



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Construcción de un coche de conducción autónomo
robótico.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Martínez Martínez, Andrés

Tutor/a: Monserrat Aranda, Carlos

Cotutor/a: Martínez Plumed, Fernando

CURSO ACADÉMICO: 2021/2022

Resumen

La conducción de coches totalmente autónomos reduciría considerablemente las muertes en carreteras y el estrés al volante. Aunque varias compañías han comenzado a desplegarlos en las calles en los últimos años, no se ha conseguido todavía una fiabilidad y escalabilidad suficientes para su adopción en masa.

Mientras que algunas de ellas prometen conseguirlo solo con varias cámaras y un pequeño procesador, otras creen que solo es posible con multitud de sensores y un maletero cargado de potencia de cómputo. El debate existente y la falta de una solución competitiva evidencian la necesidad de continuar investigando en el campo de la conducción autónoma. No obstante, debido al coste económico, la seguridad y la regulación, entre otros, resulta difícil para un particular experimentar con los coches autónomos y contribuir a su desarrollo.

Este proyecto acerca la conducción autónoma a una escala doméstica mediante la construcción de un coche robótico en miniatura. Equipando al vehículo con una Raspberry Pi y un acelerador para modelos de aprendizaje automático, se consigue que pueda ejecutar un modelo de aprendizaje profundo para la conducción autónoma en tiempo real. Este modelo, implementado en Python mediante el uso de redes neuronales convolucionales, permite al coche seguir las líneas de la calzada en circuitos que nunca ha visto previamente, y sin ser programado explícitamente para ello.

Palabras clave: conducción autónoma, visión artificial, aprendizaje automático, aprendizaje profundo, redes neuronales, redes convolucionales



Abstract

Autonomous driving cars would substantially reduce deaths on the road and driving stress. Although some companies have already started to deploy them on the streets over the last years, they haven't yet achieved enough reliability and scalability for mass adoption.

While some of them promise achieving it with only a few cameras and a small processor, others believe that it's only possible with many sensors and a trunk full of computers. The existing debate and the lack of a competitive solution evidence the need for continuing to investigate in the field of autonomous driving. However, due to the high economic cost, security issues and regulation, among others, it's hard for an individual to experiment with autonomous cars and contribute to its development.

This project brings autonomous driving closer to a domestic scale thanks to the construction of a small robotic car. By equipping the vehicle with a Raspberry Pi and an accelerator for machine learning models, it is capable to run a deep learning model for autonomous driving in real time. This model, implemented in Python using convolutional neural networks, allows the car to follow the lines of the road in circuits that it has never seen before, and without being explicitly programmed to do so.

Keywords: autonomous driving, computer vision, machine learning, deep learning, neural networks, convolutional neural networks

Índice de contenidos

| | |
|---|-----------|
| 1. Introducción..... | 9 |
| 1.1. Motivación | 9 |
| 1.2. Objetivos | 10 |
| 1.3. Estructura | 11 |
| 2. Estado del arte | 12 |
| 2.1. La visión por ordenador | 12 |
| 2.1.1. Las bases de la inteligencia artificial | 12 |
| 2.1.2. El auge de la visión por ordenador | 15 |
| 2.1.3. Avances recientes en la visión por ordenador | 16 |
| 2.2. La conducción autónoma | 19 |
| 2.2.1. Contexto del problema | 19 |
| 2.2.2. Propuestas de solución | 21 |
| 2.2.3. Percepción | 23 |
| 2.2.4. Planificación..... | 24 |
| 2.2.5. Control y supervisión | 26 |
| 3. Diseño de la solución | 27 |
| 3.1. Análisis del problema..... | 27 |
| 3.2. Planificación de necesidades | 28 |
| 3.2.1. <i>Hardware</i> | 28 |
| 3.2.2. <i>Software</i> | 30 |
| 3.3. <i>Hardware</i> empleado..... | 31 |
| 3.3.1. SunFounder PiCar-V Kit V2.0..... | 31 |
| 3.3.2. Baterías ecolle 18650 | 33 |
| 3.3.3. Raspberry Pi 3 Modelo B+ | 34 |
| 3.3.4. Acelerador Coral Edge TPU..... | 34 |
| 3.3.5. Micro SD Samsung EVO Plus 128 GB | 35 |
| 3.3.6. Accesorios | 35 |
| 3.4. <i>Software</i> empleado | 36 |
| 3.4.1. Sistema operativo: Raspberry Pi OS | 36 |
| 3.4.2. Python | 36 |
| 3.4.3. Bibliotecas: OpenCV, Tensorflow y Keras | 36 |
| 3.4.4. <i>Software</i> para la Edge TPU | 36 |
| 3.4.5. Interfaces de programación para el coche | 37 |
| 3.4.6. Entornos de desarrollo..... | 37 |



| | |
|--|-----------|
| 4. Implementación de la solución | 38 |
| 4.1. Montaje del vehículo..... | 38 |
| 4.2. Configuración e instalación del <i>software</i> | 41 |
| 4.3. Recopilación de los datos | 42 |
| 4.3.1. Conducción autónoma programada explícitamente | 43 |
| 4.3.2. Conducción manual..... | 49 |
| 4.4. Desarrollo del modelo de visión artificial | 54 |
| 4.4.1. Modelo basado en MobileNet V2 | 55 |
| 4.4.2. Modelo construido desde cero..... | 58 |
| 4.5. Despliegue y ajuste del modelo..... | 65 |
| 5. Conclusiones | 70 |
| 6. Referencias | 73 |

Índice de figuras

| | |
|--|----|
| Figura 1. Representación de una neurona artificial básica | 12 |
| Figura 2. Topología de un perceptrón multicapa..... | 13 |
| Figura 3. Ecuaciones del proceso de backpropagation..... | 13 |
| Figura 4. Representación de la arquitectura de LeNet | 14 |
| Figura 5. Filtros de la primera capa convolucional de AlexNet..... | 16 |
| Figura 6. Ejemplo de conexión residual en ResNet | 17 |
| Figura 7. Niveles de autonomía de la SAE..... | 20 |
| Figura 8. Comma Two instalado en un coche | 26 |
| Figura 9. Diagrama de los materiales necesarios para el proyecto y sus conexiones ... | 29 |
| Figura 10. SunFounder PiCar-V..... | 31 |
| Figura 11. PCB N°1 del SunFounder PiCar | 31 |
| Figura 12. PCB N°2 del SunFounder PiCar | 32 |
| Figura 13. PCB N°3 del SunFounder PiCar | 32 |
| Figura 14. Motor del SunFounder PiCar | 32 |
| Figura 15. Servos del SunFounder PiCar | 33 |
| Figura 16. Cámara del SunFounder PiCar..... | 33 |
| Figura 17. Baterías ecolle 18650 | 33 |
| Figura 18. Raspberry Pi Modelo 3 B+ | 34 |
| Figura 19. Acelerador Coral Edge TPU | 35 |
| Figura 20. Interfaz del sistema operativo Raspberry Pi OS | 39 |
| Figura 21. SunFounder PiCar-V durante el montaje..... | 40 |
| Figura 22. SunFounder PiCar-V tras completar el montaje | 41 |
| Figura 23. Fotograma de la cámara del coche ante un circuito con suelo reflectante... | 44 |
| Figura 24. Vista cenital del segundo y tercer circuito diseñados | 44 |
| Figura 25. Función para obtener los bordes rosas de una imagen..... | 45 |
| Figura 26. Fotograma del segundo circuito sin y con la máscara HSV aplicada | 45 |
| Figura 27. Vista de los bordes tras aplicar el recorte | 45 |
| Figura 28. Función para obtener las líneas blancas observadas en una imagen..... | 46 |
| Figura 29. Fotograma del segundo circuito con las líneas detectadas y la dirección a seguir | 47 |
| Figura 30. Fotograma de la cámara del coche ante una curva..... | 48 |
| Figura 31. Fotograma de la cámara del coche ante un giro brusco con su correspondiente recorte de la mitad superior | 49 |
| Figura 32. Extracto del método “drive” que aplica la conducción manual | 51 |
| Figura 33. Método para conducir mediante las entradas de teclado..... | 51 |
| Figura 34. Circuitos utilizados para la conducción manual | 52 |
| Figura 35. Distribución de los ángulos en las imágenes de entrenamiento y validación | 53 |
| Figura 36. Muestra de cuatro fotogramas de conducción con sus respectivos ángulos | 54 |
| Figura 37. Bloques convolucionales de MobileNet V2..... | 55 |
| Figura 38. Líneas de código que crean la red neuronal basada en MobileNet V2 | 56 |
| Figura 39. Arquitectura de la red neuronal basada en MobileNet V2..... | 57 |



| | |
|--|-----------|
| Figura 40. Evolución de la función de pérdida durante el entrenamiento del modelo basado en MobileNet V2..... | 58 |
| Figura 41. Representación gráfica de la arquitectura del modelo de Nvidia..... | 59 |
| Figura 42. Arquitectura de la red neuronal basada en el modelo de Nvidia | 60 |
| Figura 43. Evolución de la función de pérdida en el entrenamiento del modelo de Nvidia..... | 61 |
| Figura 44. Líneas de código para guardar los conjuntos de datos en caché..... | 62 |
| Figura 45. Salidas de las funciones ELU, Leaky ReLU y RELU para valores cercanos a 0..... | 63 |
| Figura 46. Evolución de la función de pérdida en el entrenamiento del modelo de Nvidia mejorado..... | 64 |
| Figura 47. Método para calcular el ángulo de giro mediante el modelo entrenado | 66 |
| Figura 48. Texto mostrado en el terminal al ejecutar el programa «smart_pi_car.py». | 67 |
| Figura 49. Circuito diseñado para realizar las pruebas de conducción autónoma..... | 68 |
| Figura 50. Evolución de la función de pérdida en el segundo entrenamiento del modelo de Nvidia mejorado | 69 |

1. Introducción

1.1. Motivación

Los accidentes de tráfico son una de las principales causas de muerte en países desarrollados, y la conducción autónoma tiene el potencial de reducirlos de forma drástica. Además, una conducción totalmente autónoma podría abaratar los costes de transporte de personas y mercancías y liberar tiempo a los conductores.

En los últimos años, algunas funciones de conducción asistida están llegando a manos de más conductores para aumentar la seguridad y la comodidad al volante. Redes de taxis autónomos han empezado a operar en varias ciudades, como Phoenix y San Francisco (1) (2), e incluso existen empresas que prometen vender coches totalmente autónomos antes de 2025 (3). Esto puede dar lugar a la creencia de que la conducción autónoma es un problema ya resuelto. No obstante, hay un amplio escepticismo entre los expertos en la materia sobre el cumplimiento de estas promesas (4).

Las soluciones propuestas por las empresas líderes del sector presentan una serie de problemas que evidencian la **necesidad de avances significativos** en la materia. Un conductor de un coche de Tesla¹ debe pagar \$15.000 (5) para dotar a su vehículo de una conducción autónoma aún en fase beta y que aún requiere de supervisión e intervenciones humanas (6). Por otra parte, los taxis autónomos de Waymo² solo operan en zonas poco pobladas (7) y su coste asciende a cifras superiores a \$100.000 (8). Además, requieren tanto de operarios en una central (9) como de un mapeado previo muy detallado de la zona (10).

Tanto en lo que respecta a los sensores de los coches, como a la potencia de cómputo necesaria, como a los algoritmos a utilizar, existe aún un amplio debate entre estas empresas e investigadores (11) (12). Además, cada año se presentan nuevos avances en la inteligencia artificial que cuestionan el paradigma anterior (13). Por todo esto y por la gran utilidad de esta tecnología, existe una motivación clara para seguir investigando en ella desde distintas áreas. Una de ellas, y la que ha supuesto el punto de inflexión en la viabilidad de la conducción autónoma, es la de la **visión por ordenador** (14). Gracias a algoritmos basados en inteligencia artificial y redes neuronales, es posible hacer que un coche interprete el entorno y tome decisiones de conducción en base a este (15).

Sin embargo, aunque algunas empresas han publicado el código de sus programas en internet³, la implementación y pruebas a escala real no son posibles para un particular. Además de las limitaciones obvias por seguridad y razones económicas, las necesidades computacionales de los mejores modelos de aprendizaje profundo (16) dificultan el desarrollo sin emplear granjas de servidores.

¹ <https://www.tesla.com>

² <https://waymo.com>

³ El *software* de conducción autónoma de Comma es de código abierto y está disponible en la dirección <<https://github.com/commaai>>



Por tanto, la motivación del proyecto reside en implementar una **solución a pequeña escala** para experimentar con un proyecto real de conducción autónoma. De esta forma, y haciendo público el trabajo, se pretende contribuir y facilitar la contribución de pequeños desarrolladores al avance de esta tecnología y acelerar así su adopción en masa.

1.2. Objetivos

Dada la motivación anterior, el objetivo principal del proyecto es conseguir que un pequeño coche robótico sea capaz de moverse de forma autónoma por un entorno controlado que simule una situación real de conducción.

La solución debe cumplir tres requisitos básicos:

- El coche debe recorrer el circuito completo **sin intervención humana**.
- El coche debe “aprender” a conducir sin ser explícitamente programado para ello. Esto es posible **empleando algoritmos de visión por ordenador**.
- El programa de conducción debe ejecutarse **en el hardware del coche** y sin conexión a internet.

Para conseguir una autonomía total, el coche debería responder a señales de audio como sirenas o avisos de agentes y debería poder reaccionar ante infinidad de objetos distintos. Solo algunas empresas punteras han conseguido esto, y ha sido tras una gran inversión y años de trabajo de equipos multidisciplinares. Por tanto, el objetivo del trabajo no es conseguir una autonomía total, sino una autonomía parcial que permita al coche **mantenerse dentro de las líneas de la calzada sin intervención humana**. Además, debe poder hacer esto en calzadas que no haya recorrido antes y en carreteras (suelos) de distinto color, siempre que las líneas de la calzada sean del color esperado.

Para conseguir este hito se requiere un trabajo en varios frentes. Por tanto, el proyecto se ha dividido en tres subobjetivos:

- Construir un coche robótico programable y con sensores que aporten información suficiente del entorno para permitir una conducción autónoma.
- Implementar un programa, incluyendo un modelo de inteligencia artificial entrenado, que permita al coche conseguir la autonomía parcial mencionada en este epígrafe.
- Ejecutar el programa en tiempo real y de forma local desde el robot, incluida la inferencia de la red neuronal (cálculo del resultado a partir de los datos de entrada).

Se darán los tres objetivos por conseguidos si el coche es capaz de recorrer un circuito no visto previamente de al menos dos metros de extensión sin salir de la calzada, en ambos sentidos y al primer intento.

1.3. Estructura

La presente memoria queda estructurada de la siguiente forma. En primer lugar, en el capítulo 2, se hace una revisión del estado del arte de la conducción autónoma. Por un lado, desde el marco teórico, estudiando los avances recientes en la visión por ordenador que hacen posible esta tecnología y que se utilizarán en el proyecto. Por el otro lado, desde el punto de vista práctico, analizando las soluciones implementadas por las empresas líderes y sus particularidades, con vistas a utilizar ese conocimiento para optimizar la solución aplicada.

A continuación, en el capítulo 3, se detalla la problemática que plantea el proyecto y se desarrolla una propuesta inicial de solución. Se listan las necesidades a cubrir, para posteriormente listar y describir los principales elementos escogidos para suplirlas, tanto de *hardware* como de *software*.

En el cuarto capítulo del trabajo se documenta el proceso de implementación de la propuesta anterior, con los problemas encontrados y soluciones adoptadas. En primer lugar, se explican el montaje del vehículo y la instalación y configuración del *software*, tras lo cual se obtiene un coche programable listo para conducir. Después, se continúa con la recolección de los datos, el diseño y el entrenamiento de la red neuronal empleada, para terminar con las pruebas de conducción y la optimización de la red.

Por último, en el capítulo 5, se presentan las conclusiones del proyecto, evaluando el cumplimiento de los objetivos propuestos en el epígrafe anterior. Para finalizar, cabe mencionar que el código desarrollado se encuentra accesible en el siguiente repositorio: <https://github.com/andresmm98/Smart-Pi-Car>.



2. Estado del arte

2.1. La visión por ordenador

Una de las principales disciplinas científicas que hacen posible un proyecto como este es la visión artificial, o visión por ordenador. Este término hace referencia a la capacidad de un ordenador de procesar, analizar y extraer conclusiones a partir de imágenes digitales, incluso sin haberlas visto antes. Dicha capacidad es posible gracias al uso de inteligencia artificial, que permite que, dada una serie de imágenes, un ordenador pueda encontrar patrones entre ellas y buscarlos en futuras imágenes aún no vistas.

En la actualidad, la visión por ordenador es aplicada de muchas formas y en muchos sectores. Desde el uso de drones para monitorizar campos de cultivo, hasta la detección de tumores y enfermedades, o los robots empleados en las líneas de fabricación. En los siguientes subepígrafes se describen los avances que han dado lugar a la adopción de esta tecnología, y en especial los que serán utilizados en este trabajo.

2.1.1. Las bases de la inteligencia artificial

A pesar de su reciente popularidad, el concepto de inteligencia artificial no es algo nuevo. Se suele considerar que este término pasó de idea a un campo formal de investigación alrededor de la conferencia de Dartmouth, en 1956. En una carta previa a la misma, se refieren a él como la creación de máquinas capaces de aprender por sí mismas, abstraer conocimiento y ser creativas (17).

En esta misma carta, los autores mencionan el trabajo de Warren McCulloch y Walter Pitts, que en 1943 habían creado ya un modelo de **neurona artificial** (18), similar al mostrado en la figura 1. Esta neurona es equivalente a un nodo, que realiza una suma ponderada de los valores de sus conexiones de entrada y aplica una función no lineal, conocida como **función de activación**, para devolver un valor de salida. Esta unidad de cómputo, inspirada en las neuronas biológicas, es la unidad básica sobre la que construyen las redes neuronales.

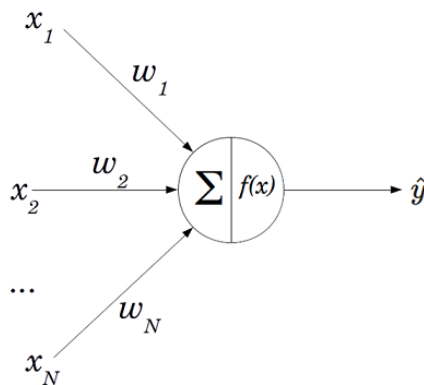


Figura 1. Representación de una neurona artificial básica
Fuente: (19)

Unos años después, en 1958, el psicólogo Frank Rosenblatt diseñó lo que algunos consideran la primera **red neuronal artificial**, a la que llamó “Perceptron” (20). La arquitectura de una red neuronal, representada en la figura 2, consiste en una serie de capas de neuronas conectadas entre sí, con unos valores en esas conexiones llamados **pesos**, que determinan la importancia relativa de las neuronas de entrada para la de salida.

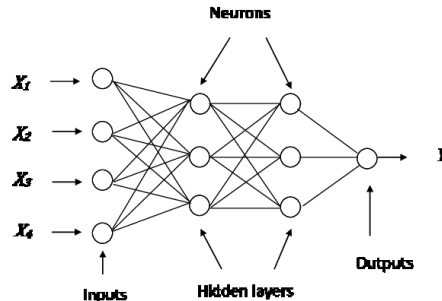


Figura 2. Topología de un perceptrón multicapa
Fuente: (21)

A pesar de la euforia inicial, las grandes expectativas a raíz de estos avances provocaron una posterior decepción por no materializarse en avances tecnológicos. Esto llevó a una reducción en la inversión e investigación en inteligencia artificial durante la década de los 70, período que fue popularmente conocido como un “invierno de la IA” (22).

Sin embargo, aunque las aplicaciones de la inteligencia artificial aún fueran escasas y la potencia de cómputo insuficiente, muchas de las herramientas que hoy día se utilizan tienen su origen en aquellos años. El algoritmo de **backpropagation** (figura 3) fue ideado en 1960 y se empezó a implementar en 1986 (23). Este algoritmo es la base del entrenamiento de las redes, permitiendo usar el error de la salida de la red para ajustar y mejorar los pesos de las conexiones previas. Mediante derivadas parciales en un proceso conocido como **descenso de gradiente**, se computa el valor que debería haber tenido cada peso para minimizar el error y se aplica una **tasa de aprendizaje** para ajustar el valor de los pesos en esa dirección.

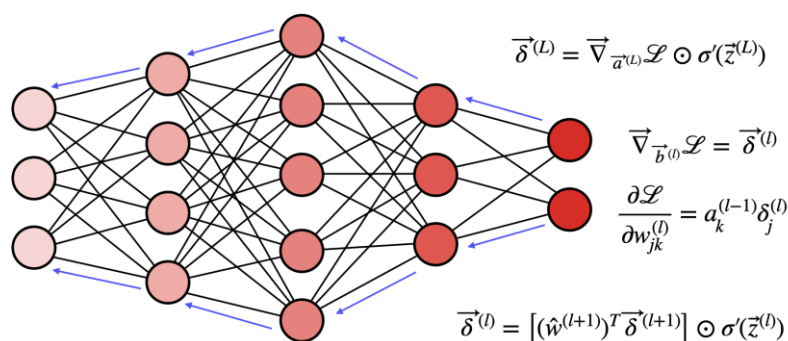


Figura 3. Ecuaciones del proceso de backpropagation
Fuente: HashPi



Así, el proceso de entrenamiento consiste, primero, en un paso de *forward propagation*, donde se calcula la salida de la red para una entrada dada y el error de la salida obtenida respecto a la deseada. A continuación, se realiza el *backward propagation*, donde se propaga el error hacia atrás para ajustar los pesos. Repitiendo estos dos pasos un número suficiente de veces y con suficientes datos bien **etiquetados**, es decir, con su valor de salida deseado, se consigue que la red devuelva la respuesta correcta a datos que nunca ha procesado. Cualquier dato que pueda ser codificado en bits puede usarse como dato de entrada, por lo que estas redes pueden usarse para traducir textos, predecir el resultado de modelos matemáticos o reconocer imágenes, entre otros.

De esta forma, las redes neuronales pueden “aprender” a encontrar patrones en datos de entrada de muchos tipos, sin ser programadas explícitamente para ello. Esta capacidad de aprender dio lugar al campo conocido como *machine learning* o **aprendizaje automático**, que aplica de forma práctica el concepto de inteligencia artificial utilizando algoritmos de aprendizaje como el descenso de gradiente. En concreto, los métodos de aprendizaje suelen clasificarse en **supervisados** y **no supervisados** según si los datos están etiquetados o no.

A partir de las primeras implementaciones de la propagación hacia atrás, las redes neuronales se fueron especializando para mejorar el rendimiento frente a distintos tipos de datos. Las **redes neuronales recurrentes** se inspiran en el mismo artículo anterior de Rumelhart et al. (23). Estas son un tipo de red pensado para procesar secuencias de datos que aún se utiliza en la actualidad, en especial para el procesamiento de texto.

En 1998, Yann LeCun y otros presentaron una de las primeras **redes convolucionales**, que más adelante fue bautizada como LeNet (24). Su trabajo planteó la innovación de aplicar filtros a las imágenes para detectar patrones, y pasar esas capas de patrones detectados a través de unas neuronas convencionales para clasificar imágenes de dígitos dibujados a mano.

Estos filtros son entrenables, suelen ser de 3x3 o 5x5 píxeles y se aplican sobre cada píxel. Esto hace que, durante el entrenamiento de la red, cada uno se optimice en detectar pequeños patrones como bordes y formas a lo largo de toda la imagen. Mediante estas capas, combinadas con el uso frecuente de *pooling layers* o **capas de agrupación**, la red disminuye en anchura y aumenta en profundidad, como se muestra en la figura 4. Finalmente, a esas capas conocidas como mapas de características, se añade un perceptrón multicapa para obtener la salida del modelo. Esta salida puede ser la clasificación de una imagen entre un conjunto de objetos, o la posición de un objeto concreto en una imagen.

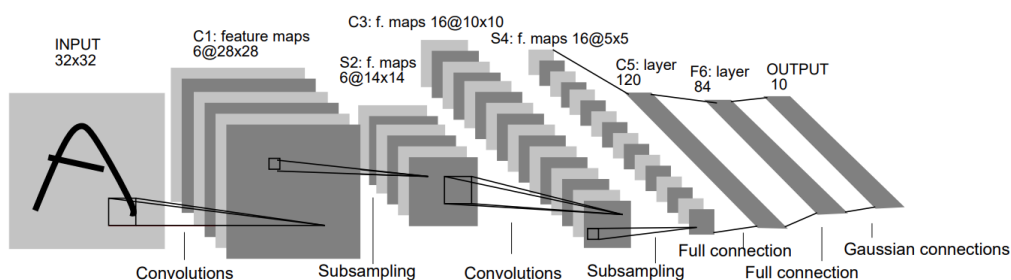


Figura 4. Representación de la arquitectura de LeNet
Fuente: (24)

2.1.2. El auge de la visión por ordenador

Como se ha comentado a lo largo del subepígrafe anterior, las bases teóricas de la inteligencia artificial fueron desarrolladas durante el siglo XX, pero no ha sido hasta el siglo XXI cuando se han comenzado a aplicar en muchos ámbitos de la vida cotidiana. En concreto, en la última década, las redes neuronales han experimentado mejoras muy grandes en su rendimiento que han hecho que del aprendizaje automático surjan varias áreas de aplicación, como el procesamiento del lenguaje natural o la visión por ordenador.

Esta mejora se ve plasmada, por ejemplo, en la tasa de error obtenida en la competición ILSVRC de clasificación de imágenes del conjunto de datos ImageNet, considerada una referencia en el sector (25). Mientras que en 2011 la mejor red neuronal tenía una tasa de acierto del 50,9%, en 2021 esa cifra alcanzó el 90% (26). Esto se debe a un conjunto de razones, de las que se destacan algunas a continuación.

- La creación de **grandes conjuntos de datos** de forma pública en la web, con más de un millón de imágenes y mil categorías. Algunos de los más utilizados son ImageNet o xView, que han permitido el entrenamiento de grandes redes neuronales.
- El aumento de la potencia de cómputo y **la paralelización de las operaciones gracias a las GPU**. Un ejemplo de ello es que la red LeNet antes mencionada necesitó tres días de entrenamiento en una estación de trabajo SUN-4/260 en 1998. Esa misma red, utilizando el último modelo del MacBook Air, puede ser entrenada actualmente en unos 90 segundos (27).
- **Mejoras en software e infraestructura**, como la computación en la nube, lenguajes de programación y bibliotecas de alto nivel, como Python, PyTorch y Keras, plataformas como Github, etc.

Estos y muchos otros factores, sumados a la gran financiación destinada por grandes empresas como Facebook y Alphabet y a la gran demanda empresarial de soluciones de inteligencia artificial, han hecho que el número de artículos científicos publicados relacionados con el aprendizaje automático se haya multiplicado por 10 en los últimos 15 años (28). Y, por supuesto, han hecho que grandes proyectos como la conducción autónoma pasen de la ciencia ficción a la realidad.

En esta explosión de la inteligencia artificial, un punto de inflexión fue el año 2012, cuando la posteriormente conocida como AlexNet consiguió una tasa de error en la competición ILSVRC del 15,4% para el top 5. Esto significa que, en el 84,6% de las ocasiones, la etiqueta correcta de la imagen estaba entre las cinco a las que el modelo había dado una probabilidad mayor. La tasa de error de AlexNet fue 10,8 puntos porcentuales más baja que el segundo mejor modelo de la competición. Esto fue posible gracias a una implementación eficiente de una red convolucional para ser entrenada en GPU, utilizando capas de *max-pooling*, que agrupan píxeles en el valor más alto de estos (29). En la figura 5 se muestra una imagen del artículo, donde pueden verse los filtros de la primera capa convolucional aprendidos por la red tras el entrenamiento.



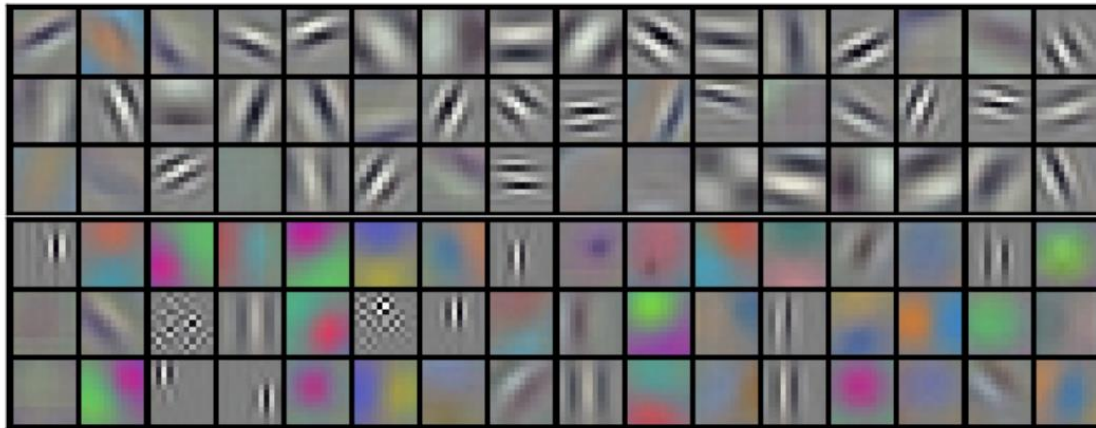


Figura 5. *Filtros de la primera capa convolucional de AlexNet*
Fuente: (29)

A partir de ese momento, las redes neuronales fueron aumentando en número de capas y parámetros, dando origen a una nueva rama del aprendizaje automático conocida como *deep learning* o **aprendizaje profundo**. Un modelo popular de aprendizaje profundo presentado en 2020, GPT-3, alcanza las 96 capas y los 175 mil millones de parámetros (16). Sin embargo, su entrenamiento requirió el equivalente a 3640 procesadores de 1 PFLOPS trabajando durante 24 horas. Incluso una red convolucional más ligera, como EfficientNet-B7, cuenta con 66 millones de parámetros y requirió el equivalente a 37 mil millones de FLOPS para su entrenamiento (30).

El elevado coste computacional dificulta la accesibilidad de la inteligencia artificial y de la visión por ordenador a un particular. Sin embargo, alguna de las soluciones que se comentarán en el siguiente subepígrafe permiten reducir el problema.

2.1.3. Avances recientes en la visión por ordenador

Desde la publicación de AlexNet en 2012 y, debido a los diversos factores comentados en los párrafos previos, el avance de las redes neuronales se ha acelerado, y esto se ha trasladado al campo de la visión artificial.

Además del aumento de uso de las GPU, otro factor diferencial para este campo ha sido la generalización del uso de las **redes convolucionales** explicadas en el subepígrafe 2.1.1, que se han convertido en el estándar para la clasificación de imágenes y detección de objetos (31). En los siguientes párrafos se describen algunas mejoras que se han ido realizando a la arquitectura de las redes convolucionales desde el año 2012. Estas serán consideradas para la elección y la optimización del modelo durante el desarrollo posterior.

Después de 2012, los siguientes años destacaron por el aumento del tamaño de las redes. Por ejemplo, GoogLeNet ganó la competición ILSVRC en 2014 con un error en el top 5 del 6,7% (32). Esto fue posible gracias a la introducción de los *inception modules*, módulos que aplican convoluciones con distinto tamaño de filtro de forma simultánea, y después concatenan el resultado. Sin embargo, otro detalle interesante fue la sustitución de los perceptrones multicapa por una simple **capa de agrupación por promedio**, mucho más liviana computacionalmente.

Un año más tarde, Microsoft Research Asia presentó ResNet (33), consiguiendo una tasa de error del 3,6% y aumentando considerablemente la profundidad del modelo, con 152 capas respecto a las 22 de GoogLeNet (32). La viabilidad de este aumento reside en que, tras la primera capa convolucional, el modelo reduce las dimensiones de la imagen a 56x56. Sin embargo, no perdieron rendimiento gracias a la introducción de **conexiones residuales**, que permiten conectar capas no consecutivas entre sí y evitar perder información útil en las convoluciones, tal y como se muestra en la figura 6.

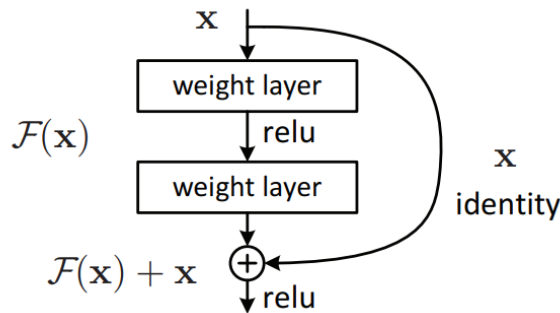


Figura 6. Ejemplo de conexión residual en ResNet
Fuente: (33)

En los años posteriores, la investigación avanzó para conseguir redes más pequeñas y eficientes, como MobileNet en 2017 (34). MobileNet separó el proceso tradicional de convolución en dos capas:

- Una capa de filtrado (*depthwise convolution*), que aplica un filtro a los 3 canales (RGB) de la imagen por separado.
- Una capa de combinación (*pointwise convolution*), en la que aplica un filtro de 1x1 para combinar los 3 canales anteriores.

De esta forma, siendo N el número de capas resultantes del filtro y D_K la anchura o altura del filtro, consiguieron una reducción en computación equivalente a:

$$\frac{1}{N} + \frac{1}{D_K^2}$$

En la práctica, esto se tradujo en un rendimiento similar al de GoogleNet con un coste computacional de un tercio, como muestran en el artículo, algo revolucionario para aplicaciones a pequeña escala, como por ejemplo la de este trabajo.

Por otra parte, también en 2017 fue presentada una nueva arquitectura que está revolucionando distintos campos de la inteligencia artificial: los **transformers** (13). Introducidos por un equipo de investigación de Google, estos modelos sustituyen las conexiones recurrentes por módulos de atención. Aplicando un codificador posicional a cada elemento de una secuencia y codificando también la importancia relativa de cada elemento respecto a otro, se consigue relacionar distintos elementos entre sí, por muy lejos que estén en la secuencia.

Aunque el uso principal de los **transformers** se ha dado en el campo del procesamiento del lenguaje natural, también está rivalizando en rendimiento con las redes



convolucionales en el campo de la visión por ordenador. Así, en los últimos años se han presentado varios modelos para intentar eliminar las convoluciones, como por ejemplo el Vision Transformer (35), que aplicó los módulos de atención a las imágenes segmentándolas en parches. Sin embargo, otros nuevos modelos, como el de ConvNeXt (36), han mostrado como las redes convolucionales siguen parejas en rendimiento y, dado que los *transformers* requieren un coste computacional más alto para tener un buen rendimiento, como se menciona en este mismo artículo, son de menor interés para un proyecto como este.

Por último, se considera importante mencionar otros avances más allá de los nuevos modelos, que también serán útiles en la posterior implementación del modelo.

Uno de ellos es la técnica conocida como *transfer learning*. Gracias a que las capas convolucionales detectan patrones útiles para reconocer objetos muy distintos, como los bordes y formas de la figura 5, las redes ya entrenadas se pueden reutilizar como base para otra red neuronal. Esto supone una generalización del aprendizaje que recuerda a la de los humanos. Sustituyendo las últimas capas del modelo, se puede ajustar un gran modelo de aprendizaje profundo para el problema deseado sin necesidad de entrenarlo desde cero. En concreto, la última capa del modelo es la que proporciona la salida. Por tanto, dependiendo de si queremos obtener una salida numérica (problema de **regresión**) o la clase de un objeto (problema de **clasificación**), añadiremos su correspondiente última capa y **función de pérdida** (función para computar el error del modelo).

Por otra parte, se han popularizado nuevas técnicas que han optimizado el rendimiento de las redes desde distintos frentes. No se va a profundizar en todas ellas, pero las más destacables y que se utilizarán más adelante son las siguientes:

- Nuevos métodos de optimización con potencial de reemplazar el descenso de gradiente: Momentum, Momentum de Nesterov y Adam.
- Nuevas funciones de activación: RELU, ELU, GELU
- Nuevas técnicas de inicialización de los pesos: Xavier uniforme y normal
- Otras nuevas técnicas para mejorar la generalización del aprendizaje. Una de ellas son las capas de *dropout*, que desactivan algunas neuronas de forma aleatoria para que la red no dependa demasiado de ellas. Otra es la normalización por lotes, otra capa que normaliza los valores de entrada de la siguiente capa antes de que entren en ella.

Por último, además de estas técnicas, en las redes neuronales existen una serie de parámetros que deben escogerse manualmente y que condicionan los resultados. Entre ellos se encuentran la tasa de aprendizaje, el tamaño de los lotes de datos para el entrenamiento, el número de etapas del entrenamiento, etc. Estos parámetros suelen recibir el nombre de **hiperparámetros**. La investigación también ha avanzado en estos frentes, llegando en muchos casos a un consenso sobre qué valores son los óptimos para mejorar el rendimiento, en función de las particularidades de cada modelo.

Así pues, a lo largo del epígrafe 2.1 se han presentado las distintas herramientas que permitirán desarrollar un programa de visión artificial lo más eficiente posible, para conseguir la conducción autónoma del coche. Sin embargo, dado que el *software* no es el

único aspecto relevante, y conviene aterrizar estos conceptos teóricos, en el siguiente epígrafe se analizan las distintas soluciones implementadas en la actualidad por las empresas líderes en conducción autónoma.

2.2. La conducción autónoma

2.2.1. Contexto del problema

Según la OCDE, en 2019 murieron en España 37 personas por cada millón de habitantes en accidentes de tráfico (37). Según el mismo estudio, en Estados Unidos esta cifra asciende a 107 personas. Si tenemos en cuenta que, según un estudio de la NHTSA, el porcentaje de accidentes causados por error humano oscila entre el 94 y el 96% (38), podemos observar el gran impacto en vidas humanas que tendría minimizar este error.

Los sistemas de conducción asistida y autónoma son una forma de conseguirlo, que, además, supondrían un gran ahorro de mano de obra en caso de eliminar completamente el conductor. En la actualidad, diversos sistemas de conducción asistida, conocidos como **ADAS** (Advanced Driver Assistance System), están implantados en muchos de los vehículos nuevos vendidos cada año. Según la Asociación Americana del Automóvil, el 92,7% de los vehículos nuevos en Estados Unidos en mayo de 2018 tenía disponible al menos una funcionalidad ADAS (39). Estas son, entre otras, aparcamiento automático, dirección asistida, control de velocidad y frenado de emergencia.

Sin embargo, no ha sido hasta 2020 cuando se ha ofrecido por primera vez al público un servicio de coches sin conductor. Así pues, para evitar engaños al consumidor respecto a las funciones de asistencia y autonomía, en la industria del automóvil se emplea un marco de referencia para catalogar el grado de autonomía de un vehículo, que parte de un documento publicado por la Sociedad de Ingenieros de Automoción⁴ (**SAE**) en 2014 (40).

La SAE distingue entre seis niveles de autonomía que serán descritos a continuación y se muestran en la figura 7: SAE Level 0, SAE Level 1, SAE Level 2, SAE Level 3, SAE Level 4 y SAE Level 5.

⁴ www.sae.org





SAE J3016™ LEVELS OF DRIVING AUTOMATION™

Learn more here: sae.org/standards/content/j3016_202104

Copyright © 2021 SAE International. The summary table may be freely copied and distributed AS-IS provided that SAE International is acknowledged as the source of the content.

| | SAE LEVEL 0™ | SAE LEVEL 1™ | SAE LEVEL 2™ | SAE LEVEL 3™ | SAE LEVEL 4™ | SAE LEVEL 5™ |
|--|---|--------------|--------------|--|--|--------------|
| What does the human in the driver's seat have to do? | You are driving whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering | | | You are not driving when these automated driving features are engaged – even if you are seated in “the driver’s seat” | | |
| | You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety | | | When the feature requests, you must drive | These automated driving features will not require you to take over driving | |

Copyright © 2021 SAE International.

| | These are driver support features | | | These are automated driving features | | |
|----------------------------|---|---|---|---|--|---|
| What do these features do? | These features are limited to providing warnings and momentary assistance | These features provide steering OR brake/acceleration support to the driver | These features provide steering AND brake/acceleration support to the driver | These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met | This feature can drive the vehicle under all conditions | |
| Example Features | <ul style="list-style-type: none"> • automatic emergency braking • blind spot warning • lane departure warning | <ul style="list-style-type: none"> • lane centering OR • adaptive cruise control | <ul style="list-style-type: none"> • lane centering AND • adaptive cruise control at the same time | <ul style="list-style-type: none"> • traffic jam chauffeur | <ul style="list-style-type: none"> • local driverless taxi • pedals/steering wheel may or may not be installed | <ul style="list-style-type: none"> • same as level 4, but feature can drive everywhere in all conditions |

Figura 7. Niveles de autonomía de la SAE

Fuente: (40)

En primer lugar, hay tres niveles considerados como **conducción asistida**, en los que el conductor debe supervisar en todo momento la conducción del vehículo y maniobrar cuando sea necesario.

- **SAE Level 0:** sin automatización de la conducción. Funciones limitadas a alertar al conductor y darle una asistencia momentánea, como frenado automático, alerta de punto ciego y aviso de cambio de carril.
- **SAE Level 1:** conducción asistida. Funciones que proporcionan dirección asistida o frenado y aceleración automáticos, pero no ambas.
- **SAE Level 2:** automatización de la conducción parcial. Funciones que proporcionan dirección asistida y frenado y aceleración automáticos.

En segundo lugar, hay tres niveles considerados como **conducción autónoma**, en los que el conductor no debe supervisar la conducción cuando las funciones están activadas.

- **SAE Level 3:** automatización de la conducción condicional. El vehículo puede ser conducido de forma autónoma bajo condiciones limitadas (por ejemplo, durante atascos) y el conductor debe conducir cuando el sistema se lo pida.
- **SAE Level 4:** automatización de la conducción elevada. El vehículo puede ser conducido de forma autónoma bajo condiciones limitadas, pero suficientes como

para que el conductor no tenga que conducir en ningún momento. Un ejemplo de este tipo serían las redes de taxis autónomos locales.

- **SAE Level 5:** automatización de la conducción completa. El vehículo puede ser conducido de forma autónoma bajo cualquier circunstancia y en cualquier lugar.

Dado que el entorno de conducción del proyecto no será una carretera real, sino un pequeño circuito artificial, es complicado definir qué nivel SAE es equivalente al objetivo del proyecto. Por un lado, se busca que el coche no requiera intervención humana en ningún momento de la conducción. Esto podría dar lugar a pensar que se trata de una autonomía SAE Level 3 o incluso SAE Level 4. No obstante, el circuito no contendrá obstáculos ni señales de tráfico, por lo que también podría decirse que se trata de una conducción asistida SAE Level 2.

Esta contradicción muestra que la clasificación anterior no tiene gran utilidad en casos como el de este proyecto. No obstante, su utilidad reside en que permite evaluar las soluciones de las distintas empresas para así identificar las mejores y tomar ejemplo de ellas.

2.2.2. Propuestas de solución

A inicios de 2022, aún no existe ninguna solución capaz de conducir de forma autónoma en todos los escenarios, cumpliendo el SAE Level 5, en una sola ciudad. Dada la gran complejidad del mundo real, los escenarios que puede encontrarse un vehículo son potencialmente infinitos y muy diversos entre sí. Por ejemplo, a pesar de que Volvo había diseñado un sistema específico solo para detectar animales grandes, el movimiento de los canguros en Australia confundía a su red neuronal (41).

Para intentar afrontar este problema, distintas empresas están probando soluciones que suelen partir de una base común, pero con enfoques diferentes. La columna vertebral de estas soluciones suele estar formada por:

- **Sensores** para obtener información del entorno.
- **Redes neuronales profundas** para comprenderlo y planificar la conducción.
- Un *hardware* para ejecutar la inferencia de las redes y transmitir las instrucciones al coche.

Sin embargo, a partir de ahí existen dos enfoques claramente diferenciados, cuyos defensores argumentan habitualmente que el suyo es el único que puede funcionar.

El primero es el de empresas como **Waymo** y **Cruise**⁵, que apuestan por utilizar sensores de alta precisión como el **LiDAR** para crear un mapa en alta definición y en tres dimensiones del entorno, y facilitar la localización exacta del vehículo y por tanto la tarea de conducción (10) (42).

Una zona habitual de pruebas es el estado de California, que publica anualmente un informe con las millas conducidas y las intervenciones de cada empresa a la que se le otorga la licencia para operar. En base al informe de 2021, las empresas más avanzadas

⁵ <https://getcruise.com>



fueron Waymo, AutoX⁶, Pony.ai⁷ y Cruise, siendo AutoX la de menor ratio de intervenciones, una cada 40.000 millas (43). Aunque hasta ahora este enfoque ha sido el único capaz de operar vehículos sin conductor y cuenta con una frecuencia de intervención humana muy baja, presenta una serie de problemas.

En primer lugar, el coste de las flotas es muy elevado. Los vehículos están equipados con numerosos sensores, entre ellos los LiDAR, cuyo precio actualmente oscila entre los 500 y 1.000 dólares por unidad (44). Además, cuentan con un potente ordenador en la parte trasera del coche que hasta hace unos años ocupaba el maletero completo (45). Según los detalles ofrecidos por el anterior director ejecutivo de Waymo John Krafcik, podemos estimar el coste de todos los añadidos de Waymo a los coches en 2021 en unos \$100.000 (8).

En segundo lugar, esta solución es totalmente dependiente de la localización del vehículo dentro de un mapa prediseñado en alta definición. Por tanto, la escalabilidad no es sencilla, ya que requiere mapear cada nueva zona dónde se quiera que operen los vehículos. Esto, por tanto, les impide salir de la zona mapeada sin perder su autonomía.

Partiendo de la base de que los humanos somos capaces de conducir usando nuestros ojos como únicos sensores, ha surgido otra propuesta para resolver el problema de la conducción autónoma. Esta prescinde del LiDAR, del radar y de los mapas en alta definición, apoyando casi todo el peso de la autonomía sobre el *software* de **visión por ordenador**. Emplea técnicas de aprendizaje profundo para estimar la posición y velocidad de los objetos del entorno, pero aun así requiere una capacidad de cómputo mucho menor.

Sin embargo, no está exenta de desventajas. Es más vulnerable ante malas condiciones climáticas, requiere de una cantidad de datos mayor para entrenar las redes neuronales y presenta menor redundancia ante fallos. Hasta ahora, ninguna empresa utilizando una solución de este tipo ha llegado a ofrecer un nivel 4 o 5 de autonomía y su distancia conducida entre intervenciones es mucho menor (46). En base a la adopción de sus productos, las dos compañías más avanzadas son Tesla y Comma.

Tesla: ofrece su solución como un paquete opcional en la compra de sus coches llamado Full Self Driving, por un precio que ha ido aumentando en los últimos años y actualmente es de \$15.000. Este paquete incluye diversas funciones avanzadas de conducción asistida SAE Level 2, entre ellas City Streets, aún en fase beta, que es capaz de navegar por escenarios urbanos y detenerse ante semáforos y señales de tráfico. Sin embargo, y aunque Tesla no publica los datos de las intervenciones, según los datos recopilados por 75 usuarios, en marzo de 2022 Full Self Driving presentaba en zonas urbanas una intervención cada 9,1 millas (46).

Comma.ai⁸: vende un dispositivo equipado con una cámara y procesadores, similar a un teléfono móvil, que puede conectarse a ciertos vehículos compatibles a través de un bus del coche tras el espejo retrovisor. Una vez conectado, el dispositivo ejecuta su *software* de código abierto OpenPilot para dotar al coche de un sistema de conducción

⁶ <https://www.autox.ai>

⁷ <https://www.pony.ai>

⁸ <https://comma.ai>

asistida SAE Level 2. La reacción ante señales de stop y semáforos está en fase de pruebas.

Por último, otra propuesta a destacar sería la de **Mobileye**⁹, que combina ambas soluciones. Por un lado, su director ejecutivo Amnon Shashua cree que es fundamental construir una solución robusta utilizando solo visión artificial mediante cámaras. No obstante, también cree que después hay que complementarlo con la redundancia que aportan los demás sensores (47).

Conocidos varios de los actores más importantes que lideran la carrera del coche autónomo y los enfoques que han adoptado, es evidente cuáles deberían ser tomados como referencia para el proyecto. Si bien es cierto que existen sensores LiDAR en miniatura por debajo de \$100 (48), éstos tienen una precisión reducida y el coste sigue siendo relevante. Además, dado que la autonomía que se busca es el seguimiento de las líneas de calzada, una cámara debería ser suficiente, requiriendo menor poder de computación y permitiendo desarrollar una solución de visión artificial más robusta.

Esto implica que, el enfoque de Tesla y, en especial el de Comma, resultan de mayor interés, por lo que a continuación se detallará más a fondo la implementación de sus soluciones. Para ello, y recuperando y formalizando la columna vertebral del problema mencionada en el subepígrafe anterior, la tarea de conducción autónoma suele dividirse en las siguientes etapas:

- **Percepción:** obtener la información necesaria del entorno mediante sensores.
- **Planificación:** decidir la trayectoria y velocidad del coche a partir de la información disponible.
- **Control:** transformar estas instrucciones en maniobras del vehículo.
- **Supervisión:** vigilancia del estado de alerta del conductor y protección ante fallos en el sistema autónomo. La primera parte no es aplicable a este proyecto.

Cada uno de estos subproblemas presenta sus retos y complicaciones con distintas formas de solventarlos, existiendo varias alternativas y enfoques relevantes.

2.2.3. Percepción

En una situación de conducción real, el objetivo es conocer con la mayor precisión posible la posición relativa del vehículo respecto a los objetos que le rodean, y obtener toda la información del entorno que pueda ser relevante para la conducción.

Así pues, los sensores que utilizan Tesla y Comma son:

1. Cámaras

Dado que los humanos conducimos principalmente usando la vista, las carreteras están preparadas para ser interpretadas por ella. Para que el coche distinga entre las distintas señales de tráfico o las distintas líneas de calzada, es necesario que tenga información sobre el color de los objetos del entorno en el espectro de luz visible.

⁹ <https://www.mobileye.com/>



Para ello se utilizan cámaras con distintas resoluciones y aperturas, que se colocan en varios puntos del vehículo. Las cámaras presentan algunas desventajas, como la dificultad de medir con exactitud la posición y velocidades relativas de los objetos, y la vulnerabilidad a las condiciones climáticas de baja visibilidad.

Los vehículos de Tesla están equipados con ocho cámaras de 1,2 MP, y los de Waymo con 19 y 29 cámaras según el modelo. En el caso de Comma, sus primeros dispositivos contaban con una sola cámara de 16 MP, y la última generación ha añadido una segunda cámara con un mayor ángulo de visión. Además, utilizan otra cámara de apoyo apuntando hacia el conductor, para ayudarse en los cambios de carril y en la vigilancia.

La baja resolución de las cámaras de Tesla y el uso de una única cámara por parte de Comma muestran la importancia del *software* y la capacidad de los modelos de extraer mucha información con pocos datos.

2. GPS

Gracias a varios satélites terrestres que comunican su distancia relativa al coche, mide la posición del vehículo sobre la Tierra con una desviación en la escala de metros. Éstos tienen un bajo coste y universalidad en la medición, pero su precisión es reducida. Además, son vulnerables a interferencias, como las provocadas por túneles, parkings o incluso la humedad.

Tal es así que en Comma han elaborado una biblioteca de código abierto para obtener resultados más precisos del GPS, incorporando estos factores que causan interferencias en la señal.

Dada la pequeña escala del proyecto, el GPS no será necesario.

3. IMU

Las unidades de medición inercial (IMU) son elementos de muy bajo coste que disponen de un velocímetro y un giroscopio que miden la velocidad y orientación del coche.

Una vez obtenida la información de los sensores, y antes de realizar la planificación, las imágenes de las distintas cámaras suelen combinarse en un espacio vectorial, en caso de tener varias.

Esto no es un proceso trivial. Según Andrej Karpathy, director de inteligencia artificial en Tesla, ellos utilizan primero redes neuronales residuales, en concreto RegNets, para obtener varios vectores de las imágenes con distintos tamaños y precisiones (49). En segundo lugar, emplean *transformers* para convertirlo en un espacio vectorial y, por último, una arquitectura algo más compleja con una red neuronal recurrente para combinar las imágenes entre sí a través del tiempo añadiendo la información de velocidad y aceleración de la IMU.

2.2.4. Planificación

Tras obtener la representación deseada del entorno, el siguiente paso es conseguir que el coche interprete esa información para tomar decisiones de conducción adecuadas.

Esta etapa plantea varias dificultades:

- En primer lugar, el objetivo de la conducción no es único sino triple. La finalidad última es llegar al destino, pero se quiere hacerlo maximizando la seguridad, la eficiencia (tiempo y consumo) y el confort de quién esté en el vehículo.
- En segundo lugar, el mundo es tan diverso que resulta muy complicado conseguir que el coche sea capaz de actuar ante cualquier escenario. La estrategia más seguida es utilizar redes neuronales convolucionales para clasificar los objetos del entorno y, a partir de ahí, emplear distintas estrategias de planificación.

Para esto último, Tesla utiliza algoritmos de búsqueda discreta en lugar de redes neuronales (49). Esto se debe a que el espacio de acciones a tomar no es convexo, y presenta muchos mínimos locales difíciles de optimizar mediante redes neuronales. Sin embargo, sí las utilizan para optimizar la trayectoria a partir del resultado de la búsqueda discreta, para que las maniobras sean lo más suaves posibles.

Estas decisiones se toman en función de la información recibida, pero esta suele ser interpretada primero. Tanto en Tesla como en Waymo y otras compañías, se clasifican los distintos objetos de la escena mediante redes convolucionales y se utilizan modelos de predicción multi-modales para los agentes que aparecen en la misma. Waymo utiliza desde 2019 el modelo MultiPath, que emplea redes neuronales basadas en ResNet para obtener una distribución normal de los movimientos esperados (50).

Dado que no se van a introducir objetos en movimiento en el circuito y no habrá bifurcaciones, estos problemas no aparecerán en el proyecto, pero muestran la enorme dificultad de implementar una solución para la conducción real.

Otra etapa fundamental para ejecutar la planificación es el entrenamiento de las redes neuronales. Una estrategia comúnmente utilizada es obtener datos durante la conducción para usarlos en el entrenamiento. Además de obtener la información del entorno, las intervenciones humanas se pueden usar como señal de error que propagar a la red. Sin embargo, la mayoría no se conforman con esto y utilizan **entornos de simulación** basados en distintos motores gráficos como Unreal Engine. En el caso de Waymo, han creado su llamada Simulation City mediante sus propias redes neuronales, incorporando en ella agentes entrenados con aprendizaje reforzado (51). Dado que ni el coche ni los circuitos empleados en el proyecto son similares a los reales, ninguno de estos métodos de obtención de datos es aplicable al proyecto, por lo que hará falta buscar una alternativa.

En el problema de planificación, Comma es la única de las empresas mencionadas anteriormente que no realiza una clasificación de los objetos. Su propuesta se basa en un sistema con una arquitectura *end to end*, justificada en la creencia de que no existe una capa comprensible por los humanos entre la percepción y la planificación (52). Entrenan su modelo con la trayectoria por la que ha conducido un humano como variable objetivo, sin hacer explícita la interpretación del entorno. En cuanto a su arquitectura, ésta es mucho más simple que la de su competencia, empleando una efficientNet b2 (52).

Por último, para el proceso de inferencia, Tesla emplea un chip de desarrollo propio que alcanza los 144 TOPS (53). Este cuenta con dos procesadores independientes para garantizar redundancia en caso de fallo. Respecto a Comma, en la versión del *hardware* que estaba a la venta hasta 2021, el Comma Two (figura 8), empleaban el teléfono móvil



LeEco LePro 3, con su procesador correspondiente. Este es el Snapdragon 821, que cuenta con la GPU Qualcomm Adreno 530 capaz de conseguir 500 GFLOPS.



*Figura 8. Comma Two instalado en un coche
Fuente: Comma.ai*

2.2.5. Control y supervisión

A partir de la planificación se obtienen, entre otros, resultados del ángulo de giro del volante, frenado y aceleración deseados. La transmisión de estas señales a las unidades de control electrónico (ECU) presenta especial dificultad cuando el fabricante del vehículo no es el mismo que el del sistema de conducción autónoma, como es el caso de Comma. Para que su sistema OpenPilot sea compatible con más de 150 vehículos, han aprovechado que los buses de casi todos los coches siguen el mismo estándar: Controller Area Network (CAN).

Una vez el dispositivo se conecta al bus CAN, se deben interceptar las señales que envía el coche para cambiarlas por las decisiones tomadas por el sistema autónomo. Esto no es trivial, dado que cada fabricante y en algunos casos cada vehículo utiliza mensajes distintos para la dirección y aceleración. Cuando se han identificado los mensajes, se cambian por los generados por OpenPilot (54). Estos deben adaptarse a las particularidades del vehículo que, en algunos casos, han de incluir la variabilidad de la respuesta en función de factores como el grosor y el desgaste de los neumáticos. Por último, debe asegurarse que el proceso es seguro y que, ni el sistema puede bloquear la conducción manual en caso de fallo, ni un agente externo puede interceptar las instrucciones y enviar otras distintas.

En cuanto a la supervisión, el objetivo de los servicios de conducción asistida, como Comma y Tesla, es asegurarse de que el conductor está listo para tomar el control del coche en todo momento. Para ello, Comma emplea una cámara que mira hacia el interior del coche para asegurarse de que el conductor está prestando atención en la carretera (55). Tesla, en cambio, obliga al conductor a ejercer una ligera presión sobre el volante cada 30 segundos (56). Por otro lado, las compañías que han desplegado taxis autónomos suelen tener operarios que vigilan los coches desde una central y les proporcionan ayuda cuando es necesario (57).

Para este proyecto, la labor será más sencilla dado que no hay vidas en juego y solo existirá un tipo de vehículo.

3. Diseño de la solución

3.1. Análisis del problema

A modo de recordatorio, el proyecto planteado consiste en la construcción de un coche robótico de pequeño tamaño capaz de moverse de forma autónoma. Como se ha explicado en los objetivos del trabajo, se busca conseguir que el robot pueda mantenerse dentro de su carril dentro de un entorno controlado, incluso en circuitos que no haya visto nunca.

Para ello, y dadas las limitaciones de presupuesto y tamaño, el proyecto presenta **tres retos principales** equivalentes a los subobjetivos del proyecto:

- La construcción de un robot móvil programable y con sensores para percibir el entorno.
- La implementación de un *software*, incluido un modelo entrenado basado en redes neuronales, que permita la autonomía.
- La ejecución de este programa en tiempo real y de forma local desde el robot, con la respectiva inferencia de la red neuronal.

Respecto al primer reto, el tamaño del robot y el presupuesto hacen imposible el uso de un gran número y variedad de sensores, como Waymo, Cruise o incluso Tesla. Sin embargo, el problema se asemeja algo más al de Comma, dado que han conseguido una solución de SAE Level 2 funcional utilizando una sola cámara, apoyada por el GPS y la UMI. Como se ha mencionado en el subepígrafe 2.2.3, el GPS no será necesario en este caso, pero sí los otros dos sensores, siendo la cámara el principal.

En cuanto al *software*, para la etapa de planificación se debe entrenar una red neuronal que estará limitada por una serie de factores, como la potencia de cómputo y la capacidad de recolección de datos. Además, sin un programa con el que controlar el coche, el modelo no servirá de nada.

En tercer lugar, para realizar la inferencia en tiempo real, Comma puede servir de referencia. Ellos son capaces de ejecutar su modelo con una capacidad de cómputo de 500 GFLOPS, inferior a la de los móviles actuales de gama media. Esto aporta una referencia para verificar la **viabilidad teórica** del proyecto y para conocer qué *hardware* será necesario.

Por tanto, en este problema en concreto, las etapas de percepción, planificación y control del coche se traducen en las siguientes tareas:

1. Captura de fotogramas de vídeo desde la cámara
2. Inferencia de la red neuronal a partir de esos fotogramas
3. Cambio de la dirección del coche a partir de la salida de la red neuronal

A partir de estas tareas, es posible concretar de qué tipo de problema de aprendizaje automático se trata para tener una idea general de la arquitectura de la futura red neuronal.



Esto es útil para elegir las herramientas de *software* utilizadas en el proyecto y para recopilar los datos adecuados.

El programa de conducción autónoma recibirá como entrada fotogramas de vídeo y deberá devolver la dirección del coche, esto es, el ángulo de giro de las ruedas delanteras. Esto es similar, de nuevo, a la solución implementada por Comma comentada en el subepígrafe 2.2.4, que computa el ángulo de giro a raíz de la trayectoria deseada.

En primer lugar, dado que la salida del modelo debe ser numérica, un ángulo, que presenta una distribución continua, se observa que se trata de un problema de **regresión**. Por otra parte, como la entrada son imágenes, la arquitectura escogida deberá ser óptima para el análisis de imágenes, como la de los modelos de **clasificación**.

Otra cuestión relevante es si es necesario añadir recurrencia al modelo. En el caso de un vehículo real, para saber a dónde dirigirse en un instante dado es necesario tener información de instantes previos para conducir. Por ejemplo, recordar una señal de tráfico o saber si un peatón está tras un objeto opaco. Sin embargo, al tratarse de un entorno sin obstáculos y sin señales, el robot puede saber a qué ángulo girar sus ruedas solo viendo lo que tiene delante en ese instante. Por tanto, el único dato de entrada necesario para la red es el fotograma del instante correspondiente.

Respecto al entrenamiento de la red, se trata de un problema de **aprendizaje supervisado**. Dentro de este tipo de métodos, se podría emplear un aprendizaje reforzado. Esto implicaría que se comenzaría a conducir el coche de forma autónoma desde cero y el coche iría aprendiendo de sus errores hasta no cometerlos. No obstante, esto implicaría tener que estar manualmente arrancando y parando el coche durante un gran número de veces, por lo que se ha considerado más sencillo entrenar el coche con datos de conducción correctamente etiquetados. Así pues, el conjunto de datos necesario para entrenar la red será un conjunto de fotogramas, etiquetados con el ángulo al que debería dirigirse el coche ante esa situación.

3.2. Planificación de necesidades

Una vez concretado el problema, se procede listando todo el material necesario para llevar a cabo la solución, en primer lugar, el *hardware*.

3.2.1. *Hardware*

Para construir un coche robótico serán necesarios:

- Un vehículo equipado con chasis, ruedas y un sensor de velocidad
- Un motor con sistema de transmisión para mover el vehículo
- Una fuente de alimentación portátil

Y para hacer que el coche pueda conducir de forma autónoma serán necesarios:

- Una cámara

- Uno o varios procesadores para ejecutar las instrucciones de conducción y la inferencia de la red neuronal
- Una memoria para almacenar las imágenes y el código
- Circuitos, buses y puertos para conectar los distintos elementos
- Un ordenador para entrenar la red neuronal e interactuar con el robot de forma remota
- Un monitor para configurar la conexión remota
- Un circuito con un carril distinguible mediante visión

Así pues, para integrar todos estos materiales se propone el sistema representado en la figura 9.

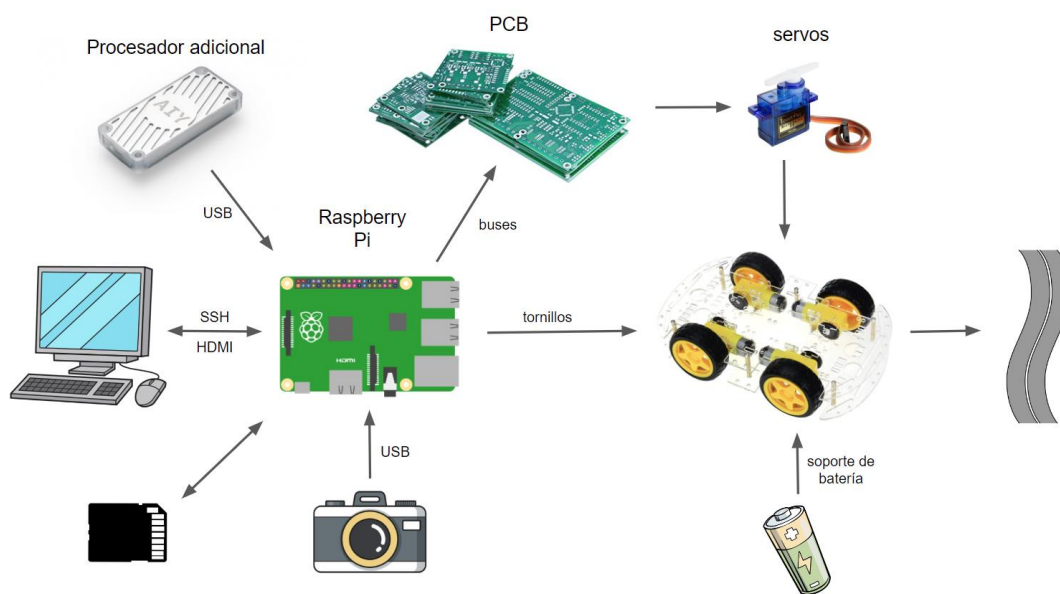


Figura 9. Diagrama de los materiales necesarios para el proyecto y sus conexiones
Fuente: elaboración propia

La solución se basa en dos elementos fundamentales: un chasis con ruedas y motores y un ordenador en miniatura que será la Raspberry Pi. La Raspberry cuenta con varios elementos que la hacen ideal para el proyecto: agujeros para atornillarla al chasis, conexiones para varios periféricos y conexión a internet para controlarla de forma remota mediante conexión por SSH.

Sin embargo, la GPU integrada en su CPU, aún en los modelos más recientes, no alcanza más de 4 GFLOPS (58), por lo que se empleará otro procesador especializado para realizar la inferencia. Tanto este como la cámara serán conectados a la Raspberry mediante sus puertos USB, mientras que el monitor se conectará por el puerto HDMI. La Raspberry cuenta también con un puerto microSD, que se aprovechará para conectar una tarjeta empleada como unidad de almacenamiento.

Por otro lado, para conectar la Raspberry Pi con el coche, se hará primero mediante tornillos de sujeción y, en segundo lugar, con diversas placas con circuitos (PCB). Estas permitirán transmitir las instrucciones de aceleración y giro al motor a través de servos, empleados junto al chasis como sistema de transmisión. Todo esto será alimentado mediante unas baterías de litio, para lo cual hará falta un soporte que las sujete al coche y un cargador.

3.2.2. Software

Las principales herramientas de *software* a utilizar serán las siguientes:

- Un sistema operativo para la Raspberry
- Un lenguaje de programación y bibliotecas para programar la red neuronal y el resto del código necesario
- Un entorno de desarrollo
- Una interfaz de programación de aplicaciones (API) para controlar el coche

Respecto al código que se deberá desarrollar, será el siguiente:

- Un modelo de regresión, que devuelva el ángulo de giro de las ruedas delanteras a partir de un fotograma de la cámara, y cuya inferencia pueda ser ejecutada en el coche en tiempo real
- Un programa para controlar el coche que permita recopilar el conjunto de datos etiquetado
- Un programa que permita transmitir la salida del modelo a la dirección del coche

Inicialmente, se puede plantear la arquitectura necesaria para la red. De los mencionados en el subepígrafe 2.1.2, dado que los *Transformers* requieren una potencia de cómputo muy por encima de las limitaciones del proyecto, se propone el uso de **redes convolucionales**.

Así pues, una vez listados los elementos necesarios, tanto de *hardware* como de *software*, en los siguientes epígrafes se describen y justifican las elecciones realizadas para cada uno de ellos.

3.3. Hardware empleado

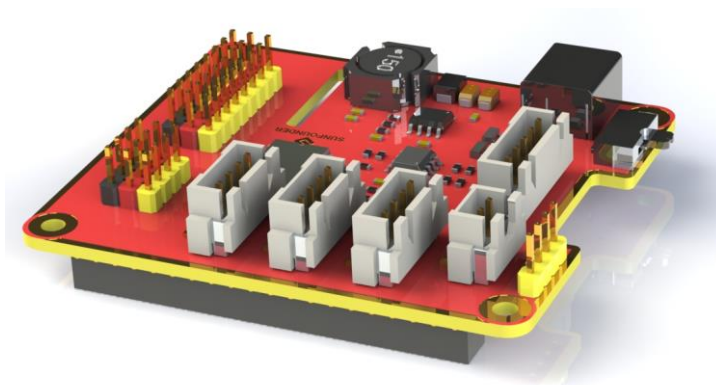
3.3.1. SunFounder PiCar-V Kit V2.0



*Figura 10. SunFounder PiCar-V
Fuente: SunFounder*

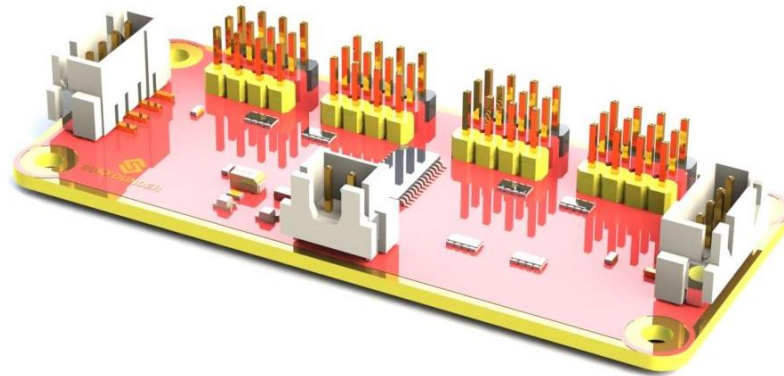
La empresa SunFounder vende un kit (figura 10) por un precio de entorno a 100€ que incluye los materiales para construir un pequeño coche funcional, preparado para utilizarse junto a una Raspberry Pi. Por tanto, se ha decidido utilizarlo dada su idoneidad. Los materiales principales que incluye el kit son los siguientes:

- **Chasis y ruedas** (figura 10)
- **PCB 1** (figura 11): la primera placa es el Robot HATS, diseñada especialmente para ser compatible con varios modelos de Raspberry Pi. Tiene un puerto para conectarse al soporte de las baterías y recibir alimentación, con un voltaje de 8,4-7,4V. Después, cuenta con 40 pines GPIO para conectarse a la Raspberry Pi, a través de los cuales le transmite la corriente, y otros puertos para transmitirla a las demás placas del coche. Tiene un interruptor para controlar el encendido y apagado.



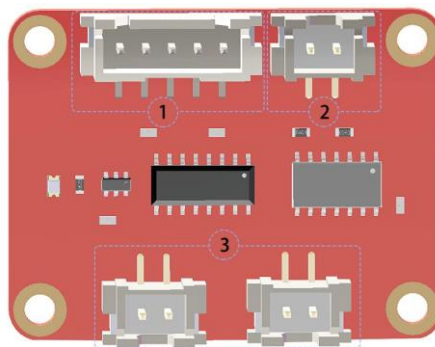
*Figura 11. PCB N°1 del SunFounder PiCar
Fuente: SunFounder*

- **PCB 2** (figura 12): placa modelo PCA9865 que recibe la corriente desde el HATS y alimenta a la placa del motor, así como a los servos.



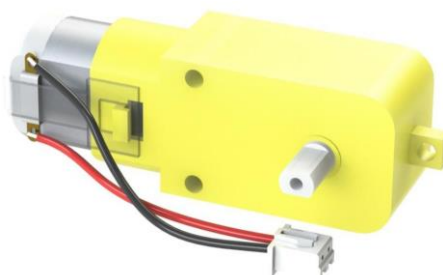
*Figura 12. PCB N°2 del SunFounder PiCar
Fuente: SunFounder*

- **PCB 3** (figura 13): módulo controlador del motor, que alimenta y ajusta la velocidad de los motores.



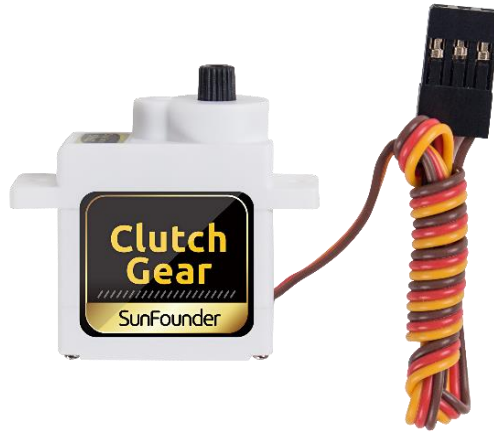
*Figura 13. PCB N°3 del SunFounder PiCar
Fuente: SunFounder*

- **Motor** (figura 14): motor de corriente continua DC Gear Motor con un voltaje de entrada de 4.5-6V.



*Figura 14. Motor del SunFounder PiCar
Fuente: SunFounder*

- **Servos** (figura 15): un servo para mover las ruedas delanteras y dos para la cámara. El modelo es SunFounder SF006C y tienen un voltaje de entrada de 4.8V y un par máximo de 1,4 Kgf · cm.



*Figura 15. Servos del SunFounder PiCar
Fuente: SunFounder*

- **Cámara** (figura 16): cámara web con conexión USB, un ángulo de visión de 120° y una resolución máxima de 480x360 píxeles.



*Figura 16. Cámara del SunFounder PiCar
Fuente: SunFounder*

3.3.2. Baterías ecolle 18650

Para no tener el vehículo permanentemente conectado, se ha optado por dotarlo de baterías de iones de litio. Las empleadas, que pueden verse en la figura 17, son dos pilas de la marca ecolle, con 65 mm de longitud y 18 mm de diámetro que transmiten una corriente de 3,7V cada una y 2500mAh.



*Figura 17. Baterías ecolle 18650
Fuente: ecolle*

Las pilas deben ser de estas dimensiones para adaptarse al soporte de baterías del kit de SunFounder. Este formato de batería ha sido utilizado en la industria del automóvil para la fabricación de numerosos coches eléctricos, como los Tesla Model S (59).

3.3.3. Raspberry Pi 3 Modelo B+

La Raspberry Pi es un ordenador de una sola placa que nos permite tener la potencia de cómputo y la conectividad necesaria en un tamaño muy reducido, suficiente para poder montarla sobre el chasis del coche. El SunFounder PiCar-V es compatible con varios modelos, de los cuáles se ha elegido el modelo 3 B+ por potencia y disponibilidad. El precio de este modelo es de unos 60€ en las páginas de venta recomendadas por el fabricante. Este modelo, mostrado en la figura 18, cuenta con una CPU Broadcom BCM2837B0, ARMv8 de 64 bits, con cuatro núcleos de 1,4GHz. Tiene 1GB de memoria SDRAM LPDDR2, cuatro puertos USB 2.0, una ranura para tarjetas microSD, un puerto HDMI, un puerto ethernet y conexión a través de WLAN.



Figura 18. Raspberry Pi Modelo 3 B+

Fuente: Raspberry

3.3.4. Acelerador Coral Edge TPU

Para realizar la inferencia de la red neuronal se hará uso de un procesador específico para inferencia, una Coral Edge TPU de Google, similar a la de la figura 19. Ésta es capaz de alcanzar 4 TOPS, superior al Snapdragon 821 empleado por Comma. Su consumo energético es de 0,5 vatios/TOP y el precio en la web del fabricante es de 49€.

El procesador recibe el nombre de unidad de procesamiento tensorial (TPU) porque realiza las operaciones usando tensores como datos de entrada, lo cual ayuda a paralelizarlas y conseguir una mayor velocidad. Sin embargo, esto implica que solo puede ejecutar modelos compilados con un formato específico, para el cual se necesitará la API

de Coral. Por último, cuenta con un cable USB que permite conectarla directamente a la Raspberry.



*Figura 19. Acelerador Coral Edge TPU
Fuente: Coral*

3.3.5. Micro SD Samsung EVO Plus 128 GB

Como unidad de almacenamiento se ha escogido una tarjeta microSD, en concreto una de 128 GB para evitar problemas de espacio. La adquirida es de marca Samsung y tiene una velocidad de lectura de 100 Mb/s y de escritura de 90 Mb/s.

3.3.6. Accesorios

Además del material montado sobre el coche, se emplearán varios accesorios para desarrollar la solución. Estos son:

- **Cargador de baterías:** cargador NIMO CAR400 con un voltaje de salida de 4,2 V e intensidad de corriente de 1 A.
- **Cinta adhesiva de color rosa:** utilizada para dibujar las líneas que simulen la calzada. Se ha elegido el rosa por ser un color que contrasta con la mayoría de los suelos.
- **Monitor, ratón y teclado:** son necesarios para configurar la conexión SSH.
- **Ordenador:** empleado para labores accesorias como la descarga del sistema operativo de la Raspberry. El ordenador disponible para el proyecto tiene un procesador AMD Ryzen 7 1700, con 8 núcleos de 3,7GHz. Cuenta con 16GB de memoria RAM DDR4 y una tarjeta gráfica Nvidia GeForce GTX 1050Ti. El sistema operativo instalado es Windows 10.

Dado que algunos de los materiales son proporcionados por la Universidad Politécnica de Valencia, no se conoce el desembolso económico total para adquirir todos los elementos de *hardware* mencionados. No obstante, buscando los precios en las páginas de venta de cada marca, el coste total se estima en alrededor de 300€.

3.4. *Software* empleado

3.4.1. Sistema operativo: Raspberry Pi OS

El sistema operativo que se instalará en la Raspberry es el proporcionado por el fabricante, anteriormente conocido como Raspbian y ahora como Raspberry Pi OS. Está basado en Debian, una distribución de Linux desarrollada en código abierto sobre la que se han construido muchas otras distribuciones, como por ejemplo Ubuntu.

Raspberry Pi OS ha sido optimizado para esta placa y está preparado para facilitar su instalación. Tiene una versión de 32 bits y una de 64. A pesar de que el procesador está optimizado para los 64 bits, se ha instalado la versión de 32 por razones que se explicarán más adelante.

3.4.2. Python

Python es el lenguaje utilizado por defecto dentro del mundo del aprendizaje automático. Esto se debe a varias razones, entre las que destacan su sintaxis simple, la facilidad de integración con otros lenguajes, y la gran cantidad de bibliotecas de las que dispone enfocadas a este campo.

El SunFounder PiCar cuenta con una API para Python de código abierto mediante la cual controlar el coche, por lo que la elección de este lenguaje es trivial. En concreto, se utilizará la versión 3 de Python, la más reciente y en la cual está escrita la API del cliente.

3.4.3. Bibliotecas: OpenCV, Tensorflow y Keras

En el desarrollo del código se emplearán algunas bibliotecas comúnmente utilizadas en Python, como Numpy o Matplotlib. Además, serán necesarias otras bibliotecas orientadas al aprendizaje automático, como OpenCV, Tensorflow y Keras.

OpenCV, desarrollada por Intel, es una biblioteca para visión artificial disponible para varios lenguajes de programación, entre ellos Python. Entre otras cosas, permite capturar vídeo e imágenes a partir de la cámara y modificar imágenes.

Tensorflow, desarrollada por Google, es una biblioteca con diversos métodos para construir y entrenar redes neuronales a partir de sus elementos básicos.

Keras, cuyo autor principal es también un ingeniero de Google, es otra biblioteca enfocada al aprendizaje profundo, similar a Tensorflow pero de más alto nivel. Cuenta con las funciones de activación, de pérdida, y las capas más utilizadas en el campo del aprendizaje profundo. En las últimas versiones de Tensorflow está integrada dentro de su API.

3.4.4. *Software* para la Edge TPU

Mientras que las bibliotecas mencionadas en el subepígrafe anterior están enfocadas al desarrollo del código, la biblioteca PyCoral ha sido desarrollada para optimizar la inferencia de los modelos en un dispositivo como la Coral Edge TPU. La TPU solo

ejecuta modelos compilados para Tensorflow Lite, por lo que esta biblioteca es imprescindible.

Además, hay algunos paquetes adicionales de *software* en la web del fabricante que también se deben descargar.

3.4.5. Interfaces de programación para el coche

Los fabricantes del kit de SunFounder han publicado en Github tres repositorios escritos en Python que permiten el control remoto del coche. El primero es el módulo básico de control que permite interactuar con los servos, la cámara y el motor. El segundo contiene una API para el servidor, pensada para ejecutarse en la Raspberry Pi, escrita en Python 2. Además, contiene la API del cliente, escrita en Python 3, pensada para ejecutarse desde otro dispositivo permitiendo así el control remoto. Las direcciones donde se alojan son https://github.com/sunfounder/SunFounder_PiCar y https://github.com/sunfounder/SunFounder_PiCar-V, respectivamente.

El tercer repositorio contiene una biblioteca de bajo nivel para controlar los servos y el motor y está alojado en https://github.com/sunfounder/SunFounder_PCA9685.

3.4.6. Entornos de desarrollo

Para desarrollar los programas necesarios se ha decidido utilizar varios entornos de desarrollo. En primer lugar, para escribir diversos programas para la Raspberry Pi, se aprovechará el acceso remoto a los archivos para utilizar Visual Studio Code en el ordenador mencionado en el subepígrafe 3.3.6.

En segundo lugar, para implementar y entrenar el modelo de aprendizaje profundo se utilizará Google Colab y sus cuadernos Jupyter. Este es un entorno de desarrollo en línea para el lenguaje Python, que ya contiene las bibliotecas preinstaladas para aprendizaje automático y permite utilizar de forma gratuita las CPU y GPU de Google para ejecutar los programas, incluyendo el entrenamiento del modelo.



4. Implementación de la solución

4.1. Montaje del vehículo

El montaje del coche robótico comenzó a partir de los elementos presentes en el kit del SunFounder PiCar-V. Además de los elementos mencionados en el subepígrafe 3.2.1, el kit contiene distintas piezas de plástico que forman el chasis, tornillos de varios tamaños y algunas herramientas para facilitar la tarea de montaje.

El kit cuenta con instrucciones de montaje por escrito y en formato digital, las cuales se siguieron para construir el coche. Los primeros pasos consistieron en atornillar diversos elementos como las ruedas traseras, los servos, el soporte de las baterías y los motores a algunas piezas de plástico, para más adelante unir estas piezas con el armazón principal, al cual se habían atornillado previamente los circuitos.

Sin embargo, la Raspberry Pi debe colocarse entre el chasis y el Robot HATS, y una vez colocada no hay espacio para conectar un monitor. Por ello, antes de seguir con el montaje se procedió a la configuración de Raspberry y a la instalación del sistema operativo, siguiendo las instrucciones de la web del fabricante.

La empresa cuenta con un programa que instala el sistema operativo escogido en una unidad de memoria. Por tanto, el primer paso fue descargar este programa, el Raspberry Pi Imager, en su versión de Windows, para posteriormente descargar el Raspberry PI OS en la tarjeta microSD. Este sistema operativo está disponible en varias versiones, siendo la recomendada por el fabricante la versión completa de 32 bits, debido a su compatibilidad con la mayoría de modelos de Raspberry. Sin embargo, tras varias consultas se observó que la versión de 64 bits tenía un mejor rendimiento en el modelo 3 B+.

Por ello, se procedió a instalar esta versión y realizar el proceso de instalación como se describe en los siguientes párrafos. No obstante, se encontró un problema con la compatibilidad de las dependencias de la librería OpenCV y finalmente tuvo que desinstalarse esta versión e instalar la de 32 bits.

Otro detalle que destacar fue la necesidad de formatear la tarjeta para cambiar el formato de archivos de exFAT a FAT32, ya que el gestor de arranque del sistema operativo solo es capaz de leer archivos de este formato o de FAT16.

Habiendo completado este proceso, se insertó la tarjeta en la Raspberry para proceder a la configuración del Raspberry Pi OS, y se conectaron los distintos elementos necesarios. Al puerto HDMI se conectó el monitor, a los puertos USB el teclado y el ratón, y al de alimentación un cargador con conexión micro USB. Este último no estaba incluido en la caja y supuso un accesorio adicional a los planteados en el subepígrafe 3.2.6. Tras encender la Raspberry y seguir las instrucciones que aparecen por pantalla, configurar la conexión Wifi y esperar unos minutos, este proceso concluyó con el reinicio del sistema y la interfaz mostrada en la figura 20 lista para su uso.

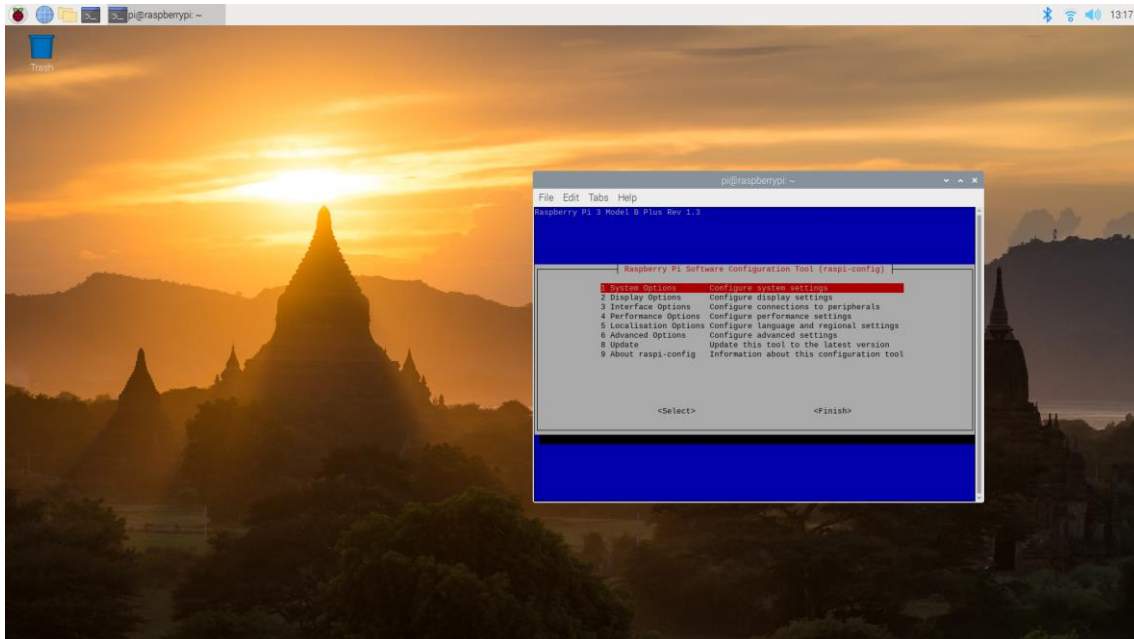


Figura 20. Interfaz del sistema operativo Raspberry Pi OS

Fuente: elaboración propia

A continuación, y tras algunas actualizaciones manuales, se prosiguió configurando el acceso remoto mediante SSH para prescindir del monitor y poder montar la Raspberry en el chasis. Para ello hizo falta obtener la dirección IP de la Raspberry y activar la conexión SSH. Esto último es posible desde la Herramienta de Configuración de *Software* de la Raspberry, dentro del apartado “Opciones de Interfaz”. Una vez activada, se abrió la aplicación de Windows de escritorio remoto e, introduciendo la IP, usuario y contraseña, el acceso remoto estaba listo.

Hecho esto, se retomó el montaje colocando la Raspberry sobre el chasis y atornillando las PCB, poniendo el Robot HATS sobre la Raspberry. Después, se ensamblaron las piezas del chasis con los motores y las ruedas al chasis principal y se conectaron los cables entre las PCB siguiendo los diagramas provistos en las instrucciones. Tras ello, se conectó el servo de las ruedas delanteras y se obtuvo el robot provisional de la figura 21, que ya podía ser encendido.

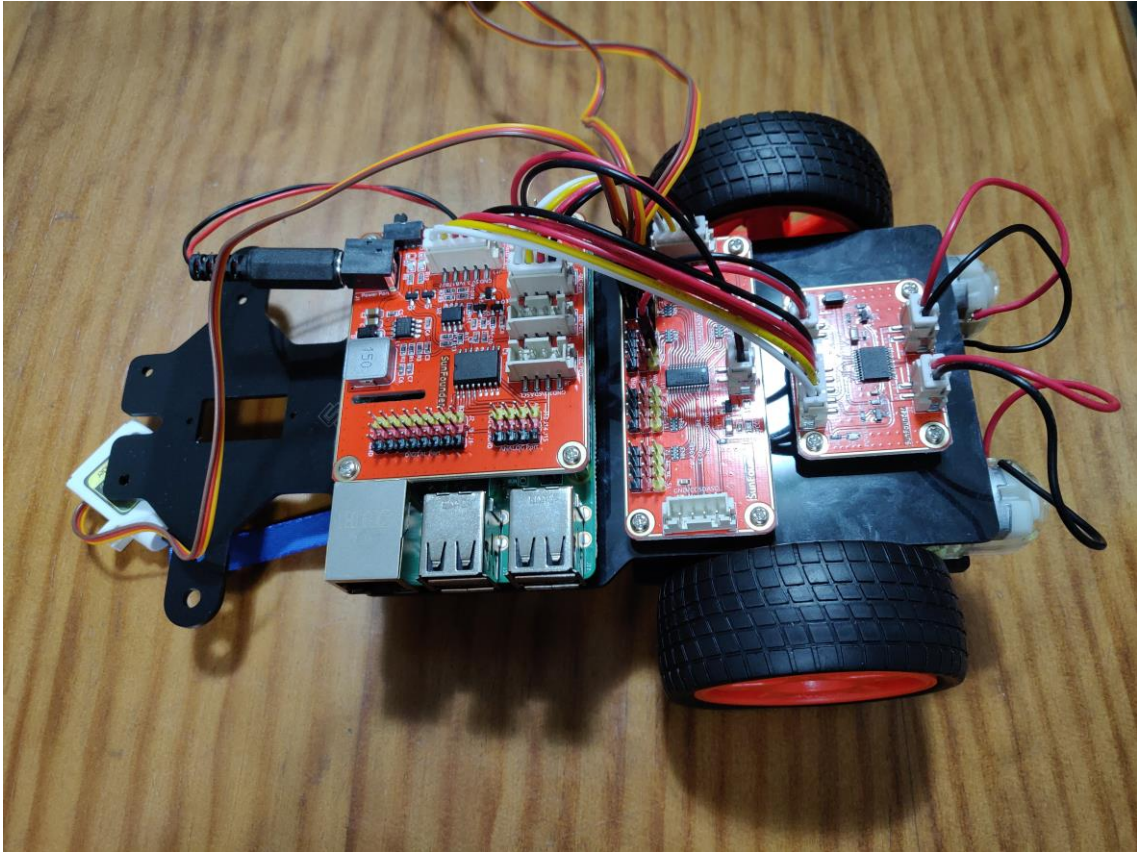
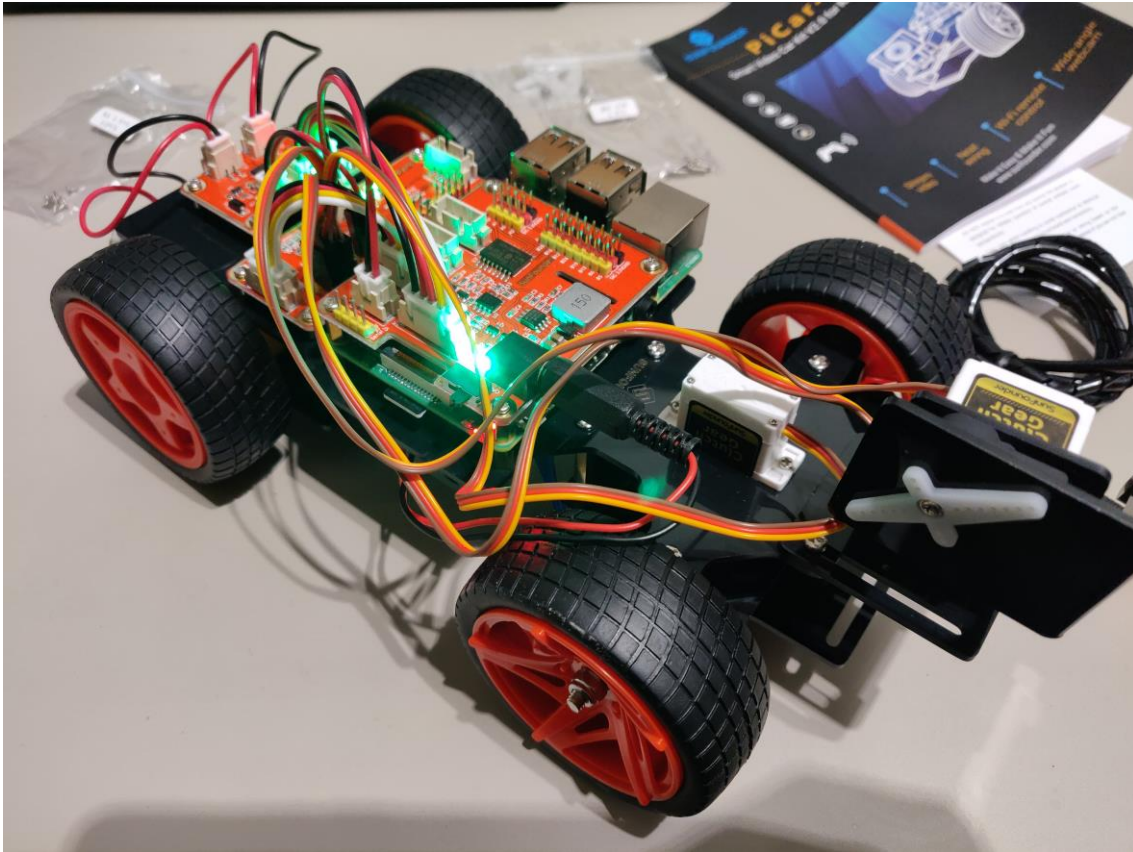


Figura 21. SunFounder PiCar-V durante el montaje

Fuente: elaboración propia

El siguiente paso es configurar mediante *software* la orientación del servo, por lo que se encendió el robot, conectándolo a la corriente mediante el cargador con micro USB, y se clonaron los repositorios ofrecidos por el fabricante. Después, se instalaron sus dependencias asociadas, y se ejecutó el programa de instalación de los servos, cuyo resultado debe ser que estos queden con una orientación de 90° respecto al eje. Sin embargo, bien por haberlo averiado durante el montaje o por defecto de fábrica, el servo quedó ligeramente desviado, algo a tener en cuenta para la futura programación de la dirección del vehículo.

Dejando esto a un lado, se procedió a terminar el montaje, ensamblando las partes restantes del chasis, las ruedas delanteras y, por último, montando la cámara con sus servos correspondientes. Así, se obtuvo el coche de la figura 22, un robot móvil ya funcional pero aún sin control remoto.



*Figura 22. SunFounder PiCar-V tras completar el montaje
Fuente: elaboración propia*

Cabe destacar que las instrucciones estaban escritas para una versión ligeramente distinta del coche, por lo que algunas piezas no eran iguales a las descritas. Esto añadió algo de complejidad adicional, debiendo desmontar parte del coche en dos ocasiones.

4.2. Configuración e instalación del *software*

Con el vehículo ya listo, el siguiente paso fue la instalación de las demás herramientas *software* necesarias.

En primer lugar, se optó por instalar un sistema de acceso remoto a los archivos, para facilitar el visionado y procesamiento de imágenes. Se decidió emplear Samba File Server. Tras instalar el programa en el coche, se configuró el acceso mediante el archivo «*smb.conf*», escribiendo en él la dirección de la carpeta pi y el acceso de lectura y escritura. Hecho esto, ejecutando el comando “net use r: «dirección IP»” desde el equipo con Windows se ganó acceso remoto a los archivos.

A continuación, se continuó descargando un programa para visualizar y grabar las imágenes de la cámara. El programa más utilizado en la Raspberry es Cheese, pero los ajustes por defecto son para una cámara de resolución mayor. Dado que Cheese no tiene un menú de configuración, no se pudo cambiar la resolución y se optó por descargar otros programas. Finalmente, el que mejor rendimiento dio fue Gucview.

En tercer lugar, se continuó con las herramientas necesarias para programar el robot. Algo fundamental es la API del coche, que como se ha indicado en el subepígrafe 3.2.5

está preparada para Python, el cual viene preinstalado en nuestro sistema operativo. No obstante, mientras que el código para el cliente utiliza la versión 3 de Python, el del servidor usa la versión 2. Como en el proyecto se quiere usar la Raspberry Pi como cliente y servidor, sería más cómodo tener ambos códigos en la misma versión de Python. Afortunadamente, más gente se ha encontrado en esta situación y un usuario de Github llamado David Tian ha modificado el código de SunFounder adaptando la API del servidor a Python 3. Para facilitar el desarrollo, se ha clonado su repositorio disponible en <https://github.com/dctian/SunFounder_PiCar> y se ha utilizado en lugar del original del fabricante.

Hecho esto, se procedió a probar el control remoto del coche desde el terminal, usando la propia Raspberry Pi como cliente. En las instrucciones de SunFounder se detalla como comprobar que el coche funciona correctamente, por lo que se siguieron los pasos indicados y se comprobó que el montaje había sido correcto.

Posteriormente, se continuó descargando algunas de las bibliotecas mencionadas en el subepígrafe 3.4.3 con sus respectivas dependencias. Como el modelo no se programará en la Raspberry y se ejecutará mediante Tensorflow Lite, no es necesario instalar todas las bibliotecas en la Raspberry. Así pues, y dado que Tensorflow presenta varios problemas de compatibilidad no resueltos con Ubuntu, no se instaló esta biblioteca en la Raspberry. Por último, la descarga de Tensorflow Lite se dejó para más adelante, de forma previa a la inferencia del modelo.

4.3. Recopilación de los datos

Llegados a este punto, ya se tiene un coche robótico que puede ser controlado de forma remota y que puede ser programado. A continuación, se debe conseguir que sea autónomo y, para ello, es necesario obtener un conjunto de datos con el que entrenar al futuro modelo. Generalmente, cuantos más datos tengamos, mayor será la precisión de este.

En primer lugar, se debe concretar cómo son los datos necesarios. En el epígrafe 3.1 se ha propuesto una solución en la que los datos de entrada serán los fotogramas de la cámara y la salida de la red neuronal el ángulo de giro deseado para cada uno de estos. Dado que se quiere hacer un entrenamiento supervisado, los datos de entrada deberán ser **una imagen de la cámara del coche junto al ángulo de giro correcto**. En cuanto a la cantidad de los datos, se prefirió no establecer una cifra fija y, en su lugar, ir recogiendo más datos en caso de que el entrenamiento de la red lo requiriera. Como inicio para empezar a entrenar, se optó por recopilar unas 1.000 imágenes.

Aclaradas las necesidades, el siguiente paso es decidir el método de recopilación de los datos. Como el circuito dónde se conducirá el coche no son carreteras convencionales, no hay datos válidos disponibles en internet y se deben generar desde cero. En este caso, hay dos formas de generarlos. La primera es escribir un programa para **conducir el coche manualmente**, guardando los fotogramas y su ángulo. La segunda es escribir un programa similar, pero con una **conducción autónoma programada de forma explícita**.

En cualquier caso, se debe escribir un programa que acceda a la API del coche para conducirlo, que se puede aprovechar más adelante para la conducción autónoma. El

código de este programa se encuentra en el archivo `«smart_pi_car.py»`, del repositorio del proyecto. Este es el programa principal de conducción, que crea una clase `SmartPiCar` inicializando la posición de los servos del coche y contiene un método llamado `drive`, mediante el cual se conduce el coche de la forma indicada por el usuario. La conducción se realiza modificando la velocidad del coche y la orientación de los servos empleando las bibliotecas de SunFounder.

David Tian, mencionado anteriormente, ha publicado un código que implementa una conducción autónoma programada de forma explícita, por lo que inicialmente se decidió adaptarlo a este caso y usarlo para recopilar los datos. Este repositorio se encuentra alojado en <https://github.com/dctian/DeepPiCar>. A pesar de que más adelante se acabó abandonando esta solución, se dedicó mucho tiempo en ella y se ha incluido en la memoria. Ambas opciones presentan problemas y, por tanto, ambas pueden ser útiles para un proyecto similar.

Por otra parte, cabe mencionar que, en este problema, las técnicas tradicionales para aumentar los datos, como recortar, girar y ampliar las imágenes, no son fácilmente aplicables. Dado que el ángulo de giro cambia con la orientación de la imagen o al eliminar partes de esta, habría que cambiar el valor de las etiquetas, lo cual supone una complejidad mayor que simplemente recopilar más datos.

4.3.1. Conducción autónoma programada explícitamente

El programa `«hand_coded_lane_follower.py»` de Tian implementa de forma explícita la conducción autónoma gracias a diversas funciones que identifican las líneas de la calzada y computan su ángulo y posición. Durante la conducción, el programa graba un vídeo mediante la cámara del coche que después se pasa al programa `«save_training_data.py»` para convertir los fotogramas en imágenes y guardarlos con su ángulo de giro en el nombre. A continuación, se profundiza en el programa de conducción para comentar los distintos cambios realizados que fueron necesarios para que este funcionara de forma adecuada. El código modificado se encuentra en el repositorio del proyecto con los mismos nombres que el original.

En primer lugar, el programa `«hand_coded_lane_follower.py»` convierte una imagen dada al espacio de color HSV, donde se pueden filtrar fácilmente los colores mediante la tercera coordenada: *Value*. A partir de ahí, se crea una máscara que solo incluya las líneas de la calzada, en color blanco. Estas tres etapas se realizan dentro del método `edges` de la clase `HandCodedLaneFollower`. Sin embargo, hay algunas condiciones en donde no es sencillo crear esta máscara. Combinando un sensor de baja calidad y un suelo reflectante con varios colores, la vista desde el coche era la que se muestra en la figura 23. Prácticamente no hay rastro del color rosa original de las líneas, complicando mucho esta etapa.





Figura 23. *Fotograma de la cámara del coche ante un circuito con suelo reflectante*
Fuente: elaboración propia

Además, con el movimiento del coche, la iluminación y los reflejos cambiaban lo suficiente como para introducir mucho ruido en la máscara. Tras un gran número de intentos fallidos, se acabó desistiendo de conducir en ese suelo y se recreó el circuito en el suelo de la figura 24, sobre el que era más sencillo realizar la conducción.



Figura 24. *Vista cenital del segundo y tercer circuito diseñados*
Fuente: elaboración propia

Tras otro período de experimentación, se consiguieron unos resultados óptimos con los parámetros que se muestran en la figura 25, que también incluye la aplicación de la función *Canny* de OpenCV para obtener los bordes de las líneas. En la figura 26 se muestra un fotograma del circuito antes y después de aplicar la máscara HSV.

```
def edges(image):  
    im_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)  
  
    # filtrar las líneas rosas  
    lower_pink = np.array([150, 50, 120])  
    upper_pink = np.array([180, 255, 255])  
    mask = cv2.inRange(im_hsv, lower_pink, upper_pink)  
  
    # filtrar los bordes de las líneas  
    edges = cv2.Canny(mask, 200, 400)  
    return edges
```

Figura 25. Función para obtener los bordes rosas de una imagen
Fuente: «hand_coded_lane_follower.py»

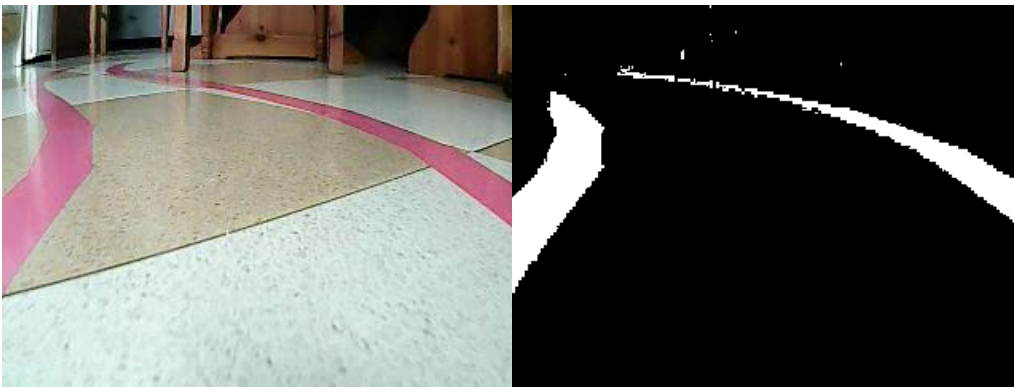


Figura 26. Fotograma del segundo circuito sin y con la máscara HSV aplicada
Fuente: elaboración propia

Posteriormente, Tian recortaba la mitad superior mediante la función llamada *crop_top* para fijarse solamente en la parte más cercana del circuito. No obstante, tras aplicarlo se eliminaba demasiada información (figura 27), por lo que se optó por diseñar circuitos más estrechos y reposicionar ligeramente la cámara para que el coche pueda ver las líneas en la mitad inferior de la imagen .

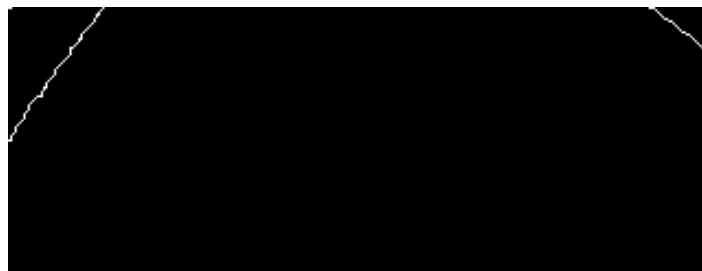


Figura 27. Vista de los bordes tras aplicar el recorte
Fuente: elaboración propia

A continuación, Tian emplea la función *HoughLinesP* de OpenCV para extraer líneas a partir de la máscara anterior. *HoughLinesP* traza líneas entre todos los puntos de la imagen con un mismo color y va acumulando las que pasan por los mismos puntos. Esta función también requirió de un proceso de prueba y error para obtener unos resultados satisfactorios, por lo que en la figura 28 se muestran los parámetros finales.

```
def get_lines(edges):  
  
    lines = cv2.HoughLinesP(  
        edges,  
        np.array([]), # vector de salida, cada línea será un vector  
        rho=1, # distancia acumuladora en píxeles  
        angle=(np.pi/180), # distancia angular acumuladora, en radianes  
        min_threshold=25, # mínimo de votos para el acumulador  
        minLineLength=10, # mínima longitud de una línea  
        maxLineGap=6 # máximo espacio para seguir siendo una sola línea  
    )  
  
    return lines
```

Figura 28. Función para obtener las líneas blancas observadas en una imagen
Fuente: «hand_coded_lane_follower.py»

Después de esto, el siguiente paso es trazar las dos líneas de calzada a partir de las distintas líneas obtenidas. Una de las complicaciones de esta etapa es cómo hacer que el programa sepa qué líneas corresponden a cada carril. Tian lo hace considerando que las líneas del tercio izquierdo son del carril izquierdo y las del tercio derecho del derecho. Después, calcula la pendiente y ordenadas promedio de las líneas y las reduce a una sola. Todo esto se realiza dentro de la función *get_lane_lines*.

Una vez obtenidas las líneas de la calzada, otra función llamada *compute_steering_angle* traza la línea de dirección a partir de estas y calcula el ángulo de giro para las ruedas. Si no se detecta ninguna línea se mantiene el ángulo, y si se detecta solo una se gira en esa dirección. Además, para evitar giros bruscos del coche, se añade otra función llamada *stabilize_steering_angle* para que no gire las ruedas más de 3 grados respecto al ángulo anterior, o 1 grado si solo ve una línea. En el caso de la figura 29, el ángulo deseado son 101 grados, pero por la limitación anterior el ángulo final son 97.

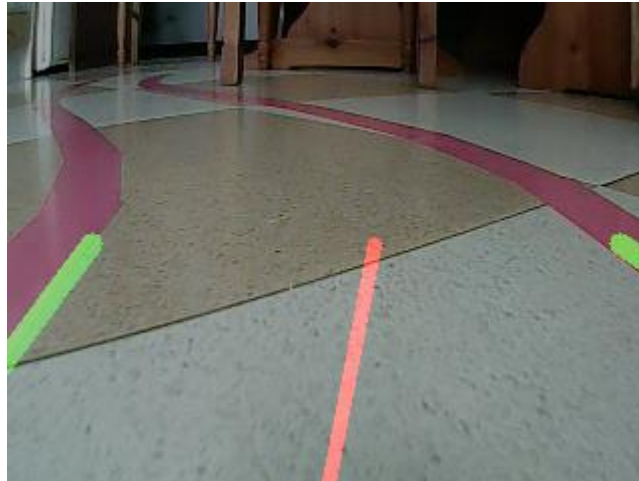


Figura 29. Fotograma del segundo circuito con las líneas detectadas y la dirección a seguir
Fuente: elaboración propia

Por último, el programa envía el ángulo a las ruedas delanteras utilizando la API del vehículo, mientras que graba la entrada de la cámara mediante OpenCV.

Durante el proceso de recopilación de los datos surgieron una serie de problemas que dificultaron la tarea. Uno de ellos fue la calibración de los servos del coche y el posicionamiento correcto de la cámara. Lo primero es una cuestión de prueba y error, pero lo segundo tiene una complejidad algo mayor. Cuando se graban vídeos en distintas sesiones de conducción, la posición y orientación de la cámara no deben variar para que los datos estén correctamente etiquetados. Si la cámara está girada un grado, o está medio centímetro más alta, hará que la conducción autónoma sea apreciablemente peor. Como la cámara no se mantiene fija sobre el chasis y son piezas pequeñas difíciles de fijar, se optó por hacer una fotografía de la calzada en una posición concreta del coche y utilizarla en sesiones posteriores. Con el robot en la misma posición, basta con posicionar la cámara para que la foto obtenida sea la misma.

Otros problemas encontrados durante la recopilación de los datos fueron debidos a fallos del vehículo. Uno de ellos era un error en el servo de la cámara que provoca que cada cierto tiempo este gire varios grados de forma impredecible, invalidando los datos recogidos a partir de ese instante. Tiempo después se descubrió que podía evitarse añadiendo la instrucción `self.vertical_servo.offset = 0` al programa `smart_pi_car.py`. No obstante, de vez en cuando este problema vuelve a suceder.

Otro inconveniente, y el más problemático, es que el modelo adquirido de Raspberry Pi tiene el lector de tarjetas microSD defectuoso. En muchas ocasiones no es capaz de leerla, siendo incapaz de encenderse y, en algunas, incluso deja de leerla con el coche encendido, provocando el apagado del mismo. Se probó a reinstalar el sistema operativo pero se acabó concluyendo que se trataba de un problema de *hardware*. Tras comprar otra tarjeta microSD y reinstalar todo el sistema y bibliotecas, se consiguió evitar este problema.

A pesar de los problemas surgidos, se consiguió que el programa condujera aceptablemente el coche en varios recorridos distintos. Mediante la ejecución posterior del programa `save_training_data.py`, que se encuentra también en el repositorio del

proyecto, se recopilaron cerca de 1000 imágenes en 10 vídeos con varios circuitos y distintas condiciones de iluminación.

No obstante, para conseguir esta muestra se habían hecho una serie de sacrificios. Había suelos por los que no se pudo conducir, se gastó mucho tiempo y se obtuvieron ángulos de giro mejorables. Además, se tuvieron que diseñar circuitos sencillos para que el algoritmo pudiera conducir sobre ellos.

Este último sacrificio se debe a un problema de difícil solución que no permite al algoritmo *get_lane_lines* funcionar correctamente. La cámara del robot no mira hacia la dirección a la que apuntan las ruedas delanteras, sino hacia la dirección a la que apunta el chasis. Es posible hacer girar la cámara mediante la API de SunFounder, pero en la práctica se comprobó que mover la cámara introducía demasiado ruido en los datos. El movimiento de la cámara difuminaba algunos fotogramas y hacía que, en ocasiones, la cámara tuviera que volver a calibrar su sensor, haciendo inservibles varios fotogramas.

Por tanto, como la cámara gira con el chasis y no con las ruedas, lo que ve el coche en las curvas no es la calzada a la que se dirige sino una esquina de la línea exterior de la curva. En la figura 30 se muestra un ejemplo de esta situación. Esa línea suele trascender el tercio izquierdo o derecho de la calzada, algo que utiliza el algoritmo *get_lane_lines* para separar entre las líneas de la izquierda y de la derecha.



Figura 30. *Fotograma de la cámara del coche ante una curva*
Fuente: elaboración propia

Esto confunde al método y empeora la conducción en todas las curvas en las que el coche tenga que girar más de 20 grados, algo que limitaría mucho la capacidad de conducción autónoma. Se probaron otros métodos para identificar las líneas, como separarlas en base a su pendiente, pero también fallaba en algunas situaciones y no se consiguió un método robusto. En la figura 31 se muestra un ejemplo en el que el coche viene de una curva cerrada hacia la derecha y está dentro de la calzada, a pesar de que no lo parezca. Pues bien, al eliminar el tercio o la mitad superior, resultaría difícil incluso para una persona saber si esa línea corresponde a la parte derecha o izquierda de la calzada. Por tanto, programar a mano un algoritmo que lo haga tampoco es una tarea fácil.



Figura 31. Fotograma de la cámara del coche ante un giro brusco con su correspondiente recorte de la mitad superior
Fuente: elaboración propia

Aun así, las imágenes obtenidas mediante este método se utilizaron inicialmente para entrenar algunos modelos de aprendizaje profundo, similares a los comentados posteriormente en el epígrafe 4.4. El error medio por giro obtenido era de unos 5 grados, algo teóricamente razonable, como se comentará al final del siguiente subepígrafe. Sin embargo, durante la conducción, el coche se salía frecuentemente de la calzada. Incluso haciendo una limpieza exhaustiva y recopilando muchas más imágenes, no se consiguió una fiabilidad aceptable.

Una posible solución era sustituir la cámara por otra con un sensor de mayor amplitud y calidad. No obstante, esto requería un desembolso económico y podría no haber solucionado el problema de la escalabilidad, al seguir fallando sobre ciertos suelos, como el de la figura 23. Además, aunque no fallara, requeriría retocar los filtros de la función *edges* para cada nuevo suelo y condiciones de iluminación en que se quisiera recopilar datos. Por tanto, se optó finalmente por desarrollar otro programa de conducción manual y volver a recopilar todos los datos, desechando los anteriores.

4.3.2. Conducción manual

Partiendo del programa «*smart_pi_car.py*» mencionado en el subepígrafe anterior, se sustituyó la llamada a los métodos de «*hand_coded_lane_follower.py*» por una a otro nuevo método de la clase *SmartPiCar*, llamado *manual_driver*, que cambia el ángulo de giro a partir de las entradas de teclado. A continuación, se detallan los aspectos más relevantes de su implementación.

Para comprobar la entrada del teclado se aprovechó la función *waitKey* de OpenCV. Se espera durante 50 milisegundos la pulsación de una tecla, y si ocurre se almacena el código de esta. Si la tecla pulsada es la “a” se decrementa el ángulo de giro en un grado y si es la “d” se incrementa. Para aprovechar la lectura del teclado se añadió también la posibilidad de parar el programa mediante la tecla “q”, de parar el coche mediante la tecla “p” y de arrancarlo mediante la tecla “g”.

Tras transmitir el ángulo a las ruedas se guarda una imagen del fotograma actual, cuyo título contiene el valor de ese ángulo. Para evitar giros extremos, se añadió un tope superior e inferior de 40 y 140 grados, respectivamente.

Este método supone varias mejoras respecto al anterior. En primer lugar, permite prescindir de los programas «*hand_coded_lane_follower.py*» y «*save_training_data.py*», reduciendo considerablemente las líneas de código y el tiempo empleado para recopilar los datos, ya que se recogen durante la conducción, y no después.

Por otra parte, permite recopilar datos bien etiquetados en cualquier entorno de conducción en que un humano pueda conducir el coche y sin tener que modificar ninguna línea de código. Esto puede ser interesante para futuras mejoras del proyecto, y permite una mayor variedad en los datos obtenidos, haciendo que el modelo sea más robusto.

Por último, si la conducción manual es buena, la precisión de los datos será mayor que con el programa autónomo. Potencialmente, permite recopilar unos datos mejor etiquetados, consiguiendo una autonomía más estable.

No obstante, este método tampoco está exento de inconvenientes. En primer lugar, es imprescindible conducir el coche con lo que se ve desde su cámara, no desde fuera. Por ejemplo, cuando el coche llega a una curva, solo ve la línea de la calzada que tiene delante, mientras que quien conduce el coche ve todo el circuito. Esto implica que, en una curva hacia la izquierda, el coche no tiene forma de saber a cuánta distancia está de la línea de la izquierda de la calzada (la que no ve). Por tanto, se debe conducir no intentando mantener el coche en el centro, sino a una distancia constante de la línea observada desde la cámara la del coche. Esto se descubrió tras la recopilación de todos los datos, obligando a repetir el proceso.

Por otra parte, debido a la necesidad de enviar instrucciones de forma remota a través del teclado, la latencia de la conexión SSH se convierte en un problema relevante. Así, los principales problemas encontrados durante la recopilación de los datos fueron la latencia y las pérdidas de conexión entre el dispositivo de control remoto y el coche. A veces, los giros tardaban en ejecutarse, llegando incluso a bloquearse las ruedas durante varios segundos.

Debido a esto, solo era posible recorrer el circuito con éxito en aproximadamente uno de cada tres intentos. Para mejorar la estabilidad se implementaron tres cambios:

- Reducción de la velocidad del coche a la mitad
- Adición de una instrucción *sleep(200)* para reducir los fotogramas por segundo guardados y por tanto la carga computacional
- Aumento del giro por clic de 1 a 3 grados, e incluso a 5 grados en circuitos con curvas más cerradas

Una vez introducidos, estos cambios permitieron recorrer los circuitos con éxito en la mayoría de los intentos, permitiendo aplicar este método para recopilar más imágenes bien etiquetadas. El código final se muestra en las figuras 32 y 33.

```
while self.camera.isOpened():
    _, frame = self.camera.read()
    cv2.imshow('Video', frame)

    self.manual_driver()

    cv2.imwrite('v%s-f%03d-a%03d.png' % (self.short_date_str, i,
self.steering_angle), frame)

    i += 1
    time.sleep(0.2)
```

Figura 32. Extracto del método “drive” que aplica la conducción manual

Fuente: «smart_pi_car.py»

```
def manual_driver(self):
    """ Drive using A and D keys """
    pressed_key = cv2.waitKey(50) & 0xFF
    if pressed_key == ord('a'):
        if self.steering_angle > 40: self.steering_angle -= 3
        self.front_wheels.turn(self.steering_angle)
    elif pressed_key == ord('d'):
        if self.steering_angle < 140: self.steering_angle += 3
        self.front_wheels.turn(self.steering_angle)
    elif key & 0xFF == ord('p'):
        self.back_wheels.speed = 0
    elif key & 0xFF == ord('g'):
        self.back_wheels.speed = speed
    elif pressed_key == ord('q'):
        self.cleanup()
```

Figura 33. Método para conducir mediante las entradas de teclado

Fuente: «smart_pi_car.py»

Una vez conseguida una conducción estable, se realizó el proceso de conducción y recopilación de los datos. Este proceso fue extenso y se tuvo que repetir por completo en varias ocasiones. Se aprovechó el aprendizaje adquirido para intentar conseguir un conjunto de datos lo más diverso y bien etiquetado posible. En la práctica, esto se tradujo en la implementación de las siguientes mejoras:

- Conducción por un total de 6 circuitos en 5 suelos distintos, mostrados en la figura 34. Estos fueron diseñados buscando la variabilidad en los recorridos y ángulos de giro. Además, se trató de mantener una distancia similar entre las dos líneas de la calzada, de aproximadamente un 120% de la anchura del robot.



Figura 34. Circuitos utilizados para la conducción manual

Fuente: elaboración propia

- Variación de la iluminación entre cada recorrido, empleando distintos focos de luz artificial y luz natural en distintas horas del día.
- Introducción de objetos contiguos a la calzada, tanto estáticos como móviles, simulando un entorno de conducción real.
- Introducción de manchas y perturbaciones en las líneas de la calzada.
- Práctica de la conducción en cada circuito para hacerla lo más uniforme posible.

A partir de estos circuitos y empleando estas técnicas, se obtuvieron imágenes bien etiquetadas en 16 recorridos distintos.

Además, durante y después del proceso de recopilación se realizó una limpieza y corrección manual de las imágenes obtenidas. Las acciones realizadas fueron las siguientes:

- Borrado de las 2-5 primeras imágenes de cada vídeo. La cámara tarda en adaptarse a la luz y los primeros fotogramas presentan una saturación muy alta.
- Corrección manual de algunos ángulos de giro en fallos manuales de conducción o en fallos de la Raspberry.
- Borrado de las últimas imágenes de cada vídeo, al llegar al final del circuito.

Tras completar el proceso de obtención y limpieza de las imágenes, se obtuvo un conjunto de datos de 4.734 imágenes etiquetadas de tamaño 320x240 píxeles. En la figura 35 se muestra la distribución de los ángulos de giro del conjunto de imágenes. A pesar de que no es uniforme, estando más representados los ángulos rectos, sí es suficientemente representativa de las situaciones de conducción que se encontrará el coche. En la figura 33 se añade una muestra de cuatro imágenes.

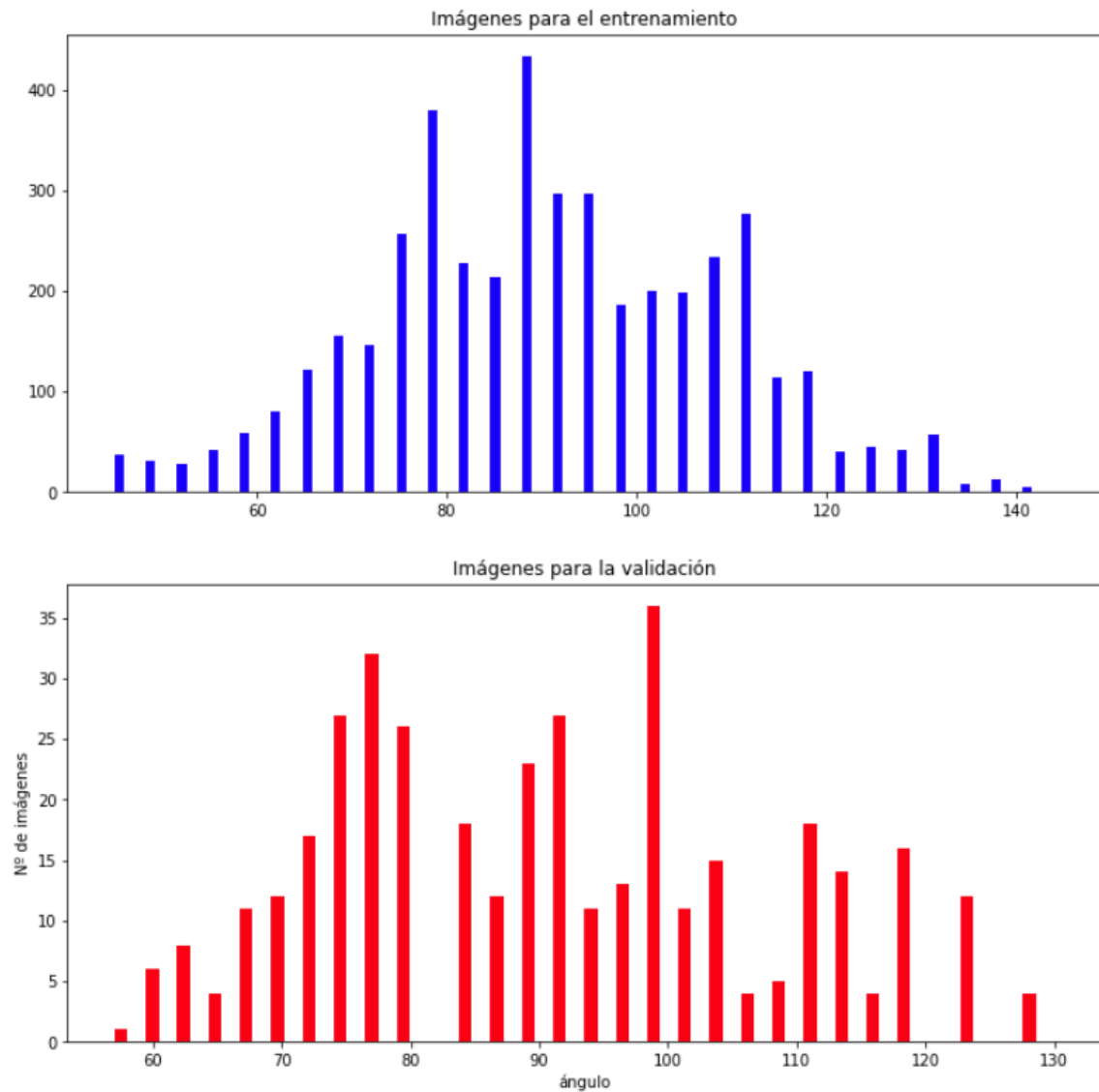


Figura 35. Distribución de los ángulos en las imágenes de entrenamiento y validación
Fuente: elaboración propia

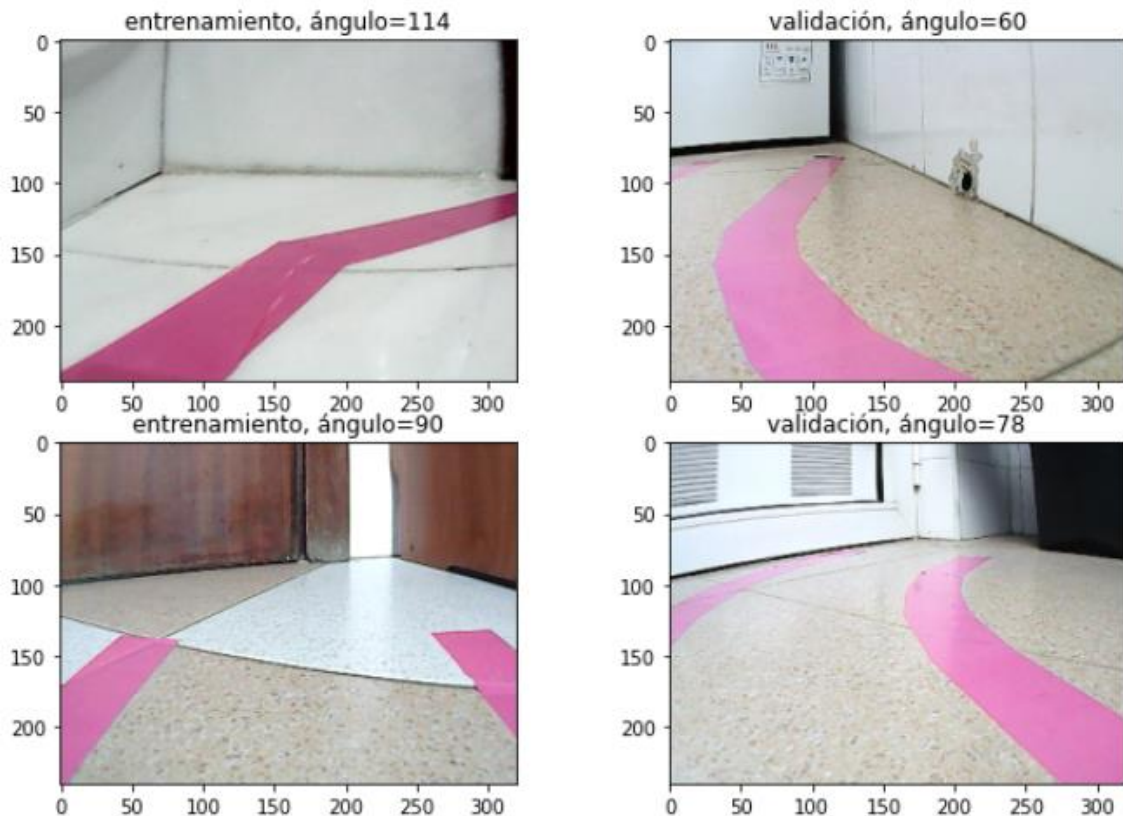


Figura 36. Muestra de cuatro fotografías de conducción con sus respectivos ángulos
Fuente: elaboración propia

Por último, antes de desarrollar y entrenar el modelo de aprendizaje profundo, es interesante conocer el error de las etiquetas del conjunto de datos. De esta forma, se puede tener una referencia del objetivo teórico del error del modelo. Desde un punto de vista teórico, el error debería ser como mínimo igual a la mitad del valor mínimo de sensibilidad en el giro. Como se ha conducido con un giro por clic de tres grados, no era posible conducir con mayor precisión que esta. Por tanto, en promedio, aunque la conducción hubiera sido perfecta, el error mínimo promedio debería ser de 1,5 grados.

Además de esto, se puede observar el error de forma empírica, aunque con menos exactitud. Para ello, basta con fijarse en dos imágenes con las líneas de la calzada en una posición similar y comparar la diferencia en el giro efectuado. En este caso, se observan diferencias de hasta 5 grados entre situaciones similares, por lo que es de esperar que un buen modelo no tenga un error muy por debajo de 5 grados. Teniendo en cuenta que en la conducción manual el coche no se ha salido de la calzada a pesar de estas pequeñas desviaciones, tampoco debería ocurrir durante la conducción autónoma.

4.4. Desarrollo del modelo de visión artificial

Una vez recopilados los datos, se procedió a desarrollar un código que permitiera una conducción autónoma gracias a un modelo de aprendizaje profundo. En este epígrafe se describen las decisiones relevantes de este proceso, las dificultades encontradas y los resultados de las pruebas realizadas durante la optimización del programa. Como se ha mencionado en el subepígrafe 3.4.6, se ha empleado Google Colab para esta sección del

trabajo. Los cuadernos con el código están también en repositorio del proyecto, en la dirección <https://github.com/andresmm98/Smart-Pi-Car/tree/master/models/lane-navigation/code>.

Para desarrollar el modelo de visión artificial, el primer paso es decidir su arquitectura. Continuando con el análisis realizado en el epígrafe 3.1, se concluyó que se trata de un problema de regresión cuyos datos de entrada son imágenes, y por tanto una red convolucional debería proporcionar un rendimiento razonablemente bueno. Para implementarla, como se ha mencionado en el subepígrafe 2.1.2, se puede hacer uso de modelos ya entrenados en vez de diseñar y entrenar uno desde cero. En este caso, habría que cambiar la función de pérdida y volver a entrenar las últimas capas del modelo. Así pues, se decidió empezar aplicando esta estrategia.

4.4.1. Modelo basado en MobileNet V2

Para elegir el modelo, debe tenerse en cuenta la limitación de capacidad de cómputo en la inferencia. En base a ello, el modelo escogido para utilizar como extractor de características ha sido el MobileNet V2 (60), entrenado en el conjunto de datos ImageNet. Este modelo parte de la red MobileNet mencionada en el subepígrafe 2.1.3 y añade *linear BottleNecks* que, de forma similar a las convoluciones en profundidad del anterior modelo, reducen la carga computacional, cambiando además la función no lineal por una lineal para no perder información. El otro avance significativo son las conexiones residuales invertidas, donde en vez de reducir la dimensionalidad para después ampliarla se realiza el proceso contrario.

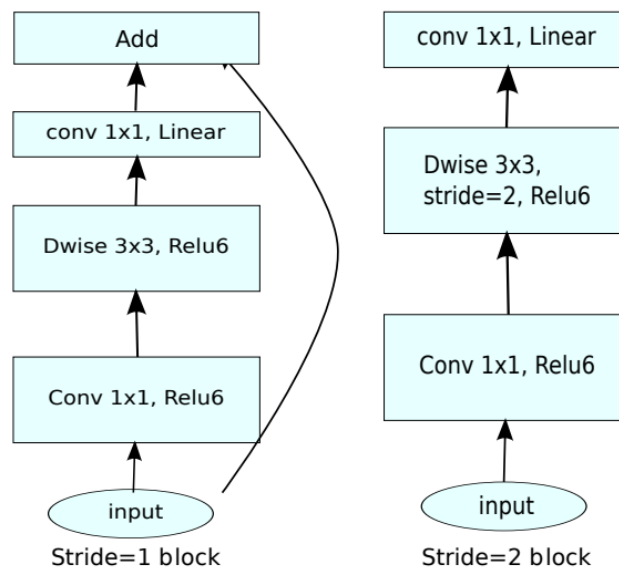


Figura 37. Bloques convolucionales de MobileNet V2

Fuente: (60)

Estas mejoras, representadas en la figura 37, junto a que todo el desarrollo de la red está enfocado a dispositivos móviles, se traducen en el segundo menor tiempo de inferencia, de todos los modelos disponibles en la API de Keras (61), solo superado por



su predecesor. Debido a la ligera mejora en rendimiento respecto a la primera MobileNet, se ha considerado que la segunda versión es la mejor para este problema.

La implementación de la red se encuentra en el archivo «*MobileNet.ipynb*». Dado que la red requiere que los datos de entrada estén en formato de tensores, se cargan los datos mediante la función `tf.keras.utils.image_dataset_from_directory`, dividiendo las imágenes en un conjunto de entrenamiento y otro de validación. Esta división no se ha hecho de forma aleatoria, sino que se ha dejado uno de los circuitos para el conjunto de validación y el resto para el entrenamiento, 4.347 y 387 imágenes, respectivamente. De esta forma, es posible identificar rápidamente el sobreajuste del modelo, ya que las imágenes del conjunto de validación no se parecen mucho a las usadas en el entrenamiento. Adicionalmente, las etiquetas se extraen a partir del nombre de los archivos, debiendo ordenarlos para mantener la equivalencia con las imágenes. Este proceso da lugar a dos variables de tipo `tf.dataset` que se podrán pasar como entrada al modelo de MobileNet una vez hechas algunas modificaciones.

El modelo se carga mediante la función `tf.keras.applications.MobileNetV2`, en la cual se especifican las dimensiones de las imágenes de entrada, que deben ser cuadradas y se han establecido en 224x224x3. También se especifica que los parámetros del modelo no se van a entrenar, y que no se quiere cargar la parte superior.

Las siguientes líneas de código se dedican a construir el resto del modelo. En primer lugar, se añaden tres capas de preprocesamiento. Una para eliminar la parte superior de las imágenes, otra para redimensionarlas y otra para cambiar la escala de los píxeles a una escala [-1,1], para facilitar el entrenamiento. Después, se añade el extractor de características ya cargado y, tras él, la cabeza de este nuevo modelo. Esta está formada por tres capas:

- Una capa convolucional con 32 filtros de 3x3 píxeles y activación RELU
- Una capa de agrupación manual, en lugar de capas totalmente conectadas, al igual que GoogleNet
- Una capa totalmente conectada sin función de activación, para obtener un solo valor de salida para cada imagen

El código que construye este modelo se muestra en la figura 38, siendo la variable `base_model` el modelo previamente cargado.

```
model = tf.keras.Sequential([
    tf.keras.layers.Cropping2D(cropping=((120,0),(0,0))),
    tf.keras.layers.Resizing(224,224),
    tf.keras.layers.Rescaling(1./127.5,offset=-1),
    base_model,
    tf.keras.layers.Conv2D(filters=32, kernel_size=3,activation='relu'),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(1)
])
```

Figura 38. Líneas de código que crean la red neuronal basada en MobileNet V2

Fuente: «*MobileNet.ipynb*»

Una vez construido se debe compilar el modelo, lo cual se hace eligiendo la función de pérdida y el optimizador. Como función de pérdida se eligió el error cuadrático medio, dado que penaliza errores muy elevados que harían que el coche saliera de la calzada. Como optimizador se ha escogido Adam, dado que suele converger más rápido que el descenso de gradiente estocástico (62). Una vez compilado y construido, el modelo en su totalidad tiene 2.626.689 parámetros, 368.705 de ellos entrenables, y su arquitectura es la que se muestra en la figura 39.

```

Model: "sequential_14"
-----
Layer (type)                Output Shape              Param #
-----
cropping2d_8 (Cropping2D)   (64, 160, 320, 3)        0
resizing_8 (Resizing)        (64, 224, 224, 3)        0
rescaling_8 (Rescaling)     (64, 224, 224, 3)        0
mobilenetv2_1.00_224 (Func  (None, 7, 7, 1280)       2257984
tional)
conv2d_8 (Conv2D)           (64, 5, 5, 32)           368672
global_average_pooling2d_5  (64, 32)                  0
(GlobalAveragePooling2D)
dense_8 (Dense)              (64, 1)                   33
-----
Total params: 2,626,689
Trainable params: 368,705
Non-trainable params: 2,257,984

```

Figura 39. Arquitectura de la red neuronal basada en MobileNet V2
Fuente: elaboración propia

Una vez construida la red, se continuó con el entrenamiento del modelo. Este se dividió en diez etapas con lotes de 32 imágenes, y se introdujeron puntos de guardado para no perder un modelo mejor que el del último entrenamiento. Tras completar el entrenamiento, el modelo con un error más bajo en el conjunto de validación fue el obtenido en la penúltima etapa, con un valor en la función de pérdida de 189. Por el contrario, el error del mismo modelo en el conjunto de entrenamiento es mucho menor, con un valor en la función de pérdida de 73. La evolución del error se muestra en la figura 40.



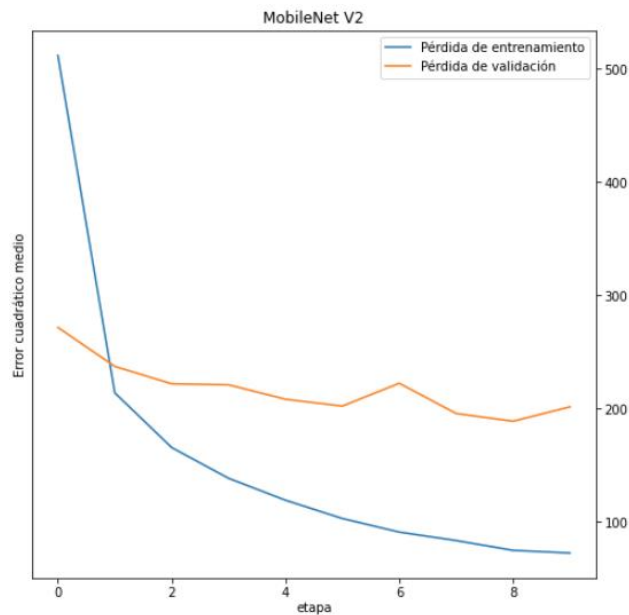


Figura 40. Evolución de la función de pérdida durante el entrenamiento del modelo basado en MobileNet V2

Fuente: elaboración propia

Esto es una muestra de un claro problema de sobreajuste, ya que el modelo está ajustando demasiado los parámetros a los circuitos sobre los que se está entrenando y no está generalizando el aprendizaje. Para intentar solucionarlo, se realizaron distintos ajustes a las capas añadidas e hiperparámetros del modelo. No obstante, los resultados fueron similares. La principal hipótesis es que esto se debe a una excesiva cantidad de parámetros y a que el aprendizaje del modelo original no es muy útil para este problema. El modelo de MobileNet está entrenado para clasificar objetos, mientras que se quiere conseguir un modelo que solo reconozca las líneas de la calzada y que distinga su posición y orientación para optimizar el giro. Por tanto, se llegó a la conclusión de que un modelo más simple entrenado desde cero podía obtener mejores resultados, por lo que se decidió probar esa alternativa.

4.4.2. Modelo construido desde cero

Diseñar la arquitectura de un modelo desde cero no es una tarea trivial y, como se ha comentado en el epígrafe 2.2, las empresas líderes del sector utilizan arquitecturas bastante diferentes entre sí. En este caso, se podría diseñar un modelo que detecte las líneas de calzada y otro que optimice el ángulo a partir de estas, pero se ha optado por utilizar un modelo *end to end* como Comma, es decir, emplear un solo modelo. Aunque la complejidad de su código es muy superior al requerido por este problema, un grupo de investigadores de Nvidia diseñó otro modelo similar pero mucho más sencillo (figura 41), que también consiguió aprender a conducir con éxito con una entrada y salida del modelo similares a las de este problema (63).

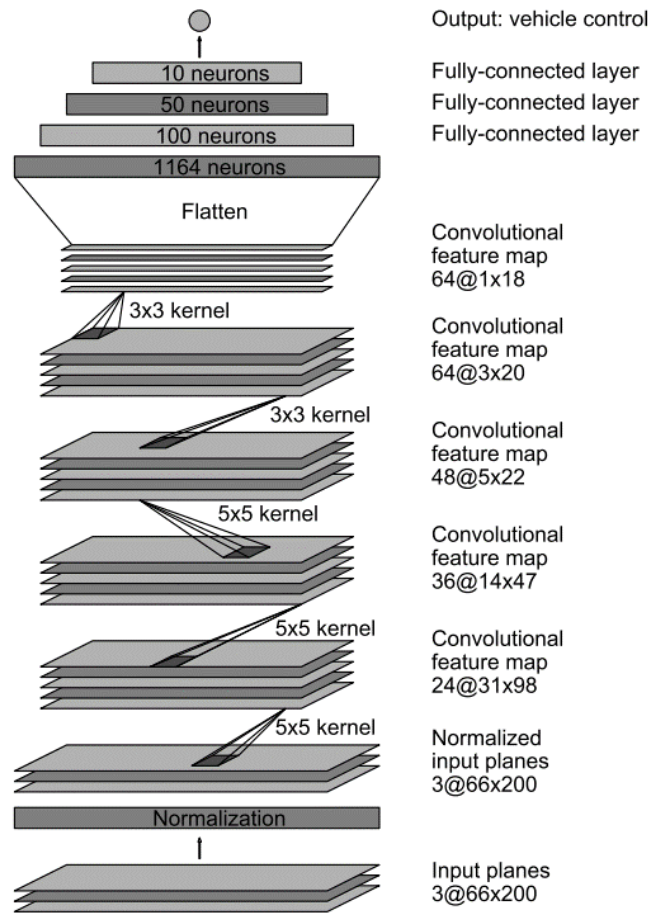


Figura 41. Representación gráfica de la arquitectura del modelo de Nvidia
Fuente: (63)

Así pues, se ha decidido partir de la arquitectura de Nvidia para diseñar el modelo. Esta es relativamente simple, consistiendo en una capa de normalización, cinco capas convolucionales, una capa aplanadora y tres capas totalmente conectadas, como se muestra en la figura 36. En total, cuenta con 9 capas, 250.000 parámetros y 27 millones de conexiones. Recibe como entrada una imagen de resolución 200x66 y devuelve el ángulo de giro. Por tanto, se creó una función llamada *img_preprocess* para cambiar el tamaño de las imágenes y cambiar los valores de los píxeles a una escala [-1,1]. Una vez implementado en Python, el resumen del modelo se muestra en la figura 42 y el código se encuentra en el archivo «*cnn-nvidia.ipynb*».

```

Model: "Model_Nvidia"
-----
Layer (type)                Output Shape                Param #
-----
conv2d_15 (Conv2D)          (None, 31, 98, 24)         1824
conv2d_16 (Conv2D)          (None, 14, 47, 36)         21636
conv2d_17 (Conv2D)          (None, 5, 22, 48)          43248
conv2d_18 (Conv2D)          (None, 3, 20, 64)          27712
conv2d_19 (Conv2D)          (None, 1, 18, 64)          36928
flatten_3 (Flatten)         (None, 1152)                0
dense_12 (Dense)            (None, 100)                 115300
dense_13 (Dense)            (None, 50)                   5050
dense_14 (Dense)            (None, 10)                    510
dense_15 (Dense)            (None, 1)                      11
-----
Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
-----
None
    
```

Figura 42. Arquitectura de la red neuronal basada en el modelo de Nvidia
 Fuente: elaboración propia

El primer entrenamiento se realizó con 10 etapas, lotes de 32 imágenes, y el mismo optimizador y función de pérdida que en el caso anterior. El resultado de este fue un error cuadrático medio en el mejor modelo de unos 13 grados, tanto en el conjunto de entrenamiento como en el de validación. En la figura 43 se aprecia como ambos errores se reducen a medida que avanza el entrenamiento, es decir, el modelo converge. Los resultados obtenidos son más prometedores que los del modelo de MobileNet y en este caso sí que parece haber una generalización del aprendizaje, por lo que se descartó el modelo anterior y se continuó optimizando este.

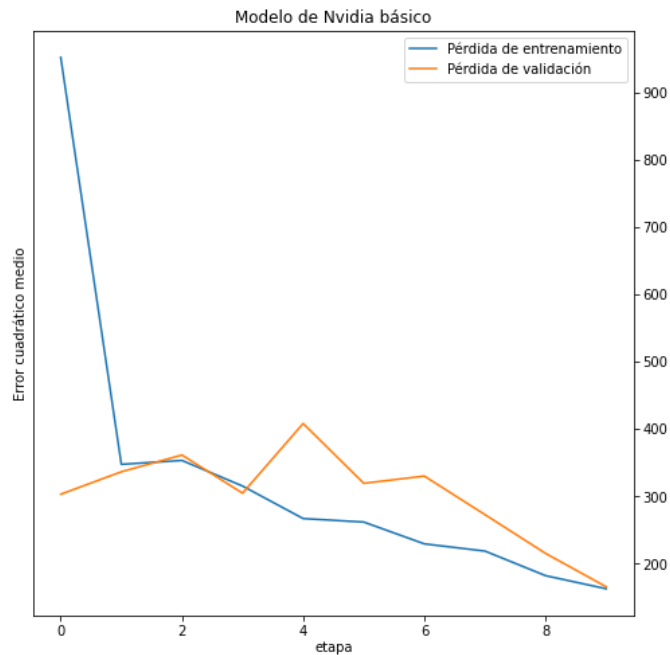


Figura 43. Evolución de la función de pérdida en el entrenamiento del modelo de Nvidia
Fuente: elaboración propia

Tras los primeros entrenamientos, se realizaron algunas pruebas de conducción. Sin embargo, estas no fueron satisfactorias, lo cual evidenció la necesidad de hacer algunas mejoras en el modelo. En primer lugar, un problema observable a simple vista es que la cantidad de datos disponibles es mucho menor que la que usaron en Nvidia para entrenar su modelo: 72 horas de vídeo de conducción. Por tanto, la principal mejora implementada fue el uso de **técnicas de aumento de datos**.

En los problemas de clasificación de imágenes, suelen hacerse modificaciones aleatorias en estas durante el entrenamiento para hacer más robusto el modelo. Sin embargo, en este problema no pueden aplicarse muchas de las modificaciones habituales, como *zoom*, giros o desplazamientos, porque cambiaría el ángulo de giro y habría que modificar las etiquetas. Por tanto, se probó a aplicar solamente cuatro técnicas de aumento: difuminado, invertido, cambio de brillo y cambio de contraste.

Invertir las imágenes también cambia el ángulo de giro deseado, pero no de forma impredecible. Al voltear las imágenes horizontalmente, su nuevo ángulo y, por tanto, su nueva etiqueta, es igual a la resta entre 180 y el ángulo anterior. Esto es cierto debido a que el coche no puede girar más de 90 grados y a que 90 grados es equivalente a continuar en línea recta. No obstante, Tensorflow no tiene una función que permita invertir las imágenes y cambiar las etiquetas, por lo que se intentó programar manualmente un **generador** de lotes de imágenes con las funciones de aumento.

En primer lugar, se programó una función que devolviera una imagen aumentada y su etiqueta, a modo de generador, para ser llamada durante el entrenamiento y generar los lotes de imágenes. No obstante, Tensorflow no permite su uso ya que, al elegir una imagen aleatoria, es posible que la misma imagen se emplee varias veces durante una misma etapa del entrenamiento.

A continuación, se probó a utilizar el método recomendado por Tensorflow. Este consiste en crear una clase heredando de `tf.keras.utils.Sequence`, para crear un objeto *Secuencia* que genere lotes de imágenes aumentadas. No obstante, esta clase tiene unas condiciones estrictas que no permitieron implementar el cambio del valor de las etiquetas. A pesar de que ambas opciones fracasaron, el código se conserva en el archivo «*cnn-nvidia.ipynb*».

Finalmente, se creó una función para invertir todas las imágenes de forma previa al entrenamiento, duplicando así el conjunto de datos inicial y obteniendo un total de 9.468 imágenes etiquetadas. Cabe mencionar que, al duplicar el tamaño del conjunto de datos, el tiempo de ejecución del código previo al entrenamiento aumentó al entorno de una hora utilizando los procesadores que ofrece Colab. El grueso de este tiempo se debía al uso de OpenCV en el preprocesamiento de las imágenes. Por tanto, para reducir este tiempo, se probó a implementar el preprocesamiento dentro del modelo y a cargar las imágenes en objetos `tf.keras.utils.data.Dataset`. Esto se implementó en el archivo «*cnn-nvidia-tensors.ipynb*» y consiguió reducir el tiempo de ejecución previo al entrenamiento al rango de unos 5 minutos.

Además, Tensorflow recomienda añadir unas líneas de código de forma previa al entrenamiento para guardar los conjuntos de datos en la memoria caché y así reducir el tiempo de entrenamiento. Este código se muestra en la figura 44 e hizo que el tiempo de entrenamiento aumentara solamente 20 segundos por etapa respecto a la implementación anterior sin tensores.

```
AUTOTUNE = tf.data.AUTOTUNE

train_dataset = train_dataset.cache().prefetch(buffer_size=AUTOTUNE)
val_dataset = val_dataset.cache().prefetch(buffer_size=AUTOTUNE)
```

Figura 44. Líneas de código para guardar los conjuntos de datos en caché

Fuente: «*cnn-nvidia-tensors.ipynb*»

No obstante, durante el entrenamiento, se observó un problema de sobreajuste que no estaba presente en la anterior implementación. Tras investigar las posibles causas, se concluyó que la más probable es la diferencia en el **mezclado aleatorio** del conjunto de datos. Antes de cada etapa del entrenamiento, si así se le ha indicado en el parámetro *shuffle* del método `model.fit`, este mezcla aleatoriamente el conjunto de datos de entrenamiento, lo cual beneficia a la generalización del aprendizaje.

Por el contrario, cuando el conjunto de datos es un objeto de tipo `Dataset`, el método delega en este objeto la tarea de mezclado. Como el método *shuffle* de la clase `Dataset` solo mezcla los lotes, y no todos los elementos, las diferencias entre cada etapa son menores. Esto podría perjudicar al entrenamiento y ser la causa de esta diferencia en el error obtenido. Debido a esto, se optó por utilizar la implementación sin tensores pese a su mayor tiempo de ejecución.

A continuación, habiendo descartado esta opción y con las imágenes ya invertidas, se continuó implementando el resto de técnicas de aumento de datos. Sin embargo, ni el difuminado, ni el cambio de contraste ni el cambio de brillo mejoraron de forma apreciable el error. Se probó a implementarlas tanto dentro como fuera del modelo y no

dieron buenos resultados, por lo que finalmente no se añadió ninguna. Aun así, el código se conserva en el archivo «*cnn-nvidia.ipynb*»

Por último, en cuanto a otras posibles mejoras de la red, se trabajó en la arquitectura del modelo y en la elección de los hiperparámetros. Los mejores resultados se consiguieron con la lista que se expone a continuación.

- Tamaño del lote: **32 imágenes**. La evidencia de diversos estudios muestra que tamaños de lote superiores a 32 aumentan el tiempo de entrenamiento y empeoran la generalización del modelo (64). En el entrenamiento de este modelo se han observado las mismas mejoras al reducir el tamaño a 32. En cambio, reduciendo a 16 imágenes no se ha notado una mejora apreciable.
- Optimizador: **Adam**. Se probaron varios optimizadores, como el descenso de gradiente estocástico con Momentum y con Momentum de Nesterov, pero estos no convergían aun cambiando la tasa de aprendizaje y el valor del Momentum. Los mejores resultados se consiguieron con Adam y con una tasa de aprendizaje de 0,001.
- Función de activación: **ELU**. Los resultados fueron ligeramente mejores a los obtenidos con las funciones RELU y GELU, convergiendo a un error más bajo. La función ELU (unidad lineal exponencial) presenta la ventaja de que no se satura ante valores negativos, a diferencia de RELU, pero mantiene un valor mínimo de -1 para evitar la propagación de ruido (figura 45).

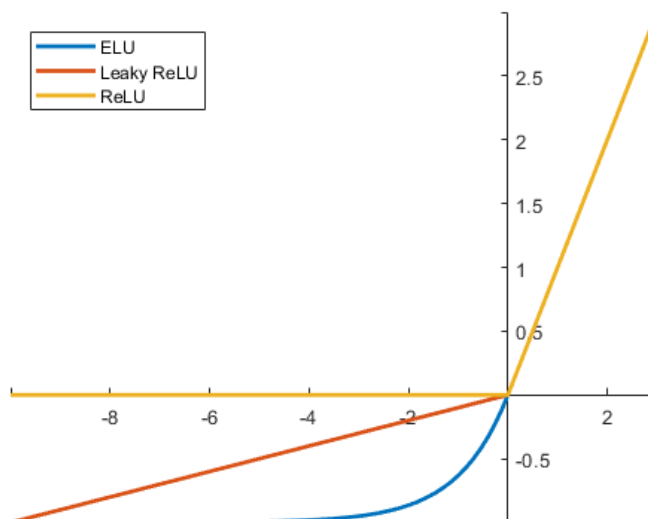


Figura 45. Salidas de las funciones ELU, Leaky ReLU y RELU para valores cercanos a 0
Fuente: (65)

- Método de inicialización de pesos: **Glorot Uniform**, también llamado Xavier Uniform. Otros inicializadores, como Glorot Normal y Lecun Uniform, dieron peores resultados. Glorot Uniform inicializa los pesos para que la varianza de las activaciones sea la misma en cada capa, y con una distribución uniforme, lo cual previene que el gradiente se desvanezca o aumente demasiado.

- Espacio de color imágenes: **RGB**. La red de Nvidia fue entrenada con imágenes en espacio de color YUV. No obstante, en este modelo se han obtenido mejores resultados entrenando la red con imágenes en el espacio RGB. Una posible razón es que en el caso de Nvidia las líneas eran blancas, mientras que en este circuito son de color rosa.
- Arquitectura del modelo:
 - Añadir capas de *dropout* aumentaba ligeramente el error, tanto en el conjunto de entrenamiento como en el de validación.
 - Cambiar las capas totalmente conectadas por una capa aplanadora o una capa de agrupación por promedio, como GoogLeNet, aumentaba el error en varios grados.
 - Cambiar el tamaño de los filtros de las capas convolucionales, o cambiar el número de estas capas, aumentaba en varios grados el error de la red.

Tras aplicar estas mejoras, el mejor modelo se consiguió en la novena etapa de un entrenamiento con los ajustes óptimos mencionados. El error cuadrático medio en el conjunto de entrenamiento fue de 20,3 y en el conjunto de validación de 23 y un error promedio aproximado en el ángulo de giro de 4,5 grados y 4,8 grados, respectivamente. En la figura 46 se muestra la evolución de la función de pérdida durante el entrenamiento.

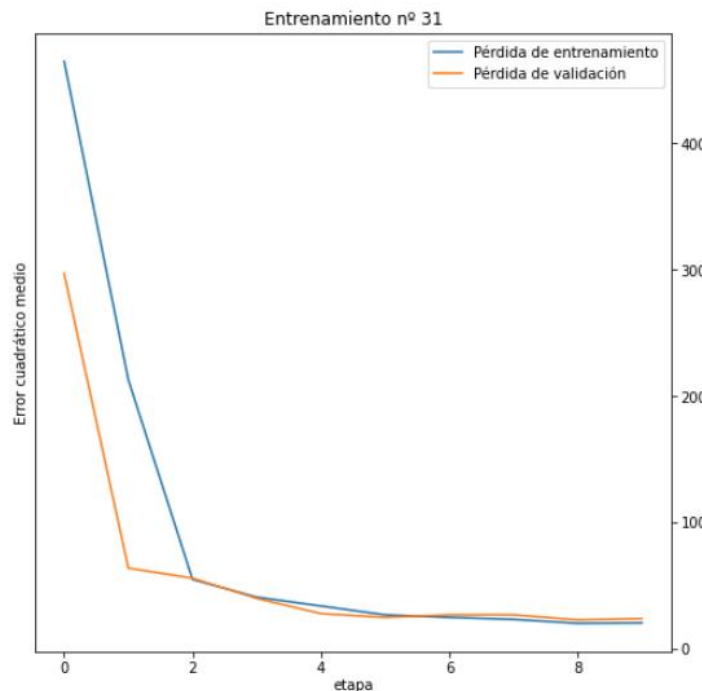


Figura 46. Evolución de la función de pérdida en el entrenamiento del modelo de Nvidia mejorado

Fuente: elaboración propia

Tal y como se ha explicado en el subepígrafe 4.3.2, un error en el ángulo inferior a 5 grados es aceptable y debería ser suficiente para que el coche pudiera conducir de forma autónoma. Por tanto, se procedió a realizar las pruebas de conducción para comprobar su efectividad.

4.5. Despliegue y ajuste del modelo

Para poder ejecutar el modelo en la Raspberry Pi, son necesarios varios pasos previos. Por un lado, se debe compilar el modelo para poder ejecutarse en el acelerador Coral. Por otro, se debe adaptar ligeramente el código original para que delegue la conducción en el modelo de conducción autónoma.

Para permitir la ejecución del modelo en el acelerador, se siguieron las instrucciones del fabricante disponibles en la web del producto. Para sistemas operativos basados en Linux, se debe instalar en ellos una biblioteca de la Edge TPU y otra de PyCoral, para Python 3. El siguiente paso, opcional, es optimizar el modelo para que toda la inferencia pueda ejecutarse en la TPU, y esta sea lo más rápida posible. En otro apartado de la web anterior está disponible una guía para realizar esta optimización. No obstante, el acelerador Coral no soporta varias de las operaciones que se realizan dentro de la red, como las funciones ELU y la capa aplanadora. Por tanto, se comprobó si la velocidad de ejecución era un cuello e botella y, como no lo era, no se adaptó el modelo para el acelerador de Google.

Por tanto, una vez descargadas las bibliotecas, el siguiente paso fue compilar el modelo original para el acelerador, mediante la función `tf.lite.TFLiteConverter.from_keras_model`. El código de conversión se encuentra al final del cuaderno «*cnn-nvidia.ipynb*». Esta función crea un modelo con extensión `.tflite` que puede ser procesado por la TPU.

A continuación, se procedió a crear un programa para implementar la conducción autónoma, aprovechando el código ya desarrollado, al que se le dio el nombre de «*autonomous_driver.py*». Este programa importa la biblioteca de Pycoral y contiene las siguientes funciones y métodos

- Un método constructor de la clase *LaneFollower* que crea un intérprete de la TPU para el modelo cargado.
- La función *img_preprocess*, similar a la empleada para el entrenamiento del modelo, que preprocesa los fotogramas de la cámara antes de enviarlos a la red neuronal.
- El método *compute_steering_angle* de la clase *LaneFollower*, que pasa el fotograma preprocesado al modelo y devuelve el ángulo de giro. El código de este método se muestra en la figura 47. Cabe destacar que la TPU almacena la salida del modelo en la clave *index* del primero de una lista de diccionarios a la que se accede mediante el método *get_output_details* de la clase *tf.lite.interpreter*.



```

def compute_steering_angle(self, frame):
    input_frame = img_preprocess(frame)
    common.set_input(self.interpreter, input_frame)

    output_details = self.interpreter.get_output_details()[0]
    self.interpreter.invoke()
    steering_angle =
self.interpreter.get_tensor(output_details['index'])

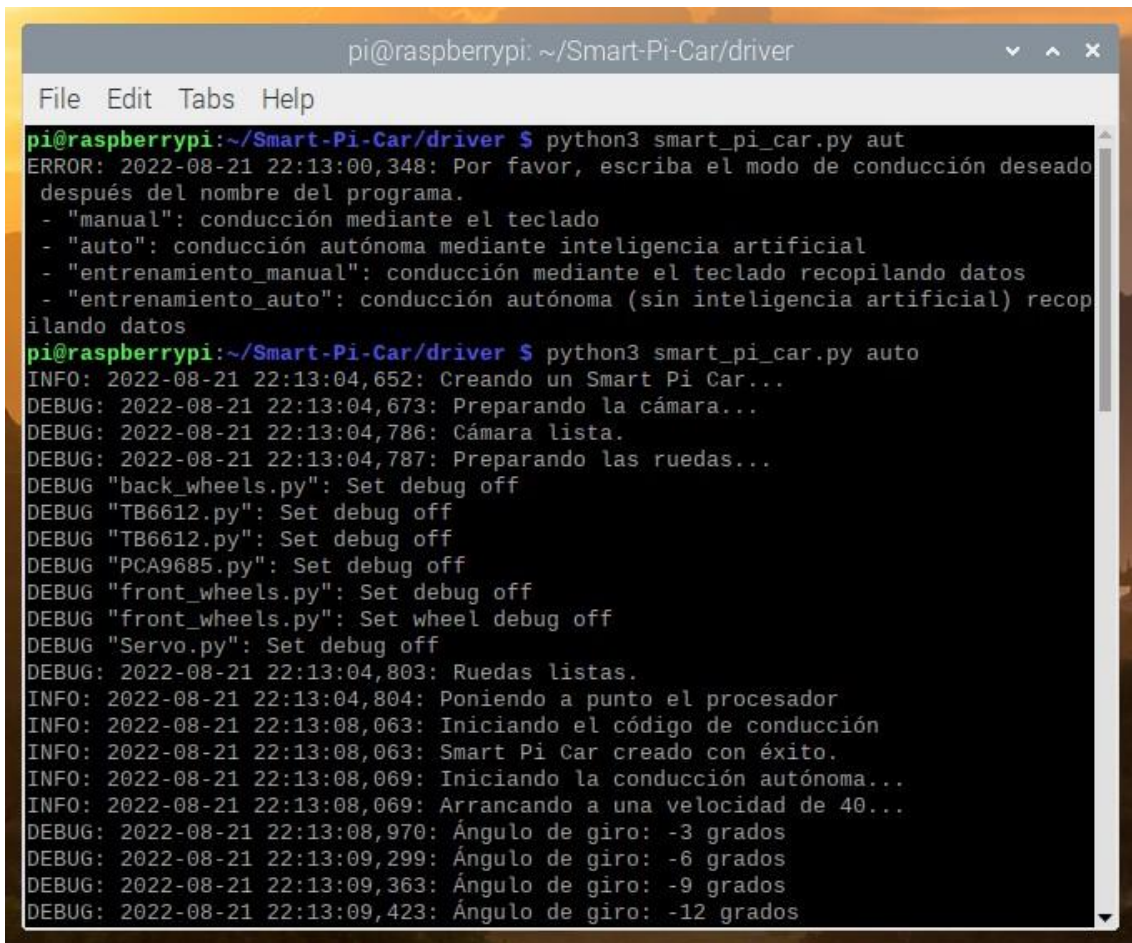
    steering_angle = int(steering_angle + 0.5) # redondeo
    logging.debug(f'Ángulo de giro: {steering_angle - 90}')
    return steering_angle
    
```

Figura 47. Método para calcular el ángulo de giro mediante el modelo entrenado

Fuente: «*autonomous_driver.py*»

- El método *stabilize_steering_angle* de la clase *LaneFollower*, que limita la diferencia entre un ángulo de giro y el anterior a 3 grados, para evitar giros bruscos y minimizar el impacto de los errores.
- La función *display_heading_line*, que, a partir de un fotograma y un ángulo de giro, devuelve el mismo fotograma con una línea roja en la dirección a la que se dirige el coche. Juntando estos fotogramas se obtiene un vídeo que se muestra por pantalla desde el programa principal, «*smart_pi_car.py*».
- El método *follow_lane* de la clase *LaneFollower*, que, a partir de un fotograma dado, llama al resto de funciones, envía el ángulo de giro calculado a las ruedas delanteras del coche y devuelve el fotograma con la línea de dirección.

Además de este nuevo programa, se hicieron algunos ajustes al programa «*smart_pi_car.py*» para que hiciera uso de las funciones anteriores. En concreto, se implementó una funcionalidad que permite al usuario escoger entre cuatro modos de conducción: “manual”, “auto” y “entrenamiento_manual” y “entrenamiento_auto”. El modo escogido debe introducirse en la línea de comandos después de la llamada al programa. Si no se introduce ninguno, el coche se conduce de forma autónoma y, si se introduce otra palabra, se muestra por el terminal una guía de los modos disponibles. En la figura 48 se muestra un ejemplo de lo que vería en pantalla un usuario al ejecutar el programa.



```
pi@raspberrypi: ~/Smart-Pi-Car/driver
File Edit Tabs Help
pi@raspberrypi:~/Smart-Pi-Car/driver $ python3 smart_pi_car.py aut
ERROR: 2022-08-21 22:13:00,348: Por favor, escriba el modo de conducción deseado
después del nombre del programa.
- "manual": conducción mediante el teclado
- "auto": conducción autónoma mediante inteligencia artificial
- "entrenamiento_manual": conducción mediante el teclado recopilando datos
- "entrenamiento_auto": conducción autónoma (sin inteligencia artificial) recopilando datos
pi@raspberrypi:~/Smart-Pi-Car/driver $ python3 smart_pi_car.py auto
INFO: 2022-08-21 22:13:04,652: Creando un Smart Pi Car...
DEBUG: 2022-08-21 22:13:04,673: Preparando la cámara...
DEBUG: 2022-08-21 22:13:04,786: Cámara lista.
DEBUG: 2022-08-21 22:13:04,787: Preparando las ruedas...
DEBUG "back_wheels.py": Set debug off
DEBUG "TB6612.py": Set debug off
DEBUG "TB6612.py": Set debug off
DEBUG "PCA9685.py": Set debug off
DEBUG "front_wheels.py": Set debug off
DEBUG "front_wheels.py": Set wheel debug off
DEBUG "Servo.py": Set debug off
DEBUG: 2022-08-21 22:13:04,803: Ruedas listas.
INFO: 2022-08-21 22:13:04,804: Poniendo a punto el procesador
INFO: 2022-08-21 22:13:08,063: Iniciando el código de conducción
INFO: 2022-08-21 22:13:08,063: Smart Pi Car creado con éxito.
INFO: 2022-08-21 22:13:08,069: Iniciando la conducción autónoma...
INFO: 2022-08-21 22:13:08,069: Arrancando a una velocidad de 40...
DEBUG: 2022-08-21 22:13:08,970: Ángulo de giro: -3 grados
DEBUG: 2022-08-21 22:13:09,299: Ángulo de giro: -6 grados
DEBUG: 2022-08-21 22:13:09,363: Ángulo de giro: -9 grados
DEBUG: 2022-08-21 22:13:09,423: Ángulo de giro: -12 grados
```

Figura 48. Texto mostrado en el terminal al ejecutar el programa «smart_pi_car.py»
Fuente: elaboración propia

Una vez implementado este código, se procedió a probar el desempeño del modelo en los circuitos. Para ello, se condujo el coche de forma autónoma por los 6 recorridos diseñados en ambos sentidos, y se diseñó un nuevo circuito sobre un suelo de un color diferente, sobre el que el coche no había sido entrenado. Este circuito, con una calzada superior a 4 metros de longitud, se muestra en la figura 49.





Figura 49. Circuito diseñado para realizar las pruebas de conducción autónoma
Fuente: elaboración propia

En los 12 recorridos iniciales, el coche solo se salió de la calzada en una ocasión. En el nuevo circuito, el coche lo recorrió con éxito en ambos sentidos con la luz de la estancia encendida. Sin embargo, al depender solo de la luz natural de menor intensidad, el coche se salió de la calzada en uno de los sentidos.

Para tratar de corregir estos errores, se recopilaron datos de las zonas problemáticas mediante una conducción manual y se realizó un segundo entrenamiento sobre el modelo inicial para ajustarlo. En total, se obtuvieron 704 imágenes que, tras invertirlas, se ampliaron a 1.408. Para evitar el sobreajuste del modelo a los nuevos datos, no se entrenó solo con estas imágenes, sino que se añadieron al conjunto de datos original. Además, se realizó un entrenamiento de solo 5 etapas con una tasa de aprendizaje inferior, de 0,00003.

En la figura 50 se muestra la evolución de la tasa de aprendizaje desde la segunda etapa del entrenamiento anterior hasta la última del segundo entrenamiento, indicando en verde el inicio de este. En los primeros lotes del nuevo entrenamiento, el error cuadrático medio aumentó hasta un valor de 30, mostrando que el modelo no había aprendido a conducir correctamente en esos escenarios. No obstante, tras varias etapas, el modelo convergió a un error en el conjunto de validación menor al del anterior modelo. El mejor modelo fue el obtenido tras la etapa número 13, con un error cuadrático medio en el conjunto de entrenamiento y en el de validación de 20,9 y 21,4. El error medio en el ángulo de giro era aproximadamente de 4 grados en ambos conjuntos de datos.

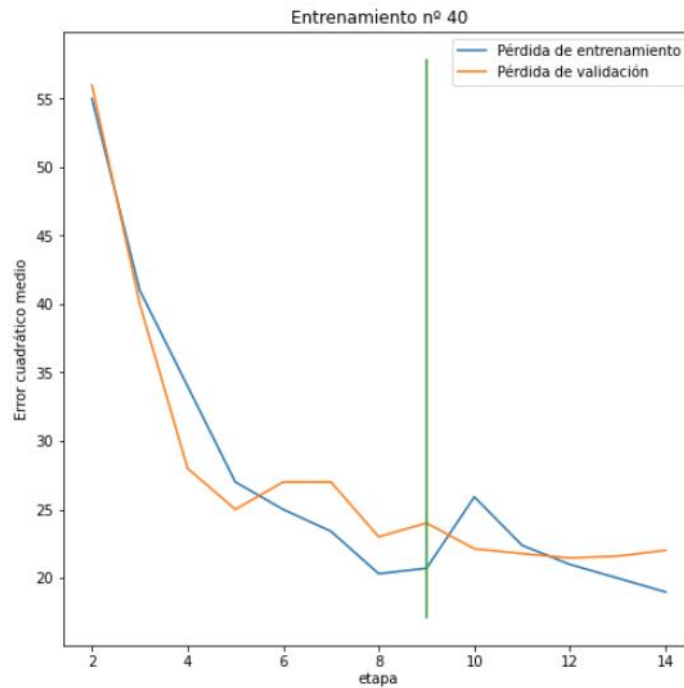


Figura 50. Evolución de la función de pérdida en el segundo entrenamiento del modelo de Nvidia mejorado
Fuente: elaboración propia

Tras volver a compilar el modelo para el acelerador, se repitieron las mismas pruebas de conducción anteriores. En esta ocasión, el coche completó con éxito los 14 recorridos sin salirse de la calzada. Dado que el entrenamiento se había realizado a una velocidad de 20 (unidad no especificada por la API del coche), se hicieron pruebas de conducción aumentando la velocidad. A medida que esta aumenta, el coche realiza los giros con un retraso mayor y se acerca más al borde de la calzada. No obstante, el coche consiguió recorrer los circuitos con éxito hasta una velocidad de 60, comenzando a fallar a partir de esa cifra.

Así pues, con esto se dieron por finalizadas las pruebas de conducción. El modelo entrenado y ajustado ha demostrado ser capaz de generalizar el aprendizaje y de no salirse de la calzada en circuitos que nunca ha recorrido, mientras que las líneas de la calzada sean de las mismas características.

5. Conclusiones

En el presente trabajo se ha abordado la tarea de construir un coche autónomo en miniatura capaz de conducir de forma autónoma bajo ciertas condiciones preestablecidas. En este último capítulo se hace una revisión del cumplimiento de los objetivos propuestos, se describen algunas conclusiones y aprendizajes obtenidos y se mencionan algunas líneas de mejora.

Los tres subobjetivos principales del trabajo eran los siguientes:

1. Construir un coche robótico programable y con suficientes sensores para permitir una conducción autónoma.

Este objetivo ha sido cumplido mediante el uso del SunFounder PiCar-V y una serie de dispositivos hardware conectados a este. Como sensores, una cámara de baja calidad y el velocímetro del coche han sido suficientes para permitir una autonomía parcial. No obstante, han existido varios cuellos de botella que limitan las capacidades del robot. Estos son los siguientes:

- La cámara: una cámara con mayor ángulo de visión y una mejor lente mejorarían la autonomía del coche.
- La placa base: la Raspberry Pi Model 3 B+ ha permitido cumplir los objetivos, pero ha provocado muchos fallos y retrasos. En especial, la tarjeta gráfica no es suficiente para renderizar en tiempo real los vídeos del coche, y la ranura de tarjetas microSD parece tener una avería. Un modelo de Raspberry más reciente acortaría el tiempo necesario para llevar a cabo el proyecto de forma considerable y mejoraría el resultado final.
- El posicionamiento de la cámara en el chasis: tener que ajustar manualmente y a ojo la posición de la cámara antes de cada sesión de conducción es algo tedioso y problemático. Para tener una solución más robusta, sería necesario diseñar un sistema de fijación de la cámara.

2. Implementar un programa que permita conseguir la autonomía parcial deseada.

Conseguir este objetivo ha sido posible gracias al desarrollo de varios programas, tanto para el control manual del coche como para el modelo de aprendizaje automático. Se considera que se ha conseguido una solución escalable, con posibilidad de mejoras futuras, y fácil de entender y utilizar para un usuario familiarizado con la programación. Se han añadido algunas funcionalidades adicionales no planteadas al inicio del trabajo, como la posibilidad de que el usuario elija entre cuatro modos de conducción, incluyendo una conducción autónoma programada de forma explícita.

Cabe destacar que para conseguir un modelo capaz de conducir de forma autónoma han sido necesarios un extenso proceso de prueba y error, la

recopilación de una gran cantidad de datos y la optimización del modelo desde muchos de los frentes posibles. El principal margen de mejora se aprecia en el correcto etiquetado de los datos, disminuyendo la diferencia de giro de 5 grados entre algunas imágenes iguales.

3. Ejecutar el programa en tiempo real y de forma local desde el robot.

Este ha sido el objetivo más sencillo de los tres, ya que el procesador de la Raspberry y el acelerador Coral tenían potencia de cómputo de sobra para realizar la inferencia del modelo. Sin embargo, si se hubiera empleado el modelo basado en la red de MobileNet V2, al tener 10 veces más parámetros, habría sido más complejo de conseguir. Para poder realizar la inferencia en el vehículo, ha bastado con descargar las librerías de Coral, compilar el modelo para la TPU y añadir algunas líneas de código en el programa.

En cuanto a las posibilidades de mejora, destaca la de adaptar el modelo para que pueda ejecutarse por completo en la TPU, y así aliviar la carga computacional del coche.

Gracias al cumplimiento de estos tres subobjetivos, se ha cumplido también el objetivo principal del trabajo: hacer que el coche recorra de forma autónoma un circuito no visto previamente, de al menos dos metros de longitud, al primer intento y en ambos sentidos. Este objetivo se ha cumplido al recorrer el circuito de la figura 49, de unos cuatro metros de longitud, y con el añadido de ser un suelo distinto a los recorridos durante el entrenamiento.

Además, se considera que hacer público el código de este proyecto podría ayudar en la realización de trabajos similares y en la experimentación de pequeños desarrolladores con la conducción autónoma. Partiendo de los programas aquí desarrollados, el tiempo necesario para desarrollar una solución similar podría ser considerablemente menor.

Por otra parte, durante la realización del trabajo se han obtenido diversos aprendizajes que pueden ser útiles para proyectos similares. Estos se listan a continuación:

- Un buen hardware es fundamental para conseguir una solución robusta y resistente a fallos.
- Con diferencia, la mejor forma de mejorar el modelo ha sido recopilar más datos y con la mayor variedad posible.
- Técnicas de aumento de datos tradicionales como el cambio de brillo y contraste aleatorios no han mejorado el aprendizaje del modelo.
- Algunas técnicas populares como las capas de *dropout* o capas de agrupamiento no han mejorado el aprendizaje del modelo.
- La elección de algunos hiperparámetros, como la función de activación, el tamaño del lote y el optimizador ha resultado ser relevante en el aprendizaje del modelo, tanto en velocidad como en rendimiento.
- Ha sido posible conseguir un robot de conducción autónoma con un presupuesto inferior a los 300€ y un número de imágenes menor a 10.000.



En cuanto a aprendizajes aplicables a la conducción autónoma a escala real, también se ha llegado a algunas conclusiones:

- Conseguir una autonomía SAE Level 4 o Level 5 no parece posible sin la supervisión de operarios. La red neuronal no deja de ser una caja negra, por lo que es imposible saber con certeza cómo va a reaccionar ante una situación hasta que se la encuentre. Por tanto, dada la infinidad de casos que puede encontrarse el coche, la supervisión humana, aunque sea remota, parece necesaria para evitar accidentes.
- La capacidad de cómputo y los sensores podrían no ser cuellos de botella importantes. Tanto Comma como Tesla evidencian esta idea, pero el trabajo ha servido para comprobar que con una cámara de muy baja calidad el *software* autónomo ha sido capaz de conducir el coche a velocidades superiores a las de un humano. Quizás, un enfoque que combine el uso de operarios con coches más livianos en cuanto a poder de cómputo y sensores podría ser interesante. No obstante, es necesario recordar que las condiciones de conducción han sido muy limitadas y la introducción de objetos en la calzada podría requerir de un aumento considerable en estos factores.

En cuanto a las líneas de mejora, estas son numerosas. El proyecto ha servido como punto de partida para conseguir un coche con una autonomía parcial. No obstante, partiendo de este punto se pueden realizar una serie de mejoras para conseguir una autonomía mayor. Algunas de ellas son las siguientes:

- Las ya mencionadas en este mismo capítulo: mejorar la cámara y fijarla al coche, mejorar la placa base, afinar el etiquetado de los datos y adaptar el modelo para que pueda ejecutarse por completo en el acelerador.
- Aprender a conducir con líneas de la calzada de varios colores.
- Ajustar la velocidad en función del giro del coche.
- Implementar un modelo de detección de objetos y ejecutarlo en el acelerador, para detectar señales de tráfico y objetos en la calzada.
- Diseñar una interfaz gráfica para que usuarios no familiarizados con la programación puedan utilizar más fácilmente el coche.

En conclusión, se dan por conseguidos los objetivos del trabajo, habiendo conseguido un pequeño coche robótico con una autonomía parcial. Además, si el proyecto llega a otros programadores, se considera que este puede facilitar la experimentación de pequeños desarrolladores con la conducción autónoma. Por último, se ha obtenido un aprendizaje útil con respecto a la conducción autónoma, que permitirá seguir avanzando en esta línea de trabajo con las ideas mencionadas en el párrafo anterior.

6. Referencias

1. Jin, Hyunjoo. Waymo offers driverless rides to San Francisco employees, expands in Phoenix. *Reuters*. [En línea] 2022. <https://www.reuters.com/business/autos-transportation/waymo-offers-driverless-rides-employees-san-francisco-expands-phoenix-2022-03-30/>.
2. Yellig, John. GM's Cruise Launches Driverless Taxi Service in San Francisco. *IoT World Today*. [En línea] 2022. <https://www.iotworldtoday.com/2022/02/17/gms-cruise-launches-driverless-taxi-service-in-san-francisco/>.
3. John, Darryn. Elon Musk says Tesla will have self-driving cars without the need for humans next year. *Drive Tesla Canada*. [En línea] 2022. <https://driveteslacanada.ca/news/elon-musk-tesla-self-driving-cars-without-humans-next-year/>.
4. Malik, Adam. Why experts say we should probably forget about autonomous vehicles. *Auto Service World*. [En línea] 2022. <https://www.autoserviceworld.com/why-we-should-probably-forget-about-autonomous-vehicles/>.
5. Musk, Elon. After wide release of FSD Beta 10.69.2, price of FSD will rise to \$15k in North America on September 5th. *Twitter*. [En línea] 2022. <https://twitter.com/elonmusk/status/1561362640261226499?s=20&t=mgX7BBL7Uo35BrEBJwk6ag>.
6. Paquette, Kim. FSD Bets 10.10.2 Tett Loop Drive 1 - not so good! *YouTube*. [En línea] 2022. <https://www.youtube.com/watch?v=JfBuMc5vbO4>.
7. Lee, Timothy B. This map perfectly explains why Waymo hasn't expanded faster. *Full Stack Economics*. [En línea] 2022. <https://fullstackeconomics.com/one-image-that-explains-the-slow-progress-of-self-driving-technology/>.
8. Moreno, Johan. Waymo CEO Says Tesla Is Not A Competitor, Gives Estimated Cost Of Autonomous Vehicles. *Forbes*. [En línea] 2021.
9. Hawkins, Andrew J. Waymo will allow more people to ride in its fully driverless vehicles in Phoenix. *The Verge*. [En línea] 2020. <https://www.theverge.com/2020/10/8/21507814/waymo-driverless-cars-allow-more-customers-phoenix>.
10. Waymo. Waymo Driver. *Waymo*. [En línea] <https://waymo.com/intl/es/waymo-driver/>.
11. *An Overview of Autonomous Vehicles Sensors and Their Vulnerability to Weather Conditions*. Vargas, Jorge, y otros. 2021, Sensors.
12. Shalev-Shwartz, Shai. Congrats to #FSD team for the great progress! *Twitter*. [En línea] 2022. https://twitter.com/shai_s_shwartz/status/1559013872945741825?s=20&t=GenWIVKgSvERaCUPKKHuw.
13. *Attention Is All You Need*. Vaswani, Ashish, y otros. 2017, NIPS.
14. Szeliski, Richard. *Computer vision: algorithms and applications*. s.l. : Springer Science & Business Media, 2010.
15. *Deep Learning for Computer Vision: A Brief Review*. Voulodimos, Athanasios, et al. 2018, Computational intelligence and Neuroscience.



16. *Language Models are Few-Shot Learners*. Brown, Tom B., y otros. 2020, NeurIPS.
17. McCarthy, J., y otros. A proposal for the Dartmouth research project on Artificial Intelligence. 1995.
18. *A logical calculus of the ideas immanent in nervous activity*. McCulloch, Warren S. y Pitts, Walter. s.l. : The bulletin of mathematical biophysics, 1943, Vol. 5.
19. Retamales, Jorge. Embeddings con Python. *Github*. [En línea] 2019. https://jretamales.github.io/2019-01-29-embeddings_101/.
20. *The Perceptron: a probabilistic model for information storage and perception in the brain*. Rosenblatt, F. s.l. : Psychological Review, 1958, Vol. 65.
21. *Recent Developments on Statistical and Neural Network Tools Focusing on Biodiesel Quality*. Brandes Marques, Delano, y otros. 2014, International Journal of Computer Science and Application.
22. AI Newsletter - January 2005. *AI Newsletter*. [En línea] 2005. https://www.ainewsletter.com/newsletters/aix_0501.htm#w.
23. *Learning representations by back-propagating errors*. Rumelhart, David E., Hinton, Geoffrey E. y Williams, Ronald J. 1986, Nature, Vol. 323.
24. *Gradient-Based Learning Applied to Document Recognition*. LeCun, Yann, y otros. 1998. Proceedings of the IEEE.
25. History of Data Science. ImageNet: A Pioneering Vision for Computers. *History of Data Science*. [En línea] 2021. <https://www.historyofdatascience.com/imagenet-a-pioneering-vision-for-computers/>.
26. Papers with code. Image Classification on ImageNet. *Papers with code*. [En línea] <https://paperswithcode.com/sota/image-classification-on-imagenet>.
27. Karpathy, Andrej. Deep Neural Nets: 33 years ago and 33 years from now. *Andrej Karpathy blog*. [En línea] 2022. <http://karpathy.github.io/2022/03/14/lecun1989/>.
28. Lima, Gabriel Camilo. The Growth of AI and Machine Learning in Computer Science Publications. *Medium*. [En línea] 2019. <https://medium.com/@thegcamilo/the-growth-of-ai-and-machine-learning-in-computer-science-publications-603d75467c38>.
29. *ImageNet Classification with Deep Convolutional Neural Networks*. Krizhevsky, Alex, Sutskever, Ilya y Hinton, Geoffrey E. 2012, NIPS.
30. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. Tan, Mingxing y Quoc V., Le. 2019, CVPR.
31. *Rethinking the Inception Architecture for Computer Vision*. Szegedy, Christian, y otros. 2016, CVPR.
32. *Going deeper with convolutions*. Szegedy, Christian y otros. 2015, CVPR.
33. *Deep Residual Learning for Image Recognition*. He, Kaiming, y otros. 2016, CVPR.
34. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Howard, Andrew G., y otros. 2017, CVPR.
35. *An image is worth 16x16 words: Transformers for image recognition at scale*. Dosovitskiy, Alexey, y otros. 2021, ICLR.

36. *A ConvNet for the 2020s*. Liu, Zhuang, y otros. 2020, CVPR.
37. OECD. Road accidents. *OECD*. [En línea] <https://data.oecd.org/transport/road-accidents.htm>.
38. Administration, National Highway Traffic Safety. *2016 Fatal Motor Vehicle Crashes: Overview*. 2017.
39. American Automobile Association. Advanced Driver Assistance Technology Names. [En línea] 2019. <https://www.aaa.com/AAA/common/AAR/files/ADAS-Technology-Names-Research-Report.pdf>.
40. SAE. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. 2014.
41. Zhou, Naaman. Volvo admits its self-driving cars are confused by kangaroos. *The Guardian*. [En línea] 2017. <https://www.theguardian.com/technology/2017/jul/01/volvo-admits-its-self-driving-cars-are-confused-by-kangaroos?>.
42. Cruise. Cruise: Technology. *Cruise*. [En línea] <https://www.getcruise.com/technology>.
43. Department of Motor Vehicles of California. Disengagement Reports, 2021. [En línea] 2022. <https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/disengagement-reports/>.
44. Newcomb, Doug. SAE International. *SAE International*. [En línea] 12 de 01 de 2022. [Citado el: 03 de 07 de 2022.] <https://www.sae.org/news/2022/01/lidar-sensors-at-ces-2022>.
45. Ahn, YooJung. Designing the 5th-generation Waymo Driver. *Waymo Blog*. [En línea] 03 de 2020. <https://blog.waymo.com/2020/03/designing-5th-generation-waymo-driver.html>.
46. Ogan, Taylor. Tesla FSD disengagement rates. *Twitter*. [En línea] 2022. <https://twitter.com/TaylorOgan/status/1501307605083295749/photo/1>.
47. Shashua, Amnon. *CES 2021: Amnon Shashua Shares that Mobileye's AVs are on the 'Go' Around the World*. Enero de 2021.
48. Resultados de búsqueda para "lidar". *Amazon*. [En línea] <https://www.amazon.es/s?k=lidar>.
49. Tesla. Tesla AI Day. *YouTube*. [En línea] 2021. <https://youtu.be/j0z4FweCy4M>.
50. *MultiPath: Multiple Probabilistic Anchor Trajectory*. Chai, Yuning, y otros. 2019, Robot Learning.
51. Waymo. Simulation City. *Waymo blog*. [En línea] 2021. <https://blog.waymo.com/2021/06/SimulationCity.html>.
52. Comma. Building a Super Human Driving Agent | COMMA_CON. *YouTube*. [En línea] 2021. <https://youtu.be/tZgfDVGXdmk>.
53. Tesla. Tesla Autonomy Day. *YouTube*. [En línea] 2019. <https://www.youtube.com/watch?v=Ucp0TTmvqOE>.
54. Comma. How Do We Control The Car? | COMMA_CON. *YouTube*. [En línea] 2021. <https://youtu.be/nNU6ipme878>.
55. —. openpilot driver monitor system outputs at night. *YouTube*. [En línea] 2021. <https://www.youtube.com/watch?v=VR9MsnOw7oE>.



56. SoyMotor - Coches. EL TESLA DE LOBATO - Cap 15: Autopilot y conducción autónoma, ¿estamos listos? | Coches SoyMotor.com. *YouTube*. [En línea] 2021. https://youtu.be/i02lmR_uUSY.
57. Zoot. Zoot teleguidance. [En línea] 2021. https://zoot.com/wp-content/uploads/zoot_teleguidance_web_hd.mp4.
58. Broadcom BCM2837B0 Specs. *Gadget Versus*. [En línea] <https://gadgetversus.com/processor/broadcom-bcm2837b0-specs/>.
59. EV Source. Tesla Model S lithium ion battery 18650. *EV Source*. [En línea] <https://evsource.com/products/tesla-model-s-lithium-ion-battery-18650-22-8-volt-5-3-kwh>.
60. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. Sandler, Mark, y otros. 2018, CVPR.
61. Keras. Usage examples for image classification models. *Keras API*. [En línea] <https://keras.io/api/applications/#usage-examples-for-image-classification-models>.
62. *Adam: a method for stochastic optimization*. Kingma, Diederik P. y Lei Ba, Jimmy. 2015, ICLR.
63. *End to End Learning for Self-Driving Cars*. Bojarski, Mariusz, y otros. 2016.
64. *On large-batch training for deep learning: generalization gap and sharp minima*. Keskar, Nitish Shrivastava, y otros. 2017, ICLR.
65. *Feature optimization of contact map predictions based on inter-residue distances and U-Net++ architecture*. Shenoy, Aditi. Estocolmo : Universidad de Estocolmo, 2019.

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

| Objetivos de Desarrollo Sostenibles | Alto | Medio | Bajo | No Procede |
|---|------|-------|------|------------|
| ODS 1. Fin de la pobreza. | | | X | |
| ODS 2. Hambre cero. | | | X | |
| ODS 3. Salud y bienestar. | | X | | |
| ODS 4. Educación de calidad. | | | | X |
| ODS 5. Igualdad de género. | | | X | |
| ODS 6. Agua limpia y saneamiento. | | | | X |
| ODS 7. Energía asequible y no contaminante. | | | X | |
| ODS 8. Trabajo decente y crecimiento económico. | | X | | |
| ODS 9. Industria, innovación e infraestructuras. | X | | | |
| ODS 10. Reducción de las desigualdades. | | | X | |
| ODS 11. Ciudades y comunidades sostenibles. | | | X | |
| ODS 12. Producción y consumo responsables. | | | X | |
| ODS 13. Acción por el clima. | | | X | |
| ODS 14. Vida submarina. | | | | X |
| ODS 15. Vida de ecosistemas terrestres. | | | | X |
| ODS 16. Paz, justicia e instituciones sólidas. | | | | X |
| ODS 17. Alianzas para lograr objetivos. | | | | X |

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El presente Trabajo Fin de Grado está relacionado de forma indirecta con la mayoría de los Objetivos de Desarrollo Sostenible. La construcción de un coche robótico capaz de conducirse de forma autónoma es un pequeño proyecto que avanza en la dirección de conseguir vehículos autónomos. A escala real, los vehículos autónomos supondrían una serie de avances sociales que podrían ir en la línea de los objetivos planteados por la ONU, por las razones que se exponen a continuación.

En primer lugar, como se menciona en el capítulo 2 del trabajo, en Estados Unidos murieron durante el año 2019 un total de 107 personas por accidentes de tráfico, según la NHTSA. De estos accidentes, se estima que entre un 94 y un 96% son causados por error humano. Por tanto, la conducción autónoma tiene el potencial de reducir considerablemente esta cifra, ayudando a la consecución del objetivo 3: **salud y bienestar**.

Además de las vidas salvadas, la conducción autónoma supondría avances en distintos frentes. Existen una serie de trabajos considerados popularmente como precarios o de riesgo que implican conducir: camionero, taxista, repartidor, etc. La conducción autónoma tiene el potencial para suplir la necesidad de realizar estos trabajos, abriendo oportunidades laborales en áreas como el desarrollo de software, hardware y sensores. Estos trabajos son más cualificados y gozan habitualmente de mejores condiciones laborales. Así pues, esto supone una clara relación con el ODS 8: **trabajo decente y crecimiento económico**.

Por otra parte, la automatización de estos puestos de trabajo supondría un abaratamiento significativo del transporte por carretera, que tendría consecuencias positivas en muchos ámbitos de la vida cotidiana. La reducción en el coste del transporte de mercancías sería especialmente notable en los productos perecederos, dado que se transportan fundamentalmente por carretera. Por tanto, el precio de algunos alimentos y recursos básicos podría verse reducido, contribuyendo potencialmente a los objetivos 1, 2 y 10: **fin de la pobreza, hambre 0 y reducción de las desigualdades**.

En lo que respecta a la sostenibilidad y el uso eficiente de los recursos, la conducción autónoma también presenta una serie de ventajas. Al no necesitar de un conductor, el tiempo de uso de cada vehículo podría ser más alto, disminuyendo el número de coches por habitante necesarios. De la misma forma, las redes de taxis autónomos podrían contribuir a una menor compra de coches. Esto, a su vez, repercutiría en una menor necesidad de espacios de aparcamiento y en una aceleración de la transición hacia el coche eléctrico. Las consecuencias de esto alcanzan varios de los ODS:

- ODS 7: **energía asequible y no contaminante**
- ODS 11: **ciudades y comunidades sostenibles**
- ODS 12: **producción y consumo responsables**



- ODS 13: **acción por el clima**

En cuanto al objetivo 5: **igualdad de género**, Arabia Saudita dejó de ser en 2018 el único país que prohibía conducir a las mujeres. No obstante, aún es considerado tabú en algunos países que una mujer conduzca. Esto, sumado al riesgo al que se enfrentan algunas mujeres al subirse a un coche de un desconocido o al transporte público, hace que el transporte aún suponga hoy en día una fuente de discriminación. La conducción autónoma, al no requerir de ninguna persona al volante, permite que cualquier mujer pueda acceder al transporte de forma segura con la vigilancia de una empresa desde una central.

Por último, se considera que el objetivo 9: **industria, innovación e infraestructura** es el que tiene una relación más directa con el proyecto. Por un lado, porque este profundiza en áreas de investigación muy relacionadas con la innovación, como son la inteligencia artificial y la robótica. Y, por otro, por que se llevan a la práctica estas áreas para que cualquier pequeño desarrollador pueda contribuir a la innovación en las mismas. Además, el mundo del automóvil tiene una estrecha relación con la industria y la infraestructura, necesitando de ambas para su utilidad.