dog_app

April 5, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

## Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- 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`.

[26]:
```
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/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.
```

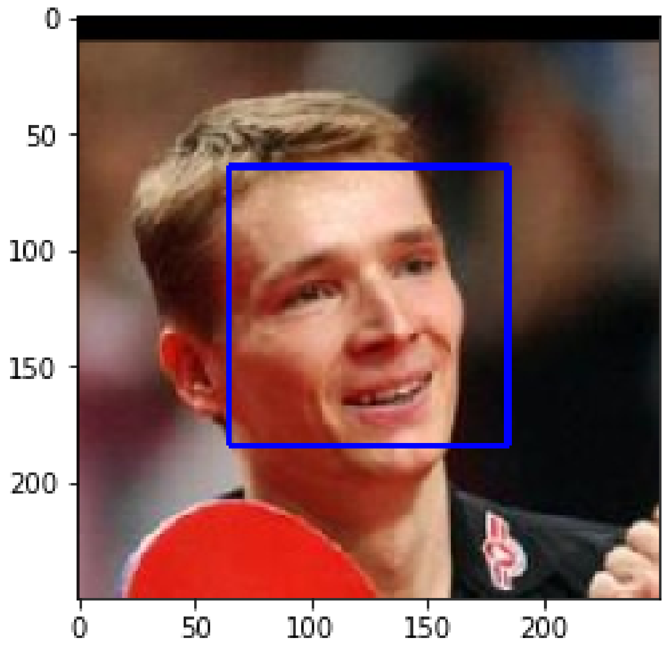
## Step 1: Detect Humans

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.

[27]:
```
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[12])
# 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.

[28]:
```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: (You can print out your results and/or write your percentages in this cell)

[29]:
```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.
cnt_human_set= 0
cnt_dog_set=0

for i in tqdm (range(100)):

    if (face_detector(human_files_short[i])):
        cnt_human_set += 1.0

    if (face_detector(dog_files_short[i])):
        cnt_dog_set += 1.0

print('true  human percent: ', cnt_human_set , '%'  )
print('false human percent: ', cnt_dog_set , '%' )
```

```
100%|      | 100/100 [00:11<00:00,  8.74it/s]
```

```
true  human percent:  99.0 %
false human percent:  12.0 %
```

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

[30]:
```
### (Optional)
### TODO: Test performance of another face 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.

[31]:
```
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()
```

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.

[32]:
```
import json
class_idx = json.load(open("data/imagenet_class_index.json"))
idx2label = [class_idx[str(k)][1] for k in range(len(class_idx))]

idx_1000cls = idx2label
```

```
#print(idx_1000cls)
```

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

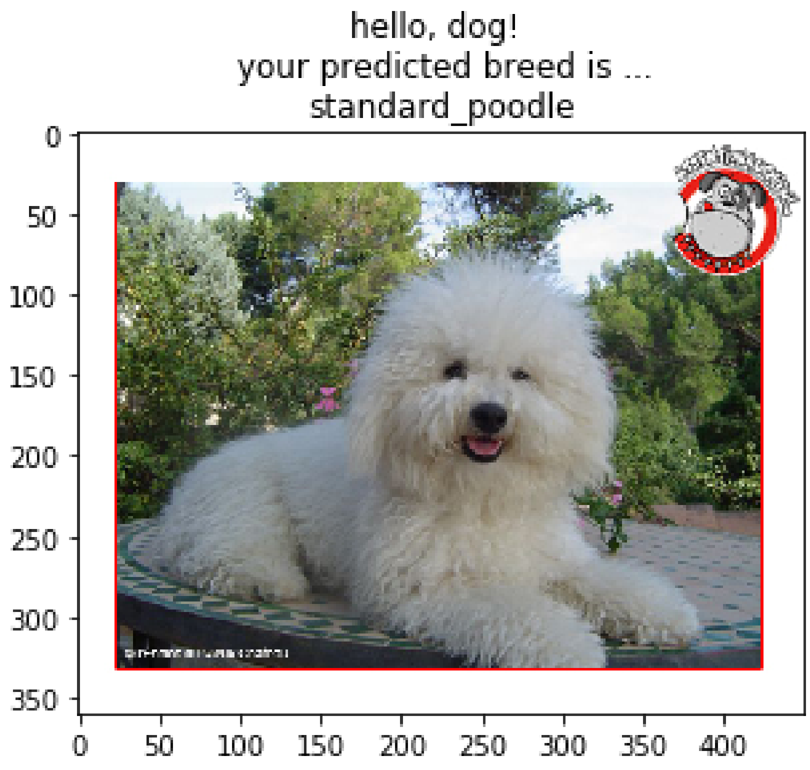
In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an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

[33]:
```
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

    input_image = Image.open(img_path)

    preprocess = 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]),
    ])

    input_tensor = preprocess(input_image)
    input_batch = input_tensor.unsqueeze(0)
    if torch.cuda.is_available():
        input_batch = input_batch.to('cuda')

    output = VGG16(input_batch)[0]

    L = output.cpu().detach().numpy()

    return idx_1000cls[np.where(L==np.amax(L,axis=0))[0][0]]
```

a test of VGG16_predict function.

[34]:
```
## test VGG16_predict()
idx = 93
output = VGG16_predict(dog_files_short[idx])

# load color (BGR) image
img = cv2.imread(dog_files_short[idx])
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image with prediction,
plt_title="hello, dog! \n your predicted breed is ...\n"+output
plt.title(plt_title)

plt.imshow(cv_rgb)
plt.show()
```



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).

[35]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    breed = VGG16_predict(img_path)

    return breed in idx_1000cls[151 : 268] # true/false

#test on dog_detector
print(dog_detector(human_files_short[9]))
print(dog_detector(dog_files_short[98]))
```

```
False
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:

[36]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
cnt_human_set= 0
cnt_dog_set=0

for i in tqdm (range(100)):

    if (dog_detector(human_files_short[i])):
        cnt_human_set += 1.0

    if (dog_detector(dog_files_short[i])):
        cnt_dog_set += 1.0

print('percentage of the images in human_files_short have a detected dog: ',␣
 ↪cnt_human_set , '%'  )
print('percentage of the images in dog_files_short have a detected dog: ',␣
 ↪cnt_dog_set , '%' )
```

```
100%|     | 100/100 [00:04<00:00, 21.20it/s]

percentage of the images in human_files_short have a detected dog:  1.0 %
percentage of the images in dog_files_short have a detected dog:  98.0 %
```

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

[37]:
```
### (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 `dogImages/train`, `dogImages/valid`, and `dogImages/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!

[38]:
```
import os
import torch
from torchvision import datasets
import torchvision.transforms as transforms
import torch.nn as nn
```

```python
import torch.nn.functional as F
import torch.optim as optim
#from torch.utils.data.sampler import SubsetRandomSampler

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

## some pics with big size,  here when we load the train / valid
## images do need resize the image.
## pictures are domained by single dog
## in most of the case.

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

transform1 = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.CenterCrop(224),
    transforms.ToTensor() ,
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
    ])

transform2 = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.CenterCrop(224),
    transforms.ToTensor() ,
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
    ])



train_data = datasets.ImageFolder('data/dogImages/train',transform=transform1)
valid_data = datasets.ImageFolder('data/dogImages/valid',transform=transform2)
test_data = datasets.ImageFolder('data/dogImages/test',transform=transform2)


classes = train_data.classes
output_classes = len(classes)



#train_idx, valid_idx = len(train_data), len(valid_data)
#print(len(train_data), len(valid_data))
```

```
#train_sampler = SubsetRandomSampler(train_idx)
#valid_sampler = SubsetRandomSampler(valid_idx)

#print(train_sampler, valid_sampler)

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:

- resize -> centercrop -> ToTensor -> Normalize

[39]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import torchvision.transforms.functional as F
# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image



# obtain one batch of training images
dataiter = iter(train_loader)

images,labels  = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
# display  images
for idx in np.arange(20):
    #ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
```

```
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

1.1.8 (IMPLEMENTATION) Model Architecture

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

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

Answer:

in order to debug and run the cells easier, I put all code in one cell.....

here is the architecture of my trainning network.

```
Net(
  (conv1): Conv2d(3, 10, kernel_size=(64, 64), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(10, 40, kernel_size=(25, 25), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(40, 60, kernel_size=(7, 7), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(60, 80, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(80, 100, 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=400, out_features=1200, bias=True)
  (fc2): Linear(in_features=1200, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=200, bias=True)
  (fc4): Linear(in_features=200, out_features=180, bias=True)
  (fc5): Linear(in_features=180, out_features=133, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
  (conv1_bn): BatchNorm2d(10, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2_bn): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3_bn): BatchNorm2d(60, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4_bn): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5_bn): BatchNorm2d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_norm): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
)
```

- conv layers >= 4 the model goes to better
- kernel size for conv1 layer is better with a big value, previously I set like 5, 11, the result was not good.
- from conv2 kernel size is less and less because maxpool will reduce the size of the image.
- fc1 layer should be (400, 1200) I set with (400,2000) and it is overfitting, then step by step 1200 is an acceptable number.
- the coming fc layers I set to be less and less, for saving calculation time.

- with this configure, the model reach 11% at Epoch 48.

```
Epoch: 57   Training Loss: 0.009092     Validation Loss: 0.091950
Validation loss decreased (0.091995 --> 0.091950).  Saving model ...

Test Loss: 3.789912
Test Accuracy: 13% (110/836)
```

[46]:
```
relu = torch.nn.functional.relu

# 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

        # convolutional layer (sees 124x124x3 image tensor)  128x128x3 origin␣
↪img size
        self.conv1 = nn.Conv2d(3, 10, 64, stride=1, padding=1) #32  filters    ␣
↪
        # convolutional layer (sees 62x62x16 tensor)       112x112  maxpooling ␣
↪then 112 channel
        self.conv2 = nn.Conv2d(10, 40, 25, stride=1,  padding=1)#128  filters
        # convolutional layer (sees 56x56x8 tensor)         56x56    maxpooling␣
↪ then 56 channel
        self.conv3 = nn.Conv2d(40, 60, 7, padding=1)#64  filters
        # convolutional layer (sees 56x56x8 tensor)         56x56    maxpooling␣
↪ then 56 channel
        self.conv4 = nn.Conv2d(60, 80, 5, padding=1)#64  filters
        # convolutional layer (sees 56x56x8 tensor)         56x56    maxpooling␣
↪ then 56 channel
        self.conv5 = nn.Conv2d(80, 100, 3,padding=1)#64  filters


        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer
        self.fc1 = nn.Linear(400, 1200) # first value should be same with x.
↪view(x.size(0),value)
        # linear layer
        self.fc2 = nn.Linear(1200, 300)
        # linear layer
        self.fc3 = nn.Linear(300, 200) # first value should be same with x.
↪view(x.size(0),value)
        # linear layer
        self.fc4 = nn.Linear(200, 180) # first value should be same with x.
↪view(x.size(0),value)
        # linear layer
        self.fc5 = nn.Linear(180, output_classes)

        self.dropout = nn.Dropout(0.25)

        #batch normalization
        self.conv1_bn = nn.BatchNorm2d(10)
        self.conv2_bn = nn.BatchNorm2d(40)
        self.conv3_bn = nn.BatchNorm2d(60)
        self.conv4_bn = nn.BatchNorm2d(80)
        self.conv5_bn = nn.BatchNorm2d(100)
        self.batch_norm = nn.BatchNorm1d(num_features=500)


    def forward(self, x):
        ## Define forward behavior
        # add sequence of convolutional and max pooling layers

        x = self.pool(relu(self.conv1_bn(self.conv1(x))))
        x = self.pool(relu(self.conv2_bn(self.conv2(x))))
        x = self.pool(relu(self.conv3_bn(self.conv3(x))))
        x = self.pool(relu(self.conv4_bn(self.conv4(x))))
        x = self.pool(relu(self.conv5_bn(self.conv5(x))))


        # flatten image input
        #x = x.view(-1, 3 * 224 * 224)
        x = x.view(x.size(0), 400 )

        x = self.dropout(relu(self.fc1(x)))
        x = self.dropout(relu(self.fc2(x)))
        x = self.dropout(relu(self.fc3(x)))
        x = self.dropout(relu(self.fc4(x)))

        x = self.fc5(x)
        return x

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

# instantiate the CNN
model_scratch = Net()
print(model_scratch)
# move tensors to GPU if CUDA is available
use_cuda=True
if use_cuda:
    model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 10, kernel_size=(64, 64), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(10, 40, kernel_size=(25, 25), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(40, 60, kernel_size=(7, 7), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(60, 80, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(80, 100, 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=400, out_features=1200, bias=True)
  (fc2): Linear(in_features=1200, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=200, bias=True)
  (fc4): Linear(in_features=200, out_features=180, bias=True)
  (fc5): Linear(in_features=180, out_features=133, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
  (conv1_bn): BatchNorm2d(10, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv2_bn): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv3_bn): BatchNorm2d(60, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv4_bn): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv5_bn): BatchNorm2d(100, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (batch_norm): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
)
```

[47]:
```
### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.05)
#optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.00001)


# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import numpy as np


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
    #use_cuda=false
    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
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the␣
↪model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model␣
↪parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass: compute predicted outputs by passing inputs to the␣
↪model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += ((1/ (batch_idx + 1)) * (loss.data - valid_loss))

        # calculate average losses
        train_loss = train_loss/len(train_loader)
        valid_loss = valid_loss/len(valid_loader)
        # 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(), 'model_scratch.pt')
            valid_loss_min = valid_loss
    # return trained model
    return model

#use_cuda=False
# train the model
model_scratch = train(100, 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: 0.014597         Validation Loss: 0.114918
Validation loss decreased (inf --> 0.114918).  Saving model …
Epoch: 2        Training Loss: 0.014270         Validation Loss: 0.110798
Validation loss decreased (0.114918 --> 0.110798).  Saving model …
Epoch: 3        Training Loss: 0.013894         Validation Loss: 0.109573
Validation loss decreased (0.110798 --> 0.109573).  Saving model …
Epoch: 4        Training Loss: 0.013657         Validation Loss: 0.107377
Validation loss decreased (0.109573 --> 0.107377).  Saving model …
Epoch: 5        Training Loss: 0.013416         Validation Loss: 0.105132
Validation loss decreased (0.107377 --> 0.105132).  Saving model …
Epoch: 6        Training Loss: 0.013270         Validation Loss: 0.104231
Validation loss decreased (0.105132 --> 0.104231).  Saving model …
Epoch: 7        Training Loss: 0.013123         Validation Loss: 0.104230
Validation loss decreased (0.104231 --> 0.104230).  Saving model …
Epoch: 8        Training Loss: 0.013003         Validation Loss: 0.102839
```

```
Validation loss decreased (0.104230 --> 0.102839).  Saving model …
Epoch: 9        Training Loss: 0.012889         Validation Loss: 0.102902
Epoch: 10       Training Loss: 0.012779         Validation Loss: 0.102117
Validation loss decreased (0.102839 --> 0.102117).  Saving model …
Epoch: 11       Training Loss: 0.012691         Validation Loss: 0.101381
Validation loss decreased (0.102117 --> 0.101381).  Saving model …
Epoch: 12       Training Loss: 0.012616         Validation Loss: 0.101447
Epoch: 13       Training Loss: 0.012496         Validation Loss: 0.100582
Validation loss decreased (0.101381 --> 0.100582).  Saving model …
Epoch: 14       Training Loss: 0.012434         Validation Loss: 0.099406
Validation loss decreased (0.100582 --> 0.099406).  Saving model …
Epoch: 15       Training Loss: 0.012383         Validation Loss: 0.100194
Epoch: 16       Training Loss: 0.012302         Validation Loss: 0.098864
Validation loss decreased (0.099406 --> 0.098864).  Saving model …
Epoch: 17       Training Loss: 0.012199         Validation Loss: 0.098320
Validation loss decreased (0.098864 --> 0.098320).  Saving model …
Epoch: 18       Training Loss: 0.012133         Validation Loss: 0.098398
Epoch: 19       Training Loss: 0.012045         Validation Loss: 0.097973
Validation loss decreased (0.098320 --> 0.097973).  Saving model …
Epoch: 20       Training Loss: 0.011996         Validation Loss: 0.097829
Validation loss decreased (0.097973 --> 0.097829).  Saving model …
Epoch: 21       Training Loss: 0.011934         Validation Loss: 0.097554
Validation loss decreased (0.097829 --> 0.097554).  Saving model …
Epoch: 22       Training Loss: 0.011864         Validation Loss: 0.097770
Epoch: 23       Training Loss: 0.011770         Validation Loss: 0.097475
Validation loss decreased (0.097554 --> 0.097475).  Saving model …
Epoch: 24       Training Loss: 0.011743         Validation Loss: 0.096767
Validation loss decreased (0.097475 --> 0.096767).  Saving model …
Epoch: 25       Training Loss: 0.011636         Validation Loss: 0.096270
Validation loss decreased (0.096767 --> 0.096270).  Saving model …
Epoch: 26       Training Loss: 0.011606         Validation Loss: 0.096194
Validation loss decreased (0.096270 --> 0.096194).  Saving model …
Epoch: 27       Training Loss: 0.011531         Validation Loss: 0.094795
Validation loss decreased (0.096194 --> 0.094795).  Saving model …
Epoch: 28       Training Loss: 0.011478         Validation Loss: 0.096299
Epoch: 29       Training Loss: 0.011398         Validation Loss: 0.096146
Epoch: 30       Training Loss: 0.011307         Validation Loss: 0.097210
Epoch: 31       Training Loss: 0.011237         Validation Loss: 0.094656
Validation loss decreased (0.094795 --> 0.094656).  Saving model …
Epoch: 32       Training Loss: 0.011172         Validation Loss: 0.094345
Validation loss decreased (0.094656 --> 0.094345).  Saving model …
Epoch: 33       Training Loss: 0.011090         Validation Loss: 0.093779
Validation loss decreased (0.094345 --> 0.093779).  Saving model …
Epoch: 34       Training Loss: 0.011053         Validation Loss: 0.093759
Validation loss decreased (0.093779 --> 0.093759).  Saving model …
Epoch: 35       Training Loss: 0.010927         Validation Loss: 0.096057
Epoch: 36       Training Loss: 0.010910         Validation Loss: 0.092985
Validation loss decreased (0.093759 --> 0.092985).  Saving model …
```

```
Epoch: 37       Training Loss: 0.010817         Validation Loss: 0.095842
Epoch: 38       Training Loss: 0.010772         Validation Loss: 0.093364
Epoch: 39       Training Loss: 0.010636         Validation Loss: 0.092481
Validation loss decreased (0.092985 --> 0.092481).  Saving model …
Epoch: 40       Training Loss: 0.010585         Validation Loss: 0.093221
Epoch: 41       Training Loss: 0.010488         Validation Loss: 0.095038
Epoch: 42       Training Loss: 0.010410         Validation Loss: 0.097226
Epoch: 43       Training Loss: 0.010286         Validation Loss: 0.093129
Epoch: 44       Training Loss: 0.010213         Validation Loss: 0.094612
Epoch: 45       Training Loss: 0.010150         Validation Loss: 0.093546
Epoch: 46       Training Loss: 0.010064         Validation Loss: 0.092630
Epoch: 47       Training Loss: 0.010002         Validation Loss: 0.092298
Validation loss decreased (0.092481 --> 0.092298).  Saving model …
Epoch: 48       Training Loss: 0.009902         Validation Loss: 0.092866
Epoch: 49       Training Loss: 0.009824         Validation Loss: 0.092971
Epoch: 50       Training Loss: 0.009740         Validation Loss: 0.092770
Epoch: 51       Training Loss: 0.009643         Validation Loss: 0.091995
Validation loss decreased (0.092298 --> 0.091995).  Saving model …
Epoch: 52       Training Loss: 0.009541         Validation Loss: 0.092981
Epoch: 53       Training Loss: 0.009501         Validation Loss: 0.094567
Epoch: 54       Training Loss: 0.009361         Validation Loss: 0.092282
Epoch: 55       Training Loss: 0.009332         Validation Loss: 0.093790
Epoch: 56       Training Loss: 0.009168         Validation Loss: 0.092721
Epoch: 57       Training Loss: 0.009092         Validation Loss: 0.091950
Validation loss decreased (0.091995 --> 0.091950).  Saving model …
Epoch: 58       Training Loss: 0.008996         Validation Loss: 0.094355
Epoch: 59       Training Loss: 0.008965         Validation Loss: 0.093764
Epoch: 60       Training Loss: 0.008872         Validation Loss: 0.092375
Epoch: 61       Training Loss: 0.008750         Validation Loss: 0.093995
Epoch: 62       Training Loss: 0.008762         Validation Loss: 0.093440
Epoch: 63       Training Loss: 0.008557         Validation Loss: 0.093907
Epoch: 64       Training Loss: 0.008466         Validation Loss: 0.094605
Epoch: 65       Training Loss: 0.008437         Validation Loss: 0.091992
Epoch: 66       Training Loss: 0.008250         Validation Loss: 0.097639
Epoch: 67       Training Loss: 0.008242         Validation Loss: 0.094342
Epoch: 68       Training Loss: 0.008048         Validation Loss: 0.093616
Epoch: 69       Training Loss: 0.008025         Validation Loss: 0.094410
Epoch: 70       Training Loss: 0.007978         Validation Loss: 0.094617
Epoch: 71       Training Loss: 0.007812         Validation Loss: 0.095024
Epoch: 72       Training Loss: 0.007742         Validation Loss: 0.096238
Epoch: 73       Training Loss: 0.007691         Validation Loss: 0.093684
Epoch: 74       Training Loss: 0.007518         Validation Loss: 0.096000
Epoch: 75       Training Loss: 0.007483         Validation Loss: 0.095983
Epoch: 76       Training Loss: 0.007415         Validation Loss: 0.096814
Epoch: 77       Training Loss: 0.007463         Validation Loss: 0.094535
Epoch: 78       Training Loss: 0.007197         Validation Loss: 0.095831
Epoch: 79       Training Loss: 0.007148         Validation Loss: 0.099258
Epoch: 80       Training Loss: 0.007100         Validation Loss: 0.098456
```

```
Epoch: 81       Training Loss: 0.006943         Validation Loss: 0.099248
Epoch: 82       Training Loss: 0.006980         Validation Loss: 0.097831
Epoch: 83       Training Loss: 0.006774         Validation Loss: 0.097013
Epoch: 84       Training Loss: 0.006701         Validation Loss: 0.098406
Epoch: 85       Training Loss: 0.006654         Validation Loss: 0.100613
Epoch: 86       Training Loss: 0.006569         Validation Loss: 0.098673
Epoch: 87       Training Loss: 0.006453         Validation Loss: 0.102965
Epoch: 88       Training Loss: 0.006467         Validation Loss: 0.098318
Epoch: 89       Training Loss: 0.006316         Validation Loss: 0.097494
Epoch: 90       Training Loss: 0.006271         Validation Loss: 0.100255
Epoch: 91       Training Loss: 0.006264         Validation Loss: 0.098356
Epoch: 92       Training Loss: 0.006071         Validation Loss: 0.103157
Epoch: 93       Training Loss: 0.006116         Validation Loss: 0.102652
Epoch: 94       Training Loss: 0.005961         Validation Loss: 0.108358
Epoch: 95       Training Loss: 0.005892         Validation Loss: 0.103350
Epoch: 96       Training Loss: 0.005798         Validation Loss: 0.103809
Epoch: 97       Training Loss: 0.005666         Validation Loss: 0.103658
Epoch: 98       Training Loss: 0.005559         Validation Loss: 0.105645
Epoch: 99       Training Loss: 0.005537         Validation Loss: 0.106757
Epoch: 100      Training Loss: 0.005479         Validation Loss: 0.106385
```

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

1.1.9 (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%.

```
[48]: 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: 3.789912


Test Accuracy: 13% (110/836)
```

## 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.

1.1.10 (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.

[49]:
```
## TODO: Specify data loaders
import os
import torch
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
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

## some pics with big size,  here when we load the train / valid
## images do need resize the image.
## pictures are domained by single dog
## in most of the case.

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


transform1 = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.CenterCrop(224),
    transforms.ToTensor() ,
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
    ])

transform2 = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.CenterCrop(224),
    transforms.ToTensor() ,
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
    ])

train_data = datasets.ImageFolder('data/dogImages/train',transform=transform1)
valid_data = datasets.ImageFolder('data/dogImages/valid',transform=transform2)
test_data = datasets.ImageFolder('data/dogImages/test',transform=transform2)


#train_idx, valid_idx = len(train_data), len(valid_data)
#print(len(train_data), len(valid_data))
#train_sampler = SubsetRandomSampler(train_idx)
#valid_sampler = SubsetRandomSampler(valid_idx)
#print(train_sampler, valid_sampler)

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_transfer = {"train":train_loader,
                   "valid":valid_loader,
                   "test":test_loader}
```

1.1.11 (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`.

[50]:
```python
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

model_transfer = models.vgg16(pretrained=True)
# print out the model structure

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

for param in model_transfer.classifier.parameters():
    param.requires_grad = True

n_inputs = model_transfer.classifier[6].in_features
print ( n_inputs )
#last_layer = nn.Linear(n_inputs, 133)
#model_transfer.classifier[6] = last_layer
model_transfer.classifier[6].out_features = 133

print(model_transfer.classifier[6].out_features)
use_cuda=True

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

```
4096
133
```

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

Answer:

- load vgg16 model pretrained mode
- freeze training for all features layers
- replace the last linear layer to my classes numbers

1.1.12 (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.

[54]:
```
import torch.optim as optim
import numpy as np
# specify loss function (categorical cross-entropy)
criterion_transfer = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.001
optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001)
#optimizer_transfer = optim.Adam(model_transfer.parameters(), lr=0.0001)
n_epochs = 10


def transfer_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
    #use_cuda=false
    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
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the␣
↪model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model␣
↪parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass: compute predicted outputs by passing inputs to the␣
↪model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += ((1/ (batch_idx + 1)) * (loss.data - valid_loss))

        # calculate average losses
        train_loss = train_loss/len(train_loader)
        valid_loss = valid_loss/len(valid_loader)
        # 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(), 'model_transfer.pt')
            valid_loss_min = valid_loss
    # return trained model
    return model
```

1.1.13 (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'`.

[55]:

```
# train the model
model_transfer = transfer_train(n_epochs, 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: 0.002797         Validation Loss: 0.014792
Validation loss decreased (inf --> 0.014792).  Saving model …
Epoch: 2        Training Loss: 0.002541         Validation Loss: 0.014431
Validation loss decreased (0.014792 --> 0.014431).  Saving model …
Epoch: 3        Training Loss: 0.002436         Validation Loss: 0.013732
Validation loss decreased (0.014431 --> 0.013732).  Saving model …
Epoch: 4        Training Loss: 0.002316         Validation Loss: 0.013602
Validation loss decreased (0.013732 --> 0.013602).  Saving model …
Epoch: 5        Training Loss: 0.002185         Validation Loss: 0.012824
Validation loss decreased (0.013602 --> 0.012824).  Saving model …
Epoch: 6        Training Loss: 0.002107         Validation Loss: 0.012297
Validation loss decreased (0.012824 --> 0.012297).  Saving model …
Epoch: 7        Training Loss: 0.002027         Validation Loss: 0.012205
Validation loss decreased (0.012297 --> 0.012205).  Saving model …
Epoch: 8        Training Loss: 0.001973         Validation Loss: 0.012267
Epoch: 9        Training Loss: 0.001866         Validation Loss: 0.011985
Validation loss decreased (0.012205 --> 0.011985).  Saving model …
Epoch: 10       Training Loss: 0.001795         Validation Loss: 0.011750
Validation loss decreased (0.011985 --> 0.011750).  Saving model …
```

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

1.1.14 (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%.

```
[107]: 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))


test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.559823


Test Accuracy: 83% (694/836)
```

1.1.15 (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.

[155]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
classes = train_data.classes
output_classes = len(classes)
#print(classes)
# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in train_data.classes]
#print(class_names)

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed

    img = Image.open(img_path)
    #plt.imshow(img)
    img_tensor = transform2(img)
    img_batch = img_tensor.unsqueeze_(0)

    use_gpu = torch.cuda.is_available()
    if use_gpu :
        img = img_batch.to('cuda')

    model_transfer.eval()
    output = model_transfer(img)
    _, preds_tensor = torch.max(output, 1)
    preds = np.squeeze(preds_tensor.numpy()) if not use_gpu else np.
↪squeeze(preds_tensor.cpu().numpy())
    print(preds)

    if(preds > 133):
        return idx_1000cls[preds]
    else:
        return class_names[preds]




img_path='/home/jinshengye/Desktop/1.jpg'
output = predict_breed_transfer(img_path)
print (output)
```

```
903
wig
```

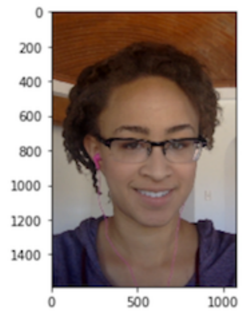
## Step 5: Write your Algorithm

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 `dog_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
```

1.1.16 (IMPLEMENTATION) Write your Algorithm

[159]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    if dog_detector(img_path) :
        output = predict_breed_transfer(img_path)
        print('dog breed is --> ' + str(output))
        img = Image.open(img_path)
        #img = cv2.imread(img_path)
        #img = cv2.cvtColor( img, cv2.COLOR_BGR2RGB)
        plt.imshow(img)
        plt.show()

    elif face_detector(img_path):
        output = predict_breed_transfer(img_path)
        print('Hi human, you look like --> a person with: ' + str(output))
        img = Image.open(img_path)
        plt.imshow(img)
        plt.show()

    else :
        print('you should enter a human or dog image :)')
```

## 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?

1.1.17 (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.

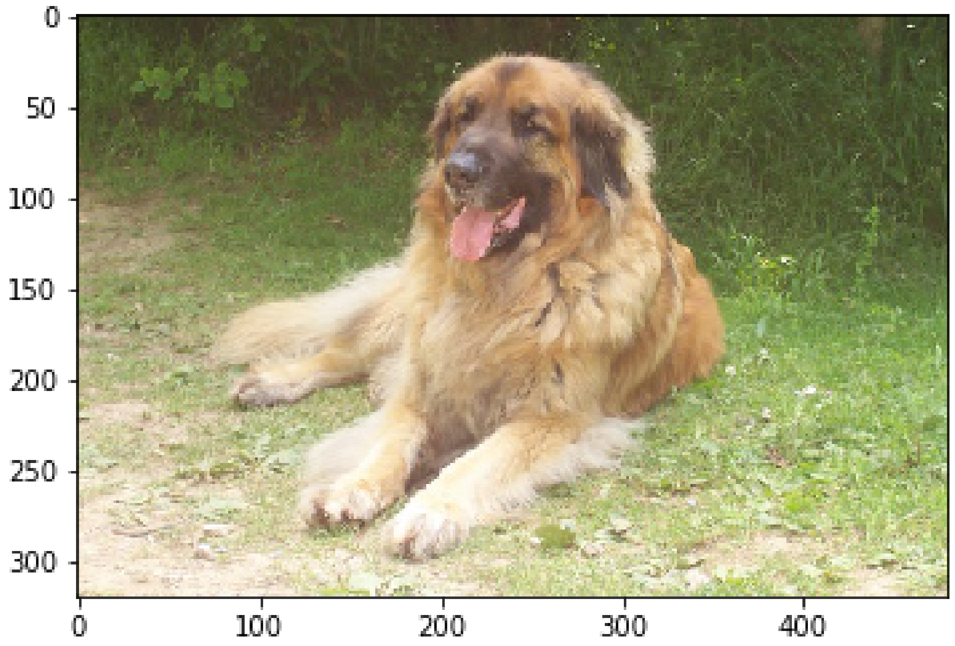
Answer: (Three possible points for improvement)

for dog classification it works fine. however, when we goes to human image, it shows error message like: **IndexError: list index out of range** Then I check the index value it seems the value returns to vgg 1000 classes. the classes can be like: wig, lab_coat, ping-pong_ball windsor_tie… etc. just not dogs or human…

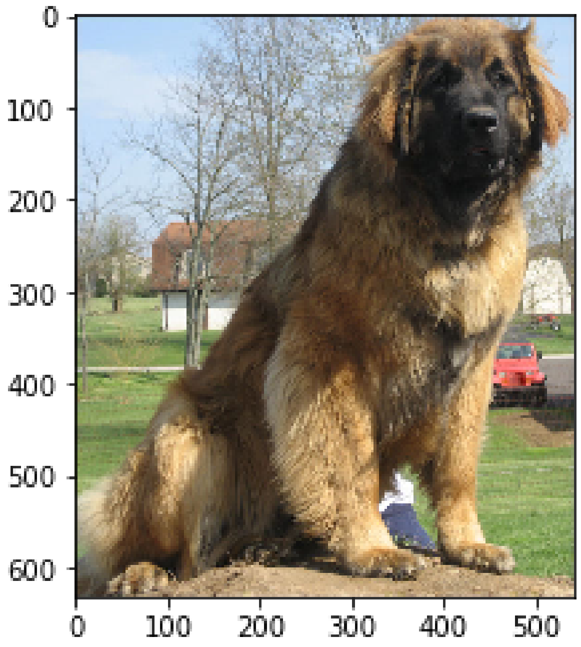
- add a human class to database
- transfer the model to a human detection CNN
- add some noise human pictures to our database

[160]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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

for file in np.hstack(dog_files[:2]):
    run_app(file)


for file in np.hstack(human_files[:23]):
    run_app(file)
```

```
97
dog breed is --> Leonberger
```

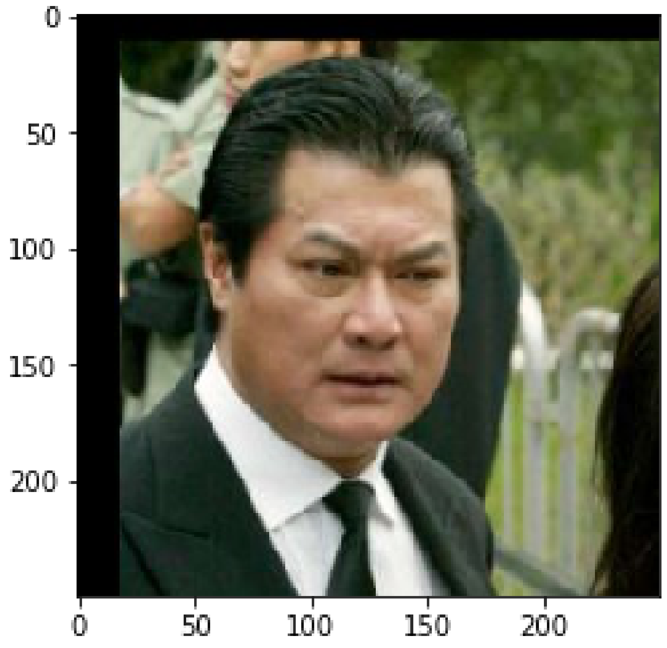


```
97
dog breed is --> Leonberger
```



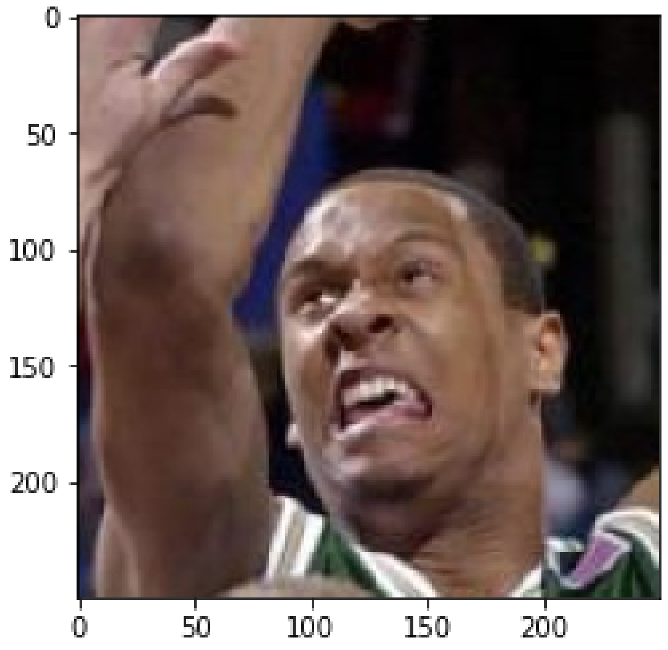
```
834
```

```
Hi human, you look like --> a person with: suit
```

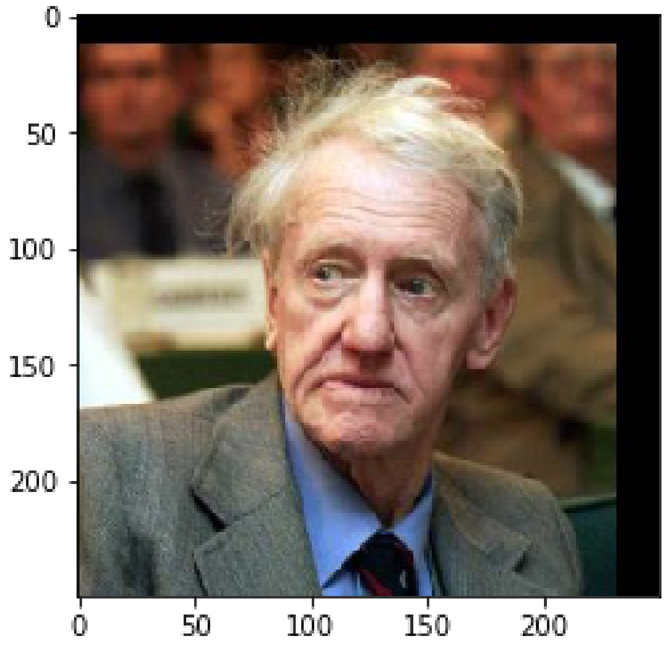


```
0
Hi human, you look like --> a person with: Affenpinscher
```

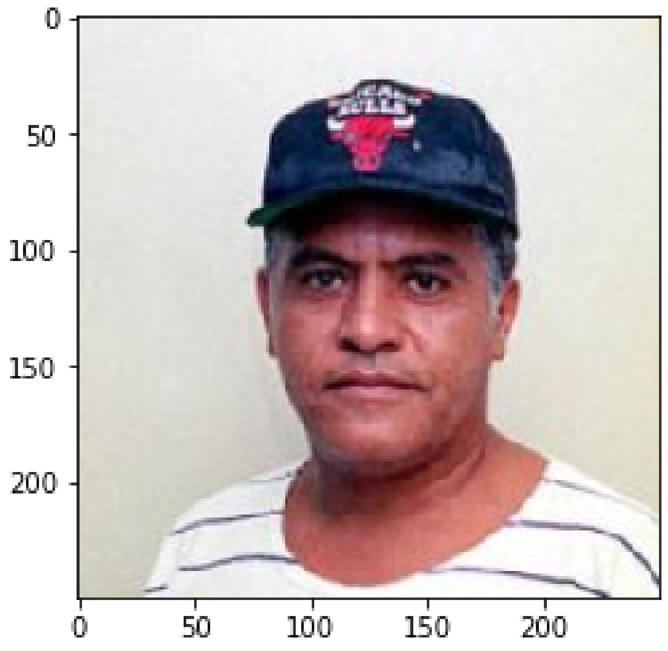
```
430
Hi human, you look like --> a person with: basketball
```



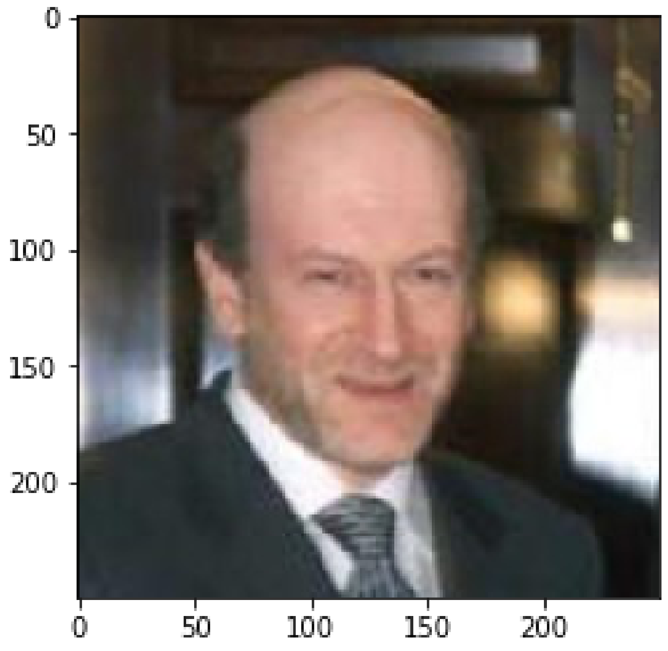
```
834
Hi human, you look like --> a person with: suit
```



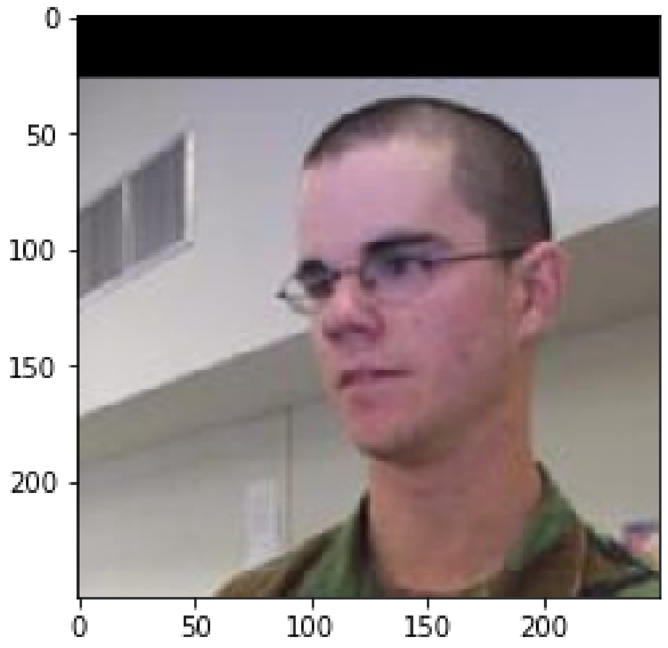
```
981
Hi human, you look like --> a person with: ballplayer
```



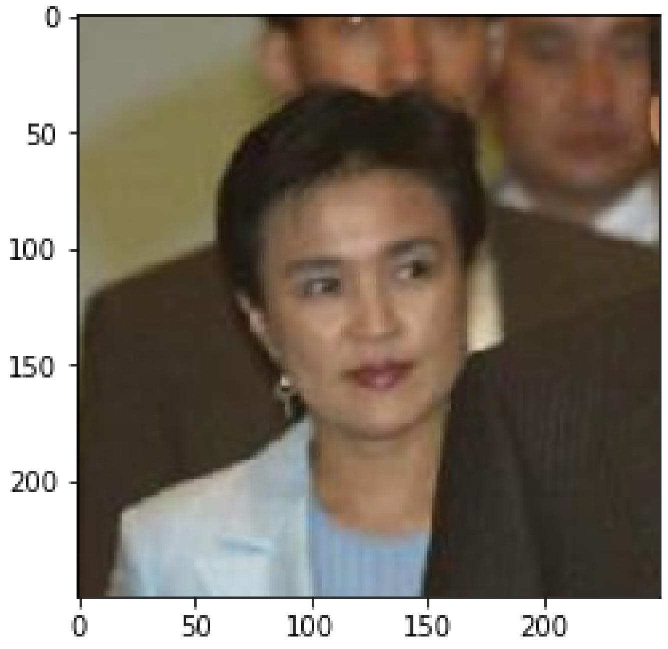
```
906
Hi human, you look like --> a person with: Windsor_tie
```



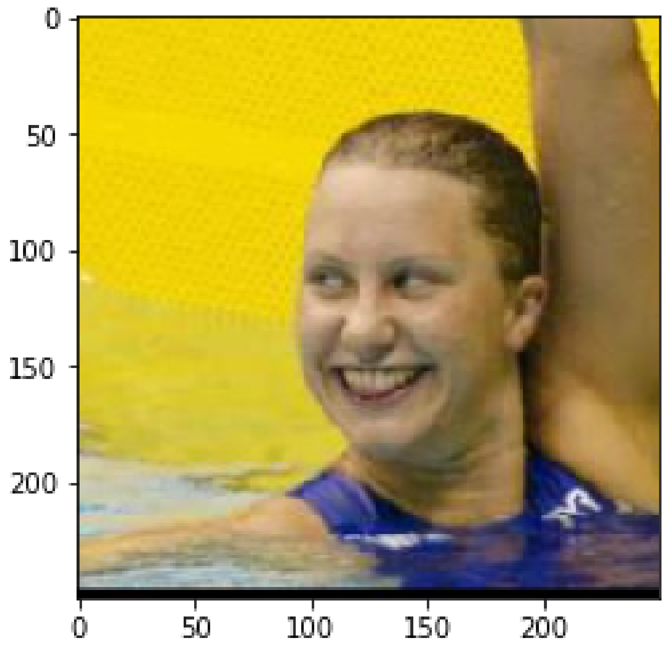
```
906
Hi human, you look like --> a person with: Windsor_tie
```



```
906
Hi human, you look like --> a person with: Windsor_tie
```

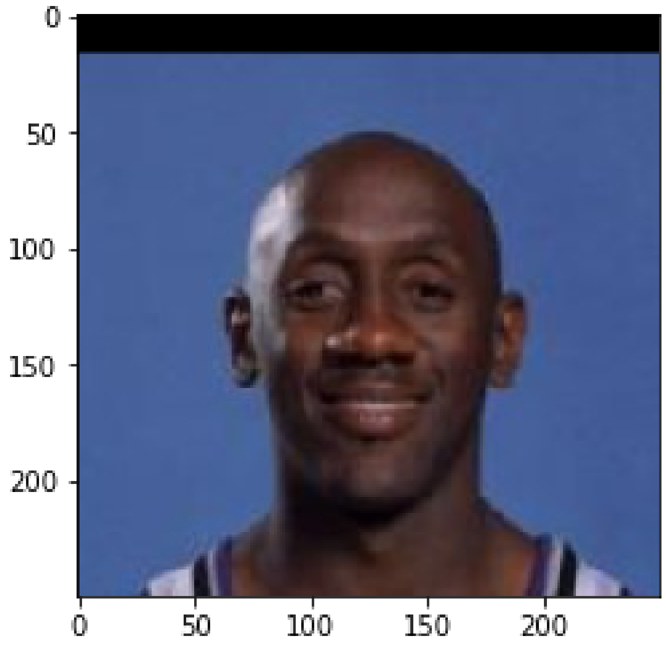


```
416
Hi human, you look like --> a person with: balance_beam
```

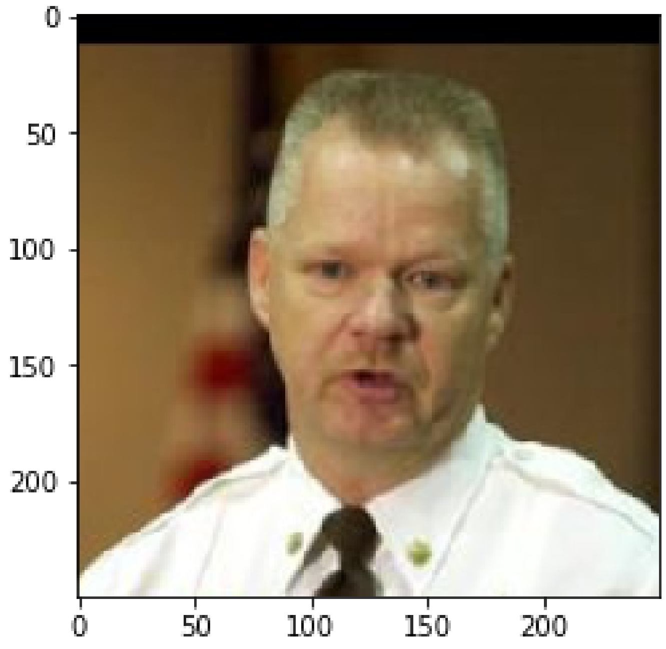


```
465
```

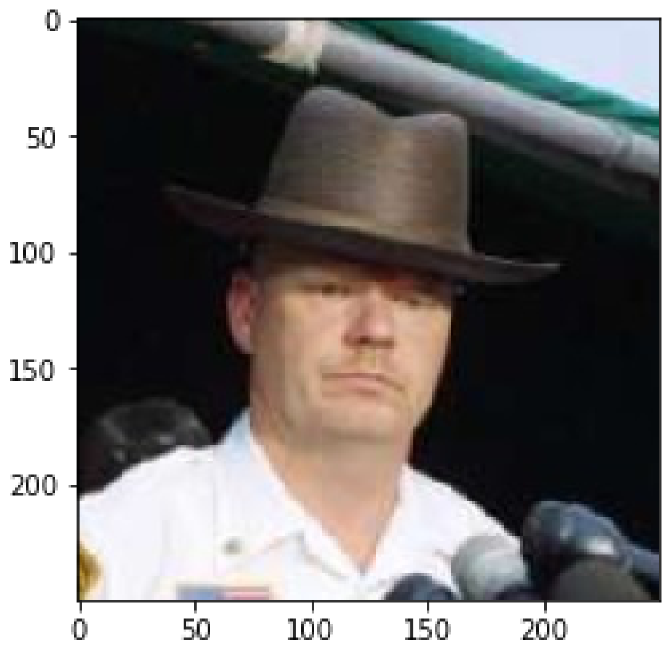
```
Hi human, you look like --> a person with: bulletproof_vest
```



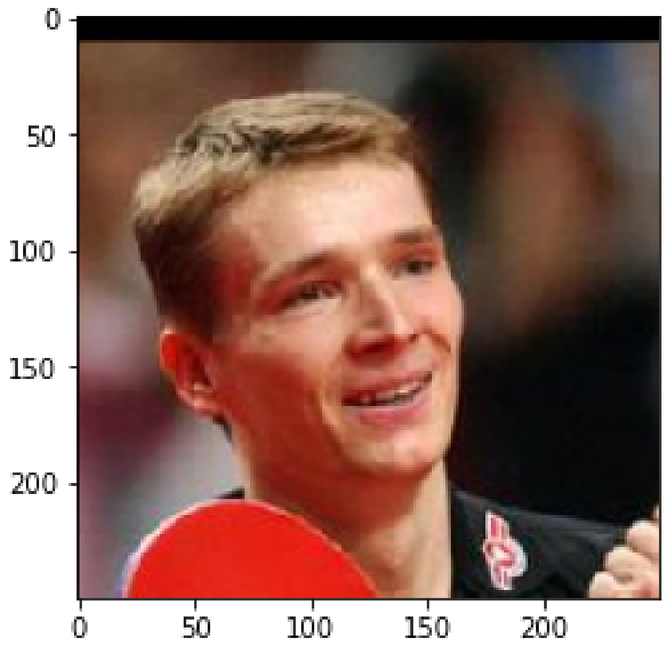
```
617
Hi human, you look like --> a person with: lab_coat
```



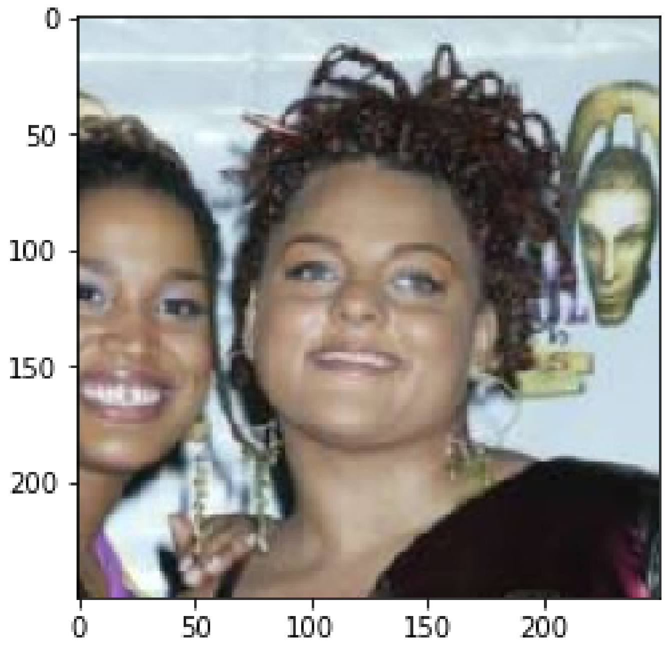
```
981
Hi human, you look like --> a person with: ballplayer
```



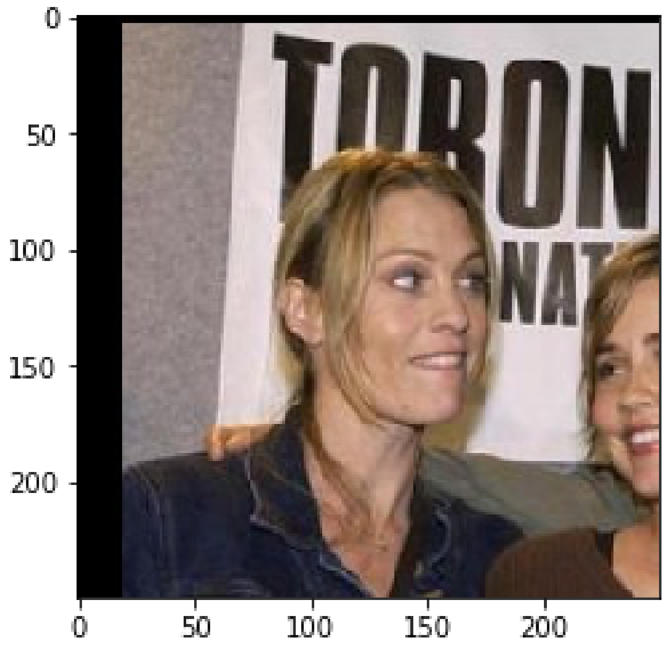
```
722
Hi human, you look like --> a person with: ping-pong_ball
```



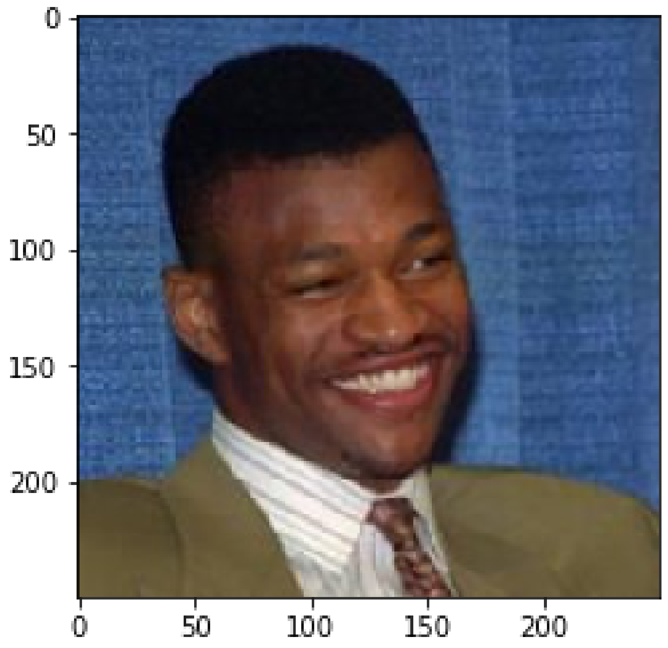
```
903
Hi human, you look like --> a person with: wig
```



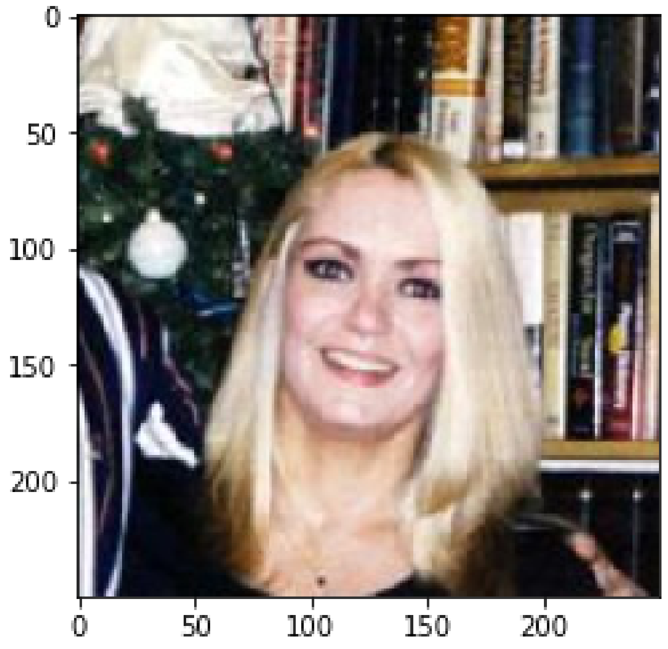
```
678
Hi human, you look like --> a person with: neck_brace
```



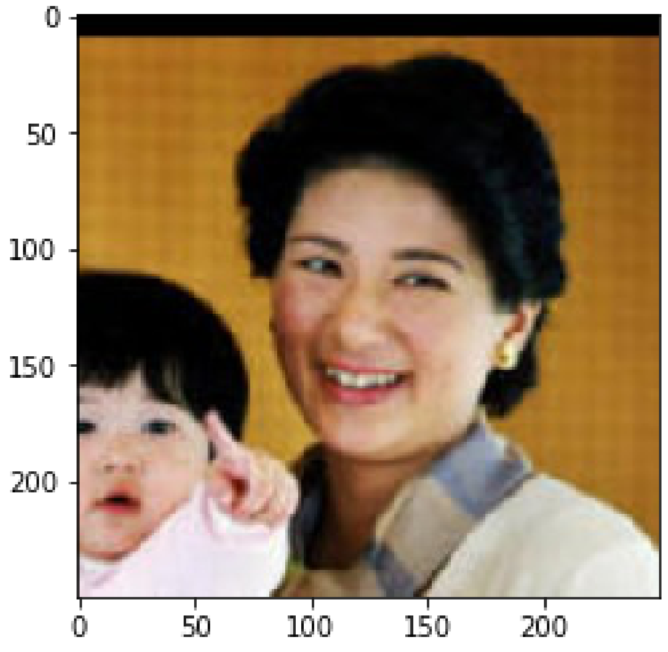
```
906
Hi human, you look like --> a person with: Windsor_tie
```



```
903
Hi human, you look like --> a person with: wig
```

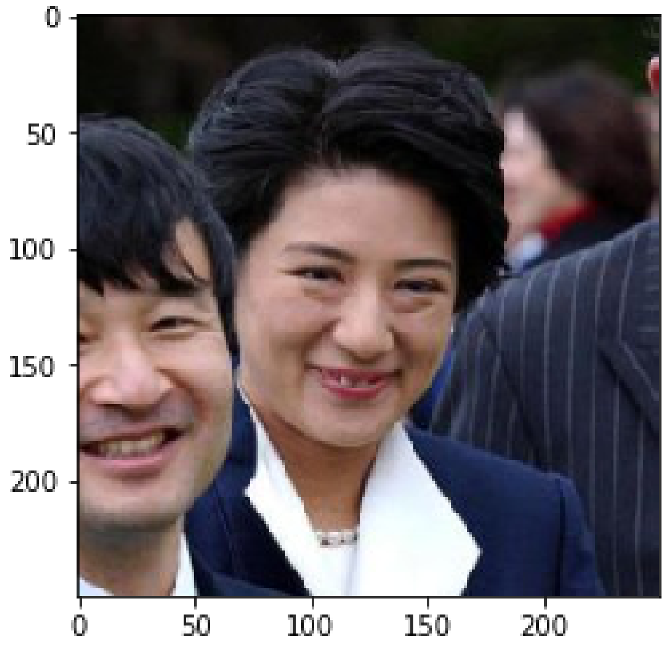


```
470
Hi human, you look like --> a person with: candle
```

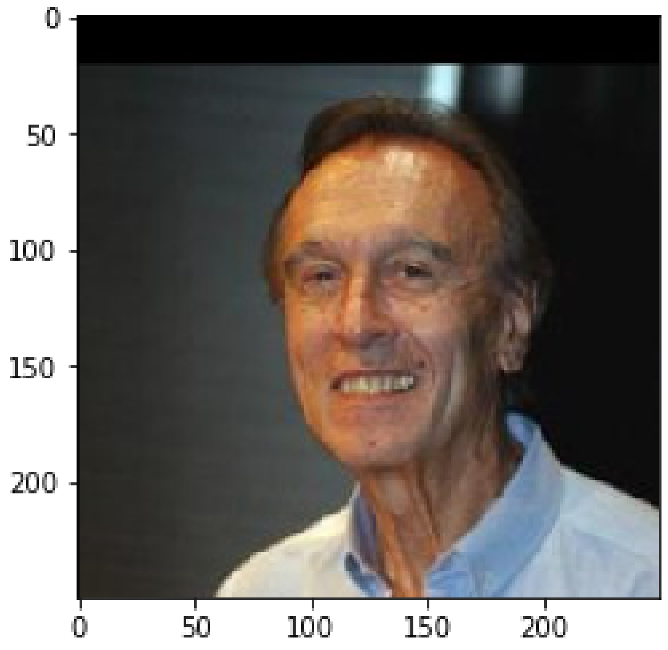


```
982
```

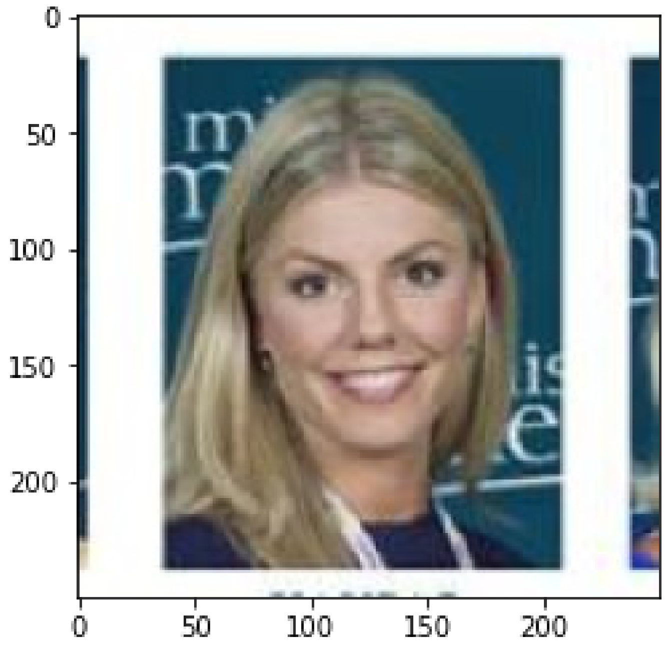
```
Hi human, you look like --> a person with: groom
```



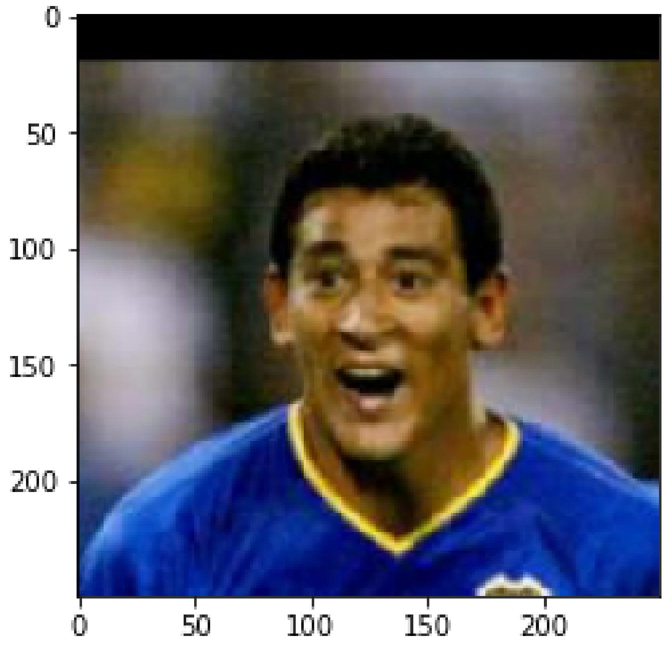
```
617
Hi human, you look like --> a person with: lab_coat
```



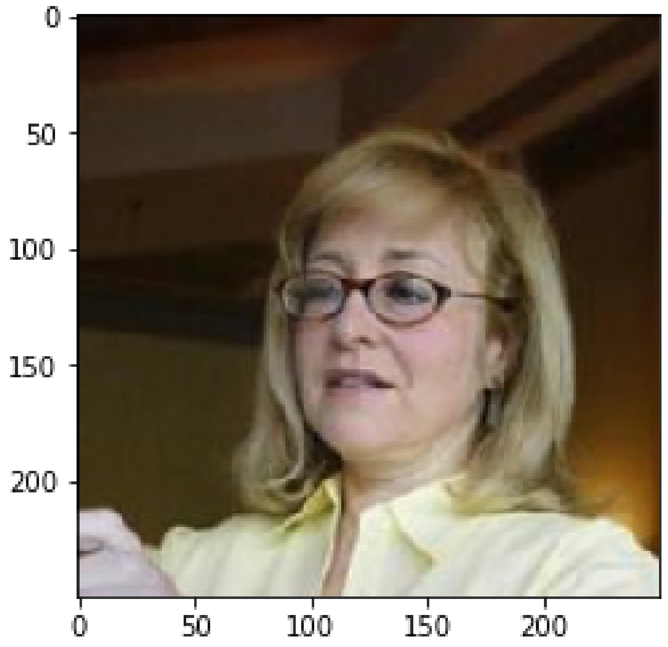
```
921
Hi human, you look like --> a person with: book_jacket
```



```
722
Hi human, you look like --> a person with: ping-pong_ball
```



```
617
Hi human, you look like --> a person with: lab_coat
```



```
[ ]:
```