UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Detección de Anomalías en Vehículos IoT con *Machine Learning*

Autor
Jose Antonio Marqués Ponce

Director
Jorge Navarro Ortiz

Co-Director
Felix Delgado Ferro

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Junio de 2025

Detección de Anomalías en Vehículos *Internet of Things* (IoT) con *Machine Learning*

Detección de colisiones de un vehículo en un entorno simulado con RNN

Autor
Jose Antonio Marqués Ponce

Director
Jorge Navarro Ortiz

Co-Director
Felix Delgado Ferro

Título del Proyecto: Detección de Anomalías en Vehículos IoT con *Machine Learning*

Jose Antonio Marqués Ponce

Palabras clave: Conducción autónoma, Detección de colisiones, Redes neuronales, Entorno simulado, *Car Learning to Act* (CARLA), *Light Detection and Ranging* (Lidar), *Long Short-Term Memory* (LSTM)

Resumen

En los últimos años se ha apreciado un avance considerable en la conducción autónoma. Sin embargo, para que esta pueda ser segura y circular con total libertad, es necesario que sea capaz de detectar y evitar en la medida de lo posible las colisiones. Las nuevas tecnologías ofrecen una capacidad de respuesta mucho más rápida que la de un humano, por lo que se podría alcanzar un nivel de seguridad superior al que nos encontramos hoy día con los sistemas actuales.

En este proyecto, se utilizan las redes neuronales para predecir posibles colisiones de un vehículo. Para ello, se utilizan redes LSTM junto con un entorno simulado, en concreto CARLA. Se han diseñado diferentes propuestas para lograr los resultados esperados. Cada una de estas propuestas, denominadas fases, utiliza combinaciones de sensores y modelos diferentes que obtienen distintos resultados.

En la última parte de este trabajo se analizan y se muestran los resultados obtenidos y se exponen futuras líneas de trabajo para ampliar la capacidad del sistema diseñado.

Project Title: Anomaly Detection in IoT Vehicles with Machine Learning

Jose Antonio Marqués Ponce

Keywords: Autonomous driving, Collision detection, Neural networks, Simulated environment, CARLA, Lidar, LSTM

Abstract

The last few years have seen considerable progress in autonomous driving. However, for autonomous driving to be safe and to be able to drive freely, it needs to be able to detect and avoid collisions as far as possible. New technologies offer a much faster response time than that of a human, which could lead to a higher level of safety than is currently the case with current systems.

In this project, neural networks are used to predict possible vehicle collisions. To do this, LSTM networks are used together with a simulated environment, specifically CARLA. Different proposals have been designed to achieve the expected results. Each of these proposals, called phases, uses different combinations of sensors and models that obtain different results.

In the last part of this work, the results obtained are analysed and shown, and future lines of work to extend the capacity of the designed system are presented.

Yo, **Jose Antonio Marqués Ponce**, alumno de la titulación TITULACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 45925421X, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Jose Antonio Marqués Ponce

Granada a 16 de Junio de 2025 .

D. **Jorge Navarro Ortiz**, Profesor del Área de Ingeniería Telemática del Departamento Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

D. **Félix Delgado Ferro**, Estudiante de doctorado en el Departamento Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado *Detección de colisiones de un vehículo en un entorno simulado con técnicas de aprendizaje automático*, ha sido realizado bajo su supervisión por **Jose Antonio Marqués Ponce**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 16 de Junio de 2025.

El director:

Jorge Navarro Ortiz **Félix Delgado Ferro**

Agradecimientos

Me gustaría agradecer a mi familia por haberme apoyado desde el comienzo de este trabajo y a lo largo de los seis años de carrera. Siempre han confiado en mí, apoyándome en todas mis decisiones.

Del mismo modo también quiero agradecer a mis compañeros por ofrecerme su ayuda cuando la he necesitado y por haber compartido conmigo este camino dentro y fuera de la universidad.

Por último, me gustaría agradecer a D. Jorge Navarro Ortiz y D. Félix Delgado Ferro por su ayuda para guiarme a lo largo de todo el trabajo, por su tiempo y disponibilidad. Sin ellos no habría sido posible este trabajo.

Muchas gracias por todo.

Índice general

1. Introducción **19**
- 1.1. Contexto . . . 19
- 1.2. Motivación . . . 21
- 1.3. Objetivos . . . 21
- 1.4. Metodología para la fase de diseño e implementación . . . 22
- 1.5. Estructura de la memoria . . . 23

2. Estado del Arte **25**
- 2.1. Conducción autónoma . . . 25
- 2.2. Detección de Colisiones . . . 26
  - 2.2.1. Reconocimiento del entorno . . . 26
  - 2.2.2. Métodos para la detección de colisiones . . . 27
- 2.3. Simuladores . . . 29

3. Fundamentos **31**
- 3.1. Redes Neuronales . . . 31
- 3.2. Redes Neuronales Recurrentes . . . 34
  - 3.2.1. Redes neuronales LSTM . . . 34
  - 3.2.2. Implementación de las *Recurrent Neural Network* (RNN) 35
  - 3.2.3. Trabajar con RNN en series temporales . . . 36
- 3.3. Consideraciones de técnicas utilizadas . . . 39
  - 3.3.1. Distancia y tiempo de detención de un vehículo . . . 39
  - 3.3.2. Curva precisión-*recall* . . . 40

4. Herramientas **43**
- 4.1. CARLA . . . 43
- 4.2. Google Colab . . . 44
- 4.3. Librerías importantes . . . 45

5. Planificación **47**
- 5.1. Planificación temporal . . . 47
  - 5.1.1. Objetivos logrados y planificación temporal final . . . 48
- 5.2. Recursos y costes . . . 50
  - 5.2.1. Recursos *Software* . . . 50

5.2.2. Recursos *Hardware* 51
5.2.3. Recursos Humanos 51
5.2.4. Costes totales 52

6. Diseño e Implementación **53**
6.1. Diseño de las fases 53
6.1.1. Fase 1 54
6.1.2. Fase 2 59
6.1.3. Fase 3 66
6.2. Implementación en CARLA 69
6.2.1. Escenario libre 70
6.2.2. Escenarios personalizados con *Scenic* 70
6.3. Desarrollo de los modelos 73
6.3.1. Formato de los datos 73
6.3.2. Transformación de los datos 74
6.3.3. Implementación de los modelos 80
6.3.4. Ajuste del modelo, para la fase 3 82
6.4. Diseño e Implementación del sistema anti-colisión 83
6.4.1. Objetivo inicial 83
6.4.2. Tipos de colisiones existentes 83
6.4.3. Planteamiento inicial 83
6.4.4. Diseño Final 86
6.4.5. Implementación con *Scenic* 86

7. Pruebas y resultados **89**
7.1. Fase 1 89
7.2. Fase 2 93
7.3. Fase 3 94
7.3.1. Diseño final 99
7.3.2. Comportamiento del modelo en el simulador 100
7.4. Sistema anti-colisión 101

8. Conclusiones **105**
8.1. Conclusiones 105
8.2. Trabajo a futuro 106

Glosario **111**

Índice de figuras

1.1. Evolución de las personas heridas hospitalizadas y no hospitalizadas en siniestros viales. España, 1960-2023 [1]. 19
1.2. Distribución de factores concurrentes en los siniestros viales y siniestros mortales Año 2023. (Cataluña y País Vasco excluidos) [1]. 20
1.3. Siniestros viales, personas fallecidas, heridas hospitalizadas y heridas no hospitalizadas. España, 2023 [1]. 21

2.1. Principales sensores utilizados en la conducción autónoma 2.1 27

3.1. *Esquema de una red neurona artificial [20]* 31
3.2. Función de activación sigmoide (izquierda), tanh (centro), ReLu (derecha), extraída de la fuente [21] 33
3.3. Diagrama de una red recurrente [22] 34
3.4. Diagrama de una red LSTM [22] 35
3.5. Transformación a series supervisadas [24] 36
3.6. *Recursive multi-step forecasting* [24] 37
3.7. *Direct multi-step forecasting* [24] 38
3.8. *Forecasting multi-output* 38
3.9. Tiempo y distancia de detención del vehículo en función de la velocidad 39
3.10. Curva precision-recall para el primer modelo de la fase 3 . . . 41

5.1. Diagrama de Gantt de la planificación inicial del proyecto . . 49
5.2. Diagrama de Gantt de la planificación final del proyecto . . . 50

6.1. Planificación para explicar cada fase 53
6.2. Configuración fase 1 55
6.3. Mapa utilizado para generar los datos de CARLA 56
6.4. Adaptación del funcionamiento de la fase 1 57
6.5. Configuración de la fase 2 y 3 61
6.6. Datos generados por el sensor Lidar 62
6.7. Detección de objetos estáticos Lidar 65
6.8. Esquema código escenario libre 71

6.9. Esquema código con *Scenic* 72
6.10. Formato de los datos 73
6.11. Salida del método info() de Pandas 74
6.12. Histograma de la característica *Distancia a otro vehículo* . . . 75
6.13. Características usadas en un modelo de la fase 2 76
6.14. Estrategia seguida en las fases 1 y 2 78
6.15. Estrategia seguida en la fase 3 79
6.16. Forma de los datos de entrada y salida para la fase 3 79
6.17. Función de pérdida personalizada para la distancia 81
6.18. Zonas de no frenado del sistema anti-colisión 84
6.19. Diseño inicial sistema anti-colisión 85
6.20. Diseño final sistema anti-colisión 86

7.1. Modelo implementado para la fase 1 90
7.2. Resultados fase 1 91
7.3. Prueba para comprobar la calidad de los datos 92
7.4. Modelo implementado para la fase 2 95
7.5. Resultados fase 2 95
7.6. Modelo implementado para la fase 3 96
7.7. Error cometido en la variable velocidad de aproximación, fase 3 97
7.8. Error cometido en la variable distancia, fase 3 98
7.9. Diseño del sistema realizado incluyendo el sistema anti-colisión 99
7.10. Prueba 1 del comportamiento del modelo en el simulador . . 100
7.11. Prueba 2 del comportamiento del modelo en el simulador . . 100
7.12. Prueba 3 del comportamiento del modelo en el simulador . . 101
7.13. Prueba 4 del comportamiento del modelo en el simulador . . 101
7.14. Prueba 1 del comportamiento del sistema anti-colisión en el simulador 102
7.15. Prueba 2 del comportamiento del sistema anti-colisión en el simulador 102
7.16. Prueba 3 del comportamiento del sistema anti-colisión en el simulador 103
7.17. Prueba 4 del comportamiento del sistema anti-colisión en el simulador 103

Índice de cuadros

5.1. Características ordenador 51
5.2. Costes de recursos hardware 51
5.3. Costes de recursos humanos 52
5.4. Costes totales proyecto 52

6.1. Vector de características calculadas fase 1 59
6.2. Características por cada objeto fase 1 59
6.3. Vector de características calculadas fase 2 66
6.4. Características calculadas vehículo EGO fase 2 y 3 66
6.5. Características calculadas objetos móviles fase 2 y 3 67
6.6. Características calculadas objetos estáticos fase 2 67
6.7. Vector de características calculadas fase 3 69
6.8. Sistema Anti-Colisión, dimensiones zona de no frenado 84
6.9. Duración del control de emergencia del Sistema Anti-Colisión 87

7.1. Vector de características fase 1 90
7.2. Vector de características fase 2 93
7.3. Características vehículo EGO fase 2 para la prueba 94
7.4. Características objetos móviles fase 2 para la prueba 94
7.5. Vector de características fase 3 95
7.6. Matriz de confusión para el modelo de steps 1 a 5 97
7.7. Matriz de confusión para el modelo de steps 6 a 10 98
7.8. Matriz de confusión para el modelo de steps 11 a 15 98

Capítulo 1

Introducción

En este capítulo se empezará comentando el contexto actual que rodea al problema. Se explicará también la motivación, la metodología seguida, los objetivos planteados que se intentarán lograr y un resumen de la estructura de este trabajo.

1.1. Contexto

La colisión entre dos vehículos se puede también entender al siniestro vial, es decir, un accidente en el que se ven implicados al menos dos vehículos. Para conocer la situación actual podemos ver la evolución de la siniestralidad en España desde el 1960 hasta el año 2023 en el siguiente gráfico (Fig. 1.1) obtenido del informe [1] de la *Dirección General de Tráfico* (DGT).

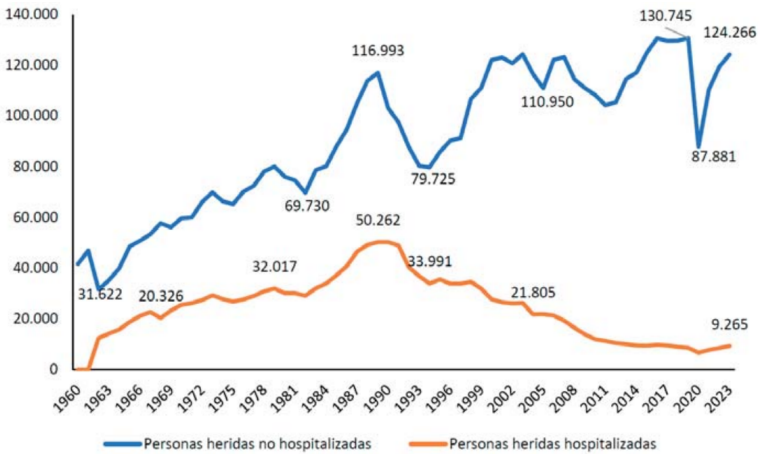

Figura 1.1: Evolución de las personas heridas hospitalizadas y no hospitalizadas en siniestros viales. España, 1960-2023 [1].

En este gráfico se observan dos tendencias. La primera con la cantidad

anual de personas hospitalizadas que ha ido disminuyendo desde los años noventa, manteniéndose mas o menos constante a partir de 2017. Y, por otro lado, la tendencia de las personas no hospitalizadas ha aumentado desde que se tienen registros. Por tanto, podemos obtener 2 conclusiones: (1) el número de accidentes o siniestros viales graves han ido disminuyendo, posiblemente debido a la mejora de los sistemas de seguridad como los *airbags*, la estructura de deformación programada que minimiza el daño en los ocupantes del vehículo o mayor uso del cinturón de seguridad entre otros. (2) El número de siniestros viales total ha ido aumentando, debido en parte posiblemente al número de conductores en España, en cualquier caso, se puede comprobar que existe un problema que todavía afecta a 124.266 españoles cada año.

Si nos fijamos en las principales causas de estos siniestros viales, en la Fig. 1.2, vemos que las más importante son las distracciones con un 17 %, seguido del consumo de drogas como el alcohol con un 13 % y, luego, aparecen otras causas como velocidad inadecuada, cansancio o no respetar la prioridad. La mayoría, por lo tanto, son causas que limitan la capacidad del conductor a reaccionar a tiempo y podrían ser evitadas con un sistema que estuviera en constante alerta.

Factor concurrente	Siniestro viales		Siniestros mortales	
	Nº de Casos	%	Nº de Casos	%
Alcohol*	3.609	13%	246	26%
Distracción	12.475	17%	409	30%
Velocidad inadecuada	5.070	7%	291	21%

Figura 1.2: Distribución de factores concurrentes en los siniestros viales y siniestros mortales Año 2023. (Cataluña y País Vasco excluidos) [1].

Una colisión o accidente puede producirse en un amplio abanico de situaciones diferentes, en función del escenarios y las causas. Por ejemplo, en situaciones como un cruce o un adelantamientos y debido a causas como la distracción de un conductor o un problema del vehículo con el pinchazo de una rueda. De esta forma, se deben estudiar cuáles son los casos más frecuentes y peligrosos. Se puede para ello, de nuevo, fijarse en los datos ofrecidos por la DGT para entender donde se suelen producir la mayoría de los siniestros viales, observando la Fig. 1.3.

Como se puede apreciar, el 65 % de los siniestros se producen en vías urbanas, de los que casi todos son en calles, que se podrían considerar principalmente representados por cruces. Por lo que este trabajo se enfocará sobre todo en simular estas situaciones donde uno de los vehículos que protagonizan el escenario se salta la señalización y se produce una colisión.

	Siniestros viales		Personas fallecidas		Personas heridas hospitalizadas		Personas heridas no hospitalizadas	
	Número	%	Número	%	Número	%	Número	%
Total	**101.306**	**100%**	**1.806**	**100%**	**9.265**	**100%**	**124.266**	**100%**
Localización								
Interurbana	35.330	35%	1.288	71%	4.345	47%	47.746	38%
Autopista	3.760	4%	92	5%	296	3%	5.656	5%
Autovía	7.727	8%	253	14%	668	7%	11.667	9%
Carretera Convencional	23.843	24%	943	52%	3.381	36%	30.423	24%
Urbana	65.976	65%	518	29%	4.920	53%	76.520	62%
Travesía	1.411	1%	31	2%	163	2%	1.694	1%
Calles	64.521	64%	486	27%	4.749	51%	74.759	60%
Autopista/Autovía urbana	44	0%	1	0,06%	8	0,09%	67	0,05%

Figura 1.3: Siniestros viales, personas fallecidas, heridas hospitalizadas y heridas no hospitalizadas. España, 2023 [1].

1.2. Motivación

Como se ha visto en los informes de la DGT, hoy día, el número de hospitalizados está en aumento, por lo que es necesario reducirlo. No existe un conductor perfecto, cualquiera puede descuidarse un momento y cometer un fallo, y las consecuencias pueden ser muy graves.

Las nuevas tecnologías representan una oportunidad para lograr reducir los accidentes. La opción de usar modelos neuronales, implementándolas como un sistema en constante alerta puede evitar que esto ocurra, aportando buenos resultados. De esta forma, una posible solución consistiría en la recopilación de información de sensores del entorno del vehículo, como una cámara o un sensor Lidar, junto con un modelo entrenado, con lo que se podría lograr reducir considerablemente los casos de colisiones, o por lo menos los daños causados por estos.

Para que la conducción autónoma llegue a ser segura es necesario que sea capaz de detectar las colisiones. De esta forma, podrá proteger tanto a los ocupantes del vehículo como los que se encuentre fuera del vehículo en la carretera. De modo que se trata de un paso indispensable.

1.3. Objetivos

El objetivo principal será tener una propuesta de solución para este problema, es decir, establecer un sistema de predicción de colisiones entre vehículos. Para ello, tendremos que plantear cómo se resolverá de principio a fin. Debido a que la idea inicial es resolverlo a través de *machile learning*, se tendrá que establecer el proceso completo para poder obtener los datos, entrenar el modelo y observar los resultados. Adicionalmente, se han añadido unos objetivos adicionales al planteamiento inicial. Estos plantean la capacidad de actuar y evitar la posible colisión a través de una mejora del

sistema. Todo esto podría dividirse en los siguientes objetivos específicos:

1. **Analizar el estado del arte**: Analizar los artículos o trabajos similares que utilicen redes neuronales para la predicción en series temporales.

2. **Proponer un diseño**: Plantear una solución con la que a través de una combinación entre sensores y un modelo entrenado se pueda predecir futuras colisiones. Esta propuesta debe ser viable con los recursos existentes, factible con el tiempo que se dispone y ofrecer garantías de que pueda obtener un buen resultado.

3. **Implementar y comprender el simulador de vehículos y tráfico**: Generar los datos que se necesiten en las condiciones deseadas, para después realizar las pruebas necesarias. Podemos dividirlo en estas tareas:

4. **Diseñar y desarrollar un modelo válido**: Entrenar un modelo que prediga con un mínimo de confianza las colisiones. Este objetivo se puede dividir en varios subobjetivos:

    a) Análisis de las redes neuronales más utilizadas y comunes en este contexto.

    b) Análisis de las características que podrían ser más útiles y factibles de obtener para las condiciones en las que se trabajará con el simulador.

    c) Estudio de cómo llevar a cabo el desarrollo del modelo una vez obtenidos los datos.

    d) Análisis de los resultados obtenidos por el modelo con el objetivo de poder mejorarlo.

5. **(Extra) Diseñar un sistema que evite la colisión**: Desarrollar una mejora sobre el sistema que actúe sobre los controles del vehículo que evite la colisión, o al menos, reduzca los daños sobre este y los ocupantes.

1.4. Metodología para la fase de diseño e implementación

Debido a que se desconoce a priori, que diseño puede funcionar mejor, la metodología de este trabajo se basará en la metodología en espiral [**metodologia˙espiral**]. Se trata de un proceso de aprendizaje y mejora continua, dividido en varias fases o diseños, donde se irán realizando cambios sucesivamente con el objetivo de obtener mejores resultados.

De esta forma cada fase tendrá un diseño distinto, sobre todo en los sensores utilizados y la información que se usará para entrenar al modelo. Cada fase estará compuesta por varias etapas que organizan el trabajo a realizar. Una vez que se tiene el diseño, las etapas que forman cada fase son:

1. **Implementación en el simulador**: se crea el código para poder comunicarse y configurarlo, de tal forma que se pueda definir el escenario y establecer los sensores que se utilizan.
2. **Generación y recopilación de datos de entrenamiento**: se ejecuta el programa y se guardan los datos.
3. **Desarrollo, análisis y pruebas en tiempo real del modelo**: se desarrolla el modelo , se analiza para medir su rendimiento o error y, finalmente, se prueba en el simulador para ver los resultados en tiempo real.

Por último, una vez que se tenga un modelo de predicción de colisiones con cierta fidelidad, se desarrollará un sistema o algoritmo para evitarlas.

1.5. Estructura de la memoria

En esta sección se describirá la estructura de capítulos seguida en esta memoria, añadiendo un breve resumen del contenido de cada uno.

1. **Introducción**: Se realiza un breve preámbulo del trabajo explicando el contexto, la motivación, la metodología seguida y la estructura del trabajo.
2. **Estado del arte**: Se explica la situación actual del problema que se está abordando, viendo las soluciones propuestas, los simuladores más usados y que redes neuronales tienen mejor rendimiento en estos casos.
3. **Fundamentos**: Se explica de forma más detallada las redes neuronales, y también consideraciones teóricas sobre partes de la solución propuesta.
4. **Herramientas**: Se detallan las herramientas más importantes utilizadas para realizar el proyecto, como son el simulador o el entorno para desarrollar el modelo.
5. **Planificación**: Se explica la planificación temporal del trabajo y los recursos y costes utilizados.
6. **Diseño e implementación**: Se describe los diseños que se han realizado, así como una breve descripción gráfica de cómo se han implementado. También se describe el desarrollo general de los modelos neuronales entrenados y el sistema para evitar la colisión.

7. **Pruebas y resultados**: Se muestran las pruebas realizadas, la capacidad del modelo para predecir las colisiones en cada una de las fases, y el rendimiento del sistema para evitar colisiones.

8. **Conclusiones**: Se comentan las conclusiones finales y las líneas futuras.

Capítulo 2

Estado del Arte

En esta sección se vera la actualidad en los distintos campos relacionados con este proyecto y la razón de las decisiones tomadas durante la realización de este.

2.1. Conducción autónoma

Hoy día la conducción autónoma está avanzando muy rápido, cada vez son más las compañías que se animan a desarrollar sus propios modelos para lograr que sus vehículos puedan conducirse sin intervención humana. Sin embargo, aún no se ha logrado en la mayoría de los casos superar el nivel 2 de autonomía y sólo algunas han llegado al nivel 3, es decir, que el conductor sigue siendo responsable y debe estar atento a cualquier incidente que pueda ocurrir.

Estos niveles de conducción autónoma vienen determinados por la *Society of Automotive Engineers* (SAE), que define en total 6 niveles de automatización [2]. Estos niveles son los siguientes:

- Nivel 0: El sistema automatizado puede realizar avisos, pero no tiene ningún control sobre el vehículo.
- Nivel 1: El sistema ofrece una asistencia al conductor, como el control de crucero o asistencia para aparcar.
- Nivel 2: El sistema tiene el control total sobre el vehículo. Sin embargo, el conductor debe estar en constante alerta. El contacto de la mano en el volante suele ser obligatorio.
- Nivel 3: El conductor puede desviar su atención, pero deberá estar atento para ciertas ocasiones donde el sistema requiera su ayuda.
- Nivel 4: Como el nivel 3, pero no requiere nunca la ayuda del conductor. Este nivel sólo está permitido en ciertas áreas o condiciones específicas.

Por ejemplo, sólo por vías bien señalizadas o con buenas condiciones meteorológicas.

- Nivel 5: Sistema autónomo total. No necesita la ayuda del conductor y puede ir por cualquier sitio en todas las condiciones.

2.2. Detección de Colisiones

Para este trabajo, ha sido necesario contemplar dos problemáticas distintas. Por un lado, las herramientas y métodos para poder reconocer el entorno y generar los datos. Como se ha mencionado anteriormente se trabaja en una simulación, por lo que se podrán probar distintas configuraciones de sensores. Por otro lado, analizar los datos para poder detectar las colisiones.

Respecto a la detección de colisiones existe una gran cantidad de documentación. Pejman Goudarzi et al. [3] realiza una comparativa de las distintas técnicas y métodos empleados, clasificándolos en distintos grupos. La idea general para detectar las colisiones suele ser el cálculo o estimación de un *Time To Colission* (TTC), que indica el tiempo estimado con el que se podría generar una colisión. Estas técnicas utilizan tecnologías como visión por computador, aprendizaje automático o *machine learning*, redes neuronales y, en algunos casos, métodos basados en reglas. Estas técnicas usan la información generada por sensores colocados sobre el vehículo autónomo que brindan la información del entorno.

2.2.1. Reconocimiento del entorno

Como se ha explicado antes, esta parte trata de recoger la información de los vehículos, peatones u otros objetos que estén alrededor del vehículo EGO, así como las operaciones que se realicen para transformar esta información en otra más útil o de mejor calidad. Hamidaoui et al. [4], discuten los sensores más utilizados en el campo de la conducción autónoma. Principalmente destacan tres: las cámaras, el sensor *RAdio Detection And Ranging* (Radar) y el sensor *Light Detection and Ranging* (Lidar). También se destaca la fusicón de varios sensores como otra opción, tal como se muestra en la Fig. 2.1. Cada sensor tiene sus fortalezas y limitaciones que impactan en su rendimiento. Por ejemplo, en condiciones de lluvia el Radar puede funcionar sin problemas, sin embargo, otros sensores como la cámara o el Lidar pueden fallar. En cambio, el Lidar es capaz de realizar un mapeo tridimensional de todo el entorno que ofrece una información muy valiosa.

- **Cámaras**: El uso de cámara es indispensable para poder reconocer algunos elementos de la carretera como pueden ser señales, peatones u otros vehículos. Por lo tanto se trata de una herramienta esencial. Sin embargo, requieren algoritmos de procesamiento intensivos como

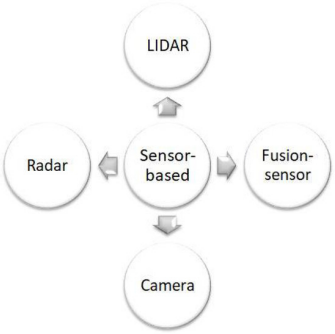

Figura 2.1: Principales sensores utilizados en la conducción autónoma 2.1

por ejemplo *Convolutional Neural Network* (CNN). Existe numerosa documentación que ayuda a mejorar el rendimiento de este sensor. Por ejemplo, en [5] proponen un método eficaz de *Posicionamiento de Vehículo a Vehículo* (V2V) a través de una cámara. El V2V es esencial en la conducción autónoma, ya que las redes neuronales necesitan conocer las relaciones espaciales entre los vehículos.

- **Radar**: Emite ondas de radio para detectar los objetos, obteniendo la distancia y velocidad a la que se encuentran, gracias al efecto Doppler. La principal ventaja es que funciona bien bajo cualquier condición climática, donde otros sensores, como la cámara, pueden fallar. En [6] se propone el sensor Radar para mejorar la precisión de V2V. Destaca la gran fiabilidad de esta herramienta frente a otros enfoques como el *Global Positioning System* (GPS) el *Global Navigation Satellite System* (GNSS) sobre todo en entornos dinámicos.

- **Lidar**: Genera un mapa tridimensional del entorno mediante pulsos láser infrarrojos. Es ampliamente utilizada en la comunidad investigadora para la estimación precisa de la posición de los vehículos. You Li et al. [7] presentan la actualidad y desafíos en el campo de la conducción autónoma en el uso de esta tecnología. A pesar de algunas limitaciones como el coste o las condiciones meteorológicas, destacan que se han propuesto numerosas soluciones y que cada vez son más los automóviles que incorporan el Lidar. En comparación con el resto de sensores, exponen que este es más preciso para medir la distancia y por lo tanto el más fiable. Por último, destacan su gran influencia en el *machine learning* y el gran crecimiento que tendrá en este campo.

2.2.2. Métodos para la detección de colisiones

A partir de los datos generados por lo sensores, los algoritmos deben interpretarlos y detectar las colisiones. Como se ha explicado antes, las

principales técnicas utilizadas son la visión por computador, aprendizaje automático y las redes neuronales profundas. Charith Chitraranjan et al. [8] propone dos grupos en los que se pueden clasificar los métodos actuales para detectar las colisiones:

- Métodos que calculan una métrica de amenaza para luego estimar las colisiones a partir de ellos. Estas métricas utilizadas son principalmente la distancia relativa, la velocidad relativa, el TTC o la combinación de estas.

- Métodos que utilizan el aprendizaje profundo directamente para predecir la probabilidad de colisión futura a partir de un vídeo como entrada.

En la mayoría de trabajos encontrados, por ejemplo [9], se emplean modelos CNN preentrenados para la detección de objetos en los fotogramas. Modelos como *YOLO v3*, *Faster R-CNN* y *SSD* [10] ofrecen muy buenos resultados que permiten detectar los objetos de manera precisa en un tiempo razonable. Cada una tiene sus ventajas y desventajas, de forma que algunas se enfocan en mejorar la precisión, pero requieren un tiempo considerable y pueden ser lentas. Y otras, en cambio, son más rápidas, pero tienen menos precisión.

Alireza Rahimpour et al. [9] proponen un sistema de predicción de colisiones con animales grandes en condiciones de poca visibilidad basado en el TTC. Usan una cámara *Cámara de Imagen Térmica* (FIR) que resaltan los animales en la oscuridad, para después detectarlos a través del modelo *Faster R-CNN*. Para predecir las trayectorias futuras de los animales emplean un modelo *encoder-decoder* con *Convolutional Long Short-Term Memory* (LSTM). Una variante de la red LSTM diseñada para datos de secuencias de imágenes o vídeos. Los resultados obtenidos por este trabajo reflejan un comportamiento muy bueno de este modelo, logrando una precisión cercana al 95 %.

Otro caso de uso de estas redes LSTM en el contexto de predicción de colisiones es el expuesto por Wentao Bao et al. [11]. Explican el proceso para detectar las colisiones entre vehículos a partir de la cámara situada en salpicadero del coche (*dashcam*). En este caso, se aplica *Graph Convolutional Networks* (GCN) para capturar las relaciones espaciales y arquitecturas recurrentes como LSTM para capturar las relaciones temporales. Por último, se estima la probabilidad de colisión mediante *Bayesian Neural Networks* (BNNs) que permiten incorporar la incertidumbre.

En resumen, se puede apreciar que el uso de las redes LSTM constituyen una técnica popular para capturar las relaciones temporales en series de datos. En concreto, muy extendida en el campo de predicción de colisiones en vehículos autónomos.

2.3. Simuladores

Un simulador es una herramienta software que crea un entorno virtual para recrear situaciones reales [12]. El objetivo es poder realizar pruebas en este entorno virtual de manera que se reduzca el tiempo y el coste económico que supondría realizarlas en la vida real. De esta forma, los simuladores tienen un papel muy importante hoy día, empleados en muchos campos.

Los simuladores de vehículos autónomos se diferencian por su complejidad, ya que deben tener un nivel de similitud frente a la vida real suficiente como para poder aceptar los resultados como válidos y fiables. Se caracterizan por poder simular entornos de tráfico realistas con varios vehículos, peatones o ciclistas, además de tener señales de tráfico o semáforos que puedan afectar al tráfico. Por último, tienen sensores que permiten recoger la información del entorno, en este sentido casi todos los simuladores permiten colocar cámaras sobre el vehículo EGO aunque, dependiendo del simulador, también dispondremos de otros sensores como Radar o Lidar.

Existen varias alternativas para simuladores de vehículos autónomos. A continuación, se verán algunos de los más importantes:

- **LGSVL**: Es un simulador [13] *Open Source* creado en *Unity* [14] con su última tecnología de *High-Definition Render Pipeline* (HDRP) que permite una alta calidad gráfica, pudiendo simular entornos virtuales fotorrealistas. Ofrece distintos escenarios o vehículos del mismo modo que ajustes del clima y de la hora del día. Aunque el soporte finalizó en 2022, aún sigue siendo una opción válida para el desarrollo de algoritmos de conducción autónoma.

- **Gazebo**: Es un simulador 3D *Open Source* más enfocado en la robótica [15]. Ofrece una gran biblioteca de herramientas de desarrollo y servicios en la nube que facilita la simulación. Al mismo tiempo construye entornos con físicas realistas que, junto a los sensores, logran crear datos de gran fidelidad. Tiene una gran comunidad que crea contenido y ofrece muchas posibilidades para poder desarrollar un amplia variedad de proyectos. Por último, ofrece compatibilidad con otras herramientas como *Robot Operating System* (ROS).

- **Carsim**: Forma parte de las herramientas que ofrece *Mechanical Simulation Corporation* [16] para la simulación de vehículos autónomos. Ofrece la posibilidad de simular vehículos, peatones o señales de tráfico y los sensores necesarios para *Automatic Driver Assistance Systems* (ADAS). Lleva desarrollándose desde hace más de 20 años por lo que es uno de los más usados y precisos para las físicas del vehículo. Sin embargo, se trata de una herramienta más sofisticada que presenta una curva de aprendizaje más pronunciada respecto al resto de simuladores.

- ***Car Learning to Act* (CARLA)**: Es un simulador *Open Source* [17] implementado sobre *Unreal Engine* [18] por lo que nos ofrece una calidad visual y de renderizado muy buena, superior al resto de simuladores. Ofrece varios mapas de ciudades y entornos rurales con una gran variedad de carreteras y cruces. Del mismo modo, ofrece muchos tipos de vehículos, ciclistas, peatones y objetos que se encuentran en la vía y que podrían afectar a la conducción de un vehículo. Al ser una herramienta tan importante existe soporte para poder utilizarla junto con otras herramientas como podrían ser ROS o *Scenic*. Soporta un gran número de sensores, cámaras y detectores con los que se puede recoger la información que se necesite. En diciembre de 2024, sacaron la versión 0.10.0, con muchas mejoras que incrementaron la capacidad de obtener datos más fiables y realistas, por lo que se trata de un proyecto que sigue avanzando. A pesar de que la curva de aprendizaje es un poco pronunciada, existe una gran comunidad gracias a ser una herramienta muy popular. Además, se ajusta perfectamente a las necesidades para realizar este trabajo, como son: (1) poder generar datos de manera fiable, (2) ofrecer entornos realistas y (3) una variedad de sensores y herramientas que facilitan en gran medida el trabajo.

Capítulo 3

Fundamentos

En este capítulo se establecerá la base teórica y conceptual de este trabajo, analizando los conceptos clave necesarios para comprenderlo.

3.1. Redes Neuronales

Una red neuronal [19] es un algoritmo o modelo de *machine learning* que se inspira en el funcionamiento del cerebro humano, imitando el comportamiento de las neuronas biológicas. Al igual que el cerebro humano, una red neuronal artificial consta de unidades o nodos conectados conocidas como neuronas artificiales. Cada una recibe señales y produce una salida de las neuronas conectadas con ella. Esta salida se calcula mediante una función no lineal de la suma de sus entradas, conocida como función de activación. Cada conexión entre neuronas tiene asociado un peso o ponderación, que indica la fuerza de la señal en esa conexión, siendo este el parámetro que se ajusta durante el entrenamiento de la red.

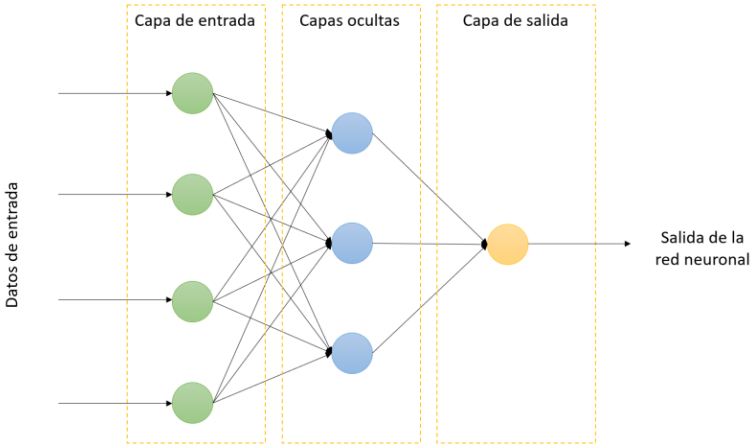

Figura 3.1: *Esquema de una red neurona artificial [20]*

Como podemos ver en la Fig. 3.1, una red neuronal, por lo general, está compuesta por distintas capas compuestas por estas neuronas artificiales: una capa de entrada, una o varias ocultas y una capa de salida. De esta forma, las señales viajan desde la capa de entrada hasta la capa de salida. Típicamente, se denomina red neuronal profunda o *deep learing* cuando se tienen tres o más capas ocultas.

Durante el entrenamiento de una red neuronal [21], una función de pérdida mide la diferencia entre la salida deseada o real para una entrada dada y la salida real obtenida de la red. En este punto se aplica una retropropagación que calcula la tasa en la que cada neurona contribuye a la pérdida total. De esta forma, se calcula el gradiente de la función de pérdida, que indica al algoritmo de optimización, como por ejemplo el descenso de grandiente, que pesos o ponderaciones ajustar para reducir las pérdidas.

Hoy día, las redes neuronales se utilizan para resolver una amplia variedad de problemas, como la visión por computador o el reconocimiento de voz. Problemas complejos de resolver con algoritmos fuera del campo del *machine learning*.

Funciones de activación

Las funciones de activación utilizadas comúnmente son:

- **La función sigmoide**: Transforma cualquier valor real en un rango comprendido entre 0 y 1. Usualmente utilizada en problemas de clasificación binaria, al poder interpretarse como una probabilidad.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- **La tangente hiperbólica** (tanh): Similar a la función sigmoide pero transforma los valores reales en un rango entre -1 y 1. Más versátiles al estar centrada en 0, ayudando a las redes neuronales a aprender eficientemente.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- **La unidad lineal rectificada** (o *Rectified Linear Unit* (ReLU)): Destaca por su simplicidad y eficacia. En este caso se transforman los valores negativos en 0 y deja los positivos igual, lo que ayuda a reducir el problema del desvanecimiento del gradiente.

$$f(x) = \text{máx}(0, x)$$

Podemos ver cada una de estas funciones en la Fig. 3.2.

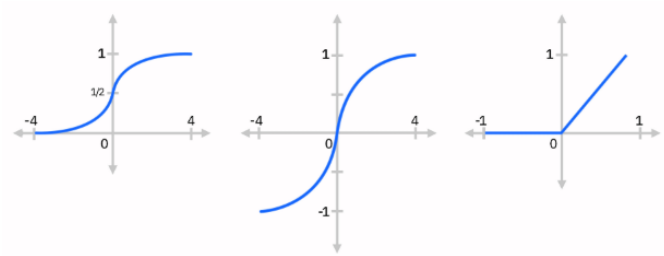

Figura 3.2: Función de activación sigmoide (izquierda), tanh (centro), ReLu (derecha), extraída de la fuente [21]

Tipos de redes neuronales

Existe una gran variedad de tipos de redes neuronales. A continuación describiremos las más comunes por uso actualmente.

- **Las redes neuronales convolucionales** (CNN): Están especializadas para procesar datos que tienen una estructura de cuadrícula, como imágenes o pistas de audio. Se inspiran en el funcionamiento del cerebro humano para procesar la información visual, resultando efectivas para reconocer patrones, clasificación de imágenes o detección de objetos. Las principales capas utilizadas en este tipo de redes son:

  - Capas de convolución: Aplica un filtro a la imagen logrando extraer las características más relevantes, como texturas, formas o bordes.
  - Capa de agrupación: Reduce la dimensionalidad de los datos, conservando la información importante, logrando limitar el sobreaprendizaje.
  - Capas totalmente conectadas: Usadas para realizar un razonamiento de alto nivel. Principalmente usadas para la clasificación final de los datos.

- **Las redes recurrentes** (*Recurrent Neural Network* (RNN)): Están diseñadas para trabajar con datos secuenciales, con el objetivo de predecir valores futuros en dicha serie. A diferencia de una red neuronal convencional, donde la información fluye desde la entradas hasta la salida sin considerar la estructura secuencial de los datos. En las RNN existen estados internos o memorias que les permiten recordar información y utilizarlos para realizar predicciones. En el siguiente capítulo abordaremos más detalladamente este tipo de red neuronal.

3.2. Redes Neuronales Recurrentes

Estas redes neuronales están pensadas para trabajar con datos secuenciales, propagándose la información a través de las conexiones recurrentes, permitiendo a la red aprender patrones temporales.

Las RNN [22] están compuestas por una unidad básica, la célula recurrente. Esta célula recibe dos entradas: (1) la entrada actual y (2) el estado oculto previo, que se podría entender como una memoria que retiene información de las iteraciones previas. Estas dos entradas conforman combinándose: (1) la salida actual y (2) el nuevo estado oculto. Este último estado oculto pasa a ser la entrada en la siguiente iteración, como se aprecia en la Fig. 3.3.

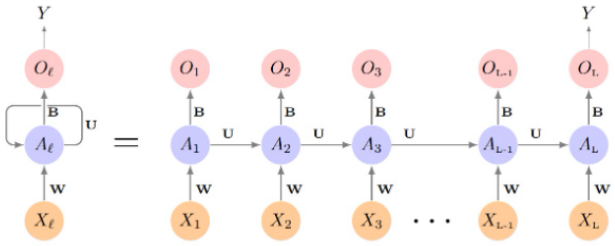

Figura 3.3: Diagrama de una red recurrente [22]

Al enfrentarse a problemas donde se necesita que la red neuronal pueda retener la información durante más tiempo las RNN no son suficientemente eficientes. Es por ello, que aparecen otras alternativas especializadas en capturar patrones en las secuencias temporales a largo plazo. Destacan principalmente las redes LSTM y las *Gated Recurrent Unit* (GRU). Las principales diferencias entre ambas son la potencia y la complejidad. Las LSTM superan en potencia y flexibilidad a las GRU, sin embargo son más complejas y lentas de entrenar, requiriendo por lo tanto más recursos.

3.2.1. Redes neuronales LSTM

En las redes LSTM, en lugar de usar neuronas, hay bloques de memoria conectados a través de las diferentes capas. Este bloque se diferencia de una neurona recurrente por tener tres puertas que gestionan que información se transfiere a su estado y a la salida, es decir, que información retener y cuál olvidar. Las tres puertas son:

- **Puerta de Olvido** (*Forget Gate*): Decide que información del estado anterior debe ser olvidada, lo que permite eliminar la información que no es útil para realizar la predicción actual.
- **Puerta de Entrada** (*Input Gate*): Decide que información nueva será almacenada en el estado del bloque, lo que permite almacenar la información que será útil para realizar futuras predicciones.

- **Puerta de Salida** (*Output Gate*): Decide que parte de la información del estado de la celda será enviada como salida y al siguiente estado oculto.

Se puede observar el diagrama de un bloque de memoria LSTM con cada una de las puertas en la Fig. 3.4.

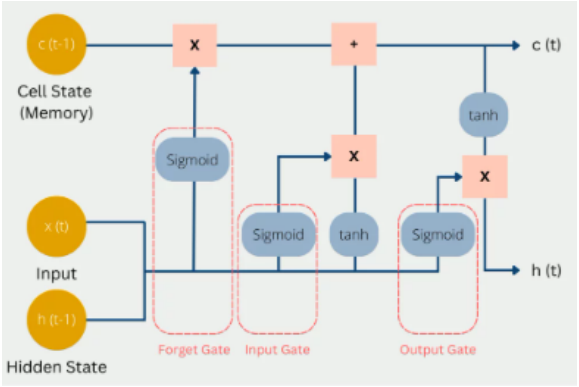

Figura 3.4: Diagrama de una red LSTM [22]

3.2.2. Implementación de las RNN

Para poder crear las RNN, hacen falta entender distintos aspectos de su implementación. En concreto, en esta sección se comentará el funcionamiento de las diferentes capas de neuronas utilizadas y el optimizador empleado para su entrenamiento.

Capas utilizadas

Como se explicó en la sección 3.1, las redes neuronales están formadas por distintas capas, por las que fluye la información desde la capa de entrada hasta la salida. En función de la configuración de esta capa o de como se reciba la información de la capa anterior encontramos diferentes tipos. Concretamente en este trabajo se han empleado las siguientes capas.

- **Capas densas**: Todas las neuronas de esta capa se conecta con todas las neuronas de la capa anterior. Requiere más recursos computacionales, pero captura muy bien las relaciones globales en los datos.
- **Capas *dropout***: Se trata de una técnica que se aplica durante el entrenamiento de la red neuronal, donde se "apagan" o se ponen a ceros aleatoriamente un porcentaje de neuronas que se especifica antes, normalmente entorno al 20 %. Podría entenderse como si durante el entrenamiento de la red, se estuvieran entrenando redes más pequeñas,

por lo que se estaría logrando reducir el sobreajuste (*overfitting*), ya que la potencia de estas redes es menor.

Optimizador *Adaptive Moment Estimation* (Adam)

Es un algoritmo ampliamente usado hoy días para el entrenamiento de redes neuronales profundas. Fue propuesto por Diederik P. Kingma y Jimmy Ba en 2014 [23]. Las ventajas de esta técnica para la optimización de las redes son:

- Adaptibilidad: Ajustando la tasa de aprendizaje o *learning rate* automáticamente durante el entrenamiento, convergiendo rápidamente.
- Eficiencia computacional: No requiere mucha memoria por lo que funciona bien con grandes conjuntos de datos o parámetros.
- Recomendado para trabajar con series temporales: Donde la distribución de los datos puede cambiar con el tiempo.

3.2.3. Trabajar con RNN en series temporales

En este problema se trabajará con series temporales, algunas más largas que otras, pero como mínimo todas con al menos unos 300 pasos. Cuando se trabaja con series temporales en *machine learning* se deben transformar estas series temporales en series supervisadas, es decir, crear series de una longitud constante de menor tamaño con uno o varios valores a predecir. Estas series supervisadas son las que puede utilizar el modelo para ser entrenado. Se puede apreciar un ejemplo visual en la Fig. 3.5, extraída de la fuente [24]. En la figura, los datos de la izquierda corresponden con la serie temporal de los datos generados y los datos de la derecha las series supervisadas obtenidas, siendo la columna roja el paso a predecir.

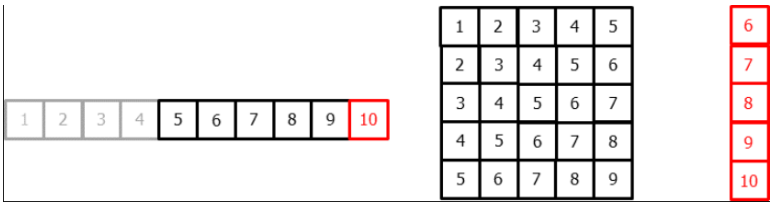

Figura 3.5: Transformación a series supervisadas [24]

Antes de continuar, aclarar que en este problema cada paso de tiempo de los datos generados es de 0,1s, es decir, que 10 datos en la serie temporal, corresponden a un segundo de simulación.

Lo siguiente que es necesario para trabajar con series temporales es determinar cómo se quieren crear estas series supervisadas. En el ejemplo anterior, se decidió utilizar 5 pasos anteriores o *lags* para predecir un paso a

futuro. Predecir únicamente un paso a futuro en este problema es evidente que se queda corto, por lo tanto se debe ser capaz de predecir varios pasos a futuro. Para poder hacerlo, existen diferentes estrategias posibles, algunas son las siguientes, extraídas de la fuente [24]:

Recursive multi-step forecasting

Esta estrategia consiste en utilizar predicciones del modelo para generar nuevas predicciones. Por ejemplo, si el modelo sólo produce una salida, el siguiente paso t_n, pero se quiere obtener t_{n+1}, se vuelve a utilizar el modelo añadiendo a los datos de entrada el valor predicho t_n. Tal como se muestra en la Fig. 3.6.

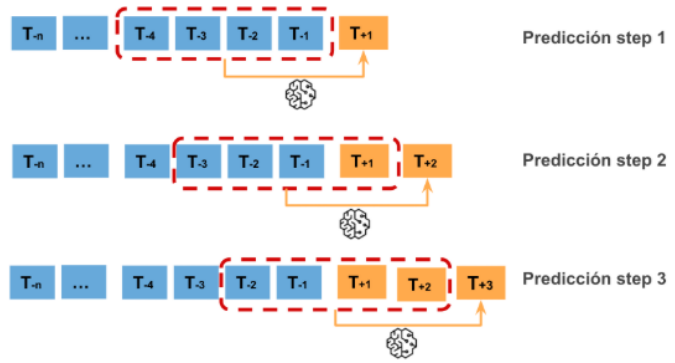

Figura 3.6: *Recursive multi-step forecasting* [24]

La principal ventaja de usar esta estrategia en su flexibilidad, es decir, con un sólo modelo entrenado, se puede ser capaz de predecir tantos valores como se necesiten, con menos precisión a medida que avancemos en el futuro dependiendo de la fiabilidad y rendimiento del modelo. Por lo tanto, al enfrentarse a problemas sencillos donde el modelo puede ajustarse con facilidad a los datos, esta técnica sería la ideal.

Como desventajas se encuentran que, en problemas más complejos, donde predecir más pasos en el futuro sea más difícil, posiblemente los resultados sean peores que otras posibles estrategias.

Direct multi-step forecasting

Como el nombre indica, consiste en entrenar un modelo independiente por cada paso en el futuro en los que se tenga interés. Por lo tanto si se quiere predecir t_n y t_{n+1}, se tendrán dos modelos que se especialicen en predecir cada paso en la serie temporal. Se puede ver en la Fig. 3.7.

Las ventaja principal es que se tienen diferentes modelos que están especializados en predecir cada uno de los pasos en el futuro, por lo que los

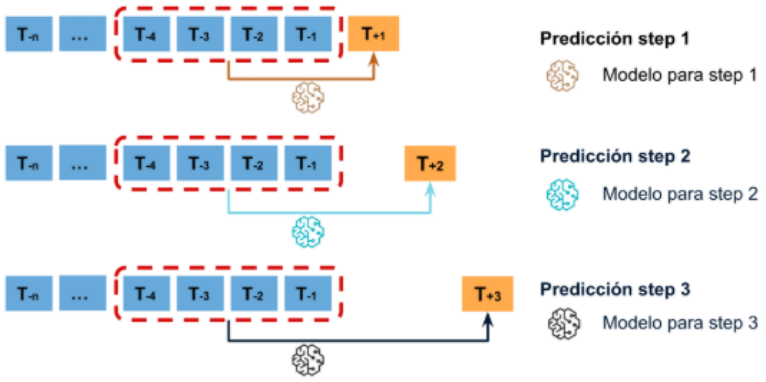

Figura 3.7: *Direct multi-step forecasting* [24]

resultados lograran ser mejores en cada uno de estos pasos en comparación de otras técnicas. Sin embargo, las desventajas son:

- Se tiene que entrenar un modelo por cada paso a futuro que se quiera predecir, es decir, mayor coste computacional.
- Menor flexibilidad, ya que si posteriormente se necesita predecir un paso que actualmente no se predice, se tiene que volver a entrenar otro modelo distinto.
- La complejidad de preparación de los datos, para construir las series temporales supervisadas, diferentes para cada modelo a entrenar.

Forecasting multi-output

Por último, esta estrategia consiste en predecir con el mismo modelo de forma simultánea varios valores a futuro de la secuencia. Es decir, que utilizando un único modelo en una sola predicción obtenemos t_n y t_{n+1}. Como podemos ver en la Fig. 3.8.

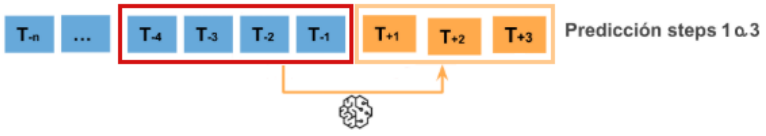

Figura 3.8: *Forecasting multi-output*

La principal ventaja sería la simplicidad ya que únicamente se tiene que entrenar un modelo para predecir los valores a futuro que se necesiten, además especializándose en todos los pasos por igual, de manera que, aunque empeore los resultados de emplear un modelo para cada paso, mejora la

técnica recursiva. Por otro lado, la desventaja sería menor flexibilidad, ya que en el caso de querer ampliar los pasos predichos, o predecir otros que actualmente no se predicen, se debería de volver a entrenar un modelo nuevo.

3.3. Consideraciones de técnicas utilizadas

En esta sección se explicarán algunos conceptos que se usarán a lo largo del trabajo para poder implementar los algoritmos o poder evaluar la eficiencia del modelo.

3.3.1. Distancia y tiempo de detención de un vehículo

Para calcular la probabilidad de colisión de un vehículo frente a cualquier objeto que se detecte en la vía se han empleado estos indicadores. De forma resumida, la distancia de detención de un vehículo es la suma de la distancia de reacción y la de frenado. La primera es la distancia recorrida en el tiempo de reacción de una persona, de media, durante $0,25s$. La segunda es la distancia desde que se empieza a frenar hasta que el vehículo está completamente detenido.

Comúnmente la velocidad de frenado es calculada de la siguiente forma:

$$DistanciaFrenado(m) = (\frac{velocidad(km/h)}{10})^2$$

Se puede ver en la Fig. 3.9 la relación entre la velocidad del vehículo frente al tiempo y la distancia de detención de un vehículo.

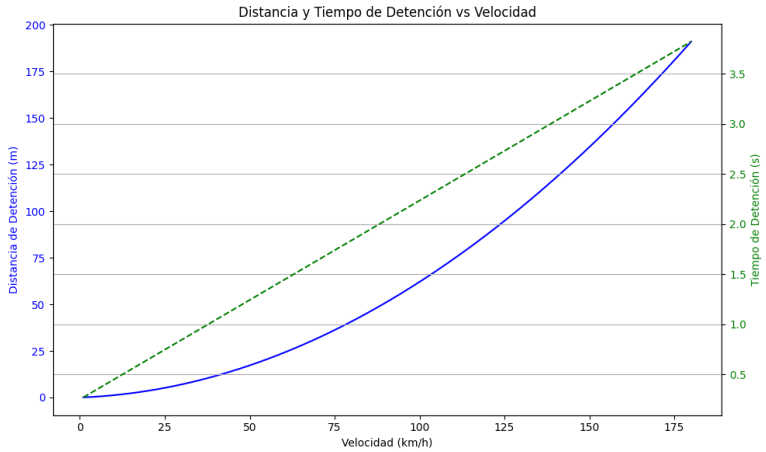

Figura 3.9: Tiempo y distancia de detención del vehículo en función de la velocidad

3.3.2. Curva precisión-*recall*

Se trata de una técnica que se puede utilizar cuando se trabaja con un clasificador binario y se quiere medir su rendimiento. Esta curva muestra la relación entre la precisión y el *recall* del modelo. Para entenderlo mejor se necesita partir de unos conceptos básicos que miden las predicciones del modelo frente a los datos reales.

- ***True Positives* (TP)**: datos positivos clasificados correctamente.
- ***True Negatives* (TN)**: datos negativos clasificados correctamente.
- ***False Positives* (FP)**: datos negativos clasificados como positivos.
- ***False Negatives* (FN)**: datos positivos clasificados como negativos.

Estos cuatro indicadores conforman la matriz de confusión. Al mismo tiempo, a partir de estos indicadores se pueden calcular las dos métricas que miden el rendimiento del modelo.

- **Precisión**: Porcentaje de datos positivos clasificados correctamente por el modelo, o dicho de otro modo, de todos los positivos que ha detectado el modelo, cuántos son realmente positivos. Se calcularía de la siguiente manera:

$$Precision = \frac{TP}{TP + FP}$$

- ***Recall* o Sensibilidad**: Porcentaje de positivos detectados por el modelo, o dicho de otro modo, entre todos los positivos reales, cuántos detecta el modelo. Se calcula de la siguiente manera:

$$Recall = \frac{TP}{TP + FN}$$

Cuando se trabaja con un modelo que debe clasificar los datos entre dos clases, en este caso, positivos o negativos, este produce una salida continua entre 0 y 1. Por lo tanto, se debe seleccionar un umbral o *threshold* sobre el cuál clasificar la salida del modelo como positiva o negativa. La elección de este umbral determina el comportamiento del modelo, es decir, de las métricas precisión y *recall*. A medida que por ejemplo se selecciona un umbral más bajo (más cercano a 0), será más fácil clasificar una salida como verdadera, se estaría aumentando el *recall* al estar disminuyendo los falsos negativos. Sin embargo, empeoraría la precisión del modelo, aumentando los falsos positivos. Es por esto que el comportamiento de estas dos métricas es inversa, siendo la decisión del umbral clave.

La curva precisión-*recall* muestra el comportamiento o relación de estas dos métricas a medida que se modifica el umbral. Se considera por lo tanto el valor óptimo del umbral en aquél punto de la curva donde esta se aproxime más a la esquina superior derecha de la gráfica. Por ejemplo, en la Fig. 3.10 se puede ver un ejemplo de curva precisión-*recall* calculada en este trabajo para uno de los modelos.

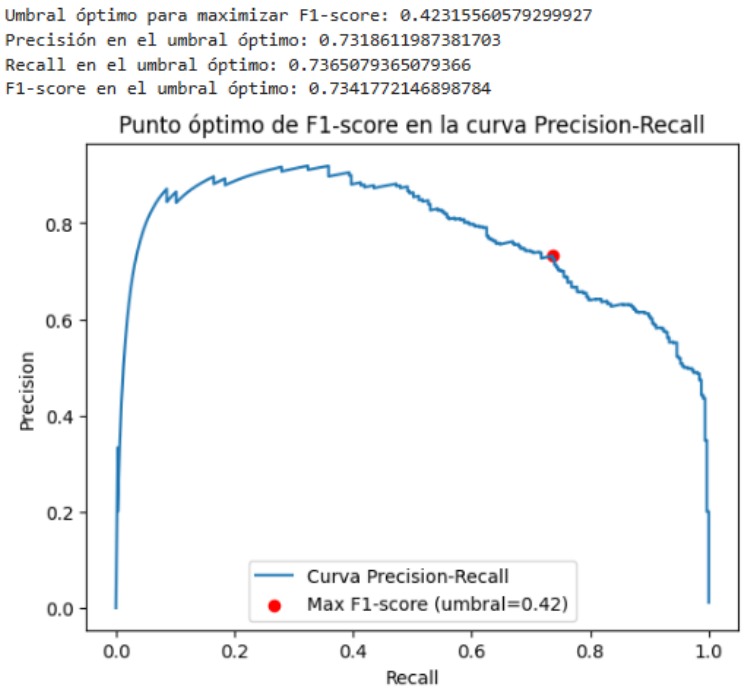

Figura 3.10: Curva precision-recall para el primer modelo de la fase 3

En esta gráfica, se aprecia otra métrica más, el *F1-score* que mide la relación entre la precisión y el *recall*. De esta forma, cuando se busca el punto óptimo de la curva, lo que se está calculando es el umbral para maximizar el *F1-score*. Este se calcula de la siguiente manera:

$$F1_{score} = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Capítulo 4

Herramientas

En este capítulo se detallarán las herramientas más importantes con las que se ha realizado el proyecto. Principalmente, el simulador escogido para realizar los escenarios y una breve explicación del entorno para desarrollar el modelo neuronal.

4.1. CARLA

CARLA es uno de los simuladores más populares hoy día de código abierto, permitiendo recrear escenarios simulados con una alta fidelidad y con gráficos de calidad, que favorece un comportamiento más realista de sensores como la cámara, Lidar, Radar, etcéteta.

A continuación, se explicarán algunas de las características más importantes de CARLA [17], que hay que tener en cuenta para entender cómo se ha diseñado el código:

- **Arquitectura Cliente-Servidor**: El simulador se ejecuta en un servidor que va recibiendo las instrucciones de los clientes, por defecto a través del puerto 2000. Este cliente utiliza una API de Python que se puede utilizar del siguiente modo:

  - **Módulo *client***: Donde viene toda la información que se necesita del simulador. A través de este modulo se puede modificar el mapa, grabar la simulación, gestionar el tráfico y además, contiene el objeto *world*.
  - **Objeto *world***: Contiene los métodos para generar los actores, cambiar el clima, obtener el estado actual del mundo o configurar la sincronización entre otros. Este objeto va asociado al mapa que se esté utilizando, por lo que si se cambia, también cambiará el objeto.

- **Sincronización**: El servidor va ejecutando cada actualización del mundo, calculando las posiciones de cada objeto, sus propiedades físicas o la información generada por cada sensor, con el objetivo de que el cliente pueda ir recibiendo esta información. Esta actualización del mundo puede realizarse de dos formas distintas, una en la que el servidor decide cuando se actualiza y otra si el cliente es quién lo decide.

    - **Modo asíncrono**: Es el servidor quien decide cuando se actualiza el mundo, tan rápido como pueda, sin esperarse a que el cliente realice sus operaciones. Esto último lleva a un problema de sincronización, ya que si el cliente es más lento que el servidor, puede perder información.
    - **Modo síncrono**: Es el cliente quién informa al servidor cuando puede actualizar el servidor, de esta forma se asegura de que no se pierda ningún dato.

De forma predeterminada, lo más sencillo para utilizar CARLA es generar un mundo, establecer unos parámetros generales y dejar ejecutándose el servidor, con un cliente que vaya guardando la información durante este periodo de tiempo. Quizás repetir este proceso con varias configuraciones distintas, para tener una mayor variedad de datos, no obstante es más complicado recrear escenarios más específicos, como por ejemplo cruces o adelantamientos. Se podría hacer filtrando todos los datos generados, pero sería menos práctico e ineficiente. Para ello, existen otras librerías que funcionan con CARLA, y que dan esa funcionalidad con una implementación más sencilla, por ejemplo está "*ScenarioRunner*" [25], que se recomienda en la página principal de CARLA, pero tras estar realizando algunas pruebas no he tenido muy buena experiencia, sobre todo por no estar muy actualizada. Por otro lado, dentro de la documentación, se menciona la herramienta de *Scenic* [26], que más adelante se explicará mejor, pero que en resumen me ha resultado más útil.

4.2. Google Colab

Es un entorno interactivo [27] que permite programar y ejecutar Python en el navegador, facilitando tareas como la configuración de dependencias y agilizando mucho el desarrollo, en este caso, de modelos neuronales. Se programa sobre cuadernos de "Jupyter", es decir, que se tienen celdas de código o texto, que podemos ir ejecutando conforme se necesiten. Se trata de una herramienta muy utilizada en el campo del aprendizaje automático, permitiendo usar librerías como "TensorFlow" o "Keras", a parte de las típicas como "Numpy", "Matplotlib", "Pandas" etcétera.

4.3. Librerías importantes

A continuación se describirán brevemente algunas de las librerías más importantes que se han utilizado para realizar este trabajo.

- **Scenic** [26]: Es una librería y lenguaje de programación probabilístico específico de dominio para modelar entornos de sistemas cibernéticos y vehículos autónomos. La principal ventaja de este lenguaje es integrar incertidumbre y restricciones que permiten generar múltiples escenarios a partir de una única definición. Por lo que cada vez que se crea un escenario a partir de esta definición, el comportamiento de cada agente y el resultado cambia, pudiendo generar datos con una gran variedad, de una forma muy sencilla. Una definición de escenario podría ser por ejemplo, dos vehículos, a una distancia de al menos 10 metros de un cruce cada uno, entrando cada uno por una entrada distinta al cruce y que sigan el carril por donde se encuentran a una velocidad como mínimo de 10 km/h.

  Esta herramienta está implementada en algunos de los simuladores más importantes, entre los que se encuentra CARLA, con funciones específicas que modifican el comportamiento de los agentes, como por ejemplo, seguir el vehículo de delante o seguir en el carril actual, que ayudan a crear cada escena. De modo que por su capacidad de generación de escenarios variados lo he escogido para crear los datos del simulador.

- **TensorFlow** [28]: Es una librería de código abierto utilizada en aprendizaje automático, desarrollada por Google. Esta diseñada para la creación de redes neuronales profundas, así como su entrenamiento. Su implementación basado en grafos de flujo de datos sobre tensores permite optimizar y distribuir la carga de computación sobre CPUs, GPUs o TPUs.

- **Keras** [29]: Es una API en Python que facilita la creación y entrenamiento de redes neuronales. Proporciona una interfaz modular y sencilla basada en otras herramientas como puede ser TensorFlow, que permite que el desarrollo de los modelos sea rápido e intuitivo, sin tener que preocuparse por detalles de bajo nivel.

Capítulo 5

Planificación

En este capítulo se explicará la planificación temporal inicial del proyecto, los objetivos logrados y los recursos y costes que se utilizarán para realizarlo.

5.1. Planificación temporal

En esta sección se detalla la organización temporal que se sigue en este proyecto, dividiéndolo en las distintas etapas que lo conforman por orden temporal y una estimación de su duración. El proyecto comienza en julio de 2024 y se extenderá hasta Junio de 2025.

1. **Tareas iniciales**

    a) **Discusión de la temática**: Reunión con los tutores para discutir el tema, la organización que se seguirá y el entorno donde se realizará el trabajo.

    b) **Presentación de la propuesta elegida**: Presentación de la temática escogida y aprobación por parte de la universidad.

2. **Estudio y Análisis**

    a) **Elección del simulador**: Investigar los simuladores más usados en este contexto y escoger aquél que se ajuste mejor con las necesidades y recursos disponibles.

    b) **Estudio del funcionamiento del simulador**: Comprender el funcionamiento del simulador para poder implementar en él las pruebas para poder generar los datos.

    c) **Estudio de las redes neuronales**: Análisis de las redes neuronales más utilizas y apropiadas para el problema que se está abordando.

d) **Estudio de desarrollo de modelos**: Análisis de documentación con desarrollos de modelos similares para comprender que estructura debe tener la red neuronal, cómo poder entrenarlo y el tipo de características necesarias o más útiles.

3. **Desarrollo del sistema para detectar colisiones**

   a) **Diseño e implementación de la fase inicial**: Primera propuesta de solución, más simple, pensada principalmente para superar los primeros objetivos y poder realizar algunos experimentos con datos y los modelos. Y a partir de este punto, mejorar el diseño, para mejorar el rendimiento del modelo.

   b) **Diseño e implementación de las siguientes fases**: diseño e implementación de las siguientes fases que tendrán como base la fase anterior, buscando obtener mejores resultados en el modelo entrenado.

4. **Desarrollo de sistema anti-colisión**

   a) **Elaboración de un diseño**: Creación de una propuesta válida y que cumpla con los requisitos del sistema anti-colisión.

   b) **Implementación en el simulador**: Implementación del sistema diseñado en el simulador.

5. **Elaboración de la memoria**

   a) **Elaboración de la memoria**: Redactar la memoria final que describa todo el trabajo realizado en el proyecto.

   b) **Entrega de la memoria**: Entrega de la memoria a la universidad en los plazos marcados.

   c) **Defensa del proyecto**: Defensa ante el tribunal de presente trabajo.

La planificación inicial se puede observar en la Fig. 5.1

5.1.1. Objetivos logrados y planificación temporal final

Tras haber finalizado el proyecto, se logra cumplir los objetivos propuestos 1.3, con la planificación propuesta. A continuación se expondrá las tres fases o diseños que finalmente se han realizado.

- **Fase 1**: Cómo se propuso, se centra en cumplir los primeros objetivos, de forma que tiene una duración superior a las demás fases.

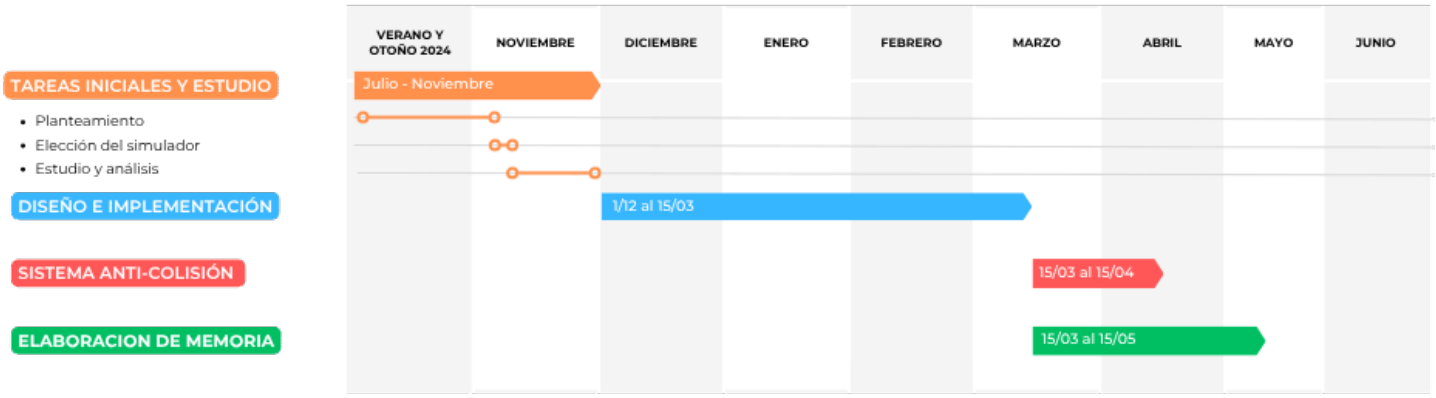

Figura 5.1: Diagrama de Gantt de la planificación inicial del proyecto

- **Fase 2**: Los principales cambios respecto a la fase anterior se realizan sobre la primera etapa de diseño e implementación, cambiando el sensor utilizado, así como la información generada.
- **Fase 3**: En esta fase se logra cumplir el cuarto objetivo, de tener un modelo con resultados fiables, por lo que a partir de estos se implementa el sistema anti-colisión.

A continuación se muestra el diagrama de Gantt que detalla la planificación temporal seguida al final del proyecto, en la Fig. 5.2.

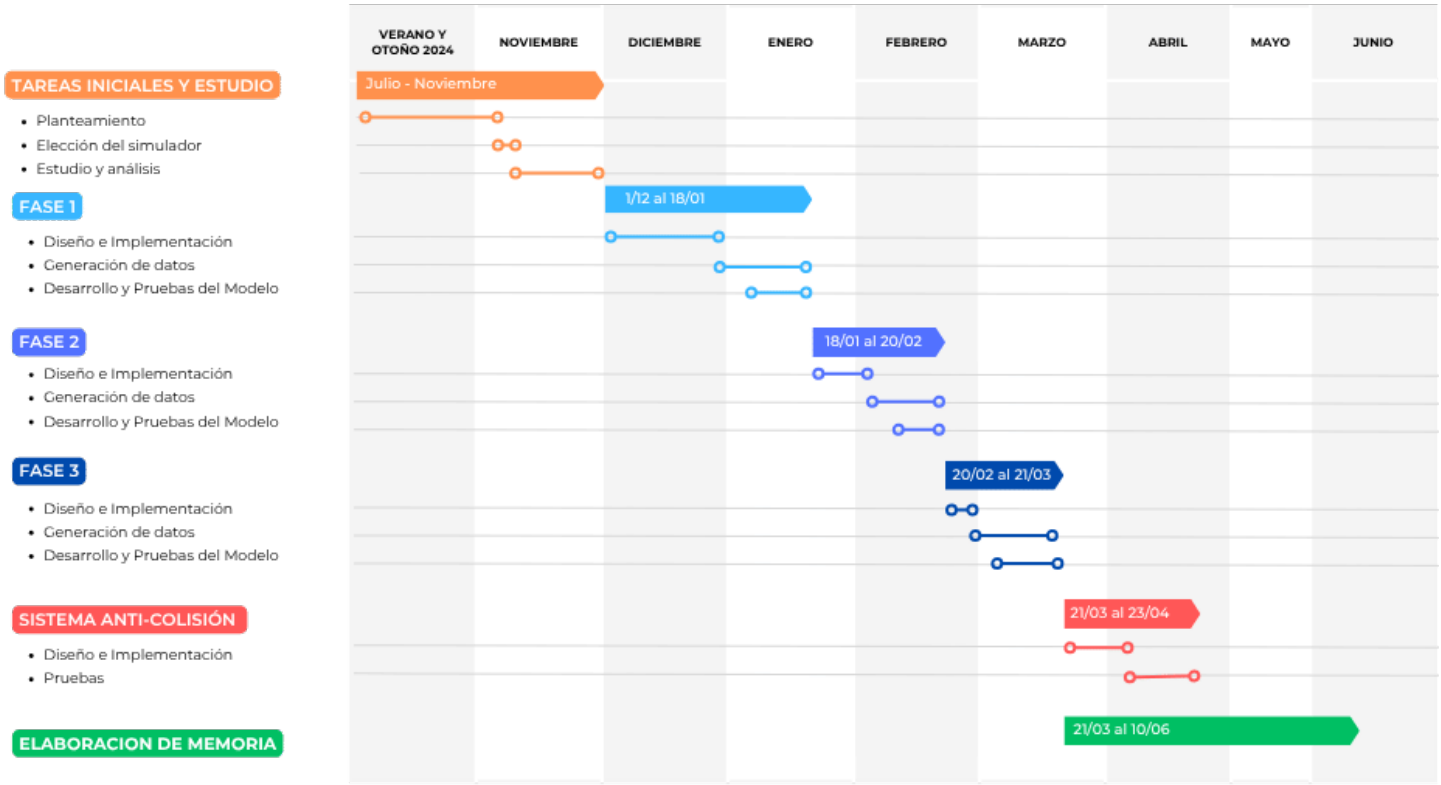

Figura 5.2: Diagrama de Gantt de la planificación final del proyecto

5.2. Recursos y costes

En esta sección se mostrarán los recursos que se utilizarán en el proyecto así como los costes derivados de estos.

5.2.1. Recursos *Software*

Los recursos *software* son los siguientes:

- **Sistema Operativo Windows 10**: El sistema operativo instalado en el ordenador que se ha utilizado para desarrollar este trabajo.
- **CARLA**: El simulador, explicado en la sección 4.1.
- **Visual Studio Code**: La herramienta donde se programa la implementación del código sobre el simulador.
- **Google Colab**: Como ya se ha explicado antes, la herramienta utilizada para desarrollar los modelos neuronales. Una de las grandes ventajas es que puedo trabajar sobre el mismo notebook desde distintos dispositivos.

- **Overleaf**: La plataforma donde se realiza la memoria, muy práctica para desarrollar documentos LaTeX entre varias personas.
- **Canva**: La plataforma donde se han realizado la mayoría de las ilustraciones que se han utilizado en este trabajo.

Todos las herramientas o programas no han tenido ningún coste por su uso, ya que la mayoría son *Open Source* o ya se tenían antes de iniciar el proyecto y estaban completamente amortizadas, como es el caso de Windows 10.

5.2.2. Recursos *Hardware*

Para la realización de este proyecto se ha utilizado únicamente un ordenador, con potencia suficiente para poder ejecutar el simulador. Sus características las podemos ver en la tabla 5.1

Componente	**Descripción**
SO	Windows 10
CPU	Intel i7 7700
GPU	GTX 4060
RAM	16 GB 2133 MHz
MEMORIA	2TB SSD 3500 MB/s

Cuadro 5.1: Características ordenador

Durante la realización de este trabajo, se invirtieron recursos en este ordenador. Los costes estimados para los recursos *hardware* se muestran en la tabla 5.2. Se estima un tiempo de amortización total de 5 años para cada componente, y un tiempo de utilización de medio año.

Componente	**Descripción**	**Coste (€)**
GPU	GTX 4060	35,50
RAM	16 GB 2133 MHz	22,10
MEMORIA	2TB SSD 3500 MB/s	24
Coste total		81,60

Cuadro 5.2: Costes de recursos hardware

5.2.3. Recursos Humanos

Las personas que han participado en el desarrollo de este proyecto han sido:

- **Jose Antonio Marqués Ponce**: alumno del doble grado de Ingeniería Informática y *Administración y Dirección de Empresas* (ADE).

- **Jorge Navarro Ortiz**: Profesor Titular de Universidad del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.
- **Félix Delgado Ferro**: Investigador Predoctoral en el Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

El coste de los recursos humanos se muestra en la siguiente tabla 5.3

Persona	**Coste / hora (€)**	**Horas total**	**Coste total (€)**
Jorge Navarro Ortiz	50	20	1000
Félix Delgado Ferro	35	30	1050
Jose A. Marqués Ponce	25	350	8750
Coste total			**10800**

Cuadro 5.3: Costes de recursos humanos

5.2.4. Costes totales

Tras ver todos los recursos utilizados en este proyecto, se mostrarán los costes totales de este en la siguiente tabla 5.4.

Recursos	**Coste (Euros €)**
Recursos *Software*	0 €
Recursos *Hardware*	81,60 €
Recursos Humanos	10800 €
Coste total	**10881,60 €**

Cuadro 5.4: Costes totales proyecto

El coste total para este proyecto es de DIEZ MIL OCHOCIENTOS OCHENTA Y UN EURO Y SESENTA CÉNTIMOS.

Capítulo 6

Diseño e Implementación

En este capítulo, se explicará primeramente cuál ha sido el proceso para llegar a la solución final, es decir, cada una de las tres fases que se han hecho en total. Después, se explicará como se ha implementado el código para poder trabajar con el simulador y recoger los datos, al mismo tiempo que predecir colisiones futuras. Posteriormente, el proceso para desarrollar las redes neuronales y, por último, el diseño e implementación del sistema anti-colisión.

6.1. Diseño de las fases

Como se explicó en la metodología, en el proyecto se ha decidido realizar en distintas fases, con diseños distintos. Cada fase usa diferentes sensores para calcular la información, del mismo modo que entrena los modelos con diferentes datos. A continuación, veremos más detalladamente el diseño de cada fase, enfocándonos en cómo funcionan, cuál es su vector de características con el que funciona la red neuronal y las razones por las que se decidió abandonar la fase actual y pasar a una nueva. El proceso en el que se explicarán cada una de las fases se muestra en la Fig. 6.1:

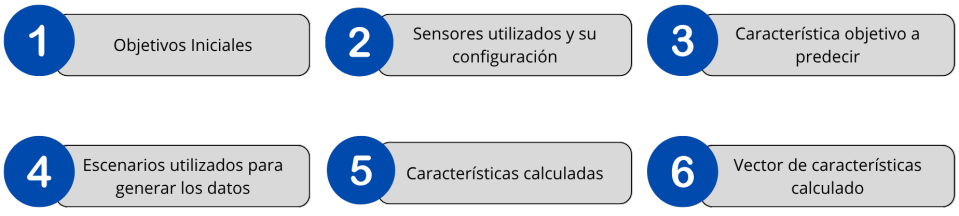

Figura 6.1: Planificación para explicar cada fase

6.1.1. Fase 1

Objetivos iniciales

Esta fase inicial pretendía superar los primeros objetivos planteados, es decir, diseñar una propuesta inicial, poder implementarla en el simulador, logrando entender su funcionamiento y correcto uso, para lograr obtener los primeros datos. De forma que el diseño es más simple, con unos resultados que conformarían el punto de partida, que se buscarían superar en las siguientes fases.

Explicación de sensores utilizados y su configuración

En esta fase se utiliza únicamente un sensor *RAdio Detection And Ranging* (Radar) colocado en la parte frontal del vehículo, de forma que sólo se recoge la información de los objetos que se sitúen por delante. En concreto, la configuración que se empleó en el sensor fue la siguiente:

- **Campo horizontal de 90º**: No se hizo más grande por que a priori la información más importante se presuponía que estuviera situada delante del vehículo. Por ejemplo con un campo más grande de 150º habría muchos más puntos con menos relevancia a los laterales del vehículo.
- **Campo vertical de 0º**: Sólo se obtiene información a una única altura, a la que se encuentra el sensor. Se hizo así puesto que queremos saber que objetos tenemos delante, saber su altura es en un principio irrelevante.
- **Distancia de 40 metros**: Al ser un primer diseño, añadido al hecho de que las colisiones se realizan a distancias cortas, se decidió tomar una distancia de 40 metros de profundidad.
- **3000 puntos por segundo**: Teniendo en cuenta que la simulación va a 20 fps, se recogen 150 puntos en cada frame, lo cuál es más bastante para detectar en cada frame con precisión los objetos que se encuentran delante.

Se puede ver en la Fig. 6.2 una representación de la configuración del sensor realizada para esta fase.

Característica objetivo de predicción

El objetivo principal es predecir colisiones, es decir, tras una secuencia de escenas detectadas por el vehículo, poder determinar si acabaría terminando en una colisión o no. Al principio, para intentar simplificar el problema, se

Figura 6.2: Configuración fase 1

optó por crear una función de probabilidad de colisión, un algoritmo que calculase esta probabilidad en cada instante de la simulación.

Es un paso muy importante, ya que el rendimiento de la red neuronal está directamente relacionada con cómo de bien funciona el algoritmo, es decir, su fiabilidad captando el peligro de colisión en cada instante. Realmente, se trata de una tarea complicada, ya que existen muchas casuísticas distintas que pueden provocar un accidente y muchas de ellas son muy difíciles de predecir.

En esta fase inicial, el diseño de este algoritmo para calcular la probabilidad de colisión es simple. Este no es preciso debido principalmente a las limitaciones del sensor Radar para captar información relevante como pueden ser las líneas de la carretera.

De esta forma, la variable o característica a predecir en esta fase es esta función de probabilidad, cuyo funcionamiento se explicará en las siguientes secciones.

Escenarios utilizados para generar los datos

En esta fase, sólo se ha hecho uso del simulador CARLA para generar los datos, sin emplear ninguna otra librería. De esta forma los datos son recogidos en el modo libre. Se escoge un mapa, se colocan los actores que se requieran, es decir, el vehículo principal EGO, los sensores que se necesiten, el resto de vehículo, peatones etcétera. Y por último, empezamos a generar las iteraciones en el simulador para generar los datos e ir guardándolos. En todas las fases se han obtenido los datos del mapa "Town10". Este mapa imita una ciudad, donde hay edificios, semáforos y una variedad de carreteras diferentes, como se puede ver en la Fig. 6.3.

Para intentar generar unos datos lo más variado posible, se ha cambiado el comportamiento de ciertos agentes o vehículos para forzar situaciones más peligrosas. Estas modificaciones afectan al 20 % de los vehículos del

Figura 6.3: Mapa utilizado para generar los datos de CARLA

simulador, excluyendo el EGO. Son las siguientes: (1) van un 30 % más rápido del límite marcado de la vía, (2) el 70 % de las veces se saltan la señalización y (3) sólo mantienen un metro de seguridad con el vehículo de delante.

Características calculadas a partir de la información de los sensores

En cada iteración de la simulación, se recoge una nube de puntos de los objetos detectados. Cada uno de los puntos posee cuatro indicadores o datos, **altitud** ($radianes$)**, latitud** ($radianes$)**, profundidad o distancia** (m) **y velocidad hacia el sensor** (m/s).

El objetivo previo a calcular la probabilidad de colisión es detectar los objetos que tenemos delante. A priori, sólo se tiene una nube de puntos, por lo que tenemos que hacer algunos cálculos para obtener de aquí los objetos presentes. El algoritmo para realizar esta tarea es el siguiente:

1. **Agrupar los puntos detectados por velocidad hacia el sensor**: ¿Que diferencia un objeto de otro? En los datos que nos proporciona el radar, únicamente la velocidad hacia el sensor nos puede ayudar a esto. Entre los objetos que tenemos alrededor, encontramos objetos inmóviles (árboles, señales, edificios etcétera) y objetos móviles (vehículos, ciclistas y/o peatones). De forma que si estos últimos se están moviendo, la velocidad hacia el sensor será distinta del resto de objetos quietos, debido al efecto Doppler, por lo que podremos diferenciarlos.

Además, no todos los objetos móviles se mueven a la misma velocidad, esto nos ayuda a poder diferenciar cada uno de ellos. De esta forma, se obtendrá un grupo para todos lo objetos estáticos y otros por cada objeto móvil detectado.

2. **Seleccionar en cada uno de los grupos detectados, aquél punto más cercano**: Este punto representará al grupo, y proporcionará la información de este objeto, su ubicación, distancia y velocidad hacia el sensor Radar (vehículo EGO). Esto se hace para simplificar el problema, en vez de tener cien puntos por cada objeto, se tiene sólo uno. Aunque se pierda información, se gana rendimiento.

Se puede apreciar en la Fig. 6.4 una adaptación realizada para poder visualizar el funcionamiento del algoritmo descrito anteriormente.

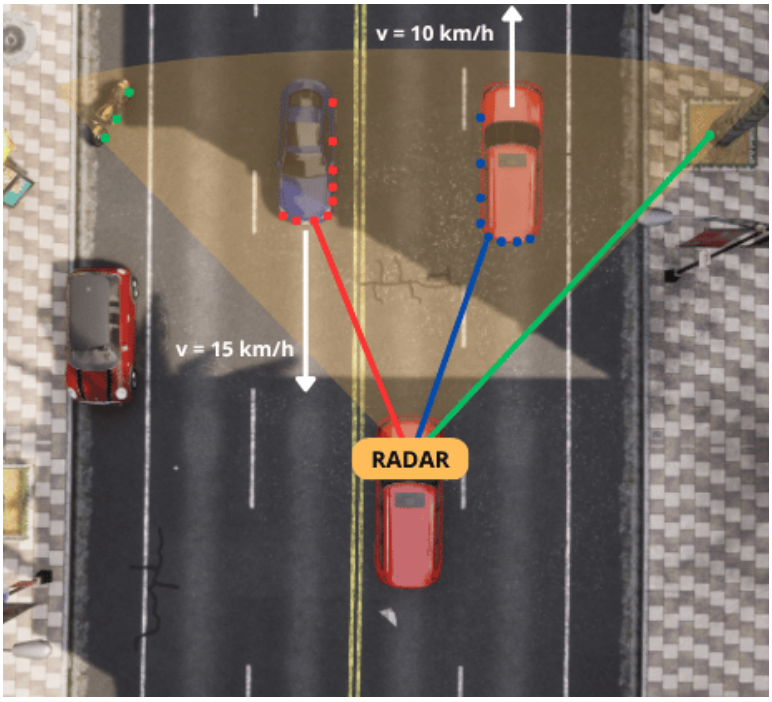

Figura 6.4: Adaptación del funcionamiento de la fase 1

De esta forma, ya se tienen los objetos detectados. El siguiente paso es calcular la probabilidad de colisión. Este cálculo se realiza de manera independiente a cada uno de los objetos detectados. Se aplica otro algoritmo que consta de varios pasos:

1. Si la distancia es 0 o la velocidad hacia el sensor es negativa, es decir, se está alejando, entonces la probabilidad es 0 %.

2. Una vez calculada la distancia de detención, cómo se explica en la sección 3.3.1 del capítulo de fundamentos, si esta es mayor que la distancia al objeto, entonces la probabilidad será 100 %.

3. En el caso de que no se cumplan ninguno de los pasos anteriores, calculamos la probabilidad de colisión cómo de la velocidad a la que se está aproximando, entre la velocidad máxima a la que se podría detener, en función de la distancia a la que se encuentra el objeto.

4. Se penaliza la probabilidad obtenida en función de cómo de alejado este de la trayectoria del vehículo. Se presupone que la trayectoria es latitud 0^{o} más el giro del volante. En función de cuanto se aleje la latitud del objeto de la latitud de nuestra trayectoria se disminuirá la probabilidad de colisión.

Una vez que tenemos calculada la probabilidad de colisión con cada uno de los objetos que hemos detectado, el último paso será seleccionar la mayor entre todas ellas. Así, obtendremos la probabilidad de ese instante de la escena, la que deberá predecir el modelo.

Vector de características

Una vez que tenemos todos los cálculos realizados, debemos pasárselos a la red neuronal, para el entrenamiento. Debemos por lo tanto decidir y organizar cómo pasar esta información. Una posibilidad sería pasar toda la nube de puntos junto con la probabilidad calculada, pero sería una información mucho más compleja que tardaría más tiempo en aprender la red neuronal. Por lo que se opta por crear las características a partir de los datos calculados, de forma que esta información es condensada y relevante. Esto permite mejorar el rendimiento de la red neuronal con menos características y menos tiempo de entrenamiento.

El siguiente problema es decidir cuantos objetos detectados pasarle a la red. Debido a que este número es variable, a veces puede ser uno y otras por ejemplo siete, tenemos que determinar cómo hacerlo. Tras un estudio de casos similares y soluciones posibles, se optó por pasar un vector fijo con los cinco objetos detectados con mayor probabilidad de colisión. Debido a que a veces pueden ser menos de cinco, se rellenará con ceros el vector restante. Para ayudar al modelo se le indicará cuántos objetos se han detectado a través de otra característica.

De esta forma, el vector final de características de esta fase es el mostrado en la tabla 6.1. La serie actual es un valor que sirve para poder diferenciar cuando acaba y comienza un nuevo escenario, más adelante se explicará cuándo se usa.

Característica	**Breve descripción**	**Nº de datos**
Serie Actual	Puede valer 0 o 1	1
Nº objetos detectados	De cero a cinco	1
Vector objetos detectados	Ver la tabla 6.2	20
Velocidad del EGO	En m/s	1
Giro volante EGO	En grados	1
Máxima probabilidad	Entre todos los objetos	1
	Total de características	**25**

Cuadro 6.1: Vector de características calculadas fase 1

Característica	**Breve descripción**	**Nº de datos**
Velocidad	Hacia el sensor en m/s	1
Latitud	En grados	1
Distancia	Al sensor en m	1
Probabilidad de colisión	En porcentaje	1
	Total	**4**
	Total cinco objetos	**20**

Cuadro 6.2: Características por cada objeto fase 1

6.1.2. Fase 2

Objetivos iniciales

Tras lograr los primeros objetivos planteados en este trabajo durante la primera fase, ahora se busca mejorar los resultados obtenidos y lograr predecir mejor las posibles colisiones. Para ello, se parte con la premisa de mejorar el algoritmo que calcula la probabilidad de colisión, con la esperanza de que se ajuste mejor a la realidad. Esto último, junto con una base de datos mucho mayor, constituían la motivación para lograr que la red neuronal lograra un mejor desempeño.

Como conclusión principal obtenida en la fase anterior, tenemos que la información que se generaba era muy confusa. No compleja, sino poco consistente, es decir, los puntos obtenidos eran muy caóticos. Para que la red neuronal sea capaz de encontrar patrones o relaciones en los datos, estas deben de estar presentes, pero en nuestro caso no ocurría. Por lo que el objetivo al final era lograr obtener una información más clara y con más calidad.

Además, a partir de esta fase, se empieza a usar la herramienta de *Scenic* para poder generar escenarios específicos. De esta forma logramos tener una base de datos un poco más balanceada, con más casos donde se producen colisiones.

Explicación de sensores utilizados y su configuración

Se decide sustituir el sensor Radar por un sensor Lidar semántico, principalmente por que este último ofrece una mayor cantidad de información. Este sensor permite inspeccionar todo el entorno alrededor, además de diferenciar claramente cada uno de los objetos presentes, de forma que no tenemos que depender del algoritmo anterior para diferenciarlos en la nube de puntos. Esto ayuda a que la información sea menos confusa, ya que por ejemplo en la fase anterior, un vehículo que estaba detenido en frente nuestra (caso de un semáforo), pasaría de formar parte del grupo de objetos estáticos a un objeto móvil, de un *frame* o iteración a otra del simulador.

Para ello, se situó el sensor en la parte superior central del vehículo, realizando un barrido de todo el entorno en cada *frame*, es decir, un giro de 360^{o}. La configuración de este sensor es la siguiente:

- **Siete canales**: Cada canal es un láser del sensor a una altura distinta. Separados por una distancia igual a, el rango que haya entre intervalo máximo y mínimo entre el número de canales.
- **Rango de 80 metros**: Se decide ampliar el rango, principalmente para poder detectar antes los vehículos que se aproximen rápidamente desde una distancia grande, con lo que logramos que se anticipe mejor la red neuronal.
- **40000 puntos por segundo**: Aunque a priori pueda parecer mucho, se sitúa entre los valores promedios en estos casos.
- **Ángulo superior -1^{o}**: Al estar situado en la parte superior del coche, no interesa detectar objetos a una altura mayor, y por eso apunta hacia abajo.
- **Ángulo inferior -18^{o}**: A diferencia del sensor Radar que estaba situado a la altura media de un coche en la parte frontal, que con una única altura podíamos detectar un objeto que estuviera al lado y otro muy lejos, ahora, al situarse en sensor en la parte de arriba del coche, se necesitan varios láseres a diferentes alturas para poder detectar objetos cercanos y lejanos. Se escoge -18^{o} por que es la menor altura posible para poder detectar objetos lo bastante cerca y que al mismo tiempo el láser no impacte contra el propio vehículo.
- **Campo horizontal de 360^{o}**: Como ya explicamos antes, para poder reconocer todo el entorno.

Debido a las limitaciones de este sensor para poder detectar las colisiones, que al final es lo más importante, debido a que los láseres impactan con la carrocería del propio vehículo cuando queremos detectar objetos más

cercanos, se decide añadir un sensor de colisión. Este detecta cuando se produce un impacto con cualquier objeto por todas las zonas del vehículo. Este sensor no necesita de ninguna configuración adicional. Se puede apreciar una adaptación de la configuración de los sensores en la Fig. 6.5.

Figura 6.5: Configuración de la fase 2 y 3

Característica objetivo a predecir

Del mismo modo que en la fase anterior, se tiene como objetivo predecir una función de probabilidad de colisión, que como se ha comentado en el primer apartado se ha logrado mejorar bastante. Debido a que ahora se han tenido en cuenta más factores y más información para calcularla, como explicaremos en las siguientes secciones.

Escenarios utilizados para generar los datos

En esta fase, como se ha mencionado anteriormente, se ha empleado la librería de *Scenic*, por lo que se generan datos tanto del modo libre como de escenarios personalizados. El escenario recreado consiste en utilizar los cruces del mapa para provocar colisiones entre los vehículos. Como se comentó en la sección de contexto 1.1, esto casos representan la mayoría de situaciones de colisiones en la vías urbanas. En concreto la descripción del escenario es el siguiente.

- Filtrar por los cruces del mapa que tengan 3 carriles.
- Se impone al vehículo EGO una trayectoria a través de este cruce.
- Se impone al vehículo adversario una trayectoria que entre en conflicto con la trayectoria del vehículo EGO.
- Los vehículos realizarán las trayectorias indicadas a una velocidad media de $20km/h$ (a veces mayor, y otras, menor).

A partir de la descripción del escenario anterior, *Scenic* creará tantos escenarios como queramos. Para generarlos, podemos establecer una serie de requisitos, para controlar de una forma más precisa cómo se producen. De esta forma, los requisitos para el escenario recreado han sido:

- La distancia a la intersección del vehículo EGO tiene que estar entre 20 y 25 metros.
- La distancia a la intersección del vehículo adversario tiene que estar entre 15 y 20 metros.
- Terminar cuando el vehículo EGO se encuentre a más de 70 metros del punto de partida o el escenario lleve ejecutándose más de 5 segundos.

Características calculadas a partir de la información de los sensores

La información con la que partimos, consiste en una nube de puntos del sensor Lidar y un indicador del sensor de colisión de si se ha producido esta. Cada punto de la nube de puntos consta de su **posición en los tres ejes** (X, Y, Z) con el sensor como punto de origen, la **etiqueta semántica** del objeto (persona, vehículo...) y el **identificador** interno en el simulador del objeto. Tal como se muestra en la Fig. 6.6.

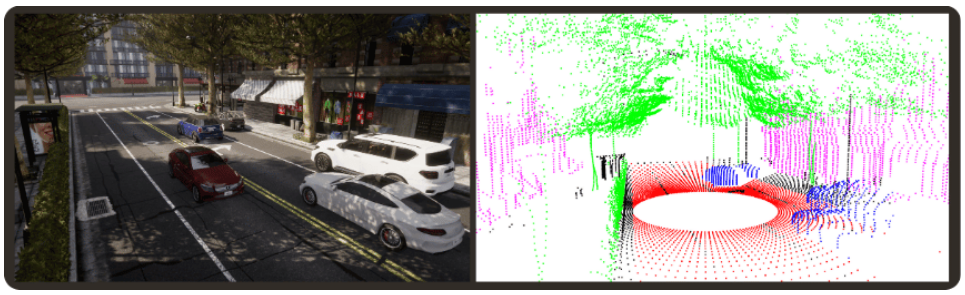

Figura 6.6: Datos generados por el sensor Lidar

Antes de comenzar con la explicación de los cálculos realizados, es importante mencionar que hay información como por ejemplo, los identificadores de los objetos o el hecho de que el Lidar sea semántico, que teóricamente en la práctica no serían posibles, pero por los recursos que se tienen y por tampoco hacerlo todo demasiado complejo, se ha decidido utilizar con el fin de poder obtener mejores resultados.

Un cambio fundamental respecto a la fase anterior, es la diferenciación entre los objetos móviles y los estáticos a la hora de crear el vector de características, calculando para cada uno, una información distinta.

De esta forma, como ya se hacía anteriormente, el primer paso es reconocer los objetos presentes en la nube de puntos, para ello se aplican filtros

con las etiquetas semánticas de los objetos que se buscan. Primero se obtienen los ***objetos móviles***, agrupando los puntos de cada uno, y de nuevo, se escoge el punto más cercano como representante del grupo. Para obtener la **distancia euclídea** hacia cada punto tenemos que realizar el primero de los cálculos, a partir de las coordenadas cartesianas.

$$d(P1, P2) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$$

En este punto se tiene un vector de puntos representando a cada objeto móvil detectado. El siguiente paso será obtener la probabilidad de colisión de cada uno respecto al vehículo EGO. Para ello, se calculará otro dato previamente, la **velocidad de aproximación**, que este sensor no nos proporciona de forma directa. Para ello, se simplifican los cálculos a través de la simulación, se obtiene de esta el vector 3d de la velocidad de cada uno de los otros vehículos. Comparándolo con el vector de velocidad 3d del vehículo EGO se puede obtener la velocidad a la que se están aproximando o alejando ambos.

Una vez que tenemos todo lo necesario, procedemos a calcular una primera estimación de la **probabilidad de colisión** de la misma forma que en la fase anterior, es decir, velocidad de aproximación entre la velocidad máxima a la que podríamos ir para frenar en la distancia que separan a ambos vehículos.

La mejora en el cálculo de esta probabilidad es la consideración de dos penalizaciones adicionales, que van en función del contexto vial.

1. **Las trayectorias**: Tener en cuenta la trayectoria de nuestro vehículo frente a la posición del otro, de forma que si no está en nuestra trayectoria se penalizará de forma considerable en función de qué tan grande sea esta diferencia.

2. **Los carriles**: Un hecho que hay que tener en cuenta cuando nos encontramos en una carretera, y es que si el otro vehículo circula por el carril contrario, a pesar de que se aproxime a gran velocidad y casi en nuestra trayectoria, la probabilidad real de que se produzca un accidente es mínima. Para poder detectar estos casos, se hicieron pruebas para detectar las líneas de los carriles a través de los puntos de sensor Lidar. A pesar de que se obtuvieron buenos resultados, hay veces que fallaba y generaba información errónea, añadiendo ruido que podrían afectar al rendimiento del modelo. Finalmente, se ha optado por identificar a través de la simulación cuando alguno de los objetos se encuentra en un cruce, de tal forma que si ninguno se encuentra en uno, la probabilidad de colisión se reduce bastante.

Por último, un pequeño ajuste final, si la distancia respecto al otro vehículo es muy pequeña, la probabilidad crece de manera inversamente

proporcional, para enfatizar más todavía en que a distancia cortas es más probable una colisión.

Con estas consideraciones se tienen calculadas todas las características de los objetos móviles, aunque se realiza un pequeño cambio en el identificador de cada vehículo. Este cambio consiste en convertirlo de un sistema global del simulador, donde cada vehículo siempre tiene el mismo, a un sistema local. De esta forma, a medida que un vehículo desaparece, libera su identificador, pudiéndose asignar a otro vehículo que posteriormente se localice. Este cambio se realiza para que la red neuronal no aprenda relaciones no deseadas, como que un cierto identificador sea más propenso a colisionar por que lo tenía asignado un vehículo más agresivo. El objetivo de incluir este identificador es ayudar a la red neuronal. Debido a que los vehículos están ordenados en el vector de objetos de mayor a menos probabilidad, normalmente cambian de posición, lo que puede resultar confuso para la red. En cambio, si se mantiene un identificador para cada vehículo, estos movimientos en el vector resultan menos confusos.

El siguiente paso es calcular las características para los objetos estáticos que, a diferencia de los móviles, son muchas menos y más simples. Lo primero es detectarlos, lo cuál se realiza de nuevo filtrando por etiquetas semántica, pero con un cambio sutil. Al contrario que pasaba con los objetos móviles, que se buscan en todo el alrededor, con los objetos estáticos no interesa saber que hay por detrás o por los laterales, ya que únicamente se podrá impactar con los que estén por delante, por lo que únicamente se tienen en cuenta estos. Además, para ayudar a la red neuronal a entender el entorno espacial y la importancia de cada objeto detectado, se diferenciarán tres zonas distintas, como se puede apreciar en la Fig. , seleccionando en cada una de estas zonas el punto estático más próximo. Finalmente, se calcula para este punto seleccionado la probabilidad de colisión a partir de una estimación del TTC, muy utilizado en trabajos similares y que se ajusta mejor para este caso, ya que al no moverse los objetos, son menos impredecibles.

Con todo esto ya se tienen calculadas todas las características necesarias, obteniendo la probabilidad de colisión global como la mayor obtenida entre todos los objetos, teniendo en cuenta además, que si el sensor de colisión se ha activado, será automáticamente 100 %.

Cuando se recibe que el sensor de colisión se activa, se actúa de manera diferente en función del escenario. Si se encuentra en un escenario generado con *Scenic*, a partir del momento en el que se ha producido la colisión, se deja de recoger datos, ya que lo que ocurra después no interesa conocerlo y posiblemente añada ruido, alterando el comportamiento de la red neuronal. Si es el escenario libre, se mantiene la colisión activa durante los 3 siguientes datos generados, no se detiene la simulación ya que casi nunca suelen producirse colisiones y en el caso de que se produzcan, no son muy fuertes y los vehículos son capaces de volver a circular con normalidad. En cambio en los escenarios de *Scenic*, los golpes suelen ser más fuertes y a veces pueden

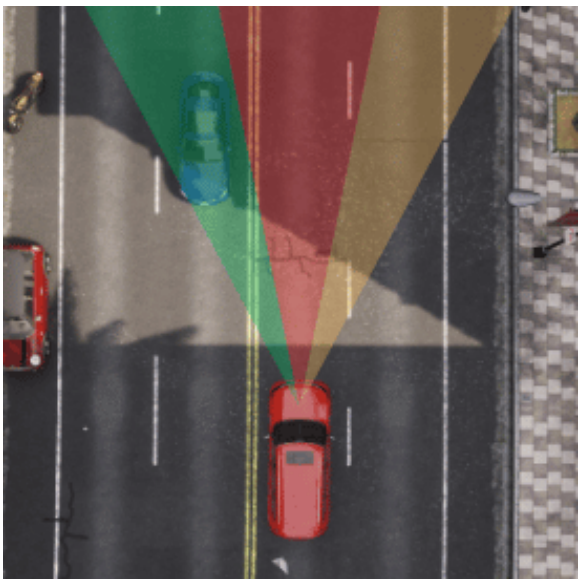

Figura 6.7: Detección de objetos estáticos Lidar

provocar movimientos bruscos.

Vector de características

Una vez que tenemos todas las características calculadas se pasa a construir el vector que utilizará la red neuronal. Del mismo modo que en la fase anterior, se usará un vector de cinco posiciones para los objetos móviles. Lo compondrán aquellos que estén más cerca, ordenados por proximidad con un valor para la máscara, que indicará si una posición es un vehículo o un valor nulo. A parte, pasaremos otro vector de tres posiciones para los objetos estáticos ordenados por las tres zonas establecidas con su respectiva máscara también.

De esta forma podemos ver la organización del vector en la tabla 6.3:

Característica	Breve descripción	Nº de datos
Serie actual	Puede valer 0 o 1	1
Características EGO	Ver la tabla 6.4	7
Objetos móviles	Ver la tabla 6.5	50
Objetos estáticos	Ver la tabla 6.6	18
Máxima probabilidad	Entre todos los objetos	1
Máx probabilidad Móv	Sólo objetos móviles	1
Máx probabilidad Est	Sólo objetos estáticos	1
Colisión detectada	1 o 0	1
	Total de características	**80**

Cuadro 6.3: Vector de características calculadas fase 2

Característica	Breve descripción	Nº de datos
Velocidad	En los ejes X e Y, en m/s	2
Giro Volante	En grados	1
Aceleración	De 0 a 1	1
Freno	De 0 a 1	1
En intersección	True o False	1
En intersección futura	True o False	1
	Total	**7**

Cuadro 6.4: Características calculadas vehículo EGO fase 2 y 3

Aunque el total de características calculadas sea 80, muchas de ellas luego no son utilizadas para entrenar el modelo. La decisión de incluirlas a priori todas, es para poder experimentar con diferentes configuraciones. Por ejemplo, si se requiere entrenar un modelo para que sólo tenga en cuenta los objetos móviles se dispone de la probabilidad máxima teniendo en cuenta únicamente estos.

En la sección de desarrollo del modelo y en el capítulo resultados se explicarán que características son usadas para las distintas pruebas realizadas.

6.1.3. Fase 3

Objetivos iniciales

Tras probar distintas configuraciones para entrenar el modelo, los resultados de la fase 2, a pesar de mejorar a los obtenidos durante la fase 1, no llegan a ser lo suficientemente buenos como para dar por válido el modelo. Se llega a varias conclusiones, partiendo de que el problema sigue siendo demasiado complejo, por lo que la modificación principal en esta fase es la simplificación de este. Ahora, en vez de generar un dato por cada instante de la simulación, combinando toda la información de todos lo objetos detectados, el modelo funcionará únicamente con la información de un único

Característica	**Breve descripción**	**Nº de datos**
Máscara	1 o 0	1
Velocidad	En los ejes X e Y, en m/s	2
Velocidad de aproximación	En m/s	1
Distancia	Al sensor en m	1
Posición relativa	En los ejes X e Y, en m	2
Ángulo	En grados	1
Identificador	Número positivo	1
Probabilidad de colisión	En porcentaje	1
	Total	**10**
	Total cinco objetos	**50**

Cuadro 6.5: Características calculadas objetos móviles fase 2 y 3

Característica	**Breve descripción**	**Nº de datos**
Máscara	1 o 0	1
Posición relativa	En los ejes X e Y, en m	2
Ángulo	En grados	1
Distancia	Al sensor en m	1
Probabilidad de colisión	En porcentaje	1
	Total	**6**
	Total tres objetos	**18**

Cuadro 6.6: Características calculadas objetos estáticos fase 2

objeto. Es decir, se ha pasado de intentar resolver el problema desde una perspectiva global, a enfocarnos en resolver cada uno de los problemas individuales que lo constituyen, los objetos del entorno, y posteriormente juntar las soluciones para resolver el problema global, es decir, comprobar si en alguna de las soluciones existe una probabilidad de colisión alta.

Se intenta dar un paso más allá, descomponiendo el problema en varias etapas. En vez de calcular la probabilidad final directamente a partir de los datos, se plantea una alternativa. Calcular primeramente las trayectorias futuras de los objetos, para después, a partir de estos datos obtener la probabilidad final.

Estas dos ideas constituyen, en resumen, la motivación para lograr buenos resultados.

Explicación de sensores utilizados y su configuración

En esta fase, los sensores utilizados y su configuración sigue siendo la misma que durante la fase 2 Fig. 6.5, obteniendo la misma nube de puntos del entorno a partir del sensor Lidar Fig. 6.6 y la señal del sensor de colisión cuando se produce una.

Característica objetivo a predecir

Cómo se ha explicado en las fases anteriores, primeramente se planteó poder predecir la función de probabilidad para cada uno de los objetos, para luego quedarnos con el valor máximo en cada instante de esta función de cada uno de estos. Sin embargo, los resultados seguían sin mejorar mucho, por lo que se planteó separar el problema en varias etapas, prediciendo en primer lugar la trayectoria futura. Se explicará con más detalle durante el capítulo 7.

La idea inicial es, a partir de estas trayectorias futuras, predecir la probabilidad de colisión, sin embargo, debido a que se realizaron varias pruebas con diferentes configuraciones, se ha encontrado una solución que muestra buenos resultados. En lugar de calcular esa probabilidad, predecir directamente el indicador de colisión. Es decir, el sensor de colisión en cada instante genera una salida binaria, si ha habido colisión o no, de forma que el modelo aprende a predecir este valor. Por lo tanto, en esta fase la característica objetivo a predecir pasa a ser determinar si se produce o no una colisión.

Escenarios utilizados para generar los datos

Para generar los datos en esta fase se ha utilizado principalmente el escenario generado con *Scenic* ya que proporciona más datos donde se producen colisiones. El escenario utilizado ha sido el mismo que el explicado durante la sección 6.1.2.

Características calculadas a partir de la información de los sensores

Se sigue usando exactamente las mismas funciones, con pequeñas modificaciones para en vez de trabajar con un número fijo de objetos, poder calcular las características para todos y cada uno de los objetos móviles detectados. De forma que los cálculos realizados siguen siendo los mismo que los explicados durante la fase 2 6.1.2.

Debido a que se separan los objetos, no tiene mucho sentido entrenar a un modelo con datos de objetos móviles y objetos estáticos, ya que cada grupo tiene comportamientos diferentes, y lo más eficiente sería entrenar con cada grupo un modelo diferente. Aun así, en esta fase, solamente nos centramos en predecir las colisiones con objetos móviles, que plantean un problema más complejo de resolver y justifican en mayor medida, el uso de una red neuronal.

Vector de características

Cómo ya se ha explicado antes, ahora el vector de características contendrá sólo las variables de un objeto detectado, a parte de las variables del vehículo EGO, como podemos ver en la figura 6.7.

Característica	**Breve descripción**	**Nº de datos**
Serie actual	Puede valer 0 o 1	1
Características EGO	Ver la tabla 6.4	7
Objeto móvil	Ver la tabla 6.5 *Sin la máscara	10
Colisión detectada	1 o 0	1
	Total de características	**19**

Cuadro 6.7: Vector de características calculadas fase 3

6.2. Implementación en CARLA

En esta sección se explicará cómo se ha implementado el código para poder trabajar con el simulador de CARLA. Todo el código de este trabajo se puede encontrar en el repositorio josemponce.

La principal dificultad a la hora de realizarlo ha sido asegurar una correcta sincronización entre el servidor, en este caso el simulador, y el cliente, nuestro código de *Python*. Posteriormente, lo más importante ha sido garantizar una configuración adecuada para los escenarios que se pretendían generar. En este sentido, por ejemplo, interesa que hubieran una gran cantidad de vehículos, para recoger la mayor cantidad de datos posible, así como forzar situaciones más peligrosas que pudieran terminar en colisiones. Esto último se lograba modificando el agente del simulador que se encargaba de controlar los vehículos, debido a que todos, incluso el EGO, estaban configurados con piloto automático. De forma que para lograr situaciones de mayor riesgo, como se explica en la sección 6.1.1, especificábamos que el 20 % de los vehículos fueran un 30 % más rápido del límite marcado, se saltaran el 70 % de las veces la señalización y sólo mantuvieran un metro de seguridad con el vehículo de delante.

A parte de una correcta configuración del servidor, en esta implementación también se debe asegurar que los datos generados se guarden correctamente. En este caso, usando archivos de tipo *Comma Separated Values* (CSV). Además de poder importar correctamente un modelo entrenado en el entorno local para realizar las predicciones correctamente.

Esta implementación se mantiene igual para todas las fases, aunque cambia un poco en función principalmente de qué sensor o sensores se estén utilizando y qué datos o información se esté generando, como se ha explicado durante la sección del diseño de las fases 6.1.

Se distinguen dos implementaciones distintas en función de si estamos usando sólo CARLA o con *Scenic*. La diferencia recae sobre el modo en el que *Scenic* funciona, el cuál obliga a cambiar la estructura del código aunque algunas partes se mantienen igual. Este cambio es debido a que *Scenic* pasa a ser el cliente, que se comunica con el servidor, por lo que el código implementado debe, en medio de esa comunicación, ajustarla para poder asegurarnos de que los datos generados sigan siendo correctos.

Para facilitar la explicación se crearán esquemas visuales que detallarán de forma más simple cómo se ha implementado en los escenarios que se han planteado.

6.2.1. Escenario libre

Como ya se viene explicando antes, este es el caso donde nuestro programa se comunica directamente con el servidor, es decir, es el cliente. De esta forma, debemos realizar toda la configuración en el servidor, desde activar el modo síncrono, hasta crear los agentes que actuarán en el escenario generado.

De esta forma, podemos ver en la Fig. 6.8 el esquema de esta implementación en el escenario libre. La flecha gris claro continua simboliza el flujo de ejecución del programa, mientras que la flecha discontinua negra refleja la comunicación entre cliente y servidor.

Cómo se puede ver en la figura, el primer paso es establecer la conexión con el servidor, para posteriormente configurarlo según las necesidades. Del mismo modo, se crean todos los agentes que se necesitan en el escenario, donde se incluyen vehículos y sensores.

Luego se entraría en el bucle donde se empieza a recoger los datos generados por el simulador. Debido a que todos los vehículos son controlados por el piloto automático, no es necesario en cada iteración especificar ningún comportamiento específico, simplemente avisar al servidor cuando debe generar el siguiente paso, *frame* o iteración de la simulación.

Finalmente, antes de terminar la comunicación con el servidor, se debe asegurar que se eliminan todos los agentes que se han creado y volver a configurar el simulador en el modo asíncrono, ya que en el caso contrario, el servidor se queda eternamente esperando a una señal para actualizarse, y el programa se queda bloqueado.

6.2.2. Escenarios personalizados con *Scenic*

El cliente pasa a ser *Scenic*, el cual se encarga de configurar el simulador, así como crear los agentes que se hayan descrito en el diseño del escenario, en un archivo ".scenic". En este archivo se añade, en el momento de la creación del vehículo EGO, una llamada a una función de mi programa de *Python*, el cual se encarga de añadir el sensor Lidar y el de colisión. De forma simple, cuando el simulador genere datos del sensor Lidar, este llamará al método `__call__` de una instancia de una clase creada para este fin, es decir, llamará a mi código. Este se encargará de calcular todas las operaciones necesarias y guardar los datos o en el caso de que se quiera predecir valores futuros, llamar a los modelos entrenados.

El principal problema para implementar el código con *Scenic* es asegurar una correcta sincronización entre este, el simulador y mi código con los

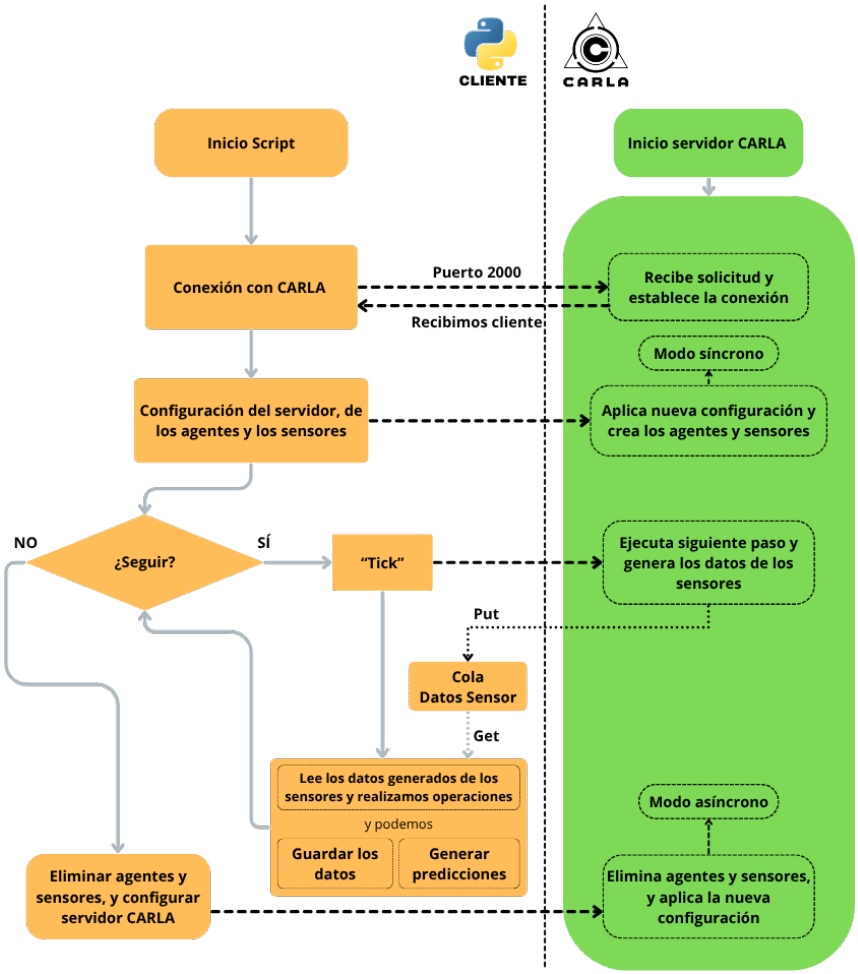

Figura 6.8: Esquema código escenario libre

sensores. Esta dificultad se debe a que cuando el simulador genera los datos y, por lo tanto, llama a la función `__call__` de mi clase, esta se ejecutaba en un hilo distinto al de *Scenic*. De esta forma, *Scenic* sigue generando la siguiente iteración de la simulación, sin esperarse a que mi código finalice de realizar los cálculos y de guardar o predecir nuevos datos. Para poder garantizar esta sincronización se ha tenido que modificar parte del código de la librería de *Scenic*, ya que de la forma que viene implementado no deja ninguna posibilidad para lograrlo. Esta modificación únicamente fue necesaria en una función, llamada `step()`, que como su nombre indica, se encarga de mandar la señal al simulador para generar el siguiente paso del escenario.

Para realizar esta sincronización se ha utilizado la librería de `threading` de *Python*, que proporciona una interfaz de alto nivel para crear y gestionar

hilos. En concreto, se ha utilizado la clase `Event()` que permite sincronizar de forma sencilla hilos a través de una bandera interna. Primeramente, desde *Scenic*, se marca la bandera a `False`, con la función `clear()`. Posteriormente, manda la señal al simulador para que genere la nueva iteración y justo después, a través de la función `wait()`, bloquea el hilo hasta que se reciba una señal `True`, emitida por el código desarrollado tras finalizar todos los cálculos.

De esta forma, se van ejecutando un escenario tras otro a partir de una única definición, hasta que se indique se detenga el programa, como se puede ver en la Fig. 6.9.

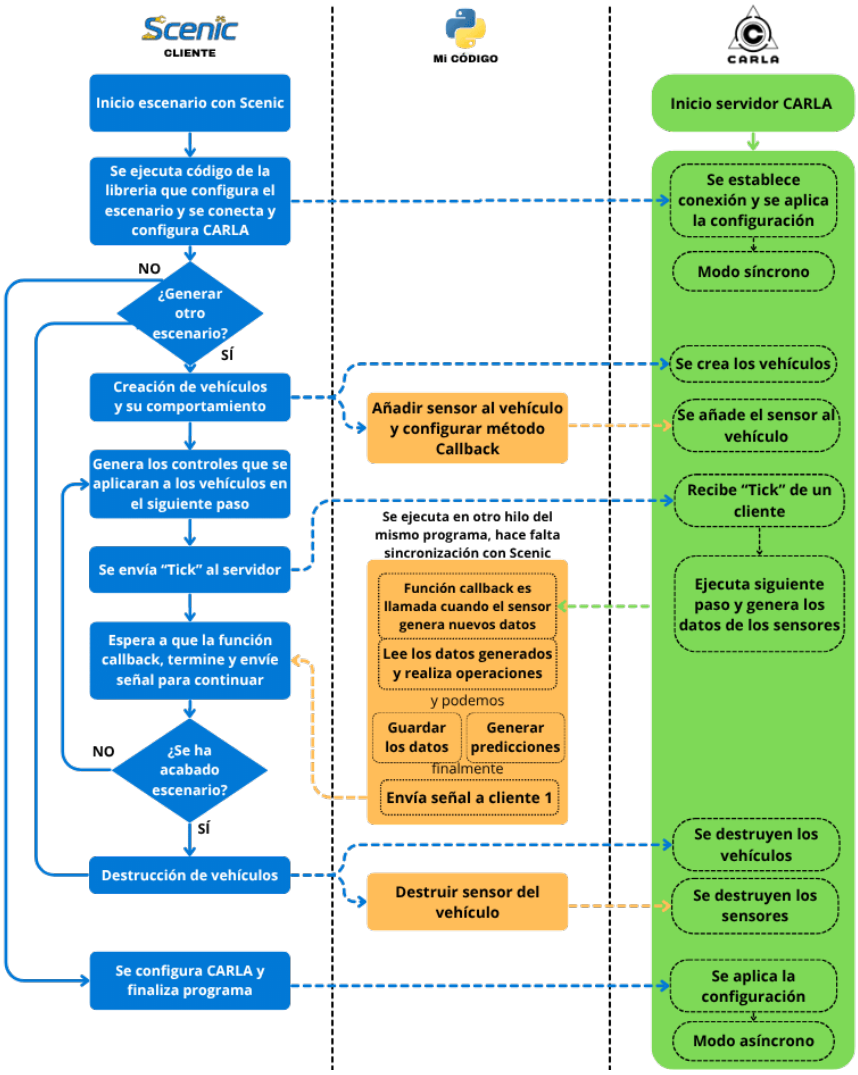

Figura 6.9: Esquema código con *Scenic*

Al finalizar el programa, *Scenic* se encarga de aplicar la configuración

necesaria para dejar el simulador en buenas condiciones para que no se bloquee.

6.3. Desarrollo de los modelos

En esta sección se explicará el proceso seguido para desarrollar cada uno de los modelos obtenidos. Se empezará explicando brevemente el formato de los datos recogidos del simulador, cómo se han guardado y cómo se han proporcionado al entorno de desarrollo de Google Colab. Posteriormente, los distintos pasos que se han ido realizando para obtener los modelos, divididos en dos grupos.

1. El primero reúne, por un lado, el análisis de los datos y, por otro lado, las transformaciones aplicadas sobre estos.

2. El segundo grupo consta de los pasos para la creación y entrenamiento de los modelos.

Con esta sección se pretende explicar el proceso general llevado a cabo, ya que, cada modelo implementa cada uno de estos pasos de forma distinta. En aquellos casos donde existan diferencias sustanciales, que influyan de forma notoria en el comportamiento del modelo, se desarrollarán más, para explicar las diferencias entre cada uno.

6.3.1. Formato de los datos

Como ya hemos explicado en secciones anteriores, los datos son guardados directamente en un formato CSV, en el ordenador local. Debido a que se desarrollan los modelos en el entorno de *Google Colab*, es necesario subirlos primeramente a *Google Drive*, para posteriormente poder acceder a ellos desde el *notebook*.

El formato de los datos puede verse en la Fig. 6.10.

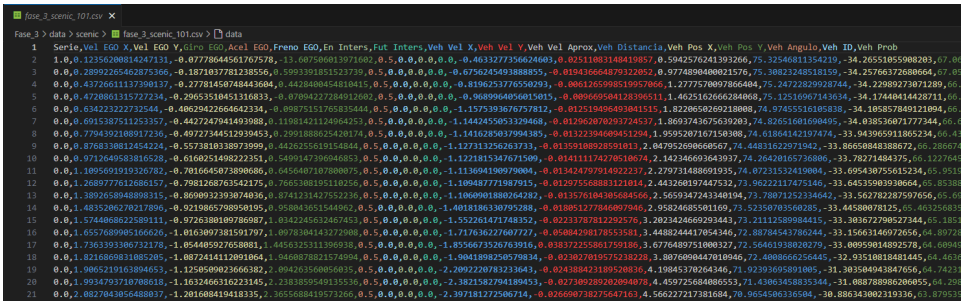

Figura 6.10: Formato de los datos

6.3.2. Transformación de los datos

En esta sección, se comentarán los distintos pasos que reúnen el estudio y análisis de los datos, y posteriormente cada una de las transformaciones aplicadas a estos, necesarias para poder entrenar los modelos y obtener los mejores resultados.

Estudio y análisis de las características

El primer paso al comenzar el desarrollo de cualquier modelo es el estudio y análisis de los datos. Debido a que se han generado los datos, se conoce cada una de las características, sus rangos y significado. Sin embargo, debo comprobar que la lectura por parte de los sensores han sido correctas, sin producir *outliers* o valores nulos.

Utilizando la librería *Pandas*, todos estos análisis se pueden realizar fácilmente, con una llamada a un método. Esta librería nos ofrece también la posibilidad de leer los datos directamente desde del *Drive*, una vez que este está ya montado en el entorno de *Google Colab*.

Para conocer la existencia de valores nulos, producidos por lecturas erróneas de los sensores, se puede utilizar la función `info()`, que nos daría una salida parecida a la Fig. 6.11. Se puede ver que ninguna de las características tiene valores nulos, por lo que los sensores han leído todos los datos correctamente. Nunca se ha producido un valor nulo por los sensores en todas las pruebas que se han realizado. De esta forma, no será necesario ningún proceso adicional para resolver el problema de valores nulos.

```
Data columns (total 15 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Serie          42817 non-null  float64
 1   Vel EGO X      42817 non-null  float64
 2   Vel EGO Y      42817 non-null  float64
 3   Giro EGO       42817 non-null  float64
 4   En Inters      42817 non-null  float64
 5   Fut Inters     42817 non-null  float64
 6   Veh Vel X      42817 non-null  float64
 7   Veh Vel Y      42817 non-null  float64
 8   Veh Vel Aprox  42817 non-null  float64
 9   Veh Distancia  42817 non-null  float64
 10  Veh Pos X      42817 non-null  float64
 11  Veh Pos Y      42817 non-null  float64
 12  Veh Angulo     42817 non-null  float64
 13  Veh ID         42817 non-null  float64
 14  Colision       42817 non-null  float64
dtypes: float64(15)
memory usage: 4.9 MB
```

Figura 6.11: Salida del método info() de Pandas

Por otro lado, para conocer la existencia de *Outliers*, así como la distribución de los datos y su rango, es muy útil la creación de gráficos estadísticos como histogramas. En estos gráficos se representa la distribución

del conjunto de datos, divididos en intervalos que representan la frecuencia o recurrencia de estos en el grupo. *Pandas* proporciona de la misma manera métodos con los que poder crear automáticamente estos histogramas sobre cada característica. La Fig. 6.12 es un ejemplo de histograma creado para visualizar la distribución de la característica *Veh Distancia*, que representa la distancia a otro vehículo. Tras realizar pruebas se comprueba que no hay *outliers* ya que todos datos obtenidos de todas las características se mueven en el rango válido.

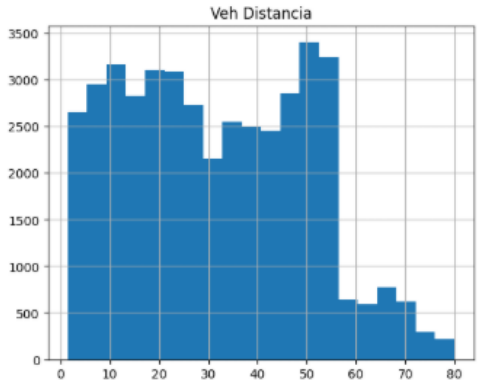

Figura 6.12: Histograma de la característica *Distancia a otro vehículo*

A través del histograma se puede apreciar también el balanceo de los datos en la variable a predecir y otras pruebas adicionales para comprender mejor las particularidades de las características.

Selección de las características

En secciones anteriores se ha comentado que, a la hora de generar los datos, son calculadas y guardadas muchas más características de las que luego se utilizarán para entrenar el modelo. En esta etapa, se escogen cuáles de estas serán utilizadas para entrenar el modelo en cuestión. Se han realizado muchas pruebas distintas a lo largo de cada una de las fases, cada una con su propio conjunto de características. Un ejemplo se puede ver en la Fig. 6.11, que representa el conjunto de características escogidas para realizar un modelo de la fase 3. También, para poder compararlo con otro ejemplo, la Fig. 6.13 representa las características usadas para entrenar un modelo de la fase 2. Se puede apreciar en este último ejemplo que se deciden usar únicamente las variables relacionadas con los objetos móviles.

Normalización

La normalización de los datos de entrada al modelo es un paso indispensable, que mejora notablemente el rendimiento del modelo. Resumiendo, las

```
RangeIndex: 22111 entries, 0 to 22110
Data columns (total 37 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Serie                22111 non-null  float64
 1   Vel EGO X            22111 non-null  float64
 2   Vel EGO Y            22111 non-null  float64
 3   Giro EGO             22111 non-null  float64
 4   En Inters            22111 non-null  float64
 5   Fut Inters           22111 non-null  float64
 6   Veh MASK 0           22111 non-null  float64
 7   Veh Vel X 0          22111 non-null  float64
 8   Veh Vel Y 0          22111 non-null  float64
 9   Veh Pos X 0          22111 non-null  float64
 10  Veh Pos Y 0          22111 non-null  float64
 11  Veh ID 0             22111 non-null  float64
 12  Veh MASK 1           22111 non-null  float64
 13  Veh Vel X 1          22111 non-null  float64
 14  Veh Vel Y 1          22111 non-null  float64
 15  Veh Pos X 1          22111 non-null  float64
 16  Veh Pos Y 1          22111 non-null  float64
 17  Veh ID 1             22111 non-null  float64
 18  Veh MASK 2           22111 non-null  float64
 19  Veh Vel X 2          22111 non-null  float64
 20  Veh Vel Y 2          22111 non-null  float64
 21  Veh Pos X 2          22111 non-null  float64
 22  Veh Pos Y 2          22111 non-null  float64
 23  Veh ID 2             22111 non-null  float64
 24  Veh MASK 3           22111 non-null  float64
 25  Veh Vel X 3          22111 non-null  float64
 26  Veh Vel Y 3          22111 non-null  float64
 27  Veh Pos X 3          22111 non-null  float64
 28  Veh Pos Y 3          22111 non-null  float64
 29  Veh ID 3             22111 non-null  float64
 30  Veh MASK 4           22111 non-null  float64
 31  Veh Vel X 4          22111 non-null  float64
 32  Veh Vel Y 4          22111 non-null  float64
 33  Veh Pos X 4          22111 non-null  float64
 34  Veh Pos Y 4          22111 non-null  float64
 35  Veh ID 4             22111 non-null  float64
 36  Prob solo vehiculos  22111 non-null  float64
dtypes: float64(37)
memory usage: 6.2 MB
```

Figura 6.13: Características usadas en un modelo de la fase 2

razones para aplicar una normalización son:

- Poder trabajar con características de diferentes rangos, que en el caso de no aplicarse, podrían ser interpretadas unas más importantes que otras por el modelo.
- Ayudar a estabilizar el proceso de optimización en el entrenamiento del modelo, ya que estos trabajan mejor cuando las entradas se mueven en un rango de 0 a 1.

Existen diferentes técnicas para normalizar los datos, en este caso se ha optado por aplicar siempre la normalización *Min-Max*, que transforma todos los valores de cada variable en un intervalo entre 0 y 1. Las razones son como se ha explicado antes, el tener diferentes características con diferentes rangos y mejorar la eficiencia del modelo. Al saber que no existen valores *outliers* y que todos los datos están comprendidos en rangos válidos, no existe ningún impedimento para aplicar esta técnica.

Este proceso de normalización se ha realizado utilizando la librería de *Scikit-learn*, a través de una función de preprocesado de los datos llama-

da `MinMaxScaler()`. Es importante mencionar que esta escala es entrenada únicamente con los datos de entrenamiento. No se puede utilizar para entrenarla los datos de validación o test, porque se cometería un error de fuga de información que sesgaría la evaluación del modelo y podría afectar a su capacidad de generalización. Una vez entrenadas, estas escalas deben ser guardadas y descargadas, para poder aplicarlas a los datos cuando se trabaje en el ordenador local.

Transformación a series supervisadas y estrategias seguidas

Como se explicó en la sección 3.2.3 de las diferentes estrategias para afrontar los problemas de series temporales con modelos neuronales, se debe utilizar alguna de estas técnicas para transformar las series temporales en series supervisadas que los modelos puedan utilizar para realizar las predicciones.

A la hora de predecir posibles colisiones entre vehículos, es interesante poder predecir varios pasos a futuro, ya que desconocemos el rendimiento que tendrá el modelo en cada uno de los pasos predecidos. De esta forma, se puede evaluar mejor el comportamiento del modelo y obtener mejores resultados.

Las dos primeras cuestiones que surgen son: (1) cuántos pasos se deben usar para predecir los valores y (2) cuántos valores se deberían predecir.

1. Se debe tener en cuenta cuantos segundos antes es recomendable utilizar para detectar una colisión. Cambia mucho en función de que tipo de colisión se produzca, pero finalmente se ha decidido usar entre 10 a 5 pasos anteriores, es decir, tomar entre medio segundo y un segundo del entorno. Esto porque si se quieren predecir futuras trayectorias no es necesario tener en cuenta mucho más pasos anteriores. Con dos pasos anteriores, teniendo en cuenta las posiciones, podrían calcularse trayectorias con un movimiento rectilíneo uniforme y con tres pasos se podrían calcular con un movimiento rectilíneo uniformemente acelerado. Se usan algunos pasos más para facilitar el trabajo del modelo, en concreto, 10 pasos anteriores para los modelos de las fases 1 y 2, y 5 pasos anteriores para los modelos de la fase 3.

2. También es resuelta de diferentes formas en cada fase. A continuación, se explicará cada decisión junto con la estrategia escogida para obtener las predicciones.

    a) Los **Modelos de las fases 1 y 2** emplean la estrategia *multi-output*, es decir, utilizar un único modelo para predecir todos los pasos. Principalmente, por la simplicidad para implementarlo y evaluar el rendimiento del modelo en los distintos pasos predecidos. En concreto, se decide predecir los 10 pasos a futuro (un

segundo), ya que sería el tiempo recomendado para poder reaccionar a tiempo e intentar evitar la colisión.

De esta forma la estrategia final es 10 *lags*, 10 *steps* y un único modelo para predecir simultáneamente todos los pasos, tal como se muestra en la figura 6.14.

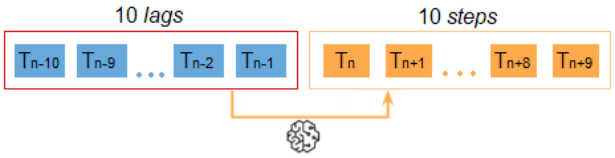

Figura 6.14: Estrategia seguida en las fases 1 y 2

b) Los **modelos de la fase 3** emplean otra estrategia diferente. Se combinan dos técnicas distintas para obtener las ventajas de cada una al mismo tiempo. Por un lado, se emplea una estrategia *Direct multi-step* donde tenemos varios modelos especializados en diferentes pasos de tiempo. Por otro lado, cada modelo emplea la estrategia de *multi-output*, por lo que predice varios pasos en el intervalo de tiempo que le corresponde, con lo que ganamos rendimiento, al no tener un modelo por cada paso a predecir. Las ventajas de esta nueva estrategia son las siguientes:

1) Mejores resultados en cada intervalo, ya que cada modelo se especializa en realizar esas predicciones.
2) Más flexibilidad, por ejemplo, para añadir otro modelo que prediga más pasos en el futuro, sin tener que cambiar ninguno de los ya entrenados. Al mismo tiempo, nos permite aplicar diferentes configuraciones del modelo para cada intervalo, si se logra apreciar diferentes comportamientos, optimizando el rendimiento de cada uno individualmente.

En la solución propuesta para la fase 3, han sido utilizados 3 modelos independientes, cada uno utiliza los 5 pasos anteriores al momento actual y predice 5 pasos futuros en un momento distinto, tal como se muestra en la figura 6.15.

Para la implementación de este código para la transformación a series supervisadas he usado una función de la fuente [30]. Esta función recibe una serie temporal, se especifica cuántos *lags* y *steps* se necesitan y devuelve los datos transformados. Posteriormente, se deben ajustar las series supervisadas para que se adapten al funcionamiento de los modelos de la fase 3.

En este punto es además cuando se tiene en cuenta la variable *Serie* de cada dato. En un mismo archivo de datos pueden haber varias series tempo-

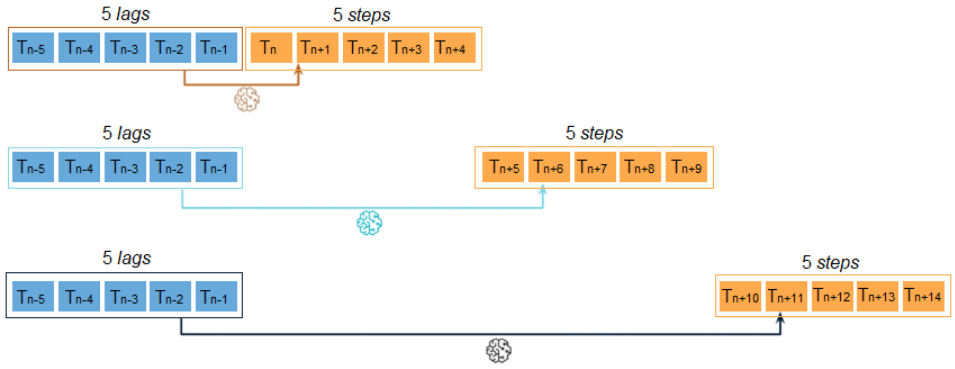

Figura 6.15: Estrategia seguida en la fase 3

rales diferentes, de modo que para dividirlas se crea esta variable. Cuando su valor es 1, quiere decir que comienza una nueva serie temporal, siendo ese dato el primero de la serie y termina hasta encontrar otro dato con la variable a 1 o llegar al final del archivo. Cada una de estas series temporales es transformada a series supervisadas individualmente, para lograr que no se mezclen datos.

Preparación de los datos para los modelos

Las redes neuronales recurrentes LSTM, explicadas en la sección 3.2.1, necesitan que se les proporcione los datos en una matriz con la forma $(datos, lags, caracteristicas)$. Los datos de salida deberán tener también la forma $(datos, steps, caracteristicas)$. Se puede ver en la figura 6.16 un ejemplo de los datos para entrenar un modelo de la fase 3, donde se usan 13 características como entrada y se quieren predecir otras 3 de salida. Hay tres conjuntos de datos para entrenar y validar, ya que como se ha explicado en la sección anterior, cada modelo necesita sus propios datos. El número 5 corresponde al número de *lags* y *steps*, que coincide.

```
Shape of training data _5:
train_X: (38049, 5, 13)
train_y: (38049, 5, 3)

Shape of validation data _5:
validation_X: (5638, 5, 13)
validation_y: (5638, 5, 3)
```

```
Shape of training data _10:
train_X: (35667, 5, 13)
train_y: (35667, 5, 3)

Shape of validation data _10:
validation_X: (5275, 5, 13)
validation_y: (5275, 5, 3)
```

```
Shape of training data _15:
train_X: (33379, 5, 13)
train_y: (33379, 5, 3)

Shape of validation data _15:
validation_X: (4922, 5, 13)
validation_y: (4922, 5, 3)
```

Figura 6.16: Forma de los datos de entrada y salida para la fase 3

6.3.3. Implementación de los modelos

Una vez que los datos están listos, se puede proceder con el entrenamiento de los modelos. En esta sección se detallará como se han creado los modelos y la metodología empleada para entrenarlos.

Creación del modelo

Para la creación de los modelos se han utilizado las librerías de *tensorflow* y *keras*, sección 4.3. Para diseñar una red neuronal, se tienen que tener en cuenta muchos aspectos distintos, tratando de escoger la mejor configuración posible para optimizar el modelo. A continuación, se detallará cada uno de los parámetros para diseñar la red neuronal, tratando de explicar que configuraciones se han utilizado.

- **Tipo y números de capas neuronales**

  Respecto al tipo de capa neuronal, como se viene explicando en capítulos anteriores, se ha decidido usar redes LSTM. De esta forma, en todos los modelos al menos la primera capa siempre ha sido con este tipo de neuronas. Dependiendo de la forma de los datos de salida, es decir, si se requería predecir únicamente una variable o varias, la última capa utilizada era distinta. En las fases 1 y 2, sólo se predice una única variable (la probabilidad), por lo que se añade una capa densa al final. En la fase 3, para lograr obtener una salida con la forma explicada durante la sección 6.3.2, se ha usado, entre distintas posibilidades, una capa *TimeDistributed*.

- **Número de neuronas por capa**

  A parte de la decisión del número de capas, el número de neuronas en cada una es el punto más importante para lograr un equilibrio entre el *overfitting* y el *underfitting*. Las redes LSTM tienen una gran capacidad de memorización, por lo que al añadir más neuronas de las necesarias, únicamente se lograría *overfitting*.

- **Función de activación de las neuronas en cada etapa**

  Una función de activación 3.1 muy popular debido a su gran rendimiento es la activación ReLU, que se asemeja a cómo funciona y realiza los cálculos un procesador, logrando agilizar el proceso de entrenamiento. Por contra, se encuentra con el problema de desvanecimiento de señal que provoca que algunas neuronas estén inactivas.

- **Función de activación de la salida**

  Dependiendo del formato de salida que se necesite se puede usar una función de activación u otra. En las dos primeras fases se hace uso de

la activación sigmoide, ya que se está intentando predecir una probabilidad.

- **Optimizador**

  Se emplea el optimizador Adam 3.2.2 que es el más popular en el campo del *machine learning*, que ayuda a mejorar el rendimiento del entrenamiento.

- **Función de pérdida**

  Para las dos primeras fases se hace uso del *Error cuadrático medio* (MSE), para acentuar más los errores grandes frente a los errores pequeños. En la tercera fase se crea una función de pérdida personalizada, ya que una de las variables que se pretende predecir es la distancia frente a otros vehículos. En este contexto de accidentes, las colisiones se producen a distancias cortas, por lo que es preferible que el error sea el menor posible en estas situaciones, restando importancia a un error más grande, pero a una distancia mucho mayor. Esta función de pérdida, por lo tanto, acentúa mucho más los errores producidos en distancias cortas frente a distancias grandes, dejando el resto de variables a predecir con el error MSE. La función creada para penalizar el error en la distancia es la función de la Fig. 6.17.

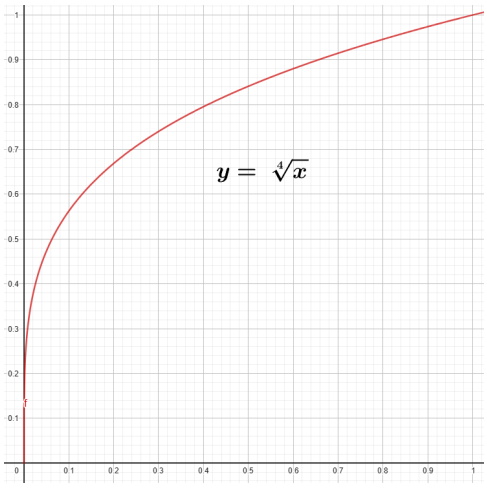

Figura 6.17: Función de pérdida personalizada para la distancia

El sensor Lidar esta configurado para capturar los láseres a una distancia máxima de 80 metros por lo que, una vez normalizados los datos, un valor próximo a 1 querrá decir que realmente son casi 80 metros. Una distancia corta que se podría considerar peligrosa para que se produzca una colisión podría ser aquella que fuera menor a 5

metros, que aplicando la normalización anterior se situaría entorno a 0'06 aproximadamente.

Entrenamiento

Para realizar el entrenamiento se debe de tomar un número de épocas lo suficientemente grande para que el modelo se ajuste bien, pero procurando evitar el sobreajuste. De media se han realizado entre 50 a 100 épocas.

Otro aspecto importante utilizado para entrenar los modelos ha sido el uso de una función de *EarlyStopping*. De esta manera, se puede detener el entrenamiento si durante un número de años fijado en esta función, no se ha logrado mejorar el error con los datos de validación.

6.3.4. Ajuste del modelo, para la fase 3

Como se explicó en la sección 6.1.3, durante la fase 3, la característica objetivo a predecir era directamente el valor de colisión, es decir, si se produce una colisión (1) o no (0). El valor que calcula el modelo para predecir esta variable es continuo, es decir, que se mueve entre 0 y 1. Por lo tanto, debemos de interpretar este valor de la mejor forma posible, para optimizar el comportamiento o desempeño del modelo.

Tal como se explica en la sección 3.3.2 del capítulo de fundamentos, se realiza un estudio sobre el umbral óptimo para cada modelo entrenado. De esta forma, logramos optimizar el rendimiento de cada uno, con la mayor precisión y *recall* posibles.

6.4. Diseño e Implementación del sistema anti-colisión

En esta sección se mostrará el proceso seguido para diseñar e implementar el sistema anti-colisión, que conforma el objetivo extra de este trabajo. Como ya se explicó anteriormente, este sistema se implementa sobre el modelo obtenido durante la fase tres, que es el que mejor resultados refleja.

6.4.1. Objetivo inicial

Los objetivos que se esperan conseguir con la implementación de este sistema son los siguientes:

- **Principal: Evitar que se produzca la colisión.**
- **Secundario: Reducir daños en los ocupantes del vehículo.**

En las siguientes secciones se mostrará el proceso seguido para lograrlo.

6.4.2. Tipos de colisiones existentes

El escenario que se plantea para la fase tres, como ya se menciona en secciones anteriores 6.1.2, consiste en un cruce donde hay dos vehículos que están llegando y uno de los dos se salta la señalización, por lo que el tipo de colisión que se produce suelen ser casi siempre laterales o frontales.

Un punto importante es que las colisiones se producen siempre con los dos vehículos en movimiento, o lo que es lo mismo, que no hay uno que esté detenido.

6.4.3. Planteamiento inicial

El tipo de acciones que podemos aplicar al vehículo son, acelerar, frenar y girar. Acelerar se descarta por que en la mayoría de ocasiones terminaría haciendo el problema mucho más complejo, por lo que a priori se plantea aplicar un freno y un giro.

Respecto al freno la idea es clara, frenar siempre que se pueda, es decir, en todas las ocasiones menos cuando el vehículo con el que vamos a colisionar esté detrás nuestra, en este caso frenar únicamente empeoraría el impacto y los daños causados. Para delimitar esta zona de no frenado, se crean tres zonas de diferente tamaño empezando desde la mitad del vehículo hacía la parte trasera, como se puede apreciar en la figura 6.18.

Cómo se ha explicado en la sección 6.3.2, en la fase 3 se detectan las colisiones con tres modelos diferentes al mismo tiempo, cada uno especializado en detectar las colisiones en un instante diferente, desde 0,5 segundos hasta 1.5 segundos. De esta forma, la distancia a la que se detectarán las colisiones contra el vehículo, serán diferente según el modelo que lo haya predicho. Esto justifica por que tenemos tres dimensiones diferentes para la

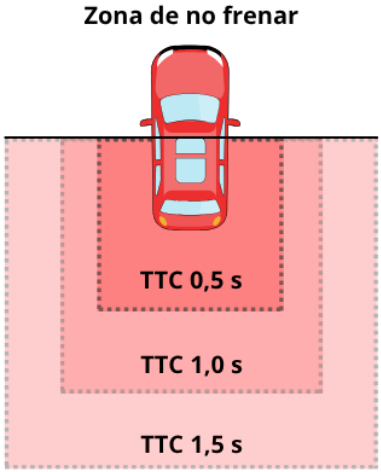

Figura 6.18: Zonas de no frenado del sistema anti-colisión

zona de no frenado. Las dimensiones escogidas para realizar estas zonas se han calculado como: la distancia (D) que recorrería el vehículo en el tiempo restante para la colisión (TTC) a la velocidad a la que se está aproximando (V_{aprox}), como se ve en la siguiente ecuación y en los ejemplos de la tabla 6.8

$$D(m) = V_{aprox}(m/s) * TTC(s)$$

Ejemplos	Modelo $1,5s$	Modelo $1,0s$	Modelo $0,5s$
$V_{aprox} = 2m/s = 7,2km/h$	$3m$	$2m$	$1m$
$V_{aprox} = 5m/s = 18km/h$	$7,5m$	$5m$	$3,5m$
$V_{aprox} = 10m/s = 36km/h$	$15m$	$10m$	$5m$

Cuadro 6.8: Sistema Anti-Colisión, dimensiones zona de no frenado

La razón de por que no siempre que se detecte un vehículo a cualquier distancia en la parte trasera frenar, es por que hay veces que aunque un vehículo pueda estar en esa zona y se vaya a producir una colisión, esta se puede evitar frenando, como por ejemplo en la incorporación a una rotonda.

Otro aspecto importante a tener en cuenta es que se puede configurar la fuerza con la que se quiere frenar, de este modo a medida que queda menos para la colisión, se frenará más fuerte. A $1.5s$, debido a que este modelo es el que tiene menos precisión 7.3 se aplica un frenado muy suave, a $1,0s$ se aplica un frenado más intenso y a $0,5s$ se aplica un frenado máximo.

La otra acción posible, como ya se ha comentado, es la aplicación de un giro. En principio, puede ayudar a evitar muchos casos de colisiones y sobre todo a reducir los daños en los ocupantes del vehículo. Sin embargo, al mismo tiempo plantea un problema mucho más complejo de resolver, ya que ahora hay que tener en cuenta muchos más aspectos. Por ejemplo, comprobar que

la zona a la que se quiere girar esté despejada, intentar en lo posible no invadir otros carriles sobre todo en dirección contraria, etcétera.

En un primer momento, se planteó aplicar este giro para intentar disminuir todavía más las posibilidades de una colisión. La idea giraba entorno a esquivar o ir en la dirección contraria en la que venía el vehículo con el que se había detectado la colisión.

Como se puede ver el proceso seguido para calcular la dirección en la que girar es:

1. Determinar desde que ángulo viene el vehículo.

2. En el caso de que venga por la parte trasera, es decir, cuadrantes III y IV, aplicar una transformación de reflexión especular al ángulo para trasladarlo a los cuadrantes I y II.

3. En el caso de que esté en el cuadrante II, es decir, ángulo resultante mayor a 0^{o}, restamos a dicho ángulo 90^{o}, que terminará quedando en el cuadrante I. Del mismo modo se hace la operación inversa si el ángulo resultante se sitúa en el cuadrante I.

4. Si tras comprobar que la zona elegida con el ángulo calculado está obstruida, se intentará girar con un ángulo mayor, en el cuadrante II y menor en el cuadrante I, hasta que se encuentre una zona despejada. En el caso de que no se logre encontrar con este método una zona despejada, no se aplicará ningún giro.

Por lo tanto, el diseño inicial planteado para el sistema anti-colisión se puede apreciar en la figura 6.19, donde se aplican controles tanto en el freno como en el volante para aplicar un giro.

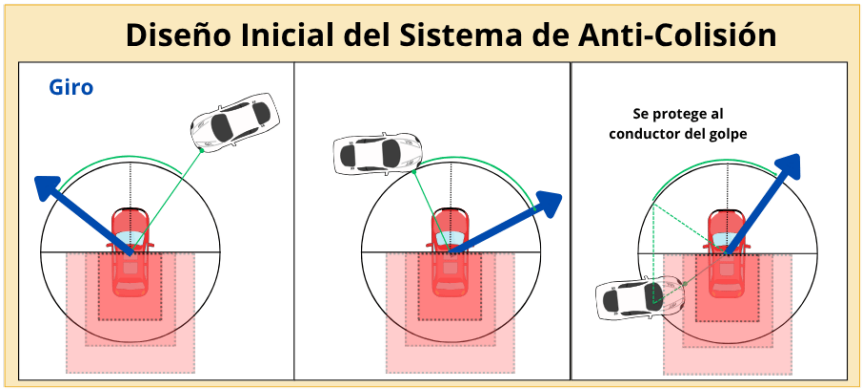

Figura 6.19: Diseño inicial sistema anti-colisión

6.4.4. Diseño Final

Aunque la aplicación de un giro puede ofrecer una gran ayuda para evitar la colisión, los impedimentos que conlleva, debido a la complejidad de la situación, han llevado a que no se tenga en cuenta para el diseño final. Cómo se ha planteado en la sección anterior, hay que tener en cuenta que la zona a la que se quiera girar esté libre, pero además, en un entorno como es la carretera donde, a parte de encontrarnos con objetos estáticos, en muchas ocasiones hay otros vehículos o personas a poca distancia. Por lo tanto, un sistema que aplique un giro al volante, debe estar muy seguro de que no genere otros problemas más graves, teniendo en cuenta otros factores como las trayectorias de otros vehículos y personas que puedan pasar por la zona a la que se plantea ir. Del mismo modo, el proceso de detectar una zona despejada para aplicar el giro, supone un problema bastante complejo para un sistema de visión por computador, por lo que con un sensor Lidar es aún más complicado.

Por todas estas razones se decide prescindir de aplicar un control sobre el volante del vehículo, quedándonos únicamente con el control sobre el freno, que se mantendrá con la misma configuración que en el planteamiento inicial. Se puede apreciar el diseño final en la Fig. 6.20.

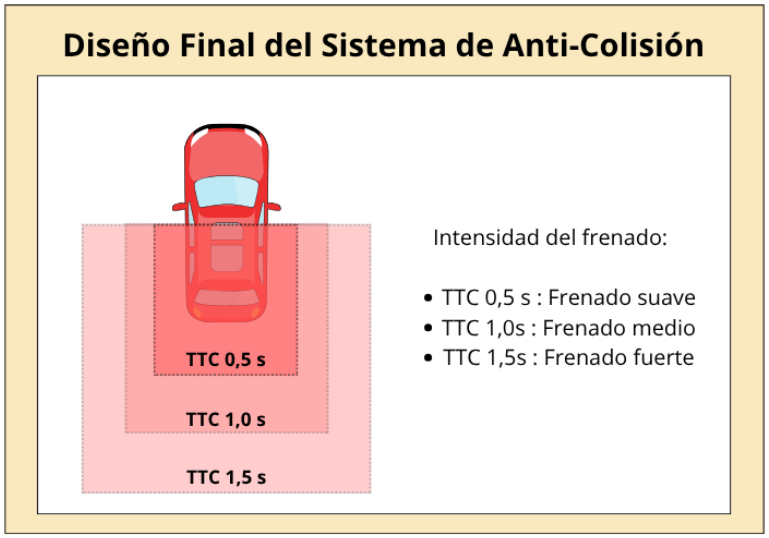

Figura 6.20: Diseño final sistema anti-colisión

6.4.5. Implementación con *Scenic*

A modo de resumen, el funcionamiento de *Scenic* consiste en calcular en cada iteración los controles que se aplicarán a cada agente del escenario para después pasárselos al simulador. Este se encargará de aplicarlos en

la siguiente iteración o *frame*. De esta manera, cuando llegamos a nuestro método `step()` de *Scenic*, los controles ya están establecidos en el simulador para el vehículo EGO. De esta forma, se debe sobrescribir esos controles con los obtenidos por el sistema anti-colisión, antes de realizar la señal al simulador para que se actualice el *frame*.

Este es el proceso seguido, a través de dos variables globales a las que acceden ambos hilos, el que ejecuta *Scenic* y el que ejecuta la llamada a la función `__call__` de nuestro código:

- La primera indica si se desea o no sobrescribir los controles.
- La segunda define a través de un objeto `carla.VehicleControl()` los controles que se desean aplicar.

Es importante mencionar que cuando se detecta una colisión se activa un contador para determinar cuanto tiempo debe de aplicarse los controles de emergencia. Cada vez que se detecte por parte de un modelo una colisión, este contador se actualizará a un valor concreto en función de qué modelo la haya detectado, siguiendo la siguiente tabla 6.9. En caso de que dos o tres modelos detecten en la misma iteración una colisión, se aplicará el mayor valor. Este contador se reducirá una unidad por cada iteración en la que no se produzcan predicciones de colisiones.

Modelo	**Nº iteraciones**	**Segundos**
Modelo $0,5s$	10	$1s$
Modelo $1,0s$	6	$0,6s$
Modelo $1,5s$	4	$0,4s$

Cuadro 6.9: Duración del control de emergencia del Sistema Anti-Colisión

Capítulo 7

Pruebas y resultados

En este capítulo se explicarán las distintas pruebas realizadas y los resultados obtenidos en cada una de las fases realizadas. En cada fase, se han ido realizando distintas pruebas. La idea era realizar distintas configuraciones de características y modelos para poder observar resultados diferentes y lograr entender mejor que planteamiento se acerca más a los objetivos planteados.

Se desarrollará principalmente la fase 3, debido a que el modelo obtenido en esta fase es el que se ha escogido como modelo final para este proyecto. De esta forma, en las fases 1 y 2 se explicará brevemente una prueba de concepto realizada en la que se obtuvieron los mejores resultados.

En general, el proceso seguido para explicar cada una de las pruebas realizadas será el siguiente:

- Análisis de las características utilizados en el modelo.
- Diseño del modelo entrenado.
- Análisis Resultados obtenidos.

En la explicación de la fase 3 se añadirán más apartados. Finalmente, se explicarán también los resultados obtenidos del sistema anti-colisión realizado.

7.1. Fase 1

Como se explicó durante la sección 6.1.1, en esta fase se ha utilizado un sensor Radar colocado en la parte frontal del vehículo. En total se generó una base de datos de 30.000 datos, obtenidos únicamente del simulador en el modo libre, sin usar *Scenic*.

A continuación, se explicará la prueba realizada durante la fase 1, con la que obtuvimos los mejores resultados.

Características utilizadas

Recordando un poco las características calculadas para la fase 1, sección 6.1.1, se guardan los 5 objetos detectados más cercanos, ordenados según la distancia, siendo el objeto 5 es el más alejado. Después de estudiar cada una de estas características, se ha observado, por un lado, que nunca se llegó a obtener información de un quinto objeto, es decir, como máximo se lograban detectar 4 objetos alrededor del vehículo. Por otro lado, la variabilidad de las características de este cuarto objeto detectado es muy poca, lo que quiere decir que la mayoría del tiempo no se han detectado más de 3 objetos. Que una variable tenga poca variabilidad indica que no proporcionará mucha información al modelo para realizar las predicciones, es decir, no aporta mucho valor. Por esto, se ha decidido prescindir de las variables de estos dos objetos más lejanos y quedarse únicamente con los tres más cercanos.

Una vez transformados los datos a series supervisadas, el vector de características final utilizado para entrenar el modelo quedaría según el cuadro 7.1:

Característica	**Breve descripción**	**Nº de datos**
Vector objetos detectados	Ver la tabla 6.2	12
Velocidad del EGO	En m/s	1
Giro volante EGO	En grados	1
Máxima probabilidad	Entre todos los objetos	1
	Total de características	**15**

Cuadro 7.1: Vector de características fase 1

También se ha decidido no utilizar la característica que indicaba el número de objetos detectados. Por último, la variable a predecir es únicamente *Máxima probabilidad*.

Diseño del modelo entrenado

En la Fig. 7.1 se puede ver el diseño del modelo implementado y su configuración.

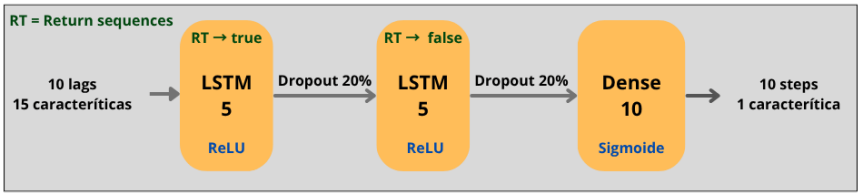

Figura 7.1: Modelo implementado para la fase 1

Como se puede observar, el modelo está compuesto por 2 capas de 5 neuronas LSTM. Cuando se quieren utilizar dos capas LSTM seguidas, es

necesario especificar el *return sequences* a *true* para que transmita cada serie temporal que recibe a la siguiente capa. Como se ha explicado en la sección 6.3.3 se usa una función de activación sigmoide en la última capa ya que se está prediciendo la probabilidad de colisión.

Resultados obtenidos

Para medir el error cometido por los modelos en esta fase se ha aplicado principalmente la métrica MSE y, por otro lado, el MAE. Este error se calcula en cada uno de los pasos predecidos, es decir, para cada uno de los 10 *steps* que en este modelo se predicen. Como el objetivo es predecir las colisiones o, dicho de otra forma, los casos donde la probabilidad sea mayor al menos a un 80 %, se crea otra prueba para medir estos casos. Para esto último, sólo se tienen en cuenta los casos donde el valor real de la probabilidad sea mayor a 80 %.

De esta forma, podemos apreciar los resultados obtenidos en la Fig. 7.2.

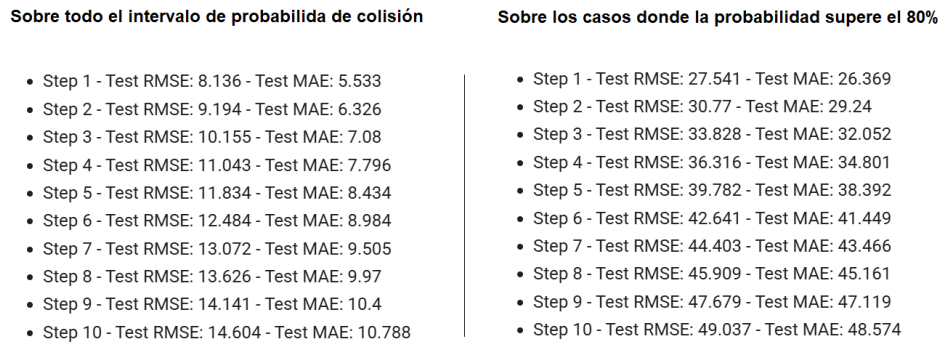

Figura 7.2: Resultados fase 1

Se pueden apreciar principalmente dos comportamientos del modelo.

- En primer lugar, se aprecia claramente que, a medida que se intenta predecir un paso más lejano en el tiempo, el error cometido por el modelo aumenta. Esto parece lógico ya que cada vez es más impredecible el comportamiento del vehículo y del entorno.
- En segundo lugar, el modelo no predice bien los casos donde la probabilidad es muy alta, ya que se puede apreciar que el error crece considerablemente. Teniendo en cuenta que en el *step* 10 el error es del 41 %, se puede corroborar que los resultados se alejan de los objetivos planteados, ya que no se estaría logrando predecir las colisiones.

Para poder comprobar la calidad de los datos, con el fin de entender los resultados, se han realizado una serie de pruebas. En estas pruebas se ha

medido el rendimiento de diferentes modelos, cada uno con una configuración distinta. Se han creado modelos compuestos por una sola capa de una única neurona LSTM hasta tres capas de 50 neuronas LSTM, es decir, desde modelos muy simples hasta más complejos. Del mismo modo, cada modelo se ha entrenado varias veces con distintos conjuntos de datos, en concreto con el 10 %, 30 %, 60 % y el 100 %. Se pueden extraer varias conclusiones a partir de los resultados de esta prueba:

- Si, a medida que se añaden más datos, los resultados del modelo mejoran, quiere decir que se está logrando generalizar mejor. Esto quiere decir que los malos resultados vienen justificados por la falta de datos.
- En cambio, si a medida que un modelo es más complejo mejoran los resultados, querrá decir que los datos no son tan simple como se pensaba y hacen falta modelos más sofisticados para predecir las series.
- Si no se aprecia ningún comportamiento anteriormente mencionado, querrá decir que los datos son de mala calidad y, por lo tanto, aunque logremos generar más datos o intentemos usar modelos más complejos, los resultados posiblemente no mejorarán.

Los resultados obtenidos de esta prueba se pueden apreciar en la Fig 7.3. El error MSE se calcula sobre todos los datos en todos los *steps* calculados.

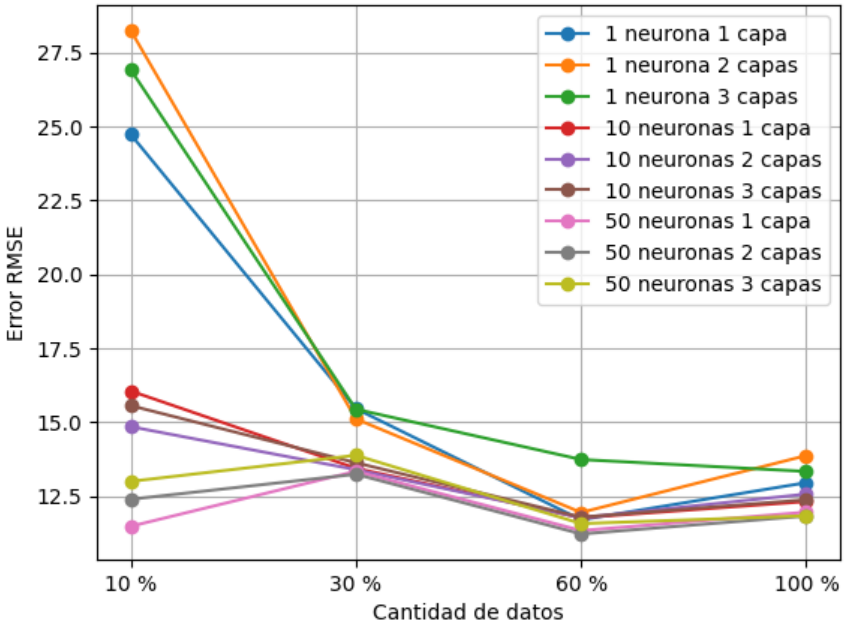

Figura 7.3: Prueba para comprobar la calidad de los datos

Como se puede comprobar, a medida que se aumenta la cantidad de los datos y/o la complejidad de los modelos, no se aprecia una mejora importante a partir de utilizar el 30 % de los datos. Este experimento sirve de prueba para demostrar que los datos, generados durante esta fase, no tienen suficiente calidad y, por lo tanto, es necesario reestructurar el diseño para generar unos datos mejores.

7.2. Fase 2

Continuando la sección 6.1.2, en esta fase se generó un total de 140.000 datos, obtenidos tanto del modo libre del simulador como de *Scenic*. Se realizaron numerosas pruebas, pero finalmente la prueba que obtuvo los mejores resultados fue la que sólo utilizó los datos de los objetos móviles para predecir la probabilidad de colisión.

Características utilizadas

El vector de características quedaría configurado, una vez transformados los datos a series supervisadas, según se muestra en el cuadro 7.3.

Característica	**Breve descripción**	**Nº de datos**
Características EGO	Ver la tabla 7.3	5
Objetos móviles	Ver la tabla 7.4	25
Máx probabilidad Móv	Sólo objetos móviles	1
	Total de características	**31**

Cuadro 7.2: Vector de características fase 2

Tampoco se hacen uso de todas las características calculadas para el vehículo EGO ni del resto de objetos, se decide eliminar algunas que proporcionan la misma información. Por ejemplo, la variable distancia podría inducirse a través de las coordenadas X e Y. Incluirlas podría facilitar al modelo el entrenamiento, pero debido a que hay muchas características se decide eliminar esta redundancia. De este modo, las variables usadas para el vehículo EGO, como para el resto de vehículos, se pueden ver en los cuadros 7.3 y 7.4.

Diseño del modelo entrenado

En la Fig. 7.4 se puede ver el diseño del modelo implementado y su configuración.

En este caso se están usando dos capas LSTM de 20 y 10 neuronas cada una. La última capa es una capa densa de 10 neuronas que generan las diez predicciones del modelo.

Característica	Breve descripción	Nº de datos
Velocidad	En los ejes X e Y, en m/s	2
Giro Volante	En grados	1
En intersección	True o False	1
En intersección futura	True o False	1
	Total	**5**

Cuadro 7.3: Características vehículo EGO fase 2 para la prueba

Característica	Breve descripción	Nº de datos
Máscara	1 o 0	1
Velocidad	En los ejes X e Y, en m/s	2
Posición relativa	En los ejes X e Y, en m	2
	Total	**5**
	Total cinco objetos	**25**

Cuadro 7.4: Características objetos móviles fase 2 para la prueba

Resultados obtenidos

Del mismo modo que en la fase anterior, para medir el rendimiento del modelo se usa tanto el MSE como el MAE. Los resultados obtenidos por este modelo se pueden visualizar en la Fig. 7.5.

Los resultados muestran una leve mejora sobre los obtenidos en la Fase 1, pero hay que tener en cuenta que la cantidad de datos con los que el modelo ha entrenado era muy distinta. En la fase 1, eran alrededor de 30.000 datos, mientras que en la fase 2 han sido cerca de 140.000. De forma que la generalización en esta fase presentaba un desafío más complejo para el modelo. Aún así, los resultados mejoran a los obtenidos en la fase 1.

Sin embargo, como se puede apreciar, sigue teniendo un gran error en los casos donde se supera el 80 % de probabilidad de colisión, por lo que no se puede considerar como un modelo fiable.

7.3. Fase 3

Características utilizadas

Como se explicó en la sección 6.1.3 , ahora el problema se ha planteado desde una perspectiva individual, buscando predecir la colisión con cada uno de los objetos que rodea al vehículo EGO. Esto se realiza con la idea de simplificar el problema y que el modelo pueda generalizar mejor. Del mismo modo, únicamente se trabaja con los objetos móviles detectados. Por último, como ya se explicó, se buscará predecir directamente el indicador o variable colisión generada por el sensor de colisión. De esta forma, la salida

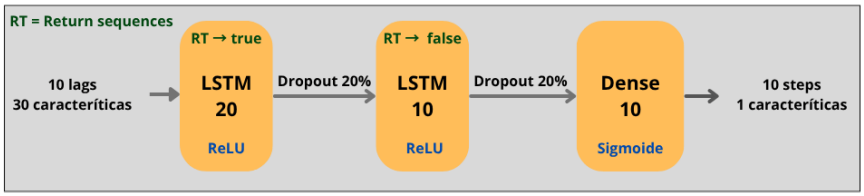

Figura 7.4: Modelo implementado para la fase 2

Sobre todo el intervalo de probabilida de colisión

- Step 1 - Test RMSE: 10.077 - Test MAE: 6.316
- Step 2 - Test RMSE: 9.587 - Test MAE: 6.066
- Step 3 - Test RMSE: 9.33 - Test MAE: 5.918
- Step 4 - Test RMSE: 9.141 - Test MAE: 5.812
- Step 5 - Test RMSE: 9.069 - Test MAE: 5.781
- Step 6 - Test RMSE: 9.121 - Test MAE: 5.826
- Step 7 - Test RMSE: 9.176 - Test MAE: 5.919
- Step 8 - Test RMSE: 9.603 - Test MAE: 6.208
- Step 9 - Test RMSE: 10.17 - Test MAE: 6.578
- Step 10 - Test RMSE: 10.84 - Test MAE: 7.077

Sobre los casos donde la probabilidad supere el 80%

- Step 1 - Test RMSE: 41.939 - Test MAE: 34.17
- Step 2 - Test RMSE: 36.629 - Test MAE: 28.971
- Step 3 - Test RMSE: 36.064 - Test MAE: 28.924
- Step 4 - Test RMSE: 33.884 - Test MAE: 26.684
- Step 5 - Test RMSE: 32.077 - Test MAE: 25.335
- Step 6 - Test RMSE: 31.564 - Test MAE: 24.441
- Step 7 - Test RMSE: 31.533 - Test MAE: 25.184
- Step 8 - Test RMSE: 32.542 - Test MAE: 26.27
- Step 9 - Test RMSE: 36.599 - Test MAE: 30.221
- Step 10 - Test RMSE: 41.508 - Test MAE: 35.736

Figura 7.5: Resultados fase 2

del modelo será una salida continua en un intervalo principalmente entre 0 y 1, ya que el modelo puede generar resultados que estén fuera del intervalo.

A pesar de necesitar únicamente una variable, la colisión, se decide predecir al mismo tiempo dos variables más, la distancia y la velocidad de aproximación del otro vehículo. Esto se ha hecho así por una serie de ventajas. Las tres variables están relacionadas entre sí, de forma que el modelo puede aprender estas representaciones compartidas y mejorar la calidad de cada variable predecida. Del mismo modo, aplica de algún modo una regularización, evitando que se sobreajuste al predecir únicamente una variable, debiendo comprender el comportamiento de las tres variables.

De esta forma, el vector de características quedaría conformado como aparece en el cuadro 7.5, una vez transformados los datos a series supervisadas.

Característica	**Breve descripción**	**Nº de datos**
Características EGO	Ver la tabla 7.3	5
Objeto móvil	Ver la tabla 6.5 [1]	10
Colisión detectada	1 o 0	1
	Total de características	**19**

Cuadro 7.5: Vector de características fase 3

Diseño del modelo entrenado

Como se comentó en la sección 6.3.2, se sigue una estrategia de usar 3 modelos diferentes, cada uno especializado en predecir un intervalo temporal distinto. Todos los modelos reciben como entrada los mismos datos, es decir, los últimos 5 pasos anteriores de la serie temporal, pero predicen 5 valores en diferente momento. A pesar de esto último, el diseño de todos los modelos es el mismo, según se puede ver en la figura 7.6.

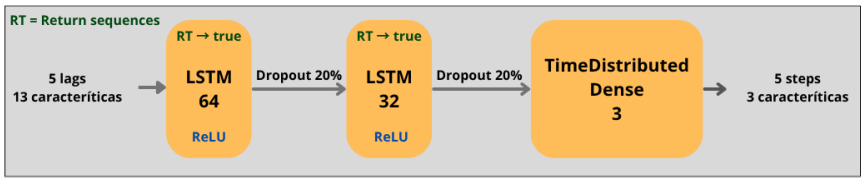

Figura 7.6: Modelo implementado para la fase 3

Se usan dos capas LSTM de 64 y 32 neuronas. Por último, se coloca una capa *TimeDistributed* que se encarga de recibir la serie de datos de la capa LSTM anterior (con *return sequences* a *true*) y generar por cada elemento de la serie una salida. Se ha podido realizar de esta forma ya que el número de *lags* y *steps* coincide. En este caso, la salida es una capa densa de 3 neuronas. De esta forma, la salida de la red son los 5 *steps* para cada una de las 3 características predecidas.

Resultados obtenidos

El modelo predice 3 variables. En las dos primeras, distancia y velocidad de aproximación, se puede medir el error cometido con las métricas de MSE y MAE, pero para la variable que indica si hay colisión o no, se usa indicador para medir el rendimiento. Para esta última variable, se aplica la técnica explicada en la sección 6.3.4, donde se calcula la precisión y el *recall* o exhaustividad.

Primero, se mostrará el error cometido por el modelo para las dos primeras variables. La Fig. 7.7 muestra el error en la velocidad de aproximación.

En la Fig. 7.8, se puede ver el error cometido al predecir la variable de distancia. Debido a que lo importante es predecir correctamente las distancias cortas, se ha creado otra métrica para evaluar únicamente los casos donde el valor real de la variable distancia sea menor a 3 metros.

Como se puede ver el error se mantiene más o menos constante a lo largo de los 15 pasos siguientes. Se puede apreciar un aumento del error, pero no llega a crecer demasiado. Por lo que podemos verificar que el modelo generaliza bien y es capaz de aprender realizar predicciones con buena calidad.

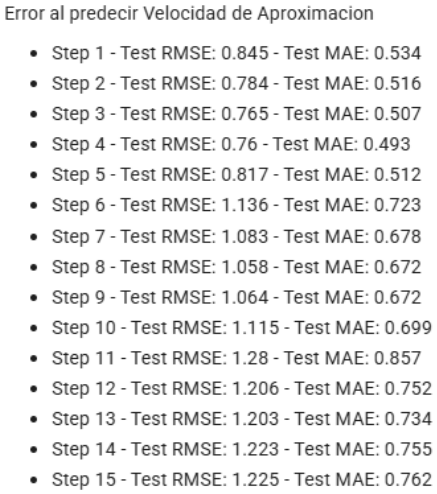

Figura 7.7: Error cometido en la variable velocidad de aproximación, fase 3

Estas dos variables anteriores, como se ha explicado antes, funcionan a modo de apoyo para ayudar al modelo a predecir con más precisión la variable de colisión. Para poder evaluar el rendimiento o el error cometido por el modelo para predecirla, se utiliza la curva precisión-*recall* explicada en la sección 3.3.2.

A continuación se irán mostrando el error cometido por cada uno de los tres modelos. Se mostrará la precisión, el *recall* y la matriz de confusión, en el umbral que maximice estos indicadores.

1. **Modelo *steps 1 a 5*** [$0,5s$ de anticipación]

   - **Umbral** 0.4231
   - **Precisión** 73,18 %
   - **Recall** 73,65 %
   - **Matriz de confusión** Cuadro 7.6

TP 232	FN 83
FP 85	TN 27790

Cuadro 7.6: Matriz de confusión para el modelo de steps 1 a 5

2. **Modelo *steps 6 a 10*** [$1,0s$ de anticipación]

   - **Umbral** 0.2788
   - **Precisión** 61,30 %

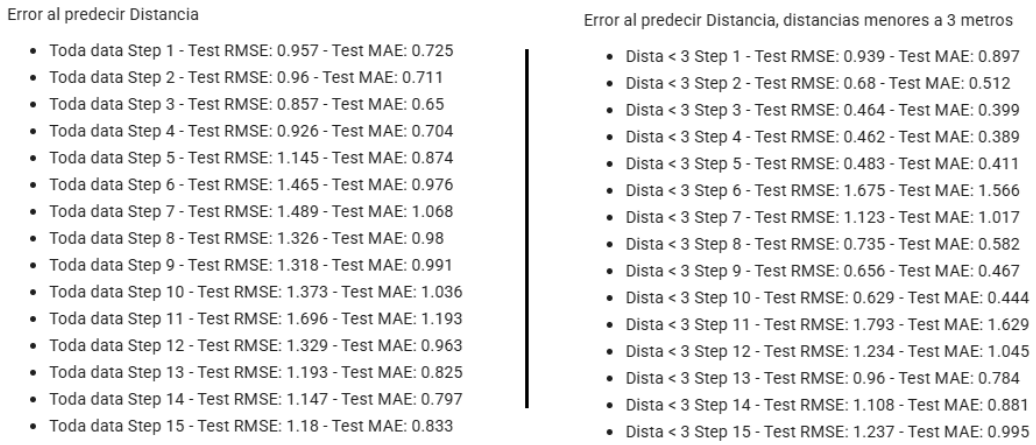

Figura 7.8: Error cometido en la variable distancia, fase 3

TP 225	FN 90
FP 142	TN 25918

Cuadro 7.7: Matriz de confusión para el modelo de steps 6 a 10

- **Recall** 71,42 %
- **Matriz de confusión** Cuadro 7.7

3. **Modelo *steps 11 a 15*** [$1,5s$ de anticipación]

   - **Umbral** 0.1603
   - **Precisión** 40,62 %
   - **Recall** 74,28 %
   - **Matriz de confusión** Cuadro 7.8

TP 234	FN 81
FP 342	TN 23953

Cuadro 7.8: Matriz de confusión para el modelo de steps 11 a 15

En nuestro problema, es importante tener un *recall* alto, es decir, el modelo debe ser capaz de detectar todos los casos positivos. Un falso positivo es, en la mayoría de casos, menos grave y conlleva menos problemas que un falso negativo, por lo que son estos último los que se deben minimizar. En los resultados obtenidos apreciamos que, aunque la precisión disminuya con cada modelo, se mantiene siempre un *recall* alto, por lo que los modelos están detectando bien casi todos los casos positivos, es decir, las colisiones.

Un modelo detectará una colisión, cuando en alguno de los cinco *steps* que predice se supere el umbral calculado. Se podría afinar todavía más el comportamiento del modelo, si para cada paso predecido se calculase su propio umbral, para maximizar el *f1-score*. Posiblemente, se lograría de esta manera aumentar la precisión del modelo, buscando no disminuir el *recall*.

7.3.1. Diseño final

Para ilustrar el diseño final realizado del sistema se ha creado la Fig. 7.9.

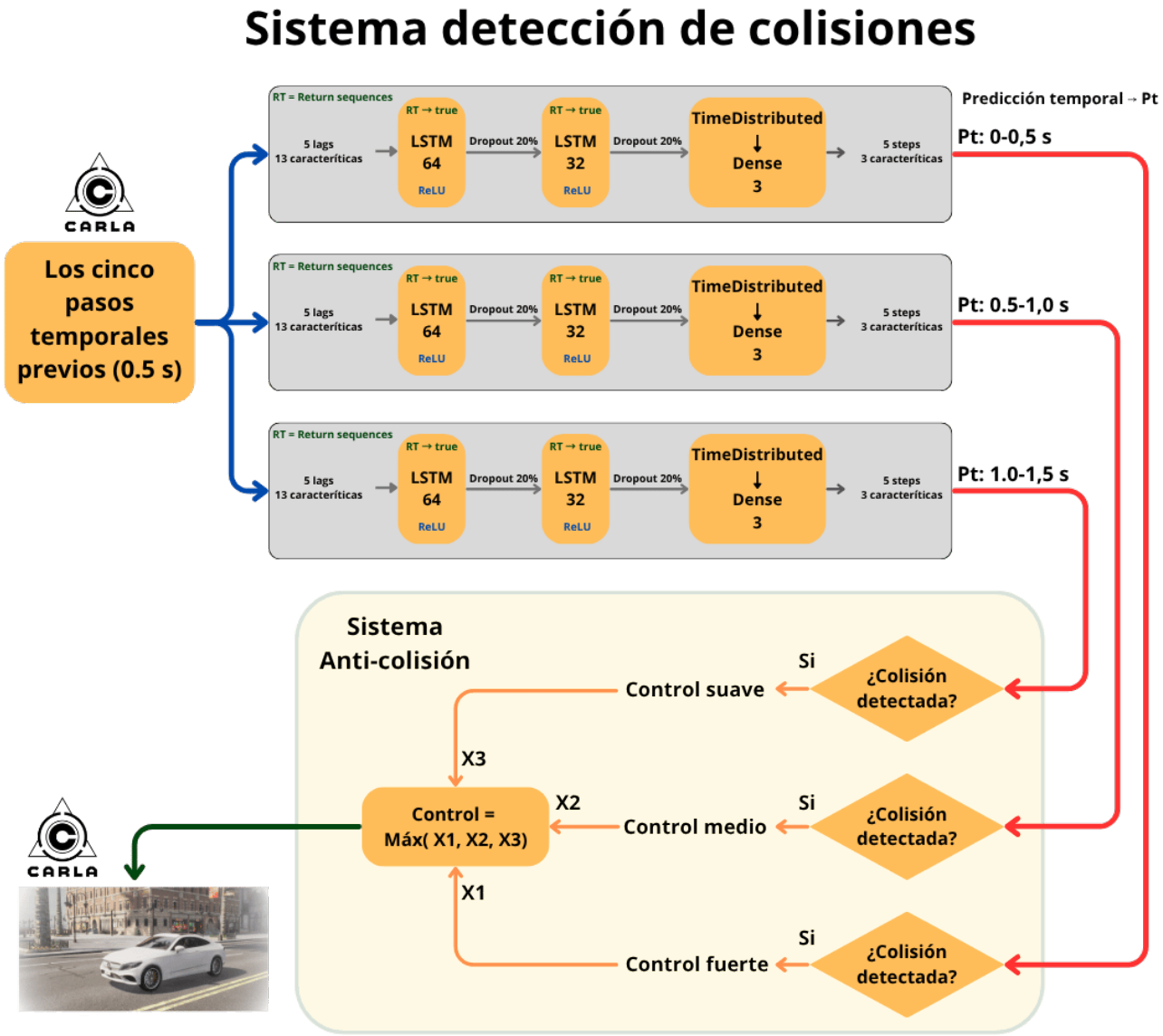

Figura 7.9: Diseño del sistema realizado incluyendo el sistema anti-colisión

7.3.2. Comportamiento del modelo en el simulador

Aunque muchos de los modelos entrenados, se han utilizado en el simulador, se mostrará el comportamiento del último modelo entrenado en la fase 3, con el que se han obtenido los mejores resultados.

Se han creado varias imágenes para poder visualizar el comportamiento de forma más clara, debido a tratarse de un comportamiento dinámico. Los mensajes realizados por el programa se realizan sobre la terminal. A medida que un modelo realice una predicción donde se considere que puede producirse una colisión, se envía por la terminal un mensaje avisando de la posible colisión.

Las figuras creadas capturan los distintos momentos en el que cada modelo predice una colisión, acompañándolo de una imagen de la situación del vehículo y su entorno. Las imágenes creadas son las Figs.7.10, 7.11, 7.12 y 7.13.

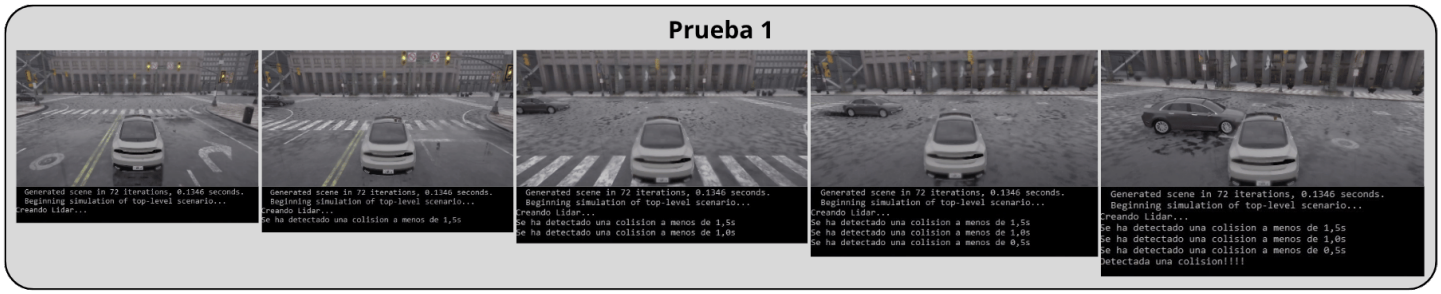

Figura 7.10: Prueba 1 del comportamiento del modelo en el simulador

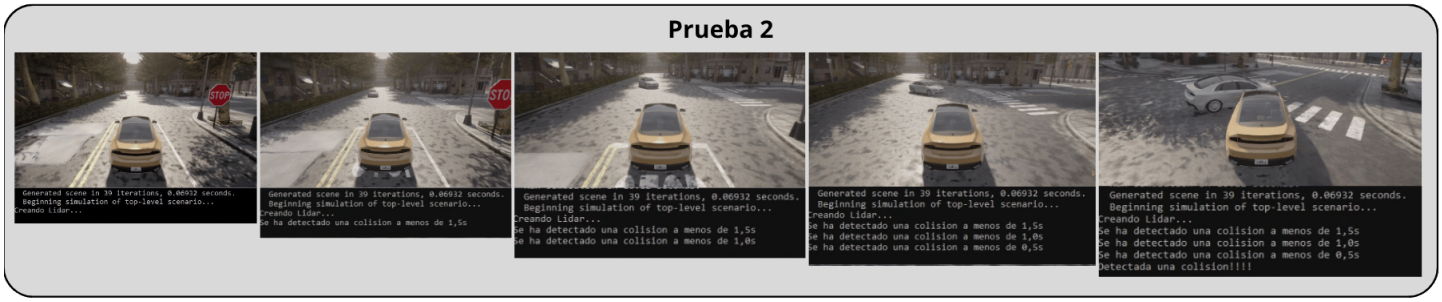

Figura 7.11: Prueba 2 del comportamiento del modelo en el simulador

Tanto en las pruebas 1 como 2 se aprecia un correcto funcionamiento del modelo. Se logra predecir correctamente con los tres modelos la colisión que posteriormente se produce. De este modo, se puede afirmar que el modelo ha sido capaz de anticiparse a la colisión $1,5s$ antes.

En la prueba 3, se ha mostrado un ejemplo donde, a pesar de que se los dos vehículos se han aproximado bastante, ninguno de los tres modelos

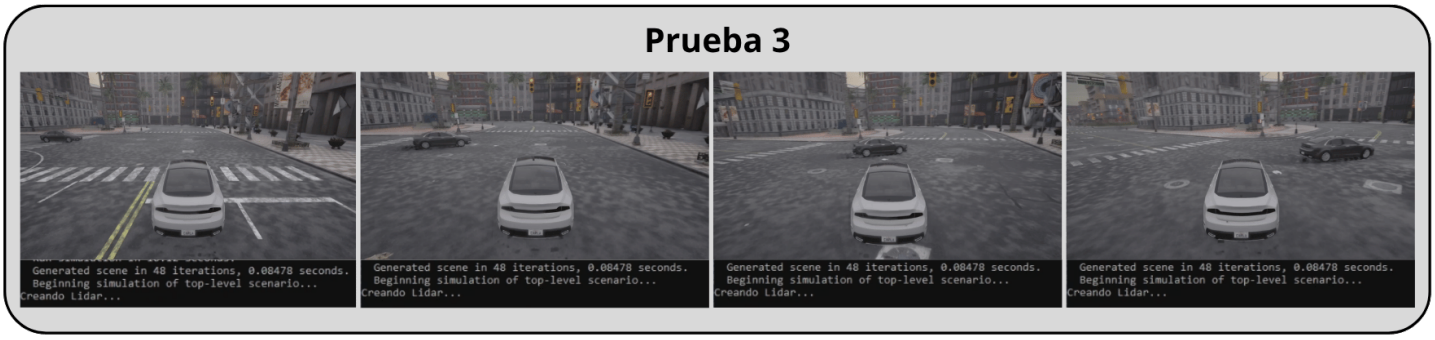

Figura 7.12: Prueba 3 del comportamiento del modelo en el simulador

avisa de la colisión. De esta forma, se puede comprobar que la precisión es bastante buena.

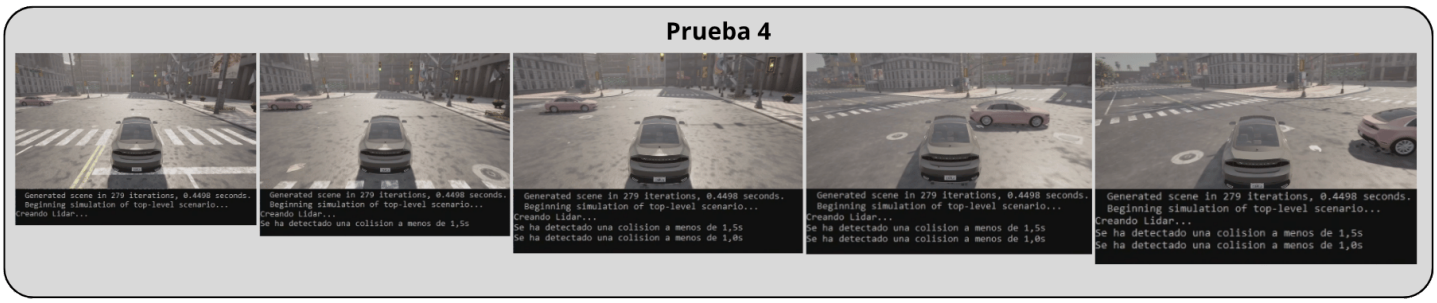

Figura 7.13: Prueba 4 del comportamiento del modelo en el simulador

Por último, en la prueba 4 se ha mostrado un ejemplo donde se ha detectado un falso positivo por parte de dos de los modelos, el de $1,5s$ y el de $1,0s$. En esta prueba, los dos vehículos se aproximan a bastante velocidad y terminan sin colisionar, pero se cruzan a muy poca distancia. Como se ha explicado en los resultados del modelo 3, estos modelos tienen una precisión baja lo que explica que fallen en estas situaciones más complicadas. Sin embargo, el modelo de $0,5s$ no da ningún aviso de colisión, demostrando una alta fiabilidad.

7.4. Sistema anti-colisión

El diseño del sistema anti-colisión es el desarrollado durante la sección 6.4. En la mayoría de pruebas realizadas el sistema es capaz de evitar las colisiones. Para poder visualizarlo mejor, se han creado imágenes que ilustran el comportamiento de este sistema en las Figs. 7.14, 7.15, 7.16 y 7.17.

Se puede apreciar que siempre se logra evitar la colisión. En la prueba 2 se puede ver un ejemplo donde se activan los tres modelos, logrando evitar

el que habría sido un impacto muy fuerte.

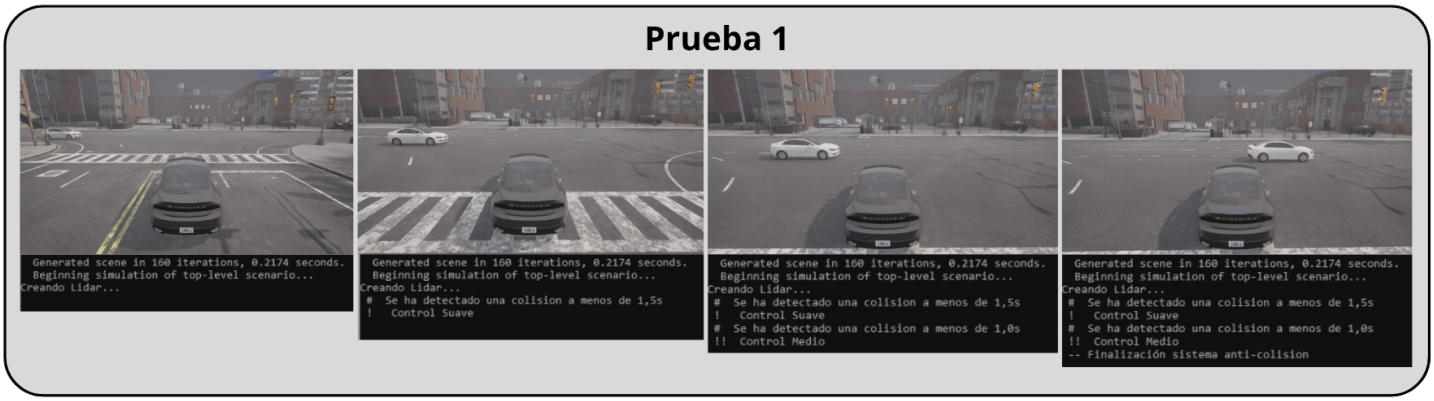

Figura 7.14: Prueba 1 del comportamiento del sistema anti-colisión en el simulador

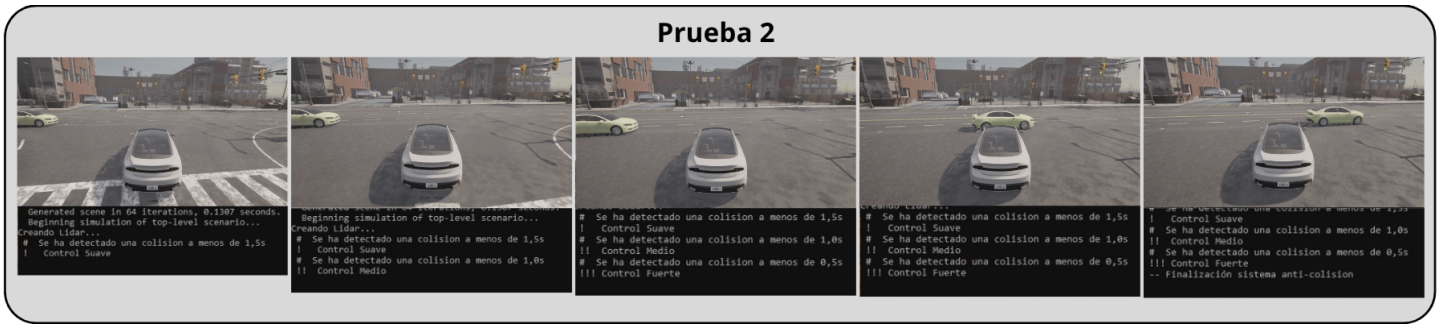

Figura 7.15: Prueba 2 del comportamiento del sistema anti-colisión en el simulador

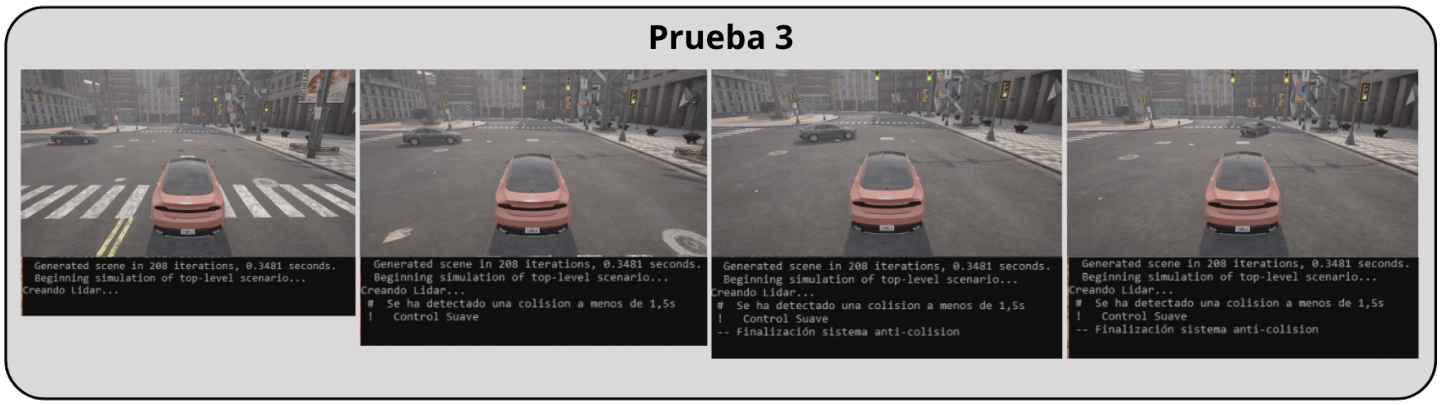

Figura 7.16: Prueba 3 del comportamiento del sistema anti-colisión en el simulador

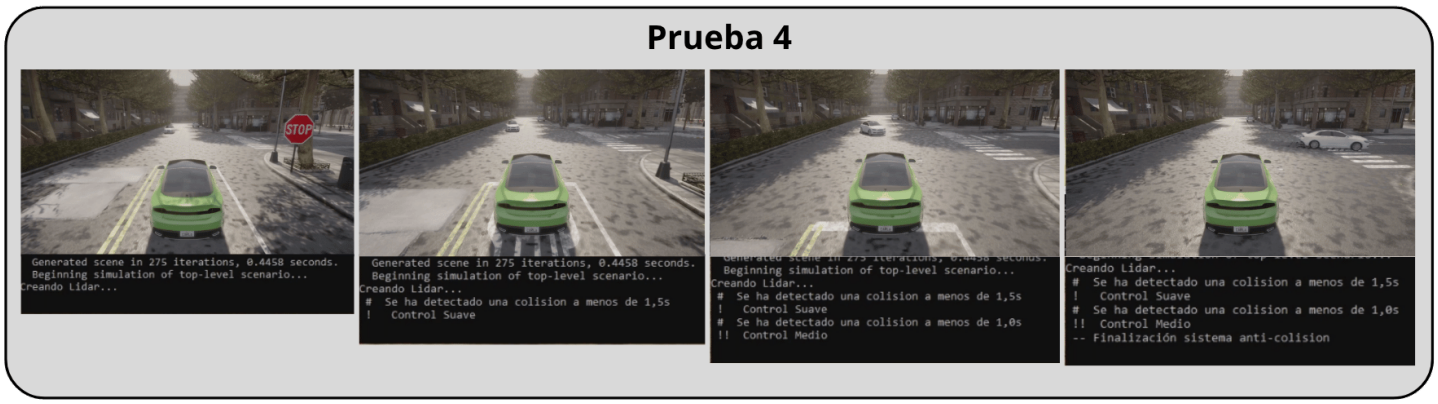

Figura 7.17: Prueba 4 del comportamiento del sistema anti-colisión en el simulador

Capítulo 8

Conclusiones

En este capítulo se describirán las conclusiones extraídas durante y al final del trabajo realizado. Del mismo modo, se plantearán algunas líneas para seguir en un trabajo futuro.

8.1. Conclusiones

En este trabajo se ha intentado predecir las colisiones que se podrían producir entre un vehículo con los objetos que se pueden encontrar en la vía. En concreto, se ha centrado en predecir las colisiones entre vehículos en uno de los puntos más comunes y peligrosos, es decir los cruces. Para ello, se ha utilizado un simulador para generar los datos y, por otro lado, se ha utilizado una red neuronal, con el objetivo de que aprendiera las relaciones entre los datos, logrando predecir colisiones.

Durante la realización del trabajo, he comprendido la complejidad del problema a resolver. Por un lado, la infinidad de posibilidades distintas en las que se puede producir un accidente, desde los diferentes lugares que habría que tener en cuenta hasta las causas o factor que puede provocarlo. Por otro lado, el factor humano que en muchos casos suele ser impredecible, por lo que sería muy difícil evitar el 100 % de las colisiones.

El uso de las redes neuronales han proporcionado una ayuda importante, facilitando el trabajo que supondría haber realizado un algoritmo que aprendiera a predecir las mismas colisiones. Se ha podido comprobar la potencia de estas redes neuronales, en concreto, de las redes LSTM, destacando su capacidad de memorización de las series.

A lo largo del estudio del problema para encontrar la mejor solución, se han experimentado con diferentes sensores y algoritmos, generando en cada caso una información distinta. Esto ha permitido comprobar el rendimiento del modelo con diferentes configuraciones. Se ha confirmado la relación existente entre la complejidad de un problema y el rendimiento del modelo para resolverlo, logrando mejorar los resultados con cada fase realizada, debido a

una mejor estructura de la información que simplificaba el problema.

Finalmente, se ha logrado obtener una solución que es capaz de detectar con una alta tasa las colisiones entre nuestro vehículo frente a otros. Concretamente, los tres modelos que conforman la solución predicen el $73,65\,\%$, el $71,42\,\%$ y el $74,28\,\%$ de las colisiones producidas respectivamente, aproximadamente 3 de cada 4. Debido a la incertidumbre del problema, es lógico que la precisión disminuya a medida que nos alejemos más en el tiempo, de forma que la precisión de cada modelo es de un $73,18\,\%$, $61,30\,\%$ y $40,62\,\%$.

Con la combinación entre los tres modelos con un buen desempeño, junto con el sistema anti-colisión diseñado, se ha logrado evitar la mayoría de las colisiones que se producen en el escenario creado. De esta forma, se ha logrado superar tanto los objetivos iniciales planteados 1.3, como el objetivo extra.

8.2. Trabajo a futuro

Existen muchas variantes posibles para mejorar los resultados de este trabajo. Por un lado, se podría considerar ampliar el número de escenarios de los que generamos los datos, logrando que el modelo funcione mejor en un mayor número de situaciones posibles.

Por otro lado, se podrían probar otros modelos neuronales diferentes u otras propuestas de diseño que aporten otro enfoque distinto al planteado en este trabajo. La posibilidad de soluciones distintas es muy grande.

Por último, sería muy interesante diseñar un sistema anti-colisión más complejo, que pueda realizar un mayor número de acciones de forma más precisa. Un muy buen enfoque podría ser aplicar un aprendizaje por refuerzo, que proporcionaría un buen resultado en un amplio número de escenarios.

Bibliografía

[1] Dirección General de Tráfico. *Las principales cifras de la siniestralidad vial (2023)*. 2024. URL: https://www.dgt.es/menusecundario/dgt-en-cifras/dgt-en-cifras-resultados/dgt-en-cifras-detalle/Las-principales-cifras-de-la-siniestralidad-en-Espana-2023/.

[2] *SAE International*. URL: https://www.sae.org/.

[3] Pejman Goudarzi y Bardia Hassanzadeh. "Collision Risk in Autonomous Vehicles: Classification, Challenges, and Open Research Areas". En: *Vehicles* 6.1 (2024), págs. 157-190. ISSN: 2624-8921. DOI: 10.3390/vehicles6010007. URL: https://www.mdpi.com/2624-8921/6/1/7.

[4] Meryem Hamidaoui et al. "Survey of Autonomous Vehicles' Collision Avoidance Algorithms". En: *Sensors* 25.2 (2025). ISSN: 1424-8220. DOI: 10.3390/s25020395. URL: https://www.mdpi.com/1424-8220/25/2/395.

[5] Jing He et al. "Effective vehicle-to-vehicle positioning method using monocular camera based on VLC". En: *Opt. Express* 28.4 (feb. de 2020), págs. 4433-4443. DOI: 10.1364/OE.382482. URL: https://opg.optica.org/oe/abstract.cfm?URI=oe-28-4-4433.

[6] Muhammad Sohail et al. "Radar sensor based machine learning approach for precise vehicle position estimation". En: *Scientific Reports* 13.1 (2023), pág. 13837.

[7] You Li y Javier Ibanez-Guzman. "Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems". En: *IEEE Signal Processing Magazine* 37.4 (2020), págs. 50-61. DOI: 10.1109/MSP.2020.2973615.

[8] Charith Chitraranjan, Vipooshan Vipulananthan y Thuvarakan Sritharan. "Vision-Based Collision Warning Systems with Deep Learning: A Systematic Review". En: *Journal of Imaging* 11.2 (2025). ISSN: 2313-433X. DOI: 10.3390/jimaging11020064. URL: https://www.mdpi.com/2313-433X/11/2/64.

[9] Alireza Rahimpour et al. *FIR-based Future Trajectory Prediction in Nighttime Autonomous Driving*. 2023. arXiv: 2304.05345 [cs.CV]. URL: https://arxiv.org/abs/2304.05345.

[10] Shadi Saleh et al. "Traffic signs recognition and distance estimation using a monocular camera". En: *6th International Conference Actual Problems of System and Software Engineering*. Vol. 2514. 2019, págs. 407-418.

[11] Wentao Bao, Qi Yu y Yu Kong. "Uncertainty-based Traffic Accident Anticipation with Spatio-Temporal Relational Learning". En: *Proceedings of the 28th ACM International Conference on Multimedia*. MM '20. ACM, oct. de 2020, págs. 2682-2690. DOI: 10.1145/3394171.3413827. URL: http://dx.doi.org/10.1145/3394171.3413827.

[12] Vaishnavi Saraf y Dipti Durgesh Patil. "Advancing Smart Autonomous Systems: Simulation and Testing Platforms". En: *2024 International Conference on IoT Based Control Networks and Intelligent Systems (ICICNIS)*. 2024, págs. 642-649. DOI: 10.1109/ICICNIS64247.2024.10823208.

[13] *Documentación LGSVL*. URL: https://unity-proj.github.io/lgsvl/.

[14] Unity Technologies. *Unity*. Ver. 2023.2.3. Game development platform. 2023. URL: https://unity.com/.

[15] *Documentación Gazebo*. URL: https://gazebosim.org/home.

[16] *Documentación CarSim*. URL: https://www.carsim.com/products/carsim/index.php.

[17] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". En: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, págs. 1-16.

[18] Epic Games. *Unreal Engine*. Ver. 5.4. 10 de abr. de 2025. URL: https://www.unrealengine.com.

[19] IBM. *¿Qué son las redes neuronales?* URL: https://www.ibm.com/es-es/think/topics/neural-networks#:~:text=Una%20red%20neuronal%20es%20un,opciones%20y%20llegar%20a%20conclusiones..

[20] Interactive Chaos. *Estructura de una red neuronal*. URL: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/estructura-de-una-red-neuronal.

[21] Dave Bergmann y Cole Stryker. *¿Qué es la retropropagación?* URL: https://www.ibm.com/es-es/think/topics/backpropagation.

[22] Joaquin Amat Rodrigo y Javier Escobar Ortiz. *skforecast*. Ver. 0.15.0. Mar. de 2025. DOI: 10.5281/zenodo.8382788. URL: https://skforecast.org/.

[23] Diederik P. Kingma y Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[24] Joaquin Amat Rodrigo y Javier Escobar Ortiz. *skforecast*. Ver. 0.15.0. Mar. de 2025. DOI: 10.5281/zenodo.8382788. URL: https://skforecast.org/.

[25] *Documentación de ScenarioRunner*. URL: https://scenario-runner.readthedocs.io/en/latest/.

[26] Edward Kim Daniel J. Fremont Eric Vin. *Documentación de Scenic*. URL: https://docs.scenic-lang.org/en/latest/.

[27] *Google Colab*. URL: https://colab.research.google.com/.

[28] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[29] François Chollet et al. *Keras*. https://keras.io. 2015.

[30] Jason Brownlee. *How to Convert a Time Series to a Supervised Learning Problem in Python*. 2019. URL: https://machinelearningmastery.com/convert-time-series-supervised-learning-problem-python/.

Glosario

Adam *Adaptive Moment Estimation.*

ADAS *Automatic Driver Assistance Systems.*

ADE *Administración y Dirección de Empresas.*

BNNs *Bayesian Neural Networks.*

CARLA *Car Learning to Act.*

CNN *Convolutional Neural Network.*

CSV *Comma Separated Values.*

DGT *Dirección General de Tráfico.*

EGO En latín 'yo'. Vehículo principal en un sistema de conducción autónoma, equipado con sensores y responsable de la percepción del entorno.

FIR *Cámara de Imagen Térmica.*

GCN *Graph Convolutional Networks.*

GNSS *Global Navigation Satellite System.*

GPS *Global Positioning System.*

GRU *Gated Recurrent Unit.*

HDRP *High-Definition Render Pipeline.*

IoT *Internet of Things.*

Lidar *Light Detection and Ranging.*

LSTM *Long Short-Term Memory.*

MAE *Error aboluto medio.*

MSE *Error cuadrático medio.*

Radar *RAdio Detection And Ranging.*

ReLU *Rectified Linear Unit.*

RNN *Recurrent Neural Network.*

ROS *Robot Operating System.*

SAE *Society of Automotive Engineers.*

TTC *Time To Colission.*

V2V *Posicionamiento de Vehículo a Vehículo.*