UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

TFG del Grado en Ingeniería Informática

Reconocimiento de señales de tráfico mediante Raspberry Pi.

Presentado por Alberto Miguel Tobar
en Universidad de Burgos — 14 de septiembre
de 2017
Tutor: César Represa Pérez

Resumen

En este trabajo se ha desarrollado una herramienta de reconocimiento de señales de tráfico mediante el uso de una *Raspberry Pi*. El sistema puede reconocer distintas señales de tráfico agrupadas por categorías analizando un flujo de imágenes capturadas en vivo, una imagen estática previamente capturada o a partir de un vídeo grabado.

El trabajo está desarrollado en *Python* haciendo uso de la librería de visión artificial *OpenCV* y para mostrar las imágenes se usa el servidor de *Flask*.

La técnica usada para el reconocimiento es *Machine learning*, usa el algoritmo *Haar* para procesar las imágenes y reconocer las señales.

Se analiza el rendimiento de la *Raspberry Pi* usando diferentes resoluciones de imagen, así como diferente número de señales a reconocer.

Descriptores

Raspberry Pi, OpenCV, señales de tráfico, reconocimiento de imágenes, clasificador en cascada, Haar.

Abstract

In this project, I have worked to develop a tool for traffic signal recognition, with de help of a *Raspberry Pi*. The system is able to recognize different types of traffic signals grouped by categories using image streams captured live, from an image that was previously taken or from a recorded video.

The project is developed with Python, making use of a computer vision library, named OpenCV. To show the images a mycroframework called Flask is used.

The technique used to recognize traffic signals is *Machine learning*. The algorithm used is the Harr cascade classifier.

The performance of the *Raspberry Pi* is analysed using images with different resolutions, as well as a different number of signals to recognize.

Keywords

Raspberry Pi, OpenCV, traffic signals, detecting images, cascade classifier, Haar.

Índice general

Índice de figuras

Índice de tablas

Introducción

1.1. Motivación

Con el avance de la tecnología se ha conseguido que los coches sean autónomos y puedan manejarse solos ente el tráfico de las ciudades. Uno de los puntos importantes para la consecución de estos logros ha sido la visión artificial, y dentro de esta el reconocimiento de objetos, uno de ellos las señales de tráfico. Así que con esto en mente he intentado emular este proceso en un entorno controlado y son unos recursos ajustados.

Hay múltiples iniciativas por parte de empresas para construir un coche inteligente, algunas como *Tesla*, *Toyota* o *Waymo* ya están haciendo pruebas por carreteras convencionales. Otras entidades como *Automotive Grade Linux (AGL)* [9] tiene un proyecto para estandarizar el software usado en la industria del automóvil.

Es un sector importante no sólo a nivel mundial sino también local, en Castilla y León hay muchas empresas dedicadas a la automovilística, como *Renault* en Palencia y Valladolid o *Michelín* en Aranda. En Burgos capital hay otros ejemplos importantes como el grupo *Antolín* o el *ITCL*. Son un tipo de empresas que invierten recursos en investigación y desarrollo y que presentan oportunidades de trabajo y ofrecen una carrera dentro de ellas.

El motivo por el que me decanté por este tema fue porque un día conduciendo un *Renault Kadjar* me di cuenta de que reconocía las señales de velocidad y mostraba en el salpicadero la velocidad máxima permitida en la vía por la que iba circulando. Cada vez que pasaba una señal de ese tipo aparecía el aviso, incluso en señales temporales de obra de las de color de fondo amarillo. Me sorprendió que hubiese coches normales con ese tipo de tecnología y pensé que me gustaría algún día hacer algo relacionado. Este es el motivo por el que elegí hacer este tipo de trabajo.

Otra motivación importante de este trabajo ha sido utilizar herramientas de software libres.

1.2. Contenido de la memoria

El proyecto está estructurado de la siguiente forma:

En la primera sección hablo sobre los objetivos del proyecto y el por qué he escogido esta temática.

En los conceptos teóricos hablo sobre el aprendizaje automático *(Machine learning)* y describo los dos procesos que tiene la librería *OpenCV* para el entrenamiento en reconocimiento visual, también hay una breve reseña sobre otros métodos posibles. Por último hay una clasificación y breve descripción de las señales de trafico.

En la parte de técnicas y herramientas me centro en las herramientas que más he usado para realizar el trabajo y explico la metodología que he usado para planificar el desarrollo. Las herramientas más importantes han sido *Python*, *OpenCV* y por supuesto la *Raspberry* y su cámara, aunque del hardware hablo en el siguiente punto.

Los aspectos relevantes que he incluido han sido:

Fase inicial donde menciono la *Raspberry Pi* y su cámara, así como *Raspbian*, *Python*, *OpenCV* y por último *Flask*.

Fase de entrenamiento aquí describo como han sido los pasos a la hora de entrenar el reconocimiento de señales y las dificultades a las que me he enfrentado. Esta parte es la más extensa del trabajo ya que me parece la más importante.

Fase de desarrollo en esta parte describo como está estructurada la parte del código y algunos aspectos que me parecen relevantes. No entro en detalles técnicos porque esa parte la he dejado para el anexo del desarrollador.

Mediciones una vez acabado el desarrollo he realizado algunas mediciones con distintos ejemplos para comprobar como funciona la aplicación bajo diferentes escenarios.

En los trabajos relacionados he añadido 2 trabajos que explican otras formas de detectar señales, un ejemplo del libro que he seguido como guía para *OpenCV* y *Python* y dos blogs en los que entrenan *OpenCV* y que me han servido de guía.

Y para terminar la parte de la conclusiones he puesto lo que me gustaría mejorar y a una breve conclusión de lo que he sacado en claro de este trabajo.

1.3. Materiales entregados

El material entregado es el siguiente:

- Memoria en formato pdf.
- Anexo en formato pdf.
- Directorio con las fuentes del trabajo, organizado según aparece en los anexos.
- Directorio con ejemplos de señales entrenadas con los *scripts* de entrenamiento y los ficheros necesarios para su funcionamiento excepto las imágenes usadas como ejemplos negativos y de fondo. En los directorios de las señales entrenadas aparecen las imágenes usadas de ejemplo y los ficheros generados en el entrenamiento.
- Vídeo en el que se muestra el funcionamiento de la página web.

Objetivos del proyecto

El objetivo principal de este trabajo es reconocer las señales de tráfico que se pueden encontrar en las carreteras españolas mediante una cámara y un ordenador. Una vez reconocidas, procesar la información que contienen y ponerla a disposición del usuario mediante una página web. No sólo hay que poder procesar una escena en directo, sino que también hay que poder procesar imágenes y un vídeo grabado.

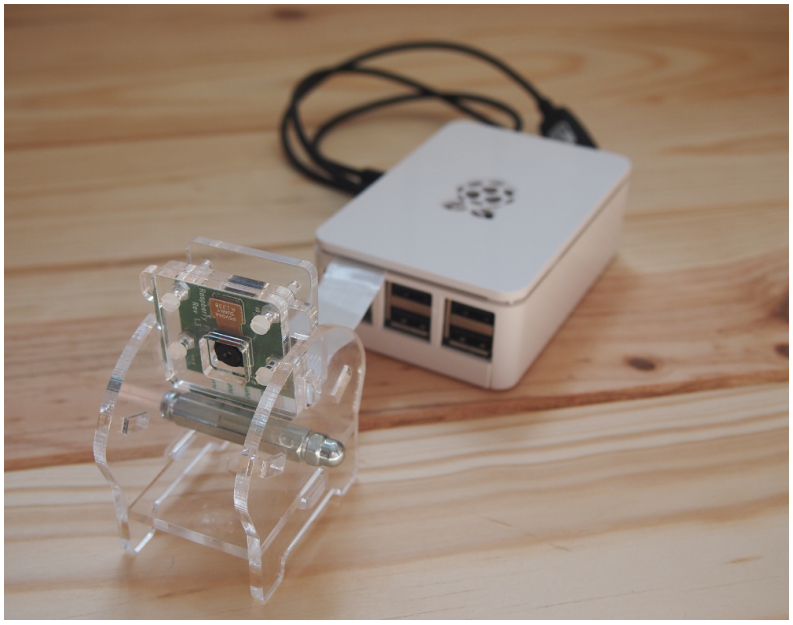

Figura 2.1: Raspberry Pi usada en el proyecto

El dispositivo que he elegido es la *Raspberry Pi* 2.1 porque es portátil, se mete en una carcasa con la cámara y una batería y se puede llevar a cualquier parte. Al ser hardware libre hay mucha información disponible en la red ya que

hay una amplia comunidad desarrollando proyectos entorno a ella. También tiene componentes que se pueden acoplar a ella como la *Picamera*, pantallas o escudos para el prototipado y manejo de elementos externos como motores, servos o *leds*.

El lenguaje de programación en el que voy a desarrollar el proyecto es *Python* porque es fácil de usar y de portar a otros dispositivos, al igual que la *Raspberry Pi* es libre y tiene gran popularidad.

La principal librería de la que voy a hacer uso es *OpenCV*, sirve entre otras cosas para el tratamiento de imágenes y la visión artificial, está disponible para la *Raspberry Pi* y se puede usar con *Python*. Tiene las funciones necesarias para entrenarla en el reconocimiento de imágenes mediante un clasificador en cascada que es la técnica que he decidido emplear.

Tanto para mostrar la información como para configurar los parámetros del programa voy a usar *Flask*, que en la página web del proyecto se describe como un *microframework* para *Python*.

Conceptos teóricos

En esta sección explico las características generales del aprendizaje automático y su clasificación. En otro apartado detallo los dos algoritmos disponibles en *OpenCV* para el reconocimiento de imágenes y cito también algunos otros métodos disponibles.

Por último presento la clasificación de las señales de tráfico en España.

3.1. Aprendizaje automático

El aprendizaje automático *(Machine learning)* es la capacidad que tienen los ordenadores de aprender en base a experiencias previas o a modelos ya estudiados.

Existen tres categorías principales en las que se clasifican estos algoritmos.

Aprendizaje supervisado: Al algoritmo se le suministran una serie de ejemplos con una entrada y su salida esperada, de esta forma tiene que determinar las reglas que se aplican para ir del estado inicial al final.

Aprendizaje no supervisado: Al algoritmo se le dan los ejemplos pero sin indicar su estado y tiene que ser capaz de encontrar la función que determina sus características.

Aprendizaje basado en refuerzo: En este caso el algoritmo interacciona con el medio para llegar a un estado final. El algoritmo recibe recompensas o castigos según su actuación.

3.2. Entrenamiento en cascada

Es un tipo de aprendizaje automático supervisado, en el que se entrena a la máquina con multitud de imágenes con el elemento a reconocer y con

algunos ejemplos negativos donde aparecen objetos distintos al requerido. En las imágenes positivas se indica las coordenadas donde se encuentra el objeto.

Para entrenar se usan imágenes en blanco y negro, ya que las características que se tienen en cuenta para identificar los objetos se basan en variaciones de luminosidad. Este método se llama detección de bordes, que es donde el brillo de la imagen cambia de modo brusco.

Figura 3.2: Aplicación de distintos algoritmos para detectar bordes. [19]

En la figura 3.2 se ve el resultado de aplicar a una imagen el algoritmo *Sobel* y *Random Forests* para detectar bordes.

Existen dos métodos disponibles para entrenar a la máquina:

- LBP [39] Patrones binarios locales *(LBP)*.
- Haar [38] Características de *Haar*.

En este trabajo voy a usar el *Haar* que es el que viene por defecto en *OpenCV*.

Local binary patterns (LBP)

Para entrenar de este modo se divide la imagen en cuadrados de varios píxeles y al del centro se le compara con ocho píxeles que estén dentro del cuadrado seleccionado y a la misma distancia. Al compararlos se crea un vector donde se guarda un '0' si el valor del centro es mayor que el del píxel comparado o un '1' si es menor, con lo que tenemos un valor de 8 bits. Al final del proceso se calcula el número en decimal que es el que va a caracterizar al píxel.

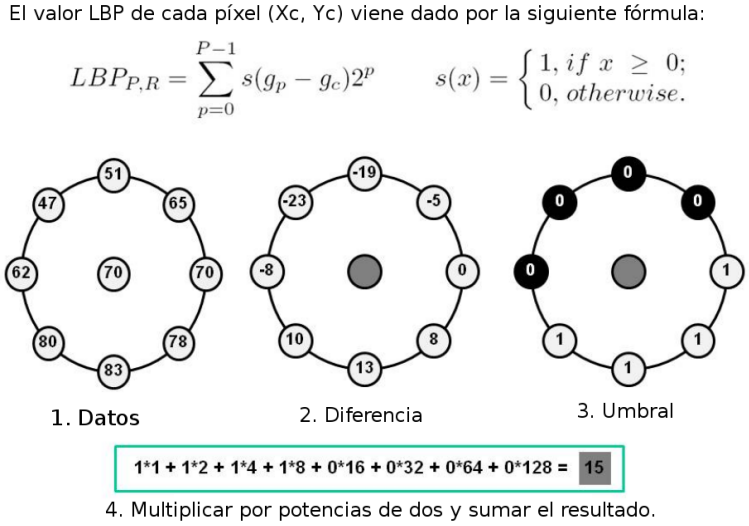

Figura 3.3: Zonas de cálculo de las variación de intensidad[31] *LBP*.

En la figura 3.3 se muestra el proceso para calcular el valor de un píxel.

El vector obtenido se procesa para obtener el número de píxeles que son menores y mayores. Una vez procesada toda la imagen y todos los vectores se obtiene un vector con las características del objeto.

Haar

En este método el clasificador se aplica a una ventana dentro de la imagen que se va moviendo dentro de ella, dentro de la cual se calcula la suma de la intensidad de una zona y luego se resta de la otra parte de la ventana. Esa diferencia es la que se usa para clasificar esa parte de la imagen. Este proceso se aplica por toda la imagen tomando como punto de partida la posición del objeto a reconocer.

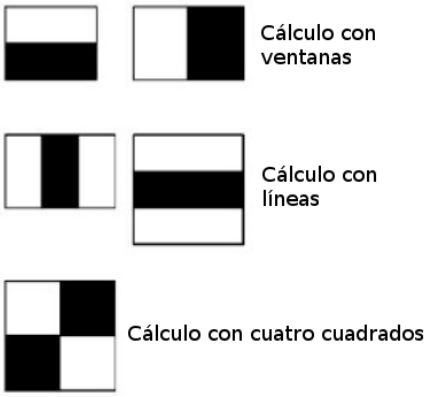

Figura 3.4: Zonas de cálculo de las variación de intensidad[20] *Haar*.

En la figura 3.4 se ve un ejemplo en el se resta la suma de las intensidades de la parte blanca de la suma de las intensidades de la parte negra.

Debido a que el cálculo obtenido no determina el objeto de forma inequívoca hay que repetirlo sucesivas veces para obtener una descripción precisa. En cada etapa se reconocen unas características básicas y se van refinando en las sucesivas.

Cálculo y aplicación del método clasificador

Para encontrar las características distintivas se calculan sobre el objeto a identificar las variaciones de intensidad con todos los posibles tamaños de ventanas y los seis modelos para el cálculo que aparecen en la figura 3.4, Para facilitar y agilizar este cálculo se aplica la fórmula de *integral images* [40] en la que se usan sólo los valores de las cuatro esquinas de la ventana actual.

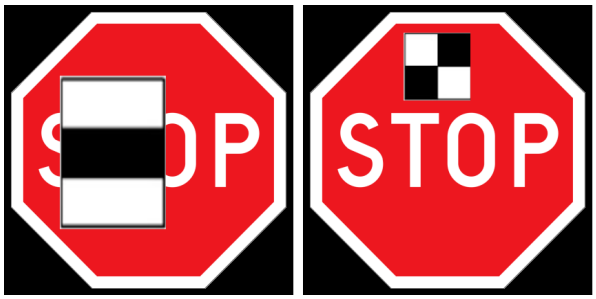

Figura 3.5: Ventana que se aplica para calcular las características.

No todos los cálculos hechos sobre el objeto sirven para caracterizar al objeto. En la figura 3.5 la parte de la izquierda tiene un cálculo aceptado

ya que hay variaciones brillo, mientras que en el la señal de la derecha sería descartada por no tener variaciones.

Todas las características distintivas sacadas de la imagen a detectar se aplican sobre las imágenes positivas y negativas del entrenamiento, para cada característica probada se calcula su ratio de errores y aciertos y se seleccionan las que menor ratio de fallos tienen.

A cada característica seleccionada se la llama clasificador débil, porque por sí sola no podría detectar el objeto entrenado. Pero procesar las imágenes con los clasificadores encontrados aún llevaría mucho tiempo, por esto se introdujo el método de cascada.

En cada etapa se usa un grupo de clasificadores para detectar el objeto, si la región procesada es descartada no se procesará en la siguiente. Las primeras etapas son las que menos número de clasificadores tienen.

3.3. Otros tipos de reconocimiento de imágenes

Dentro de la librería *OpenCV* hay más algoritmos de detección de imágenes pero son demasiado simples o no son lo suficientemente rápidos.

Harris corner detection: Detecta el cambio de intensidad de los bordes. Este algoritmo no contempla el escalado de la imagen.

Shi-Tomasi cordener detection: Detecta el cambio de intensidad en un número cerrado de bordes con mayor cambio. Al igual que el anterior tampoco funciona si el modelo cambia de tamaño.

Scale-Invarian Feature Transform (SIFT): Este algoritmo describe los los puntos más representativos de una imagen sin tener en cuenta ni su orientación ni su tamaño. Este algoritmo tiene el inconveniente de que es lento de aplicar ya que usa el filtro *Laplacian*.

Speed-Up Robust Features (SURF): Este algoritmo funciona de forma similar al *SIFT* pero para reconocer los puntos más representativos aplica otro tipo de filtro.

3.4. Señales de tráfico

En el código de circulación español las señales de tráfico están clasificadas en varios tipos:

- Señales verticales de circulación.
- Marcas viales.

- Semáforos.
- Señales y órdenes de agentes de tráfico.
- Señalización circunstancial.

En mi caso me voy a centrar en las señales verticales de circulación que se dividen en:

- Advertencia de peligro: *Las señales de advertencia de peligro tienen por objeto indicar a los usuarios de la vía la proximidad y la naturaleza de un peligro difícil de ser percibido a tiempo, con objeto de que se cumplan las normas de comportamiento que, en cada caso, sean procedentes.* [4]
- Reglamentación: *Las señales de reglamentación tienen por objeto indicar a los usuarios de la vía las obligaciones, limitaciones o prohibiciones especiales que deben observar.* [4]
- Indicación: *Las señales de indicación tienen por objeto facilitar al usuario de las vías ciertas indicaciones que pueden serle de utilidad.* [4]

Dentro de estas señales he escogido las más comunes ya que la Raspberry tiene limitada capacidad de cálculo y cuantas más señales tenga que buscar más lenta responde.

Técnicas y herramientas

En este apartado me centro en la metodología que he usado para de desarrollar el trabajo y en las herramientas utilizadas.

4.1. Técnicas

La metodología que he seguido ha sido un desarrollo iterativo con un análisis *top-down*, y las herramientas para la planificación del proyecto han sido papel y bolígrafo, al principio del proyecto marqué los distintos pasos a seguir y los he ido modificando según las distintas necesidades de cada momento.

Cada vez que tenía que empezar una nueva fase describía el problema al que me quería enfrentar y los objetivos a alcanzar. Cuando tenía los conceptos desarrollados lo dividía en partes más pequeñas con sus posibles funcionalidades y con el resultado ya podía empezar a desarrollar.

Las prioridades de cada apunte las coloreaba según su nivel de importancia, siendo el color rojo las más importantes, el color naranja las siguientes, sin ninguna marca las de prioridad normal y las de color verde las no prioritarias. Las incidencias apuntadas las iba resolviendo según su color, primero las más prioritarias.

El motivo de usar esta herramienta es que me resulta muy cómodo tomar notas estando en cualquier lado sin necesidad de depender más que de papel y lápiz.

4.2. Herramientas

Todas las herramientas para desarrollar el proyecto han sido libres y disponibles en *Linux*. En la *Raspberry Pi* la distribución de *Linux* que he usado ha sido *Raspbian* y *Fedora* en los otros ordenadores en los que he desarrollado el trabajo.

Las herramientas las he clasificado según el uso que las he dado.

Propias de programación

En este apartado cito las herramientas que he usado a la hora de programar, tanto los lenguajes como los programas.

Python [29] es un lenguaje de programación interpretado. Disponible para distintas plataformas, fácil de aprender y que tiene un gestor de paquetes propio con el que se pueden instalar distintas librerías desde su repositorio.

En mi caso me he decantado por *Python* porque lo hemos usado durante la carrera, por la comunidad de usuarios que tiene, que hace que encontrar información en internet sea fácil.

Instalar las distintas librerías que he usado es sencillo, el comando *pip3* que se encarga de compilar e instalar los paquetes disponibles.

OpenCV [18] es una librería para trabajos con visión artificial, se puede usar con *C++*, *Python* y *Java* y está disponible para *Linux* entre otros sistemas operativos.

He escogido esta opción porque se puede entrenar para reconocer imágenes con inteligencia artificial. En este caso es como vamos a trabajar. Para el trabajo he compilado el código fuente desde la *Raspberry Pi* con la opción activa de *ffmpeg*, para poder extraer frames del vídeo a reproducir.

Flask [8] es un microframework para *Python*, es ligero y versátil, está basado en *Werkzeug*, un servidor web y en *Jinja 2*, un motor de plantillas *HTML*.

He escogido este servidor porque es ligero y consume menos recursos que *Django* que es el otro que *framework* que valoré usar.

Geany [10] Es un editor de texto ligero que se usa como entorno de desarrollo para diferentes lenguajes de programación. Se puede completar su funcionalidad con complementos que le añaden funcionalidades.

He escogido este programa porque es fácil de usar y de aprender a manejar, se integra bien con *Python* y con el gestor de ventanas que uso, que es *Gnome*.

xml [5] es un lenguaje de marcado para diseñar documento. Está pensado para que lo entiendan tanto una persona como un ordenador. Se usan etiquetas de apertura y cierre para especificar cada elemento del documento.

Dentro del programa he manejado los archivos con la librería *lxml* de *Python*.

Lo he usado porque ya lo conocía de haberlo usado para algún pequeño proyecto en Python.

HTML5 [36] es un lenguaje de marcado tipo *xml* que se usa para diseñar las páginas web.

Lo he escogido porque al usar un servidor web tengo que usar el navegador para visualizar los datos y estos usan *html*.

Bootstrap [2] es un conjunto de herramientas para trabajar con *HTML*, *CSS* y *Javascript*. Sirve para modelar las páginas web y añadir componentes de forma rápida y fácil. Su instalación es sencilla, basta con copiar un archivo *CSS* y unas librerías *Javascript* que tienen dentro de la página web y hacer referencia a ellas dentro del código de la página que hay que maquetar.

Lo he escogido porque me ha parecido una forma sencilla de ordenar la página web sin necesidad de tener que crear yo archivos *CSS* y porque ya lo conocía de haberla usado en algún curso que he realizado.

Aunque ya está disponible la versión 4.0 del programa, he usado la 3.3.7 que era la última versión estable disponible cuando empecé a diseñar la página web.

4.3. Interacción con la Raspberry Pi

En este apartado describo las principales herramientas que he usado para interaccionar con la *Raspberry Pi*, ya que la he usado sin teclado ni ratón y sin pantalla.

bash [1] es un intérprete de comandos disponible para sistemas **nix*. Es compatible con otros intérpretes como *C shell* o *korn shell*, está basado en *sh* pero añadiendo más funcionalidades. Cumple las especificaciones *POSIX IEEE* para intérpretes de comandos.

Lo he usado porque es el que está disponible como opción por defecto en *Fedora* y en *Raspbian* y como en el caso de *Guake* y *Tmux* es el que estoy acostumbrado a manejar.

Vim [35] es un editor de texto altamente configurable y disponible para multitud de plataformas distintas.

Lo he escogido porque no usa apenas recursos, no es de fácil manejo ya que su curva de aprendizaje es alta, a pesar de esto lo he utilizado porque ya sabía desenvolverme en el entorno y lo tenía instalado en el ordenador. Lo he usado cuando quería editar ficheros a través de *ssh* o scripts u otros de configuración.

Guake [12] **con Tmux** [34] *Guake* es una terminal que se despliega en la parte superior de la pantalla al pulsar una combinación de teclas y *Tmux* es un multiplexador de terminal en el que dentro de una terminal puedes tener varias sesiones abiertas, que se pueden guardar para posterior mente acceder a ellas sin necesidad de tenerlas en primer plano.

He usado estos dos programas porque son con los que estoy habituado a trabajar. También porque al tener que manejar la *Raspberry Pi* por terminal tengo siempre la terminal disponible de forma rápida.

ssh [32] es un protocolo perteneciente a la *RFC*, da acceso a un dispositivo remoto por terminal de manera fiable y segura, ya que la comunicación entre el servidor y el cliente se hace de forma encriptada. Fue creado en 1995 para proteger las comunicaciones en la universidad de Finlandia de posibles ataques. Se usa para sustituir protocolos más anticuados como ftp, telnet, rlogin o rsh.

Lo he usado porque es la forma más rápida de conectarse a la *Raspberry Pi* a través de una terminal, es un estándar y una vez dentro del cliente es como estar dentro de la propia máquina. Consume pocos recursos y como la mayoría de los programas usados está disponible en los repositorios.

sshfs [33] es un programa para acceder a directorios remotos como si fuesen locales, es un equivalente al comando *mount* de *Linux* pero para directorios o volúmenes en máquinas externas.

Lo he elegido por su simplicidad de uso, ya que con un comando el directorio está disponible al usuario.

4.4. Utilidades varias

Aquí describo otros programas que me han servido para realizar otras tareas.

Firefox [7] es un navegador de internet creado por la fundación *Mozilla* y es 100 % libre.

Lo he escogido porque el trabajo no necesita ninguna característica especial que no esté presente en los navegadores modernos y es el que utilizo de manera habitual.

LaTeX [13] es un lenguaje de marcado para generar documentación técnica y científica. Es software libre y al igual que *vim* tiene una curva de aprendizaje alta.

He decido usar *LaTeX* porque ya había oído hablar de el y tenía ganas de usarlo.

LaTeXila [14] es un editor de *LaTeX* para el escritorio *Gnome*. Sus principales características son que soporta autompletado, tiene un panel con el que se puede navegar por las distintas secciones del documento. Hay una opción para cargar y crear plantillas y tiene una opción para crear proyectos.

El motivo para seleccionar este programa es que no conocía el uso de *LaTeX* y leyendo sobre diferentes programas este me pareció el adecuado para empezar a manejarlo.

GitLab [11] es un repositorio de control de versiones en el que he ido guardando el código. He usado la línea de comandos para crear, actualizar el repositorio y compartir el código en los tres ordenadores con los que he trabajado.

GitLab Tiene múltiples herramientas para gestionar gestionar el proceso de crear código, pero como ya he expuesto anteriormente lo he gestionado mediante papel y bolígrafo.

El motivo de elegir *GitLab* es que está disponible para instalarlo en cualquier ordenador, incluso tienen versión para la *Raspberry Pi* y su licencia de uso es más abierta y libre que *GitHub*.

La página del proyecto es `https://gitlab.com/amt0001/rpi_traffic_signs`.

ffmpeg [6] es un programa para el tratamiento de vídeo, con el que se puede codificar, decodificar multiplexar, demultiplexar, filtar o reproducir vídeos. Soporta gran cantidad de formatos de vídeo y se puede crear un servidor para la retransmisión de vídeo en directo.

En el trabajo lo he usado para cortar vídeo y para extraer frames de vídeos dentro del servidor y crear el vídeo de la presentación.

Aspectos relevantes del desarrollo del proyecto

En este capítulo cuento lo que me ha parecido más importante a la hora de desarrollar el trabajo, tanto en el entrenamiento, como desarrollo y en los resultados de las detecciones.

Lo he divido en cuatro secciones:

- Fase inicial en donde describo el hardware usado y los principales programas y librerías que componen el trabajo.
- Fase de entrenamiento donde describo el proceso que he seguido para preparar y entrenar el clasificador.
- Fase de desarrollo donde explico las clases que he creado y su utilidad, también hablo de alguna herramienta que me ha ayudado durante esta fase.
- Mediciones donde expongo los resultados de algunas mediciones que he realizado bajo distintos escenarios.

5.1. Fase inicial

Con este trabajo he pretendido desarrollar una aplicación que usada junto con hardware de bajo coste sea capaz de detectar señales de tráfico.

Hardware

El hardware que he escogido para realizar el trabajo ha sido:

- Raspberry Pi [22].

- Picamera [21].

El software principal ha sido:

- Python [29].
- OpenCV [18].

La parte que considero más importante del proyecto es la del entrenamiento, que está relacionada con *OpenCV* y las clase *DetectSignal* que analiza la imagen y la busca, hecha en *Python* y que se encuentra en el módulo *detect_signal.py*.

Raspberry Pi

La *Raspberry Pi* es un ordenador con todos los componentes integrados en una placa. Está desarrollada en Inglaterra y se desarrolló pensando en la enseñanza tanto a niños en los colegios como a personas en países en desarrollo.

La forma en la que la he empleado es sin conectarla a ningún monitor, sin teclado y sin ratón. El acceso al sistema operativo es mediante *ssh* y a los archivos a través de *sshfs*. Para acceder a la web desde donde se usa el programa la tengo conectada al ordenador principal a través del puerto *ethernet*, aunque se podría conectar a un *router* a través del *wifi*. He escogido esta opción porque así puedo acceder a la página web del programa sin tener una conexión a internet disponible.

Figura 5.6: Raspberry Pi V3.

En la figura 5.6 aparece la *Raspberry Pi* en su versión V3. Se pueden ver los distintos componentes de la placa.

Las principales características relevantes para este proyecto son:

- Núcleo 1.2GHz 64-bit quad-core ARMv8 CPU.
- Conexión 802.11n Wireless LAN.
- Puerto ethernet.
- Interface para la cámara (CSI).
- Ranura para tarjeta micro SD.

Picamera

La *Picamera* que uso es el modelo V1.3 es el primer modelo hardware que salió, el último ha sido V2.1, pero todas las funciones disponibles en el módulo de *Python* se pueden usar para ambos modelos.

Figura 5.7: Cámara de la Raspberry V1.3.

En la figura 5.7 aparece representada la cámara en la versión V1.3 que es la que he usado en el proyecto.

Las principales características de la cámara son:

- Sensor OmniVision OV5647.
- Resolución de la cámara 5 Megapixels.
- Máxima resolución 2592 x 1944 pixels.
- Distancia focal 3.60 mm +/- 0.01.
- Integración con Linux a través de V4L2.

Software

La elección del software a sido todo software libre.

Raspbian

Como sistema operativo a instalar en la *Raspberry Pi* he escogido Raspbian [23] porque es el que tiene soporte oficial, recibe antes las actualizaciones, tiene disponibles gran cantidad de paquetes en sus repositorios. Se puede instalar de forma fácil sin conocimientos avanzados de *Linux*.

Dispone de varias imágenes, en este caso he escogido la versión *Lite* porque no creo necesario un entorno de escritorio que consuma recursos innecesarios cuando solo se necesita una terminal para ejecutar la aplicación. La versión que hay instalada está basada en *Debian 8* ("jessie")[3], no es la última versión disponible, pero empecé con esta versión y he preferido no actualizar el sistema por si al hacerlo deja de funcionar. Aún así no debería haber problemas para correr la aplicación en la última versión ya que no hay un salto muy grande en cuanto al número usado de *Python 3* en *Debian Jessie* y *Debian Stretch*.

Python

El lenguaje de trabajo escogido ha sido *Python* que aunque es interpretado y ser mas lento que el compilado, me ha parecido que es un lenguaje ampliamente usado dentro de la comunidad del software libre, con bastante documentación disponible y con soporte tanto en la *Raspberry Pi* como en *OpenCV*.

Hay dos ramas diferenciadas de *Python* una es la 2 y la otra la 3, he escogido *Python 3* porque es la más moderna. La versión 3 es de 2008 y aunque no ha dejado de desarrollarse la 2, en algún momento dejará de tener soporte. La versión 2 tiene incompatibilidades con la 3 y se recomienda usar esta última. Al tomar esta decisión me he encontrado con que hay alguna librería que no está portada a la última versión de *Python* o que está mal portada como el paquete *PyBluez*.

OpenCV

Como librería para el tratamiento de imágenes y visión artificial he escogido *OpenCV* porque es la más conocida y la que más referencias tiene en foros y blogs. Está en constante desarrollo y mejora, tiene soporte para procesamiento multi-hilo, también soporta aceleración por hardware.

Tiene soporte para *Python* entre otros lenguajes de programación y se puede usar en *Linux*. En su página web hay documentación y tutoriales detallados en los que se pueden consultar ejemplos sencillos donde aprender como desarrollar e interaccionar con imágenes o hardware.

Para instalar *OpenCV* en la *Raspberry Pi* compilé el código fuente que se puede encontrar en la página web oficial del programa, en concreto la versión 3.2.0. Usé dos guías para su instalación una de la propia página web [17] y otra de un blog que se llama *pyimagesearch* [26] que he consultado de manera regular ya que tiene tutoriales interesantes sobre *OpenCV*.

Como el proceso de compilación tarda bastante tiempo usé *tmux* para poder ejecutar la compilación, cerrar la terminal, dejar trabajando a la *Raspberry Pi* y poder recuperar esa misma sesión una vez pasadas un par de horas.

Flask

En un principio quise transmitir las imágenes desde la *Raspberry Pi* hacia otro ordenador a través de *Bluetooth* y crear una aplicación en el cliente que se encargase de mostrar la información recibida, así se podría liberar de carga a la *Raspberry Pi*. Pero después de varias pruebas llegué a la conclusión de que las librerías que hay en *Python 3* están en una fase temprana de desarrollo o carecen de las características que necesitaba.

Una vez descartado el *Bluetooth* la mejor manera que encontré de comunicar cliente y servidor fue a través de un servidor web. De las posibles opciones que hay en *Python* las dos candidatas a elegir fueron *Django* y *Flask*. De estas dos elegí *Flask* porque es la que menos recursos consume, aunque su documentación es menos extensa que la de *Django* y tiene una comunidad de usuarios menor.

El acceso a la página web no tiene en cuenta el número de usuarios conectados, pero haciendo pruebas he visto que no es capaz de servir más de una página cada vez, ni se puede abrir dos veces el recurso de la cámara, además de ser un entorno para pruebas, no he creído necesario el restringir el acceso a un solo usuario.

5.2. Fase de entrenamiento

El primer paso del desarrollo fue instalar *OpenCV* y probar distintos parámetros del entrenamiento en cascada. Para esta tarea no he usado la *Raspberry Pi* ya que es un proceso que requiere un ordenador potente con un buen microprocesador y gran cantidad de memoria *RAM*. He usado un ordenador prestado con un *i7* de sexta generación con 16 *gigas* de *RAM* y un disco duro *SSD*, sistema operativo *Fedora 25* y *OpenCV* instalado desde los repositorios oficiales de la propia distribución. A pesar de contar con este ordenador el entrenamiento tardaba entre 2 y 4 horas por señal, haciendo pruebas con otro ordenador con un micro *i5* de quinta generación, 4 *gigas* de *RAM* y un disco duro mecánico la duración del entrenamiento superaba las 5 horas llegando en alguna prueba a las 6 horas.

Después de obtener el fichero *xml* con los descriptores de la imagen el siguiente paso fue comprobar que funciona. Después de descargar unas imágenes de Internet con las señales entrenadas programé un pequeño *script* en *Python* que cogía las imágenes de un directorio y mostraba en que imágenes encontraba la señal y la propia imagen rodeada por un cuadrado azul.

El código es el siguiente:

Script de pruebas

```
import cv2

stop_signal = cv2.CascadeClassifier('./xml/stop.xml')
for i in range(1, 10):
  nomImagen = 'stop' + str(i) + '.jpg'
  print("Imagen: " + nomImagen)
  img = cv2.imread(nomImagen)
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

  senial = stop_signal.detectMultiScale(gray, 1.3, 5)
  for (x,y,w,h) in senial:
    img = cv2.rectangle(img, (x, y), (x+w, y+h),
                        (255, 0, 0), 2)

    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    print('Encontrado en:', i)
    cv2.imshow('img', img)
    cv2.waitKey(0)

cv2.imshow('img', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Con los archivos *xml* ya creados ya pude empezar a crear el funcionamiento básico del programa donde captar imágenes en bucle desde la cámara y procesarlas. Para este paso tampoco usé la *Raspberry Pi* ya que implicaba tener que encender dos ordenadores, lo que hice fue trabajar con una *webcam* que tenía por casa y el portátil.

En esta parte teniendo en cuenta las limitaciones de la *Raspberry Pi* decidí tener dos tipos de señales a reconocer, las primarias, por ejemplo una señal de información, de peligro o la de *stop* y las secundarias, las que van dentro de las de información o de peligro. Las señales primarias serán padres de las secundarias, lo que implica que si encuentro una señal primaria padre tengo que buscar dentro de ella. Esto tiene dos mejoras principales, la primera es que reduzco en un primer bucle la cantidad de señales a reconocer y que cuando tengo que buscar la secundaria solo tengo que reconocer la imagen dentro de la primaria. Para reconocer la secundaria he elegido usar como reconocimiento también el entrenamiento *Haar*.

Reconocimiento de imágenes

Lo principal de este trabajo es el entrenamiento en el reconocimiento de imágenes. Lo primero que hice fue consultar el manual en la página de *OpenCV* [16], está muy bien explicado y es fácil de entender. Los pasos que seguí fueron los mismos que en el manual.

- Preparar los datos de entrenamiento.
  - Crear los ejemplos negativos.
  - Crear los ejemplos positivos.
- Entrenamiento en cascada.

Preparar los datos de entrenamiento

Esta parte del trabajo consta de varios pasos que hay que hacer de forma manual, la primera es obtener unas fotos para usar de modelo. Las imágenes las saqué de la colección personal de mi padre porque uno de sus *hobbies* es la fotografía y las imágenes de las señales las descargué de la página de dirección general de tráfico.

Para cada señal de tráfico tengo un directorio, con el nombre de la señal a reconocer, en el que en a su vez hay otro llamado *classifier* en el que se genera el archivo de reconocimiento de las señales. Otro donde están las imágenes a reconocer que se llama *positives*, en el caso de un *stop* solo habrá una mientras que en el caso de una señal de información hay varias, una por cada tipo de señal a reconocer, una para la fuente, otra para la gasolinera, otra para el restaurante y otra para el camping. En otro directorio una vez acabado el

proceso de creación del archivo *xml* guardo las imágenes positivas, el nombre de este no es relevante ya que se guardan ahí una vez acabado el entrenamiento.

Crear los ejemplos negativos

En esta fase hay que usar imágenes en las no aparecen señales de tráfico, así que usé las que ya tenía de mi padre. Estas hay que guardarlas en un directorio y hacer un listado en formato texto plano con ellas. Para el listado usé el comando ls redirigiendo la salida hacia un archivo de texto. El nombre del fichero donde generaba estos negativos es *bg.txt*. En la figura 5.8 se ve su contenido.

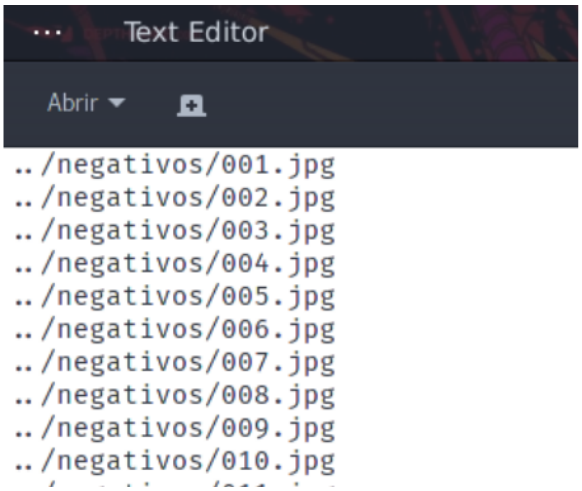

Figura 5.8: Contenido del archivo bg.txt

El comando para crear el listado sin necesidad de escribir las cosas a mano lo ejecuto desde el directorio de entrenamiento de una señal y es:

```
ls -d -1 ../negativos/* > bg.txt
```

Si se usa el *script* que creé para generar los archivos de entrenamiento basta con ejecutarlo una vez y copiar el archivo generado en el directorio principal de entrenamiento.

Crear los ejemplos positivos

Para las imágenes positivas usé la creación automática que viene con el propio *OpenCV*, el comando es *opencv_createsamples*. Este proceso tiene dos pasos, el primero es crear la imágenes positivas y un fichero como el de la figura 5.9 con extensión *.lst* en el que se guarda el nombre de la imagen, número de elementos positivos dentro de esta y sus coordenadas. En el segundo se genera un archivo propio de *OpenCV* con extensión *.vec* que luego se usa para

entrenar en el reconocimiento. Para esto me creé otro *script* que hiciese la cosas de manera automática.

Script para la generación de los archivos necesarios para el entrenamiento.

```
#!/bin/bash

if [ -d $1 ]; then
    cd $1
    i=0
    iniFile=$1
    if [ $# -eq 2 ]; then
        num_total=$2
    else
        num_total=500
    fi
    finFile=".lst"

    mkdir classifier
    cp -rf ../bg.txt ./

    ls -rtd positives/* > positives.txt

    while read -r senial
    do
        opencv_createsamples -img $senial \
            -num $num_total -bg bg.txt -info \
            $iniFile$i$finFile -bgthresh 0 \
            -maxxaangle 0.2 -maxyangle 0.2 \
            -maxzangle 0.2 -w 33 -h 52

        cat $iniFile$i$finFile >> $iniFile$finFile
        i=$((i+1))
    done < positives.txt

    echo "Se va a crear el archivo .vec"
    opencv_createsamples -bgthresh 0
        -maxxangle 0.1 -maxyangle 0.1 -maxzangle 0.1 \
        -info $iniFile$finFile -w 33 -h 52 \
        -vec $iniFile".vec" -num $num_total

else
    echo "No existe el directorio"
fi
```

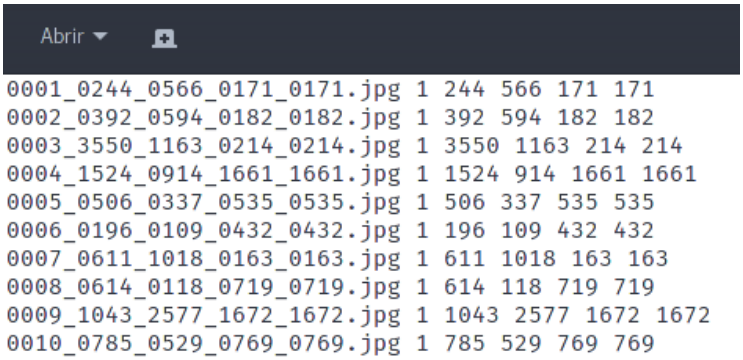

Figura 5.9: Fichero .lst generado en el primer paso de la creación.

A este *script* se le pasan dos parámetros uno es el nombre del directorio en el que se encuentran las señales a entrenar y el otro es el número de imágenes positivas que queremos entrenar. Los parámetros del comando *opencv_createsamples* los explico en el anexo del programador.

Hay que tener en cuenta un par de aspectos importantes a la hora de elegir la imagen que vamos a usar como muestra para generar los positivos. El primero es que influyen los colores que se usen en la imagen, en mi caso con las señales si pongo por ejemplo una señal de velocidad con los números negros al crear los positivos lo que ocurre es que lo hace transparente y se vería el color de fondo de la imagen, así que lo que hice fue modificarlos y ponerlos en un color oscuro.

Este comportamiento se podría cambiar al crear las imágenes con el parámetro *bgthresh* que es el que determina el brillo que se va a usar de fondo en las imágenes, en mi caso he preferido dejarlo como negro porque así es más fácil de controlar ya que el negro siempre es 0. Este parámetro se usa con el comando *opencv_createsamples* y está explicado en la parte de los anexos.

Figura 5.10: Cambio de colores en los números para que no haya zonas con transparencias de dentro del modelo a entrenar.

En la figura 5.10 se aprecia como he cambiado el el color del número de la señal para evitar el problema ya mencionado.

El otro aspecto es como hacer el fondo de la imagen. La primera vez que creé el archivo de reconocimiento lo hice con unas señales de *stop* con un poco de fondo. Y como se puede apreciar en la figura 5.11 de abajo en la izquierda aparecen unas transparencias que no aparecen en la derecha. No tener controlado esto podría suponer problemas a la hora de entrenar el reconocimiento, ya que si hay partes que desaparecen del objeto a reconocer el fichero generado al entrenar podría no ser válido.

Figura 5.11: Dos señales de muestra con distintos fondos.

Hay que tener en cuenta que esto influye a la hora de detectar las imágenes. Como se aprecia en la figura 5.12 las señales de información las reconoce mejor cuando el fondo sobre el que se muestran no es liso, sino que tiene muchos colores, líneas y cambios de brillo.

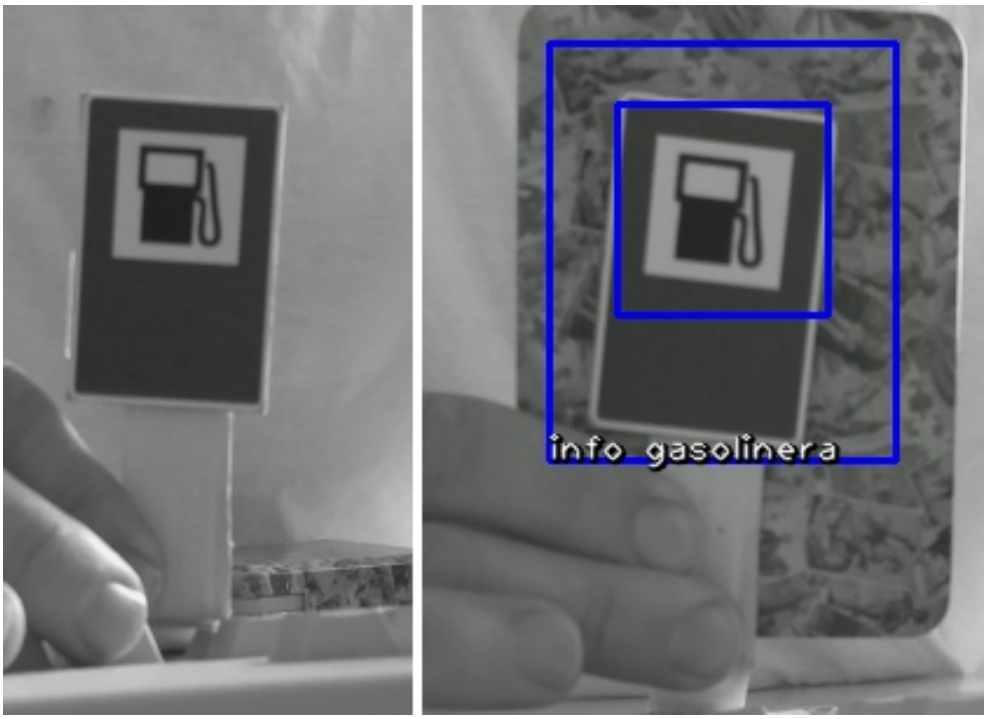

Figura 5.12: Comparación de la misma imagen con distintos fondos.

Así que a la hora de seleccionar las imágenes hay que ver qué fondo se pone, si el objeto a reconocer va a estar situado en sitios con mucha variación de colores y de formas, como un paisaje o dentro de una ciudad las imágenes para mezclar de los ejemplos positivos tienen que tener variaciones grandes de luminosidad, por el contrario si el objeto se encuentra en lugares con fondos lisos como puede ser el cielo o una pared el entrenamiento debería ser con imágenes de poca variación de luminosidad.

En la figura 5.13 hay distintos ejemplos que he usado para entrenar el reconocimiento, en la primera los márgenes están ajustados al máximo y no hay distancia entre la señal y el borde de la imagen, en la segunda los márgenes aumentan quedando espacio entre el final de la imagen y la señal. En la tercera el fondo elegido no es negro ni uniforme y queda bastante espacio entre la señal y los bordes. De las tres señales mostradas la que más positivos tiene a la hora de reconocer imágenes es la que está en el medio.

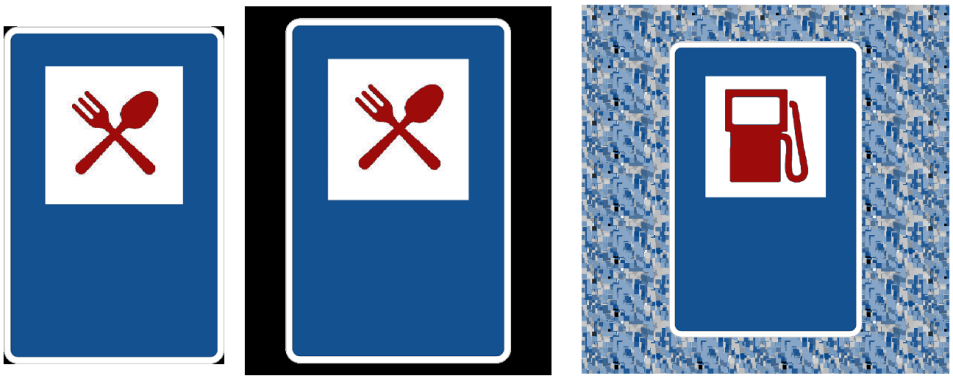

Figura 5.13: Distintas pruebas de señales para generar los positivos.

En la figura 5.14 se ve cómo quedan las imágenes de muestra con las tres señales. La primera imagen corresponde a las dos primeras señales de la figura 5.13, la diferencia que hay es que en el archivo *lst* que se genera con ellas, las coordenadas de la posición de las señales en el primer caso las coordenadas corresponden justo con la señal, mientras que en el segundo caso las coordenadas abarcan la señal y el margen correspondiente. La segunda imagen corresponde a la tercera imagen de la figura 5.13, en la muestra se puede ver como hay un margen con un fondo que corresponde a la señal.

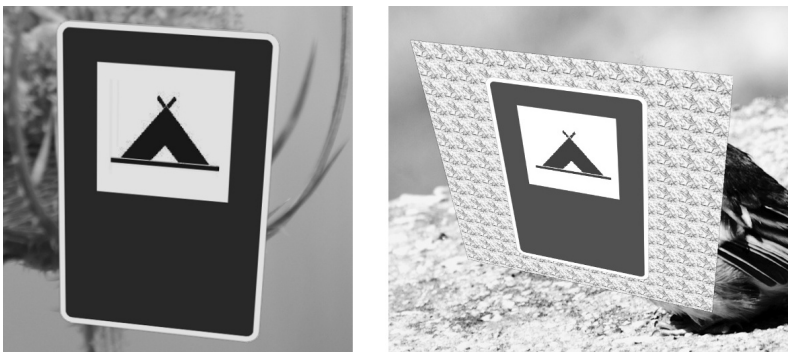

Figura 5.14: Imágenes generadas para con los positivos y los fondos para el entrenamiento en cascada.

También influye el tamaño de los objetos positivos y la distancia que haya hasta borde de la foto de muestra. En la foto de figura 5.15 se ve cómo para un mismo objeto y con el fondo de la imagen en negro, en la imagen de la izquierda no hay distancia desde que acaba el objeto y la imagen, mientras que en la de la derecha se ve cómo hay una margen negro tanto en vertical como horizontal, esto hace que la el objeto a reconocer ocupe más o menos dentro de la imagen en la que se ha encontrado.

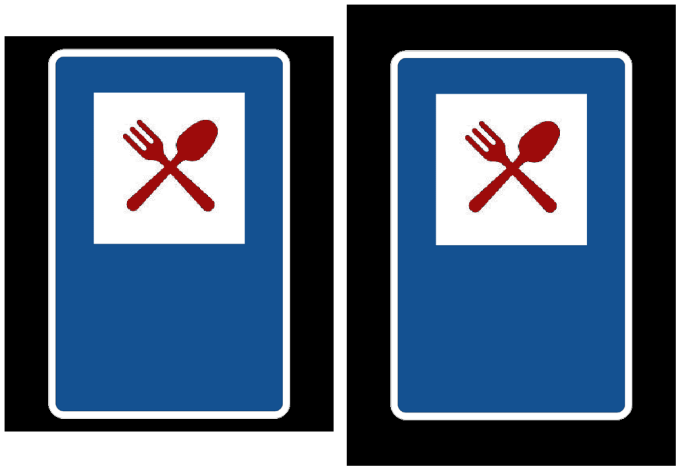

Figura 5.15: Distintas distancias entre el objeto a reconocer y el borde la imagen ambas con fondo negro

En la figura 5.16 podemos ver como cambia la distancia entre los bordes superiores en el cuadrado que encuadra la señal detectada, en la izquierda se aprecia como el borde se ajusta casi a la señal, mientras que en la derecha hay un margen mayor. Esto tiene un resultado inesperado ya que si el margen de la imagen de muestra es mayor que el margen que existe entre el borde y la señal en la imagen en la que queremos buscar no la va a conseguir detectar, así que hay que ajustar las imágenes de muestra al tamaño de la señal a detectar, pero siempre dejando un pequeño margen de fondo.

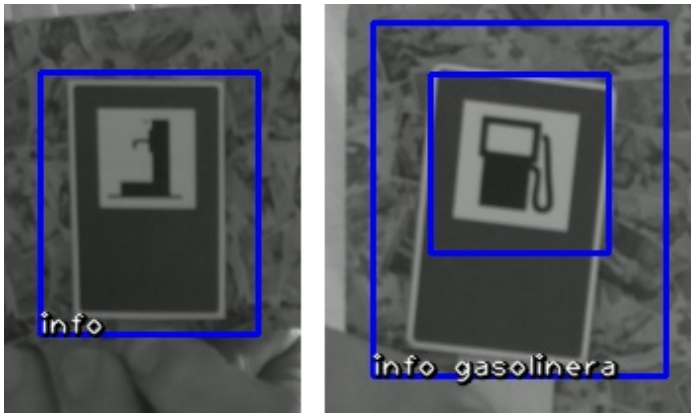

Figura 5.16: Diferencia de tamaño del recuadro marcado como zona de la señal.

Cuando se crean las imágenes de positivas y se crea el archivo *.vec* hay tres opciones importantes que determinan la rotación del objeto sobre los ejes

'X' *(-maxxangle)*, 'Y' *(-maxyangle)* y 'Z' *(-maxzangle)*. Si se quiere que el objeto se reconozca siempre en la misma posición sin ninguna distorsión estos parámetros se ponen a 0, si queremos que el elemento gire un ángulo máximo sobre un eje ponemos el valor del ángulo en radianes. En mi caso no quería que hubiese mucho ángulo en ninguno de los ejes, así que para todos elegí 0.1.

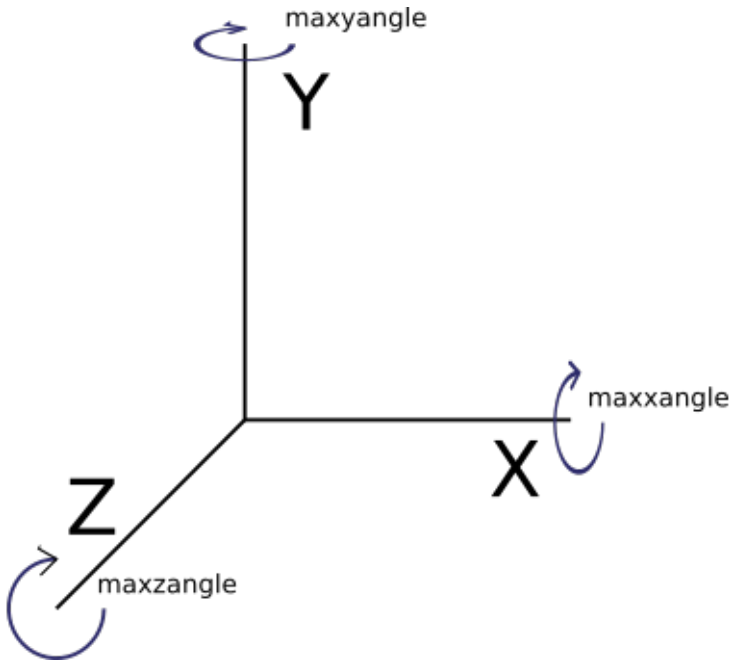

Figura 5.17: Efecto de los parámetros del ángulo sobre cada uno de los ejes.

En la figura 5.17 se representa el cambio que se generaría en cada eje con los parámetros de ángulo.

Entrenamiento

Esta es la parte más aburrida del trabajo, dejar el ordenador funcionando solo ya que no requiere ninguna intervención por mi parte. El comando usado es el siguiente:

```
opencv_traincascade -data classifier -vec $senial".vec" \
  -bg bg.txt -precalcValBufSize 5000 -precalcIdxBufSize \
  5000 -numPos 110 -numNeg 300 -nstages 20 \
  -minhitrate 0.999 -maxfalsealarm 0.5 -w 50 -h 50
```

Este proceso tarda entre 2 a 4 horas dependiendo de las características de la señal. Todas las señales las he entrenado con 20 fases, en el comando es la opción *-nstages*. En cada fase se actúa igual, primero se usan las imágenes positivas, se buscan las características de las señales se genera un archivo *xml* donde se describen las características en este caso usando el algoritmo de

Haar, luego se aplica ese algoritmo a las positivas y se anota el porcentaje de aciertos, por último se hace lo mismo con las negativas y también se anota el porcentaje de aciertos. Si no es la fase número 20 o no se han alcanzado los valore *minhitrate* y *maxfalsealarm* se continúa con una nueva fase en la que se usan los descriptores de la anterior y se busca refinarlos.

En la figura 5.18 se muestra el resultado de la fase 6 de entrenamiento, *HR* es el porcentaje de acierto en este caso es del 0.9966, *FA* es el porcentaje de falsas alarmas que es 0.38.

```
NEG current samples: 294
NEG current samples: 295
NEG current samples: 296
NEG current samples: 297
NEG current samples: 298
NEG current samples: 299
NEG current samples: 300
NEG count : acceptanceRatio    300 : 0.0114543
Precalculation time: 14
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3|        1|        1|
+----+---------+---------+
|   4| 0.996667|     0.38|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 18 minutes 17 seconds.

===== TRAINING 7-stage =====
<BEGIN
```

Figura 5.18: Salida al terminar una fase de entrenamiento con sus porcentajes de aciertos.

El número de ejemplos positivos y negativos que he usado ha sido entre 100 y 300 positivos y 200 y 300 negativos, según las consultas que hice tanto en el foro de *OpenCV* como en *blogs* y otras fuentes creí que era suficiente. Sobre todo me resultó útil esta explicación [15] que se hace en el foro de *OpenCV*.

En este paso tuve algún problema, si había generado 300 imágenes positivas y como parámetro *-numPos* le pasaba 300 había veces que me daba error, le tenía que poner un número de positivos menor.

5.3. Fase de desarrollo

En esta apartado describo los pasos que he dado para desarrollar el código de la aplicación.

Clases

El programa lo dividí en clases, la primera fue una clase que se encargase de las señales y sus características, tales como su nombre, el nombre del fichero *xml* con sus patrones de reconocimiento, el padre, si es que tiene, entre otras. En las señales no se guarda la ruta al fichero *xml* sino que se guarda el nombre, no creo que la señal deba saber donde se encuentra su descriptor.

Otra se encarga de la configuración del programa, leer los parámetros de las señales, los de configuración de la cámara y los del programa de tratamiento de las imágenes, así como también de guardar los cambios que se quieran hacer en el funcionamiento del mismo.

Con otra clase se tratan las imágenes para buscar las señales de tráfico. En esta clase es donde se cargan los *xml* con la información sobre el reconocimiento y se devuelve la imagen para mostrar.

Otra clase que se encarga de recoger las imágenes desde la cámara o desde un archivo ya grabado. En este paso he intentado usar un *pool* de hilos para mejorar la velocidad de la imagen en el procesamiento en tiempo real con la *Picamera*, pero he tenido problemas al manejar el flujo de datos que devuelve la cámara. He probado a hacer una mezcla de los ejemplos que dan en estos dos enlaces en la documentación de la cámara [25] [24] pero no lo he conseguido. Aún así al hacer pruebas para comprobar el rendimiento se ve como hace uso de los 4 núcleos al detectar imágenes. Esto pasa porque al compilar *OpenCV* con la opción WITH_TBB o WITH_OPENMP hace uso de ellos y así lo hice.

En la figura 5.19 se muestra la salida del comando *top*, en la parte de arriba se ve como está corriendo el *Flask*, está detectando una señal que le estoy mostrando a la cámara. En la parte de abajo se ve como el proceso 26603 está ejecutando *python3* hace un uso del 48.1 % de la *cpu*, y como la suma de las 4 cpus suma un 45 % también se aprecia como hay solo una tarea en activo de las 124 que hay en total.

```
192.168.1.133 - - [31/Aug/2017 20:24:52] "POST / HTTP/1.1" 200 -
192.168.1.133 - - [31/Aug/2017 20:24:52] "GET /static/css/reset.css HTTP/1.1" 304 -
192.168.1.133 - - [31/Aug/2017 20:24:52] "GET /static/css/bootstrap.min.css HTTP/1.1" 304 -
192.168.1.133 - - [31/Aug/2017 20:24:53] "GET /cam_feed HTTP/1.1" 200 -
```

```
top - 20:25:29 up  1:06,  3 users,  load average: 0.16, 0.22, 0.24
Tasks: 124 total,   1 running, 123 sleeping,   0 stopped,   0 zombie
%Cpu0  : 10.0 us,  1.0 sy,  0.0 ni, 89.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  8.7 us,  1.7 sy,  0.0 ni, 89.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  : 12.5 us,  2.0 sy,  0.0 ni, 85.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 14.0 us,  0.0 sy,  0.0 ni, 86.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:    882772 total,   209704 used,   673068 free,    18456 buffers
KiB Swap:   102396 total,        0 used,   102396 free.   103856 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26603 pi        20   0  269816  53176  31060 S  48.1  6.0   0:23.94 python3
  217 root      20   0       0      0      0 S   0.3  0.0   0:00.84 brcmf_wdog/mmc1
24300 pi        20   0    5112   2528   2164 R   0.3  0.3   0:02.50 top
26745 root      20   0       0      0      0 S   0.3  0.0   0:00.10 kworker/u8:3
    1 root      20   0   22808   3888   2680 S   0.0  0.4   0:02.75 systemd
```

Figura 5.19: Mientras está reconociendo imágenes hace uso de los 4 núcleos del micro.

Para mostrar la información de las señales encontradas tenía la opción de ir mostrando un texto dentro de la página web, pero esta opción me pareció que no era viable, ya que quería guardar la información obtenida de alguna manera, así que decidí crear un archivo *xml* donde guardar la información encontrada, con la fecha y hora a las que se empieza y acaba de procesar las imágenes y un listado con el nombre de las señales encontradas, el *frame* en el que se han encontrado y la posición dentro de la imagen.

```
<rpitrafficlog>
  <startdate>2017/07/30 - 06:03:49</startdate>
  <stopdate>2017/07/30 - 06:04:54</stopdate>
  <frames>
    <sample>
      <signal>info Camping</signal>
      <frame>13</frame>
      <position>282,55</position>
    </sample>
    <sample>
      <signal>info restaurante</signal>
      <frame>19</frame>
      <position>307,58</position>
    </sample>
    <sample>
      <signal>info Camping</signal>
      <frame>20</frame>
      <position>313,31</position>
    </sample>
    <sample>
      <signal>info restaurante</signal>
      <frame>21</frame>
      <position>312,19</position>
    </sample>
```

Figura 5.20: Datos generados y guardados en el log.

El contenido de este fichero se puede ver en la figura 5.20

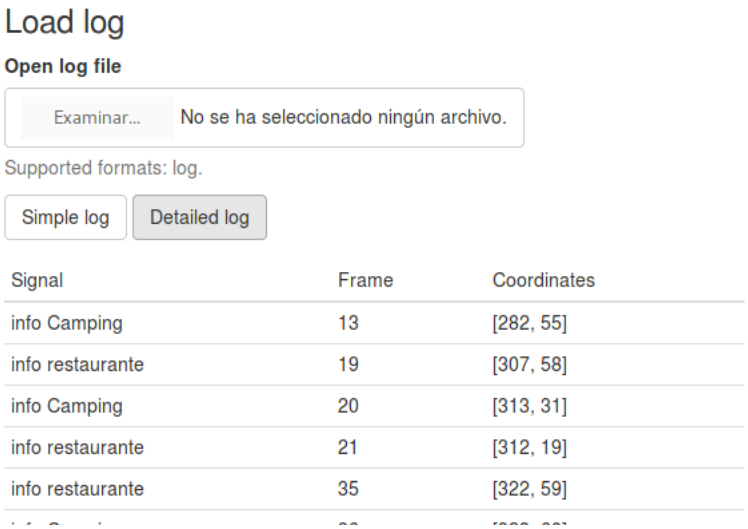

Figura 5.21: Visualización de los datos del log en la aplicación.

En la figura 5.21 se muestra un ejemplo de como se representa esa información en la página web.

Otra forma de ver qué imagen se ha encontrado que he implementado es, como se ve en la figura 5.22, mostrar un texto debajo de la señal dentro de la propia imagen que ponga el nombre de la señal encontrada.

Figura 5.22: Señal de stop detectada.

Por último el módulo con el que se crea la página web que es el que se encarga de hacer de intermediario con el usuario. Es el que está basado en *Flask*.

Herramientas

Otra decisión fue en que formato guardar la configuración del programa, en mi caso me decanté por *xml*. Los motivos fueron que es el formato en el que se guardan los datos del entrenamiento en cascada, tiene un formato bien estructurado, ya había trabajado antes con el y me resultaba conocido. *Python* tiene distintas librerías de manejo de *xml* de manera que es fácil obtener la información y guardarla.

Para cumplir con la guía de estilos de programación de *Python* uso *Pep8* [28], *pyflakes* y *pylint* [27] lo uso para encontrar posibles problemas en el código y para comprobar la cobertura de los test uso coverage3. Para no tener que ejecutar cada uno por separado he creado un *script* que ejecuto con una combinación de teclas en *Geany* o de forma manual, lo encontré en *GitHub* y lo adapté según mis necesidades:

```
#!/bin/sh
```

```
echo "=======  pep8  ======="
pep8 $1
echo "=======  pyflakes  ======="
pyflakes $1
echo "=======  pylint  ======="
pylint --msg-template="{path}:{line}:[{msg_id}({symbol}),
{obj}]{msg}" --reports=n $1
echo "=======  coverage  ======="
coverage3 run $1
coverage3 html --include=../*,./*
```

Así con esta salida veo los problemas que hay que arreglar en el código o si las pruebas ejecutan el 100 % de la cobertura.

Para la complejidad del código uso *radon* [30] que es un paquete que analiza varios parámetros de complejidad del código. Esto está explicado más en profundidad en el anexo del manual del programador.

Para generar las imágenes con la estructura de las clases he usado *pyreverse*, que es un comando que se instalar con *pylint*.

5.4. Mediciones

Para las mediciones lo primero que hay que tener en cuenta es el tamaño de las imágenes y su relación entre ellas. El programa usa tres resoluciones distintas. En la tabla 5.1 se puede ver el número de píxeles que tiene que cada imagen procesada, y la relación de tamaño que hay entre cada resolución. Sólo cuento una dimensión para el tamaño, ya que a pesar de que la imagen se puede tomar en color con formato RGB antes de procesarla para detectar las señales se transforma a blanco y negro.

Resolución	Píxeles en blanco y negro	1280x720	640x480	320x240
1280x720	921600	1	3	12
640x480	307200	1/3	1	4
320x240	76800	1/12	1/4	1

Tabla 5.1: Número de píxeles de cada tipo de imagen en blanco y negro y la relación entre cada resolución.

He realizado una serie de mediciones con distintos parámetros, cambiando el número de señales a detectar, la resolución de la cámara y poniéndola en blanco y negro. Los resultados obtenidos se pueden consultar en la tabla 5.2.

Los valores de los tiempos están tomados en el *frame* procesado número 50, para tomar el mismo valor de referencia en todos.

Para estas mediciones la *Raspberry Pi* estaba conectada a una toma de corriente, si estuviese conectada a una batería los tiempos de procesado de las imágenes hubiesen sido peores, en la tabla 5.4 se compara el procesamiento de las imágenes en blanco y negro con diferente resolución usando una batería y la toma de corriente.

En la gráfica 5.23 obtenida de esos datos se puede ver como aumenta el tiempo de procesado según aumenta el número de señales a detectar y también con el aumento del tamaño de la imagen, esto es previsible, ya que tiene más área en la que buscar y más señales. Donde apenas se nota diferencia es en el cambio de color a blanco y negro.

Resolución	Señales a procesar	T en blanco y negro (s)	T a color (s)
1280x720	5	2.087	2.132
1280x720	2	1.004	1.014
1280x720	1	0.618	0.616
1280x720	Sin señales	0.093	0.094
640x480	5	0.671	0.679
640x480	2	0.335	0.333
640x480	1	0.212	0.218
640x480	Sin señales	0.044	0.043
320x240	5	0.167	0.165
320x240	2	0.094	0.101
320x240	1	0.065	0.066
320x240	Sin señales	0.030	0.030

Tabla 5.2: Comparativa de procesado de cada frame con diferentes características. El tiempo es una media del tiempo de proceso de cada frame.

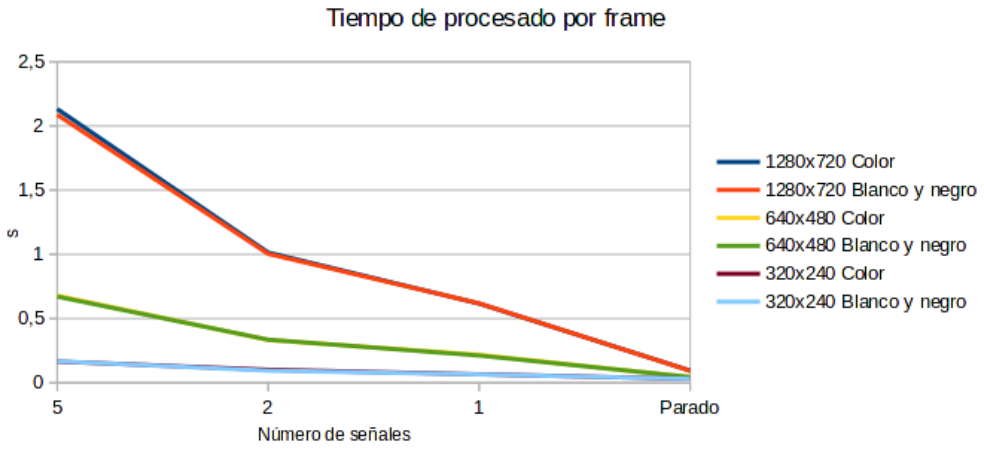

Figura 5.23: Gráfica que representa los tiempos de procesado de cada frame según el número de señales a procesar.

Para realizar estos cálculos añadí en el código de *Flask*, en concreto en la función *gen()* unos contadores y el resultado se imprime por la terminal. Para que funcione basta con descomentar el código y tener acceso a la terminal desde que se lanza la aplicación.

Los pasos que voy a seguir para calcular la relación en el tiempo de procesado de cada imagen con el resto son los siguientes:

- Voy a tomar el tiempo de cada resolución de parado como un valor a restar del tiempo de cada *frame*, porque este tiempo lo tomo como valor fijo a la hora de captar y mostrar la imagen en el navegador.
- Una vez quitado el tiempo fijo voy a dividir el valor obtenido de cada resolución.

Los datos que he escogido son los de color con 1, 2 y 5 señales a detectar y el resultado se puede ver en la tabla 5.3.

Imágenes a detectar	1280x720 / 640x480	1280x720 / 320x240	640x480 / 320x240
5	3.20	15.09	4.71
2	3.17	12.95	4.08
1	2.98	14.5	4.86

Tabla 5.3: Comparativa en el tiempo de procesado entre resoluciones.

Estos datos se acercan a lo esperado teniendo en cuenta la diferencia de píxeles de cada resolución.

Resolución	Señales a procesar	T con toma corriente (s)	T con batería (s)
1280x720	5	2.087	4.321
1280x720	2	1.004	1.808
1280x720	1	0.618	1.141
1280x720	Sin señales	0.093	0.169
640x480	5	0.671	1.458
640x480	2	0.335	0.603
640x480	1	0.212	0.397
640x480	Sin señales	0.044	0.069
320x240	5	0.167	0.347
320x240	2	0.094	0.148
320x240	1	0.065	0.105
320x240	Sin señales	0.030	0.033

Tabla 5.4: Comparativa procesando vídeos con batería y toma de corriente.

En la tabla 5.4 se puede ver como hay una pérdida de rendimiento, cuantas más imágenes hay a procesar más es la diferencia de rendimiento, en el caso de procesar cinco imágenes el tiempo de procesado es más de el doble en todos los casos con batería. En caso de no estar procesando las imágenes la pérdida de rendimiento es menor, de un 1.1 mientras que la de mayor resolución es de 1.8 veces peor.

En la tabla 5.5 también he hecho una comparación procesando tres vídeos a color de 5 segundos. Unas pruebas las hago sin usar *Flask*, ejecutando un *script* de *Python* y otras con la página web. Hay que señalar que los vídeos son distintos, así que el número de *frames* que contengan señales no tiene por qué coincidir entre ellos. El número de señales a detectar es de dos.

Resolución	Frames	Flask	T total	T por frame (s)
1280x720	145	No	149.04	1.027
1280x720	145	Si	152.79	1.054
640x480	144	No	50.41	0.350
640x480	144	Si	51.90	0.360
320x240	144	No	10.90	0.075
320x240	144	Si	11.66	0.081

Tabla 5.5: Comparativa procesando vídeos con y sin Flask.

Lo que me ha llamado la atención es que apenas hay diferencia de tiempos entre el procesado solo con el *script* y el de *Flask*. Y como era de esperar a mayor resolución mayor tiempo de procesado.

Si calculamos la relación del tiempo de procesado entre las distintas resoluciones obtenemos un valor de 2.9 para 1280x720 y 640x480, de 13 entre 1280x720 y 320x240 y de 4.6 para 640x480 y 320x240. Estos resultados se acercan a la relación que hay entre cada una de las resoluciones.

Sobre la detección de imágenes habría que compararlo con otras formas de detección y como no tengo forma de hacerlo lo que voy hacer a comparar las distintas fases de entrenamiento para una señal. Para ello en el archivo *xml* con los descriptores voy a ir borrando secciones en el apartado *stages*. Voy a usar los vídeos anteriores, los ejecutaré con el mismo *script* que antes ya que me resulta más cómodo que hacerlo con *Flask*. Aún así para ver si hay falsos positivos he procesado el vídeo en el navegador.

Al tomar como referencia el archivo con las 20 fases se aprecia como al ir quitando fases se detectan más señales sin dar falsos positivos, esto es porque hay frames que al estar la señal torcida o algo desenfocada no es tan estricto y la detecta. A mayor resolución antes tiene falsos positivos, con 1280x720 con 17 fases ya tiene falsos positivos, mientras que con las otras resoluciones es a partir de la fase 14.

Con 10 fases a mayor resolución mayor número de falsos positivos, con 1280x720 hay 1224, con la resolución de 640x480 hay 331 mientras que con la de 320x240 hay sólo 99. A pesar de ser vídeo distintos, parece que a mayor resolución hace falta un archivo de descriptores mucho más detallado. Esto

Resolución	Stage	Frames detectados	Falsos positivos
1280x720	20	66	No
1280x720	19	67	No
1280x720	18	68	No
1280x720	17	72	Si
1280x720	10	1224	Si
640x480	20	83	No
640x480	19	84	No
640x480	18	84	No
640x480	17	85	No
640x480	16	85	No
640x480	15	85	No
640x480	14	103	Si
640x480	10	331	Si
320x240	20	66	No
320x240	19	71	No
320x240	18	79	No
320x240	17	81	No
320x240	16	81	No
320x240	15	81	No
320x240	14	82	Si
320x240	10	99	Si

Tabla 5.6: Tabla comparativa con diferentes fases de detección

será porque a mayor número de píxeles mayor número de área a escanear y mayor número de patrones que se pueden encontrar.

La relación de negativos en la fase 10 con de las resoluciones 1280x720 y 640x480 es de 3.69 con las de 1280x720 y 320x240 es de13.36 y entre las de 640x480 y 320x240 es de 3.34.

Trabajos relacionados

Detección de señales de tráfico buscando zonas de color amarillo para compararlas con las formas señales conocidas.

Instituto de física – Sri Lanka, Detection and extraction of road traffic signs. www.researchgate.net

Ejemplo hecho en Mathlab que analiza los frames del vídeo en formato YCbCr y busca agrupaciones de color rojo, luego las compara con una plantilla de señales de advertencia.

MathWorks - Traffic Warning Sign Recognition. www.mathworks.com

En este proyecto se compara las formas encontradas en el imagen con las de unas señales preestrablecidas con el método Canny edge detector [37]

Miguel Ángel Sotelo - Fast road sign detection using Hough transform for assisted driving of road vehicles. www.isislab.es

Este ejemplo de detección de señales viene en un libro que compré para entender el funcionamiento de OpenCV y aprender a usarlo con python. Usan varios métodos distintos para reconocerlas pero ninguno es con este tipo de entrenamiento.

Packt publishing - Learning to recognize traffic signs. github.com

Estos dos blogs me sirvieron de ejemplo para comprender como entrenar el clasificador, no detectan señales, uno está entrenado para reconocer plátanos y el otro muñécos de lego, están algo desactualizados y no usan python pero aún así me han servido de gran ayuda.

Coding Robin - Train Your Own OpenCV Haar Classifier coding-robin.de

Electric Soup - Lego detection using OpenCV rdmilligan.wordpress.com

Conclusiones y Líneas de trabajo futuras

7.1. Conclusiones

He desarrollado una herramienta para reconocer señales de tráfico que utiliza un entrenamiento de tipo *Machine learning* que se llama *Entrenamiento en cascada basado en características Haar* y que hace uso de la *Raspberry Pi* y su cámara.

Con este tipo de entrenamiento he llegado a las siguientes conclusiones:

Me parece difícil llegar a reconocer las señales de tráfico de forma fiable, habría que aplicarle como apoyo otra técnica y así evitar falsos positivos. Esta podría ser alguna de las que aparece en los anexos, como la de MathWorks que usa regiones de color.

Otro problema de este tipo de detección que una misma señal la puede confundir dependiendo de la distancia a la que se esté tomada la imagen. Este comportamiento se puede apreciar en la figura 7.24

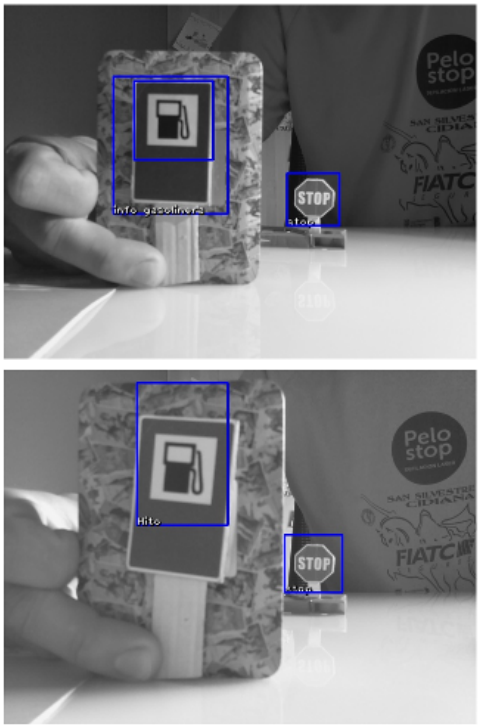

Figura 7.24: Misma señal tomada a diferente distancia.

La detección de las señales depende de las condiciones externas en las que se toma la imagen, de luz, tamaño o nitidez de la señal. Lo que conlleva que en dos imágenes tomadas de forma consecutiva y que tengan alguna pequeña variación es posible que en una detecte la imagen y en otra no. En la figura 7.25 se puede ver un ejemplo de este caso.

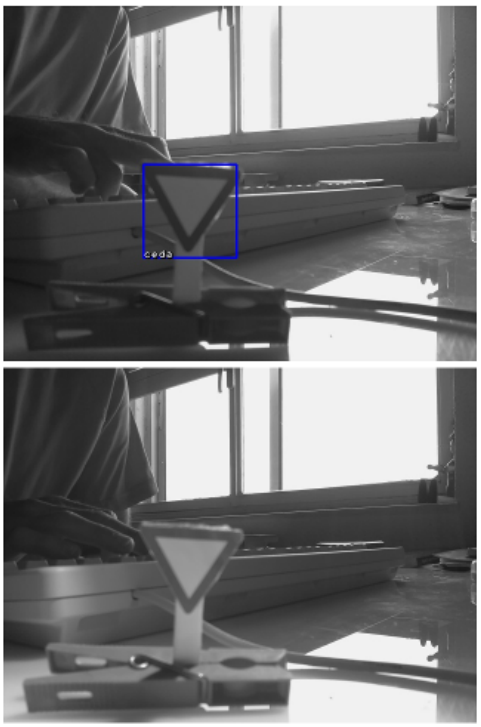

Figura 7.25: Mismo escenario con diferente iluminación.

La raspberry pi está bien como algo de aprendizaje o para desarrollar proyectos como entretenimiento, pero no para algo que requiera una alta capacidad de procesamiento, como es el caso de este trabajo, ya que los tiempos de procesado de una imagen cuando hay cinco señales no son aceptables.

7.2. Líneas de trabajo futuras

Las posibles mejoras del trabajo que se podrían realizar serían las siguientes:

El programa a pesar de no estar programado con hilos hace uso de los 4 núcleos de la *Raspberry Pi*, pero si el código se llevara a otra máquina sin que OpenCV estuviese compilado para usarlos no aprovecharía toda la capacidad de la máquina, así que una posible mejora sería programar un *pool* de hilos que fuese gestionado por el programa.

Para la interfaz web se podría usar *AJAX* para refrescar la parte necesaria de la página y no toda ella.

Habría que crear los test de la parte de *Flask* y si se añaden más funcionalidades a la página web sería necesario dividir el módulo correspondiente en varios más pequeños.

Bibliografía

[1] Bash. *Bash, Bourne Again SHell.* https://www.gnu.org/software/bash/ [Internet; descargado 1-septiembre-2017].

[2] Bootstrap. *Bootstrap, build responsive.* http://getbootstrap.com/ [Internet; descargado 1-septiembre-2017].

[3] Debian. *Devian versions.* https://www.debian.org/releases/ [Internet; descargado 1-septiembre-2017].

[4] DGT. *DGT, Nomas y señales.* www.dgt.es/Galerias/seguridad-vial/formacion-vial/cursos-para-profesores-y-directores-de-autoescuelas/XVII-Curso-de-Profesores/Normas-y-seales-Ed.-2014.pdf [Internet; descargado 1-septiembre-2017].

[5] faw3schoolske. *XML Tutorial.* https://www.w3schools.com/xml/ [Internet; descargado 1-septiembre-2017].

[6] ffmpeg. *ffmpeg, a complete, cross-platform solution to record, convert and stream audio and video.* https://ffmpeg.org [Internet; descargado 1-septiembre-2017].

[7] Firefox. *Firefox, the 100 % fresh, free-range, ethical browser.* https://www.mozilla.org/en-US/firefox/ [Internet; descargado 1-septiembre-2017].

[8] Flask. *Flask a microframework for Python.* http://flask.pocoo.org/ [Internet; descargado 1-septiembre-2017].

[9] Linux Foundation. *Automotive Grade Linux (AGL).* https://www.automotivelinux.org [Internet; descargado 1-septiembre-2017].

[10] Geany. *Geany a text editor.* http://geany.org/ [Internet; descargado 1-septiembre-2017].

[11] GitLab. *GitLab, The platform for modern developers.* https://about.gitlab.com/ [Internet; descargado 1-septiembre-2017].

[12] Guake. *Guake terminal.* http://guake.org/ [Internet; descargado 1-septiembre-2017].

[13] LaTeX. *LaTeX – A document preparation system.* http://www.latex-project.org/ [Internet; descargado 1-septiembre-2017].

[14] LaTeXila. *LaTeXila a LaTeX editor for the GNOME desktop.* https://wiki.gnome.org/Apps/LaTeXila [Internet; descargado 1-septiembre-2017].

[15] OpenCV. *about traincascade paremeters, samples, and other...* http://answers.opencv.org/question/7141/about-traincascade-paremeters-samples-and-other/ [Internet; descargado 1-septiembre-2017].

[16] OpenCV. *Cascade Classifier Training.* http://docs.opencv.org/3.1.0/dc/d88/tutorial_traincascade.html [Internet; descargado 1-septiembre-2017].

[17] OpenCV. *Installation in Linux.* http://docs.opencv.org/3.2.0/d7/d9f/tutorial_linux_install.html [Internet; descargado 1-septiembre-2017].

[18] OpenCV. *Open Source Computer Vision Library.* http://opencv.org/ [Internet; descargado 1-septiembre-2017].

[19] OpenCV. *Structured forests for fast edge detection.* http://docs.opencv.org/3.2.0/d0/da5/tutorial_ximgproc_prediction.html [Internet; descargado 1-septiembre-2017].

[20] OpenCV. *Zonas de cálculo de las variación de intensidad Haar.* https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html [Internet; descargado 1-septiembre-2017].

[21] Raspberry Pi. *Picamera.* https://www.raspberrypi.org/blog/camera-board-available-for-sale/ [Internet; descargado 1-septiembre-2017].

[22] Raspberry Pi. *Raspberry Pi learning resources.* https://www.raspberrypi.org/learning/hardware-guide/components/raspberry-pi/ [Internet; descargado 1-septiembre-2017].

[23] Raspberry Pi. *Raspbian, Foundation's official supported operating system.* https://www.raspberrypi.org/downloads/raspbian/ [Internet; descargado 1-septiembre-2017].

[24] Picamera. *API Reference*. http://picamera.readthedocs.io/en/release-0.4/api.html [Internet; descargado 1-septiembre-2017].

[25] Picamera. *Rapid capture and processing*. http://picamera.readthedocs.io/en/release-1.13/recipes2.html#rapid-capture-and-processing [Internet; descargado 1-septiembre-2017].

[26] pyimagesearch. *Install guide: Raspberry Pi 3 + Raspbian Jessie + OpenCV 3*. http://www.pyimagesearch.com/2016/04/18/install-guide-raspberry-pi-3-raspbian-jessie-opencv-3/ [Internet; descargado 1-septiembre-2017].

[27] Pylint. *Pylint - Code analisis for Python*. https://www.pylint.org/ [Internet; descargado 1-septiembre-2017].

[28] Python. *Pep8, Style Guide for Python Code*. https://www.python.org/dev/peps/pep-0008/ [Internet; descargado 1-septiembre-2017].

[29] Python. *Python programming language*. https://www.python.org/ [Internet; descargado 1-septiembre-2017].

[30] Radon. *Welcome to Radon's documentation!* http://radon.readthedocs.io/en/latest/ [Internet; descargado 1-septiembre-2017].

[31] Scholarpedia. *Zonas de cálculo de las variación de intensidad LBP*. http://scholarpedia.org/article/Local_Binary_Patterns [Internet; descargado 1-septiembre-2017].

[32] Ssh. *SSH (Secure Shell) protocol*. https://www.ssh.com/ssh/ [Internet; descargado 1-septiembre-2017].

[33] sshfs. *SSHFS, mount a remote filesystem*. https://github.com/libfuse/sshfs [Internet; descargado 1-septiembre-2017].

[34] Tmux. *Tmux, terminal multiplexer*. https://github.com/tmux/tmux [Internet; descargado 1-septiembre-2017].

[35] Vim. *Vim - the ubiquitous text editor*. www.vim.org [Internet; descargado 1-septiembre-2017].

[36] w3schools. *HTML5 Tutorial*. https://www.w3schools.com/html/default.asp [Internet; descargado 1-septiembre-2017].

[37] Wikipedia. *Canny edge detector*. https://en.wikipedia.org/wiki/Canny_edge_detector [Internet; descargado 1-septiembre-2017].

[38] the free encyclopedia Wikipedia. *Haar-like feature*. https://en.wikipedia.org/wiki/Haar-like_feature [Internet; descargado 1-septiembre-2017].

[39] the free encyclopedia Wikipedia. *Local binary patterns*. https://en.wikipedia.org/wiki/Local_binary_patterns [Internet; descargado 1-septiembre-2017].

[40] wikipedia The free Encyclopedia. *Summed-area table*. https://en.wikipedia.org/wiki/Summed-area_table [Internet; descargado 1-septiembre-2017].