



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
INFORMÁTICA

MÁSTER UNIVERSITARIO EN VISIÓN ARTIFICIAL

TRABAJO FIN DE MÁSTER

Conducción autónoma de un vehículo en un
simulador mediante aprendizaje extremo a
extremo basado en visión

Autor: Vanessa Fernández Martínez

Tutor: José María Cañas Plaza

Cotutor: Francisco Miguel Rivas Montero

Curso académico 2018/2019

Agradecimientos

Resumen

Índice general

Índice de figuras	VII
Índice de tablas	VIII
1. Introducción	2
1.1. Contexto y motivación	2
1.2. Visión artificial	4
1.3. Conducción autónoma	6
1.4. Redes neuronales artificiales	10
1.4.1. Redes neuronales convolucionales	11
1.4.2. Redes neuronales recurrentes	12
1.4.3. Tipos de capas	13
1.4.3.1. Capa Convolucional	13
1.4.3.2. Capa de <i>Pooling</i>	15
1.4.3.3. Capa <i>Fully connected</i>	15
1.4.3.4. Capa LSTM	16
2. Objetivos	17
2.1. Objetivos	17
2.2. Requisitos	18
2.3. Metodología	19
2.4. Plan de trabajo	21
3. Estado del arte	22
3.1. Bases de datos para conducción autónoma	22
3.1.1. Comma.ai	23
3.1.2. Udacity	23
3.1.3. SAIC Dataset	24
3.2. Simuladores	24
3.2.1. CARLA	25

3.2.2.	Gazebo	26
3.2.3.	Udacity's Self-Driving Car Simulator	27
3.2.4.	Deepdrive 2.0	28
3.3.	Redes neuronales	28
3.3.1.	Redes neuronales convolucionales	29
3.3.2.	Redes neuronales recurrentes	35
3.4.	Infraestructura empleada	42
3.4.1.	Simulador Gazebo	42
3.4.2.	Entorno JdeRobot	43
3.4.3.	Entorno ROS	45
3.4.4.	Python	47
3.4.5.	Biblioteca OpenCV	48
3.4.6.	PyQt	50
3.4.7.	Keras framework	51
3.4.7.1.	Modelos	52
3.4.7.2.	Capas	54
3.4.7.3.	<i>Callbacks</i>	58
3.4.8.	Formato de archivo HDF5	58
4.	Infraestructura desarrollada	60
4.1.	Objetivo de <i>Follow line</i>	60
4.2.	Modelo de coche	61
4.3.	Modelos de circuitos	63
4.4.	Mundo de Gazebo	66
4.5.	Piloto manual	69
4.6.	Creación del conjunto de datos	77
4.7.	Nodo piloto	84
4.7.1.	Interfaz gráfica	88
4.7.2.	Ejecución típica	91
4.7.3.	Tiempo de ejecución	93
5.	Redes de clasificación	95
6.	Redes de regresión	96

7. Conclusiones	97
Bibliografía	109

Índice de figuras

1.1.	Diagrama de Venn que muestra los campos que engloba la IA	3
1.2.	Navegación en robótica mediante VA	5
1.3.	Detección de cáncer de mama	5
1.4.	Detección de contenedores	6
1.5.	Conducción autónoma	6
1.6.	Comparación de neurona biológica (izquierda) y neurona artificial (derecha).	10
1.7.	Estructura de CNN	12
1.8.	Esquema de Redes Neuronales Recurrentes (RNN)	13
1.9.	Ejemplo de operación de convolución	14
1.10.	Ejemplo de capa <i>max pooling</i>	15
1.11.	Unidad LSTM	16
2.1.	Modelo en espiral	20
3.1.	Simulador CARLA	26
3.2.	Simulador Gazebo.	27
3.3.	Simulador Udacity's Self-Driving Car Simulator.	28
3.4.	Simulador Deepdrive.	29
3.5.	Arquitectura Pilotnet.	30
3.6.	Ejemplos de objetos salientes para varias imágenes de entrada.	32
3.7.	Arquitectura TinyPilotnet.	33
3.8.	Estructura de red ControlNet.	37
3.9.	Arquitectura C-LSTM.	38
3.10.	Arquitectura DeepestLSTM-TinyPilotnet.	41
3.11.	Simulador Gazebo.	42
3.12.	Ejemplo de componentes JdeRobot	44
3.13.	Interfaz del conjunto de paquetes gazebo_ros_pkgs	46
3.14.	Funciones de OpenCV	49
3.15.	Función de activación <i>ReLU</i>	57
4.1.	Modelo f1ROS	61

4.2. Modelo pistaSimple	64
4.3. Modelo monacoLine	64
4.4. Modelo nurburgrinLine	65
4.5. Modelo curveGP	65
4.6. Modelo pista_simple	66
4.7. Filtrado de color	72
4.8. Filtrado de color con cierre	73
4.9. Representación pares L1-L2 (<i>Dataset1</i> contra conducción)	80
4.10. Representación pares L1-L2 (nuevo <i>Dataset</i> contra conducción)	81
4.11. Representación pares L1-L2 (<i>Dataset_Curves</i> contra conducción)	82
4.12. Análisis de pares L1-L2 (<i>Dataset</i>) para w	83
4.13. Análisis de pares L1-L2 (<i>Dataset</i>) para v	83
4.14. Estructura del Nodo Piloto	85
4.15. Interfaz gráfica (GUI)	91
4.16. Inicialización de ROS y Gazebo	92
4.17. Inicialización del nodo Piloto y la GUI	93

Índice de tablas

4.1. Resultados del Pilotaje manual	76
---	----

Acrónimos

API Application Programming Interface.

CNN Redes Neuronales Convolucionales.

CSAIL MIT Computer Science & Artificial Intelligence Lab.

GPS Global Positioning System.

GUI Graphical User Interface.

HDF5 Hierarchical Data Format version 5.

IA Inteligencia Artificial.

ICE Internet Communications Engine.

LSTM Long Short-Term Memory.

MAE Mean Absolute Error.

MSE Mean Squared Error.

RNA Redes Neuronales Artificiales.

RNN Redes Neuronales Recurrentes.

ROS Robot Operating System.

RPC Remote Procedure Call.

SDF Simulation Description Format.

SVG Scalable Vector Graphics.

VA Visión Artificial.

XML Extensible Markup Language.

Capítulo 1

Introducción

En este capítulo se definirá el contexto en el cual se sitúa este proyecto, y la motivación principal que ha llevado a su desarrollo. Se explicará de forma general qué es la visión artificial, así como el uso de redes neuronales en la misma. Además, se expondrá qué es la conducción autónoma.

1.1. Contexto y motivación

Desde la antigüedad el ser humano ha soñado con crear máquinas capaces de pensar. Cuando surgieron los primeros ordenadores programables, las personas se plantearon la idea de lograr que estos computadores adquirieran inteligencia, adquiriendo capacidades empleadas para realizar tareas propias de los humanos. Algunos ejemplos de estas tareas son entender el habla o las imágenes, y automatizar tareas rutinarias. El campo que desarrolla estas tareas se denomina Inteligencia Artificial (IA) [1] y cada vez tiene más presencia en temas de investigación.

En IA existen diversos desafíos muy interesantes; sin embargo, en la mayoría de ellos es extremadamente difícil alcanzar el rendimiento y la eficiencia del cerebro humano. Las máquinas nos superan en tareas como procesamiento de gran cantidad de datos, almacenamiento de información o tareas de razonamiento como el juego de ajedrez. Sin embargo, algunas habilidades que el ser humano realiza inconscientemente, como caminar o ver, son muy complejas para las máquinas; y por ello el cerebro humano supera a la máquina en este tipo de tareas.

CAPÍTULO 1. INTRODUCCIÓN

La IA comprende diferentes campos (Figura 1.1): *Machine Learning*, *Knowledge Engineering*, Lingüística computacional, Redes Neuronales Artificiales (RNA) [2], Procesamiento del lenguaje natural, Minería de datos, Visión Artificial (VA), etc. Este proyecto se enfoca en la VA, que trata de analizar y procesar imágenes de tal forma que un ordenador sea capaz de interpretar dichas imágenes. La IA intenta conseguir que una máquina realice el mismo proceso que el Sistema Visual Humano de tal forma que sea capaz de tomar decisiones y actuar en función de la situación en que se encuentre.

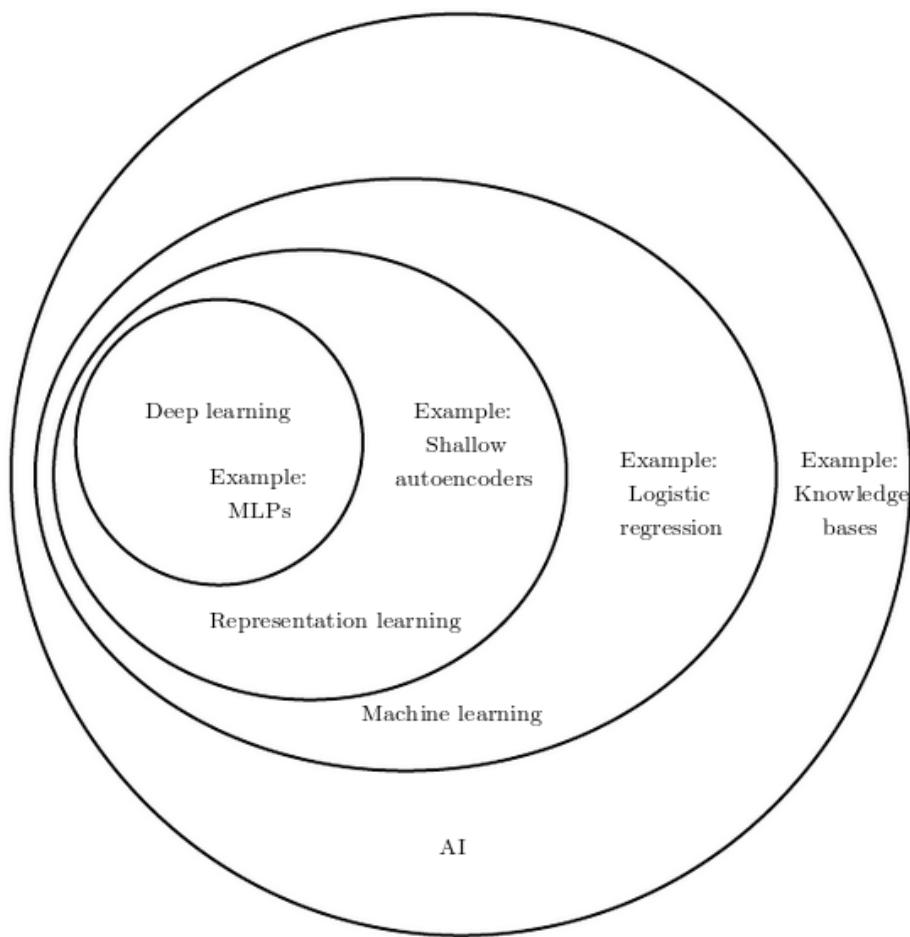


Figura 1.1: Diagrama de Venn que muestra los campos que engloba la IA

El aprendizaje de las máquinas es un punto de encuentro de diferentes disciplinas que engloba a la estadística, la geometría, la programación y la optimización, entre otras. La VA intenta simular las capacidades del ojo y el cerebro humano, empleando los conceptos de estas disciplinas.

Uno de los problemas que se está estudiando ampliamente en VA en la última década es la conducción autónoma. Los humanos somos capaces de mirar a la carretera y saber al instante si el coche que conducimos está en una curva o una recta, si hay coches alrededor y cómo interactúan entre ellos. En función a la situación en la que nos encontramos sabemos qué acciones llevar a cabo para lograr una buena conducción. Sin embargo, este procedimiento es más complicado para los ordenadores. En la actualidad se está investigando ampliamente cómo emplear las Redes Neuronales Artificiales (RNA) para predecir comportamiento autónomo en vehículos.

El objetivo principal de este proyecto es el estudio de conducción autónoma en simulación mediante Redes Neuronales Artificiales (RNA). En el estudio se incluyen diferentes arquitecturas de redes neuronales empleadas para imitar el comportamiento humano en un vehículo. Además, se presentarán los resultados de las distintas redes neuronales.

1.2. Visión artificial

Las primeras aplicaciones de la Visión Artificial datan de los años 60, donde destaca la creación del perceptrón, desarrollado por Frank Rosenblatt. Se considera la primera neurona artificial con capacidades de aprendizaje. Era capaz de distinguir figuras simples, como triángulos y cuadrados, a base de ensayo y error.

En la década de los 70 surgen las primeras aplicaciones comerciales de la VA, como por ejemplo el reconocimiento óptico de caracteres (OCR). Sin embargo, no es hasta los años 80 y 90 cuando la VA toma mayor peso. En la actualidad es una parte muy importante que contribuye a la transformación digital de diversos sectores. Por este motivo sólo es posible dar una pequeña pincelada sobre las múltiples aplicaciones en las que se ha aplicado hasta el momento.

Un claro ejemplo, es la navegación en robótica (Figura 1.2), donde la visión constituye una capacidad sensorial más para la percepción del entorno que rodea al robot. Generalmente se recurre a técnicas de visión estereoscópica con el fin de reconstruir la escena 3D. En algunas ocasiones se añade algún módulo de reconocimiento con el fin de identificar

CAPÍTULO 1. INTRODUCCIÓN

la presencia de determinados objetos, hacia los que debe dirigirse o evitar. Cualquier información que pueda extraerse mediante VA supone una gran ayuda para el movimiento del robot.



Figura 1.2: Navegación en robótica mediante VA

Otro ejemplo donde la VA supone un gran avance es en la comunidad médica, donde permite diagnosticar con mayor rapidez y detalle enfermedades y lesiones. De esta forma es posible aplicar tratamientos personalizados y eficaces en menor tiempo. Un claro ejemplo de investigadores que emplean VA es el MIT Computer Science & Artificial Intelligence Lab (CSAIL) [3], donde el desarrollo de algoritmos que analizan mamografías de una forma novedosa permite ayudar a detectar el cáncer de mama (Figura 1.3) con hasta cinco años de anticipación.

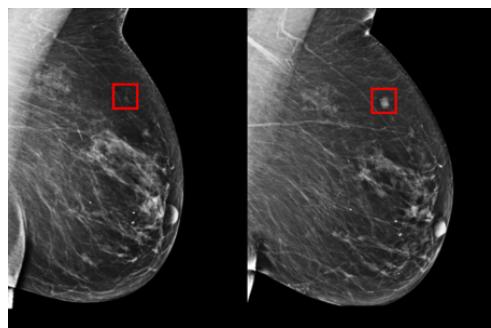


Figura 1.3: Detección de cáncer de mama

Una posible aplicación es el mantenimiento e inventariado urbano. Es posible identificar problemas en instalaciones y mobiliario urbano (averías, mal estado de

contenedores (Figura 1.4), socavones en la vía pública, etc) mediante cámaras ubicadas por ejemplo en autobuses. Los mantenimientos de infraestructuras de transporte, como vías y cables ferroviarios, pueden programarse automáticamente implantando sistemas de VA en los propios trenes.



Figura 1.4: Detección de contenedores

La reducción de accidentes gracias a vehículos autónomos es una realidad gracias a la VA, ya que los sistemas de guiado que poseen estos vehículos están basados en esta visión. Algunos ejemplos de estos sistemas (Figura 1.5) son: los sistemas de aviso de cambio de carril, o de control de velocidad de crucero.

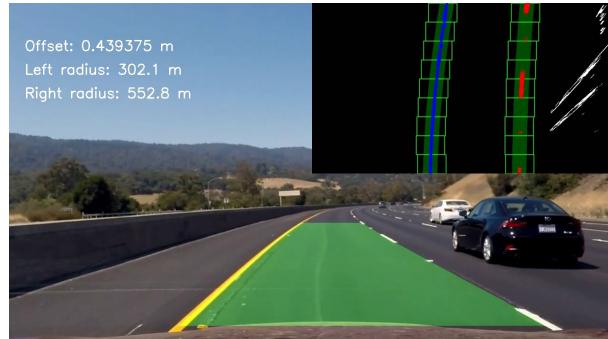


Figura 1.5: Conducción autónoma

1.3. Conducción autónoma

La conducción autónoma pretende que un vehículo sea capaz de conducir sólo en base a los datos proporcionados por determinados sensores (cámaras, LIDAR, etc), es decir, es capaz de aprender las normas de circulación. La posibilidad de crear un sistema capaz de conducir un vehículo ya se había contemplado en el siglo pasado. Sin embargo, la tecnología disponible en ese momento no permitía resolver una tarea tan compleja. Después

CAPÍTULO 1. INTRODUCCIÓN

de los avances tecnológicos cambió este hecho.

A finales del siglo pasado, algunos investigadores [4] [5] experimentaron con la creación de las primeras arquitecturas de conducción autónoma, desarrollando y probando algunos prototipos que podían conducir en calles reales. Estas pruebas se realizaron en áreas controladas y protegidas, y la conducción no fue lo suficientemente buena como para crear un producto de uso seguro. Estos experimentos dejaron claro que aún quedaba mucho para obtener una solución, pero al mismo tiempo, demostraron que la conducción autónoma podría convertirse en una perspectiva real.

En los últimos años se ha hecho mayor incapié en la investigación de la conducción autónoma con el fin de solventar el incremento de la tasa de muerte por accidentes de tráfico. Aunque algunos de estos accidentes se producen por fallos mecánicos del vehículo, la mayoría de dichos accidentes se debe a imprudencias y distracciones humanas. La conducción autónoma eliminaría estas distracciones haciendo posible la disminución de accidentes.

Hoy en día, cada vez existen más fabricantes de vehículos que incorporan tecnologías de conducción autónoma. Existe un estándar elaborado por la Sociedad de Ingenieros Automotrices (SAE), conocido como J3016 [6], que establece los niveles de conducción autónoma según la capacidad del vehículo.

- Nivel 0: No hay automatización de la conducción. Las tareas de conducción son realizadas en su totalidad por el conductor.
- Nivel 1: Asistencia al conductor. El vehículo posee algún sistema de automatización de la conducción (control de crucero, autoaparcamiento), ya sea para el control de movimiento longitudinal o el movimiento lateral, aunque no ambas cosas al mismo tiempo. El conductor realiza el resto de tareas de conducción, por lo que debe estar siempre atento.
- Nivel 2: Automatización parcial. Considera que el conductor ya no tiene que conducir en todo momento y que el coche empieza a ser realmente autónomo, aunque con ciertos matices. El vehículo es capaz de actuar de manera independiente dentro de

CAPÍTULO 1. INTRODUCCIÓN

escenarios controlados y en situaciones específicas de conducción. El conductor debe seguir prestada atención a lo que ocurre a su alrededor para evitar posibles riesgos. Un buen ejemplo de Nivel 2 de conducción autónoma pueden ser los modelos BMW Serie 7 o el Mercedes Clase E, capaces de moverse solos durante un tiempo o con el sistema de asistente de atascos.

- Nivel 3: Automatización condicional. En este nivel, el coche comienza a interactuar con el entorno que le rodea y es capaz de analizar posibles riesgos externos con el fin de evitarlos. Ya no se habla de conductor sino que hablamos de un usuario preparado para intervenir, es decir, el coche ya conduce completamente solo y el conductor es un simple vigilante de que todo funcione correctamente. El coche está preparado para ser conducido de manera habitual en cualquier momento.
- Nivel 4: Alta autonomía. En este nivel el sistema cuenta tanto con los sistemas de automatización presentes en el anterior nivel, como con sistemas de detección de objetos y eventos. Además, es capaz de responder ante ellos. El sistema de automatización de la conducción tiene un sistema de respaldo para actuar en caso de fallo del sistema principal y poder conducir hasta una situación de riesgo mínimo. En algunas situaciones es posible que el vehículo no siga conduciendo.
- Nivel 5: Autonomía total. Este nivel cuenta con todos los beneficios del sistema de automatización del nivel 4. Sin embargo, la diferencia es que en este caso el vehículo podría seguir conduciendo en todo momento o circunstancia.

Ejemplos importantes de conducción autónoma son: el DARPA Grand Challenge y el Urban Challenge. El DARPA Grand Challenge, organizado en 2004 y 2005 en Estados Unidos, fue una carrera de vehículos autónomos que debían recorrer 120 kms por el desierto de Nevada sin intervención humana y disponiendo únicamente de un listado de puntos intermedios entre el principio del circuito y el final. El Urban Challenge, organizado en 2007, fue una carrera de vehículos autónomos por zona urbana en la que debían recorrer 96 km en menos de 6 horas.

Como resultado de estos desafíos, destaca el proyecto ganador de 2005 de la Universidad de Stanford, cuyos miembros liderados por Sebastian Thrun acabaron desarrollando el vehículo autónomo de Google. En 2014, Google reveló un nuevo prototipo de su automóvil sin conductor (Firefly), que no tenía volante, pedal de acelerador o freno, siendo

CAPÍTULO 1. INTRODUCCIÓN

100 % autónomo, aunque era un prototipo empleado exclusivamente para pruebas. En la actualidad este vehículo se conoce como Waymo.

Este año, BMW y Mercedes-Benz han decidido unir fuerzas para desarrollar coches autónomos. Estas dos grandes compañías desarrollarán tecnologías para la creación de los próximos vehículos autónomos. Pretenden desarrollar ayudas a la conducción avanzadas y sistemas que automaticen la conducción en autopista y en el aparcamiento. Su objetivo es crear sistemas de conducción autónoma de nivel 4.

Tesla incluye el sistema inteligente Autopilot que alcanza el nivel de conducción 3. Sin embargo, Elon Musk ha anunciado este año que en 2020 será posible hablar de coches completamente autónomos (niveles 4 y 5), ya que su sistema Autopilot ofrecerá una conducción 100 % autónoma, donde el conductor pasaría a ser un mero espectador durante la conducción.

En la actualidad se están desarrollando sistemas que toman decisiones empleando una red neuronal profunda, la cual recibe información del entorno mediante diferentes sensores (LIDAR, radar, cámaras, etc.). A partir de los datos recogidos por los sensores la red predice unos valores de salida que serán los empleados para la conducción. Las decisiones tomadas por la red neuronal están determinadas por los datos empleados durante el entrenamiento de la red. Por lo tanto, cuanto más representativo sea el conjunto de datos, mejor rendimiento se espera que tenga la red, ya que conocerá todas las situaciones posibles en las que puede estar el vehículo.

Hoy en día el principal obstáculo para la conducción autónoma no se deriva de las limitaciones de la tecnología, sino de factores políticos, jurídicos, de regulación, de infraestructura y de responsabilidad que se deben abordar. A pesar de estas dificultades la investigación ha hecho muchos avances.

1.4. Redes neuronales artificiales

Una Red Neuronal Artificial RNA es un modelo matemático inspirado en el comportamiento biológico de las neuronas y en cómo se organizan dichas neuronas en el cerebro. Estas redes intentan imitar ciertas características propias de los seres humanos, como pueden ser la capacidad de memorizar y de asociar hechos. Estas neuronas siguen la misma estructura jerárquica que el cerebro humano, es decir, las diversas neuronas se organizan por capas.

Una comparativa entre las neuronas biológicas y las neuronas artificiales se puede observar en la Figura 1.6. Las neuronas biológicas constan de un cuerpo celular o soma que contiene un núcleo y ramas denominadas dendritas. Las dendritas transfieren la información de las células próximas mediante sinapsis al soma. Además, poseen un axón que lleva el impulso nervioso del soma a otras neuronas. Sin embargo, la neurona artificial es un modelo simplificado de las neuronas biológicas. Las sinapsis y las dendritas de la neurona artificial son las entradas al elemento procesador (soma). Cada una de estas entradas posee un peso asociado de conexión. Cada una de estas entradas es multiplicada por su peso y se suman generalmente estos productos, que pasan entonces a la función de la transferencia para generar un resultado que se transmita por la salida (axón).

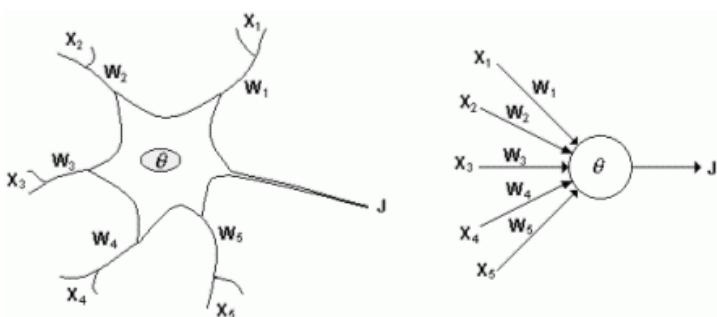


Figura 1.6: Comparación de neurona biológica (izquierda) y neurona artificial (derecha).

En este proyecto emplearemos la red multicapa, que consta de dos o más capas de neuronas interconectadas. Cada una de las capas puede hacer un tipo de transformación en su entrada, donde las señales atraviesan todas las capas. Cuando existen más de dos capas, hablamos de que la red posee capas ocultas. En esta red normalmente las capas iniciales

realizan generalizaciones simples, y en capas más profundas se hacen las generalizaciones más complejas.

La característica más especial del aprendizaje con redes neuronales es la capacidad de aprender y generalizar gracias a una base de datos específica para el problema que se debe tratar. Una vez esta red es entrenada, es capaz de estimar resultados para ejemplos que no ha visto anteriormente.

En este proyecto se emplearán dos tipos de redes neuronales para resolver el mismo problema. Por un lado se utilizan redes neuronales convolucionales y redes neuronales recurrentes.

1.4.1. Redes neuronales convolucionales

Las Redes Neuronales Convolucionales (CNN) son una clase de red neuronal artificial profunda que se emplean principalmente para clasificar imágenes, agrupar estas imágenes por similitud y realizar el reconocimiento de objetos dentro de las escenas. Este tipo de redes pueden identificar rostros, individuos, letreros de calles, tumores y muchos otros aspectos de los datos visuales.

Las CNN se basan en la arquitectura de la corteza visual del cerebro humano. Las CNN aplican una serie de filtros a los datos para extraer y aprender características de nivel superior, que el modelo puede usar para la clasificación, el reconocimiento u otro tipo de tarea. Las CNN están formadas por diferentes tipos de capas que veremos en las próximas subsecciones: capas convolucionales, capas de agrupación o *pooling*, y capas completamente conectadas o *fully connected*.

Las CNN siguen el esquema de la Figura 1.7 [7]. Normalmente este tipo de redes está formado por un conjunto de módulos convolucionales, que consisten en una capa convolucional seguida de una capa *pooling*. La capa convolución realiza una operación de convolución, mientras que la capa de agrupación o *pooling* genera características invariantes calculando estadísticas de las activaciones de convolución a partir de un campo receptivo (un pequeño campo de la capa anterior). En este tipo de redes, cada neurona de una capa oculta se conecta al campo receptivo local. En la capa convolucional, las

neuronas se distribuyen en diversas capas paralelas, denominadas mapas de características. En un mapa de características cada neurona está conectada a un campo receptivo local. Además, para cada mapa de características todas las neuronas comparten el mismo parámetro de peso conocido como *kernel* o filtro.

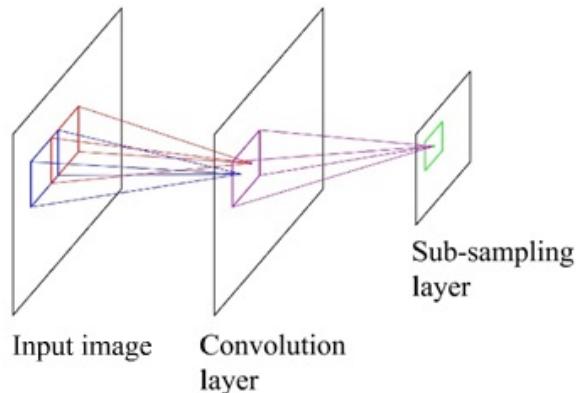


Figura 1.7: Estructura de CNN

Además, en este tipo de redes es importante tener en cuenta tanto las dimensiones de entrada como las dimensiones de las distintas capas. Las imágenes de entrada normalmente tienen dimensiones $W \times H \times C$, donde W es el ancho de la imagen, H es la altura, y C es el número de canales de la imagen. Cuando la información va atravesando las capas, normalmente se disminuye los valores $W \times H$, mientras que la profundidad de la red aumenta. Las primeras capas proporcionan una información localizada, es decir, el donde; mientras que las capas finales proporcionan información acerca del contenido de la imagen, es decir, el qué.

1.4.2. Redes neuronales recurrentes

Las Redes Neuronales Recurrentes (RNN) son un tipo de red neuronal artificial, donde la idea es usar información secuencial en vez de información independiente como en las redes tradicionales. En algunos casos emplear información independiente es mala idea, como puede ser en la predicción de la siguiente palabra en una cadena de texto, ya que sin información previa la red no es capaz de predecir la palabra.

Las RNN permiten que la información previa al instante actual persista. Una red neuronal recurrente se puede considerar como copias múltiples de la misma red, cada una de las cuales pasa un mensaje a su sucesor. En la Figura 1.8 se puede ver un esquema de RNN.

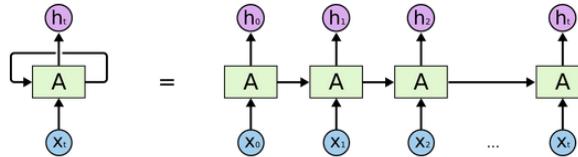


Figura 1.8: Esquema de Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes (RNN) aprenden a emplear la información pasada en los casos donde la brecha entre la información relevante y la información actual es pequeña. Pero habrá casos donde necesitemos más contexto, como por ejemplo si queremos predecir la última palabra del texto “Crecí en Francia... Hablo francés con fluidez”. La información reciente sugiere que la siguiente palabra es un idioma, pero si queremos concretar qué idioma es, necesitamos el contexto desde más atrás. Sin embargo, a medida que aumenta la brecha, las RNN no son capaces de aprender a conectar la información. En cambio las LSTM no tienen ese problema.

Las redes Long Short-Term Memory (LSTM) [8] son un tipo especial de RNN capaz de aprender dependencias a largo plazo. Esta clase de redes fueron diseñadas para recordar información de períodos de tiempo largo. Se explicará más acerca de este tipo de red en las siguientes subsecciones.

1.4.3. Tipos de capas

En las siguientes subsecciones se explican los diferentes tipos de capas empleadas en las redes CNN y LSTM.

1.4.3.1. Capa Convolucional

Las capas convolucionales son las más importantes de una CNN. La operación de convolución (Figura 1.9) recibe como entrada una imagen y luego aplica sobre ella un filtro o

kernel que devuelve un mapa de características. Con esta operación se reduce el tamaño de los parámetros. En las capas convolucionales existen diferentes parámetros a tener en cuenta:

- Dimensiones de los filtros de convolución. Suelen ser una matriz cuadrada (tamaño $M \times M$). Cada píxel de cada mapa de características solamente tendrá en cuenta los píxeles que estén dentro del filtro.
- Número de filtros de convolución. Determina la profundidad del volumen de salida. Cada filtro genera un mapa de características.
- *Stride*. Determina cuánto vamos a deslizar el filtro sobre la matriz de entrada. Por ejemplo, cuando el stride es 1 se mueven los filtros 1 píxel a la vez.
- *Padding*. Añade alrededor de la matriz de entrada ceros para evitar perder dimensiones tras la convolución.

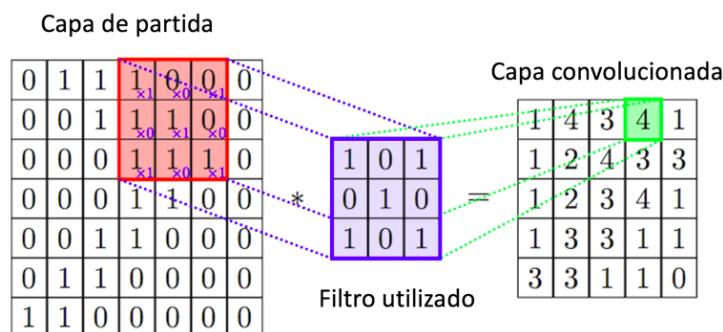


Figura 1.9: Ejemplo de operación de convolución

Se puede calcular el tamaño del volumen de salida en función al volumen de entrada (W), el tamaño del filtro de convolución (M), el stride aplicado (S) y la cantidad de zero-padding que se aplica (P). El tamaño del volumen de salida se calcula como: $(W - M + 2P) / (S+1)$.

Tras aplicar la convolución, se aplica una función de activación a los mapas de características. Esta función de activación es no lineal para conseguir modelos no lineales. La función de activación más usada es la función ReLU.

1.4.3.2. Capa de *Pooling*

La capa de *pooling* o de agrupación se coloca normalmente detrás de la capa convolucional. Se emplea para reducir las dimensiones espaciales (ancho x alto) del volumen de entrada, pero no afecta a la dimensión de profundidad del volumen.

En ocasiones la operación que realiza la capa de *pooling* se denomina reducción de muestreo debido a que la reducción de tamaño lleva a pérdidas de información. Aunque esta pérdida puede ser buena para la red por dos motivos: (1) trabaja en reducir el sobreajuste, (2) la disminución del tamaño produce un menor consumo de memoria durante el entrenamiento de las redes.

El funcionamiento de esta capa se basa en una ventana deslizante que actúa sobre el volumen de entrada. La operación realizada por esta ventana deslizante depende del tipo de *pooling* elegido. Las clases de submuestreo más empleadas son:

- *Max pooling*: Se queda con el valor máximo de los valores de la ventana deslizante. Se puede ver un ejemplo en la Figura 1.10
- *Average pooling*: Calcula cada píxel del volumen de salida realizando el promedio de los píxeles que se encuentran dentro de la ventana deslizante del volumen de entrada. Esta operación se hace canal por canal.

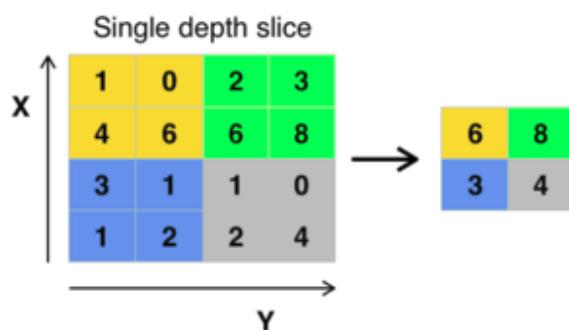


Figura 1.10: Ejemplo de capa *max pooling*

1.4.3.3. Capa *Fully connected*

Las capas completamente conectadas o *fully connected* conectan cada neurona de la capa de entrada con cada neurona de la capa de salida. Además, asignan un determinado

peso a cada conexión. La gran cantidad de conexiones produce que exista un gran número de parámetros configurables en esta capa.

1.4.3.4. Capa LSTM

La unidad LSTM puede añadir o quitar información, lo cual lo hace mediante estructuras denominadas puertas. Estas puertas son como una especie de camino para dejar pasar información. Una unidad LSTM tiene tres puertas.

- *Forget gate*. Decide qué información debe desechar.
- *Input gate*. Esta capa decide qué valores se deben actualizar.
- La unidad produce un *output* o valor de salida.

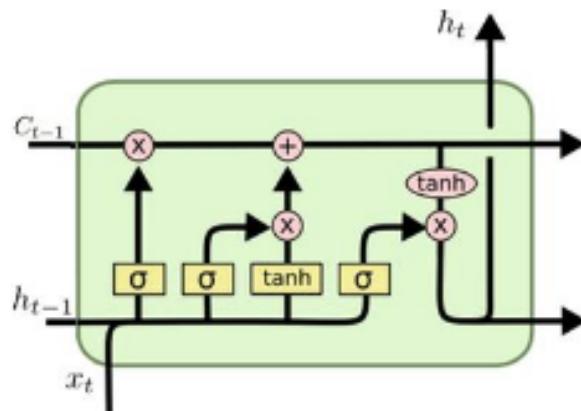


Figura 1.11: Unidad LSTM

Capítulo 2

Objetivos

Una vez explicado el contexto de este proyecto, se describirán en este capítulo los objetivos, los requisitos y la metodología empleados.

2.1. Objetivos

El propósito principal de este proyecto es el estudio de diferentes redes neuronales basadas en información visual que permitan a un vehículo ser capaz de conducir de forma autónoma.

El coche deberá ser capaz de conducir en diferentes circuitos en el Simulador Gazebo. Los entornos serán diferentes para que el vehículo sea capaz de aprender diferentes estímulos que le permitan conducir en situaciones complicadas.

Se creará un componente académico con la infraestructura necesaria que se comunica con el Simulador Gazebo, donde se podrá ver el resultado de la predicción de las redes neuronales. Para elaborar este componente se ha seguido la estructura de Robotics-Academy¹ y de dl-objectdetector². Este componente tendrá diferentes ingredientes:

- Infraestructura en el simulador.
- Nodo que permite cargar y emplear redes neuronales, además de incluir una GUI.

¹<https://jderobot.org/Robotics-Academy>

²<https://github.com/JdeRobot/dl-objectdetector>

- Fichero *MyAlgorithm.py* donde se proporciona al vehículo las órdenes de velocidad predichas por el nodo.

El objetivo principal es proporcionar una comparativa de diferentes modelos de redes neuronales que se pueden emplear para la conducción autónoma. Por este motivo, se estudiarán y se llevarán a cabo pruebas con diferentes arquitecturas de redes, como pueden ser redes neuronales convolucionales de clasificación y regresión o redes neuronales recurrentes de regresión.

2.2. Requisitos

El proyecto se desarrollará basándose en los subobjetivos mencionados anteriormente y tendrá que ajustarse a los requisitos de partida del proyecto. Estos requisitos condicionan la realización del proyecto, y son los siguientes:

1. La simulación se realizará en el simulador Gazebo, en concreto en la versión 7.15.0. El modelo de coche empleado es el modelo f1ROS (posee una cámara como sensor) creado por los desarrolladores de JdeRobot³. Este modelo se encuentra disponible en el repositorio de Github JdeRobot-assets⁴.
2. Se empleará el *middleware* robótico JdeRobot, en concreto en la versión 5.6.7. Este *middleware* se explicará en mayor detalle en el Capítulo 3. Esta plataforma simplifica el desarrollo del comportamiento del coche.
3. El sistema operativo que se empleará en este proyecto será Ubuntu 16.04.
4. El lenguaje de desarrollo empleado en los *plugins* del coche es C++. Sin embargo, en el resto de componentes se utilizarán el lenguaje Python. Debido a la compatibilidad con JdeRobot-5.6.7 y de éste con el *middleware* ROS Kinetic no se ha empleado Python-3.X, sino que se utiliza Python-2.7.
5. Se hará uso de la API de redes neuronales Keras, escrita en Python y capaz de ejecutarse sobre TensorFlow, CNTK o Theano. En este proyecto se ejecutará sobre TensorFlow y se empleará la versión 2.2.4.

³https://jderobot.org/Main_Page

⁴<https://github.com/JdeRobot/assets>

6. Las soluciones deben ser ágiles. Los algoritmos propuestos no pueden detenerse demasiado tiempo a pensar cuál será el próximo movimiento del vehículo, porque debe reaccionar rápido, en tiempo real y con movimientos suaves.

2.3. Metodología

El desarrollo del proyecto se ha realizado mediante una metodología iterativa, donde cada iteración está compuesta por varias fases: determinar objetivos, planificación, diseño e implementación, análisis de riesgos, además de reuniones periódicas con el tutor y el cotutor.

Se ha decidido seguir el modelo de desarrollo en espiral, creado por Barry Boehm [9] [10] [11]. Este modelo se adapta perfectamente a este tipo de proyectos, ya que permite separar el comportamiento final en varias subtareas más sencillas y después juntarlas. Además, el modelo permite una gran flexibilidad ante cambios en los requisitos, algo muy común en estos proyectos.

Este modelo de ciclo de vida permite obtener prototipos funcionales poco a poco, a la vez que se realiza el desarrollo del producto de forma incremental. El modelo consta de diferentes iteraciones, también conocidas como ciclos. En cada ciclo existen cuatro fases bien diferenciadas:

- Determinar objetivos: Se concretan los objetivos específicos que deben cumplirse para que el ciclo actual se considere terminado en función de los objetivos finales. Según se vayan incrementando las iteraciones, los objetivos serán más complejos.
- Análisis del riesgo: Se realiza un análisis detallado de cada posible riesgo que pueda tener el objetivo definido en la fase anterior. Se concretan los puntos a seguir con el fin de minimizar los riesgos, y después del análisis se planean estrategias alternativas.
- Desarrollar y probar: Se desarrolla el producto o las partes del producto que se han definido en las fases anteriores. Además, se realizan las pruebas necesarias que permitan asegurar la calidad de la implementación, y que pueda seguir funcionando en iteraciones futuras.

CAPÍTULO 2. OBJETIVOS

- Planificación: Se analizan los resultados obtenidos a través de las pruebas de la fase anterior, y es donde se planifica la siguiente iteración considerando los posibles errores que se han cometido.

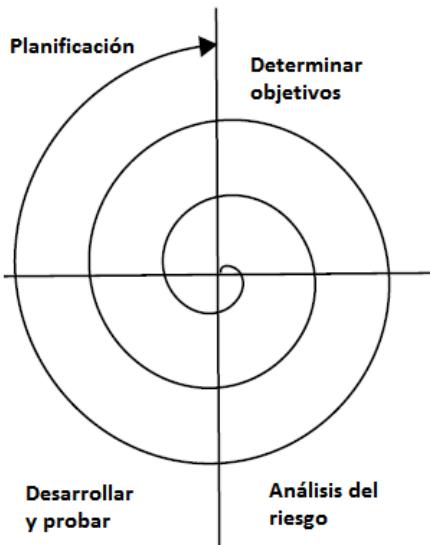


Figura 2.1: Modelo en espiral

Esta metodología se ha llevado a cabo mediante reuniones semanales con el tutor y el cotutor. En estas reuniones se analizaban los resultados de cada iteración, y en función de los resultados se fijaban nuevos objetivos. Además, en estas reuniones se analizaban los posibles fallos y se resolvían las dudas que iban surgiendo.

El código desarrollado semanalmente se ha subido al repositorio propio público de Github⁵, que emplea el sistema de control de versiones. Además, se ha desarrollado una bitácora en la página de JdeRobot⁶, donde semanalmente se han explicado los avances y se han mostrado los resultados mediante imágenes y vídeos.

El resultado del TFM, las diferentes redes neuronales desarrolladas, se encuentran disponibles en el repositorio Github, y se encuentran disponibles como software libre.

⁵<https://github.com/RoboticsURJC-students/2017-tfm-vanessa-fernandez>

⁶<https://jderobot.org/Vmartinezf-tfm>

2.4. Plan de trabajo

Las etapas en las que se divide el proyecto, que se corresponden con el modelo en espiral, son:

- Familiarización con la API de redes neuronales Keras y estudio de diferentes soluciones de aprendizaje extremo a extremo para conducción autónoma. En esta etapa se ha descargado e instalado Keras, así como todo el software necesario para desarrollar el proyecto. Además, se ha estudiado la creación de redes neuronales convolucionales en Keras, y su uso en algunos proyectos de la plataforma JdeRobot. Para tener una mejor comprensión de los diferentes modelos que se pueden emplear en conducción autónoma se han estudiado diferentes artículos.
- Desarrollo de la infraestructura necesaria en Gazebo, además de un componente académico que permita la conducción del coche, integrando una red entrenada en Keras en dicho componente.
- Creación de una base de datos que permita poder entrenar una red neuronal con la información visual del coche y los datos de velocidad.
- Estudio y mejora de redes neuronales convolucionales de clasificación aplicadas a la conducción autónoma. Se realizarán múltiples pruebas para tratar de conseguir la red más robusta posible y emplearla en el componente desarrollado.
- Estudio y mejora de redes neuronales convolucionales de regresión aplicadas a la conducción autónoma. Se realizarán diversas pruebas para intentar conseguir la red más robusta posible y emplearla en el componente desarrollado.
- Estudio y mejora de redes neuronales recurrentes de regresión aplicadas a la conducción autónoma. Se realizarán diversas pruebas para tratar de conseguir la red más robusta y emplearla en el componente desarrollado.

Capítulo 3

Estado del arte

En este capítulo se presenta el estado del arte sobre conducción autónoma mediante Redes Neuronales Convolucionales (CNN) y Redes Neuronales Recurrentes (RNN). Se describirán diferentes bases de datos para conducción autónoma como Comma.ai [12] o Udacity [13], así como diferentes arquitecturas de redes neuronales empleadas para el mismo problema como pueden ser *PilotNet* [14] o *ControlNet* [15]. Además, se describirán diferentes simuladores para conducción autónoma.

Además, se explican los diferentes ingredientes software en los que nos hemos apoyado para desarrollar el trabajo. Tales como el simulador Gazebo, el entorno JdeRobot, la librería OpenCV (empleada en el tratamiento de imagen), PyQt (para el desarrollo de la interfaz gráfica), Python como lenguaje de programación, Keras como framework para el desarrollo de redes neuronales, HDF5 como formato de archivo para guardar los modelos de redes neuronales.

3.1. Bases de datos para conducción autónoma

La conducción autónoma pretende que un vehículo sea capaz de conducir sólo en base a los datos proporcionados por determinados sensores. En concreto, la cámara es el sensor más empleado en las diferentes redes neuronales que se mencionarán en el estado del arte. Dado que queremos que el vehículo sea capaz de conducir bajo diferentes circunstancias, es decir, en diferentes entornos y diferentes iluminaciones, necesitaremos entrenar el modelo con un conjunto de imágenes representativo. Por ello, a lo largo de los últimos años han

surgido diferentes *datasets* con el fin de solucionar este problema. A continuación, se exponen algunos ejemplos de bases de datos empleadas para este propósito.

3.1.1. Comma.ai

La *startup* de conducción autónoma Comma.ai creó en 2016 un conjunto de datos [12] que permite probar modelos para controlar un vehículo autónomo. Este conjunto de datos consta de 11 videoclips grabados a 20 Hz por una cámara *Point Grey* colocada en el parabrisas de un *Acura ILX* 2016. El conjunto de datos es un archivo zip comprimido que ocupa un total de 45 GB.

Este conjunto de datos consta de un total de 7.25 horas de datos de conducción, donde los *frames* de vídeo tienen un tamaño de 160 x 320 píxeles. Junto a los archivos de vídeo se proporciona un conjunto de medidas de sensores donde se registran medidas como la velocidad, la aceleración, el ángulo de giro, la ubicación del GPS y los ángulos del giroscopio.

Además registran los *time stamps* en los que se midieron estas medidas de los sensores y los *time stamps* en que se capturaron los *frames* de la cámara. Los datos de los sensores se capturan en bruto y los *frames* de la cámara se almacenan en archivos HDF5 para que sean fáciles de usar en el aprendizaje automático y el *software* de control.

3.1.2. Udacity

Udacity posee un proyecto de código libre para conducción autónoma. El proyecto ofrece ejemplos de grabaciones de datos de más de diez horas de conducción y conjuntos de datos anotados de conducción, donde los objetos en el vídeo han sido marcados con cuadros circundantes. Además de las herramientas de código abierto, Udacity publica desafíos de programación para promover el desarrollo del proyecto.

Inicialmente, Udacity [13] poseía 40 GB de datos públicos con el fin de facilitar a las personas la construcción de modelos competitivos sin acceso al tipo de datos de conducción que Tesla o Google poseen. Sin embargo, debido a que los modelos de aprendizaje profundo necesitan muchos datos, la compañía publicó 183 GB adicionales de datos de conducción.

El conjunto de datos de Ucadity [16] consta de 223 GB de datos. Estos datos fueron grabados durante más de 70 minutos de conducción en días soleados y nublados, repartidos en dos días en *Mountain View*. Las imágenes fueron grabadas por tres cámaras frontales: izquierda, derecha y central. La variedad de imágenes aumentará la calidad de los resultados y proporcionará a los participantes datos más realistas para poder trabajar, ya que este conjunto de datos representa mejor los desafíos de la conducción en el mundo real y las condiciones variables de la carretera. Los datos almacenados constan de latitud, longitud, marcha, freno, aceleración, ángulos de dirección y velocidad.

3.1.3. SAIC Dataset

En el artículo *End-to-end Multi-Modal Multi-Task Vehicle Control for Self-Driving Cars with Visual Perceptions* [17] se creó un nuevo conjunto de datos, llamado SAIC, con el fin de obtener un conjunto de datos para pruebas reales de conducción.

El conjunto de datos incluye cinco horas de datos de conducción en el área norte de San José, principalmente en carreteras urbanas. Este conjunto contiene datos de conducción tanto de día como de noche.

El vehículo es conducido entre varios puntos y cada viaje entre los puntos tiene una duración de aproximadamente diez minutos. El estacionamiento, la espera en el semáforo y otras condiciones se consideran partes ruidosas y se filtran. Después de filtrar los vídeos ruidosos, los datos de dos horas se dividen en entrenamiento, validación y conjunto de test.

En la grabación del conjunto de datos se incluyen tres conductores para evitar sesgos hacia un comportamiento de conducción específico. De manera similar, se graban flujos de vídeo, valores de velocidad y direcciones. Las secuencias de vídeo contienen vídeos de una cámara frontal central y dos laterales con una *frame rate* de 30 fotogramas por segundo.

3.2. Simuladores

Un vehículo es caro, lo que implica que muchas investigaciones sobre conducción autónoma solamente estén disponibles para centros de investigación y corporaciones.

Cuando se emplea un vehículo puede que algo falle al probarlo, pudiendo incluso romperse el vehículo. Hoy en día existen numerosos simuladores, lo que permite a cualquier persona crear, programar y probar infinidad de vehículos y escenarios de forma segura y económica. Algunos de los simuladores más empleados se explicarán a continuación.

3.2.1. CARLA

CARLA [18] [19] es un simulador de código abierto para la investigación de conducción autónoma. Se ha desarrollado desde cero para respaldar el desarrollo, el entrenamiento y la validación de sistemas de conducción autónomos. Además, admite diferentes conjuntos de sensores y condiciones ambientales.

CARLA (Figura 3.1) simula un mundo dinámico y proporciona una interfaz simple entre el mundo y un agente que interactúa con el mundo. Para llevar a cabo esta funcionalidad, CARLA está diseñado como un sistema cliente-servidor, donde el servidor ejecuta la simulación y renderiza la escena. La API del cliente se implementa en Python y es responsable de la interacción entre el agente autónomo y el servidor a través de *sockets*. El cliente envía comandos y metacomandos al servidor y recibe las lecturas del sensor. Los comandos (dirección, aceleración y frenado) controlan el vehículo. Los metamandatos controlan el comportamiento del servidor y se utilizan para restablecer la simulación, cambiar las propiedades del entorno (condiciones climáticas, iluminación y densidad de automóviles y peatones) y modificar el conjunto de sensores.

CARLA presenta las siguientes características:

- Escalabilidad a través de una arquitectura multi-cliente servidor: varios clientes en el mismo nodo o en diferentes nodos pueden controlar diferentes actores.
- Permite a los usuarios controlar todos los aspectos relacionados con la simulación (generación de tráfico, comportamientos de peatones, climas, sensores, etc).
- Los usuarios pueden configurar diversos conjuntos de sensores (LIDAR, cámaras, sensores de profundidad, GPS, etc).
- Permite deshabilitar la representación para ofrecer una ejecución rápida de la simulación del tráfico y los comportamientos de la carretera para los que no se



Figura 3.1: Simulador CARLA.

requieren gráficos.

- Se pueden crear mapas siguiendo el estándar *OpenDrive* a través de herramientas como *RoadRunner*.
- Los usuarios pueden definir diferentes situaciones de tráfico.
- Integra ROS.

3.2.2. Gazebo

Gazebo [20] (Figura 3.2) es un simulador 3D de código abierto distribuido bajo licencia Apache 2.0. Este simulador se ha utilizado en ámbitos de investigación en robótica e Inteligencia Artificial. Es capaz de simular robots, objetos y sensores en entornos complejos de interior y exterior. Posee gráficos de gran calidad y un robusto motor de físicas (masa del robot, rozamiento, inercia, amortiguamiento, etc.). Fue elegido para realizar el DARPA Robotics Challenge (2012-2015) y está mantenido por la Fundación Robótica de Código Abierto (OSRF).

Los modelos de robots que se emplean en la simulación son creados mediante algún programa de modelado 3D (Blender, Sketchup, etc). Estos robots simulados necesitan ser dotados de inteligencia para lo cual se emplean *plugins*. Estos *plugins* pueden dotar al robot de inteligencia u ofrecer la información de sus sensores a aplicaciones externas y recibir de éstas comandos para los actuadores de los robots.

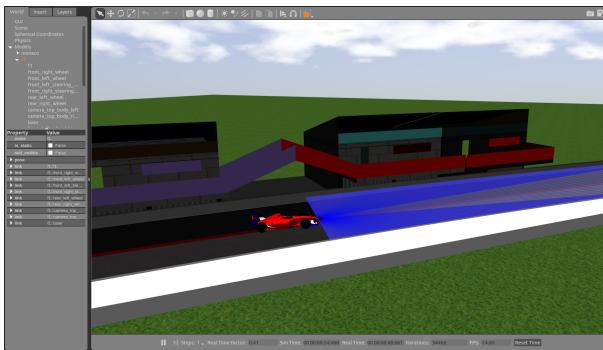


Figura 3.2: Simulador Gazebo.

3.2.3. Udacity's Self-Driving Car Simulator

Udacity's Self-Driving Car Simulator [13] [21] fue construido para Udacity's Self-Driving Car Nanodegree con el objetivo de que los estudiantes pudieran aprender cómo entrenar modelos de aprendizaje profundo que permitieran a los vehículos conducir de forma autónoma. Este simulador es de código abierto y requiere Unity.

El simulador de Udacity (Figura 3.3) permite al usuario seleccionar la escena deseada así como el modo de conducción en la pantalla principal. Existen dos modos de conducción: *Training Mode* y *Autonomus Mode*. En el modo *Training Mode* el coche se conduce manualmente mediante el teclado o el ratón y se almacenan los datos de conducción y las imágenes de las cámaras que posee el vehículo. Los datos grabados con este modo se pueden emplear para entrenar un modelo de aprendizaje automático. En el modo *Autonomous Mode* se puede probar el modelo de aprendizaje automático creado y comprobar su rendimiento en ejecución.

Técnicamente, el simulador actúa como un servidor desde el cual el programa puede conectarse y recibir un flujo de imágenes. Se puede crear un programa de Python que emplea un modelo de aprendizaje automático para procesar las imágenes de la carretera para predecir las mejores instrucciones de conducción y enviarlas de vuelta al servidor. Cada instrucción de conducción contiene un ángulo de dirección y un dato de aceleración, que cambia la dirección y la velocidad del automóvil.



Figura 3.3: Simulador Udacity’s Self-Driving Car Simulator.

3.2.4. Deepdrive 2.0

Deepdrive 2.0 [22] es un simulador de código abierto para Linux y Windows. Los simuladores actuales parecen vincularse a un *hardware* específico o no tienen forma de vincularse vehículos físicos. Deepdrive (Figura 3.4) para conseguir este propósito, incluye una amplia gama de sensores, automóviles y entornos, y facilita la transferencia a vehículos reales. Esto permitirá que un mayor número de personas utilice únicamente el simulador para hacer pruebas constantes.

Presenta algunas características únicas respecto a otros simuladores de código abierto:

- El *frame rate* es más elevado al emplear varias cámaras, ya que emplea memoria compartida en lugar de *sockets* y transferencia asíncrona.
- La superficie de la carretera no es plana, sino que incluye colinas, curvas y la anchura de la carretera varía.
- El mapa, los automóviles, la iluminación, etc. son gratuitos y son modificables en Unreal.

3.3. Redes neuronales

La conducción autónoma no es posible sin un algoritmo que tome decisiones. En algunos casos estos algoritmos pueden ser redes neuronales. En esta sección se describirán



Figura 3.4: Simulador Deepdrive.

diferentes arquitecturas de redes neuronales empleadas en la conducción autónoma.

3.3.1. Redes neuronales convolucionales

El aprendizaje de extremo a extremo para conducción autónoma se ha explorado desde finales de los años ochenta. The Autonomous Land Vehicle in a Neural Network (ALVINN) [5] se desarrolló para aprender ángulos de dirección a partir de una cámara y las medidas proporcionadas por un láser mediante una red neuronal con una sola capa oculta. Basados en esta idea de redes de extremo a extremo (dada una imagen o imágenes se preciden ángulos de dirección), existen múltiples aproximaciones [23] [24] [25] de las cuales veremos algunas a continuación.

Un buen ejemplo de red de extremo a extremo es la red PilotNet [24] [14] creada por Nvidia. En “End to end learning for self-driving cars” [24] se describe dicha red con detalle. Es una red neuronal convolucional (CNN) que mapea píxeles en crudo de una sola cámara frontal a comandos de dirección directamente. El comando propuesto por la CNN se compara con el comando deseado para la imagen en concreto y los pesos de la red se van ajustando para aproximar la salida de la red a la salida deseada. El ajuste de los pesos se realiza empleando *back propagation*.

La red PilotNet (Figura 3.5) consta de 9 capas, que incluyen una capa de normalización, 5 capas convolucionales y 3 capas *fully-connected*. La imagen de entrada se divide en planos YUV y se pasa a la red. Las capas convolucionales las diseñaron para realizar la extracción de características y las eligieron a través de experimentos que variaban las

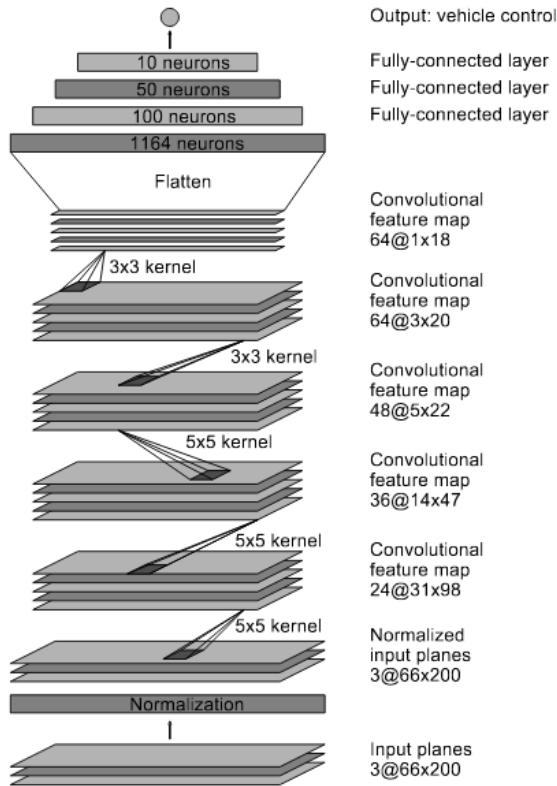


Figura 3.5: Arquitectura Pilotnet.

configuraciones de capas. Las dos primeras capas convolucionales usaban un *stride* de 2x2 y un kernel 5x5, mientras que las 3 últimas capas usaban un *non-stride* y un kernel 3x3. Las 3 capas *fully-connected* fueron diseñadas para funcionar como un controlador de la dirección, pero no es posible saber exactamente qué partes de la red funcionan principalmente como extractor de características y cuáles sirven como controlador. El sistema aprende automáticamente las representaciones internas, como la detección de características útiles de la carretera.

El objetivo de [14] es explicar lo que PilotNet aprende y cómo toma sus decisiones. Con este fin, se desarrolla un método para determinar qué elementos en la imagen de la carretera influyen más en la decisión de la dirección de PilotNet. Llaman a estas secciones de imagen objetos salientes. Se puede encontrar un informe detallado del método de detección de saliencia en “VisualBackProp: Efficient Visualization of CNNs for Autonomous Driving” [26].

La idea central de “Explaining how a deep neural network trained with end-to-end learning steers a car” [14] para discernir los objetos salientes es encontrar partes de la imagen que corresponden a ubicaciones donde los mapas de características tienen las mejores activaciones. Las activaciones de los mapas de nivel superior se convierten en máscaras para las activaciones de niveles inferiores utilizando el siguiente algoritmo:

1. En cada capa, las activaciones de los mapas de características se promedian.
2. El mapa con el promedio más alto se escala según el tamaño del mapa de la capa de abajo. El aumento de escala se realiza mediante una deconvolución. Los parámetros (*filter size* y *stride*) utilizados para la deconvolución son los mismos que se emplearon en la capa convolucional utilizada para generar el mapa. Los pesos de la deconvolución se establecen en 1.0 y los sesgos en 0.0.
3. El mapa promediado aumentado de un nivel superior se multiplica después con el mapa promediado de la capa de abajo (ahora son del mismo tamaño). El resultado es una máscara de tamaño intermedio.
4. La máscara intermedia se escala al tamaño de los mapas de la capa inferior de la misma manera que en el paso 2.
5. El mapa intermedio mejorado se multiplica de nuevo con el mapa promediado de la capa de abajo. Se obtiene una nueva máscara intermedia.
6. Los pasos 4 y 5 se repiten hasta que se alcanza la entrada. La última máscara que es del tamaño de la imagen de entrada se normaliza al rango 0-1 y se convierte en la máscara de visualización final.

Esta máscara de visualización muestra qué regiones de la imagen de entrada contribuyen más a la salida de la red. Estas regiones identifican los objetos salientes. En la Figura 3.6 se pueden ver ejemplos de objetos salientes para varias imágenes de entrada.

Los resultados muestran que PilotNet aprende a reconocer objetos relevantes en la carretera y que es capaz de mantener el vehículo en el carril con éxito en una amplia variedad de condiciones, independientemente de si las marcas del carril están presentes en la carretera o no.



Figura 3.6: Ejemplos de objetos salientes para varias imágenes de entrada.

En “Self-driving a Car in Simulation Through a CNN” [27] se propone una nueva arquitectura de red, llamada TinyPilotnet, que se deriva de la red Pilotnet [24] [14]. La red TinyPilotnet (Figura 3.7) está compuesta por una capa de entrada, en la que se introducirán imágenes de resolución 16x32 y un único canal, seguida por dos capas convolucionales de kernel 3x3, y una capa *dropout* configurada al 50 % de probabilidad para agilizar el entrenamiento. Finalmente, el tensor de información se convierte en un vector que es conectado a dos capas *fully-connected* que conducen a un par de neuronas, cada una de ellas dedicada a predecir los valores de dirección y aceleración respectivamente. La imagen de entrada tiene un solo canal formado por el canal de saturación del espacio de color HSV.

En “Event-based vision meets deep learning on steering prediction for self-driving cars” [28] se presenta un enfoque de red neuronal profunda que emplea cámaras de eventos (sensores de inspiración biológica que no adquieren imágenes completas a una velocidad de *frames* fija, sino que tienen píxeles independientes que solo producen cambios de intensidad de forma asíncrona en el momento en el que ocurren) para predecir el ángulo de giro de un vehículo. Los eventos se convierten en *frames* de eventos por acumulación de píxeles en un intervalo de tiempo constante. Posteriormente, una red neuronal profunda los asigna a los ángulos de dirección.

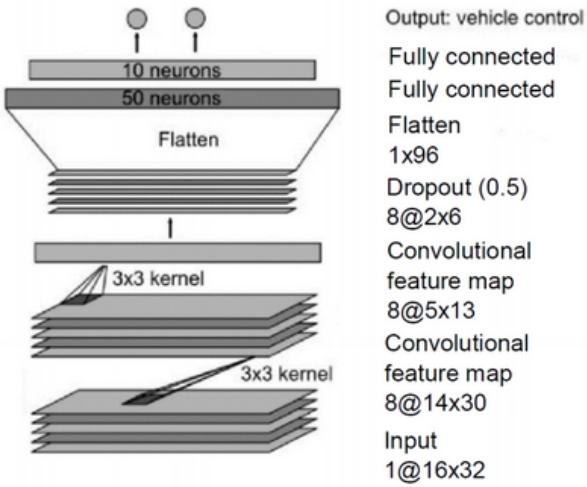


Figura 3.7: Arquitectura TinyPilotnet.

En este artículo inicialmente apilan los *frames* de eventos de diferente polaridad, creando una imagen de eventos 2D. Después, implementan una serie de arquitecturas ResNet, es decir, ResNet18 y ResNet50. Estas redes son utilizadas como extractores de características para el problema de regresión, considerando solo las capas convolucionales. Para codificar las características de la imagen extraídas de la última capa convolucional en un descriptor vectorizado, se emplea una capa *global average pooling* que devuelve la media del canal de las características. Después se agrega una capa *fully-connected* (con dimensionalidad 256 para ResNet18 y 1024 para ResNet50), seguida de una ReLU no lineal y una capa *fully-connected* unidimensional para generar el ángulo.

En este artículo [28] se preciden ángulos empleando 3 tipos de entradas: 1. imágenes en escala de grises, 2. diferencia de imágenes en escala de grises, 3. imágenes creadas por la acumulación de eventos. Analizan el rendimiento de la red en función del tiempo de integración utilizado para generar las imágenes de eventos (10, 25, 50, 100 y 200 ms). Cuanto mayor es el tiempo de integración, mayor es la traza de eventos que aparecen en los contornos de los objetos. La red funciona mejor cuando se entrena con imágenes de eventos correspondientes a 50 ms, y el rendimiento se degrada para tiempos de integración cada vez más grandes. Uno de los problemas que presentan las entradas que emplean imágenes en escala de grises es que a altas velocidades las imágenes se difuminan y la

diferencia de imágenes se vuelve muy ruidosa.

En el artículo “From Pixels to Actions: Learning to Drive a Car with Deep Neural Networks” [29] se realiza un amplio estudio donde se analiza una red neuronal de extremo a extremo para predecir las acciones de dirección de un vehículo en base a las imágenes de una cámara, así como las dependencias temporales de entradas consecutivas y la diferencia entre redes de clasificación y redes de regresión.

La arquitectura principal que emplean es una variación de la arquitectura PilotNet, AlexNet o VGG19. Para AlexNet se elimina el *dropout* de las 2 capas densas finales y se reduce el tamaño de 500 y 200 neuronas. La capa de salida de la red depende de su tipo (regresión o clasificación) y para una red de clasificación del número de clases. Para el caso de clasificación, cuantifican las medidas del ángulo de dirección en valores discretos, que representan las etiquetas de la clase. Esta cuantificación es necesaria como entrada cuando se tiene una red de clasificación y permite equilibrar los datos a través de los pesos de la muestra. Esta ponderación actúa como un coeficiente para la tasa de aprendizaje de la red para cada muestra. El peso de una muestra está directamente relacionado con la clase a la que pertenece cuando se cuantifica. La ponderación de muestra se realiza para regresión y clasificación.

Se estudia la influencia de las especificaciones de cuantización de clase en el rendimiento del sistema. Estas especificaciones consisten en la cantidad de clases y la asignación del rango de entrada de estas clases. Se comparan redes con diferentes grados de granularidad, lo que influye en el rendimiento. Se compara un esquema de cuantificación de grano grueso de 7 clases con uno de grano fino de 17 clases, obteniendo mejores resultados con el de grano grueso.

Además, en este artículo se evalúan métodos que permiten que el sistema aproveche la información de entradas consecutivas: un método que sigue una arquitectura de extremo a extremo y un método que emplea capas recurrentes (lo veremos en la siguiente subsección).

El método que emplea una CNN para la predicción, que llaman *stacked frames*, concatena varias imágenes de entrada consecutivas para crear una imagen apilada. La entrada

a la red es esta imagen apilada (para la imagen t se concatenan las imágenes $t-1$, $t-2$, etc). El tamaño de entrada será la única variable que se modifique, es decir, no se modifica la red. Por esta razón, las imágenes se concatenan en la dimensión de profundidad (canal) y no en una nueva dimensión. Por ejemplo, apilar 2 imágenes anteriores a la imagen RGB actual de $160 \times 320 \times 3$ cambaría su tamaño a $160 \times 320 \times 9$. Los resultados muestran un aumento en el rendimiento de las métricas con este método. Se cree que es debido a que la red puede hacer una predicción basada en la información promedio de múltiples imágenes. Para una sola imagen, el valor predicho puede ser o muy alto o muy bajo. En cambio, para imágenes concatenadas, la información combinada podría cancelarse entre sí, dando una mejor predicción promedio. Suponiendo que la red promedie la información, aumentar el número de imágenes podría hacer que la red perdiera la capacidad de respuesta. Por ello emplean 3 *frames* concatenados.

Además, en este artículo se demuestra cualitativamente que las métricas estándar que se emplean para evaluar redes no necesariamente reflejan con precisión el comportamiento de conducción de un sistema. Una matriz de confusión prometedora puede dar como resultado un comportamiento de conducción deficiente, mientras que una matriz con mal aspecto puede dar como resultado un buen comportamiento de conducción.

3.3.2. Redes neuronales recurrentes

Las redes neuronales recurrentes (RNNs) representan una clase de redes neuronales artificiales que utilizan células de memoria para modelar la relación temporal entre los datos de entrada y, por lo tanto, aprender la dinámica subyacente. Con la introducción de las Long Short-Term Memory (LSTM), el modelado de relaciones a largo plazo se hizo posible dentro de RNN.

En múltiples investigaciones sobre conducción autónoma se ha aprovechado la capacidad de estas redes para poder aprovechar la información de imágenes consecutivas. Algunas de estas investigaciones las veremos a continuación.

Un ejemplo de investigación donde se emplean capas LSTM es la propuesta por “Reactive ground vehicle control via deep networks” [15]. En esta investigación se presenta un controlador reactivo basado en aprendizaje profundo que emplea una arquitectura de red

simple que requiere pocas imágenes de entrenamiento. A pesar de esta estructura simple, su arquitectura de red, llamada ControlNet, supera a otras redes más complejas en múltiples entornos (entornos interiores estructurados y entornos exteriores no estructurados) utilizando diferentes plataformas robóticas. Es decir, el artículo se centra en el control reactivo, donde el robot debe evitar obstáculos que no están presentes durante la construcción del mapa.

ControlNet extrae imágenes RGB para generar comandos de control: gira a la derecha, gira a la izquierda y recto. La arquitectura de ControlNet consiste en alternar capas convolucionales con capas de *maxpooling* seguidas de capas *fully-connected*. Las capas convolucionales y la de *pooling* extraen información geométrica sobre el medio ambiente, mientras que las capas *fully-connected* actúan como un clasificador general. La capa LSTM permite al robot incorporar información temporal permitiéndole continuar moviéndose en la misma dirección sobre varios *frames*. La estructura de ControlNet (Figura 3.8) es:

- 2D Convolution, 16 filtros de tamaño 10x10
- Max Pooling, filtro de 3x3, stride de 2
- 2D Convolution, 16 filtros de tamaño 5x5
- Max Pooling, filtro de 3x3, stride de 2
- 2D Convolution, 16 filtros de tamaño 5x5
- Max Pooling, filtro de 3x3, stride de 2
- 2D Convolution, 16 filtros de tamaño 5x5
- Max Pooling, filtro de 3x3, stride de 2
- Fully connected, 50 neuronas
- ReLu
- Fully connected, 50 neuronas

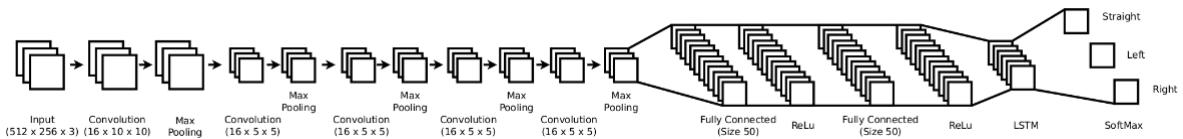


Figura 3.8: Estructura de red ControlNet.

- LSTM (5 frames)
- Softmax con 3 salidas

En “End-to-end deep learning for steering autonomous vehicles considering temporal dependencies” [30] se propone una Convolutional Long Short-Term Memory Recurrent Neural Networks, conocida como C-LSTM (Figura 3.9), que es entrenable de extremo a extremo, para aprender las dependencias visual y temporal dinámica de la conducción. El sistema investigado está compuesto por una cámara RGB frontal y una red neuronal que consta de una CNN y LSTM que estiman el ángulo del volante en función de la entrada de la cámara. Las imágenes de la cámara se procesan fotograma a fotograma por la CNN. Las características resultantes luego se procesan dentro de la red LSTM para aprender las dependencias temporales. La predicción del ángulo de dirección se calcula a través de la capa de clasificación de salida después de las capas LSTM.

Aplican el concepto de *transfer learning*. La CNN está pre-entrenada en el conjunto de datos Imagenet. Luego, transfieren la red neuronal entrenada a otra específica enfocada en imágenes de conducción. Posteriormente, en la LSTM se procesa una secuencia de vectores de características de longitud fija w de la CNN. A su vez, las capas LSTM aprenden a reconocer las dependencias temporales que conducen a una decisión de dirección y_t basada en las entradas de x_{t-w} a x_t . Los valores pequeños de t conducen a reacciones más rápidas, pero la red aprende solo las dependencias a corto plazo y la susceptibilidad a los aumentos de *frames* mal clasificados individualmente. Mientras que los valores elevados de t conducen a un comportamiento más suave y, por tanto, predicciones de dirección más estables, pero aumenta las posibilidades de aprender dependencias erróneas a largo plazo.

El concepto de ventana deslizante permite a la red aprender a reconocer diferentes

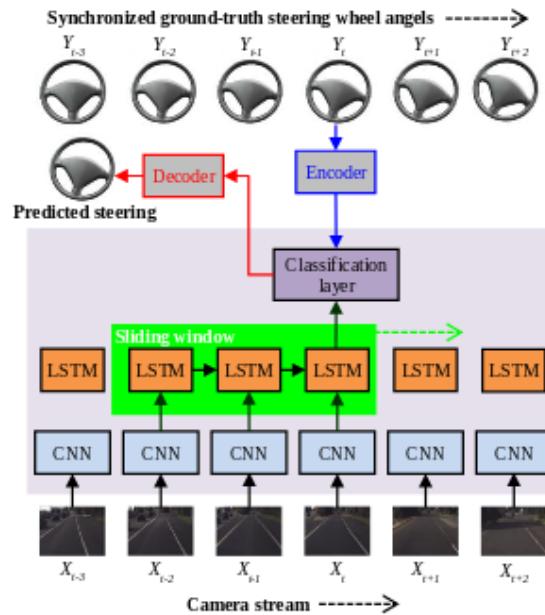


Figura 3.9: Arquitectura C-LSTM.

ángulos de dirección desde el mismo *frame* X_i pero en diferentes estados temporales de las capas LSTM. Tanto los pesos de la LSTM como de la CNN se comparten en diferentes pasos dentro de la ventana deslizante y, esto permite un tamaño de ventana arbitrariamente largo.

Plantean la regresión del ángulo de dirección como un problema de clasificación. Esta es la razón por la que el único número que representa el ángulo de dirección Y_t está codificado como un vector de activaciones de las neuronas de la capa de clasificación. Utilizan una capa totalmente conectada con activaciones *tanh* para la capa de clasificación.

En esta propuesta para el entrenamiento de dominio “específico”, la capa de clasificación de la CNN se reinicializa y se entrena con los datos de carretera de la cámara. El entrenamiento de la capa LSTM se lleva a cabo de manera múltiple, la red aprende las decisiones de dirección que están asociadas con los intervalos de conducción. La capa de clasificación y las capas LSTM emplean una mayor velocidad de aprendizaje porque se inicializan con valores aleatorios. La CNN y la LSTM se entrenan conjuntamente al mismo tiempo.

En “ Deep steering: Learning end-to-end driving model from spatial and temporal visual cues” [31] se propone un modelo basado en visión que mapea imágenes de entrada en ángulos de dirección usando redes profundas. Se segmenta la red en subredes. Es decir, los *frames* se introducen primero en una red de extracción de características, generando una representación de características de longitud fija que modela el entorno visual y el estado interno de un vehículo. Las características extraídas se envían a una red de predicción de dirección. En la subred de extracción de características emplea una Spatio-Temporal Convolution (ST-Conv) que cambia las dimensiones temporales y espaciales. Se emplea una capa *fully-connected* tras la ST-Conv para obtener un vector de características de dimensión 128. Además, en la subred de extracción de características se introducen capas LSTM, para lo cual se emplea ConvLSTM. La subred de predicción de dirección propone concatenar acciones de dirección y de estado del vehículo con el vector de características de 128 dimensiones. Para ello se añade 1 paso de recurrencia entre la salida final y las dos capas *concat* justo antes/después de la LSTM. La capa *concat* antes de la LSTM agrega la velocidad, y el par de torsión y ángulo de rueda al vector de 128 dimensiones, formando un vector de 131 dimensiones. La capa *concat* después de LSTM está compuesta por un vector de características 128-d + salida de LSTM 64-d + salida final previa 3d.

En “Interpretable learning for self-driving cars by visualizing causal attention” [25] se propone un modelo de atención visual para entrenar una red convolucional de extremo a extremo desde las imágenes hasta el ángulo de giro. El modelo de atención resalta las regiones de imagen que potencialmente influye en la salida de la red, de las cuales algunas son influencias reales y otras espurias. Su modelo predice comandos de ángulo de dirección continuos a partir de píxeles en bruto. El modelo predice el radio de giro inverso \hat{u} , pero se relaciona con el comando de ángulo de dirección mediante geometría de Ackermann.

En este método emplean una red neuronal convolucional para extraer un conjunto de vectores de características visuales codificadas, a las que se refieren como una característica convolucional cubo x_t . Cada vector de características puede contener descripciones de objetos de alto nivel que permiten que el modelo de atención preste atención selectiva a ciertas partes de una imagen de entrada al elegir un subconjunto de vectores de características. Utilizan la red PilotNet [24] para aprender un modelo de conducción, pero omiten las capas de *maxpooling* para evitar la pérdida de información de ubicación espa-

cial. Recopilan un cubo xt de características convolucionales tridimensionales de la última capa empujando la imagen preprocesada a través del modelo, y el cubo de características de salida se emplea como entrada de las capas LSTM. Utilizan una red LSTM que predice el radio de giro inverso y genera ponderaciones de atención en cada paso de tiempo t condicionado al estado oculto anterior y una característica convolucional actual xt . Asumen una capa oculta condicionada al estado oculto anterior y los vectores de características actuales. El peso de atención para cada ubicación espacial se calcula luego mediante una función de regresión logística multinomial.

El último paso de este método es un decodificador de grano fino en el que refinan un mapa de atención visual y detectan saliencias visuales locales. Aunque un mapa de atención del decodificador de grano grueso proporciona una probabilidad de importancia sobre un espacio de imagen 2D, el modelo debe determinar regiones específicas que causan un efecto casual en el rendimiento de la predicción. Obtienen una disminución en el rendimiento cuando se oculta una prominencia visual local en una imagen de entrada en bruto. En primer lugar, recopilan un conjunto consecutivo de pesos de atención e ingresan imágenes en bruto para los T pasos de tiempo especificados por el usuario. Luego, crean un mapa de atención, Mt . La red neuronal de 5 capas (basada en PilotNet) emplea una pila de filtros 5x5 y 3x3 sin ninguna capa *pooling*, y por tanto la imagen de dimensiones 80x160 se procesa para producir un cubo de características 10x20x64, conservando su relación de aspecto. Para extraer una prominencia visual local, primero muestran aleatoriamente partículas de 2D con reemplazo sobre una imagen de entrada condicionada en el mapa de atención Mt . También emplean el eje de tiempo como la tercera dimensión para considerar las características temporales de las saliencias visuales, almacenando partículas en el espacio temporales 3D. Posteriormente, aplican un algoritmo de *clustering* (DBSCAN) para encontrar una prominencia visual local agrupando las partículas 3D en *clusters*. Para los puntos de cada grupo y cada *frame* de tiempo t , calculan el algoritmo *convex hull* para encontrar una región local de cada prominencia visual destacada.

En “From pixels to actions: Learning to drive a car with deep neural networks” [29] además de aprovechar la información temporal concatenando *frames* se estudia la inclusión de capas recurrentes. Es decir, modifican su arquitectura para incluir capas LSTM, que permiten capturar información temporal entre entradas consecutivas. Las redes se

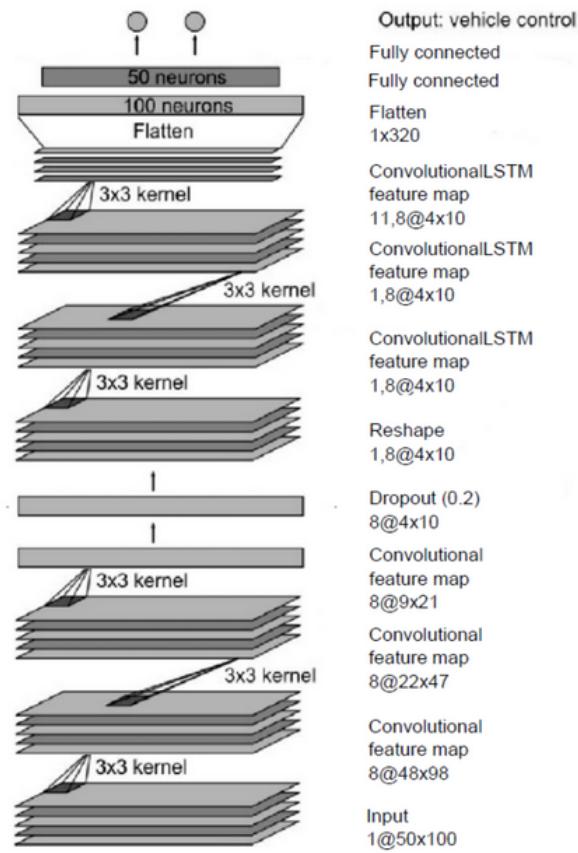


Figura 3.10: Arquitectura DeepestLSTM-TinyPilotnet.

entrenan con un vector de entrada que consiste en la imagen de entrada y una serie de imágenes anteriores, lo que se traduce en una ventana de tiempo. Comparan muchas variaciones de la arquitectura PilotNet [24]: (1) se cambia una o dos capas densas a capas LSTM, (2) se agrega una capa LSTM después de las capas densas, y (3) se cambia la capa de salida a LSTM. Todos los experimentos que realizan con redes LSTM demostraron que la incorporación de capas LSTM no aumentó ni redujo el rendimiento de la red.

En “Self-driving a Car in Simulation Through a CNN” [27] se propone una nueva arquitectura basada en la arquitectura TinyPilotnet (Figura 3.7) para mejorar el rendimiento de la misma. Esta nueva red (Figura 3.10) está formada principalmente por 3 capas convolucionales de kernel 3x3, combinadas con capas *maxpooling*, seguidas por 3 capas LSTM convolucionales 5x5 y 2 capas *fully-connected*. Las capas LSTM producen un efecto de memoria, por lo que los ángulos de dirección y los valores de aceleración dados por la CNN están influenciados por los anteriores.

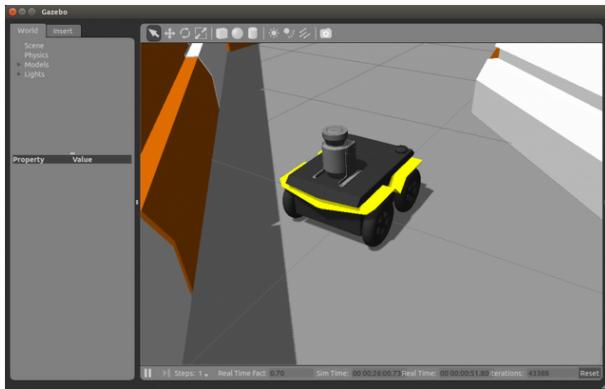


Figura 3.11: Simulador Gazebo.

3.4. Infraestructura empleada

En esta sección se explican los ingredientes software empleados para desarrollar este proyecto. Se describirá desde el lenguaje empleado hasta diferentes herramientas software que han sido necesarias.

3.4.1. Simulador Gazebo

Como hemos visto al hablar de simuladores, Gazebo¹ es un programa de código abierto distribuido bajo licencia Apache 2.0. Se emplea en el desarrollo de aplicaciones robóticas e inteligencia artificial. Es capaz de simular robots, objetos y sensores en entornos complejos de interior y exterior. Tiene gráficas de gran calidad y un robusto motor de físicas (masa del robot, rozamiento, inercia, amortiguamiento, etc.).

En este trabajo se emplea la versión 7.15.0 de Gazebo. Gracias a Gazebo se pueden incluir texturas, luces y sombras en los escenarios, así como simular la física como por ejemplo choques, empujes, gravedad, etc. Además, incluye diversos sensores, como pueden ser cámaras y láseres, los cuales podrán ser incorporados en los robots que empleemos. Todo ello hace que sea una herramienta muy potente y de gran ayuda en robótica.

¹<http://gazebosim.org/>

Los mundos simulados con Gazebo son mundos 3D, que se cargan a partir de ficheros con extensión “.world”. Son ficheros Extensible Markup Language (XML) definidos en el lenguaje Simulation Description Format (SDF). Este lenguaje contiene una descripción completa de todos los elementos que tiene el mundo y los robots, incluyendo:

- Escena: Luz ambiente, propiedades del cielo, sombras, etc.
- Mundo: Representa el mundo como un conjunto de modelos, *plugins* y propiedades físicas.
- Modelo: Articulaciones, objetos de colisión, sensores, etc.
- Físicas: Gravedad, motor físico, paso del tiempo, colisiones, inercias, etc.
- Plugins: Sobre un mundo, modelo o sensor.
- Luz: Los puntos y origen de la luz.

Las etiquetas empleadas en el fichero para representar estos elementos son: Scene, World, Model, Physics, Plugin, y Light.

Los modelos de robots que se emplean en la simulación son creados mediante algún programa de modelado 3D (Blender, Sketchup...). Estos robots simulados necesitan ser dotados de inteligencia para lo cual se emplean los *plugins*. Estos *plugins* pueden dotar al robot de inteligencia u ofrecer la información de sus sensores a aplicaciones externas y recibir de éstas comandos para los actuadores de los robots.

3.4.2. Entorno JdeRobot

JdeRobot² es un *middleware* de software libre para el desarrollo de aplicaciones con robots y visión artificial. Esta plataforma fue creada por el Grupo de Robótica de la Universidad Rey Juan Carlos en 2003 y está licenciada como GPLv3³.

Está desarrollado en C y C++, aunque contiene componentes desarrollados en lenguajes como Python y JavaScript. El entorno que ofrece está basado en componentes, los

²http://jderobot.org/Main_Page

³<https://www.gnu.org/licenses/quick-guide-gplv3.html>

cuales se ejecutan como procesos. Dichos componentes interoperan entre sí a través del *middleware* de comunicaciones ICE o de ROS messages. Tanto ICE como ROS-messages permiten la interoperación entre los componentes incluso estando desarrollados en diferentes lenguajes.

Es capaz de llevar a cabo diferentes tareas en tiempo real de forma sencilla. Cada componente *driver* está asociado a un dispositivo hardware del robot, un sensor o actuador e incluye funciones para poder emplearlo. Esto simplifica el acceso a los diferentes componentes hardware, ya que con una simple función se puede acceder a ellos.

Las aplicaciones constan de uno o varios componentes. Los que interactúan directamente con los sensores y actuadores del robot se llaman *drivers*, que son los encargados de controlar que los robots reciben órdenes a través de interfaces ICE o ROS messages. Otros llevan en su código las funciones perceptivas, procesamiento de señales o la lógica de control e inteligencia del robot. En la siguiente imagen se puede ver un ejemplo de esta comunicación con un AR Drone empleando interfaces ICE. La misma lógica de comportamiento se puede conectar al *driver* del drone real o al *driver* del drone simulado, basta con cambiar la configuración.

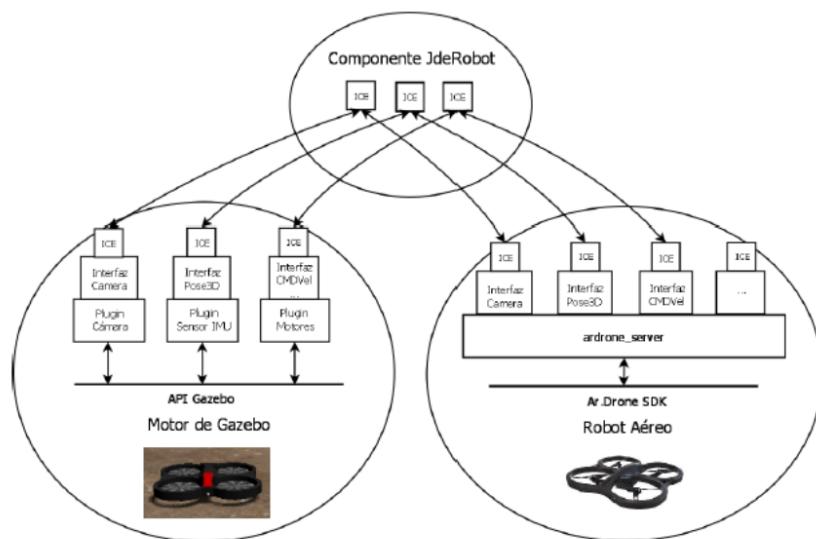


Figura 3.12: Ejemplo de componentes JdeRobot

Esta plataforma soporta gran variedad de dispositivos como el cuadricóptero AR Drone de Parrot, el robot Pioneer de MobileRobotics Inc., el robot Kobuki de Yujin Robot, el robot Pibot de JdeRobot-kids⁴, el robot Mbot de JdeRobot-kids, cámaras firewire, USB e IP, los escáneres laser LMS de SICK y URG de Hokuyo, el simulador Gazebo, sensores de profundidad como Kinect y otros dispositivos X10 de domótica. A parte de todo esto, tiene soporte para software externo como OpenCV, OpenGL, XForms, GTK, Player y GSL.

En el desarrollo del proyecto de este TFM se empleará la versión 5.6.7 de JdeRobot, ya que es la última versión estable.

3.4.3. Entorno ROS

Robot Operating System (ROS)⁵ [32] es una plataforma de software libre para el desarrollo de software de robots, que provee servicios estándar de un sistema operativo como la abstracción del hardware, el control de dispositivos de bajo nivel, mecanismos de intercambio de mensajes entre procesos y un conjunto de herramientas ampliamente utilizadas en robótica. Esta plataforma es de código abierto y se distribuye bajo licencia BSD.

Uno de los grandes beneficios del uso de ROS es la integración con el simulador Gazebo. Para realizar esta comunicación se hace uso de un conjunto de paquetes de *ros* llamado *gazebo_ros_pkgs*⁶ [33]. Gazebo se integra con *ros* mediante *ROS Messages*, servicios y reconfiguración dinámica. En la Figura 3.13 se puede observar una visión general de la interfaz *gazebo_ros_pkgs*.

⁴<https://jderobot.org/PyBoKids>

⁵<https://www.ros.org/>

⁶http://wiki.ros.org/gazebo_ros_pkgs

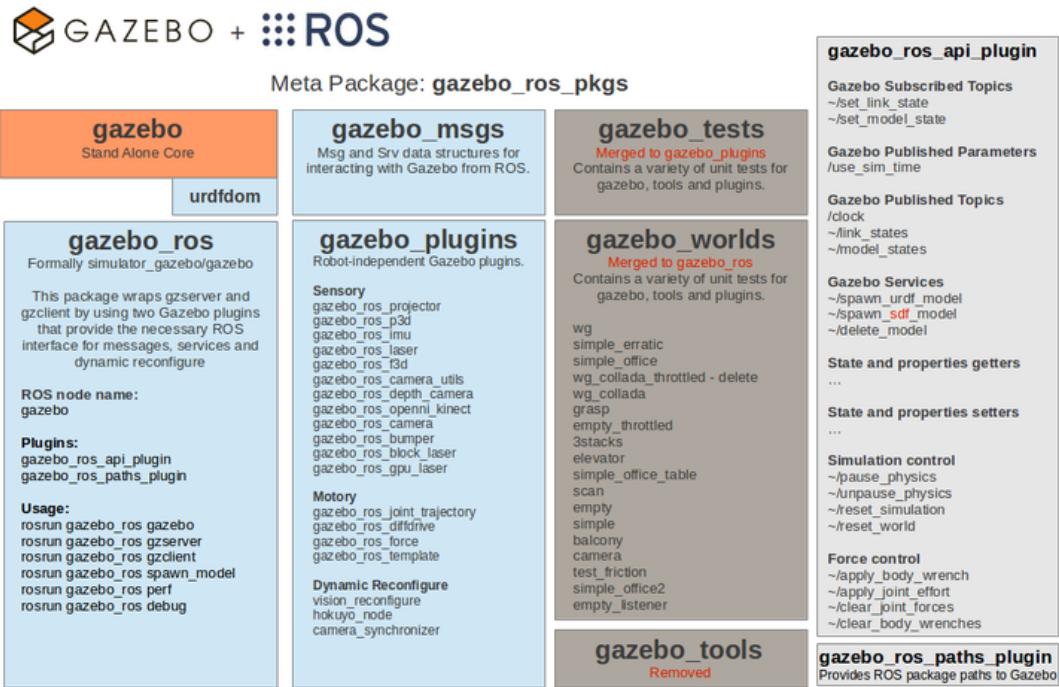


Figura 3.13: Interfaz del conjunto de paquetes `gazebo_ros_pkgs`

ROS está formado por una colección de nodos o procesos que se combinan en un gráfico, y se comunican entre ellos mediante *topics* de transmisión, servicios RPC, y el Servidor de Parámetros. El sistema de control de un robot se compone de diferentes nodos, siendo mayor el número de nodos cuanta mayor sea la funcionalidad del robot. En ROS existen distintos nodos que controlan un láser, cámaras, motores de ruedas, odometría, etc. El uso de nodos de ROS en el robot permite localizar más fácilmente los fallos que puedan surgir, ya que cada fallo se concentra únicamente en un nodo.

Los *topics* de ROS⁷ [34] son una forma de comunicación de los nodos. Los *topics* también se conocen como buses sobre los cuales los nodos intercambian mensajes. Los *topics* implementan un mecanismo de comunicación de publicación y/o suscripción. La semántica de publicación y/o suscripción de los *topics* es anónima, lo que desacopla la producción de información de consumo. De esta forma los nodos no saben con quien se están comunicando. Además, los nodos que desean recibir mensajes sobre un *topic* se deben suscribir a él para obtener la información que publique dicho *topic*. Después de suscribirse, todos los mensajes sobre el *topic* se envían al nodo que realizó la solicitud. Es

⁷<http://wiki.ros.org/Topics>

posible que existan varios suscriptores del mismo *topic*.

En ROS existen diversos *plugins*⁸ que aportan una gran variedad de funcionalidad para los distintos modelos de robots de Gazebo. Algunos de los *plugins* más destacados son libgazebo_ros_camera, que permite controlar una cámara; libgazebo_ros_laser, que controla un sensor láser; o libgazebo_ros_bumper que controla un sensor *bumper* (sensor de contacto). Los *plugins* libgazebo_ros_camera y libgazebo_ros_laser serán empleados por el coche utilizado en este proyecto.

3.4.4. Python

Python⁹ es un lenguaje de programación fácil de aprender y de alto nivel. Su creador fue Guido van Rossum, un investigador holandés que trabajaba en el centro de investigación CWI (Centrum Wiskunde & Informatica). La primera versión surgió en 1991 pero no fue publicada hasta tres años después. Guido dio el nombre de Python en honor a la serie de televisión *Monty Python's Flying Circus*.

Python incluye orientación a objetos, manejo de excepciones, listas, diccionarios, etc. A pesar de todo lo que soporta, se creó con el objetivo de que fuera un lenguaje sencillo de entender y manejar, sin perder las funcionalidades que pueden ofrecer lenguajes complejos tales como C.

Actualmente Python es un lenguaje de código abierto administrado por Python Software Foundation. Incluye módulos que permiten la entrada y salida de ficheros, *sockets*, llamadas al sistema e incluso interfaces gráficas como Qt. Además, permite dividir el programa en módulos reutilizables y no es necesario compilarlo, pues es interpretado. Es uno de los ejes de Robotics-Academy.

La última versión ofrecida por Python Software Foundation es la 3.7.3 , pero en nuestro caso se empleará la 2.7.12 por compatibilidad con JdeRobot 5.6.7, que a su vez sigue en esa versión de Python para ser compatible con ROS Kinetic. El código en el que están escritos los componentes académicos y las soluciones es Python.

⁸<http://wiki.ros.org/gazebo-plugins>

⁹<https://www.python.org/>

3.4.5. Biblioteca OpenCV

OpenCV¹⁰ es una librería de código abierto desarrollada inicialmente por Intel y publicada bajo licencia de BSD. Esta librería implementa gran variedad de herramientas para la interpretación de la imagen. Sus siglas provienen de los términos anglosajones “Open Source Computer Vision Library”, y está orientada a aplicaciones de visión por computador en tiempo real.

Esta librería puede ser usada en MacOS, Windows, Android y Linux, y existen versiones para C#, Python y Java, a pesar de que originalmente era una librería en C/C++. Además, hay interfaces en desarrollo para Ruby, Matlab y otros lenguajes.

OpenCV principalmente implementa algoritmos para las técnicas de calibración, detección de rasgos, para el rastreo, análisis de la forma, análisis del movimiento, reconstrucción 3D, segmentación de objetos y reconocimiento. Los algoritmos se basan en estructuras de datos flexibles acopladas con estructuras IPL (Intel Image Processing Library), aprovechándose de la arquitectura de Intel en la optimización de más de la mitad de las funciones.

Fue diseñado para tener una alta eficiencia computacional. Está escrito en C y puede aprovechar las ventajas de los procesadores multinúcleo. La biblioteca de OpenCV contiene más de 500 funciones que abarcan muchas áreas de la visión artificial. También tiene una librería de aprendizaje automático (MLL, Machine Learning Library) destinada al reconocimiento y agrupación de patrones estadísticos.

¹⁰<http://opencv.org/>

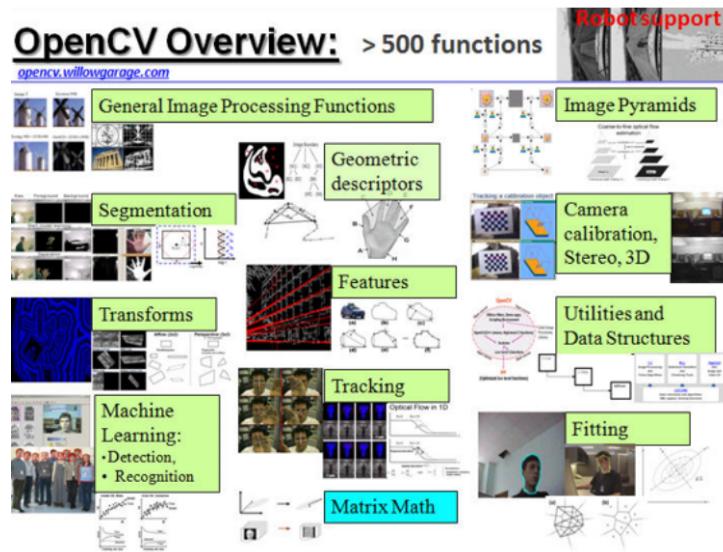


Figura 3.14: Funciones de OpenCV

OpenCV está compuesto por numerosas librerías con las cuales podemos manejar estructuras de datos, detectar bordes y esquinas, escalar o rotar imágenes, modificar el espacio de color de una imagen, realizar emparejamiento, detectar líneas y círculos, tratar objetos en 3D, crear ventanas y asociar eventos a dichas ventanas, etc. Incorpora funciones básicas para modelar el fondo, sustraer dicho fondo, generar imágenes de movimiento MHI (Motion History Images), etc. Además, incluye funciones para determinar dónde hubo movimiento y en qué dirección.

Desde su aparición OpenCV ha sido usado en numerosas aplicaciones. Hay una gran cantidad de empresas y centros de investigación que emplean estas técnicas como IBM, Microsoft, Intel, SONY, Siemens, Google, Stanford, MIT, CMU, Cambridge e INRIA.

En este trabajo se ha empleado la versión 3.3.1 de OpenCV en Python. Esta librería se empleará para realizar todo lo relacionado con el tratamiento de imágenes. Con ella se extraerán datos que puedan emplearse a la hora de tomar decisiones para que el coche funcione correctamente.

3.4.6. PyQt

PyQt [35] [36] es un conjunto de enlaces Python para el conjunto de herramientas Qt, las cuales se emplean para el desarrollo de interfaces gráficas. Fue desarrollado por Riverbank Computing Ltd y es soportado por Windows, Linux, Mac OS/X, iOS y Android.

Qt es un entorno multiplataforma orientado a objetos desarrollado en C++ que permite desarrollar interfaces gráficas e incluye *sockets*, hilos, Unicode, bases de datos SQL, etc. PyQt combina todas las ventajas de Qt y Python, pues permite emplear todas las funcionalidades ofrecidas por Qt con un lenguaje de programación tan sencillo como Python.

En este proyecto se ha empleado la versión 5 (en concreto la versión 5.5.1) de PyQt. PyQt5 es un conjunto de enlaces Python para Qt5, disponible en Python 2.x y 3.x. Tiene más de 620 clases y 6000 funciones y métodos. PyQt5 dispone de una licencia dual, es decir, los desarrolladores pueden elegir entre una licencia GPL (General Public Licence) o una licencia comercial.

La interfaz gráfica del componente académico creado en este proyecto está escrita usando PyQt. Las clases de PyQt5 se dividen en ciertos módulos, tales como QtCore, QtGui, QtWidgets, QDom, QSql, etc. En este proyecto se ha hecho uso de los siguientes módulos:

- QtCore: contiene las funcionalidades principales que no tienen que ver con la GUI. Este módulo se emplea para trabajar con archivos, diferentes tipos de datos, hilos, procesos, url, etc.
- QtGui: contiene clases para el desarrollo de ventanas, gráficos 2D, imágenes y texto.
- QtWidgets: dispone de clases que proporcionan un conjunto de elementos de interfaz de usuario para crear GUIs clásicas de escritorio.
- QtSvg: proporciona clases para mostrar el contenido de archivos Scalable Vector Graphics (SVG). SVG es un lenguaje para describir gráficos bidimensionales y aplicaciones gráficas en XML.

3.4.7. Keras framework

Keras¹¹ es un *framework* de alto nivel para redes neuronales, escrito en Python y capaz de correr sobre los *frameworks* TensorFlow, CNTK, o Theano. Fue desarrollado para facilitar la experimentación rápida.

Keras fue desarrollado con el fin de que la implementación de modelos de aprendizaje profundo fuera lo más fácil y rápido posible para la investigación y el desarrollo.

Este *framework* se ejecuta en Python 2.7-3.6, y es posible ejecutarlo tanto en CPU como en GPU. Keras se liberó bajo la licencia permisiva del MIT [37], y fue desarrollado y mantenido por François Chollet, un ingeniero de Google que utiliza cuatro principios:

- Facilidad de uso: Keras es una API diseñada basándose en la experiencia del usuario, es decir, ofrece un API consistente y simple, proporciona comentarios claros y procesables en caso de error del usuario.
- Modularidad: Un modelo se entiende como una secuencia o un gráfico de módulos independientes, totalmente configurables, que se pueden conectar con la menor cantidad de restricciones posible. En concreto, las capas neuronales, las funciones de coste, los optimizadores, los esquemas de inicialización, las funciones de activación y los esquemas de regularización son módulos independientes que se pueden combinar para crear nuevos modelos.
- Fácil extensibilidad: Los nuevos módulos son fáciles de agregar, y los módulos existentes proporcionan amplios ejemplos. La posibilidad de crear fácilmente nuevos módulos permite una extensibilidad total, lo que hace que Keras sea adecuado para la investigación avanzada.
- Trabajo con Python: No hay archivos de configuración de modelos separados en un formato declarativo, sino que los modelos se describen en el código de Python, facilitando la depuración de código y permitiendo la extensibilidad.

La versión principal utilizada en este proyecto es Keras 2.2.4, y se ha ejecutado sobre TensorFlow. Keras ha sido empleado para entrenar e implementar diferentes arquitecturas

¹¹<https://keras.io/>

de redes neuronales.

En las próximas subsecciones, se analizan los elementos principales que forman una red neuronal convolucional y una red neuronal recurrente (LSTM) construida con Keras, comenzando con el objeto modelo.

3.4.7.1. Modelos

En Keras cada red neuronal se define como un modelo, es una forma de organizar las capas. La clase de modelo más simple es el modelo *Sequential*, que es una pila lineal de capas. Es posible construir arquitecturas más complejas, aunque se debe utilizar la API funcional de Keras, que permite crear gráficos de capas arbitrarios.

Los modelos *Sequential* tienen diferentes métodos, y algunos son imprescindibles para el proceso de aprendizaje, como son:

- **.compile()**: Configura el modelo para entrenamiento. Los principales argumentos son los siguientes:
 - *optimizer*: Nombre del optimizador que actualizará los valores de los pesos durante el entrenamiento para minimizar la función de pérdida. Existen diferentes optimizadores como Adadelta, SGD, RMSProp, Adagrad, Adamax o Adam. En las diferentes redes implementadas se ha empleado el optimizador Adam [38].
 - *loss*: Nombre de la función de coste que mide la diferencia entre la predicción y la etiqueta real. En este proyecto en las redes de clasificación se ha empleado *categorical cross-entropy*, también conocida como *log loss*. Esta función es muy utilizada en problemas de clasificación multiclase. Esta función devuelve la entropía cruzada entre una distribución aproximada q y una distribución verdadera p , y sigue la siguiente fórmula [39]:

$$H(p, q) = -\sum_x p(x) \log(q(x)) \quad (3.1)$$

En las redes neuronales de regresión se ha empleado como función de coste *Mean Squared Error (MSE)*, que da una medida de cómo de lejos están las

medidas predichas de las reales, pero acentúa los errores grandes. La fórmula de MSE es:

$$MSE = \frac{1}{N} \sum_{j=1}^N (y_j - \hat{y}_j)^2 \quad (3.2)$$

Otra posible función a emplear es *Mean Absolute Error (MAE)*, que nos da una medida de cuán lejos están las medidas predichas de las medidas reales.

- *metrics*: Nombre de las funciones que se emplean para medir el rendimiento del modelo durante el entrenamiento y el *test*. En este proyecto las métricas empleadas son *accuracy*, MSE y MAE. *Accuracy* es el número de predicciones correctas realizadas por el modelo sobre todo tipo de predicciones realizadas en los modelos de clasificación. La fórmula para *accuracy* es la siguiente:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (3.3)$$

donde TP es *True Positive* (casos en los que la clase real del dato es 1 y la clase predicha es 1), TN es *True Negative* (la clase del dato es 0 y la predicha es 0), FP es *False Positive* (la clase real es 0 y la clase predicha es 1), y FN es *False Negative* (la clase real es 1 y la predicha es 0).

Mean Absolute Error (MAE), como hemos dicho anteriormente es una medida del error de la predicción, y sigue la siguiente fórmula:

$$MAE = \frac{1}{N} \sum_{j=1}^N |y_j - \hat{y}_j| \quad (3.4)$$

- **.fit()**: Entrena el modelo para un número dado de épocas (iteraciones en un conjunto de datos). Los siguientes argumentos son necesarios:
 - *x*: Muestras de entrenamiento. Se debe definir como un *Numpy array* o una lista de *Numpy arrays*.
 - *y*: Etiquetas de entrenamiento. Se debe definir como un *Numpy array* o una lista de *Numpy arrays*.
 - *batch_size*: Número de muestras que serán evaluadas antes de actualizar los pesos. Si no se especifica, *batch_size* será por defecto 32.

- *epochs*: Número de iteraciones sobre todo el conjunto de datos.
 - *callbacks*: Lista de *callbacks* (ver la subsección 3.4.7.3) que se aplican durante el entrenamiento y la validación.
 - *validation_split o validation_data*: En Keras hay dos posibilidades para establecer el conjunto de validación: *validation_split* o *validation_data*. *validation_split* es la fracción de los datos de entrenamiento (número entre 0 y 1) que se utilizarán como datos de validación. *validation_data* es una tupla de valores sobre la cual se debe evaluar la pérdida y cualquier métrica del modelo al final de cada época. El modelo no tendrá en cuenta el conjunto de validación al entrenar el modelo.
 - *shuffle*: booleano que determina si se barajan los datos de entrenamiento o no. Si los datos no son barajados durante el entrenamiento, las muestras de una misma clase pueden aparecer de forma consecutiva. En este caso, el modelo tendrá que aprender las características de una determinada clase. Cuando el modelo empieza a ver muestras de la siguiente clase, se ajusta a los nuevos datos y se olvida de la característica aprendida anteriormente. Si los datos están ordenados por clases, este proceso sigue y conduce a un peor resultado.
- **.predict()**: Genera predicciones de salida para las muestras de entrada.
 - **.evaluate()**: Devuelve el valor de *loss* y los valores de *metrics* para el modelo en *test*.
 - **.save()**: Guarda un modelo en un solo archivo Hierarchical Data Format version 5 (HDF5), que contendrá la arquitectura del modelo, los pesos del modelo, la configuración de entrenamiento, y el estado del optimizador (permite reanudar el entrenamiento por donde se quedó).
 - **.load_model()**: Carga un modelo desde un archivo HDF5.

3.4.7.2. Capas

Como hemos visto anteriormente, los modelos se componen de un conjunto de capas. Estas capas se añaden al modelo empleando el método *.add()* de Keras. Dentro de este método se define el tipo de capa y los parámetros de cada capa. Existen diferentes tipos

de capas en Keras, pero solamente veremos las empleadas en el proyecto, que ya han sido descritas con más detalle en la Sección 1.4.3.

- *Convolutional layer*: Es la capa principal de una red CNN, como vimos en la Sección 1.4.3, donde se explica con detalle su funcionamiento. Keras proporciona distintos tipos de capas convolucionales en función de las dimensiones de los datos de entrada: *Conv1D*, *Conv2D*, y *Conv3D*. En nuestro proyecto emplearemos la capa *Conv2D*, ya que nuestros datos de entrada son imágenes.

Los argumentos principales que hay que definir en una capa convolucional en Keras son:

- *filters*: Número de filtros. Las capas *Conv2D* intermedias aprenderán más filtros que las primeras capas *Conv2D*, pero menos filtros que las capas más cercanas a la salida.
- *kernel_size*: Especifica la anchura y altura de los filtros. Puede ser un solo entero para especificar el mismo valor para todas las dimensiones espaciales, o puede ser una tupla o lista de 2 enteros.
- *strides*: Entero o tupla/lista de 2 enteros, que especifica cuántos píxeles debe desplazarse el filtro antes de aplicar la siguiente convolución. El valor por defecto es 1.
- *padding*: Puede ser *valid* o *same*. Si se emplea *valid*, no se aplica relleno, dando lugar a una salida con una dimensión más pequeña que la entrada. Sin embargo, si empleamos *same*, la entrada se llenará con ceros para dar lugar a una salida que conserve las dimensiones de la entrada. El valor por defecto es *valid*.
- *BatchNormalization Layer*: Normaliza las activaciones de la capa anterior en cada lote, es decir, aplica una transformación que mantenga la activación media cerca de 0 y la desviación estándar de activación cerca de 1. El argumento más importante es *axis*, que indica el eje que debe normalizarse. Por ejemplo, después de una capa *Conv2D* donde establecemos *data_format = "channels_first"*, el valor de *axis* será 1. Mientras que si establecemos *data_format = "channels_last"*, el valor de *axis* será -1.

- *Pooling layer*: Como vimos en la Sección 1.4.3, esta capa reduce las dimensiones espaciales del volumen de entrada, reduce el coste computacional, y evita el sobreajuste.

En Keras, dependiendo de las dimensiones de entrada y la operación empleada, existen diferentes capas de *pooling*: MaxPooling1D, MaxPooling2D, MaxPooling3D, AveragePooling1D, AveragePooling2D, AveragePooling3D, GlobalMaxPooling1D, GlobalMaxPooling1D, etc. En Keras, los principales argumentos necesarios para definir estas capas son:

- *pool_size*: Factor por el cual se reduce la escala (vertical, horizontal), donde el factor es un número entero o una tupla de 2 enteros. Si solo se especifica un número entero, se utilizará la misma longitud de ventana para ambas dimensiones. Por ejemplo, si empleamos un *pool_size* de (2, 2) se reducirá a la mitad la entrada en ambas dimensiones espaciales.
- *strides*: Indica cuántos píxeles debe desplazarse la ventana antes de aplicar la siguiente operación. Su valor es un entero, o una tupla de 2 enteros, o *None*.
- *Dense layer*: En Keras, las capas *fully-connected* se definen como *Dense layers*. El argumento principal de este tipo de capa es:
 - *units*: número de neuronas.
- *Activation layer*: En Keras, una función de activación se puede declarar como una capa en sí misma o como un argumento dentro del método *.add()* de la capa anterior. Keras proporciona varias funciones de activación, como *sigmoid*, *linear*, *ReLU* y *softmax*. En este proyecto se han empleado las funciones de activación:
 - ReLU: es una función de activación no lineal, donde la salida es igual a 0 si la entrada es menor que 0, y si la entrada es mayor que 0 la salida es igual a la entrada. La función *ReLU* sigue la siguiente fórmula:

$$g(x) = \max(0, x) \quad (3.5)$$

En la Figura 3.15 se muestra la función de activación *ReLU* en el intervalo [-10, 10].

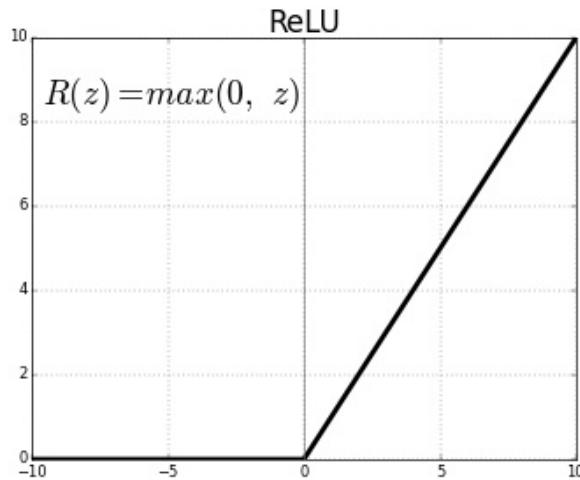


Figura 3.15: Función de activación *ReLU*

- Softmax: Esta función de activación es muy empleada en la capa de salida de los problemas de clasificación. La función *softmax* escala las salidas de cada unidad para que estén entre 0 y 1, al igual que una función sigmoide, pero también divide cada salida de tal manera que la suma total de las salidas sea igual a 1.

La función softmax se puede expresar matemáticamente como vemos a continuación, donde z es un vector de las entradas a la capa de salida, y j indexa las unidades de salida, entonces $i = 1, 2, \dots, K$.

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad \text{for } j = 1, \dots, K. \quad (3.6)$$

- *Flatten layer*: Aplana la entrada, es decir, modifica sus dimensiones. Por ejemplo, convierte los elementos de una matriz de imágenes de entrada en un array plano. Esta capa no afecta al *batch_size*.
- *Dropout layer*: Esta capa consiste en establecer aleatoriamente una tasa de fracción (*rate*) de unidades de entrada en 0 en cada actualización durante el tiempo de entrenamiento, lo que ayuda a evitar el sobreajuste. El argumento principal de esta capa es *rate*, que es la tasa de fracción mencionada anteriormente. El valor de *rate* debe estar entre 0 y 1.

- *LSTM layer*: Implementa una capa Long Short-Term Memory (LSTM). Esta capa tiene algunos argumentos esenciales:
 - *units*: Número de celdas LSTM.
 - *return_sequences*: Booleano que indica si se debe devolver la última salida en la secuencia de salida o la secuencia completa. Si se establece a *True* se devuelve la secuencia completa.

3.4.7.3. *Callbacks*

Un *callback* es un conjunto de funciones que se aplicarán en determinadas etapas del proceso de entrenamiento. Se puede emplear los *callbacks* para obtener un vistazo de los estados internos y las estadísticas del modelo durante el entrenamiento. En este proyecto se han empleado los siguientes *callbacks*:

- *.ModelCheckpoint()*: Guarda el modelo y sus pesos después de cada época. Es posible configurar *ModelCheckpoint* para que sobreescriba el modelo solamente si una métrica que indicamos ha mejorado respecto al mejor resultado anterior. De esta forma se guarda la mejor versión del modelo.
- *.TensorBoard()* [40]: Es un conjunto de herramientas de visualización proporcionado por *TensorFlow*, que facilita la comprensión, la depuración y la optimización de los programas. Se puede emplear TensorBoard para visualizar el gráfico proporcionado por TensorFlow, trazar métricas cuantitativas sobre la ejecución del gráfico, así como histogramas de activación para las diferentes capas en el modelo, y mostrar datos adicionales como las imágenes que pasan a través de él.
- *.CSVLogger()*: Escribe un archivo de registro CSV que contiene información sobre las épocas, el *accuracy* y *loss* en el disco, dando la posibilidad de inspeccionarlo más tarde. De esta forma se pueden crear gráficos a partir de estos datos o mantener un registro del proceso de entrenamiento del modelo a lo largo del tiempo.

3.4.8. Formato de archivo HDF5

Hierarchichal Data Format version 5 (HDF5) [41] [42] es una librería de propósito general y al mismo tiempo un formato de ficheros para el almacenamiento de datos

científicos. HDF5 fue creado con el fin de facilitar el trabajo a los ingenieros y científicos que trabajan en entornos con altas prestaciones y con un uso masivo de datos. Keras emplea el formato de archivo HDF5 para guardar modelos y leer conjuntos de datos. La tecnología HDF5 incluye:

- Un modelo de datos versátil que puede representar objetos de datos complejos y una gran variedad de metadatos.
- Un formato de archivo completamente portable sin límite en el número o tamaño de los objetos de datos de una colección.
- Una biblioteca software que se ejecuta en diversas plataformas computacionales como ordenadores portátiles o sistemas masivamente paralelos. Además, implementa una API de alto nivel con interfaces C, C++, Fortran 90 y Java.
- Un gran conjunto de funciones de rendimiento que permiten optimizar el tiempo de acceso y el espacio de almacenamiento.
- Herramientas y aplicaciones para manejar, manipular, visualizar y analizar datos.
- El modelo de datos HDF5, el formato de archivo, la biblioteca y las herramientas son de código libre.

En este trabajo se han empleado los archivos HDF5 para guardar los modelos de las diferentes redes. Para tratar con archivos HDF5 se emplea la biblioteca h5py ¹² para Python.

¹²www.h5py.org

Capítulo 4

Infraestructura desarrollada

En este capítulo se explica la infraestructura software en la que nos hemos apoyado para desarrollar el proyecto. Se definirán el coche y los circuitos empleados tanto para el entrenamiento de las redes como para el *test*. Además, se explica cómo se ha creado el conjunto de datos a partir de un piloto manual basado en visión artificial, así como el nodo piloto creado para facilitar el uso de las redes neuronales en conducción.

4.1. Objetivo de *Follow line*

El propósito de este proyecto es que un coche autónomo sea capaz de conducir en diferentes circuitos mediante distintas redes neuronales que son capaces de aprender control visual. El coche dispondrá de una cámara que le proporciona información de su entorno, y además posee un actuador de movimiento basado en velocidad lineal y velocidad de giro.

El vehículo, como hemos mencionado, debe ser capaz de aprender determinadas acciones. Para que el coche pueda aprender es necesario disponer de una serie de datos, por lo que se ha creado un conjunto de entrenamiento que se verá en la Sección 4.6. Con el fin de grabar este conjunto de datos se ha creado un piloto manual que es capaz de dar vueltas alrededor del circuito de forma autónoma. Este piloto tiene un algoritmo, basado en visión, que veremos en la Sección 4.5.

El piloto se ha creado como solución a la práctica *Follow line* de JdeRobot Robotics-Academy¹, donde el objetivo es realizar un control PID y completar una vuelta de un

¹<https://jderobot.org/Robotics-Academy>

circuito Fórmula 1 basándose en el análisis de una imagen proporcionada por el vehículo.

4.2. Modelo de coche

El coche que se ha empleado en este proyecto es un modelo de robot creado por los desarrolladores de JdeRobot mediante un programa de modelado 3D (como pueden ser Blender, SketchUp, etc). El modelo de coche empleado es *f1ROS*, el cual es un robot creado para poder moverse de forma autónoma o teledirigida por un escenario. El modelo *f1ROS* posee tres sensores: un láser y una cámara, que le permiten extraer datos del entorno que le rodea; así como sensores de odometría que permiten conocer la velocidad del vehículo. Además, cuenta con motores que le permiten moverse por el escenario de la manera adecuada.

El modelo *f1ROS* es empleado en las prácticas *Follow line* y *Obstacle avoidance with VFF*. Este coche tiene unas dimensiones pequeñas, ya que mide aproximadamente 0.8 metros de largo, 0.4 metros de ancho y posee una altura de 0.3 metros. Este modelo pesa 10 kg y se puede ver en la figura 4.1.

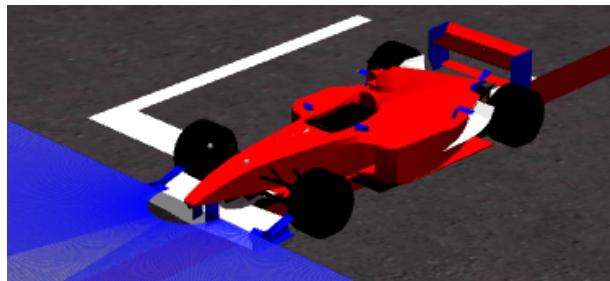


Figura 4.1: Modelo f1ROS

El modelo en la parte frontal izquierda tiene una cámara instalada que captura imágenes, que son una gran fuente de información para que el coche pueda recorrer el circuito. Los *plugins* de la cámara tienen soporte para una cámara conectada por USB con una tasa de refresco de las imágenes de 20 fps, que permite al coche obtener imágenes a una velocidad suficiente para analizar la situación en la que se encuentra.

Las imágenes recogidas por la cámara periódicamente tienen unas dimensiones de 640 x 480 píxeles, y se obtienen en formato RGB en crudo. Las imágenes recogidas por el *plugin* son capturadas, por ejemplo, mediante el nodo de ROS, que a través de un API bastante sencillo, permite que se pueda trabajar con las imágenes capturadas y que se aplique un procesado a la imagen.

Esta cámara será el sensor que se empleará fundamentalmente tanto en el piloto manual creado como en las redes neuronales de conducción que se han empleado. Además, el modelo de coche cuenta con un sensor láser que recoge un array de 180 medidas, pudiendo medir distancia alrededor de 180 grados en milímetros. Este sensor no ha sido empleado en este proyecto, pero está disponible en el modelo por si se necesitará en otros casos.

Además, este coche cuenta con sensores de posición (sensores de odometría), ya que son una gran fuente de información para los algoritmos en los que se apoya el pilotaje de nuestro vehículo. La odometría se emplea para estimar la posición (x, y, orientación) de un robot móvil en todo momento. Por lo tanto, emplearemos este tipo de sensores para estimar la posición del Fórmula 1 en el mundo de Gazebo, y a partir de este dato estimar su velocidad. Estos sensores de odometría estiman la posición de las ruedas izquierda y derecha en un intervalo de tiempo concreto. La plataforma JdeRobot apoyándose en ROS aísla de la complejidad de los sensores de odometría, facilitando una variable que contiene la posición (x, y, orientación) en el mundo.

En este proyecto se han empleado tres *plugins* de ROS con el fin de dotar de movimiento al modelo, odometría, captación de imágenes y captación de datos del sensor láser. Los *plugins* son *drivers* de los sensores y actuadores del modelo, y permiten crear una conexión e intercambiar mensajes de ROS (*topics*) con diferentes aplicaciones. Desde las aplicaciones se puede acceder fácilmente a ellos, empleando tres bibliotecas:

- `libgazebo_ros_camera`: Los componentes harán uso de este *plugin* para captar imágenes.
- `libgazebo_ros_laser`: Este *plugin* será usado por los componentes para obtener información de la distancia que hay hasta los obstáculos.

- libgazebo_ros_planar_move: es el *plugin* que permite dotar al coche de velocidad, tanto velocidad de tracción como velocidad de rotación. Además, proporciona la posición del coche en tiempo real.

4.3. Modelos de circuitos

El objetivo de este proyecto es que nuestro Fórmula 1 sea capaz de conducir de forma autónoma por un circuito, por lo que tendremos que crear diferentes entornos (circuitos) donde se moverá. Para facilitar la solución de la práctica *Follow line*, y por tanto la solución del piloto manual, los circuitos tienen una línea roja pintada en el suelo.

Los modelos de circuitos fueron creados con una herramienta de modelado 3D (Blender, SketchUp, etc). La mayoría de los circuitos corresponden con circuitos de grandes dimensiones. En estos circuitos no veremos muchas gradas alrededor, ni público u otros elementos habituales de los circuitos reales, sino que se ha simplificado su creación para que sea rápido en la ejecución del simulador. Los mundos que tienen muchos detalles son más costosos computacionalmente de simular. Lo que podremos ver en estos circuitos son elementos propios de carreteras como son rectas, curvas simples o pronunciadas, una línea de salida, paredes que evitan que el coche se salga del recorrido, y en algunos casos una grada y césped de adorno.

El primer modelo de circuito empleado se llama *pistaSimple*. Es un circuito de carreras de gran tamaño, que consta de una carretera con una línea roja en el suelo, una línea de salida, y las paredes para evitar que el coche se salga del circuito. Este circuito no posee ningún elemento de adorno, ya que haría que la simulación fuera muy lenta. Este modelo se puede observar en la Figura 4.2.

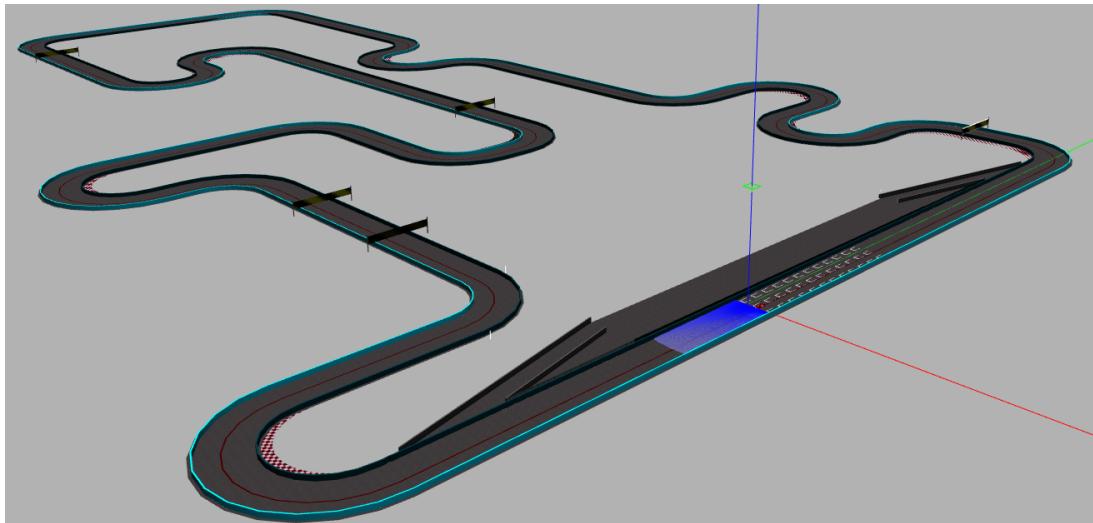


Figura 4.2: Modelo pistaSimple

El segundo modelo de circuito se denomina *monacoLine* y simula el circuito de carreras de Mónaco (Principado de Mónaco), también conocido como Montecarlo. Este modelo consta del circuito en sí mismo, así como línea de salida, paredes que rodean el circuito, césped de adorno, y una grada pequeña. El modelo *monacoLine* se puede ver en la Figura 4.3.

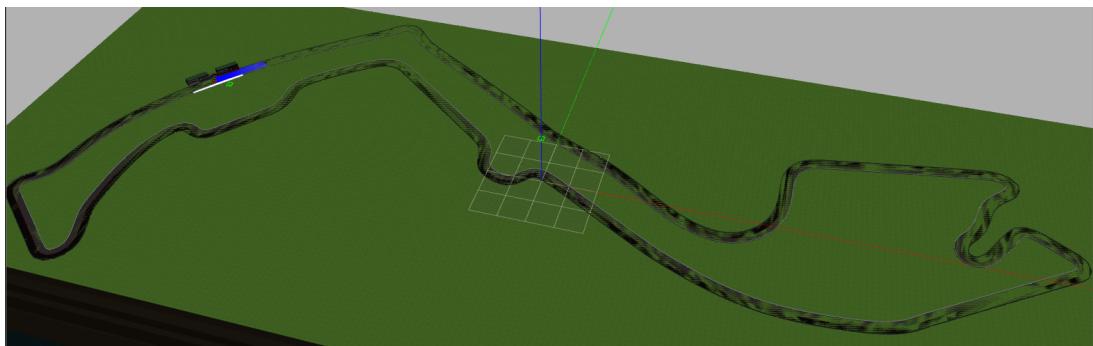


Figura 4.3: Modelo monacoLine

El tercer modelo simula el circuito de carretas Nürburgring (Alemania) acortado, llamado *nurburgrinLine*. Se ha modelado el circuito con una línea de salida, la carretera, paredes para evitar que el robot se salga del recorrido, una grada pequeña y césped de adorno. Este modelo se puede observar en la Figura 4.4.

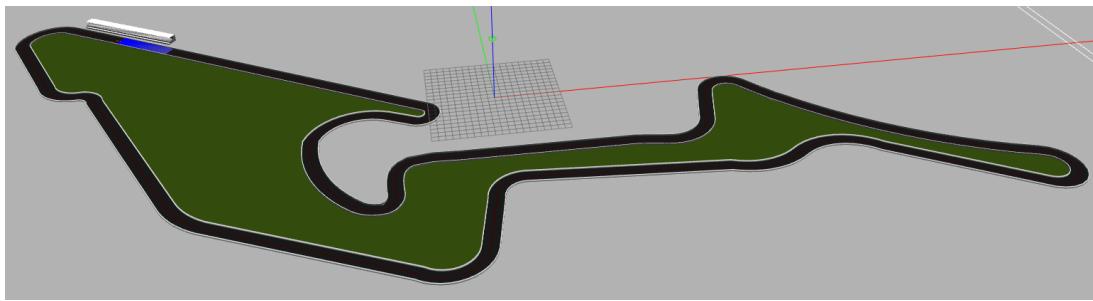


Figura 4.4: Modelo nurburgrinLine

El cuarto modelo de circuito, llamado *curveGP*, simula un circuito que no tiene ninguna curva, solamente tiene curvas, tanto curvas leves como abruptas. Este circuito es de gran tamaño, y por este motivo únicamente simula el circuito, y paredes que rodean al mismo, no tiene ningún elemento de adorno. El modelo *curveGP* se puede ver en la Figura 4.5.

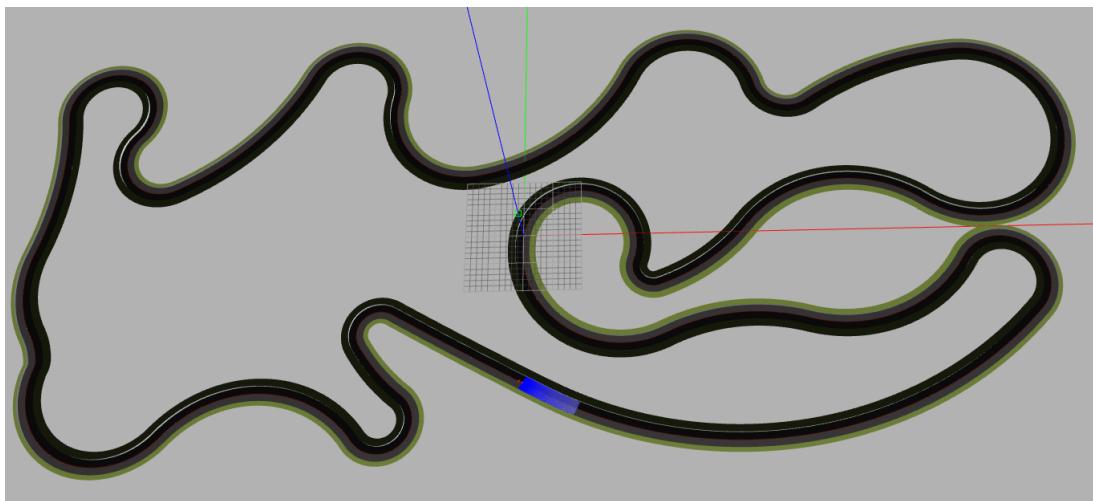


Figura 4.5: Modelo curveGP

El quinto modelo de circuito se denomina *pista_simple*. Este circuito es el más corto de todos, aunque únicamente simula el circuito, y paredes que rodean al mismo, no tiene ningún elemento de adorno. En la Figura 4.6 se puede ver el modelo *pista_simple*.

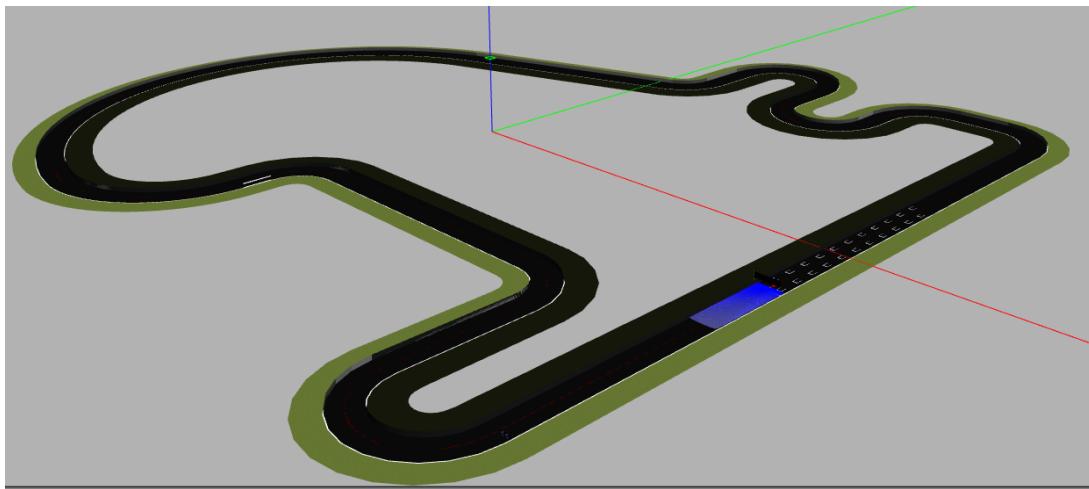


Figura 4.6: Modelo pista_simple

Los modelos de circuitos *pistaSimple*, *monacoLine*, *nurburgrinLine* son empleados para el entrenamiento y el *test* de las redes neuronales de extremo a extremo. En los modelos con dependencias temporales (como LSTM) se añade el modelo *curveGP* tanto para entrenamiento como para *test*. Este modelo es empleado para adquirir más datos de curvas, ya que en los anteriores circuitos teníamos muchos más datos de rectas que de curvas, y es necesario que las redes posean más datos de este tipo para que sean capaces de aprender ciertos comportamientos. El modelo *pista_simple* se emplea únicamente para *test*.

4.4. Mundo de Gazebo

Los mundos que se simulan con Gazebo son mundos 3D. Estos mundos se cargan en ficheros con extensión .world, que no son más que ficheros XML definidos en el lenguaje SDF. Este lenguaje contiene una descripción completa de todos los elementos que tiene el mundo y los robots.

Se ha creado un mundo en Gazebo para cada circuito compuesto por uno de los cinco modelos de circuito y el modelo del coche (*f1ROS*). Los archivos del mundo de cada circuito son iguales en todos los casos, las únicas diferencias son que el nombre del modelo de circuito cambia, y además la posición del modelo del coche también cambia, ya que en cada circuito la línea de salida está en un lugar diferente. Por ejemplo, el archivo

f1-simple-circuit.world (mundo del circuito *pistaSimple*) tiene el siguiente aspecto:

```

<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <scene>
      <grid>false</grid>
    </scene>
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://pistaSimple</uri>
    <pose>-160 17 0 0 0 0</pose>
    </include>
    <include>
      <uri>model://f1ROS</uri>
      <pose>1 0 0 0 0 -1.57</pose>
    </include>

    <scene>
      <sky>
        <clouds>
          <speed>12</speed>
        </clouds>
      </sky>
    </scene>
  <light name='user_directional_light_0' type='directional'>
    <pose frame=''>0 0 10 0 -0 0</pose>
    <diffuse>0.8 0.8 0.8 1</diffuse>
    <specular>0.2 0.2 0.2 1</specular>
    <direction>0.1 0.1 -0.9</direction>
    <attenuation>
      <range>1000</range>
    </attenuation>
  </light>
</world>
</sdf>
```

```
<constant>0.9</constant>
<linear>0.01</linear>
<quadratic>0.001</quadratic>
</attenuation>
<cast_shadows>1</cast_shadows>
</light>

</world>
</sdf>
```

Además de este fichero de configuración, es necesario crear un archivo con extensión .launch, que arranca los *plugins* y *drivers* de ROS-Kinetic. En este fichero es necesario indicar a Gazebo algunos argumentos como el nombre del fichero de configuración con el escenario (archivo del mundo), se establece el tiempo que empleará el escenario (como por ejemplo tiempo simulado), la posibilidad de lanzar una GUI, y algunas opciones de depuración. Es necesario crear un archivo .launch por cada modelo de circuito o escenario. Todos estos archivos serán iguales, excepto que en el argumento “world_name” se modificará el valor del archivo .world empleado. Por ejemplo, para emplear el archivo de configuración *f1-simple-circuit.world*, se ha creado el archivo *f1.launch* que se puede ver a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <!-- We resume the logic in empty_world.launch, changing only the name
      of the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="f1-simple-circuit.world"/> <!-- Note: the
      world_name is with respect to GAZEBO_RESOURCE_PATH environmental
      variable -->
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
```

```
<arg name="verbose" default="false"/>  
</include>  
</launch>
```

4.5. Piloto manual

Como hemos mencionado el objetivo del proyecto es que un vehículo sea capaz de aprender determinadas acciones que le permitan conducir de forma autónoma. Para lograr este objetivo es necesario disponer de una serie de datos, por lo que se ha creado un conjunto de datos (Sección 4.6). Con el fin de grabar este conjunto de datos se ha creado un piloto manual que es capaz de dar vueltas alrededor del circuito de forma autónoma.

El piloto manual [43] [44] se ha creado como solución a la práctica *Follow line* de JdeRobot Robotics-Academy [45], donde el objetivo es programar al robot *f1ROS* para que sea capaz de navegar rápidamente por un circuito de Fórmula 1 siguiendo una línea roja pintada en el suelo. Para ello, el coche dispone de una cámara en la parte frontal izquierda y unos motores a los que se envían órdenes de velocidad (velocidad de tracción y velocidad de rotación).

El coche tendrá una parte perceptiva y una parte de control. En la parte perceptiva el vehículo deberá extraer la información relevante de las imágenes: dónde está la línea roja, si está en una recta o en una curva, si ha perdido la línea roja, etc. En la parte de control, el coche deberá hacer un control reactivo PID o bien un control PD, que sea capaz de corregir la velocidad de giro para mantener al coche por encima de la línea.

Antes de explicar la solución del piloto manual, es necesario saber qué es un control PID. Un control PID consta de tres parámetros distintos: el proporcional, el integral y el derivativo. Para explicar mejor cada una de las partes tomaremos como ejemplo una calefacción con termómetro, que es un ejemplo clásico de control PID. En este ejemplo el sensor es el termómetro y el objetivo es obtener una determinada temperatura, por lo que tendremos un error que será la desviación entre la temperatura observada y la temperatura deseada. El error tendrá magnitud y signo, y el controlado PID deberá dar órdenes a la calefacción o el aire acondicionado para conseguir minimizar el error y obtener la

temperatura deseada.

La parte proporcional (P) manda a los actuadores una corrección proporcional al error, de forma que si el error es pequeño se corrige suavemente y si es grande la corrección es mayor. De no existir desviación no se modifica la temperatura. El control P tiende a conseguir que el sistema obtenga la situación deseada, pero a veces lo logra con muchas oscilaciones. Una posibilidad para suavizar las oscilaciones es emplear la parte Derivativa (D), que realiza una corrección proporcional a la derivada del error. Si el error está creciendo el control D aumenta la corrección, y si por el contrario está disminuyendo, suaviza la corrección.

En algunos casos no es suficiente con un control PD, ya que puede que se estabilice en una situación no deseada pese a las correcciones. Para poder eliminar estos *offsets* se puede emplear la parte Integral (I). Este tipo de control actúa en función de la acumulación del error, es decir, si ha pasado bastante tiempo sin que el error tienda a cero, entonces aumenta la corrección por parte del control.

El control Proporcional-Integral-Derivativo (PID) permite anular un determinado error de manera reactiva. Dependiendo de la aplicación concreta podremos denominar error a una cosa u otra. En nuestro caso podemos considerar error (desviación) a la diferencia horizontal en píxeles entre el centro de la línea cuando el robot está realmente recto sobre ella (este es el valor objetivo) y el centro de la línea observado en el instante actual. En el caso de que el error sea cero, es que el coche estará completamente centrado sobre la línea.

Una vez que ya se ha explicado de forma general la parte perceptiva y la parte de control PID, ya podemos hablar del procesado de imagen empleado en el piloto, así como el control realizado y las herramientas necesarias.

Con el fin de crear el piloto manual se empleará el simulador Gazebo, la plataforma JdeRobot y la librería OpenCV ampliamente usada para procesamiento de imágenes. Para poder ver el piloto en funcionamiento, deberemos lanzar primero el simulador Gazebo con el mundo correspondiente, que contendrá el coche fórmula 1 y el circuito que queremos usar; y después ejecutar nuestro código para ver cómo se comporta el coche en el circuito.

CAPÍTULO 4. INFRAESTRUCTURA DESARROLLADA

En un terminal lanzamos Gazebo:

```
roslaunch /opt/jderobot/share/jderobot/gazebo/launch/f1.launch
```

En otro terminal se debe lanzar la aplicación académica:

```
python2 ./follow_line.py follow_line_conf.yml
```

El coche posee una cámara situada en la parte frontal izquierda como hemos visto anteriormente. Las imágenes que capta la cámara se pueden obtener con la siguiente instrucción:

```
self.getImage()
```

Como hemos mencionado antes, el piloto automático debe dar una vuelta al circuito. Para ello, será necesario detectar la línea roja que está en el centro de la carretera, es decir, será necesario realizar una umbralización de esta línea. Para realizar esta umbralización, primero debemos transformar las imágenes al espacio de color HSV, ya que HSV es más robusto que RGB ante cambios de iluminación.

```
image_HSV = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
```

Tras realizar la transformación a HSV, se ha realizado una umbralización de la imagen empleando la función *cv2.inRange* de OpenCV. Este filtrado se hace en función al rango de valores de rojo de la línea de la carretera.

```
value_min_HSV = np.array([0, 235, 60])
value_max_HSV = np.array([80, 255, 255])
image_HSV_filtered = cv2.inRange(image_HSV, value_min_HSV, value_max_HSV)
```

Si mostramos la imagen tras el filtrado vemos que aparecen puntos en negro que pertenecen a la línea roja y no debería ser así. Esto ocurre en la línea de salida del circuito, ya que se puede ver que la línea parpadea y por lo tanto no tiene siempre el tono de rojo habitual. Este problema lo podemos ver en la Figura 4.7.

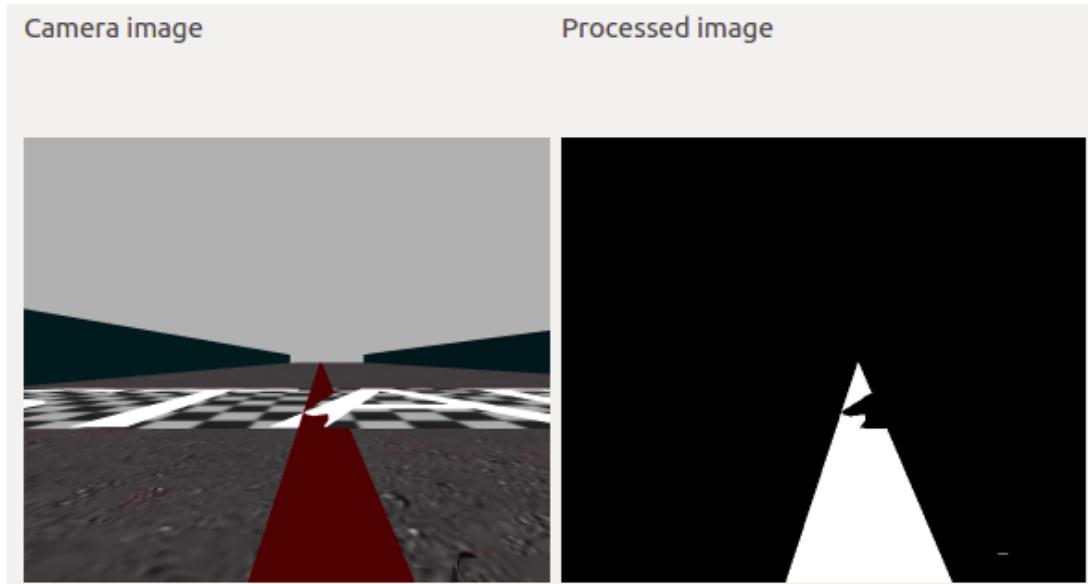


Figura 4.7: Filtrado de color

Para solventar este problema, se ha aplicado un cierre morfológico (primero hace una dilatación y después una erosión) empleando un elemento estructurante con forma de ellipse. Esta operación hace que los pixeles que aparecían en negro en la línea filtrada ahora pertenezcan a la línea filtrada, es decir, que aparezcan en blanco. Para llevar a cabo esta operación morfológica se emplean las siguientes líneas de código:

```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (31, 31))
image_HSV_filtered = cv2.morphologyEx(image_HSV_filtered, cv2.MORPH_CLOSE,
                                         kernel)
```

Se puede observar en la Figura 4.8 cómo después de aplicar el cierre morfológico no quedan huecos en la parte segmentada al filtrar la línea roja.

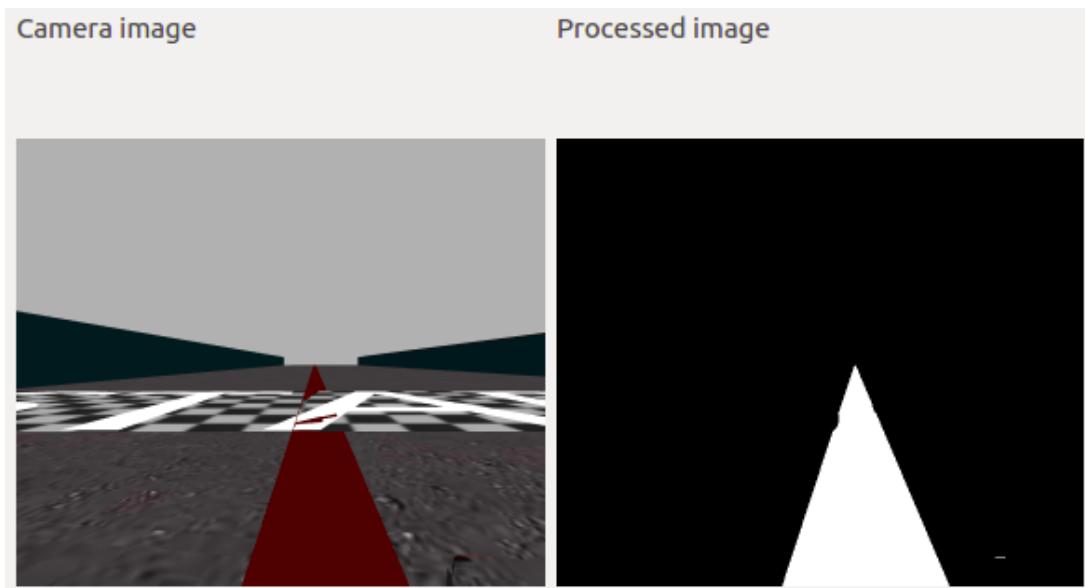


Figura 4.8: Filtrado de color con cierre

Tras obtener una imagen con la línea roja en blanco, se ha tenido en cuenta que si el coche circula sobre la carretera correctamente la línea debería aparecer más o menos centrada en la imagen. Si por el contrario, la línea aparece en la parte izquierda o derecha de la imagen, es debido a que el coche debería girar puesto que está un poco desviado de la línea roja. De esta forma se puede conocer el giro que debe realizar el coche.

Con el fin de analizar la información proporcionada por el filtrado de la imagen es necesario analizar alguna fila de la imagen filtrada. En el caso de que solamente analicemos una única fila de la imagen, no obtendremos suficiente información para saber si nos encontramos en recta o en curva. Por este motivo, se han analizado tres filas de la imagen con el fin de saber si el coche está situado encima de una recta o una curva, o si por otro caso se ha salido el coche de la línea. En nuestro caso analizamos las filas situadas en las posiciones $y_1 = 260$, $y_2 = 310$, e $y_3 = 350$. En estas líneas se calcula el centro de la línea roja, para lo cual se comprueban los valores de la imagen filtrada. Con estos centros (centro de cada fila) podremos saber si nos encontramos en recta o curva.

El siguiente paso es procesar esta información. El centro de la línea roja situado en la fila y_1 (260) será el único que no perderemos siempre que nos encontremos cerca de la línea roja, ya que es la posición que está situada más arriba. Los centros de la línea roja

situados en las filas y2 (310) e y3 (350) es posible que se pierdan al llegar a una curva grande. Si no encontramos la línea roja en la posición y1 es debido a que nos hemos salido de la línea roja. En esta situación el coche deberá ir hacia atrás y girar hacia el circuito para continuar su recorrido. Si la última vez que se ha visto la línea estaba a la izquierda de la imagen, quiere decir que el coche se ha salido hacia la derecha, y por tanto tendrá que girar a la izquierda y viceversa. Este caso se ha tenido en cuenta al elaborar el piloto manual.

La desviación o error que tenemos en cada instante se calcula respecto a la posición ideal. Esta posición ideal o de referencia puede no ser el centro de la imagen, ya que la cámara no está centrada en el vehículo y mirando exactamente la recta. En este caso, hay que analizar las imágenes cuando el coche está en recta y ver cuál es el centro de referencia (posición ideal). Haciendo algunas pruebas se ha determinado que la posición ideal es $x = 326$. Las desviaciones se calcularán respecto a este valor objetivo. En nuestro caso calculamos la diferencia entre la posición x de y1 (260) y la posición ideal. Esta es la desviación que tendremos en cuenta a la hora de hacer los diferentes controles PD que se llevan a cabo. Además, como hemos mencionado anteriormente, se tiene en cuenta un caso por si el coche se ha salido del circuito.

Otro de los casos que se han tenido en cuenta en el piloto manual es si el centro situado en la fila situada más abajo ($y3 = 350$) se ha perdido. Esto puede ocurrir si estamos en una curva grande. Si se da este caso se realizará un control PD adaptado a esta excepción. En función de la desviación que hay entre el centro situado en la fila de arriba (y1) y el centro situado en la fila de en medio (y2), se darán diferentes situaciones en las que se ajustará un PD diferente.

En el piloto manual se ha decidido emplear un control PD, ya que puede ser que el uso de un control P no sea suficiente, puesto que el coche puede oscilar sobre la línea roja. Por lo tanto, es mejor emplear un control PD para evitar estos vaivenes del coche. El control PD (definido por un control derivativo y un control proporcional) sigue la siguiente fórmula:

```
Correccion = kp * error + kd * (error - errorAnterior)
```

Los valores de las constantes kp y kd se han ajustado experimentalmente. En función de la desviación obtenida se tienen diferentes controladores PD para controlar la velocidad de rotación, y se mantienen diferentes velocidades constantes en función de cada caso para la velocidad de tracción.

Otro caso que evaluaremos es si nos situamos en recta o curva. Si estamos en recta se aplicarán unas situaciones de control PD y si estamos en curva otras. Para diferenciar si nos encontramos en recta o en curva se empleará la diferencia entre la posición x de y3 (350) y de y1 (260). Lo que se hace exactamente es calcular la recta que pasa por el centro de y1 e y3, y después se mira la posición de x que se encuentra en esta recta para la fila y2 (310). De este modo conociendo el centro de la fila y2 y el punto x que se encuentra en la recta calculada, podemos saber si estamos en curva o recta. Esto se explica mejor a continuación. Para saber si es curva o recta se siguen los siguientes pasos:

1. Conociendo la ecuación de una recta ($y = m(x - x_1) + y_1$) podemos calcular la pendiente de la recta del siguiente modo:

$$y = m(x - x_1) + y_1$$

$$260 - 350 = m(x_{arriba} - x_{abajo})$$

$$m = -90/(x_{arriba} - x_{abajo})$$

Donde x_{arriba} es el centro de la fila y1 (arriba) y x_{abajo} es el centro de la fila y3 (abajo).

2. Una vez conocida la ecuación de la recta que pasa por estos dos puntos, se calcula el punto x de la fila de en medio (y2) que pasaría por esta recta. Este valor de x lo calculamos del siguiente modo:

$$y2 - y3 = m(x - x_{abajo})$$

$$(310 - 350)(x_{arriba} - x_{abajo})/(-90) = (x - x_{abajo})$$

$$x = (310 - 350)(x_{arriba} - x_{abajo})/(-90) + x_{abajo}$$

Tabla 4.1: Resultados del Pilotaje manual

Circuitos	Tiempo simulado
pistaSimple (horario)	1min 35 seg
pistaSimple (anti-horario)	1min 33 seg
monacoLine (horario)	1min 15 seg
monacoLine (anti-horario)	1min 15 seg
nurburgrinLine (horario)	1min 02 seg
nurburgrinLine (anti-horario)	1min 02 seg
curveGP (horario)	2min 13 seg
curveGP (anti-horario)	2min 09 seg
pista_simple (horario)	1min 00 seg
pista_simple (anti-horario)	59 seg

3. Una vez tenemos el punto x que pasa por la recta calculada a partir de y1 e y3, podemos calcular la desviación de x respecto al centro de y2. Si esta desviación es mayor a un umbral es que estamos en curva y si no es que estamos en recta. Como hemos mencionado antes, aplicamos unas situaciones de PD para recta y otras situaciones para curva.

A la hora de probar el piloto manual se han hecho pruebas para ver su robustez. Una de estas pruebas consiste en modificar la posición del Fórmula 1 en la línea de salida para que no esté situado el coche justamente encima de la línea roja. En este caso el coche es capaz de volver encima de la línea roja. Otra prueba ha sido mover el vehículo mediante el teleoperador durante la ejecución del piloto manual. El coche es capaz de volver otra vez encima de la línea roja a pesar de intentar girar el coche con el teleoperador.

El piloto manual se ha ejecutado en los diferentes circuitos, tanto en sentido horario como en sentido anti-horario. Se ha medido los tiempos de simulación que tarda el vehículo en dar una vuelta al circuito en ambos sentidos. En la tabla 4.1 se muestran los resultados de este piloto manual.

4.6. Creación del conjunto de datos

En este proyecto el objetivo es que un vehículo sea capaz de conducir bajo diferentes circunstancias, es decir, en diferentes entornos y diferentes iluminaciones. Además, el objetivo es emplear diferentes redes neuronales con este fin. Para que las redes neuronales sean capaces de aprender es necesario crear un conjunto de datos.

Como se menciona en el libro “Deep Learning, Introducción práctica con Keras” [46], el conjunto de datos se debe dividir para poder configurar y evaluar el modelo de forma correcta. En *Deep Learning* estos datos se dividen en tres conjuntos: datos de entrenamiento (*training*), datos de validación (*validation*) y datos de prueba (*test*).

Los datos de entrenamiento son aquellos que se utilizan para que la red obtenga los parámetros del modelo. Cuando entrenamos un modelo con un conjunto de entrada lo que ocurre es que hacemos que el modelo sea capaz de aprender de forma general un concepto. De esta forma cuando le consultamos por nuevos datos el modelo será capaz de comprender estos nuevos datos y devolver un resultado fiable en función de su capacidad de generalización. Sin embargo, si este modelo no es capaz de adaptarse a los datos de entrada (por ejemplo se produce *underfitting* u *overfitting*), en este caso modificaremos los hiperparámetros del modelo, y después de entrenar el modelo de nuevo con los datos de entrenamiento evaluaremos el modelo con los datos de validación.

Los hiperparámetros se pueden ir ajustando guiados por los datos de validación hasta obtener unos resultados de validación que consideremos apropiados. Si hemos seguido este método, lo que ha sucedido es que el modelo se ha ajustado también a los datos de validación. Por este motivo, es necesario reservar un conjunto de *test* que solamente emplearemos al evaluar el modelo cuando consideremos que ya hemos terminado de ajustar los hiperparámetros.

Como hemos visto para realizar el entrenamiento y la evaluación de las redes neuronales es necesario disponer de bases de datos donde el contenido se corresponda con el estímulo del problema. Por este motivo es necesario crear una conjunto de datos para conducción autónoma en el simulador Gazebo. Esta base de datos se ha creado a partir del piloto manual que se ha explicado en la Sección 4.5. Como se mencionó en esta Sec-

ción, este piloto es capaz de conducir de forma autónoma mediante una solución basada en visión que calculaba las órdenes de velocidad que hay que enviar al vehículo.

El conjunto de datos creado es un *dataset* “casero” donde se han almacenado las imágenes de la cámara del piloto manual en cada instante, así como los datos de velocidad necesarios. Los datos de velocidades se han guardado en un archivo *.json*. En este fichero se han guardado diferentes datos relacionados con la velocidad. Por un lado, se han almacenado en un diccionario los datos numéricos de v y w con las claves *v* y *w*. Por otro lado se han almacenado datos que servirán para entrenar las redes de clasificación. Para almacenar estos datos se han creado diferentes claves con sus valores en función de diferentes clasificaciones. Se han creado diferentes clasificaciones para ver el efecto que tienen en las redes neuronales. A continuación, se puede observar cómo se almacenan los datos de velocidad correspondientes a la primera imagen del *dataset*:

```
{"class_v_5": very_fast, "class_w_9": slight,
"class3": "very_fast", "class2": "slight",
"classification": "left", "w": 0.029500000000000002, "v": 13}
```

En este ejemplo, se puede observar que hay diferentes claves con su valor como habíamos mencionado. Las claves correspondientes a las diferentes clasificaciones son:

- *classification*: Esta clasificación divide los datos entre las clases “left” y “right” en función de los datos de velocidad de giro. Si la velocidad de giro es negativa el dato se corresponde con la clase “right”, y si por el contrario es positiva se corresponde con la clase “left”.
- *class2*: Esta clasificación divide los datos de la velocidad lineal en 4 clases. Estas clases son: “slow” si la velocidad es menor o igual que 7; “moderate” si la velocidad es mayor que 7 y menor o igual que 9; “fast” si la velocidad es mayor que 9 y menor o igual que 11; y “*very_fast*” si la velocidad es mayor que 11.
- *class3*: En esta clasificación se dividen los datos de velocidad angular en 7 clases. Las clases son: “radically_left” si la velocidad de rotación (*w*) es mayor o igual a 1; “moderately_left” si *w* es menor que 1 y mayor o igual que 0.5; “slightly_left” si *w* es

menor que 0.5 y mayor o igual que 0.1; “slight” si w es menor que 0.1 y mayor a -0.1; “slightly_right” si w es menor o igual que -0.1 y mayor que -0.5; “moderately_right” si w es menor o igual que -0.5 y mayor que -1; y “radically_right” si la velocidad de rotación es menor que -1.

- *class_v_5*: Se dividen los datos de la velocidad lineal (v) en 5 clases. Estas clases son: “negative” si v es menor que 0, “slow” si la velocidad es menor o igual que 7 y mayor que 0; “moderate” si la v es mayor que 7 y menor o igual que 9; “fast” si la v es mayor que 9 y menor o igual que 11; y “very_fast” si la velocidad es mayor que 11.
- *class_w_9*: Se dividen los datos de velocidad angular en 9 clases. Las clases son: “radically_left” si la velocidad de rotación (w) es mayor o igual a 2; “strongly_left” si w es menor que 2 y mayor o igual que 1; “moderately_left” si w es menor que 1 y mayor o igual que 0.5; “slightly_left” si w es menor que 0.5 y mayor o igual que 0.1; “slight” si w es menor que 0.1 y mayor a -0.1; “slightly_right” si w es menor o igual que -0.1 y mayor que -0.5; “moderately_right” si w es menor o igual que -0.5 y mayor que -1; “strongly_right” si w es menor o igual que -1 y mayor a -2; y “radically_right” si la velocidad de rotación es menor que -2.

Cuando se comenzó el proyecto se creó un conjunto de datos que constaba de 5006 pares de datos, de los cuales 2803 pares eran datos de entrenamiento, 701 eran datos de validación, y 1502 eran datos de *test*. Este conjunto fue grabado gracias al piloto manual que conducía de forma autónoma únicamente por el circuito *pistaSimple* en un mismo sentido. Este conjunto tenía un problema y es que únicamente el vehículo poseía datos de un circuito, por lo que no era capaz de generalizar y aprender a conducir en otros entornos.

Con el objetivo de analizar los datos de los que disponíamos y solventar este problema se ha creado una gráfica de estadísticas de los datos. En cada imagen se han analizado dos fila y se ha calculado el centroide de la línea roja del circuito para cada una de las filas (filas 350 y 260). En el eje x de la gráfica, se representa el centroide de la fila 350 (L2), y el eje y representa el centroide de la fila 260 (L1) de la imagen. En la Figura 4.9 podemos ver la representación de esta estadística del conjunto de entrenamiento (círculos rojos) contra los datos de una conducción fallida (cruces azules).

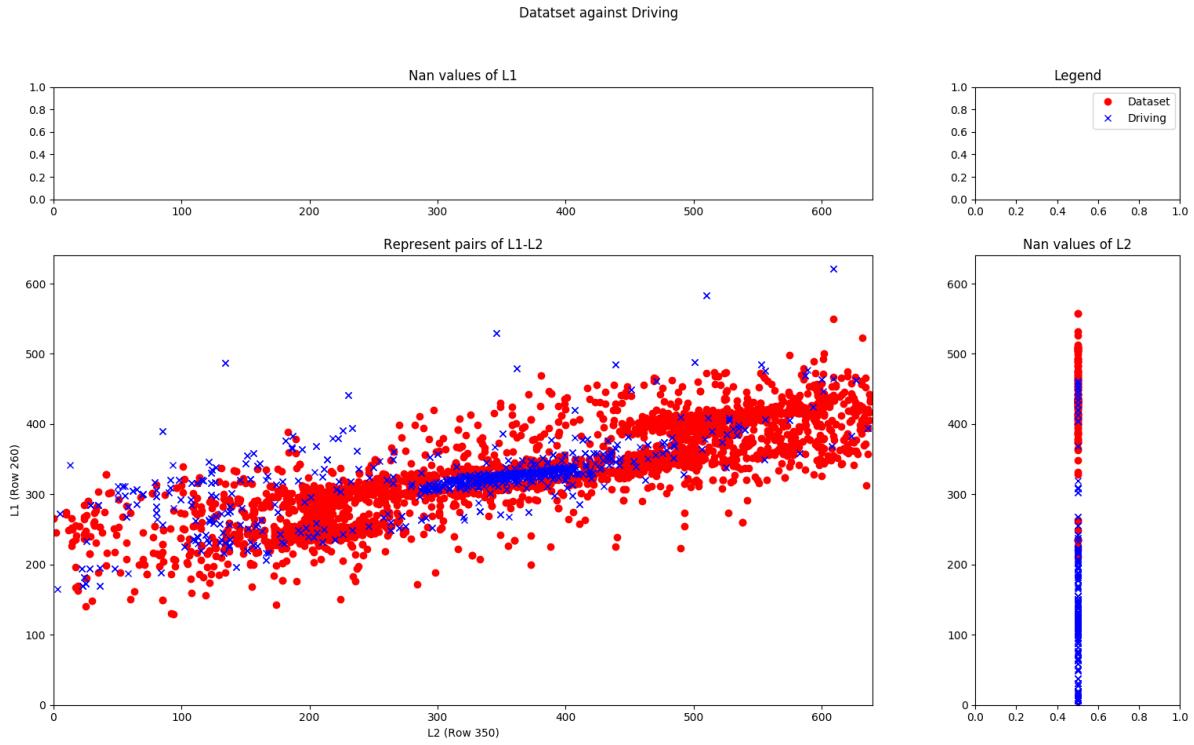


Figura 4.9: Representación pares L1-L2 (*Dataset1* contra conducción)

En la Imagen 4.9 se puede observar que hay bastantes casos conocidos para el coche, como pueden ser probablemente situaciones donde se encuentra en recta el coche; pero sin embargo hay zonas de la gráfica donde se representan cruces azules (conducción fallida) y no hay ningún círculo rojo (conjunto de entrenamiento) alrededor. Esto quiere decir que el coche está ante situaciones desconocidas, y por tanto, no sabe qué hacer.

La gráfica 4.9 nos hace pensar que es necesario entrenar el modelo con un conjunto de imágenes bastante más representativo. Por este motivo se ha creado un nuevo conjunto de datos que trata de solventar este inconveniente.

El nuevo conjunto de datos (denominado *Dataset*) se ha grabado en tres circuitos diferentes: *pistaSimple*, *monacoLine*, *nurburgrinLine*. El piloto ha dado varias vueltas a los 3 circuitos en ambos sentidos para poder grabar este conjunto. Este *dataset* consta de 17341 pares de imágenes-datos. Se han dividido los datos obteniendo 9710 pares de datos de entrenamiento, 2428 pares de validación, y 5203 pares para *test*.

La misma gráfica estadística que se ha empleado para evaluar el dataset “fallido” se ha utilizado con el fin de evaluar los nuevos datos. En la Figura 4.10 se puede ver cómo el nuevo conjunto de datos es muy representativo, ya que consigue abarcar la gran mayoría de los casos que eran desconocidos para el coche.

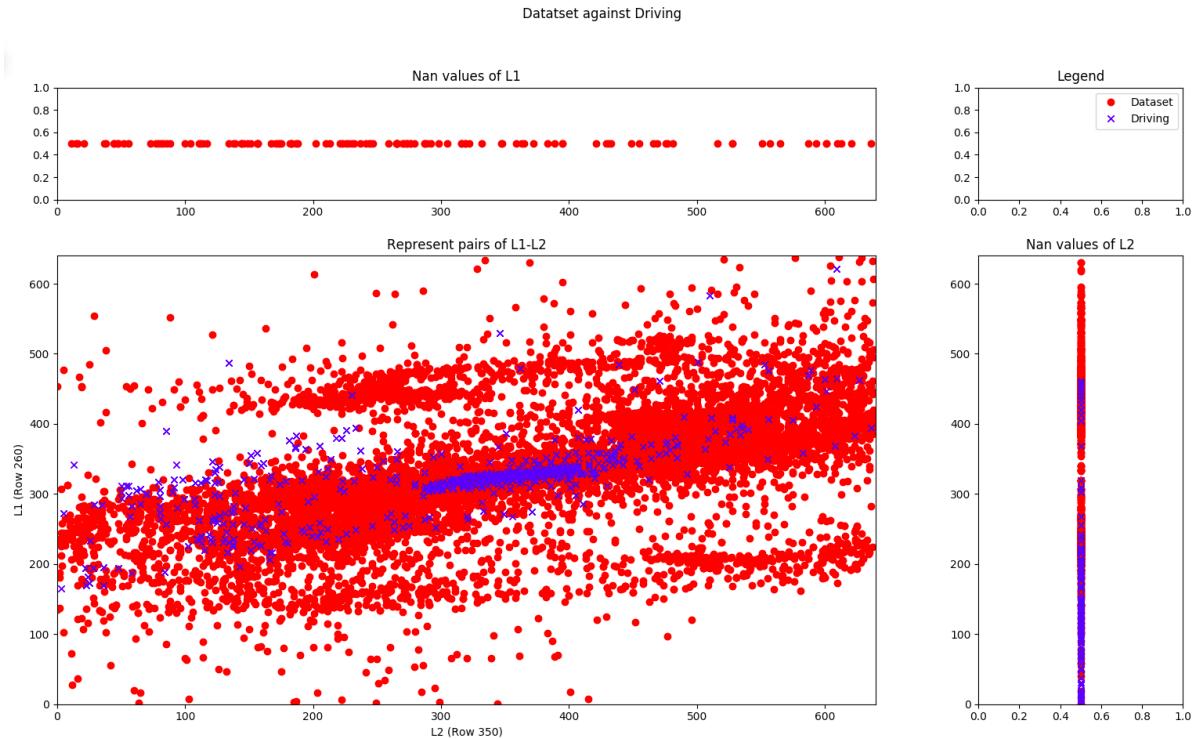


Figura 4.10: Representación pares L1-L2 (nuevo *Dataset* contra conducción)

En los circuitos utilizados para grabar el nuevo conjunto de datos existen más datos de rectas que de curvas, por este motivo se ha grabado un *dataset* complementario al anterior de un circuito que solamente posee curvas (*curveGP*). Este conjunto consta de 5268 pares de imágenes datos.

Este conjunto únicamente se ha añadido al entrenamiento de las redes neuronales recurrentes, ya que en las redes neuronales convolucionales ha sido suficiente con entrenar con el *dataset* anterior. En la Figura 4.11 se puede ver la gráfica estadística de este conjunto de datos (*Dataset_Curves*).

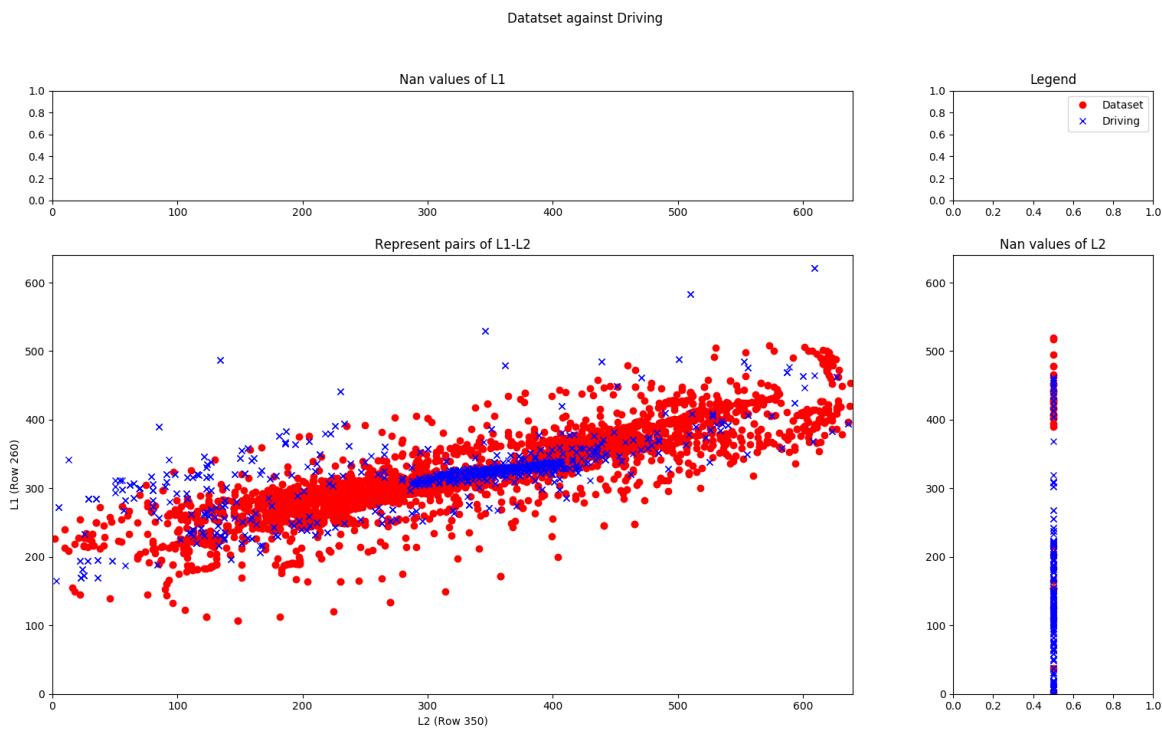


Figura 4.11: Representación pares L1-L2 (*Dataset_Curves* contra conducción)

La gráfica estadística también se puede emplear para evaluar cómo se distribuyen los datos de las clases, tanto para la velocidad lineal (v) como para la velocidad de rotación (w). En la Figura 4.12 se representa el análisis de los pares L1-L2 (centroides de las filas 260 y 350) para 7 clases de w (“radically_left”, “moderately_left”, “slightly_left”, “slight”, “slightly_right”, “moderately_right”, “radically_right”). En la Figura 4.13 se representa el análisis de los pares L1-L2 para 5 clases de v (“negative”, “slow”, “moderate”, “fast”, “very_fast”).

En estas imágenes (Figuras 4.12 y 4.13) se observa cómo los ejemplos quedan más o menos agrupados por clases entorno a un rango de valores de L1 y L2.

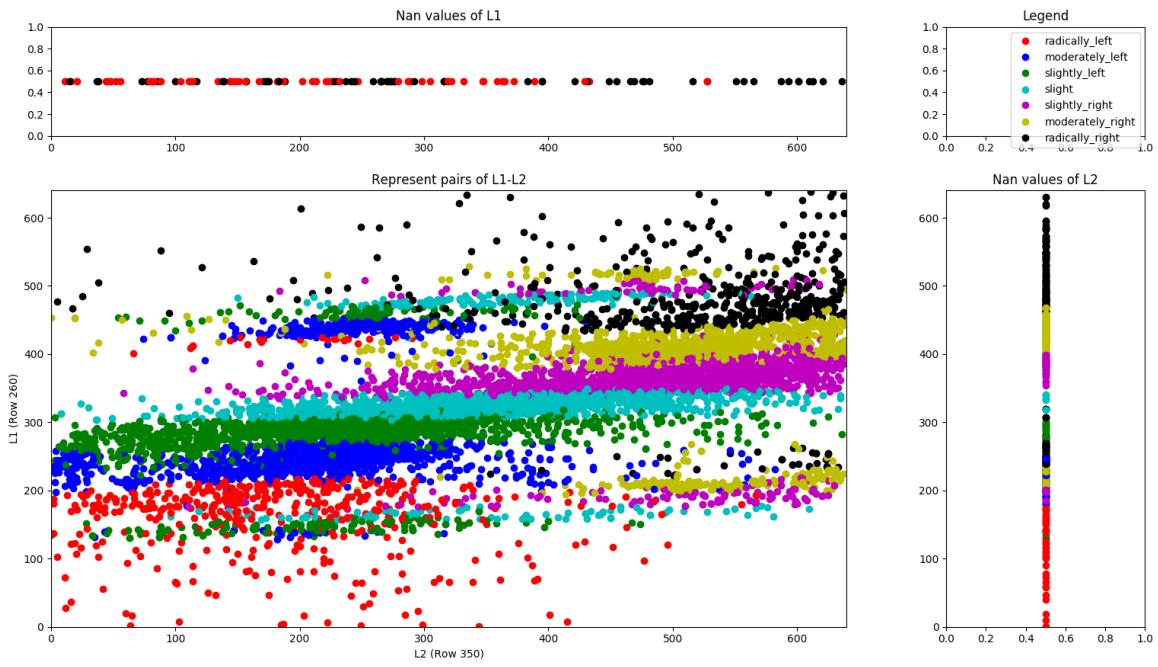


Figura 4.12: Análisis de pares L1-L2 (*Dataset*) para w

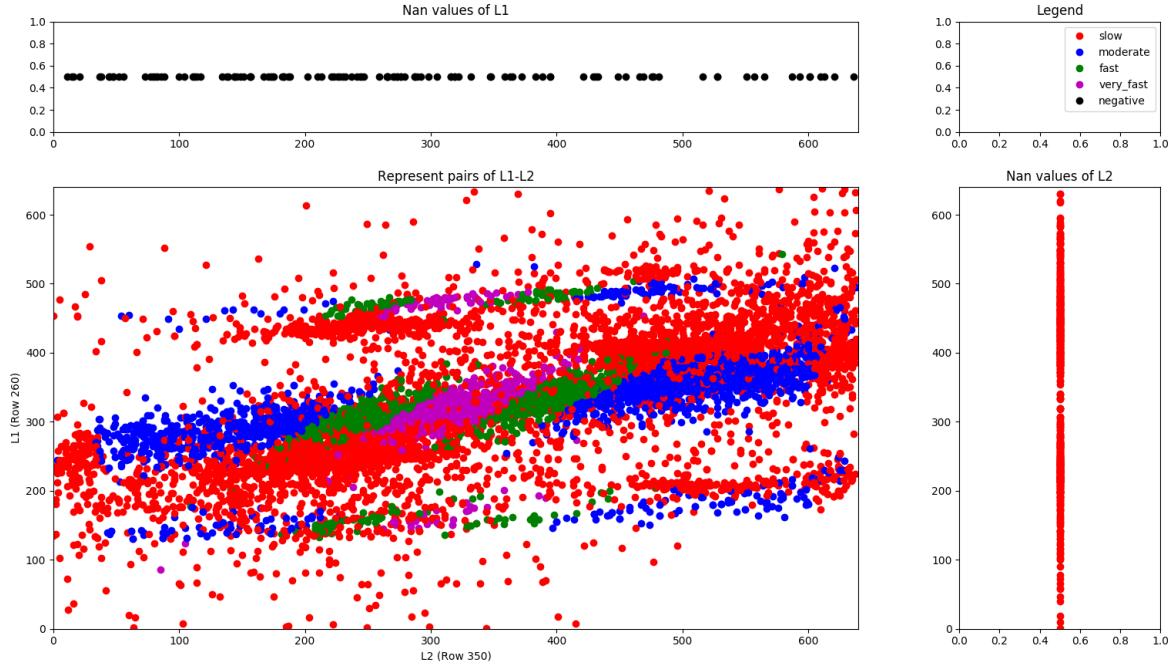


Figura 4.13: Análisis de pares L1-L2 (*Dataset*) para v

En el entrenamiento de las redes neuronales convolucionales se ha empleado el conjun-

to de datos *Dataset*; mientras que en el entrenamiento de las redes neuronales recurrentes se ha utilizado dicho conjunto, y adicionalmente el conjunto *Dataset_Curves*.

En ejecución, el piloto entrenado con las redes se ha probado en todos estos circuitos mencionados (*pistaSimple*, *monacoLine*, *nurburgrinLine*, *curveGP*) en la creación de los *datasets*, y además se ha probado en un circuito que las redes no han empleado para entrenamiento. Este circuito se denomina *pista_simple* y se ha utilizado en ambos sentidos.

4.7. Nodo piloto

El nodo piloto (componente académico), creado para cargar y emplear redes neuronales de conducción, resuelve varias funcionalidades: (a) ofrece una interfaz gráfica al usuario que le ayuda a depurar el código; (b) ofrece acceso a sensores y actuadores en forma de métodos simples (oculta el *middleware* de comunicaciones); (c) incluye código auxiliar para poder emplear las predicciones realizadas por las redes (bien de clasificación o de regresión). Lo deja todo atado para que el usuario sólo tenga que incluir su red y retoque el fichero *MyAlgorithm.py*, donde se proporciona al vehículo las órdenes de velocidad predichas por el nodo. En la Figura 4.14 se puede observar la estructura que tiene este nodo piloto.

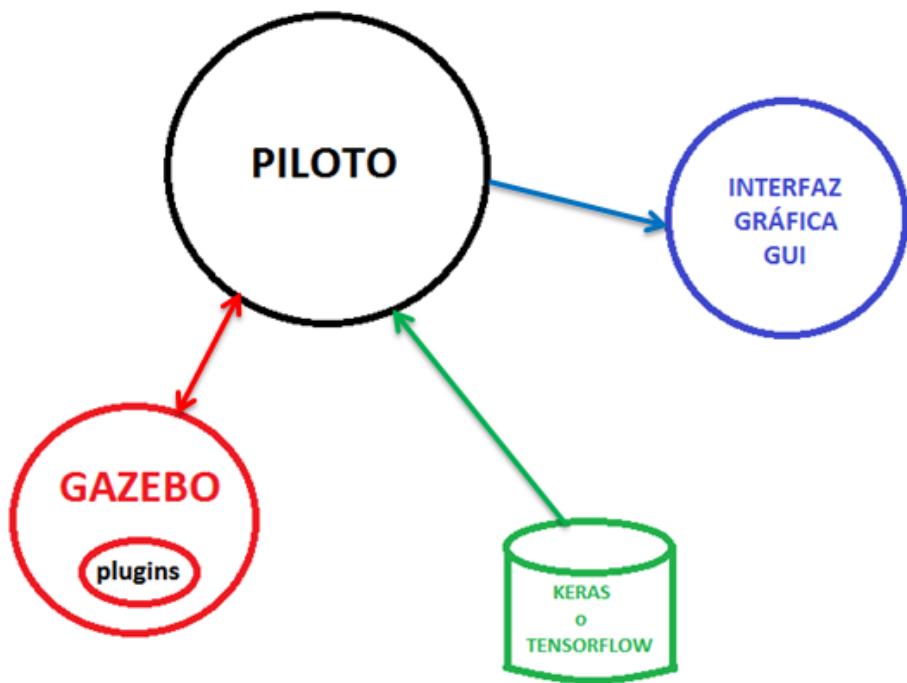


Figura 4.14: Estructura del Nodo Piloto

Este nodo ofrece al programador un Application Programming Interface (API) de sensores y actuadores, y de predicciones de la red. A continuación se puede observar el API concreto de este proyecto:

- `camera.getImage`: Permite obtener la imagen de la cámara del coche.
- `motors.sendV`: Para establecer la velocidad lineal.
- `motors.sendW`: Para establecer la velocidad de giro.
- `network.prediction_v`: Para obtener la predicción de velocidad lineal de la red.
- `network.prediction_w`: Para obtener la predicción de velocidad de giro de la red.

Es necesario emplear la biblioteca `comm` incluida en JdeRobot para realizar la comunicación entre distintos componentes. Podemos usar esta librería empleando ROS, o a través de un proxy de ICE (protocolo de comunicaciones). Para poder crear un comunicador con `comm` en JdeRobot, es necesario emplear el archivo de configuración YML².

²YAML Ain't Markup Language format

CAPÍTULO 4. INFRAESTRUCTURA DESARROLLADA

En el archivo YML se indican los puertos de los *plugins* que emplea el coche de carreras. Además, tenemos que indicar el *framework* de redes neuronales que emplearemos (Keras o Tensorflow), así como los pesos de modelos de redes neuronales que ya hemos entrenado y que cargaremos para poder predecir datos de velocidad. Este fichero (driver.yml) en el proyecto tiene el siguiente aspecto:

```
Driver:  
  
    CameraLeft:  
        Server: 2 # 0 -> Deactivate, 1 -> Ice , 2 -> ROS  
        Proxy: "cam_f1_left:tcp -h localhost -p 8995"  
        Format: RGB8  
        Topic: "/F1ROS/CameraL/image_raw"  
        Name: FollowLineF1CameraLeft  
  
    Motors:  
        Server: 0 # 0 -> Deactivate, 1 -> Ice , 2 -> ROS  
        Proxy: "Motors:tcp -h localhost -p 9999"  
        Topic: "/F1ROS/Motors"  
        Name: FollowLineF1Motors  
  
    robot: F1  
  
Network:  
    Framework: Keras # Currently supported: "Keras" or "TensorFlow"  
    Model_Classification_w: models/model_smaller_vgg_7classes_biased_cropped_w.h5  
    Model_Classification_v: models/model_smaller_vgg_5classes_biased_cropped_v.h5  
    Model_Regression_v: models/model_controlnet_v.h5  
    Model_Regression_w: models/model_controlnet_w.h5  
    Dataset: Net/Dataset  
  
NodeName: Driver
```

Podemos ver que los motores emplean el Puerto 9999, mientras que la cámara emplea

el Puerto 8995. Además, podemos ver que en este archivo se indica que se utilizará el *framework* Keras, así como los modelos (para v y w) que cargaremos en el caso de emplear una red de clasificación o una red de regresión.

Se han creado dos clases que permiten cargar los modelos de v y w indicados en el archivo de configuración *.yml*, así como predecir los valores de v y w, y almacenarlos en las variables *self.network.prediction_v* y *self.network.prediction_w*. La clase de Python creada para las redes neuronales de clasificación es *ClassificationNetwork*, y para redes de regresión es *RegressionNetwork*. En el archivo principal (driver.py) se indicará cuál de estas dos clases queremos emplear al ejecutar el nodo Piloto. De esta forma desde el fichero *MyAlgorithm.py* podremos emplear las velocidades predichas por la red e indicar las órdenes de velocidad al vehículo.

Se ha dividido el nodo piloto (componente académico) en diferentes partes, por lo que emplearemos hilos de ejecución para llevar a cabo diferentes tareas de forma simultánea. En este proyecto existen tres procesos diferenciados:

- Hilo de percepción y control: Es el encargado de actualizar los datos de los sensores y los actuadores. El tiempo de refresco de este hilo es muy importante, y debe ser un periodo de tiempo muy corto, ya que se encarga de establecer la velocidad y la dirección del vehículo en todo momento. Si este intervalo de tiempo fuera muy grande, las decisiones que modifican la trayectoria del coche podrían ser incorrectas. Este hilo (*ThreadPublisher*) se utiliza para actualizar los datos de la cámara y enviar órdenes a los motores. Se actualiza cada 80 milisegundos.
- Hilo de la interfaz gráfica de usuario (GUI): Se encarga de actualizar la GUI. El intervalo de actualización de este hilo es muy importante, ya que tenemos que mostrar la imagen que ve el coche en tiempo real. Por lo que el intervalo de tiempo debe ser pequeño. El hilo de ejecución de la GUI (*ThreadGUI*) se actualizará cada 50 ms.
- Hilo de la red: Es el encargado de inferir valores a partir la última imagen recibida, de forma asíncrona. Cuando termina la inferencia, se almacena el valor dentro del elemento de red. Cuando el vehículo necesita los últimos datos de inferencia, solamente toma estos datos sin bloquear ningún proceso ni llamda. Este hilo

(*ThreadNetwork*) se debe actualizar en intervalos de tiempo pequeños para que el coche sea capaz de conducir. Este hilo se actualiza cada 50 ms.

4.7.1. Interfaz gráfica

La intergaz gráfica (GUI) del proyecto es una ayuda para proporcionar datos durante el pilotaje del vehículo. Esta interfaz se ha creado empleando PyQt5, dado que permite realizar interfaces con numerosos objetos gráficos (imágenes, botones, etc).

La GUI del proyecto (Figura 4.15) contiene la imagen que captura la cámara del vehículo. Esta imagen está situada en la parte superior izquierda de la GUI. Gracias a ella, el usuario puede tener una idea de la visión del coche y emplear estas imágenes como datos de entrada a las redes entrenadas.

En la parte derecha de la imagen mostrada en la GUI hay 4 leds que se corresponden con diferentes rangos de velocidades lineales; mientras que en la parte inferior hay 7 leds que se corresponden con velocidades angulares del coche. En función del valor predicho por la red se encenderá un led verde de la parte inferior y un led azul de la parte derecha. De esta forma, es más fácil obtener una visualización de las órdenes de velocidad aproximadas que está recibiendo el coche.

Si denotamos los leds (velocidad angular) que aparecen debajo de la imagen con los números de 1 a 7 empezando por la izquierda, se encenderá cada uno de los leds en el siguiente caso:

- Led 1: La velocidad angular del vehículo es mayor o igual a 1, es decir, el coche gira bruscamente a la izquierda.
- Led 2: La velocidad angular del vehículo está en el rango [0.5, 1), es decir, el coche gira moderadamente a la izquierda.
- Led 3: La velocidad angular del coche se encuentra en el rango [0.1, 0.5), es decir, el coche gira ligeramente a la izquierda.
- Led 4: La velocidad angular del coche está en el rango (-0.1, 0.1), el coche está en recta.

- Led 5: La velocidad angular del vehículo se encuentra en el rango $(-0.5, -0.1]$, el coche gira ligeramente a la derecha.
- Led 6: La velocidad angular del vehículo está en el rango $(-1, -0.5]$, el coche gira moderadamente a la derecha.
- Led 7: La velocidad angular del coche es menor o igual que -1 , el coche gira bruscamente a la derecha.

Si denotamos los leds (velocidad lineal) que a la derecha de la imagen con los números de 1 a 4 empezando por abajo, se encenderá cada uno de estos leds en el siguiente caso:

- Led 1: La velocidad es menor o igual que 7. Esta situación se corresponde con los casos donde el coche se encuentra en una curva y necesita reducir la velocidad o incluso dar marcha atrás en algún caso.
- Led 2: La velocidad se encuentra en el rango $(7, 9]$, es decir, o bien estamos en recta y nos encontramos algo desviados o estamos en una curva muy leve.
- Led 3: La velocidad se encuentra en el rango $(9, 11]$, es decir, estamos en recta y nos encontramos un “pelín” desviados del centro de la línea roja.
- Led 4: La velocidad es mayor que 11. En esta situación nos encontramos en una recta y el coche va a mucha velocidad.

Además, para añadir más información de velocidad que le permita al usuario depurar fallos, a la derecha de los leds que indican la velocidad lineal se han añadido las órdenes de velocidades que se envían a los motores del coche. Por un lado, tenemos el valor de la velocidad lineal, que aparece indicado con una v ; mientras que el valor de la velocidad angular se indica con una letra w .

Esta interfaz gráfica además muestra un teleoperador justo debajo del botón “Play Code”. Mediante este teleoperador se puede mover manualmente el coche en el mundo de Gazebo si se desea. En la esquina superior derecha, donde se muestra las órdenes de velocidad del coche, aparecerán los valores de velocidad que tiene el mismo cuando se teledirige. La velocidad lineal del coche se puede controlar moviendo el *joystick* en sentido vertical. Cuanto más subamos el *joystick* más velocidad tendrá el coche hacia delante,

y si lo bajamos del todo más velocidad lineal tendrá hacia atrás. La velocidad angular del vehículo se controla moviendo el *joystick* en sentido horizontal, según lo movamos a izquierda o a la derecha, el robot girará en un sentido u otro.

En la aplicación gráfica también hay cuatro botones importantes, dos que sirven para controlar lo que sucede con el algoritmo que controla al coche, y otros dos que se emplean en el manejo de un *dataset*. El botón inferior izquierdo, en el que aparece un símbolo de STOP, es el que emplearemos cuando teledirigimos el coche y queremos que pare en un punto y no siga navegando. El botón que se encuentra entre la imagen y el teleoperador, en el cual pone “Play Code”, es el botón con el que le ordenamos al componente que comience a ejecutar el código del fichero *MyAlgorithm.py*. Este botón cambia de color al pulsarlo. Si queremos que este código pare en un determinado momento, pulsaremos el mismo botón haciendo que pare; y si queremos reanudar su comportamiento lo volveremos a pulsar.

Los otros dos botones se emplean para el manejo de un *dataset*. Por un lado, tenemos el botón de la parte superior derecha, en el cual pone “Save Dataset”, que sirve para crear un nuevo *dataset* que guarde los datos tanto de velocidad como las imágenes que ve el coche durante la ejecución de su algoritmo. Este conjunto de datos se crea con la misma estructura que el conjunto de datos mencionado en la Sección 4.6, y se almacenará en el directorio “Dataset”. Este conjunto de datos únicamente se crea si pulsamos este botón, de no ser así simplemente se ejecuta el algoritmo que permite navegar al vehículo. Por otra parte, el botón que se sitúa en la parte inferior derecha, permite borrar el *dataset* creado.

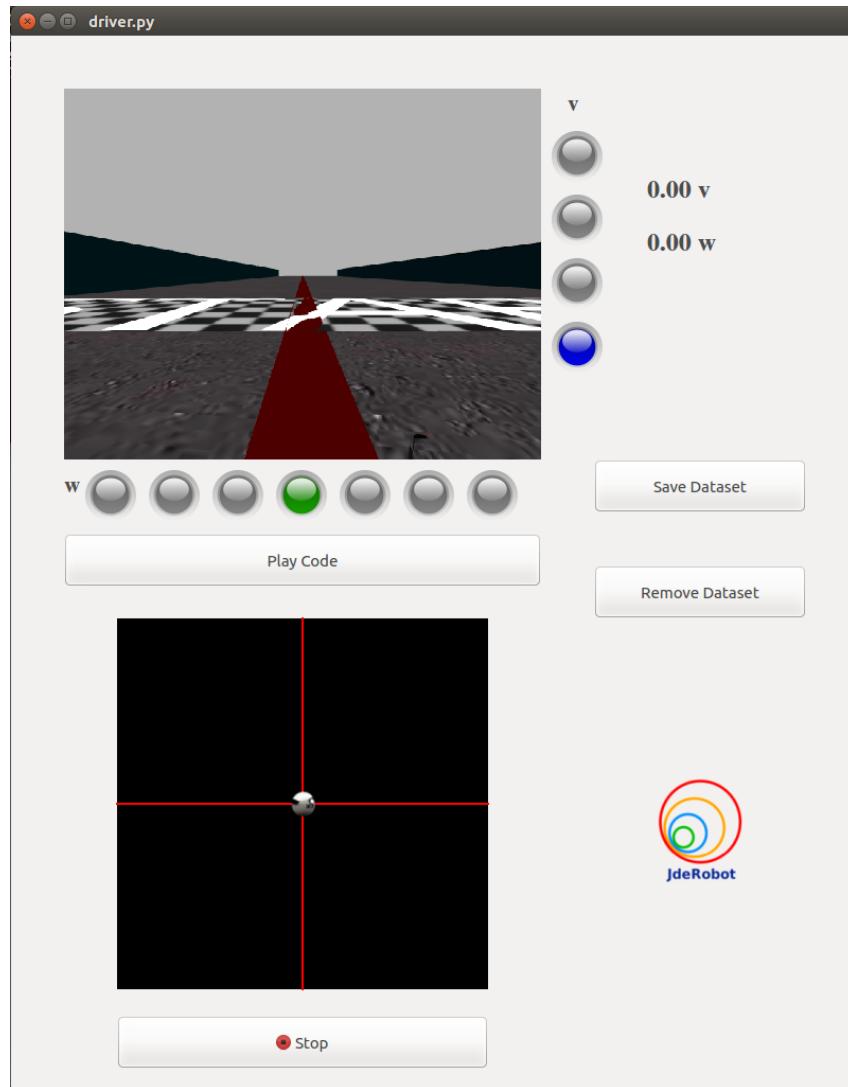


Figura 4.15: Interfaz gráfica (GUI)

4.7.2. Ejecución típica

Para ejecutar el nodo piloto es necesario lanzar en un terminal el fichero de configuración de ROS que permite lanzar el mundo de Gazebo y los *plugins* del coche, y que se ha descrito en 4.4. Para lanzar este fichero hay que ejecutar el siguiente comando si ejecutamos el mundo con el circuito *pistaSimple*:

```
roslaunch /opt/jderobot/share/jderobot/gazebo/launch/f1.launch
```

CAPÍTULO 4. INFRAESTRUCTURA DESARROLLADA

Si en vez de esta pista de carreras queremos ejecutar otra, será necesario lanzar el fichero de configuración correspondiente. Una vez hemos lanzado este fichero se abrirá Gazebo con el escenario que hemos elegido incluyendo circuito y coche (Figura 4.16).

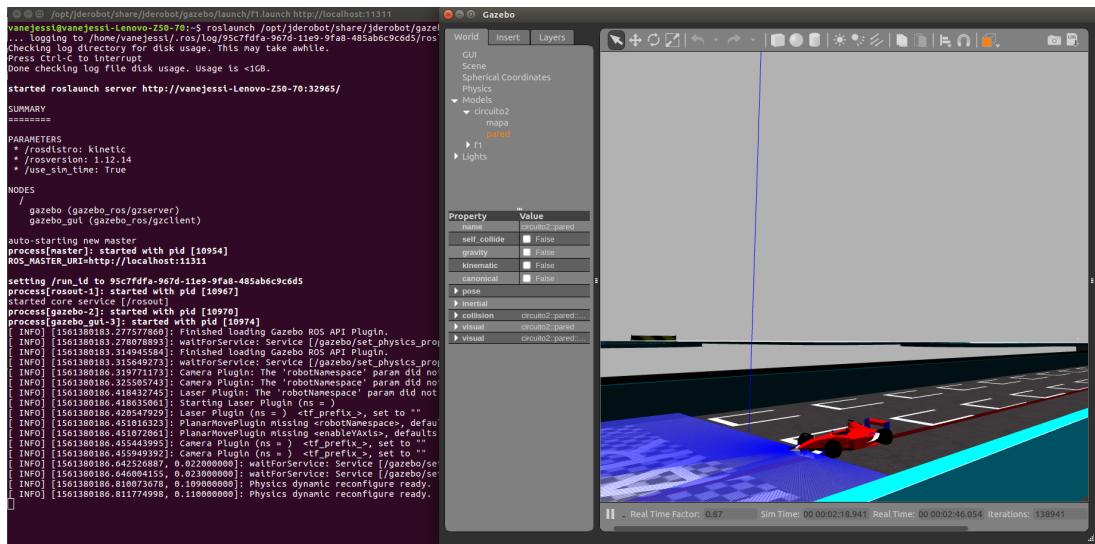


Figura 4.16: Inicialización de ROS y Gazebo

Para poder lanzar el nodo Piloto es necesario abrir otra terminal y ejecutar el siguiente comando:

```
cd 2017-tfm-vanessa-fernandez/Follow\ Line/dl-driver/
python2 driver.py driver.yml
```

Una vez hemos ejecutado este comando, el nodo Piloto enlaza los sensores y actuadores de ROS-Kinetic con las variables asociadas a los mismos: *self.camera* y *self.motors*. Mediante estas variables, el nodo Piloto se comunica con los *drivers* de ROS-Kinetic.

Además, cuando ejecutamos este comando, se lanzará también la Graphical User Interface (GUI) en la que se pueden observar las imágenes obtenidas por la cámara del coche, así como los botones de control, los datos de velocidad, y el teleoperador (Figura 4.17).

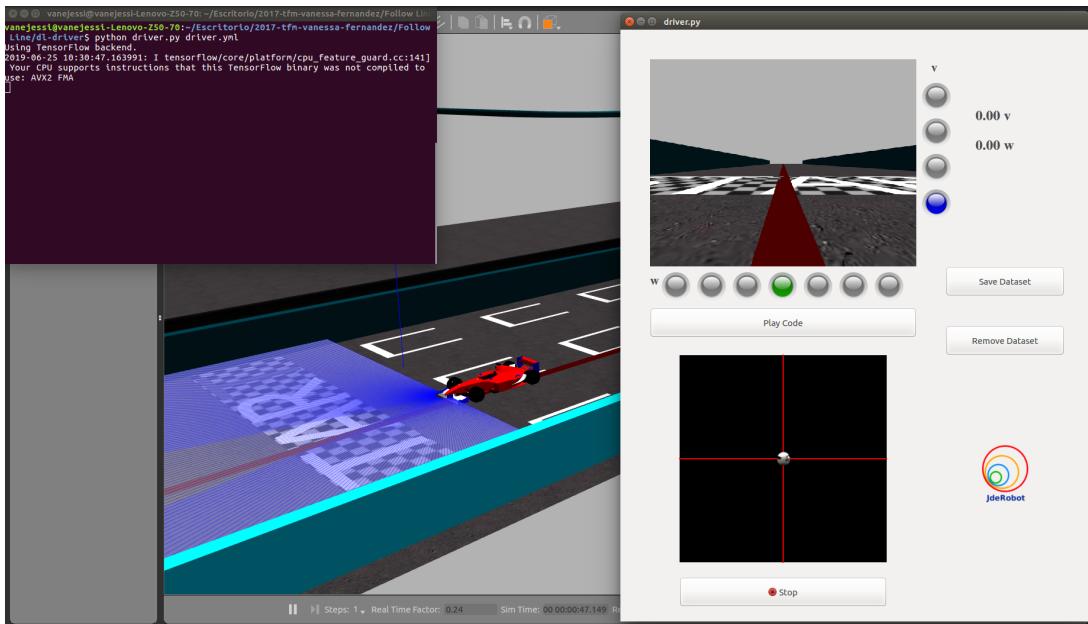


Figura 4.17: Inicialización del nodo Piloto y la GUI

Una vez se han inicializado el escenario en Gazebo, el nodo Piloto y la GUI, se puede ejecutar el código que emplea las predicciones de las redes para conducir el coche. Este código se ejecutará pulsando el botón “Play Code”. De esta forma somos capaces de visualizar en el simulador y en la GUI el resultado de la conducción mediante redes neuronales.

4.7.3. Tiempo de ejecución

En el proyecto es muy importante el tiempo de ejecución, ya que este tiempo influye en las decisiones que toma el vehículo, y cuanto más rápido sea el algoritmo mejor. En el tiempo que tarda en tomar decisiones el coche influirá el ordenador que empleemos. Este es un inconveniente, ya que quien posea mejor ordenador obtendrá tiempos de ejecución menores que quien tenga un ordenador sin tantas capacidades. La ejecución de Gazebo consume muchos recursos del ordenador haciendo que el coche sea más lento.

En la parte inferior de Gazebo se puede ver el *Real Time*, el *Sim. Time* (tiempo simulado) y el *Real Time Factor*, los cuales tienen mucho que ver en el tiempo de ejecución de Gazebo. El parámetro *Real Time* expresa el tiempo real en ejecución. El factor *Sim. Time* expresa el tiempo simulado. Si empleáramos un ordenador con grandes capacidades

entonces el *Sim. Time* debería estar próximo al *Real Time*. Mientras que si utilizamos un ordenador con menos capacidades veremos que el *Sim. Time* es mucho menor que el *Real Time*. Por su parte, el factor *Real Time Factor* es un producto de la tasa de actualización y el tamaño del paso. Si queremos obtener un tiempo de simulación bajo, este factor tendrá que tener un valor entorno a 1. Si este parámetro es menor que 1 veremos que la ejecución es más lenta, y cuando se aproxima a 0.2 o menos es demasiado lenta.

En el caso del ordenador que se ha empleado el *Real Time Factor* es muy bajo en algunas ocasiones durante el pilotaje, lo que hace que el *Real Time* sea mucho mayor que el *Sim. Time*. En este ordenador el *Real Time Factor* normalmente oscila entre 0.2 y 0.6, siendo en grandes ocasiones cercano a 0.2.

Capítulo 5

Redes de clasificación

Capítulo 6

Redes de regresión

Capítulo 7

Conclusiones

Bibliografía

- [1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [2] Anyu. "RNA – Redes Neuronales Artificiales", *Bitácoras de un Ingeniero*. <http://andrealezcano.blogspot.com.es/2011/04/rna-redes-neuronales-artificiales.html>, 2011. [Accedido 29 de Mayo de 2019].
- [3] MIT Computer Science & Artificial Intelligence Lab. Using AI to predict breast cancer and personalize care. <https://www.csail.mit.edu/news/using-ai-predict-breast-cancer-and-personalize-care>, 2019. [Accedido 30 de Mayo de 2019].
- [4] Janosch Delcker. The man who invented the self-driving car (in 1986). <https://www.politico.eu/article/delf-driving-car-born-1986-ernst-dickmanns-mercedes/>, 2018. [Accedido 30 de Mayo de 2019].
- [5] Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 305–313. Morgan-Kaufmann, 1989.
- [6] Automated driving levels of driving automation are defined in new SAE International Standard J3016. https://www.smmt.co.uk/wp-content/uploads/sites/2/automated_driving.pdf, 2014. [Accedido 3 de Junio de 2019].
- [7] S.Kom M.Eng. Dewi Suryani. Convolutional Neural Network, *Binus University - School of Computer Science*. <http://soc.s.binus.ac.id/2017/02/27/convolutional-neural-network/>, 2017. [Accedido 31 de Mayo de 2019].
- [8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

- [9] Galo Fariño R. Modelo Espiral de un proyecto de desarrollo de software, *Administración y Evaluación de Proyectos*. <http://www.ojovisual.net/galofarino/modeloespiral.pdf>, 2011. [Accedido 31 de Mayo de 2019].
- [10] Modelo Espiral. <http://modeloespiral.blogspot.com.es/>, 2009. [Accedido 31 de Mayo de 2019].
- [11] Herramientas software. https://moodle2.unid.edu.mx/dts_cursos_mdl/licIEL/HS/S04/HS04_Lectura.pdf. [Accedido 31 de Mayo de 2019].
- [12] Eder Santana and George Hotz. Learning a driving simulator. *CoRR*, abs/1608.01230, 2016.
- [13] Self Driving Car Engineer. <https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013>, 2017. [Accedido 30 de Abril de 2019].
- [14] Mariusz Bojarski, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence D. Jackel, and Urs Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. *CoRR*, abs/1704.07911, 2017.
- [15] Keith Sullivan and Wallace Lawson. Reactive ground vehicle control via deep networks. 2017.
- [16] Udacity's Datasets. <https://github.com/udacity/self-driving-car/tree/master/datasets>, 2016. [Accedido 30 de Abril de 2019].
- [17] Zhengyuan Yang, Yixuan Zhang, Jerry Yu, Junjie Cai, and Jiebo Luo. End-to-end multi-modal multi-task vehicle control for self-driving cars with visual perceptions. *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 2289–2294, 2018.
- [18] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio López, and Vladlen Koltun. Carla: An open urban driving simulator. In *CoRL*, 2017.
- [19] CARLA: Open-source simulator for autonomous driving research. <http://carla.org/>, 2017. [Accedido 30 de Abril de 2019].
- [20] Gazebo. <http://gazebosim.org/>, 2011. [Accedido 30 de Abril de 2019].

- [21] Udacity’s Self-Driving Car Simulator. <https://github.com/udacity/self-driving-car-sim/>, 2017. [Accedido 30 de Abril de 2019].
- [22] Deepdrive: self-driving AI. <https://deepdrive.io/>, 2017. [Accedido 30 de Abril de 2019].
- [23] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L. Cun. Off-road obstacle avoidance through end-to-end learning. In Y. Weiss, B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 739–746. MIT Press, 2006.
- [24] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.
- [25] Jinkyu Kim and John F. Canny. Interpretable learning for self-driving cars by visualizing causal attention. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2961–2969, 2017.
- [26] Mariusz Bojarski, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Larry J. Ackel, Urs Muller, Philip Yeres, and Karol Zieba. Visualbackprop: Efficient visualization of cnns for autonomous driving. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8, 2018.
- [27] Javier del Egio, Luis Bergasa, Eduardo Romera, Carlos Gómez Huélamo, Javier Araluce, and Rafael Barea. *Self-driving a Car in Simulation Through a CNN: Proceedings of the 19th International Workshop of Physical Agents (WAF 2018), November 22-23, 2018, Madrid, Spain*, pages 31–43. 01 2019.
- [28] Ana I. Maqueda, Antonio Loquercio, Guillermo Gallego, Narciso García, and Davide Scaramuzza. Event-based vision meets deep learning on steering prediction for self-driving cars. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5419–5427, 2018.
- [29] Jonas Heylen, Seppe Iven, Bert De Brabandere, M. Oramas JoséOramas, Luc Van Gool, and Tinne Tuytelaars. From pixels to actions: Learning to drive a car with

- deep neural networks. *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 606–615, 2018.
- [30] Hesham M. Eraqi, Mohamed N. Moustafa, and Jens Honer. End-to-end deep learning for steering autonomous vehicles considering temporal dependencies. *CoRR*, abs/1710.03804, 2017.
- [31] Lu Chi and Yadong Mu. Deep steering: Learning end-to-end driving model from spatial and temporal visual cues. *CoRR*, abs/1708.03798, 2017.
- [32] Simone Ceriani and Martino Migliavacca. Middleware in robotics. *Advanced Methods of Information Technology for Autonomous Robotics*. [Accedido 14 de Junio de 2019].
- [33] Gazebo. Tutorial: ROS integration overview. http://gazebosim.org/tutorials? tut=ros_overview, 2014. [Accedido 15 de Junio de 2019].
- [34] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming Robots with ROS, A PRACTICAL INTRODUCTION TO THE ROBOT OPERATING SYSTEM*. O Reilly, 2015. http://marte.aslab.upm.es/redmine/files/dmsf/p_drone-testbed/170324115730_268_Quigley_-_Programming_Robots_with_ROS.pdf.
- [35] Jan Bodnar. Introduction to PyQt5. <http://zetcode.com/gui/pyqt5/introduction/>, 2017. [Accedido 10 de Junio de 2019].
- [36] What is PyQt?, *Riverbank Computing Limited*. <https://riverbankcomputing.com/software/pyqt/intro>, 2016. [Accedido 9 de Junio de 2019].
- [37] Keras team. Licencia de Keras, *Github*. <https://github.com/keras-team/keras/blob/master/LICENSE>, 2019. [Accedido 12 de Junio de 2019].
- [38] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015. [Accedido 13 de Junio de 2019].
- [39] Theano. Ops for neural networks, *Theano*. http://deeplearning.net/software/theano/library/tensor/nnet/nnet.html#theano.tensor.nnet.categorical_crossentropy, 2017. [Accedido 13 de Junio de 2019].

CAPÍTULO 7. CONCLUSIONES

- [40] TensorFlow. TensorBoard: Visualizing Learning, *TensorFlow Core*. https://www.tensorflow.org/guide/summaries_and_tensorboard, 2019. [Accedido 14 de Junio de 2019].
- [41] The HDF5 Library & File Format. <https://www.hdfgroup.org/solutions/hdf5/>, 2019. [Accedido 12 de Junio de 2019].
- [42] What is HDF5? <https://support.hdfgroup.org/HDF5/whatishdf5.html>, 2019. [Accedido 12 de Junio de 2019].
- [43] Vanessa Fernández Martínez. Práctica 1: Follow line (Prueba 1), *Visión en Robótica*. <http://vanessavisionrobotica.blogspot.com/2018/03/practica-1-follow-line-prueba-1.html>, 2018. [Accedido 16 de Junio de 2019].
- [44] Vanessa Fernández Martínez. Práctica 1: Follow line (Prueba 2), *Visión en Robótica*. <http://vanessavisionrobotica.blogspot.com/2018/05/practica-1-follow-line-prueba-2.html>, 2018. [Accedido 16 de Junio de 2019].
- [45] Robotics-Academy. Visual follow-line behavior on a Formula1, *JdeRobot*. https://jderobot.org/Robotics-Academy#Visual_follow-line_behavior_on_a_Formula1, 2019. [Accedido 16 de Junio de 2019].
- [46] Jordi Torres. *Deep Learning, Introducción práctica con Keras*. WATCH THIS SPACE, 2018. <https://torres.ai/deep-learning-inteligencia-artificial-keras>.
- [47] Ibáñez. De 0 a 5: cuáles son los diferentes niveles de conducción autónoma, a fondo,xataka. <https://www.xataka.com/automovil/de-0-a-5-cuales-son-los-diferentes-niveles-de-conduccion-autonoma>, 2017. [Accedido 29 de Mayo de 2019].
- [48] Diccionario de Internet y Tecnologías de la Información y la Comunicación. Paradoja de Moravec : que es, definición y significado, descargar videos y fotos, *Internet y Tecnologías de la Información y la Comunicación*. <https://www.paraisodigital.org/internet/11-paradoja-de-moravec-que-es-definicion-y-significado-descargar-videos-y-fotos.html>, 2018. [Accedido 29 de Mayo de 2019].

- [49] Betzaida Zambrano and Jorge Hernández. Técnicas y campos de la Inteligencia Artificial. <https://es.slideshare.net/beshi/tecnicas-y-camposdelaibzjh>, 2013. [Accedido 29 de Mayo de 2019].
- [50] Indra. Una imagen vale más que mil palabras: Visión Artificial. https://www.minsait.com/sites/default/files/newsroom_documents/unaimagenvalemasquemilpalabras.pdf. [Accedido 30 de Mayo de 2019].
- [51] Pedro Javier Oscar Sergio Alejandro, Vicente and Carlos. Introducción al Diseño de Micro Robots Móviles2009/10: Sistemas De Visión Artificial. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&uact=8&ved=2ahUKEwjm3_TF9tziAhUuxYUKHdmADZQQFjAFegQIAxAC&url=http%3A%2F%2Fwww.roboticaeducativa.org%2Fmod%2Fresource%2Fview.php%3Fid%3D2051&usg=A0vVaw1WxJXLKEuuu7WpK08VivRb, 2010. [Accedido 30 de Mayo de 2019].
- [52] AbdelmalikMoujahid Pedro Larra naga, I nakiInza. Tema8. RedesNeuronale. <http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t8neuronales.pdf>. [Accedido 31 de Mayo de 2019].
- [53] Skymind. A Beginner's Guide to Neural Networks and Deep Learning, *A.I. Wiki*. <https://skymind.ai/wiki/neural-network>. [Accedido 31 de Mayo de 2019].
- [54] Antonio Blanco Emilio Soria. Redes neuronales artificiales. https://www.acta.es/medios/articulos/informatica_y_computacion/019023.pdf. [Accedido 31 de Mayo de 2019].
- [55] Fernando Sancho Caparrini. Redes Neuronales: una visión superficial. <http://www.cs.us.es/~fsancho/?e=72>. [Accedido 31 de Mayo de 2019].
- [56] Damián Jorge Matich. Redes Neuronales: Conceptos Básicos y Aplicaciones. https://www.frro.utn.edu.ar/repositorio/catedras/quimica/5_anio/orientadora1/monografias/matich-redesneuronales.pdf, 2001. [Accedido 31 de Mayo de 2019].
- [57] Mayank Mishra. Convolutional Neural Networks, Explained. <https://www.datascience.com/blog/convolutional-neural-network>, 2019. [Accedido 31 de Mayo de 2019].

- [58] Raul E. Lopez Briega. Redes neuronales convolucionales con TensorFlow. <https://relopezbriega.github.io/blog/2016/08/02/redes-neuronales-convolucionales-con-tensorflow/>, 2016. [Accedido 31 de Mayo de 2019].
- [59] Omar Emilio Contreras Zaragoza. *Desarrollo de una red neuronal convolucional para el procesamiento de imágenes placentarias*. PhD thesis, Universidad Nacional Autónoma de México, 2018. [Accedido 1 de Junio de 2019].
- [60] Pablo Pastor Martín. *Usando Redes Neuronales Convolucionales Para Convertir Características Visuales en Estímulos Sonoros*. PhD thesis, Universidad de La Laguna, 2018. [Accedido 1 de Junio de 2019].
- [61] John Marturet Rodrigo. *Evaluación de redes neuronales convolucionales para la clasificación de imágenes histológicas de cáncer colorrectal mediante transferencia de aprendizaje*. PhD thesis, Universitat Oberta de Catalunya, 2018. [Accedido 1 de Junio de 2019].
- [62] Jaime Durán Suárez. *Redes Neuronales Convolucionales en R*. PhD thesis, Escuela Técnica Superior de Ingeniería, Universidad de Sevilla, 2017. [Accedido 1 de Junio de 2019].
- [63] José Francisco Núñez Castro. *Aprendizaje automático en fusión nuclear con Deep Learning*. PhD thesis, Pontifica Universidad Católica de Valparaíso, 2017. [Accedido 1 de Junio de 2019].
- [64] Oinkina and Hakyll. Understanding LSTM Networks. <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>, 2015. [Accedido 1 de Junio de 2019].
- [65] Secretaría de Estado de Educación y Formación Profesional. Visión Artificial: Aplicación práctica de la visión artificial en el control de procesos industriales. http://visionartificial.fpcat.cat/wp-content/uploads/UD_1_didac_Conceptos_previos.pdf, 2012. [Accedido 29 de Mayo de 2019].
- [66] Philipp Kandal. From Zero to Waymo: The Story of Google's Driverless Car. <https://kandal.com/essays/from-zero-to-waymo-the-story-of-googles-driverless-car>, 2017. [Accedido 30 de Mayo de 2019].

- [67] David Villarreal. BMW y Mercedes-Benz unen fuerzas para desarrollar coches autónomos, Diariomotor. <https://www.diariomotor.com/noticia/bmw-mercedes-unen-fuerzas-coche-autonomo/>, 2019. [Accedido 30 de Mayo de 2019].
- [68] Gustav von Zitzewitz. Survey of neural networks in autonomous driving. 07 2017.
- [69] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntaek Lee, Xinyan Yan, Evangelos A. Theodorou, and Byron Boots. Agile autonomous driving using end-to-end deep imitation learning. In *Robotics: Science and Systems*, 2018.
- [70] Lucas Martín. Gazebo, simulador de robótica, *Automatismos Mar del Plata*. <http://www.automatismos-mdq.com.ar/blog/2017/01/gazebo-simulador-de-robotica.html>, 2017. [Accedido 30 de Abril de 2019].
- [71] Gazebo Simulator: simular un robot nunca fue tan fácil, *Robologs*. <https://robologs.net/2016/06/25/gazebo-simulator-simular-un-robot-nunca-fue-tan-facil/>, 2016. [Accedido 30 de Abril de 2019].
- [72] Follow line: JdeRobot RoboticsAcademy. https://github.com/JdeRobot/RoboticsAcademy/tree/master/exercises/follow_line, 2017. [Accedido 30 de Septiembre de 2018].
- [73] Javier Antón Alonso and Xuebo Zhu Chen. *Estudio y simulación de un vehículo autopilotado en Unity 5 haciendo uso de algoritmos de aprendizaje automático*. PhD thesis, Universidad Complutense Madrid, 2018. [Accedido 2 de Junio de 2019].
- [74] Joshué Manuel Pérez Rastelli. *Agentes de control de vehículos autónomos en entornos urbanos y autovías*. PhD thesis, Universidad Complutense Madrid, 2012. [Accedido 2 de Junio de 2019].
- [75] Chenyi Chen, Ari Seff, Alain L. Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2722–2730, 2015.
- [76] Antonio Paladini. *End-to-end Models for Lane Centering in Autonomous Driving*. PhD thesis, Politecnico di Milano, 2018. [Accedido 6 de Junio de 2019].

- [77] Simon Kardell and Mattias Kuosku. *Autonomous vehicle control via deep reinforcement learning*. PhD thesis, Chalmers University of Technology, 2017. [Accedido 7 de Junio de 2019].
- [78] David Ungurean. *DeepRCar: An Autonomous Car Model*. PhD thesis, Faculty of Information Technology CTU in Prague, 2018. [Accedido 7 de Junio de 2019].
- [79] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J. Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3357–3364, 2017.
- [80] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39:677–691, 2015.
- [81] Tharindu Fernando, Simon Denman, Sridha Sridharan, and Clinton Fookes. Going deeper: Autonomous steering with neural memory networks. *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*, pages 214–221, 2017.
- [82] Arthur Emidio T. Ferreira, Ana Paula Goncalves Soares de Almeida, and Flavio de Barros Vidal. Autonomous vehicle steering wheel estimation from a video using multichannel convolutional neural networks. In *ICINCO*, 2018.
- [83] David Gerónimo Gómez. *Visión Artificial aplicada a vehículos inteligentes*. PhD thesis, Universitat Autònoma de Barcelona, 2004. [Accedido 1 de Junio de 2019].
- [84] Keras documentation. Keras: The Python Deep Learning library. <https://keras.io/>. [Accedido 10 de Junio de 2019].
- [85] Jesús Utrera Burgal. Deep Learning básico con Keras (Parte 1). <https://enmielocalfunciona.io/deep-learning-basico-con-keras-parte-1/>. [Accedido 10 de Junio de 2019].
- [86] Jason Brownlee. Introduction to Python Deep Learning with Keras. <https://machinelearningmastery.com/introduction-to-python-deep-learning-with-keras/>

- introduction-python-deep-learning-library-keras/l, 2016. [Accedido 12 de Junio de 2019].
- [87] Carlos Santana. Historia de Python. <https://www.codejobs.biz/es/blog/2013/03/03/historia-de-python>, 2013. [Accedido 11 de Junio de 2019].
- [88] Python Software Foundation. Tutorial de Python. <http://docs.python.org.ar/tutorial/3/real-index.html>, 2017. [Accedido 11 de Junio de 2019].
- [89] Python, EcuRed. <https://www.ecured.cu/Python>, 2017. [Accedido 11 de Junio de 2019].
- [90] J.M Cañas. *Programación de robots con la plataforma Jderobot*. PhD thesis, Universidad de Málaga, 2009. [Accedido 12 de Junio de 2019].
- [91] Julio Manuel Vega. *Navegación y autolocalización de un robot guía de visitantes*. PhD thesis, Universidad Rey Juan Carlos. Ingeniería Informática, 2009. [Accedido 12 de Junio de 2019].
- [92] Página Oficial de OpenCV. <http://opencv.org/>, 2019. [Accedido 12 de Junio de 2019].
- [93] V. M. Arévalo, J. González, and G. Ambrosio. *La Librería de Visión Artificial OpenCV, Aplicación a la Docencia e Investigación*. PhD thesis, Dpto. De Ingeniería de Sistemas y Automática, Universidad de Málaga, 2004. [Accedido 12 de Junio de 2019].
- [94] Ji Yang. ReLU and Softmax Activation Functions. <https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions>, 2017. [Accedido 13 de Junio de 2019].
- [95] ROS.org. Topics de ROS. <http://wiki.ros.org/Topics>, 2019. [Accedido 15 de Junio de 2019].
- [96] Gazebo. Gazebo plugins in ROS. http://gazebosim.org/tutorials?tut=ros_gzplugins, 2014. [Accedido 15 de Junio de 2019].

CAPÍTULO 7. CONCLUSIONES

- [97] Packt. ROS Architecture and Concepts. <https://hub.packtpub.com/ros-architecture-and-concepts/>, 2016. [Accedido 15 de Junio de 2019].
- [98] JdeRobot. Página del repositorio dl-objectdetector. <https://github.com/JdeRobot/dl-objectdetector>, 2018. [Accedido 18 de Junio de 2019].
- [99] Raul E. Lopez Briega. Machine Learning con Python, *Matemáticas, análisis de datos y python*. <https://relopezbriega.github.io/blog/2015/10/10/machine-learning-con-python/>, 2015. [Accedido 22 de Junio de 2019].
- [100] Juan Ignacio Bagnato. Qué es overfitting y underfitting y cómo solucionarlo, *Aprende Machine Learning*. <https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/>, 2017. [Accedido 23 de Junio de 2019].
- [101] Vanessa Fernández Martínez. *Nuevas Prácticas en el Entorno Docente de Robótica JdeRobot-Academy*. PhD thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, 2017. [Accedido 23 de Junio de 2019].
- [102] Ignacio Condés Menchén. *Deep Learning Applications for Robotics using TensorFlow and JdeRobot*. PhD thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, 2018. [Accedido 24 de Junio de 2019].
- [103] Marcos Pieras Sagardoy. *Visual people tracking with deep learning detection and feature tracking*. PhD thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, 2017. [Accedido 24 de Junio de 2019].
- [104] Nuria Oyaga de Frutos. *Análisis de Aprendizaje Profundo con la plataforma Caffe*. PhD thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, 2017. [Accedido 24 de Junio de 2019].
- [105] David Pascual Hernández. *Study of Convolutional Neural Networks using Keras Framework*. PhD thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, 2017. [Accedido 24 de Junio de 2019].

CAPÍTULO 7. CONCLUSIONES

- [106] Pablo Moreno Vera. *Nuevas Prácticas Docentes en el Entorno Robotics-Academy.* PhD thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, 2019. [Accedido 24 de Junio de 2019].