Lab-1 (Part-2)

Camera-Based Motion Detection

Lab Goal:

By the end of this lab, you will have created a basic security camera using your raspberry pi.

Components Required:

Raspberry Pi with Pi Camera, SenseHat, and Power Supply

Background:

In this lab, we will interface with the camera module and apply Open-CV's HAAR Cascade Model to track human faces. Additionally, you will learn the basics of background subtraction and implement it as a new feature of your Pi spy camera.

Install pre-reqs on your Pi:

```
>> sudo apt install -y python3-opencv opencv-data
```

Building a Surveillance Camera:

First, let us show a live camera feed from the raspberry pi's camera

1. download and open **Picamera2Capture.py** script.
2. The code is written to continuously capture a live camera feed from the Raspberry Pi Camera
3. Press the spacebar to save images in the same directory as the python file. Make sure to click on the window with the camera feed to ensure your space key presses are being registered correctly.
4. To exit the program, enter input command 'q'.

Finding and Tracking Faces

In this part, we will use the OpenCV's face detection model, specifically the Haar Cascade model, to detect and track faces from the live video streams from Raspberry Pi camera. OpenCV Haar Cascade is a machine learning object detection method used to identify objects or features in images or video.
Specifically, it's a classifier designed for detecting objects with a particular shape or pattern, and it's widely used for tasks like face detection.

1. Download the **haarcascade_frontalface_default.xm**l file from the following:

https://github.com/kipr/opencv/tree/master/data/haarcascades

This model has been trained to detect faces. It's worth noting that OpenCV's HaarCascade library offers other pretrained models for tasks like tracking eyes, detecting the human body, identifying smiles, and more.

2. Add the following line before the while loop and copy the path where the .xml file is saved. Add the absolute path to the file in the CascadeClassifier function as shown below:

```
faceCascade=cv2.CascadeClassifier("/path/to/haarcascade_frontalface_default.xml")
```

Inside the while loop,add the following lines after the frame capturing:

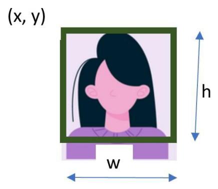
```
frameGray=cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)

faces=faceCascade.detectMultiScale(frameGray,1.3,5)
```

This calls the detector function, i.e. , detectMultiScale function once the .xml file is loaded. This function has three parameters; image, scaleFactor, and minNeighbours.

- *image:* Gray scaled image/frame
- *scaleFactor*:Parameter value for how much image size is reduced at each image scale. A smaller scaleFactor implies reducing the size of the image by a small amount to do more accurate object detection as it moves across different scales. Preferable values are between 1.1 to 1.5.
- *minNeighbours*: Parameter specifying how many neighbors each candidate rectangle should have to retain it. It specifies the minimum number of rectangles required to group an area as an object.

This returns a data structure, *faces* (array of arrays), where each array contains the coordinates of the bounding boxes in the picture. If a face is found it will contain four values → [[x, y, w, h]]. The x,y correspond to the coordinates of the upper left corner of the bounding box around the face. w is the width of the bounding box and h is the height of the bounding box. If more than 1 face is found, it will return a group of faces as multiple arrays inside the larger array. If it detects more than one face, resulting array will look like:

[[x1,y1,w1,h1], [x2,y2,w2,h2]]



- Once the detector detects faces, add the following lines to iterate over all the detected faces and draws a rectangular box around them

```
      print(faces)

      for face in faces:

            x,y,w,h=face

        cv2.rectangle(frame, (x,y), (x+w, y+h),

  (255,0,0),3) cv2.imshow("Camera Frame", frame)

  time.sleep(0.5)

      key=cv2.waitKey(1) & 0xFF
```

3. save the code named Picamera2FaceTracking.py and run it in IDE. The output should show bounding boxes around the detected faces

Checkpoint-1:

Write a program to count the number of faces in a video stream, and print a message (on console) with the number of faces detected. Call the TA once you complete this exercise.

Detecting motion from a live surveillance camera

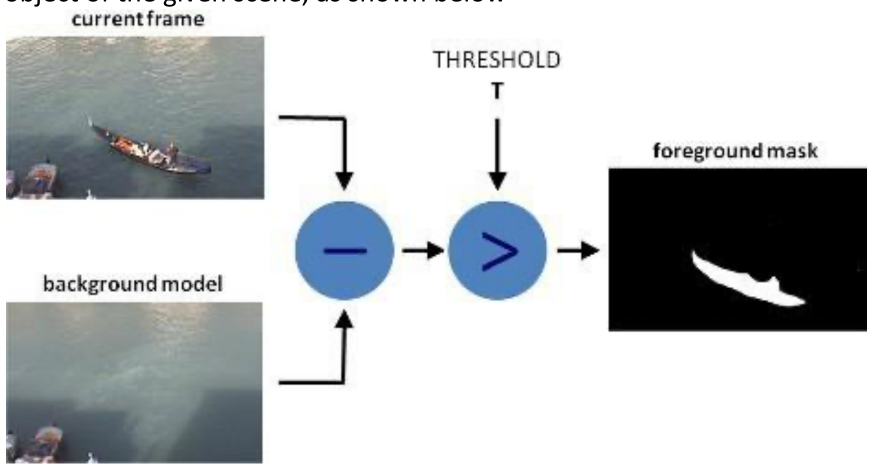
Background subtraction is a technique that is commonly used to identify moving objects in a video stream. The idea behind background subtraction is that once you have obtained a model of the background, you can detect foreground objects by examining the difference between the current video frame and the background frame (containing the static part of the scene)

In this lab, we will compute the background model by taking the weighted mean of previous frames and comparing that mean to the current frame. This way, the background model is dynamically adjusted and compared against the current frame to detect motion. You can also initialize the first frame as the background model instead of the weighted mean of previous frames under the assumption that the first frame is a good representation of the background model.

However, this assumption can easily break under scenarios where new objects are introduced into the frame or the lighting conditions change, causing false motion detections. Therefore, it is important to update the background model dynamically. Based on the weighted average of the frames, we subtract the weighted average from the current frame to compute the frame delta.

delta=|background_model - current_frame|

We can then threshold this delta to find regions of image that have substantial differences from the background model. These regions correspond to the detected motion from our live video streams. Using this background subtraction technique, we can generate a **foreground mask**. A foreground mask is a binary image containing the pixels belonging to a moving object of the given scene, as shown below



Source: https://docs.opencv.org/4.x/d1/dc5/tutorial_background_subtraction.html

We are using OpenCV's MOG2 background subtraction algorithm which is a Gaussian Mixture based foreground and background segmentation method for motion detection. More details about it can be found here: link-1, link-2, link-3.

You will now be modifying your previous security camera code to know when the motion is detected by the camera. You can also modify Picamera2Capture.py provided to you on Canvas. Save your modified security camera code as picamera2_motion_detect.py.

1. Add the following lines after the picam2.start() function and before the While loop to define a background subtractor object.

```
picam2.start()
 ## Number of previous frames to update background model
num_history_frames=10
 ## Gets the background subtractor object
back_sub= cv2.createBackgroundSubtractorMOG2(history=num_history_frames, \ varThreshold=25, \
detectShadows=False)
time.sleep(0.1)
max_foreground=127 # (0-255)
```

Note that the background subtraction function takes the following three parameters: history, varThreshold, and detectShadows.

history: Defines the number of previous frames to update the background model

varThreshold: Threshold on the squared distance between the pixel and background model to decide if a pixel is foreground or background. A lower varThreshold value makes the algorithm more sensitive, classifying more pixels as foreground even with subtle changes in the background. The classification is based on how well the pixel's color fits the background model.

detectShadows: If true, it will detect shadows. It decreases the processing speed.

***Note:** You can tune the history and varThreshold parameters as you see fit.*

2. Now add the following lines in the While loop. This will apply the background subtractor object to the frame in question.

```
frame=picam2.capture_array()
            ## Frame is a large 2D array of rows and cols
fgmask = back_sub.apply(frame) ## obtains the foreground mask
```

1. The following lines of code are responsible for settling the threshold value for what is considered foreground and what is considered to be the background based on the grayscale color from the mask. If a pixel is less than 127, it is considered black (background) and set to 0. Otherwise, it is white or foreground, and set to 255.

```
              # If a pixel is less than 127 , it is considered black (background) # otherwise, it is
      white (foreground). 255 is upper limit.
              # modify the number after fgmask as you seem fit.
_,fgmask=cv2.threshold(fgmask, max_foreground, 255, cv2.THRESH_BINARY)
```

2. Next, using our background and foreground we find the *contours* around the object so that we can get the areas around the moving object. A contour in OpenCV is defined as a boundary curve around a region of pixels. We want to find the contours so that we can compute the area of moving objects. Knowing the area of each moving object allows us to find the largest foreground object. Add the following lines to the code:

```
            # Find the contours of the object inside the binary image
            contours, hierarchy = cv2.findContours(fgmask, cv2.RETR_TREE, \
    cv2.CHAIN_APPROX_SIMPLE)[-2:]
            areas=[cv2.contourArea(c) for c in contours]
```

3. Save this file as **Picamera2MotionDetect.py** and run it from the Thonny IDE or via a terminal to see how things look. You will probably see little spots of motion even if there is no motion happening at all. Take this time to make sure your camera is detecting motion, even if there are false positives.

4. Now let us perform some filtering techniques to make our motion detection more efficient. A large reason why we see random detections is because there is a lot of noise in the image that OpenCV is analyzing. To filter the noise, first define what is called a *kernel*. A kernel is a small image that is scanned over another image to apply some kind of filtering process to it. (For more information, look here.) The kernel size can be changed depending on the level of noise reduction we want and the input video characteristics. Add the following code before the while loop:

```
 max_foreground=127 # (0-255)
 ## Create a kernel for morphological operation to remove noise from binary images.
 ## You can tweak the dimensions of the kernel ## e.g. instead
of 20,20 you can try 30,30.
 ## This creates a square matrix of 20x20 filled with ones, suitable for closing operations.
 ## Closing operations smoothen out a binary image by removing small holes/gaps in detected objects
kernel= np.ones((20,20), np.uint8)
```

5. The following OpenCV morphologyEx() function is used for applying the closing operation to close any small holes inside the fgmask or to remove small black points in the fgmask. Add the following code inside the while loop after applying the background subtractor object:

```
fgmask = back_sub.apply(frame) ## obtains the foreground mask #
closing the gaps
fgmask=cv2.morphologyEx(fgmask, cv2.MORPH_CLOSE, kernel)
```

6. The following line of code is used to smooth the boundaries of the foreground mask. This is necessary because the foreground detection process generates salt and pepper noise along the border of the foreground region and the background region. A median blur filter is effective at removing this type of noise from our foreground mask. Add the following medianBlur() function right below the previous line of code:

```
fgmask=cv2.medianBlur(fgmask,5) # Remove salt and pepper noise using the
filter
```

The rest of the code for subsequent processing like thresholding and contour detection will remain the same.

7. Test the code again; now it should be able to detect major motion from the camera feed with fewer false positives.

***Note**: Make sure the camera is stable during the testing.*

Checkpoint 2:

Even though the filters improve the motion detection performance, there is still some room for improvement. You can modify what the maximum color that identifies as foreground (max_foreground) as well as how many frames we should wait with no motion before we classify what is currently on display as our new "background" i.e. *max_foreground* and *num_history_frames* can be modified based on lighting conditions or other factors. So try to change the values for the above parameters and test it in different lighting conditions. When you are satisfied with how your spy camera is working, call over a TA and demonstrate it to them by showing that 1) your choice of parameters reduces false detection noise and 2) your spy camera can still detect true positives (actual motion).

Post-lab 1 Assignments

Create a github repo with separate script folders for each assignment below. Include 2 videos of your functioning codes in the readMe file of your github repo as well as bonus. Submit the repo link on canvas and make sure it's accessible to anyone with the link to repo.

Post Lab Assignment-1 (5 points): Develop a program that enables the movement of a solitary pixel (represented by a single LED) within the SenseHat matrix. Control this pixel's movement using the Joystick interface. For example, assume the initial (x,y) coordinates of the pixel to be illuminated is set as (3,5). When the joystick is pushed in the *right* direction, the subsequent pixel along the positive X-axis should light up. Likewise, pressing the joystick upwards, downwards, or to the left should illuminate the respective corresponding pixel. Exit the program in case of the middle-click.

Hint: https://github.com/raspberrypilearning/astro-pi-guide/blob/master/inputs-outputs/joystick.md

Post Lab Assignment-2 (5 points): Cameras are computationally intensive for Raspberry Pis. To optimize their use and efficiency, let us explore a practical application by integrating it with other sensing modalities. Write a Python program that utilizes the Sense-Hat module to estimate the presence of a person based on changes in the ambient temperature or humidity. The program should automatically initiate video recording when the temperature/humidity changes by 1 degree Celsius and perform face recognition. (You can create the temperature/humidity change by blowing onto the sensors.)

Bonus assignment - optional (4 points): Download another pretrained model from Haarcascades link below and develop a new cool smart feature for your camera.

https://github.com/kipr/opencv/tree/master/data/haarcascades

Bonus assignment - optional (4 points)

Write a script that captures the RPi IP address and shows it on the LED matrix. Set the script to run at the boot time.

Helpful Links:
https://www.diyrobocars.com/2017/11/27/displaying-your-raspberry-pi-ip-address-on-bootup/
https://stackoverflow.com/questions/2311510/getting-a-machines-external-ip-address-with-python