

Reporte de Servicio Social

Victor Cruz y Gustavo Reyes

27 de septiembre de 2021

Índice

1. Detector de carril	1
2. Seguidor de carril	8
3. Detección de semáforos	19
4. Detector de peatones con YOLO versión 3	24
5. Detector de peatones con OpenVINO	36
6. Entrenamiento de RNA Yolov3	42

1. Detector de carril

El objetivo del detector de carril es detectar, únicamente, los carriles de una carretera a partir de la imagen original de la cámara de un coche, se usan librerías de OpenCV para cumplir este objetivo. Se puede ver con claridad en la imagen “RGB Image” de la figura 1 que las líneas son de color blanco. Esta propiedad se puede aprovechar para segmentar los carriles, ya que son los únicos que tienen la mayor cantidad de píxeles juntos de color blanco que cualquier otro objeto.

Para comenzar, usamos la imagen original de la cámara con la que puedes realizar cualquier tipo de programa que tenga que ver con la información que está provee. La imagen original es “BGR Image” de la figura 1.

La imagen “Binary image” muestra en color blanco todos los píxeles de la imagen original que son de color blanco y en negro todo lo que no es de color blanco. Para segmentar este color, es necesario usar la siguiente función:

```
1 cv2.inRange(original_image ,lower_white ,upper_white)
```

En la función anterior indicamos la imagen original, el umbral máximo y mínimo en el que se encuentra el color blanco.

Después de hacer algunas pruebas con herramientas web como “color picker” para saber el código del color blanco en hsv que nos proporciona la imagen de la cámara, se descubrió que en algunas partes de la carretera el color de la división de los carriles no es completamente blanco, es de un color que tiende a ser gris, por lo que es necesario ajustar el umbral mínimo, pues el máximo es

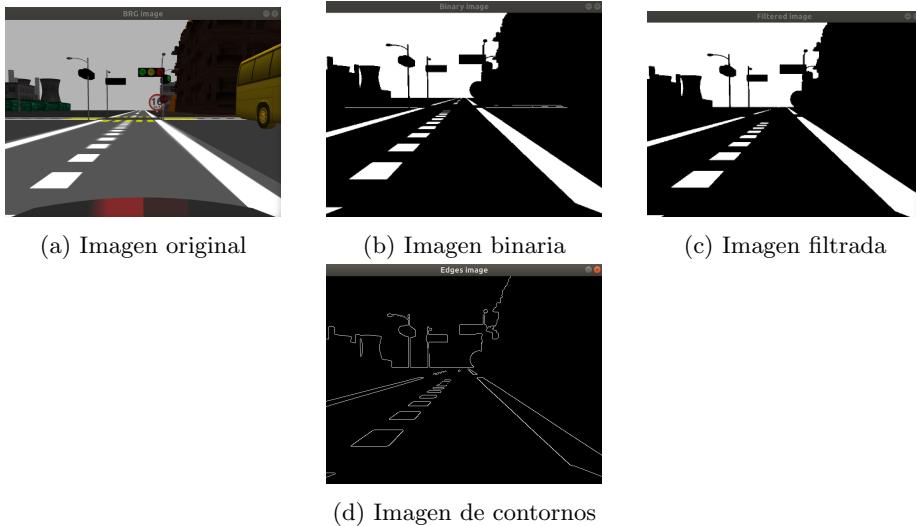


Figura 1: Imágenes procesadas

completamente blanco. De acuerdo a la imagen original, se llegó a la conclusión de que el valor de los umbrales debe ser el siguiente:

```

1 #threshold = [HUE, SATURATION, VALUE]
2 lower_white = [177,177,177] #gris claro
3 upper_white = [255,255,255] #blanco

```

Para continuar con la segmentación de los carriles es recomendable aplicar un filtro a la imagen “Binary image” con el objetivo de despreciar aquellos píxeles color blanco cuya densidad es mínima, es decir, despreciar píxeles blancos que están muy dispersos entre sí. En la imagen “Filtered image” de la figura 1 se puede ver el resultado de haber aplicado el filtro, las líneas de los carriles horizontales ya no aparecen en la imagen filtrada, ya que la cantidad de píxeles blancos que están juntos es mucho menor que la de los píxeles de los carriles verticales. Para obtener la imagen filtrada puedes usar la función cv2.morphologyEx(binary_image, cv2.MORPH_OPEN, kernel). Es necesario que en el primer parámetro de la función indiques la imagen que será filtrada y que en los dos últimos uses los datos que se recomiendan en la documentación de OpenCV.

Con el fin de identificar los carriles por medio de líneas, el primer paso es obtener los contornos de la imagen filtrada, el contorno nos proporciona información próxima a una línea. Para obtener la imagen de contornos “Edges image” de la figura 1, es necesario usar la función cv2.Canny(filtered_image, cv2.threshold1, threshold2). Se requiere indicar la imagen filtrada y los umbrales correspondientes. Después de hacer algunas pruebas, se llegó a la conclusión de que los umbrales recomendados son:

```

1 threshold1 = 100
2 threshold2 = 200

```

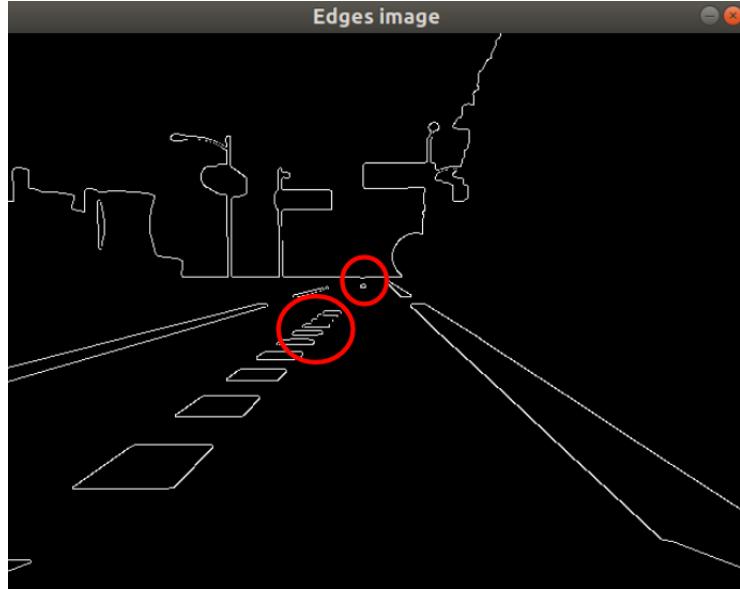


Figura 2: Parametro Threshold1 > 100 o Threshold1 < 100. Threshold2 > 200 o Threshold2 < 200

Si cambiamos estos umbrales aumentando o disminuyendo el valor recomendado, el comportamiento tiende a ser el siguiente. Figura 2.

En la figura 2 aparecen más contornos de color blanco y además comienzan a distorsionar el contorno original de la división de los carriles.

Podemos ver que desde la posición en la que se encuentra el coche, es posible interpretar la perspectiva y ver que el camino termina en un punto, formando un triángulo como el que se muestra en la figura 3.

Lo que realmente interesa es segmentar los carriles de la carretera, entonces se puede aprovechar esta geometría para reducir el campo de visión de estudio y obtener únicamente la imagen en la que aparecen los carriles. Observa la figura 4.

En la figura 4 se muestra la región de interés segmentada por un triángulo cuyos puntos están definidos en pixeles por coordenadas en el eje “x” y “y” de la imagen original. Los puntos con los que se construye el triángulo se definen de forma experimental de acuerdo a las dimensiones de la imagen, basta con indicar 3 puntos para construirlo. Se puede ver únicamente la imagen contenida en el interior del triángulo y se descarta todo lo que está en el exterior.

Para obtener la imagen de la figura 4 usamos las siguientes instrucciones:

```

1 def region_of_interest(img_edges):
2     height = img_edges.shape[0]          #Altura de la imagen en
3     width = img_edges.shape[1]           #Ancho de la imagen en
4     mask = np.zeros_like(img_edges)    #Arreglo de ceros con
                                         #las dimensiones de la imagen de contornos
5
6     #Definimos el triángulo
7     triangle = np.array([[
```

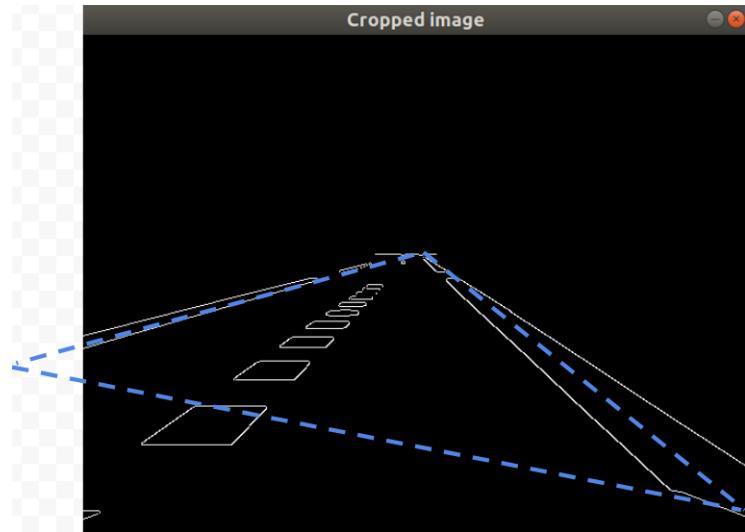


Figura 3: Perspectiva de inicio y final de la carretera en forma de triángulo.

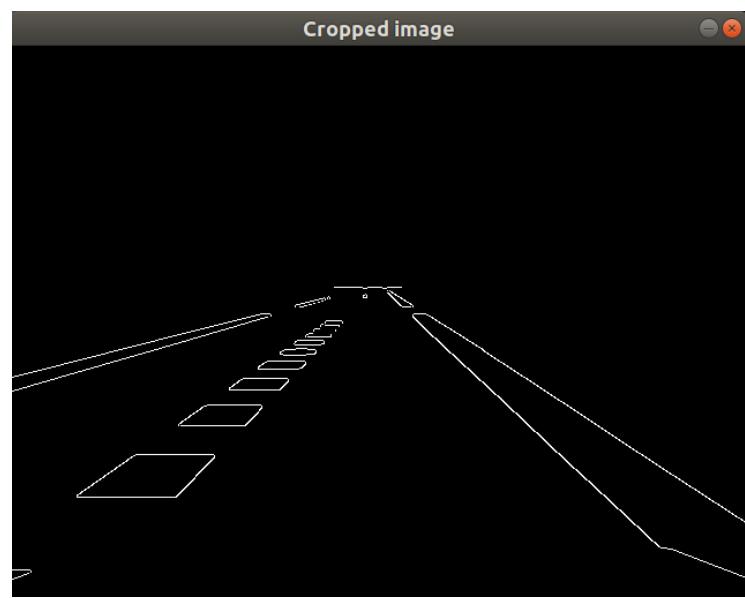


Figura 4: Imagen de los contornos de la sección de interés.

```

8      (-850, height), #Primer punto del triángulo.Vértice
9      inferior izquierdo
10     (320, 200),      #Segundo punto del triángulo. Vé
11     rtice central
12     (900, height)   #Tercer punto del triángulo. Vé
13     rtice inferior derecho
14     ],
15     np.int32)
16 cv2.fillPoly(mask, triangle, 255)
17 img_masked = cv2.bitwise_and(img_edges, mask)
18
19 return img_masked

```

Vemos que la función `region_of_interest()` recibe como argumento a la imagen de contornos de la figura 1.(b), partimos de esta imagen porque contiene, únicamente, los contornos de los elementos de la imagen original, esto quiere decir que posee pocos elementos y es más sencilla de procesar.

Por otro lado, la función `cv2.fillPoly()` sirve para enmascarar (quitar visibilidad) a toda la imagen con excepción de la parte interna de triángulo, y después de que la parte externa ya no es visible la eliminamos con la función `cv2.bitwise_and()` y por supuesto mantenemos la imagen interna del triángulo. Finalmente, la imagen `img_masked` que devuelve la función `region_of_interest()` es la que se muestra en la figura 4.

Esta imagen es la que necesitamos para calcular y dibujar las líneas de los carriles sobre la imagen original “BGR image”. Lo primero que hay que hacer, es calcular las líneas por medio de la función `cv2.HoughLinesP()`. Los parámetros que se usan son los siguientes:

```

1 cv2.HoughLinesP(
2 cropped_image,           #Imagen con la región de interés
3 rho = 1,                 #Valor recomendado en la documentación
4 theta = numpy.pi/180,    #Valor recomendado en la documentació
4     n
5 threshold = 60,          #Valor recomendado por experimentación
6 lines = None,            #Valor recomendado en la documentación
7 minLineLength = 175,     #Valor recomendado por experimentació
7     n
8 maxLineGap = 150         #Valor recomendado por experimentación
9 )

```

Cuando aumentamos el valor del parámetro “threshold” el número de líneas calculadas disminuye, es decir, si dibujamos todas las líneas calculadas por la función `cv2.HoughLinesP()` con un valor pequeño de “threshold”, por ejemplo 2, el número de líneas es mayor que si usamos un valor igual a 200. Se recomienda un valor de 60 para que el número de líneas sea parecido al número de divisiones de los carriles que hay en la imagen. En la figura 5 se muestra la diferencia entre un valor de “threshold” igual a 2, 60 y 200, respectivamente.

Cuando aumentamos el valor del parámetro “minLineLength” el número de líneas disminuye. Entre más pequeño sea este valor, el número de líneas aumenta. Tienen un comportamiento similar al del parámetro “threshold”. El valor que se ajusta bien en cuanto al número de líneas dibujadas y el número de líneas de división de carriles es de 175.

En cuanto al parámetro “maxLineGap”, entre menor sea el valor, las líneas tienden a ser más largas, entonces solo se calculan y dibujan líneas largas. Si el valor es grande solo se calculan y dibujan líneas cortas. Este parámetro también afecta el número de líneas, es necesario encontrar un equilibrio entre longitud y número de líneas. Tenga en cuenta que cada línea se dibuja con respecto a

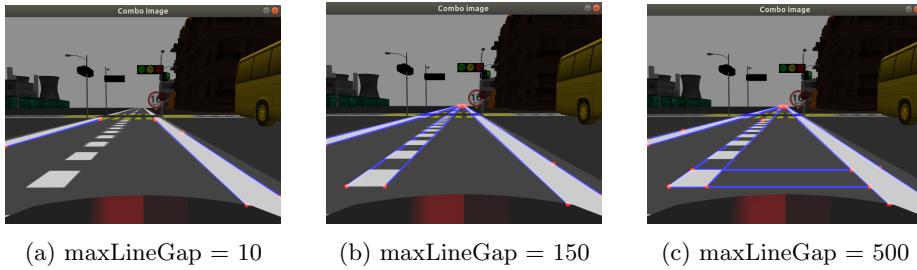


(a) threshold = 2

(b) threshold = 60

(c) threshold = 200

Figura 5: Imágenes con diferente valor del parámetro threshold



(a) maxLineGap = 10

(b) maxLineGap = 150

(c) maxLineGap = 500

Figura 6: Imágenes con diferente valor del parámetro maxLineGap

dos puntos con coordenadas “x” y “y” cada uno. Vea la diferencia entre usar un valor de “maxLineGap” de 10, 150 y 500 en la figura 6.

Se recomienda usar un valor de 150 para que la longitud y el número de líneas se ajuste al número de líneas de división de los carriles.

Con el procedimiento anterior y usando los valores recomendados, el resultado final se muestra en la figura 7.



Figura 7: Segmentación de los carriles de la carretera.

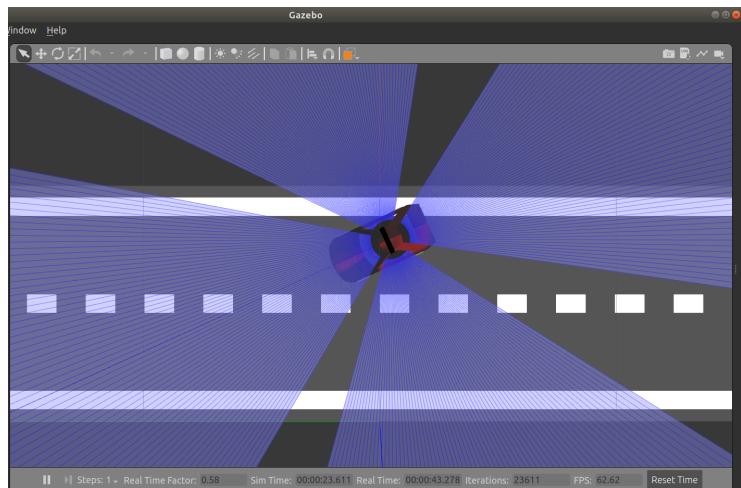


Figura 8: Al comienzo de la simulación el coche no está correctamente alineado con los carriles.

2. Seguidor de carril

El seguidor de carril es un programa que usa visión computacional para mover a un coche a escala por medio de la segmentación de los carriles de una carretera. Usamos los carriles ya segmentados para que el coche pueda avanzar sin salirse de su carril. Para fines didácticos y demostrativos, probaremos nuestro programa en una simulación tridimensional en la que vamos a poder ver interactuar al automóvil a escala en un circuito de asfalto. Usaremos el simulador 3D "Gazebo".

En la simulación se pretende que el coche sea capaz de seguir un carril recto sin salirse de él. Lo interesante es que el coche tiende a salirse del carril a propósito, este comportamiento lo indicamos en el programa y, afortunadamente, el control que implementamos evita que salga. Otro aspecto importante, es que podemos comprobar en la simulación que el control que programamos corrige la dirección del coche sin importar que no esté bien alineado desde el comienzo de la simulación.

En la figura 8 se muestra un ejemplo de las condiciones iniciales.

Para comenzar, usaremos las imágenes de la cámara que tiene el coche en el techo para poder ver los elementos que lo rodean y enfocarnos, únicamente, en los carriles blancos. Entonces para empezar con el programa. Requerimos de los siguientes paquetes de python:

- CvBridge: Convierte los mensajes de imágenes de ROS a imágenes de OpenCv.
- Cv2: Librería diseñada para resolver problemas de visión computacional.

Iniciamos obteniendo la imagen original que proporciona la cámara del coche con la siguientes líneas:

```
1 bridge = CvBridge()  
2 img_bgr = bridge.imgmsg_to_cv2(msg, "bgr8") #Convierte los  
mensajes de imágenes de ROS a imágenes de OpenCv en el  
espacio de color bgr, siguiendo el orden: azul-verde-  
rojo (blue-green-red)
```

Para visualizar la imagen, basta con usar las siguientes líneas:

```
1 cv2.imshow("Original Image", img_bgr) #Indicamos el nombre  
de la ventana en la que aparecerá la imagen y el nombre  
de la imagen que vamos a mostrar. Para todas las imá-  
genes que mostraremos más adelante hacemos exactamente  
lo mismo.  
2 cv2.waitKey(33) #Cada frame es mostrado aproximadamente por  
33 milisegundos. Es indispensable escribir esta línea  
para ver las imágenes, de otro modo no será posible.
```

A partir de la imagen original de la figura 9, segmentamos las líneas de los carriles. Aprovechamos que estos son los únicos elementos de color blanco. Usamos las siguientes líneas de código:

```
1 # [HUE, SATURATION, VALUE]  
2 lower_white = np.array([177,177,177]) #elegimos el umbral  
inferior del color blanco. [177,177,177] es gris claro  
3 upper_white = np.array([255,255,255]) #elegimos el umbral  
superior del color blanco. [255,255,255] es el color  
blanco
```

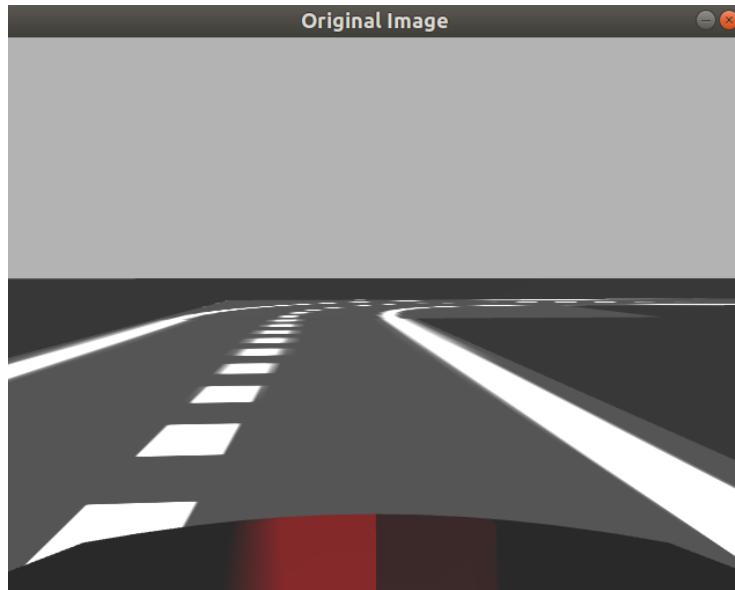


Figura 9: Imagen original de la cámara del coche.

```

4
5  #binary image
6 img-bin = cv2.inRange(img-bgr , lower-white , upper-white) #
    genera la nueva imagen en blanco y negro. Figura 10.
    Veremos en color blanco, únicamente, las líneas de los
    carriles y en negro todo lo que no sea de color blanco.
7
8 #filtered image
9 kernel = cv2.getStructuringElement(cv2.MORPHRECT,(2,2))
10 img-noise-filter = cv2.morphologyEx(img-bin , cv2.MORPH_OPEN
    , kernel)
11 #noise filter

```

Se recomienda generar la imagen de la figura 11 imagen en la que se eliminan los pequeños puntos que coinciden con el color que segmentamos, es decir, nos interesa segmentar el color blanco porque es el color dominante de las líneas de los carriles, pero puede haber algún píxel o un pequeño grupo de píxeles que también sean blancos, lo más probable es que no pertenezcan a las líneas de los carriles. Por esta razón filtramos la imagen binaria y eliminamos los pequeños grupos de píxeles mencionados.

Se puede ver que desde la perspectiva de la cámara las líneas de los carriles tienden a formar un triángulo, observa la figura 12.

A partir de esta observación, vemos que los elementos que están dentro del triángulo son las líneas de los carriles, entonces nos enfocaremos en esta sección de la imagen que contiene únicamente lo que nos interesa. A esta sección la llamaremos “región de interés”. Comenzamos por generar una nueva imagen en la que solo se muestre el interior del triángulo. Usamos las siguientes instrucciones:

```

1 img-edges = cv2.Canny(img-noise-filter , 100, 200) #
    Transformaremos las líneas de los carriles en líneas
    rectas más sencillas para aproximarlos a algo más
    parecido al triángulo azul de la figura 5. Se genera

```

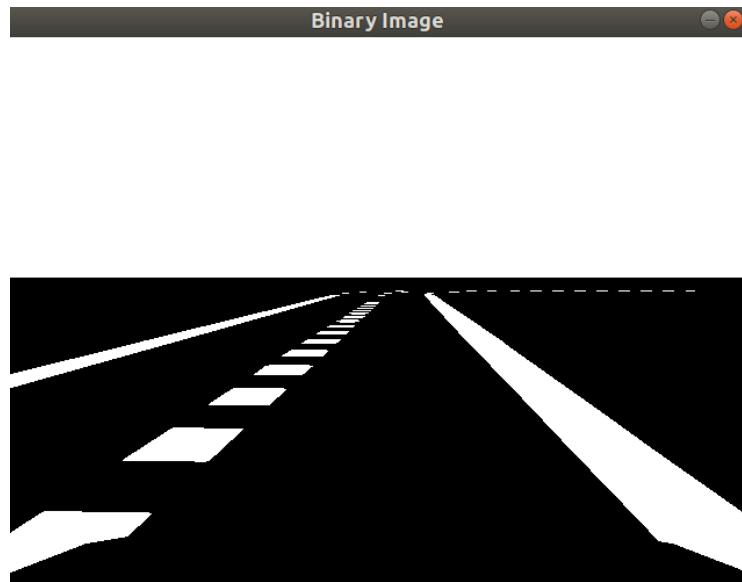


Figura 10: Imagen binaria “Binary Image”. En color blanco se muestra la segmentación del color de interés.

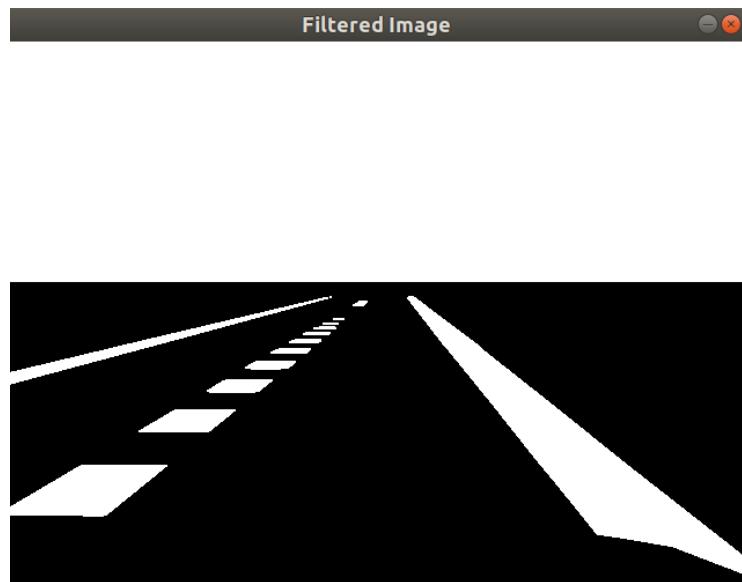


Figura 11: Imagen Filtrada “Filtered Image”. Se puede ver que es prácticamente igual a la imagen binaria, pues en la imagen binaria los únicos elementos blancos con las líneas de los carriles.

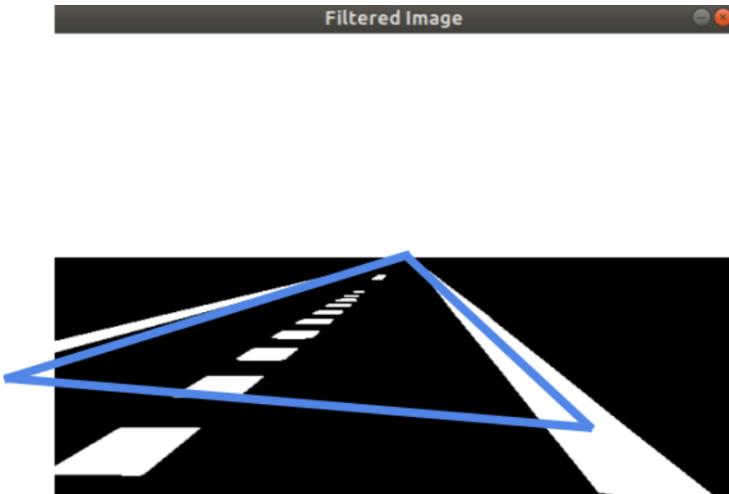


Figura 12: Forma de triángulo de los carriles vistos desde la perspectiva de la cámara.

una imagen con los contornos de los carriles. Veremos, únicamente, líneas que describen su perímetro. Vea la figura 13.

En la función cv2.Canny(img_noise_filter, 100, 200) indicamos la imagen de la que extraemos los contornos, el umbral inferior y superior. Los umbrales 100 y 200 fueron elegidos de acuerdo al procedimiento sugerido en el manual “lane detector” o detector de carril. Se recomienda echarle un vistazo.

```

1 def region_of_interest(img_edges):
2     height = img_edges.shape[0] #Obtenemos la altura de la
3         #imagen en pixeles
4     width = img_edges.shape[1] #Obtenemos el ancho de la
5         #imagen en pixeles
6     mask = np.zeros_like(img_edges)
7     #Definimos los 3 puntos que componen al triángulo
8     triangle = np.array([[
9         (-850, height),           #primer punto. Esquina inferior
10        izquierda
11         (320, 200),            #segundo punto. Punto intermedio
12         (900, height)          #tercer punto. Esquina inferior
13         derecha
14     ]], np.int32)
15     cv2.fillPoly(mask, triangle, 255) #Genera una máscara
16         que cubre la parte exterior del triángulo
17     img_masked = cv2.bitwise_and(img_edges, mask) #Elimina
18         los pixeles que están fuera del triángulo y conserva
19         la imagen interior
20     return img_masked
21
22
23 img_cropped = region_of_interest(img_edges) #En esta imagen
24     se muestra la imagen contenida en el interior del triángulo,
25     también le llamamos región de interés. Figura
26     14.

```

Ahora tenemos una imagen en la que las líneas de los carriles se parecen cada vez más a simples líneas rectas. Para obtener únicamente estas últimas,

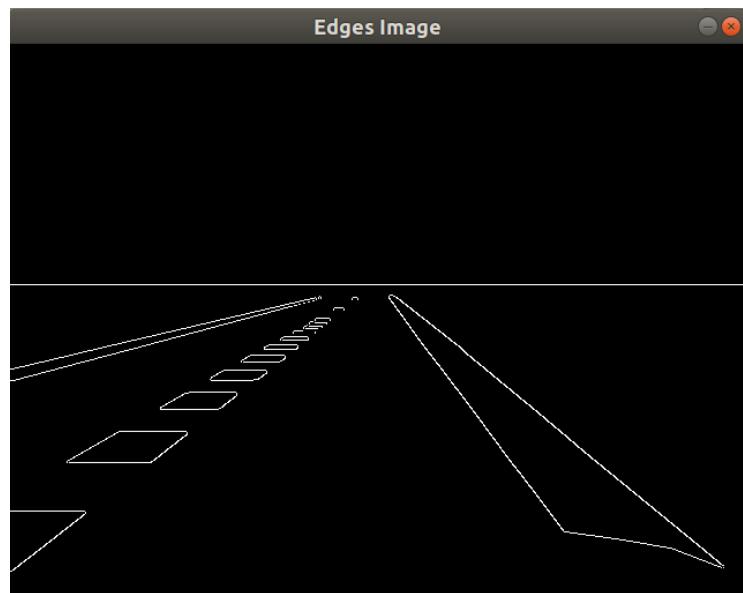


Figura 13: Imagen con los contornos de las líneas de los carriles.

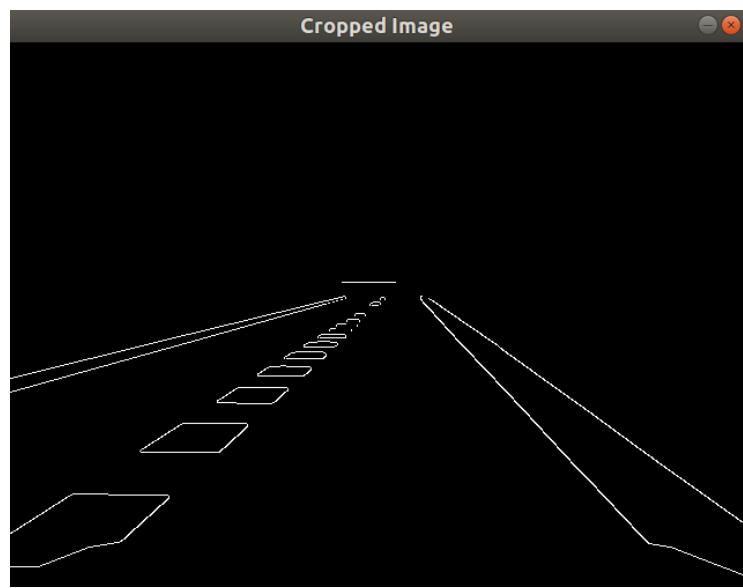


Figura 14: Imagen recortada. Región de interés.

usamos la transformada de línea Hough estándar y probabilística. Este método nos permite obtener líneas rectas mediante el el cálculo de dos puntos que al unirse forman una línea. Usamos las siguientes instrucciones:

```

1 def hough_lines(img_cropped): #img_cropped es la imagen con
    la sección de interés de la figura 14.
2     #probabilidad de donde pueden estar las líneas
3     lines = cv2.HoughLinesP(
4         img_cropped,
5         rho=1.0,
6         theta=np.pi/180,
7         threshold=60,
8         lines=None,
9         minLineLength=175,    #Entre más grande sea el valor ,
10        habrá menos líneas
11        maxLineGap=150)      #Entre mayor sea el valor las lí
12        neas serán más largas
13        return lines          #regresa una lista de líneas.
14        Coordenadas de los dos puntos que las componen
15        #lines list
16        lines = hough_lines(img_cropped) #llamamos a la función
17        hough_lines y devuelve una lista de líneas

```

El uso del valor de los parámetros rho,theta, threshold, lines, minLineLength y maxLineGap, se explican en el manual “lane detector” o detector de carriles.

Ahora que conocemos las coordenadas de los dos puntos que componen a cada una de las líneas, las dibujamos en una imagen para ver su ubicación. Lo hacemos de la siguiente manera:

```

1 def display_lines(img_bgr, average_lines):
2     #Dibujar líneas
3     #El número "3" representa los 3 canales rgb
4     img_lines = np.zeros((img_bgr.shape[0], img_bgr.shape
5                           [1],3), dtype=np.uint8)
6     line_color = [255, 0, 0]      #El color de las líneas será
7     azul
8     line_thickness = 4 #El grosor de las líneas será de 4
9     pixeles
10    dot_color = [0, 0, 255]      #El color de los dos puntos
11    de cada línea será rojo.
12    dot_size = 5 #El tamaño de los dos puntos de cada línea
13    será rojo
14    img_lines = display_lines(img_bgr, lines) #Imagen donde se
15    dibujan las líneas y sus dos puntos.

```

Mostramos la imagen en pantalla. Figura 15.

```

1 cv2.imshow("Lines image", img_lines)
2 cv2.waitKey(33)

```

Para visualizar las líneas de azules sobre la imagen original \bgr_img" basta con usar las siguientes instrucciones:

```

1 def add_weighted(img_bgr, img_lines):
2     #make lines more visible
3     try:
4         return cv2.addWeighted(src1=img_bgr, alpha=0.8, src2
5                               =img_lines, beta=1.0, gamma=0.0)
6     except:
7         pass

```

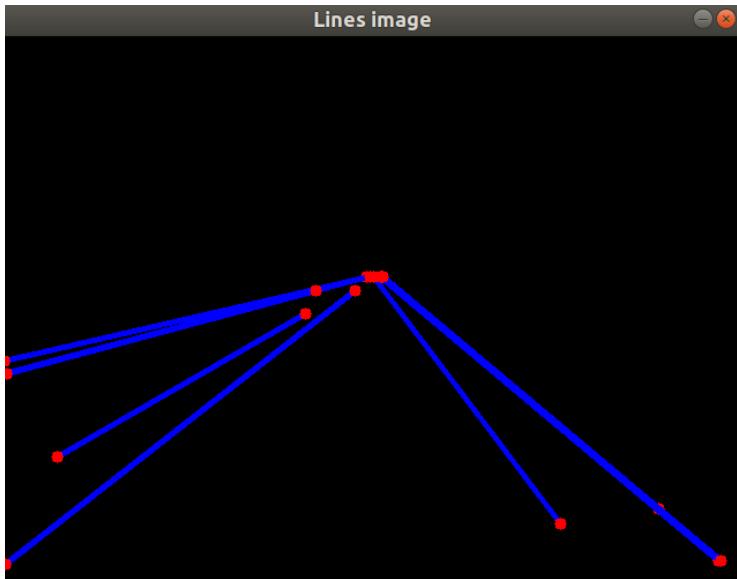


Figura 15: Representación de las líneas de los carriles por medio de líneas rectas.

```
7 img_overlaid = add_weighted(img_bgr, img_lines) #Imagen
    original con las líneas azules dibujadas sobre las lí
    neas de los carriles.
```

En el parámetro src1 indicamos la imagen sobre la que queremos visualizar las líneas azules y en src2 se indica la imagen que contiene a las líneas. El valor de los parámetros alpha,beta y gamma son los que se recomiendan en la documentación de OpenCv. Veamos el resultado en la figura 16.

Hasta aquí tenemos listo y funcionando todo lo relacionado a visión computacional con OpenCV. Lo que se requiere ahora es mover al coche. Recordemos que el coche tiene que avanzar dentro de su mismo carril, y en caso de desviarse hay que corregir la dirección. El control que vamos a usar corrige la dirección del coche comparando la inclinación de las líneas azules con la dirección del coche. El principal recurso serán las coordenadas de los puntos de las líneas azules de la figura 9 con los que calcularemos el ángulo de inclinación de cada línea. Estas líneas nos las devuelve la función `hough_lines` que usamos en proceso de visión computacional.

Para el cálculo de la pendiente de las líneas azules se usará la siguiente expresión matemática:

$$m = \frac{y_2 - y_1}{x_2 - x_1} \quad (1)$$

Se utilizará la siguiente ley de control para centrar al coche cuando este próximo a salirse de su carril:

Sí el ángulo de inclinación del coche o de las líneas azules es mayor a 36.5° quiere decir que el coche está centrado, entonces le asignamos una velocidad relativamente alta y un ángulo al volante que le permita estar alineado para que sea posible apreciar su desplazamiento. El coche está bien centrado cuando su

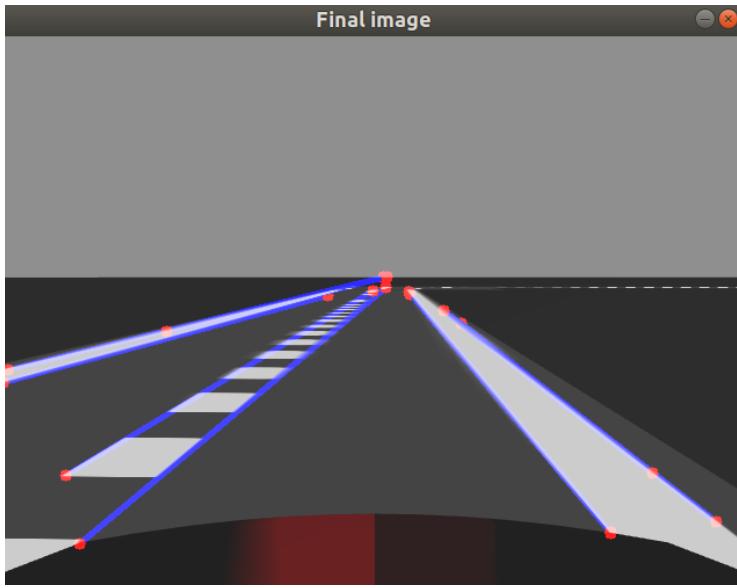


Figura 16: Líneas calculadas por medio de la transformada de línea Hough estándar y probabilística dibujadas sobre la imagen original.

ángulo de inclinación es de aproximadamente 38° , entonces aceptamos un error de 1.5° .

Si el ángulo es menor o igual a 36.5° , modificamos el ángulo del volante para que quede centrado, así mismo disminuimos su velocidad para que tenga oportunidad de centrarse y no salirse del carril. Usamos la siguiente expresión para corregir el ángulo:

errorDeAngulo = 39 - AnguloDeMayorMagnitudDeDesviacionDelCodigo

```

1 def calculate_inclination_angle_list(lines):
2     #A veces la longitud de las líneas len(lines) es igual a
3     #1
4     try:
5         #A veces el coche puede estar fuera de los carriles del
6         #camino y estos a la vez fuera del campo de visión de la
7         #cámara, entonces no habrá carriles, por lo que la
8         #lista de líneas no tendrá ningún elemento.
9         if len(lines) == 0:
10             print("No lines found")
11             pass
12         Else:
13             #lista donde se almacenará el ángulo en grados de inclinaci
14             #ón de cada línea:
15             slope_list = []
16             #obtenemos las coordenadas x1,y1 y x2,y2 de cada de las lí
17             neas
18             for line in lines:
19                 x1 = line[0][0]
20                 y1 = line[0][1]
21                 x2 = line[0][2]
22                 y2 = line[0][3]
23                 #cálculo de la pendiente

```

```

18         slope = (float(y2)-float(y1)) / (float(x2)-
19             float(x1)) #cálculo del ángulo de inclinación de la lí-
20             nea en grados
21             slope = (math.atan(slope)*180)/math.pi
22             slope_list.append(slope)
23             return slope_list
24         except:
25             pass
26 #Lista de ángulos en grados
27 angle_list = calculate_inclination_angle_list(lines)
28 if isinstance(angle_list, list) == True: #De esta manera
29     sabemos si la lista de ángulos es iterable o no. Cuando
30     no hay carriles dentro del campo de visión de la cá-
31     mara tampoco hay elementos numéricos (ángulos) dentro
32     de la lista. Cuando esto sucede, la lista angle_list es
33     un elemento de tipo NoneType que no es iterable.
34
35         upper_angle_list = [] #Aquí se almacenarán los á-
36         ngulos de mayor magnitud de cada lista de ángulos.
37         for angle in angle_list:
38             #Si el ángulo es menor que 0, se reciben las lí-
39             neas del carril derecho. Para fines didácticos de este
40             manual usaremos las líneas del lado derecho del carril
41             derecho como referencia.
42             if angle < 0:
43                 upper_angle_list.append(angle)
44 #Se ordenan todos los elementos de la lista
45         upper_angle_list en orden descendente:
46             upper_angle_list.sort()
47             #upper_angle_list [0] es el ángulo más grande de la
48             lista
49 #Cuando el coche está derecho las líneas azules tienen un á-
50             ngulo promedio de 38 grados, entonces el error aceptado
51             es de aproximadamente 1.5 grados para que se considere
52             que el coche está derecho con respecto a las líneas
53             del carril en el que se encuentra.
54             if abs(upper_angle_list [0]) > 36.5:
55 #Indicamos que el coche está derecho
56             print("The car IS straight")
57 #El rango de velocidad del coche está entre -2000 y 2000
58             unidades. Cuando la velocidad indicada es menor que
59             cero, el coche avanza y cuando la velocidad es mayor
60             que cero, retrocede.
61             vel.data = -500
62 #El coche está centrado cuando la dirección promedio es de
63             87.8 grados. Indicamos una dirección de 90 grados para
64             que el coche tienda a desviarse hacia la izquierda
65             mientras avanza y tengamos la posibilidad de observar
66             como nuestro controlador corrige la dirección.
67             direction.data = 90
68         else:
69             print("The car is NOT straight")
70             #Cálculo del error
71             angle_error = 39 - abs(upper_angle_list [0])
72             vel.data = -350
73             direction.data = 90 - (int(angle_error)*90)/39
74 #Publicamos la velocidad y dirección del carro.
75             pub_vel.publish(vel)
76             pub_direction.publish(direction)
77         else:
78             pass

```

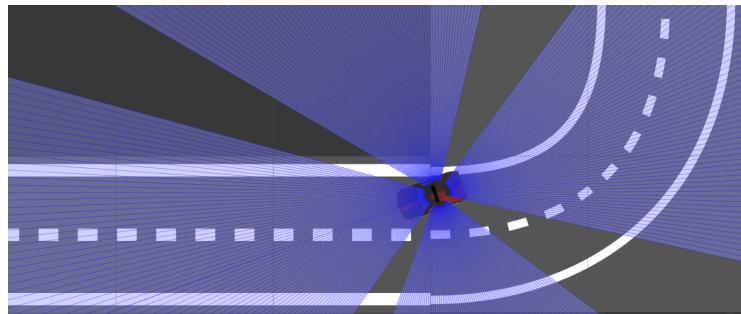


Figura 17: Inicio de la simulación con el coche inclinado.

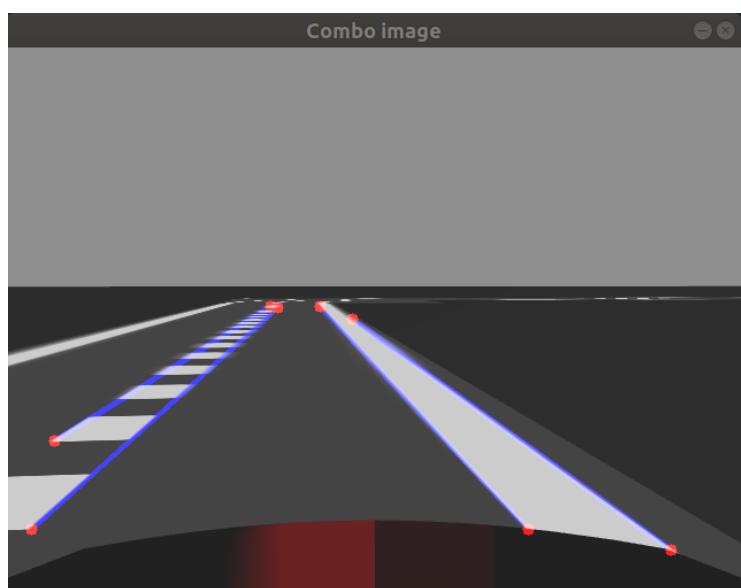


Figura 18: Se corrige la dirección del coche.

Podemos probar el funcionamiento de nuestro controlador corriendo la simulación con el coche un poco inclinado con respecto a las líneas de los carriles. Observa la figura 17.

En la figura 18 se puede ver como se corrige la dirección del coche.



Figura 19: Imagen original de la cámara de coche.

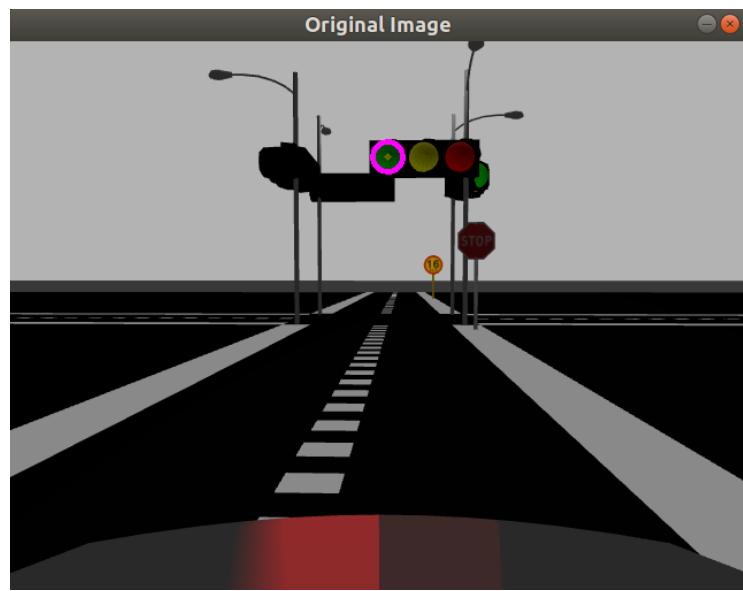


Figura 20: Luz verde del semáforo identificada por el círculo rosa.

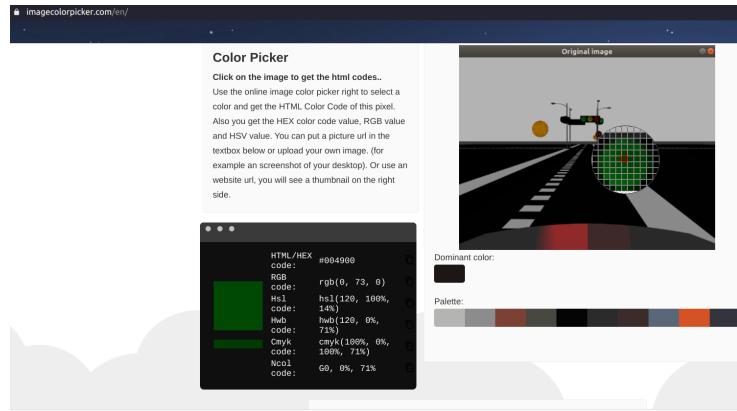


Figura 21: Uso de la herramienta “imagecolorpicker.com” para saber el código de color.

3. Detección de semáforos

El detector de color es un programa que usa visión computacional para detectar un color elegido por el programador. El objetivo de este programa es detectar cualquiera de los tres colores que posee un semáforo vial, verde, ámbar o rojo e identificar la zona del semáforo que posee este color. Observa la figura 19.

Para fines demostrativos vamos a elegir el color verde del semáforo y lo vamos a identificar encerrándolo dentro de un círculo color rosa. Figura 20.

La manera en la que obtenemos y mostramos la imagen original de la cámara se describe en el manual “Seguidor de carril”, en el que usamos el paquete “CvBridge” para convertir las imágenes de la cámara en mensajes de ros. Ahora que tenemos la imagen original debemos identificar el código de color verde del semáforo en el espacio de color HSV. Podemos usar herramientas web como “imagecolorpicker.com”, basta con proporcionar la imagen (puede ser una captura de pantalla) y poner el cursor sobre el color verde que nos interesa. Figura 21.

A partir del código de color HSV original que nos brinda “imagecolorpicker.com” experimentamos aumentando y disminuyendo los valores de Hue, Saturation y Value del umbral inferior y superior, tal como lo hicimos en el programa “Detector de carril”, en el que identificamos el umbral superior e inferior del color blanco. Puedes experimentar cambiando los parámetros HSV de los umbrales y observar los colores que se segmentan conforme haces cambios, esto te ayudará a identificar correctamente los píxeles que coinciden con el color verde del semáforo. Ten en cuenta que no todos los píxeles de color verde tienen el mismo código de color, si observas detenidamente la imagen con el zoom que hace “imagecolorpicker.com” podrás observar que algunos son más claros u oscuros que otros, por lo que es importante que ajustes los umbrales para que se segmenten correctamente los píxeles de color verde claro y oscuro que componen a la luz verde del semáforo. Después de varias pruebas y ajustes a los umbrales, el resultado es el siguiente:

```
1 lower_green = numpy.array ([0 ,40 ,0])      #Verde oscuro
```

```
2 upper_green = numpy.array([35,250,35]) #Verde claro
```

Para ver la imagen en la que aparece únicamente el color verde ya segmentado, hacemos lo siguiente:

- Generamos la imagen binaria en la que aparecerán en color blanco todas las zonas de la imagen original que son de color verde. Dicho de otra manera, en la imagen binaria se mostrarán en color blanco todos los píxeles de la imagen original que están dentro del umbral que definimos, y en color negro todos los píxeles que estén fuera del umbral. Figura 22.

```
1 img_bin = cv2.inRange(img_bgr, lower_green,  
2 upper_green)
```

- Se recomienda filtrar el ruido de la imagen binaria. Esto permite que los píxeles dispersos en las diferentes zonas de la imagen que están dentro del umbral sean eliminados y sólo permanezcan aquellos que están muy cerca uno del otro. Definimos la imagen filtrada:

```
1 img_filtered = cv2.morphologyEx(img_bin, cv2.  
2 MORPH_OPEN, kernel)
```

Para tener la imagen filtrada es necesario que, primeramente, definamos un kernel:

```
1 kernel = cv2.getStructuringElement(cv2.MORPH_RECT  
2 ,(4,4))
```

Los parámetros del kernel son los que se recomiendan directamente en la documentación de OpenCv. El kernel sirve para definir la tolerancia de los píxeles coincidentes con los umbrales que serán eliminados, en este caso si en lugar de definir este vector (4,4) lo definimos como un vector más pequeño (2,2), se eliminarán menos píxeles, por el contrario si lo definimos un vector más grande (8,8) se eliminarán más píxeles. Figura 23. Se puede ver que en la figura 1 que los únicos elementos que son de color verde son las luces verdes de dos semáforos, por lo que el filtro no es necesario pero se acostumbra a usarlo porque el programa puede ser usado en cualquier otra simulación en la que haya más elementos.

- Finalmente, para visualizar la segmentación en el color original, usamos la función `cv2.bitwise_and` con los parámetros que se recomiendan en la documentación de OpenCV:

```
1 img_filtered_color = cv2.bitwise_and(img_bgr,  
2 img_bgr, mask=img_bin)
```

En el primer y segundo argumento indicamos la imagen original y el tercer argumento indicamos la imagen binaria, o bien la imagen filtrada en caso de que sea necesario usar el filtro y el resultado es el siguiente. Figura 24.

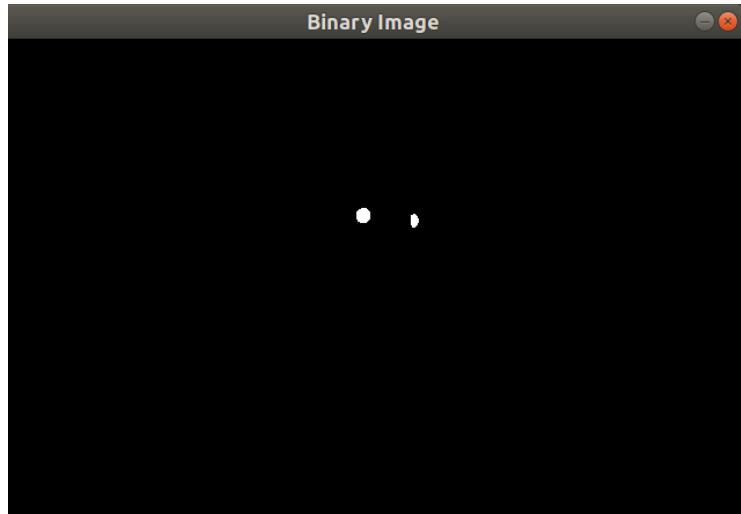


Figura 22: Imagen binaria. Segmentación del color verde del semáforo.

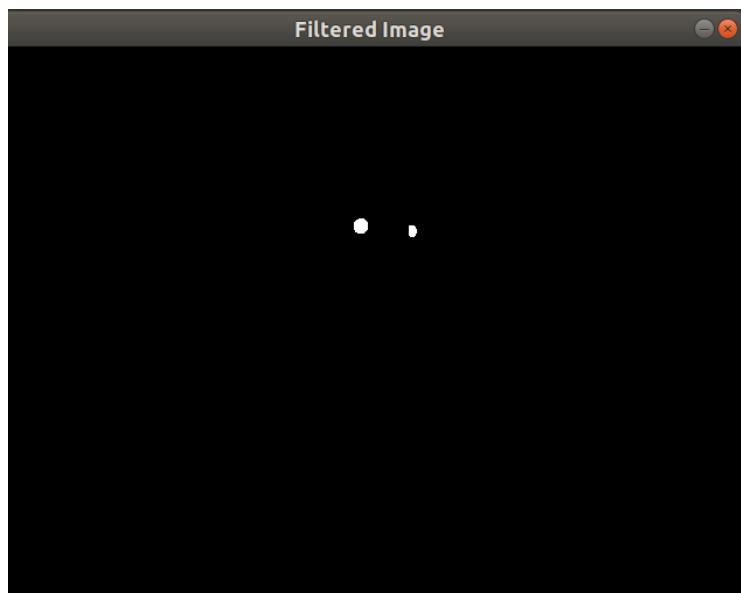


Figura 23: Imagen filtrada.

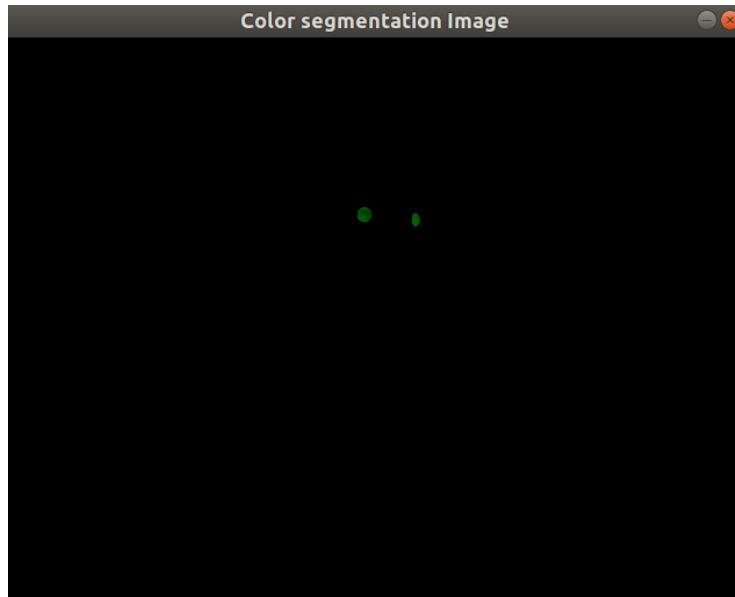


Figura 24: Segmentación final del color verde del semáforo.

En la figura 24 se puede ver que se han detectado dos elementos de color verde. Si comparamos esta imagen con la figura 1, vemos que corresponden a las luces verdes de los semáforos. Si queremos aproximarnos a la realidad lo más prudente sería poner atención al semáforo que corresponde al carril en el que transitamos, por esta razón implementaremos un algoritmo que se encargue de detectar únicamente los elementos circulares de la imagen y que los resalte con un color (usaremos el color rosa). Esto quiere decir que la luz verde del semáforo que tenemos de frente será la única resaltada por una circunferencia rosa, ya que la luz verde que está a la derecha se parece más a la mitad de un círculo que a un círculo completo.

Para detectar elementos circulares usaremos el algoritmos propuesto en la documentación de OpenCV llamado “HoughCircles” que es similar al algoritmo “HoughLines” que usamos en el manual de detección de líneas, la diferencia es que “HoughCircles” detecta geometrías circulares en lugar de líneas. Para empezar, convertimos la imagen segmentada de la figura 6 a escala de grises usando la siguiente línea recomendada en la documentación de OpenCv:

```
1 gray = cv2.cvtColor(img_filtered_color, cv2.COLOR_BGR2GRAY)
```

En la instrucción anterior la función “cv2.cvtColor” requiere de dos parámetros, el primero es la imagen filtrada de la figura 6 y el segundo es la instrucción recomendada en la documentación de OpenCv para convertir la imagen a escala de grises. El resultado es el siguiente. Figura 25.

Se sugiere emborronar o empañar a la imagen en escala de grises para depreciar contornos irregulares y claridad excesiva de la imagen, en este caso se pretende que las orillas o perímetros irregulares de las luces segmentadas se deprecien para que se vean mas redondas y con menos picos. Usamos la siguiente instrucción:

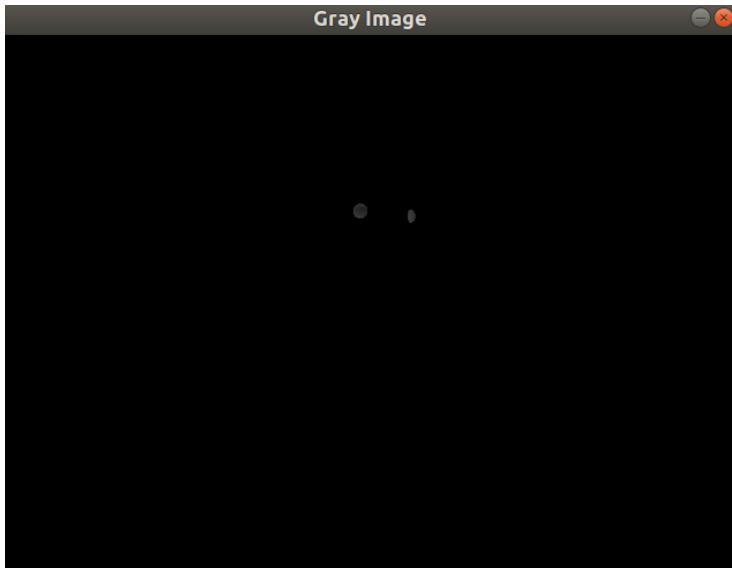


Figura 25: Imagen segmentada en escala de grises.

```
1 gray = cv2.medianBlur(gray, 5)
```

La función “cv2.medianBlur” requiere de dos parámetros, el primero es la imagen en escala de grises y el segundo indica que tanto desenfoque le queremos aplicar, cuando se incrementa el número del último parámetro el desenfoque aumenta y si este decrementa, el desenfoque será menor. Observa la figura 26.

La diferencia entre la figura 25 y 26 no es evidente porque la geometría de las luces segmentadas son circulares y semicirculares, no hay picos ni irregularidades pronunciadas. En otros casos el desenfoque resulta mucho más útil para difuminar las irregularidades.

Ahora aplicamos el algoritmo de detección de círculos que se encarga de resaltar, únicamente, la luz verde del semáforo que está justamente en frente del coche. Podemos encontrar este algoritmo en la documentación de OpenCv y podemos usarlo tal como se recomienda pero hay que hacer unos cuantos ajustes para que funcione propiamente. El algoritmo es el siguiente:

```
1 rows = gray.shape[0]
2 circles = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, 1,
3 rows / 8,
4                                     param1=1, param2=20,
5                                     minRadius=6, maxRadius=20)
6 if circles is not None:
7     circles = np.uint16(np.around(circles))
8     for i in circles[0, :]:
9         # Cálculo del centro del círculo
10        center = (i[0], i[1])
11        # Se dibuja el centro del círculo
12        cv2.circle(img_bgr, center, 1, (0, 100, 100), 3)
13        # Se calcula radio del círculo
14        radius = i[2]
15        # Se dibuja el contorno o perímetro del círculo
16        cv2.circle(img_bgr, center, radius, (255, 0,
```

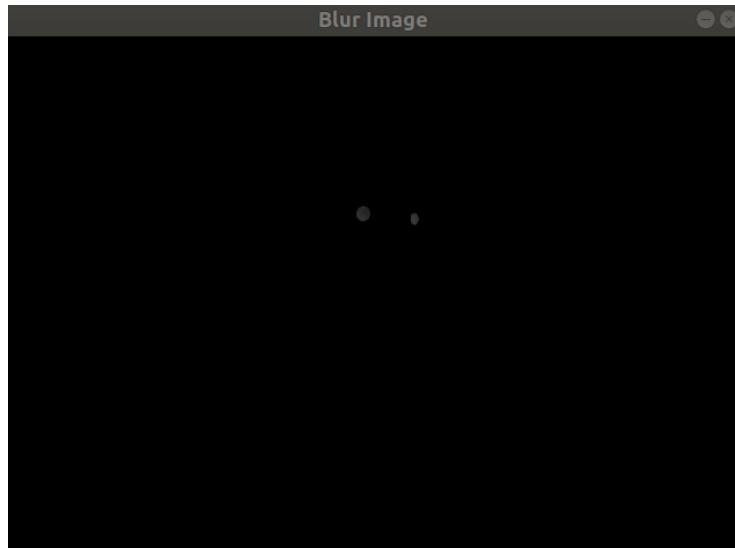


Figura 26: Imagen en escala de grises con desenfoque.

Haremos los únicos ajustes necesarios en los parámetros de la función “HoughCircles”, específicamente, en el primer argumento especificaremos el nombre de la imagen de la que se detectarán los círculos, en este caso “gray”. En el cuarto argumento indicamos la distancia mínima que habrá entre los círculos detectados. En caso de que la distancia entre círculos sea muy grande, se calcularán únicamente círculos que estén alejados uno del otro, por el contrario, si la distancia definida entre círculos es pequeña, se calcularán círculos más cercanos uno del otro. La distancia entre círculos se mide del centro de un círculo al centro de otro. Finalmente, la función “HoughCircles” devuelve los círculos encontrados en la imagen que le pasamos como parámetro “gray”.

La función “cv2.circle” dibuja el centro del círculo y para ello requiere de los siguientes parámetros: (imagen sobre la que se dibujara el centro, centro del círculo devuelto por la función “HoughCircles”, radio del círculo, color del centro en espacio de color rgb en formato de tupla, ancho). Para dibujar el contorno del círculo usamos exactamente la misma función y sólo cambiamos el radio y el color.

El resultado final del programa detector de luces de semáforo se puede ver en la figura 20 en donde se puede observar que se señala o resalta con un círculo de color rosa la luz verde del semáforo que está justamente delante de la cámara.

4. Detector de peatones con YOLO versión 3

El detector de personas es un programa que puede identificar personas dentro de una imagen a partir de algoritmos de código abierto de detección de objetos en tiempo real como YOLOv3 (You Only Look Once) y OpenCV. Anteriormente, se ha estado trabajando con OpenCV, por lo que ya se tiene una cantidad de herramientas suficientes para poder trabajar con algunas herramientas simples de YOLO versión 3.

Para comenzar, es importante tener una idea general de lo que se va hacer, en pocas palabras, el detector de personas consta de tres pasos importantes que se desarrollarán a en este documento:

- Se verá cómo preparar las imágenes para entrenar al modelo de reconocimiento de personas con YOLOv3.
- Se verá cómo cargar las imágenes a los servidores de google (la nube) para proceder con el entrenamiento del modelo. Desde ahora se debe tener en cuenta que no se requiere tener una computadora ni tarjeta gráfica poderosas, todo el proceso de entrenamiento se hará en los potentes servidores que google nos presta de forma gratuita, más adelante veremos las ventajas y limitaciones de este servicio de google.
- Se probará el modelo generado en el paso dos y se verá si realmente funciona.

Primeramente, se prepara el set de datos o dataset de imágenes. El dataset contiene muchas imágenes en donde aparece el objeto al que interesa hacer el reconocimiento, en este caso, el objeto que interesa es el de un peatón que transita normalmente por las calles pero hay que tener en cuenta que se puede elegir el objeto que se deseé. Para echar un vistazo a las imágenes y archivos que se usarán en este ejemplo, visite el siguiente enlace: <https://drive.google.com/drive/folders/11CfPcR3UkhfEJMuY8lbe3c1KotMuIQ6I?usp=sharing> y abra el directorio llamado \pedestrian_detector".

Dentro del directorio principal \pedestrian_detector" hay una carpeta que se llama "images", dentro de ella se encuentran diez imágenes diferentes de peatones con las que se estará trabajando. Es importante señalar que entre más imágenes se tengan y que además sean diferentes (diferentes escenarios, con objetos diferentes alrededor, ángulos, acercamiento, iluminación, etcétera) es mucho mejor. Por ejemplo, cuando se trabaje con proyectos robustos se recomienda usar al menos cien imágenes y si son más imágenes, mejor.

Ahora que ya se tienen identificadas las imágenes, se requiere descargar un programa gratuito que no requiere ser instalado llamado LabelImg que está disponible para Windows, Linux y MacOs. A continuación se proporciona el enlace a su sitio web para descargarlo:

LabelImg: <https://tzutalin.github.io/labelImg/>

En el sitio de descargas, sólo se puede encontrar el programa para Windows y para Linux, pero también existe para MacOs y lo podemos encontrar en el siguiente sitio:

LabelImg: <https://pypi.org/project/labelImg/>

Este programa sirve para preparar las imágenes y tenerlas listas para entrenar al modelo de reconocimiento de objetos con YOLOv3. Si se trabaja con Windows se recomienda ampliamente descargar la última versión de LabelImg, a fecha de hoy 16 de noviembre de 2020 la versión más reciente es: Windows_v1.8.0. Si se trabaja con Linux también se recomienda descargar la versión más reciente. Al día de hoy la versión más reciente del para linux es: Linux_v1.4.3 y por desgracia esta versión y las anteriores no cuentan con la herramienta que se necesita para poder preparar las imágenes para entrenar al modelo con YOLOv3, por lo que si se trabaja con linux, se puede hacer la preparación de las imágenes en otro sistema operativo como Windows o MacOs.

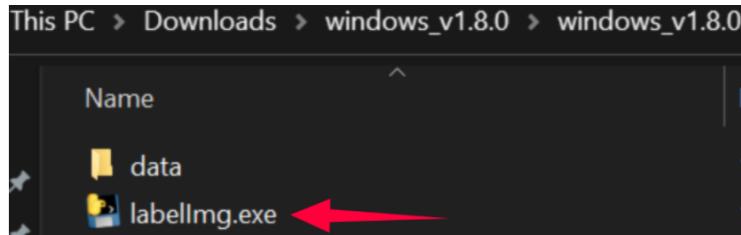


Figura 27: Archivo ejecutable del programa labelImg.

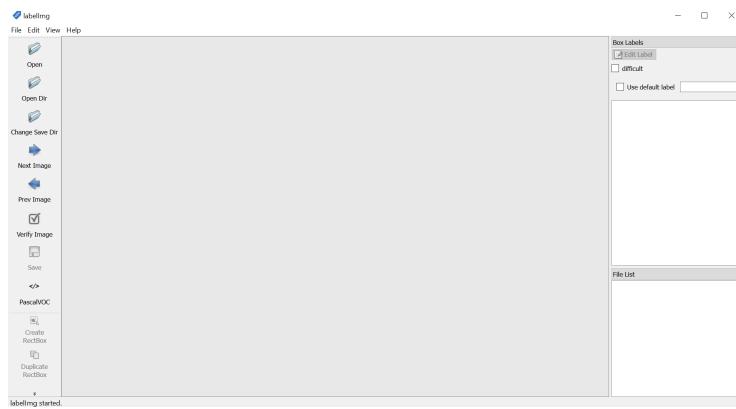


Figura 28: Interfaz del programa: labelImg.

La descarga de labelImg nos genera una carpeta, dentro de ella hay un archivo ejecutable que se debe ejecutar para abrir el programa, figura 27. Al ejecutar el programa se abre la interfaz de la figura 28.

Una vez abierto el programa labelImg se selecciona el directorio en el que se encuentran las imágenes:

- Del lado izquierdo de la ventana seleccione el icono “Open dir”. Figura 29.
- Busque y seleccione el directorio con las imágenes. Figura 30.

Ahora se requiere seleccionar el directorio en el que se guardaran los archivos necesarios para realizar el entrenamiento del modelo:

- Del lado izquierdo de la ventana seleccione el icono “Change Save Dir”. Figura 31.
- Seleccione el mismo directorio en el que están las imágenes. Figura 32.

Se trabajará con YOLOv3 para entrenar el modelo, entonces se requiere configurar al programa labelImg para trabajar con el formato que requiere yolo:

- En la parte izquierda de la ventana se puede ver que por default labelImg trabaja con el formato de “PascalVOC”. Figura 33.
- Es necesario cambiar el formato de PascalVOC a YOLO. Entonces de click sobre el icono de “PascalVOC” para hacer el cambio. Figura 34.

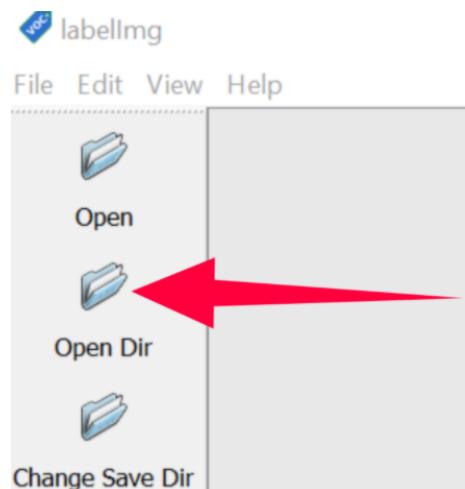


Figura 29: Icono “Open Dir”

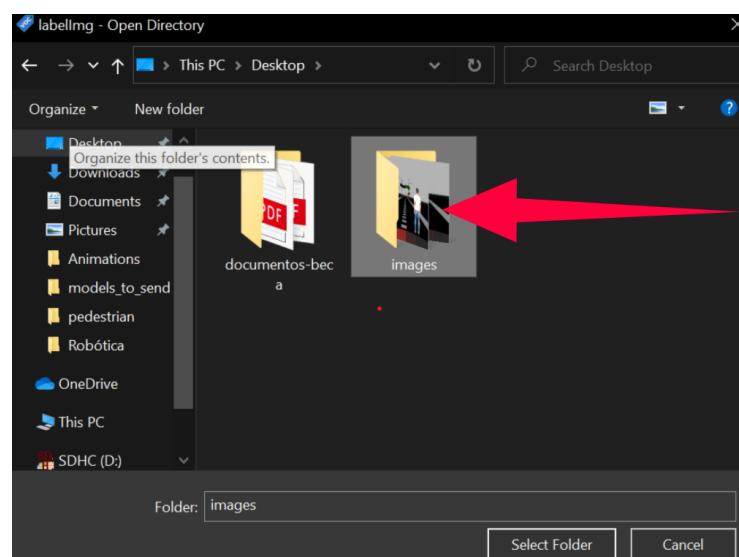


Figura 30: Selección del directorio con las imágenes.



Figura 31: Icono “Change Save Dir”

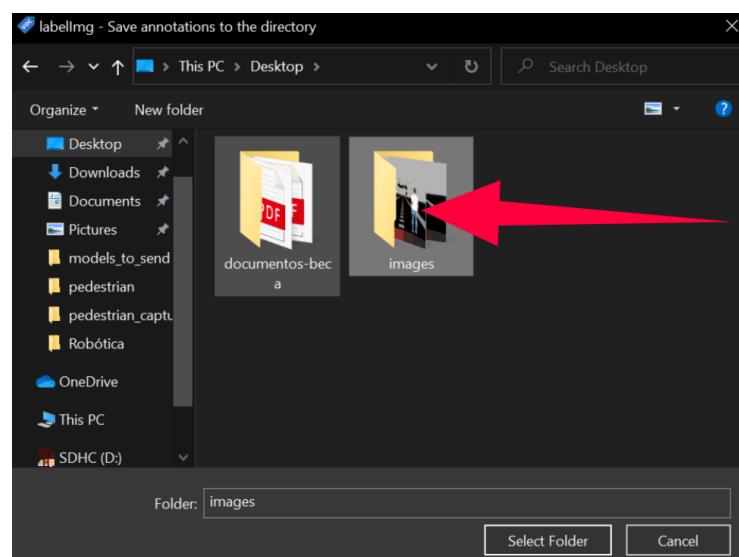


Figura 32: Selección del directorio con las imágenes.

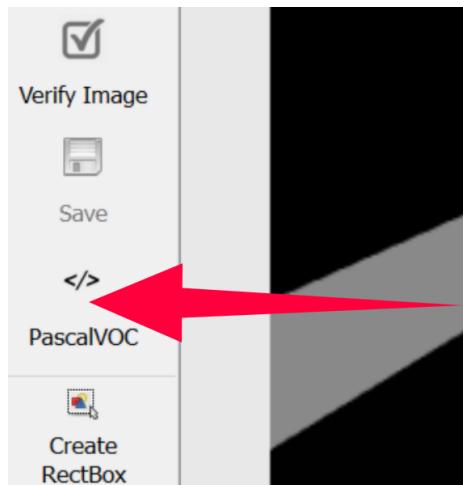


Figura 33: Icono “PascalVOC”

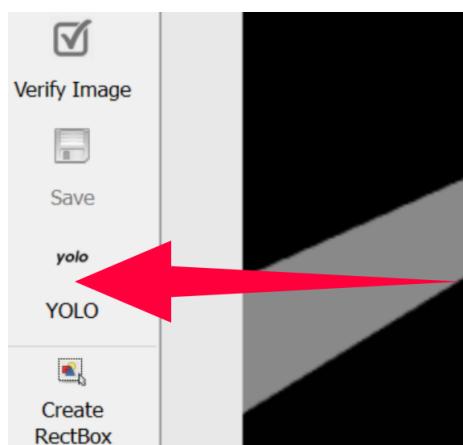


Figura 34: Cambio del formato PascalVOC al formato YOLO.

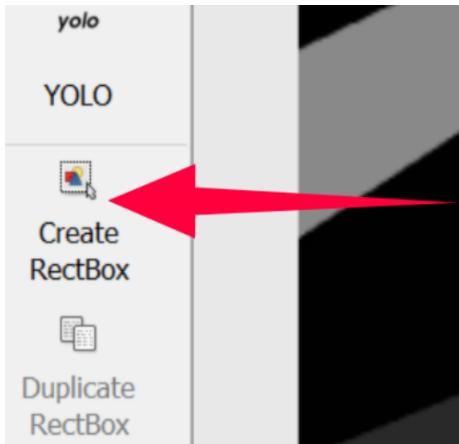


Figura 35: Icono Create RectBox

Con las configuraciones anteriores ya es posible comenzar a preparar las imágenes. Primeramente, debe seleccionar el ícono “Create RectBox” que se encuentra del lado izquierdo de la ventana. Figura 35.

Una vez seleccionado aparecerá un cursor en forma del símbolo “+” con el cual se seleccionará dentro de un rectángulo al objeto de interés, para este ejemplo, el objeto de interés es la persona que aparece en la imagen. Figura 36.

Para etiquetar la imagen de acuerdo a una categoría, el programa muestra una ventana que contiene una lista con algunas sugerencias. En el caso de que el elemento de interés no se encuentre en la lista es posible proponer el nombre de etiqueta escribiendo directamente en el espacio en blanco de la figura 11 y presionar “OK”. En este ejemplo el objeto de interés se encuentra en la lista con el nombre “person” (persona), si se desea usar esta etiqueta selecciónela o proponga el nombre que prefiera y seleccione “OK”. Figura 37

Después de etiquetar la imagen seleccione el ícono “Save” para guardar la etiqueta y su relación con la imagen seleccionada. Figura 38.

Ya se ha seleccionado al objeto de interés de la primera imagen y se han guardado los cambios. Ahora seleccione el botón “Next Image” para realizar el mismo proceso con las nueve imágenes restantes. Figura 39.

Para cada imagen etiquetada se generará un archivo de texto en la carpeta “images”, este archivo está asociado a la imagen, la asociación se puede ver por el nombre que le asigna el programa a cada archivo de texto, el nombre es el mismo que el de su imagen correspondiente. Cuando termine de etiquetar a todas las imágenes el contenido de la carpeta “images” será el de la figura 40.

Ahora seleccione todos los archivos que están dentro de la carpeta “images” (imágenes y archivos de texto) y guarde todo dentro de un archivo comprimido “.zip” llamado “images.zip”. Para simplificar la comprensión de este ejemplo se proponen los nombre de los archivos pero hay que tener en cuenta que no tienen que ser estrictamente los mismos. El archivo “images.zip” será el dataset que se usará para entrenar al modelo de reconocimiento de personas con YOLOv3.

Es momento de cambiar a Google Colab. Este es un servicio que ofrece Google para que los usuarios puedan usar GPUs gratuitas y listas en cualquier momento. Para comenzar con Google Colab:



Figura 36: Selección del objeto de interés.

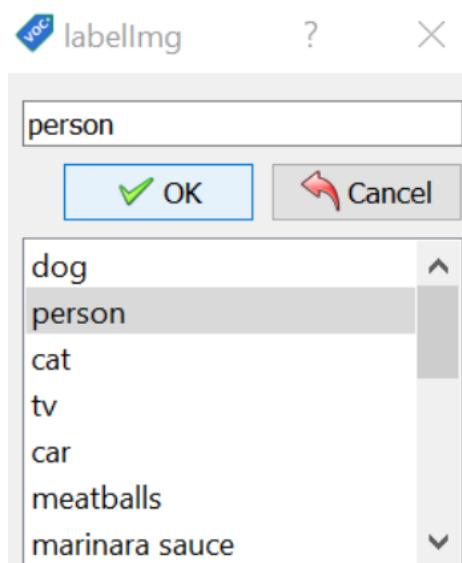


Figura 37: Etiquetar imágenes.

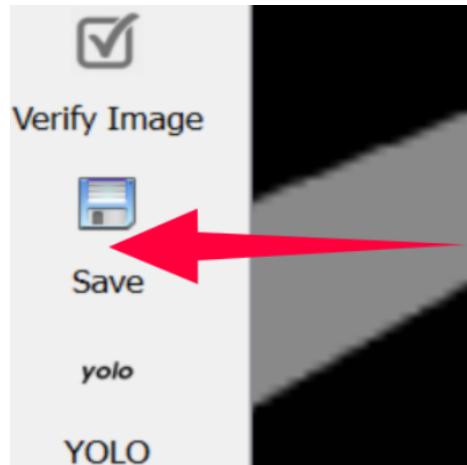


Figura 38: Botón “Save”.



Figura 39: Icono “Next Image”

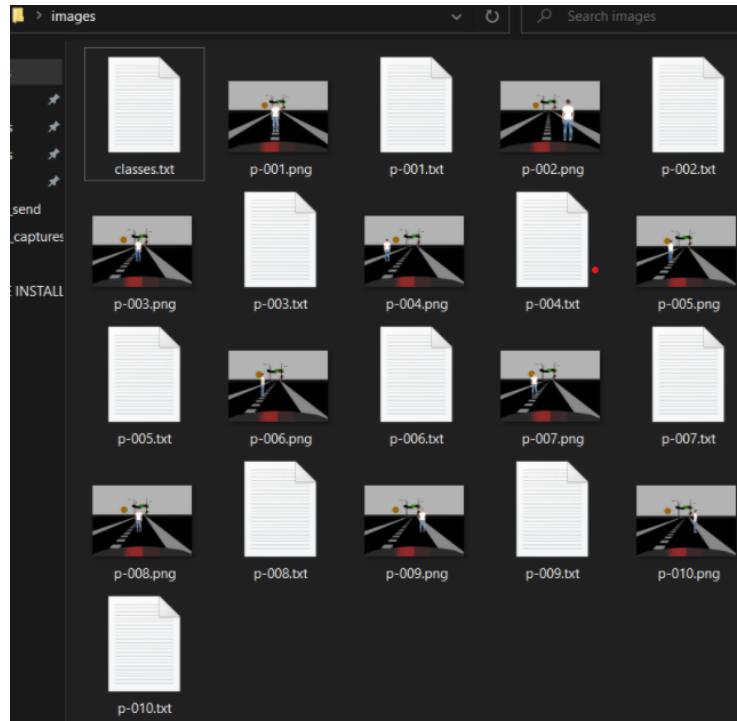


Figura 40: imágenes con sus respectivas etiquetas.

- Inicie sesión con su cuenta de Google y diríjase a Google Drive
- Genere un directorio llamado “yolov3” en la raíz de su espacio de Google Drive
- Dentro de la carpeta “yolov3” cargue el archivo “images.zip”
- Descargue el archivo \Train_YoloV3.ipynb” del siguiente enlace y almacene el archivo en su espacio de Google Drive: <https://drive.google.com/drive/folders/11CfPcR3UkhfEJMuY81be3c1KotMuIQ6I?usp=sharing>
- Busque en su navegador de internet Google Colaboratory e inicie sesión si aún no lo ha hecho.
- Diríjase al menú de la parte superior, seleccione “Archivo” y de la lista que se despliega seleccione “Cargar cuaderno”. Aparecerá la ventana de la figura 41.
- Arrastre el o cargue el archivo “Train_YoloV3.ipynb” a la ventana de la figura 41. Después de subir el archivo se abrirá el archivo en cuaderno que tiene el aspecto de la figura 42.
- Ahora diríjase al menú superior y seleccione “Runtime” o “Tiempo de ejecución”, se despliega una lista de opciones, seleccione “Run all” o “Ejecutar todo” y aparecerán algunos datos técnicos de la GPU que Google le está prestando. Así mismo, aparece un campo en blanco que pide que introduzca su código de autorización. Figura 43.

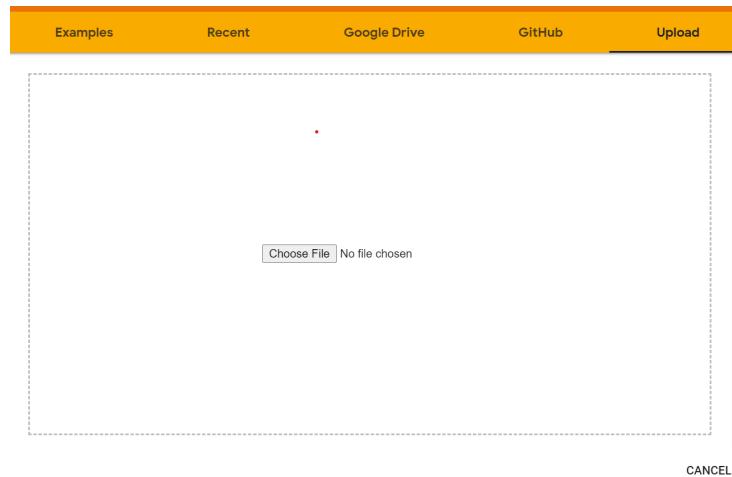


Figura 41: Ventana para cargar y abrir el archivo “Train_YoloV3.ipynb”.

```

Train_YoloV3.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Connect google drive
[ ] # Check if NVIDIA GPU is enabled
!nvidia-smi

[ ] from google.colab import drive
drive.mount('/content/gdrive')
!ln -s /content/gdrive/My\ Drive/ /mydrive
!ls /mydrive

1) Clone the Darknet

[ ] !git clone https://github.com/AlexeyAB/darknet

2) Compile Darknet using Nvidia GPU

[ ] # change makefile to have GPU and OPENCV enabled
%cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!make

```

Figura 42: “Train_YoloV3.ipynb” abierto en un cuaderno de Google Colaboratory.



```

from google.colab import drive
drive.mount('/content/gdrive')
!ln -s /content/gdrive/My\ Drive/ /mydrive
!ls /mydrive

... Go to this URL in a browser: https://accounts.google.com/

Enter your authorization code:

```

Figura 43: “Campo en blanco para introducir código de autorización.

- De click en el enlace de color azul, acepte los términos y condiciones de Google, copie el código de autorización que se le proporciona, peguelo dentro del campo en blanco de la figura 43 y presione “Enter”. Se sugiere que lea los términos y condiciones de Google antes de aceptar cualquier cosa.
- Ahora espere que el programa de entrenamiento termine de ejecutarse. Esto puede tomar varias horas.
- Al finalizar el entrenamiento, dentro del directorio “yolov3” se generarán algunos archivos con la extensión “.weights”. De todos los archivos, descargue el que se llama “yolov3_training_last.weights”, este es el archivo que contiene la información completa del entrenamiento. Se sugiere que cambie el nombre por uno que se relacione con la detección de personas que se pretende hacer, para este ejemplo se llamará “pedestrian_detector.weights”.
- Ahora diríjase al siguiente enlace <https://drive.google.com/drive/folders/11CfPcR3UkhfEJMuY81be3c1KotMuIQ6I?usp=sharing>, abra el directorio llamado “pedestrian_detector” y descargue el archivo “pedestrian_detection.py”. Este es el programa principal con el que se mostrará el funcionamiento del detector de personas.

Asegúrese de tener en un mismo directorio los siguientes archivos:

- “pedestrian_detection.py”
- “Pedestrian_detector.weights”
- “Yolov3_configuration.cfg”
- El directorio con las imágenes originales
- También puede descargar del siguiente enlace <https://drive.google.com/drive/folders/11CfPcR3UkhfEJMuY81be3c1KotMuIQ6I?usp=sharing> el directorio llamado “pedestrian_detector” que ya contiene todos los archivos necesarios.
- Abra el archivo “pedestrian_detection.py” con su editor de texto preferido para observar la configuración que se ha hecho para que el programa acceda a los demás archivos del directorio.

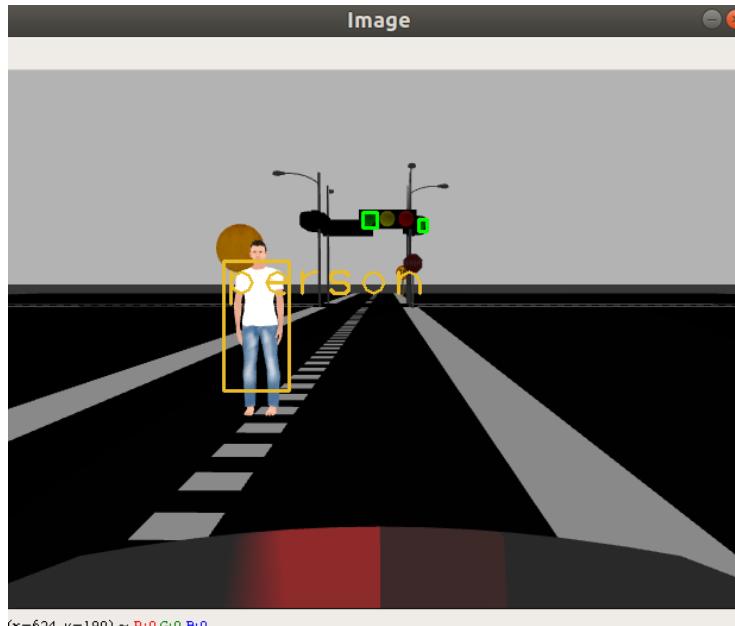


Figura 44: Resultado final del programa detector de personas.

- Ejecute el programa con python 3, ya que la versión 3 de yolo no funciona con versiones inferiores a python 3.
- Al ejecutar el programa aparecerá la ventana de figura 44 en la que se detecta a la persona que aparece en la imagen. Para hacer la prueba de la detección de peatones con las diez imágenes cambie de imagen con la flecha derecha del teclado hasta haber usado todas las imágenes. Con esto concluye el programa detector de personas.

5. Detector de peatones con OpenVINO

El detector de personas de este documento es un programa que identifica personas dentro de una imagen usando algoritmos de detección de objetos en tiempo real. Se usarán las herramientas de OpenVINO para facilitar la optimización de un modelo de aprendizaje profundo en hardware Intel.

Para tener una idea más clara de lo que se pretende hacer, se hará una lista de los pasos importantes que se desarrollarán:

- Obtener y etiquetar las imágenes con las que se entrenará el modelo de reconocimiento de personas.
- Entrenar el modelo de reconocimiento con herramientas de OpenVINO como tensorflow.
- Entrenar el modelo de reconocimiento con herramientas de OpenVINO como tensorflow.

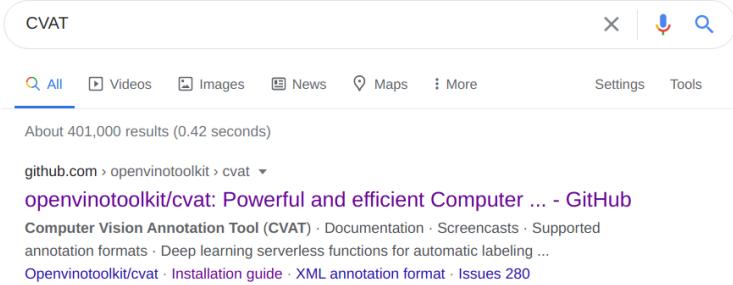


Figura 45: Página de Github de CVAT.

Primeramente, es necesario preparar el set de datos o dataset de las imágenes. El set de datos contiene varias imágenes en las que aparece el objeto al que interesa hacer el reconocimiento. En este documento el objeto de interés es un peatón o persona que transita por las calles de una simulación de RVIZ, en todas y cada una de las imágenes del set de datos aparece el peatón en diferentes ángulos de visión respecto a la cámara. Hay que tener en cuenta que se puede elegir cualquier objeto que se desee reconocer en imágenes.

Para ver el set de datos con el que se trabajará, visite el siguiente enlace: shorturl.at/tBLV1 y abra el directorio llamado “images”. En este directorio hay diez imágenes en formato PNG en las que aparece el objeto de interés, es decir, el peatón.

Como comentario adicional, cabe señalar que a mayor número de imágenes (en diferentes escenarios, objetos diferentes alrededor del objeto de interés, ángulos de la cámara, acercamiento, iluminación, etc) es mejor. Cuando se trabaja con proyectos más robustos se recomienda usar al menos cien imágenes para etiquetarlas y realizar el entrenamiento del modelo de reconocimiento de objetos.

Una vez que se tienen listas las imágenes ya es posible realizar el etiquetado. Para ello se usará la herramienta web CVAT (Computer Vision Annotation Tool).

- Escriba en su navegador “CVAT”
- Diríjase a la página de Github. Figura 45.
- En la página de Github de CVAT hay una sección llamada “Documentation”, diríjase al primer apartado llamado “Installation guide”. Figura 46.
- Elija el sistema operativo con el que esté trabajando. En este caso se elegirá la opción **Ubuntu 18.04 (x86_64/amd64)**.
- Siga las instrucciones de instalación copiando y pegando los comandos en su terminal. Al final generará un usuario y contraseña.
- Una vez terminada la instalación de los requerimientos de CVAT se le pedirá que se dirija a la siguiente dirección **localhost:8080** en su navegador CHROME.

Documentation

- [Installation guide](#)
- [User's guide](#)
- [Django REST API documentation](#)
- [Datumaro dataset framework](#)
- [Command line interface](#)
- [XML annotation format](#)
- [AWS Deployment Guide](#)
- [Frequently asked questions](#)
- [Questions](#)

Figura 46: Guía de instalación de CVAT.

- Escriba su usuario y contraseña para acceder a la herramienta web de CVAT.
- Dentro de la interfaz web de clic sobre el botón azul **Create new task**.
- Ahora se requiere proporcionar la información del proyecto. Figura 47.
- En el requerimiento **Name** escriba el nombre del proyecto. En este caso se decidió llamarlo **Person detector**.
- No es necesario llenar el apartado **Project**.
- Debajo del apartado **Labels** hay dos opciones: **Raw** y **Constructor**, seleccione **Constructor** y abajo aparecerá una opción llamada **Add label**, selecciónela. Escriba el nombre de la etiqueta. En la figura 3 se el nombre de la etiqueta es ”pedestrian”, se hace referencia al objeto de interés pero puede escribir el nombre que desee. Una vez proporcionado el nombre de la etiqueta de clic sobre el botón azul **Done**.
- En el apartado **Select files** cargue las imágenes que desee etiquetar. En este ejemplo se cargaron las diez imágenes en formato PNG que se proponen al inicio del documento.
- Una vez cargadas las imágenes de clic sobre el botón azul **Submit**.
- En la esquina superior izquierda de la interfaz hay dos opciones: **Projects** y **Tasks**, seleccione la opción **Tasks**, se desplegará la tarea que acaba de crear, ábrala dando click sobre el botón **Open**.
- En la parte inferior de la tarea aparece una sección llamada **Jobs**, dentro de esta sección aparece en color azul **Job** , de clic sobre las letras azules. Aparecerá una interfaz como la de la figura 48.
- Con esta herramienta ya es posible realizar el etiquetado de las imágenes. Para el etiquetado es necesario seleccionar el objeto de interés dentro de un

Create a new task

Basic configuration

* Name
Pedestrian detector

Project:
Select project

* Labels:
 Raw Constructor Copy
 pedestrian Add an attribute +
 Done Continue Cancel

* Select files:
 My computer Connected file share Remote sources
 Click or drag files to this area
 Support for a bulk images or a single video

> Advanced configuration

Submit

Figura 47: Información del proyecto.

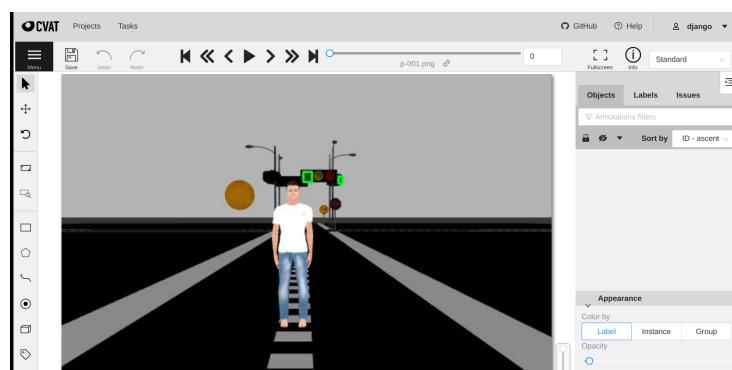


Figura 48: Interfaz de CVAT.

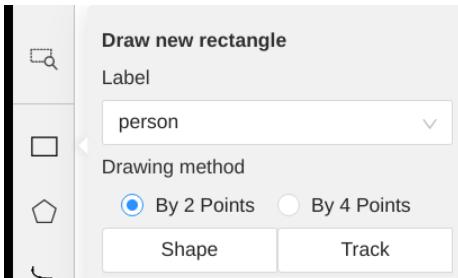


Figura 49: Herramienta delimitadora de objetos.

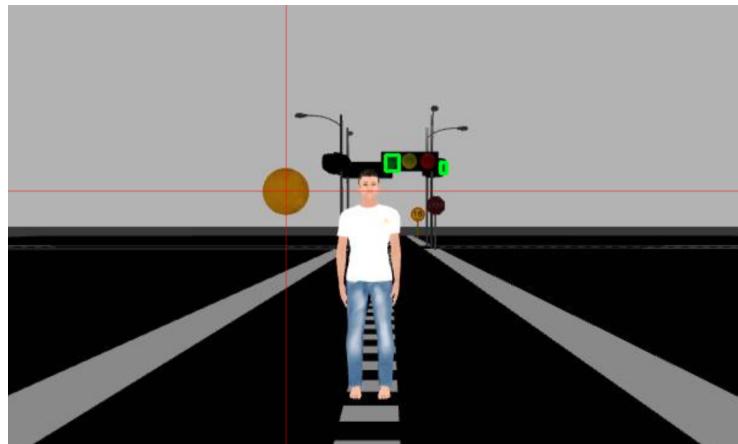


Figura 50: Cursor con líneas rojas.

rectángulo para delimitarlo de otros objetos a su alrededor. Seleccione el rectángulo que aparece en la parte izquierda de la ventana, al seleccionarlo se recomienda usar la opción “By two points” y “Shape”. Figura 49.

- Al seleccionar la herramienta “Draw new rectangle” con el ícono de un rectángulo aparece un cursor con líneas rojas que ayuda a dibujar el rectángulo de delimitará el objeto. Figura 50.
- Delimite el objeto de interés dentro de un rectángulo. Figura 51.
- Presione la tecla “F” para cambiar a la siguiente imagen. Es necesario realizar el mismo proceso para cada una de ellas, es decir, para cada imagen debe limitar al objeto dentro de un rectángulo para que pueda ser etiquetado.
- Cuando termine de etiquetar todas las imágenes guarde los cambios dando click sobre el botón “Save” que está en la esquina superior izquierda. Figura 52.
- Para finalizar con el etiquetado, diríjase al “Menu” y seleccione la opción “Dump annotations”. Aparece una lista de diferentes formatos para el set de datos. En este ejemplo se selecciona el formato “COCO 1.0”. Al seleccionar el formato, se descarga automáticamente un archivo JSON

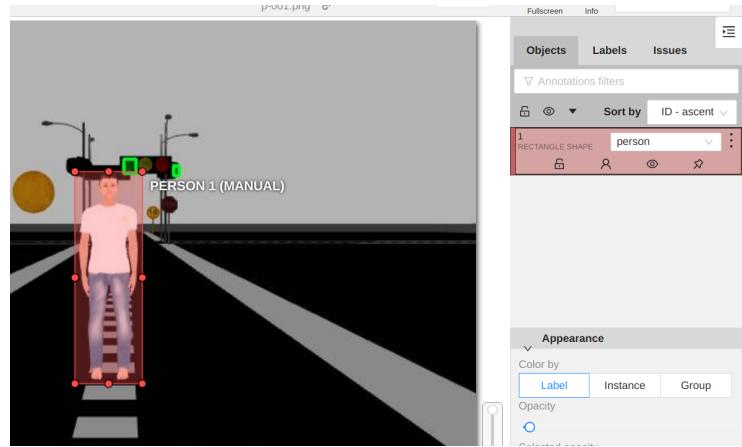


Figura 51: Cursor con líneas rojas.

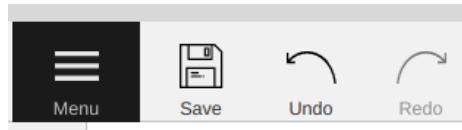


Figura 52: Botón guardar.

con la información de todas las etiquetas. Este archivo se usará para el entrenamiento. Figura 53.

- En la carpeta compartida `shorturl.at/tBLV1` puede encontrar el archivo JSON con el siguiente nombre: `instances_default.json`.

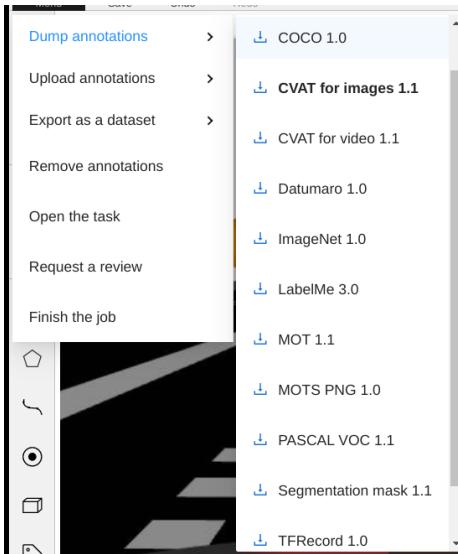


Figura 53: Elección del formato del set de datos.

6. Entrenamiento de RNA Yolov3

Este apartado explica el procedimiento para entrenar una red neuronal artificial basada en la arquitectura Yolov3 para la detección de objetos relevantes en un ambiente de simulación de un vehículo autónomo. El procedimiento involucra la generación del dataset para el entrenamiento, el entrenamiento de la red como tal, así como la exportación de la matriz de pesos y los vectores de sesgos de la matriz a un script que pueda ejecutarse de manera local.

El primer paso para el entrenamiento de la red neuronal consiste en la generación de los datasets para entrenamiento y validación de la red. Este dataset consistió en un conjunto de alrededor de 2 mil capturas de pantalla, las cuales contienen instancias de los elementos que se desea entrenar a la red para que los aprenda a identificar. Estas imágenes deberán de ir acompañadas de un conjunto de etiquetas, las cuales indiquen la ubicación de los elementos relevantes para el entrenamiento de la red. Los elementos escogidos para su identificación fueron: semáforo, señal de límite de velocidad, señal de stop, y paso de cebra.

El procedimiento para generar el dataset de entrenamiento comienza con la generación de un archivo de video de la simulación en ejecución, en el que se realice un recorrido dentro de la simulación que capture los elementos a identificar en múltiples posiciones, variando su cercanía, ubicación y orientación. El objetivo de este video es generar tantas instancias tan variadas como sea posible de los elementos a identificar. Para este ejemplo se utilizó el software Open Broadcast Software, <https://obsproject.com/es>.

Una vez generado este video, se utiliza el siguiente script de Python para extraer una porción de los frames del video, y guardarlas como imágenes .jpg, las cuales serán posteriormente etiquetadas para usarlas en el entrenamiento de la red neuronal.

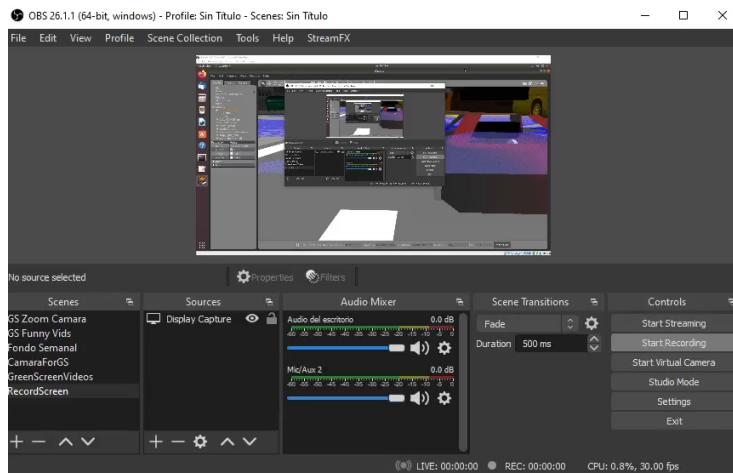


Figura 54: Captura de video de la simulación

```

1 import cv2
2 import time
3 # Capturar video
4 cap = cv2.VideoCapture('SS_AutoNOMOS_Video03.mp4')
5
6 # Fps
7 index = 1485
8 contador = 0
9
10 # Verificar video abierto
11 if cap.isOpened() == False:
12     print("Error al abrir archivo de video")
13
14 while cap.isOpened():
15     # Leer archivo de video
16     ret, frame = cap.read()
17     if ret == True:
18         cv2.imshow('frame', frame)
19         if contador == 16:
20             cv2.imwrite('SS_ImagenesBruto_Video03\\{}.jpg'.format(index), frame)
21             contador = 0
22             index += 1
23             contador += 1
24
25             if cv2.waitKey(25) & 0xFF == ord('q'):
26                 break
27             else:
28                 break
29
30 cap.release()
31 # Cerrar ventanas
32 cv2.destroyAllWindows()

```

Como en muchos de los scripts de este proyecto, se utiliza la librería OpenCV. La primera línea del código se utiliza para importar el archivo de video creado

en el paso anterior. Este archivo debe de estar guardado en formato .mp4.

```
1 import cv2
2 import time
3 # Capturar video
4 cap = cv2.VideoCapture('SS_AutoNOMOS_Video03.mp4')
```

Una vez verificado que se haya abierto correctamente el archivo, se realiza una iteración en la cual se extrae un frame del video por cada 16 frames que se reproducen, es decir, se extae un aproximado de dos cuadros por segundo. Este valor se puede ajustar dependiendo de qué tanta similitud se esté dispuesto a tolerar para las imágenes que constituirán el dataset de la red neuronal a entrenar. Las imágenes extraídas se guardan en una carpeta con el mismo nombre del video, y se enumeran consecutivamente a partir del índice indicado al inicio del código.

```
1 while cap.isOpened():
2     # Leer archivo de video
3     ret, frame = cap.read()
4     if ret == True:
5         cv2.imshow('frame', frame)
6         if contador == 16:
7             cv2.imwrite('SS_ImagenesBruto_Video03\\{}.jpg'.format(index), frame)
8             contador = 0
9             index += 1
10            contador += 1
```

Finalmente se incluyen unos comandos de salida, se libera el objeto de captura de Opencv y se cierran las ventanas que se abrieron para mostrar los frames que fueron capturados.

El resultado obtenido debería de ser un carpeta con algunos cuantos cientos o miles de imágenes numeradas (dependiendo de la duración del video) que contienen una o varias instancias de los objetos que se desean detectar dentro del ambiente de la simulación. A continuación, se procederá a crear manualmente las etiquetas de estas imágenes que le indiquen al algoritmo de entrenamiento de la red neuronal la ubicación y el tipo de objeto identificado en cada una de ellas.

Para la creación de estas etiquetas, nos apoyaremos en el software **labelImg**, disponible en el repositorio <https://github.com/tzutalin/labelImg>. Los detalles de la instalación se pueden consultar en el archivo **readme**.

Es necesario crear un documento de texto indicando el número y el tipo de clases que se desea etiquetar. En nuestro caso, las clases configuradas fueron cuatro: **Traffic_light**, **Traffic_sign**, **Stop_sign** y **Zebra_path**.

Para ejecutar el script en Python de **labelImg** (después de haber instalado los requerimientos), se deberá de ejecutar la siguiente línea, en la cual se indique la ruta a la carpeta de imágenes, así como el archivo con los nombres de las clases.

```
1 $ python labelImg.py [RUTA DE IMAGENES] [ARCHIVO CLASES]
```

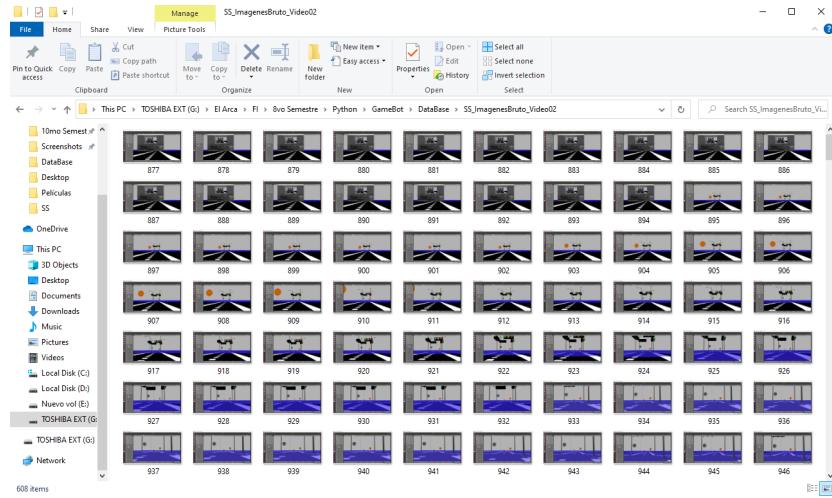


Figura 55: Carpeta con los frames capturados

Una vez dentro del ejecutable, se tendrá una interfaz como la mostrada en la imagen 56. En esta interfaz se deberá de dibujar sobre las imágenes rectángulos que ubiquen a los objetos que se desean identificar en el entrenamiento, y posteriormente de deberá de indicar la clase a la que pertenecen estos objetos.

Este procedimiento ha de repetirse para todas las imágenes que constituirán los datasets de entrenamiento y validación. La documentación de Yolov3 recomienda que se tengan al menos mil instancias etiquetadas para cada clase que se desea identificar, y que las imágenes estén divididas en un 80 % para el dataset de entrenamiento, y el resto en el dataset de validación.

Una vez concluido el etiquetado de las imágenes, éstas deberán de ser comprimidas en un archivo .zip, el cual se subirá a nuestro Drive para continuar con el proceso de entrenamiento de la red en la nube.

Antes de continuar con el entrenamiento en la nube, es necesario configurar algunos archivos de texto que incluirán información acerca de las clases que se desean entrenar, así como algunas configuraciones a la arquitectura de Yolov3 para adecuarla a la cantidad de clases y los parámetros de entrenamiento de nuestro dataset particular.

El primer archivo de texto que se ha de crear es simplemente uno que contiene los nombres de las clases a identificar, cada uno en una línea. Estos nombres no deben de contener espacios. El archivo deberá de ser llamado **obj.names**.

```

1 Traffic_light
2 Traffic_sign
3 Stop_sign
4 Zebra_path

```

El siguiente archivo que se debe de modificar es uno al que llamaremos **obj.data**, el cual indicará el número de clases que se desean entrenar, así como las rutas a los archivos que le indican al algoritmo de entrenamiento la ubicación

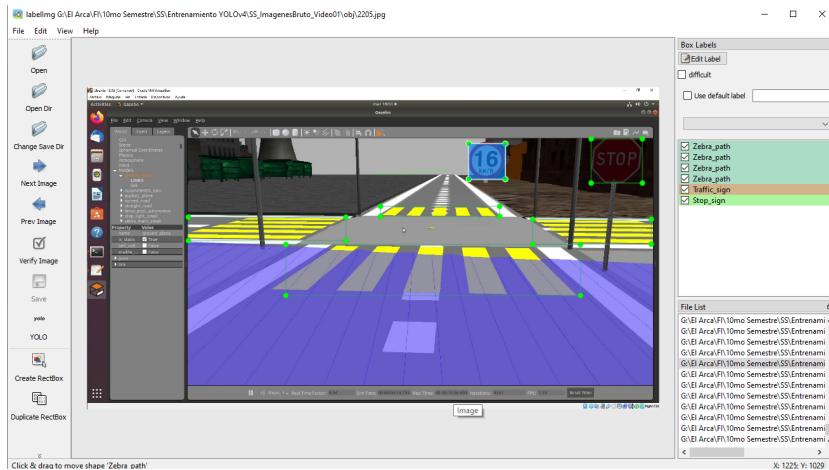


Figura 56: Interfaz de labelImg

```
2205 - Notepad
File Edit Format View Help
3 0.314844 0.440278 0.275521 0.073148
3 0.604948 0.568981 0.515104 0.158333
3 0.617188 0.386574 0.208333 0.030556
3 0.883333 0.449537 0.207292 0.076852
1 0.699219 0.231019 0.064062 0.110185
2 0.926042 0.229630 0.089583 0.137037
Ln 1, C 100% Unix (LF) UTF-8
```

Figura 57: Etiqueta de una imagen

de los datasets de entrenamiento y validación, así como la carpeta de destino en donde se guardarán los archivos de los pesos de la red neuronal generados.

```
1 classes = 4
2 train = data/train.txt
3 valid = data/test.txt
4 names = data/obj.names
5 backup = /mydrive/SS-Computer-Vision/yolov3/backup
```

El último archivo que debe de ser configurado es el archivo que describe la arquitectura de la red neuronal Yolov3. Se deberán configurar ciertos parámetros relativos al entrenamiento de la red para ajustarlos a la cantidad de clases con las que cuenta nuestro dataset particular. El archivo original puede ser descargado del repositorio <https://github.com/AlexeyAB/darknet> dentro de la carpeta **cfg**, bajo el nombre **yolov4.cfg**.

```
1 [ net ]
2 #Testing
3 #batch=1
4 #subdivisions=1
5 #Training
6 batch=64
7 subdivisions=16
8 width=416
9 height=416
10 channels=3
11 momentum=0.9
12 decay=0.0005
13 angle=0
14 saturation = 1.5
15 exposure = 1.5
16 hue=.1
```