dog_app

November 30, 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.

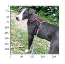
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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.

Why We're Here

In this note- book, you will make the first steps to- wards de- vel- op- ing an algo- rithm that could be used as part of a mo- bile or web app. At the end of this project, your code will ac- cept any user- supplied im- age as in- put. If a dog

```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```



In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog

---

The Road Ahead We break the note-book into sep-a-rate steps. Feel free to use the links be-low to nav-igate the notebook.

* Step 0: Import Datasets
* Step 1: Detect Humans
* Step 2: Detect Dogs
* Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
* Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
* Step 5:

## 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`.

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

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

# 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

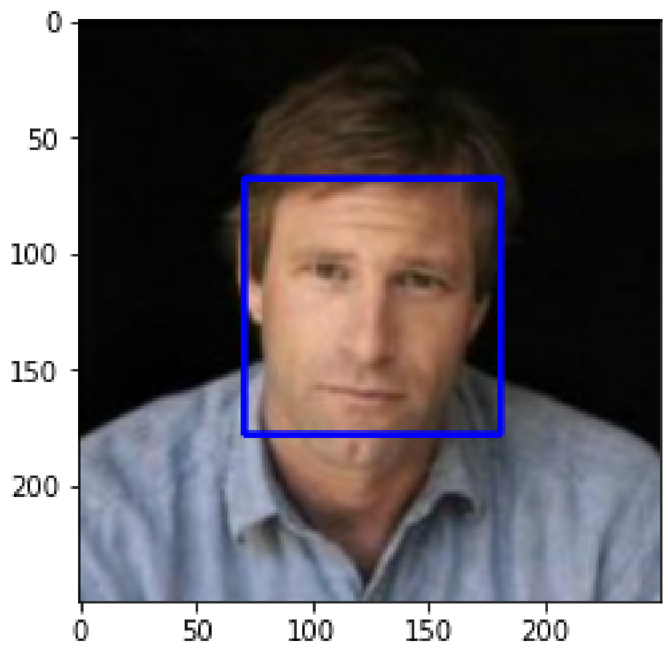
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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.

[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.

[3]:
```python
# 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: 96% for `human_files` and 18% for `dog_files`

[4]:
```python
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_performance = 0
dog_performance = 0

for image in human_files_short:
    if face_detector(image) == True:
        human_performance += 1
print(str(human_performance) + "%")


for image in dog_files_short:
    if face_detector(image) == True:
        dog_performance += 1
print(str(dog_performance) + "%")
```

```
96%
18%
```

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`.

[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.

[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()
```

```
C:\Users\iorlf\anaconda3\lib\site-packages\torch\cuda\__init__.py:52:
UserWarning: CUDA initialization: Found no NVIDIA driver on your system. Please
check that you have an NVIDIA GPU and installed a driver from
http://www.nvidia.com/Download/index.aspx (Triggered internally at
..\c10\cuda\CUDAFunctions.cpp:100.)
  return torch._C._cuda_getDeviceCount() > 0
```

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.

[7]:
```python
from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''


    ## 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

    img = Image.open(img_path)

    img_transform = transforms.Compose([
                        transforms.Resize(img.size),
                         transforms.CenterCrop(size=224),
                         transforms.ToTensor(),
                         transforms.Normalize((0.485, 0.456, 0.406),
                                              (0.229, 0.224, 0.225))])
    image = img_transform(img).unsqueeze(0)

#     image= image.cuda()

    VGG16.eval() # eval mode

    _, index = torch.max(VGG16(image),1)

    return index.item()
```

[8]:
```
VGG16_predict(dog_files_short[10])
```

[8]:
```
160
```

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).

[9]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.

    if VGG16_predict(img_path) in range(151, 269):
        return True
    else:
        return False
```

[10]:
```
dog_detector(dog_files_short[11])
```

[10]:
```
True
```

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:

The percentage of the images in the `human_files_short` is 0% The percentage of the images in the `dog_files_short` is 93%

[11]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

dog_human_performance = 0
dog_dog_performance = 0

for img_path in human_files_short:
    if dog_detector(img_path) == True:
        dog_human_performance += 1
print(str(dog_human_performance) + "%")


for img_path in dog_files_short:
    if dog_detector(img_path) == True:
        dog_dog_performance += 1
print(str(dog_dog_performance) + "%")
```

```
0%
93%
```

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`.

[12]:
```
### (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!

[13]:
```
import os
from torchvision import datasets
import torchvision.transforms as transforms
from PIL import ImageFile


### TODO: Write data loaders for training, validation, and test sets
ImageFile.LOAD_TRUNCATED_IMAGES = True

# define training and test data directories
data_dir = "C:/Users/iorlf/Desktop/Data Science_SH/Udacity_Deep Learning_Nano/
 ↪deep-learning-v2-pytorch-master/Project 2 - Dog Classification/dogImages"
```

```
train_dir = os.path.join(data_dir, 'train/')
valid_dir =  os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

## Specify appropriate transforms, and batch_sizes
# number of subprocesses to use for data loading
num_workers = 0

# how many samples per batch to load
batch_size = 20

data_transform = {

    # RandomHorizontalFlip() & RandomRotation() to augement data in train␣
 ↪transformation
    'train' : transforms.Compose([transforms.Resize(256),
                                  transforms.RandomResizedCrop(224),
                                  transforms.RandomHorizontalFlip(),
                                  transforms.RandomRotation(10),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
                                                       std=[0.229, 0.224, 0.
 ↪225])]),

    'valid' : transforms.Compose([transforms.Resize(256),
                                  transforms.CenterCrop(224),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
                                                       std=[0.229, 0.224, 0.
 ↪225])]),

    'test' : transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
                                                      std=[0.229, 0.224, 0.
 ↪225])])
}


train_data = datasets.ImageFolder(train_dir, transform=data_transform['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform['valid'])
```

```
test_data = datasets.ImageFolder(test_dir, transform=data_transform['test'])

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,␣
 ↪num_workers=num_workers, shuffle=True)
valid_loader = torch.utils.data.DataLoader(train_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:

1. I resized the images to 256 pixels and cropped them to 224 pixels.
2. Yes I used horizontal flip and rotation for 10 deg.

1.1.8 (IMPLEMENTATION) Model Architecture

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

[24]:
```
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
        # 224 * 224 * 3
        self.conv1 = nn.Conv2d(3, 16, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)

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

        #Fully connected layer
        self.fc1 = nn.Linear(7 * 7 * 128, 512)
        self.fc2 = nn.Linear(512, 133)

        # drop-out
        self.dropout = nn.Dropout(0.4)


    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 = self.pool(F.relu(self.conv5(x)))

        # flatten
        x = x.view(x.size(0), -1)

        x = self.dropout(x)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = self.fc2(x)

        return x

#-#-# You do 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()


print(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): 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=6272, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.4, inplace=False)
)
```

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

Answer:

1. The first input layer is 224x224x3 and the output has 133 classes.
2. The first convolutional layer is 112x112x16; after maxpooling, it becomes 56x56x16.
3. The second convolutional layer is 56x56x32; after maxpooling, it becomes 28x28x32.
4. The third convolutional layer is 28x28x64; after 14x14x64
5. The fourth convolutional layer is 14x14x128, after ReLu activation function, it becomes 7x7x128.
6. Two fully connected layers: the first one goes from 6272 features to 512 features as output. The second goes from 512 features to 133 classes.
7. dropout = 0.3
8. These parameters are obtained after several experiments.

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.

[25]:
```
import torch.optim as optim

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


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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

[29]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True


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_loss))

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

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass
            output = model(data)
            # batch loss
            loss = criterion(output, target)
            # update validation loss
            valid_loss += loss.item() * data.size(0)

        # calculate average losses
        train_loss = train_loss/len(loaders['train'].dataset)
        valid_loss = valid_loss/len(loaders['valid'].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
        if valid_loss <= valid_loss_min:
            print('Validation loss decreased ({:.6f} --> {:.6f}).    Saving␣
 ↪model...'.
                 format(valid_loss_min, valid_loss))
            torch.save(model.state_dict(), save_path)
            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: 4.233361         Validation Loss: 4.110986
Validation loss decreased (inf --> 4.110986).    Saving model…
Epoch: 2        Training Loss: 4.210989         Validation Loss: 4.085627
Validation loss decreased (4.110986 --> 4.085627).    Saving model…
Epoch: 3        Training Loss: 4.163762         Validation Loss: 4.048190
Validation loss decreased (4.085627 --> 4.048190).    Saving model…
Epoch: 4        Training Loss: 4.155178         Validation Loss: 4.026704
Validation loss decreased (4.048190 --> 4.026704).    Saving model…
Epoch: 5        Training Loss: 4.130531         Validation Loss: 3.984756
Validation loss decreased (4.026704 --> 3.984756).    Saving model…
Epoch: 6        Training Loss: 4.110177         Validation Loss: 3.953700
Validation loss decreased (3.984756 --> 3.953700).    Saving model…
Epoch: 7        Training Loss: 4.087589         Validation Loss: 3.936505
Validation loss decreased (3.953700 --> 3.936505).    Saving model…
```

```
Epoch: 8        Training Loss: 4.063428        Validation Loss: 3.921946
Validation loss decreased (3.936505 --> 3.921946).    Saving model…
Epoch: 9        Training Loss: 4.028560        Validation Loss: 3.885422
Validation loss decreased (3.921946 --> 3.885422).    Saving model…
Epoch: 10       Training Loss: 4.000980        Validation Loss: 3.891244
Epoch: 11       Training Loss: 3.987342        Validation Loss: 3.839429
Validation loss decreased (3.885422 --> 3.839429).    Saving model…
Epoch: 12       Training Loss: 3.966410        Validation Loss: 3.841684
Epoch: 13       Training Loss: 3.950411        Validation Loss: 3.773225
Validation loss decreased (3.839429 --> 3.773225).    Saving model…
Epoch: 14       Training Loss: 3.925167        Validation Loss: 3.785482
Epoch: 15       Training Loss: 3.919394        Validation Loss: 3.728841
Validation loss decreased (3.773225 --> 3.728841).    Saving model…
Epoch: 16       Training Loss: 3.863620        Validation Loss: 3.725016
Validation loss decreased (3.728841 --> 3.725016).    Saving model…
Epoch: 17       Training Loss: 3.892025        Validation Loss: 3.751821
Epoch: 18       Training Loss: 3.845278        Validation Loss: 3.712613
Validation loss decreased (3.725016 --> 3.712613).    Saving model…
Epoch: 19       Training Loss: 3.844303        Validation Loss: 3.698217
Validation loss decreased (3.712613 --> 3.698217).    Saving model…
Epoch: 20       Training Loss: 3.824936        Validation Loss: 3.710656
Epoch: 21       Training Loss: 3.806488        Validation Loss: 3.595903
Validation loss decreased (3.698217 --> 3.595903).    Saving model…
Epoch: 22       Training Loss: 3.778316        Validation Loss: 3.618128
Epoch: 23       Training Loss: 3.772528        Validation Loss: 3.568526
Validation loss decreased (3.595903 --> 3.568526).    Saving model…
Epoch: 24       Training Loss: 3.746706        Validation Loss: 3.562263
Validation loss decreased (3.568526 --> 3.562263).    Saving model…
Epoch: 25       Training Loss: 3.714434        Validation Loss: 3.510066
Validation loss decreased (3.562263 --> 3.510066).    Saving model…
Epoch: 26       Training Loss: 3.720721        Validation Loss: 3.575320
Epoch: 27       Training Loss: 3.695088        Validation Loss: 3.516500
Epoch: 28       Training Loss: 3.687641        Validation Loss: 3.477462
Validation loss decreased (3.510066 --> 3.477462).    Saving model…
Epoch: 29       Training Loss: 3.650174        Validation Loss: 3.522171
Epoch: 30       Training Loss: 3.633583        Validation Loss: 3.503656
Epoch: 31       Training Loss: 3.626010        Validation Loss: 3.396955
Validation loss decreased (3.477462 --> 3.396955).    Saving model…
Epoch: 32       Training Loss: 3.613072        Validation Loss: 3.416016
Epoch: 33       Training Loss: 3.631413        Validation Loss: 3.351913
Validation loss decreased (3.396955 --> 3.351913).    Saving model…
Epoch: 34       Training Loss: 3.569676        Validation Loss: 3.339841
Validation loss decreased (3.351913 --> 3.339841).    Saving model…
Epoch: 35       Training Loss: 3.570176        Validation Loss: 3.313715
Validation loss decreased (3.339841 --> 3.313715).    Saving model…
```

```
[29]: <All keys matched successfully>
```

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%.

[30]:
```
def test(loaders, model, criterion, use_cuda):

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

    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: 3.398047


Test Accuracy: 19% (161/836)
```

## 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.

[31]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets
import torchvision.transforms as transforms
from PIL import ImageFile


### TODO: Write data loaders for training, validation, and test sets
ImageFile.LOAD_TRUNCATED_IMAGES = True

# define training and test data directories
data_dir = "C:/Users/iorlf/Desktop/Data Science_SH/Udacity_Deep Learning_Nano/
 ↪deep-learning-v2-pytorch-master/Project 2 - Dog Classification/dogImages"


train_dir = os.path.join(data_dir, 'train/')
valid_dir =  os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

## Specify appropriate transforms, and batch_sizes
# number of subprocesses to use for data loading
num_workers = 0

# how many samples per batch to load
batch_size = 20

data_transform = {

    # RandomHorizontalFlip() & RandomRotation() to augement data in train␣
 ↪transformation
    'train' : transforms.Compose([transforms.Resize(256),
                                  transforms.RandomResizedCrop(224),
                                  transforms.RandomHorizontalFlip(),
                                  transforms.RandomRotation(10),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.
↪406],
                                                      std=[0.229, 0.224, 0.
↪225])]),

    'valid' : transforms.Compose([transforms.Resize(256),
                                  transforms.CenterCrop(224),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.
↪406],
                                                      std=[0.229, 0.224, 0.
↪225])]),

    'test' : transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485, 0.456, 0.
↪406],
                                                     std=[0.229, 0.224, 0.
↪225])])
}

#224 * 224 *3
train_data = datasets.ImageFolder(train_dir, transform=data_transform['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform['valid'])
test_data = datasets.ImageFolder(test_dir, transform=data_transform['test'])

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,␣
 ↪num_workers=num_workers, shuffle=True)
valid_loader = torch.utils.data.DataLoader(train_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_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`.

[32]:
```
import torchvision.models as models
import torch.nn as nn
```

```
## TODO: Specify model architecture
model_transfer = models.vgg16_bn(pretrained=True)

# Freeze training for all "features" layers
for param in VGG16.features.parameters():
    param.requires_grad = False

# fully connected layer as the last layer
model_transfer.classifier = nn.Sequential(nn.Linear(25088, 133))

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

[33]:
```
model_transfer
```

[33]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (36): ReLU(inplace=True)
    (37): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (39): ReLU(inplace=True)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=133, bias=True)
  )
)
```

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

Answer:

1. I started with VGG16 model, replaced with one fully connected layer as a classifier that takes 25088 inputs and 133 outputs.
2. The learning rate was changed from 0.01 and 0.001. The later one has better performance.
3. The initial train doesn't give good results. So I used VGG16 with batch normalization layers.

This time the result is 72% accuracy.
4. I think this architecture is suitable for this problem as it is pretrained on IMAGENET which is similar to the dog breed classification problem.

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.

[34]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.
 ↪001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

[35]:
```
# train the model
model_transfer = train(15, loaders_transfer, model_transfer,␣
 ↪optimizer_transfer, criterion_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: 3.774912         Validation Loss: 2.406760
Validation loss decreased (inf --> 2.406760).    Saving model…
Epoch: 2        Training Loss: 2.599025         Validation Loss: 2.153608
Validation loss decreased (2.406760 --> 2.153608).    Saving model…
Epoch: 3        Training Loss: 2.344269         Validation Loss: 2.055963
Validation loss decreased (2.153608 --> 2.055963).    Saving model…
Epoch: 4        Training Loss: 2.349879         Validation Loss: 2.042208
Validation loss decreased (2.055963 --> 2.042208).    Saving model…
Epoch: 5        Training Loss: 2.267749         Validation Loss: 2.129164
Epoch: 6        Training Loss: 2.281118         Validation Loss: 1.912228
Validation loss decreased (2.042208 --> 1.912228).    Saving model…
Epoch: 7        Training Loss: 2.167209         Validation Loss: 1.876957
Validation loss decreased (1.912228 --> 1.876957).    Saving model…
Epoch: 8        Training Loss: 2.292864         Validation Loss: 1.988397
Epoch: 9        Training Loss: 2.229920         Validation Loss: 1.834872
Validation loss decreased (1.876957 --> 1.834872).    Saving model…
Epoch: 10       Training Loss: 2.157330         Validation Loss: 1.760168
Validation loss decreased (1.834872 --> 1.760168).    Saving model…
Epoch: 11       Training Loss: 2.082944         Validation Loss: 1.887617
Epoch: 12       Training Loss: 2.108066         Validation Loss: 1.962073
```

```
Epoch: 13       Training Loss: 2.193261         Validation Loss: 1.903293
Epoch: 14       Training Loss: 2.133721         Validation Loss: 1.809213
Epoch: 15       Training Loss: 2.243020         Validation Loss: 1.834044
```

[35]:
```
<All keys matched successfully>
```

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%.

[36]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 2.298791


Test Accuracy: 72% (610/836)
```

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.

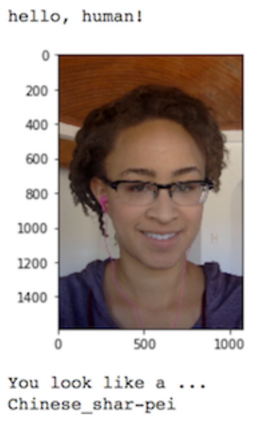
[37]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

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

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img_transforms = transforms.Compose([transforms.CenterCrop(224),
                                         transforms.ToTensor()])
    img = Image.open(img_path)
    img_tensor = img_transforms(img)[None, :]
    if use_cuda:
        img_tensor = img_tensor.cuda()

    model_transfer.eval()
    output = model_transfer(img_tensor)
    _, preds_tensor = torch.max(output, 1)

    return class_names[preds_tensor.item()]
```

## 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!



```
hello, human!
```

```
You look like a ...
Chinese_shar-pei
```

1.1.18 (IMPLEMENTATION) Write your Algorithm

[38]:
```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

    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    if dog_detector(img_path):
        prediction = predict_breed_transfer(img_path)
        print("Hello, dog!\nIt looks like a {0}".format(prediction))

    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(img_path)
        print("Hello, human!\nYou look like a {0}".format(prediction))

    else:
        print("There is an error.")
```

## 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!

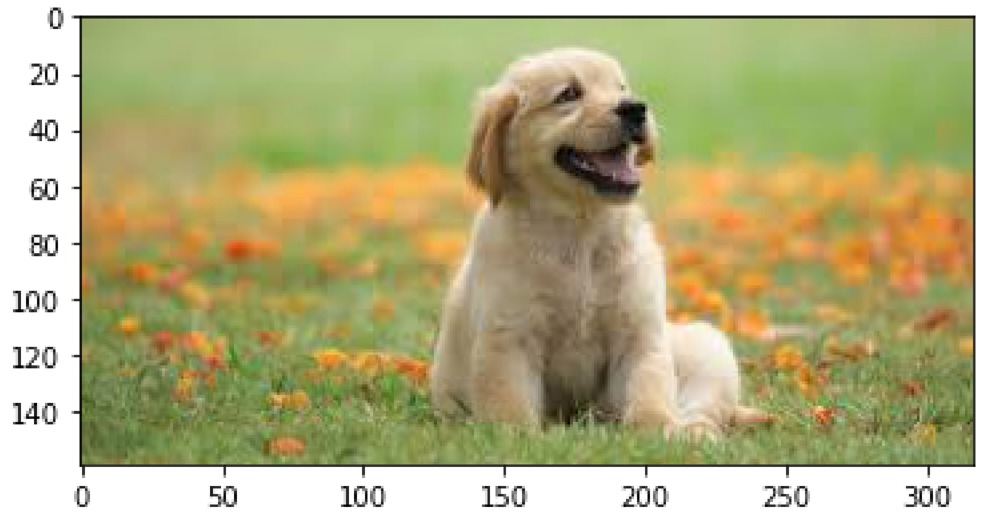
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

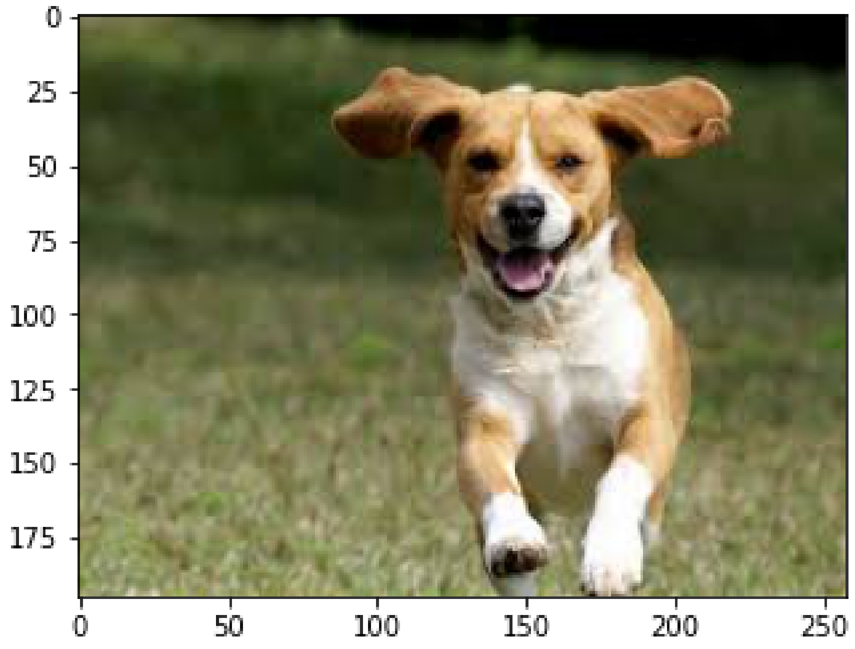
Answer: (Three possible points for improvement)

1. More image augmentations may help improve the accuracy.
2. The parameters can be udpated (e.g. learning rate, batch size, etc.).
3. May use another pretrained network that works better.
4. May need more training data.

[42]:
```
## 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

images = np.array(glob("Random Image/*"))

for file in np.hstack(images):
    run_app(file)
```

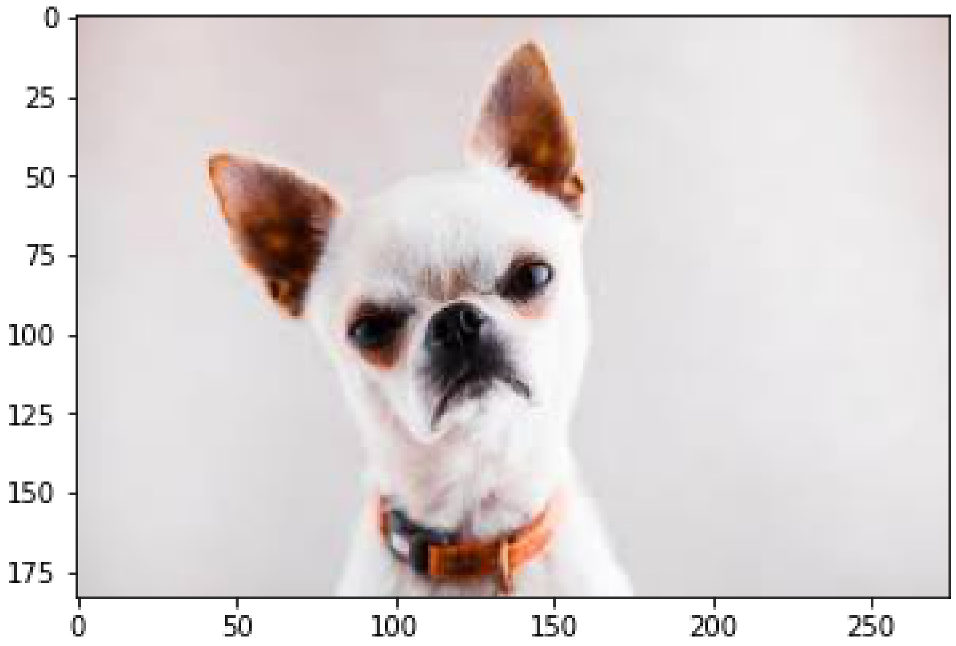


```
There is an error.
```



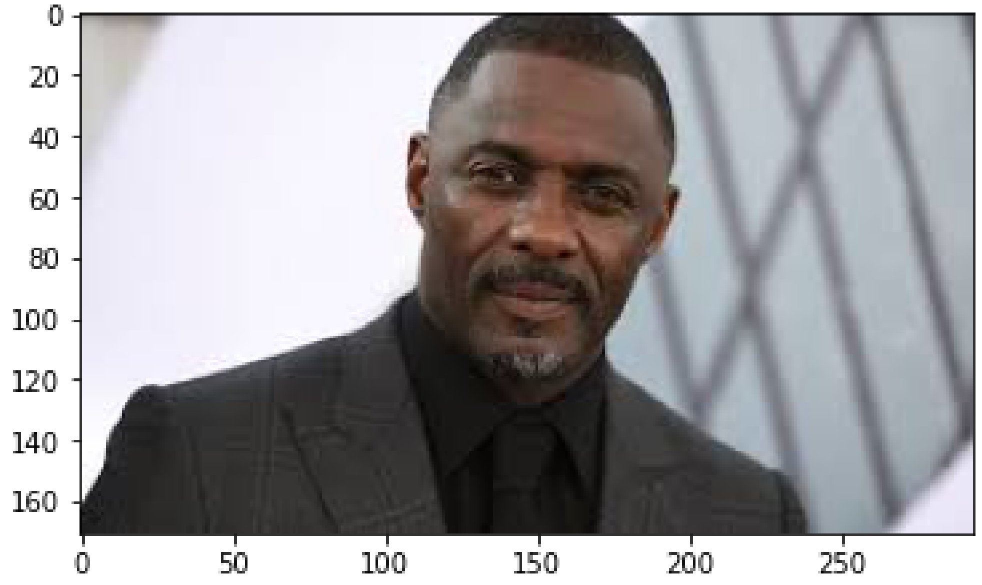
```
Hello, dog!
It looks like a Chinese shar-pei
```

```
Hello, dog!
It looks like a Pointer
```



```
Hello, dog!
```

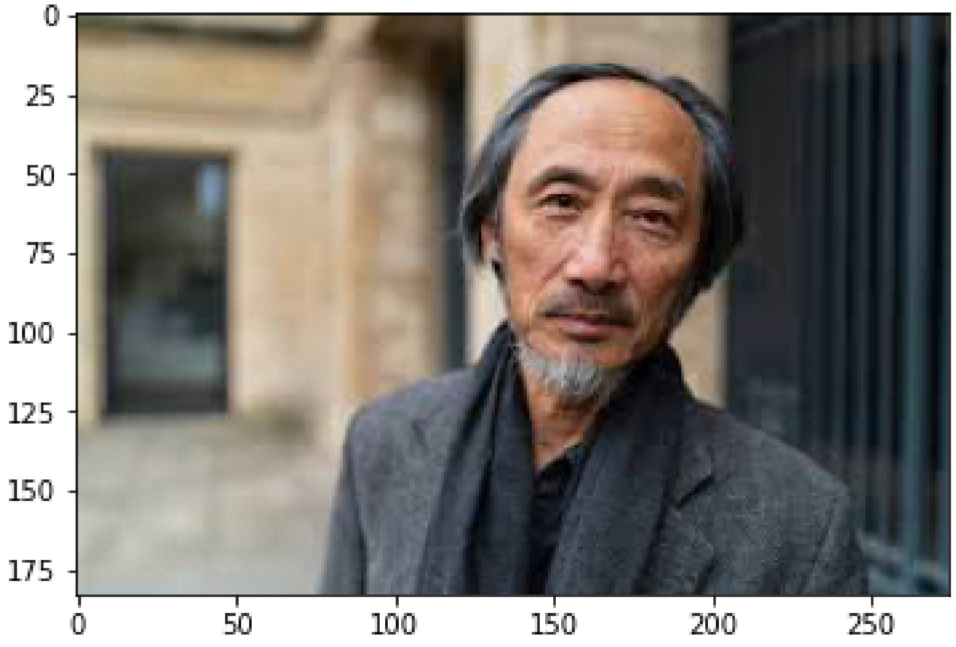
```
It looks like a Chinese crested
```



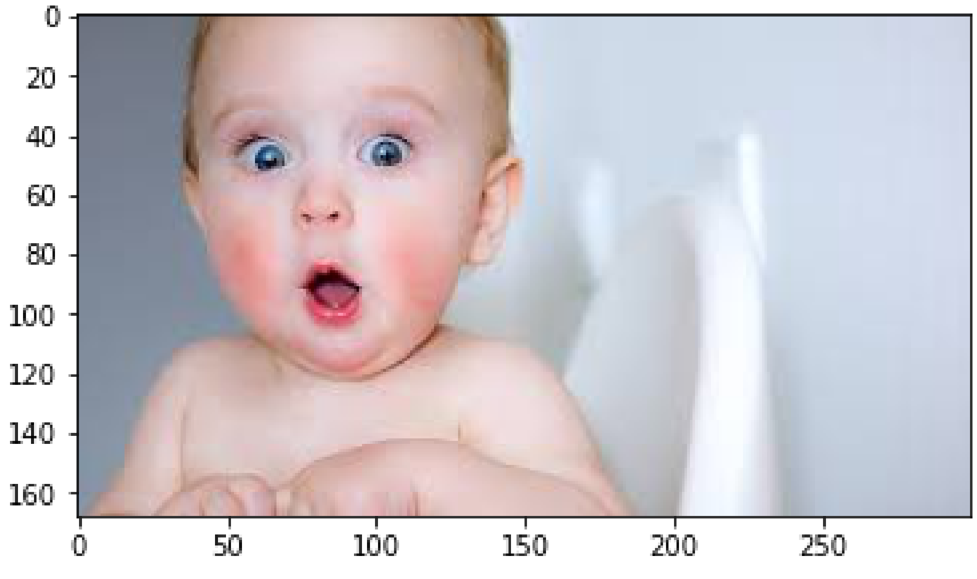
```
Hello, human!
You look like a Pharaoh hound
```

```
Hello, human!
```

```
You look like a English springer spaniel
```



```
Hello, human!
You look like a Clumber spaniel
```



```
Hello, human!
You look like a American eskimo dog
```