Intro. To Artificial Intelligence

Face recognition on image, video, and webcam using HOG / CNN

1. A brief introduction to the problem:

During the second semester of my freshman year, COVID outbreaks, and some of the students that signed up for service-learning was arranged to guard the only entrance of engineering building. We had to made sure everyone swipes their ID cards to enter the building.

And here's the question that popped out in my mind : "Couldn't we just reuse the infrared camera to record people who entered the building?"

In this project, I want to know:

1. Is it possible to train a averagely good model with only one picture(specifically, the one on our ID cards)? What is the accuracy using HOG and CNN(only image) respectively?
2. How does the accuracy differ in image, video, and real-time webcam video.

2. A brief introduction to the GitHub code you refer to:

There's no code on GitHub that satisfied my request of use, but I did find a video that introduces how to use the face_recognition library on images and some useful sample code in the creator's blog.

The blog explains a general installation of dependencies. Then it introduces how to load datas in local machine and to label all the unknown faces with the library.

Here are the sample codes and brief introductions about them(detailed explanation are commented in the code)

```python
import face_recognition
import os
import cv2


KNOWN_FACES_DIR = 'known_faces'
UNKNOWN_FACES_DIR = 'unknown_faces'
TOLERANCE = 0.6
FRAME_THICKNESS = 3
FONT_THICKNESS = 2
MODEL = 'cnn'  # default: 'hog', other one can be 'cnn' - CUDA accelerated (if available) deep-learning pretrained model
```

Figure 2.1 Importing the essential dependencies and the face_recognition library. And setting up directories and parameters.

```python
# Returns (R, G, B) from name
def name_to_color(name):
    # Take 3 first letters, tolower()
    # lowercased character ord() value rage is 97 to 122, substract 97, multiply by 8
    color = [(ord(c.lower())-97)*8 for c in name[:3]]
    return color
```

Figure 2.2 A fancy function that calculates color(in the form of (R, G, B)) from name, so each name has its unique color.

```python
print('Loading known faces...')
known_faces = []
known_names = []

# We oranize known faces as subfolders of KNOWN_FACES_DIR
# Each subfolder's name becomes our label (name)
for name in os.listdir(KNOWN_FACES_DIR):

    # Next we load every file of faces of known person
    for filename in os.listdir(f'{KNOWN_FACES_DIR}/{name}'):

        # Load an image
        image = face_recognition.load_image_file(f'{KNOWN_FACES_DIR}/{name}/{filename}')

        # Get 128-dimension face encoding
        # Always returns a list of found faces, for this purpose we take first face only (assuming one face per image as you can't be twice on one image)
        encoding = face_recognition.face_encodings(image)[0]

        # Append encodings and name
        known_faces.append(encoding)
        known_names.append(name)
```

Figure 2.3 Start loading the images in KNOWN_FACES_DIR, and appending the encodings and labels of the image to two lists(known_faces and known_names)

```python
print('Processing unknown faces...')
# Now let's loop over a folder of faces we want to label
for filename in os.listdir(UNKNOWN_FACES_DIR):

    # Load image
    print(f'Filename {filename}', end='')
    image = face_recognition.load_image_file(f'{UNKNOWN_FACES_DIR}/{filename}')

    # This time we first grab face locations - we'll need them to draw boxes
    locations = face_recognition.face_locations(image, model=MODEL)

    # Now since we know loctions, we can pass them to face_encodings as second argument
    # Without that it will search for faces once again slowing down whole process
    encodings = face_recognition.face_encodings(image, locations)

    # We passed our image through face_locations and face_encodings, so we can modify it
    # First we need to convert it from RGB to BGR as we are going to work with cv2
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
```

Figure 2.4 Start loading the images in UNKNOWN_FACES_DIR, grabbing the location of faces in the image, and encoding the according faces.

```python
    # But this time we assume that there might be more faces in an image - we can find faces of dirrerent people
    print(f', found {len(encodings)} face(s)')
    for face_encoding, face_location in zip(encodings, locations):

        # We use compare_faces (but might use face_distance as well)
        # Returns array of True/False values in order of passed known_faces
        results = face_recognition.compare_faces(known_faces, face_encoding, TOLERANCE)

        # Since order is being preserved, we check if any face was found then grab index
        # then label (name) of first matching known face withing a tolerance
        match = None
        if True in results:  # If at least one is true, get a name of first of found labels
            match = known_names[results.index(True)]
            print(f' - {match} from {results}')

            # Each location contains positions in order: top, right, bottom, left
            top_left = (face_location[3], face_location[0])
            bottom_right = (face_location[1], face_location[2])

            # Get color by name using our fancy function
            color = name_to_color(match)

            # Paint frame
            cv2.rectangle(image, top_left, bottom_right, color, FRAME_THICKNESS)

            # Now we need smaller, filled grame below for a name
            # This time we use bottom in both corners - to start from bottom and move 50 pixels down
            top_left = (face_location[3], face_location[2])
            bottom_right = (face_location[1], face_location[2] + 22)
```

```python
            # Paint frame
            cv2.rectangle(image, top_left, bottom_right, color, cv2.FILLED)

            # Wite a name
            cv2.putText(image, match, (face_location[3] + 10, face_location[2] + 15), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (200, 200, 200), FONT_THICKNESS)
```

Figure 2.5 For each face in the image, use .compare_faces() to compare to each known_faces by its encodings. And draw a rectangle around the face and put the corresponding label under it.

```python
    # Show image
    cv2.imshow(filename, image)
    cv2.waitKey(0)
    cv2.destroyWindow(filename)
```

Figure 2.6 Show the labeled image

3. My algorithm, provide details and differences from the reference code:

Since we are comparing the different accuracies between image, videos, and webcam, I modified it into three different codes for each(face_rec_image, face_rec_video, face_rec_webcam), and I'll introduce them sequentially.

Details and differences for face_rec_image.py :

```python
print("loading known faces")
known_faces = []
known_names = []

for name in os.listdir(KNOWN_FACES_DIR):
    # Next we load every file of faces of known person
    for filename in os.listdir(f'{KNOWN_FACES_DIR}/{name}'):
        # Load an image
        image = face_recognition.load_image_file(f'{KNOWN_FACES_DIR}/{name}/{filename}')
        # Get 128-dimension face encoding
        # Always returns a list of found faces, for this purpose we take first face only (assuming one face per image as you can't be twice on one image)
        temp_encoding = face_recognition.face_encodings(image)
        if len(temp_encoding) >0 :
            encoding = temp_encoding[0]
        else:
            print("no face found in", name, filename)
            quit()
        # Append encodings and name
        known_faces.append(encoding)
        known_names.append(name)
```

Figure 3.1.1 Looping through all the images in KNOWN_FACES_DIR and append their corresponding encoding and label to known_faces and known_names.

I added in an if-else statement so that it prints out a string when there's no face in the image or the model can't find one. The original code is troublesome because it accesses the 0th element without checking whether it's valid.

```python
print('Processing unknown faces...')
# Now let's loop over a folder of faces we want to label
for filename in os.listdir(UNKNOWN_FACES_DIR):
    # Load image
    print(f'Filename {filename}', end='')
    image = face_recognition.load_image_file(f'{UNKNOWN_FACES_DIR}/{filename}')
    temp_image = image
    image = cv2.resize(image, (0, 0), None, 0.9, 0.9) # Scaling the image to try to speed up the computation
    # This time we first grab face locations - we'll need them to draw boxes
    locations = face_recognition.face_locations(image, model=MODEL)
    # Now since we know loctions, we can pass them to face_encodings as second argument
    # Without that it will search for faces once again slowing down whole process
    encodings = face_recognition.face_encodings(image, locations)
    # We passed our image through face_locations and face_encodings, so we can modify it
    # First we need to convert it from RGB to BGR as we are going to work with cv2
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
```

Figure 3.1.2 Looping through all the images in UNKNOWN_FACES_DIR, grabbing the locations of faces in the image, and encode the faces.

I added in a scaling because pictures nowadays are extremely huge and in high-quality. By scaling, the program could speed up about 20% and merely affect the results.

```python
for face_encoding, face_location in zip(encodings, locations):
    # We use compare_faces (but might use face_distance as well)
    # Returns array of True/False values in order of passed known_faces
    results = face_recognition.compare_faces(known_faces, face_encoding, TOLERANCE)
    # Returns array of values in order of passed known_faces
    faceDis = face_recognition.face_distance(known_faces, face_encoding)
    # Since order is being preserved, we check if any face was found then grab index
    # then label (name) of first matching known face withing a tolerance
    match = None
    if True in results:  # If at least one is true, get a name of the least distance
        matchIndex = np.argmin(faceDis)
        match = known_names[matchIndex]
        print(f' - {match} from {results}')
    else:  # If there's none similar faces, name he/she 'unknown'
        match = str("unknown")
    # print distance of the unknown_faces between each known_faces
    print(faceDis)
    # Each location contains positions in order: top, right, bottom, left
    top_left = (face_location[3], face_location[0])
    bottom_right = (face_location[1], face_location[2])
    # Get color by name using our fancy function
    color = name_to_color(match)
    # Paint frame
    cv2.rectangle(image, top_left, bottom_right, color, FRAME_THICKNESS)
    # Now we need smaller, filled grame below for a name
    # This time we use bottom in both corners - to start from bottom and move 50 pixels down
    top_left = (face_location[3], face_location[2])
    bottom_right = (face_location[1], face_location[2] + 22)
    # Paint frame
    cv2.rectangle(image, top_left, bottom_right, color, cv2.FILLED)
    # Wite a name
    cv2.putText(image, match, (face_location[3] + 10, face_location[2] + 15), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (200, 200, 200), FONT_THICKNESS)
```

Figure 3.1.3 From the encodings and locations, compare one face to one face. Using the compare_faces(), we can get the Boolean value whether the face is similar or not. And draw a rectangle around the face with the label under it.

I deleted all original statement and added in a completely new if-else statement. The original code didn't take into consider of the case when there's no TRUE value in results, and it matches the unknown face with the label of the first TRUE value in results, which is also really troublesome, especially in a large-scale database. Therefore, I brought in a new face_distance() function, this function calculates the distance between the unkown face and other known faces encodings. Then I use the argmin function to get the label which has the least distance, label the unkown face. And if there's no TRUE value in results, I name the unknown face "unknown".

```python
# Show image
imageS = cv2.resize(image, (0, 0), None, 0.75, 0.75) # Scaling the image b/c some of the images are too large to fit in the screen
cv2.namedWindow(filename, cv2.WINDOW_NORMAL) # Let the window be adjustable
cv2.imshow(filename, imageS)
# Overlooking if there's key pressed
key = cv2.waitKey(0)
if key == ord('q') or key == 27: # Esc
    print('halting face_rec')
    sys.exit()
cv2.destroyWindow(filename)
```

Figure 3.1.4 Showing the labeled images.

I added in another scaling due to the size of the image, and I also put a nameWindow() function to let the output window be adjustable. Also, I inserted a if statement because the program sometimes might crash unexpectedly or we simply don't need to see all the results.

Details and differences for face_rec_video.py :

Due to the great working of face_rec_image.py, I only need to delete/modify some useless sentences and make some of them satisfy how videos operates from my own code.

```python
import face_recognition
import os
import cv2
import numpy as np

KNOWN_FACES_DIR = 'known_faces'
TOLERANCE = 0.45
FRAME_THICKNESS = 3
FONT_THICKNESS = 2
MODEL = 'hog'  # default: 'hog', other one can be 'cnn' - CUDA accelerated (if available) deep-learning pretrained model

video = cv2.VideoCapture("hwaa.mp4")
```

Figure 3.2.1 Importing the essential dependencies and the face_recognition library, setting up directories and parameters, and load in the video file.

I simply remove the UNKNOWN_FACE_DIR and added in the VideoCapture() here.

```python
print('Processing unknown faces...')
# Now let's loop over a folder of faces we want to label
while True:
    ret, image = video.read()
    # This time we first grab face locations - we'll need them to draw boxes
    locations = face_recognition.face_locations(image, model=MODEL)
    # Now since we know loctions, we can pass them to face_encodings as second argument
    # Without that it will search for faces once again slowing down whole process
    encodings = face_recognition.face_encodings(image, locations)
    # But this time we assume that there might be more faces in an image - we can find faces of dirrerent people
    for face_encoding, face_location in zip(encodings, locations):
        # We use compare_faces (but might use face_distance as well)
        # Returns array of True/False values in order of passed known_faces
        results = face_recognition.compare_faces(known_faces, face_encoding, TOLERANCE)
```

Figure 3.2.2 Showing the video frame by frame and label the faces in the video.

For the case of video, we don't have image to loop from, so I modified the for loop into a while loop. And because we don't have the unknown face image, we have to grab it from the video, here's where the function read() comes in. The function grabs the image frame by frame so that we can put it in the face recognition model and get its label.

Details and differences for face_rec_webcam.py :

This code is almost the same as face_rec_webcam.py, we only modify the source of the video, and the rest works just the same.

```python
video = cv2.VideoCapture(0) # webcam on my local = 0, maybe differ in local machines
```

Figure 3.3.1 Connect the video source to the local webcam device.

4. Experiment results, note how the results are conducted, and think about the experiment setting and quality measure:

The accuracy is conducted as followings:

1. Image accuracy(%): (faces recognized correctly) / (faces recognized correctly + faces recognized incorrectly + faces missed)
2. Video accuracy(%): (total time of full faces recognized correctly) / (total time of each appearing full face)
3. Webcam accuracy(%): (total time of full faces recognized correctly) / (total time of each appearing full face)

The time is measured by myself using a phone stopwatch, this maybe somehow troublesome, but it’s hard to find a better way to measure time, so this can only give a rough idea on how the algorithm works.

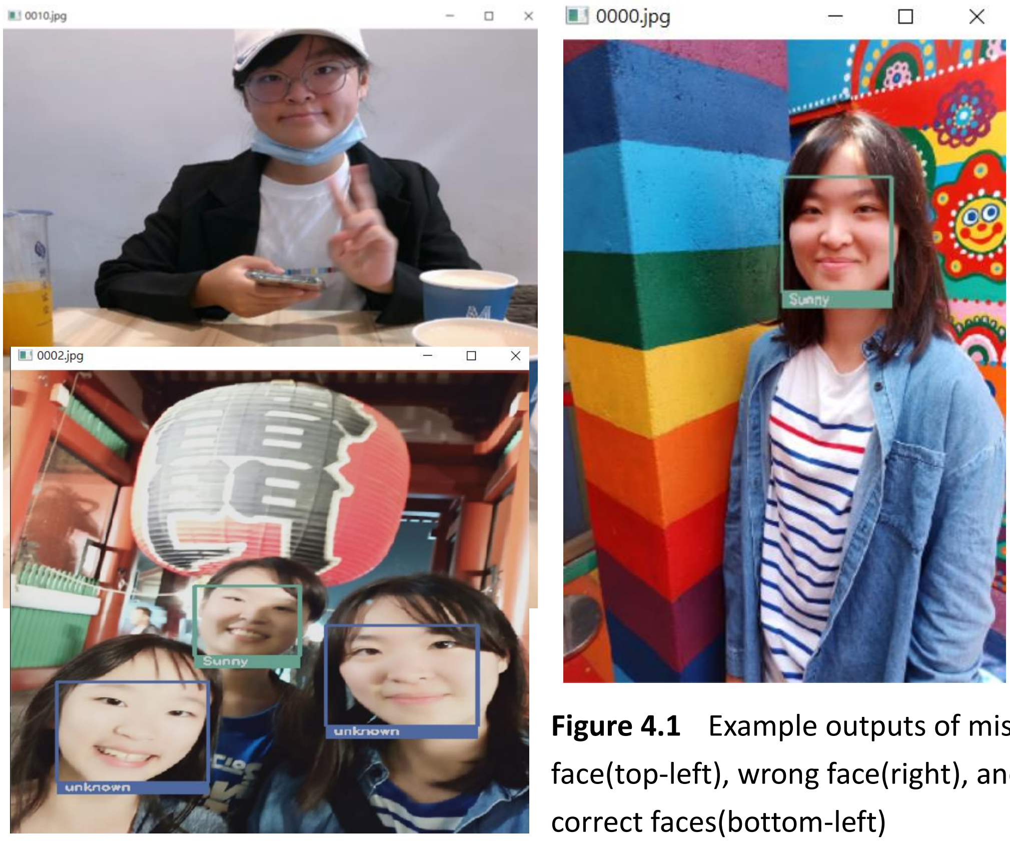

Figure 4.1 Example outputs of missed face(top-left), wrong face(right), and correct faces(bottom-left)

Results of using HOG :

a. Image accuracy: 18 / (18 + 4 + 6) ≒ 64.3%
b. Video accuracy: 29.66 / 1:09.65 ≒ 42.6%
c. Real-time webcam accuracy: 5.97 / 21.64 ≒ 27.6%

Results of using CNN :

a. Image accuracy: 22 / (22 + 6 + 0) ≒ 78.6%

The reason we use CNN only on images is because my local machine is not good enough to compute CNN on videos and real-time webcams. Although I’m also really curious about the result due to its good performance on images. Maybe when I have a chance to get a better computer, I can try run on its GPU.

Summary :

1. From the results I got from using HOG, I would say it’s really hard to get a great accuracy if we only train the model with one image. But for CNN, it might be very promising to get a 50% accuracy. So the answer to this question is ‘yes’, it’s possible to train a good model using one training image.
2. The accuracy decreases significantly between three different sources, and for images, CNN performs much better than HOG.