Actividad 3: Signal Identification

Curso: TE3002B.561 Implementación de Robótica Inteligente
Profesor: Diego López Bernal
Alumnos:

- Jennifer Lizeth Avendaño Sánchez - A01656951
- Juan Francisco García Rodríguez - A01660981

Fecha: 21 de mayo de 2024

1. Documentación del proceso

1. Se utilizan imágenes de señales de tráfico como referencia para el reconocimiento. Las imágenes se cargan en escala de grises, lo que simplifica el procesamiento porque solo se consideran las intensidades de luz y no los colores. La función `cv2.imread` se utiliza para cargar las imágenes de referencia de las señales de tránsito en escala de grises.

```
    right_arrow_img = cv2.imread('turnright.jpg', cv2.IMREAD_GRAYSCALE)
    left_arrow_img = cv2.imread('turnleft.jpg', cv2.IMREAD_GRAYSCALE)
    give_way_img = cv2.imread('giveaway.jpg', cv2.IMREAD_GRAYSCALE)
    work_in_progress_img = cv2.imread('workinprogress.jpg', cv2.IMREAD_GRAYSCALE)
    forward_arrow_img = cv2.imread('straigth.jpg', cv2.IMREAD_GRAYSCALE)
    turn_around_arrow_img = cv2.imread('turnaround.jpg', cv2.IMREAD_GRAYSCALE)
    stop_img = cv2.imread('stop.jpg', cv2.IMREAD_GRAYSCALE)
```

2. Se crea un objeto SIFT. SIFT (Scale-Invariant Feature Transform) es un algoritmo para detectar y describir características locales en imágenes. El objeto creado se utilizará para identificar puntos clave en las imágenes.

```
    sift = cv2.SIFT_create()
```

3. Para cada imagen de referencia, se detectan los *keypoints* y se calculan los descriptores. Los *keypoints* son características distintivas en una imagen, como esquinas o bordes. Los descriptores son vectores que describen las características alrededor de estos puntos clave.

```
    kp1, des1 = sift.detectAndCompute(right_arrow_img, None)
    kp2, des2 = sift.detectAndCompute(left_arrow_img, None)
    kp3, des3 = sift.detectAndCompute(give_way_img, None)
    kp4, des4 = sift.detectAndCompute(work_in_progress_img, None)
    kp5, des5 = sift.detectAndCompute(forward_arrow_img, None)
    kp6, des6 = sift.detectAndCompute(turn_around_arrow_img, None)
    kp7, des7 = sift.detectAndCompute(stop_img, None)
```

4. Se utiliza la cámara de la computadora para capturar video en tiempo real. Cada *frame* del video se procesará para detectar las señales de tránsito.

```
    cap = cv2.VideoCapture(0)
```

5. Se define el umbral mínimo de coincidencias, el cual determina cuántas coincidencias se necesitan entre los puntos clave del video y las imágenes de referencia para considerar que una señal ha sido detectada.

```
    MIN_MATCH_COUNT = 20
```

6. Se utiliza un buffer para almacenar los resultados de las últimas detecciones. Esto ayuda a estabilizar la detección y evitar falsos positivos, sobre todo en las señales de tránsito que tienen muchos puntos comúnes. Para este caso, el buffer se determinó de 7 elementos.

```
    buffer_size = 7
    results_buffer = []
```

7. Se inicia un bucle que se ejecuta continuamente para capturar y procesar cada *frame* del video.

```
    while True:
        ret, frame = cap.read()
        if not ret:
            break
```

8. Se convierte cada *frame* capturado del video a escala de grises para simplificar el procesamiento de la imagen. En este caso se utiliza `cv2.cvtColor`, función de OpenCV que convierte una imagen de un espacio de color a otro.

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

9. Similar a las imágenes de referencia, se detectan puntos clave y se calculan descriptores en el *frame* actual del video.

```
    kp_frame, des_frame = sift.detectAndCompute(gray, None)
```

10. Se establece un contador para las coincidencias de cada señal.

```
    match_counts = {
    'turnright': 0,
    'turnleft': 0,
    'give_way': 0,
    'work_in_progress': 0,
    'straight': 0,
    'turnaround': 0,
    'stop': 0
    }
```

11. Se comparan los descriptores del *frame* actual con los descriptores de las imágenes de referencia utilizando el algoritmo de coincidencia *BFMatcher* y un umbral para el filtrado de coincidencias. *BFMatcher* crea un objeto para el emparejamiento de descriptores utilizando el método *Brute-Force*, lo cual le permite obtener un conjunto de coincidencias que se utilizan para determinar si una señal de tránsito ha sido detectada.

```
    for name, (kp_ref, des_ref) in [('turnright', (kp1, des1)), ('turnleft', (kp2,
        des2)), ('give_way', (kp3, des3)), ('work_in_progress', (kp4, des4)), ('
        straight', (kp5, des5)), ('turn_around', (kp6, des6)), ('stop', (kp7, des7))]:

        bf = cv2.BFMatcher()
        matches = bf.knnMatch(des_ref, des_frame, k=2)

        good_matches = []
        for m, n in matches:
            if m.distance < 0.6 * n.distance:
                good_matches.append(m)

        match_counts[name] = len(good_matches)
```

12. Se identifica la señal que tiene la mayor cantidad de coincidencias en el *frame* actual.

```
    best_match_name = max(match_counts, key=match_counts.get)
```

13. El resultado de la señal detectada se almacena en el buffer, y si el buffer supera su tamaño máximo, se elimina el resultado más antiguo.

```
    results_buffer.append(best_match_name)
    if len(results_buffer) > buffer_size:
        results_buffer.pop(0)
```

14. Se calcula la moda del buffer para determinar la señal más común en los últimos *frames*.

```
    most_common_result = Counter(results_buffer).most_common(1)[0][0]
```

15. Si la cantidad de coincidencias de la mejor señal supera el umbral mínimo de coincidencias, se considera que la señal ha sido detectada y se muestra en la imagen la etiqueta de dicha señal. Para dibujar el texto en la imagen se utiliza la función `cv2.putText`.

```
    if match_counts[best_match_name] >= MIN_MATCH_COUNT:
        cv2.putText(frame, most_common_result, (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1,
            (0, 255, 0), 2)
```

16. Se muestra el *frame* procesado con la señal detectada.

```
    cv2.imshow('Traffic Signs Detection', frame)
```

17. El bucle de captura de video se cierra si se presiona la tecla 'x'. Finalmente, se liberan los recursos y se cierran todas las ventanas de OpenCV.

```
    if cv2.waitKey(1) & 0xFF == ord('x'):
        break

    cap.release()
    cv2.destroyAllWindows()
```

Antes de implementar SIFT (*Scale-Invariant Feature Transform*), se exploraron diversas técnicas más eficientes desde el punto de vista computacional para identificar las señales de tránsito, como la detección de contornos, HoughCircles y otros algoritmos similares; sin embargo, estas técnicas no lograron la precisión deseada. A pesar de su alto costo computacional, SIFT resultó ser altamente eficaz en la detección precisa de las señales. Se intentó también con ORB (*Oriented FAST and Rotated BRIEF*), un algoritmo más rápido que SIFT, pero los resultados no fueron satisfactorios, ya que no ofrecía la misma precisión. Por último, se consideró usar SURF (*Speeded-Up Robust Features*), que combina rapidez y precisión, pero este algoritmo no está disponible en la versión de OpenCV utilizada (cv2), lo que impidió su implementación. En última instancia, SIFT se mantuvo como la mejor opción debido a su robustez y fiabilidad en el reconocimiento de las imágenes deseadas.

Al principio, el algoritmo de reconocimiento de señales de tránsito se implementó utilizando exclusivamente el algoritmo SIFT junto con BFMatcher (*Brute-Force Matcher*). Aunque esta combinación era capaz de detectar varias señales, se presentaban numerosos errores en la identificación de señales similares, especialmente entre *turnright*, *turnleft* y *straight*. Estas confusiones ocurrían debido a las características visuales de estas señales, que comparten muchas similitudes, dificultando su correcta diferenciación solo con SIFT y BFMatcher. Para mejorar la precisión del reconocimiento, se introdujo el filtro de moda, que le brinda más estabilidad al algoritmo. Como se mencionó anteriormente, este filtro almacena los resultados de las últimas detecciones en un buffer y determina la señal más comúnmente detectada en ese periodo. De esta manera, se reduce la posibilidad de falsos positivos y se mejora la confiabilidad del sistema, ya que la decisión final se basa en una tendencia de las detecciones recientes en lugar de en una sola instancia que podría ser errónea.

En todas las pruebas realizadas, el sistema demostró un alto grado de acierto en la detección de las señales. Aunque en algunas ocasiones el algoritmo llega a confundir *turnright*, *turnleft* y *straight*, la implementación del filtro de moda mejoró significativamente esta parte, logrando que el sistema acierte con mayor frecuencia que los errores. El algoritmo fue probado en diversas condiciones de iluminación, incluyendo escenarios de poca luz y ambientes muy brillosos, y en todos los casos demostró ser confiable y robusto. Estos resultados destacan la eficacia de SIFT combinado con el filtro de moda, ofreciendo una solución viable y precisa para el reconocimiento en tiempo real.