



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto fin de Grado

Grado en Ingeniería Informática - Tecnologías Informáticas

Sistema integral de ayuda a la conducción con sistemas empotrados de bajo coste

**Realizado por
Juan Arteaga Carmona**

**Dirigido por
Francisco Luna Perejón
Manuel Jesús Dominguez Morales**

**Departamento
Arquitectura y tecnología de computadores**

Sevilla, 21 de septiembre de 2020

Abstract

Driving assistance systems are one of the most important systems in today's vehicles. They've become an important consideration for clients when choosing a new vehicle, as they want a secure vehicle that keeps their families safe, as well as for the companies that create them because this systems are a required step in order to eventually achieve full self-driving.

However, this type of systems are usually only found in luxury vehicles or they are behind a paywall.

Because of this, in this project we describe the design and implementation of an integral driving assistance system based on embedded systems, computer vision and artificial intelligence models that is relatively easy to install in any type of vehicle.

The main objective is to be able to warn the driver, acoustically and/or visually, when we detect any type of anomaly.

In order to meet this objective we've developed a piece of Python software that allow us to connect to a simulator and obtain information about the interior and exterior of the vehicle thanks to the use of a few types of simulated hardware sensors, process this information and analyze the current and future state of the vehicle.

Resumen

Los sistemas de ayuda a la conducción son unos de los más importantes en los vehículos de hoy día. Se han convertido en un punto importante a tener en cuenta tanto para los usuarios finales, que desean adquirir vehículos seguros que mantengan protegidos a sus familias, como para las propias empresas que los crean, puesto que es un paso intermedio necesario para la eventual transformación a la conducción autónoma.

Sin embargo, este tipo de sistemas suelen estar restringidos a modelos de alta gama o se encuentran tras un *paywall* bastante considerable.

Es por esto que en este proyecto se describe el diseño e implementación de un sistema integral de ayuda a la conducción basado en sistemas empotrados, visión por computador y modelos de inteligencia artificial que puede ser incluido en cualquier vehículo con relativa facilidad.

El objetivo principal es conseguir detectar y avisar al conductor ante estados anómalos de forma acústica y/o visual.

Para cumplir con este objetivo se ha desarrollado una aplicación en Python que nos permite conectarnos a un simulador y obtener información sobre el exterior e interior del vehículo gracias al uso de varios tipos de sensores *hardware* simulados, procesarla y realizar un análisis del estado actual y futuro del vehículo.

Agradecimientos

No puedo continuar con este trabajo sin antes agradecer a todas las personas que me han ayudado durante cada uno de los días de mi etapa universitaria. Desde los profesores que han hecho que sus clases sean tan interesantes como cautivadoras, hasta todas esas nuevas personas a las que tengo el placer de llamar amigos. Pasando, por su puesto, por mi familia, siempre apoyandome en todas las decisiones que he tomado.

Índice general

Índice general	IV
Índice de tablas	VII
Índice de figuras	VIII
Índice de código	X
1 Introducción	1
1.1 Introducción y motivación	1
1.2 Estructura de esta memoria	1
1.3 Objetivos del proyecto	2
1.3.1 Objetivos técnicos	2
1.3.2 Objetivos académicos	2
1.4 Estado del arte	2
1.4.1 Tesla Autopilot	3
1.4.2 Cadillac SuperCruise	3
1.4.3 OpenPilot	4
1.5 Elicitación de requisitos	5
1.6 Análisis de riesgos	5
1.6.1 Identificación de los posibles riesgos	5
1.6.2 Plan de contingencia ante los posibles riesgos	6
2 Ejecución del proyecto	7
2.1 Diseño del sistema	7
2.1.1 Arquitectura	7
2.2 Tecnologías consideradas y empleadas	9
2.2.1 Tecnologías hardware	9
2.2.1.1 NVIDIA Jetson AGX Xavier	9
2.2.1.2 Sensores RGBD	10
2.2.1.2.1 Microsoft XBOX 360 Kinect	10
2.2.1.2.2 ASUS Xtion Pro Live	11
2.2.1.2.3 Conclusiones sobre el uso de sensores RGBD	11
2.2.1.3 Sensores RGB	11
2.2.1.3.1 SONY PlayStation Eye	11
2.2.1.3.2 Aukey PC-LM1	12
2.2.1.4 Conclusiones sobre el uso de sensores RGB y cambio a sensores virtuales	13
2.2.1.5 Tobii Eyetracker 4C	13
2.2.2 Tecnologías software	14

2.2.2.1 Python	14
2.2.2.2 OpenCV	14
2.2.2.3 C# y WPF	14
2.2.2.4 Simulador CARLA	14
2.2.2.4.1 Servidor	15
2.2.2.4.2 Vehículos	15
2.2.2.4.3 Sensores	15
2.2.2.4.4 API	15
2.2.2.5 Interfaz gráfica con Qt5	17
2.2.2.5.1 Qt Designer	17
2.2.3 <i>Datasets</i> utilizados para el entrenamiento de los modelos	18
2.2.3.1 LISA traffic lights dataset	18
2.2.3.2 Dataset de intersecciones	18
2.2.3.3 Dataset para <i>Road Director</i>	18
2.3 Desarrollo	20
2.3.1 Preparando el entorno de desarrollo y despliegue	20
2.3.1.1 NVIDIA JetPack	20
2.3.1.1.1 OpenCV	20
2.3.1.1.2 Tensorflow	20
2.3.1.2 jetson-inference	20
2.3.1.3 CARLA	21
2.3.1.3.1 Servidor	21
2.3.1.3.2 Conductor	21
2.3.1.3.3 Cliente Python para la NVIDIA Jetson AGX	21
2.3.1.4 Entorno Windows 10 para sensor de seguimiento ocular	22
2.3.2 Obteniendo datos de los sensores	24
2.3.2.1 Imágenes de las cámaras RGB	24
2.3.2.2 Sensores OBD	24
2.3.2.2.1 Velocidad del vehículo	25
2.3.2.2.2 Ángulo del volante	25
2.3.2.2.3 Límite de velocidad	25
2.3.2.3 Sensor de atención del conductor	25
2.3.2.3.1 Estructura de la aplicación	26
2.3.2.3.2 Obtención del estado del conductor	26
2.3.2.3.3 Envío de los datos a nuestra aplicación	26
2.3.3 Obteniendo conocimiento a partir de los datos adquiridos	28
2.3.3.1 Detección de intersecciones	28
2.3.3.2 Detección de objetos	31
2.3.3.3 Transformación de perspectiva sobre las imágenes de las cámaras	32
2.3.3.4 Detección de marcas de la calzada	32
2.3.3.4.1 Filtrado de la imagen para obtener marcas en la calzada	35
2.3.3.4.2 <i>Template Matching</i> con OpenCV	37
2.3.4 Analizando los datos	38
2.3.4.1 Clasificación del estado de los semáforos	38
2.3.4.1.1 Modelo de clasificación de semáforos	38
2.3.4.2 Algoritmo de seguimiento de vehículos y predicción de la posición futura	40
2.3.4.3 Algoritmo de seguimiento de peatones	42
2.3.5 Toma de decisiones	43
2.3.5.1 Control del límite de velocidad	43

2.3.5.2 Control de la atención del conductor	43
2.3.5.3 Aviso de cambio de semáforo	43
2.3.5.4 Aviso de vehículo en ángulo muerto	43
2.3.5.5 <i>Road director</i>	44
2.3.5.5.1 Entrenamiento y resultados obtenidos	45
2.3.6 Interfaz gráfica y notificaciones visuales	46
2.3.6.1 Interfaz inicial basada en OpenCV	46
2.3.6.2 Interfaz gráfica con Qt5	46
2.3.6.2.1 Cámaras, detección de objetos y vista cenital . . .	48
2.3.6.2.2 Indicadores	48
2.3.6.2.3 Notificaciones	48
2.3.6.2.4 <i>Road Director</i>	48
2.3.7 Notificaciones auditivas	49
2.3.7.1 Reproducción de sonidos con playsound	49
2.3.7.2 Reproducción de sonidos con mpv	49
2.4 Pruebas del sistema	50
3 Planificación del proyecto	51
3.1 Planificación temporal inicial	51
3.2 Planificación financiera inicial	51
3.3 Retrasos provocados por diversos motivos y cambios en el trabajo	52
3.4 Planificación temporal final	52
3.5 Planificación financiera final	52
3.6 Precios utilizados	52
3.7 Estudio de mercado	54
3.7.1 Clientes potenciales	54
3.7.2 Plan de comercialización	54
4 Trabajo futuro	55
4.1 Aspectos a mejorar del proyecto	55
4.2 Testeo en un vehículo real	56
4.3 Independencia del hardware utilizado	56
4.4 Ejecución con un sistema mucho más potente	56
5 Conclusiones	57
5.1 Comparativa con los objetivos y requisitos iniciales	57
5.2 Aspectos a mejorar	58
5.3 Impresión personal del proyecto	58
5.3.1 Aspectos positivos	58
5.3.2 Aspectos negativos	59
Apéndices	60

Índice de tablas

2.1	Resoluciones y tasas de refresco de la cámara PS Eye	12
2.2	Sensores disponibles en el simulador CARLA	16
2.3	Parámetros utilizados en el entrenamiento del modelo de detección de intersecciones.	28
2.4	Objetos del dataset COCO	31
2.5	Resultados obtenidos durante las pruebas del sistema	50
3.1	Precios utilizados para el cálculo de la planificación financiera	54

Índice de figuras

1.1 Ejemplos del funcionamiento interno del sistema Tesla Autopilot	4
1.2 Detalle del sistema de control del conductor de Cadillac Superruise	4
1.3 Ejemplos del funcionamiento del sistema Openpilot	5
2.1 Arquitectura de la aplicación	7
2.2 Placa de desarrollo NVIDIA Jetson AGX Xavier y dispositivo con refrigeración integrada	9
2.3 Aspecto exterior de los dispositivos RGBD	10
2.4 Comparativa entre la imagen RGB y el <i>depthmap</i> recibido	11
2.5 Cámara Sony PlayStation Eye	12
2.6 Webcam Aukey PC-LM1	12
2.7 Sensor de seguimiento ocular Tobii Eyetracker 4C	13
2.8 Ejemplos de entornos que se pueden encontrar en el simulador	14
2.9 Diversidad de vehículos en el simulador y vehículos finalmente seleccionado	15
2.10 Ejemplos de sensores que se pueden conectar a vehículos	16
2.11 Interfaz del software QtDesigner	17
2.12 Ejemplo de datos obtenidos con el dataset LISA	18
2.13 Imágenes de ejemplo del dataset de detección de intersecciones	19
2.14 Interfaz de la aplicación de conducción modificada	22
2.15 Datos de las cámaras recibidos desde el simulador	24
2.16 Arquitectura del sistema de control de atención del conductor	25
2.17 Servidor de atención del conductor ejecutándose.	27
2.18 Estructura del modelo de detección de intersecciones	29
2.19 Ejemplo de imagen recibida por el detector de intersecciones	30
2.20 Métricas obtenidas durante el entrenamiento del detector de intersecciones .	30
2.21 Aplicación del cambio de perspectiva	32
2.22 Imágenes de las cámaras con el cambio de perspectiva	32
2.23 Vista cenital de la zona alrededor del vehículo	35
2.24 Ejemplo de aislamiento incorrecto de las marcas en la calzada	36
2.25 Vista cenital de la zona delante del vehículo con filtro de marcas en calzada aplicado.	36
2.26 <i>Templates</i> utilizadas para el reconocimiento mediante patrones de OpenCV	37
2.27 Detección de marcas en la calzada con <i>template matching</i>	37
2.28 Método de selección de semáforo principal	38
2.29 Recorte y ajuste del tamaño del semáforo a una imagen de 25x25 píxeles .	38
2.30 Estructura del modelo de detección del estado del semáforo	39
2.31 Métricas obtenidas del entrenamiento del analizador de semáforos	40
2.32 Diagrama de funcionamiento del algoritmo de seguimiento de vehículos .	41
2.33 Predicción de posición futura del vehículo	41
2.34 Predicción de posición futura de peatones	42

2.35 Indicación de vehículo en ángulo muerto	43
2.36 El director de vuelo (<i>Flight director</i>) en la pantalla principal de vuelo de una aeronave	44
2.37 Visualización del ángulo de volante real y predecido	44
2.38 Ejemplos de interfaz gráfica generada con OpenCV	46
2.39 Captura de la interfaz gráfica desarrollada	47
2.40 Notificaciones visuales consideradas en la interfaz	48
3.1 Diagrama de Gantt de la planificación inicial	51
3.2 Diagrama de planificación de gastos inicial	52
3.3 Diagrama de Gantt de la planificación final	53
3.4 Diagrama de planificación de gastos final	53

Índice de código

2.1	Comandos para la compilación de <i>jetson-inference</i>	20
2.2	Instanciación de la cámara central del vehículo en el simulador	24
2.3	Grid principal de la aplicación	26
2.4	Cuadros de texto para selección del servidor y puerto	26
2.5	Función encargada de enviar los datos a la NVIDIA Jetson AGX	27
2.6	Script utilizado para convertir un SavedModel de tensorflow a un modelo optimizado de TensorRT	30
2.7	Constructor de la clase personal de vehículos	33
2.8	Función encargada de realizar la inferencia con el modelo de detección de objetos	34
2.9	Fragmento de código encargado de realizar la transformación de perspectiva de la cámara central	34
2.10	Filtro utilizado para extraer las marcas de la calzada	35
2.11	Fragmento de código utilizado para la detección de marcas en calzada	37
2.12	Clase encargada de la reproducción de notificaciones acústicas	49

CAPÍTULO 1

Introducción

1.1– Introducción y motivación

La principal motivación para la realización de esta memoria es la de cumplir con los requisitos necesarios para presentar el trabajo de fin de grado del grado en Ingeniería Informática - Tecnologías Informáticas aunque obviamente también existe la motivación personal del autor de documentar extensivamente el trabajo realizado así como la de profundizar en el área de los sistemas de ayuda a la conducción y, en general en la tecnología a bordo de los vehículos de hoy día.

1.2– Estructura de esta memoria

Esta memoria se encuentra estructurada en varios bloques:

- **Bloque 1:** Introducción

En este primer bloque se introduce el proyecto así como la temática general en la que se va a centrar. Además también se indicarán los objetivos, tanto técnicos como académicos que se esperan cumplir.

- **Bloque 2:** Ejecución del proyecto

A continuación nos encontraremos con el bloque dedicado a la explicación exhaustiva del diseño e implementación del sistema así como de las pruebas realizadas.

- **Bloque 3:** Planificación del proyecto

En este bloque se tratará la planificación temporal y financiera de este proyecto tanto la planeada inicialmente como los resultados finales tras la elaboración completa del trabajo.

- **Bloque 4:** Conclusiones

Tras las explicaciones anteriores, en esta sección se remarcarán los objetivos cumplidos así como una pequeña conclusión personal.

- **Bloque 5:** Trabajo futuro

En este bloque se tratarán algunos aspectos cuya ampliación en el futuro creemos que es un aspecto interesante.

- **Referencias bibliográficas y apéndices**

Finalmente nos encontraremos con las referencias bibliográficas y los distintos apéndices de este proyecto.

1.3– Objetivos del proyecto

Como para cualquier proyecto será necesario proponer unos objetivos que nos permitan llevar una evaluación del progreso.

1.3.1. Objetivos técnicos

Respecto a objetivos técnicos creemos conveniente que se cumplan los siguientes para dar por finalizado este trabajo.

- **Objetivo 1:** Detección y seguimiento del límite de velocidad
- **Objetivo 2:** Detección y aviso cambio de semáforo
- **Objetivo 3:** Detección y aviso de salida de carril
- **Objetivo 4:** Detección y aviso de vehículos en angulo muerto
- **Objetivo 5:** Detección y aviso de peatones en la trayectoria del vehículo
- **Objetivo 6:** Satisfactoria implementación del sistema en un sistema empotrado

A modo de resumen, como objetivo general que englobe a todos los demás se puede considerar el diseño e implementación de un sistema integral de ayuda a la conducción basado en sistemas empotrados de bajo coste que sea capaz de detectar situaciones y estados anómalos y notifique al conductor de forma visual y auditiva.

1.3.2. Objetivos académicos

Además de los objetivos técnicos enunciados en la sección anterior también se podrán considerar una serie de requisitos distintos relacionados con el carácter educativo y académico de este proyecto.

La elaboración de este trabajo no es más que el culmen final del trabajo realizado durante todos los años de estudios por lo que es natural que nos planteemos los siguientes objetivos que nos permitan demostrar y ampliar nuestro conocimiento en esta materia.

- **Objetivo 1:** Creación de una aplicación robusta y funcional que cumpla los objetivos técnicos especificados con anterioridad
- **Objetivo 2:** Ampliar conocimiento y profundizar en el estudio de distintas técnicas de *Machine Learning*
- **Objetivo 3:** Ampliar conocimiento sobre la elaboración de aplicaciones con interfaz de usuario gráfica

1.4– Estado del arte

El sistema que proponemos no es algo novel ya que actualmente existen tanto soluciones similares como soluciones mucho más complejas y completas en el mercado.

En los últimos años se ha producido una gran explosión en la investigación y comercialización de estos sistemas debido a las peticiones de los clientes que desean más seguridad para sus familias y que tienen la mirada puesta en la eventual conversión a la conducción autónoma total.

Los sistemas de ayuda a la conducción han sido unos de los primeros pasos que se han dado en la búsqueda de la conducción autónoma por parte de las empresas automovilísticas. Así pues nos podemos encontrar con sistemas como Mobileye[?], que forman la base de sistemas de ayuda a la conducción como Nissan ProPilot y las primeras versiones del sistema Autopilot de Tesla.

Empresas como Waymo y Uber también han creado sistemas de conducción autónoma con sensores LIDAR que obtienen una nube de puntos de alta resolución de los alrededores del vehículo. Además, existen otros sistemas basados en visión por computador creados por las propias empresas automovilísticas como SuperCruise de Cadillac, que consigue navegar en autopistas sin problema alguno gracias a la ayuda de *HD maps* generados previamente[?], o las últimas versiones del sistema Autopilot de Tesla [?], que alcanza un nivel de autonomía 2 en cualquier tipo de calzada con marcas claramente visibles.

SuperCruise es uno de los ejemplos más interesantes en cuanto al control del conductor ya que gracias a una serie de sensores situados en el volante son capaces de vigilar la atención del conductor. El uso de estos sensores es mucho mas avanzado que el simple sensor de fuerza de torsión que usa Tesla para la misma función, aunque cabe destacar que los últimos modelos de la marca (Model 3 y Model Y) incorporan una cámara que tiene visión del interior del vehículo [?] así que es muy posible que en el futuro cercano se comience a utilizar para vigilar al conductor mediante el uso de visión por computador.

El ejemplo que más se parece a nuestro proyecto sería OpenPilot, de Comma.ai [?], un proyecto basado en visión por computador e inteligencia artificial que, junto con los datos del radar de los vehículos compatibles, es capaz de analizar el entorno y controlar el vehículo hasta un nivel 2 de autonomía.

Por supuesto, a pesar de ser el sistema más parecido al propuesto, nuestro sistema quedará lejos de este otro pues la capacidad tecnológica del autor no podrá ser comparada con la del grupo de personas que desarrollan OpenPilot.

1.4.1. Tesla Autopilot

El sistema Tesla Autopilot es el sistema de conducción autónoma de nivel 2 disponible para la mayoría de vehículos creados por la empresa estadounidense y que ofrece la capacidad de controlar el vehículo en diversos entornos. La recolección de datos se centra principalmente en las imágenes obtenidas por las cámaras del vehículo aunque también se trata la información generada por el radar central y los sensores de ultrasonidos.

En diversas conferencias Andrej Karpathy ha compartido detalles sobre el funcionamiento interno de este sistema y como este ha ido evolucionando para, con el tiempo, ir convirtiéndose en un sistema mucho más centrado en los sistemas de inteligencia artificial.

El sistema autopilot está compuesto por más de 48 redes neuronales que realizan más de 1000 predicciones por cada fotograma analizado. Uno de los sistemas más interesantes es el sistema de predicción de calzada en vista cenital que se puede observar en la figura 1.1b. Este sistema es capaz de generar un mapa bidimensional del entorno alrededor del vehículo con un precisión bastante acertada utilizando únicamente los datos obtenidos por las cámaras del vehículo [?] [?] [?].

Una parte de nuestro trabajo, tal y como se verá en el siguiente capítulo, esta inspirada en este sistema.

1.4.2. Cadillac SuperCruise

En cuanto al sistema de conducción semiautónoma de Cadillac, SuperCruise, se puede destacar que es un sistema que utiliza mapas de alta resolución generados mediante LIDAR, Radar, GPS y cámaras convencionales para permitir que el vehículo pueda conducir sin la interacción directa del conductor [?].

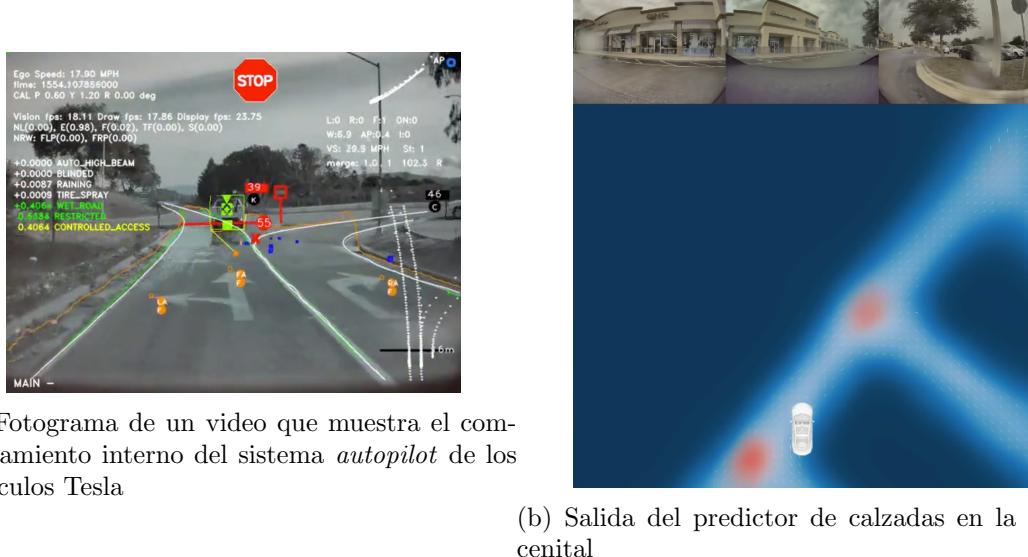


Figura 1.1: Ejemplos del funcionamiento interno del sistema Tesla Autopilot

A pesar de que apenas existe información sobre este sistema, ya que tan solo se puede utilizar en varias autopistas de EEUU y Canadá y que la cantidad de personas con acceso a él es mucho menor que a la de los sistemas de sus competidores, en la figura 1.2 se puede observar el dispositivo encargado de controlar la atención del conductor que, además de un led indicador, parece tener 6 leds infrarrojos que proyectarán un patrón fácilmente reconocible en la pupila del conductor. Este sistema, parece tener gran similitud con los sistemas de seguimiento ocular para consumidores de la empresa Tobii, uno de los cuales será utilizado en este proyecto.



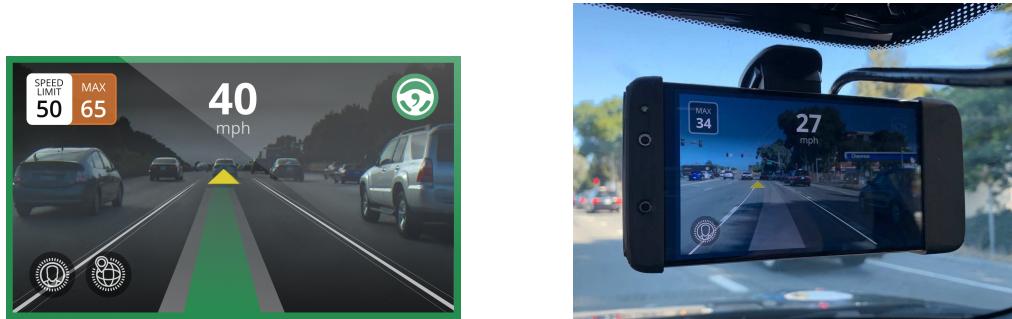
Figura 1.2: Detalle del sistema de control del conductor de Cadillac Supercruise

1.4.3. OpenPilot

Openpilot es un sistema de conducción autónoma open-source desarrollado por George Hotz capaz de detectar las líneas de la calzada e interaccionar con el radar de los vehículos compatibles para controlarlo hasta un nivel de autonomía 2. Además también realiza alertas visuales y auditivas ante situaciones anómalas, como por ejemplo la superación del límite de velocidad. Inicialmente, el sistema se ejecutaba en un teléfono móvil convencional

pero posteriormente se ha trasladado a una solución hardware propietaria.

Uno de los puntos interesantes de esta solución es la compatibilidad con más de 85 vehículos de distintas marcas, algo solo posible al tratarse de un proyecto *opensource*.



(a) Interfaz gráfica del sistema Openpilot

(b) Ejemplo de uso de Openpilot en un vehículo real

Figura 1.3: Ejemplos del funcionamiento del sistema Openpilot

1.5– Elicitación de requisitos

A continuación se indican los requisitos que nuestra aplicación deberá cumplir para considerar una entrega final como satisfactoria.

- El sistema deberá recibir la información del vehículo y mostrarla
- El sistema deberá ser capaz de reconocer los vehículos a su alrededor
- El sistema deberá detectar los cambios de los semáforos
- El sistema deberá controlar la atención del conductor
- El sistema deberá avisar acústica y visualmente ante situaciones anómalas

1.6– Análisis de riesgos

Por supuesto, como cualquier otro proyecto es posible que se den algunos problemas. A continuación se describen los posibles riesgos que se pueden dar.

1.6.1. Identificación de los posibles riesgos

- Posibles retrasos derivados de las distintas obligaciones del autor
Puesto que la elaboración de este proyecto se realizará en paralelo a la finalización de los estudios del autor y de la entrada al mundo laboral es bastante posible que la priorización de otras tareas no relacionadas con la ejecución del proyecto afecten a los tiempos de desarrollo del mismo.
- Pérdida parcial o completa del código fuente
Al trabajar con un proyecto importante siempre es preocupante que debido a problemas de hardware se produzca la pérdida parcial o completa del código fuente.

- Imposibilidad del acceso a las herramientas de desarrollo

Otro de los posibles problemas que nos podemos encontrar será la imposibilidad de acceder a las herramientas de desarrollo de nuestro proyecto por causas de fuerza mayor.

- Dificultad de implementación

Otro posible riesgo que nos podemos encontrar sería las posibles complicaciones durante la implementación de alguna parte del proyecto. Estas complicaciones podrían aparecer a raíz del desconocimiento de algún tipo de herramienta o librería por parte de los autores.

1.6.2. Plan de contingencia ante los posibles riesgos

En el caso de que alguno de los riesgos enunciados anteriormente llegue a materializarse es conveniente conocer las distintas opciones disponibles para minimizar las partes del trabajo que se verían afectadas.

- Retrasos en la elaboración del proyecto

La principal solución a los problemas de retrasos consistirá en la reorganización de las tareas y, en el caso de que sea necesario, el retraso de la entrega a las distintas convocatorias disponibles durante el curso escolar.

- Pérdida del código fuente

La perdida del código fuente supondría el aplazamiento completo de las tareas y consecuentes retrasos por lo que este es un riesgo que no podemos permitirnos. Afortunadamente, en nuestra industria, la perdida parcial o completa es un riesgo menor gracias a los distintos sistemas que existen para llevar un seguimiento de datos. Para reducir este riesgo y que nunca se llegue a dar, todo el código utilizado en este proyecto se encuentra archivado siguiendo la regla copias de seguridad 321 [?]. En concreto, para nuestro proyecto existen al menos 5 copias en 2 medios distintos y con al menos 2 copias *offsite*.

- Imposibilidad del acceso a las herramientas de desarrollo

En el caso de que perdamos el acceso a las herramientas de desarrollo de nuestro proyecto o no podamos acceder a ellas por causas de fuerza mayor deberemos modificar la planificación para intentar que el periodo durante el que no tengamos acceso se pueda invertir en otras tareas que no dependan de estas herramientas. Aun así, si este riesgo llegase a materializarse se deberá aceptar el hecho de que los retrasos serán necesarios.

- Dificultad de implementación

En el caso de que nos encontremos con distintos problemas durante la implementación de alguna parte del proyecto nos veremos obligados a realizar una extensiva investigación para resolver los problemas con los que nos podamos encontrar. Afortunadamente tanto las herramientas como el hardware utilizados parecen estar muy bien documentados y tienen una gran comunidad de personas detrás que nos permitirán, casi con total seguridad, encontrar las respuestas a nuestras preguntas. Si los frutos de la investigación no son suficientes siempre podremos apoyarnos en la ayuda que nuestros tutores nos puedan aportar.

CAPÍTULO 2

Ejecución del proyecto

En este capítulo se podrá encontrar la explicación detallada de las tecnologías consideradas y finalmente utilizadas así como los detalles de la implementación realizada.

2.1– Diseño del sistema

2.1.1. Arquitectura

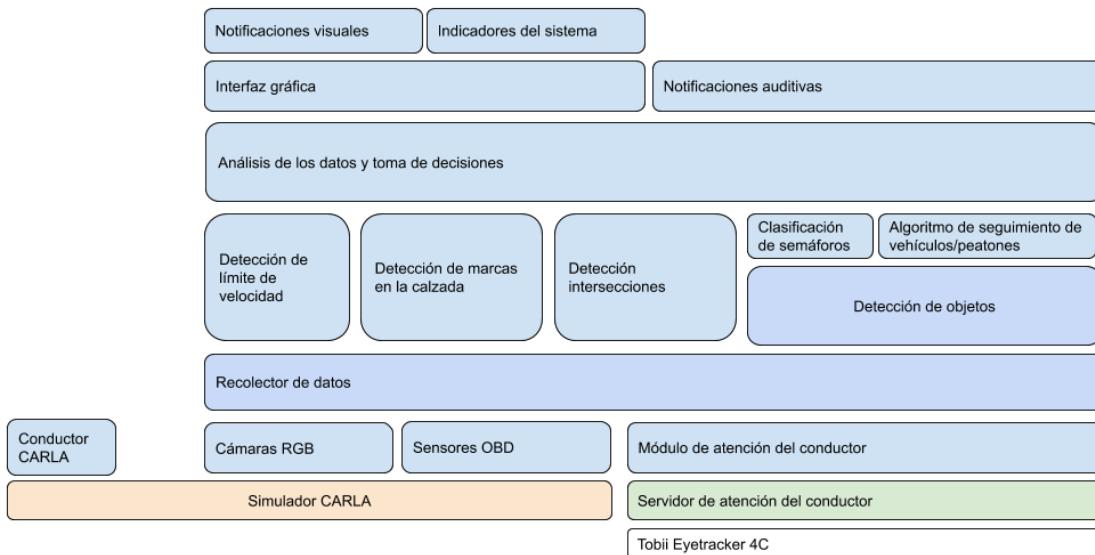


Figura 2.1: Arquitectura de la aplicación

La idea principal de la aplicación consiste en la recolección de los datos aportados por el simulador CARLA, en el cual controlamos un vehículo mediante el Conductor CARLA, y el servidor de atención del conductor, que utiliza el sensor Tobii Eyetracker 4C, para posteriormente ser analizados por los distintos módulos de detección y, una vez tengamos conocimiento sobre el estado del vehículo, poder tomar las decisiones y alertar al conductor en el caso de que se crea necesario.

A continuación se comentan con un poco más de detalle los puntos más importantes de entre los que se pueden observar en la figura 2.1:

- **Simulador y conductor CARLA**

El simulador CARLA y la aplicación de conducción nos proporcionarán el mundo y el vehículo sobre el cual se ejecutará nuestro sistema. El usuario tiene control total sobre el vehículo de la misma forma que tendría control sobre un vehículo real.

- **Servidor de atención del conductor**

El servidor de atención del conductor será la pequeña aplicación encargada de controlar la atención del conductor gracias al uso del sensor Tobii EyeTracker 4C

- **Recolector de datos**

El módulo recolector de datos será el encargado de obtener la información del simulador y del servidor de atención del conductor para ponerla a disposición del resto de la aplicación.

- **Módulos de detección**

Una vez obtenidos los datos será necesario analizarlos para obtener conocimiento, este proceso se delega a los distintos módulos que se encargarán de detectar intersecciones, objetos y marcas en la calzada.

- **Análisis de los datos y toma de decisiones**

Cuando se haya obtenido conocimiento acerca de la situación del vehículo será necesario determinar si es necesario emitir alguna notificación. Este módulo será el encargado de realizar esta decisión

- **Interfaz gráfica y notificaciones**

Finalmente, la interfaz gráfica proporcionará un lugar donde podrán observarse todas los datos recogidos así como las notificaciones visuales. Por otra parte, el módulo de notificaciones auditivas permitirá la reproducción de sonidos que alerten al conductor.

2.2– Tecnologías consideradas y empleadas

Para la elaboración de este trabajo hemos investigado el uso de distintos dispositivos para obtener información acerca del exterior e interior del vehículo. La mayoría de los dispositivos no han sido utilizados finalmente puesto que se decidió adaptar nuestra aplicación para que funcione sobre un simulador. Aun así, creemos que es importante aportar información sobre estos dispositivos pues, tal y como se hablará en el capítulo 4, es posible que en el futuro se utilicen en el caso de que se desee adaptar la aplicación para su uso en un vehículo real. A continuación se explicarán detalladamente cada uno de estos dispositivos.

2.2.1. Tecnologías hardware

2.2.1.1. NVIDIA Jetson AGX Xavier



Figura 2.2: Placa de desarrollo NVIDIA Jetson AGX Xavier y dispositivo con refrigeración integrada

La NVIDIA Jetson AGX Xavier se trata de la placa de desarrollo por excelencia para aplicaciones de inteligencia artificial *on the edge* de la compañía NVIDIA. Está basada en un procesador ARM de 8 núcleos, posee una GPU Volta con 512 *Tensor cores* optimizados para cálculos de inteligencia artificial y 16GB de RAM. NVIDIA asegura que el dispositivo es capaz de alcanzar una potencia computacional de aproximadamente 32 TeraOPS con un consumo máximo de 30W, lo que la hace la opción ideal para la utilización en nuestro proyecto ya que nos permitirá realizar los cálculos necesarios así como las inferencias de los distintos modelos de inteligencia artificial sin que nos tengamos que preocupar por falta de potencia.

En este proyecto utilizaremos la versión con la *carrier board*, que tal y como se puede ver en la figura 2.2 ya tiene incorporado el sistema de refrigeración.

Uno de los puntos importantes a destacar es que la GPU incluida en este SoC cuenta con 512 *tensorcores*, los cuales son núcleos optimizados para el cálculo de la aritmética utilizada durante la inferencia de los modelos de inteligencia artificial. Gracias a estos núcleos y a la optimización del modelo con la librería TensorRT obtendremos una gran mejora de rendimiento durante la inferencia de los distintos modelos que vamos a crear.

En cuanto a sistema operativo, el dispositivo ejecuta Ubuntu 18.04 sobre la versión 4.9 del kernel de Linux. En este aspecto se puede considerar que la NVIDIA Jetson AGX se trata de un PC normal siendo los únicos puntos a destacar la arquitectura de esta, que es ARM64, y las restricciones de energía.

2.2.1.2. Sensores RGBD



(a) Microsoft Xbox 360 Kinect

(b) ASUS Xtion Pro Live

Figura 2.3: Aspecto exterior de los dispositivos RGBD

Inicialmente se pensó en la utilización de sensores RGBD para la obtención de los datos del exterior del vehículo.

Los sensores RGBD, además de comportarse como una simple cámara RGB también nos aportan un *stream* de datos que nos indica la distancia de cada píxel de la imagen al sensor, generalmente en milímetros.

Estos sensores funcionan de la siguiente forma:

1. Una cámara RGB recoge la información de la luz visible de la escena.
2. Un emisor infrarrojo inunda la estancia en la que se encuentre el sensor con una nube de puntos
3. Una cámara infrarroja recoge una imagen en el espectro del infrarrojo cercano en la que se pueden diferenciar claramente los puntos del emisor.
4. Internamente se calcula un *depthmap* del entorno de acuerdo con la calibración de fábrica del dispositivo.

Sin embargo, uno de los problemas de estos sensores es que dependen de la nube de puntos emitida para poder calcular correctamente la distancia de los objetos y para aplicaciones en el exterior, como era nuestro caso antes del cambio al simulador, la nube de puntos se veía completamente eclipsada por la luz solar. En la figura 2.17 se puede observar un ejemplo de los datos que recibimos desde el dispositivo en una escena exterior. Finalmente se decidió contra la utilización de este tipo de cámaras puesto que aunque es posible utilizarlas como cámaras RGB normales el tamaño de estas es mucho mayor que el de cualquier cámara RGB, lo que dificulta la instalación de estos sensores en un vehículo real.

2.2.1.2.1. Microsoft XBOX 360 Kinect

El dispositivo Kinect es un periférico de la consola Xbox 360 de Microsoft. Inicialmente desarrollado para competir con otros dispositivos como el mando *PS Move* de Sony y el *Wii Remote* de Nintendo este dispositivo permitía la interacción del usuario en los juegos con todo su cuerpo gracias a la detección del esqueleto del jugador.

En realidad, este periférico no es más que una cámara RGBD que ofrecía los valores de distancia de la imagen capturada a la consola la cual junto con diversos algoritmos realizaba la identificación de la posición del jugador.

Apenas 3 horas después de la salida del dispositivo Héctor Martín ya había creado un driver *open-source* con el que utilizar este dispositivo en un PC [?]. A partir de este punto la comunidad de *makers* comenzó a utilizar este dispositivo en todo tipo de aplicaciones robóticas.



Figura 2.4: Comparativa entre la imagen RGB y el *depthmap* recibido. En este último, el color blanco se corresponde a las regiones sin datos y el negro con regiones demasiado cercanas (reflejos del parabrisas)

2.2.1.2.2. ASUS Xtion Pro Live

El dispositivo ASUS Xtion Pro Live se trata de un dispositivo similar al Kinect que utiliza el mismo tipo de tecnología y del que se pueden obtener los mismos resultados.

La única diferencia razonable es que este dispositivo es mucho más ligero ya que no incorpora el motor en la base y que funciona únicamente con un cable USB mientras que Kinect necesita una fuente de alimentación externa.

2.2.1.2.3. Conclusiones sobre el uso de sensores RGBD

Como ya hemos visto, el uso de este tipo de sensores para la obtención de un *depthmap* no es posible. Sin embargo, también se tuvieron en cuenta otros casos en los que estas cámaras nos podrían haber resultado útiles.

Una de esas opciones sería la utilización de las dos cámaras al mismo tiempo para obtener una vista estereoscópica de la escena y de esta forma poder calcular la profundidad, una idea especialmente interesante en el caso del dispositivo ASUS Xtion Pro Live, puesto que sus cámaras son simétricas respecto al centro del dispositivo y por su menor peso en comparación con el otro dispositivo a nuestra disposición. Sin embargo, a pesar de lo que podría parecer a primera vista, ninguno de los dos dispositivos a nuestra disposición son capaces de aportar estos dos streams de datos al mismo tiempo.

Finalmente se decidió contra la utilización de este tipo de cámaras puesto que aunque es posible utilizarlas como cámaras RGB normales el tamaño de estas es mucho mayor que el de cualquier otra cámara RGB, lo que dificulta la instalación de estos sensores en un vehículo real.

2.2.1.3. Sensores RGB

Una vez visto como los sensores RGBD no eran la mejor opción, pasamos a considerar la opción de cámaras RGB normales.

2.2.1.3.1. SONY PlayStation Eye

Una de las cámaras consideradas fue la SONY PlayStation Eye. Como su nombre indica, esta cámara es un periférico utilizado por la consola Sony PlayStation 3 para realizar el seguimiento de los mandos con control de movimiento PS Move.

El hecho de que haya sido utilizada por una empresa tan importante como Sony en uno de sus productos principales nos da a entender que esta cámara, a pesar de su redu-



Figura 2.5: Cámara Sony PlayStation Eye

cido tamaño aporta grandes resultados, algo cuyas especificaciones, las cuales se pueden comprobar en la tabla 2.1, parecen corroborar.

Además de ser la calidad de imagen obtenida por la cámara, en la parte superior del dispositivo nos encontramos con una colección de micrófonos que también serán accesibles y que se podrían utilizar para ayudar en el reconocimiento de, por ejemplo, el claxon de otros vehículos o las sirenas de vehículos de emergencias.

Especificaciones con parámetros por defecto	
Resolución	Tasa de refresco
640x480	60 FPS
320×240	120 FPS
Especificaciones con parámetros manuales	
640×480	Hasta 75 FPS (con <i>artefacting</i>)
320×240	Hasta 187 FPS (con <i>artefacting</i>)

Tabla 2.1: Resoluciones y tasas de refresco de la cámara PS Eye

2.2.1.3.2. Aukey PC-LM1

Otra cámara que se consideró utilizar fue la Aukey PC-LM1. Esta cámara, se trata de una *webcam* USB con una resolución de 1080p a una tasa de refresco de 30 FPS. A pesar de que la tasa de refresco sea menor que la que se obtiene con la PS Eye, se pensó que el gran aumento de la resolución nos podría resultar útil para realizar detecciones más precisas.



Figura 2.6: Webcam Aukey PC-LM1

2.2.1.4. Conclusiones sobre el uso de sensores RGB y cambio a sensores virtuales

Tras el cambio del proyecto al simulador CARLA, la utilización de sensores físicos quedó obsoleta puesto que la información del exterior del vehículo se recibiría mediante la API de conexión con el simulador.

2.2.1.5. Tobii Eyetracker 4C



Figura 2.7: Sensor de seguimiento ocular Tobii Eyetracker 4C

Uno de los sensores que si se ha mantenido desde la consideración inicial ha sido el sensor de seguimiento ocular Tobii Eyetracker 4C. El uso de este sensor nos permite conocer el estado de la visión del conductor y es sin duda uno de los pilares principales de nuestro sistema pues nos permite llevar el control del estado de atención del conductor.

Existe poca información sobre el funcionamiento del hardware ya que se trata de un producto comercial con precio considerable. Aún así, todo parece apuntar a que varios leds infrarrojos proyectan un patrón fácilmente reconocible en la pupila del usuario el cual es reconocido por las dos cámaras situadas en los extremos del sensor e internamente se calcularán los datos que finalmente se ofrecerán al usuario.

Para acceder a la información obtenida por el sensor Tobii nos ofrece una API con la que podremos obtener distinta información como las coordenadas de la mirada del usuario, la distancia a la mirada y otras más. Sin embargo, como se verá en la sección 2.3.2.3 nuestro dispositivo está bastante limitado al no tratarse de un dispositivo *research*.

2.2.2. Tecnologías software

2.2.2.1. Python

Este lenguaje de programación ha sido seleccionado por la gran facilidad de implementación así como por ser el lenguaje con el que el autor está más familiarizado. Además, dejando de lado la familiaridad del autor con este lenguaje, Python es uno de los lenguajes más utilizados para el prototipado rápido de aplicaciones de *Machine Learning* puesto que existen *bindings* para la mayoría de las librerías y el hecho de que el *overhead* del lenguaje no es excesivamente grande.

2.2.2.2. OpenCV

Para el tratamiento de imágenes se utilizará la librería OpenCV por tratarse de la librería *standard* en tareas de visión por computador con Python. Además, esta librería es relativamente conocida por el autor.

2.2.2.3. C# y WPF

Para obtener los datos que nos permitan controlar la atención del conductor utilizaremos una aplicación escrita en C# con el marco de interfaz de usuario Windows Presentation Foundation, que nos permitirá crear una pequeña interfaz gráfica que sea reactiva a la visión del conductor para, de esta forma, saber si el conductor está prestando atención a la calzada o se encuentra distraído. Como se ha visto en 2.2.1.5 utilizaremos las funciones disponibles de la API de Tobii para obtener los datos que se correspondan con la vista del conductor. Concretamente, en nuestro caso haremos reactiva a la vista una ventana y posteriormente utilizaremos las funciones standard de C# para enviar esta información a la NVIDIA Jetson AGX Xavier.

A pesar del completo desconocimiento del autor de este lenguaje, la gran facilidad de uso así como el gran parecido con Java hacen que programar sea sencillo y que en apenas unas horas se puedan obtener resultados muy buenos.

2.2.2.4. Simulador CARLA

El simulador CARLA [?] es un simulador *open source* creado por personal del Toyota Research Institute, Intel Labs y personal del Computer Vision Center de Barcelona. Está creado con Unreal Engine [?] y ofrece control total sobre los vehículos y el mundo que les rodea por lo que el uso de este simulador está indicado a todas las personas que deseen investigar y desarrollar sistemas de ayuda a la conducción.



Figura 2.8: Ejemplos de entornos que se pueden encontrar en el simulador

2.2.2.4.1. Servidor

CARLA utiliza una arquitectura Cliente-Servidor en la cual el servidor es el encargado de la creación del mundo, el control de los distintos NPCs, ya sean vehículos o peatones, y la renderización de los datos de los sensores.

De entre todos los mapas disponibles en el simulador nos hemos centrado en el mapa Town03 ya que creemos que será posible cumplir nuestros objetivos con el mundo que nos ofrece este mapa.

2.2.2.4.2. Vehículos

En la figura 2.9a se puede observar una muestra de la diversidad de vehículos disponibles en las primeras versiones del simulador. Desde varias de las últimas versiones del simulador se han incluido diversos cambios que mejoran el comportamiento de los vehículos así como permiten el control de diversos aspectos de los mismos. Uno de los más interesantes y actuales ha sido la inclusión del control de luces.



(a) Diversidad de vehículos que se pueden encontrar en el simulador (b) Tesla Model 3 en el simulador

Figura 2.9: Diversidad de vehículos en el simulador y vehículos finalmente seleccionado

Para nuestro proyecto hemos seleccionado el vehículo Tesla Model 3 ya que se trata de un vehículo que posee amplias cualidades de conducción autónoma en el mundo real y creemos que se trata de una buena base para comenzar nuestro trabajo. Uno de los aspectos que hemos intentado imitar es la posición de las cámaras externas virtuales, las cuales se han colocado aproximadamente en la posición de las cámaras del vehículo real.

Además al tratarse de un vehículo eléctrico obtendremos una conducción mucho más suave y no nos tendremos que preocupar por los cambios de marchas, lo que puede ser de ayuda para nuestros algoritmos.

2.2.2.4.3. Sensores

En CARLA existen diversos tipos de sensores que podemos añadir a la simulación, en la tabla 2.2 se muestran todos los disponibles en la versión 0.99. Dos de los más interesantes y que fueron considerados para la utilización en este proyecto son las cámaras RGB, que por supuesto han sido las utilizadas finalmente, y las cámaras con datos de profundidad, que no se utilizaron finalmente pues trabajar con datos en 3D ampliaría la complejidad de nuestro sistema.

2.2.2.4.4. API

Para controlar la simulación y obtener datos se utiliza una API de conexión que nos permite modificar varios parámetros del simulador así como recibir los datos de los sensores que hayamos conectado a algún vehículo. El único punto a destacar sobre esta API es que no se encuentra disponible para la arquitectura de nuestro sistema y deberá ser compilada para que funcione en la NVIDIA Jetson.

Sensores disponibles en CARLA
Detector de colisiones
Sensor de profundidad
Sensor GNSS
Acelerómetro, giroscopio y brújula
Detector de invasión de carril.
LIDAR
LIDAR semántico
Detector de obstáculos
Radar
Cámaras RGB
Cámara de segmentación semántica
Cámara DVS

Tabla 2.2: Sensores disponibles en el simulador CARLA

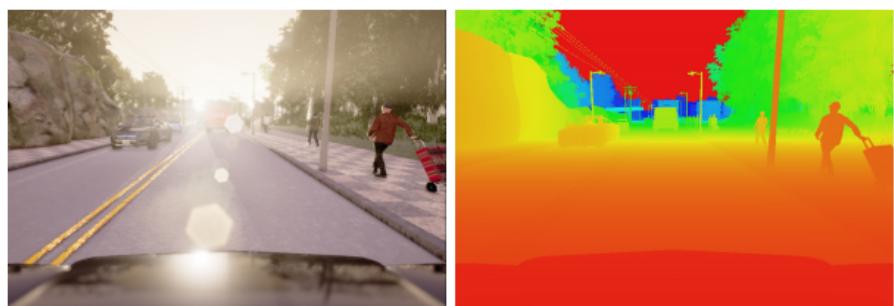


Figura 2.10: Ejemplos de sensores que se pueden conectar a vehículos. El primero de ellos es una cámara RGB, el segundo es un sensor de profundidad en escala logarítmica

2.2.2.5. Interfaz gráfica con Qt5

2.2.2.5.1. Qt Designer

Para el diseño de la interfaz gráfica hemos utilizado el *software* QtDesigner, que nos permite crear interfaces gráficas con la librería Qt5, la cual podremos controlar desde nuestro código python.

El uso de QtDesigner es extremadamente simple y, tal y como se puede observar en la figura 2.11, recuerda a las herramientas de creación de interfaces de otros *softwares* como Android Studio y Xcode. Una vez tengamos definida una interfaz necesitaremos transformarla a código python para que la podamos utilizar en nuestra aplicación. Este proceso se realiza mediante el uso de la herramienta `pyuic` a la cual le aportamos el archivo con la definición de la interfaz y nos devolverá una clase python con la que podremos ejecutar la interfaz.

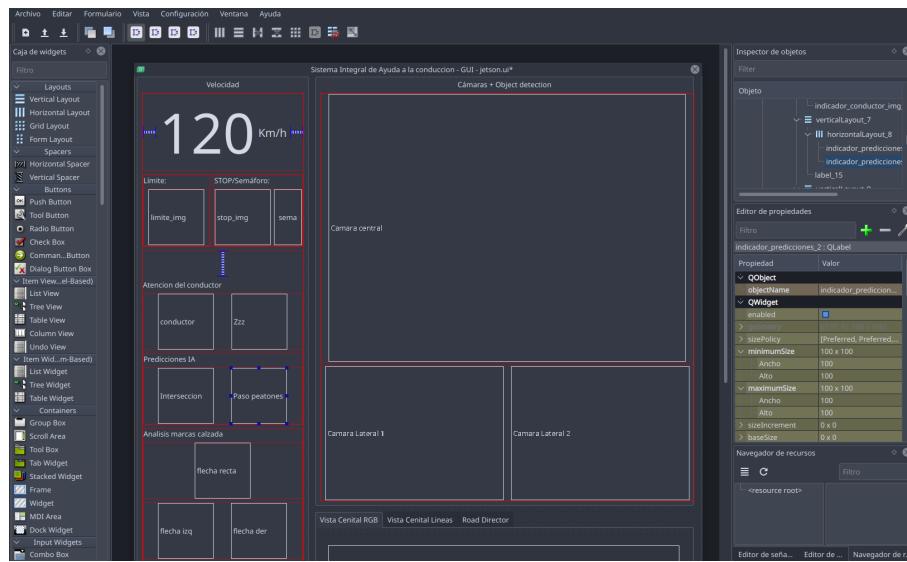


Figura 2.11: Interfaz del software QtDesigner

2.2.3. Datasets utilizados para el entrenamiento de los modelos

A continuación se realiza una pequeña explicación de cada uno de los datasets utilizados para el entrenamiento de los distintos modelos de inteligencia artificial.

2.2.3.1. LISA traffic lights dataset

El dataset LISA [?] se trata de un conjunto de imágenes extraídas de unos videos grabados en la parte frontal de un vehículo que han sido anotados para permitir la identificación de distintos tipos de señales de tráfico. Las imágenes han sido grabadas con varios tipos de cámaras por lo que nos encontramos con distintas resoluciones que varían entre 640x480 y 1024x522.



Figura 2.12: Ejemplo de datos obtenidos con el dataset LISA

El dataset completo contiene 47 tipos de señales y 7855 anotaciones para las 6610 imágenes que nos ofrece. En nuestro caso hemos reducido este dataset únicamente a la detección de semáforos, por lo que nos hemos quedado con 27518 imágenes para las cuales hemos recortado los semáforos y clasificados en dos grupos, semáforos en verde y semáforos en rojo y ámbar.

2.2.3.2. Dataset de intersecciones

El dataset que se ha utilizado para la detección de intersecciones ha sido obtenido por el autor de este trabajo de forma manual. El dataset consiste en 137 imágenes obtenidas del módulo de análisis de datos, concretamente de la vista cenital de la parte frontal del vehículo. En la figura 2.13 se pueden observar dos ejemplos de imágenes que contienen este dataset.

2.2.3.3. Dataset para *Road Director*

Al igual que el dataset para la detección de intersecciones, el dataset utilizado para el entrenamiento del modelo *Road Director* se trata de un dataset personal creado con imágenes de la vista cenital del vehículo. En este caso, el dataset fue creado de forma automática gracias a la funcionalidad “Autopilot” del simulador CARLA, la cual permite que el vehículo se mueva por el mapa de forma automática.



(a) Ejemplo positivo

(b) Ejemplo negativo

Figura 2.13: Imágenes de ejemplo del dataset de detección de intersecciones

Mientras el vehículo se encontrase en movimiento y con una velocidad superior a 3 km/h un pequeño script se encarga de guardar la imagen de la vista cenital inmediatamente delante del vehículo así como la información de la posición del volante y acelerador.

2.3– Desarrollo

A continuación se realiza la extensiva explicación de desarrollo realizado para cumplir con los objetivos de este proyecto.

2.3.1. Preparando el entorno de desarrollo y despliegue

2.3.1.1. NVIDIA JetPack

La instalación de NVIDIA JetPack se realiza durante la puesta en marcha del dispositivo NVIDIA Jetson AGX y consiste en la inicialización del dispositivo e instalación de librerías principales.

Para realizar esta inicialización es necesario conectar la Jetson AGX a un ordenador con arquitectura x64 y sistema operativo Ubuntu 18.04 puesto que deberemos instalar el *NVIDIA SDK Manager*, el cual únicamente se encuentra disponible para este sistema. Una vez descargado conectaremos el dispositivo al ordenador y los pondremos en *recovery mode* manteniendo pulsado el botón correspondiente. Una explicación más exhaustiva se podrá encontrar en [?].

2.3.1.1.1. OpenCV

Tal y como se ha comentado anteriormente, una de las herramientas que utilizaremos en este proyecto será OpenCV y, afortunadamente, al realizar la instalación de JetPack se nos dió la opción de instalar OpenCV al mismo tiempo por lo que una vez se inicialice el dispositivo tendremos acceso inmediato a la librería.

2.3.1.1.2. Tensorflow

Al igual que con OpenCV, al realizar la instalación inicial de NVIDIA Jetpack se nos dió la opción de instalar al mismo tiempo Tensorflow. Al realizar la instalación de esta forma nos aseguramos que Tensorflow se instale correctamente y aproveche sin problemas la GPU del dispositivo puesto que también se instalan automáticamente las distintas dependencias así como CUDA para permitir el uso de la GPU en las inferencias.

La versión de Tensorflow utilizada en este proyecto es la 2.0, concretamente la versión 2.3.0.

2.3.1.2. jetson-inference

Para utilizar la librería jetson-inference necesitaremos compilarla y por tanto tendremos que obtener el código fuente del repositorio en GitHub[?]. Tras esto, necesitaremos compilar el repositorio. A modo de resumen, una vez tengamos clonado el repositorio e instaladas las dependencias, estos son los comandos que se deben ejecutar para compilar e instalar la librería:

```
1 $ cd jetson-inference
2 $ mkdir build
3 $ cd build
4 $ cmake ../
5 $ make -j$(nproc)
6 $ sudo make install
7 $ sudo ldconfig
```

Código 2.1: Comandos para la compilación de *jetson-inference*

Durante la compilación se nos preguntará qué modelos deseamos instalar junto a la librería. Nuestro sistema es compatible con todos los modelos indicados por la aplicación (a fecha 21 de septiembre de 2020) luego es posible instalar todos los modelos disponibles

en la sección de *object detection* e ir modificando el código fuente para ver como modifica el comportamiento.

Cabe destacar que es posible que se deseen modificar varios archivos C del código fuente de la API *jetson-utils* puesto que si no los modificamos obtendremos una salida constante en la terminal que nos impedirá observar otros tipos de salidas. A pesar de que la recomendación sea la de modificar los archivos esta decisión queda a disposición del lector.

2.3.1.3. CARLA

Tal y como se ha comentado en la sección 2.2.2.4 el desarrollo y testeo de nuestro software se ha realizado con el simulador de conducción CARLA [?].

En cuanto a la utilización de CARLA en este proyecto es necesario diferenciar entre dos partes claramente diferenciadas. Se podrá hacer la distinción entre el servidor CARLA, encargado de la lógica, cálculos y renderizado del mundo y la aplicación de conducción, que creará y ofrecerá el control de un vehículo en el mundo que se está ejecutando en el servidor.

2.3.1.3.1. Servidor

La preparación del servidor de CARLA es simple, una vez descargada la *release* deseada procederemos a descomprimir este archivo y a ejecutar el binario `Carla.sh`. Se recomienda encarecidamente utilizar los parámetros `--quality-level=Low` y `--opengl` puesto que mejorará el rendimiento del simulador utilizando unos gráficos más simples y el motor OpenGL el cual se recomienda para los sistemas GNU/Linux por los propios creadores del simulador[?].

A modo de información, durante la elaboración de este proyecto se ha utilizado la versión 0.9.9 de este simulador, tanto para el servidor como para el cliente y es posible que el funcionamiento entre distintas versiones no sea el correcto ya que pueden existir muchos cambios al tratarse de un proyecto aún en desarrollo.

2.3.1.3.2. Conductor

Por otra parte, la aplicación de conducción se trata de un cliente de CARLA que tras conectarse hace aparecer un vehículo en el mundo y nos ofrece el control de este. Gracias a esta aplicación tendremos el control total de un vehículo al que le podemos añadir los sensores deseados para que nuestro software principal pueda entender el estado del vehículo.

La aplicación de conducción se trata de una modificación de los ejemplos de interacción de CARLA, y por lo tanto se encuentra escrita en Python con la ayuda de la librería pygame. En la figura 2.14 se puede observar la interfaz de esta aplicación.

Entre las modificaciones realizadas al archivo original se puede destacar la eliminación de los sensores de los que no deseamos recibir datos como por ejemplo el LIDAR y las distintas cámaras de segmentación semántica y de proximidad. Además, también se ha modificado el control del acelerador y freno para que la conducción con teclado sea mucho más suave y realista.

2.3.1.3.3. Cliente Python para la NVIDIA Jetson AGX

Para recibir datos en nuestro sistema es necesario conectar nuestra aplicación con el simulador CARLA. Este proceso es bastante sencillo puesto que lo único que debemos hacer es utilizar la API de cliente de CARLA para conectarnos. Sin embargo, la API de conexión con CARLA no está disponible en los repositorios para la arquitectura de la NVIDIA Jetson AGX por lo tanto tendremos que compilarla manualmente.



Figura 2.14: Interfaz de la aplicación de conducción modificada

Para realizar la compilación de la API de conexión deberemos obtener el código fuente del simulador, el cual se puede obtener en el repositorio oficial en GitHub [?]. Una vez hayamos obtenido el código fuente y nos hayamos asegurado de tener todas las dependencias cubiertas procederemos a compilar la API para obtener el archivo `egg` que posteriormente podremos instalar y de esta forma se nos permitirá importar el módulo `carla` desde cualquier archivo python.

Una vez se haya instalado la API podremos acceder a los datos generados por el simulador desde nuestro dispositivo tal y como se describe en la sección 2.3.2.

2.3.1.4. Entorno Windows 10 para sensor de seguimiento ocular

Uno de los principales problemas que supone la utilización del sensor de seguimiento ocular de Tobii es la incompatibilidad de este con nuestro entorno de desarrollo basado en GNU/Linux. Aunque Tobii asegura que el sensor de seguimiento ocular 4C es compatible con sistemas GNU/Linux, esto es parcialmente cierto puesto que para que sea compatible se deberá utilizar el Pro SDK, un SDK enfocado a los productos *enterprise* de la marca y restringido a tan solo varios de los productos de consumidor. En nuestro caso, el Tobii 4C se encuentra bajo el grupo de productos de consumidor lo que nos obligaría registrar el dispositivo como un dispositivo *research*, pagando la numerosa cantidad de 2160€, para poder utilizarlo con este SDK y de esta forma poder recibir y utilizar información sobre la dirección de la vista del usuario, algo totalmente fuera de nuestras posibilidades. Sin embargo, bajo sistemas Windows es posible utilizar este dispositivo utilizando el SDK general siempre que no se de un uso que incumpla con la política de uso analítico de Tobii [?].

De esta forma necesitaremos un entorno basado en Windows para poder ejecutar un servidor que nos envíe información, no sobre los datos de la vista del conductor, pues esto incumpliría la política de uso analítico de Tobii, sino la información de interacción del usuario, cuyo uso entraría dentro del uso adecuado de este dispositivo, la interacción del jugador en juegos y aplicaciones de PC.

Para cumplir con este objetivo se ha utilizado un sistema virtualizado basado en Win-

dows Server 2019 al que nos conectamos desde nuestro sistema de desarrollo mediante virt-manager y al que redirigimos el dispositivo de seguimiento ocular.

2.3.2. Obteniendo datos de los sensores

Una vez configurado el entorno de desarrollo podremos pasar a la obtención de los datos desde el simulador.

2.3.2.1. Imágenes de las cámaras RGB

La recolección de imágenes de las cámaras se realiza con el módulo de recolección de datos. Al tratarse de las imágenes de las cámaras RGB, recibiremos esta información utilizando las funciones de la clase `connectCARLAcompleto`.

Para instanciar una cámara RGB deberemos buscar la plantilla y ajustar los parámetros de esta para luego conectarla al vehículo que hemos creado con anterioridad. En el fragmento de código 2.2 se muestra la instanciación de la cámara central del vehículo.

```

1  camera_bp = blueprint_library.find('sensor.camera.rgb')
2
3  camera_bp.set_attribute('image_size_x', '640')
4  camera_bp.set_attribute('image_size_y', '480')
5  camera_bp.set_attribute('fov', '56')
6
7  camera_transform_central = carla.Transform(carla.Location(x=0.40, y=0.0, z
     =1.35))
8  camera = world.spawn_actor(camera_bp, camera_transform_central, attach_to=
     ego_coche)

```

Código 2.2: Instanciación de la cámara central del vehículo en el simulador

La posición de las cámaras puede ser modificada modificando la variable `camera_transform_central` la cual aplicará el *offset* que le indiquemos a la posición que se corresponde con el centro del vehículo. En nuestro caso, las cámaras han sido colocadas aproximadamente en la misma posición que se encuentran las del coche real.

En la figura 2.15 se puede observar la imágenes de las cámaras que recibimos desde el simulador.



Figura 2.15: Datos de las cámaras recibidos desde el simulador

2.3.2.2. Sensores OBD

En este apartado hablaremos de otros tipos de sensores que recibimos desde el simulador. Este tipo de sensores se corresponden con los que en un vehículo real podríamos obtener si conseguimos acceso al bus CAN del vehículo.

2.3.2.2.1. Velocidad del vehículo

El sensor de velocidad del vehículo, tal y como su nombre indica nos proporcionará la velocidad actual de nuestro vehículo.

Para obtener esta información utilizamos el vector velocidad que podemos obtener desde el simulador y le aplicamos la siguiente fórmula

$$3,6 * \sqrt{v.x^2 + v.y^2 + v.z^2}$$

donde $v.x$, $v.y$ y $v.z$ son los correspondientes componentes del vector recibido.

Estos datos son actualizados cada 0.03 segundos lo que permite tener gran resolución temporal y reaccionar adecuadamente ante cambios bruscos de velocidad.

2.3.2.2.2. Ángulo del volante

Para obtener el ángulo de volante podremos utilizar la función `get_control()` del objeto Vehículo de la API de CARLA. Utilizando esta función obtendremos un número entre -0.7 y 0.7 que nos indicará la posición del volante, siendo -0.7 el valor que indica que el volante se encuentra girado completamente a la izquierda y 0.7 que el volante está completamente girado a la derecha.

2.3.2.2.3. Límite de velocidad

El siguiente dato que obtendremos será el del límite de velocidad que está afectando al vehículo. La idea inicial para la detección del límite de velocidad se basaba en la detección de objetos mediante el algoritmo YOLO [?], sin embargo finalmente se decidió utilizar la información que nos devuelve el simulador ya que la implementación no se finalizó a tiempo.

2.3.2.3. Sensor de atención del conductor

Tal y como se ha comentado en la sección 2.3.1.4 este sensor se encuentra configurado en una máquina virtual con Windows Server 2019 en la que estamos utilizando un programa que detecta la atención del conductor y envía estos datos a la NVIDIA Jetson AGX Xavier.

Para la creación de este software hemos utilizado la plataforma Windows Presentation Foundation y el lenguaje C# puesto que la integración con nuestro dispositivo es directa.

La idea principal es utilizar el SDK de Tobii para comprobar si el usuario se encuentra mirando a la ventana de nuestro servidor, que será maximizada, para que de este modo se nos indique si que el conductor se encuentra atento al monitor en el que el simulador está siendo mostrado.

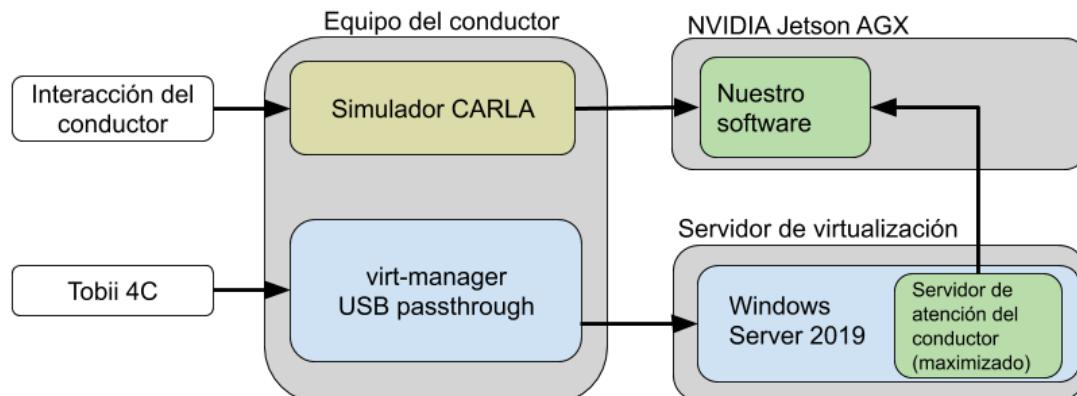


Figura 2.16: Arquitectura del sistema de control de atención del conductor

2.3.2.3.1. Estructura de la aplicación

Una aplicación WPF está generalmente estructurada en dos partes. Archivos .xaml que definen las pantallas y archivos C# que controlan la lógica de la aplicación.

En nuestro caso hemos definido una vista compuesta por varios elementos, siendo el principal un grid, capaz de reaccionar al evento `HasGazeChanged` del SDK de Tobii, que nos permitirá saber si el conductor se encuentra mirando a la ventana.

```

1  <Grid x:Name="LayoutRoot"
2    tobii:Behaviors.IsGazeAware="True"
3    tobii:Behaviors.HasGazeChanged="LayoutRoot_funciontobii">

```

Código 2.3: Grid principal de la aplicación

También hemos añadido varios cuadros de texto que para que podamos cambiar fácilmente la dirección IP y puerto del cliente al que enviaremos esta información.

```

1  <TextBox x:Name="ip_tbox" HorizontalAlignment="Left" Height="23" Margin="98,35,0,0" TextWrapping="Wrap" Text="10.0.0.6" VerticalAlignment="Top" Width="120" TextChanged="TextBox_TextChanged"/>
2  <TextBox x:Name="puerto_tbox" HorizontalAlignment="Left" Height="23" Margin="98,62,0,0" TextWrapping="Wrap" Text="1234" VerticalAlignment="Top" Width="120" TextChanged="TextBox_TextChanged_1"/>

```

Código 2.4: Cuadros de texto para selección del servidor y puerto

2.3.2.3.2. Obtención del estado del conductor

Para obtener el estado del conductor utilizamos el Grid de la aplicación, el cual, tal y como hemos comentado anteriormente, es reactivo a la visión del usuario gracias a las funciones del SDK de Tobii.

Para tener un control interno de este estado y no solamente modificar el color de nuestra aplicación utilizamos una función que se encarga de modificar una propiedad privada de la clase cuando se produce un cambio en la atención del conductor.

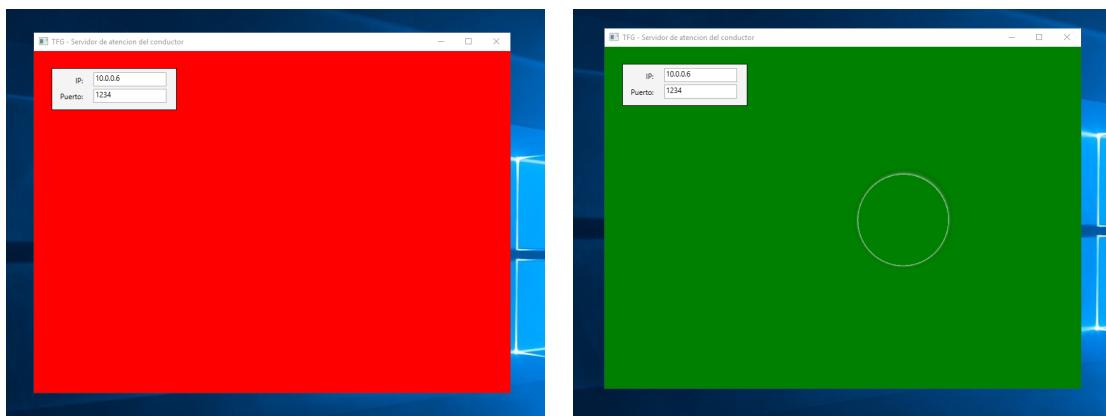
2.3.2.3.3. Envío de los datos a nuestra aplicación

Por otra parte, la función encargada de recibir los eventos de cambio de atención también llama a la función encargada de enviar estos datos a nuestro software principal. Este envío de información se produce a través del protocolo UDP. Nuestra NVIDIA Jetson AGX se encontrará escuchando en este mismo puerto a la espera de recibir el paquete enviado por el servidor de atención. Para la creación del puerto utilizamos las librerías standard de C# y la única particularidad de esta función es que se están controlando los posibles errores que puedan darse durante el cambio de servidor con un bloque try/catch.

```

1 public void enviaEstadoConductor()
2 {
3     try
4     {
5         IPAddress serverAddr = IPAddress.Parse(this.ip);
6         IPEndPoint endPoint = new IPEndPoint(serverAddr, this.puerto);
7
8         string estadoConductor = this.conductorPendiente.ToString();
9
10        byte[] buffer = Encoding.ASCII.GetBytes(estadoConductor);
11        this.servidor.SendTo(buffer, endPoint);
12        Console.WriteLine("Se ha enviado un paquete UDP: " + estadoConductor);
13    }
14    catch
15    {
16        IPAddress serverAddr = IPAddress.Parse("127.0.0.1");
17        IPEndPoint endPoint = new IPEndPoint(serverAddr, 1234);
18
19        string estadoConductor = this.conductorPendiente.ToString();
20
21        byte[] buffer = Encoding.ASCII.GetBytes[!lstlisting]{h!, language=
sharpC,caption=Función encargada de enviar los datos a la NVIDIA Jetson
AGX,label={cod:servidorUDPwpf}]
es(estadoConductor);
this.servidor.SendTo(buffer, endPoint);
Console.WriteLine("Se ha enviado un paquete UDP: " + estadoConductor);
22    }
23
24 }
25 }
26 }
```

Código 2.5: Función encargada de enviar los datos a la NVIDIA Jetson AGX



(a) Servidor de atención del conductor si el conductor no está atento. (b) Servidor de atención del conductor cuando este se encuentra atento

Figura 2.17: Servidor de atención del conductor ejecutándose. En la figura 2.17b, el círculo semitransparente representa la mirada del conductor

2.3.3. Obteniendo conocimiento a partir de los datos adquiridos

2.3.3.1. Detección de intersecciones

La primera forma de obtener conocimiento será un análisis de la vista cenital del vehículo que nos permita detectar si nos encontramos acercándonos a una intersección. Para obtener este conocimiento utilizaremos un clasificador binario entrenado sobre el dataset descrito en 2.2.3.2 en un modelo basado en una simplificación del propuesto por NVIDIA en “End-to-End Deep Learning for Self-Driving Cars”[?]. Como se puede observar en la figura 2.18 únicamente utilizamos 4 capas de funciones Conv2D y 3 capas *fully-connected* frente a las correspondientes 5 y 4 de NVIDIA.

Para realizar esta intersección creamos un módulo que se encarga de recibir la imagen cenital y realizar la clasificación. En la figura 2.19 se puede observar un ejemplo de la imagen que recibe nuestro modelo.

El entrenamiento de este modelo se realizó con los parámetros de la tabla 2.3:

Parámetros de entrenamiento del modelo	
batch_size	1024
epochs	10
validation_split	0.2

Tabla 2.3: Parámetros utilizados en el entrenamiento del modelo de detección de intersecciones.

Una vez finalizado el entrenamiento podremos comprobar los datos que se han obtenido durante este, los cuales se encuentran disponibles en la figura 2.20, y veremos que los valores de precisión sobre el subset de validación no son muy buenos. Aun así, tras un testeo extensivo el resultado de este modelo es bastante satisfactorio puesto que consigue distinguir claramente si nos encontramos acercándonos a una intersección correctamente. Los bajos valores de la precisión se podrán explicar por el pequeño número de ejemplos que nuestro dataset posee.

Una vez testeado el modelo se procede a la optimización de este, la cual convertirá el modelo en un modelo de TensorRT y permitirá la ejecución de este en la GPU de nuestra NVIDIA Jetson AGX Xavier. La optimización del modelo se realiza con la ayuda del script que se puede observar en el fragmento de código 2.6.

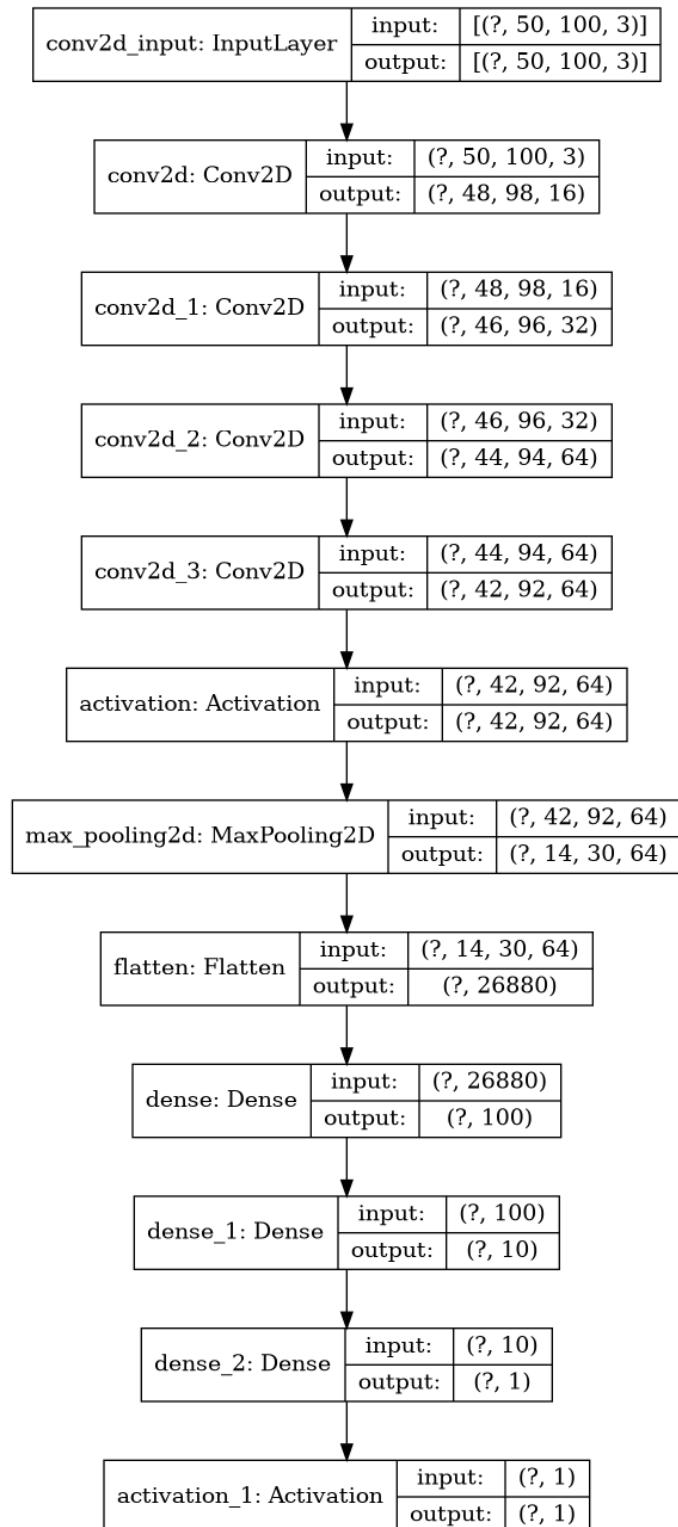
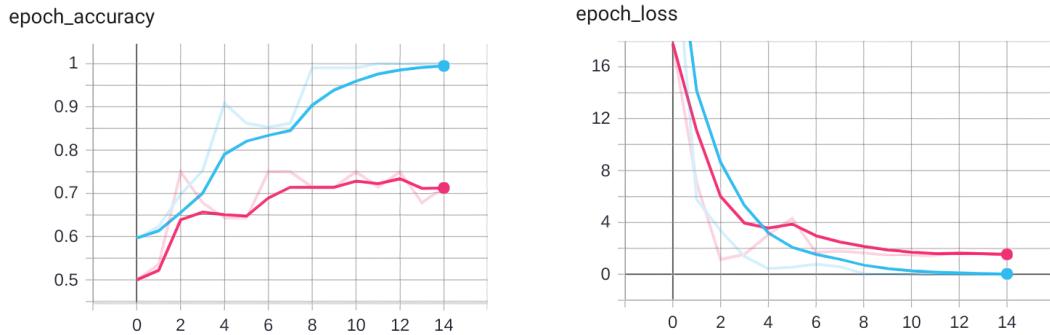


Figura 2.18: Estructura del modelo de detección de intersecciones



Figura 2.19: Ejemplo de imagen recibida por el detector de intersecciones



(a) Evolución de la precisión durante el entrenamiento
(b) Evolución de la función *loss* durante el entrenamiento

Figura 2.20: Métricas obtenidas durante el entrenamiento de la red convolucional. Las líneas azul y rosa indican la precisión y *loss* sobre el conjunto de entrenamiento y validación respectivamente.

```

1 import tensorflow as tf
2 import sys
3 from tensorflow.python.compiler.tensorrt import trt_convert as trt
4
5 path = sys.argv[1]
6
7 conversion_params = trt.DEFAULT_TRT_CONVERSION_PARAMS
8 conversion_params = conversion_params._replace(precision_mode="FP16")
9 conversion_params = conversion_params._replace(maximum_cached_engiens=100)
10
11 converter = trt.TrtGraphConverterV2(input_saved_model_dir=path, conversion_params
12     =conversion_params)
13 converter.convert()
14 converter.save(f'{path}_optimizadoTRT_FP16')

```

Código 2.6: Script utilizado para convertir un SavedModel de tensorflow a un modelo optimizado de TensorRT

2.3.3.2. Detección de objetos

Para cumplir uno de los objetivos principales del proyecto deberemos encontrar distintos tipos de objetos en la imagen de las cámaras, siendo los principales los vehículos y peatones.

Para ello, aprovechando la arquitectura modular de nuestra aplicación hemos creado un módulo que nos permita realizar la detección de objetos de una forma sencilla, únicamente deberemos pasarle la imagen de la cámara deseada y nos devolverá una lista con los objetos que ha detectado.

En cuanto a la detección de objetos en si utilizamos la librería *jetson-inference* que nos permitirá aplicar diversos modelos de inteligencia artificial de una forma muy sencilla y con gran rendimiento en los sistemas NVIDIA Jetson.

El modelo que hemos utilizado con esta librería ha sido *ssd-mobilenet-v2* que por supuesto se trata de la segunda versión del famoso modelo SSD de mobilenet entrenado en las 80 clases del *dataset COCO* [?].

Objetos del dataset COCO		
person	tie	chair
bicycle	suitcase	couch
car	frisbee	potted plant
motorcycle	skis	bed
airplane	snowboard	mirror
bus	sports ball	dining table
train	kite	window
truck	baseball bat	desk
boat	baseball glove	toilet
traffic light	skateboard	door
fire hydrant	surfboard	tv
street sign	tennis racket	laptop
stop sign	bottle	mouse
parking meter	plate	remote
bench	wine glass	keyboard
bird	cup	cell phone
cat	fork	microwave
dog	knife	oven
horse	spoon	toaster
sheep	bowl	sink
cow	banana	refrigerator
elephant	apple	blender
bear	sandwich	book
zebra	orange	clock
giraffe	broccoli	vase
hat	carrot	scissors
backpack	hot dog	teddy bear
umbrella	pizza	hair drier
shoe	donut	toothbrush
eye glasses	cake	hair brush
handbag		

Tabla 2.4: Objetos del dataset COCO

Para nuestro uso, las clases más importantes serán aquellas que se correspondan con

vehículos, personas y señales de tráfico. Para estas clases además, hemos creado una clase distinta en python para poder añadirle distintos parámetros como por ejemplo, en el caso del vehículo, si nos encontramos acercándonos a él o la posición predecida en los siguientes frames. A modo de ejemplo en el fragmento de código 2.7 se muestra el constructor de la clase vehículo.

Para realizar la inferencia llamamos a la función `deteccionRGBframe640` para las imágenes de la cámara central y `deteccionRGBframe320` para las laterales puesto que estas tienen la mitad de la resolución.

2.3.3.3. Transformación de perspectiva sobre las imágenes de las cámaras

Para obtener la vista cenital a partir de la imagen de las cámaras será necesario cambiar la perspectiva de la vista utilizando las funciones de OpenCV para este cometido.

En concreto utilizaremos la función `warpPerspective` que utilizaremos en conjunción con una matriz de transformación de perspectiva definida a partir de cuatro puntos de nuestra imagen. En la figura 2.21 se puede ver el proceso de aplicación de esta función y en el fragmento de código 2.9 se indica el código necesario para transformar la perspectiva de la camara central.

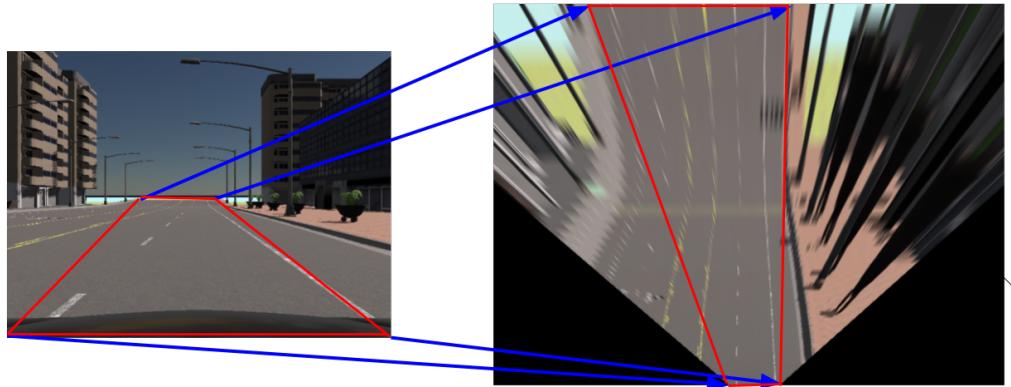


Figura 2.21: Aplicación del cambio de perspectiva

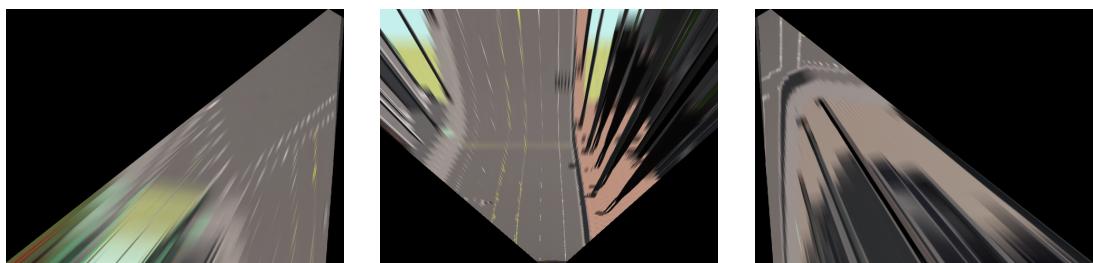


Figura 2.22: Imágenes de las cámaras con el cambio de perspectiva

Una vez obtenidas las imágenes que se pueden observar en la figura 2.22 procederemos a unirlas para obtener una vista cenital como la que se puede ver en la figura 2.23

2.3.3.4. Detección de marcas de la calzada

Una vez obtenida la vista cenital de la zona alrededor del vehículo podremos intentar detectar las marcas en la calzada.

```
1 def __init__(self, objNvidia, idLoop):
2     # Datos NVIDIA
3     self.Area = objNvidia.Area
4     self.Bottom = objNvidia.Bottom
5     self.Center = objNvidia.Center
6     self.ClassID = objNvidia.ClassID
7     self.Confidence = objNvidia.Confidence
8     self.Height = objNvidia.Height
9     self.Instance = objNvidia.Instance
10    self.Left = objNvidia.Left
11    self.Right = objNvidia.Right
12    self.Top = objNvidia.Top
13    self.Width = objNvidia.Width
14
15    # Datos personales
16    self.Centro = centroPersonal(objNvidia)
17    self.ID = idLoop
18    self.numeroFramesSinDeteccionNueva = 0 # Para controlar frames en los que no
19        se detecte
20
21    # Memorias de las ultimas 8 posiciones
22    self.memoriaTop = deque([], 8)
23    self.memoriaTop.append(objNvidia.Top)
24
25    self.memoriaBottom = deque([], 8)
26    self.memoriaBottom.append(objNvidia.Bottom)
27
28    self.memoriaRight = deque([], 8)
29    self.memoriaRight.append(objNvidia.Right)
30
31    self.memoriaLeft = deque([], 8)
32    self.memoriaLeft.append(objNvidia.Left)
33
34    # Prediccion -> Inicialmente el mismo
35    self.predictTop = int(objNvidia.Top)
36    self.predictBottom = int(objNvidia.Bottom)
37    self.predictLeft = int(objNvidia.Left)
38    self.predictRight = int(objNvidia.Right)
39
40    self.AcercandoseOalejandose = None
```

Código 2.7: Constructor de la clase personal de vehículos

```
1 def deteccionRGBframe640(self, frame):
2     # Lo convertimos a RGBA
3     frame = cv2.cvtColor(frame, cv2.COLOR_RGB2RGBA)
4
5     # Lo pasamos a CUDA para que se pueda usar con la net
6     frame = jetson.utils.cudaFromNumpy(frame)
7
8     detections = self.net.Detect(frame, 640, 480)
9
10    frame = jetson.utils.cudaToNumpy(frame, 640, 480, 4)
11    frame = cv2.cvtColor(frame, cv2.COLOR_RGBA2RGB).astype(np.uint8)
12
13    self.detections = detections
14    self.frame = frame
15    self.fps = self.net.GetNetworkFPS()
16
17    return self.detections, self.frame, self.fps
```

Código 2.8: Función encargada de realizar la inferencia con el modelo de detección de objetos

```
1 src = np.float32([[300, 250], [350, 250], [0, 480], [640, 480]])
2 dst = np.float32([[280, 0], [370, 0], [300, 480], [350, 480]])
3
4 M = cv2.getPerspectiveTransform(src, dst)
5
6 topDown = cv2.warpPerspective(camaraCentral, M, (640, 480))
```

Código 2.9: Fragmento de código encargado de realizar la transformación de perspectiva de la cámara central



Figura 2.23: Vista cenital de la zona alrededor del vehículo

2.3.3.4.1. Filtrado de la imagen para obtener marcas en la calzada

Antes de comenzar con la detección deberemos aislar las marcas en la calzada de toda la imagen. Para realizar esta separación convertimos la imagen al formato HLS y aplicamos un filtro sobre el canal L. Esto nos separará las marcas de la calzada del resto de la imagen pues estas tienen un valor de L mucho mayor que los demás píxeles.

```
1 topdownMasked = self.vistaTopDown.copy()
2 topdownMasked = cv2.GaussianBlur(topdownMasked, (3, 3), 0)
3 hls = cv2.cvtColor(topdownMasked, cv2.COLOR_RGB2HLS)
4 L = hls[:, :, 1]
```

```
5 mask_blanco = cv2.inRange(L, 130, 255)
6 topdownMasked = cv2.bitwise_and(topdownMasked, topdownMasked, mask=mask_blanco)
```

Código 2.10: Filtro utilizado para extraer las marcas de la calzada

El principal problema de utilizar este filtro es que estamos *hardcodeando* los valores del canal L que el filtro debe filtrar y por lo tanto la generalización de este procedimiento no es muy buena. Por ejemplo, el funcionamiento solo es correcto en varios de los mapas que CARLA ofrece, pero no en otros. En la figura 2.24 se puede observar un ejemplo en el que no se aíslan correctamente las marcas en la calzada.

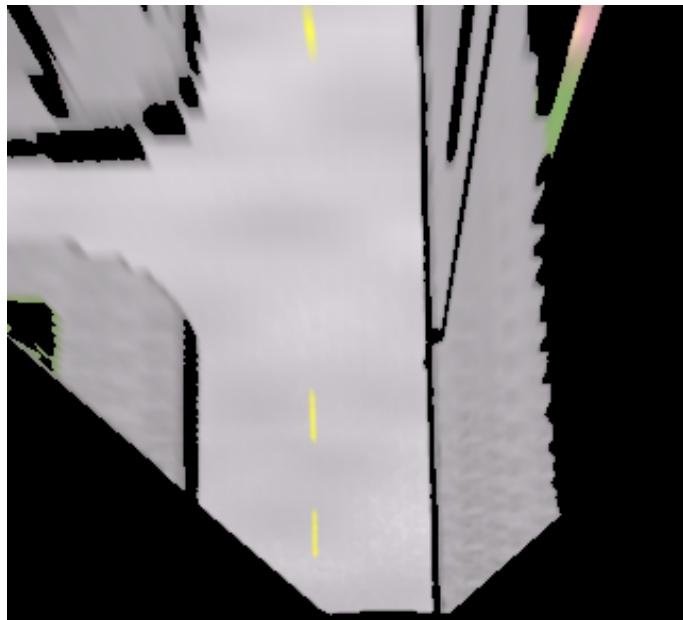


Figura 2.24: Ejemplo de aislamiento incorrecto de las marcas en la calzada

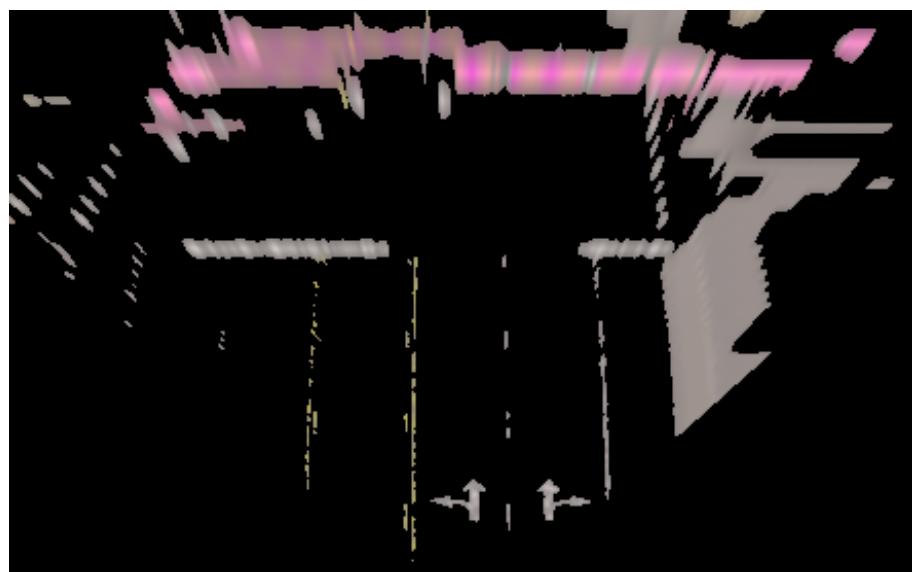


Figura 2.25: Vista cenital de la zona delante del vehículo con filtro de marcas en calzada aplicado.

2.3.3.4.2. Template Matching con OpenCV

Para la identificación de las marcas en la calzada utilizamos un sistema de detección mucho más simple. Puesto que las marcas como pasos de peatones, flechas de dirección y letras de STOP se adhieren a un standard y siempre son iguales se ha investigado la identificación de estas mediante la técnica de *Template Matching*.

Para ello, se utilizan los patrones que se pueden comprobar en la figura 2.26. El algoritmo de OpenCV se encarga de devolvernos, en el caso de que detecte algún patrón, la posición de la imagen en la que se está detectando.

En el fragmento de código 2.11 se pueden observar las llamadas a los distintos detectores y en la figura 2.27 podemos ver un ejemplo de detección correcta.



Figura 2.26: *Templates* utilizadas para el reconocimiento mediante patrones de OpenCV

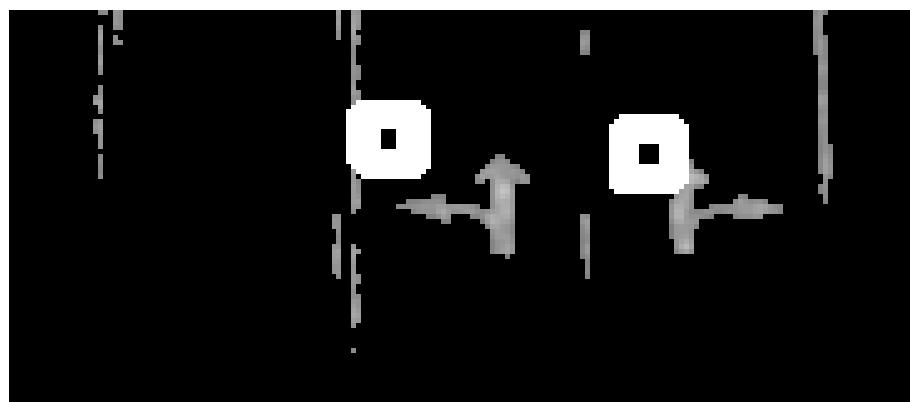


Figura 2.27: Recorte de la vista cenital. Los cuadrados indican la correcta identificación de la marca en la calzada

```

1 img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
2
3 matchsSTOP = cv2.matchTemplate(img, self.templateSTOP, cv2.TM_CCOEFF_NORMED)
4 matchsCebra = cv2.matchTemplate(img, self.templateCebra, cv2.TM_CCOEFF_NORMED)
5
6 matchsDelanteDerecha = cv2.matchTemplate(img, self.templateDelanteDerecha, cv2.
    TM_CCOEFF_NORMED)
7 matchsDelanteIzq = cv2.matchTemplate(img, self.templateDelanteIzq, cv2.
    TM_CCOEFF_NORMED)
8
9 matchsIzq = cv2.matchTemplate(img, self.templateIzq, cv2.TM_CCOEFF_NORMED)
10 matchsDerecha = cv2.matchTemplate(img, self.templateDerecha, cv2.TM_CCOEFF_NORMED
    )

```

Código 2.11: Fragmento de código utilizado para la detección de marcas en calzada

2.3.4. Analizando los datos

2.3.4.1. Clasificación del estado de los semáforos

La selección del semáforo principal, es decir, la selección del semáforo que se encuentra afectando a nuestro vehículo, es sin duda un problema con una dificultad mucho mayor que la de su detección [?]. Por esto, nuestra *approach* a este problema se ha basado en realizar una selección muy sencilla. En nuestro caso, consideraremos como semáforo principal aquel que se encuentre más cerca del punto central de la horizontal de la imagen de la cámara principal tal y como se puede comprobar en la figura 2.28. Este comportamiento simplifica considerablemente la decisión de selección del semáforo que afecta al vehículo y nos ofrece unos resultados relativamente buenos puesto que, generalmente, el semáforo que afecta a nuestro vehículo se encontrará frente a este.

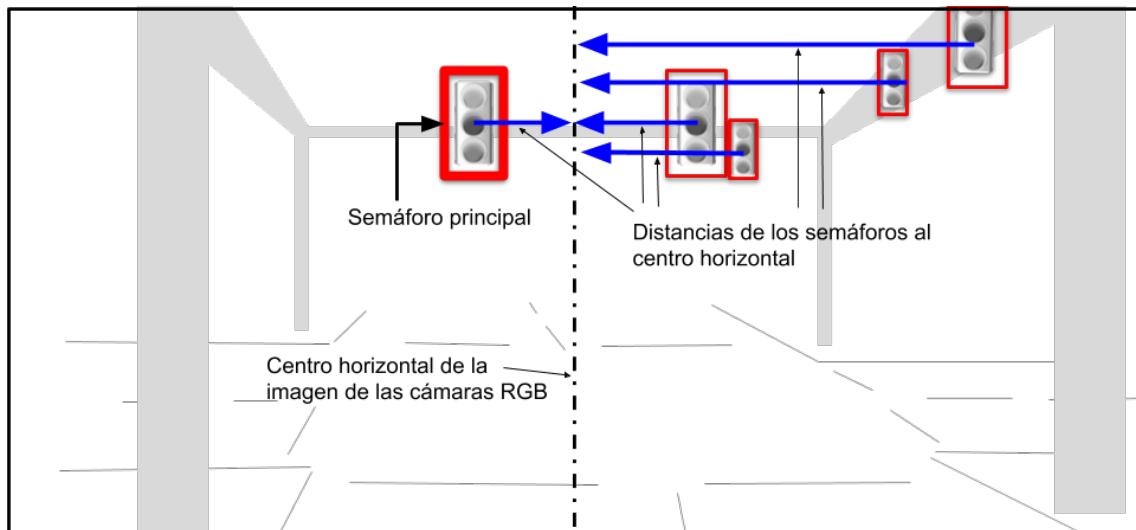


Figura 2.28: Método de selección de semáforo principal

Una vez hemos seleccionado el semáforo que deberemos considerar necesitaremos realizar un recorte de la imagen para quedarnos con la región en la que se ha detectado el semáforo. Sin embargo, debido a que la calidad de las detecciones del módulo de detección de objetos dejan bastante que desear también tenemos en cuenta un pequeño *padding* que nos permitirá asegurarnos de que el semáforo se encontrará dentro del rectángulo que recortaremos.

Una vez hayamos realizado el recorte procederemos a cambiar el tamaño de la imagen a una imagen de 25x25 píxeles, que es el tamaño que espera el modelo de clasificación.

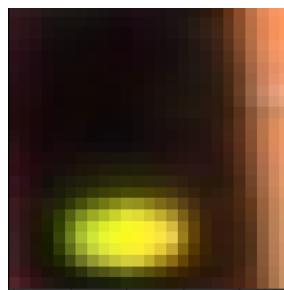


Figura 2.29: Recorte y ajuste del tamaño del semáforo a una imagen de 25x25 píxeles

2.3.4.1.1. Modelo de clasificación de semáforos

Para la detección del estado del semáforo se ha utilizado un clasificador binario entrenado sobre el dataset LISA.

La estructura de la red se compone de una simple capa de Conv2D, su correspondiente MaxPooling, dos capas de 10 neuronas *fully-connected* y como salida de la red una única neurona cuya salida se corresponde con la función de activación sigmoide.

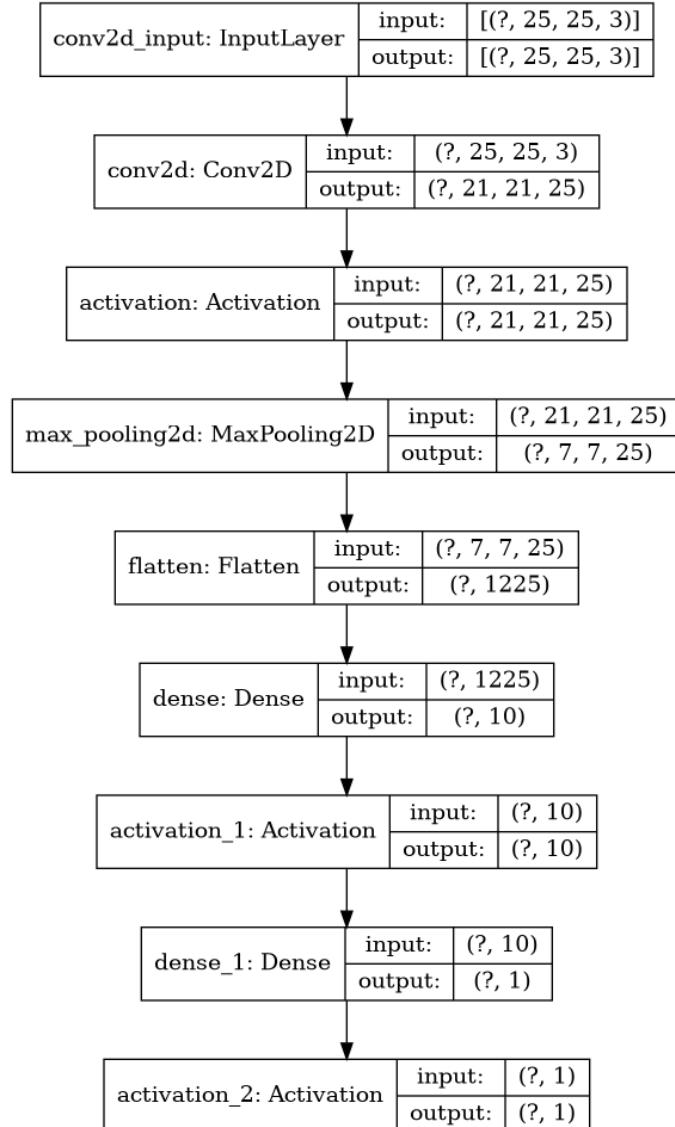
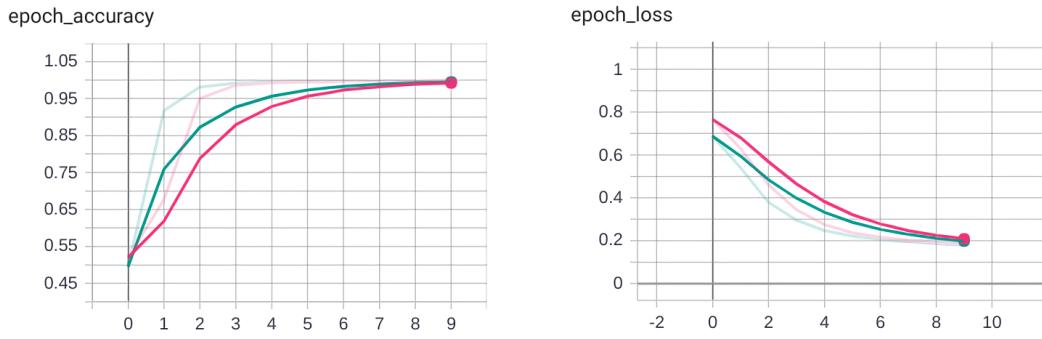


Figura 2.30: Estructura del modelo de detección del estado del semáforo

Una vez realizado el entrenamiento se obtuvieron los resultados que se pueden comprobar en la figura 2.31, los cuales son realmente buenos. La gran exactitud de las predicciones de este modelo se puede atribuir al gran dataset que hemos utilizado para entrenarlo.

A pesar de estos resultados siempre es posible que la red realice predicciones erróneas por lo que siempre devolvemos el valor más repetido de las últimas 10 predicciones.



(a) Evolución de la precisión durante el entrenamiento
(b) Evolución de la función *loss* durante el entrenamiento

Figura 2.31: Métricas obtenidas durante el entrenamiento de la red convolucional. Las líneas azul y rosa indican la precisión y *loss* sobre el conjunto de entrenamiento y validación respectivamente.

2.3.4.2. Algoritmo de seguimiento de vehículos y predicción de la posición futura

A pesar de haber realizado la detección de objetos con el módulo de detección de objetos, aún debemos tratar estos datos para intentar obtener algún conocimiento sobre estos. Para ello utilizaremos el algoritmo de seguimiento de vehículos que se explica a continuación.

El algoritmo utilizado para el seguimiento de vehículos es un algoritmo de invención propia que intenta relacionar las detecciones del frame actual con las del frame anterior y asignarle el mismo identificador. Gracias a este algoritmo podremos identificar si un vehículo detectado es el mismo que uno que ya había sido detectado anteriormente y realizar un seguimiento de su posición que nos permita obtener aún más información.

Esta relación de las detecciones con las del frame anterior se realiza teniendo en cuenta las distancias de los puntos centrales de las detecciones (tanto del frame actual como del anterior), de forma que la detección de un vehículo será relacionada con la detección del fotograma anterior (es decir consideraremos dos detecciones como un único vehículo que se ha movido y no como dos vehículos distintos) siempre y cuando la distancia entre las detecciones del frame $t - 1$ y t se encuentren lo suficientemente cerca.

En nuestro sistema se ha elegido un valor de distancia entre dos detecciones de 300 píxeles. Este valor ha sido definido mediante la experimentación.

En la figura 2.32 se puede observar un pequeño diagrama que ayudará a la comprensión del funcionamiento de este algoritmo.

A medida que vamos relacionando los vehículos con las predicciones anteriores podemos ir guardando las posiciones de estos para de esta forma tener una ‘memoria’ de posiciones que nos permitirá intentar predecir la posición futura de este.

La predicción futura de la posición del vehículo se realiza con la ayuda de la función `polyfit` de la librería numpy.

Una vez tengamos la predicción de la posición futura del vehículo podremos calcular si nos estamos acercando o alejando de él. Este cálculo se realiza comparando el ancho de la detección actual y el ancho de la predicción. Si este ancho aumenta esto indicará que nos estamos acercando al vehículo, si por el contrario el ancho disminuye esto nos indicará que el vehículo se está alejando.

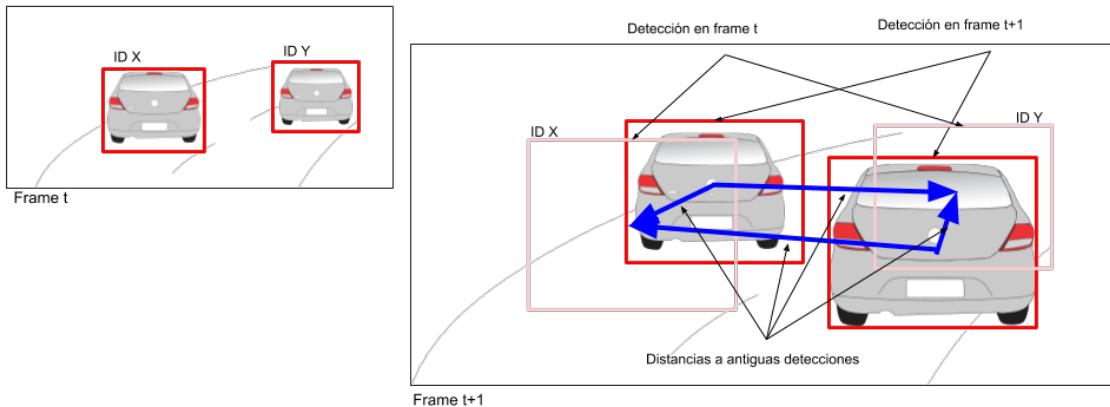


Figura 2.32: Diagrama de funcionamiento del algoritmo de seguimiento de vehículos. En este caso el vehículo de la izquierda será relacionado con el vehículo con identificador “X”. En el caso del vehículo de la derecha este será relacionado con el vehículo con ID “Y” puesto que esa antigua predicción es la más cercana.

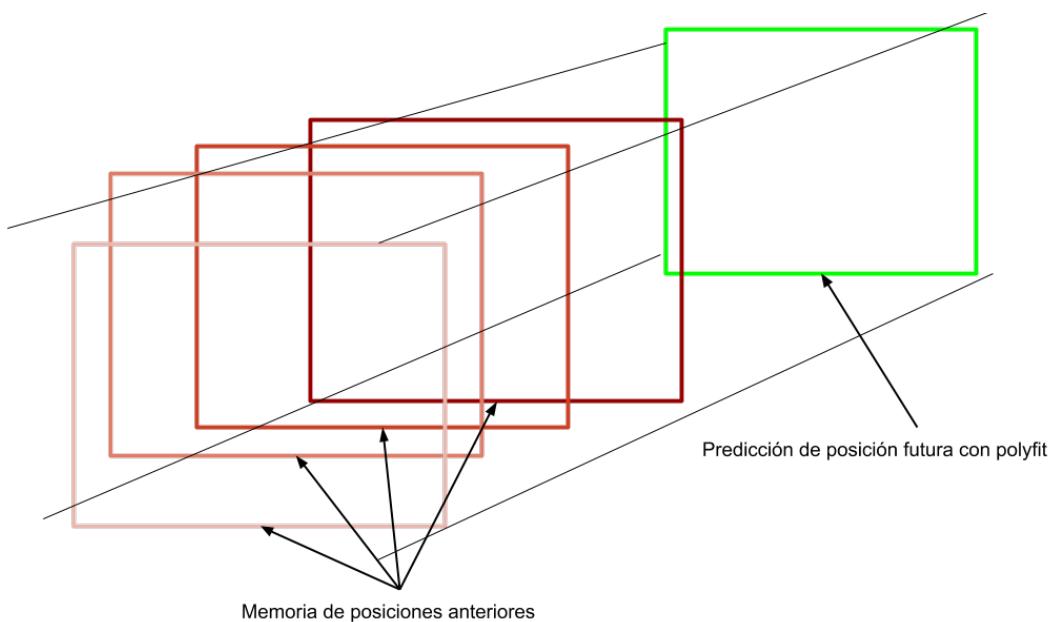


Figura 2.33: Predicción de posición futura del vehículo

2.3.4.3. Algoritmo de seguimiento de peatones

El algoritmo de predicción de la posición de peatones es similar al de la predicción de posición de vehículos. La única diferencia importante es que las detecciones de entrada se corresponden con las de los peatones y no las de los vehículos.

Si la predicción de la posición futura del peatón está lo bastante cerca del punto medio horizontal de la imagen de las cámaras activaremos el indicador visual correspondiente con el paso de peatones.

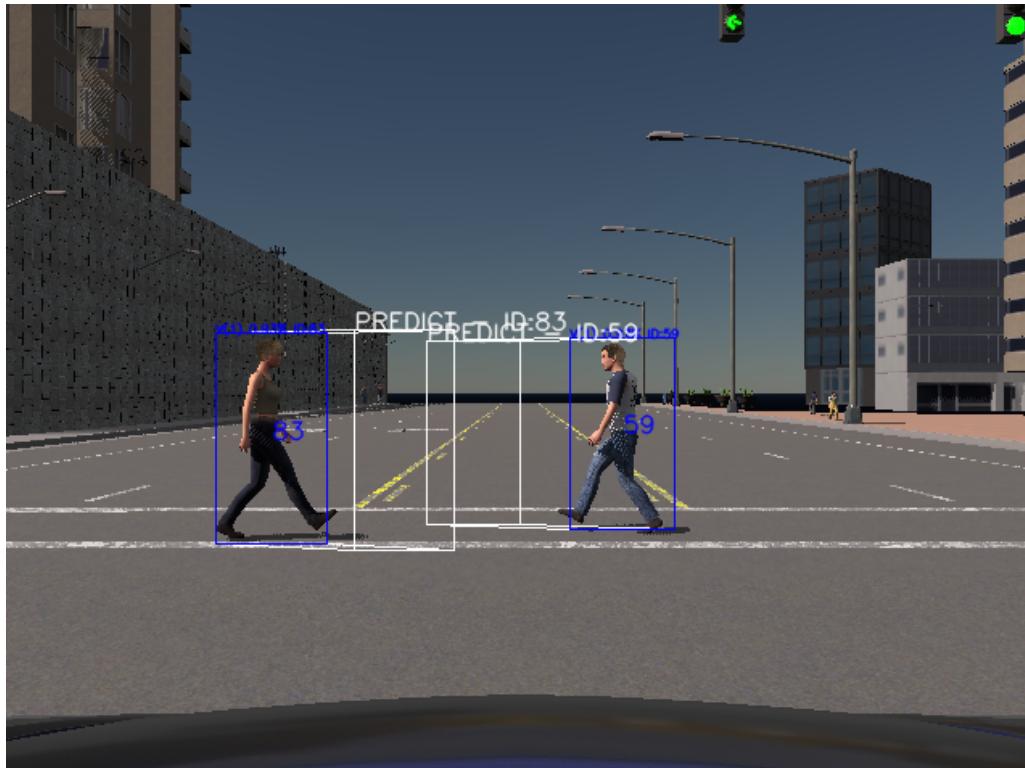


Figura 2.34: Predicción de posición futura de peatones

2.3.5. Toma de decisiones

A continuación se explica cómo se realiza la toma de decisiones que nos permite saber si se deben mostrar y reproducir notificaciones.

2.3.5.1. Control del límite de velocidad

El control del límite de velocidad es sin duda una de las decisiones más simples que nuestro sistema realiza. Esta decisión se trata de la comparación entre la velocidad actual del vehículo y la velocidad del límite de velocidad detectado por el módulo de detección. Si la velocidad actual del vehículo es mayor que la velocidad del límite de velocidad decidiremos actuar ante esta situación permitiendo que la interfaz gráfica y el módulo de notificaciones muestren y hagan sonar la notificación correspondiente.

2.3.5.2. Control de la atención del conductor

Para el control de la atención del conductor utilizamos la información obtenida desde el servidor de atención para iniciar un temporizador cuando el conductor deja de estar atento. Una vez este temporizador haya alcanzado un valor de 1.5 segundos permitimos que la interfaz gráfica muestre la notificación visual y que el módulo de notificaciones auditivas comience a reproducir un sonido hasta que el conductor vuelva a estar atento.

2.3.5.3. Aviso de cambio de semáforo

En cuanto a la notificación de cambio de semáforo esta se emite cuando se detecta un cambio del semáforo principal, desde la información que nos proporciona el módulo de clasificación de semáforos, desde el color rojo al color verde. La notificación desaparecerá cuando las cámaras dejan de ver el semáforo principal o cuando este vuelve a cambiar al color rojo. Un ejemplo de la notificación que se muestra se podrá encontrar en la figura 2.40.

2.3.5.4. Aviso de vehículo en ángulo muerto

Otro tipo de análisis que realizamos es la identificación de vehículos en el ángulo muerto del vehículo. Para ello utilizamos los datos del detector de objetos y en el caso de que detectemos algún vehículo en la imagen de las cámaras que miran hacia atrás comprobamos su área para conocer si se encuentra cerca o no. En el caso de que se encuentre lo suficientemente cerca como para que suponga un problema se activa el indicador de vehículo en ángulo muerto tal y como se puede ver en la figura 2.35.



Figura 2.35: Indicación de vehículo en ángulo muerto

2.3.5.5. Road director

El modulo *Road director* es el encargado de llevar un seguimiento de la posición del volante del conductor y compararlo con la posición predecida por una red entrenada con información del sistema *autopilot* del simulador.

De esta forma, si existe una gran diferencia entre la posición del volante del conductor y la posición que el sistema espera se activarán las notificaciones correspondientes.

El nombre *Road director* viene inspirado por el sistema *Flight director* de los sistemas *autopilot* de la aeronaves, cuya función es la de indicarle al piloto la maniobra que el sistema realizaría si este se encontrase en control del avión.

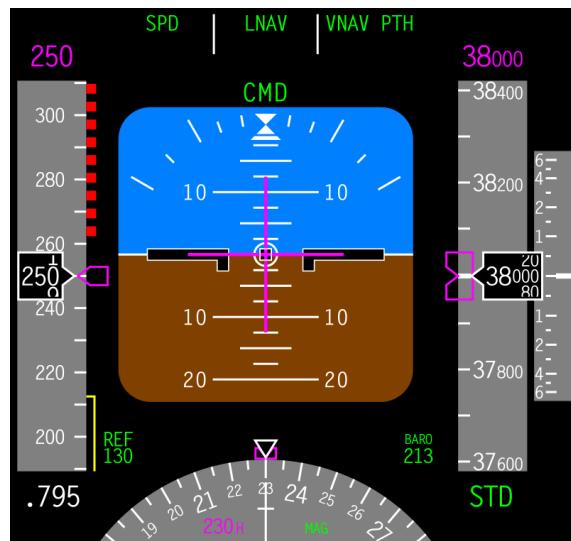


Figura 2.36: Director de vuelo (*Flight director*) en la pantalla principal de vuelo de una aeronave. La cruz de color rosa indica la maniobra que el *autopilot* del avión realizaría si tuviese el control. En este caso el avión se encuentra nivelado por lo que se mantendría el rumbo sin modificación alguna.

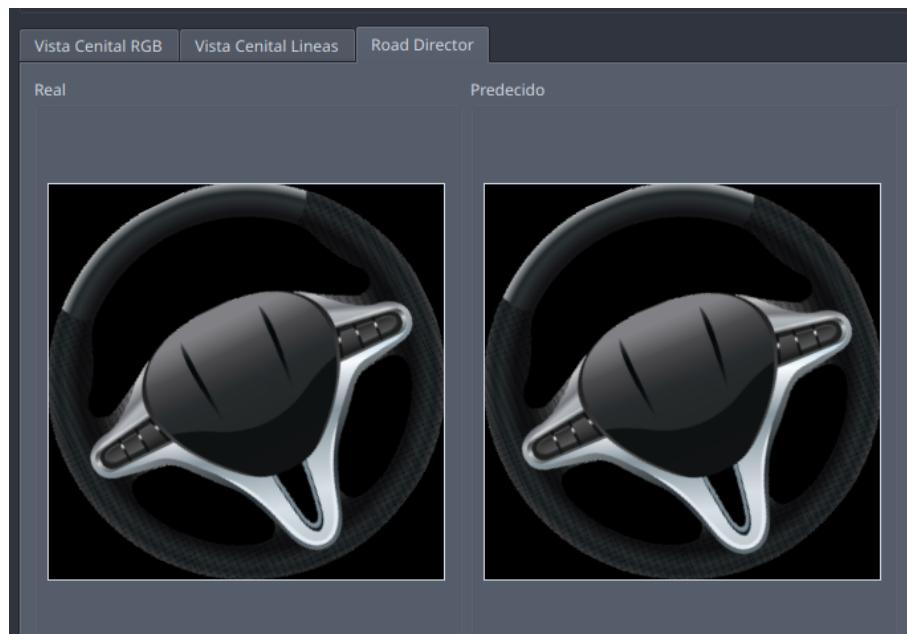


Figura 2.37: Visualización del ángulo de volante real y predecido

Para conseguir realizar la comparación entre estos dos valores será necesario obtener el valor que el sistema espera del conductor, para esto se ha utilizado una red neuronal que observando la imagen cenital de la zona inmediatamente delante del vehículo es capaz de devolvernos un valor que podemos transformar al rango comprendido entre el valor -0.7 y 0.7, que tal y como se ha visto en la sección 2.3.2 se trata del mismo rango que los datos que obtenemos del sensor de ángulo de volante.

2.3.5.1. Entrenamiento y resultados obtenidos

Esta red neuronal ha sido entrenada con el dataset explicado en la sección 2.2.3.3. Hasta el momento los resultados obtenidos no son satisfactorios y por lo tanto no se está realizando ninguna toma de decisiones que permita realizar notificaciones acústicas o visuales. El hecho de que no sean completamente satisfactorios no indica que no se encuentre funcionando correctamente, de hecho bajo ciertas circunstancias el funcionamiento es el correcto sin embargo, como no es completamente correcto finalmente se decidió deshabilitar la capacidad de enviar notificaciones.

2.3.6. Interfaz gráfica y notificaciones visuales

Una de las formas más efectivas de que el usuario se encuentre cómodo al utilizar nuestro sistema es asegurarnos de presentar la información que estamos obteniendo y analizando de una forma simple que permita al usuario comprobar que todo está funcionando correctamente. Para cumplir con este cometido, y por supuesto cumplir uno de los objetivos indicados en la sección 1.3, desde el comienzo del proyecto siempre se ha tenido en cuenta la necesidad de representar los datos en pantalla.

2.3.6.1. Interfaz inicial basada en OpenCV

Las primeras versiones de esta aplicación utilizaban como interfaz gráfica una única imagen generada con la librería OpenCV, en la que se iba superponiendo distinta información.



(a) Salida del detector de objetos



(b) Salida del detector de semáforos

Figura 2.38: Ejemplos de interfaz gráfica generada con OpenCV. En las dos imágenes anteriores se pueden observar las distintas salidas sobreuestas sobre las imágenes de las cámaras

2.3.6.2. Interfaz gráfica con Qt5

Más tarde, durante el desarrollo de la aplicación se decidió cambiar a una interfaz gráfica desarrollada con Qt que nos permitiera tener opciones más interesantes que las que podríamos conseguir con modificaciones sobre una imagen en OpenCV. En la figura 2.39 se puede observar una captura de esta interfaz durante el funcionamiento del sistema.

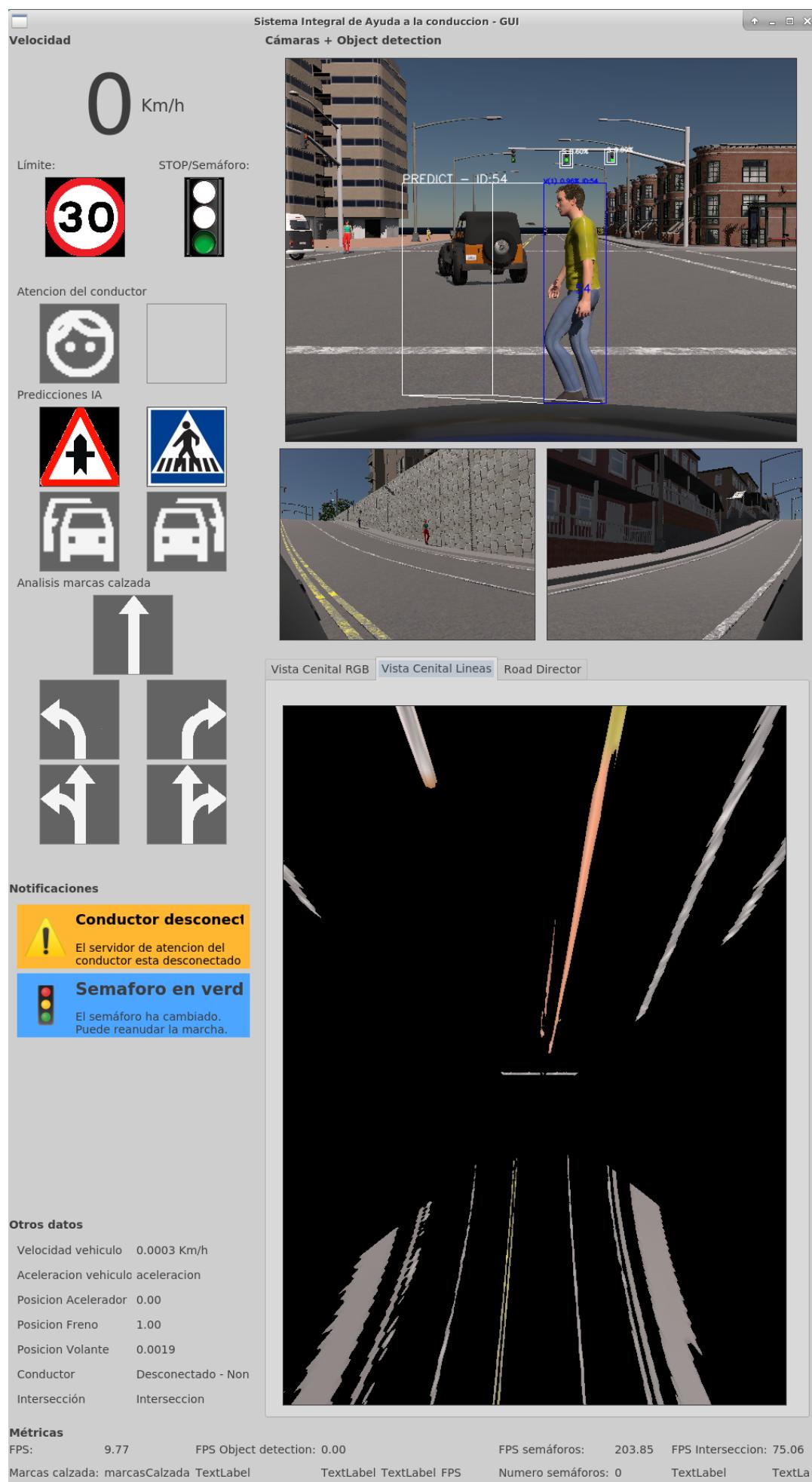


Figura 2.39: Captura de la interfaz gráfica desarrollada

2.3.6.2.1. Cámaras, detección de objetos y vista cenital

Aunque finalmente se decidiese utilizar Qt para la elaboración de la interfaz gráfica no se dejó de utilizar OpenCV para generar *overlays* en las imágenes de las cámaras. En concreto, en la nueva interfaz gráfica se utiliza gran parte de las imágenes generadas con OpenCV como por ejemplo en la vista cenital y en las cámaras RGB con la información del detector de objetos.

2.3.6.2.2. Indicadores

Los indicadores de la interfaz simulan el comportamiento de un panel de instrumentos clásico en el que estos se encienden a medida se van detectando las distintas situaciones. Un gran ejemplo de estos indicadores son el indicador de límite de velocidad, que indica el límite de velocidad obtenido desde el simulador y el indicador de semáforo, que indica el estado del semáforo que está afectando al vehículo si se detecta alguno.

2.3.6.2.3. Notificaciones

A medida que se vayan produciendo situaciones que requieran de una notificación visual estas aparecerán en la zona de notificaciones. Algunas de las notificaciones que se han considerado pueden verse en la figura 2.40.

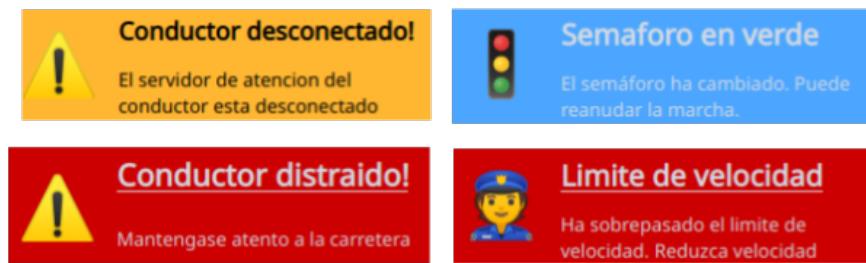


Figura 2.40: Notificaciones visuales consideradas en la interfaz

2.3.6.2.4. Road Director

En la zona de *Road Director* podemos ver una comparativa entre la posición real del volante y la posición que el sistema tomaría de acuerdo a lo descrito en la sección 2.3.5.5

2.3.7. Notificaciones auditivas

2.3.7.1. Reproducción de sonidos con playsound

Inicialmente la reproducción de sonidos se realizaba con el módulo *playsound*, sin embargo, debido a varios problemas de compatibilidad con el entorno de desarrollo se decidió cambiar y utilizar un reproductor para reproducir las notificaciones auditivas.

2.3.7.2. Reproducción de sonidos con mpv

Finalmente, para evitar problemas en el sistema de desarrollo se eligió utilizar un reproductor de audio y video para reproducir las señales acústicas. En concreto se está utilizando el reproductor *mpv* al cual se llama mediante una llamada a la función *sys.os* de python.

La clase completa que se encarga de reproducir las alertas de sonido es la que se puede observar en el fragmento de código 2.12

```
1 class notificacionesSonido:
2
3     def __init__(self):
4
5         self.sonido = None
6
7         # Thread
8         t = threading.Thread(target=self._reproduceSonidoSiHayCola)
9         t.daemon = True
10        t.start()
11
12    def _reproduceSonidoSiHayCola(self):
13        while True:
14            if self.sonido is not None:
15                print('Sonido!')
16                os.system(f'mpv resources/quindarTones/{self.sonido}.ogg')
17                self.sonido = None
18
19                time.sleep(0.1)
20
21    def reproduceAlertaQuindarStart(self):
22        self.sonido = 'start'
23
24    def reproduceAlertaQuindarEnd(self):
25        self.sonido = 'end'
```

Código 2.12: Clase encargada de la reproducción de notificaciones acústicas

2.4– Pruebas del sistema

Durante todo el desarrollo del proyecto se ha ido comprobando la correcta ejecución de todos los sistemas, pero para obtener unos resultados concretos hemos realizado distintas pruebas. A continuación, en la tabla 2.5 ofrecemos los resultados de estas.

Resultados de las distintas pruebas realizadas				
Tipo de prueba	Pruebas realizadas	Aciertos	Fallos	Porcentaje de aciertos
Detección del límite de velocidad	20	20	0	100 % ¹
Detección de marca de STOP en calzada	20	18	2	90 %
Detección de semáforos	50	35	15	70 %
Correcta clasificación del estado del semáforo	50	33	17	66 %
Notificación de cambio de semáforo	30	28	2	93.3 %
Reacción ante pérdida de atención del conductor	20	20	0	100 %
Correcta predicción de intersección	50	42	8	84 %
Correcta detección de marcas en la calzada	70	59	11	84.29 %

Tabla 2.5: Resultados obtenidos durante las pruebas del sistema

A continuación se destacan los puntos más importantes acerca de los aciertos y fallos obtenidos en las pruebas.

- **Detección de marca de STOP en calzada:** Cuando el vehículo se acerca a una marca de STOP desde un ángulo no natural el detector tiene problemas para detectar la marca.
- **Detección de semáforos:** El detector de objetos tiene problemas distinguiendo los semáforos que tienen algún edificio detrás. Algunas veces también detecta, erróneamente, semáforos en las copas de los árboles. Bajo ciertas circunstancias, el rectángulo de la predicción aparece en una posición un poco alejada y no contiene al semáforo.
- **Notificación de cambio de semáforo:** La notificación de cambio de semáforo no se lanza si se pierde la detección del semáforo en el momento en el que este cambia.
- **Correcta clasificación del estado del semáforo:** Cuando la zona recortada no contiene un semáforo la mayoría de las veces se considera un semáforo en verde. Si el semáforo es muy pequeño también existen problemas al detectar el estado.
- **Correcta predicción de intersección:** Para algunos tipos de intersecciones extrañas el detector no es capaz de reconocerlas correctamente, algo razonable debido a la poca cantidad de imágenes del dataset que se ha utilizado.

Otro de los puntos interesantes a destacar es la velocidad de procesamiento que se obtiene al ejecutar el sistema en el dispositivo NVIDIA Jetson AGX Xavier. Con el perfil de consumo máximo de energía, el cual se corresponde con 30W, obtenemos aproximadamente unos 15 FPS con picos de 18 FPS un valor muy bueno puesto que nos mantenemos en un margen que se puede considerar *real-time* y que, además, se asimila a la velocidad de procesamiento de otras soluciones con un mayor consumo de energía como por ejemplo el sistema Tesla *autopilot* (en la figura 1.1a, obtenida de [?], se puede observar como este sistema se mantiene entre 13 y 20 FPS. Este sistema está diseñado con una restricción de presupuesto de energía de 100W [?]).

¹El porcentaje de acierto es un 100 % porque se están utilizando los datos que nos aporta el simulador

CAPÍTULO 3

Planificación del proyecto

A continuación se realiza una explicación extensiva de la planificación inicial y final de nuestro proyecto.

3.1– Planificación temporal inicial

En la figura 3.1 se puede observar el diagrama de Gantt con la planificación inicial de este proyecto.

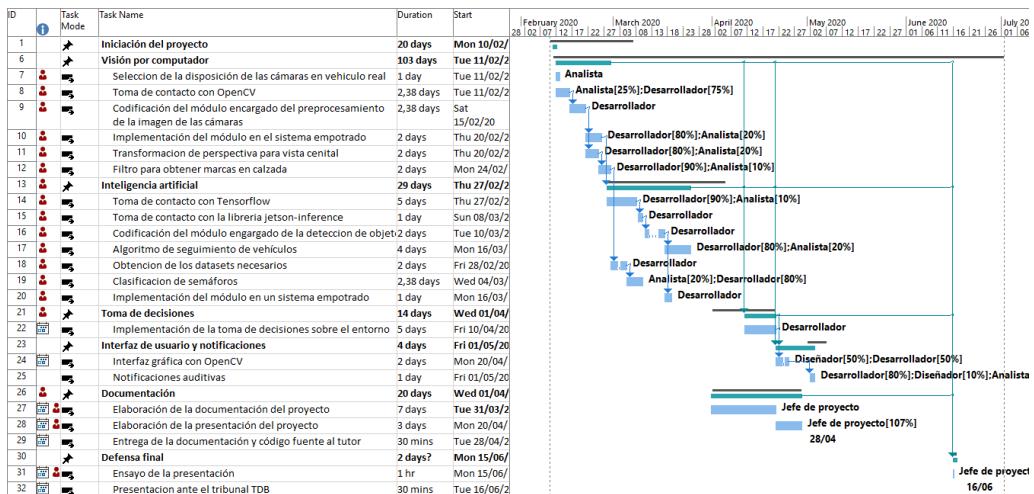


Figura 3.1: Diagrama de Gantt de la planificación inicial

3.2– Planificación financiera inicial

En cuanto a la planificación financiera inicial, se esperaba que el desarrollo del proyecto costase aproximadamente unos 20.125€ y que tuviese una evolución como la que se puede observar en la figura 3.2

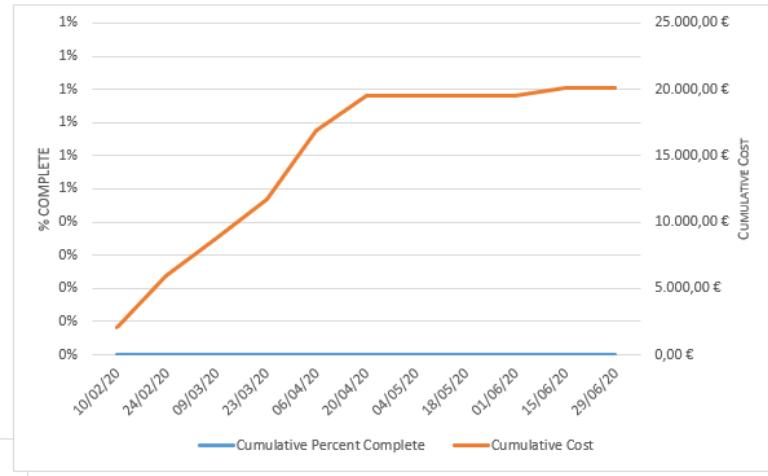


Figura 3.2: Diagrama de planificación de gastos inicial

3.3– Retrasos provocados por diversos motivos y cambios en el trabajo

A pesar de haber planificado la realización de las tareas del proyecto de acuerdo a lo descrito en los apartados anteriores, a medida que se iba realizando el proyecto se fueron encontrando diversas dificultades que impidieron el seguimiento de la planificación.

La primera de ellas fue el gran retraso que se produjo tras la imposibilidad de acceder al hardware de desarrollo debido al estado de alarma nacional decretado debido a la crisis sanitaria global provocada por el virus SARS-CoV-2. Tras los retrasos provocados por la falta del hardware se decidió aplazar la fecha de entrega de este trabajo a la siguiente convocatoria.

Durante el periodo de cuarentena también se decidió migrar la ejecución del sistema desde un vehículo real a un simulador ya que, debido a la situación, depender de la conducción de un vehículo real no parecía ser una opción razonable.

Finalmente otro de los problemas encontrados durante la elaboración del proyecto fue el cambio de dos a un único autor, lo que supuso una mayor carga de trabajo ya que, como se ha observado, incluso la planificación inicial del trabajo se trataba de una planificación para 2 personas.

3.4– Planificación temporal final

Finalmente, tras los cambios, la planificación temporal al finalizar el trabajo coincide con la que se puede observar en la figura 3.3.

3.5– Planificación financiera final

La planificación financiera final del trabajo se ha visto modificada para coincidir con la nueva planificación. Tal y como se puede comprobar en la figura 3.4 la cifra final asciende a 39.546 €.

3.6– Precios utilizados

Los precios utilizados para el cálculo de la planificación financiera han sido un redondeo de los precios medios que se pueden observar en [?]. A modo informativo, en la tabla 3.1 se pueden observar cuales han sido estas cantidades.

3. Planificación del proyecto

53

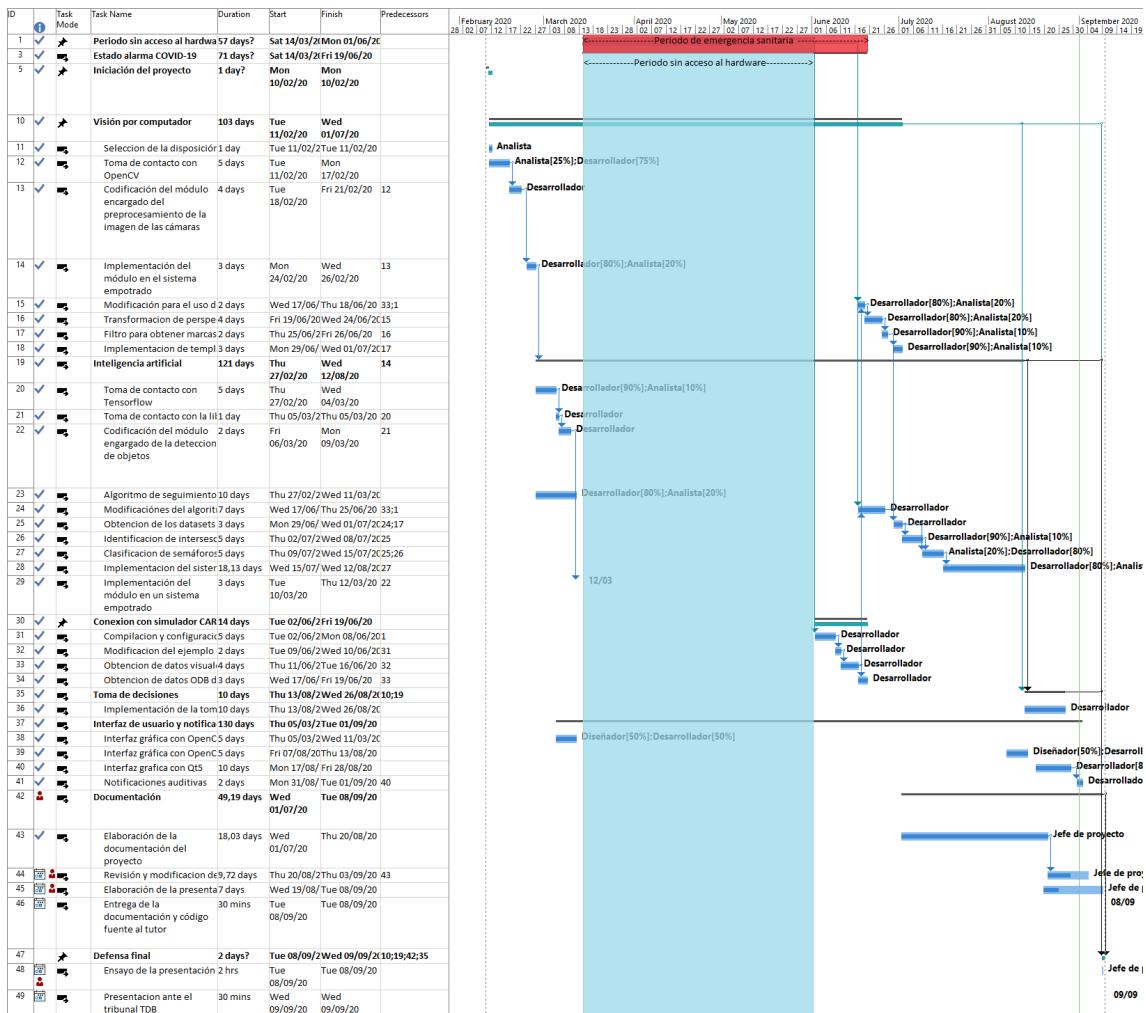


Figura 3.3: Diagrama de Gantt de la planificación final

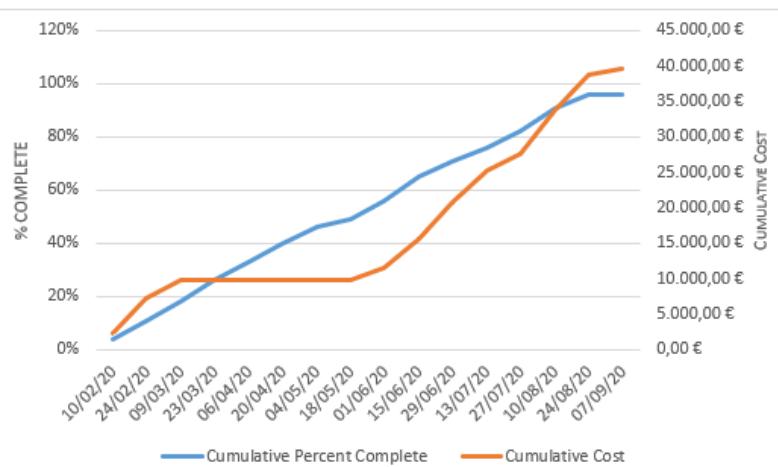


Figura 3.4: Diagrama de planificación de gastos final

Posición	Salario/hora
Jefe de proyecto	37 €
Analista	30 €
Desarrollador	25 €
Diseñador	31 €

Tabla 3.1: Precios utilizados para el cálculo de la planificación financiera

3.7– Estudio de mercado

3.7.1. Clientes potenciales

Los principales clientes potenciales de este sistema serán las empresas automovilísticas que no dispongan de un equipo centrado en el desarrollo de sistemas de conducción y que estén dispuestos a considerar una opción como la nuestra.

3.7.2. Plan de comercialización

Teniendo en cuenta el precio final del sistema desarrollado y considerando lo que aportaría a una empresa automovilística, para la cual los precios de los extras suelen ser abismales creemos que el precio final al que se debería vender un sistema parecido al desarrollado sería de 50000 €, lo que nos proporcionaría un beneficio de 10454 €. Por otra parte, para la empresa el precio de este sistema sería bastante competitivo. Si Suponemos que un acuerdo con NVIDIA le permitiría obtener el hardware a un precio reducido de 500 € podrían recuperar la inversión en apenas la venta de 100 vehículos si se vende nuestro software en un paquete de 1000 €.

CAPÍTULO 4

Trabajo futuro

4.1– Aspectos a mejorar del proyecto

Tal y como se ha comentado en el capítulo anterior existen bastantes puntos que podrían haber sido mejorados durante la elaboración de este proyecto. De entre todos ellos a continuación realizaremos un pequeño comentario sobre los más interesantes.

- Detección de señales de tráfico

La detección del límite de velocidad finalmente se ha realizado obteniendo los datos de *ground truth* desde el propio simulador. Uno de los aspectos que se debería considerar para ampliar en el futuro sería la detección de las señales de tráfico para permitir el funcionamiento de este sistema fuera de un simulador.

- Mejora del algoritmo de seguimiento de vehículos

El algoritmo de seguimiento de vehículos explicado en la sección 2.3.4.2 podría ser modificado para mejorar el rendimiento así como la selección del vehículo del frame anterior con la que la nueva detección se corresponde. Para mejorar este algoritmo sería interesante relacionar las nuevas detecciones con aquellas cuya intersección del área del rectángulo detectado y el área de los rectángulos de las detecciones anteriores sea máximo.

- Mejora de la selección del semáforo que afecta al vehículo

Como se comentó en la sección 2.3.4.1 la selección del semáforo principal se basa en la distancia de todos los semáforos detectados al centro de la imagen. Este sería uno de los aspectos a considerar más interesantes pero al mismo tiempo sería bastante complicado puesto que para obtener este conocimiento necesitaríamos obtener información mucho más precisa del espacio alrededor del vehículo.

- Obtención de otros tipos de datos del exterior del vehículo

Un punto interesante que se consideró al comienzo del desarrollo y que lamentablemente se tuvo que omitir tras el cambio a trabajar con un simulador fue el uso de los micrófonos de las cámaras PS Eye para reconocer distintos tipos de señales acústicas. En un supuesto trabajo futuro de este proyecto este sería un punto bastante interesante, en especial si se combina con la posibilidad de testear nuestro sistema en un vehículo real.

4.2– Testeo en un vehículo real

Al comienzo del trabajo, antes de la decisión de cambiar a un simulador, se pensaba aplicar nuestro software a un vehículo real y permitir reaccionar ante las situaciones anómalas que se pudiera dar durante la conducción en entornos controlados. Por supuesto, tras la decisión de realizar el proyecto con un simulador desechamos esta idea. Una vez el sistema se haya mejorado sería interesante ejecutar este en un vehículo real y analizar los resultados que se obtienen.

4.3– Independencia del hardware utilizado

Otro de los puntos importantes a considerar en cuanto a trabajo futuro es la independencia del sistema con el hardware. En el estado actual, tal y como se ha comentado en la sección 2.3.1.2, el detector de objetos se ha implementado utilizando una librería de NVIDIA. En el mercado existen diversas opciones que se podrían considerar para sustituir a la librería *jetson-inference* siendo la más interesante la propia API de detección de objetos de Tensorflow la cual ha sido recientemente actualizada para hacerla compatible con Tensorflow 2.0 que, como se ha comentado anteriormente, es la versión de Tensorflow que este proyecto está utilizando.

La reescritura del módulo de detección de objetos con una librería sin dependencias de hardware nos permitiría trasladar la ejecución de nuestro sistema a otros dispositivos empotrados así como en equipos de escritorio y, con gran seguridad, se obtendrían mejores resultados.

4.4– Ejecución con un sistema mucho más potente

De reescribirse el módulo de detección de objetos tal y como se ha comentado en la sección 4.3 el siguiente paso sería la ejecución del sistema en un equipo mucho más potente para de esta forma poder comprobar si los resultados obtenidos son mejores que los que se han especificado en 2.4.

CAPÍTULO 5

Conclusiones

A continuación procederemos a la conclusión de este trabajo comentando varios puntos interesantes.

5.1– Comparativa con los objetivos y requisitos iniciales

Como se indicó en el primer capítulo existen dos tipos de objetivos que se habían propuesto al comienzo de este trabajo: los objetivos técnicos y los objetivos académicos.

Esta vez comenzaremos por los requisitos académicos, que recordemos eran los siguientes:

- Creación de una aplicación robusta y funcional que cumpla los objetivos técnicos del proyecto
- Ampliar conocimiento y profundizar en el estudio de distintas técnicas de *Machine Learning*
- Ampliar conocimiento sobre la elaboración de aplicaciones con interfaz gráfica

En este aspecto se han cumplido los requisitos sin problema alguno.

Con la elaboración de este trabajo se ha demostrado poseer los conocimientos necesarios para implementar una aplicación y al mismo tiempo se ha ampliado conocimiento en las áreas en las que se deseaba. Tras el desarrollo de este proyecto se han aprendido conocimientos esenciales acerca de *Machine Learning* gracias al uso de Tensorflow, TensorRT y jetson-inference. En cuanto al desarrollo de aplicaciones con interfaz gráfica se ha aprendido a crear interfaces basadas en Qt y la integración de esta con OpenCV es sin duda uno de los puntos más interesante de entre los que han sido aprendidos.

Respecto a los objetivos técnicos que nos planteamos al comienzo del proyecto también podemos asegurar que han sido superados.

- **Detección y seguimiento del límite de velocidad:** El límite de velocidad es recibido por nuestra aplicación y, en el caso de que lo superemos, mostramos una notificación visual y reproducimos una auditiva por lo que podemos asegurar que este objetivo ha sido cumplido.
- **Detección y aviso cambio de semáforo:** Nuestra aplicación es capaz de reconocer el estado del semáforo que afecta a nuestro vehículos y reacciona una vez este cambia de color avisándonos visual y acústicamente por lo tanto este objetivo ha sido cumplido.

- **Detección y aviso de salida de carril:** El módulo *Road director* de nuestra aplicación es el responsable de controlar que los movimientos del volante del vehículo del conductor se correspondan con los esperados. A pesar de que el sistema no funciona excesivamente bien, en casos extremos el comportamiento es el esperado y con un poco de trabajo futuro podría tratarse de un gran sistema de ayuda a la conducción e incluso un sistema primitivo de conducción autónoma.
- **Detección y aviso de vehículos en ángulo muerto:** Nuestro sistema es capaz de realizar un seguimiento de los vehículos a su alrededor por lo tanto también es capaz de detectar los vehículos en su punto muerto. Además se muestra una notificación visual cuando se detecta un vehículo en alguna de las cámaras que apuntan hacia atrás. Por lo tanto podemos afirmar que se ha cumplido este objetivo.
- **Detección y aviso de peatones en la trayectoria del vehículo:** Al igual que con el tercer objetivo este necesitaría de un poco de trabajo futuro para poder ser considerablemente mejorado. El principal problema aparece a raíz de los problemas de detección del modelo de detección de objetos cuya precisión no es lo suficientemente buena como para realizar una predicción al igual que con los vehículos. Sin embargo, el módulo de detección de objetos consigue detectar y predecir la posición de las personas y se muestra una icono en nuestra aplicación si se predice que un peatón entrará en la trayectoria del vehículo luego el objetivo ha sido cumplido.
- **Satisfactoria implementación del sistema en un sistema empotrado:** El sistema se encuentra ejecutándose en un dispositivo NVIDIA Jetson AGX Xavier por lo tanto este objetivo ha sido cumplido.

5.2– Aspectos a mejorar

Como se ha visto en el capítulo anterior existen bastantes puntos a mejorar en este trabajo y esto puede hacer que parezca que el resultado final no se corresponde con el esperado. Sin embargo, hay que tener en cuenta que desde el principio la planificación del trabajo ha sido para dos personas y finalmente el trabajo ha sido desarrollado por una única persona por lo que es más que razonable que algunos puntos hayan quedado peor que otros y necesiten un poco de trabajo futuro para llegar al nivel que se esperaba.

Aún así se han cumplido los objetivos por lo que la valoración neta es positiva.

5.3– Impresión personal del proyecto

A continuación se recoge la opinión personal del autor respecto a la elaboración del trabajo.

5.3.1. Aspectos positivos

Puesto que los objetivos técnicos de este trabajo han sido cumplidos es más que razonable considerar que la impresión personal es completamente positiva.

El principal aspecto positivo de este trabajo ha sido la adquisición de nuevo conocimiento respecto a técnicas de inteligencia artificial y creación de interfaces aunque sin duda no se puede dejar de lado la capacidad de planificación de este ya que se ha conseguido realizar un trabajo inicialmente planificado para dos personas por una única persona.

5.3.2. Aspectos negativos

Sin embargo, también hay aspectos negativos que se deben tener en cuenta. El principal aspecto negativo han sido los retrasos que se han ido acumulando durante la elaboración del trabajo los cuales han impedido cumplir con los plazos que se esperaban y nos han obligado a retrasar las entregas. también se podría añadir a esta lista los aspectos a considerar como trabajo futuro, siendo el más negativo la falta de implementación del sistema en un vehículo real, que era la idea inicial de este trabajo.

Aun así, en general, mi valoración personal es completamente positiva.

Apéndices

En la siguientes páginas se encuentran los distintos apéndices de este trabajo, entre los que se incluyen el acta de adjudicación así como cualquier otro documento o información adicional que se haya referenciado anteriormente y se crea necesario.

Acta de adjudicación

.pdf” .png” .jpg” .mps” .jpeg” .jbig2” .jb2” .PDF” .PNG” .JPG” .JPEG” .JBIG2” .JB2” .eps”