PRACTICAL 4

MOTION DETECTION

This practical continues the work done in Practical 3 and will re-use/modify the scripts generated there. Be sure of having completed Practical 3 beforehand.

TASK 1: GMM

In this task, we will perform background subtraction using adaptative Gaussian Mixture Models. To assist you in this process, you have some available functions to download from QoI. You will have to develop a couple extra.

STEP 1: Create a new script where you will load the video viptraffic. Similarly to Practical 3,Task 1 Step 5, 6 and 9, you will need to extract the current frame within a loop and convert it to gray scale and doubles.

STEP 2: Before starting the loop, you will need to instantiate the function `InitialiseGMM` to initialise the Mixture of Gaussians

```
GMM = InitialiseGMM(height, width);
```

You need to provide this function with the width and height of the video frames. The output variable GMM is a structure which contains the mean, standard deviation and weight of each Gaussian for every pixel in the image. It also contains all the parameters that can be tuned. Please open the function file and try to understand the code. How are the means initialised?

STEP 3: Inside the loop you will need to call the function RunGMM at each iteration/frame.

```
[Foreground, Background, GMM] = RunGMM(currentFrameGray,GMM);

figure(1),subplot(2,3,1),imshow(uint8(currentFrameGray)), title(['Frame: ',
num2str(t)])
subplot(2,3,2),imshow(uint8(Background)), title('Background')
subplot(2,3,3),imshow(uint8(Foreground)), title('Foreground')
```

This function takes and input the new frame and the structure containing the Mixture of Gaussians. It returns this MoG updated according to the new frame, and the current estimation of the background and the foreground (moving objects).

If you try to run the script, an error will be display. This is because there are 3 functions missing that you will need to complete.

STEP 4: The first function will calculate the distance between a pixel value and the Gaussian mean. This is needed to calculate the **probability** of such pixel to belong to any of the Gaussians

$$p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \longrightarrow \boldsymbol{d = |x-\mu|}$$

Create the function `distance2Gauss`, and fill it with a simple formula which calculates the equation in red above.

Please remember that the two hour duration of a practical class will probably not be enough time to complete all sections. There is an expectation that you will work outside of class as an independent learner.

```
function distance = distance2Gauss(pixelVal, mean)
```

STEP 5: The second function will implement the **condition** to check if a pixel belongs to a Gaussian. Therefore its output will be a Boolean. The equation that defines this condition is:

$$\frac{|x-\mu|}{\sigma} \leq D \quad \Rightarrow \quad \frac{d}{\sigma} \leq D$$

where D is the positive deviation threshold, which has been initialised by the function `InitialiseGMM`. Create the function `belong2Gauss`, and fill it with a simple formula which calculates the condition above. Please note that since the numerator of the fraction has already been calculated, we are passing directly the distance to the mean.

```
function belonging = belong2Gauss(distance, sd, threshold)
```

STEP 6: The last function to implement is in charge of updating the Gaussian values (weight, mean and standard deviation) once the pixel has been assigned to it. The equation involved in this update is very similar to the ones seen in the lecture, with some small differences:

$$w_{t+1}^k = (1-\alpha) * w_t^k + \alpha$$

$$\mu_{t+1}^k = \left(1 - \frac{\alpha}{w_{t+1}^k}\right) * \mu_t^k + \frac{\alpha}{w_{t+1}^k} * x$$

$$\sigma_{t+1}^k = \sqrt{\left(1 - \frac{\alpha}{w_{t+1}^k}\right) * \left(\sigma_t^k\right)^2 + \frac{\alpha}{w_{t+1}^k} * \left(x - \mu_{t+1}^k\right)^2}$$

This function should also have an alternative behavior if no pixel is assigned to such a Gaussian:

$$w_{t+1}^k = (1-\alpha) * w_t^k$$

$$\mu_{t+1}^k = \mu_t^k$$

$$\sigma_{t+1}^k = \sigma_t^k$$

Create the function `UpdateGMM`, and implement previous equations. The alternative behaviour can be implemented by counting the number of parameters passed (using `nargin`): if there is no pixel value, then it will run the second set of equations.

```
function [w, mean, sd] = UpdateGMM(w, mean, sd, alpha, pixval)

      if nargin<5



      else



      end
end
```

STEP 7: Now that we understand some of the crucial bits, open the `RunGMM` file and try to understand the different steps and how each of the pixels and each of the Gaussians is processed.

STEP 8: Run the code and observe how the background and foreground evolved. Does it have sense (remember the question in Step 2)? Do you notice something strange at the middle of the video? Was that happening with the previous background subtraction method?

STEP 9: Finally we can convert the foreground image into a binary image with a simple instruction

```
Blobs = Foreground > 0;
```

Complete the script by adding morphological operators, connected component labelling and bounding box extraction and plotting, as in the previous tasks.

TASK 2: OPTICAL FLOW

Optical flow allows estimating the motion flow of the image. It can also be used as motion detector, especially useful when camera is moving.

STEP 1: Create a new script where you will load the video viptraffic. Create a for loop to have access to the current frame and also its previous frame. The loop should start in the second frame to avoid an out of bounds error for the first iteration

```
nframes = size(vidFrames,4);

for t = 2:nframes
    currentFrame= vidFrames(:,:,:,t);
    currentFrameGray = rgb2gray(currentFrame);
    currentFrameGray = double(currentFrameGray);

     previousFrame= vidFrames(:,:,:,t-1);
     previousFrameGray = rgb2gray( previousFrame);
     previousFrameGray = double( previousFrameGray);

end
```

STEP 2: Call the provided function HS, which calculated the optical flow using the Horn-Schunck method

```
    [u, v] = HS(previousFrameGray, currentFrameGray);
```

This function returns 2 matrices, u containing the horizontal component of the motion vector for each pixel, and v containing the vertical component of the motion vector for each pixel. We can now combine both components to reconstruct the motion vectors and plot them.

STEP 3: To draw the vector, Matlab has a handy function available called quiver. Draw the vectors on the current frame:

```
    figure(1)
    imshow(previousFrameGray,[0 255]), hold on
    quiver(u, v, 4, 'color', 'b', 'linewidth', 2);
    set(gca,'YDir','reverse');
    hold off
```

STEP 4: Stop in one of the frames and make zoom with the tool provided in the figure window in the area around the car. Observe how the pixels are match and their motion is estimated.

As you can see, some of the pixels are noise or badly estimated, however, the global motion of the object is capture. We can take advantage of this to detect the moving areas in the image. To do that we are going to calculate the magnitude of the vectors and considered in motion every pixels with a magnitude of bigger than 3 pixels

STEP 5: Convert the optical flow in motion blobs by using the magnitude.

```
mag = sqrt(u.^2+v.^2);
vel_th = 3;
Blobs = mag >= vel_th;
```

STEP 6: Complete the scripts by adding morphological operators, connected component labelling and bounding box extraction and plotting. How this motion detection compares with the previous ones? Is it better or worse? What are the potential advantages? How could it be improved?

TASK 3: NEW VIDEOS

STEP 1: Now run all the previous scripts in this practical, as well as the motion detection from Practical 3, on the video shopping_center.mpg. Does it work? How many parameters did you have to change/ tune?

OTHER DETECTIONS

TASK 4: TEMPLATE MATCHING

In this task we are going to implement template matching using our convolution functions that we learned in practical 2.

STEP 1: Create a function

```
function BB=templateMatching(I,template)
```

that will contain all the logic to perform the template matching. The first 2 instructions will convert both the image I and the template into doubles. Use only grayscale.

STEP 2: Using either the convolution function `filter2`() or your own convolution function `myconvolution`() created in Practical 2 , calculate the correlation map between the full image and the template.

STEP 3: Using the function `max`(), obtain the coordinates x and y of the most likely position of the template in the image.

STEP 4: Convert the coordinates x and y into a Bounding box in the form (similar to what you did in Practical 3, Task 3, Step 4.

):

```
BB = [xmin ymin xmax ymax];
```

STEP 5: Load the provided image '`letters.jpg`' and the first template '`k`'`jpg`'. Calulate the resulting bounding-box using the function `templateMatching()` that you just completed. In a figure, display both the image, the template as well as the image with the returned bounding-box on it.

STEP 6: Load the other template `'s.jpg'` and display the new results. Display the correlation map too and analyse if the system is working.

STEP 7: Load the last set of image/template '`image1.jpg`' and '`image2.jpg`'. What is your conclusion now?

STEP 8: Modify the template matching function so:

```
function BBs=templateMatching2(I,template,N)
```

it returns not only one BB but a matrix Nx4 with the N most likely bounding boxes.

STEP 9: Apply the new function to both the letters and the building image and get your conclusions. Can you think on an alternative to get a better performance of template matching in natural images?