Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

# Trabajo de Fin de Grado

Grado en Ingeniería Informática

# Sistema de asistencia a la conducción mediante el procesamiento de imágenes procedentes de una dash-cam

*Driving assistant system through a dash-cam image processing*

Alexis Rodríguez Casañas

La Laguna, 10 de septiembre de 2019

D. **Néstor Morales Hernández**, con N.I.F. 54.048.001-W profesor Titular de Universidad adscrito al Departamento de **Ingeniería informática y sistemas** de la Universidad de La Laguna, como tutor

D. **Jonay Tomás Toledo Carrillo**, con N.I.F. 78.698.554-Y profesor Titular de Universidad adscrito al Departamento de **Ingeniería informática y sistemas** de la Universidad de La Laguna, como cotutor

**C E R T I F I C A (N)**

Que la presente memoria titulada:

*"Sistema de asistencia a la conducción mediante el procesamiento de imágenes procedentes de una dash-cam"*

ha sido realizada bajo su dirección por D. **Alexis Rodríguez Casañas**,

con N.I.F. 54.054.901-W.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de septiembre de 2019

Agradecimientos

A mi tutor y a todos los profesores que he encontrado a lo largo de mi vida académica.

A mis padres.

# Licencia

* Si quiere permitir que se compartan las adaptaciones de tu obra y NO quieres permitir usos comerciales de tu obra indica:

## Resumen

*El objetivo de este trabajo ha sido el desarrollo de un sistema de asistencia a la conducción mediante el procesamiento de imágenes procedentes de una dash-cam. El proyecto ha intentado sentar las bases para un sistema genérico que detecte el entorno del vehículo que lo utiliza, detectando otros vehículos, señales de tráfico, personas, carriles, obstáculos, etc.*

*Para la obtención de imágenes se utiliza una dash-cam, que no es más que una pequeña cámara situada en el salpicadero del vehículo. Estas cámaras se han hecho muy populares y son utilizadas en diversos países para grabar lo ocurrido en caso de accidente. Si bien en España estas imágenes no se admiten como prueba en un juicio, su uso no está prohibido.*

*Nos encontramos ante un tema complejo en el que actualmente se encuentran trabajando grandes equipos de expertos del sector, bajo la tutela y los punteros recursos computacionales y económicos de empresas líder. Además, nos encontramos ante un sistema de tiempo real donde se ven comprometidas vidas humanas. Estos sistemas cumplen unas características a todos los niveles que necesitan de un gran número de recursos humanos, temporales y tecnológicos.*

*Por ello, este proyecto persigue, más que lograr un sistema completamente funcional y apto para la vida real, servir de introducción a este campo de estudio y comprobar hasta qué punto la evolución de la tecnología permite que a día de hoy sea posible realizar un sistema de estas características con recursos de muy bajo coste y basados en software libre.*

*Las principales aportaciones que pretende realizar este proyecto son las siguientes:*

- *Desarrollo de un prototipo de asistencia a la conducción completo basado en imágenes tomadas pro una dash-cam.*
- *Análisis y estudio de los diferentes métodos de detección de objetos disponibles basados en redes neuronales convolucionales, atendiendo tanto a su eficiencia computacional como a su tasa de acierto.*
- *Análisis y estudio de métodos de detección de carreteras. Comparando métodos clásicos de visión por computador y redes neuronales.*
- *Desarrollo modular de los componentes, de forma que puedan ser reutilizados en otros proyectos y basados en filosofía de software libre.*



**Palabras clave:** redes neuronales, visión por computador, ROS, conducción, coche, detección de objetos

## Abstract

*This project is about a driving assistant system using a dash-cam. This work tries to lay the foundations for a generic system which can detect the vehicle environment, such as other vehicles, pedestrians, obstacles or road lanes.*

*To obtain images, a dash-cam is used. A dash-cam is a small camera located on the dashboard of the vehicle. These cameras have become very popular and are used in various countries to record what happened in case of an accident. Although in Spain these images are not admitted as evidence in a trial, their use is not prohibited.*

*This work faces complex topics which are currently being investigated and developed by large teams of the best experts, using the best available technology and resources. In addition, this is a real time system where human lives are compromised. Building a real and ready to use system like this requires a huge amount of human, technological and temporary resources.*

*For this reason, this project seeks, rather than achieving a fully functional and suitable system ready to use in real life, to serve as an introduction to this computer science field. It also pretends to verify how technology has evolved allowing to create such a complex system with very low cost and open source based resources.*

*The main contributions that this project intends to achieve are the following:*

- *Development of a complete driving assistance prototype based on images taken by a dash-cam*
- *Analysis and study of the different methods of detecting objects available in convolutional neural networks, taking into account both their computation efficiency and success rate.*
- *Analysis and study of road lane detection, comparing classic methods of computer vision and neural networks.*
- *Modular development of the components so that they can be reused in other projects, everything done with opensource tools.*



**Keywords:** neural networks, computer vision, ROS, driving, car, object detection

# Índice general

# Índice de figuras

# Capítulo 1
# Introducción

## 1.1 Antecedentes

### 1.1.1 Breve historia de la visión por computador

La detección de objetos, pese a estar tan de moda en la actualidad, no es un campo reciente en las ciencias de la computación. Ya en los años sesenta se experimentaba en varias universidades con esta tecnología. Ingenieros en computación y matemáticos trabajaron de la mano en las siguientes décadas enriqueciendo cada vez más este campo. De hecho, en la década de los setenta se formaron los cimientos de muchos de los algoritmos de visión por computador que hoy se utilizan, como la extracción de bordes, el etiquetado de líneas, el modelado poliédrico y no poliédrico o la estimación del movimiento.

En las siguientes décadas se comenzaron a utilizar conjuntamente técnicas de aprendizaje estadístico como los famosos *Eigenfaces* (1987), conjuntos de vectores propios utilizados para el reconocimiento de caras.

### 1.1.2 Breve introducción a las redes neuronales

Igual que ocurre con la visión por computador, las redes neuronales no son en realidad ningún concepto reciente. Ya en los años cuarenta existían conceptualmente en forma de modelos matemáticos. El motivo por el que no se había oído hablar de ellas hasta hace poco es que se necesita una gran cantidad de recursos computacionales para poder entrenar y ejecutar una red con buenos resultados. En los últimos años, se han conseguido avances significativos gracias a la mejora del hardware y al uso de las GPUs, que han permitido materializar este tipo de computaciones.

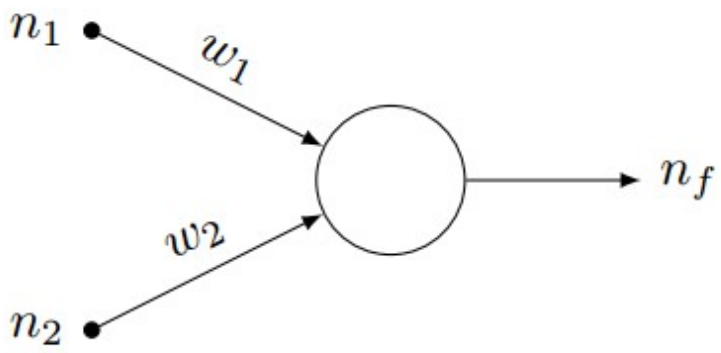
Sin embargo, una vez que se tienen los requisitos necesarios para su implementación, las redes neuronales parecen cobrar vida propia y sus beneficios son exponenciales. De hecho, en el campo de la visión artificial, por ejemplo, parecen haber superado sin demasiado esfuerzo y de forma casi natural, algoritmos que los humanos habíamos ido puliendo a lo largo de las últimas décadas. Ahora bien, ¿cómo funciona una red neuronal?

Las redes neuronales deben su nombre a la idea de imitar el funcionamiento

de las neuronas en los organismos vivos: un conjunto de neuronas conectadas entre sí que trabajan en conjunto y que van creando y reforzando ciertas conexiones para “aprender” algo que se queda fijo en el tejido.

En realidad, dicho de forma simplificada, las redes neuronales se valen de las matemáticas (sobre todo de la estadística) y sólo hacen una cosa: dados unos parámetros, buscar la forma idónea de combinarlos para predecir un cierto resultado. Una vez se ha encontrado el “peso” correcto de todos los parámetros, se dice que la red está “entrenada” y esta es capaz de predecir resultados para casos que no había visto antes.

La unidad básica de una red neuronal es el perceptrón, el cual sería la neurona estableciendo un símil con el modelo biológico. El siguiente perceptrón posee dos entradas, **n1 y n2**, a dichas entradas se les asigna una importancia o peso, **w1 y w2** respectivamente. Durante la fase de entrenamiento, los pesos se van ajustando hasta producir la salida deseada. Una vez el perceptrón está ajustado o entrenado, es lógico que podrá predecir casos similares.




Los perceptrones se agrupan en capas, formando así las redes multicapa. Las capas exploran los datos y sacan todas las características relevantes. Dichas capas acaban a su vez “aprendiendo” a encontrar y detectar las características que mejor ayudan a clasificar los datos. Normalmente, con más capas se pueden mejorar las predicciones en conjuntos de datos más complicados.

En el caso del presente trabajo, se utilizarán las redes neuronales convolucionales. Las redes neuronales convolucionales son un tipo de red neuronal muy utilizadas para el tratamiento de imágenes y gracias a la llamada convolución, permiten ganar eficiencia y reducir la cantidad de parámetros de la red. El uso de una red neuronal con una imagen de alta definición tiende a ser inviable. Tengamos en cuenta que con una imagen de 200x200 píxeles RGB, una sola neurona plenamente conectada en una primera capa oculta tendría 200x200x3= 120.000 pesos. Esta cantidad no es desorbitada pero

podemos hacernos una idea de la complejidad con la que puede escalar el problema.

Las redes neuronales convolucionales trabajan modelando de forma consecutiva pequeñas piezas de información y luego combinando esta información en las capas más profundas de la red. Por lo general, estas redes poseen una estructura con tres tipos de capas bien diferenciadas:

Capa convolucional: utiliza la operación de convolución. Esto consiste en aplicar un filtro o kernel sobre la imagen de entrada que devuelve un mapa con las características más significativas de la imagen original, pero reduciendo considerablemente el tamaño de los parámetros y los cálculos.

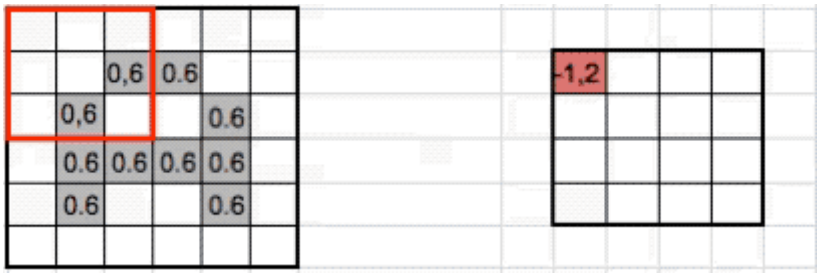



Figura 1.1: A la izquierda, imagen original con el kernel superpuesto en rojo. A la derecha, mapa resultante.

Es necesario mencionar que en realidad no se aplica un solo kernel, sino muchos. Por ejemplo, en este primer proceso de convolución se podrían tener 32 filtros, con lo cual acabaríamos obteniendo 32 matrices de salida (conjunto conocido como feature mapping"). Estas 32 imágenes nuevas poseen ciertas características de la imagen original que en el futuro, ayudarán a distinguir un objeto.

Capa de subsampling, reducción o pooling: esta capa se sitúa generalmente después de la capa convolucional y su utilidad radica en la reducción de dimensiones espaciales. La operación que se suele utilizar es max-pooling, que divide la imagen en un conjunto de rectángulos y de cada subregión se queda con el máximo valor.

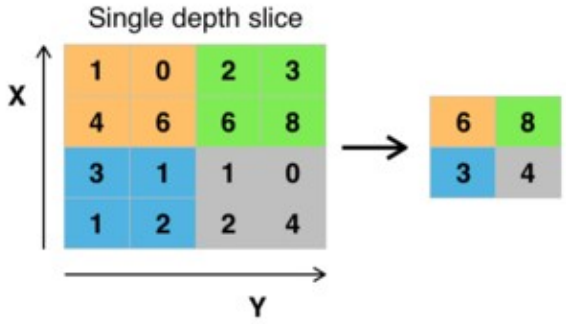



Figura 1.2: Operación de max-pooling

Como se observa en la figura anterior, luego de hacer el subsampling nos queda una matriz mucho más reducida y la imagen que representa, en teoría, sigue almacenando la información más importante para detectar características deseadas.

Hasta este punto, todo el proceso anterior describe una primera fase de convolución. Es decir, imagen de entrada, aplicación de los kernels con función de activación, obtención de los feature mapping y aplicación de pooling a estos últimos. Con esta primera convolución la red es capaz de detectar las características más primitivas como líneas o curvas. A medida que se repita el proceso, los mapas de características serán capaces de reconocer formas más complejas.

Tras iterar este proceso el número de veces necesario, la última capa es conectada a una capa de neuronas tradicionales. A esta capa, se le aplica una función llamada softmax, que conecta contra la capa de salida final y que tiene una cantidad de neuronas correspondientes a las clases que se desea clasificar. Las salidas del entrenamiento tendrán el formato conocido como one-hot-encoding y la función softmax se encarga de pasar a probabilidad el resultado de las neuronas de salida. Por ejemplo, si estamos clasificando perros y gatos, el one-hot-encoding será [1, 0] y [0, 1], y si la función softmax nos da una salida como [0, 2, 0, 8] quiere decir que hay un 20% de probabilidad de que sea perro y un 80% de probabilidad de que sea gato.

Recordemos que en una red neuronal tradicional, mediante el backpropagation se mejora el valor de los pesos hasta que se ajustan de forma óptima. Sin embargo, en el caso de una red neuronal convolucional, el valor que se ajusta es el de los kernels o filtros. Esto es una gran ventaja, ya que como se vio con anterioridad, cada kernel es de un tamaño reducido.

## 1.2 Tecnologías y herramientas utilizadas

### 1.2.1 ROS: Robotic Operating System

ROS (Robotic Operating System) es un sistema que provee librerías y herramientas para ayudar a los desarrolladores a crear aplicaciones en el mundo de la robótica. ROS provee abstracción de hardware, controladores, librerías, herramientas de visualización y mucho más. ROS es de código abierto, pues está bajo la licencia open source BSD.

Utilizaremos ROS durante el presente proyecto para sacar partido de su abstracción y modularidad, de forma que el proyecto sea fácil de compartir o modificar. Además, ROS posee una característica muy interesante que es el paso de mensajes. ROS nos permite hacer programas con una estructura cliente-servidor sin que tengamos que preocuparnos de la implementación. De cara a un sistema de tiempo real esto es muy interesante porque entre otras cosas, podemos tener un mayor control y modularidad del sistema, haciendo que si por ejemplo un elemento del mismo cae, el resto del sistema no lo haga

o incluso podamos recuperarlo o actuar en consecuencia.

En ROS, el programa se divide en nodos, que son otros programas en ejecución. Los nodos publican los datos a través de tópicos y otros nodos, pueden suscribirse a dichos tópicos. Un nodo puede publicar y estar suscrito a más de un tópico. Todo esto esconde en realidad, una implementación basada en sockets como habría que programar manualmente en cualquier aplicación cliente-servidor.

### 1.2.2 TensorFlow

Tensorflow es una librería de código abierto para aprendizaje automático desarrollada por Google, que utiliza grafos de flujo de datos. Los nodos en los grafos representan operaciones matemáticas, mientras que las aristas de los grafos representan las matrices de datos multidimensionales (tensores) comunicadas entre ellos.

Tensorflow es una formidable herramienta para construir y entrenar redes neuronales. Su arquitectura le permite implementar el cálculo a una o varias CPUs, GPUs y puede ejecutarse en equipos de escritorio o dispositivos móviles con una sola API.

### 1.2.3 OpenCV

OpenCV (OpenSource Computer Vision) es una librería de visión por computador. Al estar bajo licencia BSD es de código abierto, pudiendo este ser utilizado y modificado. Posee una comunidad de alrededor de 50.000 personas y más de 7 millones de descargas. La librería contiene más de 2500 algoritmos relacionados con el mundo de la visión artificial, machine learning y otras utilidades. Se usa en aplicaciones como reconocimiento de objetos, caras, seguimiento, monitorización, ayuda de navegación para robots, etc.

OpenCV ofrece además un binding para Python, entre otros lenguajes. Esto es de una gran utilidad, porque significa que podemos aprovecharnos de la simplicidad de codificar en este lenguaje sin la penalización de su bajo rendimiento con respecto al C++, que es el lenguaje en que está escrita originalmente.

## 1.3 Estado del arte

Como ya se dijo en la introducción, gracias a los avances tecnológicos, la pareja formada las redes neuronales y la visión por computador está viviendo un auge sin precedentes y que parece haber llegado para quedarse. En este punto, concretaremos además dentro de este campo para referirnos al mundo de la automoción.

Ya desde hace algunos años los coches cuentan con sistemas de ayuda a la conducción que cada vez cobran más importancia y protagonismo. Hoy en día es habitual que un coche detecte las señales de tráfico, realice una frenada de emergencia si detecta un obstáculo delante o haga correcciones en el volante para no salirse del carril. Google y otras compañías trabajan para lograr

vehículos totalmente autónomos con la visión por computador como protagonista, y parece que empresas como Tesla no van nada mal encaminadas para lograr este objetivo.

Es un hecho tan seguro que el futuro toma esta dirección que diversas empresas como Nvidia están apostando fuertemente por el sector. Esta última, ha creado un departamento orientado a la conducción autónoma y ofrece todo tipo de herramientas de hardware y software a clientes y desarrolladores. Actualmente, Nvidia posee toda una suite orientada a la conducción autónoma. Posee todo un conjunto de herramientas de software, simuladores, servidores para entrenar las redes neuronales, entornos de desarrollo, hardware como plataformas de cálculo autónomas , o redes neuronales propias, como WaitNet, que se encarga específicamente de detectar cruces, o ClearSightNet, que se ocupa de evaluar las posibilidades de las cámaras de ver claramente y determinar causas de oclusiones, bloqueos y reducciones de visibilidad

## 1.4 Problemática

A pesar de su auge, las redes neuronales y los sistemas inteligentes orientados a la conducción no están libres de problemas. Algunos de estos problemas son retos técnicos evidentes, pero también existen relativos a ámbitos insospechados a primera vista, como éticos, psicológicos o medioambientales.

En este apartado se brinda una visión general sobre la problemática tanto de las redes neuronales como de la incorporación de la tecnología inteligente a la conducción.

### 1.4.1 Problemática de las redes neuronales

Las redes neuronales no están exentas de desventajas. Entre ellas, las más significativas son la complejidad del tiempo de aprendizaje para tareas complicadas. Requieren una inmensa cantidad de datos para ser entrenadas y si no existen, el proceso de obtención y preparación de los mismos es laborioso.

Un conocido aspecto negativo de las redes neuronales es su “oscuridad” con respecto a la interpretación humana. Cuando una red neuronal produce una solución, no hay forma de saber por qué y cómo la ha encontrado. Esto puede reducir la confianza en la red bajo determinadas circunstancias. Además, no existe una regla específica por el momento para determinar qué estructura debe tener una red neuronal artificial, y es común ver incluso a los mayores expertos del sector utilizando un método de ensayo y error.

Otra famosa desventaja son los elevados recursos de hardware que se necesitan. No es ningún secreto que si se desea entrenar o ejecutar una red neuronal con buenos resultados, se necesita potencia, pero esto a gran escala produce una desventaja sobre la que no suele hablarse, y es la contaminación. La computación necesita electricidad y la producción de esta (de forma tradicional) contamina. En realidad, esta enorme dependencia de energía en un

centro de procesamiento de datos no se debe tanto al proceso de computación en sí (que tampoco es despreciable) sino al proceso de mantener a una temperatura adecuada la infraestructura. Los costes de refrigerar un CPD son conocidos y elevados. Algunos expertos hablan de que en un futuro, el coste en contaminación de ciertas redes neuronales excesivamente potentes sería un factor tan significativo como para tenerlo en cuenta.

### 1.4.2 Problemática de la conducción inteligente

Los problemas a la hora de enfrentarse a retos relacionados con los comportamientos inteligentes y la conducción no son triviales y los expertos que trabajan en el sector pueden dar fe de ello. Para empezar, en realidad uno de los mayores retos de la robótica sigue siendo uno de los problemas más simples a nivel conceptual: que el sistema conozca su situación o posición con respecto al entorno. Este problema puede ser resuelto en un entorno controlado, por ejemplo una habitación, colocando balizas con las que el robot se orienta. También puede ser medianamente tratado en exteriores utilizando un GPS, pero definitivamente aún no podemos hablar de un sistema con capacidad de sentido espacial propio, como hacemos los humanos, que simplemente con las imágenes que vemos por nuestros ojos conocemos nuestra situación exacta en un lugar.

Continuando con los aspectos técnicos, hay situaciones que pueden ser un verdadero reto, como la nieve en los países nórdicos, donde las carreteras muchas veces simplemente desaparecen. Para un humano es instintivo seguir la trayectoria natural de la curva, pero esto no es tan sencillo para un sistema robótico. Otro ejemplo son las carreteras en mal estado, las que no presentan señalización ni ningún tipo de marca, o simplemente las que por un error humano, poseen marcas equivocadas. Algo poco común, pero existente y por ello no despreciable. Existen infinidad de factores que los humanos pasamos por alto día a día pero que en realidad son muy complejos de resolver por una máquina. Por ejemplo, uno de los mayores problemas de la empresa Volvo en Australia es la detección de canguros para evitar que sean atropellados. Sin embargo, la mayoría de sistemas de cálculo de distancia se apoyan en utilizar el suelo como referencia y el canguro, se desplaza saltando. Por ello, cuando está en el aire el sistema interpreta que se ha alejado pero al aterrizar, vuelve a “aparecer” cerca de nuevo

Además, debemos incluir el factor psicológico humano que produce excepciones en todo tipo de situaciones. Por ejemplo, un conductor sin preferencia puede cedernos el paso, un ciclista puede hacer una señal gestual para realizar una determinada acción o un peatón puede detenerse en un paso de peatones para hablar con una persona, pero no cruzar. Todas las personas que interactúan a diario en la carretera, la mayoría de las veces resuelven todos estos problemas con una simple mirada a los ojos o un ligero movimiento de la cabeza, ya que estamos naturalmente preparados para ello. Estas interacciones, en cambio, son muy difíciles de lograr en un sistema robótico.

Continuando en esta línea, en ocasiones el vehículo debe reconocer y sopesar rápidamente excepciones puntuales. Por ejemplo, no puede detenerse en seco en una autovía con el riesgo de provocar un accidente si detecta un obstáculo como una caja de cartón, pero sí debería hacerlo ante una persona.

En definitiva, la mezcla resultante de los sistemas inteligentes y la conducción es un tema apasionante, en auge y que parece ser el camino hacia el futuro, pero pese a los abrumadores avances vistos en los últimos años, en realidad aún estamos muy lejos de poder confiar plenamente en todas las situaciones en un sistema tanto autónomo como de asistencia.

# Capítulo 2
# Objetivos

El objetivo de este trabajo es intentar lograr un sistema de asistencia a la conducción mediante las imágenes procedentes de una dash-cam desde un punto de vista didáctico y académico. Se trata de un sistema muy complejo en el que se encuentran trabajando gigantescos equipos de empresas pioneras en el sector. Además, se trata de un sistema de tiempo real donde están en juego vidas humanas. Todo ello hace que sea muy difícil lograr un proyecto realista en el plazo dado, pero sí que puede servir de introducción en el campo de estudio y además, puede ser interesante ver hasta qué punto ha evolucionado la tecnología como para ver qué resultados pueden obtenerse con recursos caseros de bajo coste y código abierto.

## 2.1 El pipeline de procesamiento con ROS

El primer objetivo será lograr establecer un pipeline de procesamiento con ROS. Independientemente de que nuestros nodos estén vacíos y no efectúen trabajo alguno, no es impedimento para que los datos fluyan de un nodo a otro, por lo que desde el principio del proyecto siempre tendremos un vídeo de entrada y una imagen de salida.

## 2.2 El detector de obstáculos

El detector de obstáculos se encarga de reconocer vehículos, personas u otros objetos.

Para implementar el detector de obstáculos se utilizará un modelo preentrenado de los muchos que ofrece el Model Zoo de TensorFlow. Se escribirá el código pertinente para utilizar este modelo dentro de un nodo de ROS que haga posible la detección, pero no debemos preocuparnos de realizar entrenamiento. Se probarán diversos modelos entrenados con diferentes data sets para estudiar y decidir cuál se adapta mejor al proyecto.

De cara al diseño del nodo de ROS, se utilizará el mostrado en la imagen siguiente. Donde se puede observar que el nodo recibe como entrada la imagen proveniente del nodo anterior, que puede ser una fuente de vídeo o una cámara, resultando transparente para el resto del sistema. Lo más

destacable de este nodo es que publicará la imagen con la información ya mostrada sobre ella, pero también publicará dicha información de detección por separado. El objetivo de esto es poder utilizar esa información, dependiendo de si deseamos armar un diseño lineal o paralelo como el de las figuras 3.1 y 3.2 respectivamente, mostradas más adelante. Es posible diseñar los nodos desde el principio con esta estructura y preocuparse más tarde por aspectos de si es posible su implementación o no.

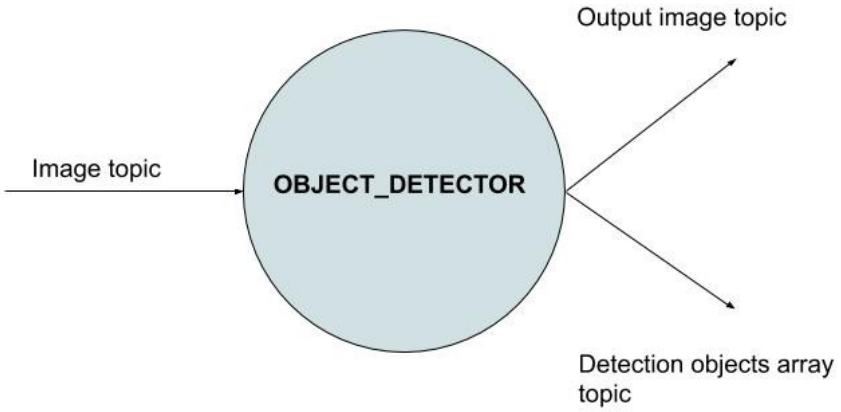



Figura 2.1: Nodo detector de obstáculos

## 2.3 El detector de carril

La idea principal consiste en utilizar también un modelo preentrenado como puede ser LaneNet. No obstante, debido a las limitaciones del sistema se priorizó optar por probar algunas aproximaciones manuales basadas en técnicas de visión por computador tradicional. El detector de carril se encarga de detectar las líneas de la carretera entre las cuales circula el vehículo. En una situación real podría advertir, por ejemplo, si dicho vehículo se sale del carril.

Atendiendo a detalles de diseño en ROS, el nodo tiene el aspecto que se muestra en la siguiente figura. Obsérvese que recibe una imagen y que dicha imagen puede ser una previamente procesada por el nodo detector de obstáculos, como la imagen original de la fuente. Esto permite que los nodos sean fácilmente reutilizables y recolocables tanto por nosotros mismos como por cualquier otro programador. Nótese que no se ha entrado en detalles de implementación ni se ha especificado si lo que contendrá el nodo es una red neuronal o una aproximación tradicional, ya que para el sistema es indiferente.

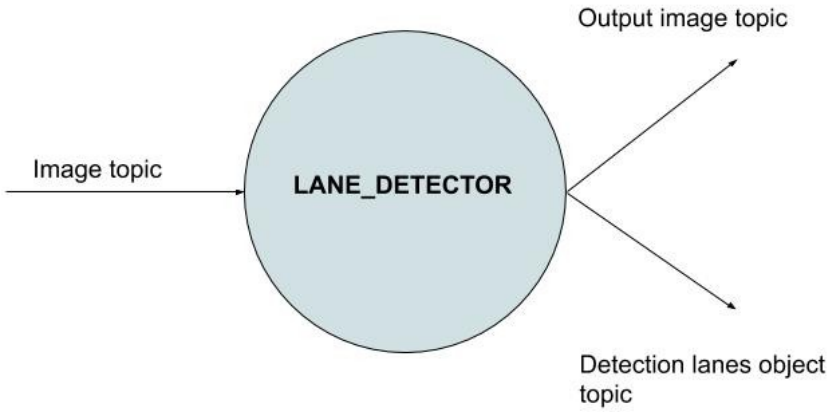



Figura 2.2: Nodo detector de carreteras

## 2.4 El HUD o visualizador

Por supuesto, la información debe visualizarse para saber que la estamos procesando adecuadamente y poder mostrarla al usuario. En principio, esto no es una tarea complicada ya que al trabajar con TensorFlow, este nos ofrece las funciones necesarias para mostrar toda la información de forma sencilla, sin necesidad de hacerlo manualmente en OpenCV.

Como se observa en la figura siguiente, este nodo está suscrito a tres tópicos. Por un lado, recibe mensajes que contienen la información de las líneas de la carretera y de los obstáculos. Por otra parte, recibe una imagen, que puede ser la imagen final procesada por los nodos anteriores, si se realiza un diseño en forma de tubería, o la imagen sin procesar de la fuente original, para superponerle la información obtenida si se opta por un esquema en paralelo. Esto se puede observar con más detalle en el siguiente capítulo.

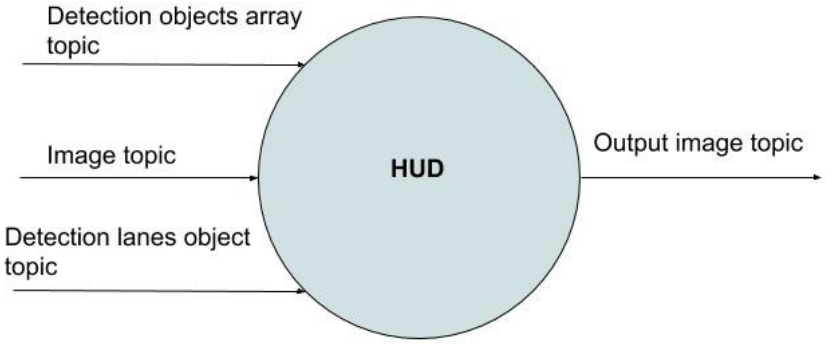



Figura 2.3: Nodo HUD o visualizador

# Capítulo 3
# Desarrollo

En este capítulo se entrará en detalles sobre cada fase del proyecto, tanto a nivel de implementación como de resultados. Téngase en cuenta que no se mostrará absolutamente todo el código fuente necesario para conseguir un determinado fin sino únicamente las partes más relevantes, con el objeto de no entrar en detalles triviales de implementación ajenos al tema que nos ocupa. El código completo se encuentra en un repositorio referenciado en el anexo, al final de este trabajo.

## 3.1 El pipeline de procesamiento con ROS

### 3.1.1 Primer diseño

Como vimos con anterioridad, nuestro sistema no va a ser un programa, sino que en realidad estará dividido en varios programas o nodos de ROS, en nuestro caso. Lo primero, por ello, es diseñar y crear un pipeline de procesamiento. Como podemos ver en la figura, un nodo de ROS se encarga de obtener la fuente de vídeo y enviar la imagen al siguiente nodo. Aquí comenzamos a ver la modularidad de ROS: al resto del sistema le es indiferente que la imagen provenga de una cámara en directo o de un vídeo pregrabado. Esta imagen ingresará en el detector de objetos, que hará su función y volverá a publicar la imagen con la información añadida. El pipeline sigue su camino de forma lineal hasta terminar en el nodo visualizador, que es el encargado de mostrar una ventana con el contenido del resultado final.


Figura 3.1: Esquema de trabajo lineal

En la siguiente figura, se puede observar el fichero de configuración *.launch* de ROS que representa el pipeline de la figura anterior. Los ficheros *.launch* son muy cómodos porque nos permiten con un solo comando lanzar todos los nodos con una serie de parámetros que previamente hemos establecido. También, por supuesto especificamos a qué tópicos se suscriben estos nodos. Podemos establecer un sinfín de parámetros. Uno realmente interesante es *respawn* con el cual si un nodo cae, ROS intentará volver a lanzarlo sin que nosotros tengamos que preocuparnos de ninguna implementación.

```
<launch>

    <!-- Lanzar el nodo que hace streaming del vídeo-->
    <include file="$(find video_stream_opencv)/launch/camera.launch" >
        <arg name="camera_name" value="videofile" />
        <!-- Ruta al fichero de vídeo -->
        <arg name="video_stream_provider" value="/home/alex/catkin_ws/video.mp4" />
        <arg name="fps" value="30" />
        <arg name="frame_id" value="videofile_frame" />
        <arg name="camera_info_url" value="" />
        <arg name="flip_horizontal" value="false" />
        <arg name="flip_vertical" value="false" />
        <arg name="visualize" value="false" />
    </include>

    <!-- Lanzar el nodo detector de carriles-->
    <node pkg="bridge" type="bridge.py" name="opencv_bridge" output="screen">
        <remap from="image" to="/videofile/image_raw" />
    </node>

    <!-- Lanzar el nodo que detecta los objetos-->
    <node pkg="tensorflow_object_detector" type="detect_ros.py" name="tensorflow_object_detector" output="screen">
            <remap from="image" to ="opencv_bridge_output" />
    </node>

    <!-- Lanzar el nodo que visualiza la salida en una ventana-->
    <node pkg="image_view" type="image_view" name="image_view" output="screen">
        <remap from="image" to="/result"/>
    </node>

</launch>
```

Figura 3.2: Fichero de configuración ROS que representa un pipeline de procesamiento lineal

### 3.1.2 Segundo diseño

El esquema anterior presenta un problema y es que al tener forma de tubería, si un nodo cae, todo el sistema lo hará, ya que la información no puede seguir su camino.

La siguiente figura muestra un diseño modular que sería ideal conseguir. Como se puede observar, ahora el detector de objetos no publica una imagen, sino la información que hay que pintar en dicha imagen y es el visualizador el que se encargará de hacerlo. Lo mismo ocurre con el detector de carreteras. De esta forma, si un nodo cayese, el resto del sistema seguiría funcionando con

normalidad e incluso podríamos recuperar ese nodo en tiempo de ejecución o tomar las medidas oportunas. Si el sistema escalase incorporando más sensores y funcionalidades, está claro que este diseño sería el que deberíamos perseguir, ya que sería impensable que todo el sistema autónomo del vehículo se viese comprometido por un fallo de un determinado módulo encargado de una tarea menor.

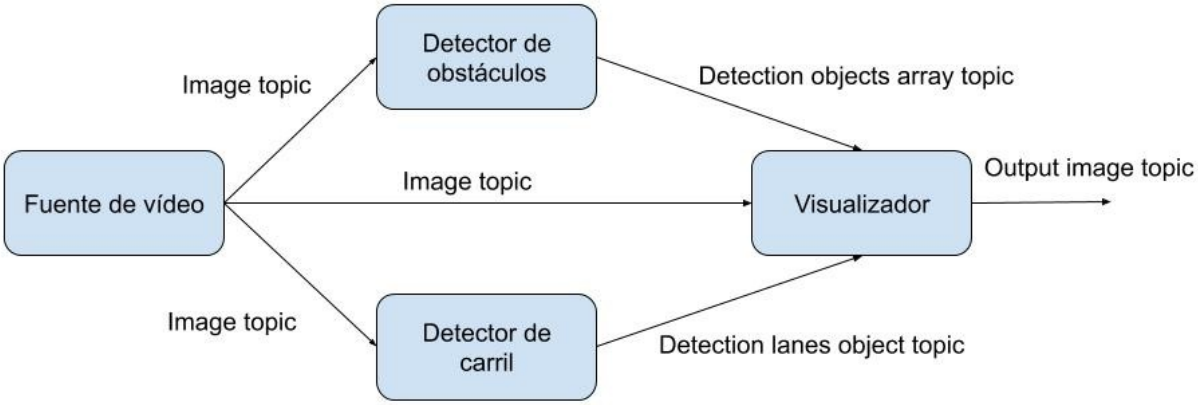



Figura 3.3: Esquema en paralelo

La siguiente figura muestra el fichero *.launch* necesario para conseguir el esquema anterior. Nótese que el nodo visualizer está suscrito tanto al nodo detector de carriles, como al detector de objetos y el nodo de streaming, pero no recibe la imagen sino la información que estos obtienen de la imagen a la cual están suscritos. Luego, el nodo visualizador utiliza esta información y la imagen en blanco para juntarlo todo en una sola salida, que envía al nodo que muestra todo en una ventana.

Como se puede observar, lo más interesante de este esquema es que, si por ejemplo, el nodo detector de carril cae, el visualizador sigue recibiendo información de los obstáculos e imagen y por tanto el sistema seguiría funcionando, mostrando los obstáculos mientras el nodo caído intenta ser recuperado. Incluso en un caso donde ambos nodos detectores caen, el sistema no se colgaría y continuaría mostrando la fuente de vídeo. Si el proyecto escalase incorporando más sensores u otros sistemas tanto hardware como software y contásemos con, por ejemplo, una docena de nodos, resulta fácil ver la elegancia y robustez de este diseño.

```xml
<launch>

    <!-- Lanzar el nodo que hace streaming del vídeo-->
    <include file="$(find video_stream_opencv)/launch/camera.launch" >
        <arg name="camera_name" value="videofile" />
        <!-- Ruta al fichero de vídeo -->
        <arg name="video_stream_provider" value="/home/alex/catkin_ws/video.mp4" />
        <arg name="fps" value="30" />
        <arg name="frame_id" value="videofile_frame" />
        <arg name="camera_info_url" value="" />
        <arg name="flip_horizontal" value="false" />
        <arg name="flip_vertical" value="false" />
        <arg name="visualize" value="false" />
    </include>

    <!-- Lanzar el nodo detector de carriles-->
    <node pkg="bridge" type="bridge.py" name="opencv_bridge" output="screen">
        <remap from="image" to="/videofile/image_raw" />
    </node>

    <!-- Lanzar el nodo que detecta los objetos-->
    <node pkg="detector" type="detector.py" name="detector" output="screen">
            <remap from="image" to ="/videofile/image_raw" />
    </node>

    <node pkg="visualizer" type="display.py" name="visualizer" output="screen">
            <remap from="image" to="/videofile/image_raw" />
            <remap from="boxes" to="/objects" />
            <remap from="lanes" to="/lane_lines" />
    </node>


    <!-- Lanzar el nodo que muestra la salida en una ventana-->
    <node pkg="image_view" type="image_view" name="image_view" output="screen">
        <remap from="image" to="/visualizer_output"/>
    </node>

</launch>
```

Figura 3.4: Fichero de configuración ROS que representa el pipeline de procesamiento en paralelo

Es necesario mencionar que este diseño no siempre es posible de conseguir y en muchas ocasiones no es trivial, ya que como vimos, los nodos en ROS se comunican a través de mensajes, pero existen tipos de datos para estos mensajes, no pudiendo ser las estructuras o tipos que se nos antojen. Esto es especialmente complicado cuando no somos nosotros los que implementamos

todo desde cero, sino que reaprovechamos módulos ya existentes, como se hace en este proyecto. A veces es muy difícil convertir la información o el módulo que usamos ni siquiera nos ofrece dicha información debido a su diseño o encapsulación. En estas ocasiones, será tarea del programador reimplementar la solución al problema que desea resolver y obtener los datos en estructuras que sea posible publicar a través de mensajes de ROS. Por otra parte, en el caso de este proyecto en particular existe un problema de sincronización. La fuente de vídeo funciona a una determinada tasa de FPS, pero los nodos detectores no son capaces de funcionar al mismo ritmo. Esto significa que cuando recibamos la información de dónde pintar el recuadro alrededor de un obstáculo, dicho obstáculo ya no estará allí. No obstante, esto es un problema del hardware que se está utilizando para realizar este proyecto, por lo que técnicamente no debería ser  un problema en un proyecto real

## 3.2 El detector de objetos

Para el detector de objetos se ha utilizando una red neuronal preentrenada de las muchas que ofrece el Model Zoo de Tensorflow.

Lo primero que debemos hacer para comenzar los preparativos estableciendo los datos y parámetros necesarios

```
14
15  import object_detection
16  from object_detection.utils import label_map_util
17  from object_detection.utils import visualization_utils as vis_util
18
19  # Definicion y configuracion del modelo, grafo, ficheros de etiquetas, etc
20  MODEL_NAME =  'ssd_mobilenet_v1_coco_2018_01_28'
21  MODEL_PATH = os.path.join(os.path.dirname(sys.path[0]),'data','models' , MODEL_NAME)
22  GRAPH_PATH = MODEL_PATH + '/frozen_inference_graph.pb'
23  LABEL_NAME = 'mscoco_label_map.pbtxt'
24  LABELS_PATH = os.path.join(os.path.dirname(sys.path[0]),'data','labels', LABEL_NAME)
25  NUM_CLASSES = 90
```

Nótese que la mayoría del código está adecuadamente comentado, por lo que no hacen falta mayores explicaciones para la mayoría de las líneas.

```
27  detection_graph = tf.Graph()
28  with detection_graph.as_default():
29      od_graph_def = tf.GraphDef()
30      with tf.gfile.GFile(GRAPH_PATH, 'rb') as fid:
31          serialized_graph = fid.read()
32          od_graph_def.ParseFromString(serialized_graph)
33          tf.import_graph_def(od_graph_def, name= '')
34
35
36  # Carga del label map
37  # El label map es un mapa que relaciona las categorias con un numero. Cuando la red predice '5',
38  # ese valor corresponde a 'avion'. No es necesario programar esta parte porque ya existen las funciones
39  labelMap = label_map_util.load_labelmap(LABELS_PATH)
40  categories = label_map_util.convert_label_map_to_categories(labelMap, max_num_classes=NUM_CLASSES, use_display_name = True)
41  categoryIndex = label_map_util.create_category_index(categories)
42
43  # Ajustes de GPU
44  # En mi caso no es necesario pero es importante e interesante saber que estan ahi
45  GPU_FRACTION = 0.4
46  config = tf.ConfigProto()
47  config.gpu_options.per_process_gpu_memory_fraction = GPU_FRACTION
```

En el caso de los ajustes de GPU, no han sido necesarios en este proyecto debido al hardware disponible y la versión de las herramientas utilizadas, pero por supuesto el conocimiento de su existencia es de vital importancia en un proyecto de esta naturaleza.

Con todo establecido, ahora sí puede comenzar la implementación de la clase. Obsérvense las características propias de ROS en el constructor de la clase. En la línea 55, por ejemplo, estamos suscribiéndonos a un tópico llamado “image”. Esta imagen es publicada por el primer nodo del pipeline, independientemente de que se trate de una cámara en directo o un vídeo. Además, véase que como parámetro podemos establecer un callback de forma que se ejecute cada vez que recibamos datos.

La línea 62 es digna de mención puesto que se trata de algo clave a la hora de trabajar con ROS y OpenCV. Las imágenes que nos llegan o publicamos a través de mensajes de ROS no son imágenes con las que podamos trabajar con OpenCV. Por ello, debemos utilizar el módulo cv_bridge para convertir esta imagen antes de comenzar a trabajar con ella. Por supuesto, una vez la hayamos procesado o modificado, justo antes de publicarla deberemos nuevamente invocar a cv_bridge para realizar el proceso inverso.

Las líneas siguientes consisten en detalles técnicos de la API de Tensorflow como se puede leer en los comentarios que las acompañan y no merecen especial mención.

```python
class Detector:

    def __init__(self):
        self.image_pub = rospy.Publisher("result", Image, queue_size = 1)
        self.object_pub = rospy.Publisher("objects", Detection2DArray, queue_size = 1)
        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("image", Image, self.image_cb, queue_size = 1, buff_size = 2**24)
        self.sess = tf.Session(graph = detection_graph, config = config)

    def image_cb(self, data):
        objectArray = Detection2DArray()
        try:
            # Recordar que es necesario el uso de cv_bridge
            cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
        except CvBridgeError as e:
            print(e)
        image=cv2.cvtColor(cv_image,cv2.COLOR_BGR2RGB)

        # Representamos y guardamos la imagen como un array. Se usara mas tarde para presentar el resultado
        # con las cajas y etiquetas
        npArrayImage = np.asarray(image)

        npArrayImageExpanded = np.expand_dims(npArrayImage, axis = 0)
        image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
        # Cada caja representa una seccion de la imagen donde un objeto fue detectado
        boxes = detection_graph.get_tensor_by_name('detection_boxes:0')
        # Obtenemos los scores o nivel de confianza de la prediccion
        scores = detection_graph.get_tensor_by_name('detection_scores:0')
        # Obtenemos la clase que tambien sera mostrada junto con el score
        classes = detection_graph.get_tensor_by_name('detection_classes:0')
        num_detections = detection_graph.get_tensor_by_name('num_detections:0')
        print("Detectando!")
```

Figura 3.5

En estas líneas es donde realmente ocurre la magia de TensorFlow. Aquí usamos la API para no solo detectar sino dibujar toda la información en la imagen.

```python

        (boxes, scores, classes, num_detections) = self.sess.run([boxes, scores, classes, num_detections], feed_dict = {image_tensor: npArr

        objects = vis_util.visualize_boxes_and_labels_on_image_array(
            image,
            np.squeeze(boxes),
            np.squeeze(classes).astype(np.int32),
            np.squeeze(scores),
            categoryIndex,
            use_normalized_coordinates = True,
            line_thickness=2)

```

Figura 3.6

La imagen quedaría lista para ser publicada y recogida por el nodo HUD o visualizador. Sin embargo, en lugar de detenernos aquí, se intentó implementar una característica añadida para permitir el diseño modular del pipeline de ROS que se explicó en la sección 3.1.2.

```python
        objectArray.detections = []
        objectArray.header = data.header
        object_count = 1

        for i in range(len(objects)):
            object_count += 1
            objectArray.detections.append(self.object_predict(objects[i], data.header, npArrayImage, cv_image))

        self.object_pub.publish(objectArray)
```

Figura 3.7

Recordemos que en la figura 3.3 hemos obtenido los objetos y los hemos almacenado en una variable homónima. En lugar de publicar la imagen, vamos a encapsular esos objetos en una estructura de datos adecuada y publicarlos para que sean recogidos por el nodo visualizador. De esta forma no enviamos la imagen, sino únicamente la información que hay que pintar sobre ella y sería el nodo visualizador quien realizaría este último paso. En la figura siguiente se detalla la implementación de la función utilizada en la línea 107. Como curiosidad, obsérvese en la línea 109 lo simple que es publicar un mensaje con ROS.

```
123     def object_predict(self, object_data, header, npArrayImage, image):
124         image_height, image_width, channels = image.shape
125         obj=Detection2D()
126         obj_hypothesis= ObjectHypothesisWithPose()
127 
128         object_id=object_data[0]
129         object_score=object_data[1]
130         dimensions=object_data[2]
131 
132         obj.header=header
133         obj_hypothesis.id = object_id
134         obj_hypothesis.score = object_score
135         obj.results.append(obj_hypothesis)
136         obj.bbox.size_y = int((dimensions[2]-dimensions[0])*image_height)
137         obj.bbox.size_x = int((dimensions[3]-dimensions[1] )*image_width)
138         obj.bbox.center.x = int((dimensions[1] + dimensions [3])*image_height/2)
139         obj.bbox.center.y = int((dimensions[0] + dimensions[2])*image_width/2)
140 
141         return obj
142 
143 def main(args):
144     rospy.init_node('detector_only')
145     obj=Detector()
146     try:
147         rospy.spin()
148     except KeyboardInterrupt:
149         print("Exit")
150     cv2.destroyAllWindows()
```

Figura 3.8

Esta función en realidad no posee ninguna línea digna de especial mención. Como se puede observar, tan sólo almacena y deja la información lista para ser enviada. A modo de curiosidad, obsérvese en la línea 141 como ROS necesita un método main, pero también su simplicidad.

## 3.3 El nodo visualizador

El nodo visualizador se encarga de recibir la información y enviarla al módulo de ROS que abre una ventana del sistema y muestra la imagen.

```
17  class image_converter:
18
19      def __init__(self):
20          self.image_pub = rospy.Publisher("visualizer_output",Image)
21
22          self.bridge = CvBridge()
23          self.image_sub = rospy.Subscriber("image", Image, self.callback)
24
25          self.boxes_sub = rospy.Subscriber("boxes", Detection2DArray, self.drawBoxes, queue_size = 1)
26
27      def callback(self,data):
28          try:
29              cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
30          except CvBridgeError as e:
31              print(e)
32
33          try:
34              self.image_pub.publish(self.bridge.cv2_to_imgmsg(cv_image, "bgr8"))
35          except CvBridgeError as e:
36              print(e)
```

Figura 3.9

Véanse las primeras líneas del constructor, donde se establece que la imagen final será publicada y donde el nodo es suscrito a la imagen (línea 23) y a la información de la detección (línea 25). Obsérvese la función de callback en la línea 27, donde se observa el proceso completo del uso del módulo cv_bridge ya mencionado con anterioridad. Existe una función que trataba de pintar la información recibida sobre la imagen para lograr el diseño modular mencionado en la sección 3.1.2, aunque su implementación no es trivial y finalmente no pudo ser conseguida debido a la falta de tiempo. No obstante, esto no altera en absoluto las líneas vistas en la figura anterior y la imagen será publicada al nodo final de la misma forma, se haya obtenido con la información dibujada en ella o se realice ese proceso en este nodo.

## 3.4 El detector de carriles

### 3.4.1 Aproximaciones manuales

Debido al miedo de enfrentarnos a grandes costes de rendimiento, se consideró la idea de optar por una aproximación basada en visión técnicas de visión por computador tradicional con OpenCV.

Dado que un vídeo no es más que una sucesión de imágenes, este problema se abordó utilizando una imagen estática (concretamente un frame de uno de los vídeos utilizados en el proyecto). De esta forma se puede controlar mejor cada paso. Más tarde simplemente hay que realizar unas pequeñas adaptaciones teniendo en cuenta que el programa va a ser en realidad un nodo de ROS, pero esto solamente afecta a la forma en que obtenemos los datos al principio, siendo igual el resto del código.


Figura 3.10: Imagen de muestra utilizada para comenzar el desarrollo.

Para abordar el problema debemos comenzar por preguntarnos por qué nosotros como humanos, sabemos que lo que vemos son las líneas del carril. En nuestro caso, al utilizar una dash-cam que estará siempre en la misma posición, podemos asumir que las líneas del carril casi siempre estarán en una determinada región de la imagen. Además, casi siempre tendrán una forma característica, siendo más anchas en la parte más cercana y alejándose formando casi un triángulo sin llegar nunca al horizonte. Por ejemplo, no tiene sentido que intentemos detectar las líneas del carril en el cielo o en un lateral como si se encontrasen en una pared. Por ello, como punto de partida, se procede a recortar la imagen y quedarnos con una región de interés.

Para calcular la región de interés, debemos pensar en una figura geométrica que podamos aplicar a la imagen y que nos devuelva un resultado que contenga la información que deseamos. Dada la explicación del párrafo anterior, un triángulo con vértices en las esquinas inferiores y aproximadamente el centro de la imagen, debería comprender la región que nos interesa. Es preciso recordar que normalmente en el mundo de los gráficos, el origen se encuentra en la esquina superior izquierda de la pantalla.

```
38
39          RegionOfInterestVertices = [(0, height), (width / 2.1, height / 1.6), (width, height),]
40
```

Figura 3.11

Ahora, necesitamos una función que dados unos vértices, nos devuelva la región de la imagen que queda comprendida en la figura que dibujan.

```
59
60      #Vertices de la region que nos interesa
61      def getRegionOfInterest(self, img, vertices):
62          mask = np.zeros_like(img)
63          match_mask_color = 255
64          cv2.fillPoly(mask, vertices, match_mask_color)
65          masked_image = cv2.bitwise_and(img, mask)
66          return masked_image
```

Figura 3.12

Hecho lo anterior, solo nos queda llamar a nuestra función adecuadamente en el código y observar el resultado:

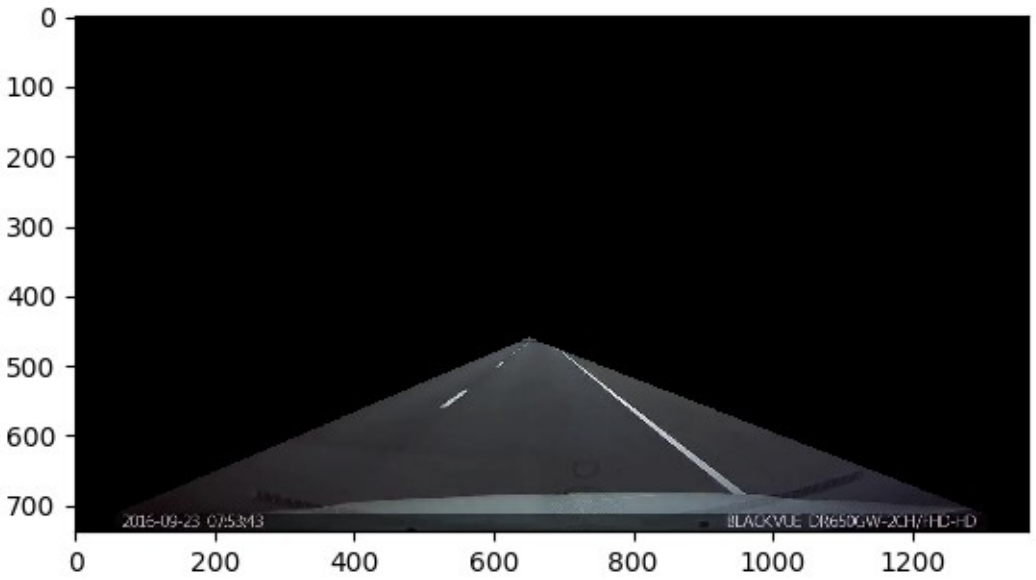



Figura 3.13: Imagen después de ser recortada

Con la imagen obtenida, hemos descartado mucha información que no era necesaria. El siguiente paso será utilizar esta imagen para realizar una detección de bordes. Aunque OpenCV cuenta con muchas funciones que evitan que tengamos que dominar ampliamente algunos conceptos matemáticos que se esconden detrás del tratamiento de imágenes, si es necesario que tengamos unas nociones básicas sobre lo que necesitamos y queremos hacer.

Si prestamos atención a la imagen, saber cómo debemos pensar para reconocer un borde no es demasiado complejo: un borde es un área donde los valores de color cambian muy bruscamente. Así, detectar bordes en una imagen se reduce a un problema matemático que consiste en detectar áreas donde existen cambios bruscos en el valor de los píxeles con respecto a sus vecinos. Este problema es relativamente fácil de abordar y por suerte, existe un famoso algoritmo inventado por John F. Canny que además se encuentra

entre las funciones incluídas en OpenCV. El algoritmo de detección de bordes de Canny básicamente detecta áreas de la imagen que poseen un fuerte gradiente en la función de color de dicha imagen. Al igual que con muchos otros algoritmos relativos al campo del tratamiento de imágenes, es habitual trabajar con parámetros que sirven de umbral.

En realidad para resolver este problema, no necesitamos utilizar información relativa al color. Por ello, para simplificar ligeramente el proceso de detección se convertirá la imagen a escala de grises. El algoritmo de Canny se limitará a encontrar áreas donde existan cambios bruscos en la intensidad de los píxeles.

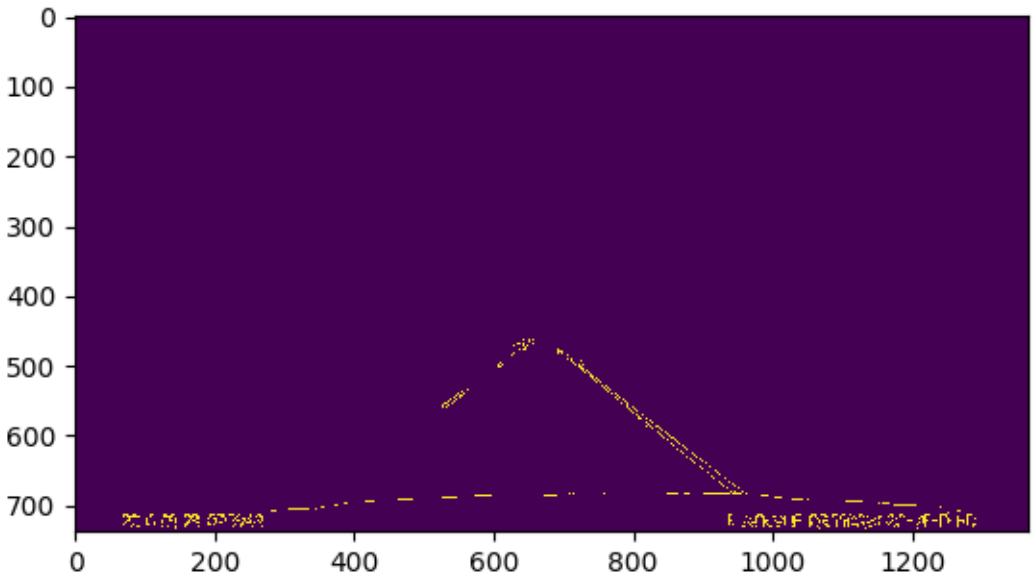



Figura 3.14

Ahora sí, se puede proceder a detectar las líneas, y para ello nuevamente es necesario hablar de matemáticas.

En términos de visión por computador, una línea es una estructura que comparte muchos puntos alineados en común. Dicho de otra forma, si trazamos todas las líneas posibles que pasan por un determinado pixel, existirá un conjunto de píxeles que compartirán la misma línea. Si este número de píxeles es significativo seguramente nos encontremos ante una línea. En resumen, nuestro problema es determinar líneas que compartan múltiples píxeles que representan bordes al mismo tiempo. Sin embargo, existe un número muy elevado de líneas que pasan por un determinado pixel, por lo que resulta inviable comprobar todas las líneas de todos los puntos y si existen a su vez otros puntos que comparten estas líneas. Por suerte, este problema también está matemáticamente resuelto y la forma de hacerlo es a través de la llamada transformada de Hough.

Dado que la transformada de Hough es ampliamente usada en el mundo de la visión por computador para resolver este y otros problemas, OpenCV incorpora su implementación. Únicamente debemos preocuparnos de ajustar

```
41            #Pasamos la imagen a escala de grises
42            grayImage = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
43            #Aplicamos el algoritmo de Canny para detectar bordes
44            cannyed_image = cv2.Canny(grayImage, 100, 200)
45            #Recortamos la imagen
46            croppedImage = self.getRegionOfInterest(cannyed_image, np.array([RegionOfInterestVertices], np.int32), )
47
48            #Utilizamos la transformada de Hough para obtener las lineas que necesitamos
49            lines = cv2.HoughLinesP(croppedImage, rho=6, theta=np.pi / 60, threshold=160, lines=np.array([]), minLineLength=40,
50
```

Figura 3.15

los parámetros (línea 46).

Por último, tan solo necesitamos una función que dadas las líneas, las pinte sobre la imagen, algo que no merece especial mención.

```
68      #Este metodo dibuja las lineas que recibe como parametro sobre la imagen
69      def drawLines(self, img, lines, color=[255, 0, 0], thickness=3):
70          if lines is None:
71              return
72          img = np.copy(img)
73          line_img = np.zeros(
74              (
75                  img.shape[0],
76                  img.shape[1],
77                  3
78              ),
79              dtype=np.uint8,
80          )
81          for line in lines:
82              for x1, y1, x2, y2 in line:
83                  #La linea es valida unicamente si supera una cierta pendiente
84                  slope = (y2 - y1) / (x2 - x1)
85                  if math.fabs(slope) > 0.5:
86                      cv2.line(line_img, (x1, y1), (x2, y2), color, thickness)
87          img = cv2.addWeighted(img, 0.8, line_img, 1.0, 0.0)
88          return img
```

Figura 3.16

Al tratarse de una aproximación manual, a partir de este punto el límite es el tiempo disponible, ya que siempre podrán encontrarse nuevos casos y mejoras. Una muy sencilla, es no considerar aquellas líneas demasiado horizontales, es decir, cuya pendiente forme un cierto ángulo menor que cierto umbral con respecto al eje horizontal.

No nos quedaremos con una sola línea por cada lado del carril, porque en el futuro podría desearse implementar una línea trazada a partir de todas las anteriores. Si no se hace de esta forma, en carriles con líneas discontinuas únicamente se detectan dichas líneas, pero no existe un trazo continuado que delimite el carril.

### 3.4.2 Aproximación por redes neuronales (LaneNet)

Debido a diversos problemas, la implementación de esta aproximación resultó ser mucho más complicada de lo esperado y las soluciones a estos problemas no resultan triviales, ya que tienen que ver con el funcionamiento de TensorFlow, el modelo y ROS conjuntamente.

No obstante, pese a no poderse probar el rendimiento en cuanto a tasa de FPS, sí se dispone de dichos datos por fuentes externas y además, sí es posible

realizar una comparación basada en imágenes estáticas. El tiempo de procesamiento para cada imagen fue de una media de 2.2 segundos. Los resultados como se puede ver son mucho más precisos, detectando incluso cerradas curvas de giros en cruces, como se muestra en la siguiente imagen.

La detección responde bien en todas las situaciones e incluso predice el rumbo de la carretera aunque un vehículo impida la visión del carril, como se observa en la última imagen (de muestra ya proporcionada en el repositorio)

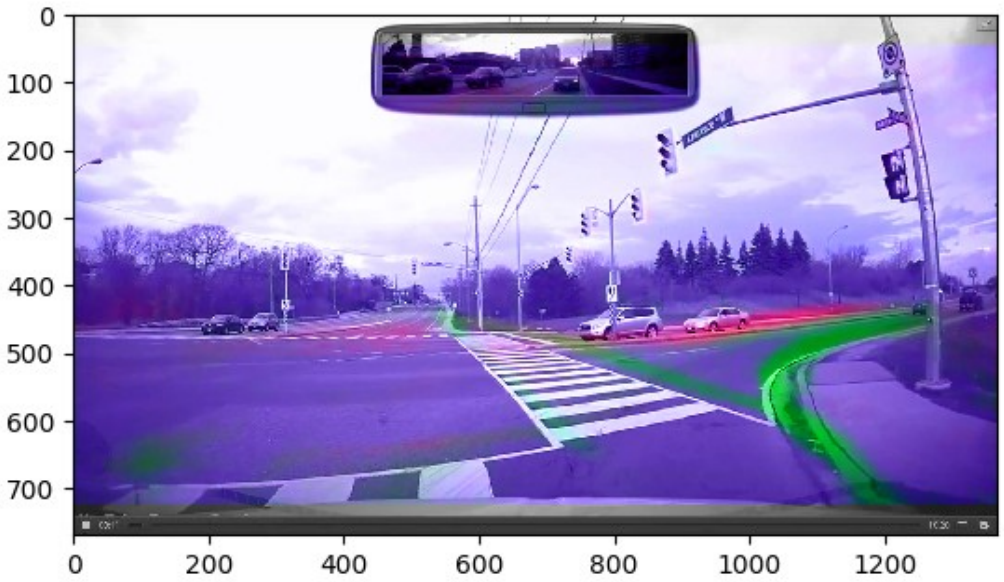



Figura 3.17

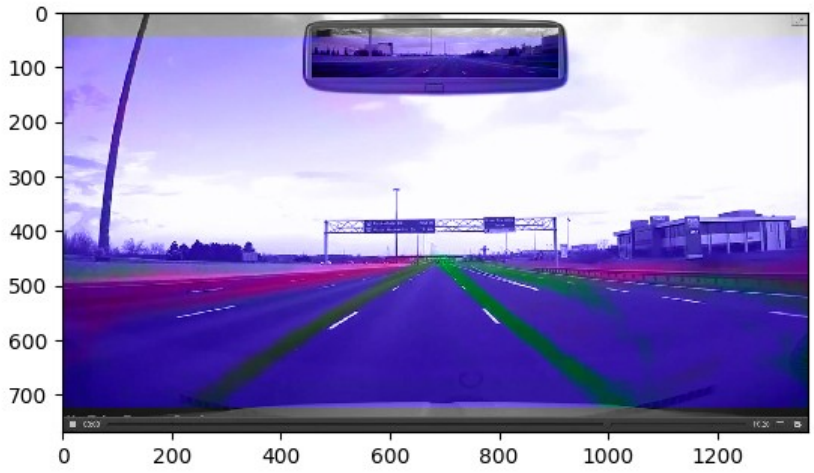



Figura 3.18

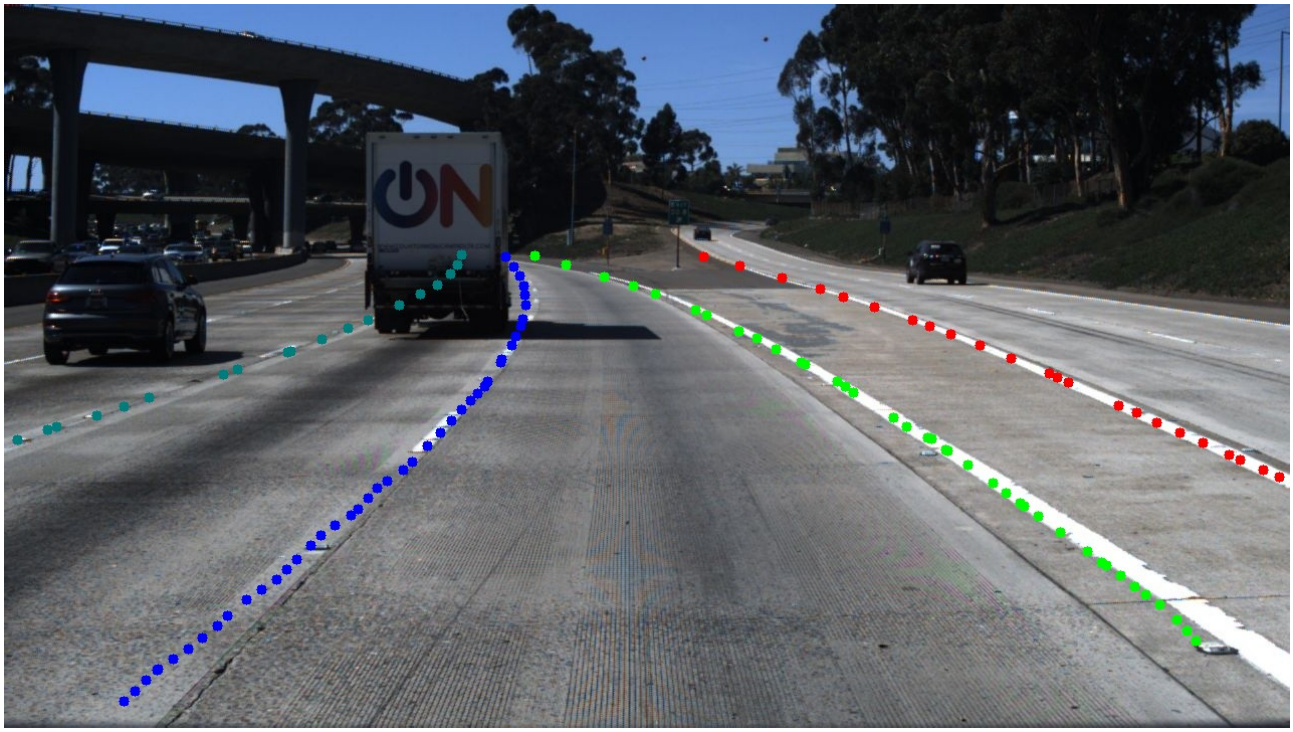


Figura 3.19: Como se observa, la red es capaz de intuir la línea de carril que avanza a la izquierda de la azul.

## 3.5 Resultados

Una vez todo listo, tan solo debemos cambiar la ruta para probar diferentes modelos. A continuación se detalla en una gráfica los diferentes modelos probados y su rendimiento. Para obtener dicha gráfica, se ejecutó el sistema durante veinte segundos, recogiendo en todo momento los FPS obtenidos en tiempo real. A continuación, un simple script de Python obtenía la media de estos datos. Normalmente, la medida instantánea de FPS no distaba demasiado de la media, pero con este proceso nos aseguramos no medir justo en una caída de la frecuencia y obtener así unos datos más consistentes.

### 3.5.1 Tasa de FPS

Para calcular la tasa de FPS se utilizó el comando *rostopic hz*, que permite conocer la frecuencia de un determinado tópico. Para evitar sesgar los datos con caídas puntuales de FPS, se optó por ejecutar el sistema durante 30 segundos y desviar la salida del comando anterior a un fichero. Un sencillo script procesa el fichero de salida y calcula los FPS medios de todo el tiempo de ejecución. Cabe decir que este tiempo medio no se desvía demasiado de las medidas tomadas por separado, dado que el sistema suele funcionar casi siempre a un ritmo constante.

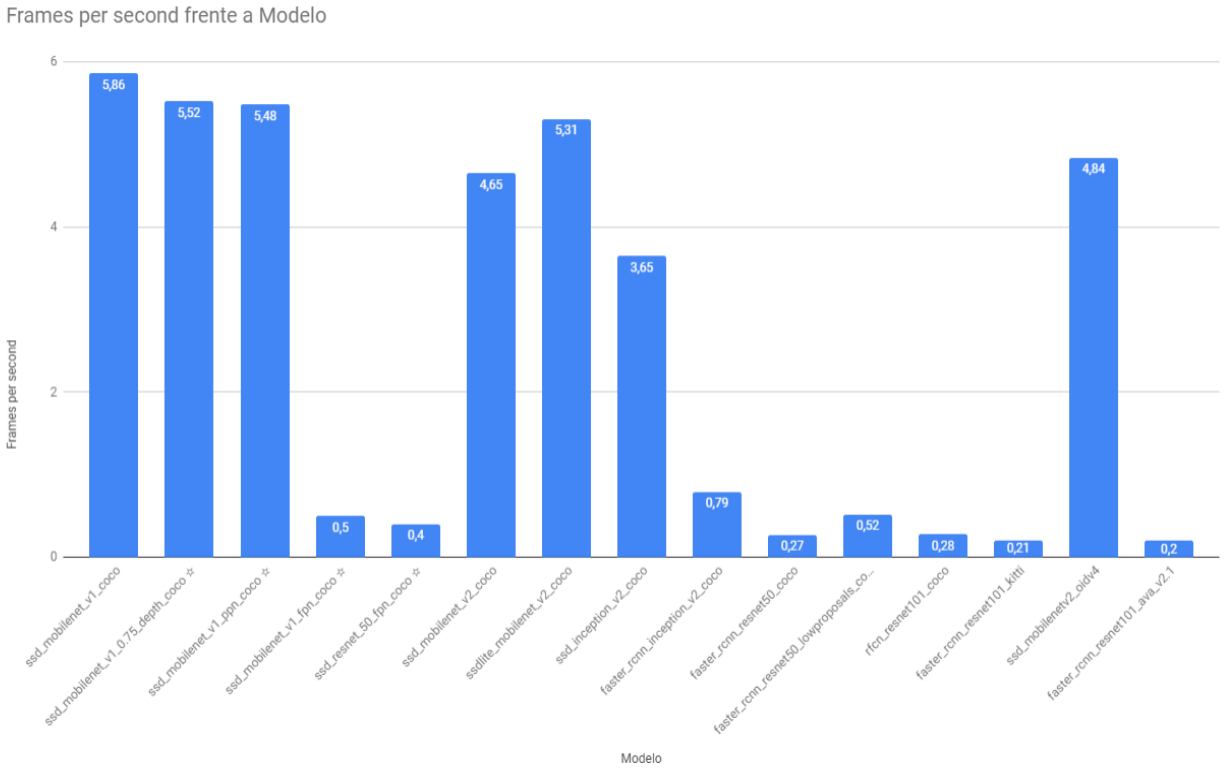



Figura 3.20: Tasa de FPS media para cada modelo. Los modelos que muestran una estrella en su nombre soportan TPU. Pruebas realizadas en un i3 2.0GHz / 4GB RAM

### 3.5.2 Tasa de precisión

Los datos obtenidos experimentalmente vistos en la figura anterior, en realidad coinciden casi a la perfección con los suministrados por el repositorio de TensorFlow. En vista de esto y debido a la laboriosidad de la tarea junto con la falta de tiempo, para realizar el estudio de la tasa de precisión se utilizaron los datos ya suministrados por el repositorio.

Antes de presentar estos datos, conviene explicar brevemente su significado. Normalmente, el indicador utilizado se denota como mAP y proviene del inglés “mean Average-Precision”. Esto mide la tasa de acierto y precisión del modelo sobre un conjunto de datos. Para realizar esta medición se conoce un concepto abreviado como IoU, que proviene de Intersection over Union. Este proceso es importante porque permite establecer un umbral de lo que se considera un acierto o un error. Por ejemplo, poniendo un ejemplo extremo, si una imagen contiene un coche y el modelo dibuja un recuadro que comprenda toda la imagen, está claro que técnicamente es cierto que su detección contiene el coche, pero esto, obviamente, no puede ser válido

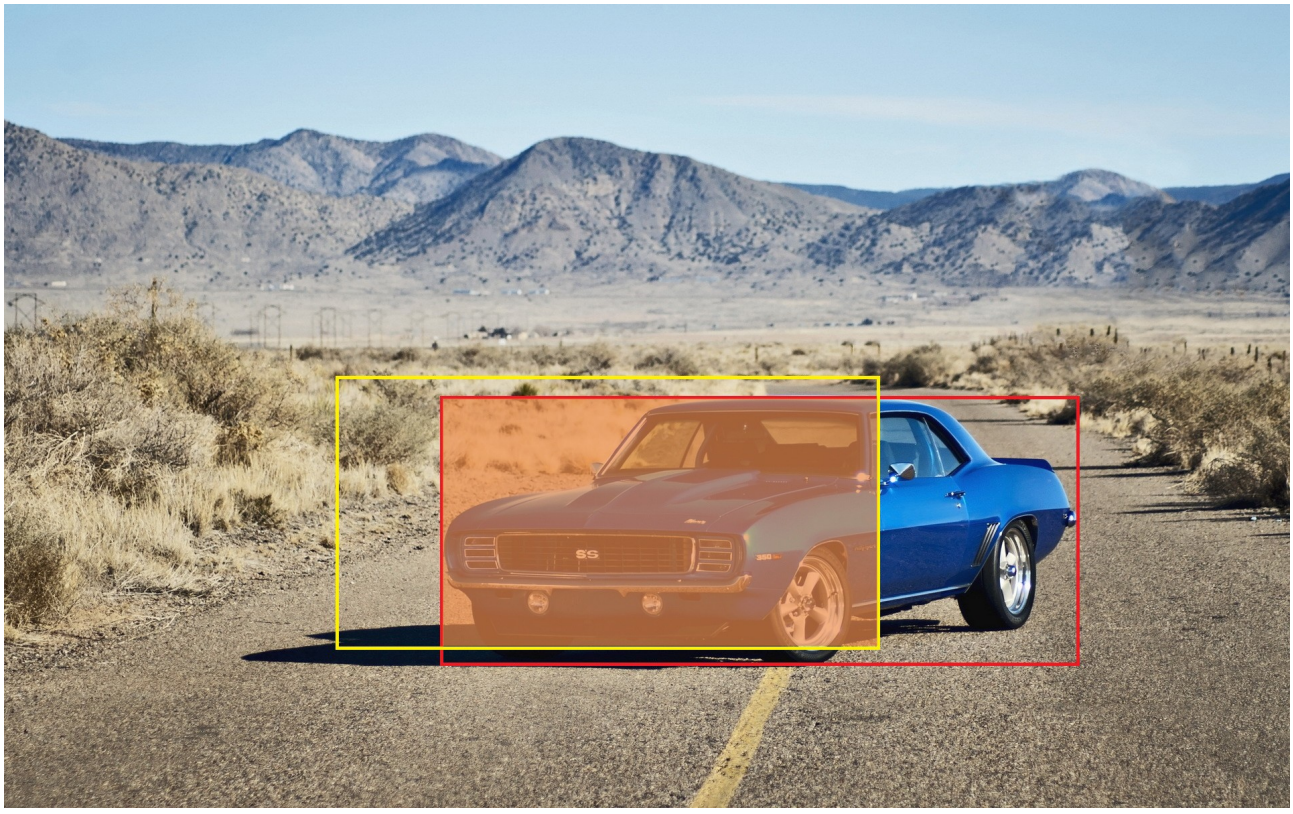


Figura 3.21: Definición de IoU

El concepto de IoU, como puede observarse en la imagen anterior es muy intuitivo, y es el resultado de calcular el cociente entre el área total resultante de la unión de los dos recuadros y el área superpuesta (en naranja). Donde el

recuadro rojo contiene el coche y el recuadro amarillo es la predicción que ha hecho el modelo.

$$IoU = \frac{Área\ de\ la\ intersección}{Área\ de\ la\ unión}$$

Normalmente, se considera que la predicción es correcta si la IoU es $\geq 0.5$ y se considera la predicción como lo llamado un falso positivo en caso contrario. Así, se define la precisión como el cociente entre las predicciones correctas y el número de predicciones totales incluyendo los falsos positivos.

$$Precisión = \frac{Positivos\ verdaderos}{Positivos + Falsos\ positivos}$$

Atendiendo a los resultados disponibles, en la gráfica mostrada a continuación se exponen los diferentes modelos utilizados y su correspondiente valor de mAP

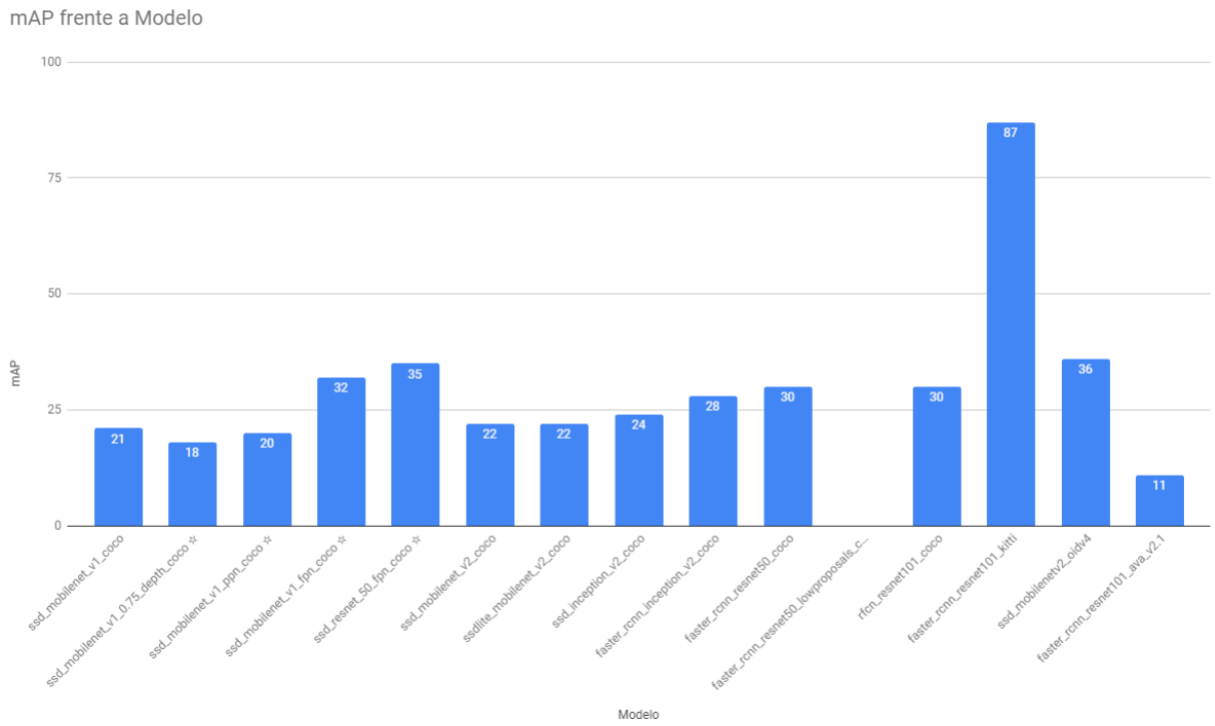



Figura 3.22: mAP para cada modelo. Obsérvese que hay un modelo para el cual no existen datos.

Con los datos anteriores, se puede estudiar la decisión de utilizar un modelo

u otro. Un primer acercamiento puede ser establecer un ratio entre tasa de FPS y tasa de detección, produciendo la gráfica resultante que se muestra en la siguiente imagen. Por supuesto, este método no es la respuesta, ya que considera igual de importantes ambas tasas y esto no tiene por qué ser cierto. No obstante, fue satisfactorio ver el resultado obtenido, ya que empíricamente, cuando se probaron los distintos modelos y sin recoger los datos, se tuvo la sensación de que el modelo *ssd_mobilenet_v1_ppn_coco* era el que presentaba un mejor equilibrio entre detección y rendimiento, y de hecho este modelo resulta aparecer como la segunda mejor opción en la gráfica.

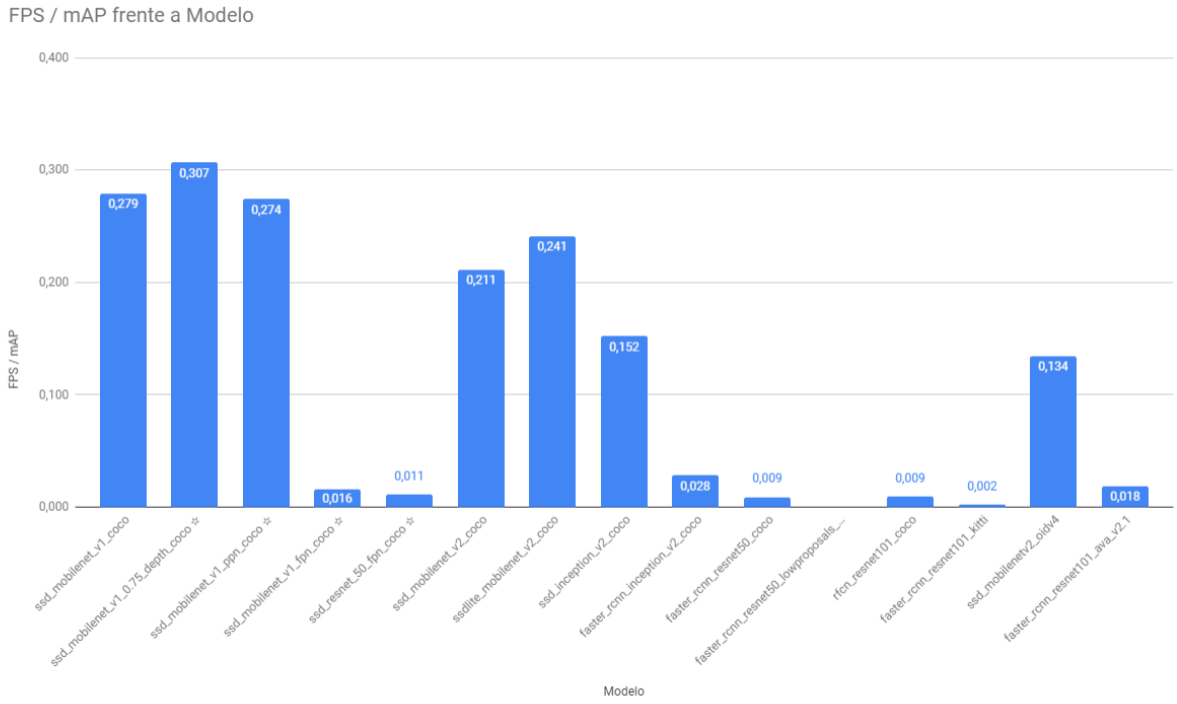



Figura 3.23: Ratio FPS / mAP

A la vista de los resultados, se decidió utilizar el modelo con mayor puntuación. El resultado final se muestra a continuación y como se puede observar, logra una buena detección incluso a unas distancias o situaciones de baja iluminación considerables. La tasa de FPS del sistema final ronda los 5 FPS, una medida nada despreciable teniendo en cuenta los bajos recursos del sistema donde se ejecuta, que ni siquiera cuenta con GPU, algo vital en los sistemas de este tipo.

En la gráfica que se muestran los mismos datos pero representados de otra forma. En el eje X se muestran los FPS y en el eje Y el mAP. Si tuviésemos un caso donde FPS = 0.1 y mAP = 0.1, y otro caso donde FPS = 1 y mAP = 1, en ambos casos el ratio FPS / mAP sería 1 en ambos casos, cuando el segundo caso es claramente mejor. La representación que se muestra a continuación ayuda a visualizar mejor los datos obtenidos.

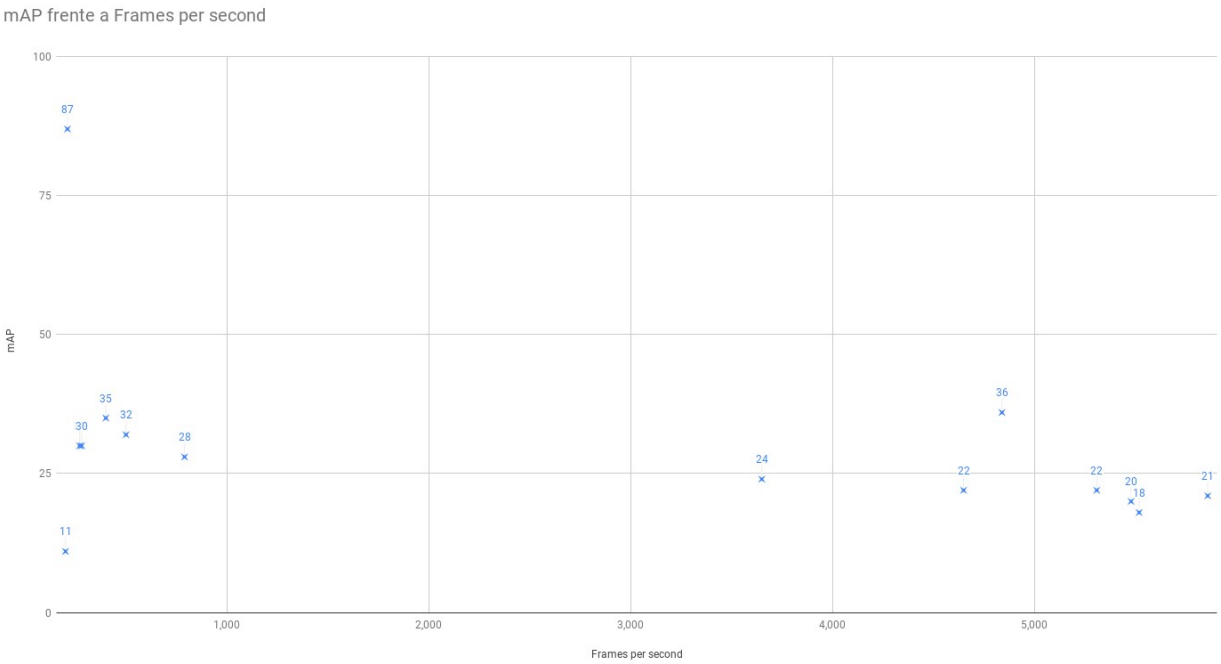



Figura 3.24

Por supuesto, esto no quiere decir que sea la elección más aconsejable. En la figura 3.22 se puede observar la alta tasa de acierto del mejor modelo, que además ha sido entrando sobre el dataset *Kitti* especialmente pensado para entornos de carretera, y si bien su rendimiento necesita muchos más recursos, esto no es un problema existiendo el hardware adecuado.


Figura 3.25: Nótese que la detección presenta una calidad y velocidad significativas, siendo detectado sin problema un coche que acaba de aparecer por un extremo de la imagen y en una zona oscura. Recordemos que la detección de carril se basa en la detección de bordes y sufre problemas en desniveles y curvas pronunciados, pero en situaciones controladas su desempeño no es despreciable como punto de partida.

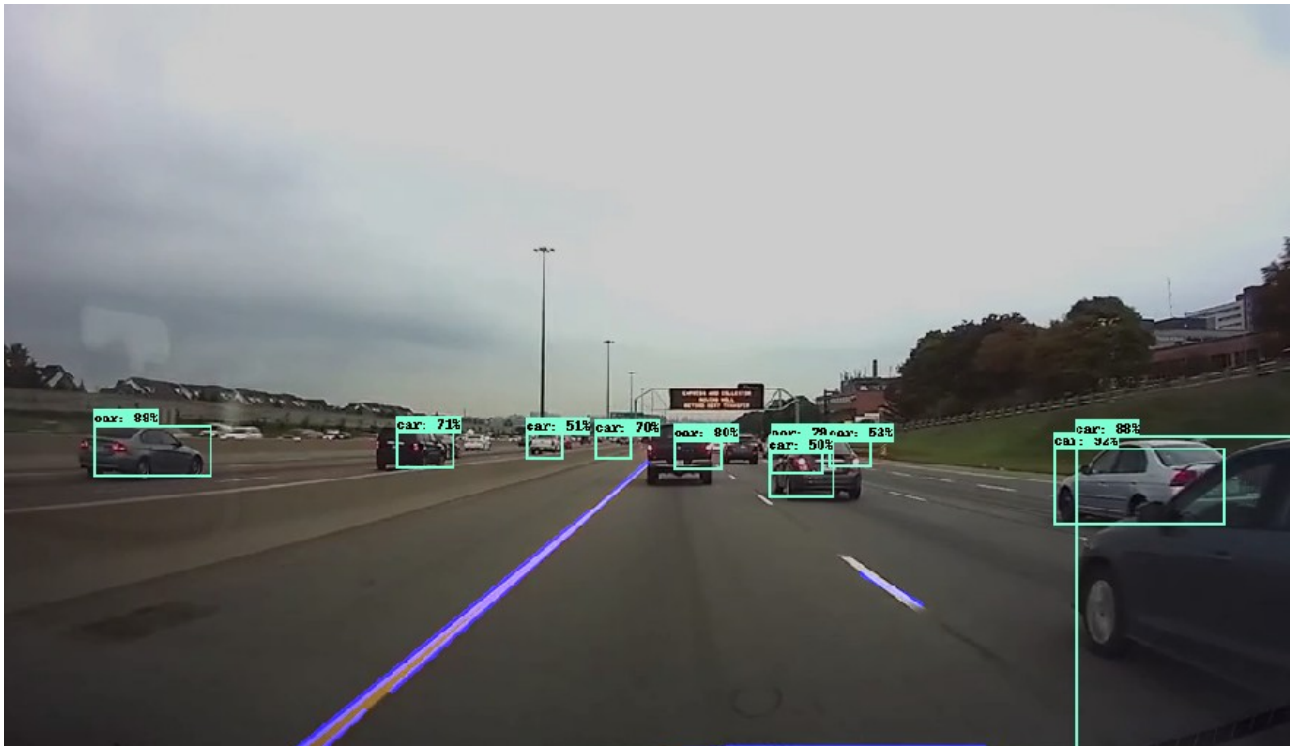



Figura 3.26: La detección de carril es problemática si hay un vehículo excesivamente cerca, sobre todo cuando es de un color que resalta con la carretera, pero como se puede apreciar en esta imagen, el resultado en una situación normal es bueno. Obsérvese que todos los vehículos son detectados.

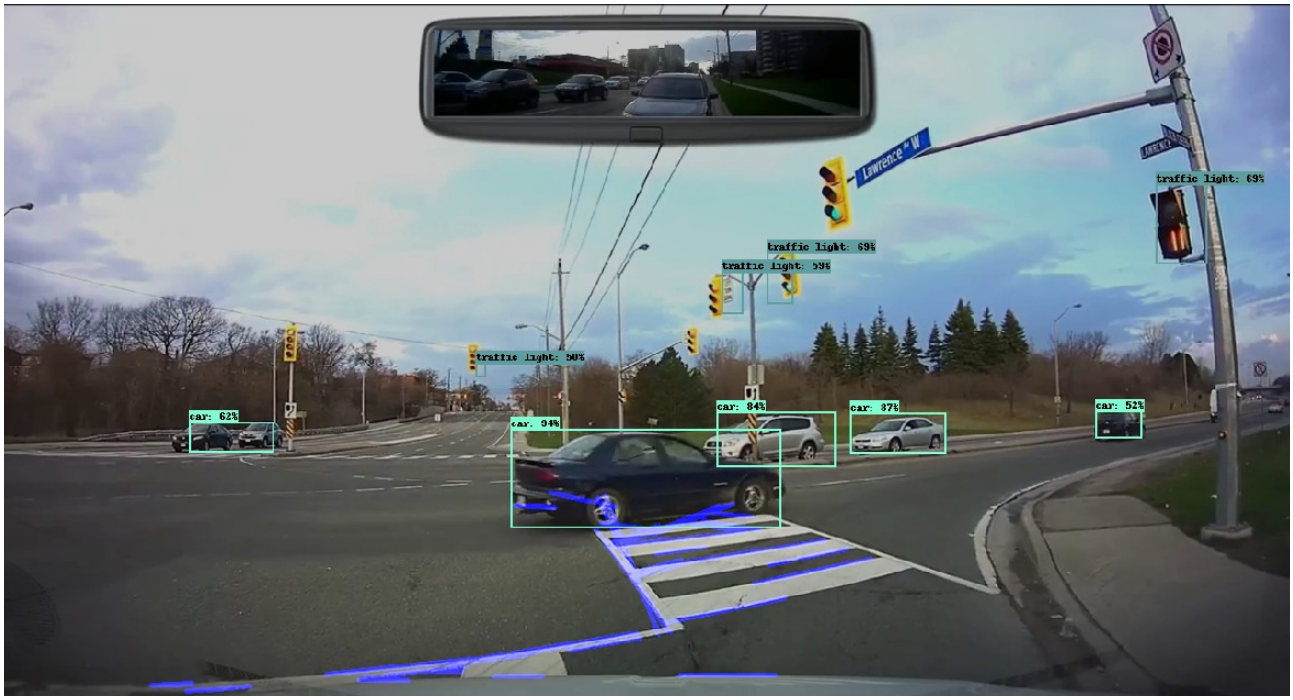



Figura 3.27: Una situación algo sobrecargada donde tenemos diferentes vehículos, líneas en el pavimento y semáforos. Obsérvense los errores en el detector de carril sobre el coche que hay delante.

# Capítulo 4
# Conclusiones y líneas futuras

## 4.1 Conclusiones

En general, el proyecto ha sido muy instructivo y me ha servido para adentrarme en los principios básicos de esta rama de la informática, la cual desconocía.

Como se mencionó en el capítulo de introducción, en un proyecto de estas características nos enfrentamos no solo al reto de que el sistema funcione, sino que al tratarse de un sistema de tiempo real en un entorno donde están en juego vidas humanas, se debe tener un especial cuidado en todos los niveles del desarrollo y la forma de hacer las cosas en este campo de la ingeniería suele distar de lo tradicional y lo estudiado en el ámbito académico. Por ello, este trabajo pretendía tener un carácter instructivo y experimental. Existe el hardware (y su coste no es desmesurado) para que este proyecto funcione con un gran rendimiento sin preocuparnos de nada, pero se pretendía estudiar hasta que punto han avanzado las tecnologías y qué resultados se podían obtener con recursos caseros de muy bajo costo y siempre basados en software libre. Creo que la verdadera conclusión del trabajo es el hecho de que en muy poco tiempo y con un equipo personal con bajos recursos, se ha conseguido realizar un sistema que funciona relativamente bien. En mi opinión, es un indicador de que el futuro toma este camino y de hecho esto es tan seguro, que como hemos visto, hace tiempo que no es futuro sino presente.

Respecto a las herramientas utilizadas, el mayor descubrimiento ha sido ROS, no solo debido al enorme abanico de funcionalidades y posibilidades que ofrece, sino a su gran modularidad y la posibilidad de lograr un paralelismo entre sistemas funcionando al unísono tan cómodamente. Sin duda es una herramienta a tener en cuenta en cualquier proyecto de estas características.

En el ámbito de la detección, me ha resultado muy instructiva la comparativa de las aproximaciones manuales y las redes neuronales. Las aproximaciones manuales requieren grandes esfuerzos de programación y se tiene la sensación de que siempre habrá un caso particular más que no se ha contemplado, por mucho tiempo que se invierta. Por otro lado, su consumo de recursos tampoco es que sea exageradamente menor que un enfoque basado en redes neuronales, al menos en el tema que nos ocupa. Por ello, teniendo en

cuenta una relación entre coste, esfuerzo, rendimiento y resultados, existiendo las redes neuronales, es inevitable preguntarse si merece la pena plantearse un enfoque tradicional.

## 4.2 Líneas futuras

Como líneas futuras de trabajo en el proyecto, me parece de vital importancia la consecución de que los diferentes nodos que representan diferentes sistemas trabajen en paralelo y tener en cuenta siempre este esquema de trabajo, ya que esto permite que el sistema continúe funcionando con normalidad ante el fallo eventual de un nodo en particular. Creo que este diseño es la base sobre la que siempre debe sustentarse el proyecto, independientemente de cuantas funcionalidades se añadan.

Otra línea de trabajo que encuentro sumamente interesante y en caso de conseguirse abriría todo un nuevo abanico de trabajo es el cálculo de distancias. Para abordar este problema existen diversas técnicas, aunque con una sola cámara no lograremos un resultado fiable y esto no puede ser en un sistema donde debe primar la seguridad. Debemos tener en cuenta que introducir una cámara estéreo podría requerir replantearnos los recursos necesarios para el proyecto. Actualmente, muchos equipos de investigación y desarrollo líderes en el sector utilizan una sola cámara que se apoya en otros recursos de hardware y software para calcular la corrección de la lente, el desnivel y ángulo de la carretera y otros factores, para finalmente calcular la distancia de forma fiable.

En el anteproyecto de este trabajo también se presentó como valor añadido el entrenamiento de la red neuronal con señales europeas, para que reconociera las mismas. Es una característica fundamental para un sistema como este y su implementación es sencilla, pero se necesitan recursos para realizar el entrenamiento.

# Capítulo 5
# Summary and Conclusions

In general, I've found the project very instructive and it helped me to get into the basics of this computer science branch.

As mentioned in the introduction chapter, we face also the challenge of making a real time system where human lives are compromised. This kind of systems are complex and their design and development usually distracts from the traditional and studied in the academic life. Therefore, this project was intended to be instructive and experimental. It exists the hardware for making this project work without worrying about performance, but the intention was to check what could be achieved with very low cost and open source based resources. Personally, I believe that the main conclusion of this work is the fact that in a very shot time and with very low resources, a system that works relatively well was achieved. In my opinion this is an indicator of the huge possibilities that the future can offer to us.

Regarding the tools used for the project, my personal greatest discovery was ROS, not only due to its enormous amount of functionalities and possibilities but also for its great modularity, allowing to build parallel systems very easily. It is certainly a tool to consider for any project of this nature.

In the field of object detection, the comparison between neural networks and manual approaches has been very instructive. Manual approaches are difficult to become generic and there are always particular cases that were not contemplated by the programmer, no mtter how much time is spent. On the other hand, resource consumption of neural networks is not non-viable. Therefore, in my opinion, taking into account a relationship between cost, effort, performance and results, it is hard to think in a good reason for considering traditional approaches over neural networks.

As future lines of work in the project, I found very important to achieve a parallel design with ROS nodes, since this allows the system to continue working in case of an eventual failure of a particular node. I believe this behaviour must be the foundation of a project like this.

I also find extremely interesting achieving distance calculation. This would open a whole new range of possibilities. There are several techniques for facing this problem, although a single camera will not give us a reliable result and this is not valid in an environment where security must prevail. A common solution is using a stereo camera but it has a resources cost. Currently, many industry-

leading research and development teams use  single camera that relies on other hardware and software resources to calculate lens correction, slope and angle of the road and other factors, to finally calculate the distance reliably.

In the preliminary draft of this work, the training of the neural network with European signals was also presented as an added value. It is a fundamental characteristic for a system like this and its implementation is simple, but resources are needed to carry out the training.

# Capítulo 6
# Presupuesto

A continuación se detalla lo que podría ser un presupuesto aproximado del proyecto.

## 6.1 Sección Uno

| Tipos | Descripción |
|---|---|
| Dash-cam 1080p | 79,90€ |
| Nvidia Jetson AGX Xavier | 729,90€ |
| **TOTAL** | 809,80€ |

**Tabla 6.1:** Resumen de tipos

# Capítulo 7
# Anexo

## 7.1 Repositorio del código

Con el objeto de no sobrepasar el límite de páginas establecido para la presente memoria y para una lectura más cómoda, en lugar de adjuntar el código en este anexo, se ofrece simplemente el enlace al repositorio donde reside el mismo:

https://github.com/alexrcas/ROSdashCam

Latest commit `f61eb72`

# Bibliografía


https://www.smithslawyers.com.au/post/self-driving-car-problems

https://www.hisour.com/es/computer-vision-42799/

http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

https://www.xataka.com/robotica-e-ia/las-redes-neuronales-que-son-y-por-que-estan-volviendo

https://medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec

https://blogs.nvidia.com/blog/2019/06/19/drive-labs-distance-to-object-detection/

https://arxiv.org/abs/1807.01726

https://github.com/MaybeShewill-CV/lanenet-lane-detection