Moto GP frames Challenge

Robin Blanchard

October 8, 2018 -

1 Description of the issue

Frames coming from a fixed camera are given. On these images, there can be one or more motos, or not. The goal of the challenge is to build a model permiting to state if there is a moto on the picture or not.

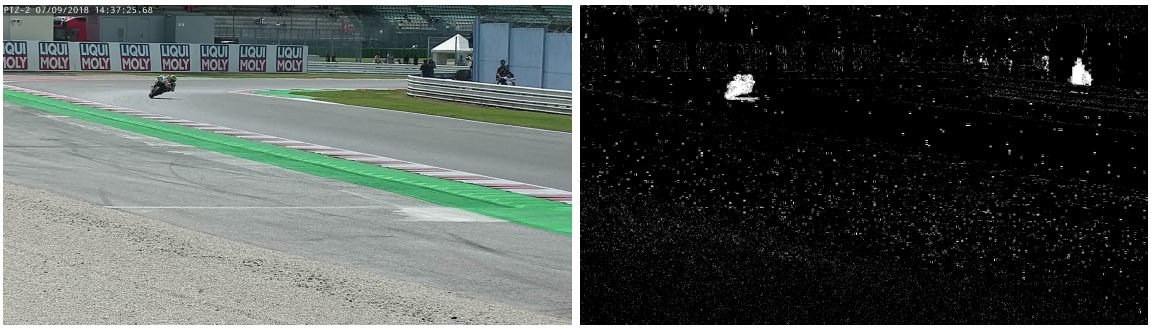
2 Solution using OpenCV

First, I used computer vision techniques, imported from the OpenCV package, to analyse the frames.

2.1 Background Substraction

By observing the given frames, I understood that all the frames come from a fixed camera, looking in a fixed direction. Indeed, most of the picture stays the same accross all pictures. My first idea was then too remove the background of the pictures, in order to obtain pictures with what is moving in white, and the rest in black. To do so, I used the createBackgroundSubtractorMOG2 function from OpenCV, applied this filter to every frame by using the following function.

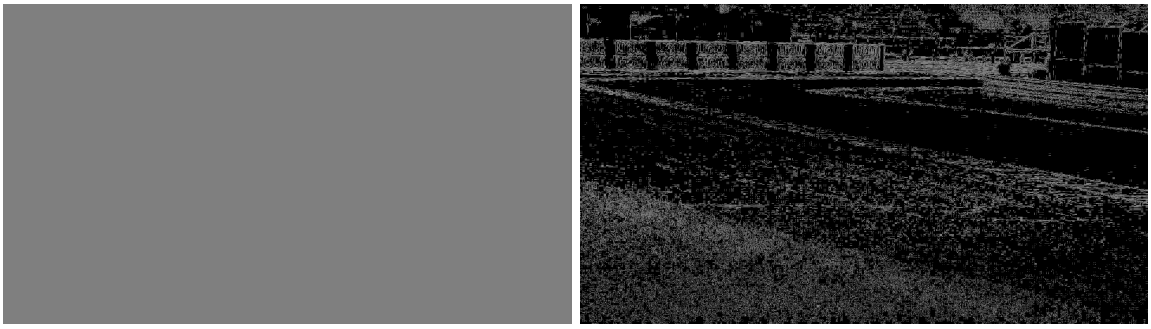
```
def fg_extraction():
    """
    This function extracts the foreground from the background and
                                        delivers black
    and white images, the foreground being in white.
    """

    fgbg = cv2.createBackgroundSubtractorMOG2()
    for i in range(30,192):
        if i<100:
            img_name="00"+str(i)
        else:
            img_name="0"+str(i)
        img = cv2.imread(img_name+'.jpg')
        fgmask = fgbg.apply(img)
        cv2.imwrite('Foreground\\fg_'+img_name+'.jpg',fgmask)
```



(a) Original picture (b) After Foreground Extraction

Figure 1: Background substraction of frame 154

But then, I observed that the results of the first pictures aren't great. This makes sense because the background substractor has not been trained on enough pictures to detect what is not background on first pictures. Here are some examples:



(a) Output for the first picture (b) Output for the third picture

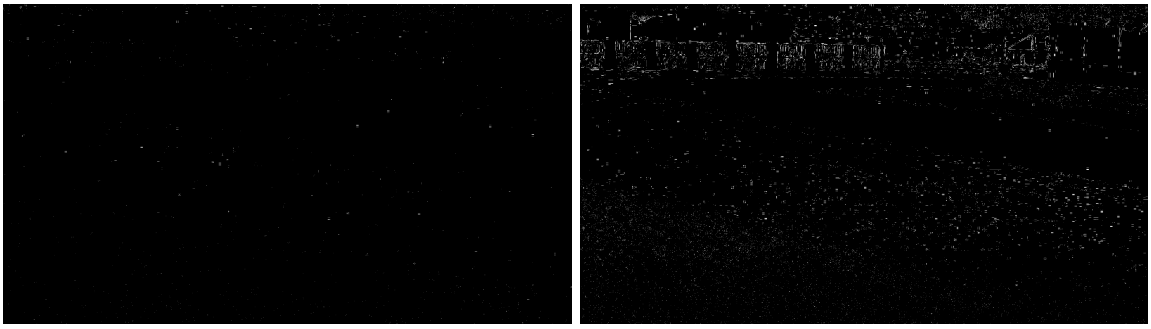
Figure 2: Errors because of poorly trained background substracter

So, I updated the fg_extraction by going through all the frames twice.

```
def fg_extraction():
    """
    This function extracts the foreground from the background and
                                        delivers black and white images,
                                         the foreground being in white.
    """

    fgbg = cv2.createBackgroundSubtractorMOG2()
    for k in range(2):
        for i in range(30,192):
            if i<100:
                img_name="00"+str(i)
            else:
                img_name="0"+str(i)
```

```
            img = cv2.imread(img_name+'.jpg')
            fgmask = fgbg.apply(img)
            cv2.imwrite('Foreground\\fg_'+img_name+'.jpg',fgmask)
```

Here are the new results for the first and third photos.



(a) Output for the first picture (b) Output for the third picture

Figure 3: Errors solved

2.2 Post-processing

Now, I looked at the different outputs I had and I realized that most of the foregrounds extracted also contain small irrelevant white points, corresponding to noise. Consequently, I implemented a post_processing function.

```
def post_processing(img):
    median = cv2.medianBlur(img,5)
    thresh = cv2.threshold(median, 200, 255, cv2.THRESH_BINARY)[1]
    thresh = cv2.erode(thresh, None, iterations=2)
    post = cv2.dilate(thresh, None, iterations=4)
    return post
```

The first step of the post processing is applying a median filter using the medianBlur function. ksize, the dimension of the aperture, is taken as 5. This results in smoothing the picture. Applying a threshold after this allows to remove completely most of the noise. Finally, the erode and dilate functions use the pixel neighborhood to change the pixel, respectively using the minimum and maximum in the neighborhood values. The results of this function on frame 154 are following.

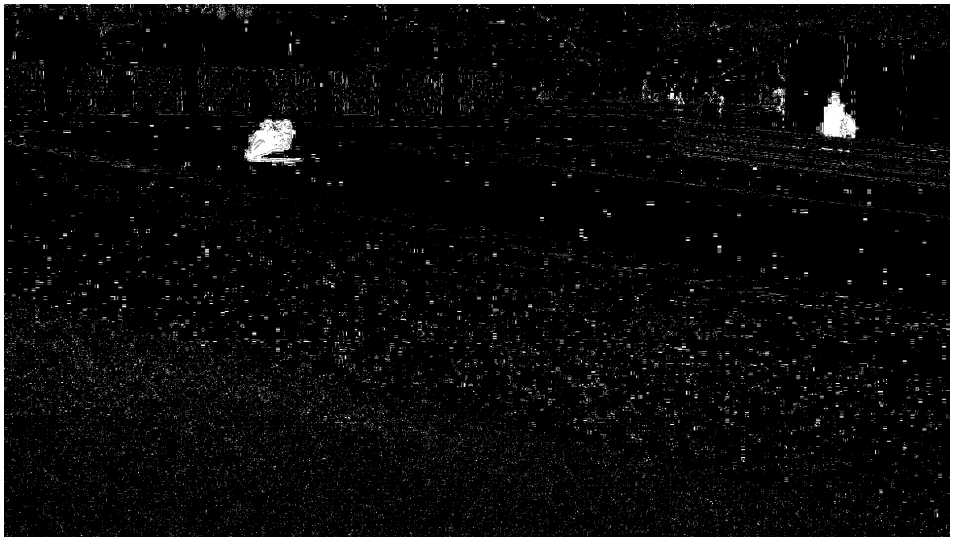


Figure 4: Before Post-processing

Figure 5: After Post-processing

I am then able to finalize my foreground mask extraction function. I only have to post process every foreground mask before saving the picture.

```
def fg_extraction():
    """
    This function extracts the foreground from the background and
                                        delivers black and white images,
                                         the foreground being in white.
    """
```

```
    fgbg = cv2.createBackgroundSubtractorMOG2()
    for k in range(2):
        for i in range(30,192):
            if i<100:
                img_name="00"+str(i)
            else:
                img_name="0"+str(i)
            img = cv2.imread(img_name+'.jpg')
            fgmask = fgbg.apply(img)
            post=post_processing(fgmask)
            cv2.imwrite('Foreground\\fg_'+img_name+'.jpg',fgmask)
            cv2.imwrite('Foreground\\fg_post_'+img_name+'.jpg',post)
```

2.3 Find the connected components

Let's now observe one of the foreground masks.

Figure 6

Two kinds of white points can be seen: large groups of points, that are going to be called blobs, and isolated sole points. To be able to find all the groups, and filter through them only to get the blobs, we need to find the connected components of the picture. To do so, I used the connectedComponentsWithStats function from OpenCV and designed the following function.

```
def blob_count(img,threshold=50,connectivity=4):
    img = cv2.imread(img,0) #0 is needed to open the image in grayscale.
```

```
    output = cv2.connectedComponentsWithStats(img, connectivity, cv2.
                                              CV_32S)
    num_labels = output[0]
    labels = output[1]
    stats = output[2]
    centroids = output[3]
    count=0
    for label in range(1,num_labels):
        width = stats[label, cv2.CC_STAT_WIDTH]
        height = stats[label, cv2.CC_STAT_HEIGHT]
        #x = stats[label, cv2.CC_STAT_LEFT]
        #y = stats[label, cv2.CC_STAT_TOP]
        if width>=threshold and height>=threshold: #can instead use the
                                                   number of white pixels.
            #roi = img[y:int(y+height), x:int(x+width)]
            #cv2.imwrite('blob_'+str(label)+'.jpg', roi)
            count+=1
    return count
```

The blob_count function returns the number of blobs for a particular image. All is left to do is to compute this function for every foreground mask, and print the result. This is done with the is_there_motos function.

```
def is_there_motos():
    for i in range(30,192):
        if i<100:
            img_name="00"+str(i)
        else:
            img_name="0"+str(i)
        count=blob_count('Foreground\\fg_post_'+img_name+'.jpg')
        if count==0:
            print('There isn\'t any moto in the picture {}.'.format(
                                                    img_name))
        elif count==1:
            print('There is one moto in the picture {}.'.format(img_name)
                                                    )
        else:
            print('There are {} motos in the picture {}.'.format(count,
                                                    img_name))
```

Here is a sample of the output.

There isn't any moto in the picture 0035.
There isn't any moto in the picture 0036.
There is one moto in the picture 0037.
There are 2 motos in the picture 0111.
There are 2 motos in the picture 0112.
There are 2 motos in the picture 0113.
There are 3 motos in the picture 0114.
There are 3 motos in the picture 0115.

There are 3 motos in the picture 0116.
There are 2 motos in the picture 0117.

2.4 Removing the extra moto

Let's now look at picture 0114, since it is stated that 3 motos appear on it.

Figure 7: Frame 114

Only two motos are on this frame. Where does the 3 motos come from? In order to find out where the issue comes from, let's give a look at the foreground mask.

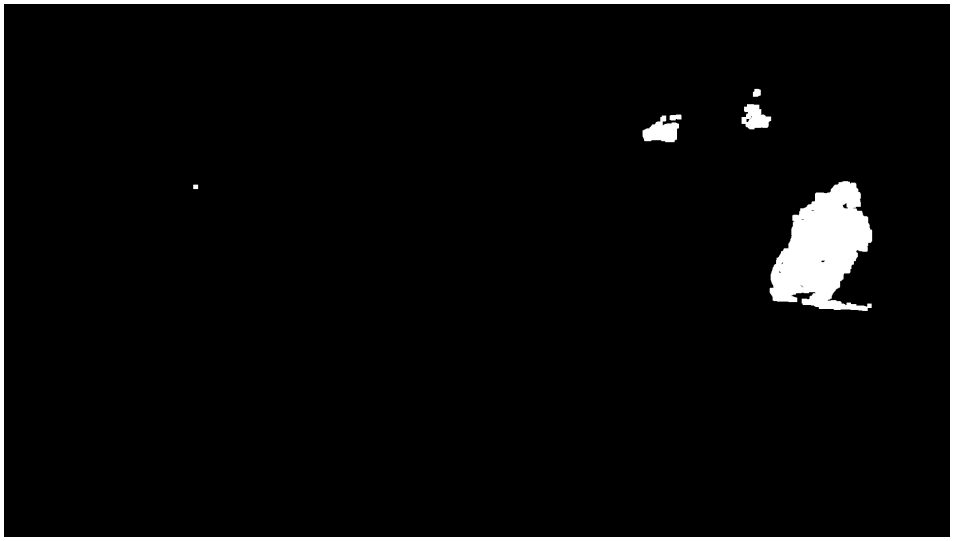


Figure 8: Frame 114 foreground

The two race motos are indeed forming blobs on the foregound, but the media moto is also taken into account. To remove it, I replaced the top right corner by a black filter, as following.

Figure 9: Removal of media moto

I add this black filter inside the blob function, just before counting the blobs.

```
def blob_count(img,threshold=50,connectivity=4):
    img = cv2.imread(img,0) #0 is needed to open the image in grayscale.
    img[0:300,1400:]=0 #remove top_right corner
    output = cv2.connectedComponentsWithStats(img, connectivity, cv2.
                                              CV_32S)
    ...
```

Here is a sample of the new output.

There isn't any moto in the picture 0035.
There isn't any moto in the picture 0036.
There is one moto in the picture 0037.
There is one moto in the picture 0038.
There is one moto in the picture 0039.
There is one moto in the picture 0110.
There is one moto in the picture 0111.
There is one moto in the picture 0112.
There is one moto in the picture 0113.
There are 2 motos in the picture 0114.
There are 2 motos in the picture 0115.
There are 2 motos in the picture 0116.

There is one moto in the picture 0117.
There is one moto in the picture 0118.
There is one moto in the picture 0119.
There is one moto in the picture 0120.
There is one moto in the picture 0121.

2.5 Colors of moto

After having counted the motos on an image, I wanted to determine the color of each moto. It is pretty easy to select only the part of the image corresponding to the photo by using the coordinates, height and width of the connected components being larger than the threshold. However, in order to find the color of the restricted image, I tried to compute the average color of it. By doing so, most of the outputs were brown. I did some research on the Internet and it seems that what I am looking for is the dominant color, and not the average color.

I first tried to compute the most dominant colors manually, counting the RGB values for each pixel. The advantage of doing it manually is that I can implement the fact that the most dominant color is going to be black, so don't take black pixels into account. I can then see which of the RGB components is the most dominant.

```
def find_color(img):
    height, width = len(img), len(img[0])

    red = 0
    green = 0
    blue = 0
    count = 0

    for i in range(width):
        for j in range(height):
            colors=img[j][i]

            pix_red=colors[0]
            pix_green=colors[1]
            pix_blue=colors[2]

            if pix_red!=0 or pix_green!=0 or pix_blue !=0:
                red += pix_red
                green += pix_green
                blue += pix_blue
                count+=1

    return [red/count,green/count,blue/count]
```

There are unfortunately many drawbacks to this solution, because I only have which of Red, Green and Blue is most present, but it doesn't give me the exact color of the moto. If there is one red moto and one purple (more red than blue) moto, this solution wouldn't make the difference . Moreover, this solution is computationnally very slow, although it is quite quick as we only look at the color of the blobs.

I then decided to look on the Internet how people usually handle this issue. The most famous one is using the K-means algorithm, such as explained here : https://adamspannbauer.github.io/2018 icon-dominant-colors/.

I built again the find_color function, applying the K means algorithm as explained, and took arbitrarily the 4 most dominant colors.

```
def find_color2(img_name):
    image = cv2.imread(img_name+'.jpg')
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # reshape the image to be a list of pixels
    image = image.reshape((image.shape[0] * image.shape[1], 3))

    # cluster the pixel intensities
    clt = KMeans(n_clusters = 4)
    clt.fit(image)

    return clt.cluster_centers_
```

Let's look at the following picture:

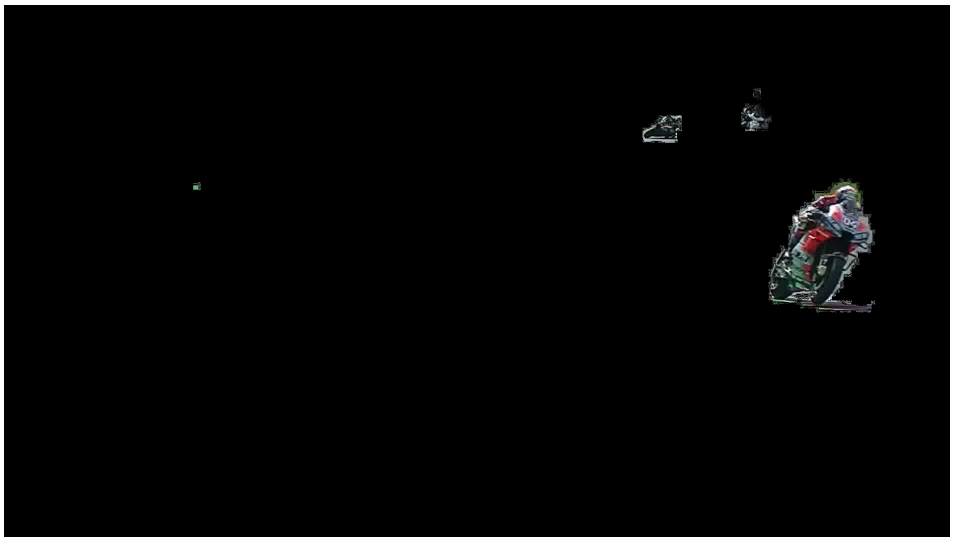


Figure 10: Foreground of a picture

The four most dominant colors of this picture are:

```
[[5.76891038e-02 8.48352454e-02 8.28727143e-02]
 [5.55884942e+01 5.20353977e+01 5.35132359e+01]
 [1.92703574e+02 1.94218445e+02 1.96451637e+02]
 [1.19949720e+02 1.21750280e+02 1.21953237e+02]]
```

Figure 11: Dominant colors

We observe that most of these pictures are shades of grey. This solution then needs to be improved. The issue is that the foregound extraction isn't very accurate on the edges of the motos and that we have computed the most dominant colors and the whole image, whereas most of it is grey. We should restrain our computation to blobs.

2.6 Colors and Ranking

Finally, I tried to improve the is_there_motos function by adding to the input sentences like this "The moto in the front is red. The moto in the back is green". I used the first solution of the color detection. I used the fact that the moto in the front should lead to a connected component with a larger area than the moto in the back. Then, for every moto, I compute the most dominant color, present on the picture restricted to the blob corresponding to the moto. I also had to store the positions and sizes of the blob, so I modified the blob_count function:

```
def blob_count(img,threshold=50,connectivity=4):
    img = cv2.imread(img,0) #0 is needed to open the image in grayscale.
    img[0:300,1400:]=0 #remove top_right corner
    output = cv2.connectedComponentsWithStats(img, connectivity, cv2.
                                              CV_32S)
    num_labels = output[0]
    labels = output[1]
    stats = output[2]
    centroids = output[3]
    count=0
    blobs=[]
    for label in range(1,num_labels):
        width = stats[label, cv2.CC_STAT_WIDTH]
        height = stats[label, cv2.CC_STAT_HEIGHT]
        x = stats[label, cv2.CC_STAT_LEFT]
        y = stats[label, cv2.CC_STAT_TOP]
        if width>=threshold and height>=threshold: #can instead use the
                                                   number of white pixels.
            #roi = img[y:int(y+height), x:int(x+width)]
            #cv2.imwrite('blob_'+str(label)+'.jpg', roi)
            count+=1
            blobs.append([x, y, width, height])
    return count, blobs

colors={0:'red',1:'green',2:'blue'} #dictionnary to store the
                                    corresponding colors
```

```
def is_there_motos():
    for i in range(30,192):
        if i<100:
            img_name="00"+str(i)
        else:
            img_name="0"+str(i)
        count, blobs = blob_count('Foreground\\fg_post_'+img_name+'.jpg')
        if count==0:
            print('There isn\'t any moto in the picture {}.'.format(
                                                img_name))
        elif count==1:
            print('There is one moto in the picture {}.'.format(img_name)
                                                )

            img = cv2.imread(img_name+'.jpg')
            fg= cv2.imread('Foreground\\fg_post_'+img_name+'.jpg',0)
            fg_color=cv2.bitwise_and(img,img,mask = fg)

            for blob in blobs:
                x,y,width,height=blob
                img_blob=fg_color[y:int(y+height), x:int(x+width)]
                most_present_color=find_color(img_blob)
                most_present_color=most_present_color.index(max(
                                                most_present_color))
                                                 #index of the most
                                                dominant color

            print('The moto is {}.'.format(colors[most_present_color]))

        else:
            print('There are {} motos in the picture {}.'.format(count,
                                                img_name))

            img = cv2.imread(img_name+'.jpg')
            fg= cv2.imread('Foreground\\fg_post_'+img_name+'.jpg',0)
            fg_color=cv2.bitwise_and(img,img,mask = fg)

            colors_ranking=[]

            for blob in blobs:
                x,y,width,height=blob
                img_blob=fg_color[y:int(y+height), x:int(x+width)]
                most_present_color=find_color(img_blob)
                most_present_color=most_present_color.index(max(
                                                most_present_color))
                                                 #find the most
                                                dominant color for
                                                each blob
                colors_ranking.append([width*height,most_present_color])
                                                #also store the area
                                                 of the blob
```

```
        colors_ranking=sorted(colors_ranking, key=lambda x:x[0],
                                              reverse=True) #reverse
                                              it to start with largest
                                               blob
        for i in range(len(colors_ranking)):
            print('The moto in position {} is {}.'.format(i+1, colors
                                                  [colors_ranking[i][1
                                                  ]]))
```

Finally, I have this input:

There is one moto in the picture 0107. The moto is red.
There is one moto in the picture 0108. The moto is red.
There is one moto in the picture 0109. The moto is red.
There is one moto in the picture 0110. The moto is red.
There is one moto in the picture 0111. The moto is red.
There is one moto in the picture 0112. The moto is red.
There is one moto in the picture 0113. The moto is red.
There are 2 motos in the picture 0114. The moto in position 1 is red. The moto in position 2 is green.
There are 2 motos in the picture 0115. The moto in position 1 is red. The moto in position 2 is green.
There are 2 motos in the picture 0116. The moto in position 1 is blue. The moto in position 2 is green.
There is one moto in the picture 0117. The moto is green.
There is one moto in the picture 0118. The moto is green.
There is one moto in the picture 0119. The moto is green.
There is one moto in the picture 0120. The moto is green.
There is one moto in the picture 0121. The moto is green.

The first moto in the timeline is said to be red, the second moto in the timeline is said to be green, and that is exactly what we can observe on the pictures. Whereas I thought this solution slow and naive, it seems to be accurate and not so slow since we execute it only on the blobs. This solution working means that we could be able to tell if their is an overtaking and more importantly when: if the colors of the front and back motos invert at some point, it means that there has been an overtaking.

2.7 Issues to solve

Finally, I imagined what issues I could face using this solution. It seems to work when the motos are separated. Here are somes cases where my solution wouldn't work:

-In case of overtaking, we could have an image where the motos are not separated. Computationally, it means we would find only one single connected component for two motos. My program would then count as only one moto the blob.

-In case of an accident, it is current to see the pilot and the moto go in different directions on the track. My program would then see two things moving on the image:

the pilot and the moto, and both interpret them as a moto. It would then count two blobs for one moto.

3 Probabilistic model

The first solution is able to count how many motos are on the picture. But it doesn't compute the probability of the results being correct. To do so, we need to build a probabilistic model. One way to do this could be to calculate features on the image. For example, calculate the connected components and then try to find round shapes, that could correspond to the shape of the helmet or the shape of the wheels. Moreover, one could build a neural network, using TensorFlow for example, to compute the probabilities from this kind of features.