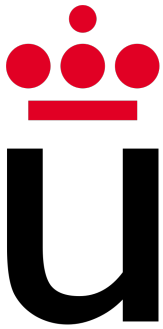


Universidad
Rey Juan Carlos

TRABAJO FIN DE GRADO

# Aprendizaje por Imitación y Mezcla de Redes Especialistas en Conducción Autónoma

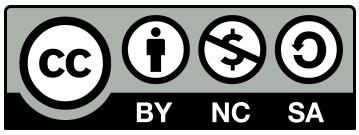
## Grado en Ingeniería Robótica de Software

Escuela de Ingeniería de Fuenlabrada


**Realizado por**
Juan Simó Álvarez

**Dirigido por**
José María Cañas Plaza
Sergio Paniego Blanco


**Curso académico 2023/2024**



# Agradecimientos

Este trabajo no hubiera sido posible sin la ayuda y el apoyo de muchas personas especiales.

En primer lugar, quiero agradecer a mis profesores por su paciencia infinita, especialmente a Roberto, Pedro y Centeno. Sin vosotros, este camino habría tenido muchas menos risas y anécdotas que contar.

Por supuesto a mis tutores de TFG, Jose María y Sergio, por vuestra tutela, la increíble ayuda técnica y vuestra persistencia para llegar hasta aquí a pesar de las dificultades.

A mi familia, que siempre estuvo ahí para recordarme que hay que cerrar cada etapa para pasar bien a la siguiente. Gracias por vuestro amor incondicional y por darme los pequeños empujones que necesitaba para no acabar de detenerme nunca, "lento pero sin pausa".

A mis amigos, por los incontables momentos de distracción necesarios para mantener mi cordura. Gracias por hacerme reír, por las cenas compartidas y por estar siempre dispuestos a escuchar mis quejas sobre la vida académica (gracias Juan).

A Juanjo Molina, quién sin saberlo me ha dado uno de los mejores años deportivos, y me ha permitido mantenerme sano pese a las infinitas horas frente al ordenador.

Y no podía faltar, a Óscar, la persona que más me ha acompañado, escuchado, ayudado y motivado. Con quién a lo largo de estos años he compartido momentos duros, pero sobre todo éxitos. Porque nunca te paras y me obligas a seguirte, porque ver tu ilusión y motivación contagia a cualquiera que te rodee, porque sin ti, no estaría aquí.

A todos vosotros, mil gracias.

***"Life before death.***
***Strength before weakness.***
***Journey before destination."***
— Brandon Sanderson

# Resumen

Este Trabajo de Fin de Grado (TFG) explora el desarrollo y la implementación de un sistema de control para un coche de carreras autónomo basado en redes neuronales. El objetivo principal es comparar dos enfoques: un piloto monolítico y una estrategia de mezcla de especialistas.

El piloto monolítico utiliza una única red neuronal entrenada de extremo a extremo para controlar el coche. La red procesa las imágenes capturadas por la cámara del coche y genera comandos de velocidad. Se describen los detalles del procesamiento de imágenes, la arquitectura de la red neuronal y la implementación del sistema de control en ROS2.

Por otro lado, la estrategia de mezcla de especialistas divide la tarea de control en dos redes especializadas: una para rectas y otra para curvas. Cada red se entrena con datos específicos, mejorando así su rendimiento en su respectiva tarea. La clasificación de las imágenes entrantes determina qué red se utiliza en cada momento.

Se realiza una comparación detallada de ambos enfoques, evaluando su rendimiento en varios circuito de carreras. Los resultados muestran que la estrategia de mezcla de especialistas ofrece mejoras significativas en términos de estabilidad y eficiencia.

**Palabras clave:** robótica, inteligencia artificial, ROS2, conducción autónoma, Gazebo, redes neuronales.

# Acrónimos

**TFG** - Trabajo de Fin de Grado
**ROS** - Robot Operating System
**ANN** - Artificial Neural Network
**CNN** - Convolutional Neural Network
**PLC** - Programmable Logic Controllers
**CPU** - Central Processing Units
**GPU** - Graphics Processing Units
**IA** - Inteligencia Artificial
**LIDAR** - Light Detection and Ranging
**DARPA** - Defense Advanced Research Projects Agency
**CARLA** - Car Learning to Act
**DART** - Dynamic Architecture for Robotics Toolkit
**RGB** - Red Green Blue
**PID** - Proporcional Integral y Derivativo
**CSV** - Comma-Separated Value
**RTF** - Real Time Factor

# Índice general

# Índice de figuras

# 1. Introducción

La conducción autónoma es una tecnología innovadora que tiene el potencial de transformar el transporte y la movilidad. Este campo interdisciplinario integra la robótica, la visión artificial, la inteligencia artificial y las redes neuronales, entre otros, combinando avances en estas áreas para crear vehículos capaces de navegar de manera segura y eficiente sin intervención humana.

Este capítulo ofrece una visión general de los conceptos clave y las tecnologías subyacentes que respaldan la conducción autónoma. A través de él, se establece el contexto necesario para comprender el contenido de este trabajo.

## 1.1. Robótica

La robótica es una rama interdisciplinaria de la ingeniería y la ciencia que incluye campos como la mecánica, la electrónica, la informática y la inteligencia artificial. Su objetivo principal es diseñar, construir, operar y aplicar robots en una amplia variedad de sectores, como la industria, la medicina, la agricultura o la exploración espacial. Los robots son sistemas automáticos que pueden realizar tareas complejas de manera autónoma o semiautónoma. La robótica ha evolucionado significativamente con el avance de la tecnología, especialmente en las últimas décadas.

Aunque la historia de la robótica tiene sus raíces en la antigüedad, su desarrollo significativo comenzó en el siglo XX. Las primeras aplicaciones se centraron principalmente en la industria, destacándose el primer robot comercial, Unimate, utilizado en las cadenas de montaje. Desde entonces, la robótica ha experimentado un crecimiento exponencial, extendiendo sus aplicaciones a diversos sectores.

Los componentes esenciales de los robots son:

- **Actuadores:** Dispositivos que convierten señales de control en acciones físicas. Pueden generar movimiento (como motores), luz (como LEDs), sonido (como altavoces), o cualquier otra forma de interacción con el entorno físico.
- **Sensores:** Dispositivos que permiten al robot percibir información, ya sea del

entorno o su propia situación. Los sensores incluyen cámaras, sensores de proximidad, acelerómetros y giroscopios.

- **Controladores:** Unidades que procesan la información de los sensores y generan comandos para los actuadores. Pueden ser microcontroladores, PLCs, CPUs o GPUs.

Sobre estos componentes, se ejecutan programas que definen el comportamiento del robot, incluyendo algoritmos de control, sistemas de navegación y módulos de inteligencia artificial.

La programación de robots puede ser compleja debido a los algoritmos avanzados y el procesamiento de información sensorial requerido. Además, la falta de estandarización en el *middleware* y los lenguajes de programación utilizados por diferentes fabricantes dificultaba la reutilización de código. Para abordar estos desafíos, se desarrollaron soluciones como ROS [5], un *middleware* que soporta lenguajes como Python y C++, y que se ha convertido en el estándar de facto en robótica de servicios.

Antes de la industrialización, los robots requieren extensas pruebas, que pueden ser costosas y peligrosas debido al alto costo del *hardware* y la dificultad de recrear entornos específicos. Para mitigar estos problemas, se utilizan simuladores como Webots [6] y Gazebo [7] para diseñar y probar robots en entornos virtuales. Para tareas especializadas como la conducción autónoma, existen simuladores avanzados como CARLA [8], que ofrecen vehículos y escenarios fotorrealistas.

Los robots pueden clasificarse en varias categorías según su aplicación y diseño:

- **Robots Industriales:** Utilizados en la manufactura para tareas como soldadura, ensamblaje y pintura.
- **Robots de Servicio:** Realizan tareas útiles para humanos o equipos, excluyendo aplicaciones de automatización industrial.

Dentro de los robots de servicio, encontramos aplicaciones en:

- **Limpieza:** Robots diseñados para la limpieza de hogares y espacios comerciales, como las aspiradoras robóticas y los limpiadores de ventanas automatizados (ver Figura 1.1).

**Figura 1.1:** Robot de limpieza Rumba de iRobot [1].

- **Salud:** Robots que asisten en procedimientos médicos, proporcionan rehabilitación, ayudan en la cirugía, y ofrecen compañía y asistencia a pacientes (ver Figura 1.2).

**Figura 1.2:** Robot quirúrgico Da Vincci de Intuitive Surgical [2].

- **Logística:** Robots que automatizan la gestión de almacenes, el transporte de mercancías, y la preparación de pedidos en centros de distribución (ver Figura 1.3).

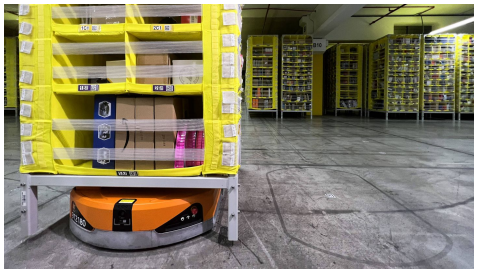


**Figura 1.3:** Robots empleados en los almacenes de Amazon.

- **Conducción autónoma:** Vehículos que navegan de manera autónoma en diferentes entornos, mejorando la seguridad y la eficiencia en el transporte (ver Figura 1.4).

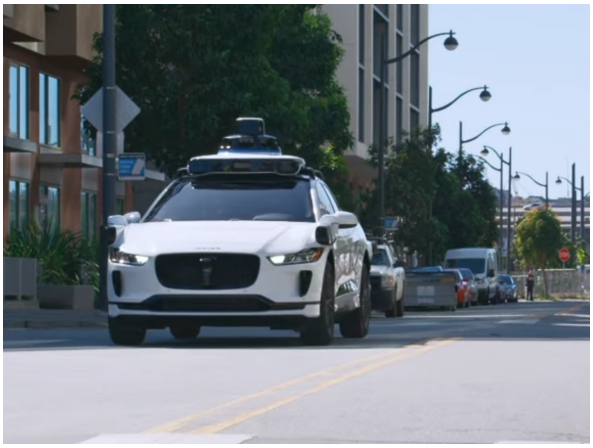


**Figura 1.4:** Coche de Waymo circulando [3].

## 1.2. Visión artificial

En la robótica moderna, las cámaras han emergido como el sensor principal debido a la abundancia de información que pueden proporcionar y a su bajo costo en comparación con otros sensores como los LIDAR. No obstante, estas cámaras también presentan desafíos significativos, ya que gran parte de la información que capturan puede ser irrelevante para ciertas aplicaciones, como en la conducción autónoma, donde los colores de los edificios o del cielo no son críticos. La visión artificial ha ganado relevancia en este contexto al maximizar la utilidad de las cámaras como sensores principales.

La visión artificial, también conocida como visión por computadora, es un campo de la informática que se centra en permitir a las máquinas interpretar y comprender el contenido visual del mundo. Utiliza cámaras, algoritmos de procesamiento de imágenes y modelos de aprendizaje automático para analizar imágenes y videos con el objetivo de extraer información útil y tomar decisiones basadas en dicha información.

La visión artificial comenzó a tomar forma a mediados del siglo XX con los primeros intentos de procesamiento de imágenes digitales. A lo largo de los años, ha evolucionado significativamente gracias a los avances en *hardware*, algoritmos y técnicas de aprendizaje automático. A finales del siglo XX, se desarrollaron los primeros sistemas comerciales de visión artificial para aplicaciones industriales. En la actualidad, la visión artificial se utiliza en una amplia variedad de aplicaciones, desde el reconocimiento facial hasta la conducción autónoma.

Los sistemas de visión artificial constan de varios componentes y utilizan diversas técnicas para procesar y analizar imágenes. Algunos de los componentes y técnicas más comunes incluyen:

- **Adquisición de Imágenes:** Uso de cámaras y sensores para capturar fotogramas.
- **Procesamiento de Imágenes:** Técnicas para mejorar la calidad de las imágenes y preparar los datos para su análisis, como filtrado, segmentación y detección de bordes.
- **Reconocimiento de Patrones:** Algoritmos para identificar y clasificar objetos dentro de las imágenes.
- **Aprendizaje Automático:** Modelos de aprendizaje supervisado y no supervisado que permiten a las máquinas aprender de los datos y mejorar su precisión con el tiempo.

La visión artificial tiene aplicaciones en numerosos campos. En la industria, se utiliza para la inspección de calidad y control de procesos. En la medicina, ayuda en el diagnóstico y seguimiento de enfermedades mediante el análisis de imágenes médicas. En la seguridad, se emplea para la vigilancia y el reconocimiento facial. Además, es una tecnología clave en el desarrollo de vehículos autónomos, donde permite a los vehículos percibir y comprender su entorno para tomar decisiones de conducción.

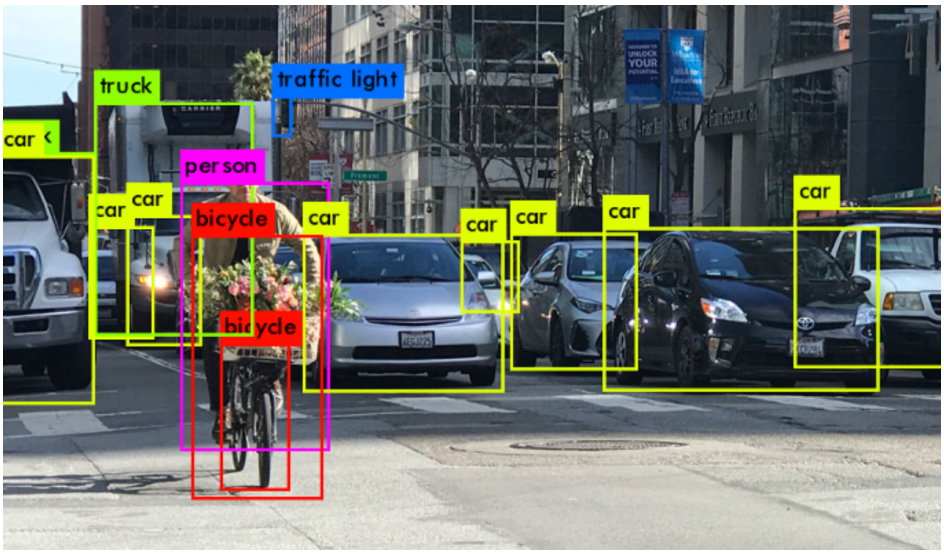



**Figura 1.5:** Algoritmo YOLO detectando y clasificando objetos.

## 1.3. Conducción autónoma

La conducción autónoma es una tecnología emergente que permite a los vehículos operar sin intervención humana. Se basa en la integración de diversas tecnologías, incluyendo la inteligencia artificial, la visión por computadora, la robótica y los sistemas de comunicación, para percibir el entorno, planificar rutas y tomar decisiones en tiempo real. El principal motivo que impulsa el desarrollo de esta tecnología es reducir el número de accidentes que ocurren en carretera que, según un estudio realizado entre 2005 y 2007 [9], se deben en más del 90 % de los casos al error humano.

La idea de vehículos autónomos ha existido desde mediados del siglo XX, pero los avances significativos comenzaron en la década de 2000 con proyectos como el DARPA Grand Challenge, que incentivaron el desarrollo de tecnologías para vehículos sin conductor. Desde entonces, empresas y universidades de todo el mundo han trabajado en prototipos y sistemas avanzados de conducción autónoma, llevando a la creación de vehículos capaces de operar de manera segura en diversas condiciones de tráfico y ambientales.

Los sistemas de conducción autónoma constan de varios componentes y tecnologías clave:

- **Sensores:** Cámaras, radares, LIDAR y sensores ultrasónicos que proporcionan una visión completa del entorno del vehículo.

- **Percepción:** Algoritmos de visión artificial y procesamiento de señales que interpretan los datos de los sensores para identificar objetos, peatones y otros vehículos.
- **Localización:** Sistemas de GPS y mapas digitales de alta definición que permiten al vehículo conocer su ubicación precisa y planificar rutas.
- **Planificación y Control:** Algoritmos de planificación de rutas y control del vehículo que determinan la mejor manera de llegar a un destino evitando obstáculos y cumpliendo con las normas de tráfico.

La conducción autónoma se clasifica en cinco niveles según la Sociedad de Ingenieros Automotrices (SAE):

- **Nivel 0:** Sin automatización. El conductor controla todas las funciones del vehículo.
- **Nivel 1:** Asistencia al conductor. Sistemas como el control de crucero adaptativo asisten en algunas funciones.
- **Nivel 2:** Automatización parcial. El vehículo puede controlar la dirección y la velocidad, pero el conductor debe supervisar y estar preparado para intervenir.
- **Nivel 3:** Automatización condicional. El vehículo puede manejar todas las funciones de conducción en ciertas condiciones, pero el conductor debe estar disponible para tomar el control cuando se le solicite.
- **Nivel 4:** Alta automatización. El vehículo puede operar de manera autónoma en la mayoría de las situaciones, sin necesidad de intervención del conductor.
- **Nivel 5:** Automatización completa. El vehículo es totalmente autónomo en todas las condiciones y no requiere conductor.

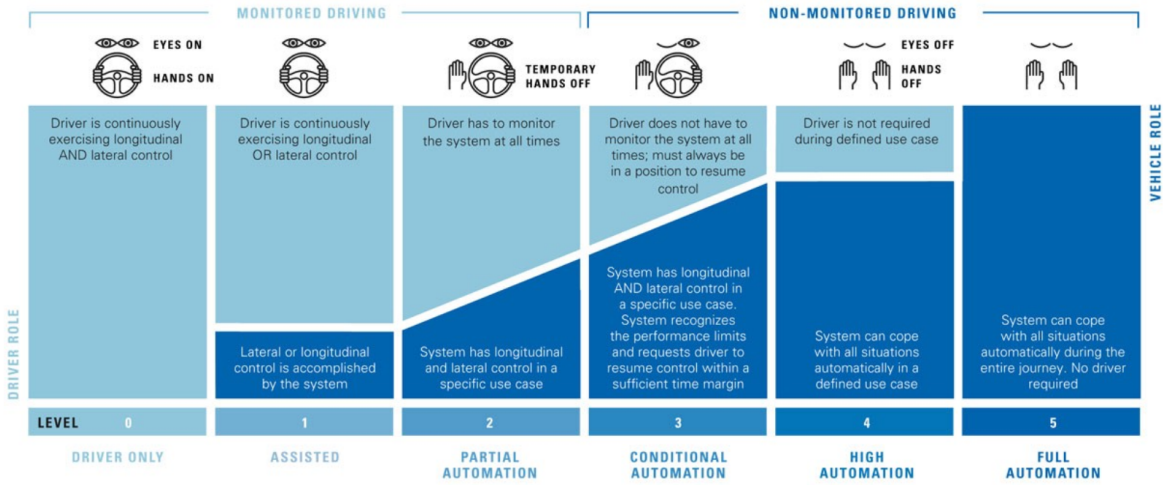



**Figura 1.6:** Niveles de autonomía en conducción.

La conducción autónoma promete revolucionar el transporte, mejorando la seguridad, reduciendo el tráfico y aumentando la accesibilidad. Entre sus aplicaciones destacan:

- **Transporte Público:** Autobuses y taxis autónomos que pueden operar en rutas fijas y bajo demanda.
- **Logística y Entrega:** Vehículos autónomos que pueden realizar entregas de manera eficiente y sin intervención humana.
- **Vehículos Privados:** Coches personales que pueden conducir de manera autónoma, permitiendo a los propietarios centrarse en otras actividades durante el viaje.

Se han identificado dos enfoques principales para el desarrollo de sistemas de conducción autónoma: los enfoques de extremo a extremo y los enfoques modulares.

El enfoque modular combina varios módulos especializados, cada uno específico para actividades particulares del proceso de conducción, como percepción, planificación y control. Estos módulos se comunican entre sí y su combinación permite la conducción del vehículo. Este enfoque es muy popular debido a su flexibilidad. Sin embargo, presenta ciertos inconvenientes importantes: los errores en un módulo pueden ser difíciles de rastrear y tienen el potencial de propagarse al resto del sistema. Además, la coordinación entre múltiples módulos incrementa significativamente la complejidad del sistema global, lo que puede dificultar su mantenimiento.

Los sistemas de extremo a extremo en conducción autónoma se refieren a arquitecturas en las que un único modelo, típicamente una red neuronal profunda, toma las entradas sensoriales (como imágenes de cámaras) y devuelve las acciones de control del vehículo (como la dirección, aceleración y frenado). Este enfoque contrasta con los sistemas tradicionales que dividen el proceso de conducción en múltiples etapas (percepción, planificación, control), cada una con sus propios algoritmos especializados. Los sistemas de extremo a extremo simplifican el *pipeline* al integrar todas estas etapas en un solo modelo entrenado de manera conjunta.

En el CARLA Leaderboard 2.0[1], los modelos *end-to-end* han logrado puntuaciones destacadas. Por ejemplo, el modelo ReasonNet [10] ha alcanzado el primer lugar en la categoría de sensores, destacándose por su capacidad para razonar temporal y globalmente sobre el entorno de conducción.

La conducción autónoma basada en visión de extremo a extremo ha ganado considerable atención en la literatura reciente, aunque aún presenta algunas limitaciones [11], especialmente su falta de interpretabilidad. A pesar de ello, los avances recientes han sido significativos. Por ejemplo, algunos enfoques combinan grandes modelos neuronales para la conducción [12] y emplean información exteroceptiva sobre el entorno del vehículo [13] [14].

Un ejemplo destacado de un modelo de extremo a extremo es PilotNet [15], propuesto por investigadores de Nvidia. Este modelo emplea una CNN para extraer características y generar comandos de control a partir de imágenes de entrada. Se ha mejorado agregando capas completamente conectadas para alcanzar robustez y conducción razonables [14].

## 1.4. Aprendizaje automático

Tradicionalmente, la programación de robots se ha realizado de manera explícita, describiendo detalladamente el comportamiento del *hardware* mediante código o circuitos. Esto implica que un robot solo puede realizar tareas para las cuales ha sido específicamente programado. Sin embargo, cuanto más general se desea que sea un robot, más difícil resulta programarlo explícitamente. Por ejemplo, diferenciar entre tareas como fregar o cocinar puede ser complejo, ya que, aunque la información sensorial pueda ser similar, las partes relevantes de esta información y las acciones necesarias son distintas. Esto aumenta

[1] https://leaderboard.carla.org

exponencialmente la complejidad de la programación.

Este desafío contribuyó al desarrollo de la inteligencia artificial (IA), que, con el avance del aprendizaje automático, encontró en la robótica un área de aplicación cada vez más relevante con el avance de las técnicas de aprendizaje automático.

La inteligencia artificial es una rama de la informática que busca crear sistemas capaces de realizar tareas imitando la inteligencia humana, como el reconocimiento de voz, la toma de decisiones y la traducción de idiomas. La IA tiene sus raíces en la década de 1950, cuando los primeros investigadores comenzaron a desarrollar algoritmos para simular el pensamiento humano. Desde entonces, la IA ha evolucionado significativamente, con hitos importantes como el desarrollo de sistemas expertos en la década de 1970, el auge de los algoritmos de *machine learning* en los años 2000, y los recientes avances en *deep learning* y redes neuronales profundas.

El aprendizaje automático es una rama de la inteligencia artificial que permite a los sistemas aprender de manera autónoma, sin necesidad de programación explícita. Engloba diversas técnicas, entre ellas el aprendizaje supervisado (como el aprendizaje neuronal profundo), el aprendizaje no supervisado (como la búsqueda de patrones o la segmentación), y el aprendizaje por refuerzo, donde los agentes aprenden a través de la interacción con su entorno y la retroalimentación que reciben. Las redes neuronales artificiales (ANNs) están inspiradas en la estructura y funcionamiento del cerebro humano y están compuestas por capas de neuronas artificiales, también llamadas nodos, que están interconectadas. Las ANNs se pueden clasificar en varios tipos según su arquitectura y el tipo de problemas que abordan:

- **Redes Neuronales *Feedforward*:** Las neuronas están organizadas en capas y la información fluye en una sola dirección, desde la entrada hasta la salida.
- **Redes Neuronales Recurrentes:** Incluyen conexiones que forman ciclos, permitiendo que la información persista en el tiempo, lo que las hace adecuadas para el procesamiento de secuencias.
- **Redes Neuronales Convolucionales:** Utilizadas principalmente para el procesamiento de datos estructurados en forma de imágenes, utilizan operaciones de convolución para detectar características locales.
- ***Transformers*:** Una arquitectura moderna basada en el mecanismo de atención, que permite capturar relaciones globales en una secuencia sin recurrencia.

Estas redes neuronales requieren una gran cantidad de datos para ajustar sus

parámetros y aprender a resolver los problemas ejemplificados en los datos. En el ámbito de la robótica, se aplican en tareas perceptivas como la detección de objetos, la segmentación semántica o la estimación de profundidad, donde la red interpreta la información sensorial del entorno. Además, también se emplean en enfoques extremo a extremo, donde el modelo no solo procesa las percepciones, sino que toma decisiones completas que llegan hasta los actuadores del robot. Dentro de este último enfoque se encuentra el aprendizaje por imitación, técnica utilizada en este TFG, en la que el sistema aprende a replicar el comportamiento de un experto humano a partir de demostraciones.

El aprendizaje por imitación, o *imitation learning*, consiste en entrenar al modelo para que imite el comportamiento de un experto. Una práctica común para lograr esto se denomina clonación del comportamiento o *behavioral cloning* [16]. Esto se logra mediante la recopilación de datos de conducción (humana o de programación explícita), que luego se utilizan para entrenar la red neuronal. El objetivo es que el modelo aprenda a replicar las decisiones de un conductor experto a partir de las mismas entradas sensoriales.

El aprendizaje por imitación ha demostrado ser efectivo en la conducción autónoma y otros dominios, como los agentes de juegos. Por ejemplo, en la investigación de [17], se combina con éxito el *behavioral cloning* con el manejo de restricciones difíciles para escenarios no vistos en los datos recopilados y que pueden llevar a fallos críticos de seguridad.

Esta técnica también es aplicable a otros dominios fuera de la conducción autónoma, como los agentes de juegos, donde el modelo se entrena a partir de datos del agente experto de la misma manera [18] [19] [14].

Sin embargo, generar un modelo exitoso en esta técnica requiere un conjunto de datos amplio y equilibrado. En experiencias normales de conducción, la mayoría de los casos registrados pueden no ser relevantes, llevando a conjuntos de datos desequilibrados. Esto compromete el rendimiento del vehículo en situaciones nuevas o complejas, como se ilustra en la Figura 1.7.

Para abordar estas limitaciones, se emplean enfoques como el sobremuestreo de casos extremos y técnicas de aumento de datos, comunes en *machine learning* y soportadas por marcos de *deep learning* como *Albumentations* [20]. Estas estrategias ayudan a mejorar la capacidad de generalización del modelo y a manejar situaciones inesperadas con mayor eficacia.

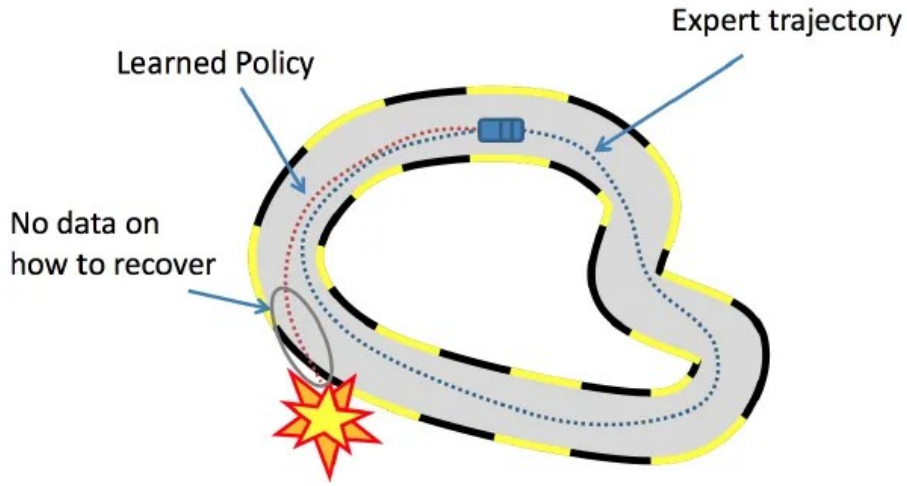



**Figura 1.7:** El aprendizaje por imitación falla al encontrar situaciones nuevas.

Además del aprendizaje por imitación, las técnicas de *reinforcement learning* también juegan un papel crucial en la conducción autónoma. Estas permiten al vehículo interactuar con el entorno, aprendiendo a través de exploración y explotación basada en una función de recompensa. Integrar estos enfoques en sistemas híbridos es fundamental para desarrollar sistemas autónomos capaces de enfrentar tareas complejas de manera efectiva [21] [14].

# 2. Objetivos y metodología

Tras la introducción y explicación del contexto de este trabajo, se detallarán los objetivos, la metodología aplicada y el plan de trabajo implementado.

## 2.1. Objetivos

El objetivo principal de este trabajo es programar, analizar y comparar dos enfoques distintos para el control de vehículos autónomos en un entorno simulado. Ambos sistemas utilizan redes neuronales profundas entrenadas con el mismo conjunto de datos supervisado, y están diseñados para seguir una línea en un circuito utilizando únicamente la información visual proporcionada por una cámara. El primer enfoque emplea una red neuronal monolítica, mientras que el segundo se basa en la coordinación de dos redes especializadas: una para curvas y otra para rectas. La evaluación de ambos modelos se realizará en varios circuitos dentro del simulador Gazebo, tanto en trayectorias incluidas en los datos de entrenamiento como en nuevas, no vistas previamente, con el objetivo de estudiar su capacidad de generalización.

Para alcanzar este objetivo principal, se han definido varios subobjetivos:

1. Desarrollar un piloto experto y grabar un *dataset* supervisado.
2. Generar un piloto entrenando una red neuronal monolítica.
3. Generar un piloto que combine dos redes neuronales expertas (una en rectas y otra en curvas).
4. Validar y comparar experimentalmente los pilotos generados.

## 2.2. Metodología

El desarrollo de este trabajo se ha llevado a cabo mediante *sprints* de Scrum, una metodología ágil que ha permitido un enfoque sistemático y flexible para abordar los diversos desafíos del proyecto. Cada semana se han seguido los siguientes pasos:

1. Establecimiento de los objetivos semanales en colaboración con mis tutores. Esta fase inicial ha sido importante para asegurar que los objetivos estén alineados con el objetivo principal del trabajo y sean realistas y alcanzables dentro del plazo disponible.
2. Planificación del trabajo y definición de los pasos necesarios para alcanzar los objetivos establecidos. Esta planificación ha incluido la identificación de posibles obstáculos y las estrategias para superarlos.
3. Implementación del código y realización de las pruebas correspondientes. Durante esta fase, se ha desarrollado el software necesario y se han llevado a cabo pruebas para garantizar que todo funcione correctamente.
4. Actualización de un blog[1] con el propósito de documentar los avances realizados cada semana.
5. Reunión semanal con mis tutores para revisar los objetivos alcanzados, discutir los problemas encontrados y establecer los objetivos para la siguiente iteración.

Gracias a esta metodología iterativa, ha sido posible descomponer cada problema en subobjetivos manejables y abordar cada uno de manera sistemática. Este enfoque ha permitido un avance constante y eficiente hacia el logro del objetivo principal del trabajo, asegurando un desarrollo ágil.

## 2.3. Plan de trabajo

Este TFG, desarrollado a lo largo del curso académico 2023/2024 y 2024/2025, se ha dividido en distintas etapas:

1. **Estudio y familiarización:** En esta etapa inicial, se realizó un estudio profundo de la API de redes neuronales de PyTorch y de trabajos previos en conducción autónoma que sirvieron como punto de partida. Se experimentó con redes neuronales simples y se revisaron TFGs y otros proyectos de la organización JdeRobot[2], los cuales ayudaron a consolidar y modelar algunas de las herramientas utilizadas en este trabajo.
2. **Desarrollo del primer piloto:** El objetivo de esta fase era desarrollar un primer piloto experto programado de forma explícita, que permitiera grabar

[1] https://roboticslaburjc.github.io/2023-tfg-juan-simo/
[2] https://jderobot.github.io/

los datos de la conducción de un coche holonómico mediante ROS2 *bags* y entrenar los primeros modelos neuronales. La prioridad en esta fase era comprender el funcionamiento de los distintos componentes necesarios para realizar el trabajo.

3. **Iteraciones de mejora:**

   (a) **Mejora de expertos y generación de *datasets*:** Se programaron varios pilotos expertos, seleccionando aquel que, aunque no era el más rápido, ofrecía la conducción más robusta siendo a la vez suficientemente rápida.

   Para alcanzar los objetivos finales, se incrementó la cantidad y variedad de datos, se implementaron técnicas de aumento de datos durante el entrenamiento, y se desarrollaron clasificadores de casos difíciles para mejorar la representación de estos en los datos de entrenamiento. También se creó un clasificador para dividir los datos entre rectas y curvas con el fin de entrenar las redes especialistas.

   Por último se optó por aplicar lo desarrollado a un piloto con dinámica de Ackermann con el objetivo de aumentar la complejidad de la tarea a realizar.

   (b) **Mejora del entrenamiento:** Tras las primeras pruebas, se desarrollaron programas para la extracción, almacenamiento y análisis de los datos, así como para el aumento y balanceo de datos. También se mejoraron los códigos de entrenamiento y las redes neuronales utilizadas, basándose en el trabajo de Sergio Paniego[3]. El objetivo siendo obtener procesos y códigos más limpios, estandarizados y fáciles de depurar y mejorar.

4. **Pruebas y resultados:** A lo largo de todas las fases del proyecto se han empleado herramientas desarrolladas por el autor del TFG para monitorizar el entrenamiento de las redes neuronales, evaluar la precisión de las redes entrenadas con tests *offline* y aumentar la eficiencia de las iteraciones de mejora. Además, en una última etapa del trabajo, se probaron las redes de manera experimental y metódica, y se desarrollaron programas para obtener métricas objetivas, permitiendo evaluar el rendimiento y comparar el piloto monolítico con el piloto de especialistas combinados.

[3] https://github.com/JdeRobot/DeepLearningStudio

# 3. Herramientas de desarrollo

En este capítulo se describen las herramientas de desarrollo utilizadas en este trabajo de fin de grado. Comprender estos aspectos es esencial para apreciar las capacidades y limitaciones de las soluciones propuestas. Nos enfocaremos en varios componentes clave: el lenguaje de programación Python, la biblioteca de aprendizaje profundo PyTorch, el sistema operativo de robots ROS2, el simulador Gazebo y la biblioteca de aumentado de datos Albumentations.

## 3.1. Lenguaje Python

Python [22] es un lenguaje de programación interpretado, de alto nivel y de propósito general, ampliamente utilizado en diversos campos, incluyendo la robótica y la inteligencia artificial. Su sintaxis simple y clara facilita el desarrollo rápido y eficiente de aplicaciones complejas, haciendo de Python una elección popular entre desarrolladores e investigadores. Su vasto ecosistema de bibliotecas y su fuerte comunidad de soporte lo convierten en una herramienta fundamental para proyectos de ciencia de datos, aprendizaje automático y desarrollo de *software* en general.

A continuación, se destacan algunas de sus características más importantes que lo diferencian de otros lenguajes de programación:

- **Sintaxis clara y legible:** Una de las principales ventajas de Python es su sintaxis clara y legible, que facilita la escritura y comprensión del código. A diferencia de lenguajes como C++ o Java, Python utiliza indentación para definir bloques de código en lugar de llaves o palabras clave específicas. Esto no solo hace que el código sea más limpio, sino que también reduce la probabilidad de errores sintácticos y mejora la mantenibilidad.
- **Tipado dinámico y gestión automática de memoria:** Python es un lenguaje de tipado dinámico, lo que significa que no es necesario declarar explícitamente el tipo de una variable cuando se define. Esto permite una programación más rápida y flexible. Además, Python gestiona automáticamente la memoria a través de un recolector de basura, liberando a los desarrolladores de la gestión manual de la memoria que es común en

lenguajes como C o C++.

- **Extensa biblioteca estándar:** Python cuenta con una extensa biblioteca estándar que proporciona módulos y paquetes para una amplia gama de tareas, desde manejo de archivos y operaciones matemáticas hasta desarrollo web y manipulación de datos. Esto lo hace muy adecuado para aplicaciones científicas, de datos y de inteligencia artificial.
- **Orientación a objetos y programación funcional:** Aunque Python es conocido por su soporte para la programación orientada a objetos, también ofrece potentes capacidades de programación funcional. Los desarrolladores pueden utilizar características como funciones de primera clase, cierres y generadores para escribir código funcional, lo que añade otra capa de flexibilidad y expresividad al lenguaje.
- ***Open source* y comunidad activa:** La comunidad de Python es una de las más grandes y activas del mundo de la programación. Esta comunidad no solo contribuye al desarrollo continuo del lenguaje y sus bibliotecas, sino que también ofrece un vasto recurso de documentación, tutoriales y soporte, facilitando el aprendizaje y la resolución de problemas.
- **Aplicaciones en ciencia de datos y aprendizaje automático:** Python ha emergido como el lenguaje preferido en el campo de la ciencia de datos y el aprendizaje automático. Bibliotecas como NumPy, pandas, SciPy, y scikit-learn, junto con herramientas de visualización como Matplotlib y Seaborn, proporcionan una infraestructura robusta para análisis de datos, modelado estadístico y desarrollo de algoritmos de aprendizaje automático. Además, *frameworks* de aprendizaje profundo como TensorFlow y PyTorch, escritos principalmente en Python, subrayan la relevancia del lenguaje en la investigación y desarrollo de inteligencia artificial.

Aunque Python ofrece muchas ventajas, también tiene algunas desventajas significativas. Su rendimiento computacional es inferior al de lenguajes compilados como C++ o Java debido a su naturaleza interpretada, lo que puede hacer que sea más lento. Además, Python puede consumir más memoria debido a su gestión automática de la misma. La existencia del *Global Interpreter Lock* limita la ejecución concurrente de *threads*, afectando el rendimiento en aplicaciones multiproceso. Tampoco es ideal para el desarrollo de aplicaciones móviles, y su tipado dinámico puede llevar a errores que solo se detectan en tiempo de ejecución. Pese a ello se ha considerado que las ventajas superan a las desventajas, y debido a ello es el lenguaje que se ha escogido para el desarrollo del TFG, donde se ha empleado la

versión 3.10.12.

## 3.2. PyTorch

PyTorch [23] es una biblioteca de código abierto para el aprendizaje automático desarrollada por el laboratorio de investigación de inteligencia artificial de Facebook. Es especialmente popular en la comunidad de investigación y desarrollo de inteligencia artificial y aprendizaje profundo debido a su flexibilidad y facilidad de uso.

Una de las características más destacadas de PyTorch es su uso de tensores, una estructura de datos similar a los *arrays* de NumPy, pero con la capacidad adicional de ser procesados en GPUs para acelerar los cálculos. Esto es particularmente útil para tareas de aprendizaje profundo que requieren operaciones de alto rendimiento.

Además, PyTorch incluye una funcionalidad de diferenciación automática conocida como *autograd*, que permite calcular gradientes automáticamente. Esto simplifica la implementación de algoritmos de retropropagación necesarios para el entrenamiento de redes neuronales, permitiendo a los desarrolladores enfocarse en la construcción y experimentación con modelos en lugar de los detalles matemáticos de los gradientes.

Otra característica importante es su diseño dinámico de grafos computacionales. A diferencia de otras bibliotecas que utilizan grafos estáticos, PyTorch construye el grafo de operaciones sobre la marcha, lo que facilita la depuración y modificación del modelo. Este enfoque dinámico permite un flujo de trabajo más intuitivo y flexible, especialmente útil en la investigación donde los modelos cambian frecuentemente.

PyTorch se utiliza en una variedad de aplicaciones, incluyendo visión por computadora, procesamiento de lenguaje natural, generación de modelos de lenguaje y conducción autónoma. Su versatilidad y compatibilidad con otras bibliotecas y herramientas, como torchvision para visión por computadora y HuggingFace para procesamiento de lenguaje natural, hacen que sea una opción preferida tanto en la academia como en la industria. Además, la comunidad activa y el soporte continuo de Facebook aseguran que PyTorch siga evolucionando y adaptándose a las necesidades emergentes en el campo del aprendizaje automático.

En este trabajo, PyTorch (2.2.0) se ha utilizado para implementar, entrenar y validar las redes neuronales encargadas de pilotar el coche. Concretamente, se han

utilizado módulos como `torch.nn` para definir la arquitectura de las redes, `torch.optim` para los algoritmos de optimización, `torchvision.transforms` para el preprocesamiento de imágenes, y `DataLoader` junto con `SubsetRandomSampler` para gestionar la carga y muestreo de los datos. Además, `TensorBoard` se ha empleado a través del módulo `SummaryWriter` para monitorizar el proceso de entrenamiento.

## 3.3. OpenCV

OpenCV (*Open Source Computer Vision Library*) [24] es una biblioteca de software libre enfocada en aplicaciones de visión por computadora y aprendizaje automático. Desarrollada inicialmente por Intel, OpenCV se ha convertido en una herramienta fundamental tanto en la industria como en la investigación.

Una de las principales características de OpenCV es su amplia colección de algoritmos optimizados para tareas de visión por computadora, tales como el reconocimiento de objetos, detección de rostros, seguimiento de movimientos, reconstrucción 3D y análisis de imágenes. Estos algoritmos están altamente optimizados para un rendimiento eficiente, permitiendo su uso en aplicaciones en tiempo real.

OpenCV es compatible con múltiples lenguajes de programación, incluyendo Python, C++, Java y MATLAB, lo que facilita su integración en diversos entornos de desarrollo. En particular, la interfaz de Python es muy popular debido a su simplicidad y a la extensa comunidad de usuarios que contribuyen con documentación y ejemplos prácticos.

Otro aspecto destacado de OpenCV es su capacidad para trabajar con múltiples tipos de entradas y salidas de medios, como imágenes y videos, de diversas fuentes y formatos. También se integra bien con otras bibliotecas y *frameworks* de aprendizaje profundo, como TensorFlow, PyTorch y Caffe, lo que amplía sus capacidades para incluir tareas de aprendizaje profundo y análisis avanzado de datos.

OpenCV se utiliza en una amplia gama de aplicaciones prácticas, desde sistemas de seguridad y vigilancia, hasta automóviles autónomos y aplicaciones de realidad aumentada. Su robustez, eficiencia y la comunidad activa de desarrolladores que la respalda aseguran que OpenCV siga siendo una herramienta esencial en el desarrollo de soluciones avanzadas de visión por computadora.

A lo largo de este TFG ha sido empleado para todo aquello que ha requerido

lectura, procesamiento y muestra de imágenes, siendo transversal para el trabajo realizado. OpenCV (4.11.0) ha tenido presencia desde el primer piloto, siendo la librería usada para la lectura de la imagen así como para el filtro que permite al piloto experto seguir la línea. Esta librería ha sido también empleada para el procesamiento de ROS2 *bags* para generar los *datasets*, leyendo, cortando (como se explica en la Subsección 4.1.3) y almacenando la información captada por la cámara del coche. Durante el entrenamiento de las redes ha sido empleado para algunos de los aumentos de datos usados, como el *flip* explicado en la Sección 4.3 y en la visualización de las gráficas de entrenamiento (ver Figura 4.21).

## 3.4. ROS2

ROS2 (Robot Operating System 2) es un *middleware* diseñado para el desarrollo de *software* en robótica. Proporciona servicios operativos como abstracción de *hardware*, controladores de dispositivos, implementación de bibliotecas, transmisión de mensajes entre procesos y gestión de paquetes. A diferencia de su predecesor, ROS, ROS2 ha sido reescrito desde cero para abordar limitaciones críticas en términos de seguridad, fiabilidad y rendimiento en sistemas distribuidos. Sus características más destacadas incluyen [25]:

- ***Middleware* DDS (*Data Distribution Service*):** ROS2 utiliza DDS, lo que mejora significativamente la comunicación en sistemas distribuidos y proporciona soporte nativo para *Quality of Service*.
- **Seguridad mejorada:** Ofrece mecanismos avanzados de seguridad, como autenticación, autorización y cifrado, que son esenciales para aplicaciones críticas y colaborativas.
- **Gestión de ciclo de vida de los nodos:** Introduce un modelo de gestión de ciclo de vida de los nodos, permitiendo un control más preciso sobre el estado y la configuración de los componentes robóticos.
- **Soporte para sistemas en tiempo real:** ROS2 está diseñado para funcionar en sistemas en tiempo real, lo cual es crucial para aplicaciones que requieren respuestas rápidas y determinísticas.
- **Comunidad activa y soporte extensivo:** La comunidad de ROS2 es muy activa, ofreciendo un vasto ecosistema de paquetes y herramientas que facilitan el desarrollo de aplicaciones robóticas complejas.

El uso de ROS2 se extiende a diversas aplicaciones robóticas, desde la automatización industrial hasta la robótica de servicio, vehículos autónomos y drones. Su arquitectura modular y escalable permite a los desarrolladores crear soluciones personalizadas adaptadas a las necesidades específicas de cada proyecto, aprovechando las ventajas de una plataforma moderna y bien soportada.

En el centro de ROS2 yace el grafo de computación, que define la estructura de los nodos del sistema y su comunicación a través de un diseño basado en publicadores y subscriptores. Los componentes que conforman ROS2 son los siguientes:

- **Nodos:** Unidad de ejecución básica en ROS2 que realiza cálculos y se comunica con otros nodos.
- **Publicadores y subscriptores (Pub/Sub):** Mecanismo fundamental para la comunicación entre nodos en ROS2. Los nodos pueden publicar mensajes en un tema (*topic*) y otros nodos pueden suscribirse a esos temas para recibir los mensajes.
- **Servicios:** Permiten la comunicación síncrona entre nodos, donde uno solicita un servicio y otro lo proporciona, similar a las llamadas a procedimientos remotos.
- **Acciones:** Extienden el concepto de servicios permitiendo una comunicación asincrónica más compleja, útil para tareas que requieren acciones prolongadas o *feedback* periódico.
- **Parámetros:** Facilitan la configuración dinámica de los nodos durante la ejecución, permitiendo ajustes en tiempo real sin necesidad de reiniciar los nodos.
- **Gráficos de composición:** Herramientas que permiten describir y lanzar conjuntos de nodos y su interconexión de manera estructurada y automatizada.
- ***Topics*:** Canales de comunicación en los cuales los nodos pueden publicar y suscribirse a mensajes, facilitando la comunicación indirecta y distribuida.
- ***Launch*:** Archivos de configuración que describen cómo iniciar múltiples nodos y configurar sus parámetros y conexiones de manera simultánea y coordinada.
- **Descubrimiento de nodos:** Mecanismos utilizados por ROS2 para que los nodos encuentren dinámicamente a otros nodos en la red, permitiendo la

configuración automática y la adaptación a cambios en la topología del sistema.

- ***Bags*:** Archivos de registro que pueden capturar y reproducir datos de mensajes, útiles para pruebas, depuración y análisis de datos.

En el transcurso de este trabajo se ha empleado la versión de ROS2 Humble Hawksbill, siendo las aplicaciones desarrolladas en el Capítulo 4, para el control del coche, nodos de ROS2 ejecutando inferencias de redes neuronales y empleando el sistema publicador-suscriptor para controlar el coche. Además se han empleado *launchers* y *bags* para la ejecución de los mundos simulados y la recopilación de datos.

## 3.5. Simulador Gazebo

Gazebo es un simulador 3D de robots potente y versátil, ampliamente utilizado para el desarrollo y prueba de aplicaciones robóticas. Proporciona un entorno de simulación realista que incluye física avanzada, gráficos de alta calidad y la capacidad de simular una amplia gama de sensores y actuadores. Gazebo permite a los desarrolladores probar algoritmos, diseñar robots y realizar pruebas de regresión sin la necesidad de hardware físico, reduciendo costos y acelerando el desarrollo. Entre otras cuenta con las siguientes características.

- **Simulación realista:** Gazebo ofrece un entorno de simulación avanzado que incluye física precisa y gráficos realistas, lo que permite representar con fidelidad escenarios robóticos complejos. Utiliza motores de física como ODE (*Open Dynamics Engine*), *Bullet* y DART para simular colisiones, fricción y dinámica de cuerpos rígidos. Para el renderizado, emplea motores como OGRE (*Object-Oriented Graphics Rendering Engine*), lo que permite una visualización detallada de sensores como cámaras RGB y *depth*.
- **Amplia compatibilidad de sensores y actuadores:** Permite simular una amplia variedad de sensores y actuadores, lo que facilita el desarrollo y prueba de sistemas robóticos completos.
- **Reducción de costos y aceleración del desarrollo:** Al eliminar la dependencia de *hardware* físico, Gazebo permite a los desarrolladores iterar rápidamente en el diseño y la implementación de algoritmos y robots.
- **Integración con ROS2:** Su integración con ROS2 proporciona una plataforma robusta para el desarrollo de aplicaciones robóticas, aprovechando las ventajas

del grafo de computación y las herramientas de desarrollo de ROS.

- **Comunidad y soporte:** Gazebo es una herramienta de código abierto con una comunidad activa que contribuye con mejoras, modelos de robots y *plugins*, asegurando un desarrollo continuo y soporte extendido. En este trabajo nos hemos beneficiado ampliamente de esta característica aprovechando modelos desarrollados y usados por Alejandro Moncalvillo [26].

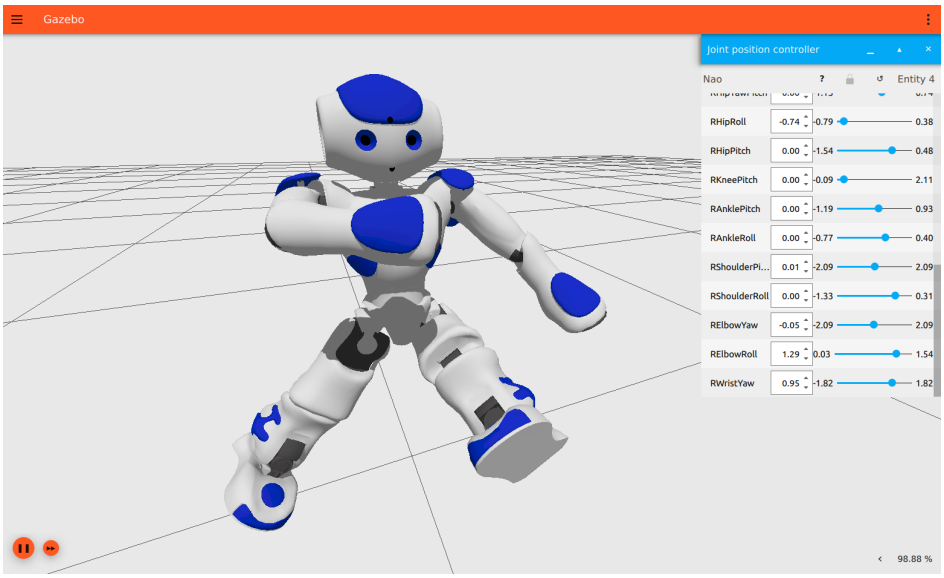



**Figura 3.1:** Robot en Gazebo.

El empleo de Gazebo 11, mantenido por la *OpenRobotics Software Foundation*[1], ha sido esencial para la generación de *datasets* supervisados, probar los pilotos explícitos y neuronales así como visualizar y analizar los resultados del trabajo realizado.

## 3.6. Albumentations

Albumentations[2] es una biblioteca de aumentado de datos para imágenes, diseñada para mejorar el rendimiento de los modelos de aprendizaje profundo. Ofrece una amplia gama de técnicas de aumentado que pueden aplicarse de manera eficiente a grandes conjuntos de datos, permitiendo a los desarrolladores aumentar la diversidad de sus datos de entrenamiento y mejorar la robustez de los modelos. Albumentations es compatible con otras bibliotecas populares de

[1]https://www.openrobotics.org
[2]https://albumentations.ai/

aprendizaje automático y procesamiento de imágenes, lo que facilita su integración en diversos flujos de trabajo de investigación y producción.

- **Técnicas de aumentado:** Albumentations proporciona una variedad de técnicas como recortes, rotaciones, cambios de brillo y contraste, entre otras, que pueden aplicarse de manera configurable para mejorar la generalización de los modelos de aprendizaje profundo.
- **Eficiencia y escalabilidad:** La biblioteca está diseñada para manejar grandes volúmenes de datos de manera eficiente, permitiendo una rápida aplicación de las transformaciones de aumentado durante la fase de entrenamiento.
- **Integración con otras bibliotecas:** Albumentations es compatible con *frameworks* y bibliotecas ampliamente utilizadas en aprendizaje automático como TensorFlow, PyTorch y scikit-learn, facilitando su adopción en diferentes entornos y flujos de trabajo.
- **Personalización y extensibilidad:** Los usuarios pueden definir fácilmente nuevas transformaciones y ajustar parámetros para adaptarse a requisitos específicos de aplicaciones y conjuntos de datos.
- **Contribución y comunidad:** La biblioteca es de código abierto y cuenta con una comunidad activa de desarrolladores que contribuyen con nuevas características, mejoras y documentación.

Como se menciona en la Sección 4.3, la versión 1.3.1 de esta biblioteca ha sido empleada para el aumentado de datos con el objetivo de enriquecer el entrenamiento de los modelos neuronales utilizados haciendo uso de técnicas de aumentado de datos como *affine*.

## 3.7. LaTeX

LaTeX es un sistema de composición de textos ampliamente utilizado en ámbitos académicos y técnicos para la creación de documentos de alta calidad tipográfica. Su principal fortaleza radica en su capacidad para manejar la estructura y el formato de documentos complejos de manera consistente y profesional. Algunas de las características distintivas de LaTeX incluyen:

- Tipografía de alta calidad.
- Formato estructurado.

- Soporte para fórmulas matemáticas.
- Gestión automática de referencias.
- Portabilidad y compatibilidad.
- Personalización avanzada.

Para la escritura de esta memoria se ha hecho uso de esta herramienta en el entorno de Overleaf, una plataforma en línea que facilita la edición, colaboración y publicación de documentos LaTeX. Permite a los usuarios trabajar en proyectos LaTeX directamente desde su navegador web, eliminando la necesidad de configurar y mantener un entorno LaTeX local. Además, Overleaf ofrece características como control de versiones, sincronización automática y compilación en tiempo real, lo que hace que la escritura y edición de documentos LaTeX sea más eficiente y accesible para equipos de trabajo distribuidos.

## 3.8. Nvidia CUDA

CUDA (*Compute Unified Device Architecture*)[3] es una plataforma de computación paralela y una API desarrollada por NVIDIA que permite el uso de la GPU para realizar cálculos generales de alto rendimiento. A través de CUDA, los desarrolladores pueden acelerar aplicaciones exigentes aprovechando la potencia de procesamiento masivamente paralelo de las tarjetas gráficas modernas.

- **Ejecutación acelerada en GPU:** CUDA permite que aplicaciones como Gazebo y bibliotecas de aprendizaje profundo utilicen la GPU en lugar del procesador, mejorando significativamente la velocidad de simulación y el entrenamiento o inferencia de modelos neuronales.
- **Compatibilidad con PyTorch:** En este trabajo, CUDA ha sido utilizado mediante PyTorch para entrenar e inferir redes neuronales de forma más eficiente. El modelo se transfiere automáticamente a la GPU si está disponible, como se muestra en el código de selección de dispositivo.
- **Mejoras en simulación con Gazebo:** Aunque Gazebo puede ejecutarse en CPU, el uso de CUDA permite aprovechar la aceleración por *hardware* para el renderizado en tiempo real, haciendo que la simulación sea más fluida y receptiva, especialmente en escenas complejas con sensores visuales.

[3]https://developer.nvidia.com/cuda-toolkit

- **Versionado y compatibilidad:** Para este proyecto se ha utilizado la versión CUDA 12.2, asegurando la compatibilidad tanto con la versión instalada de PyTorch como con los *drivers* y *hardware* del sistema de desarrollo.

Gracias al soporte de CUDA, ha sido posible mantener tiempos de entrenamiento razonables incluso con modelos complejos y conjuntos de datos aumentados, además de garantizar una experiencia fluida durante la ejecución de las simulaciones en Gazebo.

# 4. Aprendizaje por imitación

Este capítulo aborda el proceso de aprendizaje por imitación, desde la generación de datos mediante pilotos expertos hasta el entrenamiento de modelos neuronales. Se exploran los métodos utilizados para tratar y aumentar los datos, así como las estrategias de diseño y entrenamiento de redes monolíticas y especializadas.

## 4.1. Generación de conjuntos de datos

En esta sección, se detalla cómo se generaron los conjuntos de datos necesarios para el entrenamiento de los modelos neuronales. Los datos fueron recopilados utilizando un piloto experto programado explícitamente, que simula el comportamiento deseado.

### 4.1.1. Experto

El piloto experto, programado de manera implícita, es utilizado para generar datos de alta calidad. Este piloto sigue comportamientos predefinidos y proporciona una referencia precisa para el modelo de aprendizaje por imitación.

El objetivo de este piloto es ser capaz de completar diversos circuitos de carrera en Gazebo con un modelo de un coche con dinámica de Ackermann de F1 que será el mismo para todos los pilotos. Estos circuitos cuentan con una línea roja en el centro como se muestra en las Figuras 4.1 y 4.2, creadas en el TFG de Alejandro Moncalvillo [26].

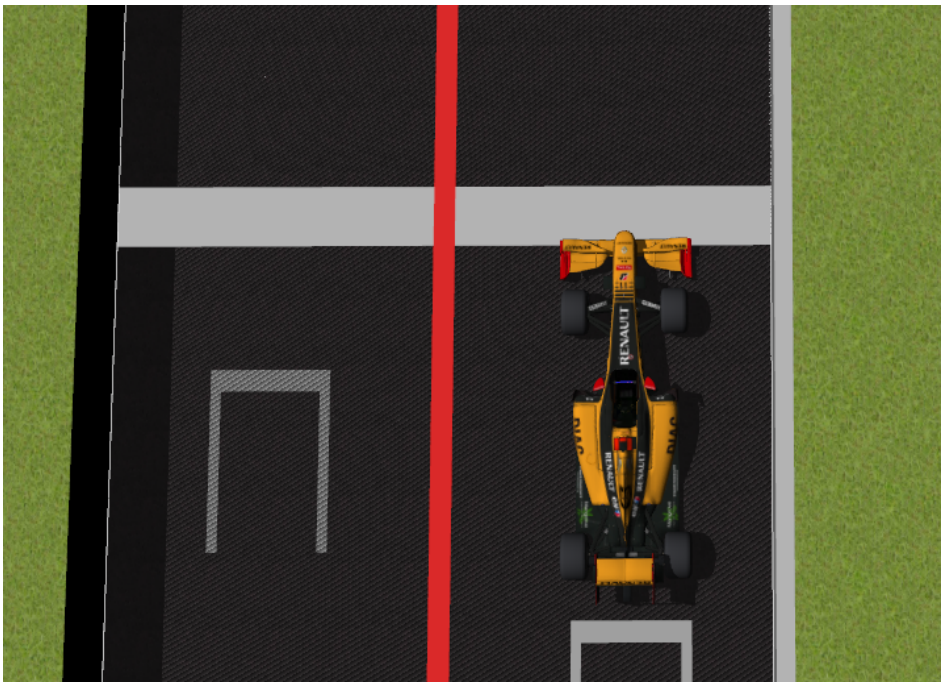



**Figura 4.1:** Circuito y modelo del coche en Gazebo.

**Figura 4.2:** En funcionamiento con cámara siguiendo al coche.

El piloto experto ha sido programado como un nodo de ROS2 capaz de controlar de forma autónoma un coche con dinámica de Ackermann dentro de los circuitos simulados en Gazebo. Este nodo recibe imágenes desde la cámara frontal del vehículo, detecta la línea roja central del circuito, y aplica un sistema de control reactivo basado en PID para seguir dicha trayectoria de forma robusta y eficiente.

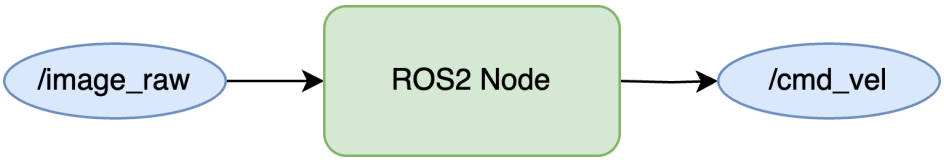



**Figura 4.3:** Diagrama simple de los pilotos.

El sistema se compone de los siguientes elementos clave:

- **Subscripciones:** Se reciben mensajes de la cámara frontal del coche (`/f1ros2/cam_f1_left/image_raw`).
- **Filtrado de imagen:** Se realiza un filtrado por color para extraer la línea roja central del circuito. La imagen se procesa con OpenCV y `cv_bridge`.
- **Detección del punto a seguir:** Este punto se usa como referencia para estimar la curvatura de la trayectoria.
- **Control PID:** Se usan dos controladores PID, uno para la velocidad angular y otro para la velocidad lineal. Estos adaptan la respuesta del vehículo a la geometría de la curva detectada.
- **Publicación de comandos:** Se publica un mensaje de tipo `Twist` en el *topic* `/f1ros2/cmd_vel` para controlar la velocidad del coche.

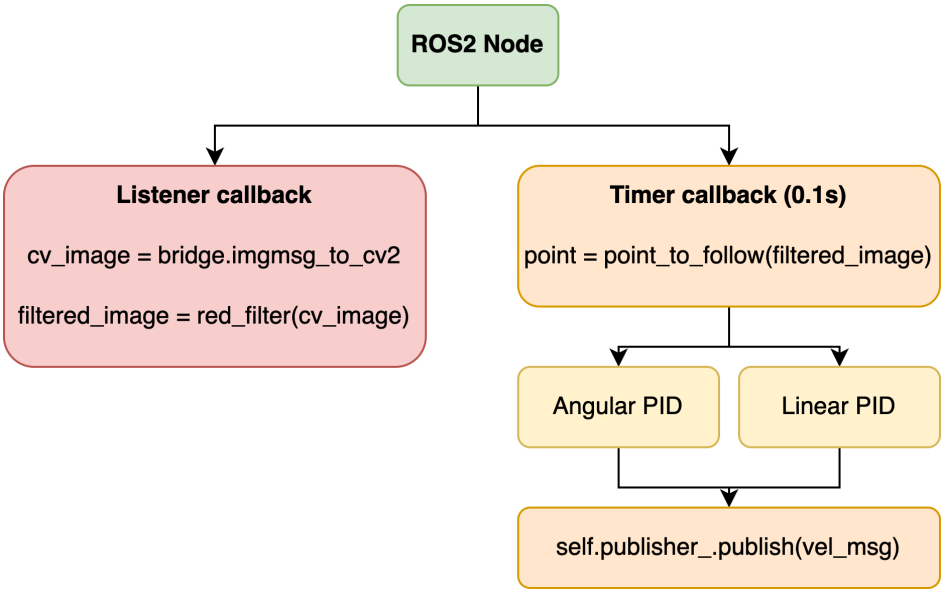



**Figura 4.4:** Diagrama del funcionamiento del piloto Experto.

La implementación del piloto experto se realiza utilizando clases en Python. Las clases proporcionan una forma organizada y modular de agrupar funciones y datos

relacionados, facilitando la mantenibilidad y la extensión del código. En este caso, la clase principal gestiona la suscripción a *topics*, el procesamiento de imágenes y el control de las velocidades del vehículo.

El piloto experto se suscribe al *topic* de la cámara para recibir imágenes en tiempo real del entorno y se procesa la imagen recibida de la cámara mediante un filtro de color que identifica la línea roja en el centro del circuito. A continuación, se determina un punto a seguir haciendo la media de los píxeles rojos dentro de un umbral cuidadosamente escogido. Si este umbral es demasiado pequeño, se anticipa demasiado a las curvas; si es demasiado grande, es muy inestable; y si está demasiado cerca del coche, no es capaz de anticiparse lo suficiente y se sale en las curvas. En este umbral, cuyos límites están marcados por las líneas horizontales verdes en las Figuras 4.5 y 4.6, se tienen en cuenta las últimas 25 filas de píxeles con línea roja en ellas. Esta decisión mejora la robustez del seguimiento de la línea y refleja la realidad de la configuración física del sensor en el vehículo.

A continuación, se presentan los fragmentos clave del código responsable del sistema de percepción visual.

```
    self.imageSubscription = self.create_subscription(
        Image,
        '/f1ros2/cam_f1_left/image_raw',
        self.listener_callback,
        10)

    def listener_callback(self, msg):
        bridge = CvBridge()
        cv_image = bridge.imgmsg_to_cv2(msg, "bgr8")
        red_lower = np.array([17, 15, 70], dtype="uint8")
        red_upper = np.array([50, 56, 255], dtype="uint8")
        red_mask = cv2.inRange(cv_image, red_lower, red_upper)
        self.filteredImage = cv2.bitwise_and(cv_image, cv_image, mask=red_mask)
```

**Extracto de código 4.1:** Suscripción a la cámara y filtrado de color rojo

```
    def choose_point(self, image):
        img_width = image.shape[1]
        height_mid = int(image.shape[0] / 2)
        x = y = count = 0

        for row in range(height_mid, height_mid + LIMIT_UMBRAL):
            for col in range(img_width):
```

```
                if not (image[row][col] == np.array([0, 0, 0])).all():
                    y += row
                    x += col
                    count += 1

        if count == 0:
            count = 1
        return [int(x / count), int(y / count)]
```

**Extracto de código 4.2:** Detección del punto a seguir de la línea roja

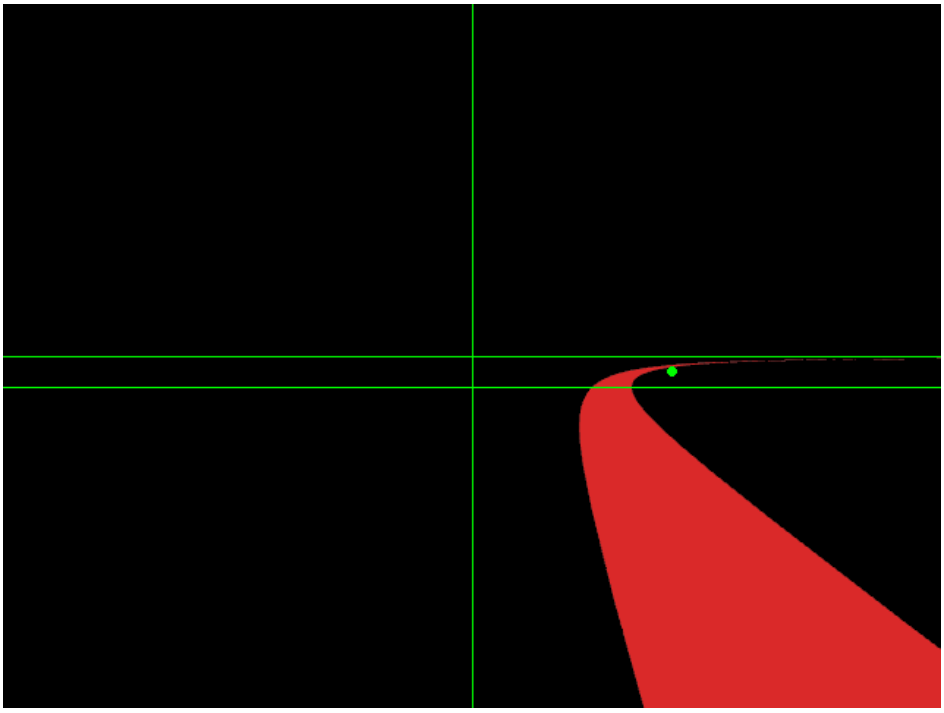


**Figura 4.5:** Imagen de curva filtrada con lineas para depuración.

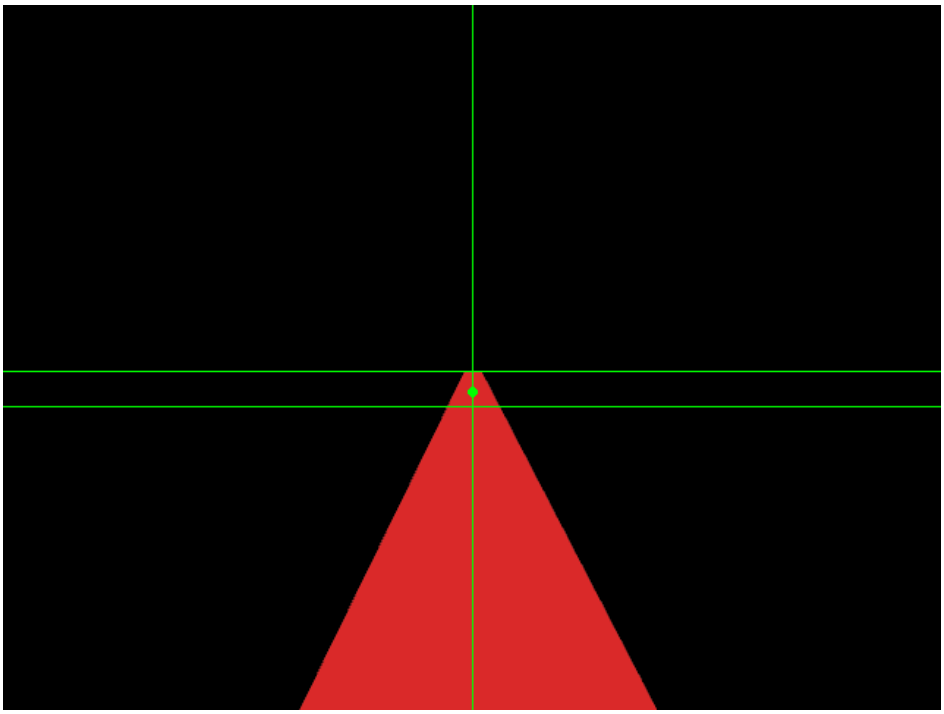


**Figura 4.6:** Imagen de recta filtrada con lineas para depuración.

La velocidad lineal y angular del vehículo proceden a calcularse a través de un controlador PID como mostrado en el Extracto de código 4.3. Este controlador calcula la diferencia entre la posición actual del vehículo y la trayectoria deseada (error) y aplica correcciones basadas en tres términos: proporcional, integral y derivativo. El término proporcional responde a la magnitud del error, el término integral acumula el error a lo largo del tiempo y el término derivativo considera la tasa de cambio del error. Esta combinación permite una respuesta suave y precisa, minimizando oscilaciones y mejorando la estabilidad.

```
    def get_pid(self, error):
        self.int_error += error
        self.int_error = max(self.min, min(self.int_error, self.max))
        dev_error = error - self.prev_error
        self.prev_error = error
        out = self.KP * error + self.KI * self.int_error + self.KD * dev_error
        out = max(self.min, min(out, self.max))
        return out
```

**Extracto de código 4.3:** Definición del PID.

La velocidad lineal del vehículo se ajusta dinámicamente en función de la velocidad angular y su tasa de cambio. En tramos rectos, donde ésta es menor y

más constante, la velocidad lineal se incrementa para mejorar el rendimiento del vehículo. Este ajuste se realiza aumentando el valor de referencia dado por el controlador PID lineal, permitiendo una aceleración controlada y segura.

Además, se tiene en cuenta el desplazamiento horizontal del punto a seguir para determinar si se va a afrontar una curva, y la velocidad del desplazamiento para detectar curvas especialmente fuertes. Esto permite al piloto acelerar más en rectas y empezar a frenar a tiempo; de otra forma, derrapa y se sale o debe ir más lento en todo el circuito.

Finalmente, el piloto experto publica información en *topics* específicos. Esto incluye la imagen filtrada para la visualización y depuración, y las velocidades calculadas para el control del vehículo junto con la imagen sin filtrar para generar adecuadamente los conjuntos de datos, de lo cual hablaremos más en la Subsección 4.1.3.

Tras varias versiones con distintos PIDs y otros mecanismos de control, optamos por utilizar como piloto experto, no al más rápido, sino a aquél que mejor se ajustaba a la línea.

Esta metodología asegura que el piloto experto sigue la línea de manera precisa y a una velocidad relativamente alta, proporcionando datos para el aprendizaje por imitación. Y ha permitido probar al piloto experto en todos los circuitos disponibles: Simple, Many Curves, Montreal, Montmeló y Nürburgring.

### 4.1.2. Errático

El piloto errático se utiliza para añadir variabilidad y ruido a los datos, simulando situaciones de recuperación que permiten al modelo aprender a manejar situaciones inesperadas y errores comunes.

Este piloto funciona igual que el anterior, pero de forma aleatoria gira bruscamente hacia uno u otro lado y vuelve a ceder el control del coche al PID, que recupera la situación y vuelve a lograr que el coche se centre [27]. Para ello, en cada iteración del código genera un número aleatorio del 1 al 100 y un 2 % de las veces asume el comportamiento errático durante 2 iteraciones.

El objetivo de esto es introducir más variabilidad en el *dataset* y con ello aumentar la robustez de la red de control extremo a extremo, ya que, como vimos en la Sección 1.4, los pilotos generados por clonación del comportamiento tienen problemas al encontrarse con situaciones nuevas. En este piloto es muy importante

desactivar la publicación de velocidades e imágenes del experto para evitar que estos datos se graben junto al resto y enseñen a las redes a salirse de la línea sin motivo aparente.

### 4.1.3. Conjunto de datos para red monoítica

Para el entrenamiento de redes neuronales necesitamos conjuntos de datos suficientemente grandes, variados y balanceados que permitan a las redes aprender a imitar el comportamiento deseado.

Para ello, lo primero es grabar los datos utilizando *bags* de ROS2, almacenando la información publicada en los *topics* del piloto experto. Es por esto que el piloto publica la imagen junto con las velocidades, para que se registren por pares en las bolsas y no haya disparidades. Si se grabara solo lo que publica la cámara del coche y las velocidades por separado, podría haber desincronización debido a las diferentes tasas de publicación.

Una vez que los datos están almacenados, es necesario extraer y procesar la información. Esto implica guardar las imágenes y crear un archivo CSV con las velocidades.

Para el procesamiento de los datos, primero se debe verificar que las rutas de almacenamiento existen y crearlas si no es así. Luego, se lee el *bag* de ROS2 y se accede a los mensajes publicados en los *topics* de interés. Las imágenes se procesan para convertirlas en un formato legible y se guardan en el directorio correspondiente. Para las velocidades, se escribe un archivo CSV con las velocidades lineales y angulares extraídas de los mensajes. Además, para poder usar posteriormente la función *sort*, se nombran las imágenes numéricamente, correspondiéndose el número de cada imagen a la fila en la que están sus valores de salida en el CSV (-1 ya que la primera fila es la que nombra las columnas).

En el caso de nuestros circuitos, la información que aportan las imágenes es nula por encima del horizonte, ya que todo lo que queda por encima es un cielo monocromático idéntico en todos los puntos y circuitos. Así que, para facilitar el entrenamiento y reducir el coste de memoria de los conjuntos de datos, se guardan cortadas horizontalmente.

Para garantizar que los datos están organizados y disponibles para el entrenamiento, se listan los directorios donde se almacenan los *bags*, y se procesan cada uno de ellos, verificando el tipo de datos y almacenando las imágenes y las

velocidades en los directorios y archivos correspondientes.

Este proceso asegura que disponemos de un conjunto de datos coherente y bien estructurado, esencial para el entrenamiento efectivo de las redes neuronales que imitarán el comportamiento del piloto experto.

Adicionalmente, para poder observar los datos y escoger si queremos guardarlos permanentemente o no, contamos con un programa que nos permite ver gráficamente la distribución de datos del *bag* previamente, como se muestra en la Figura 4.7. Esto es de utilidad para saber no solo si queremos repetir la grabación de datos, sino para saber cómo nos conviene balancearlos, lo cual se puede hacer desde la extracción, cogiendo varias veces aquellos datos menos representados.

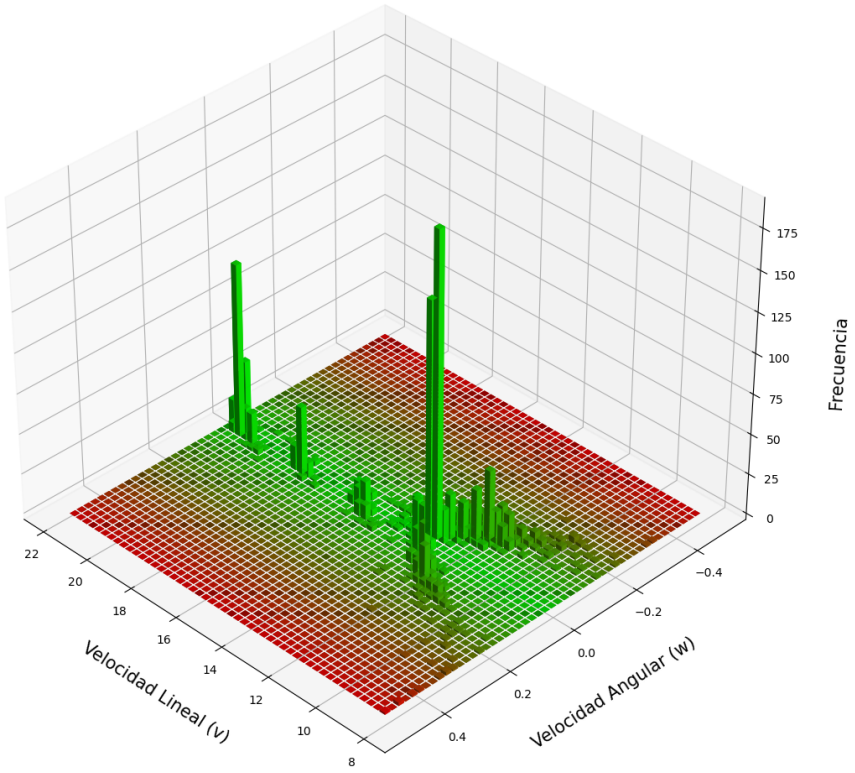



**Figura 4.7:** Representación gráfica de los datos del circuito Simple.

Como se puede comprobar en la Figura 4.7 los datos de velocidades angulares altas tienen poca representación, esto puede compensarse durante el entrenamiento, pero también conviene encontrar una forma de que los datos estén suficientemente balanceados al empezar los entrenamientos. Para ello, se ha creado un programa que lee los datos y las imágenes desde los directorios especificados. Las imágenes se procesan aplicando un filtro rojo para resaltar la línea roja en la

pista. A continuación, la imagen se analiza para determinar si contiene secciones difíciles de la pista:

- Se consideran difíciles aquellas que no tienen píxeles rojos en la línea inferior, esto significa que el coche está alejado de la línea y no encima de ella (en general recuperando).
- Se consideran difíciles aquellas que no tienen píxeles rojos en las líneas superiores, significando esto que hay una curva cerrada.
- Se consideran difíciles aquellas que no tienen píxeles rojos en los dos tercios derechos o izquierdos de la imagen, representando estos casos aquellos donde la red va a ver poca línea y debe saber qué hacer.

El programa evalúa las imágenes mediante dicho conjunto de reglas. Si una imagen se considera difícil, se guarda en un directorio separado junto con sus velocidades asociadas en un archivo CSV. Este proceso permite crear un conjunto de datos balanceado a partir del *dataset* crudo al identificar y almacenar casos difíciles por separado y representar este conjunto varias veces en el conjunto de datos de entrenamiento.

Podemos dividir los datos generados en los cuatro circuitos donde son generados:

- **Simple:** Este circuito es el más simple como indica su nombre y sus datos cuentan con menos representación en el *dataset* final empleado en los entrenamientos.
- ***Many curves*:** Una versión más compleja que la anterior compuesta exclusivamente de curvas aportando. Además cuenta con un color de recta más oscuro, añadiendo robustez a la detección de la línea de las redes.
- **Montmeló:** Uno de los dos circuitos complejos, con algunas de las situaciones más difíciles para los pilotos, incluyendo curvas muy cerradas y curvas en distintas direcciones una detrás de otra.
- **Montreal:** El segundo de los circuitos complejos. Tuvo que ser eliminado tras la implementación del programa para separar rectas y curvas, ya que la peor calidad de las imágenes suponía que casi todo se clasificaba como curva. Esto se puede apreciar en la Figura 4.15, la calidad de la línea debería ser como en la Figura 4.14 perteneciente al circuito de Montmeló.

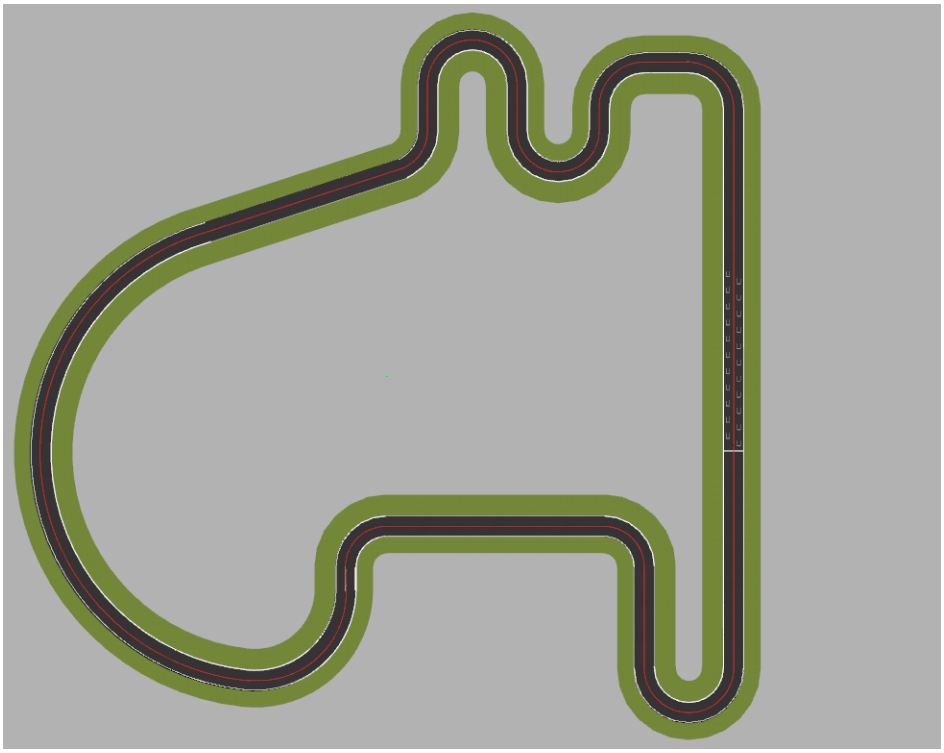


**Figura 4.8:** Vista de pájaro del circuito Simple.

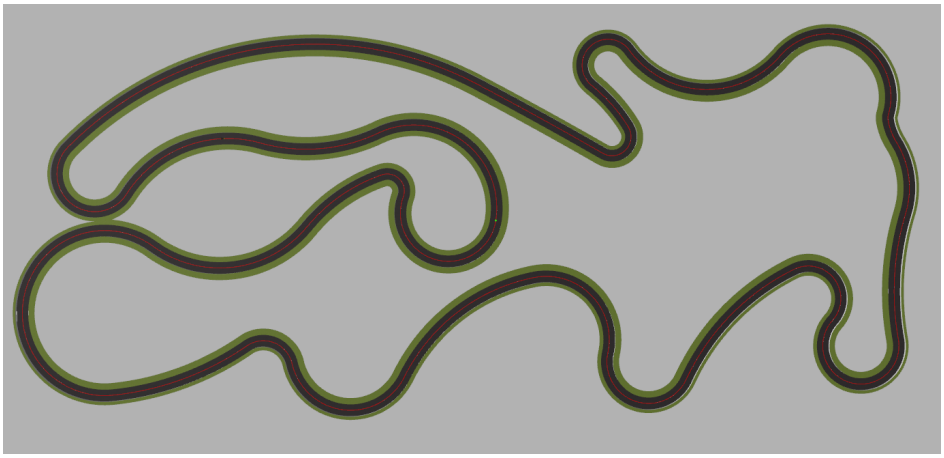


**Figura 4.9:** Vista de pájaro del circuito *Many curves*.

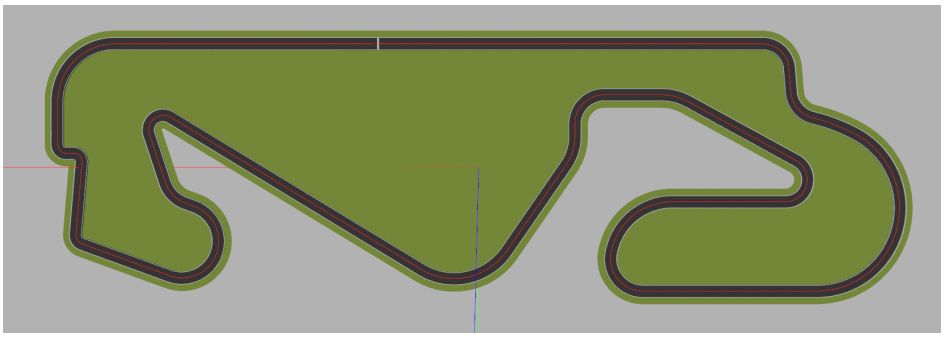


**Figura 4.10:** Vista de pájaro del circuito Montmeló.

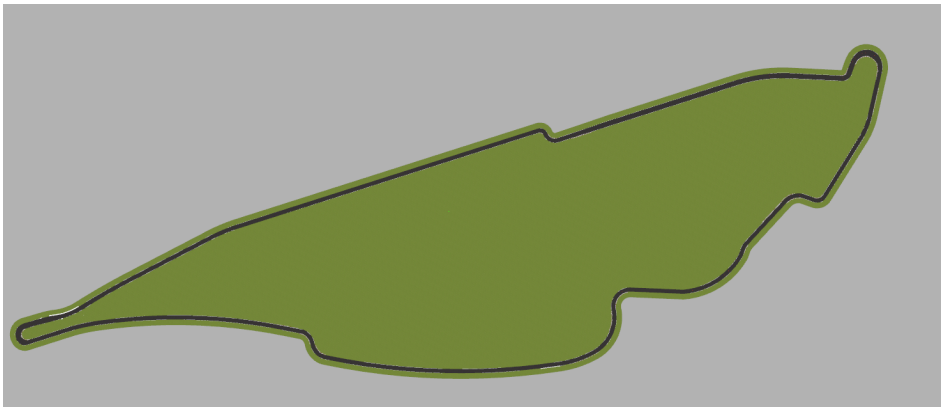


**Figura 4.11:** Vista de pájaro del circuito Montreal.

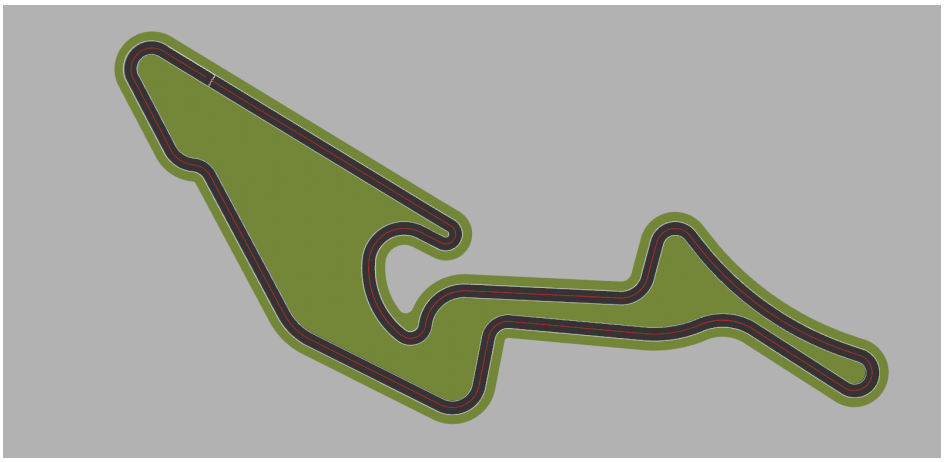


**Figura 4.12:** Vista de pájaro del circuito Nürburgring.

El *dataset* balanceado previo al aumento de datos explicado en la Sección 4.3 consta de 148526 pares de imagen/velocidades. Este *dataset* generado en frío contiene: los datos generados en los circuitos, los datos de los casos difíciles y los datos generados por el piloto errático de la Subsección 4.1.2.

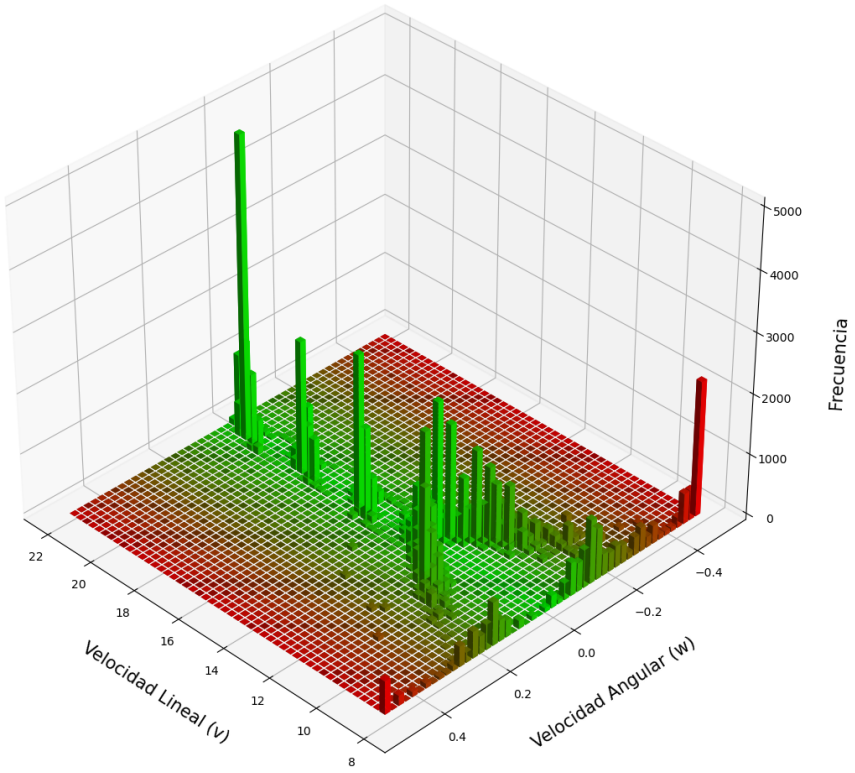



**Figura 4.13:** Representación gráfica del *dataset* crudo previo al aumentado de datos.

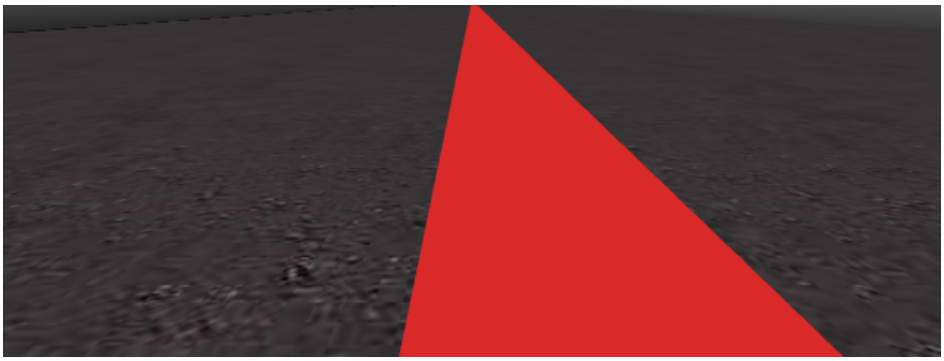


**Figura 4.14:** Imagen del circuito Montmeló.

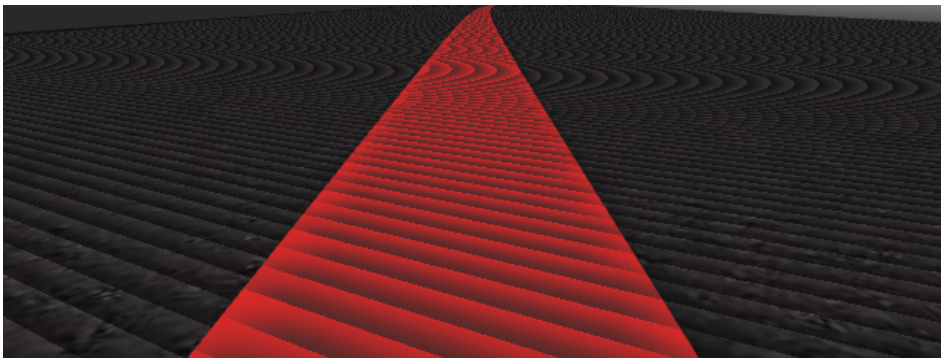


**Figura 4.15:** Imagen del circuito Montreal.


**Figura 4.16:** Imagen del circuito Montmeló siendo clasificado.

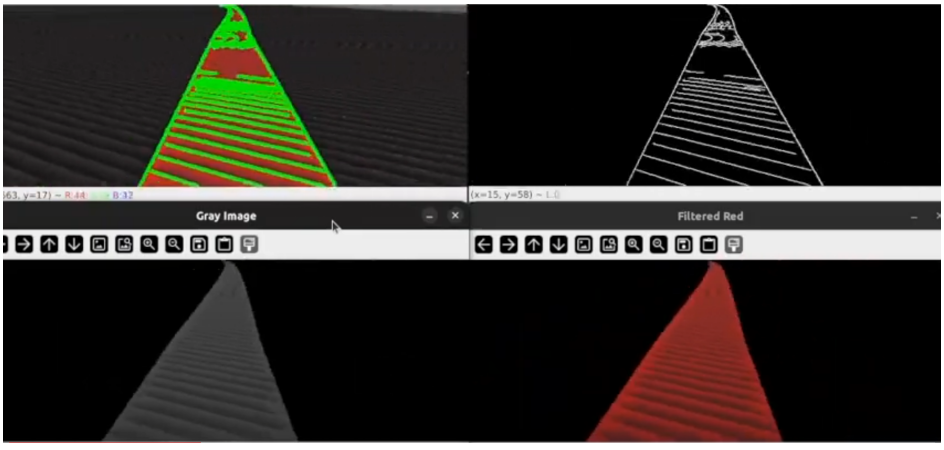



**Figura 4.17:** Imagen del circuito Montreal siendo clasificado.

### 4.1.4. Conjunto de datos para redes especialistas

Los especialistas también necesitan conjuntos de datos especializados y, si bien podríamos dividir los datos en función de su velocidad angular, se ha optado por un criterio más fiable y que se puede usar para clasificar imágenes en tiempo real y así ser útil posteriormente a la hora de crear el piloto de redes especialistas.

Para ello, hemos creado un programa que clasifica las imágenes y las velocidades asociadas en dos categorías: líneas rectas y curvas. Las imágenes se procesan para determinar si representan una línea recta o una curva siguiendo el siguiente proceso:

- Primero se filtra la línea roja.
- Después se pasa la imagen a escala de grises.
- A continuación se emplea *cv2.Canny* para encontrar bordes.
- Posteriormente se saca el contorno de la línea usando *cv2.findContours* y escogiendo los dos más largos con *sorted*.
- Para acabar se emplean *cv2.arcLength* y *cv2.approxPolyDP* para escoger si el contorno corresponde o no a una línea recta.

Para observar esto de forma gráfica se han tomado las imágenes obtenidas por el piloto experto y en qué posición del mundo han sido obtenidas. Posteriormente se han pasado estas posiciones a píxeles como se explica en el Capítulo 5, y se han pasado las imágenes al mismo clasificador que separa los *datasets* y que escoge qué red neuronal toma el control en el piloto mezcla de expertos durante la ejecución.

Con el resultado de dicha clasificación se ha generado una representación visual:

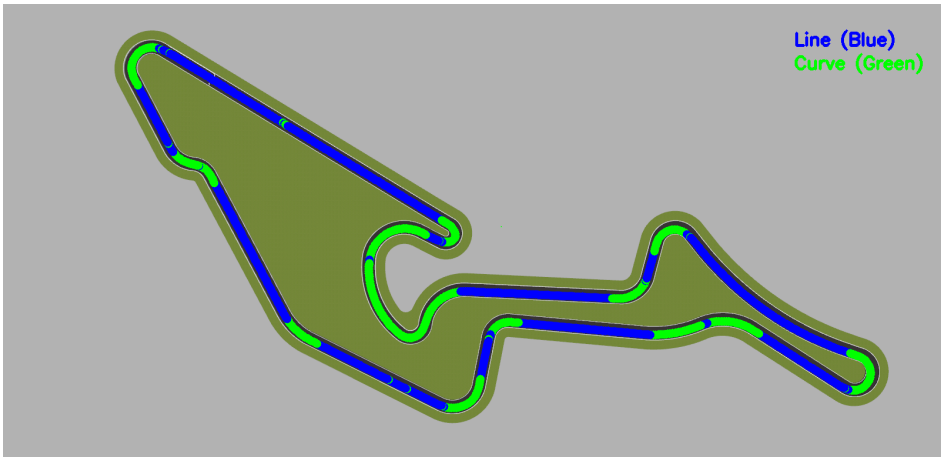



**Figura 4.18:** Clasificación de rectas y curvas en Nürburgring.

Podemos observar que nuestro clasificador no es perfecto y hay algunos puntos en rectas que considera curvas y viceversa. Esto es principalmente debido a que en estos puntos el coche pilotado se había separado de la línea más de lo debido. Pero en general, vemos que la clasificación es muy buena.

Podemos por lo tanto determinar que los subconjuntos de datos empleados para el entrenamiento de las redes especialistas permiten una efectiva especialización en curvas o rectas.

El proceso de clasificación en rectas y curvas de los conjuntos de datos puede observarse siguiendo el enlace a pie de página[1], donde se observa cómo se procesan unas imágenes de Montreal y Montmeló. También puede apreciarse un problema mencionado abajo viéndose la comparativa entre ambos, véanse las Figuras 4.16 y 4.17.

Después de procesar las imágenes, estas se almacenan en directorios separados dependiendo de si representan una línea recta o una curva, y se guardan las velocidades asociadas en archivos CSV separados, contando el *dataset* de rectas previo al aumento de datos 62004 pares de imagen/velocidades y el de curvas 86522.

## 4.2. Modelo neuronal

Aquí se describe la estructura y arquitectura del modelo neuronal utilizado para el aprendizaje por imitación. Se detalla el uso de PilotNet [15] como base para la red neuronal y las modificaciones realizadas para adaptarla a las necesidades específicas del proyecto. Esta arquitectura ha sido también usada en el desarrollo de los trabajos de otros alumnos [28] [29] [26].

El modelo neuronal se ha desarrollado utilizando la biblioteca PyTorch y se basa en la arquitectura PilotNet, diseñada originalmente para el control de vehículos autónomos. La red toma imágenes de entrada y produce etiquetas de salida correspondientes a comandos de control.

La arquitectura del modelo es la siguiente (ver Figura 4.19):

- Una capa de normalización por lotes (*Batch Normalization*) para estabilizar y acelerar el aprendizaje.

[1] https://www.youtube.com/watch?v=udUKPIobrwc

- Cinco capas convolucionales, cada una seguida de una función de activación ReLU:
  - La primera capa convolucional tiene 24 filtros, un tamaño de núcleo de 5 y un paso de 2.
  - La segunda capa convolucional tiene 36 filtros, un tamaño de núcleo de 5 y un paso de 2.
  - La tercera capa convolucional tiene 48 filtros, un tamaño de núcleo de 5 y un paso de 2.
  - La cuarta capa convolucional tiene 64 filtros, un tamaño de núcleo de 3 y un paso de 1.
  - La quinta capa convolucional tiene 64 filtros, un tamaño de núcleo de 3 y un paso de 1.
- Una capa de aplanamiento (*Flatten*) para convertir las salidas de las capas convolucionales en un vector unidimensional.
- Cinco capas completamente conectadas (*Fully connected*):
  - La primera capa completamente conectada tiene 1164 unidades.
  - La segunda capa completamente conectada tiene 100 unidades.
  - La tercera capa completamente conectada tiene 50 unidades.
  - La cuarta capa completamente conectada tiene 10 unidades.
  - La quinta y última capa produce el número de etiquetas de salida especificado por el modelo.

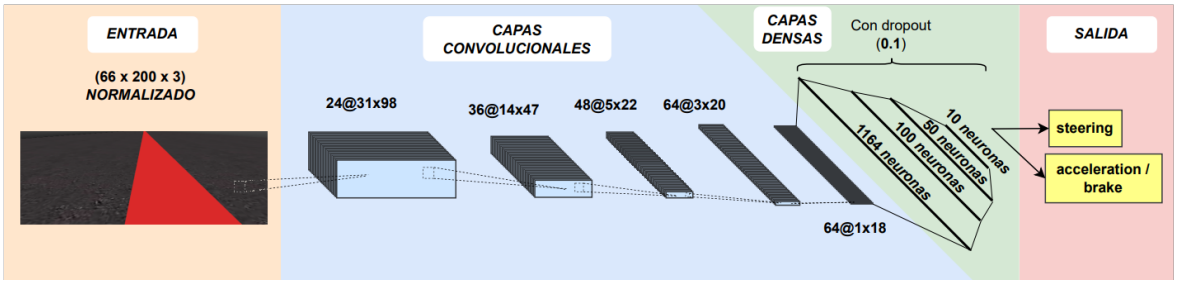



**Figura 4.19:** Arquitectura del modelo PilotNet [4].

## 4.3. Entrenamiento

### 4.3.1. Preparación de los datos

Se definen rutas a varios directorios que contienen los datos de entrenamiento en formato CSV. Estos datos incluyen diferentes escenarios y condiciones de conducción, como curvas cerradas y situaciones con o sin comportamiento errático.

Se implementan transformaciones sobre las imágenes de entrada como parte del preprocesamiento. Estas incluyen técnicas de recorte y normalización integradas directamente en el conjunto de datos, así como estrategias de aumentado de datos aplicadas antes del entrenamiento. El objetivo de estas transformaciones es aumentar la variabilidad del conjunto de entrenamiento y mejorar la capacidad de generalización del modelo neuronal frente a cambios en el entorno visual.

Para cargar y preparar los datos, se define la clase `PilotNetDataset`, que hereda de `Dataset` de `PyTorch`. Esta clase realiza las siguientes tareas:

Primero, se define el constructor que recibe la ruta a los datos, las transformaciones y las opciones de preprocesamiento. Dependiendo de las opciones, se configura el tipo de imagen y datos a utilizar.

A continuación, se cargan las imágenes y etiquetas desde los archivos CSV. Se ordenan las imágenes con *sort* para asegurar que el orden de las imágenes en al *array* corresponda con sus pares de datos. El resultado es un *array* con pares de velocidades separados por comas y otro con las direcciones a las imágenes. Para que estos datos sean usables debemos leer las imágenes y transformarlas al tamaño que recibe nuestra red (200, 66, 3), también tenemos que procesar las etiquetas de datos para obtener una lista que contenga tuplas para acceder fácilmente a cada velocidad.

Posteriormente, se aplican las técnicas de aumentado de datos en caliente elegidas con las opciones de preprocesamiento recibidas:

- ***Flip*:** Invierte la imagen horizontalmente (efecto espejo) y cambia de signo la velocidad angular correspondiente. Esta transformación se justifica por la naturaleza simétrica de la tarea de seguir una línea central. Al tratarse de una tarea donde el comportamiento es equivalente a izquierda y derecha, el volteo permite equilibrar los datos de entrenamiento para ambos sentidos de giro.

- ***Affine*:** Haciendo uso de la biblioteca Albumentations (ver Sección 3.6) aplicamos esta transformación, que consiste en desplazar las imágenes horizontalmente en cualquiera de los dos sentidos una distancia aleatoria (con un máximo de 40 % de desplazamiento) y podría además rotarlas, lo cual no utilizamos. Por supuesto esto significa que hay que ajustar la velocidad angular de forma proporcional al desplazamiento. De esta forma generamos artificialmente datos de situaciones complicadas y de recuperación como podemos observar en la Figura 4.20.

También se aplica *normalización*, que normaliza ambas etiquetas de velocidad lineal y angular entre -1 y 1, lo cuál acelera mucho la velocidad de entrenamiento. Es importante que los pilotos cuenten con una función de desnormalización para poder utilizar las redes entrenadas con esta opción.

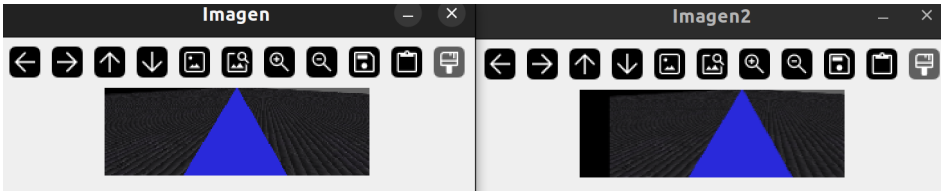



**Figura 4.20:** Comparativa de una imagen con y sin *affine*.

Tras este proceso tenemos definido nuestro conjunto de datos, que tras el aumentado de datos cuenta con 594104 pares de imagen/velocidades, y lo dividimos entre datos de entrenamiento (90 %) y de validación de forma aleatoria.

### 4.3.2. Hiperparámetros y configuración del entrenamiento

Se carga la arquitectura del modelo PilotNet y se configura para utilizar la GPU si está disponible. Esto permite un entrenamiento más rápido y eficiente.

Se utiliza la función de pérdida MSE (*Mean Squared Error*) y el optimizador *Adam* para ajustar los pesos del modelo durante el entrenamiento. La elección del criterio de pérdida y el optimizador es crucial para la convergencia del modelo.

La función de pérdida MSE se define como:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

donde $N$ es el tamaño del conjunto de entrenamiento. En nuestro caso particular:

$$\ell(x, y) = \text{mean}(L)$$

En este contexto, $x$ representa la salida del modelo, es decir, las velocidades predichas (lineal y angular) por la red neuronal para una determinada imagen de entrada. Por otro lado, $y$ son las etiquetas reales, es decir, las velocidades objetivo asociadas a dicha imagen en el conjunto de entrenamiento.

Ambas velocidades se expresan como tensores bidimensionales por muestra, por lo que cada $x_n$ e $y_n$ es un vector de dimensión 2: `[velocidad_lineal, velocidad_angular]`. El modelo busca minimizar el error cuadrático medio entre la predicción del modelo y el valor real para cada muestra del lote de entrenamiento.

### 4.3.3. Proceso de entrenamiento

El modelo se entrena durante varias épocas. En cada época, se itera sobre el conjunto completo de entrenamiento, calculando la pérdida y ajustando los pesos del modelo mediante retropropagación. Se utiliza TensorBoard[1] (una herramienta que ayuda con la representación gráfica en aprendizaje automático) para registrar las pérdidas de entrenamiento y validación.

El estado del modelo se guarda periódicamente, y las pérdidas se registran para un seguimiento detallado del rendimiento. Esto ayuda a monitorizar el progreso y detectar problemas como el sobreajuste.

Después de cada época, se evalúa el modelo en el conjunto de datos de validación para monitorizar su rendimiento. Si el modelo mejora, se guarda como el mejor modelo basado en la validación.

Tras cada evaluación, usando la función de pérdida MSE del apartado anterior, se compara con iteraciones anteriores: si el resultado es mejor que el último mejor resultado, se actualiza éste y se guardan los valores de la red; al final del entrenamiento quedan guardadas la red con los últimos valores y la red con los valores que obtuvieron el mejor resultado. Esto asegura que el modelo más preciso y robusto esté disponible para despliegue.

Al finalizar el entrenamiento, se generan gráficas de la pérdida de entrenamiento a lo largo de las épocas, como por ejemplo las mostradas en la Gráfica 4.21. Estas gráficas ayudan a analizar el comportamiento del modelo y a realizar los ajustes necesarios para mejorar el rendimiento.

[1] https://www.tensorflow.org/tensorboard?hl=es-419

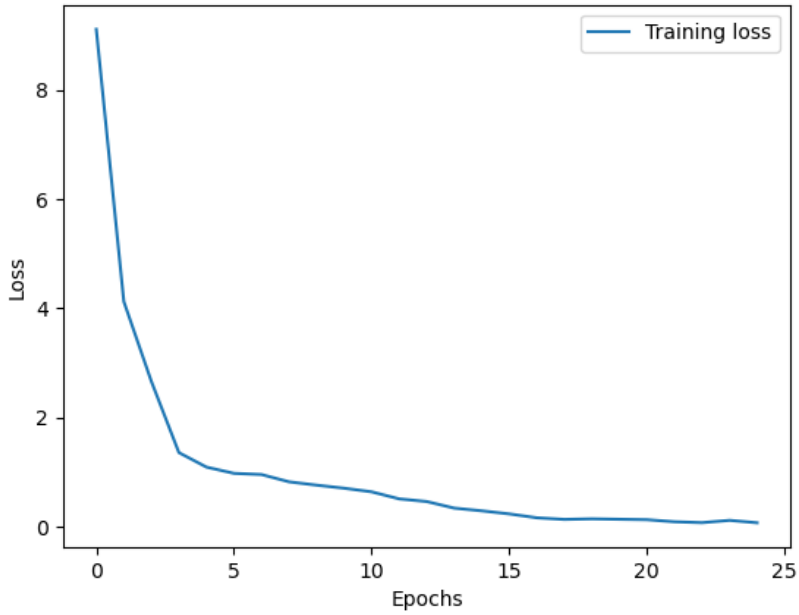



**Figura 4.21:** Ejemplo de gráfica de entrenamiento.

Este proceso detallado asegura que el modelo PilotNet esté bien entrenado y sea capaz de realizar predicciones precisas en diversos escenarios de conducción.

## 4.4. Monolítico

En este trabajo, hemos desarrollado un piloto monolítico que controla el coche utilizando una red neuronal de extremo a extremo. Este enfoque permite que el coche siga una línea en la pista basándose únicamente en las imágenes capturadas por una cámara montada en el vehículo.

Al igual que el piloto experto, el monolítico ha sido programado como un nodo de ROS2 capaz de controlar de forma autónoma un coche con dinámica de Ackermann dentro de los circuitos simulados en Gazebo. Recibiendo imágenes desde la cámara frontal del vehículo y publicando las velocidades a través de los mismos *topics*.

Dentro de esta clase, se configura el dispositivo de PyTorch para utilizar la GPU si está disponible, lo que mejora significativamente el rendimiento del modelo. Luego, se carga el modelo entrenado de la forma explicada en la sección anterior.

```
    if torch.cuda.is_available():
        self.device = torch.device("cuda")
        device_type = "cuda"
    else:
        self.device = torch.device("cpu")
        device_type = "cpu"

    self.model = PilotNet([200, 66, 3], 2).to(self.device)
    self.model.load_state_dict(torch.load(MODEL_PATH))
    self.model.to(self.device)
```

**Extracto de código 4.4:** Selección de dispositivo y carga del modelo.

Cada vez que se recibe una nueva imagen de la cámara se convierte a una imagen que pueda entender OpenCV y se recorta para centrarse en la mitad inferior, donde no solo se encuentra la pista sino que de esta manera se corresponde con la forma de los datos con los que ha sido entrenado. La imagen recortada se redimensiona al tamaño que recibe nuestra red y se convierte en un tensor que la red neuronal puede procesar.

```
    bridge = CvBridge()
    self.img = bridge.imgmsg_to_cv2(msg, "bgr8")
    half_height = self.img.shape[0] // 2
    bottom_half = self.img[half_height:, :, :]

    img = cv2.resize(bottom_half, (int(200), int(66)))
```

**Extracto de código 4.5:** Corte de la mitad superior de la imagen recibida.

Una vez preprocesada, la imagen se pasa a la red PilotNet, que predice las velocidades lineal y angular necesarias para seguir la pista que se desnormalizan para publicarlas como un mensaje de velocidad para controlar el coche.

```
    preprocess = transforms.Compose([
        transforms.ToTensor()
    ])

    img_tensor = preprocess(img).to(self.device)
    img_tensor = img_tensor.unsqueeze(0)

    predictions = self.model(img_tensor)

```

```
10      v = predictions.data.cpu().numpy()[0][0]
11      w = predictions.data.cpu().numpy()[0][1]
12      if self.norm:
13          v = denormalize(v, MIN_LINEAR, MAX_LINEAR)
14          w = denormalize(w, -MAX_ANGULAR, MAX_ANGULAR)
```

**Extracto de código 4.6:** Inferencia neuronal de la red monolítica.

El enfoque monolítico simplifica el desarrollo y despliegue del sistema, ya que todos los componentes necesarios para el control del coche se encuentran en una sola entidad. Este método demuestra la capacidad de las redes neuronales para aprender comportamientos complejos basándose únicamente en entradas visuales.

## 4.5. Mezcla de especialistas

En esta sección se describe una estrategia avanzada en la que se combinan dos modelos especializados para mejorar el rendimiento del coche en el circuito. A diferencia del modelo monolítico, aquí se utilizan dos redes neuronales distintas: una para rectas y otra para curvas. Cada red se entrena con conjuntos de datos específicos generados por un clasificador, como se explica en la Subsección 4.1.3.

El proceso comienza con la suscripción del piloto al *topic* de la cámara para recibir las imágenes publicadas por el nodo del coche. Si está disponible, se utiliza una GPU y se cargan ambas redes especializadas.

Una vez configurado, el sistema espera a recibir imágenes. Estas imágenes se convierten a un formato manejable con CSV y se recortan para adaptarse al formato requerido por las redes neuronales. La diferencia principal respecto al piloto monolítico es la clasificación de la imagen antes de redimensionarla y transformarla en un tensor. Para ello se emplea el mismo clasificador programado explícitamente para la clasificación de las imágenes y la generación de los *datasets* particulares de rectas y curvas. Según el resultado de la clasificación en inferencia, se selecciona la red especialista correspondiente para el pilotaje del coche.

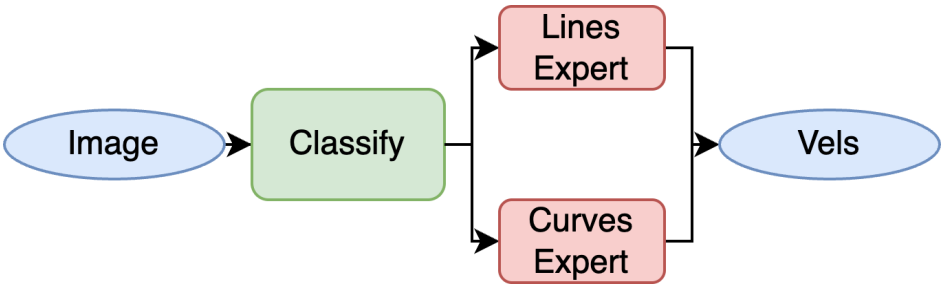



**Figura 4.22:** Diagrama simple de la mezcla de especialistas.

```
    shape = classify_line_shape(bottom_half)
    preprocess = transforms.Compose([
        transforms.ToTensor()
    ])

    img_tensor = preprocess(img).to(self.device)
    img_tensor = img_tensor.unsqueeze(0)

    if shape == 'LINE':
        predictions = self.l_model(img_tensor)
        self.classify = "Line"
    else:
        predictions = self.c_model(img_tensor)
        self.classify = "Curve"

    v = predictions.data.cpu().numpy()[0][0]
    w = predictions.data.cpu().numpy()[0][1]
    if self.norm:
        v = denormalize(v, MIN_LINEAR, MAX_LINEAR)
        w = denormalize(w, -MAX_ANGULAR, MAX_ANGULAR)
```

**Extracto de código 4.7:** Clasificación e inferencia neuronal de la mezcla de especialistas.

La ventaja de esta estrategia es que cada especialista puede concentrarse en su caso específico. La red especializada en curvas no necesita preocuparse por la aceleración en rectas y viceversa. Durante el entrenamiento, la red de curvas solo se expone a datos de curvas, evitando confusiones y eliminando la necesidad de balancear el *dataset*. De igual manera, la red de rectas se enfoca especialmente en la aceleración, optimizando su rendimiento en esa situación específica.

# 5. Validación experimental

Las soluciones de control extremo a extremo (*end-to-end control*) basadas en aprendizaje profundo, como las redes neuronales entrenadas para conducción autónoma, se evalúan habitualmente utilizando tanto métricas cuantitativas como cualitativas. Estas métricas permiten determinar en qué medida el comportamiento del sistema entrenado se aproxima al comportamiento deseado, generalmente proporcionado por un piloto humano o una política de referencia.

Entre las métricas cuantitativas más empleadas se encuentran el error cuadrático medio (*Mean Squared Error*, MSE) entre las salidas predichas por la red neuronal y las salidas del piloto experto (velocidades lineales y angulares), la duración promedio de cada episodio (por ejemplo, tiempo sin colisión) y la distancia recorrida antes de cometer un error crítico. En este trabajo se ha optado por el análisis del MSE respecto al experto y el error de trayectoria con respecto a una línea ideal trazada sobre el circuito.

Asimismo, se utilizan representaciones visuales del recorrido del vehículo, que permiten observar patrones de comportamiento inadecuado o inconsistencias en la trayectoria. Esta validación cualitativa resulta particularmente útil en entornos simulados, donde pueden observarse fácilmente desviaciones visuales del trazado ideal sin necesidad de instrumentación adicional.

El uso de datos grabados para generar métricas objetivas y reproducibles es una práctica común cuando se trabaja con simuladores como Gazebo, y resulta especialmente útil cuando los modelos y mundos tienen un alto coste computacional o cuando se busca comparar múltiples políticas bajo las mismas condiciones.

Para lograr esto se han generado ROS2 *bags* que capturaban la posición, tiempo de simulador, velocidades e imágenes; y posteriormente procesando estas *bags* y creando CSVs.

Cada una de estas *bags* consta de la ejecución de n vueltas de uno de los pilotos en uno de los circuitos, y antes de sacar métricas se procesa dicho CSV dividiéndolo en datos de cada vuelta a través de los datos de odometría y usando estos mismos datos se genera un CSV con las medias de velocidades, posiciones y tiempos.

Para el procesamiento de estos CSVs se ha desarrollado un programa que, con

estos datos, nos da las siguientes herramientas para evaluar a los pilotos:

- Una representación del recorrido realizado por los tres pilotos de forma simultánea teniendo en cuenta su posición en distintos instantes de tiempo.
- Tiempo medio por vuelta de los pilotos en cada uno de los circuitos evaluados.
- Una gráfica con la distancia absoluta en metros a la línea roja de cada piloto. Y el error cuadrático medio de todos los pilotos asumiendo que la perfección es lograr una distancia 0 a la línea.
- Gráficas comparando las salidas de los dos pilotos neuronales, monolítico y mezcla de especialistas, a las salidas del piloto experto. Además el error cuadrático medio de las salidas de los pilotos neuronales en comparación al experto al que intentan imitar.

Cabe destacar que la red monolítica y ambas redes especialistas han seguido el mismo entrenamiento, con las mismas épocas, tasas de aprendizaje, semillas, aumentado de datos, etc. Y por supuesto también han sido entrenadas con los mismos conjuntos de datos, siendo igual el número de datos con los que se entrenó la red monolítica a la suma de los conjuntos de las redes especialistas (más de 500.000 durante el entrenamiento tras el aumentado de datos).

Gracias a seguir todo lo anterior metódicamente podemos comparar de forma clara y objetiva las dos aproximaciones: piloto monolítico y piloto por mezcla de especialistas.

## 5.1. Ejecuciones típicas

Todos los pilotos han pilotado un mismo coche y nunca de forma simultánea, por ello para poder observar sus ejecuciones cualitativamente se ha tomado una vista de pájaro de tres circuitos (véanse las Figuras 4.8, 4.10, 4.12) usando las herramientas de visualización de Gazebo para rotar la cámara de forma paralela al plano formado por los ejes x e y. Haciendo esto se ha podido establecer mediante regresión lineal una conversión de posiciones en el mundo de Gazebo y posiciones en píxeles en el mapa.

De esta forma, nos es posible representar sus posiciones en nuestra vista de pájaro de forma simultánea, haciendo avanzar el tiempo de forma simulada y actualizando las posiciones de los distintos pilotos en función de él, obteniendo así los siguientes resultados por circuito:

- **Simple**: Circuito empleado en el entrenamiento para añadir volumen de datos, el más fácil de los circuitos. Puede verse el resultado de una vuelta de los pilotos en el vídeo a pie de página[1].
- **Montmeló**: Circuito principal empleado en el entrenamiento dada su aportación a los casos difíciles. Se trata del circuito más difícil por sus varias curvas cerradas y en especial por dos curvas difíciles seguidas y en sentidos contrarios hacia el final del circuito, esto puede apreciarse en los siguientes apartados en los tiempos y errores medios de los pilotos. Puede verse el resultado de una vuelta de los pilotos en el vídeo a pie de página[2].
- **Nürburgring**: Circuito empleado en evaluación por su compleja primera curva que ha permitido probar pilotos de forma rápida. Pasada la primera curva se trata de un circuito más complejo que el Simple pero que no da muchos más problemas a los pilotos. Puede verse el resultado de una vuelta de los pilotos en el vídeo a pie de página[3].

Todos los modelos y mundos de Gazebo pueden encontrarse en el repositorio del proyecto[4], habiéndose usado el *f1_renault_camera* y los circuitos *big_circuit_name* para el coche de Ackermann.

## 5.2. Evaluación *offline*

La evaluación *offline* consiste en aplicar las redes neuronales directamente sobre el conjunto de datos de validación, entregándoles como entrada las imágenes capturadas por el vehículo, y comparando sus salidas con los comandos de control generados por el piloto experto. Este procedimiento permite estimar qué tan bien las redes logran aproximar el comportamiento del controlador sin necesidad de ejecutar en el entorno real.

Para cada arquitectura evaluada, se calcula el error cuadrático medio (MSE) tanto para la velocidad lineal $v$ como para la velocidad angular $w$.

[1] https://youtu.be/1YMXtw_ocXk
[2] https://youtu.be/bS_hbgCbdL0
[3] https://youtu.be/JzfMb0Q-MUo
[4] https://github.com/RoboticsLabURJC/2023-tfg-juan-simo

### 5.2.1. Red monolítica

La red monolítica presenta un desempeño destacado en el test *offline*, con un MSE de 0,0415 para la velocidad lineal y de 0,00051 para la velocidad angular. Esto indica que la red logra seguir con buena precisión las salidas del piloto experto.

En la Figura 5.1 se puede observar cómo las predicciones de la red (línea naranja) se ajustan estrechamente a las señales generadas por el experto (línea azul), con pequeñas desviaciones principalmente en los tramos más dinámicos. El modelo mantiene una buena regularidad, sin fluctuaciones bruscas, lo que sugiere una buena capacidad generalizadora.

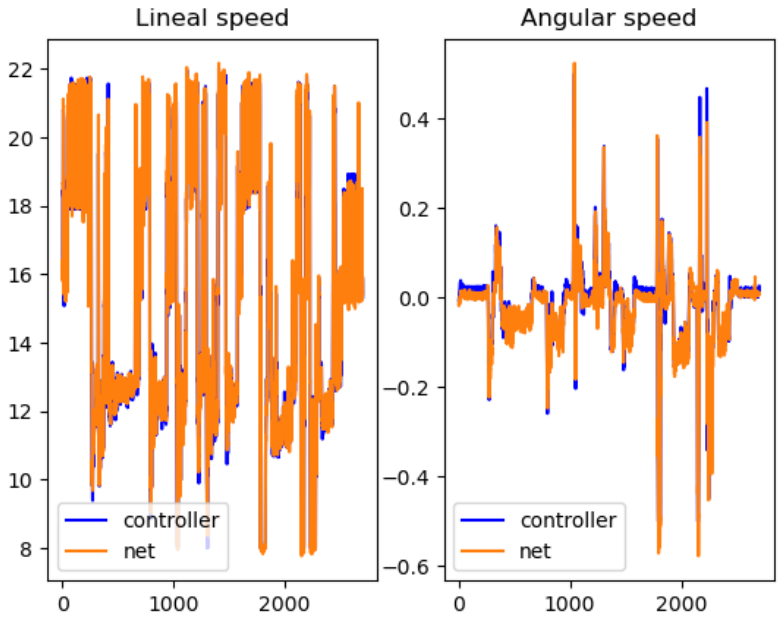



**Figura 5.1:** Evaluación *offline* de la red monolítica.

### 5.2.2. Mezcla de especialistas

En contraste, la mezcla de especialistas presenta un MSE significativamente mayor: 1,1694 para la velocidad lineal y 0,00261 para la angular. Si bien sigue capturando tendencias generales del comportamiento del experto, muestra mayor variabilidad y desviaciones más marcadas, especialmente en la velocidad lineal.

La Figura 5.2 ilustra estas diferencias: la señal generada por la mezcla de especialistas (naranja) reproduce en líneas generales la forma de la salida del experto (azul), pero con oscilaciones más notables y ciertos desajustes temporales.

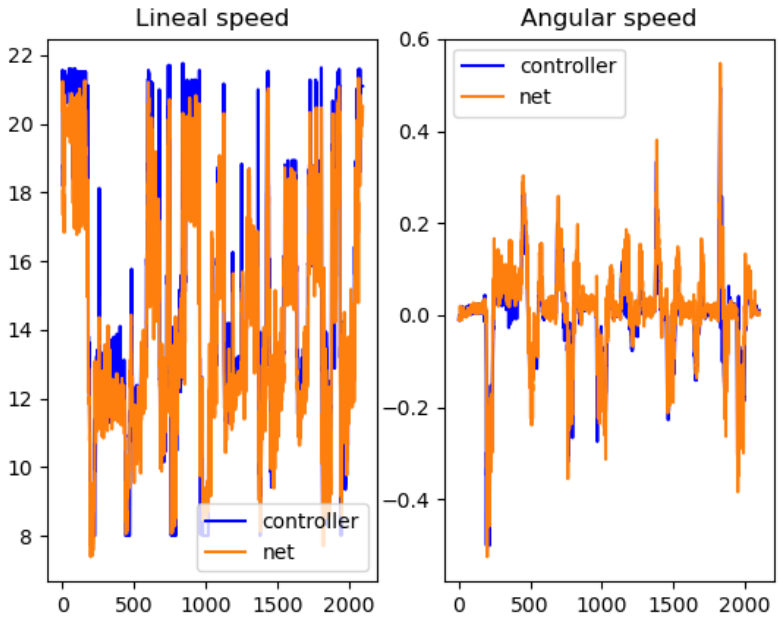



**Figura 5.2:** Evaluación *offline* de la mezcla de especialistas.

Los resultados indican que, en términos de aproximación directa al comportamiento del controlador, la red monolítica supera claramente a la mezcla de especialistas en el entorno *offline*.

No obstante, este análisis no contempla aspectos como la robustez frente a perturbaciones o el desempeño en ejecución real, donde la arquitectura de mezcla podría beneficiarse de una mayor capacidad de adaptación en distintos contextos. Estos aspectos se explorarán en las evaluaciones *online*.

## 5.3. Evaluación *online*

Para todos los circuitos se han tomado datos de 5 vueltas y se han procesado como ha sido explicado en la introducción a este Capítulo 5.

### 5.3.1. Tiempos

Dado el peso de las redes y los mapas el RTF no se mantenía de forma constante en 1. Siendo el RTF el factor que relaciona el tiempo de simulación con el tiempo real, si el RTF es 1 cada segundo de reloj equivale a un segundo en la simulación,

pero si este bajase a 0,5 harían falta dos segundos de reloj para que avanzase uno la simulación.

Esto implica que el tiempo de la simulación no se correspondía al tiempo real y, por lo tanto, un simple cronómetro no nos dice cuánto tardan los pilotos en completar los circuitos. Además, dado que el RTF varía, tampoco es el tiempo real una comparativa precisa entre ellos. Para esto se ha tomado como tiempo en los CSVs el *simulation time* o tiempo de simulación que proporciona Gazebo teniendo en cuenta el RTF.

También se ha de tener en cuenta que el tiempo que tardan en lanzarse los procesos puede variar así como el tiempo que tardan los procesos en cargar las redes y empezar a pilotar. Para ello se ha tomado empezado a tomar el tiempo cuando la primera consigna de velocidades se ha dado, y posteriormente se han preprocesado los *datasets* para hacer de este el tiempo 0.

Dividiendo los datos en vueltas como ha sido explicado y comparando el tiempo de simulación al inicio y al final de cada una se puede obtener el *tiempo medio por vuelta* de cada uno de los pilotos en cada uno de los circuitos, constando con los resultados cuantitativos siguientes:

| **Circuito** | **Experto (s)** | **Monolítico (s)** | **Mezcla de especialistas (s)** |
|---|---|---|---|
| Simple | 162.259 | 175.401 | 170.390 |
| Montmeló | 271.763 | 287.682 | 282.142 |
| Nürburgring | 202.172 | 221.127 | 212.973 |

**Tabla 5.1:** Tiempos en segundos por circuito y piloto.

Como podemos apreciar en las Tablas 5.2 y 5.1 el coche pilotado por la mezcla de expertos es aproximadamente un 3 % más rápido que el pilotado por la red monolítica.

| **Circuito** | **Experto** | **Monolítico** | **Mezcla de especialistas** |
|---|---|---|---|
| Simple | 100 % | 92.49 % | 95.22 % |
| Montmeló | 100 % | 94.47 % | 96.32 % |
| Nürburgring | 100 % | 91.43 % | 94.94 % |

**Tabla 5.2:** Porcentaje de velocidad respecto al experto (100 %) por circuito y arquitectura.

### 5.3.2. Métricas de error espacial

Calculando una distancia euclídea desde la posición del coche obtenida a la posición en el mundo de la línea roja (fija e igual para todos los pilotos) es posible obtener la distancia en metros a la línea en cada momento.

Para poder representar esta distancia en una gráfica donde se muestren los resultados de los tres pilotos, y así facilitar la comparación, se ha establecido como eje vertical la distancia en metros absoluta a la línea. Para establecer una referencia común se ha calculado mediante la misma distancia euclídea la diferencia entre posiciones para generar una longitud en metros del circuito y así tener una referencia común para todos los pilotos.

Las gráficas obtenidas son las siguientes:

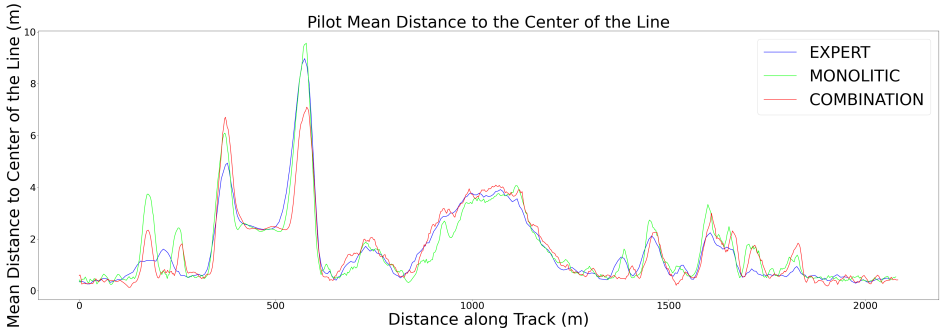



**Figura 5.3:** Distancia en metros a la línea roja en el circuito Simple.

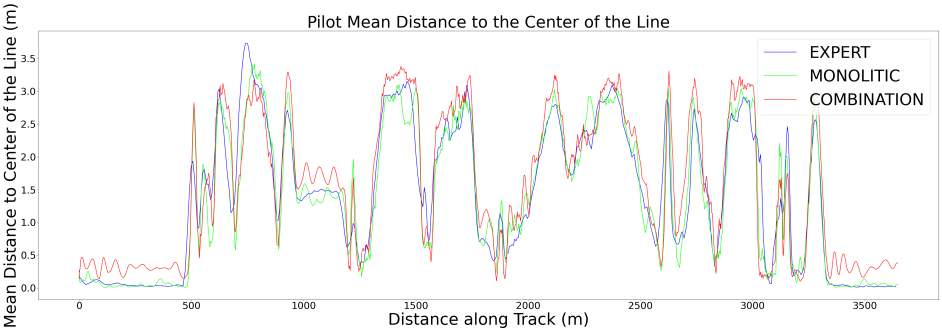



**Figura 5.4:** Distancia en metros a la línea roja en el circuito Montmeló.

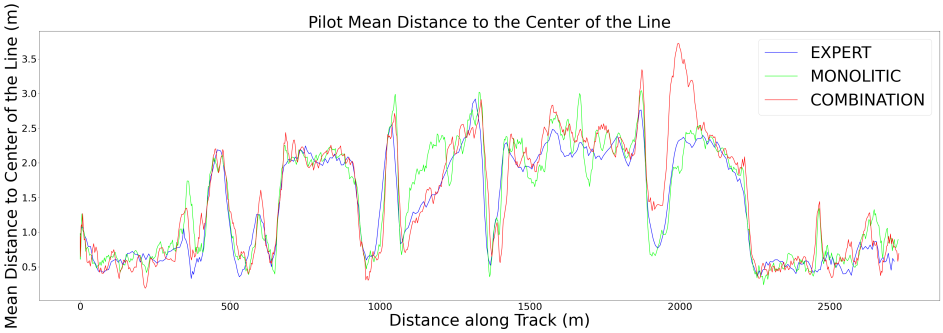



**Figura 5.5:** Distancia en metros a la línea roja en el circuito Nürburgring.

Podemos comprobar que los pilotajes conseguidos con las redes han sido buenos y las diferencias entre pilotos no son excesivamente grandes, aunque sí hay algunos puntos donde se aprecian diferencias. Sin embargo, estos datos se pueden resumir en un número gracias al error cuadrático medio a lo largo de los circuitos como puede observarse en la Tabla 5.3.

El análisis de este error nos permite concluir que la mezcla de especialistas es consistentemente mejor en adherirse a la línea que el monolítico. Además, observamos que en los circuitos observados durante el entrenamiento supera incluso al experto. Esto probablemente sea gracias al aumentado de datos.

| Circuito | Experto (MSE) | Monolítico (MSE) | Mezcla de especialistas (MSE) |
|---|---|---|---|
| Simple | 5.0053 | 5.2449 | 4.8693 |
| Montmeló | 8.4559 | 8.2883 | 8.2109 |
| Nürburgring | 2.4653 | 2.9369 | 2.7515 |

**Tabla 5.3:** Error cuadrático medio (MSE) por circuito y piloto.

### 5.3.3. Métricas de velocidad a lo largo del circuito

Debemos así mismo comparar las salidas que devuelven nuestras redes a las generadas por el piloto experto que ha generado los *datasets*. Para ello vamos a coger las medias de velocidades obtenidas durante las ejecuciones de los tres pilotos en los tres circuitos y establecer una comparación de manera similar a las distancias. Además, igual que en el apartado anterior, debemos concretar en un número, siendo en este caso el error cuadrático medio de las salidas de velocidades de los pilotos neuronales en relación a las generadas por el experto.

Para mayor simplicidad comparemos por separado las velocidades lineales y angulares.

### Velocidad lineal

En el caso de la velocidad lineal obtenemos los siguientes resultados de nuestro procesado de los CSVs:

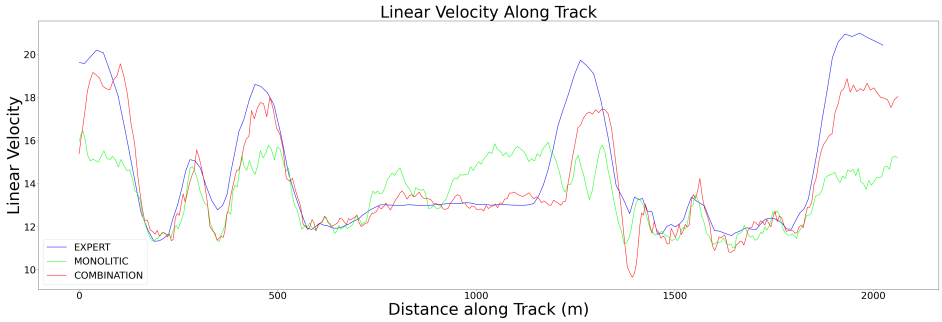



**Figura 5.6:** Velocidades lineales en el circuito Simple.

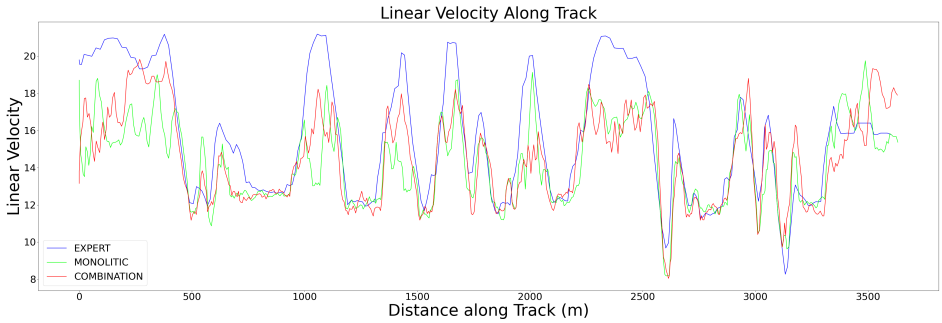



**Figura 5.7:** Velocidades lineales en el circuito Montmeló.

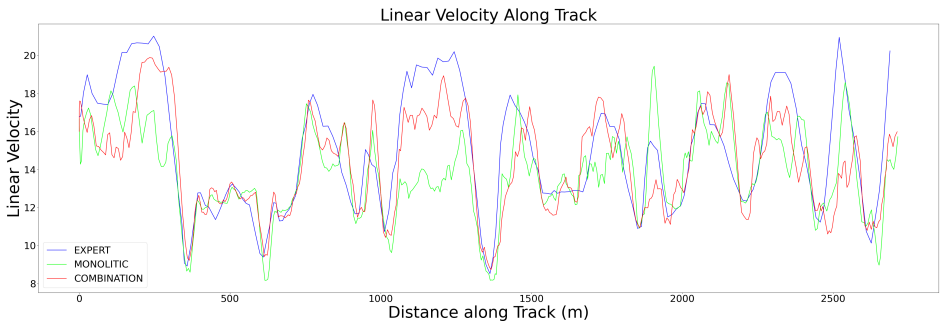



**Figura 5.8:** Velocidades lineales en el circuito Nürburgring.

| Circuito | Monolítico (MSE) | Mezcla de especialistas (MSE) |
|---|---|---|
| Simple | 10.22895 | 12.53712 |
| Montmeló | 13.77317 | 13.39087 |
| Nürburgring | 14.07739 | 16.10997 |

**Tabla 5.4:** MSE de velocidad lineal por circuito.

### Velocidad angular

En el caso de la velocidad angular obtenemos los siguientes resultados de nuestro procesado de los CSVs:

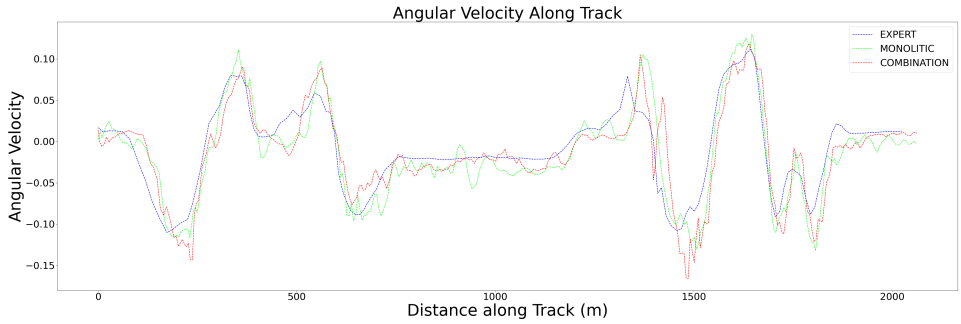



**Figura 5.9:** Velocidades angulares en el circuito Simple.

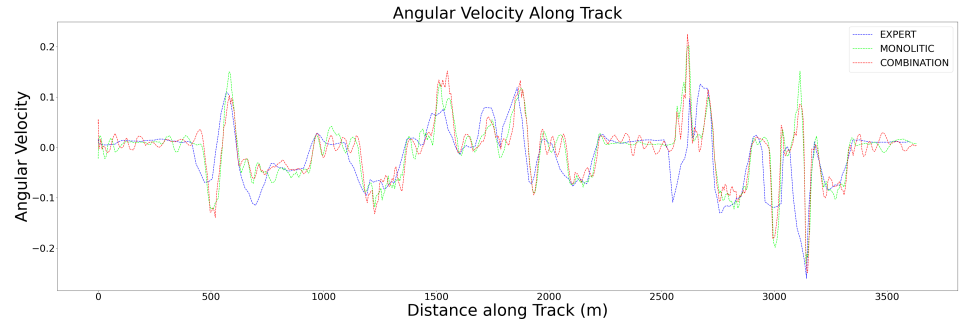



**Figura 5.10:** Velocidades angulares en el circuito Montmeló.

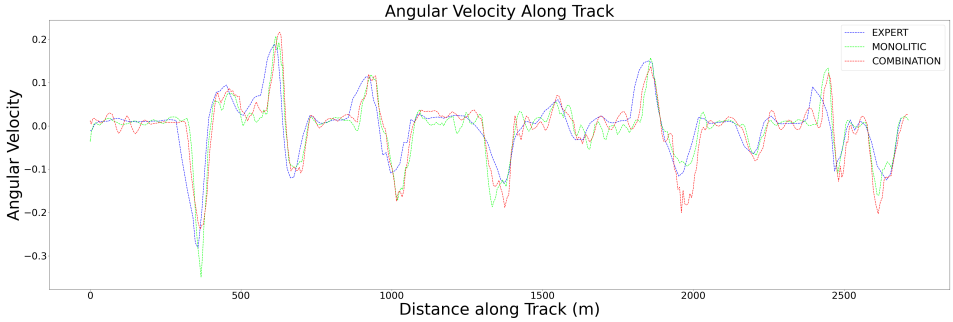



**Figura 5.11:** Velocidades angulares en el circuito Nürburgring.

| Circuito | Monolítico (MSE) | Mezcla de especialistas (MSE) |
|---|---|---|
| Simple | 0.00660 | 0.00699 |
| Montmeló | 0.00823 | 0.00763 |
| Nürburgring | 0.01057 | 0.01120 |

**Tabla 5.5:** MSE de velocidad angular por circuito.

Observando todos los datos de velocidades obtenidos no parece que haya una clara superioridad por parte de ninguna de los pilotos neuronales en su similitud al experto. Podemos comprobar que en las velocidades angulares se parecen ambas redes mucho al experto, pero en las lineares se diferencian considerablemente.

## 5.4. Discusión

Si prestamos atención detallada a las gráficas de velocidades podemos apreciar una tendencia del coche pilotado por la mezcla de especialistas a mantener velocidades más altas en los puntos altos que el pilotado por la red monolítica, mientras que en los puntos bajos también suele bajar más. Esto probablemente se debe a que la red monolítica, al estar entrenadas con todos los datos, da unas velocidades más promediadas en todo momento, mientras que la mezcla de especialistas en rectas acelera más y en curvas reduce más la velocidad.

Siendo este precisamente el resultado buscado y observando el efecto que esto ha tenido en los tiempos medios por vuelta y distancias a la línea de los pilotos; podemos decir que la aproximación por especialistas es cuantitativamente superior a la monolítica.

# 6. Conclusiones

## 6.1. Cumplimiento de objetivos

A lo largo del trabajo presentado se ha logrado:

- Desarrollar un piloto experto satisfactorio, capaz de recorrer todos los circuitos de manera robusta, así como crear unos conjuntos de datos balanceados para el entrenamiento de redes neuronales a través del uso de herramientas y conocimientos de ROS2 y visión artificial.
- Generar satisfactoriamente un piloto monolítico que fuese capaz de recorrer todos los circuitos de forma consistente empleando una red extremo a extremo con la arquitectura PilotNet y un conjunto de datos aumentado con técnicas como *affine*.
- Generar un piloto que mezcle dos redes especialistas y las coordine para completar los circuitos requeridos mediante un clasificador visual de rectas y curvas implementado de forma explícita y probada su validez.
- Validar de forma experimental todos los pilotos, así como desarrollar métricas cuantitativas y objetivas para comparar los pilotos entre sí.

Además, mediante la comparación de los pilotos neuronales generados, se puede concluir que la mezcla de expertos es una aproximación superior al piloto monolítico en su capacidad de aprender a través de los mismos datos a recorrer circuitos complejos siguiendo una línea de modo ligeramente más preciso y rápido. Esto puede verse en los resultados del Capítulo 5, en especial en los tiempos por vuelta.

Por todo ello se puede concluir que se han cumplido satisfactoriamente todos los objetivos planteados al principio de esta memoria.

## 6.2. Competencias empleadas

A lo largo del desarrollo del presente proyecto se han puesto en práctica diversas competencias tanto técnicas como transversales adquiridas a lo largo del

grado. Estas competencias han sido fundamentales para el diseño, implementación y validación del sistema de conducción autónoma basado en visión y aprendizaje profundo. A continuación se detallan las más relevantes:

- **Programación en Python:** Se ha utilizado Python como lenguaje para el desarrollo tanto del sistema ROS2 como del modelo de red neuronal, empleando herramientas como OpenCV, NumPy, y PyTorch de forma intensiva. Este lenguaje ha sido transversal en el grado, siendo introducido en la primera asignatura de programación "Introducción a la programación" y habiendo estado presente hasta cuarto en "Robótica de Servicios".
- **Desarrollo sobre ROS2:** Se han implementado nodos ROS2 personalizados para realizar tareas de percepción, control, publicación de datos y sincronización temporal. Se han empleado conceptos como calidad de servicio (QoS), temporizadores y suscripciones simultáneas, mostrando un dominio de esta tecnología moderna de robótica. Como no podía ser de otra manera ROS2 ha estado presente en el grado de Ingeniería Robótica Software desde segundo con su introducción en "Arquitecturas Software para Robots" y se ha mantenido presente desde entonces en una gran variedad de asignaturas.
- **Visión artificial:** Se han desarrollado técnicas de preprocesado de imagen como filtros de color, segmentación y trazado de líneas guía. Estas competencias fueron adquiridas en la asignatura de tercero "Visión Artificial".
- **Control automático:** Implementación y ajuste de controladores PID para la regulación de la velocidad lineal y angular del vehículo, en función de la percepción visual de la trayectoria. Si bien el control de robots se ha empleado en muchas asignaturas, cabe destacar "Ingeniería de Control" a la hora del diseño de controladores y "Robótica Móvil" en el control de robots en simulador.
- **Simulación de sistemas autónomos:** Uso de entornos de simulación compatibles con ROS2 para la validación del comportamiento del vehículo en situaciones controladas, permitiendo iterar de forma rápida y segura sobre el diseño del sistema. Los simuladores, y en concreto Gazebo, introducidos por primera vez en "Arquitecturas Software para Robots" han sido también una parte fundamental del grado, cabiendo destacar "Modelado y Simulación de Robots" como la asignatura que más competencias ha aportado a la hora de manipular los modelos del coche y mundos para adaptarlos a las necesidades

del trabajo.

- **Integración de sistemas complejos:** Coordinación de múltiples subsistemas (percepción, control, simulación, entrenamiento) dentro de un flujo de trabajo coherente y funcional, respetando restricciones de tiempo real. Estas competencias han sido adquiridas a lo largo de la carrera, pero de especial influencia en el trabajo desarrollado ha sido "Robótica de Servicios", cuyas prácticas son precursoras de este TFG.

## 6.3. Competencias adquiridas

Además de aplicar conocimientos previamente adquiridos, este proyecto ha permitido desarrollar nuevas competencias clave, entre las cuales destacan:

- **Entrenamiento de modelos de aprendizaje profundo:** Se ha trabajado con redes neuronales convolucionales, aplicando técnicas como *data augmentation*, recorte, normalización, partición del conjunto de datos y evaluación de la pérdida.
- **Análisis crítico y toma de decisiones técnicas:** Capacidad para evaluar alternativas (por ejemplo, diferentes técnicas de preprocesamiento o arquitecturas de red) y justificar técnicamente las decisiones adoptadas.
- **Interpretación del comportamiento de modelos de IA:** Evaluación de predicciones erróneas, identificación de fallos comunes y comprensión del impacto de los datos de entrenamiento sobre el desempeño final.
- **Trabajo con flujos asíncronos y temporizados:** Gestión de entradas y salidas de sensores con diferentes frecuencias de muestreo y sincronización en tiempo real en un sistema basado en ROS2.
- **Autonomía y aprendizaje autodidacta:** Adquisición de conocimientos específicos sobre librerías, herramientas y enfoques que no habían sido tratados directamente en asignaturas previas. Como *flip* de OpenCV o la librería Albumentations.
- **Gestión del proyecto y documentación técnica:** Planificación del flujo de trabajo, organización del código y redacción de documentación formal, incluyendo la memoria técnica con explicaciones detalladas, figuras y referencias.

## 6.4. Futuras líneas de desarrollo

Si bien aquí acaba este trabajo, deben ser mencionadas ciertas mejoras posibles y futuras líneas de trabajo. En primer lugar, las diferencias apreciadas entre ambos pilotos no parecen ser exageradas; por ello, se plantea como futura línea de trabajo repetir los pasos seguidos con un experto llevado al máximo, capaz de sacar el máximo partido a las características del coche y el entorno, pues, cuanto más rápido sea, mayores deberán ser las diferencias entre pilotos. Para ello, podría ser beneficioso buscar mejoras tanto en el *hardware* como en el *software* para aumentar la cantidad y calidad de los datos utilizados.

Es otra línea de desarrollo interesante la de aumentar la complejidad de la situación, avanzando a simuladores más realistas como CARLA para afrontar tareas de mayor nivel de dificultad, observando si, a medida que ésta aumenta, también lo hace la diferencia entre pilotos neuronales.

# Bibliografía


[1] Roomba. iRobot Roomba i1. URL https://www.irobot.es/es_ES/irobot-roomba-i1/I155640.html. Acceso: 23 de junio de 2024.

[2] Intuitive Surgical. da Vinci Surgical System, Acceso: 24 de junio de 2024. URL https://www.intuitive.com/en-us/products-and-services/da-vinci.

[3] Waymo LLC. Waymo, Acceso: 24 de junio de 2024. URL https://waymo.com/intl/es/.

[4] Enrique Y. Shinohara Soto. Conducción autónoma en tráfico usando aprendizaje profundo extremo a extremo, 2023. Máster Universitario en Visión Artificial.

[5] Open Robotics. ROS - Robot Operating System, 2024. URL https://www.ros.org/. Acceso: 23 de junio de 2024.

[6] Cyberbotics Ltd. Cyberbotics - webots: Robot simulator, 2024. URL https://cyberbotics.com/. Acceso: 23 de junio de 2024.

[7] Open Robotics. Gazebo - robot simulation made easy, 2024. URL https://gazebosim.org/home. Acceso: 23 de junio de 2024.

[8] CARLA Team. Carla: Open-source simulator for autonomous driving research, 2024. URL https://carla.org/. Acceso: 23 de junio de 2024.

[9] N. H. T. S. Administration, U.S. Department of Transportation. Critical reasons for crashes investigated in the national motor vehicle crash causation survey, 2015.

[10] Hao Shao, Letian Wang, Ruobing Chen, Steven L. Waslander, Hongsheng Li, and Yu Liu. Reasonnet: End-to-end driving with temporal and global reasoning. *arXiv preprint arXiv:2305.10507*, 2023. doi: 10.48550/arXiv.2305.10507. URL https://arxiv.org/abs/2305.10507. CVPR 2023.

[11] Felipe Codevilla, Eder Santana, Antonio M. López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. *arXiv preprint arXiv:1904.08980*, 2019. URL https://arxiv.org/abs/1904.08980.

[12] Li Chen, Penghao Wu, Kashyap Chitta, Bernhard Jaeger, Andreas Geiger, and Hongyang Li. End-to-end autonomous driving: Challenges and frontiers. *arXiv preprint arXiv:2306.16927*, 2023. URL https://arxiv.org/abs/2306.16927.

[13] Jelena Kocic, Nenad Jovićić, and Vujo Drndarević. An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. *Sensors*, 19(9), 2019. doi: 10.3390/s19092064. URL https://www.mdpi.com/1424-8220/19/9/2064.

[14] Sergio Paniego Blanco. *End-to-end vision-based autonomous driving using deep learning*. Tesis Doctoral, Universidad Rey Juan Carlos, 2024. Programa de Doctorado en Tecnologías de la Información y las Comunicaciones.

[15] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. URL https://arxiv.org/abs/1604.07316.

[16] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint*, arXiv:1805.01954, 2018. URL https://arxiv.org/abs/1805.01954.

[17] Christopher Diehl, Janis Adamek, Martin Krüger, Frank Hoffmann, and Torsten Bertram. Differentiable constrained imitation learning for robot motion planning and control. *arXiv preprint arXiv:2210.11796*, 2023. URL https://arxiv.org/abs/2210.11796.

[18] Yue Ying. Mastering first-person shooter game with imitation learning. In *2nd International Conference on Networking, Communications and Information Technology (NetCIT)*, pages 587–591, 2022. doi: 10.1109/NetCIT57419.2022.00139.

[19] Derek Yadgaroff, Alessandro Sestini, Konrad Tollmar, Ayca Ozcelikkale, and Linus Gisslén. Improving generalization in game agents with data augmentation in imitation learning. *arXiv preprint arXiv:2309.12815*, 2023. URL https://doi.org/10.48550/arXiv.2309.12815.

[20] Alexander Buslaev, Alex Parinov, Eugene Khvedchenya, Vladimir I. Iglovikov, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2):125, 2020. doi: 10.3390/info11020125. URL https://doi.org/10.3390/info11020125.

[21] Felipe Codevilla, Matthias Müller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4693–4700. IEEE, 2018. doi: 10.1109/ICRA.2018.8460487. URL https://doi.org/10.1109/ICRA.2018.8460487.

[22] Python Software Foundation. Python language reference, version 3.8, 2021. URL https://www.python.org/. Acceso: 1 de julio de 2024.

[23] PyTorch Contributors. Pytorch, 2024. URL https://pytorch.org/. Acceso: 1 de julio de 2024.

[24] OpenCV Developers. Opencv: Open source computer vision library, 2024. URL https://opencv.org/. Acceso: 1 de julio de 2024.

[25] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[26] Alejandro Moncalvillo González. Seguimiento de carril por visión y conducción autónoma con aprendizaje por imitación, 2024. Grado en Ingeniería Robótica de Software.

[27] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. *arXiv preprint arXiv:1011.0686*, 2011. URL https://arxiv.org/abs/1011.0686.

[28] Vanessa Fernández Martínez. Conducción autónoma de un vehículo en simulador mediante aprendizaje extremo a extremo basado en visión, 2019. Máster Universitario en Visión Artificial.

[29] Adrián Madinabeitia Portanova. Programación de drones con aprendizaje por imitación y redes neuronales, 2024. Grado en Ingeniería Robótica de Software.