



ESCUELA DE INGENIERÍA DE FUENLABRADA

GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

TRABAJO FIN DE GRADO

Conducción autónoma en CARLA basado en
aprendizaje por refuerzo

Autor: Juan Camilo Carmona Sánchez

Tutor: Dr. Roberto Calvo Palomino

Curso académico 2023/2024

Agradecimientos

Madrid, 30 de junio de 2023

Juan Camilo Carmona Sánchez

Resumen

La conducción autónoma representa una de las revoluciones tecnológicas más grandes y significativas del siglo XXI. Los pequeños avances que se logran día a día en este ámbito nos ponen un poquito más cerca de un futuro que tiempo atrás parecía utópico e inalcanzable, en el que las personas podremos pasarle el testigo de la movilización humana a las máquinas y dejarlas encargadas por completo de nuestro transporte a lo largo de ciudades, países y continentes. En este proyecto se busca aportar un pequeño avance más de los ya mencionados, explorando la aplicación del aprendizaje por refuerzo (RL, por sus siglas en inglés) en el ámbito de la conducción autónoma, utilizando el simulador CARLA como plataforma experimental y de desarrollo. CARLA,

con su entorno realista y parámetros de control finamente detallados, ofrece un terreno fértil para investigar cómo los agentes basados en RL pueden aprender políticas de conducción eficientes, seguras y, sobre todo, autónomas a partir de la interacción con su entorno, sin necesidad de indicaciones previas ni ningún tipo de razonamiento humano detrás de las decisiones efectuadas en cada momento. Esta línea de desarrollo se alinea con el núcleo conceptual de los vehículos autónomos, que deben ser capaces de adaptarse y responder a situaciones imprevistas en tiempo real. En este trabajo se describe de

manera detallada y profunda la creación, actuación, rendimiento y comparación de un agente de aprendizaje por refuerzo dotado de la capacidad para aprender por sí mismo a navegar de manera fluida y acertada por un carril de carretera. Se abordan desafíos específicos relacionados con la alta dimensionalidad del espacio de acción y observación, la naturaleza estocástica del tráfico tanto en el entorno urbano como el interurbano y la necesidad de proveer un comportamiento que pueda garantizar la integridad tanto del vehículo como de los posibles pasajeros que puedan ocupar este mismo. Para lograr esta hazaña, se implementarán técnicas de aprendizaje por refuerzo junto con algoritmos avanzados de real-time y redes neuronales, entre otros efectos del mundo de la inteligencia artificial. El análisis de los resultados obtenidos pone de manifiesto la capacidad del agente para aprender políticas de conducción complejas, así

como los retos inherentes al equilibrio entre exploración y explotación en un dominio donde los errores pueden tener consecuencias significativas. Se discuten las limitaciones actuales del enfoque y se esbozan direcciones para futuras investigaciones, incluyendo la integración de otras fuentes de información y la adaptación a condiciones de conducción más desafiantes. Además, todo esto se compara con los resultados proporcionados por una amplia gama de métodos más tradicionales de conducción autónoma en los que la inteligencia artificial no está presente y el agente está sujeto a las indicaciones, etiquetas y reglas previamente definidas por el humano. En conclusión, este proyecto busca arrojar luz sobre el potencial del aprendizaje por refuerzo como herramienta para avanzar en el desarrollo de sistemas de conducción autónoma, al tiempo que subraya la importancia de la simulación y experimentación en entornos controlados y realistas como CARLA. Y, sobre todo, acercar a la sociedad humana la conducción del futuro.

Acrónimos

TFG Trabajo Fin de Grado

IA Inteligencia Artificial

PID Proporcional Integral Derivativo

GUI Intefaz Gráfica de Usuario

TF TensorFlow

FPS *Frames Per Second*

SAE *Sociedad de ingenieros automotrices*

ADAS *Sistemas avanzados de ayuda a la conducción*

ISA *Asistente inteligente de velocidad*

LKA *Sistema de mantenimiento de carril*

REV *Sistema detector de marcha atrás*

RL *Reinforcement learning*

DL *Deep learning*

RT *Real time*

ROS *Robot operating system*

TFM *Trabajo de fin de máster*

ROS *Robot Operating System*

Índice general

1. Introducción	1
1.1. La robótica	1
1.2. Los robots móviles	2
1.3. La conducción autónoma	3
1.4. La inteligencia artificial en la conducción autónoma	9
1.5. Conducción autónoma en CARLA basada en aprendizaje por refuerzo	10
2. Objetivos	11
2.1. Descripción del problema	11
2.2. Objetivos	11
2.3. Requisitos	12
2.4. Metodología	12
2.5. Plan de trabajo	13
3. Plataformas de desarrollo y herramientas utilizadas	15
3.1. Lenguajes de programación	15
3.1.1. Python	15
3.2. Entornos de programación	16
3.2.1. CARLA	16
3.2.2. Ros2	17
3.2.3. CARLA to ros bridge	18
3.2.4. Visual studio code	19
3.2.5. Servidor landau de la URJC	20
3.3. Herramientas de inteligencia artificial	21
3.3.1. Redes neuronales	21
3.3.2. Aprendizaje por refuerzo	22
4. Diseño	23
4.0.1. Arquitectura	24

4.1.	Instalación del entorno de trabajo	26
4.1.1.	Instalación de CARLA	26
4.1.2.	Instalación de ROS2	28
4.1.3.	Carla to ROS Bridge	28
4.2.	Creación de un teleoperador	30
4.2.1.	Teleoperador en CARLA basado en ROS2	31
4.3.	Sigue carril basado en visión artificial tradicional y un controlador PID	35
4.3.1.	Método 1: Filtro Canny	38
4.3.2.	Método 2: Filtro de color HSV + filtro Canny	40
4.4.	Sigue carril basado en inteligencia artificial y un controlador PID	51
4.4.1.	Método 1: Redes neuronales	51
4.5.	Comparación de los métodos de detección	54
4.6.	Evaluación de la Compatibilidad entre ROS y CARLA a través de ROS Bridge	57
4.7.	Sigue carril basado en redes neuronales y Qlearning	58
4.7.1.	Sigue carril tradicional basado en Q-learning	58
4.7.2.	sigue carril con reacción a obstáculos en la vía	64
4.7.3.	Sigue carril con adaptación al tráfico	69

Bibliografía**71**

Índice de figuras

1.1.	Disciplinas que componen la robótica	1
1.2.	Ilustración de un robot móvil generado por IA	2
1.3.	Ilustración de coches autónomos generada por Inteligencia Artificial (IA)	4
1.4.	Niveles de conducción autónoma según el estándar J3016	6
1.5.	Vehículo tesla	7
1.6.	Interior de un robotaxi de Waymo	7
1.7.	Ilustración de una red neuronal	10
2.1.	Ilustración de la metodología scrum	13
3.1.	Imagén de vehículos en CARLA	17
3.2.	logotipo de Ros2	18
3.3.	carla to ros bridge	19
3.4.	carla to ros bridge	19
4.1.	Arquitectura del TFG	26
4.2.	Logotipoo de ros2 foxy	28
4.3.	HMI de un teleoperado para un vehículo de CARLA	32
4.4.	Gráfica de tasa de fps del sigue carril basado en canny	39
4.5.	Gráfica de uso de cpu del sigue carril basado en canny	40
4.6.	Gráfica de tasa de fps del sigue carril basado en canny + hsv	43
4.7.	Gráfica de uso de cpu del sigue carril basado en canny + hsv	43
4.8.	Gráfica de tasa de fps del sigue carril basado en algoritmo sliding window	50
4.9.	Gráfica de uso de cpu del sigue carril basado en algoritmo sliding window	50
4.10.	Gráfica de tasa de fps del sigue carril basado en deeplearning	53
4.11.	Gráfica de uso de cpu del sigue carril basado en deeplearnig	53
4.12.	Gráfica de comparación de los FPS de todos los algoritmos	55
4.13.	Gráfica de ruta seguida por todos los algoritmos siguiendo un carril	55
4.14.	Gráfica de los tipos movimientos de los PID de todos los algorimtos	56
4.15.	Gráfica de la función ecdef de la insensidad de los giros de cada algoritmo	56

4.16. Estados del algoritmo de Q-learnig	59
4.17. Entorno de entrenamiento del algoritmo de Q-learnig	61
4.18. Evolución del ratio de exploración del algoritmo Q-learnig	62
4.19. Evolución de la recompensa del algoritmo Q-learnig	63
4.20. Histograma de las acciones tomadas por el agente de Q-learning tras el entrenamiento	63
4.21. Evolución del ratio de exploración del algoritmo Q-learnig	68
4.22. Histograma de acciones dl algoritmo sigue carril con reacción ante obstáculos	69

Listado de códigos

3.1.	Hola Mundo en Python	15
4.1.	Añadir repositorio de CARLA al sistema	27
4.2.	Comandos para instalar CARLA	27
4.3.	Comando para lanzar el simulador CARLA	27
4.4.	git clone de carla to ros bridge 1	29
4.5.	Como incluir carla to ros bridge en un CMake.list	29
4.6.	Ejemplo de uso del carla to ros bridge	30
4.7.	Eventos de pygame para controlar el teleoperado	34
4.8.	Controlador PID de los sigue líneas basados en visión artifcial	37
4.9.	Controlador de velocidad fija del vehículo	38
4.10.	Implementación de filtro canny	39
4.11.	Rangos de detección del filtro de color HSV	41
4.12.	Filtro de color HSVI	42
4.13.	Umbralización de color y gradiente	44
4.14.	Transformación de perspectiva	45
4.15.	Inversión de la transformación de perspectiva	45
4.16.	Filtrado de carril mediante algoritmo sliding window	47
4.17.	Filtrado de carril mediante algoritmo sliding window	48
4.18.	Filtrado de carril mediante algoritmo sliding window	49
4.19.	Pipeline completo de filtrado de carril	49
4.20.	Acciones disponibles para el algoritmo de qlearning	60
4.21.	Función recompensa del algoritmo de qlearning	61
4.22.	Función para actualizar la tabla Q del algoritmo con reacción a obstáculos	65
4.23.	Función para elegir una acción de la tabla Q del algoritmo con reacción a obstáculos	65
4.24.	Función recompensa del algoritmo con reacción a obstáculos	67

Índice de cuadros

Capítulo 1

Introducción

1.1. La robótica

La robótica es una disciplina compleja que se conforma de distintas áreas y campos de la ciencia y la tecnología. Entre ellas, las más destacables serían la mecánica, la electrónica y la informática, la ingeniería de control, la física y la inteligencia artificial. Todas estas disciplinas se unen en una que pretende idear, diseñar y construir robots, máquinas capaces de realizar de manera automática y preferiblemente autónoma una o un conjunto de tareas para las cuales esta ha sido designado [1]

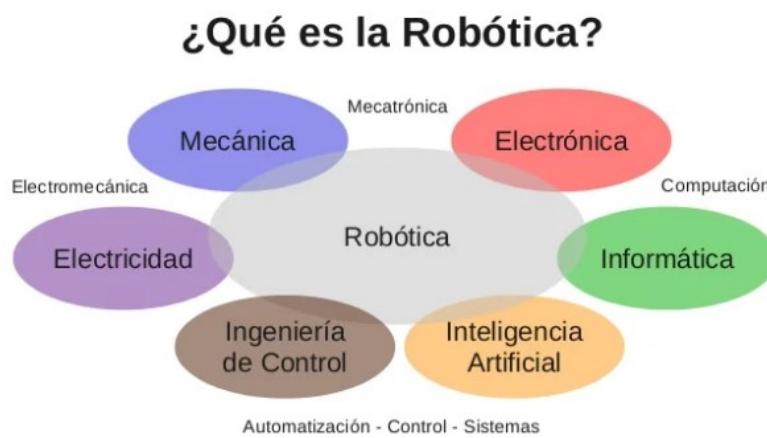


Figura 1.1: Disciplinas que componen la robótica.

Con los recientes avances en mecatrónica, informática e inteligencia artificial, la robótica ha encontrado su lugar en una amplia gama de sectores. En el sector industrial, los brazos robóticos desempeñan roles cruciales en cadenas de montaje, automatizando y refinando diversos procesos. Paralelamente, en el sector doméstico, la robótica ha transformado nuestras rutinas diarias: desde aspiradoras inteligentes que se desplazan

con precisión por nuestros hogares, hasta avanzados robots de cocina que simplifican la preparación de alimentos.

1.2. Los robots móviles

Los robots, a lo largo de su evolución, han sido clasificados de diversas maneras en función de su diseño, propósito y contexto de aplicación. Los robots humanoides, con su semejanza a la figura humana, buscan emular nuestros movimientos y comportamientos para adaptarse a nuestro mundo. Existen también robots colaborativos que, en lugar de reemplazar a los humanos, están diseñados para trabajar junto a nosotros en entornos compartidos. También podemos distinguir los ya mencionados robots industriales, sin embargo y a pesar de esta diversidad, existe una clasificación que destaca sobre el resto, debido a su gran importancia en la historia de la robótica y a su continuo desarrollo, estos son los robots móviles. Un robot móvil es un sistema robótico que puede desplazarse en distintos entornos y que cuenta con distintas capacidades que les permiten ejecutar tareas complejas, ya sea de forma autónoma o controlados por un operador humano. [2]

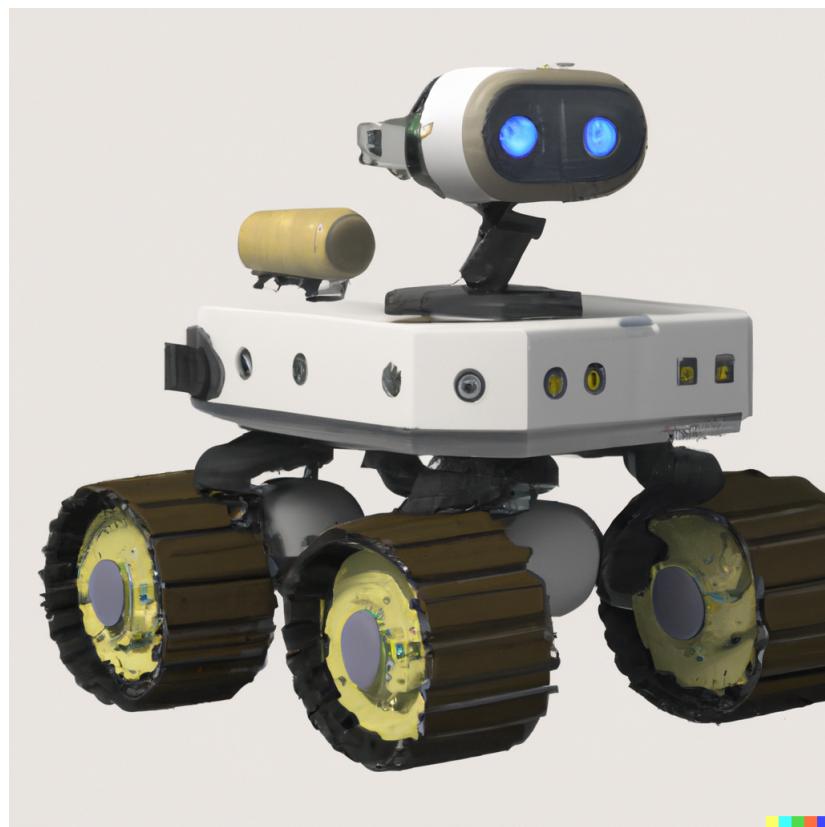


Figura 1.2: Ilustración de un robot móvil generado por IA.

A diferencia de los robots fijos, que permanecen estacionarios y realizan tareas desde una posición establecida, los robots móviles cuentan con la capacidad de desplazarse de un sitio a otro. Esto les otorga una libertad sin precedentes y les permite realizar todo tipo de acciones que para muchas otras máquinas resultan imposibles. Esta versatilidad se debe, en gran parte, a avanzados sistemas de sensores que les permiten percibir su entorno; algoritmos de navegación y control que delinean sus trayectorias; sistemas de comunicación que los conectan con otras máquinas, bases de datos y teleoperadores; y a sofisticados actuadores, tales como ruedas, patas, hélices y un largo etcétera, en función de a qué medios deba adaptarse el robot.

En el panorama de la robótica móvil, a pesar de la diversidad y versatilidad de sus aplicaciones, una categoría ha destacado del resto. Los vehículos autónomos son, probablemente, los protagonistas de la robótica móvil del siglo XXI. Estos robots son vehículos equipados con sofisticados sensores que les permiten percibir su entorno, procesar esa información mediante algoritmos avanzados para reaccionar ante él de manera inteligente, imitando el comportamiento humano. Estos vehículos consiguen esta gran hazaña gracias a una característica muy importante: la conducción autónoma.

1.3. La conducción autónoma

La conducción, en su esencia, se refiere a la acción de guiar o controlar un vehículo, ya sea motorizado o no, con la finalidad de trasladarse de un lugar a otro. Desde tiempos prehistóricos, la humanidad ha tenido la necesidad de trasladarse y transportar bienes, materias primas y otras personas. Esta necesidad impulsó la creación de vehículos sencillos como carros tirados por animales o vehículos impulsados por la propia potencia muscular del conductor, como las bicicletas.

Con el paso del tiempo y el avance de la ingeniería y la ciencia en el mundo de la automoción, en el siglo XIX aparecieron los primeros vehículos motorizados. Estos prototipos, movidos inicialmente por vapor y luego por combustibles fósiles, marcaron el inicio de una revolución en la movilidad y transformaron la manera en que las personas y mercancías se desplazaban. Sin embargo, estos vehículos requerían determinadas habilidades manuales y cognitivas por parte del conductor, además de amplios conocimientos de la máquina que se operaba. Así, la conducción se convirtió en una habilidad que las personas debían estudiar, practicar y aprender. Adicionalmente, el acto de conducir, para ser ejecutado de manera correcta y segura, requiere de atención, calma y claridad mental, estados que en ocasiones resultan difíciles de

mantener para los seres humanos, sobre todo en situaciones desconocidas, inciertas y estresantes, situaciones que son el pan de cada día de cualquier conductor. Todo esto ha hecho que la conducción sea una de las habilidades más difíciles de adquirir y a la vez más valoradas en la actualidad.

Debido a esto, la idea de vehículos que pudieran conducirse por sí mismos, sin necesidad de intervención humana y de las habilidades y atención del conductor, ha sido una aspiración de la humanidad por mucho tiempo, muy probablemente desde el inicio mismo de la conducción. Sin embargo, no fue hasta finales del siglo XX cuando esta idea empezó a parecer viable. Durante este período, la emergencia de la computación, la inteligencia artificial y una amplia gama de sensores avanzados resultaron en vehículos capaces de interpretar su entorno, tomar decisiones y operar sin intervención humana directa en ciertas condiciones estableciendo así el inicio de la conducción autónoma. La conducción autónoma se define como la capacidad total o parcial de que un vehículo pueda conducir autonomamente sin necesidad de un operador externo. [?]

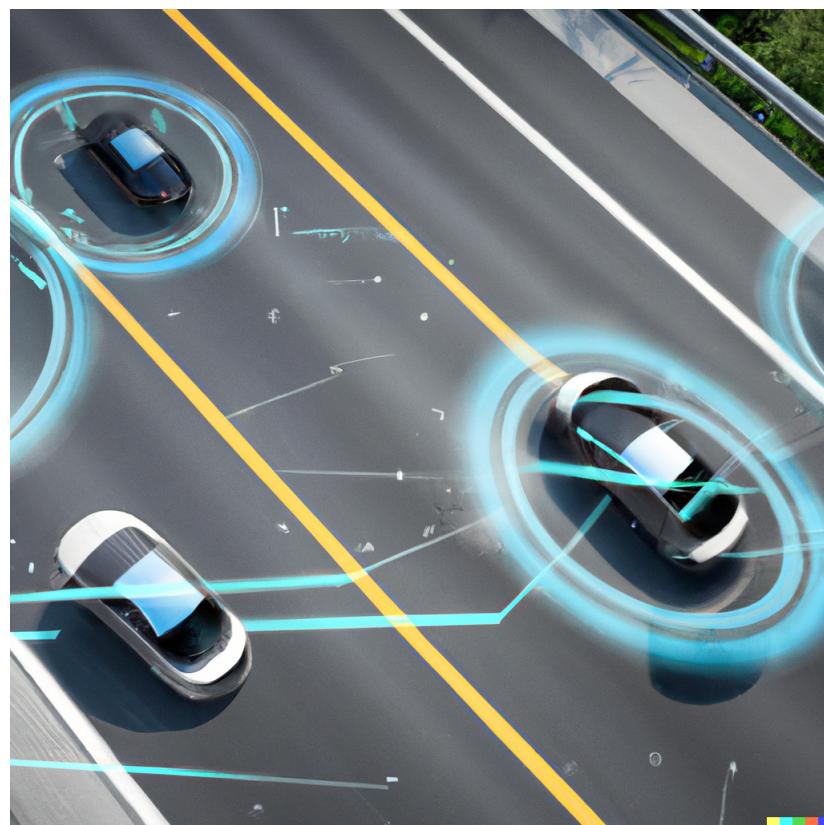


Figura 1.3: Ilustración de coches autónomos generada por IA.

Existen 6 niveles de conducción autónoma según el estándar establecido por la *Sociedad de ingenieros automotrices* (SAE) en el estándar J3016 [3]

1. **Nivel 0 (No Automation)**: El conductor humano es responsable de todas las tareas de conducción, incluso si el vehículo ofrece alguna intervención momentánea.

2. **Nivel 1 (Driver Assistance)**:

El vehículo puede asistir al conductor en una única tarea de conducción (por ejemplo, control de crucero). El conductor sigue siendo responsable de la mayoría de las tareas y debe estar atento en todo momento.

3. **Nivel 2 (Partial Automation)**:

El vehículo puede controlar simultáneamente dos tareas, como dirección y aceleración. A pesar de esta automatización, el conductor debe supervisar el sistema en todo momento.

4. **Nivel 3 (Conditional Automation)**:

El vehículo puede realizar la mayoría de las tareas de conducción en ciertas condiciones, pero requerirá intervención humana cuando el sistema lo solicite. El conductor debe estar disponible para tomar el control, pero no necesita tener las manos en el volante todo el tiempo.

5. **Nivel 4 (High Automation)**:

En ciertos escenarios o zonas geográficas específicas, el vehículo puede manejar todas las tareas de conducción. Fuera de estas zonas, el vehículo podría requerir que el conductor tome el control.

6. **Nivel 5 (Full Automation)**:

El vehículo es completamente autónomo en todos los escenarios y condiciones. No necesita un volante, pedales ni un conductor humano.

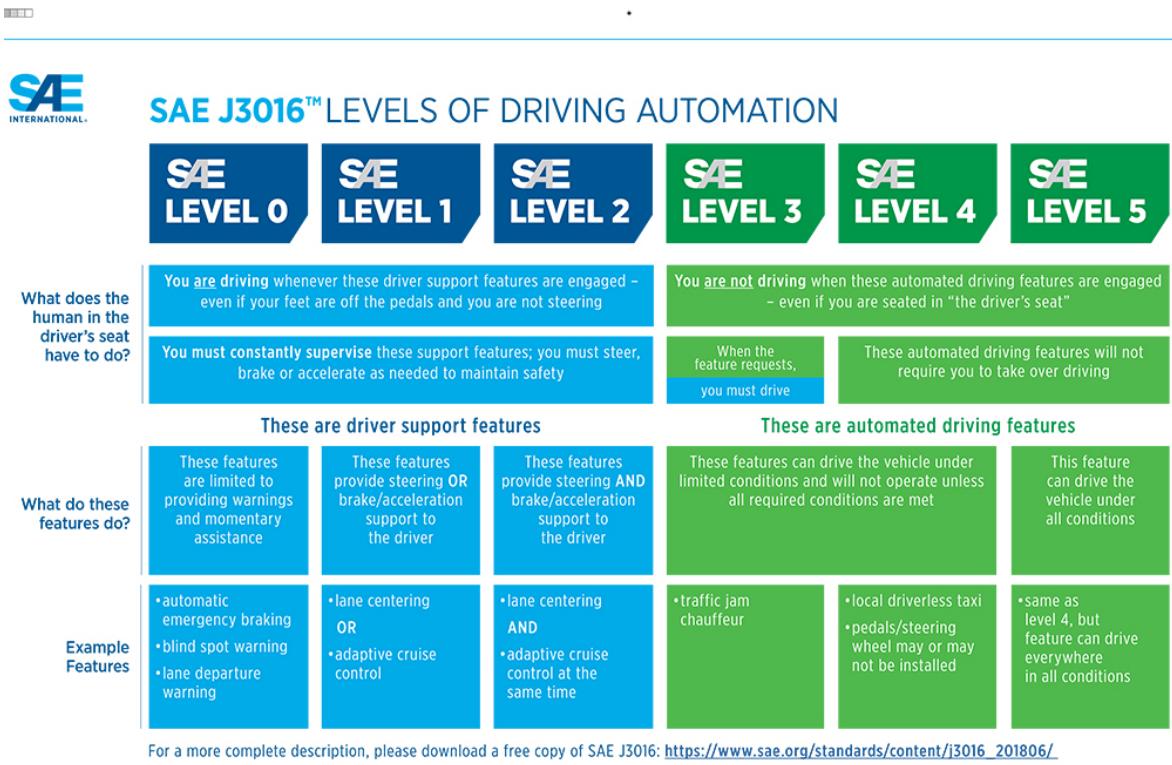


Figura 1.4: Niveles de conducción autónoma según el estándar J3016.

El final del siglo XX y el comienzo del siglo XXI marcaron una era significativa en el avance de la conducción autónoma. Los primeros sistemas y algoritmos para la conducción autónoma derivaban de los sistemas *Sistemas avanzados de ayuda a la conducción* (ADAS). Según la página oficial de la DGT [4] Estos sistemas representan un conjunto de tecnologías integradas en vehículos que no solo mejoran la seguridad sino también la experiencia del conductor. Operan con diferentes grados de autonomía y pueden influir en múltiples funciones del vehículo, como frenado, aceleración, dirección y señalización. Por ejemplo, sistemas como el *Asistente inteligente de velocidad* (ISA), que regula constantemente la velocidad del vehículo; el *Sistema detector de marcha atrás* (REV), que alerta sobre obstáculos al retroceder; y el *Sistema de mantenimiento de carril* (LKA), que asegura que el vehículo permanezca dentro de un carril, sitúan a los coches que los incorporan dentro de los primeros 2 niveles de autonomía. No obstante, con el auge de la inteligencia artificial, la emergencia de las redes neuronales y los avances en computación, los sistemas ADAS han evolucionado con rapidez. Este progreso ha permitido a empresas como Tesla desarrollar los primeros vehículos comerciales de nivel 2. En cuanto a los niveles de autonomía más avanzados tenemos a compañías como Waymo, quien como se explica en ese artículo académico [5] está embarcado en un proyecto para crear una flota de taxis autónomos.



Figura 1.5: Vehículo tesla.

Los llamados robotaxis [6] , son actualmente una serie de prototipos de taxis autónomos de nivel de autonomía 4 que operan en algunas ciudades de Estados Unidos. Estos vehículos estan pensados para transportar pasajeros de manera completamente autónoma, sin la necesidad de un conductor humano, de hecho estos taxis futuristas no incluyen pedales ni volante como se puede observar en la figura 1.6. Los robotaxis actualmente aún son prototipo que realizan viajes con pasajeros reales seleccionados en algunas zonas de Phoenix y San Francisco con el fin de recopilar datos y en general probar y madurar esta nueva tecnologia para así en un futuro próximo convertir los robotaxis en un producto comercial que revolucione el transporte en las ciudades



Figura 1.6: Interior de un robotaxi de Waymo.

Sin embargo, a pesar de todos los avances de los últimos años en el ámbito de la conducción autónoma, esta problemática sigue sin estar completamente solucionada y los vehículos de nivel 4 y 5 de autonomía todavía son solo prototipos y no productos comerciales certificados y testados como podemos leer en este TFM [7]. Esto se debe a que la conducción autónoma es una de las áreas más desafiantes dentro de la ingeniería y la robótica, dada la inmensa complejidad y variabilidad de los escenarios en los que estos vehículos deben operar. Estos entornos no son estáticos, sino que están en constante cambio y movimiento. Carreteras en constante transformación, condiciones meteorológicas cambiantes, variaciones de luz, obstáculos inesperados y una amplia variedad de usuarios de la vía, desde peatones hasta ciclistas y otros vehículos, hacen que la carretera sea uno de los entornos más impredecibles. Añadiendo una capa adicional de complejidad, se encuentra el factor humano. No solo los vehículos autónomos deben anticipar y responder a las acciones de los conductores humanos, que pueden ser a menudo ilógicas o imprevistas, sino que además deben garantizar la máxima seguridad para los peatones y otros usuarios de la vía. Convivir en un espacio compartido con seres humanos requiere de un cuidado y precisión extraordinarios, pues el más mínimo error podría tener consecuencias extremadamente graves como se menciona en este artículo académico en la revista nature [8]

No obstante, una conducción autónoma perfecta promete revolucionar radicalmente el paradigma de movilidad y seguridad en nuestras carreteras tal y como explica este artículo publicado en la revista ScienceDirect [9]. La mayoría de los accidentes en la carretera son causados por errores humanos, ya sea por distracción, fatiga o decisiones erróneas en situaciones críticas. Los vehículos autónomos, operando con una combinación de sensores avanzados y algoritmos sofisticados, tienen el potencial de minimizar estos errores, reaccionando más rápidamente y de manera más precisa que un humano ante situaciones imprevistas. Además, los sistemas de conducción autónoma podrían gestionar de forma más eficiente el flujo de tráfico. Al poder comunicarse entre sí, los vehículos podrían coordinarse para evitar atascos, optimizar el uso de carriles y reducir las congestiones, resultando en viajes más rápidos y eficientes para todos. Por último, pero no menos importante, liberar a los humanos de la tarea de conducir abre un mundo de posibilidades. Las personas podrían ocupar todo el tiempo que dedicamos hoy en día a la conducción en otras acciones más provechosas o entretenidas: leer, ver una película, trabajar o incluso descansar. Esto mejoraría la calidad de vida al proporcionar tiempo adicional para actividades personales o productivas aparte. En definitiva, las ventajas que nos presenta la conducción autónoma son múltiples y muy

interesantes; es una tecnología que, sin duda, cambiará el mundo.

1.4. La inteligencia artificial en la conducción autónoma

La IA ha desempeñado un papel fundamental en la evolución de la conducción autónoma, permitiendo que los vehículos interpreten su entorno, tomen decisiones y se adapten a situaciones cambiantes. Al principio, la IA en la conducción autónoma se basaba en algoritmos más básicos. Sin embargo, pronto se hizo evidente que para alcanzar los niveles más altos de autonomía era necesario un enfoque más sofisticado para gestionar la complejidad. Esto condujo a la adopción y desarrollo de técnicas más avanzadas.

Las redes neuronales se han convertido en herramientas esenciales en el campo de la conducción autónoma. Estas redes, que imitan la estructura y el funcionamiento de las neuronas humanas, son particularmente aptas para tareas como la detección y la identificación de objetos. Al ser entrenadas con grandes cantidades de datos, pueden reconocer patrones y categorizar objetos en tiempo real con una precisión impresionante como se muestra en el siguiente TFG de un estudiante de la universidad de ULPCC [10]

El aprendizaje por refuerzo es otra técnica de IA crucial en la conducción autónoma. Este método entrena modelos de IA a través de recompensas y penalizaciones, permitiéndoles aprender a tomar decisiones óptimas en situaciones específicas. Por ejemplo, un vehículo autónomo podría ser recompensado por evitar obstáculos y penalizado por acercarse demasiado a ellos, aprendiendo con el tiempo a navegar de manera más segura y eficiente. En la documentación de continuación se puede encontrar información más detallada sobre el aprendizaje por refuerzo [11]

El imitation learning, o aprendizaje por imitación, es una técnica que permite a los sistemas de IA aprender a partir de la observación directa de acciones realizadas por humanos u otros agentes. En el contexto de la conducción autónoma, se puede utilizar para enseñar a los vehículos a imitar las decisiones y acciones de conductores humanos experimentados en diversas situaciones, permitiendo que los vehículos aprendan comportamientos más naturales. En el *Trabajo de fin de máster* (TFM) que a continuación se cita se explora una solución para la conducción autónoma basada en imitation learning [12].

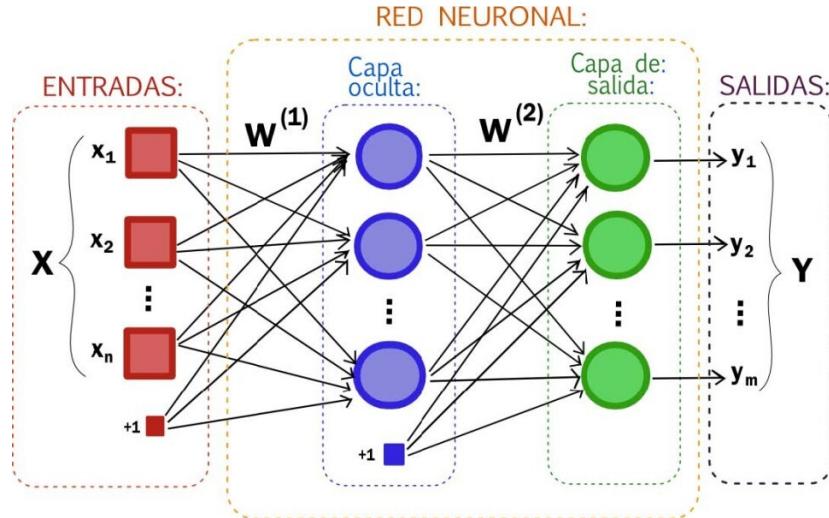


Figura 1.7: Ilustración de una red neuronal.

En resumen, la inteligencia artificial ha revolucionado significativamente el ámbito de la conducción autónoma. Las principales empresas tecnológicas y automotrices, como Waymo, Tesla y Cruise, han desarrollado y probado vehículos con niveles avanzados de autonomía, principalmente en niveles 2 y 3, y en algunos casos con capacidades limitadas de nivel 4 en entornos específicos. A pesar de los logros técnicos, la complejidad que representa navegar por nuestras carreteras sitúa aún lejos vehículos capaces de alcanzar niveles 4 y 5 completos de autonomía.

1.5. Conducción autónoma en CARLA basada en aprendizaje por refuerzo

El Trabajo de Fin de Grado (TFG) que a continuación se presenta aborda una profunda investigación académica sobre la conducción autónoma basada en aprendizaje por refuerzo. Se pretende solucionar varias problemáticas relacionadas con el seguimiento de carriles por un vehículo autónomo de manera fluida, elegante y segura. Por otro lado, aparte de este hito este Trabajo Fin de Grado (TFG) pretende llegar un paso mas allá mejorando este sistema y otorgandole la capacidad de evitar la colisión frontal con obstáculos que puedas aparecer en la carretera.

Finalmente, el TFG concluirá con la creación de un sistema incluso más avanzado de conducción autónoma aprovechando técnicas de inteligencia artificial. Este sistema estará especialmente diseñado para seguir carriles, evitar obstáculos y adaptarse al tráfico presente en la vía.

Capítulo 2

Objetivos

En el primer capítulo, se ha descrito e introducido el contexto en el que se enmarca este trabajo de fin de grado. En este segundo capítulo, que a continuación se expone, se procederá a exponer los objetivos específicos que se han establecido para este proyecto.

2.1. Descripción del problema

Como se ha mencionado en el primer capítulo, la conducción autónoma promete ser un avance tecnológico que cambie completamente la manera en la que se concibe la movilidad en nuestra sociedad. Este trabajo de fin de grado tiene como principal objetivo realizar una investigación sobre la utilización de distintas técnicas de inteligencia artificial, como son el deep learning y el aprendizaje por refuerzo, para la creación de una solución completa para la problemática de la navegación por un carril. Se presentará un comportamiento dotado de la capacidad de seguir un carril, adaptarse al tráfico de este y evitar colisiones en caso de que el tráfico se detenga.

2.2. Objetivos

1. Instalación y configuración del simulador CARLA, ROS2, logrando además una comunicación entre ambos.
2. Creación de un comportamiento sigue carril autónomo basado en *Reinforcement learning* (RL) y *Deep learning* (DL).
3. desarrollar un comportamiento para, además de navegar por un carril, detenerse antes de colisionar con obstáculos de la carretera.
4. Elaboración de un comportamiento que permita a un vehículo adaptarse a la velocidad del tráfico a la hora de navegar por un carril utilizando RL.
5. Análisis de métricas de cada método anteriormente mencionado y realización de una comparativa entre estas métricas.

2.3. Requisitos

Los requisitos que ha de cumplir este trabajo son los siguientes:

- El trabajo ha de realizarse en el simulador fotorealista de conducción autónoma CARLA.
- Los sistemas a desarrollar deben ser reactivos, es decir deben ser capaces de reaccionar a su entorno de manera rápida y precisa.
- El vehículo debe navegar de manera adecuada, natural y segura.

2.4. Metodología

El TFG comenzó en octubre de 2022 y finalizó en septiembre de 2023. A lo largo de estos 11 meses se siguieron las siguientes directrices para la realización del mismo:

- Reuniones semanales con mi tutor de TFG para establecer micro-objetivos semanales y reales. Gracias a estas reuniones, fue fácil mantener un control del avance del proyecto en cada momento e ir solucionando de manera rápida y efectiva los problemas que surgían.
- Realización de un blog¹, en el que se añadían periódicamente entradas con información sobre el avance del proyecto, además de los problemas enfrentados en cada etapa, a modo de bitácora para documentar todo el proceso de desarrollo.
- Se utilizó la plataforma de comunicación Microsoft Teams para realizar las reuniones periódicas con mi tutor del TFG y el correo de la universidad para mantener contacto con él en todo momento, con el fin de resolver problemas que pudieran surgir, notificar avances y solicitar consejos.
- Se empleó la plataforma de desarrollo GitHub para alojar todo el código desarrollado en el TFG².
- En el desarrollo de este TFG, se adoptó la metodología Scrum como sistema de trabajo. A lo largo del desarrollo de este trabajo, se establecieron diferentes sprints, cada uno de ellos con una duración determinada, durante los cuales se plantearon mini objetivos específicos a alcanzar. Esta estructura no solo permitió una organización y planificación eficiente del trabajo, sino que también ofreció

¹<https://roboticslaburjc.github.io/2022-tfg-juancamilo-carmona/>

²<https://github.com/RoboticsLabURJC/2022-tfg-juancamilo-carmona>

la flexibilidad necesaria para adaptarse a cambios y nuevos requerimientos que surgieron a lo largo del desarrollo del TFG.

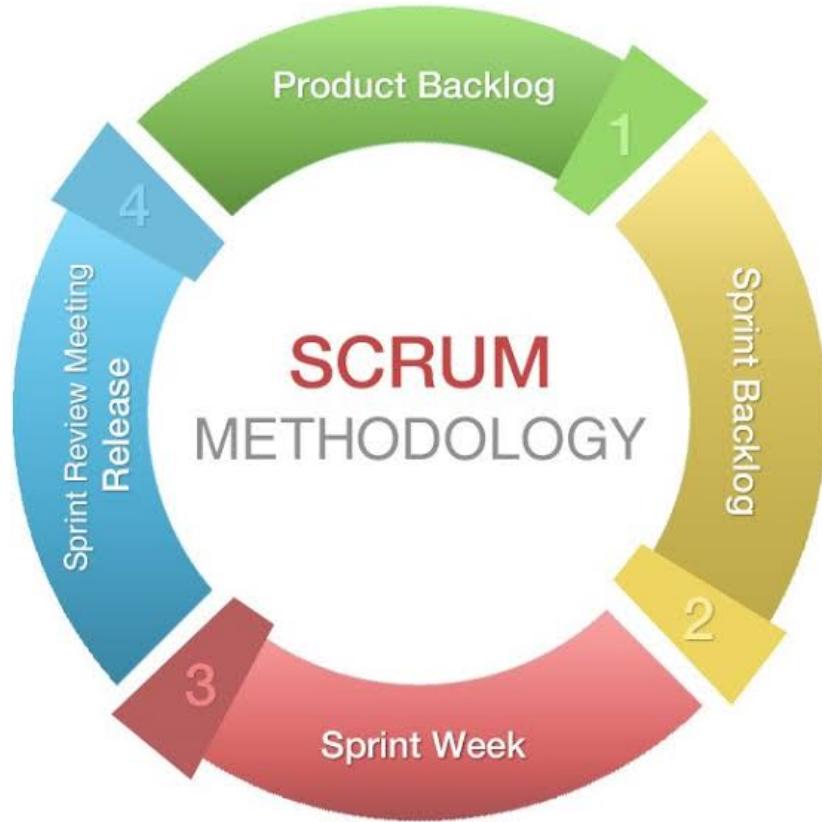


Figura 2.1: Ilustración de la metodología scrum.

2.5. Plan de trabajo

Durante los 11 meses en los que se ha desarrollado este proyecto, se ha seguido un plan de trabajo con la siguiente estructura:

1. Etapa de configuración: Esta etapa consistió básicamente en preparar todo el entorno para la realización del TFG. Aquí se instalaron todos los drivers, software y aplicaciones necesarias para empezar a trabajar.
2. Comienzo del TFG: Una vez el entorno de trabajo estaba listo, se empezó a desarrollar todo el código.
 - En primer lugar, se inició con el desarrollo de un teleoperador sencillo de un vehículo.
 - En segunda instancia, una vez el teleoperador estaba terminado, se continuó con los algoritmos de seguimiento de carriles basados en visión artificial tradicional.

- El siguiente paso después de explorar métodos más tradicionales, fue programar un sigue carril basado en DL y RL.
 - Una vez terminado el comportamiento sigue carril, se procedió a añadir un comportamiento el cual permitiera no chocarse al encontrar obstáculos en la carretera.
 - Finalmente se trabajó en una solución completa de conducción autónoma que se adaptara al tráfico, navegando por el carril correctamente, deteniéndose en caso de encontrar un vehículo detenido u obstáculo y adaptándose a la velocidad del tráfico sin colisionar ni detenerse en caso de que este existiera.
3. Una vez finalizado desarrollo, la siguiente etapa consistió en analizar las métricas de todos los algoritmos y compararlas para llegar a una conclusión sobre la mejor solución para la problemática.
 4. Para finalizar, se procedió a la redacción de la memoria del Trabajo Fin de Grado.

Capítulo 3

Plataformas de desarrollo y herramientas utilizadas

En el capítulo que a continuación se presenta se van a describir las distintas plataformas y herramientas que se han utilizado a lo largo desarollo de este TFG

3.1. Lenguajes de programación

3.1.1. Python

Python es un lenguaje de programación orientado a objetos, de alto nivel y fácil de interpretar, con una sintaxis sencilla y legible [13]. Actualmente, es uno de los lenguajes de programación más populares y resulta especialmente útil para el desarrollo de prototipos. Esto se debe a su facilidad de uso, que permite codificar algoritmos complejos de manera rápida y ágil. Además, es muy utilizado para desarrollar aplicaciones y algoritmos de inteligencia artificial, gracias a la gran cantidad de bibliotecas especializadas que este lenguaje ofrece [14]. Todo el código desarrollado para este TFG ha sido realizado íntegramente en Python, lo cual ha facilitado enormemente tanto la fase de investigación como la de implementación, permitiéndonos centrarnos más en la lógica de los algoritmos y menos en las complicaciones del lenguaje en sí. Esta elección ha demostrado ser acertada y ha contribuido significativamente al éxito del proyecto.

```
print("Hola Mundo")
```

Código 3.1: Hola Mundo en Python

3.2. Entornos de programación

3.2.1. CARLA

Dentro del extenso panorama de simuladores dedicados a la conducción autónoma, encontramos un sinfín de opciones. Sin embargo, entre toda esta variedad de simuladores, CARLA destaca como una de las herramientas más interesantes. CARLA es un simulador de conducción autónoma realista de código abierto que ha ganado prominencia en la comunidad de investigación y desarrollo, su alto grado de fotorealismo, el amplio control que ofrece sobre el entorno simulado y la gran cantidad de vehículos y sensores disponibles la convierten en una de las mejores elecciones a la hora de desarrollar y probar todo tipo de proyectos relacionados con la conducción autónoma. Fue creado por el equipo del Computer Vision Center en Barcelona en colaboración con Intel Labs y Toyota Research Institute. Desde su lanzamiento en 2017, CARLA ha sido una de las herramientas más prometedoras del mundo de la conducción autónoma.

La principal y más importante fortaleza de este simulador probablemente sea su nivel de fotorealismo. CARLA ofrece un alto nivel de detalle y precisión en sus simulaciones, que garantiza que los desarrolladores puedan probar algoritmos en condiciones que emulan fielmente situaciones del mundo real, como cambios de luz y de sombras, choques y colisiones, situaciones de altas y bajas velocidades, entre muchas otras características que sitúan a este simulador un paso por delante en cuanto a este aspecto se refiere se refiere.

Otra de las fortalezas más destacables de CARLA es su elevado número de escenarios. Estos abarcan desde entornos urbanos, como calles de barrios, rotundas, etc., hasta entornos interurbanos como autopistas y carreteras convencionales e incluso entornos rurales como granjas. Esto nos permite a los ingenieros y desarrolladores simular innumerables contextos y situaciones de conducción. Adicionalmente, CARLA brinda gran control sobre las condiciones ambientales. Los usuarios tienen la ventaja de ajustar aspectos tan vitales como la luminosidad, hora del día y condiciones climáticas, creando un espacio flexible y diverso para pruebas y experimentos especialmente aquellos relacionados con la IA.

La diversidad de activos probablemente sea la última de las fortalezas de CARLA. Su biblioteca contiene una amplia gama de vehículos, peatones y sensores, abarcando desde cámaras RGB hasta sistemas LIDAR, radares y electroópticos, ofreciendo así una rica paleta de herramientas para las simulaciones.



Figura 3.1: Imagén de vehículos en CARLA

CARLA es el corazón de este TFG es el escenario virtual en el que se desarrollan, prueban y evalúan todos los algoritmos y modelos diseñados para las problemáticas de conducción autónoma planteadas.¹

3.2.2. Ros2

ROS2, la evolución de segunda generación de ROS, es un meta sistema operativo de código abierto diseñado específicamente para ser utilizado en robots. Proporciona servicios que se esperarían de un sistema operativo, incluyendo abstracción de hardware, control de dispositivos de bajo nivel e implementación de funcionalidades [15]. Una de las características fundamentales de ROS es su modelo de comunicación basado en el sistema publicador-suscriptor. En este sistema, los nodos (componentes individuales de software en un sistema ROS) pueden comunicarse entre sí de manera asincrónica. Un nodo que tiene información para compartir se convierte en un publicador y envía mensajes a un topic. Otros nodos que estén interesados en recibir esa información se suscriben a ese y recibirán los mensajes publicados en él. Esto permite una comunicación eficiente y flexible entre los diferentes nodos que conformen nuestro sistema.

¹<https://carla.org/>

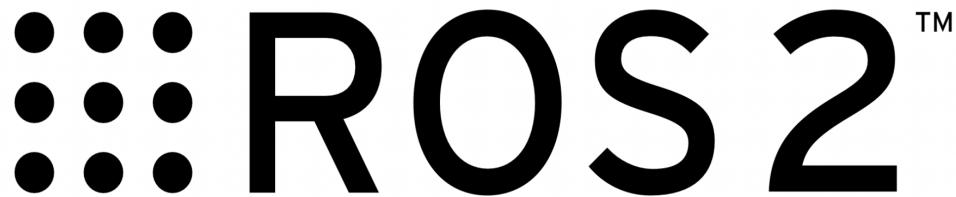


Figura 3.2: Logotipo de Ros2

En el desarrollo de este Trabajo de Fin de Grado, ROS 2 se utilizó para elaborar una sección específica del código y de los algoritmos de conducción autónoma. La decisión de integrar ROS 2 con el simulador CARLA tuvo un propósito muy claro: evaluar la viabilidad de unir las capacidades de este sistema, que es una referencia en el mundo de la robótica, con el entorno de simulación de conducción autónoma que CARLA ofrece. Esta integración se realizó con el objetivo de no solo generar algoritmos más robustos y eficaces sino también de explorar nuevas posibilidades y métodos para la creación de sistemas de conducción autónoma. Esta fusión entre ROS 2 y CARLA se convirtió en un elemento clave para entender y desarrollar más a fondo los complejos retos que plantea la conducción autónoma, permitiendo una interacción más fluida y adaptable entre los diferentes componentes del sistema.²

3.2.3. CARLA to ros bridge

Carla to ros brisge es una herramienta oficial desarrollada por el equipo de CARLA. Esta herramienta tiene el objetivo de ser, un medio de comunicación bidireccional entre CARLA y un entorno de desarrollo ROS o ROS2. Por un lado el bridge se encarga de traducir la información de CARLA a topics y mensajes de ROS de manera que el framework pueda entenderlos sin mayor dificultad. Por otro lado el bridge se encarga también de traducir los mensajes publicados en los topics generados para CARLA en mensajes basados en la API del simulador para que, de esta manera un nodo ROS pueda comunicarse con CARLA como lo haría con cualquier robot. carla to ros bridge puede ser encontrado de su repositorio oficial de github, en el aparte del código encontraremos intrucciones para su descarga y uso³

²<https://www.ros.org/>

³<https://github.com/carla-simulator/ros-bridge>

carla-simulator/ros-bridge

ROS bridge for CARLA Simulator



41

Contributors

98

Issues

377

Stars

287

Forks



Figura 3.3: carla to ros bridge

en este este TFG carla to ros bridge ha tenido un papel fundamental en la fase de investigación y experimentación sobre la viabilidad del desarrollo de algoritmos de conducción autónoma en ROS2 para el simulador CARLA, permitiendo que CARLA y ROS2 se comunicarán y haciendo posible.

3.2.4. Visual studio code

Visual studio code ⁴ es un editor de código gratis, ligero, profesional y fácil de usar disponible tanto para Linux, como para Windows, como para macOS. Probablemente representa la elección favorita de la gran mayoría de los programadores de hoy en día a la hora de elegir un editor de código. Tanto en ambientes de investigación como en entorno profesionales de producción de aplicaciones y programas Visial studio code es amnpliamente conocido y utilizado.



Figura 3.4: Visual studio code logo

⁴<https://code.visualstudio.com/>

Visual studio code ha sido el editor elegido para la programación de todo el código de este TFG. Los algoritmos, herramientas y aplicaciones que más tarde se explicarán han sido todas gestados en este famoso editor.

3.2.5. Servidor landau de la URJC

Una de las principales limitaciones que los ingenieros nos encontramos a la hora de trabajar con IA, es si duda, la necesidad de un Hardware potente para el correcto funcionamiento de esta. La ejecución y entrenamiento de algoritmos de inteligencia artificial y aprendizaje automático suelen ser computacionalmente muy pesado lo cual dificulta su ejecución de manera correcta y fluida sin el Hardware adecuado. Generalmente lo más importante en este aspecto es una GPU potente. En el caso del TFG que en esta memoria se presenta, existe el problema añadido de que como anteriormente se ha mencionado se utiliza el simulador fotorealista CARLA el cuál por otra parte también requiere un gran cantidad de memoria de GPU para poder funcionar. Estos dos factores hicieron que para realizar este tfg fuera vital disponer de una GPU muy potente capaz de correr CARLA y a la vez ser capaz de ejecutar algoritmos pesados de IA.

Esta problemática hubiera sido probablemente uno de los problemas más grandes a los que este TFG se hubiera tenido que afrontar, debido a que el hardware que se necesitaba era sumamente costoso y probablemente inasumible. Afortunadamente esto no fue así ya que La Universidad Rey Juan Carlos se ofreció a proporcionar el hardware necesario ofreciendo acceso a un servidor propio y privado de la Universidad el cuál permitía ejecutar código en racks de gráficas Nvidia alojadas en la misma universidad.

De esta manera y después de que me fueran otorgados los permisos para acceder a este servidor, este se convirtió en un pilar casi imprescindible de este TFG debido a que fue en esta plataforma donde se ejecutaron CARLA y todos los algoritmo y aplicaciones de conducción autónoma de una manera lo suficientemente fluida como para permitir un comportamiento *Real time* (RT).

3.3. Herramientas de inteligencia artificial

Como se ha mencionado con anterioridad una de los principales objetivos del trabajo realizado para este proyecto es, crear un sistema de conducción autónoma seguro y eficaz basado en inteligencia artificial. A continuación se describen las herramientas de inteligencia artificial utilizadas en la temtativa de lograr este objetivo

3.3.1. Redes neuronales

A principios del XXI hace no muchos años, más concretamente en el año 2011, los avances en el mundo del aprendizaje automático desembocaron en la creación de una nueva rama de esta disciplina, el llamado deep learning o aprendizaje profundo. El deep learning una rama hermana del machine learning, pero con tecnologías y técnicas mucho más sofisticadas. En este en vez de emplear algoritmos basados en regresión lineal o arreglos decisiones, como era típico en el machine learning, el DL se fundamentó en un nuevo concepto: Las redes neuronales profundas. Las redes neuronales, son un sistema informático que busca emular el funcionamiento de las conexiones neuronales biológicas generadas en el cerebro humano. Las redes neuronales profundas se pueden definir como un tipo de red neuronal, con un gran número de capas las cuales se transmiten y procesan la información entre sí generando un sistema complejo con mucha ‘profundidad’ de niveles lo que genera una complejidad añadida y las acerca más a imitar el comportamiento de neuronas reales.

Generalmente, una red neuronal se compone de tres partes:

1. Una capa de entrada, con nodos que simbolizan los campos de entrada. Estas son todas las variables que la red neuronal acepta y es capaz de tomar en cuenta
2. Una o varias capas ocultas, en el caso de las redes profundas suelen ser un número superior a 10 capas. Estas capas procesan la información de manera transparente para el usuario de la red
3. Una capa de salida, con un único nodo, o varios nodos, que simbolizan el campo o campos de destino. Las unidades establecen conexiones con fuerzas de conexión variables.

La primera capa recibe los datos de entrada, y los valores se transmiten de cada neurona a todas las neuronas de la siguiente capa y viceversa. Finalmente, toda la

información procesada se envía a la capa de salida y de esta se ofrece un resultado al usuario.

3.3.2. Aprendizaje por refuerzo

El aprendizaje por refuerzo (Reinforcement Learning, RL) es una rama del aprendizaje automático en la que un agente autónomo toma decisiones tratando de maximizar una recompensa acumulada a lo largo del tiempo. Funciona mediante la interacción del agente con un entorno. En cada paso temporal, el agente selecciona una acción basada en una política definida, por cada acción que realice el agente recibirá una recompensa. El valor de esta dependerá de como se haya definido la función recompensa previamente, una vez finalizada la acción y evaluada la recompensa de esta el agente avanzará a un nuevo estado en el que se repetirá este proceso. Esta dinámica se guía por el principio de maximizar la recompensa a largo plazo [16].

Los componentes esenciales en un sistema de RL son: el agente, que es la entidad que toma decisiones; el entorno, que representa el mundo externo con el que el agente interactúa; la política, que es la estrategia o conjunto de reglas que el agente sigue para decidir qué acción tomar en un estado dado; la recompensa, que es una señal escalar que indica qué tan bien lo está haciendo el agente después de haber tomado una acción en un estado particular; y el estado, que es una representación o descripción del escenario actual en el que se encuentra el agente

Capítulo 4

Diseño

El Trabajo de Fin de Grado (TFG) que a continuación se describe tiene como objetivo realizar una investigación sobre la conducción autónoma mediante inteligencia artificial, más concretamente mediante aprendizaje por refuerzo. Se tomará como entorno base el simulador fotorealista CARLA. Adicionalmente, se profundizará en la sinergia entre CARLA y ROS2, evaluando en particular la eficacia y viabilidad del carla to ros bridge, herramienta que se encarga de actuar como 'puente' entre estos dos programas, permitiendo que puedan comunicarse de manera efectiva entre sí. Además, también se evaluará la fusión de ROS y CARLA como herramienta esencial en el diseño de soluciones integradas y robustas para la conducción autónoma.

Dentro del contexto de la conducción autónoma primero se exploran distintas soluciones para un sistema de seguimiento de carril con un enfoque tradicional. La detección de carril en este sistema se basará en métodos fundamentados en la visión artificial y el procesamiento de imágenes, mientras que el control del vehículo estará gobernado por un controlador PID. A pesar de que estos algoritmos no son considerados de vanguardia en la actualidad, desempeñaron un papel fundamental en los primeros desarrollos de vehículos autónomos.

A continuación, se investigarán soluciones adicionales en las cuales se volverá a diseñar un sigue carril, pero en este caso, el algoritmo de detección de carril estará basado en IA. Se utilizará DL, específicamente una red neuronal convolucional, para la detección del carril que el agente estará encargado de seguir. Nuevamente, para implementar el control y la reacción del vehículo, se utilizará un controlador PID, similar al caso anterior.

Después de exponer estos dos enfoques para crear un sigue carril, uno tradicional y otro más moderno, en los cuales se utiliza el mismo tipo de controlador para gestionar los movimientos, se recopilarán datos con los cuales se generarán gráficos que nos permitirán hacer análisis que destacará las fortalezas y debilidades de cada técnica.

en diversas situaciones y contextos de conducción a la hora de filtrar y detectar de carriles. Este análisis arrojará luz sobre qué método es más efectivo y robusto para esta tarea, permitiendo así tomar una decisión fundamentada sobre cuál método elegir en la solución final de consucción autónoma.

De la misma manera repetiremos este mismo proceso con el sistema de control del vehículo. Se plantearán dos enfoques, se tomará como enfoque tradicional el método que se utilizó para la comparación de los métodos de detección de carril, es decir, un controlador PID y como método más actual basado en IA se diseñará un método de control basado en aprendizaje por refuerzo, más concretamente con una implementación de aprendizaje por refuerzo. Al igual que en el caso anterior se extraerán métricas con las que se compararán ambos métodos fundamentando así la decisión de la elección para el controlado de la solución final de conducción autónoma.

Finalmente, el TFG concluirá con la creación de dos sistemas avanzados de seguimiento de carril en los cuales la detección del carril y el control del vehículo estarán a cargo de los métodos seleccionados después de los dos análisis anteriores.

El primer sistema estará especialmente diseñado para seguir carriles de manera efectiva y evitar colisiones con otros vehículos u objetos en caso de encontrarlos en la vía frenando y deteniendo la marcha, garantizando así una conducción segura y eficiente en el entorno simulado de CARLA.

El segundo sistema será una versión mejorada del primero. Además de frenar para evitar colisiones con obstáculos, adaptará la velocidad en caso de que los obstáculos estén en movimiento, de modo que no se produzcan colisiones, pero sin llegar a detenerse por completo. Esto creará un sistema capaz de adaptarse de manera efectiva a situaciones de tráfico dinámico.

4.0.1. Arquitectura

La arquitectura de este TFG se compone de varios elementos clave que se integran para llevar a cabo la investigación y el desarrollo de las soluciones de seguimiento de carril. A continuación, se presenta una descripción general de esta arquitectura.

El entorno de simulación CARLA sirve como el denominador común y la plataforma central para la realización de pruebas y evaluación de algoritmos.

Una gran cantidad de las implementaciones de sigue carril que se presentan en este TFG se han desarrollado en el entorno de *Robot Operating System* (ROS) 2 el cuál se

comunican con CARLA a través del puente carla to ros bridge como ya se ha comentado en anteriores ocasiones. Estas implementaciones incluyen:

- **Teleoperador:** Permite el control manual del vehículo en el entorno CARLA a través de comandos de ROS.
- **Algoritmo de Seguimiento de Carriles basado en Canny + PID:** Utiliza el algoritmo de detección de bordes Canny y un controlador Proporcional Integral Derivativo (PID) para el seguimiento de carriles.
- **Algoritmo de Seguimiento de Carriles basado en Canny + Filtro de Color HSV + PID:** Combina la detección de bordes Canny con un filtro de color basado en espacio de color HSV y un controlador PID para el seguimiento de carriles.
- **Algoritmo de Seguimiento de Carriles basado en Sliding Window + PID:** Utiliza un algoritmo sliding window junto con un controlador PID para el seguimiento de carriles.
- **Algoritmo de Seguimiento de Carriles basado en Redes Neuronales + PID:** Implementa un sistema de seguimiento de carriles basado en redes neuronales convolucionales y utiliza un controlador PID para el control del vehículo.

Por otro lado Además de las implementaciones en ROS, se han desarrollado también diferentes programas en la API del lenguaje Python de CARLA que se ejecutan directamente en el entorno de simulación CARLA. Estas implementaciones incluyen:

- **Seguimiento de Carriles basado en Redes Neuronales y Q-Learning:** Utiliza una red neuronal junto con el algoritmo de aprendizaje por refuerzo Q-Learning para el seguimiento de carriles.
- **Seguimiento de Carriles basado en Redes Neuronales, Q-Learning y Detección de Obstáculos:** Amplía la implementación anterior al detener el vehículo si se detecta un obstáculo en la vía.
- **Seguimiento de Carriles Adaptativo al Tráfico:** Implementa un sistema de seguimiento de carriles que se adapta al tráfico.

En el diagrama de la figura 4.1 se puede ver una representación gráfica de la arquitectura que acaba de ser mencionada

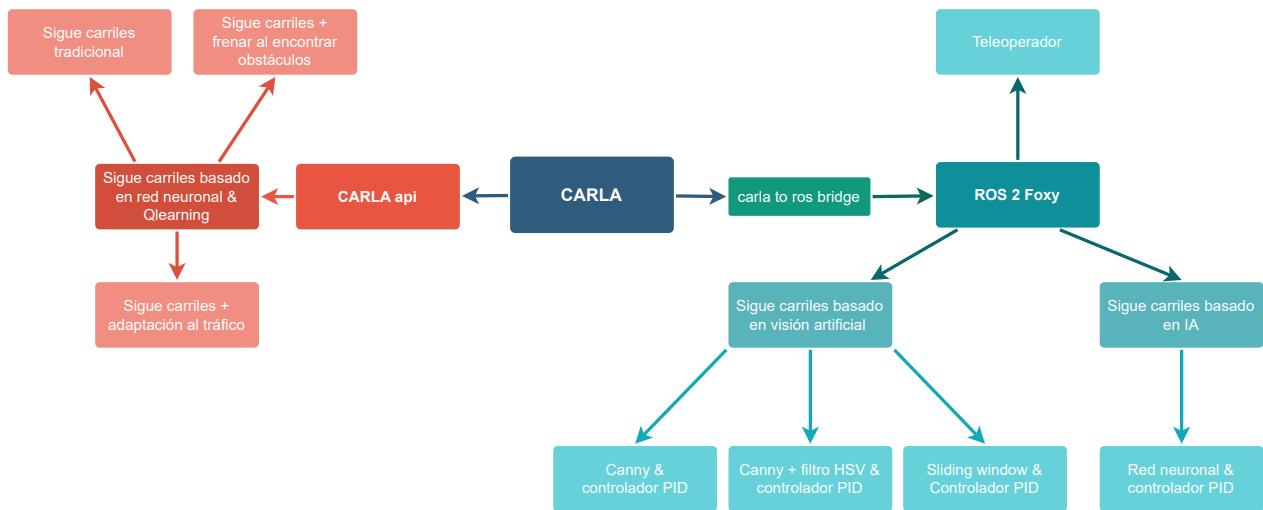


Figura 4.1: Arquitectura del TFG

4.1. Instalación del entorno de trabajo

Este TFG comenzó con la búsqueda y lectura de documentación e información relevante sobre todos los componentes del entorno de trabajo en el cual se iba a desarrollar el proyecto. Posteriormente, se procedió con la instalación de todos los programas necesarios para crear dicho entorno: el simulador fotorealista de conducción autónoma CARLA, el sistema operativo robótico ROS 2 y un ‘puente’ para comunicar ambos, el carla to ros bridge.

4.1.1. Instalación de CARLA

Para la instalación de CARLA, el procedimiento que se siguió consistió en seguir las directrices de instalación detalladas en la página oficial del simulador¹.

¹<https://carla.org/>

En primer lugar, se añadió el repositorio Debian de CARLA al sistema utilizado para trabajar.

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
1AF1527DE64CB8D9
sudo add-apt-repository "deb [arch=amd64] http://dist.carla.org/carla
$(lsb_release -sc) main"
```

Código 4.1: Añadir repositorio de CARLA al sistema

A continuación, se procedió a la instalación de la última versión de CARLA disponible utilizando el siguiente comando:

```
sudo apt-get update
sudo apt-get install carla-simulator
```

Código 4.2: Comandos para instalar CARLA

Después de completar la instalación del simulador, el siguiente paso, como era de esperar, fue ejecutarlo y verificar su correcto funcionamiento. Fue en este punto donde surgió el primer problema en este TFG. A pesar de estar ejecutándolo en una máquina con una potente tarjeta gráfica dedicada, específicamente una NVIDIA RTX 3060 para portátiles, el simulador utilizaba automáticamente la tarjeta gráfica integrada del ordenador. Esto resultó en una simulación extremadamente lenta, con poca fluidez y una tasa de fotogramas muy baja.

La solución para este problema se encontró en una respuesta a una entrada en un issue.^{en} en el GitHub oficial de CARLA² que hacía referencia a un problema similar. Se descubrió que al ejecutar CARLA con la bandera -prefernvidia, el simulador podía ser forzado a utilizar la tarjeta gráfica NVIDIA en caso de que estuviera disponible en el equipo. A partir de este punto, el comando utilizado para ejecutar CARLA durante todo este proyecto fue el siguiente:

```
/opt/carla-simulator/CarlaUE4.sh -prefernvidia
```

Código 4.3: Comando para lanzar el simulador CARLA

²<https://github.com/carla-simulator/carla/issues/4716>

4.1.2. Instalación de ROS2

Una vez instalado el simulador CARLA, el siguiente elemento necesario en el entorno de trabajo era ROS 2. Como se explica en la documentación oficial de ROS[17], ROS se divide en una serie de versiones específicas de paquetes ROS que se publican en momentos concretos llamadas distribuciones. Por lo tanto, una tarea importante antes de instalar ROS era elegir qué distribución se iba a utilizar. Debido a que ya se había trabajado con ella anteriormente en otros proyectos y que es una versión estable y probada, se eligió ROS 2 Foxy³ como la distribución para desarrollar el TFG.

Una vez que se tuvo claro qué versión de ROS se utilizaría, se navegó hasta la página oficial de ROS 2 Foxy para seguir los pasos de instalación⁴. De los dos métodos que se presentan en la página web, se eligió el método de instalación mediante paquetes DEB.



Figura 4.2: Logotipo de ros2 foxy

4.1.3. Carla to ROS Bridge

Como se ha mencionado previamente en este documento, CARLA y ROS 2 no están diseñados para funcionar conjuntamente de forma nativa. Para que estos dos programas puedan trabajar juntos, es necesario que puedan comunicarse de alguna manera. Para lograr esto, los desarrolladores de CARLA ofrecen una herramienta muy interesante: Carla to ROS Bridge. Este programa se encarga, tal como se menciona en su repositorio

³<https://docs.ros.org/en/foxy/index.html>

⁴<https://docs.ros.org/en/foxy/Installation.html>

oficial en GitHub[18], de crear un método de comunicación bidireccional entre ROS y CARLA, permitiendo una simbiosis entre ambos.

Instalación de Carla to ROS Bridge

Las instrucciones de instalación de Carla to ROS Bridge se encuentran en una sección de la documentación oficial del simulador CARLA⁵. No obstante, la instalación y el método de uso son bastante sencillos. Primero se debe clonar el repositorio de GitHub de Carla to ROS Bridge en el proyecto ROS en el que deseamos utilizarlo. Esto se puede realizar con el siguiente comando:

```
git clone https://github.com/carla-simulator/ros-bridge.git
```

Código 4.4: git clone de carla to ros bridge

Una vez clonado el repositorio para poder utilizarlo en un proyecto, es necesaria su inclusión en el archivo Cmake.list como se enseña a continuación

```
find_package(catkin REQUIRED ...
carla_ros_bridge
...
)
catkin_package(
...
CATKIN_DEPENDS
...
carla_ros_bridge
...
)
```

Código 4.5: Como incluir carla to ros bridge en un CMake.list

Después de configurar el Cmake.list, solo se tendrá que incluir en el nodo ROS que necesite comunicarse con CARLA las funciones, tipos de mensajes, etc., que vayamos a requerir del ROS Bridge, y luego compilar el paquete de manera habitual.

A continuación 4.6, se presenta un código de ejemplo escrito en Python para la plataforma ROS 2 Foxy que realiza la acción de comandar a un vehículo de CARLA mediante un publicador de ROS, utilizando el valor máximo posible de aceleración del vehículo.

⁵<https://carla.readthedocs.io/projects/ros-bridge/en/latest/>

```

import rclpy
from carla_msgs.msg import CarlaEgoVehicleControl

self.vehicle_control_publisher = self.create_publisher(
    CarlaEgoVehicleControl, "/carla/ego_vehicle/vehicle_control_cmd", 10)

self.control_msg = CarlaEgoVehicleControl()

self.control_msg.throttle = 1.0
self.control_msg.steer = 0.0
self.control_msg.brake = 0.0
self.control_msg.hand_brake = False
self.control_msg.reverse = False
self.control_msg.gear = 0

self.vehicle_control_publisher.publish(self.control_msg)

```

Código 4.6: Ejemplo de uso del carla to ros bridge

4.2. Creación de un teleoperador

Tras la puesta a punto del entorno de trabajo, ya era posible comenzar a desarrollar el cuerpo del TFG. No obstante, la conducción autónoma es una tarea muy compleja en la que intervienen numerosos factores y variables. Aventurarse en ella sin un conocimiento previo podría dejarte enfrentando un mundo inhóspito y desconocido en el cuál muy probablemente podrías acabar perdido. Además de esto, se añade la complicación de trabajar en un entorno que integra dos programas que no están explícitamente diseñados para trabajar juntos, como son ROS y CARLA. Este hecho agrega un nivel adicional de dificultad a una tarea muy compleja de por si.

Debido a esto, el desarrollo de este TFG comenzó con una pequeña tarea de iniciación que proporcionaría una primera toma de contacto con el entorno de trabajo y que nos dotaría de los conocimientos, las herramientas y la soltura necesarios para, posteriormente, afrontar las tareas principales de este TFG de manera fluida y precisa. Esta tarea inicial debía ser algo que permitiera practicar con el control de vehículos de CARLA y su entorno, con la comunicación de CARLA con ROS bridge y con los métodos para enviar comandos de control a CARLA desde ROS. Teniendo todo esto en cuenta, la tarea que resultó más adecuada para satisfacer todos estos requisitos fue la creación de un teleoperador para un vehículo de CARLA en ROS 2.

4.2.1. Teleoperador en CARLA basado en ROS2

El primer paso del desarrollo del teleoperador fue lanzar un mundo de CARLA generando en él, además, un vehículo que controlar. Para esto, nos ayudamos de un launcher ya presente en el repositorio de carla to ros bridge, más concretamente del launcher `carla_generate_vehicle.launch.py`. Este launcher se encarga de lanzar tanto el mundo de CARLA en una ciudad definida, como un vehículo de la librería de vehículos disponibles en CARLA, en este caso, un vehículo Tesla, como también el carla to ros bridge que a su vez se encarga de publicar todos los topics asociados al vehículo lanzado para poder interactuar con él desde ROS.

El control de un vehículo en CARLA se asemeja en gran medida a la operación de un vehículo real. En lugar de simplemente enviar comandos de velocidad lineal, se controla la cantidad de throttle que se comanda al coche o la cantidad de fuerza de freno que se aplica, esto equivale a la cantidad de presión que se le aplique al acelerador o al freno en un automóvil real. Además, la dirección del vehículo se controla indicando la cantidad de giro que se debe aplicar al volante del vehículo, estas características de control añaden un nivel adicional de complejidad en comparación a otro tipo de simuladores.

Es importante destacar que, en CARLA, también se tiene la capacidad de controlar las marchas del vehículo. Sin embargo, para simplificar nuestras pruebas y evitar complicaciones innecesarias, se ha optado por utilizar un vehículo automático en lugar de uno con transmisión manual.

El siguiente paso fue crear una interfaz gráfica para visualizar y controlar el vehículo de CARLA. En este caso, se utilizó la librería de Python pygame⁶. Según su página de Wikipedia oficial [19], Pygame es una biblioteca de Python que se utiliza para el desarrollo de videojuegos y aplicaciones multimedia interactivas, por lo que resultaba una opción perfecta para crear una interfaz gráfica en la que se visualizarían imágenes del vehículo y que además nos permitiera controlar sus movimientos al presionar el teclado, algo bastante parecido a lo que podría ser un videojuego.

Para crear la interfaz gráfica se creó una ventana en pygame, la cual se dividió en dos partes por la mitad. En la mitad izquierda se visualizaría el vehículo visto desde una perspectiva de tercera persona, mientras que en la mitad derecha se mostraron las imágenes de la cámara frontal con la que iba equipado el vehículo. Esta cámara

⁶<https://www.pygame.org/wiki/GettingStarted>

se encontraba ubicada en el capó del vehículo, justo en la parte media de este.. Adicionalmente, en la parte superior izquierda de la mitad derecha, también se añadió un pequeño indicador que mostraba la tasa de refresco de *Frames Per Second* (FPS) de la imagen. El resultado final de la interfaz se puede ver en la figura 4.3.

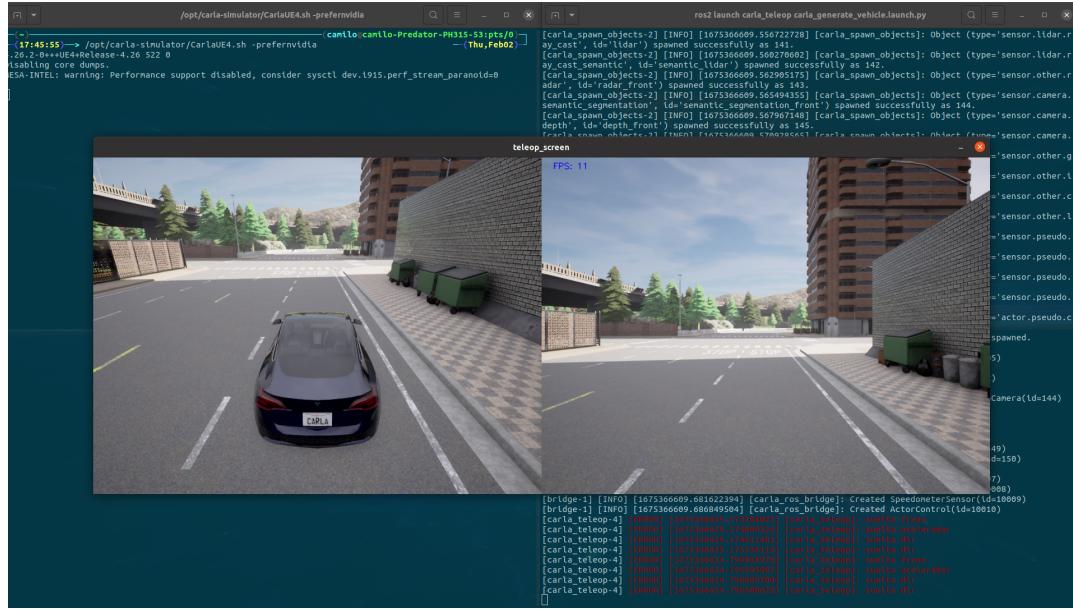


Figura 4.3: HMI de un teleoperado para un vehículo de CARLA

Una vez finalizado la Intefaz Gráfica de Usuario (GUI) de esta aplicación nos encontramos con la primera limitación del carla to ros brisge. Y es que a pesar de que el simulador CARLA por si solo funciona a unos 60 FPS en la maquina que se estaba desarrollando la aplicación, las imagenes recibidas por la GUI luego de que estas fueran traducidas de CARLA a ros por el ros bridge no superaban los 15 FPS. Tras investigar un poco se descubrió que esto se debe a un cuello de botella que genera el carla to ros bridge, en las imágenes que traduce de CARLA. Todas las imágenes que carla to ros bridge traduzca a ROS mediante topics tienen un límite máximo de tasa de refresco de alrededor de 15 FPS. Este problema significa que ninguna de las aplicaciones que se hagan utilizando CARLA y ROS con el ros bridge como método de comunicación podrán superar este frame rate máximo, algo sin duda muy grave cuando hablamos de conducción autónoma ya que a altas velocidades necesitamos buenas tasas de refresco de FPS para poder ser reactivos y adaptarnos a cambios repentinos. A pesar de esto a velocidades no muy altas 15 FPS resultan suficientes para tener un comportamiento fluido

Terminada la GUI del teleoperador, se prosiguió implementando el control del vehículo para poder manejarlo utilizando el teclado del computador. Para realizar esta

tarea, se utilizaron los eventos de pygame⁷. Mediante estos eventos es posible capturar qué teclas del teclado se están presionando en cada momento. Posteriormente, se asoció una serie determinada de teclas al control del vehículo. Esta implementación puede verse con más claridad en el código 4.7 De esta manera, cada vez que se apretaba una de estas teclas, se publicaba en los topics del vehículo la acción asociada a esa tecla. Luego, era trabajo del carla to ros bridge traducir los mensajes de estos topics a comandos de CARLA.

⁷<https://www.pygame.org/docs/ref/event.html>

```

def control_vehicle(self):

    self.set_autopilot()
    self.set_vehicle_control_manual_override()

    while True:

        for event in pygame.event.get():

            if event.type == KEYDOWN:
                keys = pygame.key.get_pressed()

                if (keys[K_DOWN]):
                    self.control_msg.brake = 1.0

                if (keys[K_UP]):
                    self.control_msg.throttle = 1.0

                if (keys[K_LEFT]):
                    self.control_msg.steer = -1.0

                if (keys[K_RIGHT]):
                    self.control_msg.steer = 1.0
            elif event.type == KEYUP :
                keys = pygame.key.get_pressed()
                if not(keys[K_DOWN]):
                    self.control_msg.brake = 0.0

                if not(keys[K_UP]):
                    self.control_msg.throttle = 0.0

                if not(keys[K_LEFT]):
                    self.control_msg.steer = 0.0

                if not(keys[K_RIGHT]):
                    self.get_logger().error("suelta dir")
                    self.control_msg.steer = 0.0

    self.vehicle_control_publisher.publish(self.control_msg)

```

Código 4.7: Eventos de pygame para controlar el teleoperador

Finalmente, para que el procesamiento de los eventos de pygame no interfiriera con el procesamiento de las imágenes del vehículo, cada una de estas funcionalidades fue implementada en un hilo distinto. De esta manera, un hilo del programa controla el HMI y la visualización de imágenes, mientras que el otro controla el manejo del vehículo

mediante el teclado.

En el siguiente **video** en la plataforma YouTube, se puede visualizar el funcionamiento del teleoperador que acaba de ser explicado.

4.3. Sigue carril basado en visión artificial tradicional y un controlador PID

Tras finalizar esta tarea inicial, como era lógico y esperado, observamos un aumento significativo en nuestra familiaridad con el entorno. La interacción entre ROS y CARLA se tornó más intuitiva, resaltando así que se habían adquirido las competencias fundamentales necesarias para trabajar de manera cómoda y fluida con estos dos programas. Con esta base establecida, nos encontrábamos en una posición óptima para empezar con el desarrollo de las tareas principales TFG.

Como se comentó en el capítulo 2 el objetivo de este TFG es crear una investigación completa y detallada sobre la conducción autónoma basada en inteligencia artificial, para ello es fundamental justificar porque utilizar inteligencia artificial es la opción más acertada para este tipo de tareas. Con el fin de demostrar esta premicia, los primeros apartados de este TFG van a consistir en diseñar tres algoritmos de seguimiento de carril cuya detección estará basada en efoques mas tradicionales donde se utilizará visión artificial y procesamiento de imágenes para detectar el carril de una carretera. Luego estos tres métodos se comparán con una solución en la que se detectará el carril utilizando IA concluyendo después de las pruebas que método resulta más efectivo.

Como nuestra misión es determinar que algoritmo nos proporciona una mejor detección de carril, en todos los métodos se utilizará el mismo controlador para manejar la dirección del vehículo y lograr el comportamiento de seguimiento de carril. Este controlador consistirá en una implementación de un controlador PID.

En todos los métodos se seguirá la siguiente dinámica: Primero se filtrarán las dos líneas que delimitan el carril utilizando un método de filtrado de cada implementación, a continuación se calculará el centro del carril filtrado en la carretera, posteriormente este centro se comparará con el centro de la imagen, calculando el error que hay entre ambos. Este error será el error del controlador PID, de esta manera se consigue que el controlador PID siempre procure mantener el centro del carril alineado con el centro de la imagen ya que en este caso el error sería igual a 0 y como en nuestro vehículo

la cámara se localiza en el centro del capó, mantener el centro del carril y el centro de la imagen alineados genera que el centro del vehículo se mantenga en el centro del carril evitando así que el vehículo se salga del carril. El controlador tendrá la siguiente implementación:

```

def control_vehicle(self):

    actual_error = self.error

    actual_error = (actual_error) / 100 #error
    d_error = actual_error - self.last_error #derivative error

    self.i_error = self.i_error + actual_error #integral error

    if actual_error >= 0:
        self.curling = 1

    if actual_error <= 0:
        self.curling = -1

    if ((actual_error < 10/100) and ( actual_error > -10/100)):
        stering = 0.0
        self.control_msg.steer = stering
        self.curling = 0.0

    elif ((actual_error < 50/100) and ( actual_error > -50/100)):
        stering = actual_error* self.kp_straight +
                  d_error*self.kd_straight + self.i_error*self.ki_straight

        if stering > 1:
            self.control_msg.steer = 1.0

        elif stering < -1.0:
            self.control_msg.steer = -1.0

        else:
            self.control_msg.steer = stering
    else :
        stering = actual_error*self.kp_turn + d_error*self.kd_turn +
                  self.i_error*self.ki_turn

        if stering > 1:
            self.control_msg.steer = 1.0

        elif stering < -1.0:
            self.control_msg.steer = -1.0

        else:
            self.control_msg.steer = stering

    self.last_error = actual_error

```

Código 4.8: Controlador PID de los sigue líneas basados en visión artifcial

Además del controlador que mantendrá al vehículo dentro del carril, para que el

vehículo avance por este se le otorgará una velocidad lineal fija, que se mantendrá durante todo el recorrido y que no variará manteniéndose lo más estable posible. Esta velocidad será de 20 kkm/h. Como ya se explicó antes en CARLA el movimiento del vehículo se controla mediante el thortel y no comandando una velocidad lineal, por lo que para mantener una velocidad fija se ha diseñado un pequeño controlador para aplicar thortel cuando la velocidad baje de la deseada y dejar de aplicarlo cuando supere la misma. Su implementación se puede ver a continuación

```
if self.speed >= 20:  
    self.control_msg.throttle = 0.0  
else:  
    self.control_msg.throttle = 1.0
```

Código 4.9: Controlador de velocidad fija del vehículo

4.3.1. Método 1: Filtro Canny

El primer método utilizado para detectar el carril se basa en un enfoque muy básico. En este método, se utilizará un filtro Canny. Canny es un algoritmo que, mediante la detección de cambios abruptos de intensidad utilizando cálculos de gradientes y umbralización, es capaz de detectar bordes en una imagen. Este filtro resulta especialmente útil en nuestro caso, ya que nos ofrece la posibilidad de detectar los bordes que delimitan las líneas del carril en nuestra carretera. Para realizar esta tarea de filtrado, se eligió utilizar la librería OpenCV⁸. OpenCV es una de las herramientas más utilizadas en el campo de la visión artificial debido a su accesibilidad, eficiencia y comunidad activa. Además, esta librería nos ofrece una gran cantidad de funciones para el filtrado y procesamiento de imágenes, entre ellas, ofrece el filtro Canny.

Luego de filtrar las líneas del carril con Canny, se delimita un área de interés en la que el carril debería localizarse para, de esta manera, eliminar el ruido de posibles interferencias generadas por el propio entorno fuera de nuestra zona de interés, edificios, árboles, carteles etc. y finalmente se utilizará la función de OpenCV HoughLinesP⁹ para detectar estas líneas ya filtradas y guardar los píxeles que las componen. Esto nos permitirá posteriormente analizarlos con la finalidad de encontrar el centro del carril y controlar nuestro vehículo. A continuación se deja la implementación de este método de filtrado de carril

⁸<https://opencv.org/>

⁹https://docs.opencv.org/3.4/d3/de6/tutorial_js_houghlines.html

```

def line_filter(self, img):

    # Convert to grayscale here.
    gray_image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)# Call Canny Edge
        Detection here.
    cannyed_image = cv2.Canny(gray_image, 100, 150)

    h, w = cannyed_image.shape[:2]
    region_of_interest_vertices = [ (0, h*0.7),(w/2 , h/2) ,(w, h*0.7), ]
    cropped_image = self.region_of_interest(cannyed_image,
        numpy.array([region_of_interest_vertices], numpy.int32))

    lines = cv2.HoughLinesP(cropped_image,rho=9,theta=numpy.pi / 60,
        threshold=50,lines=numpy.array([]),minLineLength=20,maxLineGap=25)

```

Código 4.10: Filtro canny

Como se puede observar en el código 4.10 primero aplicamos el filtro canny a las imágenes obtenidas de la cámara, luego recortamos la zona de interés anteriormente mencionada y finalmente aplicamos houghlines para detectar y guardar las coordenadas de las líneas

Este método tiene la ventaja de que al ser muy sencillo resulta computacionalmente muy ligero, ya que el único procesamiento de imagen que se realiza es un simple canny y posteriormente la detección de líneas mediante houghlines. Esto lo podemos ver facilmente reflejado en las dos gráficas de a continuación en las que se muestra la tasa de refresco de FPS, figura 4.4 y el consumo de cpu del algoritmo la figura 4.5

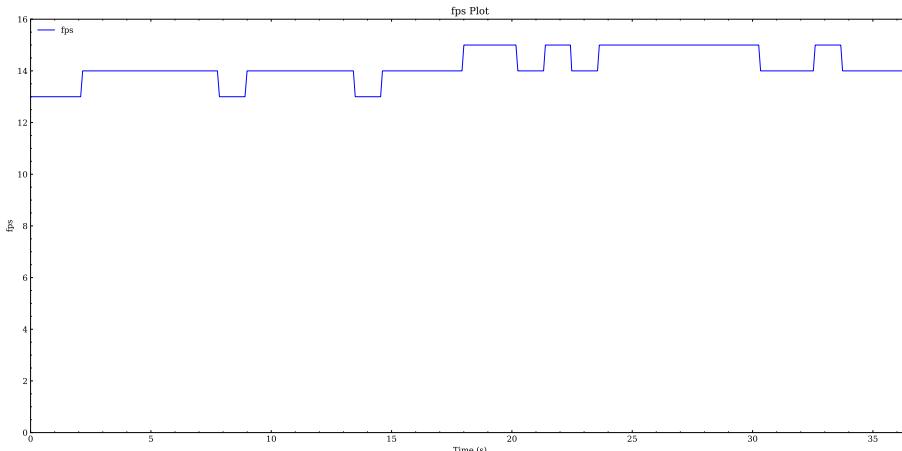


Figura 4.4: Gráfica de tasa de fps del sigue carril basado en canny

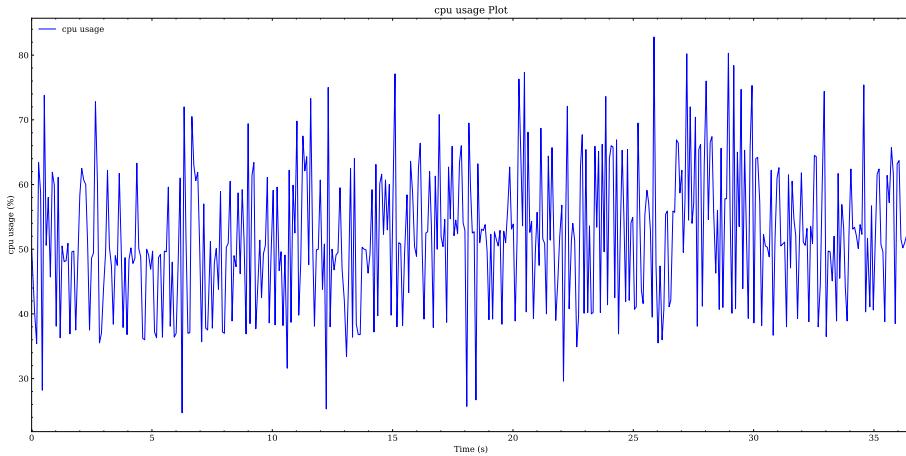


Figura 4.5: Gráfica de uso de cpu del sigue carril basado en canny

Como vemos los FPS de las imágenes de este método no se aleja mucho de los 15 FPS que obtenemos al mostrar una imagen sin procesamiento obteniendo FPS.

En cuanto al consumo de cpu, en sistemas Linux, el uso de la CPU se muestra en porcentajes y se basa en la cantidad de núcleos disponibles en la computadora. En este caso la maquina que ha ejecutado este y los demás métodos cuenta con 6 núcleos, cada núcleo se considera al 100 % de su capacidad cuando está trabajando al máximo. de manera que 600 % de uso de CPU significaría que el algoritmo esta consumiendo todos los recursos de procesamiento de la maquina. En este caso como se ven en la gráfica de la figura 4.5, el algoritmo probado consume alrededor del 50 % de uso de CPU esto quiere decir que el algoritmo esta necesitando tan solo la mitad de uno de los 6 núcleos del sistema para ser procesado lo cuál constituye un consumo pequeño de los recursos del sistema.

Pero por otro lado nuevamente al ser un método tan sencillo el filtro de carril no es muy robusto y presenta numerosas limitaciones. El carril de la vía no se llega a filtrar completamente, además de que aparecen fallos y errores de detección en muchos momentos sobretodo al intentar filtrar líneas discontinua. A pesar de esto ajustando el controlador PID de los movimientos del vehículo se puede conseguir un comportamiento mas o menos fluido en un entorno controlado. A continuación, se presenta un enlace a un **video** en el que se puede observar el funcionamiento de este método.

4.3.2. Método 2: Filtro de color HSV + filtro Canny

El segundo método utilizado para detectar el carril de la vía es muy parecido al primero. En este método, nuevamente se utilizará la función de detección de líneas HoughLines y un filtro Canny, pero con el añadido de que ahora, además del filtro

Canny, se añadirá una segunda función de filtrado de carril: un filtro de color. Las líneas que delimitan los carriles generalmente son de color blanco, mientras que la carretera, por el contrario, es de color negro o gris. Esta clara diferencia de color entre ambos elementos permite detectar las líneas de la carretera filtrando los colores blanco, negro y gris presentes en la imagen de la cámara.

Primero, antes de pasar a implementar este método, debemos determinar el rango de color que vamos a filtrar y en qué formato. En visión artificial, es común utilizar el formato HSV para filtros de color debido al control que ofrece sobre el contraste y la iluminación. Estas características se adaptan bien a nuestro caso, en el que no buscábamos separar diferentes colores, sino distinguir el negro del blanco. Para definir el espectro de color que deseábamos filtrar, posicionamos el vehículo de CARLA en diferentes ubicaciones del mapa y capturamos imágenes que la cámara recogía del carril. Una vez que teníamos estas imágenes, diseñamos una sencilla herramienta que mostrara la imagen en pantalla y nos permitiera ajustar los valores de HSV que se aplicaban a la imagen mediante controles deslizantes. Con esta herramienta, buscábamos encontrar un rango de valores HSV común a todas las imágenes que filtrara las líneas blancas del carril de la mejor manera posible.

Sin embargo, al realizar el filtrado de las líneas mediante HSV, nos encontramos con la primera limitación de este método. Debido al realismo de CARLA, los cambios de luz, sombras y reflejos eran comunes, y dependiendo de estos factores, el rango de color blanco de las líneas del carril variaba significativamente. Esto era especialmente evidente cuando el carril estaba expuesto a luz directa o cuando estaba bajo una sombra. En estos casos, era imposible filtrar las líneas del carril utilizando el mismo rango HSV.

Finalmente, concluimos que era imposible encontrar un rango de color blanco que fuera válido para todas las situaciones. Por lo tanto, intentamos encontrar un rango que abarcara el mayor número posible de situaciones. Luego de explorar muchos valores diferentes, elegimos el siguiente rango.

```
#HSV color filter range
lower_white = numpy.array([0, 0, 200])
upper_white = numpy.array([255, 255, 255])
```

Código 4.11: Rangos de detección del filtro de color HSV

La implementación final del filtro de color HSV fue la siguiente:

```

def line_filter(self, img):

    hsv_image = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    lower_white = numpy.array([0, 0, 200])
    upper_white = numpy.array([255, 255, 255])

    color_mask = cv2.inRange(hsv_image, lower_white, upper_white)
    filtered_image = cv2.bitwise_and(img, img, mask=color_mask)

    h, w = filtered_image.shape[:2]
    region_of_interest_vertices = [ (0, h*0.75),(w/2 , h/2) ,(w,
        h*0.75) , ]
    cropped_image = self.region_of_interest(filtered_image,
        numpy.array([region_of_interest_vertices], numpy.int32))

    lines = cv2.HoughLinesP(cropped_image,rho=9,theta=numpy.pi / 60,
        threshold=50,lines=numpy.array([]),minLineLength=20,maxLineGap=25)

```

Código 4.12: Filtro de color HSV

Con este método, nuevamente nos encontramos con la ventaja de que resulta ser computacionalmente muy ligero, ya que el único procesamiento añadido a la imagen es un simple filtro de color HSV. Esto se refleja en la tasa de FPS de la imagen, ya que sigue sin variar mucho de los FPS que obtenemos al mostrar una imagen sin procesamiento ni los que obtenemos en el método anterior. No obstante, debido al cuello de botella ya mencionado anteriormente, seguimos limitados a una tasa de refresco de 15 FPS.

Con este método, nuevamente nos encontramos con la ventaja de que resulta ser computacionalmente muy ligero, ya que el único procesamiento añadido a la imagen es un filtro de color HSV. Esto de nuevo se ve reflejado en las dos gráficas de a continuación en las que nuevamente se muestra la tasa de refresco de FPS, figura 4.6 y el consumo de cpu este algortimo la figura ??

Como vemos los FPS de las imágenes tras el procesamiento necesario para este método siguen sin mucho de los 15 FPS máximos, sin embargo si que podemos observar que la media de FPS es un poco más baja que en el caso anterior debido al procesamiento extra.

El consumo de cpu, en este caso como se ve en la gráfica de la figura 4.5, es un poco mayor que en caso anterior, rondando ahora el 60 % de uso de CPU. Esto se debe

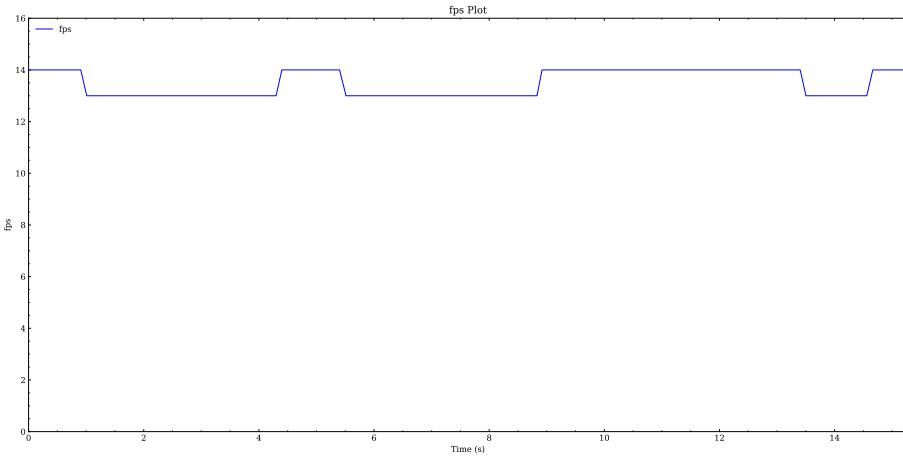


Figura 4.6: Gráfica de tasa de fps del sigue carril basado en canny + hsv

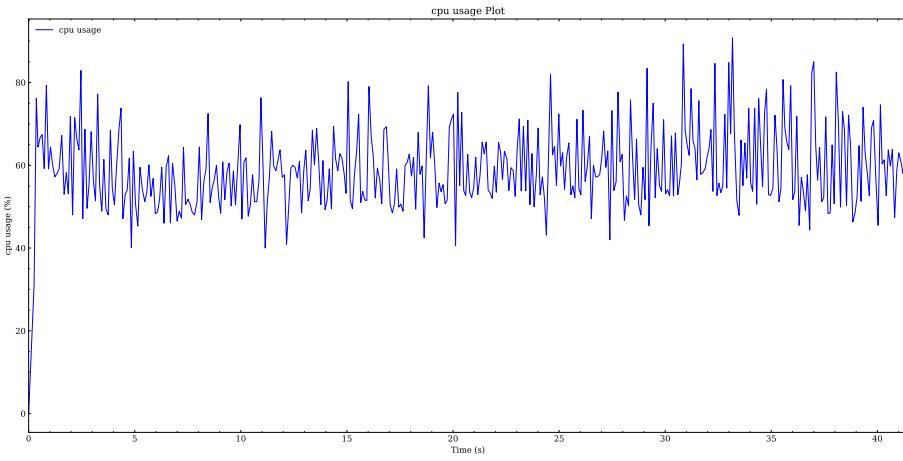


Figura 4.7: Gráfica de uso de cpu del sigue carril basado en canny + hsv

nuevamente al procesamiento de imagen extra añadido a este método, sin embargo, el incremento es bastante pequeño debido a la naturaleza sencilla del mismo.

A continuación, se presenta un enlace a un **video** en el que se puede observar el funcionamiento de este método. Como vemos, el filtrado de las líneas del carril mejora ligeramente con respecto al método anterior; sin embargo, sigue siendo bastante defectuoso.

[sectionMétodo 3: Pipeline de procesamiento de imágenes con algoritmo Sliding window](#)

Los dos primeros métodos que se han expuesto son métodos muy sencillos que no se suelen utilizar por si solos en soluciones del mundo real, ya que no aportan la robustez suficiente para funcionar en varios ambientes. El tercer método que vamos a utilizar para filtrar el carril es un método bastante más sofisticado y complejo con que los dos anteriores, a cambio este método promete entregarnos resultados más robustos y eficaces.

Este método consiste en un pipeline de varias transformaciones, filtros y procesamientos de imágenes que se le aplican a la imagen obtenida de la cámara, para filtrar el carril y guardar los puntos que componen este. El pipeline utilizado es el siguiente:

1. **Umbralización de color y gradiente:** Esta etapa consiste en una combinación de umbrales de color y gradiente para generar una imagen binaria de los carriles de la carretera. En el código de continuación se puede ver con más detalle.
-

```
def pipeline(self,img, s_thresh=(200, 255), sx_thresh=(50, 255)):

    img = numpy.copy(img)

    # Convert to HLS color space and separate the V channel
    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS).astype(numpy.float)
    l_channel = hls[:, :, 1]
    s_channel = hls[:, :, 2]
    h_channel = hls[:, :, 0]

    # Sobel x detecta los bordes en x
    sobelx = cv2.Sobel(l_channel, cv2.CV_64F, 1, 1) # Take the
        derivative in x
    abs_sobelx = numpy.absolute(sobelx) # Absolute x derivative to
        accentuate lines away from horizontal
    scaled_sobel = numpy.uint8(255*abs_sobelx/numpy.max(abs_sobelx))

    # Threshold x gradient
    sxbinary = numpy.zeros_like(scaled_sobel)
    sxbinary[(scaled_sobel >= sx_thresh[0]) & (scaled_sobel <=
        sx_thresh[1])] = 1

    # Threshold color channel
    s_binary = numpy.zeros_like(s_channel)
    s_binary[(s_channel >= s_thresh[0]) & (s_channel <= s_thresh[1])] = 1

    #color_binary = numpy.dstack((numpy.zeros_like(sxbinary), sxbinary,
        s_binary)) * 255

    combined_binary = numpy.zeros_like(sxbinary)
    combined_binary[(s_binary == 1) | (sxbinary == 1)] = 1

    return combined_binary
```

Código 4.13: Umbralización de color y gradiente

2. **Transformación de perspectiva:** La imagen es modificada para obtener una

vista desde arriba o vista de pájaro de la imagen binaria de los carriles. Esto facilita la identificación y seguimiento de las líneas del carril al eliminar la perspectiva que podría distorsionar la percepción. En el código de a continuación se muestra la función que implementa esta transformación

```
def perspective_warp(self, img,
                      dst_size=(800, 600),
                      src=numpy.float32([(0.39,0.57),(0.62,0.57),(0.1,1),(1,1)]),
                      dst=numpy.float32([(0,0), (1, 0), (0,1), (1,1)])):
    img_size = numpy.float32([(img.shape[1],img.shape[0])])
    src = src* img_size
    dst = dst * numpy.float32(dst_size)

    # Given src and dst points, calculate the perspective transform
    # matrix
    M = cv2.getPerspectiveTransform(src, dst)
    # Warp the image using OpenCV warpPerspective()
    warped = cv2.warpPerspective(img, M, dst_size)

    return warped
```

Código 4.14: Transformación de perspectiva

Pero de la misma manera que invertimos la perspectiva luego hay que desinvertirla para poder utilizar el resultado generado sobre la imagen original, para esto se creo la siguiente función que reinvierte la perspectiva.

```
def inv_perspective_warp(self, img,
                          dst_size=(800, 600),
                          src=numpy.float32([(0,0), (1, 0), (0,1), (1,1)]),
                          dst=numpy.float32([(0.39,0.57),(0.62,0.57),(0.1,1),(1,1)])):
    img_size = numpy.float32([(img.shape[1],img.shape[0])])
    src = src* img_size
    dst = dst * numpy.float32(dst_size)
    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, dst_size)
    return warped
```

Código 4.15: Inversión de la transformación de perspectiva

3. **Algoritmo sliding window:** Al buscar las líneas del carril por primera vez o cuando el algoritmo tiene incertidumbre sobre las ubicaciones de las líneas, se

realiza una búsqueda de ventanas mediante un algoritmo sliding window¹⁰. Esta búsqueda toma un histograma de la mitad inferior de la imagen binaria, revelando el vecindario donde comienzan las líneas del carril. Luego, se divide la imagen en segmentos horizontales, deslizando una ventana de búsqueda en cada segmento para encontrar las áreas de mayor frecuencia. Una vez identificadas estas áreas para los carriles izquierdo y derecho, se realiza un ajuste polinómico de segundo orden utilizando la función np.polyfit() para obtener una curva que se adapte a la línea.

¹⁰<https://www.geeksforgeeks.org/window-sliding-technique/>

```

def sliding_window(self,img, nwindows=10, margin=150, minpix = 1,
draw_windows=True):
    left_fit_ = numpy.empty(3)
    right_fit_ = numpy.empty(3)
    out_img = numpy.dstack((img, img, img))*255

    histogram = self.get_hist(img)
    # find peaks of left and right halves
    midpoint = int(histogram.shape[0]/2)
    leftx_base = numpy.argmax(histogram[:midpoint])
    rightx_base = numpy.argmax(histogram[midpoint:]) + midpoint

    # Set height of windows
    window_height = numpy.int(img.shape[0]/nwindows)
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = img.nonzero()
    nonzeroy = numpy.array(nonzero[0])
    nonzerox = numpy.array(nonzero[1])
    # Current positions to be updated for each window
    leftx_current = leftx_base
    rightx_current = rightx_base

    # Create empty lists to receive left and right lane pixel indices
    left_lane_inds = []
    right_lane_inds = []

    # Step through the windows one by one
    for window in range(nwindows):
        # Identify window boundaries in x and y (and right and left)
        win_y_low = img.shape[0] - (window+1)*window_height
        win_y_high = img.shape[0] - window*window_height
        win_xleft_low = leftx_current - margin
        win_xleft_high = leftx_current + margin
        win_xright_low = rightx_current - margin
        win_xright_high = rightx_current + margin
        # Draw the windows on the visualization image
        if draw_windows == True:
            cv2.rectangle(out_img,(win_xleft_low,win_y_low),(win_xleft_high,win_y_high),(100,255,255), 3)
            cv2.rectangle(out_img,(win_xright_low,win_y_low),(win_xright_high,win_y_high),(100,255,255), 3)

        # Identify the nonzero pixels in x and y within the window
        good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy <
win_y_high) &
(nonzerox >= win_xleft_low) & (nonzerox <
win_xleft_high)).nonzero()[0]
        good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy <
win_y_high) &
(nonzerox >= win_xright_low) & (nonzerox <
win_xright_high)).nonzero()[0]

```

Código 4.16: Filtrado de carril mediante algoritmo sliding window

```

# Append these indices to the lists
left_lane_inds.append(good_left_inds)
right_lane_inds.append(good_right_inds)

# If you found > minpix pixels, recenter next window on their
# mean position
if len(good_left_inds) > minpix:
    leftx_current =
        numpy.int(numpy.mean(nonzerox[good_left_inds]))
if len(good_right_inds) > minpix:
    rightx_current =
        numpy.int(numpy.mean(nonzerox[good_right_inds]))

# Concatenate the arrays of indices
left_lane_inds = numpy.concatenate(left_lane_inds)
right_lane_inds = numpy.concatenate(right_lane_inds)

# Extract left and right line pixel positions
leftx = nonzerox[left_lane_inds]
lefty = nonzeroy[left_lane_inds]
rightx = nonzerox[right_lane_inds]
righty = nonzeroy[right_lane_inds]

ploty = numpy.linspace(0, img.shape[0]-1, img.shape[0] )

if lefty.any() and leftx.any() :

    left_fit = numpy.polyfit(lefty, leftx, 2)
    self.line_detected_num = self.line_detected_num +1

    self.left_a.append(left_fit[0])
    self.left_b.append(left_fit[1])
    self.left_c.append(left_fit[2])

    left_fit_[0] = numpy.mean(self.left_a[-10:])
    left_fit_[1] = numpy.mean(self.left_b[-10:])
    left_fit_[2] = numpy.mean(self.left_c[-10:])

    left_fitx = left_fit_[0]*ploty**2 + left_fit_[1]*ploty +
    left_fit_[2]

else:
    left_fitx = numpy.empty(0)

if righty.any() and rightx.any() :
    right_fit = numpy.polyfit(righty, rightx, 2)
    self.line_detected_num = self.line_detected_num +1

    self.right_a.append(right_fit[0])
    self.right_b.append(right_fit[1])
    self.right_c.append(right_fit[2])

```

```

        right_fit_[0] = numpy.mean(self.right_a[-10:])
        right_fit_[1] = numpy.mean(self.right_b[-10:])
        right_fit_[2] = numpy.mean(self.right_c[-10:])
        # Generate x and y values for plotting
        right_fitx = right_fit_[0]*ploty**2 + right_fit_[1]*ploty +
                     right_fit_[2]
    else:
        right_fitx = numpy.empty(0)

    return (left_fitx, right_fitx)

```

Código 4.18: Filtrado de carril mediante algoritmo sliding window

4. **Dibujo del carril:** Finalmente, se procede a dibujar y resaltar el carril detectado sobre la imagen original, proporcionando una representación visual clara del camino a seguir.

Una vez analizadas todas las funciones individualmente, en el código 4.19 se puede ver la implementación del pipeline completo.

```

def line_filter(self, img):

    img_ = self.pipeline(img)
    img_ = self.perspective_warp(img_)
    curves = self.sliding_window(img_, draw_windows=True)

    if curves[0].any() and curves[1].any():
        img = self.draw_lanes(img, curves[0], curves[1])
        self.draw_centers(img)
    else:
        center_x = int(img.shape[1]/2)
        cv2.line(img, (center_x, 450), (center_x, 600), [0, 255, 0], 2)

    return img

```

Código 4.19: Pipeline completo de filtrado de carril

Este método es mucho más pesado computacionalmente que los dos anteriores, debido a la gran cantidad de procesamiento de imagén y a los complejos métodos que se utilizan en el, por consiguiente y como era de esperar en este caso la tasa de FPS de la imagen se ve reducida, esto se puede observar en la gráfica de a continuación 4.9

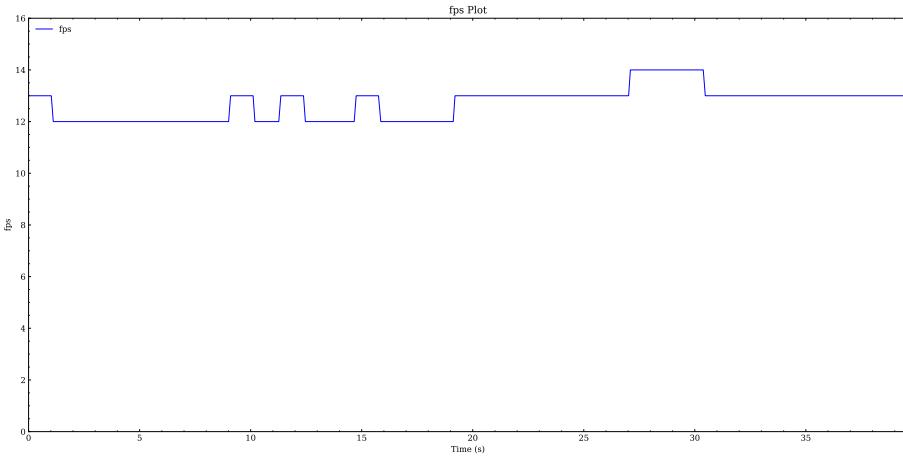


Figura 4.8: Gráfica de tasa de fps del sigue carril basado en algoritmo sliding window

En este algoritmo vemos como la media de FPS ronda los 12 FPS alejandonos ahora si un poco más de los 15 FPS máximos. Por otro lado y como vemos en la gráfica de la figura 4.9 el consumo de cpu también se ve como es lógico incrementando superando ahora el 100 % de uso de cpu y alcanzando incluso picos del 120 %, lo que significa que nuestra maquina esta utilizando completamente uno de sus núcleos de procesamiento de sus seis núcleos de procesamiento en este algoritmo, e incluso está necesitando del uso de otro núcleo extra.

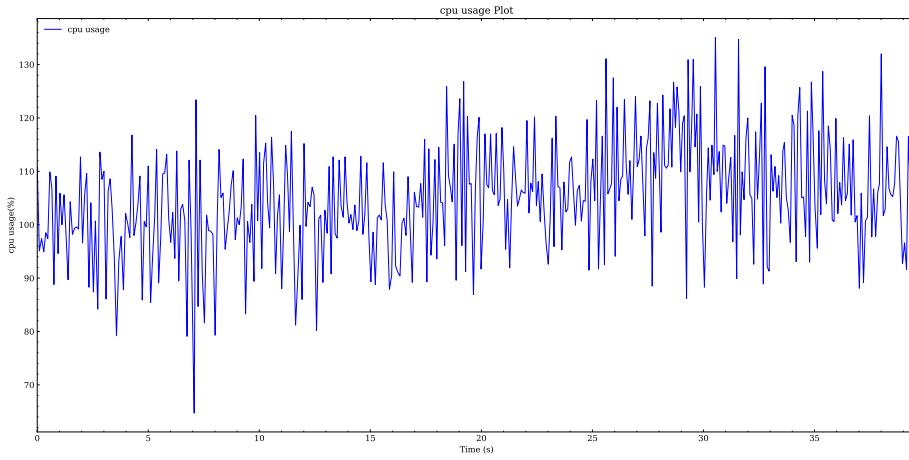


Figura 4.9: Gráfica de uso de cpu del sigue carril basado en algoritmo sliding window

Sin embargo el desempeño de este método de filtrado de carril a la hora de detectar un carril es muy notable, superando con creces el de los dos métodos anteriores. Pudiendo filtrar carriles de manera efectiva en distintas situaciones con distintas iluminaciones e incluso ofreciendo buenos resultados con líneas discontinuas.

Este método tiene la ventaja de que al ser muy sencillo resulta computacionalmente muy ligero, ya que el único procesamiento de imagen que se realiza es un simple canny

y posteriormente la detección de líneas mediante houghlines. Esto lo podemos ver facilmente reflejado en las dos gráficas de a continuación en las que se muestra la tasa de refresco de FPS, figura 4.4 y el consumo de cpu del algortimo la figura 4.5

En cuanto al consumo de cpu, en sistemas Linux, el uso de la CPU se muestra en porcentajes y se basa en la cantidad de núcleos disponibles en la computadora. En este caso la maquina que ha ejecutado este y los demás métodos cuenta con 6 núcleos, cada núcleo se considera al 100 % de su capacidad cuando está trabajando al máximo. de manera que 600 % de uso de CPU significaría que el algoritmo esta consumiendo todos los recursos de procesamiento de la maquina. En este caso como se ven en la gráfica de la figura 4.5, el algoritmo probado consume alrededor del 50 % de uso de CPU esto quiere decir que el algoritmo esta necesitando tan solo la mitad de uno de los 6 núcleos del sistema para ser procesado lo cuál constituye un consumo pequeño de los recursos del sistema.

A continuación, se presenta un enlace a un **video** en el que se puede observar el funcionamiento de este tercer método de filtrado de carril. En carriles rectos y curvas abiertas el desempeño que ofrece es muy preciso y robusto incluso en situaciones con cambios de luz y reflejos, superando concretos el desempeño de los dos métodods anteriores. No obstante el algoritmo no es perfecto y cuando se encuentra en situaciones con curvas cerradas la detección sufre bastantes fallos causando en muchos casos que el vehículo se llegue hasta salir del carril

Este tercer método de filtrado de carril esta inspirado en este proyecto [20] realizado por Charles Wong.

4.4. Sigue carril basado en inteligencia artificial y un controlador PID

4.4.1. Método 1: Redes neuronales

Tras haber explorado distintos enfoques básados en visión artificial y analizar sus ventajas y desventajas. Podemos dar paso a una nueva línea de investigación en la que se creará y analizará un sigue carril cuya detección de carril estará basada en IA. Como en el caso anterior el control del vehículo será dominado por un controlador PID con las mismas características que el visto en los métodos anteriores 4.8 , aunque nuevamente con la variación de un ajuste de las constantes Kd, Kp y Kd personalizado para este caso.

En este método la detección del carril a seguir se realizará mediante la inferencia de red neuronal. Más concretamente se utilizará una red neuronal convolucional ¹¹ específica para tareas de segmentación. Esta red ha sido previamente entrenada con un dataset de imágenes de distintos carriles pertenecientes a una gran variedad de mundos presentes en el simulador CARLA en todo tipo de situaciones climatológicas y escenarios con diferentes iluminaciones, sombras y reflejos. Esta red ha sido creada con la librería de IA de python pytorch ¹². Pytorch tal y como se explica en su documentación oficial [] es una biblioteca optimizada de tensores para aprendizaje profundo capaz de utilizar tanto GPUs como CPUs. Pytorch Se ha convertido en una de las herramientas más populares y ampliamente utilizadas en el campo de la inteligencia artificial y el aprendizaje automático por su increíble potencia a la par que fácil uso debido a su cercanía con el lenguaje python.

Para hacer uso de la red neuronal y lograr la detección de carriles mediante su inferencia han de seguirse una serie de pasos.

1. Primero tendremos que cargar la red neuronal en nuestro programa mediante las herramientas proporcionadas por pytorch.
2. Posteriormente proporcionaremos las imágenes de la cámara de nuestro vehículo a la red neuronal como entrada
3. El tercer y último paso guardar los datos de salida de la red neuronal en los que se encuentra filtrado el carril de la imagen

No obstante, al intentar probar este método nos encontramos con un problema. La limitación de recursos en la tarjeta gráfica de nuestra máquina, que estaba equipada con 6 GB de memoria, se volvió evidente al lanzar conjuntamente el simulador de conducción Carla y el algoritmo de detección basado en redes neuronales, ya que en ese momento, se generó un error que indicaba la insuficiencia de memoria gráfica disponible para llevar a cabo estas tareas simultáneamente.

Para superar este obstáculo, se adoptó una estrategia que implicó realizar una pequeña modificación en la configuración. Optamos por iniciar Carla con el flag -low-quality. Esta configuración específica tenía como resultado la disminución de la calidad visual de las imágenes generadas en el entorno virtual de Carla. Aunque esta decisión

¹¹https://es.wikipedia.org/wiki/Red_neuronal_convolucional

¹²<https://pytorch.org/>

reducía la calidad de las representaciones visuales, también se traducía en un consumo de recursos significativamente menor por parte de la tarjeta gráfica.

La implementación de esta modificación demostró ser una solución efectiva para nuestro proyecto. Permitió que el algoritmo de detección basado en redes neuronales funcionara de manera adecuada y fluida, produciendo como se ve en la grafica de la figura 4.10 una tasa de refresco de alrededor de 13 FPS lo cual era suficiente para ver el comportamiento del algoritmo. Además como se puede ver en la figura de la gráfica 4.11 El consumo de CPU de este algoritmo de detección rondaba el 100 %, sin embargo, también hay que tener en cuenta que debido a la configuración establecida, en este caso particular, el procesamiento de imágenes por parte de la red neuronal se realizaba utilizando la gráfica del equipo y no la cpu.

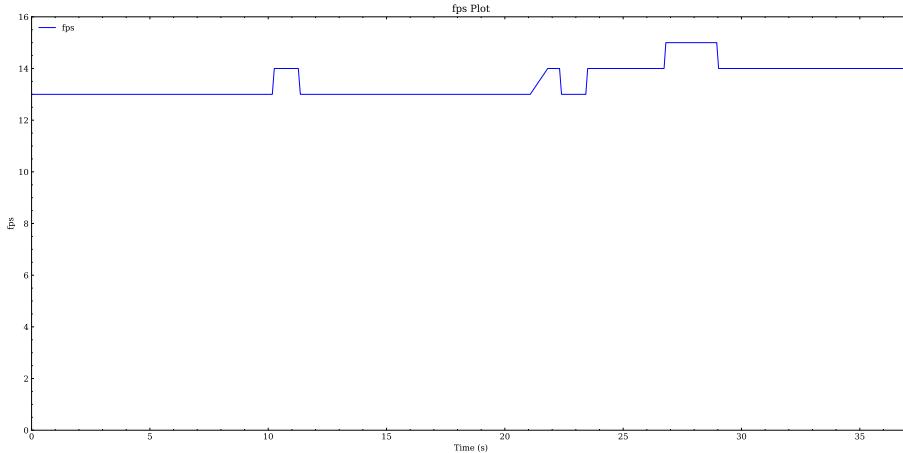


Figura 4.10: Gráfica de tasa de fps del sigue carril basado en deeplearning

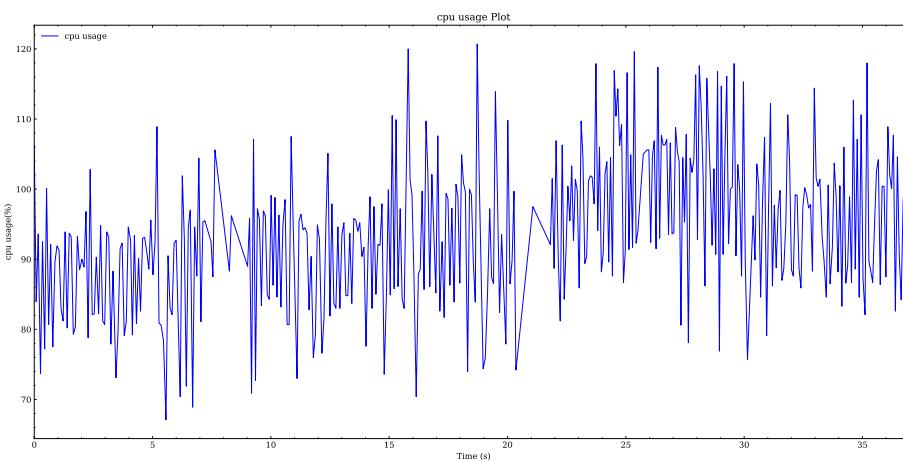


Figura 4.11: Gráfica de uso de cpu del sigue carril basado en deeplearnig

A continuación, se presenta un enlace a un [vídeo](#) en el que se puede observar el funcionamiento de este método. Como vemos el resultado obtenido en la detección del

carril es mucho mas robusto que todos los anteriores vistos. La red neuronal es capaz de filtrar el carril en todo tipo de situaciones, incluso con reflejos cambios de luz e incluso líneas cerrada. Solamente presentó fallos en la detección de líneas discontinuas cuando la distancia entre líneas es muy grande, sin embargo este problema podría ser facilmente solucionado reentrenando la red neuroanal, añadiendo imágenes de de carriles con este tipo de líneas discontinuas al dataset de entrenamiento.

4.5. Comparación de los métodos de detección

Después de exponer y analizar los resultados de los tres algoritmos basados en visión artificial y un algoritmo basado en redes neuronales, podemos llevar a cabo una comparación entre ellos para llegar a una conclusión acerca del método más eficaz en la detección de carriles.

En cuanto a los FPS, observamos en la figura 4.12 que los dos algoritmos más simples, que requieren menos procesamiento, logran una mayor cantidad de FPS, acercándose significativamente al límite impuesto por el cuello de botella del rosbridge. A continuación, se encuentra el algoritmo sliding window, que como sabemos demanda un mayor procesamiento debido a esto, este algoritmo tiene la menor tasa de FPS de todos, aún así logra mantener una cantidad aceptable de FPS por lo que el comportamiento del algoritmo sigue logrando ser fluido y reactivo. Finalmente estaría el algoritmo algoritmo basado en redes neuronales. Para este caso, es importante tener en cuenta que este algoritmo a diferencia de los demás utiliza la tarjeta gráfica del PC para su procesamiento y en el modo de baja calidad de imágenes de CARLA, como ya se explicó anteriormente. Dentro de este contexto los FPS que ofrece este algoritmo son suficiente altos, situandose cerca de los algoritmos más sencillos.

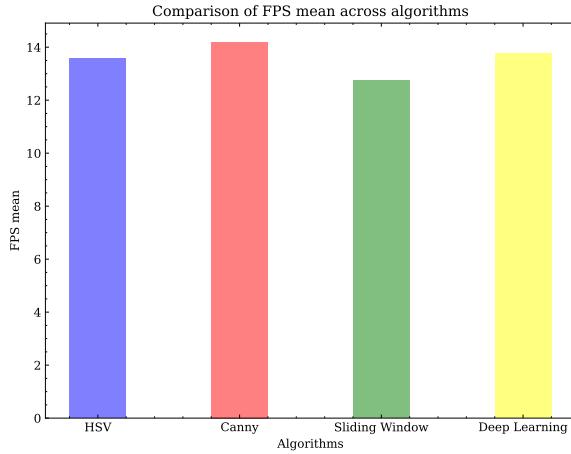


Figura 4.12: Gráfica de comparación de los FPS de todos los algoritmos

ahora vamos a ver una comparacion de las gráficas de las rutas que han seguido los distintos algoritmos. En general todos los algoritmos consiguen filtrar el carril de manera efectiva y seguir el mismo durante el recorrido entero. sin embargo exosten diferencias sutiles en los trayectos de cada algoritmos. Específicamente, los algoritmos más sencillos, como el método de Canny y la combinación de Canny con HSV, tienden a trazar las curvas de manera más cerrada y abrupta. Por otro lado, los algoritmos más complejos y robustos generalmente describen las curvas de una manera más abierta y natural, lo cual podría ser preferible en aplicaciones donde se busca una conducción más suave y segura.

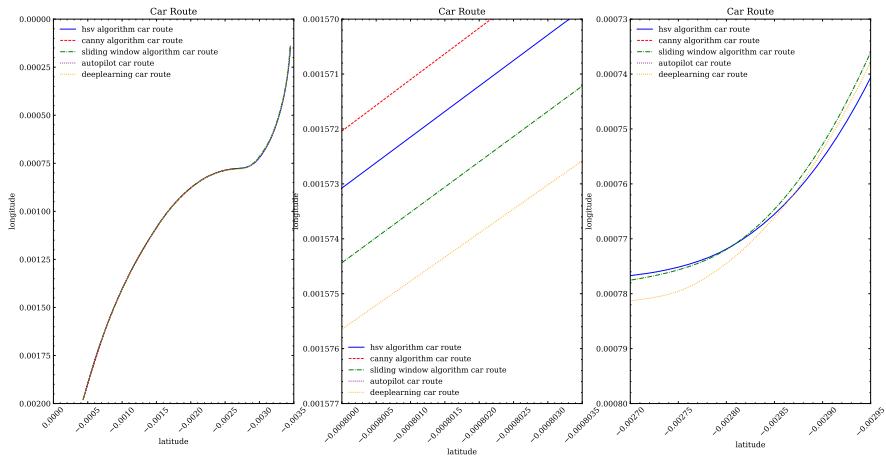


Figura 4.13: Gráfica de ruta seguida por todos los algoritmos siguiendo un carril

Más concretamente si miramos la gráfica de la figura 4.14 en donde se muestra que porcentaje de los movimientos totales representa cada tipo de movimiento(giros a la izquierda, a la derecha y ausencia de giros) y la gráfica de la figura 4.15 donde se observa la función ecdf de la intensidad de los giros realizados por el PID. Vemos como

en los métodos más complejos y especialmente en el algoritmo de detección basado en redes nueronales, se relizan un menor número de acciones de giros manteniendo al vehículo durante más tiempo recto sin tener que corregir su posición 4.14 y además observando la gráfica cdef de la insentsidad de los giros 4.15, vemos como además de que se realizan menos giros, estos también son menos intensos. Lo que podemos concluir de esta información es que debido a la mejora de detección el controlador PID tiene que realizar una menor cantidad de giros con menos intensidad para corregir la trayectoria y seguir el carril lo cuál resulta en una conducción más suave y fluida.

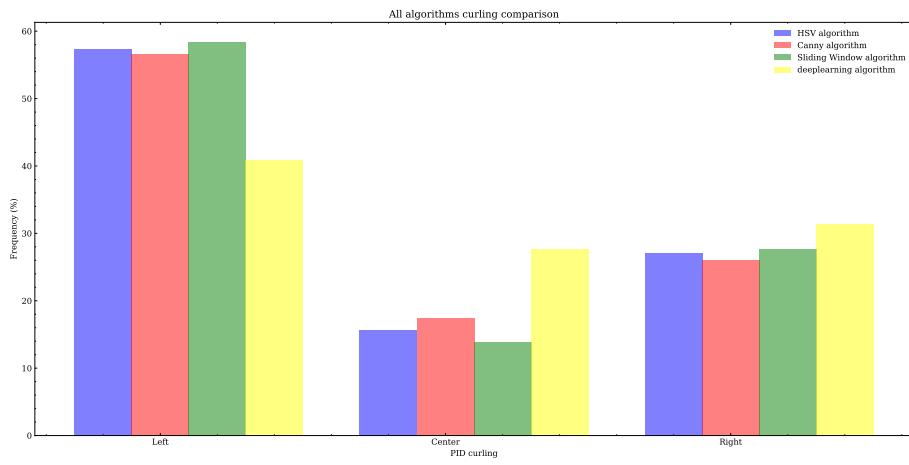


Figura 4.14: Gráfica de los tipos movimientos de los PID de todos los algoritmos

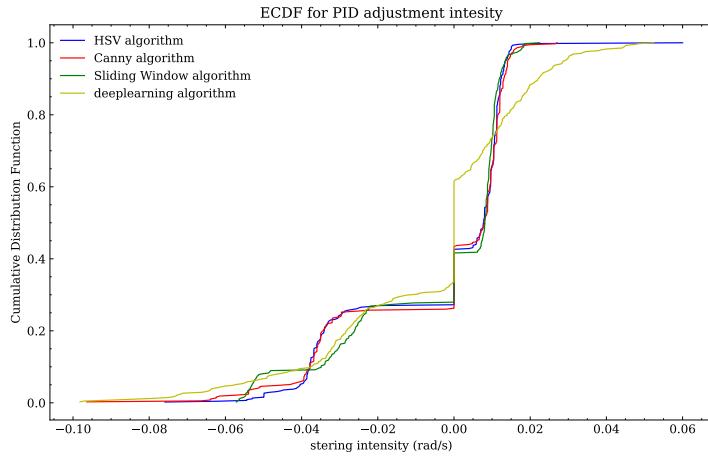


Figura 4.15: Gráfica de la función ecdef de la insensidad de los giros de cada algoritmo

En términos generales todos los algoritmos que hemos puesto a prueba logran, en su medida, filtrar el carril de manera lo suficientemente efectiva como para realizar un seguimiento del mismo. Sin embargo, aunque para nuestro caso concreto y en condiciones ideales los algoritmos sustentados por métodos más sencillos, como lo son el algoritmo basado en el filtro Canny y el algoritmo basado en filtro Canny + filtro

de color HSV, resulten eficaces tienen una enorme vulnerabilidad ante cambios de luz, sombras y brillos, esto genera que no sean algoritmos robustos ni generalistas que puedan funcionar de manera adecuada en un entorno real.

Por otro lado, los algoritmos más complejos como el basado en Sliding Window y el que utiliza una Red neuronal como método de detección demuestran una mayor robustez, siendo capaces de operar eficazmente en un rango más amplio de escenarios. No obstante, el algoritmo de *Sliding Window* ha mostrado algunas deficiencias en casos de iluminación extremadamente alta o ubicaciones muy oscuras, donde el algoritmo basado en Redes Neuronales ha tenido un rendimiento superior.

Además de esto, el algoritmo basado en Redes Neuronales ofrece una tasa de cuadros por segundo FPS más alta, debido a una inferencia más rápida y, por lo tanto, una mayor reactividad en la toma de decisiones del vehículo. Este método de detección por otro lado, también ofrece la flexibilidad de ser reentrenado para adaptarse a nuevos escenarios para los cuales actualmente no este preparado o para mejorar su rendimiento, sin la necesidad de modificar el código fuente. Esta es una ventaja significativa sobre el algoritmo de Sliding Window, donde cualquier mejora en de la aplicación requeriría una revisión y modificación sustancial del código. Por estos motivos y con los argumentos presentados se ha elegido el método

4.6. Evaluación de la Compatibilidad entre ROS y CARLA a través de ROS Bridge

Tras realizar probar distintos algoritmos utilizando RO2 y CARLA mediante el uso de ROS Bridge, podemos concluir que la posibilidad de controlar CARLA a través de los métodos y *topics* que ofrece ROS representa una herramienta sumamente útil. Este enfoque abre un amplio abanico de posibilidades para el desarrollo de nuevas aplicaciones dentro de este sistema operativo, las cuales pueden ser probadas en un entorno simulado tan realista como el que ofrece CARLA. Sin embargo, actualmente con la limitación de la tasa FPS impuesta por el cuello de botella que produce carla to ros bridge trabajar de manera profesional con estos dos entornos todavía no es posible. desarrollar aplicaciones de conducción autónoma con un máximo de 15 FPS en las imágenes que recibimos, aunque puede ser suficiente para la creación de prototipos, resulta insuficiente para aplicaciones más serias y completas.

Debido a esta restricción, se ha decidido que para las implementaciones finales de

este TFG no se utilizará ROS Bridge. En su lugar, se migrará todo el código necesario para continuar con el desarrollo de las aplicaciones finales a la API nativa de CARLA en python.

4.7. Sigue carril basado en redes neuronales y Qlearning

Tras la imposición de la red neuronal como el mejor método para la detección de carriles, se ha consolidado su papel central como método de filtrado en las implementaciones finales de los sistemas de seguimiento de carriles. El siguiente paso gira en torno a seleccionar un método para controlar el manejo del vehículo. Hasta este momento, se había empleado un controlador PID, el cuál constituye un enfoque tradicional y que funciona muy bien cuando se adapta específicamente a una tarea o un entorno determinado. No obstante, para ofrecer una solución final más generalista y robusta en más situaciones para los diseños de los sigue carriles finales se implementará un aproximación más innovadora basada nuevamente en IA. En particular se ha optado por un algoritmo de aprendizaje por refuerzo, en particular, una variante propia de Q-learning para controlar el manejo del vehículo.

Como se detalló en la sección 4.4.1, la máquina en la que se venía ejecutando los algoritmos presentados hasta ahora, mostró ciertas limitaciones, en lo que respecta a la capacidad gráfica del equipo. Esto se manifestó como una incapacidad para ejecutar de manera óptima la simulación en CARLA junto con el algoritmo de detección de carriles basado en redes neuronales al mismo tiempo.

Dadas estas limitaciones, se ha decidido trasladar el desarrollo, entrenamiento e implementación de los algoritmos que ahora se van a desarrollar al servidor Landau de la Universidad Rey Juan Carlos explicado con más detalle en la sección 3.2.5. En este entorno, la capacidad gráfica no representará una barrera, permitiendo así un desarrollo más ágil y eficiente de los distintos componentes del proyecto.

4.7.1. Sigue carril tradicional basado en Q-learning

Al implementar el algoritmo de Q-learning para el seguimiento de carriles, es esencial definir sus componentes fundamentales. En este contexto, identificamos los siguientes elementos clave:

- **Estados:** Para esta implementación, dividimos verticalmente la imagen de la cámara en múltiples franjas. Cada una de estas franjas representa un estado del algoritmo de Qlearning. Hay 11 franjas separadas equidistantemente por 20 pixeles empezando en el centro de la imagen y extendiéndose a cada lado, luego hay dos últimas franjas desde donde terminan estas 11 iniciales hasta el resto de la imagen, esto resulta en un total de 24 estados tal y como se ve en la figura 4.16. La determinación del estado actual se basa en las coordenadas de píxeles en los que se encuentra ubicado el centro del carril detectado.

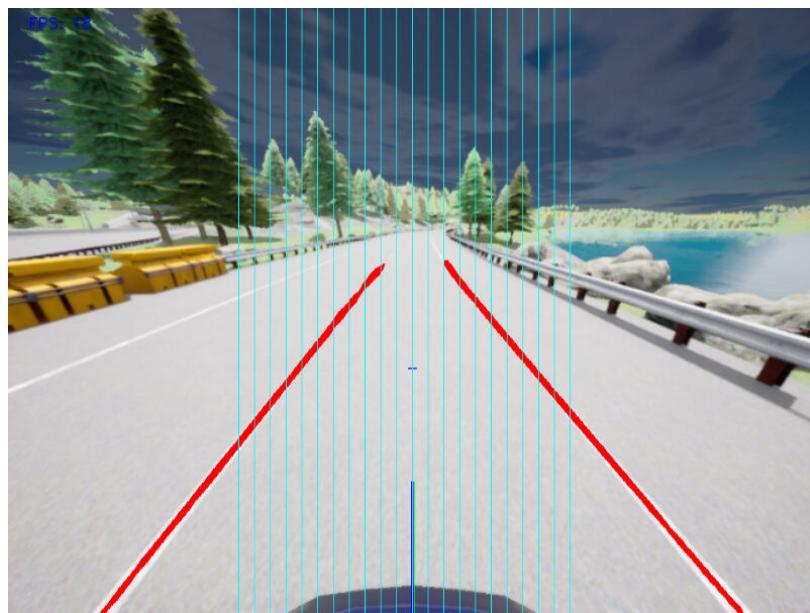


Figura 4.16: Estados del algoritmo de Q-learning

- **Agente:** El agente en este contexto es el vehículo en sí. Es el componente que realiza las acciones y busca maximizar la recompensa acumulada a lo largo del tiempo, lo que conduce a un comportamiento de conducción efectivo y seguro.
- **Acciones:** Se han definido una serie de acciones que el agente puede llevar a cabo. Estas acciones incluyen giros, donde especificamos la cantidad de rotación que se aplica al volante del vehículo. Hemos considerado 9 acciones para girar a la derecha, 9 acciones para girar a la izquierda y una acción de giro nulo, lo que suma un total de 19 acciones relacionadas con el control de la dirección. Además, hemos incorporado acciones relacionadas con la velocidad lineal, definiendo 4 velocidades lineales distintas. Estas acciones permiten regular la velocidad del vehículo a medida que avanza por el circuito. En código 4.20 podemos ver la implementación de las acciones definidas junto con sus valores para este caso. El gir del volante esta normizado entre -1 y 1 de manera que -1 es quivale a

girar el volante completamente a la izquierda, +1 equivale a girar el volante completamente a la derecha y 0 a mantener el volante en el centro

```

self.ACTIONS = [
    'forward', #stering = -0.0
    'left_1', #stering = -0.02
    'left_2', #stering = -0.04
    'left_3', #stering = -0.06
    'left_4', #stering = -0.08
    'left_5', #stering = -0.1
    'left_6', #stering = -0.12
    'left_7', #stering = -0.14
    'left_8', #stering = -0.16
    'left_9', #stering = -0.18
    'left_10', #stering = -0.2
    'right_1', #stering = 0.02
    'right_2', #stering = 0.04
    'right_3', #stering = 0.06
    'right_4', #stering = 0.08
    'right_5', #stering = 0.1
    'right_6', #stering = 0.12
    'right_7', #stering = 0.14
    'right_8', #stering = 0.16
    'right_9', #stering = 0.18
    'right_10' #stering = 0.2
]
self.SPEED = [
    'speed_1', #stering = 3.5 m/s
    'speed_2', #stering = 4.0 m/s
    'speed_3', #stering = 4.5 m/s
    'speed_4' #stering = 5.0 m/s
]

```

Código 4.20: Acciones disponibles para el algoritmo de qlearning

- **Entorno:** El entorno es el componente de un algoritmo de Qlearning donde el agente toma decisiones y ajusta su comportamiento en función de las acciones al estado actual y a la función recompensa. El entorno utilizado para el entrenamiento de este algoritmo es el mismo circuito que hemos estado utilizando en las pruebas de los demás algoritmos del proyecto. Dentro de este circuito además se han elegido tres puntos distintos para generar el vehículo a la hora de realizar el entrenamiento de Q-learning. Estas localizaciones además de el propio circuito se pueden ver en la figura 4.17

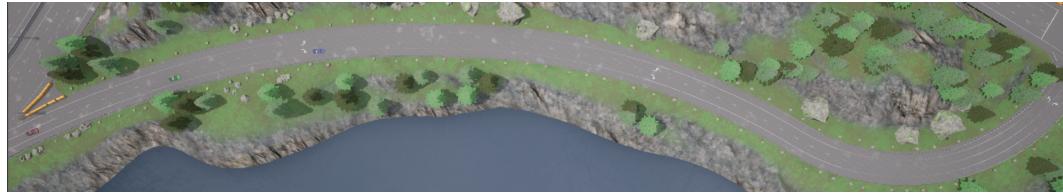


Figura 4.17: Entorno de entrenamiento del algoritmo de Q-learning

- **Función de Recompensa:** La función de recompensa desempeña un papel fundamental en la adaptación del comportamiento del vehículo. En esta implementación, la función de recompensa se diseña para fomentar velocidades lineales más rápidas, lo que impulsa al vehículo a mantener la velocidad media más alta posible. Además, se recompensa al vehículo por mantener el centro del carril alineado con el centro de la imagen, lo que garantiza que el vehículo se mantenga en una posición central en la carretera y además también se premia al vehículo por mantener un ángulo adecuado con respecto al carril, asegurando que se conduzca de manera paralela al mismo. La implementación de la función recompensa puede verse en el código 4.21 de a continuación.

```
def reward_function(self, error, angle_error, car_crashed):

    if self.lane_lines < 1:
        reward = 0.0
        return reward

    if car_crashed:
        reward = 0.0
        return reward

    normalized_error = abs(error)

    reward = (((1 / (normalized_error + 1)) + self.speed/100) -
              angle_error/100))

    reward = np.clip(reward, -1.0, 1.0)
    reward = 1 / (1 + np.exp(-reward))

    return reward
```

Código 4.21: Función recompensa del algoritmo de qlearning

- **Función de Recompensa:** Finalmente el último componente del algoritmo de Qlearning que nos queda por mencionar son los parámetros del algoritmo.

se optó por un ratio de aprendizaje de 0.5 para permitir un equilibrio en la incorporación de nuevas recompensas y conocimientos previos. El factor de descuento se estableció en 0.95 para dar importancia a las recompensas futuras en una secuencia de decisiones. Además, se inició con un ratio de exploración de 0.95 que se reducía en 0.1 cada cierto número de episodios hasta llegar a 0 cuando se establecía un valor fijo muy pequeño sin llegar a ser cero para el resto del entrenamiento como se puede observar en la figura 4.18. Esto se realizaba con el objetivo de explorar ampliamente el espacio de estados al principio y posteriormente enfocarse en explotar lo aprendido aunque sin cerrar completamente la ventana de seguir aprendiendo.

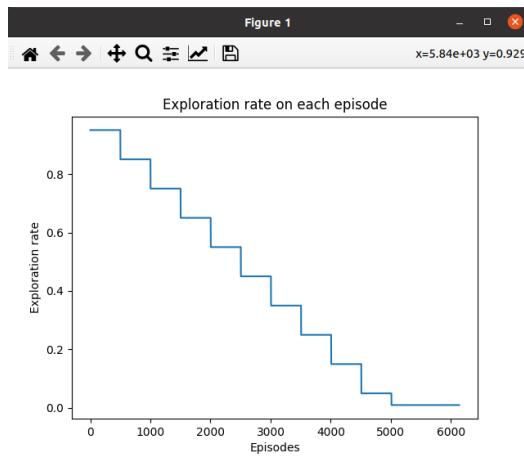


Figura 4.18: Evolución del ratio de exploración del algoritmo Q-learning

Para las fases de entrenamiento, la metodología consistía en dejar entrenando el algoritmo de Qlearning en el servidor landau de la URJC durante periodos que oscilaban entre las 12 y las 16 horas aproximadamente. De esta manera nos asegurábamos que el algoritmo realizará un entrenamiento completo y convergiera como se puede ver en la grafica

Como vemos el algoritmo converge en una franja de entr las 250 y las 300 unidades de recompensa y no siempre a una recompensa concreta como suele suceder en algoritmos de Q-learning. Esto no significa que el algoritmo no converja adecuadamente, ya que este fenomeno se debe al hecho de que este algoritmo ha sido entrenado en un circuito líneal como se puede ver en la figura 4.17 donde el agente empezaba cada episodio en uno de los tres sitios seleccionados, sin embargo siempre terminaba el mismo sitio (al terminar la carretera). Esto genera es que en los episodios donde el agente empieza más alejado de la meta al tener que recorrer más camino la recompensa que se acumula es mayor y en los episodios en los que el agente empieza más cerca del

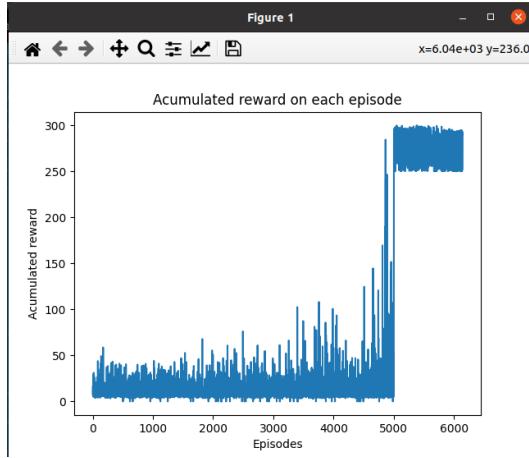


Figura 4.19: Evolución de la recompensa del algoritmo Q-learning

final del circuito al recorrer menos espacio la recompensa es menor. Dandonos como resultado una gráfica en la que a pesar que vemos que el algoritmo converge este lo hace en una franja amplia de valores dependiendo de donde comience el episodio.

Tras terminar el proceso de entrenamiento, evaluando ahora el funcionamiento del resultado obtenido, el algoritmo de Q-learning demostró un rendimiento altamente satisfactorio en la navegación por el carril. Se mantuvo una conducción fluida y segura, sin giros bruscos y manteniendo una trayectoria recta, tal como se puede apreciar en la figura 4.20 donde podemos ver como el mayor porcentaje de movimientos son mantener el volante recto sin realizar ningún giro o realizar correcciones pequeñas sin efectuar giros abruptos.

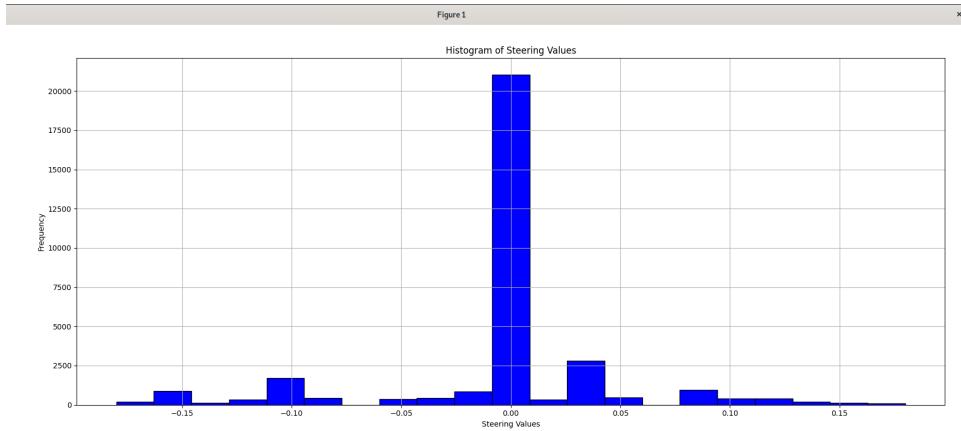


Figura 4.20: Histograma de las acciones tomadas por el agente de Q-learning tras el entrenamiento

El algoritmo final no solo es robusto, sino que también podría ser fácilmente extrapolable a otros circuitos. Mientras la detección del carril sea precisa, se espera que el vehículo navegue de manera adecuada. por todo tipo de circuitos con curvas

similares a las usadas en el entrenamiento. En el siguiente **video** se puede observar una demostración práctica del funcionamiento de esta solución de sigue carril.

4.7.2. sigue carril con reacción a obstáculos en la vía

En esta fase del proyecto, abordamos un nivel avanzado de complejidad en el sistema de seguimiento de carriles. Además de lograr una conducción fluida y segura en carreteras despejadas libres de objetos en la vía, nuestro objetivo ahora es garantizar que el vehículo pueda navegar de manera segura incluso cuando se enfrenta a obstáculos inesperados en la carretera. Para lograr esta funcionalidad, hemos integrado un sensor LiDAR en el vehículo, lo que nos permite detectar obstáculos en tiempo real. El sistema se basa en el uso de dos tablas Q en lugar de una. Cada tabla Q se emplea en situaciones específicas:

1. **Tabla Q para Navegación Normal:** En situaciones en las que no hay vehículos u obstáculos presentes en la carretera, el sistema utiliza una tabla Q diseñada para la navegación continua por el carril, como se ha demostrado en implementaciones anteriores. Esta tabla se enfoca en mantener una velocidad constante y un posicionamiento preciso en el carril.
2. **Tabla Q para Detección de Obstáculos:** Cuando el sensor LiDAR detecta la presencia de un obstáculo, el sistema cambia automáticamente a una tabla Q especializada para abordar este escenario. La tabla Q de detección de obstáculos se centra en tomar decisiones que eviten colisiones con los obstáculos detectados. Esto incluye estrategias para detener el vehículo o realizar maniobras de evasión en función de la proximidad y tamaño del obstáculo.

Para lograr que esta idea de usar dos tablas Q funcionara, era necesaria la realización de ciertos cambios clave en el código del algoritmo. Principalmente estos cambios debían realizarse en las funciones del programa que interactuaban con la tabla Q y consistían en añadir un if que determinara qué tabla se iba a utilizar en función de la presencia o ausencia de un obtáculo detectado por el LIDAR. A continuación se puede ver dos ejemplos de estos cambios realizados para la adaptación del código inicial de Qlearning con una tabla Q al código de este método con dos tablas Q, el código 4.22 se corresponde con la función que se encarga de actualizar la tabla Q y en código 4.23 se corresponde con la función de elegir una acción leyendo las entradas de la tabla Q.

```

def update_q_table(self, current_state, steering_action,
                   acceleration_action, reward, next_state, object_in_front):

    if object_in_front:
        current_q_table = self.q_table_with_object
    else:
        current_q_table = self.q_table_without_object

    future_max_q = np.max(current_q_table[next_state, :, :])

    current_q_value = current_q_table[current_state, steering_action,
                                       acceleration_action]
    new_q = (1 - self.learning_rate) * current_q_value + \
            self.learning_rate * (reward + self.discount_factor * future_max_q)

    current_q_table[current_state, steering_action, acceleration_action] = new_q

    if self.exploration_rate_counter > 800:
        self.exploration_rate = self.exploration_rate - 0.1
        self.exploration_rate_counter = 0

```

Código 4.22: Función para actualizar la tabla Q del algoritmo con reacción a obstáculos

```

def choose_action(self, state):

    if self.object_in_front:
        current_q_table = self.q_table_with_object
    else:
        current_q_table = self.q_table_without_object

    if np.random.uniform(0, 1) < self.exploration_rate:
        action = np.random.randint(len(self.ACTIONS))
        self.random_counter += 1
    else:
        action_values = current_q_table[state]
        action = np.unravel_index(action_values.argmax(),
                                  action_values.shape)[0]
        self.table_counter += 1

    return action

```

Código 4.23: Función para elegir una acción de la tabla Q del algoritmo con reacción a obstáculos

En adición a esto una modificación de la función recompensa también era necesaria para conseguir que el agente recibiera las recompensas correctas para lograr el nuevo comportamiento, estos consistieron en lo siguiente:

- En ausencia de obstáculos, la función de recompensa premia la navegación continua y efectiva por el carril, como se ha hecho en las implementaciones anteriores. Se fomenta mantener una velocidad constante y una posición central en el carril.
- Cuando se detecta un obstáculo, la función de recompensa otorga la máxima recompensa si el vehículo se detiene por completo para evitar una colisión inminente con el obstáculo. La recompensa disminuye gradualmente si el vehículo continúa avanzando hacia el obstáculo sin detenerse.

A continuación en el código 4.24 podemos ver la función recompensa actualizada con los cambios descritos

```

def reward_function(self, error, angle_error, car_crashed):

    if not self.object_in_front and self.speed == 0.0:
        reward = 0.0
        print("penaliza por parar ", self.speed)
        return reward

    elif self.object_in_front and self.speed == 0.0:
        reward = 1.0
        print("premia por parar ", self.speed)
        return reward

    if self.lane_lines < 1:
        reward = 0.0
        return reward

    if car_crashed:
        reward = 0.0
        return reward

    normalized_error = abs(error)

    reward = (((1 / (normalized_error + 1)) + self.speed/100) -
              angle_error/100))

    reward = np.clip(reward, -1.0, 1.0)

    reward = 1 / (1 + np.exp(-reward))

    print("reward: ", reward)
    return reward

```

Código 4.24: Función recompensa del algoritmo con reacción a obstáculos

Otra ventaja clave de este concepto de utilizar dos tablas Q es su alta modularidad y flexibilidad. La incorporación de dos tablas Q distintas, brinda la capacidad de entrenar y operar cada contexto por separado. En primer lugar, podemos entrenar la tabla Q destinada a la navegación en carriles despejados, perfeccionando así las habilidades de conducción fluida y segura en condiciones ideales. Simultáneamente, podemos llevar a cabo el entrenamiento de la tabla Q específica para la detección de obstáculos. Esto implica enseñar al vehículo a reaccionar adecuadamente cuando se encuentre con un obstáculo en la carretera. Una vez que ambas tablas Q estén entrenadas y afinadas de manera óptima, el sistema puede operar en dos modos distintos: en modo carretera despejada en modo "detección de obstáculos".

Una vez finalizada la implementación del algoritmo, avanzamos a la fase de entrenamiento, siguiendo la misma metodología que en la sección anterior. Para este proceso, se dejó entrenar el modelo en los laboratorios Landau de la universidad, pero con una duración extendida en comparación a la implementación 4.7.2 de aproximadamente 24 horas esto debido a que entrenar dos tablas Q de manera correcta suponía una tarea más demorada que la de entrenar una sola tabla. De esta manera las gráficas que obtenemos luego de este entrenamiento resultan muy parecidas a las anteriores. donde la gráfica de la recompensa acumulada que se puede ver en la figura 4.21 es muy similar a la del método 4.7.2 pero convergiendo ahora recompensas más altas, debido a las altas recompensas que obtiene nuestro algoritmo al detenerse ante obstáculos.

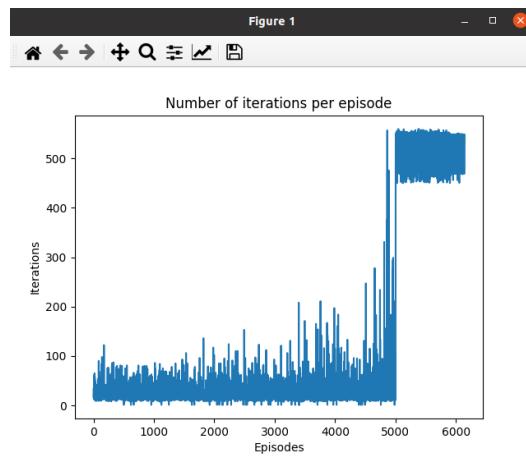


Figura 4.21: Evolución del ratio de exploración del algoritmo Q-learning

En cuanto a la gráfica de acciones observables en la figura ?? nuevamente podemos comprobar que la mayoría de acciones de giro ejecutadas por nuestro agente son la ausencia de giro, dejado el volante si mover o giros pequeños que no resultaran muy bruscos y favorecieran una conducción fluida.

Los resultados del entrenamiento fueron altamente satisfactorios. Al igual que en el experimento previo, se logró un comportamiento de seguimiento de carril fluido y eficaz. Una adición significativa fue la capacidad del vehículo para detenerse en presencia de un obstáculo en el carril y no reanudar la marcha hasta que este se hubiese retirado. Una demostración de este comportamiento exitoso se puede observar en el siguiente [vídeo](#).

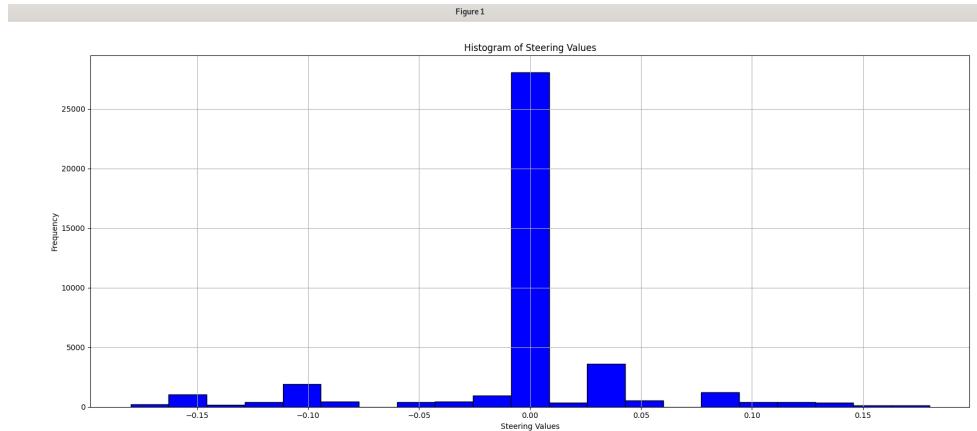


Figura 4.22: Histograma de acciones del algoritmo sigue carril con reacción ante obstáculos

4.7.3. Sigue carril con adaptación al tráfico

La última y más compleja implementación en este trabajo de investigación es un sistema de seguimiento de carriles capaz de adaptarse al tráfico en tiempo real. En este escenario, el vehículo debe tomar decisiones inteligentes tanto cuando se encuentra en condiciones de carretera despejada como cuando se enfrenta a obstáculos o vehículos en movimiento que circulan a una velocidad inusualmente lenta.

Para lograr esta adaptación al tráfico, se utiliza un sensor LiDAR para medir la distancia al obstáculo o vehículo de adelante. Se establecen varios umbrales de distancia, dividiendo el rango de medición en segmentos que van desde 0 a 2 metros, de 2 a 5 metros y de 5 a 10 metros. Dependiendo de en qué rango se encuentre el vehículo de adelante, el sistema tomará una decisión sobre la velocidad a la que debe circular.

La implementación se basa en la creación de múltiples tablas Q para el aprendizaje por refuerzo. En este caso, se utilizan cuatro tablas Q distintas. Una tabla Q se utiliza cuando no hay obstáculos en la carretera y el vehículo puede navegar normalmente. Las otras tres tablas Q se utilizan para cada uno de los umbrales de distancia mencionados anteriormente. Esto da como resultado un total de cuatro tablas Q, cada una diseñada para manejar un escenario específico.

La función de recompensa se modifica para premiar la toma de decisiones que mantienen una distancia segura con el vehículo de adelante y permiten una adaptación eficiente al tráfico. Se otorga una recompensa máxima cuando el vehículo se encuentra en el rango de distancia deseado y toma una velocidad dentro de las acciones de velocidad lineal predefinidas. Estas acciones de velocidad lineal también se ajustan

para incluir cuatro velocidades distintas: 0 m/s (detenerse), 2.5 m/s (adaptación al tráfico), 3.5 m/s (navegación normal) y 5 m/s (velocidad rápida).

Este enfoque permite que el sistema tome decisiones inteligentes y adapte su velocidad en función de las condiciones del tráfico en tiempo real. La modularidad de las tablas Q y la flexibilidad en la función de recompensa brindan un control preciso sobre el comportamiento del vehículo en una variedad de situaciones de conducción, lo que lo convierte en un sistema altamente adaptable y seguro.

Bibliografía

- [1] Wikipedia. La robótica. <https://es.wikipedia.org/wiki/Robot>.
- [2] Angel Eduardo Gil Pérez. Robótica móvil: Qué es y sus aplicaciones. <https://openwebinars.net/blog/robotica-movil-que-es-y-sus-aplicaciones/>.
- [3] Sae standards news: J3016 automated-driving graphic update. SAE Internacional, 2019.
- [4] Dirección General de Tráfico. Sistemas avanzados de ayuda a la conducción (adas), 2023.
- [5] Digitalization in modern transport of passengers and freight.
- [6] Robo-taxi service fleet sizing: assessing the impact of user trust and willingness-to-use. 2019.
- [7] Pasado, presente y futuro del coche autónomo. 2021.
- [8] Road safety, health inequity and the imminence of autonomous vehicles. 2021.
- [9] Not if, but when: Autonomous driving and the future of transit. 2018.
- [10] Identificación y detección de objetos móviles mediante redes neuronales empleando el sistema nvidia jetson tx2. 2020.
- [11] Aprendizaje por refuerzo. 2023.
- [12] Tfm: Autonomous driving in traffic using end-to-end deep learning. 2023.
- [13] que es python. <https://developer.oracle.com/es/learn/technical-articles/what-is-python>.
- [14] Andrii Mazur. Python para el desarrollo de ia: ¿por qué es tan importante?, 2022.
- [15] ROS Documentation. Ros documentation.

- [16] Reforcement Learning an introduction. *Título del Libro*. 2018.
- [17] Ros. Ros distributions.
- [18] carla to ros bridge oficial github. carla to ros bridge oficial github.
- [19] Wikipedia. Pygame.
- [20] Charles Wong. advanced lane finding using sliding window search.
- [21] Conducción autónoma — niveles y tecnología. <https://www.km77.com/reportajes/varios/conduccion-autonoma-niveles>.
- [22] que es python. <https://developer.oracle.com/es/learn/technical-articles/what-is-python>.
- [23] Pytorch team. Pytorch documentation.