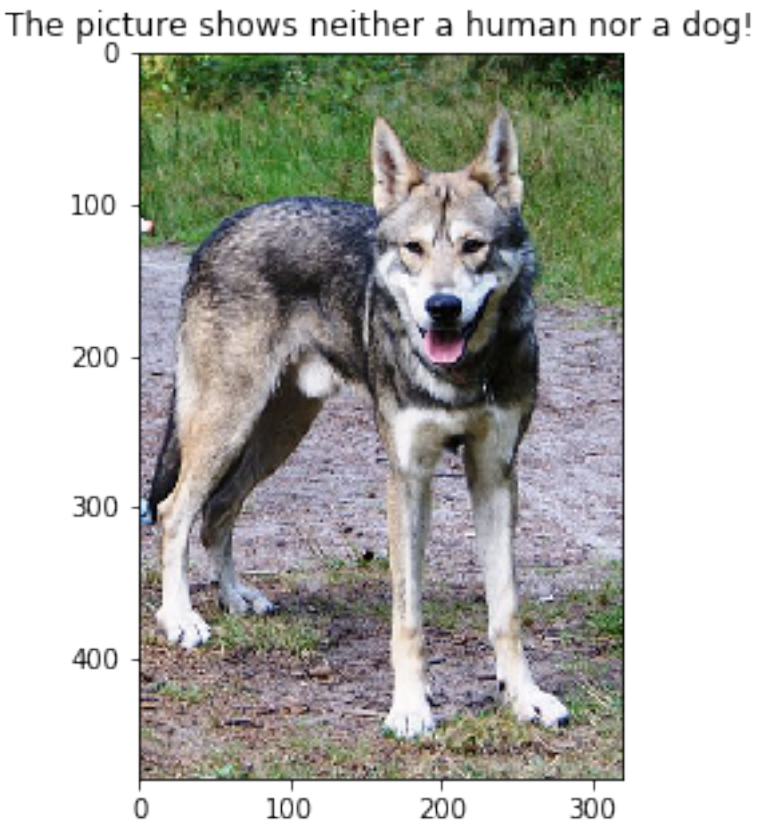

```
./test_pics/dog2.jpg
```



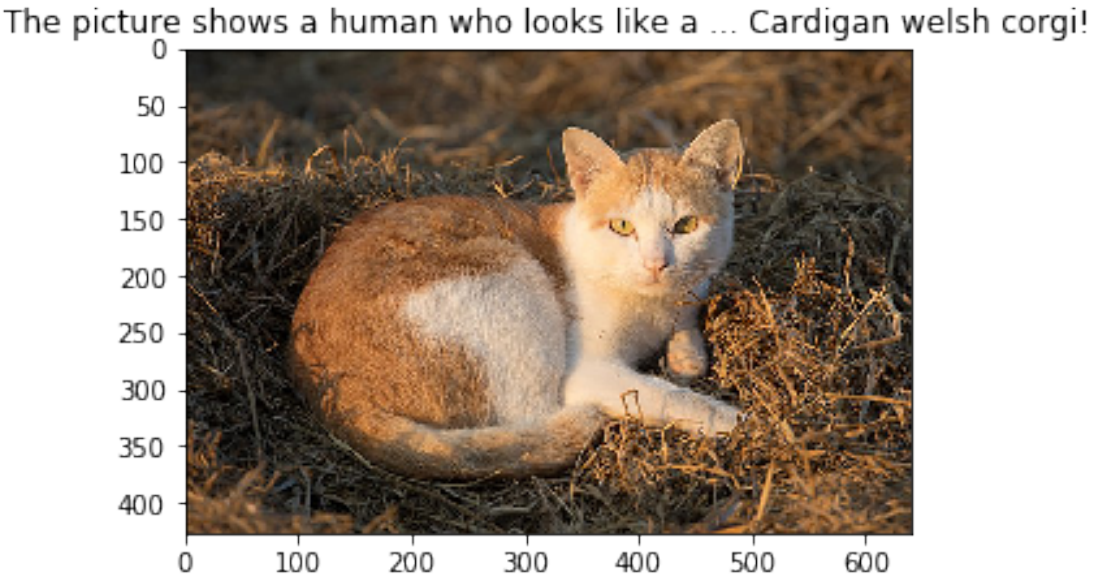

```
./test_pics/human2.jpg
```



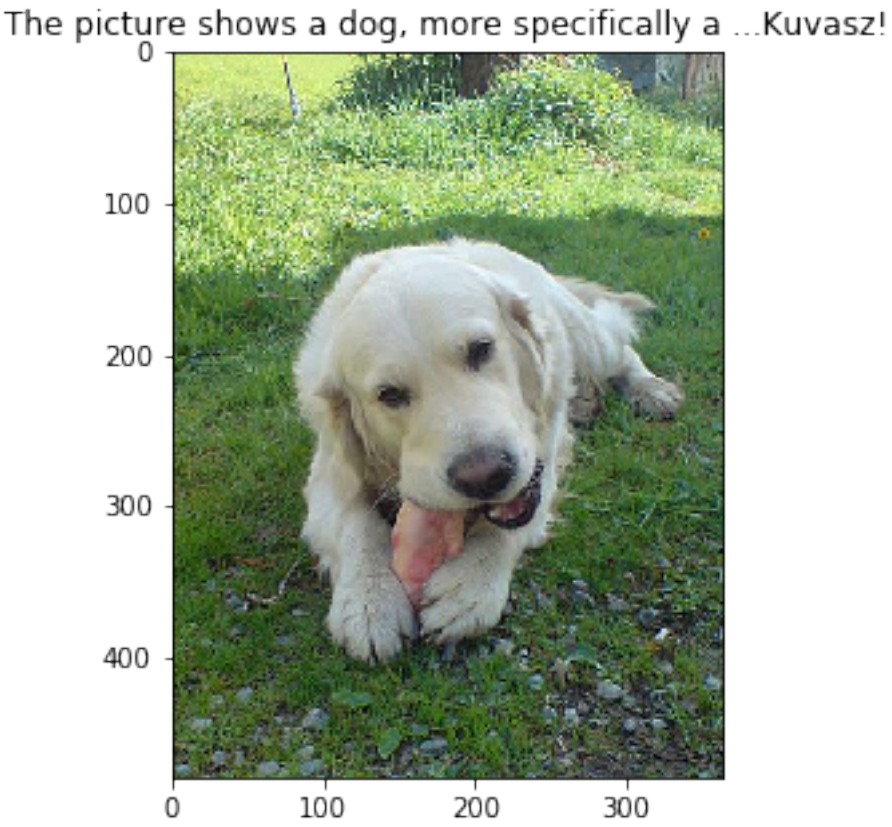

```
./test_pics/dog3.jpg
```




```
./test_pics/neither1.jpg
```



```
./test_pics/dog1.jpg
```



1.2 Picture sources:

Because I don't own a pet and I don't want to use personal pictures, I used open source pictures for testing. Sources are listed below:

1.2.1 Dog pictures from Wikipedia:

- By Denhulde - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=4300940
- By Chris Barber - Flickr, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=2253341
- By Elbieta Wojtko Orinek7 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=4912928

1.2.2 Human pictures from Wikipedia:

- By Hamed Saber from Tehran, Iran - Persian Beauty, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=51290535

- By Louise Phillips, wife of Alison Phillips - Alison Phillips, Public Domain, https://commons.wikimedia.org/w/index.php?curid=29640639
- By Lorenzo Lippi - The Yorck Project (2002) 10.000 Meisterwerke der Malerei (DVD-ROM), distributed by DIRECTMEDIA Publishing GmbH. ISBN: 3936122202., Public Domain, https://commons.wikimedia.org/w/index.php?curid=153925

1.2.3 Neither dog nor human from Wikipedia:

- By Basile Morin - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=685080
- By Carlos Delgado - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=3466
- By User:Frinck51 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=59307

In [ ]: