

# Visión por computador. Proyecto final.

## Detección de señales de tráfico.

*José Carlos Martínez Velázquez*

*Benjamín Vega Herrera*

*10 de Febrero de 2017*

## Contents

<b>1. Descripción del problema.</b>	<b>2</b>
<b>2. Enfoque de la implementación y detalles sobre la eficiencia de la misma</b>	<b>2</b>
2.2. Fase de visión por computador . . . . .	2
2.2.1. Obtención de información relativa a la forma . . . . .	2
2.2.2. Obtención de información relativa al color . . . . .	4
2.2.3. Reducción de la imagen y preparación para la fase de machine learning . . . . .	7
2.3. Fase de machine learning . . . . .	8
2.3.1. Obtención de ejemplos y construcción del dataset. . . . .	8
2.3.2. Elección del modelo . . . . .	9
<b>3. Experimentos realizados</b>	<b>9</b>
<b>4. Valoración de los resultados</b>	<b>11</b>
<b>5. Conclusiones</b>	<b>12</b>
<b>6. Referencias</b>	<b>14</b>

## 1. Descripción del problema.

Nuestro problema consiste en la detección de señales de tráfico. Nuestro objetivo final es, dada una imagen, detectar si hay o no señales en la imagen y en caso afirmativo, señalárla en la misma de forma intuitiva. Durante los sucesivos apartados vamos a detallar una **primera aproximación** a la solución de este problema, uniendo los conocimientos adquiridos en las materias de visión por computador y aprendizaje automático.

## 2. Enfoque de la implementación y detalles sobre la eficiencia de la misma

El enfoque utilizado trata de aproximarse a una solución usando visión por computador y machine learning. Trataremos de construir un algoritmo de dos fases. En la primera fase obtendremos características relevantes sobre la imagen usando los conocimientos de visión por computador y en la segunda fase, utilizaremos estas características para construir un dataset, ajustar un modelo y entrenarlo.

### 2.2. Fase de visión por computador

En esta fase vamos a decidir qué características podrían ser relevantes de cara al ajuste del modelo de machine learning que usaremos para predecir. Pensando en cómo lo hacemos los humanos, decidimos tratar de extraer información sobre forma y color. En primera instancia, parecen suficientes para discriminar que en una imagen existe una señal de tráfico o no. En este apartado, trataremos de construir un dataset, que contenga la información anteriormente citada. Esta parte del problema, consiste en, dada una imagen cualquiera (**en principio**), construir un vector de características que se componga de la siguiente información:

característica	descripción	rango de valores
$x_1$	Existe forma conocida o no	(“forma”   “nada”)
$x_2$	Bin en el eje rojo	[0-16]
$x_3$	Bin en el eje verde	[0-16]
$x_4$	Bin en el eje azul	[0-16]
$y$ (salida)	Existe señal en la imagen o no	(1   -1)

Tal y como se puede apreciar, el vector de características de una imagen trata con variables categóricas (forma) y cuantitativas, que codifican las coordenadas de una imagen en el eje de color. Profundizaremos en qué significan, por qué esos valores y cómo obtener cada una de estas características en los apartados correspondientes.

#### 2.2.1. Obtención de información relativa a la forma

En primer lugar necesitamos saber qué formas queremos que se detecten. Para ello necesitamos imágenes bien definidas de estas formas. Nosotros detectaremos triángulos (peligro y ceda el paso), círculos (obligación y prohibición), octógono (stop), cuadrados (información y recomendación). Para almacenar esta información utilizaremos las siguientes imágenes:

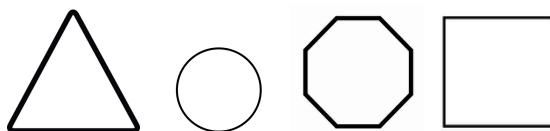


Figura 1. Formas geométricas en que se basan las señales.

Extraeremos su contorno mediante la función cuyo código se presenta a continuación, que usa el algoritmo `adaptiveThresold` de OpenCV, un algoritmo de detección de fronteras más adecuado a este problema que el algoritmo de Canny. Además se usa la función `findContours()` de OpenCV para buscar el contorno dentro de la imagen que devuelve `adaptiveThresold`:

```
"""
Lee los contornos de las figuras geometricas.
"""

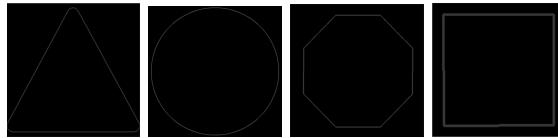
def leerContornos():
    CV_RETR_EXTERNAL=0
    CV_CHAIN_APPROX_TC89_L1=3
    imgs=[

        loadImage("./dataset/contornos/"+str(i)+".png", "GRAYSCALE")
        for i in xrange(4)]
    _contoursTest=[]

    for im in imgs:
        output = cv2.adaptiveThreshold(
            im,255,
            cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
            cv2.THRESH_BINARY_INV,51,2)
        im2, ctest, hierarchy=cv2.findContours(
            output,
            CV_RETR_EXTERNAL,
            CV_CHAIN_APPROX_TC89_L1)
        _contoursTest.append(ctest)

    return _contoursTest
```

Obtenemos los siguientes resultados, que almacenaremos durante la ejecución de memoria:



**Figura 2.** Contornos de las formas geométricas básicas.

De forma similar, sacaríamos el contorno de una imagen, mediante una función similar, de la que se extraen múltiples contornos:

```
"""
Lee los contornos de una imagen en concreto
"""

def leerContornosImagen(img):
    CV_RETR_EXTERNAL=0
    CV_CHAIN_APPROX_TC89_L1=3
    output = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY_INV,11,2)
    im2, ctest, hierarchy=cv2.findContours(output, CV_RETR_EXTERNAL,
        CV_CHAIN_APPROX_TC89_L1)

    return ctest
```

De ejecutar este código con diferentes ejemplos de imágenes, obtenemos los siguientes contornos:



**Figura 2.** Contornos de las imágenes.

Ahora, con una última función, tratamos de aproximar los contornos de las imágenes a los contornos de los polígonos básicos usando la función `matchShapes()` de OpenCV. De este modo, se obtiene una distancia a cada polígono básico y nos quedamos con la menor. Si la menor distancia es menor que un determinado umbral (fijado experimentalmente a 0.05), podemos determinar que existe una forma conocida (incluso podríamos determinar qué forma es si nuestro problema lo requiere), no siendo así en caso contrario. De este modo obtenemos la característica  $x_1$ , rellenándola con el valor “forma” o con el valor “nada”:

```
def buscarForma(contornos_figuras, imagen, umbral):
    contornos_imagen=leerContornosImagen(imagen)
    best_dis=999999.9
    best_fig=-1
    best_cont=None
    for cont_img in contornos_imagen:
        if(len(cont_img)>10):
            for i,cont in enumerate(contornos_figuras):
                dis=cv2.matchShapes(cont_img,cont[0], 1, 0.0)
                if(best_dis>dis):
                    best_dis=dis
                    best_fig=i
                    best_cont=cont_img
    if(best_dis>umbral):
        best_fig=-1
    return best_cont, best_fig, best_dis
```

### 2.2.2. Obtención de información relativa al color

La lógica de esta parte radica en que, en una imagen con señal se supone que la variabilidad de color de la parte que no sea señal sera alta, no siendo así en la parte con señal. Si aceptamos esto, deberíamos quedarnos con un valor de color representativo de la imagen, pero debido a la luz, una señal no tendrá un solo color, sino que el color de esta se moverá en un determinado rango.

Definiremos un sistema de representación gráfica para cada imagen en tres ejes de color (R,G,B). Dado que los colores se mueven en rangos, vamos a introducir en bins (clases de equivalencia) algunos valores que representarán lo mismo en este sistema de referencia. El máximo valor que puede tomar un píxel para cada valor R,G,B es 255. La descomposición factorial de 255 es:

$$255 = 3 \cdot 5 \cdot 17$$

Utilizaremos entonces el máximo factor como número de bins. De este modo, tendremos 17 clases de equivalencia o bins donde cada una representa a 15 valores de color por cada eje.

Resuelta la parte de representación de una imagen, vamos a tratar de explicar cómo asignaremos a una imagen sus coordenadas en el sistema de referencia. Dado que deseamos extraer el valor más representativo, definiremos un sistema de votaciones, por cada imagen, que constituirá un histograma. Dada una imagen,

separamos las tres capas de color. En una capa concreta, cada píxel vota en un bin de su eje. Calcularemos en qué bin caería dicho valor mediante la operación:

$$valor_{nuevo} = \left\lfloor \frac{\overbrace{17}^{num. bins} - 1}{255} \cdot valor_{anterior} \right\rfloor$$

Obtenidos los tres histogramas, nos quedamos con el bin más votado en cada capa. Los bins más votados constituirán las coordenadas de la imagen en el sistema de representación. Veámoslo más claro con un ejemplo:

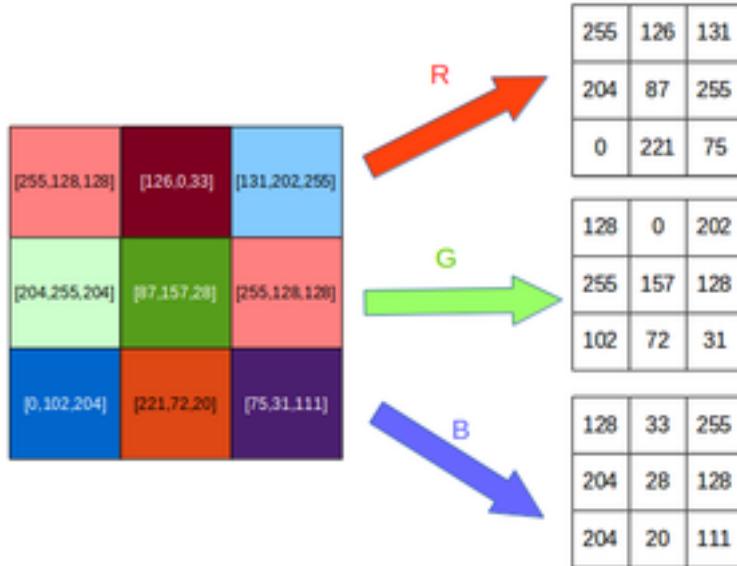


Figura 3. Paso 1: Separación de las capas.

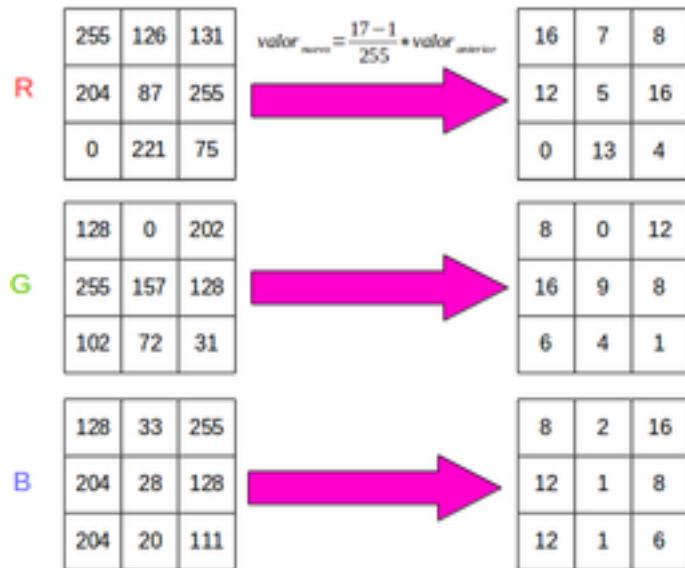
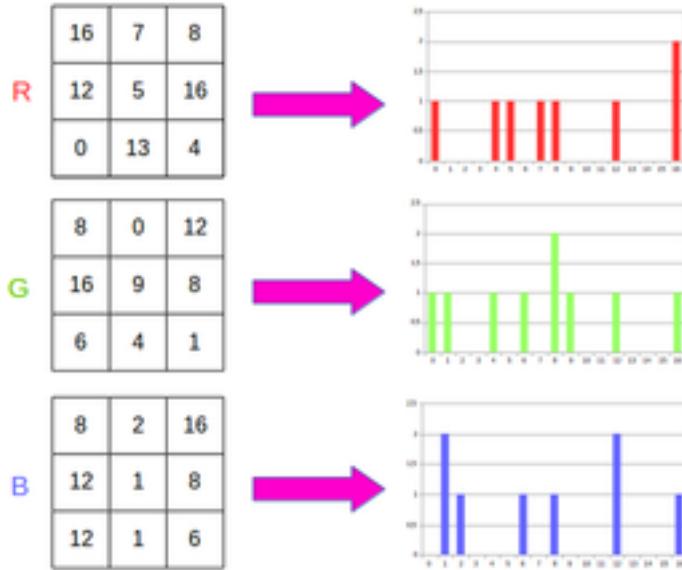


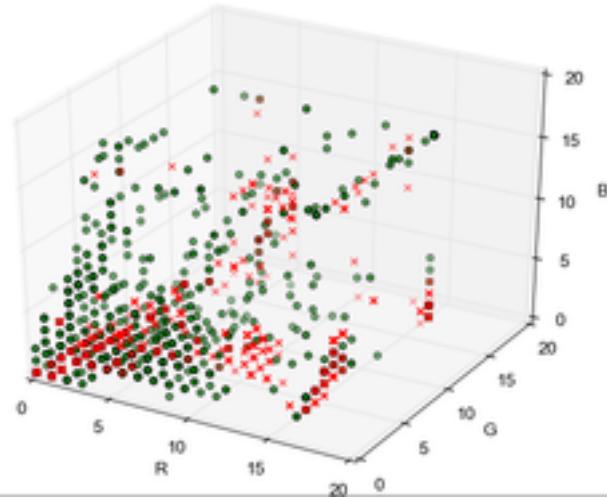
Figura 4. Paso 2: Cálculo del bin correspondiente.



**Figura 5.** Paso 3: Construcción de los histogramas.

Del último paso, podemos deducir que las coordenadas de la imagen en el sistema de representación son  $(16, 8, 1)$ . En realidad también podrían ser  $(16, 8, 12)$ , pero si existe ambigüedad es porque ambos valores (en el caso del histograma azul) hablan con la misma fuerza. Si nos vamos al caso de una imagen real, ya no tenemos  $3 \times 3$  píxeles y por lo tanto la probabilidad de que sucedan repeticiones se reduce drásticamente.

Para hacernos una idea de cómo están distribuidos los ejemplos obtenidos para el dataset de entrenamiento (completaremos la explicación del proceso para su obtención en los sucesivos apartados), hemos realizado su representación gráfica, donde los círculos representan ejemplos donde hay señal y las aspas ejemplos donde no las hay:



**Figura 6.** Representación gráfica del dataset en el eje de colores.

Esta gráfica 3D nos permite dilucidar que se tratará de una potencial candidata al positivo (hay señal) siempre que las coordenadas en este sistema de representación no superen el valor 10, a pesar de la dispersión. Fijémonos que los candidatos a negativo (no hay señal) por lo general tendrán valores G menores que 10, B menores que 5, no pudiendo sacar conclusiones en el eje R. De modo informal diremos que los ejemplos negativos están “delante”, por lo general, de los ejemplos positivos, esto significa que, los ejemplos negativos tendrán un valor de G más bajo.

### 2.2.3. Reducción de la imagen y preparación para la fase de machine learning

La mecánica de detección de señales se basará en un rastreo por ventana multiescalar. Coloquialmente, esto significa que pasaremos ventanas de distintos tamaños por la imagen y preguntaremos a un modelo de machine learning en base a los ejemplos aprendidos si esa sección de la imagen contiene o no señal. Esto es computacionalmente costoso por lo que convendría reducir la imagen en una cierta escala. Otra ventaja de reducir la imagen es que, si conseguimos llevar cualquier tamaño de imagen a un tamaño que nos convenga, sólo habrá que optimizar los tamaños de ventana para el tamaño de imagen que vamos a calcular. El objetivo que nos proponemos es, dada una imagen, convertirla a una imagen equivalente reducida, fijando ciertos parámetros a un tamaño que nos interese, respetando siempre su aspect ratio.

Supongamos que una cierta imagen, que queremos reducir, tiene un ancho  $x$  (número de columnas) y un alto  $y$  (número de filas), calculamos su aspect ratio como:

$$ar = \frac{y}{x}$$

La proporción  $ar$  nos indica cuántos píxeles de ancho hay por cada uno de alto, dicho de otro modo, cuántas columnas hay por cada fila. Supongamos ahora que queremos obtener una imagen equivalente reducida, con ancho  $x_{new}$  y alto  $y_{new}$ . Entonces, para respetar su aspect ratio, se debe cumplir que:

$$ar = \frac{y}{x} = \frac{y_{new}}{x_{new}}$$

Como se puede apreciar, sólo podemos fijar uno de los nuevos parámetros  $x_{new}$  ó  $y_{new}$ . Fijemos por ejemplo el nuevo ancho, entonces, podemos calcular el nuevo alto como sigue:

$$ar = \frac{y}{x} = \frac{y_{new}}{x_{new}} \Rightarrow y_{new} = \frac{y}{x} \cdot x_{new}$$

Conocidas la nueva altura y anchura, podemos calcular unos factores que nos indiquen cuántas filas y columnas hay que extraer de la imagen original para construir la nueva. Denominaremos a estos factores *slicers* y se calculan como sigue:

$$\begin{aligned} slicer_{alto} &= \left\lfloor \frac{y}{y_{new}} \right\rfloor \\ slicer_{ancho} &= \left\lfloor \frac{x}{x_{new}} \right\rfloor \end{aligned}$$

Un *slicer*, como se puede intuir, nos dice cuántas veces es más grande la imagen original con respecto al ancho o alto que la que estamos solicitando. Dicho de otro modo,  $slicer_{alto}$  nos dice que tenemos que quedarnos con una fila de cada  $\frac{y}{y_{new}}$  filas en la imagen original, análogamente  $slicer_{ancho}$  nos dice lo mismo con respecto a las columnas. Obviamente, dado que los cocientes pueden devolver numeros decimales, obtendremos una imagen reducida de la forma más cercana al parámetro fijado, de forma que los *slicers* sean números enteros

Conocido el nuevo tamaño y los *slicers* vamos a proceder a reducir la imagen. Este proceso se denomina **subsampling** y no puede considerarse satisfactorio sin emplear un filtro Gaussiano. Para no perder demasiado detalle en la imagen que vamos a tratar emplearemos un filtro Gaussiano de ventana  $3 \times 3$ . Esto es para tratar de no destrozar los contornos de las posibles señales que puedan existir en nuestra imagen original.

El proceso completo aplicado a una imagen de 1024 píxeles de alto por 1536 píxeles de ancho, fijando un  $x_{new} = 500$ , resulta en una nueva imagen de 342 píxeles de alto por 512 píxeles de ancho:



**Figura 7.** Comparación de una imagen de test original y reducida.

Como se puede apreciar, lo que se pierde en calidad es pequeño en comparación con la ganancia en eficiencia, pues el algoritmo tardará menos si se ha consegido reducir. La función para reducir una imagen es la siguiente:

```
def reduceImagen(imagen,nuevo_ancho,tamanio_ventana):
    if(nuevo_ancho<imagen.shape[1]):
        nuevo_alto=int((float(imagen.shape[0])/float(imagen.shape[1]))*nuevo_ancho)
        slicer_alto=int(float(imagen.shape[0])/nuevo_alto)
        slicer_ancho=int(float(imagen.shape[1])/nuevo_ancho)
        nueva_imagen=cv2.GaussianBlur(
            imagen[
                0:imagen.shape[0]:slicer_alto,
                0:imagen.shape[1]:slicer_ancho,
                ::,
                ],
            tamanio_ventana,
            sigmaX=0,
            sigmaY=0)
        return slicer_alto,slicer_ancho,nueva_imagen
    else:
        return 1,1,imagen
```

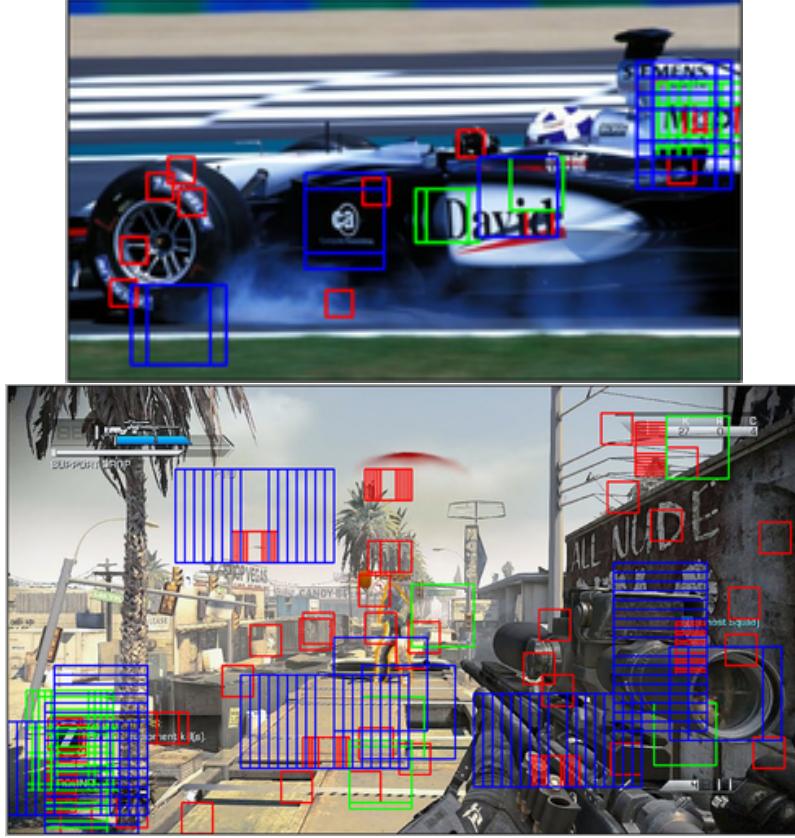
Obviamente, si la imagen solicitada es mayor a la original, no es necesario reducir. Los *slicers* serán igual a 1 y obtendremos la imagen original.

## 2.3. Fase de machine learning

### 2.3.1. Obtención de ejemplos y construcción del dataset.

En este proceso de machine learning se necesita entrenar un modelo con datos conocidos y etiquetados (positivos y negativos) y luego predecir sobre imágenes nuevas que no se conocen. El dataset usado, que puede obtenerse en [1] se compone de recortes de las imágenes completas que contienen las señales. Estos recortes se pueden utilizar como entrada al algoritmo descrito en los dos apartados anteriores y obtener salidas etiquetadas como positivas.

Tan pronto como llegamos a este punto, nos dimos cuenta que **el verdadero problema no es detectar cuándo había señales, sino determinar dónde no hay señal**. Determinar que hay señal es relativamente sencillo si se entrena con un número razonable de ejemplos positivos pero determinar que no hay señal es altamente complicado, pues lo que puede aparecer en una imagen que no tenga señal puede ser cualquier cosa. ¿Cómo decidimos solucionar esto? entrenando con imágenes que no tuvieran señales y etiquetándolas negativamente, aquí algunos ejemplos:



**Figura 8.** Entrenando con imágenes donde no hay señal (obteniendo ejemplos negativos).

Como se intuye, estamos ante un tipo de aprendizaje supervisado y la proporción de ejemplos negativos debería ser muchísimo mayor a los ejemplos positivos. En concreto, el dataset construído se compone de alrededor del 8% de ejemplos etiquetados positivamente (hay señal) y el 92% de ejemplos etiquetados negativamente (no hay señal).

### 2.3.2. Elección del modelo

Decidimos utilizar como modelo de aprendizaje un árbol de decisión. La elección del modelo usado se puede justificar de dos modos. En primer lugar, la justificación intuitiva es la decisión en base a la forma y color, esto es, en base a nuestro vector de características. Por ejemplo, si tenemos un círculo verde o un triángulo marrón lo más probable es que no se trate de una señal de tráfico. En segundo lugar, la justificación técnica se basa en la difícil separabilidad vista en la representación gráfica del apartado 2.2.2. La dimensión de Vapnik-Chervonenkis de un árbol de decisión es infinita, lo que significa que el número máximo de ejemplos que puede separar el modelo no está acotado superiormente.

## 3. Experimentos realizados

Se han entrenado con todas las imágenes del dataset [1] y se ha utilizado como test otro dataset [2]. En el dataset de test, que se compone de 80 imágenes se han apreciado 32 falsos positivos, lo que supone un total de 0.4 falsos negativos de media. Se han observado también 41 falsos negativos, lo que supone un total de 0.51 falsos positivos de media.

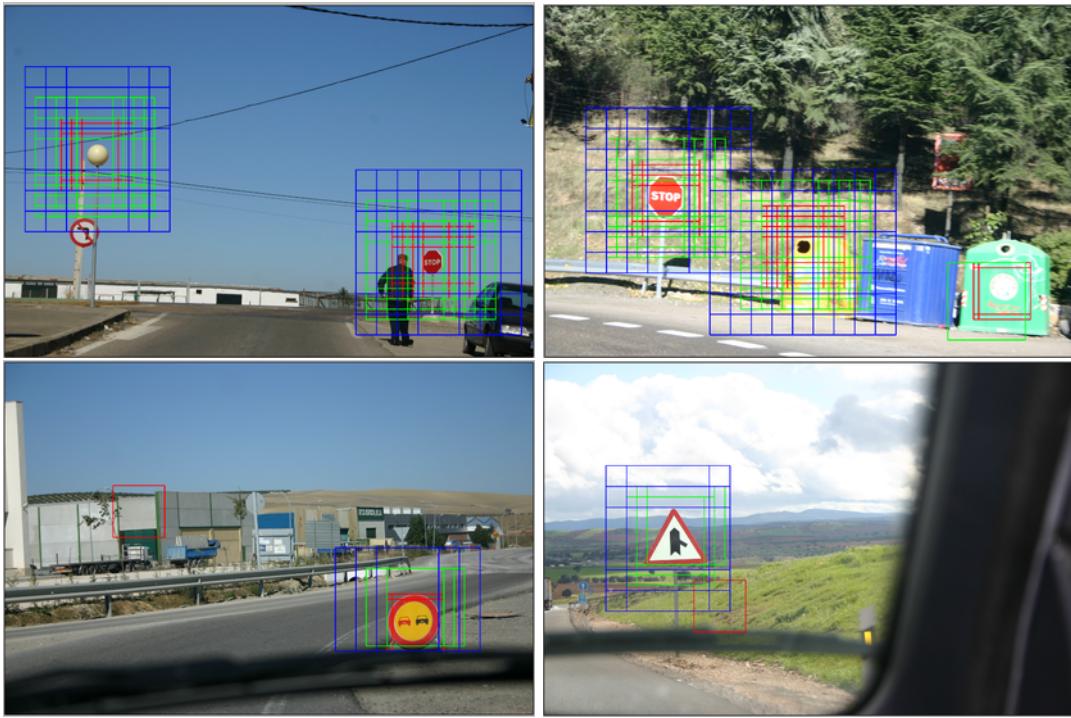
Algunos ejemplos a continuación:



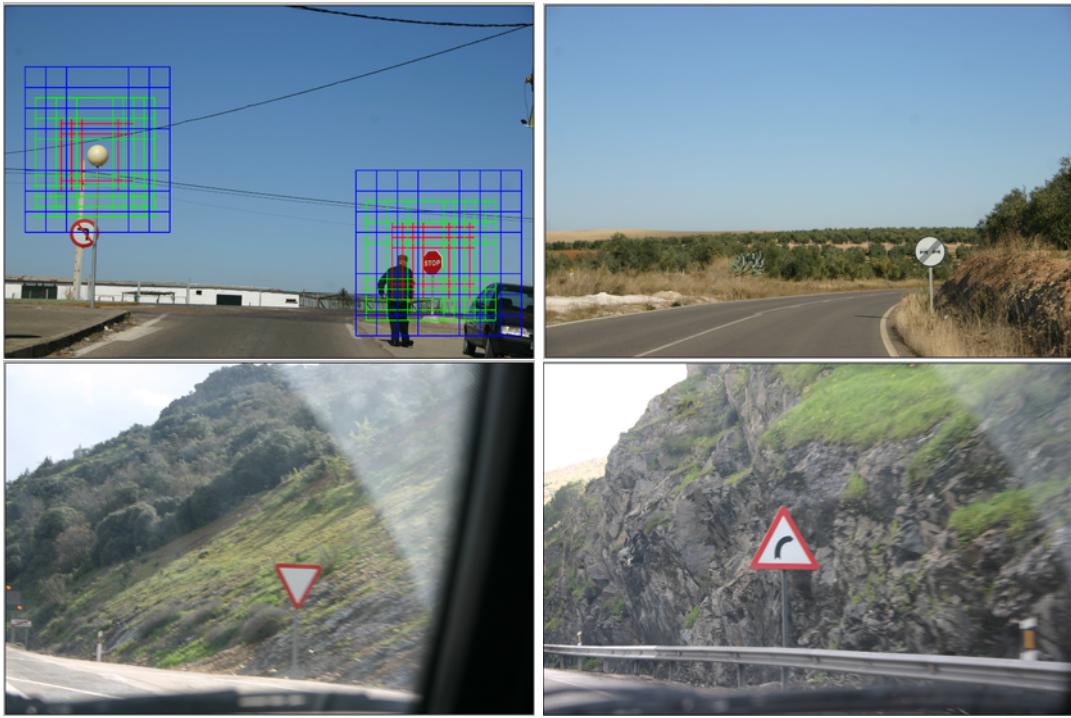
**Figura 9.** Verdaderos positivos.



**Figura 10.** Verdaderos negativos.



**Figura 11.** Falsos positivos.



**Figura 12.** Falsos negativos.

#### 4. Valoración de los resultados

En base a los experimentos realizados, debemos hacer algunos comentarios. Este sistema está pensado para ser instalado en un vehículo que analice las imágenes en tiempo real. Algunos de los falsos negativos son

causados por los tamaños de ventana. Este es un aspecto tan importante como delicado, pues aunque hemos realizado un gran esfuerzo por reducir las posibilidades de tamaño de las diferentes imágenes, llevándolas a un tamaño para optimizar las ventanas de las diferentes escalas, no resultó tan efectivo como pensamos, aunque sí es cierto que, si no hubiésemos llevado a cabo esta tarea, aún esto sería menos eficaz. Con todo ello, algunos falsos negativos se convierten en verdaderos positivos cuando el hipotético vehículo en el que el sistema va instalado avanza unos metros, como muestra el siguiente ejemplo:



**Figura 13.** El vehículo avanza, detectando una señal que no detectó en un instante anterior.

En líneas generales, consideramos que es una buena aproximación a la solución del problema aunque requeriría aún de mucho entrenamiento para alcanzar unos resultados óptimos. Los resultados obtenidos quizás no son los que nos hubieran gustado, pero está claro que el algoritmo implementado converge a la zona de señal, siempre que la haya. Nos parece importante recalcar la importancia de que el algoritmo tenga preferencia por indicar que no hay señal cuando la hay (generar falsos negativos) que indicar que sí hay cuando en realidad esto no es cierto. Es por ello que no es habitual que el algoritmo devuelva falsos negativos cuando no hay señal en la imagen, tal como se puede apreciar en las imágenes del apartado anterior.

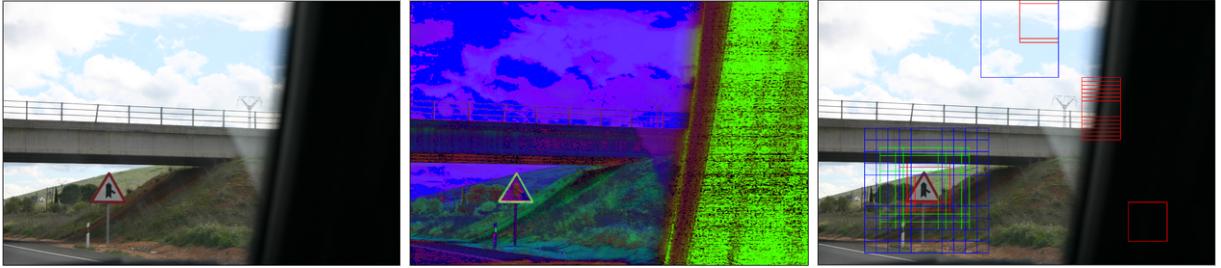
Hemos investigado sobre este problema y hemos encontrado resultados y errores de los sistemas para este cometido implementados por las grandes marcas [3]. Teniendo en cuenta que son sistemas maduros, sometidos (suponemos) a fuertes controles de calidad, validación y verificación, podemos estar satisfechos con los resultados obtenidos teniendo en cuenta el tiempo empleado.

## 5. Conclusiones

El problema al que nos enfrentamos es un problema complejo, que requiere una solución compleja. Aunque seguramente haya métodos más sofisticados y eficaces para resolver el problema, no es una mala aproximación. Creemos que existen varias maneras de mejorarlo, de entre las cuales, proponemos las siguientes:

- Usar un mejor detector de contornos: el detector utilizado comete fallos cuando hay algún objeto que corta, por así decirlo, el contorno de una señal. Por ello, si una señal tiene una planta (por ejemplo) delante, aunque no obstaculice demasiado, tendrá efectos negativos en el detector de contornos. Por motivos de tiempo y poca accesibilidad a la información, no hemos podido implementar el detector **ShapeContext** de Serge Belongie [4], aunque sin duda mejoraría la detección de contornos.
- Mejorar el entrenamiento en lo que no es señal: debido a la variabilidad de lo que no es señal (podría ser cualquier cosa que imaginemos) la decisión de que en una zona de la imagen no hay señal no es para nada trivial, lo que hace aumentar considerablemente la necesidad de añadir ejemplos negativos al dataset para mejorar el entrenamiento del sistema en este aspecto.
- Explorar en otros espacios de color: el espacio R,G,B quizás no es el más adecuado para resolver el problema, algo de lo que nos dimos cuenta a posteriori. Dado que hacemos que cada capa hable por separado, no tenemos en cuenta su interacción, de manera que si buscamos un rojo en un rango,

podemos obtener un ejemplo positivo sin tener en cuenta que los valores de este mismo ejemplo de las demás capas son altos, de manera que obtendríamos una respuesta alta en rojo cuando en realidad podríamos tener un color morado, que podría acabar marcando como señal una flor, por ejemplo. La solución estaría en añadir características que no solo hablaran del color sino de su matiz y saturación, esto es, trasladarnos al espacio de color HSV. Probamos este caso y mejora considerablemente los resultados:



**Figura 13.** Imagen analizada por el sistema explicado (izq, falso negativo). Imagen trasladada al espacio HSV (centro). Imagen analizada en el espacio HSV (der)

Ahora se detectan señales que antes no se detectaban gracias a que se obtienen datos relativos al matiz y saturación, lo que facilita la detección de los colores uniformes de la imagen. Aunque con este nuevo sistema aparecen falsos positivos, esto se solucionaría volviendo a entrenar con ejemplos cuyo vector de características se modificase añadiendo las características  $x_5$  y  $x_6$ :

característica	descripción	rango de valores
$x_1$	Existe forma conocida o no	(“forma”   “nada”)
$x_2$	Bin en el eje rojo	[0-16]
$x_3$	Bin en el eje verde	[0-16]
$x_4$	Bin en el eje azul	[0-16]
$x_5$	Saturación en el espacio HSV	
$x_6$	Matiz en el espacio HSV	
$y$ (salida)	Existe señal en la imagen o no	(1   -1)

Por falta de tiempo no hemos podido extraer de nuevo el dataset con estas nuevas características, pero estamos convencidos de que si todo se llevara a cabo, el sistema mejoraría considerablemente.

## 6. Referencias

1. *The German Traffic Sign Detection Benchmark.* Institut für neuroinformatik.  
<http://benchmark.ini.rub.de/?section=gtsdb&subsection=dataset>
2. *Traffic Signs UAH Dataset.*  
[agamenon.tsc.uah.es/Investigacion/gram/research\\_es.html](http://agamenon.tsc.uah.es/Investigacion/gram/research_es.html)
3. *Sistemas de reconocimiento de señales de tráfico en turismos.* Real Automóvil Club de Cataluña  
[http://imagenes.w3.racc.es/uploads/file/22207\\_Sistema\\_Reconocimiento\\_Seniales.pdf](http://imagenes.w3.racc.es/uploads/file/22207_Sistema_Reconocimiento_Seniales.pdf)
4. *Shape Context.* Wikipedia.  
[https://en.wikipedia.org/wiki/Shape\\_context](https://en.wikipedia.org/wiki/Shape_context)