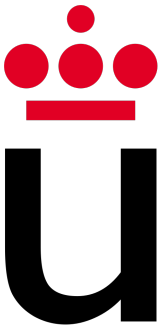


Universidad
Rey Juan Carlos

TRABAJO FIN DE GRADO

Seguimiento de carril por visión y conducción autónoma con Aprendizaje por Imitación

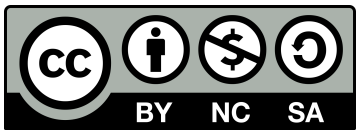
Grado en Ingeniería Robótica de Software

Escuela de Ingeniería de Fuenlabrada

Realizado por
Alejandro Moncalvillo González

Dirigido por
José María Cañas

Curso académico 2023/2024



Agradecimientos

En primer lugar quiero agradecer a mi tutor del TFG, Jose María, todo el apoyo dado durante el desarrollo del trabajo y la paciencia que ha tenido conmigo.

A mis amigos, que me han ayudado a desconectar, reír, y disfrutar para tener energías y seguir trabajando.

A mis padres y a mi hermana, que me han ayudado a persistir y continuar con el trabajo aún en los momentos más difíciles.

The Road goes ever on and on,
down from the door where it began.
Now far head the Road has gone,
and I must follow, if I can.
J. R. R. Tolkien

Resumen

La robótica es un sector que ha evolucionado enormemente en los últimos años, extendiendo su uso más allá del ámbito industrial hacia tareas en entornos complejos y no estructurados. Este desarrollo es debido a mejoras en sensores, actuadores y unidades de procesamiento, sumado al uso de algoritmos modernos y de técnicas de aprendizaje automático que permiten a los robots adaptarse para coexistir con los humanos y realizar tareas cada vez más útiles.

Algunas de estas nuevas áreas de la robótica son los robots de logística o la conducción autónoma. Los vehículos autónomos son sistemas capaces de imitar las capacidades humanas de manejo y control, percibir el medio que le rodea y navegar en consecuencia. Estudios contemporáneos han investigado el uso de la percepción basada en la visión en el campo de la conducción autónoma tanto haciendo uso de arquitecturas modulares[1], como de extremo a extremo [2].

El objetivo de este trabajo es demostrar la viabilidad de las soluciones de control de extremo a extremo basadas en visión para tareas relacionadas con la conducción autónoma. Para ello hemos creado un total de 3 aplicaciones que se encargan de resolver los siguientes problemas: sigue línea en un circuito simulado; sigue carril en un circuito simulado; y sigue carril en un escenario real. Estas aplicaciones tienen como controladores modelos neuronales programados indirectamente mediante la técnica de aprendizaje automático denominada *aprendizaje por imitación*.

Conceptos clave: robótica, aprendizaje automático, inteligencia artificial, extremo a extremo, redes neuronales, simulaciones, aprendizaje por imitación.

Acrónimos

CNN - Convolutional Neural Network
CPU - Central Processing Unit
CSV - Comma Separated Values
CUDA - Compute Unified Device Architecture
FPGA - Field Programable Gate Array
GPU - Graphics Processing Unit
IA - Inteligencia Atrificial
LSTM - Long Short-Term Memory
PID - Proportional–Integral–Derivative
RAM - Random Access Memory
ROS 2 - Robot Operating System 2
TFG - Trabajo de Fin de Grado

Índice general

Índice de figuras

Índice de extractos de código

1. Introducción

En este capítulo se introducirá el contexto en el que este TFG ha sido desarrollado. Para ello es necesario proporcionar definiciones y ejemplos del del estado del arte de los campos que tienen conexión con el trabajo realizado: la robótica, la inteligencia artificial, y la conducción autónoma.

1.1. Robótica

La robótica es una rama interdisciplinaria de la ingeniería que se dedica al diseño, construcción y programación de robots. Combina partes de ingeniería mecánica, ingeniería eléctrica, informática y otras disciplinas para crear robots. Los robots son sistemas autónomos capaces de realizar tareas que requieren inteligencia y agilidad humanas. Todos los sistemas robóticos tienen tres componentes fundamentales: sensores (cámaras, ultrasonidos, láser, etc), actuadores (principalmente motores), y procesadores (CPUs, GPUs, FPGAs, etc.). Estos últimos son los encargados de analizar la información proporcionada por los sensores, y enviar órdenes a los actuadores para llevar a cabo una tarea determinada.

1.1.1. Historia

Las siguientes subsecciones muestran una breve historia de la robótica.

Orígenes

El concepto de autómata se remonta a las civilizaciones antiguas. En la antigua Grecia, el matemático Arquitas de Tarento diseñó un pájaro mecánico propulsado por vapor (ver Figura 1.1).

La Edad Media y el Renacimiento

En los siglos siguientes, los autómatas que se desarrollaron consistían en mecanismos sofisticados y meticulosos que podían realizar tareas más

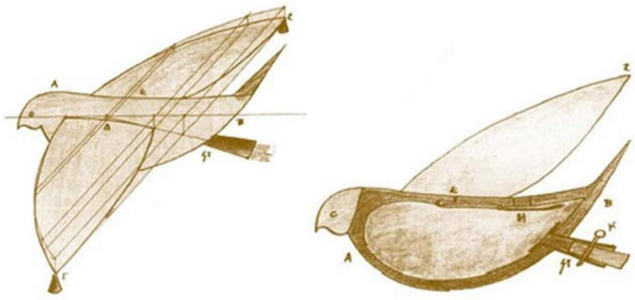


Figura 1.1: Paloma a vapor de Arquitas de Tarento.

complicadas. En el siglo XIII, el libro .El libro del conocimiento de ingeniosos dispositivos mecánicos", escrito por el inventor musulmán Ismail Al-Jazari, muestra muchos dispositivos automáticos como un autómata para lavarse las manos (ver Figura 1.2).

Figura 1.2: Autómata de Al-Jazari.

Siglos XVIII a XX

La Revolución Industrial (siglos XVIII y XIX) supuso un salto significativo en el desarrollo de la robótica. El uso de la energía del vapor y la fabricación mecanizada hicieron posible múltiples innovaciones en la maquinaria automatizada. En particular, en 1801, Joseph Marie Jacquard inventó el telar Jacquard (ver Figura 1.3), que utilizaba tarjetas perforadas para controlar el tejido de patrones

complejos. Esta máquina programable se considera el origen de las computadoras y sistemas automáticos modernos.

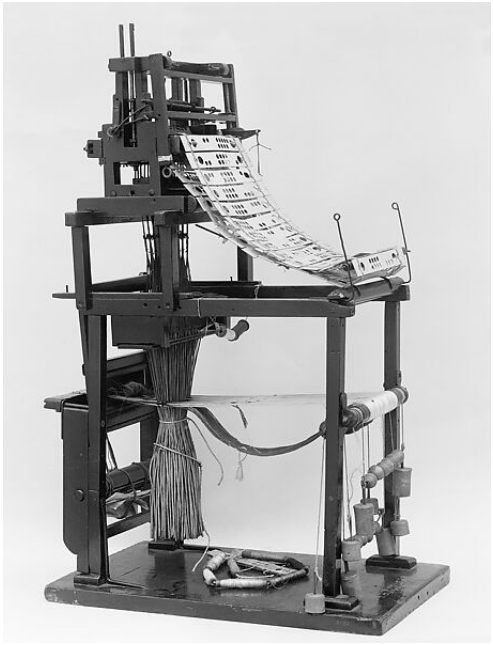


Figura 1.3: Telar Jacquard.

El siglo XX fue testigo de rápidos avances en robótica, impulsados por las necesidades de la fabricación industrial, las guerras, y la exploración espacial. El término *robot* fue acuñado por el escritor checo Karel Čapek en su obra de 1920 R.U.R. (Rossum's Universal Robots)", en la que se representan como seres sintéticos al servicio de los humanos. Durante la Segunda Guerra Mundial, los avances en electrónica y computación llevaron a la creación de una de las primeras máquinas eléctricas programables por parte de Alan Turing y su equipo de investigación (ver Figura 1.4). En la segunda mitad del siglo XX se inició la comercialización de robots industriales. En 1961, el Unimate (ver Figura 1.5), el primer robot industrial, se instaló en una fábrica de General Motors para realizar tareas repetitivas y peligrosas.

1.1.2. La Robótica del siglo XXI

El siglo XXI se ha caracterizado por avances significativos en inteligencia artificial (IA) y aprendizaje automático, que han mejorado las capacidades de los robots y han redefinido lo que son capaces de lograr. Los robots modernos pueden aprender de su entorno, adaptarse a nuevas situaciones y realizar tareas complejas con gran precisión.

Figura 1.4: Máquina Enigma.

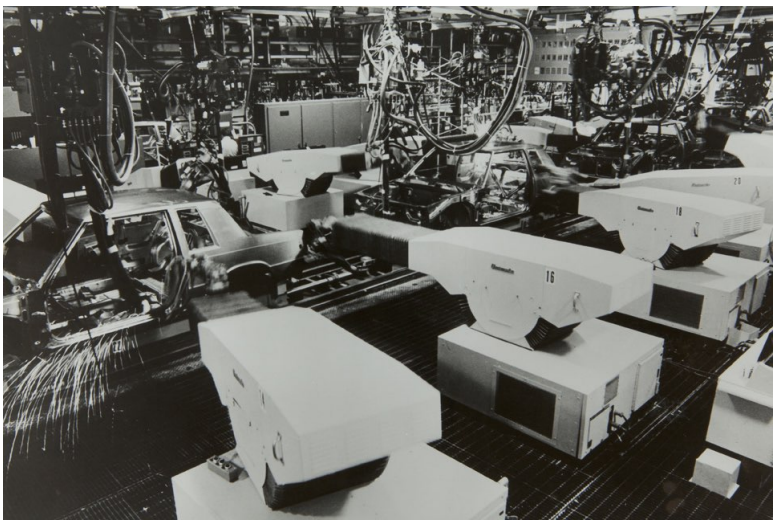


Figura 1.5: Robots Unimate trabajando en una fábrica de automóviles.

1.1.3. Estado del arte

En los últimos años, la autonomía y la capacidad de decisión de los robots se ha incrementado enormemente gracias a técnicas de IA, como el uso de redes neuronales. Algunos de los sectores de la robótica que más han crecido son:

- **Robots para Logística:** el objetivo de estos robots es optimizan las operaciones de transporte de objetos entre ubicaciones, como por ejemplo en un almacén (ver Figura 1.6). Estos sistemas tienen sensores (como un láser o cámaras) y sistemas de planificación que les permiten navegar ente un punto hasta otro de manera eficiente, mejorando los tiempos de entrega y reduciendo los errores humanos.

Figura 1.6: Robots de logística en un almacén de Amazon

- **Robots usados en Medicina:** este tipo de robots ayudan o mejoran procesos médicos como la detección de enfermedades, las intervenciones quirúrgicas, o el uso de prótesis. Un dispositivo de este grupo bastante conocido es el *da Vinci* (ver Figura 1.7), se trata de un robot teleoperado que hace posible procedimientos de cirugía menos invasivos y más precisos.

Figura 1.7: Robot médico da Vinci

- **Robots de Limpieza:** estos robots se encargan de la limpieza de diversos entornos como: hogares, centros comerciales, espacios públicos, etc. Incorporan sensores para detectar suciedad y obstáculos, junto con algoritmos para planificar rutas de limpieza eficientes. Un ejemplo notable es Roomba (ver Figura 1.8), un robot de aspiración autónomo de la empresa IRobot, que comenzó su desarrollo en 1989 pero no se introdujo en el mercado hasta 2002. Modelos más recientes de este robot han incorporado técnicas de IA para el reconocimiento de objetos.

Figura 1.8: Robot de limpieza Roomba

- **Conducción Autónoma:** otro campo con grandes avances es la conducción autónoma, que se trata más en detalle en la sección 1.3.

1.1.4. Desarrollo de aplicaciones robóticas

Las aplicaciones robóticas han ido ganando complejidad para cubrir casos de uso más avanzados. El procesamiento suele estar distribuido en varios componentes que se ejecutan a diferentes frecuencias y requieren distinta información para su funcionamiento. Las aplicaciones robóticas deben combinar respuestas rápidas para la interacción con el mundo real con situaciones espontáneas junto con algoritmos de planificación a largo plazo. Para facilitar el desarrollo de software robótico se han creado herramientas como:

Middlewares robóticos

Son herramientas de software que se encargan de la abstracción del hardware, gestión de comunicaciones o integración con diversos simuladores y hacen posible programar aplicaciones de alto nivel con más facilidad. El *middleware* más extendido en investigaciones robóticas es ROS 2, explicado en profundidad en la sección 3.5. ROS 2 permite la programación de aplicaciones como un conjunto de nodos que realizan distintas tareas y se comunican entre ellos.

Simuladores

Permiten depurar y verificar las aplicaciones robóticas en entornos físicamente realistas, proporcionando herramientas para el modelado de robots, sensores y actuadores. Para ello, es necesario replicar condiciones físicas como: la fricción, la gravedad, la colisión de objetos, la iluminación, etc. Además, los simuladores ofrecen la capacidad de probar nuevos algoritmos sin la necesidad de hacer ensayos con sistemas reales, acelerando el proceso de desarrollo y la seguridad. Uno de los más utilizados es Gazebo (sección 3.6), una simulación de ejemplo se puede apreciar en la Figura 1.9.

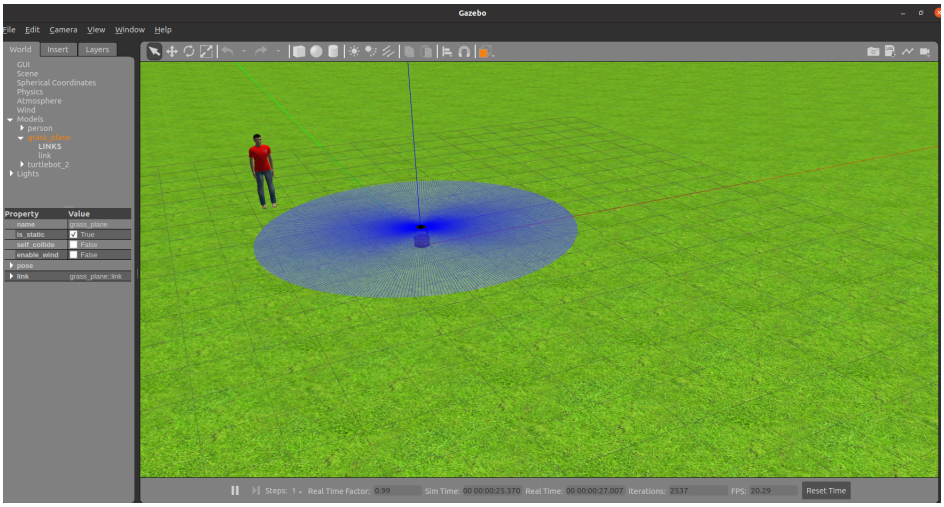


Figura 1.9: Simulación de Gazebo con una persona y un *turtlebot 2*.

1.1.5. Tipos de elementos en las aplicaciones robóticas

Los distintos componentes que se emplean para crear una aplicación robótica tienen funciones diversas que se adaptan mejor a unos casos u otros:

- **Componentes reactivos:** proporcionan una respuesta inmediata a los cambios del entorno. Operan sobre el principio de estímulo-respuesta, permitiendo al robot actuar de manera rápida, utilizando un algoritmo ligero, ante situaciones imprevistas, sin necesidad de una planificación previa. Estos componentes se encargan de tareas que requieren tiempos de respuesta breves, como evitar obstáculos o el seguimiento de una ruta preestablecida. Es por ello que se suelen ejecutar a frecuencias elevadas.
- **Componentes deliberativos:** están diseñados para llevar a cabo tareas complejas para las que se analizan de manera anticipada varios factores y

consecuencias potenciales. Permiten al robot tomar decisiones basadas en modelos internos del mundo. Se encargan de acciones que necesitan un análisis detallado y toma de decisiones estratégicas, como la navegación global o la manipulación de objetos conocidos. Estos componentes generalmente se ejecutan a una frecuencia menor que los reactivos, generando planes de actuación a largo plazo.

- **Componentes de gestión de la ejecución:** ofrecen un modelo para organizar y coordinar las acciones del robot. Existen varios tipos, siendo los dos más populares:
  - Máquinas de estado: permiten modelar la lógica de control en situaciones donde el comportamiento del robot puede clasificarse en un conjunto finito de estados, cuya transición está determinada por un conjunto de reglas claras y deterministas.
  - Árboles de Comportamiento: proporcionan una estructura para manejar decisiones y comportamientos más complejos, basada en la selección dinámica de tareas y acciones reutilizables.

Normalmente, una aplicación robótica se compone de los tres elementos: los deliberativos, para crear planes a largo plazo; los reactivos, para llevarlos a cabo; y los de gestión para servir de enlace entre los otros dos y decidir cuándo interviene cada uno.

1.2. Inteligencia Artificial

La Inteligencia Artificial (IA) es un campo de la informática centrado en la creación de modelos capaces de realizar tareas que normalmente requieren inteligencia humana. Estas tareas pueden ser: razonamiento, resolución de problemas, percepción de objetos o personas, análisis del lenguaje, etc.

1.2.1. Aprendizaje Automático

El aprendizaje automático (en inglés *Machine Learning*), es una rama de la inteligencia artificial, cuyo objetivo es desarrollar técnicas que permitan que los sistemas automáticos aprendan a desempeñar una o mas tareas. Se dice que un agente aprende cuando su desempeño mejora con la experiencia y mediante el uso de datos; es decir, cuando la habilidad no estaba presente en su genotipo o rasgos

de nacimiento. En el Aprendizaje Automático un sistema observa datos, construye un modelo basado en esos datos y utiliza ese modelo a la vez como una hipótesis acerca del mundo y una pieza de software que es capaz de resolver problemas. Existen varios tipos de de algoritmos de Aprendizaje Automático, cada uno de ellos se suele usar para diferentes tipos de tareas y estructuras de datos. Los principales son:

1. Aprendizaje Supervisado: en este enfoque, el sistema se entrena con un conjunto de datos etiquetados, lo que significa que cada punto de datos de entrada tiene una etiqueta de salida o de destino correspondiente. Es el método de entrenamiento más común para redes neuronales.
2. Aprendizaje No Supervisado: implica entrenar el sistema con datos sin etiquetas explícitas. El objetivo es identificar patrones, grupos o asociaciones con el objetivo de aprender la estructura o lógica que hay detrás de los datos.
3. Aprendizaje por Refuerzo: se entrena un sistema a través de múltiples interacciones en un entorno. El agente aprende a realizar tareas recibiendo una puntuación positiva o negativa, en función de sus acciones y una función de recompensa desconocida para el sistema. Se basa en el proceso de aprendizaje de prueba y error.

Para este trabajo optamos por utilizar el Aprendizaje por Imitación, que es un tipo de Aprendizaje Supervisado, donde el sistema aprende imitando el comportamiento de un agente externo. Este tipo de aprendizaje es particularmente útil en escenarios donde es difícil de definir una función de recompensa y tenemos múltiples datos de experiencias de un controlador experto (provenientes de una aplicación o un humano).

1.2.2. Redes Neuronales

Las redes neuronales son un componente importante de la IA. Están inspirados en la estructura y funcionalidad del cerebro humano, estando compuestas por nodos (neuronas) interconectados y organizados en capas. Las neuronas son las unidades básicas de una red neuronal, recibiendo información, procesándola y produciendo una salida. Cada neurona aplica una función (a menudo no lineal) a su entrada y pasa el resultado a una o más neuronas de la siguiente capa. Una red neuronal simple compuesta por una sola neurona se llama "Perceptrón"(ver Figura 1.10), que fue implementada físicamente y presentada por primera vez en 1958 en el artículo *The Perceptron: a probabilistic model for information storage and organization in the brain* ,

escrito por Frank Rosenblatt.

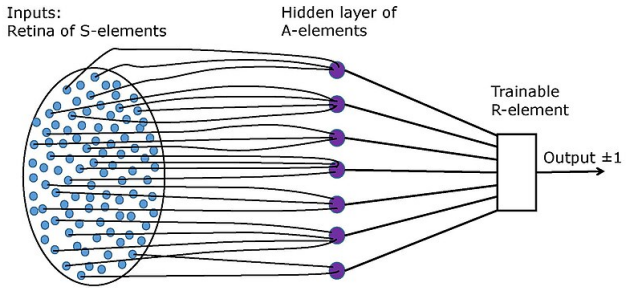


Figura 1.10: Estructura del Perceptrón.

Las redes neuronales modernas están compuestas por miles de neuronas que se dividen en múltiples capas interconectadas, podemos dividir su estructura (ver Figura 1.11) en:

1. Capa de entrada: recibe los datos iniciales (en nuestro caso imágenes).
2. Capas ocultas: capas intermedias donde se produce la transformación de datos y la extracción de características a través de conexiones ponderadas y funciones de activación.
3. Capa de salida: obtiene el valor de inferencia final basado en los datos procesados (en nuestro caso los valores para el acelerador y la dirección del automóvil).

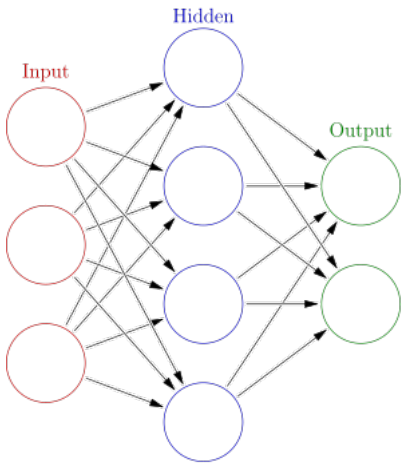


Figura 1.11: Estructura general de una red neuronal.

Además, las redes neuronales se pueden clasificar en dos tipos principales:

1. Redes neuronales Feedforward: en esta configuración de red la información se mueve solo en una dirección, de entrada a salida, sin ningún tipo de ciclos o bucles.

2. Redes neuronales recurrentes: diseñadas para datos secuenciales, como series de tiempo o procesamiento del lenguaje, este tipo de redes pueden ir hacia atrás, permitiendo que la salida de algunas neuronas impacte en la entrada de las anteriores.

Asimismo, existen múltiples subtipos más específicos que los dos mencionados anteriormente, como:

1. Redes neuronales de larga memoria a corto plazo (LSTM): es un tipo de red neuronal recurrente cuyo objetivo es proporcionar una memoria a corto plazo que pueda durar miles de pasos de tiempo, es decir, una "memoria larga a corto plazo". Suelen ser utilizadas en aplicaciones relacionadas con: la escritura, análisis de la conversación, traducción automática, videojuegos, etc.

2. Redes neuronales convolucionales (CNN): es un tipo regularizado de red neuronal Feedforward que se especializa en procesar estructuras de datos en forma de cuadrícula, como imágenes, mediante el uso de capas convolucionales para detectar patrones y características mediante una optimización del kernel que se usa. En la figura 1.12 se puede ver una estructura de red neuronal convolucional. Se pueden emplear para tareas de reconocimiento de imágenes, procesamiento de vídeos o clasificación de series de tiempo, entre otros. En este trabajo hemos utilizado una arquitectura perteneciente a este tipo de red.

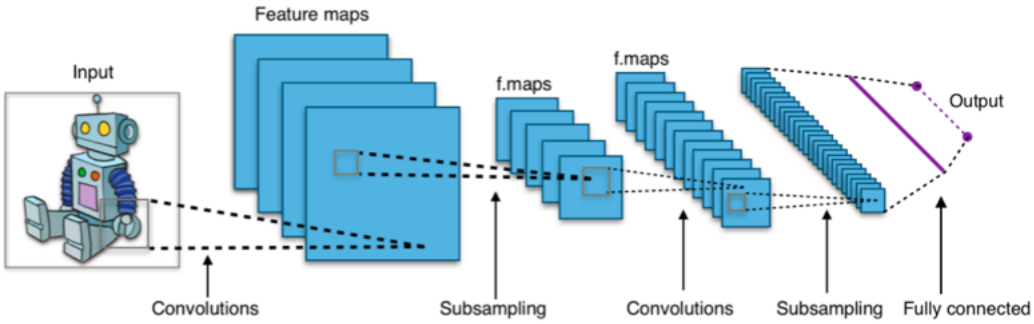


Figura 1.12: Estructura de una red neuronal convolucional.

3. Transformadores: es un tipo de red neuronal desarrolada por Google con una arquitectura más compleja (ver Figura 1.13), que se centra principalmente en detectar patrones del lenguaje. Un texto se convierte en una representación numérica que se analiza y contextualiza en un marco siguiendo un algorítmo de atención multi-nivel. Actualmente se usan en aplicaciones de LLM como los modelos GPT de OpenAI y Gemini de Google.

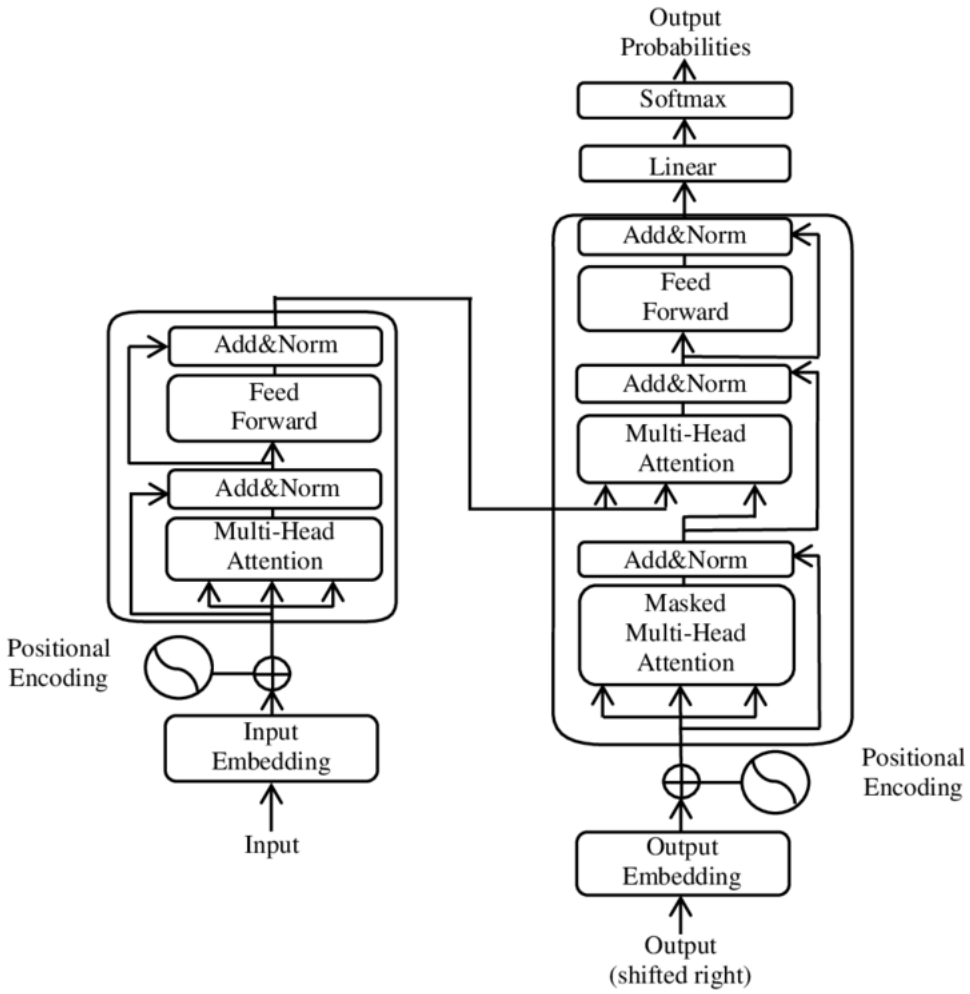


Figura 1.13: Arquitectura de un Transformador.

Para poder conseguir resultados que se adapten a la tarea que se quiere resolver es necesario entrenar un modelo. Entrenar una red neuronal implica ajustar los pesos (valores numéricos asociados a las conexiones entre neuronas en diferentes capas). Esto se hace mediante un proceso llamado retropropagación, que utiliza algoritmos de optimización, como el del descenso de gradiente, para minimizar el error actualizando iterativamente los pesos en cada capa. En este trabajo hemos utilizado el cálculo de pérdidas, donde la salida se compara con los valores objetivo reales utilizando una función de pérdida, para cuantificar el error.

1.3. Conducción autónoma

La conducción autónoma, que alguna vez fue un concepto de ciencia ficción, se ha convertido en un campo que avanza rápidamente y está preparado para

revolucionar los medios de transporte. El desarrollo de vehículos autónomos implica la integración de múltiples tecnologías, incluida la inteligencia artificial (IA), el aprendizaje automático, los sensores y la informática avanzada.

La visión de los vehículos autónomos se remonta a principios del siglo XX. En 1925, Houdina Radio Control Company mostró un automóvil controlado por radio que navegaba por las calles de la ciudad de Nueva York (ver Figura 1.14), mostrando el potencial de los vehículos operados a distancia. Sin embargo, los avances prácticos en la conducción autónoma estuvieron limitados hasta la última mitad del siglo debido a restricciones tecnológicas. La era moderna de la conducción autónoma comenzó en la década de 1980, impulsada en gran medida por la investigación científica y militar. La Agencia de Proyectos de Investigación Avanzada de Defensa (DARPA), inició varios proyectos para desarrollar vehículos terrestres autónomos, el punto de inflexión fueron los DARPA Grand Challenges (ver Figura 1.15). Estas competiciones, celebradas por primera vez en 2004, fomentaron la investigación y el desarrollo en conducción autónoma y en la edición de 2005 el equipo de la Universidad de Stanford completó el recorrido de 132 millas en un desierto. El siguiente DARPA Challenge, esta vez con el apellido *Urban*, en 2007, impulsó aún más la límites al requerir que los vehículos naveguen por entornos urbanos complejos.

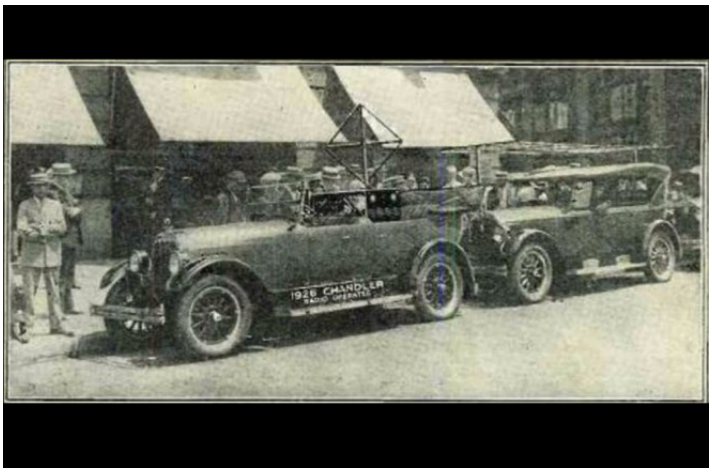


Figura 1.14: Coche de Houdina Radio Control Company.

Después de este periodo, importantes empresas automotrices y tecnológicas comenzaron a invertir fuertemente en la investigación de la conducción autónoma, como Waymo, Tesla, Huawei, Nuro... (ver Figura 1.16) y probablemente se agregarán más empresas a esa lista en los próximos años. Este interés comercial ha impulsado el campo de la investigación en inteligencia artificial y, gracias al uso de hardware como GPU y técnicas como DeepLearning y Reinforcement Learning, la

Figura 1.15: DARPA Grand Challenge (superior), DARPA Urban Challenge (inferior).

velocidad de desarrollo se ha visto rápidamente aumentada.

Estas soluciones autónomas se pueden clasificar en uno de los 5 niveles de autonomía, donde se incrementa la automatización en cada nivel. Las soluciones de las empresas anteriormente mencionadas caen en el segundo y tercer nivel de la escala, sin embargo, solamente a partir del cuarto nivel podemos encontrar vehículos totalmente autónomos sin que se requiera intervención humana.

Los beneficios de la conducción autónoma son inmensos e incluyen una mayor seguridad vial, una reducción de la congestión del tráfico y una mayor movilidad para las personas que no pueden conducir vehículos tradicionales. Sin embargo, hoy en día persisten desafíos importantes, como: leyes para regular su uso,

Figura 1.16: Imagen promocional de Waymo (derecha), vista de la solución de conducción autónoma de Tesla (izquierda).

consideraciones éticas, y garantizar la robustez de los sistemas de IA en condiciones de la vida real con agentes inesperados.

1.3.1. Conducción autónoma con técnicas de IA

Las soluciones de última generación para la conducción autónoma incluyen tanto la percepción como la toma de decisiones. La mayoría de ellas siguen un enfoque modular. En ellas la aplicación se descompone en varios módulos, resolviendo cada uno un aspecto, que se combinan entre sí para obtener una correcta funcionalidad. Normalmente hay módulos de percepción, fusión de sensores, localización, planificación de ruta global, evitación de obstáculos, control de bajo nivel, etc.

Por ejemplo, en [1] se propone una arquitectura híbrida que combina módulos de estrategia, táctica y ejecución. El módulo de estrategia define la trayectoria a seguir y el módulo de decisión táctica emplea un algoritmo de optimización de políticas proximal y un aprendizaje de refuerzo profundo.

Hace unos años se propusieron algunos enfoques [2] [3] de extremo a extremo, que consistían en una red neuronal alimentada directamente con los sensores del vehículo (por ejemplo cámaras), y conectada directamente a los actuadores del vehículo. Un enfoque de extremo a extremo relevante es la arquitectura PilotNet desarrollada por NVIDIA [4] (ver sección 3.10).

La principal inspiración para este trabajo es un TFM[5], donde se entrenaron redes neuronales de clasificación y regresión con Aprendizaje por Imitación para lograr el objetivo de conducir un coche de F1 simulado en un circuito. Trabajos inspiradores más recientes del mismo grupo de investigación y enfoque son [6, 7, 8].

2. Objetivos y metodología

2.1. Objetivos

El objetivo principal de este trabajo es resolver dos problemas relacionados con la conducción autónoma. Para lograr estas dos operaciones simplificadas, Sigue Línea y Sigue Carril, pretendemos entrenar un modelo de red neuronal con Aprendizaje por Imitación para utilizarlo como controlador del coche. En una primera iteración, las soluciones se probarán en un entorno simulado con un vehículo holonómico y un vehículo con dinámica de Ackermann, y luego, intentaremos replicar el comportamiento de la solución del Sigue Carril en un dispositivo robótico físico, el AWS Deepracer (ver sección 3.3).

2.2. Metodología

Para poder trabajar eficazmente en este proyecto, resolver dudas lo más rápido posible y establecer un registro del trabajo realizado hemos seguido estas pautas:

1. Para conseguir la ayuda del tutor en el TFG se han realizado reuniones semanales en el que se resolvieron dudas, se discutieron los avances respecto a la semana anterior y se propusieron próximos pasos de acción para el desarrollo del proyecto. Con el tutor dando retroalimentación en todos estos. Las reuniones fueron realizadas vía videollamada o presenciales.
2. Slack se utilizó para comunicarse tanto con el tutor, para resolver problemas simples, como con el resto de integrantes del equipo de Robotics Academy que ha trabajado en aprendizaje automático en el pasado.
3. Se creó un repositorio de GitHub[1] dentro del grupo de RoboticsLabURJC, donde se puede consultar todo el código y sus cambios. Además, para registrar el progreso de lo realizado hemos escrito un blog donde se explica todo el avance que se ha realizado en cada período de tiempo.

[1]https://github.com/RoboticsLabURJC/2022-tfg-alejandro-moncalvillo

2.3. Plan de Trabajo

El desarrollo de este TFG se puede dividió en estas fases:

1. Para tener una primera aproximación a los métodos de entrenamiento de redes neuronales comenzamos recreando el ejercicio de clasificación de dígitos de la plataforma educativa Unibotics [2]. Los resultados se pueden ver en [3].

2. Después de comprender cómo funcionan las bases de datos y el ciclo de entrenamiento en PyTorch, empezamos a preparar las simulaciones.

3. En la siguiente fase, transformamos y adaptamos el modelo holonómico del repositorio JdeRobot [4] para conseguir un coche con dinámica de Ackermman. También modificamos los modelos de circuitos en consonancia.

4. Una vez preparados los escenarios comenzamos con el proceso de obtención de datos, entrenamiento y evaluación de los modelos:

   a) Creamos un controlador experto para cada tarea usando un controlador PID[5].

   b) Luego obtuvimos los conjuntos de datos obtenidos de las ejecuciones simuladas del controlador experto.

   c) A continuación, entrenamos los modelos con una parte de los conjuntos de datos.

   d) Finalmente, probamos los modelos con los conjuntos de datos no utilizados para el entrenamiento y con inferencia en vivo en las simulaciones.

5. Como último experimento concluyente, decidimos seguir los mismos pasos mencionados en la anterior fase para conseguir un modelo neuronal para la tarea de seguimiento del carril en un agente físico, el AWS Deepracer (ver sección 3.3).

[2] http://jderobot.github.io/RoboticsAcademy/exercises/ComputerVision/dl_digit_classifier
[3] https://www.youtube.com/watch?v=X7luURncLwM
[4] https://github.com/JdeRobot/RoboticsInfrastructure
[5] https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller

3. Infraestructura y herramientas

En este capítulo enumeraremos y explicaremos el software (componentes virtuales) y el hardware (componentes físicos) que hicieron posible el desarrollo de este TFG.

3.1. Hardware de Cómputo

Todas las simulaciones y entrenamiento de los modelos se han realizado en un ordenador con una memoria de 32GB(RAM) y Ubuntu 22.04 como sistema operativo. En las siguientes subsecciones se explicarán dos de los elementos hardware que fueron fundamentales en el desarrollo del TFG.

3.2. Tarjeta gráfica

Las tarjetas gráficas , también conocidas como GPU (Unidades de procesamiento de gráficos), son un tipo de hardware que se utiliza principalmente para renderizar imágenes, vídeos y animaciones. Se utilizan ampliamente en videojuegos, visualización profesional de estructuras, animación, centros de gestión de datos e inteligencia artificial.

La serie *Geforce* de NVIDIA tiene componentes de funciones impulsados por IA como *Tensor Cores* y *RT Cores*, que permiten el uso de CUDA . La utilizada en este proyecto es la tarjeta *Geforce RTX 3070* (8GB de VRAM) (ver Figura 3.1).

Figura 3.1: NVIDIA Geforce RTX 3070.

3.3. Robot físico

El robot utilizado es el AWS DeepRacer (ver Figura 3.2). Se trata de un un coche de carreras a escala 1/18 con capacidad de conducción totalmente autónoma compuesto por los siguientes componentes:

1. 4 ruedas.
2. 1 motor continuo.
3. 1 servomotor.
4. Procesador Intel Atom™ (4 GB de RAM).
5. Memoria de 32GB para almacenamiento.
6. Wi-Fi 802.11ac.
7. Batería de motores (7,4 V/1100 mAh).
8. Batería del ordenador (13600mAh).
9. Cámara de 4 MP con MJPEG.
10. Puertos: 4x USB-A, 1x USB-C, 1x Micro-USB, 1x HDMI.

Figura 3.2: AWS Deepracer.

El propósito original del robot es utilizar la consola de AWS y sus simulaciones para entrenar un modelo mediante Aprendizaje por Refuerzo, pero al ser un dispositivo de código abierto, en nuestro caso, se ha utilizado para realizar inferencia en vivo con un modelo entrenado con Aprendizaje por Imitación. Para obtener un entorno de desarrollo similar al utilizado para las simulaciones actualizamos el sistema operativo del AWS Deepracer a Ubuntu 20.04 Focal Fossa (una distribución de GNU/Linux), también instalamos la distribución de ROS 2 Foxy Fitzroy y actualizamos Python a la versión 3.8.

3.4. Python

Python[1] es un lenguaje de programación muy conocido por su sencillez y versatilidad, y actualmente está establecido como uno de los más utilizados en el mundo de la ingeniería software. La extensa biblioteca estándar de Python proporciona una amplia gama de módulos para diversos fines, desde desarrollo web y análisis de datos hasta inteligencia artificial y cálculos científicos. Su amplia comunidad mejora continuamente las capacidades del lenguaje con bibliotecas innovadoras, solidificando la posición de Python como estándar en la industria. Consulte el Extracto de código 3.1 para ver un código de ejemplo.

```python
import turtle

screen = turtle.Screen()
turtle = turtle.Turtle()

side_size = float(input("Enter the size of the square sides: "))

for _ in range(4):
  turtle.forward(side_size)
  turtle.right(90)
```

Extracto de código 3.1: Programa simple que dibuja un cuadrado de un tamaño definido por el usuario usando la biblioteca ”turtle”.

La versión más reciente de Python, a día de hoy, es la 3.12.3. Para este TFG utilizamos la versión 3.10.12 ya que era la última disponible en el momento de realizar el código.

Otras bibliotecas específicas de Python que se han utilizado para el proyecto son: *numpy*, para realizar operaciones matriciales; *time* y *datetime*, para obtener medidas de tiempo y controlar la frecuencia del programa; *csv*, para operar con archivos con valores separados por comas; *matplotlib*, para visualizar los resultados; *PIL*, para gestionar conversiones de imágenes a matrices; *glob* y *os*, para obtener los datos necesarios del sistema de archivos del ordenador.

[1] https://www.python.org/

3.5. Middleware Robótico

El middleware robótico que se ha usado ha sido la distribución *Humble*[2] de ROS 2.

ROS 2 (Robot Operating System 2) es un kit de desarrollo de software de código abierto para aplicaciones de robótica. ROS 2 se ha creado teniendo como base a ROS 1, que se utiliza hoy en día en múltiples aplicaciones de robótica en todo el mundo. A menudo denominado "middleware", ROS 2 ofrece al usuario un canal de comunicación simplificado entre la aplicación robótica y los sensores y actuadores del sistema. El propósito de ROS 2 es ofrecer una plataforma de software estándar a desarrolladores de todas las industrias.

ROS 2 proporciona una amplia gama de herramientas, bibliotecas y capacidades robóticas necesarias para desarrollar aplicaciones complejas. Es de código abierto, el código ROS 2 usa la licencia Apache 2.0, con código ROS 1 trasladado bajo la licencia *3-clause BSD*. Ambas licencias permiten un uso permisivo del software, sin restricciones sobre la propiedad intelectual del usuario. El medio de comunicación predeterminado en ROS 2 utiliza estándares industriales como IDL, DDS y DDS-I RTPS, que están presentes actualmente en una variedad de aplicaciones industriales.

Se puede ver una estructura simple de ROS 2 en la figura 3.3. Generalmente está compuesto por múltiples nodos, estos nodos pueden tener cualquier número de suscriptores (*subscribers*) y publicadores(*publishers*), los cuales pueden comunicarse publicando o recibiendo mensajes de un tema específico (*topic*).

3.6. Simulador

El simulador que se ha utilizado es Gazebo.

Gazebo[3] es un simulador robótico de código abierto líder que se encuentra a la vanguardia de la investigación y el desarrollo de robótica. Reconocido por su versatilidad y robustez, Gazebo proporciona una plataforma integral para simular sistemas robóticos complejos con realismo. Su arquitectura modular permite una integración perfecta con distintos modelos de robots, sensores y entornos, lo que permite a los investigadores y desarrolladores simular una amplia gama de escenarios y probar algoritmos en un entorno virtual antes de implementarlos en el

[2]https://docs.ros.org/en/humble/
[3]https://gazebosim.org

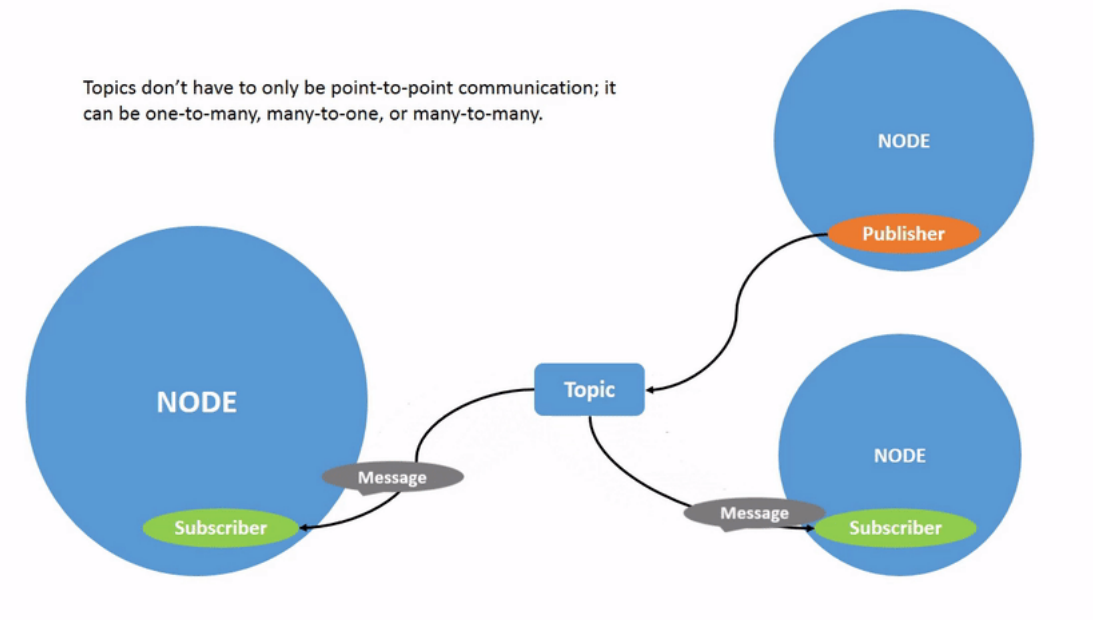


Figura 3.3: Representación de la estructura interna de ROS 2.

mundo real (ver Figura 3.4). La comunidad de Gazebo fomenta la colaboración y la

Figura 3.4: Simulación de un almacén en Gazebo.

innovación, añadiendo continuamente al simulador nuevas funciones y extensiones. Está equipado con un amplio conjunto de características como la simulación de físicas, la simulación de sensores y la compatibilidad con *plugins*. Gazebo permite a los usuarios crear simulaciones complejas que pueden imitar fielmente las condiciones del mundo real y pone a disposición de cualquiera la experimentación robótica sin la limitación de tener acceso a los dispositivos físicos.

Gazebo utiliza modelos de objetos SDF. Los modelos SDF pueden ser desde formas simples hasta robots complejos. Los modelos se componen de una etiqueta principal *¡model¿* y varias etiquetas opcionales que describen el modelo:

1. *Links*: contienen las propiedades físicas de una parte del modelo. Puede ser una rueda o un eslabón de una cadena conjunta. Cada *Link* suele contener varios elementos adicionales que pueden ser:
    - *Collision*: Un elemento de colisión describe una geometría que se utiliza para comprobar la colisión con el resto de los elementos.
    - *Visual*: Un elemento visual se utiliza para poder renderizar las partes definidas en un *Link*.
    - *Inertial*: El elemento inercial describe las propiedades dinámicas del enlace, como la masa y la matriz de inercia rotacional.
    - *Sensor*: Un sensor recopila datos del mundo, generalmente tiene un *plugin* asociado.
2. *Joints*: se encarga de conectar dos enlaces. Se establece una relación padre-hijo junto con otros parámetros como el eje de rotación y los límites de las articulaciones.
3. *Plugin*: se trata de una biblioteca compartida creada por un tercero para proporcionar funcionalidades extra, como por ejemplo, controlar el movimiento de un modelo.

La versión utilizada para este trabajo es la 11.10.2, y los *plugins* que hicieron posibles las simulaciones son: *libgazebo_ros_camera*, para la cámara; *libgazebo_ros_planar_move*, para realizar movimientos holonómicos; y *libgazebo_ros_ackermann_drive*, para simular la dinámica de Ackermann.

3.7. OpenCV

OpenCV (Open Source Computer Vision Library)[4] es una biblioteca de software de código abierto desarrollada para proporcionar un conjunto completo de herramientas para tareas de visión por computadora y aprendizaje automático. Lanzada inicialmente por Intel en 2000, desde entonces se ha convertido en una de las bibliotecas más utilizadas en estos campos y está integrada en una varios de lenguajes de programación, incluidos C++, Python y Java.

Las principales características de OpenCV que se utilizan para este trabajo son:

[4]https://opencv.org/

1. Procesamiento de imágenes: OpenCV ofrece amplias funciones para procesar imágenes, incluido el filtrado, la detección de bordes, la ecualización de imágenes y las transformaciones geométricas.
2. Captura y análisis de vídeo: la biblioteca admite la captura y el análisis de vídeo en tiempo real, lo que permite tareas como el seguimiento de objetos o la detección de movimiento.
3. Calibración de cámara y visión 3D: OpenCV incluye herramientas para calibrar cámaras, corregir la distorsión de las lentes y reconstruir modelos 3D a partir de múltiples imágenes 2D.

En este trabajo se ha utilizado la versión 4.6.0.

3.8. Biblioteca Pytorch de *DeepLearning*

PyTorch[5] es una biblioteca de aprendizaje automático de código abierto desarrollada principalmente por el laboratorio de investigación de IA de Facebook (*FAIR*). Proporciona un marco flexible y dinámico para construir y entrenar redes neuronales, lo que lo hace particularmente popular entre investigadores y profesionales en los campos del aprendizaje profundo y la inteligencia artificial.

Las principales características de PyTorch que están intrínsecamente relacionadas con este trabajo son:

1. Tensores: En esencia, PyTorch utiliza tensores, que son matrices multidimensionales similares a las matrices de *NumPy* pero con capacidades adicionales para la aceleración por GPU, lo que acorta los tiempos de entrenamiento e inferencia.
2. Módulo de red neuronal (torch.nn): PyTorch incluye un módulo de red neuronal de alto nivel que simplifica la creación y el entrenamiento de redes neuronales. Proporciona capas prediseñadas, funciones de pérdida y optimizadores, lo que facilita la construcción de modelos complejos.
3. Optimizadores para entrenamiento: PyTorch ofrece una variedad de algoritmos de optimización, como SGD, Adam (que es el utilizado en este proyecto), L-BFGS y RMSprop, que son esenciales para entrenar redes neuronales. Estos optimizadores se pueden integrar fácilmente en el ciclo de entrenamiento.

[5]https://pytorch.org/

4. Compatibilidad con GPUs de NVIDIA: PyTorch se puede integrar perfectamente con CUDA, lo que permite un cálculo eficiente en las GPUs NVIDIA. Esto acelera el entrenamiento de redes neuronales complejas y modelos de aprendizaje profundo.

Para este TFG hemos utilizado la versión 2.1.2+cu121 (que incluye herramientas para utilizar CUDA).

3.9. CUDA

CUDA es la plataforma de cálculo paralelo y el modelo de programación de NVIDIA que permite grandes aumentos en el rendimiento informático de tareas específicas, como el entrenamiento de modelos de redes neuronales, mediante el uso de la arquitectura distintiva de la GPU.

3.10. Modelo neuronal

Para este trabajo hemos utilizado la arquitectura de red neuronal Pilotnet desarrollada por NVIDIA para solucionar problemas de conducción autónoma[4]. Tiene alrededor de 27 millones de conexiones y 250 mil parámetros y consta de 9 capas divididas en: 1 capa de normalización, 5 capas convolucionales y 3 capas completamente conectadas (*fully-connected*) (ver Figura 3.5). Las capas convolucionales se utilizan para obtener las características relevantes de las imágenes de la carretera y las completamente conectadas (*fully-connected*) para actuar como controlador del ángulo de giro de dirección de las ruedas.

Esta aquitectura de red, en el formato utilizado por Pytorch, se puede encontrar en [6].

[6] https://github.com/RoboticsLabURJC/2022-tfg-alejandro-moncalvillo/blob/main/dl_car_control/Ackermann/utils/pilotnet.py

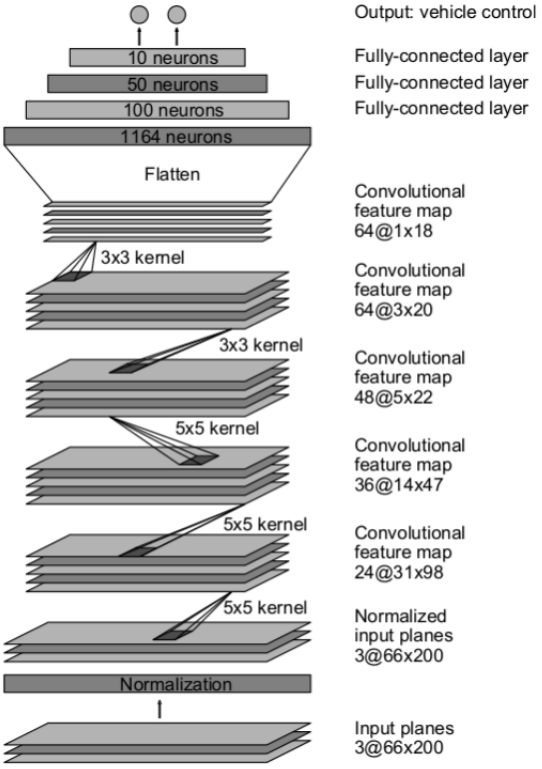


Figura 3.5: Arquitectura PilotNet

4. Aplicación Sigue Línea

En primer lugar, entrenamos un modelo para realizar la tarea Seguir línea, donde el coche, utilizando como datos las imágenes obtenidas por una cámara a bordo, sigue una línea roja para completar la vuelta del circuito. Los pasos seguidos para lograr el modelo de Aprendizaje por Imitación que imita un controlador de robot visual extremo a extremo son:

1. Preparar el robot y el escenario (en este caso las simulaciones).
2. Para recopilar un conjunto de datos supervisados de un agente experto, la entrada sensorial y la salida supervisada se registran y guardan para usarse como datos para el entrenamiento. Después es necesario un preprocesamiento del conjunto de datos.
3. Elección de un modelo de red neuronal y entrenar la red con el conjunto de datos.
4. Evaluar el rendimiento del modelo entrenado.

4.1. Configuración de la simulación para la Aplicación Sigue Línea

Para la configuración hemos utilizado ROS 2 [9] Humble (una de las distribuciones actuales de ROS 2) (ver sección 3.5) como plataforma robótica para controlar y obtener información del coche simulado. Gazebo [10] (ver Sección 3.6) es el simulador seleccionado.

El modelo de coche original obtenido del repositorio de JdeRobot [1] tenía un tipo de movimiento holonómico (que permite la rotación en el lugar, común en aspiradoras y robots cilíndricos de interior).La estructura del modelo SDF del coche holonómico se puede ver en la Figura 4.1. Pero los vehículos reales con ruedas holonómicas no son muy comunes, así que decidimos modificarlo usando un *plugin* de Gazebo[2] para conseguir un movimiento más realista. El *plugin* se basa en la geometría diferencial de Ackermann, que es un principio de diseño utilizado

[1] https://github.com/JdeRobot/RoboticsInfrastructure
[2] https://docs.ros.org/en/ros2_packages/rolling/api/gazebo_plugins/generated/

en los mecanismos de dirección de los vehículos para resolver el problema de las ruedas de un vehículo que necesitan trazar diferentes ángulos de giro al girar(se pueden ver los dos modelos uno al lado del otro en la Figura 4.2). Esto es necesario porque las ruedas interiores tienen un radio de giro más corto en comparación con las ruedas exteriores, lo cual hace obligatorio la necesidad de girar en un ángulo más pronunciado. Para obtener este modelo necesitábamos dividir el modelo original del coche en chasis, ruedas y volante. Se puede ver la estructura del modelo SDF del coche con dinámica Ackermann en la Figura 4.3.

```
<?xml version='1.0'?>
<sdf version="1.4">
<model name="f1_renault">
  <pose>0 0 0 0 0 0</pose>
  <static>false</static>
  <link name="f1">…
  <inertial>
        <mass>10</mass>
        <inertia>…
      </inertia>
    </inertial>
    <collision name="collision">…
    </collision>
    <visual name="visual">…
    </visual>
    <visual name='left_cam'>
      <pose>0.45000 0.040000 0.1000000 0.000000 -0.000000 0.0000000</pose>
      <geometry>
        <sphere>
          <radius>.005</radius>
        </sphere>
      </geometry>
    </visual>
    <sensor name='cam_f1_left' type='camera'>
      <pose>0.45000 0.040000 0.1000000 0.000000 -0.000000 0.0000000</pose>
      <update_rate>20.000000</update_rate>
      <camera name='cleft'>…
      </camera>
      <plugin name="camera_controller_left" filename="libgazebo_ros_camera.so">…
      </plugin>
    </sensor>

  </link>

  <plugin name="object_controller" filename="libgazebo_ros_planar_move.so">…
  </plugin>
</model>
</sdf>
```

Figura 4.1: Estructura SDF del modelo de coche holonómico.

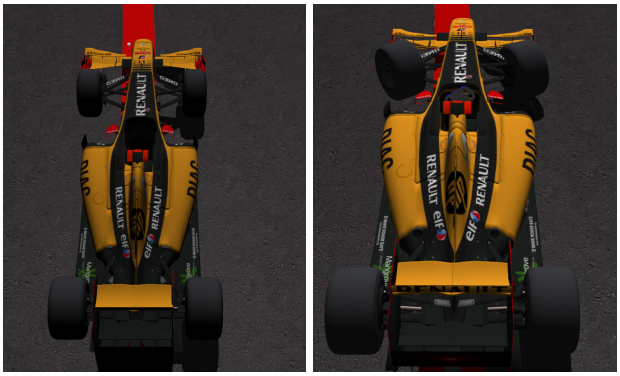


Figura 4.2: Modelo holonómico utilizado (izquierda), modelo con dinámica de Ackermann utilizado (derecha).

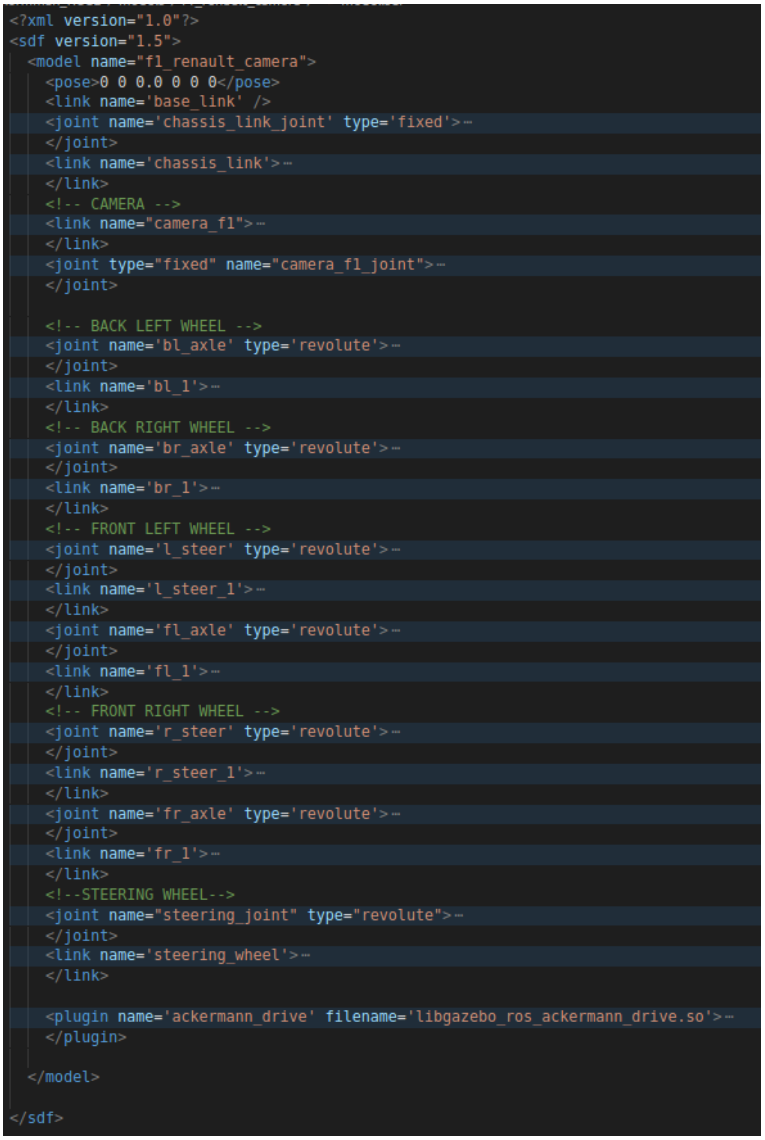


Figura 4.3: Estructura SDF del modelo de coche Ackermann.

Se puede encontrar un vídeo del modelo creado de coche Ackermann en acción aquí[3].

4.1.1. Escenarios del Sigue Línea Simulado

En cuanto a los escenarios, se desarrollaron varios circuitos (Figura 4.4) imitando algunos circuitos famosos de Fórmula 1 como Montmeló, Nurburgring o MonteCarlo. Algunos de ellos fueron utilizados como escenarios de entrenamiento para recopilar los conjuntos de datos (imágenes y velocidades). Otros se utilizaron como escenarios de prueba. Todos ellos están disponibles públicamente en el

[3]https://www.youtube.com/watch?v=PXaBTjq7vok

repositorio de JdeRobot[4].

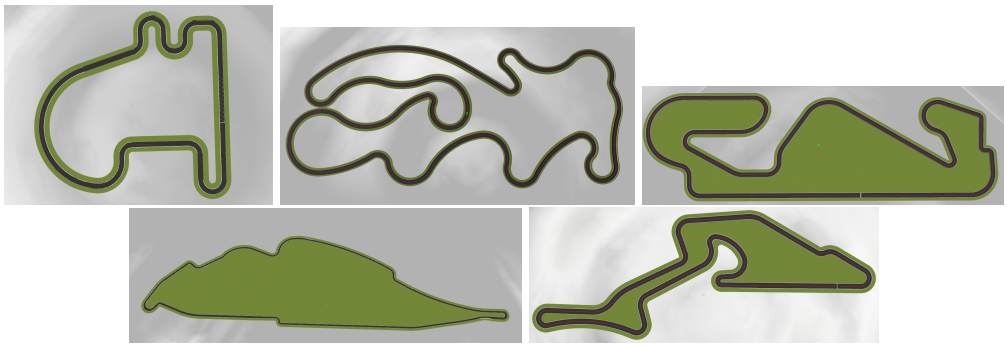


Figura 4.4: Vista de los circuitos en Gazebo.

El modelo de automóvil con dinámica de Ackermann es con diferencia más grande que el modelo holonómico. Este problema hizo imposible girar en algunas curvas, por lo que para que cupiese en todas las partes de los circuitos, escalamos los modelos de los circuitos para resolver el problema. Los resultados se pueden ver en la Figura 4.5.

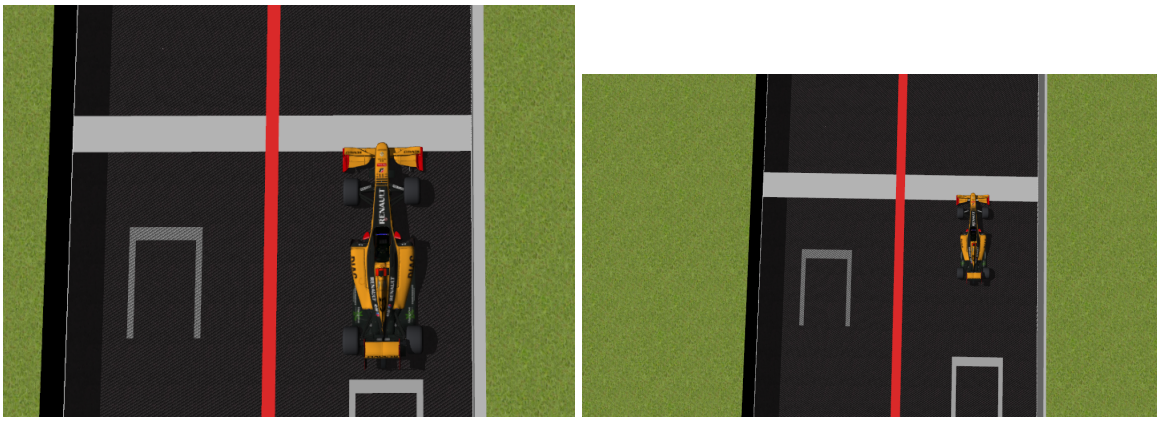


Figura 4.5: Modelo de coche en el circuito original (arriba) y modelo de coche en el circuito modificado (abajo).

4.2. Conjuntos de datos para Sigue Línea Simulado

Para crear las etiquetas del conjunto de datos el primer paso era crear una aplicación que completase la tarea haciendo uso de un algoritmo.

En cuanto a la arquitectura de una aplicación para completar la tarea de Sigue Línea basada en visión, en términos de ROS 2, se puede ver en la figura 4.6. En este ejemplo hay un total de tres nodos:

[4]https://github.com/JdeRobot/RoboticsInfrastructure

1. *Camera Node*: publica las imágenes que obtiene de un dispositivo de cámara en el tema */image*.
2. *Processor Node*: recibe los datos necesarios del tema */image*, luego, los procesa para obtener la salida deseada y envía el resultado a cualquier nodo suscrito al tema */speed*.
3. *Motors node*: recibe los mensajes publicados en el tema */speed* y activa los motores de acuerdo con la información recibida.

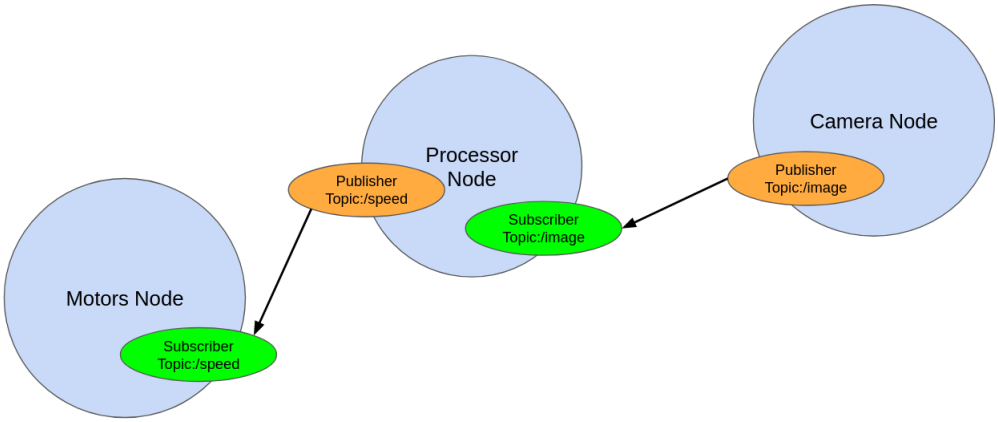


Figura 4.6: Representación interna de en ROS 2 de una aplicación para Sigue Línea basada en visión.

Además de utilizar ROS 2, creamos un módulo adicional *HAL* (Human Abstraction Layer), que crea otra capa de abstracción (ver Figura 4.7) a parte de la funcionalidad de ROS 2 . El usuario tiene acceso a tres métodos para controlar y recibir datos del coche: *getImage()*, para obtener la imagen de la cámara en formato OpenCV; *setV()* y *setW()*, para cambiar el acelerador y la dirección del coche respectivamente. Finalmente, el usuario tiene acceso a un método adicional, *main()*, que establece la frecuencia de la aplicación del usuario (ver Extracto de código 4.1).

```
#initialize the ROS 2 python library
rclpy.init()
def main(user_main):
    #Create node executor
    executor = MultiThreadedExecutor()
    executor.add_node(camera_subscriber_node)
    executor.add_node(speed_publisher_node)
    frequency = 60
```

```
    time_cycle = 1000.0 / frequency
    try:
        while rclpy.ok:
            #get the time before the user code
            start_time = datetime.now()
            #execute the user code
            user_main()
            #get the time after the user code
            finish_time = datetime.now()
            #calculate the time difference
            dt = finish_time - start_time

            ms = (dt.days * 24 * 60 * 60 + dt.seconds) * 1000 + dt.
    microseconds / 1000.0
            # If the time diference is lower than the selected one the
    program sleeps until
            # it is reached
            if(ms < time_cycle):
                time.sleep((time_cycle - ms) / 1000.0)
            # Finally we activate the ROS 2 nodes to recieve or send
    information
            executor.spin_once()
    except KeyboardInterrupt:
        # The program stops when there is a KeyboardInterrupt event
        pass
    finally:
        # Destroy the ROS2 nodes and shutdown the ROS 2 client library
    for Python
        Destroy_nodes()
        rclpy.shutdown()
```

Extracto de código 4.1: HAL *main()*.

Para obtener las etiquetas del conjunto de datos utilizamos un controlador experto que utiliza un PID y un filtro de color. El controlador experto utiliza un filtro de color (ver Listado 4.2) para obtener la situación del automóvil con respecto a la línea, luego el controlador PID proporciona la velocidad lineal y angular deseada para permanecer en la línea o carril, si el sistema no encuentra la línea, el vehículo retrocede hasta que sea visible nuevamente (consultar Extracto de código

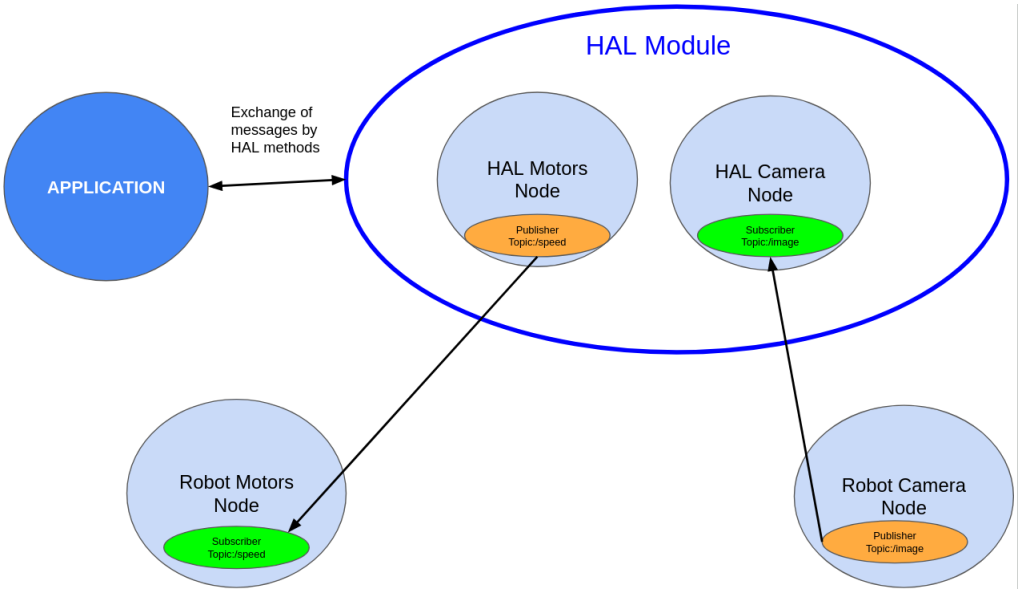


Figura 4.7: Estructura funcional del módulo HAL.

4.3).

```
image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
lower_red = np.array([0, 50, 50])
upper_red = np.array([180, 255, 255])
image_mask = cv2.inRange(image_hsv, lower_red, upper_red)

points = get_relevant_pixels(image_mask)
```

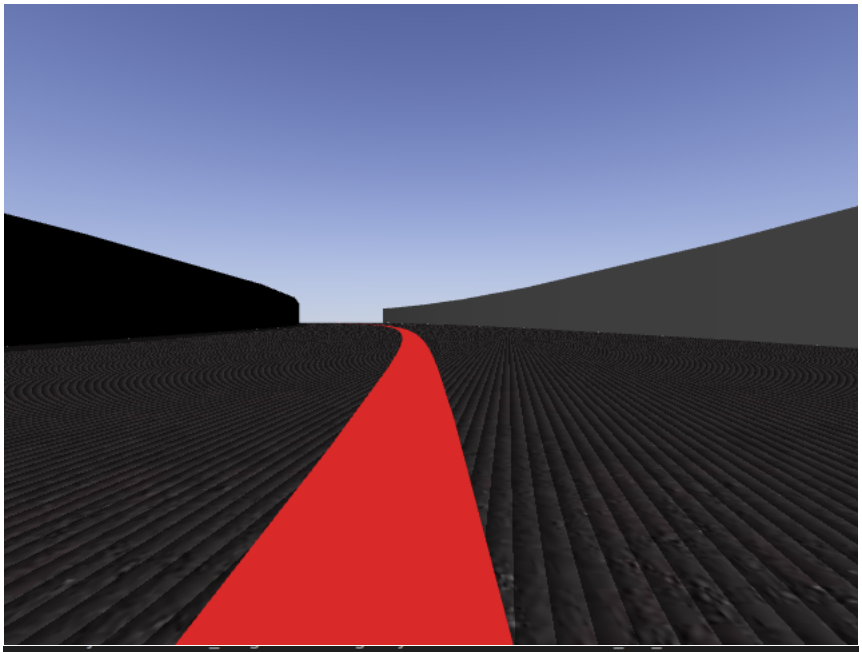
Extracto de código 4.2: Filtro de color que usa el agente experto

```
if line_not_found(points) == True:
    speed, rotation = missing_case()
else:
    if is_straight_line(points):
        speed, rotation = straight_case(points)
    elif is_curve(points):
        speed, rotation = curve_case(points)
    else:
        speed, rotation = exception_case(points)
```

Extracto de código 4.3: Pseudo-código de toma de deciciones del agente experto.

Los conjuntos de datos se componen de múltiples imágenes obtenidas del controlador experto y un único archivo *.csv* que contiene la información sobre la

velocidad lineal y angular correspondiente para cada imagen (ver Figura 4.8).



```
1   image_name,v,w
2   1.png,5,-0.374
3   2.png,5,-0.352
4   3.png,5,-0.352
5   4.png,5,-0.352
6   5.png,5,-0.352
7   6.png,5,-0.352
8   7.png,5,-0.352
9   8.png,5,-0.352
10  9.png,5,-0.352
11  10.png,5,-0.352
12  11.png,13,0.1857
13  12.png,13,0.1782
14  13.png,5,-0.352
15  14.png,5,-0.352
```

Figura 4.8: Imagen del conjunto de datos (arriba) y archivo *.csv* (abajo).

Como se cuenta en [5], dado que casi la mitad de la imagen no proporciona información relevante para nuestro caso, utilizamos imágenes recortadas en el entrenamiento para obtener mejores resultados (ver Figura 4.9).

Finalmente, para obtener más datos y mejorar la robustez del modelo, realizamos un aumento de datos sobre el conjunto de datos. Las dos técnicas utilizadas son : imágenes en espejo y aplicar difuminado gaussiano[5].

[5]https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#gaabe8c836e97159a9193fb0b11ac52cf1

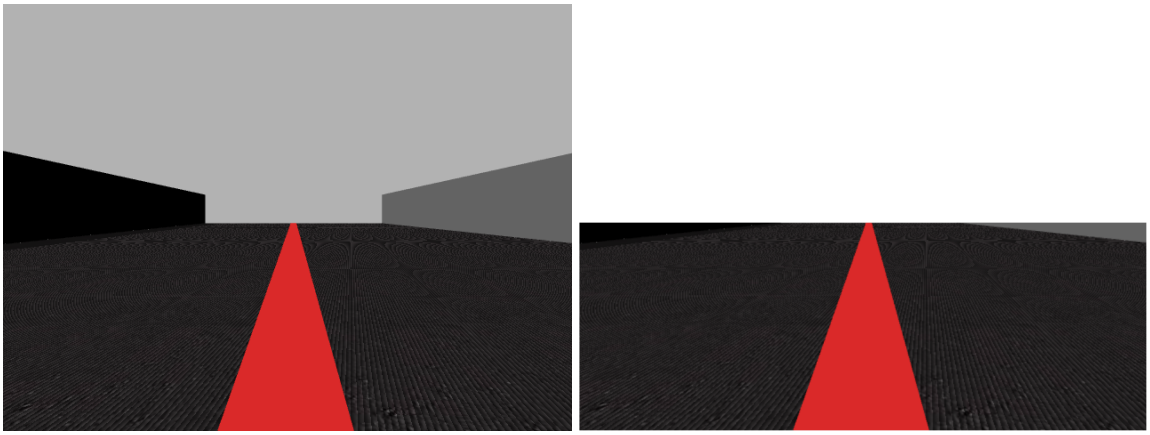


Figura 4.9: Imágenes del conjunto de datos: sin procesar (izquierda) y recortadas (derecha).

4.3. Modelo neuronal y entrenamiento de la red para Sigue Línea Simulado

El modelo de red utilizado en este caso fue una variante de PilotNet (ver sección 3.10),ya que ha demostrado buenos resultados en la literatura para el aprendizaje en casos de control extremo a extremo. En nuestro caso la estructura de la red se ha modificado ligeramente (ver Figura 3.5) para inferir no solo el ángulo de dirección sino también el valor del acelerador, similar al del TinyPilotNet [3]. La red fue implementada en PyTorch[6],y entrenada usando el algoritmo de Adam[7] con un valor de pérdida obtenido del error cuadrático medio [8] entre los valores de inferencia y los reales.

El pseudocódigo del bucle de entrenamiento se puede ver en el Listado 4.4.

```
1
2     num_epochs = arguments.num_epochs
3
4     # Load data
5     train_dataset = PilotNetDataset(path_to_train_dataset)
6     validation_dataset = PilotNetDataset(path_to_validation_dataset)
7     # Load Model
8     pilotModel = PilotNet()
9     # Select loss and optimizer to be used
```

[6]https://pytorch.org/
[7]https://pytorch.org/docs/stable/generated/torch.optim.Adam.html
[8]https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html

```
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam()

    # Train the model
    for epoch in range(0, num_epochs):
        #Train phase
        pilotModel.train()
        for images, labels in train_dataset:
            outputs = pilotModel(images)
            loss = criterion(outputs, labels)
            # Backpropagation and perform Adam Optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        #Validation phase
        pilotModel.eval()
        for images, labels in validation_dataset:
            outputs = pilotModel(images)
            val_loss += criterion(outputs, labels)
        # compare
        if val_loss < GLOBAL_MINUMUM_LOSS:
            GLOBAL_MINUMUM_LOSS = val_loss
            best_model = deepcopy(pilotModel)
            torch.save(best_model)
```

Extracto de código 4.4: Pseudocódigo del bucle de entrenamiento.

Uno de los elementos más importantes a tener en cuenta son las épocas (*epochs*), que es la cantidad de veces que se realizará el ciclo de entrenamiento. Probamos dos modelos entrenados con el mismo conjunto de datos, pero con diferentes valores de épocas, con un conjunto de datos simple de 240 imágenes. El efecto en los resultados de la inferencia del modelo se puede ver en la Figura 4.10.

Ambos modelos fueron entrenados durante unas 400 épocas.

En este primer caso, para el entrenamiento del modelo holonómico utilizamos un conjunto de datos compuesto por 52.000 imágenes, y para el modelo de Ackermann el conjunto de datos tiene alrededor de 70.000 imágenes. Todos los conjuntos de datos se analizaron para balancear el número de imágenes

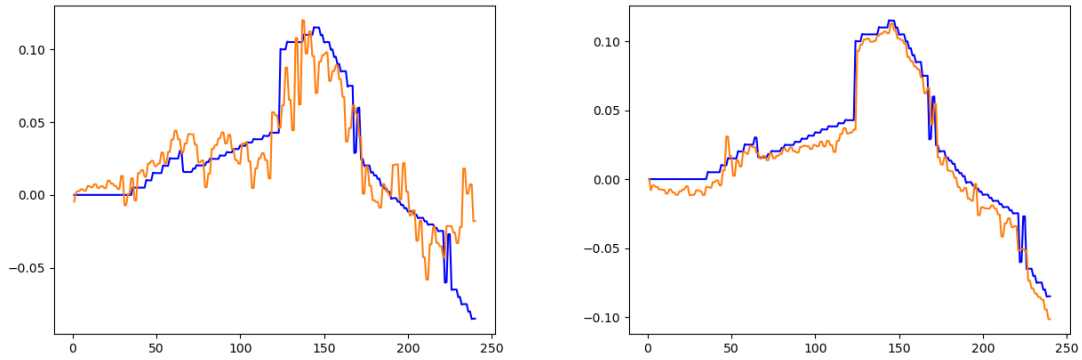


Figura 4.10: Comparativa de los resultados de inferencia de un modelo (en color naranja) entrenado en 100 épocas (izquierda) y de un modelo entrenado en 500 épocas (derecha) contra los valores del conjunto de datos (en color azul en ambas imágenes).

representativas de cada caso (línea recta, curva, excepción, o línea perdida), dándole una mayor importancia a las curvas, ya que se tratan del caso más común (ver Figura 4.11).

Debido a la cantidad de datos utilizados, decidimos utilizar CUDA 3.9, para aprovechar la capacidad de cálculo de la GPU con el fin de disminuir el tiempo de entrenamiento.

4.4. Validación experimental de la Aplicación Sigue Línea Simulado

En cuanto a la evaluación experimental, se han creado *scripts* de Tests *Offline* y en vivo:

- Test *Offline*: utiliza las imágenes del conjunto de datos pasadas al programa y un modelo de red neuronal para realizar inferencias, después muestra una gráfica para comparar las salidas de la red con los datos almacenados en el archivo *.csv* del conjunto de datos (consulte el Extracto de código 4.5).

```
1     #load the model
2     pilotModel = PilotNet()
3     pilotModel.load_state_dict(aruments.net_name)
4     pilotModel.eval()
5     # Select the dataset data file
```

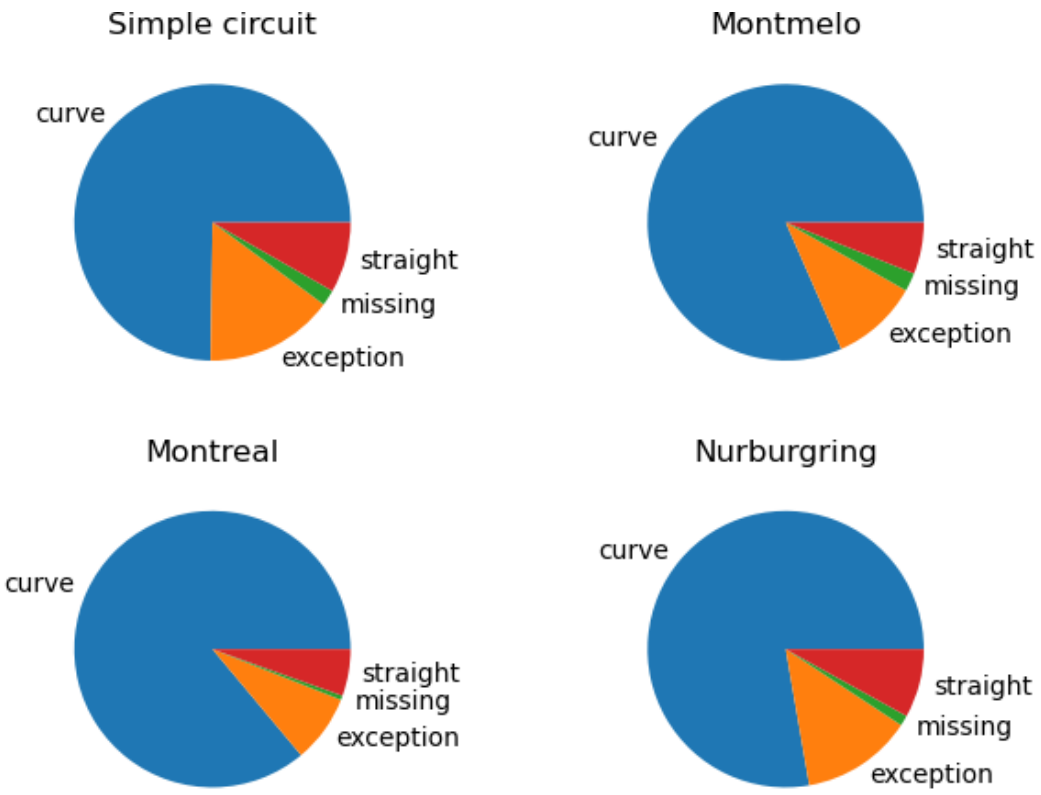


Figura 4.11: Gráficos circulares que muestran la cantidad de entradas de distintas situaciones posibles en los conjuntos de datos de cuatro circuitos diferentes.

```
6     data_file = open(aruments.dataset_data)
7
8     for line in data_file:
9
10        #get image
11        image = cv2.imread(line[0])
12
13        #crop the image
14        cropped_image = image[240:480, 0:640]
15
16        output = pilotModel(cropped_image)
17
18        net_v_array.append(output[0])
19
20        net_w_array.append(output[1])
21
```

```
        data_v_array.append(float(line[1]))
        data_w_array.append(float(line[2]))

    #plot the results
    plot_outputs(net_v_array,net_w_array,data_v_array,data_w_array)

```

Extracto de código 4.5: Pseudocódigo del script de test offline.

- Test en vivo: utiliza un modelo de red neuronal y el módulo HAL para realizar inferencias en vivo sobre imágenes obtenidas de un componente robótico (simulado o real) (ver Extracto de código 4.6).

```

    #load the model
    pilotModel = PilotNet()
    pilotModel.load_state_dict(aruments.net_name)
    pilotModel.eval()

    def main():
        HAL.main(application_main)

    def application_main():
        #get image
        image= HAL.getImage()
        #crop the image
        cropped_image = image[240:480, 0:640]

        output = pilotModel(cropped_image)

        v = output[0]
        w = output[1]

        HAL.setV(v)
        HAL.setW(w)


```

Extracto de código 4.6: Pseudocódigo del script de test de inferencia en vivo

En la evaluación *Offline* el modelo entrenado para el modelo holonómico presentó unos resultados más parecidos al controlador experto que el modelo de Ackermann (ver Figuras 4.12, 4.13). Sin embargo, ambos pueden imitar el controlador PID hasta un grado que permite completar el circuito.

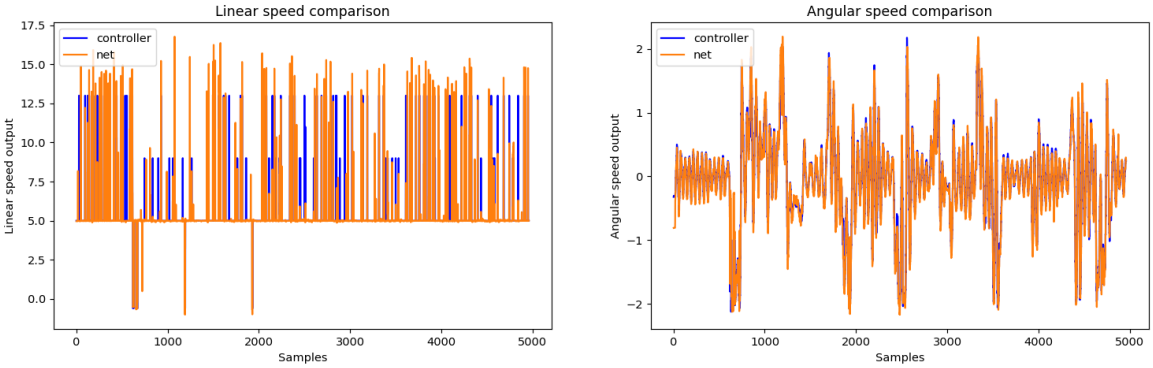


Figura 4.12: Comparación entre la salida del modelo entrenado (en naranja) y la salida del controlador experto (en azul) para el modelo de automóvil holonómico para la tarea de seguimiento de línea, usando unas 5000 imágenes como datos de entrada

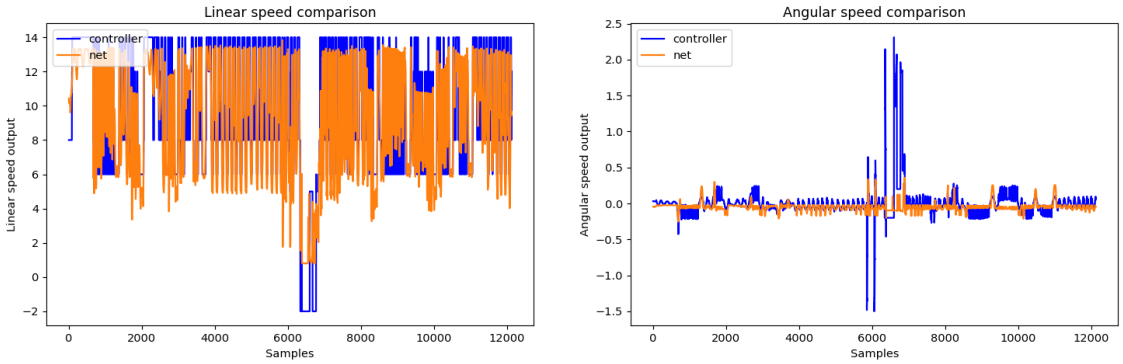


Figura 4.13: Comparación entre la salida del modelo entrenado (en naranja) y la salida del controlador experto (en azul) para el modelo de coche con dinámica de Ackermann para la tarea de seguimiento de línea, usando alrededor de 12000 imágenes como datos de entrada

En este vídeo [9] se puede ver una ejecución típica del modelo neuronal entrenado para el coche con movimiento holonómico y en este otro [10] una ejecución del modelo neuronal entrenado para el coche de Ackermann.

Algunas situaciones características de ambas ejecuciones que se pueden comentar son:

[9] https://youtu.be/zmlrHEOTUmI
[10] https://youtu.be/eaSRJ3pmuT0

- Tramos Rectos (Figura 4.14): ambos tipos de coches realizan movimientos similares en estos tramos, en el caso de Ackermann se puede ver ligeramente cómo las ruedas están giradas

Figura 4.14: Imágenes de comparación entre ejecuciones del Sigue Línea holonómico (derecha) y Ackermann (izquierda) en un tramo recto.

- Tramos de Curva (Figura 4.15): en estas partes de los circuitos se puede apreciar una mayor diferencia. El coche holonómico tiene un control más permisivo que permite realizar giros como el visto en la imagen derecha de la Figura 4.15. Mientras que el de Ackermann necesita un control más suave para no desviarse, ya que no puede rectificar con tanta facilidad.

Figura 4.15: Imágenes de comparación entre ejecuciones del Sigue Línea holonómico (derecha) y Ackermann (izquierda) en una curva.

Una de las lecciones aprendidas con estas pruebas experimentales es que el coche de Ackermann tiene una dinámica más sensible que el holonómico, lo que significa que es más difícil de controlar adecuadamente debido a que tiene un sistema más complejo (ver Sección 4.1). Por este motivo, para obtener un modelo exitoso hemos necesitado obtener conjuntos de datos más grandes.

5. Aplicación Sigue Carril

Después de obtener resultados exitosos con el caso Follow Line, pasamos a una aplicación más cercana a un escenario realista, el Sigue Carril. En este caso el sensor principal sigue siendo una cámara situada en la parte frontal del vehículo, y, de igual manera, las salidas del sistema son la velocidad linear y el giro de las ruedas delanteras del coche. Para abordar el desarrollo de esta aplicación se han seguido los mismos pasos que en el capítulo 4, primero para las aplicaciones de los automóviles simulados, y después para la aplicación del robot físico.

5.1. Configuración de la Simulación para la Aplicación Sigue Carril

La configuración para la simulación en este caso es la misma que la descrita en la sección 4.1. Usando los mismos programas y modelos de coches, tanto el holonómico como el de dinámica de Ackermann

5.1.1. Escenarios del Sigue Carril Simulado

Como el resto de la configuración, los escenarios para este caso son los mismos que los del apartado4.1.

5.2. Conjuntos de datos para Sigue Carril Simulado

Para el caso de Sigue Carril, en contraste con el Sigue Línea, tuvimos que cambiar la forma en que el controlador experto detecta la posición deseada de la línea, ya que debe estar en el lado izquierdo de la imagen en lugar de que esté centrada en la imagen. Además, existen otros problemas debido a que solo se tiene la mitad del espacio disponible para maniobrar cuando el vehículo intenta tomar una curva (ver Figura 5.1). Esto tiene cierto impacto en el control del coche, principalmente en la velocidad lineal, haciendo que se más dificil. Para solucionar este asunto, es necesario cambiar el controlador experto cambiando la velocidad

lineal usada al tomar una curva a una menor menor para no salirse de la carretera ni colisionar.

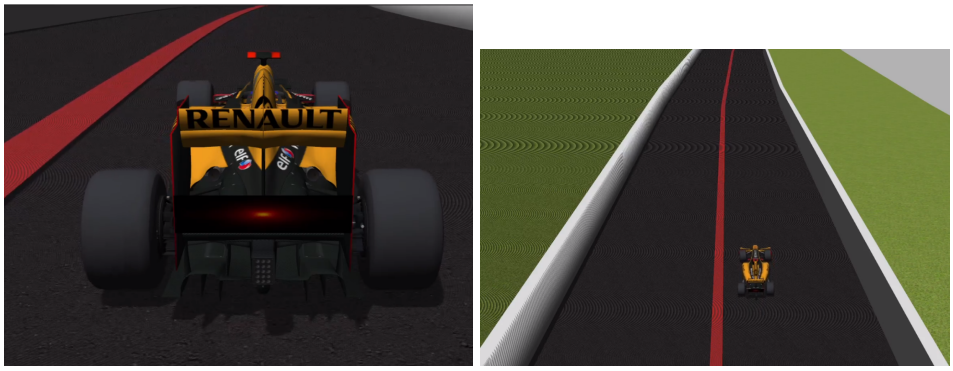


Figura 5.1: Imágenes de la ejecución de la aplicación de red neuronal de Sigue Carril del coche con dinámica de Ackermann

Después de modificar el controlador experto para completar esta tarea, obtuvimos las imágenes y etiquetas para crear los conjuntos de datos.

Como se indica en la sección 4.2, también recortamos las imágenes. En cuanto a las técnicas de aumentos de datos, para esta tarea solo usamos la adición de difuminado gaussiano a las imágenes, ya que solo queríamos que el coche siguiera el carril del lado derecho.

En este caso, para el entrenamiento del modelo holonómico simulado generamos un conjunto de datos compuesto por 40.000 imágenes, y para el modelo de Ackermann simulado el conjunto de datos tiene alrededor de 67.000 imágenes. Los conjuntos de datos también fueron balanceados en consonancia para reflejar los distintos casos que hay en cada circuito.

5.3. Modelo neuronal y entrenamiento de la red para Sigue Carril Simulado

El modelo neuronal y proceso de entrenamiento que utilizamos es el mismo que el descrito en la sección 4.3. En este caso, los modelos también se entrenaron durante unas 400 épocas.

5.4. Validación Experimental de la Aplicación de Sigue Carril Simulado

Después de entrenar los modelos con los conjuntos de datos, utilizamos los mismos scripts que en la sección 4.4 para obtener los resultados de la validación experimental.

Similar a lo visto en la sección 4.4, Los valores de inferencia del modelo holonómico parecen más cercanos a los supervisados en la fase de validación *Offline* que los obtenidos de la red para el modelo de Ackermann . (see Figures 5.2, 5.3).

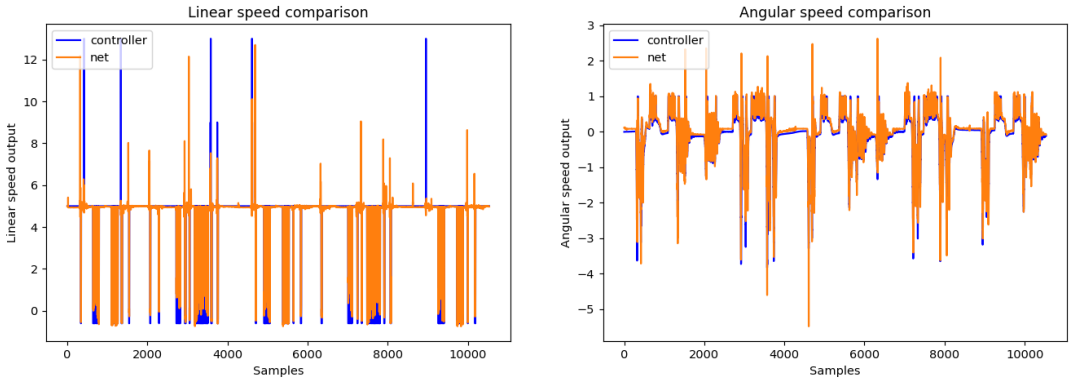


Figura 5.2: Comparación entre la salida de PilotNet y la salida del controlador experto para el modelo de automóvil holonómico para la tarea de seguimiento de carril

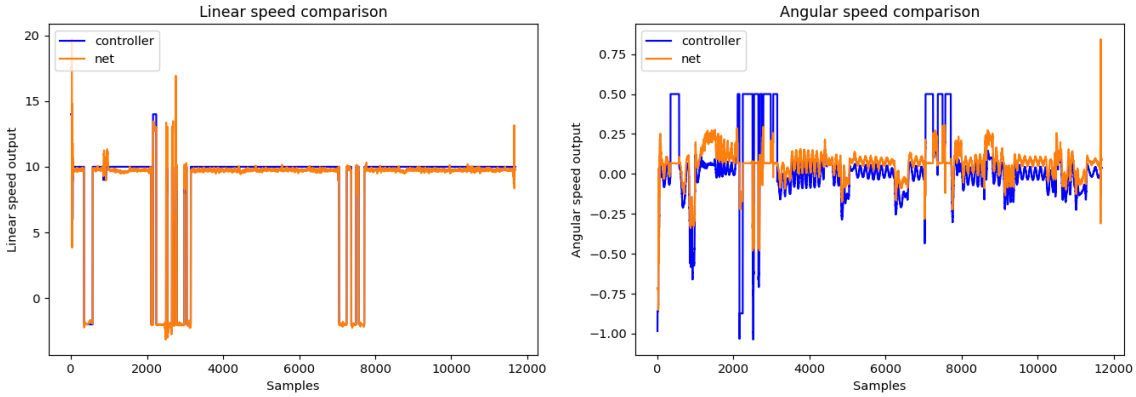


Figura 5.3: Comparación entre la salida de PilotNet y la salida del controlador experto para el modelo de coche con dinámica de Ackermann para la tarea de seguimiento de carril

A pesar de ello, ambos modelos de red completan de manera exitosa la tarea

propuesta en las simulaciones. Una ejecución típica del modelo de Ackermann se puede ver en este vídeo[1], y una del holonómico en este otro [2].

Algunas situaciones características de ambas ejecuciones que se pueden comentar son:

- Inicio del circuito (Figura 5.4): al comienzo de la simulación, cuando el coche está centrado, se puede apreciar el giro de las ruedas indicando que el modelo es capaz de distinguir el carril que debe seguir.

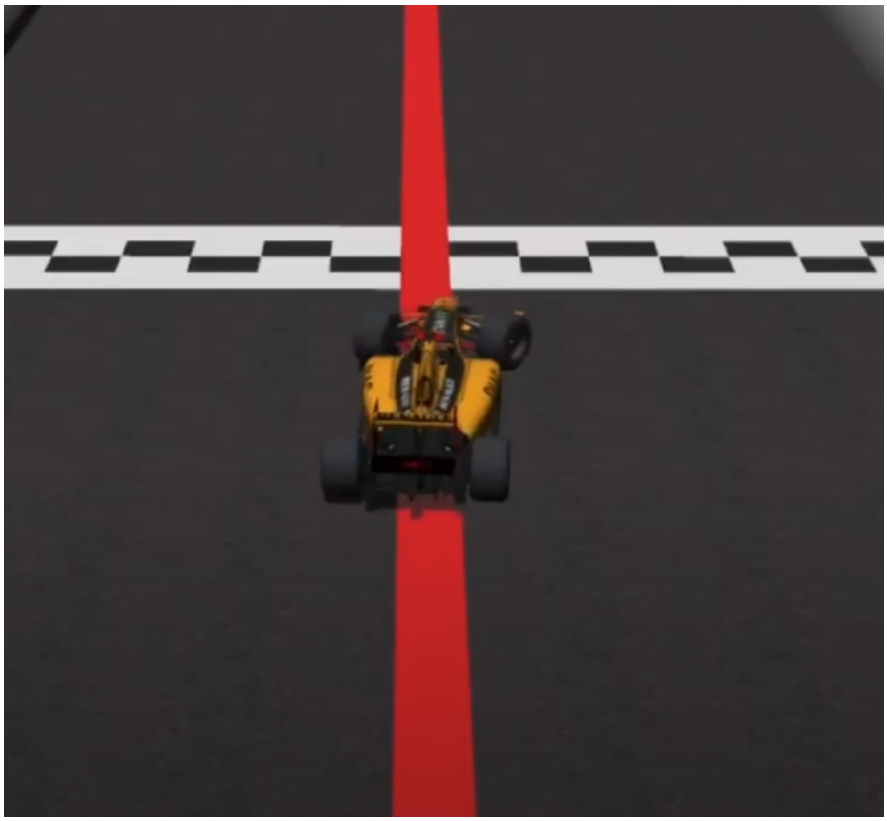


Figura 5.4: Coche de Ackerman girando las ruedas hacia el carril derecho.

- Tramos Rectos (Figura 5.5): en este caso los dos tipos de coches también realizan movimientos similares, como en el Sigue Línea.

- Tramos de Curva (Figura 5.6): análogamente al Sigue Carril, el coche holonómico es capaz de realizar giros más cerrados, mientras que el de Ackermann tiene que tomar las más curvas más suavemente para no invadir el carril contrario ni salirse del suyo.

[1] https://www.youtube.com/watch?v=MA6YnOSDr_8
[2] https://youtu.be/lZfpuWrUlW4

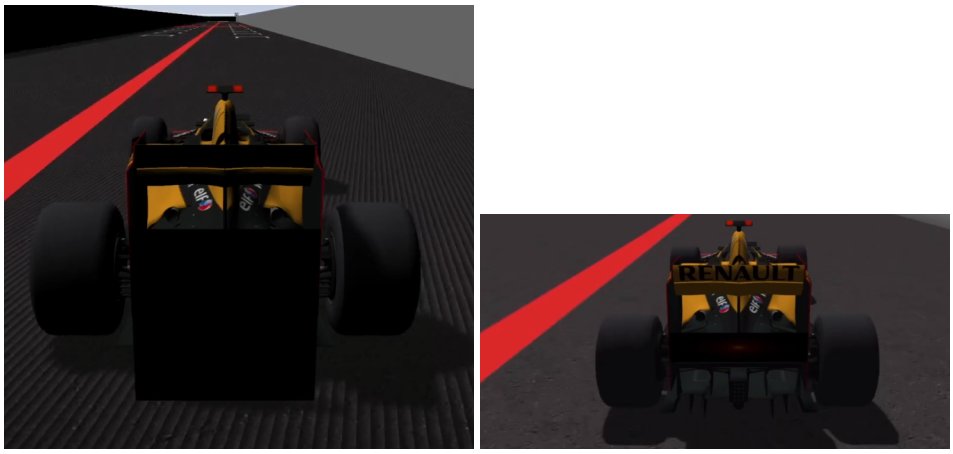


Figura 5.5: Imágenes de comparación entre ejecuciones del Sigue Carril holonómico (derecha) y Ackermann (izquierda) en un tramo recto.

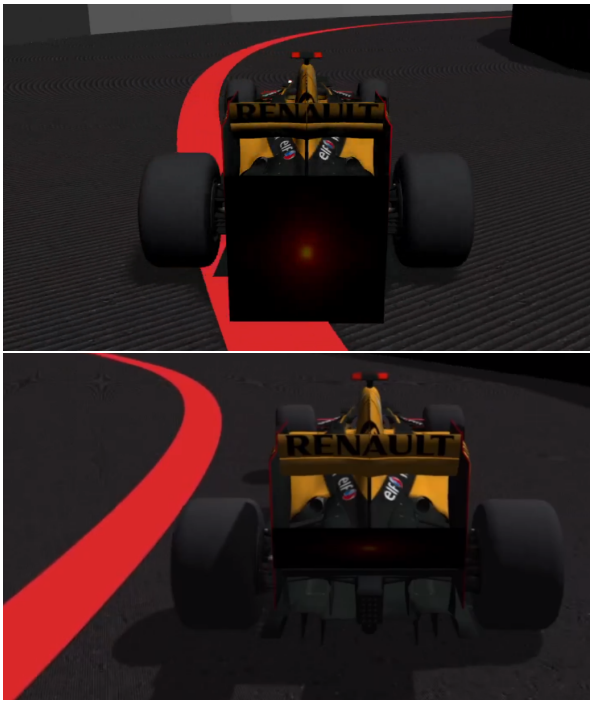


Figura 5.6: Imágenes de comparación entre ejecuciones del Sigue Carril holonómico (arriba) y Ackermann (abajo) en una curva.

- El coche no se encuentra bien posicionado en el carril (Figura 5.7): el modelo es capaz de comandar velocidades negativas cuando no se encuentra la línea roja que sirve como referencia del carril.

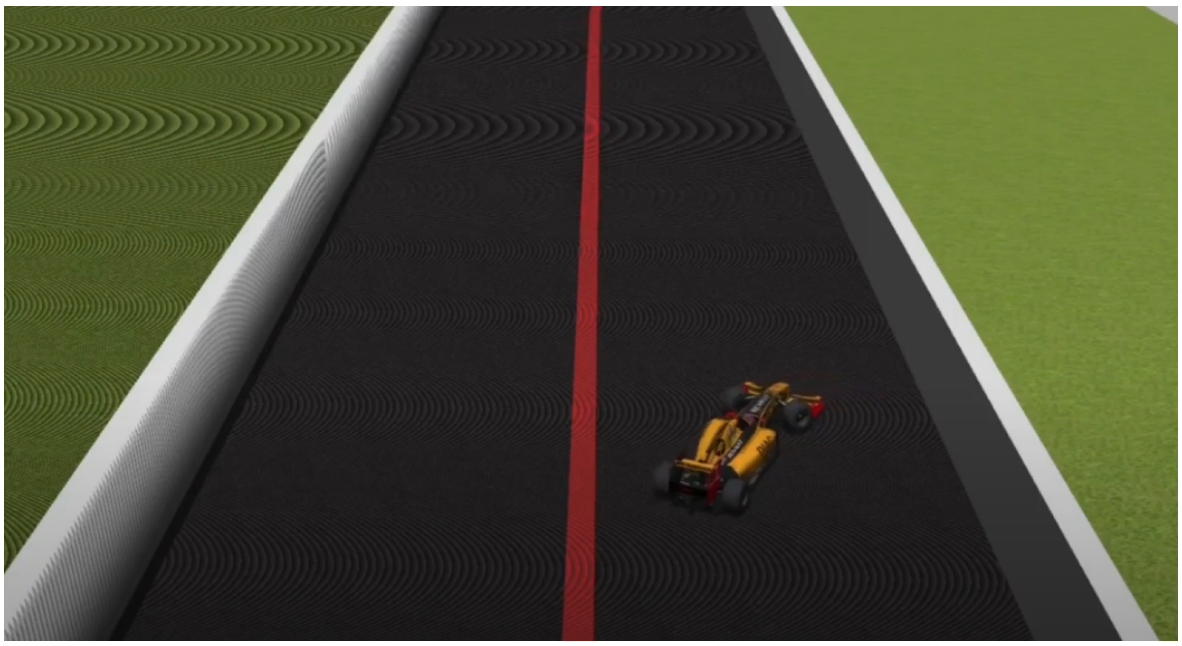


Figura 5.7: Coche de Ackermann dando marcha atrás para posicionarse centrado en el carril.

5.5. Configuración del AWS DeepRacer real

El AWS DeepRacer es un robot que ya se ha utilizado para experimentar y explorar soluciones en el campo de la conducción autónoma [11]. Para la configuración, se utilizó ROS2 Foxy (una distribución de ROS 2 anterior que Humble) para controlar y obtener información del robot AWS DeepRacer. Para conseguir los componentes de ROS 2 para comunicarse con los motores y la cámara, utilizamos los mismos que permiten la integración del *Navigation Stack*[3] de ROS 2. Asimismo, para comunicarnoValidacions con los motores y la cámara desde la aplicación programada utilizamos el módulo HAL.

La principal diferencia entre el coche con dinámica de Ackermann simulado utilizado (Figure 4.2 (derecha)) y el DeepRacer real es la escala. Ambos son modelos de coche Ackermann y tienen una interfaz de ROS 2 que hace posible la comunicación entre el proceso, los actuadores y los sensores. Esto hizo que la migración de la aplicación de uno a otro fuera bastante sencilla.

Figura 5.8: Coche robótico AWS DeepRacer, con geometría de Ackermann.

La única cosa distinta entre los sistemas que requirió cierta adaptación fue la cámara. La imagen obtenida de la cámara oficial DeepRacer tiene una distorsión de ojo de pez, que se solucionó determinando la matriz de distorsión usando llas funciones del modelo de cámara de ojo de pez de OpenCV (ver sección 3.7) [4].Las matrices de distorsión para nuestra cámara real del robot son:

[3]https://github.com/aws-deepracer/aws-deepracer
[4]https://docs.opencv.org/4.x/db/d58/group__calib3d__fisheye.html

$$K = \begin{bmatrix} 502{,}482 & 0 & 320{,}49 \\ 0 & 502{,}454 & 238{,}255 \\ 0{,}0 & 0{,}0 & 1{,}0 \end{bmatrix} D = \begin{bmatrix} -0{,}087 \\ -0{,}292 \\ 0{,}678 \\ -0{,}348 \end{bmatrix}$$

5.5.1. Escenarios para el AWS DeepRacer real

Para el escenario real, construimos un pequeño circuito en el porche trasero de una casa, con líneas rojas para delimitar el carril, tal y como se muestra en la Figura 5.9. Tiene un total de 11 metros de longitud y 5 curvas con diferente radio y dirección. Parecía apropiado para el DeepRacer real y se utilizó para crear los conjuntos de datos y probar el modelo final.

Figura 5.9: Circuito utilizado para el experimento del Deepracer.

5.6. Conjuntos de datos para AWS DeepRacer real

El primer paso para obtener los conjuntos de datos fue programar un controlador explícito que pudiera completar el circuito. Como no obtuvimos buenos resultados con el enfoque del controlador experto original, basado en un PID, intentamos simplificarlo para tener solo dos tipos de salida para el acelerador, positiva o negativa, dejando la dirección como un control proporcional pero

redondeando los valores a un único dígito decimal (ver Lcapaz de imitar las capacidades humanas de manejo y control. Como vehículo autónomo, es capaz de percibir el medio que le rodea y navegar en consecuencia.isting 5.1).

```
if line_not_found(points) == True:
    speed, rotation = exception_case()
else:
    speed, rotation = normal_case(points)
```

Extracto de código 5.1: Declaraciones de control para el AWS Deepracer

Incluso entonces, el controlador no pudo completar la tarea por lo que optamos por reemplazarla con una aplicación teleoperada para generar con ella, grabando sus datos, el conjunto de datos supervisado. Se basa en la misma estructura que tenía el controlador explícito, pero sustituyendo el PID por las acciones de un teleoperador externo. Se puede ver una visión general del código utilizado para teleoperar en el Extracto de código 5.2. Para este controlador también permitimos solo dos tipos de salida para el acelerador y la dirección es un número con un solo dígito decimal.

```
key = pressed_key

if key == "w":
    linear_speed = Positive_throttle_value
if key == "s":
    linear_speed = Negative_throttle_value
if key == "a":
    if angular_speed < 0:
        angular_speed = 0.1
    elif angular_speed >= 0 and angular_speed < 1:
        angular_speed = angular_speed + 0.1
if key == "d":
    if angular_speed > 0:
        angular_speed = - 0.1
    elif angular_speed <= 0 and angular_speed > -1:
        angular_speed = angular_speed - 0.1
if key == "q":
    Exit_program()
```

Extracto de código 5.2: Pseudo-código del programa que registra las acciones del teleoperador en un teclado.

Sumados a los problemas de la cámara descritos en la sección 5.5, también nos encontramos con que el brillo no siempre era el mismo y debíamos tenerlo en cuenta al obtener el conjunto de datos (ver Figura 5.10).

Figura 5.10: Dos imágenes tomadas casi desde la misma posición en dos días diferentes y en momentos diferentes.

También recortamos las imágenes como en los casos simulados, ver Figura

5.11, y utilizamos los mismos tipos de aumentos de datos que los de la sección 4.2 (imágenes en espejo y difuminado gaussiano).

Figura 5.11: Imagen recortada utilizada para entrenamiento.

5.7. Modelo neuronal y entrenamiento de red para AWS DeepRacer real

En cuanto al modelo neuronal escogido y el proceso de entrenamiento de la red utilizamos los mismos que en los anteriores.

La única diferencia en este caso fue que el entrenamiento se realizó en una máquina (el ordenador con la tarjeta gráfica) que no era la misma que se utilizó para la inferencia de la fase de validación experimental (el AWS deepracer).

Para el entrenamiento, esta vez el conjunto de datos balanceado que se utilizó tenía alrededor de 20.000 imágenes, y la red se entrenó durante 500 épocas.

5.8. Validación Experimental de AWS DeepRacer real

Después de entrenar el modelo con los conjuntos de datos obtuvimos resultados bastante buenos para este circuito que se pueden ver en estos videos [5][6][7]. Algunas fotografías obtenidas durante las ejecuciones se pueden ver en la Figura 5.12.

Algunas características del modelo que se pueden comentar son:

[5]https://www.youtube.com/watch?v=QyGXuK_4xHc
[6]https://www.youtube.com/watch?v=x38dDHj7Stc
[7]https://www.youtube.com/watch?v=KW5HiTAqDVc

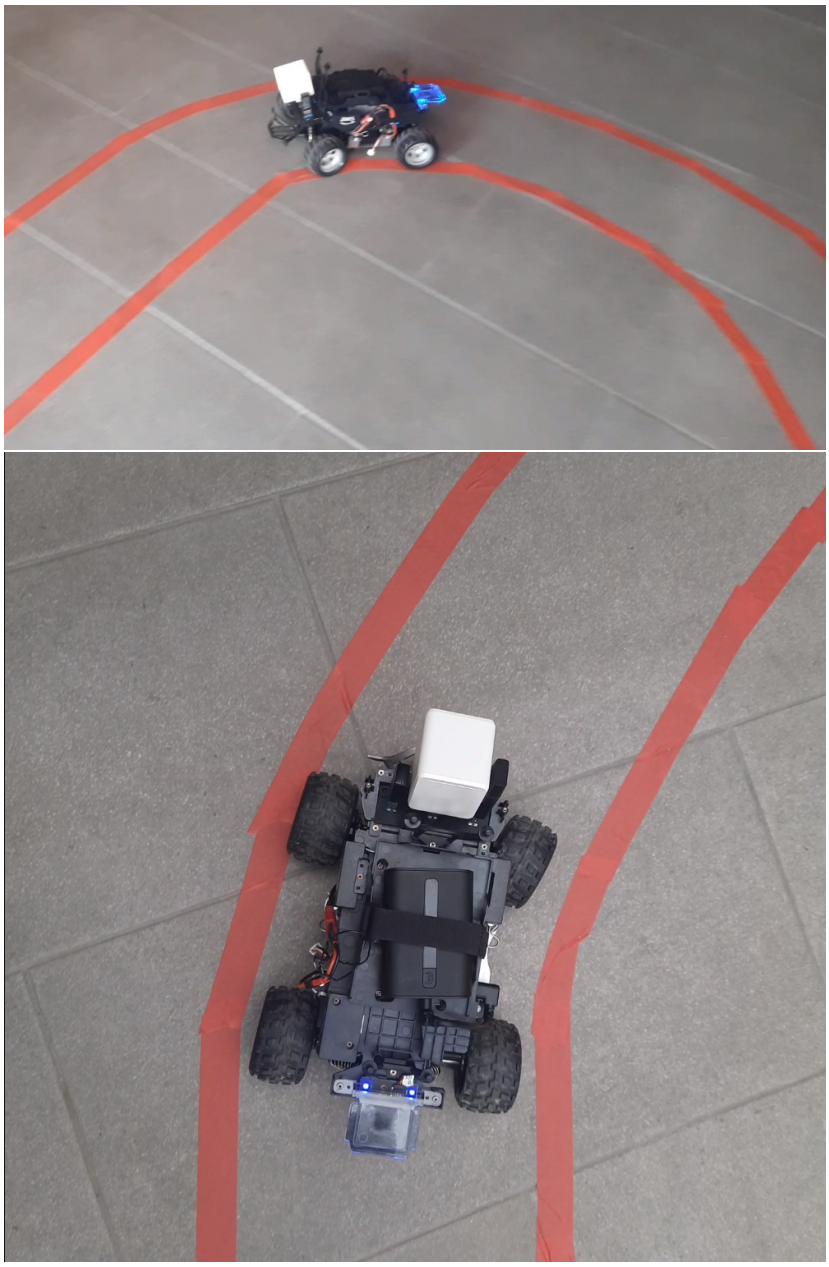


Figura 5.12: Imágenes del AWS DeepRacer durante la ejecución de la aplicación de red neuronal de sigue carril.

- El modelo presenta robustez ante ciertos cambios en las condiciones de luminosidad del entorno (ver Figura 5.13).
- El modelo es capaz de completar el circuito en un sentido y en el opuesto (ver Figura 5.14).
- El modelo recula y da marcha atrás cuando se sale del circuito tal y como hacía el controlador teleoperado (ver Figura 5.15).

Figura 5.13: Imágenes del AWS DeepRacer tomando una curva con dos zonas, una con menos luminosidad (derecha), y otra con más (izquierda).

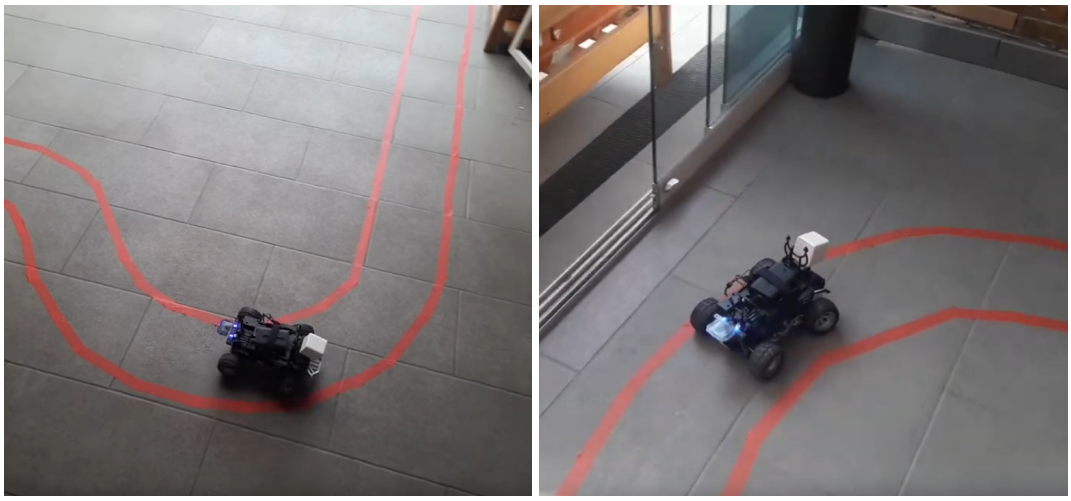


Figura 5.14: Imágenes del AWS DeepRacer siguiendo el carril en sentido horario (izquierda), y antihorario (derecha).

Figura 5.15: Imagen del AWS DeepRacer justo en el momento que empieza a recular para volver al carril.

También realizamos una evaluación *Offline* comparando los resultados de inferencia con los resultados supervisados, ver Figura 5.16.

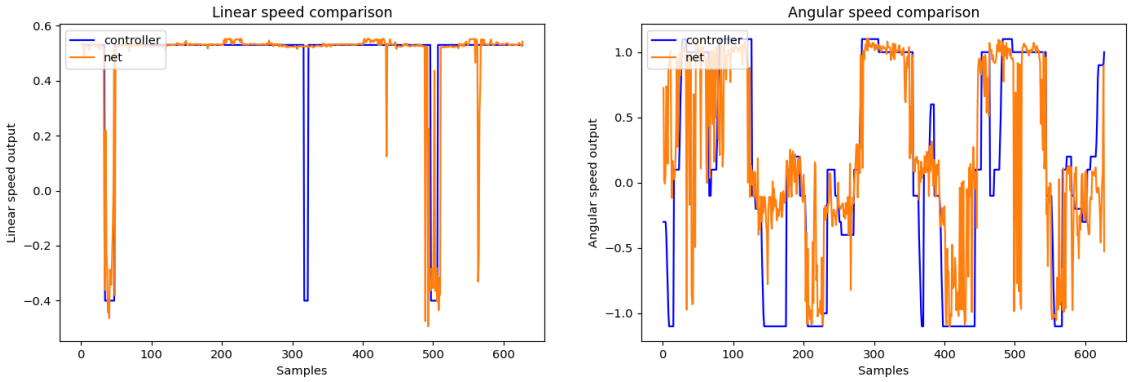


Figura 5.16: Comparación entre la salida del modelo entrenado (en naranja) y la salida del controlador experto (en azul) para la tarea de sigue carril del AWS DeepRacer real, usando unas 650 imágenes como datos de entrada

6. Conclusiones

En este capítulo se valorarán los resultados obtenidos en este TFG y se propondrán futuras vías de desarrollo relacionadas con los mismos.

6.1. Objetivos conseguidos

Con este trabajo hemos conseguido resolver satisfactoriamente las tareas de conducción autónoma propuestas mediante una programación indirecta de un controlador extremo a extremo usando Aprendizaje por Imitación. Siendo ambas soluciones aplicaciones reactivas basadas en la visión. Con el Sigue Línea, determinamos que el coche de Ackermann tiene un control más sensible y complicado que el holonómico. El Sigue Carril también se pudo resolver a pesar de ser más exigente, ya que es más fácil perder de vista la línea central si el coche se desvía al tomar una curva. Con estos dos primeros experimentos realizados en el simulador Gazebo obtuvimos la prueba de concepto para poder utilizar la misma metodología en un robot físico. En la prueba final intentamos acercarnos a una situación real de conducción autónoma, pero utilizando un coche pequeño (el AWS Deepracer), y abordando el problema del Sigue Carril. La aplicación conseguida con Aprendizaje por Imitación para este último también funcionó de manera exitosa, incluso teniendo un conjunto de datos menor y mostrando una robustez notable a condiciones variadas de iluminación.

Los modelos han sido validados experimentalmente incluso en circuitos no usados para su entrenamiento. Teniendo en cuenta estos resultados, podemos confirmar que la arquitectura de red *PilotNet* es capaz de lograr múltiples objetivos relacionados con la conducción autónoma.

Finalmente, para obtener los conjuntos de datos para el Aprendizaje por Imitación, demostramos que se puede utilizar tanto un agente experto programado explícitamente como un agente basado en acciones teleoperadas para generarlos. Cuanto más amplio y heterogéneo sea el conjunto de datos, mejor.

Todos los *scripts*, modelos, redes, simulaciones y otro contenido es *software* libre y se puede encontrar en [1].

[1] https://github.com/RoboticsLabURJC/2022-tfg-alejandro-moncalvillo

6.2. Trabajos futuros

Con el método de Aprendizaje por Imitación validado experimentalmente para ambos escenarios (virtual y físico), proponemos las siguientes vías de acción futuras:

- Probar soluciones basadas en de Aprendizaje por Imitación en el simulador CARLA[2], que es un estándar dentro de lacomunidad internacional de investigación en conducción autónoma.
- Probar técnicas de Aprendizaje por Transferencia con el robot real, de modo que se utilicen los datos simulados como base para el entrenamiento modelo del coche real y se pruebe el modelo entrenado con ellos en otro circuito real completamente distinto.
- Utilizar la misma metodología de Aprendizaje por Imitación para resolver tareas más sofisticadas como seguir el carril adaptándose al tráfico, otros vehículos en el mismo carril, señales de tráfico, etc.
- Explorar el alcance de las aplicaciones de control extremo a extremo basadas en de Aprendizaje por Imitación en otros entornos, como por ejemplo con robots aéreos como los drones (que ya se está investigando en [3]).

[2]https://carla.org//
[3]https://github.com/RoboticsLabURJC/2023-tfg-adrian-madinabeitia

Bibliografía

[1] Rodrigo Gutiérrez-Moreno, Rafael Barea, Elena López-Guillén, Felipe Arango, Navil Abdeselam, and Luis M. Bergasa. Hybrid decision making for autonomous driving in complex urban scenarios. In *2023 IEEE Intelligent Vehicles Symposium (IV)*, pages 1–7, 2023. doi: 10.1109/IV55152.2023.10186666.

[2] Ardi Tampuu, Tambet Matiisen, Maksym Semikin, Dmytro Fishman, and Naveed Muhammad. A survey of end-to-end driving: Architectures and training methods. *IEEE Transactions on Neural Networks and Learning Systems*, 33(4):1364–1384, 2020.

[3] Javier del Egio, Luis Miguel Bergasa, Eduardo Romera, Carlos Gómez Huélamo, Javier Araluce, and Rafael Barea. Self-driving a car in simulation through a cnn. In *Advances in Physical Agents: Proceedings of the 19th International Workshop of Physical Agents (WAF 2018), November 22-23, 2018, Madrid, Spain*, pages 31–43. Springer, 2019.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[5] Vanessa Fernández Martínez. Conducción autónoma de un vehículo en simulador mediante aprendizaje extremo a extremo basado en visión. Master's thesis, Máster Universitario Visión Artificial, Universidad Rey Juan Carlos, 2019.

[6] Sergio Paniego, Nikhil Paliwal, and JoséMaría Cañas. Model optimization in deep learning based robot control for autonomous driving. *IEEE Robotics and Automation Letters*, 9(1):715–722, 2023.

[7] Sergio Paniego, Enrique Shinohara, and JoséMaría Cañas. Autonomous driving in traffic with end-to-end vision-based deep learning. *Neurocomputing*, page 127874, 2024.

[8] Sergio Paniego, Roberto Calvo-Palomino, and JoséMaría Cañas. Behavior metrics: An open-source assessment tool for autonomous driving tasks. *SoftwareX*, 26:101702, 2024.

[9] 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):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[10] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004. doi: 10.1109/IROS.2004.1389727.

[11] Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, Eddie Calleja, Sunil Muralidhara, and Dhanasekar Karuppasamy. Deepracer: Autonomous racing platform for experimentation with sim2real reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2746–2754, 2020. doi: 10.1109/ICRA40945.2020.9197465.