CS585 Assignment 2: Scuba Hand Signal Recognition

Hao Qi
Together with: Seunghwan Hyun
February 14, 2024

1 Problem Definition

The topic of image and video computing has many application scenarios, and a possible one is recognizing scuba hand signals. In the silent underwater world where verbal communication is impossible, divers rely on hand signals to convey essential information, manage safety procedures, and respond to emergencies. Integrating computer vision techniques can lead to innovative interactions with diving equipment, minimizing the risk of miscommunication and leading to underwater exploration and research advancements. We hope to use the most classic computer vision methods we have learned recently to recognize several of the signals.

1.1 Data

After reading some online tutorials, we decided to use the guidance from a professional website to select the following six signals as our initial detection objects.

Up	Down	Ok
		
Leak/Bubbles	**Hold**	**Decompression**
		

Several pairs of them have strong similarities in morphology. For example, Signal Up and Down look the same in many aspects other than orientation.

1.2 Goals and difficulties

We plan to implement several classic algorithms using some of the library functions of OpenCV, such as skin color detection, contour detection, circularity measurement, template matching, etc. By combining these algorithms, we aim to improve recognition accuracy and enhance the resistance to background environmental interference. We will create

our dataset to evaluate the performance of the model quantitatively and will also design a graphical interface to detect inputs from the camera dynamically.

The key to accurate recognition is to effectively extract and utilize the diverse morphological features of hand signals. We need to be familiar with the library functions we use and choose the appropriate hyperparameters (such as multiple thresholds) through tests, which I will explain one by one below. Other potential issues include converting image size and channels, selecting the testing environment, etc.

2 Methods and Implementation

In our collaboration, I was primarily responsible for building the skeleton and implementing algorithms that use contour detection and circularity to distinguish signals; my teammate mainly took care of preparing the data, implementing template matching, and integrating and fine-tuning all these methods.

2.1 Graphical user interface

Creating the interface is a physically demanding but not a mentally demanding task; all I need to do is set the positions and callback functions of various controllers. The Tkinter-based GUI, with a fixed size of 1000x700 pixels, includes functionalities for file selection, file opening, and activating a camera for live feeds. It uses OpenCV for media manipulation, displaying recognition results in a text entry. When processing a video stream, it reads a frame every ten milliseconds and hands it to the model for processing. This is implemented using the Tkinter event loop.

```
# read and process frames
def open_video(self):
    res, img=self.video.read()
    if res == True and np.any(img):
        img1, img2, self.result_text = self.image_handler.get_hand(img)
        self.display_image1(img1)
        self.display_image2(img2)
    self.after = self.root.after(10, self.open_video)
```

2.2 Image preprocessing

The process includes applying Gaussian blur to smooth the image, obtaining a binary image using a specific range, and performing morphological operations (closing followed by dilation) to improve the result. We tried both directly using a predefined skin color range for filtering and converting to the HSV color space before filtering, using a smaller kernel size since the hand occupies a relatively small portion of the image.

```
# take skin color detection as an example
def preprocess_with_skin_color(self):
    self.img = cv2.GaussianBlur(self.src_img, (5, 5), 0)
    self.img = cv2.inRange(self.img, self.low_skin, self.high_skin)
    kernel = np.ones((3, 3), np.uint8)
    self.img = cv2.morphologyEx(self.img, cv2.MORPH_CLOSE, kernel)
    self.img = cv2.morphologyEx(self.img, cv2.MORPH_DILATE, kernel)
```

The program also cropped the images to speed up recognition.

```
def resize_img(self, img, size):
    size = [size[1], size[0], size[2]]  # [h, w, c]
    mask = np.zeros(size, dtype=np.uint8)
    h, w = img.shape[0:2]

    dwh = min([size[0] / h, size[1] / w])
    img = cv2.resize(img, None, fx=dwh, fy=dwh)

    if h > w:
        dxy = int((size[1] - img.shape[1]) / 2)
        mask[:, dxy:img.shape[1] + dxy, :] = img
    else:
        dxy = int((size[0] - img.shape[0]) / 2)
```

```
        mask[dxy:img.shape[0] + dxy, :, :] = img

    return mask
```

2.3 Contour Detection

The process starts by finding contours in the preprocessed image and then iterates through these contours. For each contour, it calculates the area and the perimeter, checking if these values fall within predefined maximum and minimum ranges. Contours that meet these criteria are approximated into polygon shapes, which simplifies the contour shapes based on a specified accuracy (epsilon, calculated as a certain percentage of the contour's perimeter) to obtain a list of contour vertices.

```
# code snippet
contours, _ = cv2.findContours(self.img, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
for contour in contours:
    area = cv2.contourArea(contour)
    length = cv2.arcLength(contour, True)
    if self.max_area > area > self.min_area and self.max_length > length > self.
                                                    min_length:
        epsilon = 0.02 * cv2.arcLength(contour, True)
        self.points_list = cv2.approxPolyDP(contour, epsilon, True)
        self.points_list = self.points_list.reshape(len(self.points_list), 2)
        self.points_list = np.array(self.points_list, dtype=np.int32)
```

I tried to use the maximum distance between the points forming the contour as a basis for distinguishing the number of fingers, but finding a suitable threshold for discrimination was too tricky. There are more viable methods than this.

2.4 Circularity Calculation

Instead of referring to the class material on the analysis on binary images, we used a more common approach to calculate the circularity of an object:

$$C = \frac{4 * \pi * S}{L * L}$$

where S represents the area of the shape, and L stands for the perimeter.

This feature might help us directly identify certain hand signals, which can serve as an auxiliary tool.

3 Experiments

3.1 Data

We collected a total of 90 images from a white background with a mobile phone in a fixed angle and lighting. 90 images can be broken down to 15 images per each hand sign. For each hand sign, we used the first 5 to calculate the circularity and produce the templates. However, we used all 15 images per hand sign for calculating Accuracy and F1-score for our model. Since we tried multiple methods, the output of the model is not necessarily the result of template matching.

3.2 Hand segmentation

The goal of our first experiment was to use different methods to have clear and exact inputs of hand images. It was hard to determine the evaluation metric at this stage, so we checked the output of sample data and chose the parameters that we thought had the best segmentation of hand in a naked eye.

3.3 Circularity

We attempted to calculate the average circularity for each hand sign to improve the performance of our prediction model. Because each hand sign has a different shape, if the circularity of each hand sign differs greatly from the others, using circularity to predict the hand sign would show great performance. We used 5 images for each hand sign to calculate the circularity and apply it to our model.

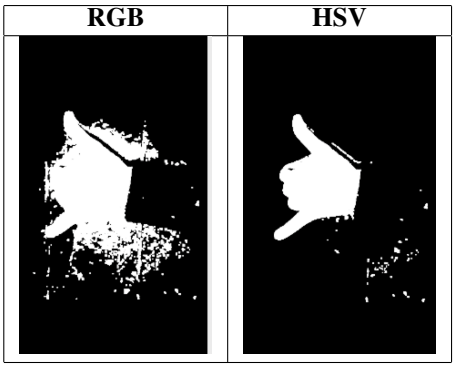
3.4 Template Matching

Template matching is a simple and direct method that calculates the similarity between a given image and the template. There have been different ways of calculating the difference between the target image and the template which is what we experimented with, the 'Template Matching Mode'. We tried different template matching modes in the OpenCV website. We used Accuracy and F1-Score as a evaluation metric for how each hand sign template accurately predicts the hand sign of input image.

4 Results

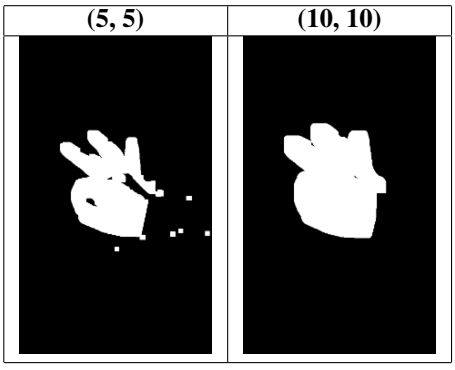
4.1 RGB vs. HSV

For the hyperparameters we used, converting the image into the HSV channel before binarization yielded better results. Therefore, we used it as the method for skin detection.



4.2 Kernel size

We need to use kernels when applying Gaussian blur, closing, and dilation operations. If the kernel is too large, it may lead to loss of image information, while if it is too small, it may result in excessive noise. In our experiments, we used relatively small kernels because we first cropped the image to a size of 480x480 before processing it, and the images were not large, with the hand being relatively small in the image.



4.3 Circularity

'Hold' sign had a circularity around 0.6 which was significantly greater than the other hand signs. However, other hand signs like 'Bubbles', 'Ok', 'Up' did not show significant differences when it comes to mean circularity of each hand signs. Although there was a subtle difference in the mean of the circularity for each hand sign, the large range made it difficult to predict the hand sign based solely on circularity.

	1	2	3	4	5
Bubbles	0.281	0.251	0.237	0.263	0.268
Decompress	0.317	0.258	0.331	0.302	0.259
Down	0.376	0.413	0.408	0.400	0.394
Hold	0.618	0.627	0.591	0.664	0.651
Ok	0.291	0.229	0.314	0.269	0.250
Up	0.290	0.322	0.399	0.225	0.287

Table 1: Circularity for Hand Signs

4.4 Template matching mode

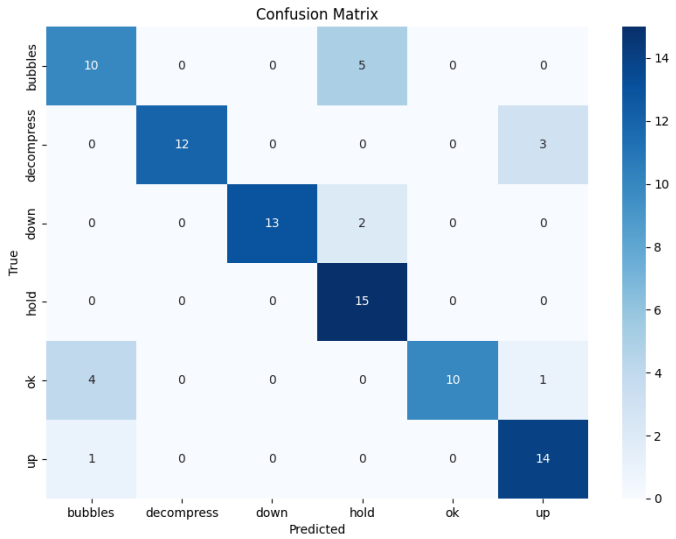
We discovered that normalization modes outperformed their corresponding modes. Template matching's weakness is that they are not robust to variations in lighting and contrast to the surroundings. This could be the reason why normalization modes showed better performance because **CCOEFF_NORMED** showed the best performance in both Accuracy and F1-score.

	Accuracy	F1-Score
TM_CCOEFF	0.22	0.12
TM_CCOEFF_NORMED	0.40	0.30
TM_CCORR	0.17	0.05
TM_CCORR_NORMED	0.23	0.15
TM_SQDIFF	0.17	0.05
TM_SQDIFF_NORMED	0.17	0.05

Table 2: Template Matching Operation Performance

4.5 Model performance

Our model achieved an accuracy of 82.2% on the test set of 90 images we created ourselves. The confusion matrix is as follows:



We have good recognition results for most of the signals, but the performance for 'bubbles' and 'Ok' are relatively poor. Both the demo video and the source code have been posted.

5 Discussion

We chose a project with practical significance and attempted to achieve the best possible recognition effect with simple methods. This is just a preliminary attempt, and we have left interfaces in anticipation of future improvements. Here are my analyses of some key points while writing the code and conducting tests.

5.1 Selecting values for hyperparameters

We used so many hyperparameters: the size for cropping images, the RGB or HSV range for converting color images to binary ones, the size of the kernel for morphological operations, options for finding contours, thresholds for filtering out small contours, thresholds for division based on circularity, options for template matching, weights for combining different results, and so on. We conducted detailed tests on some of these, but for many others, we used default or assumed correct values, which means there is significant room for improvement.

5.2 Finding more features

The accuracy of our method for calculating circularity is not so satisfactory. Searching for the orientation of objects from the perspective of image moments and calculating the circularity may yield better results. We should also seek more resources and implement other recognition strategies.

5.3 Defining problem boundaries

Unless we continuously adjust the values of hyperparameters, our method shows significant fluctuations in recognition capability when the background color, the brightness of the light, or its direction changes. We need to specify more information, such as the environment of the data source or the proportion of the hand in the entire image, to make our program more robust. In this project, I gained experience processing images with OpenCV, but to achieve better results, I need to learn better methods or have a clearer understanding of the problem.

6 Conclusion

I learned how to process video streams, how to convert images to binary and analyze multiple properties, and also set up a usable visualization interface. The effectiveness of our model largely depends on the quality of the input images (or frames). Next time, I will consider a more fixed topic and seek advice from people with relevant experience (such as teaching assistants) more often.

7 Credits and Bibliography

The websites that were referred to have been added to the report in the form of hyperlinks. The lab sessions for the course were very helpful in completing the assignment.

Generative AI (mainly ChatGPT) was used to generate the initial code framework and for some translation tasks.