



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
INFORMÁTICA

MÁSTER UNIVERSITARIO EN VISIÓN ARTIFICIAL

TRABAJO FIN DE MÁSTER

Conducción autónoma de un vehículo en
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

En primer lugar, me gustaría dar las gracias a mi familia por el apoyo que me ha dado a lo largo de este tiempo. Todos ellos han mostrado interés por el trabajo y lo han hecho más llevadero. Sobre todo quiero mencionar a mis padres y a mi hermana, los cuales me han animado, me han dado todo su cariño, y me han ayudado en los momentos más difíciles, pues sin ellos no hubiese sido posible conseguirlo.

Gracias a José María y a Fran, por su dedicación y apoyo durante todos estos meses de desarrollo del trabajo, por los ánimos, por su paciencia, por sus ideas y su ayuda. Y por supuesto, por haberme transmitido su pasión por la visión artificial, la robótica y la investigación.

Por otro lado, quiero dar las gracias a mis compañeros de la universidad que me han animado durante todos estos años y han compartido conmigo el estrés de los exámenes: Nuria, Irene, Carolina, Fran, Aitor, Carlos, Nacho, Miguel Ángel, Celia, etc. Además, gracias a mis amigos que me han apoyado durante tantos años en los buenos y los malos momentos: Marcos, Nerea, Lizaveta, Chaimae, Laura, Manal, Marian, Isabella, etc.

Por último, no puedo olvidarme de dar las gracias al CNAH, no solamente por la natación, sino sobre todo por la gente que lo forma, especialmente el equipo femenino. Gracias a las más jóvenes del equipo, por sacarme una sonrisa en todo momento. A las mayores: Jessi, Concha, María, Elena y Laura, que me han hecho desconectar cada día del estrés y pasármelo como nunca. No puedo olvidarme, de Manolo Revilla, que desde que llegué al club me ha tratado como una amiga y se ha preocupado por mí.

¡Muchas gracias a todos!

Resumen

En la última década, en Visión Artificial (VA) se está estudiando ampliamente la conducción autónoma. Los humanos somos capaces de mirar a la carretera y saber al instante que acción llevar a cabo. 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 materializar comportamiento autónomo en vehículos. En este proyecto se estudia la conducción autónoma en simulación mediante redes neuronales basadas en información visual.

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. Además, el coche deberá ser capaz de conducir en diferentes entornos. Por ello, se ha creado un conjunto de datos propio a partir de un piloto autónomo programado explícitamente que conduce de forma autónoma usando visión.

El objetivo principal de este trabajo es comparar diferentes modelos de redes neuronales que se pueden emplear para la conducción autónoma. Por este motivo, se han estudiado y probado diferentes arquitecturas de redes. Primero, se ha estudiado el empleo de redes neuronales convolucionales *de clasificación* en conducción autónoma, realizando múltiples pruebas para tratar de conseguir la red más robusta posible y emplearla en el pilotaje del vehículo. Segundo, se ha estudiado el empleo de redes neuronales convolucionales y *recurrentes de regresión* en conducción autónoma. Se han llevado a cabo diversas pruebas también para intentar conseguir la red más robusta posible y emplearla en el pilotaje del coche.

Índice general

Índice de figuras	VII
Índice de tablas	VIII
1. Introducción	2
1.1. Visión artificial	2
1.2. Conducción autónoma	5
1.3. Redes neuronales artificiales	8
1.3.1. Redes neuronales convolucionales	9
1.3.2. Redes neuronales recurrentes	10
1.3.3. Tipos de capas	11
1.3.3.1. Capa Convolucional	11
1.3.3.2. Capa de <i>Pooling</i>	13
1.3.3.3. Capa <i>Fully connected</i>	13
1.3.3.4. Capa LSTM	14
2. Objetivos	15
2.1. Objetivos	15
2.2. Requisitos	16
2.3. Metodología	17
2.4. Plan de trabajo	18
3. Estado del arte	20
3.1. Bases de datos para conducción autónoma	20
3.1.1. Comma.ai	21
3.1.2. Udacity	21
3.1.3. SAIC Dataset	22
3.2. Simuladores para conducción autónoma	23
3.2.1. CARLA	23
3.2.2. Gazebo	24

3.2.3.	Udacity’s Self-Driving Car Simulator	25
3.2.4.	Deepdrive 2.0	26
3.3.	Redes neuronales	27
3.3.1.	Redes neuronales convolucionales	27
3.3.2.	Redes neuronales recurrentes	33
3.4.	Infraestructura empleada	39
3.4.1.	Simulador Gazebo	39
3.4.2.	Entorno ROS	41
3.4.3.	Entorno JdeRobot	43
3.4.4.	Lenguaje Python	44
3.4.5.	Biblioteca OpenCV	44
3.4.6.	Interfaces gráficas con PyQt	45
3.4.7.	Middleware neuronal Keras	46
3.4.7.1.	Modelos en Keras	47
3.4.7.2.	Capas en Keras	50
3.4.7.3.	<i>Callbacks</i> en Keras	53
3.4.8.	Formato de archivo HDF5	54
4.	Infraestructura desarrollada	56
4.1.	Circuitos de carreras en Gazebo	56
4.2.	Piloto autónomo programado	62
4.3.	Creación de conjunto de datos para entrenamiento neuronal	67
4.4.	Piloto autónomo basado en redes neuronales	73
4.4.1.	Interfaz gráfica	77
5.	Redes de clasificación	80
5.1.	Arquitecturas de red	81
5.1.1.	LeNet-5	81
5.1.2.	SmallerVGGNet	82
5.2.	Experimentos	82
5.2.1.	Métricas de evaluación	84
5.2.2.	Imágenes de distintas dimensiones	87
5.2.3.	Número de clases	89
5.2.4.	Influencia de los datos de entrenamiento	92

6. Redes de regresión	98
6.1. Arquitecturas de red	99
6.1.1. PilotNet	99
6.1.2. TinyPilotNet	100
6.1.3. LSTM-TinyPilotNet	100
6.1.4. DeepestLSTM-TinyPilotNet	101
6.2. Experimentos	103
6.2.1. Métricas de evaluación	103
6.2.2. Comparativa entre redes de regresión	106
6.2.3. Aumentado de los datos	111
6.2.4. Imágenes de distintas dimensiones	112
6.2.5. Tipo de imagen de entrada	116
7. Conclusiones	121
7.1. Conclusiones	121
7.2. Trabajos futuros	125
Bibliografía	139

Índice de figuras

1.1. Navegación en robótica mediante VA	4
1.2. Detección de cáncer de mama	4
1.3. Detección de contenedores	5
1.4. Conducción autónoma	5
1.5. Estructura de CNN	10
1.6. Esquema de Redes Neuronales Recurrentes (RNN)	11
1.7. Ejemplo de operación de convolución	12
1.8. Ejemplo de capa <i>max pooling</i>	13
1.9. Unidad LSTM	14
2.1. Modelo en espiral	18
3.1. Simulador CARLA.	24
3.2. Simulador Gazebo.	25
3.3. Simulador Udacity's Self-Driving Car Simulator.	26
3.4. Simulador Deepdrive.	27
3.5. Arquitectura Pilotnet.	28
3.6. Ejemplos de objetos salientes para varias imágenes de entrada.	30
3.7. Arquitectura TinyPilotnet.	31
3.8. Estructura de red ControlNet.	35
3.9. Arquitectura C-LSTM.	37
3.10. Simulador Gazebo.	40
3.11. Interfaz del conjunto de paquetes gazebo_ros_pkgs	42
3.12. Modelo f1ROS	43
3.13. Funciones de OpenCV	45
3.14. Función de activación <i>ReLU</i>	52
4.1. Modelo pistaSimple	57
4.2. Modelo monacoLine	58
4.3. Modelo nurburgrinLine	58
4.4. Modelo curveGP	59

4.5. Modelo pista_simple	59
4.6. Filtrado de color	63
4.7. Filtrado de color con cierre	64
4.8. Representación pares L1-L2 (<i>Dataset1</i> contra conducción)	70
4.9. Representación pares L1-L2 (nuevo <i>Dataset</i> contra conducción)	71
4.10. Representación pares L1-L2 (<i>Dataset_Curves</i> contra conducción)	72
4.11. Análisis de pares L1-L2 (<i>Dataset</i>) para w	72
4.12. Análisis de pares L1-L2 (<i>Dataset</i>) para v	73
4.13. Estructura de la aplicación de control visual basada en redes neuronales . .	74
4.14. Interfaz gráfica (GUI)	78
5.1. Arquitectura Lenet-5.	81
5.2. Arquitectura SmallerVGGNet.	83
5.3. Imagen completa (izquierda) e imagen recortada (derecha)	87
5.4. Pilotaje del coche en el circuito nurburgrinLine	96
5.5. Pilotaje del coche en el circuito monacoLine	96
5.6. Pilotaje del coche en el circuito pistaSimple	97
5.7. Pilotaje del coche en el circuito pistaSimple	97
6.1. Arquitectura PilotNet.	100
6.2. Arquitectura TinyPilotNet.	101
6.3. Arquitectura LSTM-TinyPilotNet.	102
6.4. Arquitectura DeepestLSTM-TinyPilotNet.	102
6.5. Pilotaje del coche en el circuito monacoLine	109
6.6. Pilotaje del coche en el circuito nurburgrinLine	109
6.7. Pilotaje del coche en el circuito curveGP	110
6.8. Pilotaje del coche en el circuito pista_simple	110
6.9. Imagen de la cámara	112
6.10. Imagen tras realizar la operación <i>flip</i>	113
6.11. Pilotaje del coche en el circuito pistaSimple	114
6.12. Imagen diferencia	117

Índice de tablas

4.1. Resultados del Pilotaje autónomo explícito	66
5.1. Métricas de test de redes de clasificación (w, imagen recortada)	86
5.2. Métricas de test de redes de clasificación (v, imagen recortada)	86
5.3. Resultados de conducción con redes de clasificación (imagen completa e imagen recortada)	88
5.4. Resultados de conducción con redes de clasificación modificando la combinación del número de clases (imagen recortada)	91
5.5. Resultados de conducción con redes de clasificación (estudio de la influencia de los datos de entrenamiento)	95
6.1. Métricas de test de redes de regresión (v, imagen completa)	105
6.2. Métricas de test de redes de regresión (w, imagen completa)	105
6.3. Resultados de conducción con redes neuronales de regresión (imagen completa)	108
6.4. Resultados de conducción con redes neuronales recurrentes de regresión (imagen completa)	110
6.5. Resultados de conducción con redes neuronales de regresión (imagen recortada)	114
6.6. Resultados de conducción con redes neuronales recurrentes de regresión (imagen recortada)	115
6.7. Resultados de conducción con redes neuronales de regresión introduciendo temporalidad (imagen completa)	118
6.8. Métricas de test de redes de regresión (v, imagen completa)	119
6.9. Métricas de test de redes de regresión (w, imagen completa)	119

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. Visión artificial

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 aún muy complejas para las máquinas.

La IA comprende diferentes campos: *Machine Learning*, *Knowledge Engineering*, Lingüísti-

ca 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 VA 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.

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 humanos, 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 materializar comportamiento autónomo en vehículos.

Un claro ejemplo, es la navegación en robótica (Figura 1.1), 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 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 1.1: 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 1.2) con hasta cinco años de anticipación.

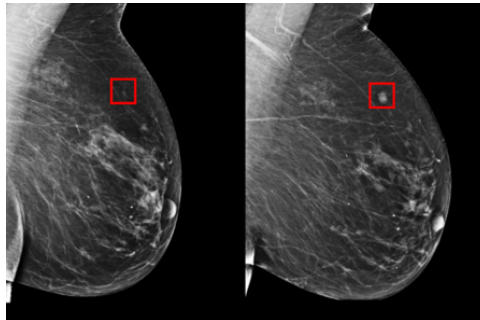


Figura 1.2: Detección de cáncer de mama

Una posible aplicación es el mantenimiento e inventariado urbano. Es posible identificar problemas en instalaciones y mobiliario urbano (averías, mal estado de contenedores (Figura 1.3), 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 1.3: 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 1.4) son: los sistemas de aviso de cambio de carril, o de control de velocidad de crucero.

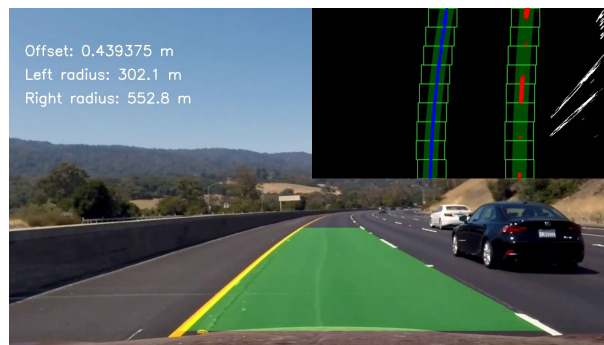


Figura 1.4: Conducción autónoma

1.2. 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.

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 hincapié en la investigación de la conducción autónoma con el fin de reducir 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 escenarios controlados y en situaciones específicas de conducción. El conductor debe seguir prestando 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 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.

1.3. 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 se organizan por capas.

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 generar resultados satisfactorios 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.

1.3.1. Redes neuronales convolucionales

Las Redes Neuronales Convolucionales (CNN) son una clase de red neuronal artificial profunda que se emplea 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 1.5 [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 de la imagen. 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.

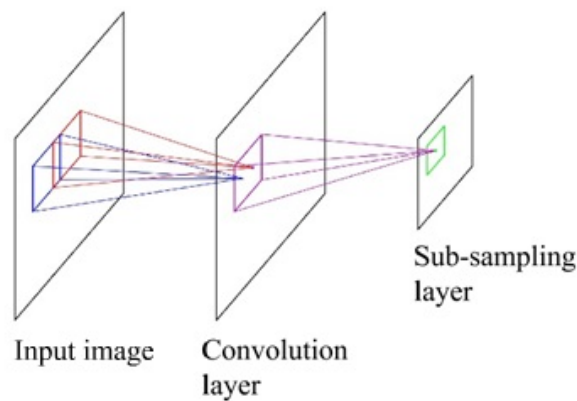


Figura 1.5: Estructura de CNN

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 disminuyen 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.

1.3.2. 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 1.6 se puede ver un esquema de RNN.

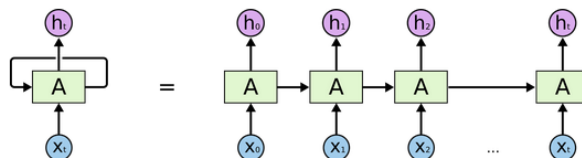


Figura 1.6: Esquema de Redes Neuronales Recurrentes (RNN)

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 periodos de tiempo largo. Se explicará más acerca de este tipo de red en las siguientes subsecciones.

1.3.3. Tipos de capas

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

1.3.3.1. Capa Convolutiva

Las capas convolucionales son las más importantes de una CNN. La operación de convolución (Figura 1.7) 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.

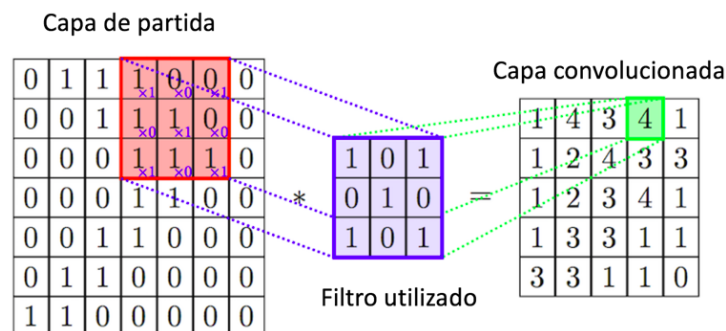


Figura 1.7: Ejemplo de operación de convolución

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.

1.3.3.2. 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 1.8
- *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.

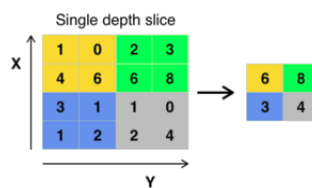


Figura 1.8: Ejemplo de capa *max pooling*

1.3.3.3. 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.

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

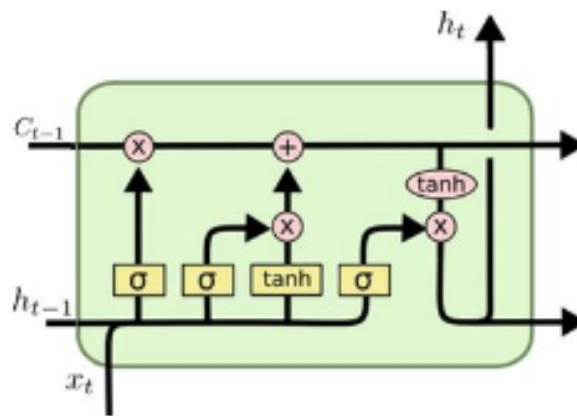


Figura 1.9: Unidad LSTM

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 de entrenamiento y de prueba serán diferentes para que el vehículo sea capaz de aprender diferentes estímulos que le permitan conducir en situaciones variadas.

Este objetivo genérico se ha articulado en cuatro subobjetivos concretos:

1. Se creará una aplicación de control visual con la infraestructura necesaria que se comunica con el simulador Gazebo, donde se podrá ver el resultado de la predicción de las redes neuronales. Esta aplicación tendrá diferentes ingredientes:
 - Infraestructura de conexión en el simulador, tanto para recoger imágenes de la cámara a bordo del coche como para enviar órdenes al acelerador y al volante.
 - Permitirá cargar y emplear diferentes redes neuronales, además de incluir una GUI.
 - En un fichero específico se guarda el código que genera las órdenes de velocidad predichas por la red neuronal y se envían al coche simulado.

2. Se crearán varias bases de datos propias de conducción autónoma con un algoritmo de pilotaje basado en visión sencillo, que permitan el entrenamiento supervisado de diferentes redes neuronales.
3. Se realizará un estudio y se llevarán a cabo pruebas de diferentes modelos de redes neuronales convolucionales de clasificación que se pueden emplear para la conducción autónoma.
4. Se realizará un estudio y se llevarán a cabo pruebas de diferentes modelos de redes neuronales convolucionales y redes neuronales recurrentes de regresión que se pueden emplear para la conducción autónoma.

2.2. Requisitos

El proyecto se desarrollará basándose en los subobjetivos mencionados anteriormente y tendrá que ajustarse a los requisitos de partida del proyecto:

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 la organización JdeRobot ¹. Este modelo se encuentra disponible en el repositorio de Github JdeRobot-assets ².
2. Se empleará el *middleware* robótico ROS, en concreto en la versión *ROS Kinetic*. Este *middleware* que simplifica el desarrollo de software robótico se explicará en mayor detalle en el Capítulo 3.
3. El sistema operativo que se empleará en este proyecto será Ubuntu 16.04.
4. El lenguaje de desarrollo empleado será Python. Debido a la compatibilidad 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.

¹https://jderobot.org/Main_Page

²<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 siguiendo 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 los tutores.

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 subtarefas 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: (1) Se concretan los objetivos específicos que deben cumplirse para que el ciclo actual se considere terminado en función de los objetivos finales; (2) se realiza un análisis detallado de cada posible riesgo que pueda tener el objetivo definido y se planean estrategias alternativas; (3) se desarrolla el producto y se realizan las pruebas necesarias; (4) se analizan los resultados obtenidos a través de las pruebas, y se planifica la siguiente iteración.

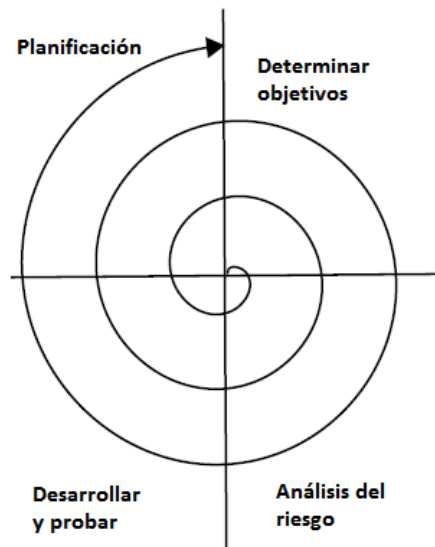


Figura 2.1: Modelo en espiral

Esta metodología se ha llevado a cabo mediante reuniones semanales con los tutores en las que 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 Github ³, que emplea el sistema de control de versiones. Además, se ha desarrollado una bitácora en la página de JdeRobot ⁴, 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 como software libre.

2.4. Plan de trabajo

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

³<https://github.com/RoboticsURJC-students/2017-tfm-vanessa-fernandez>

⁴<https://jderobot.org/Vmartinezf-tfm>

- 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 organización JdeRobot.
- Desarrollo de la infraestructura necesaria en Gazebo, y de una aplicación de control visual que permita la conducción del coche integrando una red entrenada en Keras.
- Creación de una base de datos que permita 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 la aplicación desarrollada.
- Estudio y mejora de redes neuronales convolucionales y recurrentes de regresión aplicadas a la conducción autónoma. Se realizarán diversas pruebas para intentar conseguir la red más robusta posible y emplearla en la aplicación desarrollada.

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.

Se presentan también 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, PyQt, Python como lenguaje de programación, Keras como entorno para el desarrollo de redes neuronales y 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 solo 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, en distintos 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 ¹ 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 fotogramas 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 sellos temporales en los que se midieron estas medidas de los sensores y los *time stamps* en que se capturaron los fotogramas de la cámara. Los datos de los sensores se capturan en bruto y los fotogramas 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 ² 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

¹<https://comma.ai/>

²<https://github.com/udacity/self-driving-car/tree/master/datasets>

profundo necesitan muchos datos, la compañía publicó 183 GB adicionales de datos de conducción.

Actualmente el conjunto de datos de Udacity [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 un *frame rate* de 30 fotogramas por segundo.

3.2. Simuladores para conducción autónoma

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 explican 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 metamandos 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).



Figura 3.1: Simulador CARLA.

- 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 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 el *middleware* robótico 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 Open Source Robotics Foundation (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

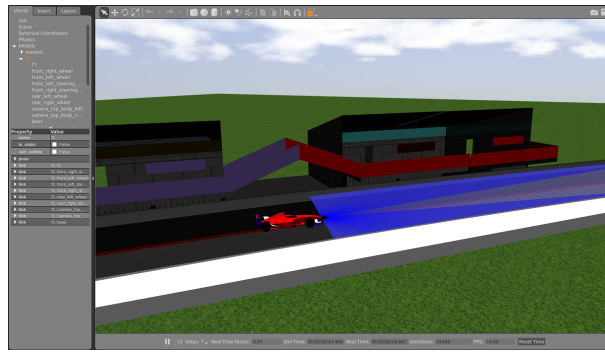


Figura 3.2: Simulador Gazebo.

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.

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 *Autonomous 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 al 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.



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

Cada instrucción de conducción contiene un ángulo de dirección y un dato de aceleración que cambia la dirección y la velocidad del automóvil.

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 a vehículos físicos. Para conseguir este propósito Deepdrive (Figura 3.4) 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.



Figura 3.4: Simulador Deepdrive.

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 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*.

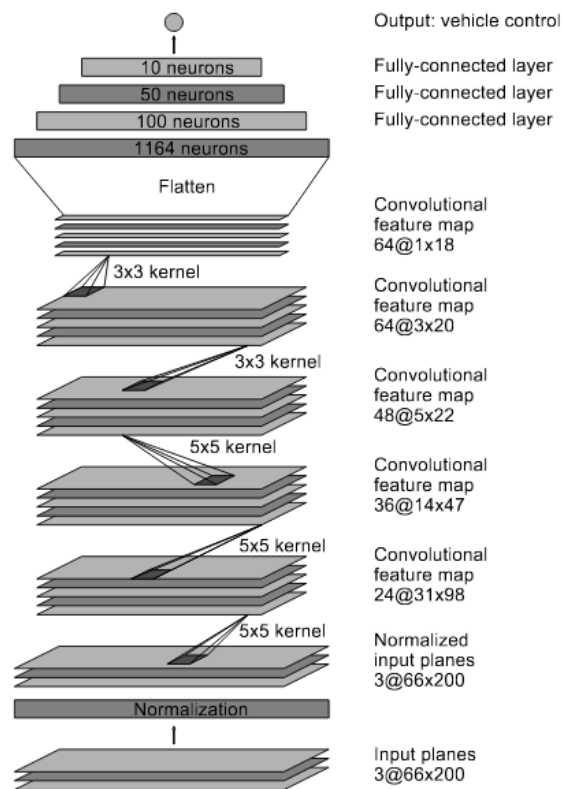


Figura 3.5: Arquitectura Pilotnet.

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 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 desarrolló un método para determinar qué elementos en la imagen de la carretera influyen más en la decisión de la dirección de PilotNet. Llamamos 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.



Figura 3.6: 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.

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 16×32 y un único canal, seguida por dos capas convolucionales de kernel 3×3 , 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 sólo producen cambios de intensidad de forma asíncrona en el momento en el que ocurren) para predecir el ángulo de giro

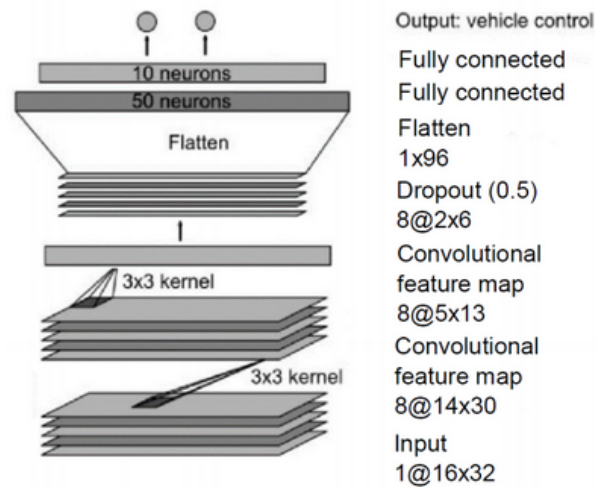


Figura 3.7: Arquitectura TinyPilotnet.

de un vehículo. Los eventos se convierten en fotogramas de eventos por acumulación de píxeles en un intervalo de tiempo constante. Posteriormente, una red neuronal profunda los asigna a los ángulos de dirección.

En este artículo inicialmente apilan los fotogramas 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 $160 \times 320 \times 3$ cambiaría 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 fotogramas 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 prometedoras 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 (RNN) 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 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 fotogramas. 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
- 2D Convolution, 16 filtros de tamaño 5x5
- Max Pooling, filtro de 3x3, *stride* de 2
- Fully connected, 50 neuronas

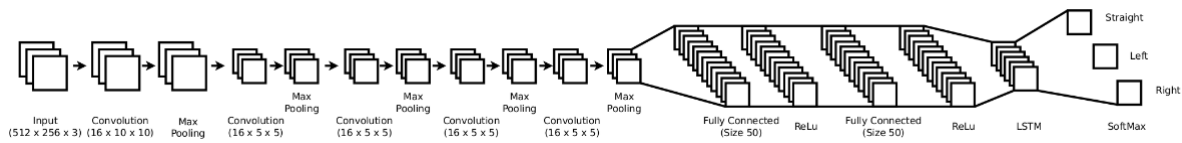


Figura 3.8: Estructura de red ControlNet.

- ReLu
- Fully connected, 50 neuronas
- 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 fotogramas 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 ángulos de dirección desde el mismo fotograma 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 fotogramas 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

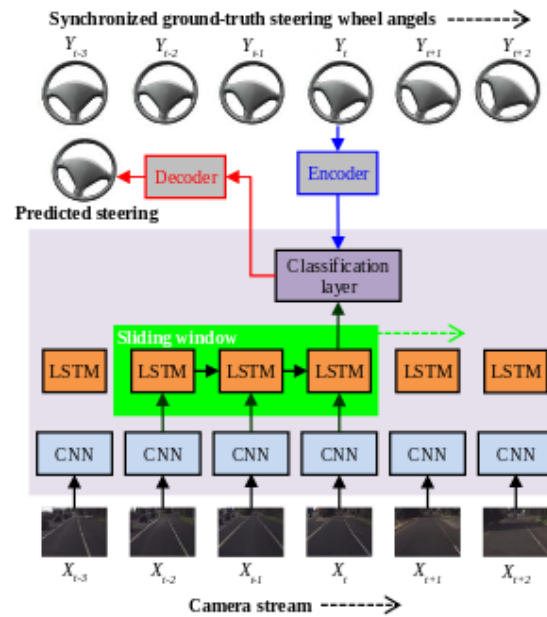


Figura 3.9: Arquitectura C-LSTM.

de 128 dimensiones. Para ello se añade un 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 influyen en la salida de la red, de las cuales algunas son influencias reales y otras espúreas. Su modelo predice comandos de ángulo de dirección continuos a partir de píxeles en bruto. El modelo predice el radio de giro inverso $\hat{u}t$, 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 espacial. Recopilan un cubo x_t 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 x_t . 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, M_t . La red neuronal de 5 capas (basada en PilotNet) emplea una pila de filtros 5×5 y 3×3 sin ninguna capa *pooling*, y por tanto la imagen de dimensiones 80×160 se procesa para producir un cubo de características $10 \times 20 \times 64$, conservando su relación de aspecto. Para extraer una prominencia visual local, primero muestrean aleatoriamente partículas de 2D con reemplazo sobre una imagen de entrada condicionada en el mapa de atención M_t . También emplean el eje de tiempo como la tercera dimensión para considerar las características temporales de las saliencias visuales, almacenando partículas 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 fotograma 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 fotogramas 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 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, conocida como DeepestLSTM-TinyPilotNet, introduce capas LSTM que 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. Esta red se explicará en mayor detalle en la Sección 6.1.4.

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, Gazebo ³ 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.).

³<http://gazebosim.org/>

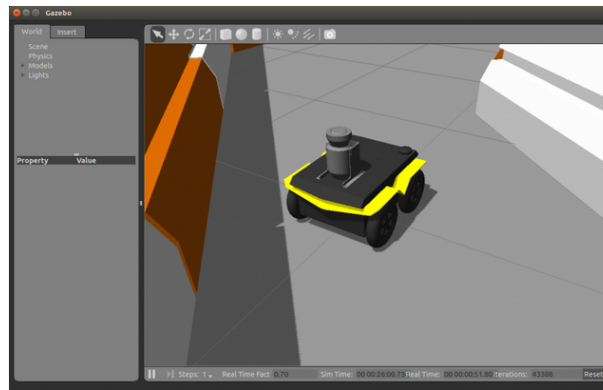


Figura 3.10: Simulador Gazebo.

En este trabajo se emplea la versión 7.15.0 de Gazebo. Gracias a él 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.

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). Cada uno de estos ficheros contiene una descripción completa de todos los elementos que forman 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 comportamiento 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 ROS

Robot Operating System (ROS) ⁴ [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` ⁵ [33]. Gazebo se integra con ROS mediante *ROS Messages*, servicios y reconfiguración dinámica. En la Figura 3.11 se puede observar una visión general de la interfaz `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.

⁴<https://www.ros.org/>

⁵http://wiki.ros.org/gazebo_ros_pkgs

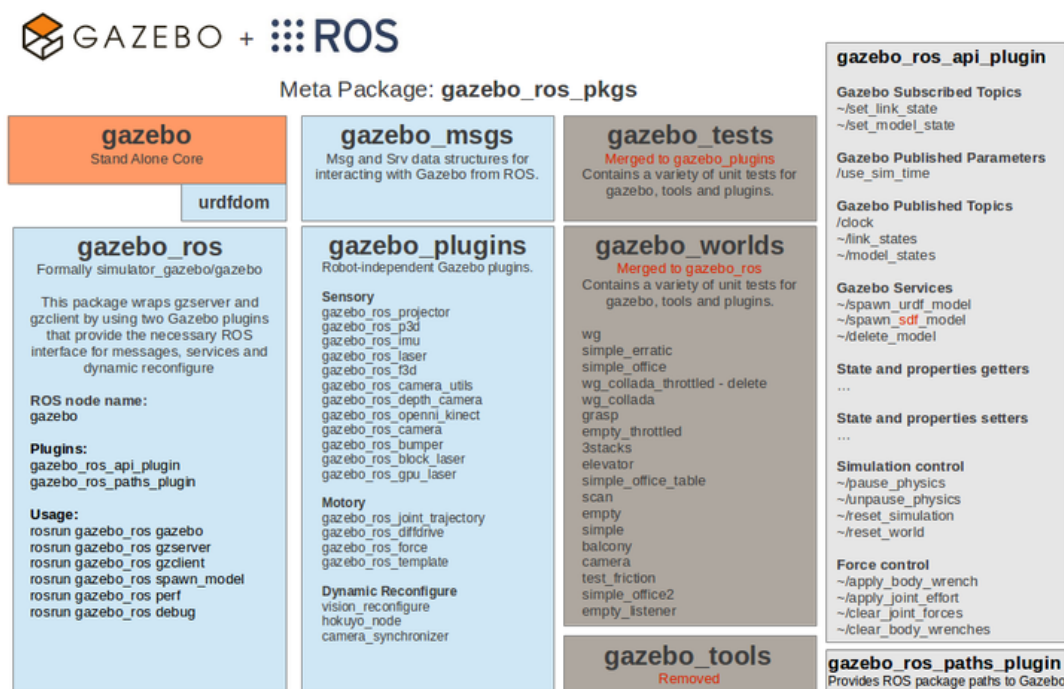


Figura 3.11: Interfaz del conjunto de paquetes gazebo_ros_pkgs

Los *topics* de ROS ⁶ [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 posible que existan varios suscriptores del mismo *topic*.

En ROS existen diversos *plugins* ⁷ 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

⁶<http://wiki.ros.org/Topics>

⁷http://wiki.ros.org/gazebo_plugins

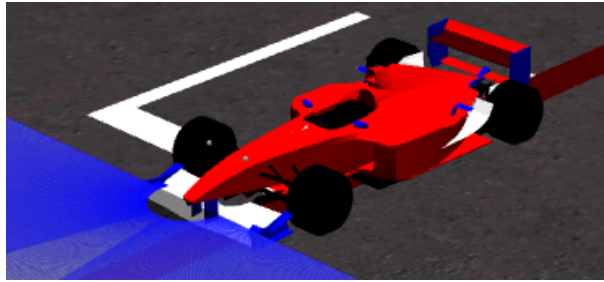


Figura 3.12: Modelo f1ROS

el coche utilizado en este proyecto.

3.4.3. Entorno JdeRobot

JdeRobot ⁸ 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 GPLv3 ⁹.

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 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.

En este proyecto hemos empleado el *driver* del coche de Fórmula1 (Figura 3.12) de la organización JdeRobot que está basado en ROS. En el *driver* del F1 se han empleado *plugins* de ROS con el fin de dotar de movimiento al modelo y captar las imágenes de la cámara del coche. En el desarrollo del proyecto de este TFM se empleará la versión 5.6.7 de JdeRobot, ya que es la última versión estable.

⁸http://jderobot.org/Main_Page

⁹<https://www.gnu.org/licenses/quick-guide-gplv3.html>

3.4.4. Lenguaje Python

Python ¹⁰ es un lenguaje de programación fácil de aprender y de alto nivel. Python incluye orientación a objetos, manejo de excepciones, listas, diccionarios, etc. 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.

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 el *middleware* ROS Kinetic. El código en el que está escrito la aplicación de control visual neuronal es Python.

3.4.5. Biblioteca OpenCV

OpenCV ¹¹ 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 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.

¹⁰<https://www.python.org/>

¹¹<http://opencv.org/>

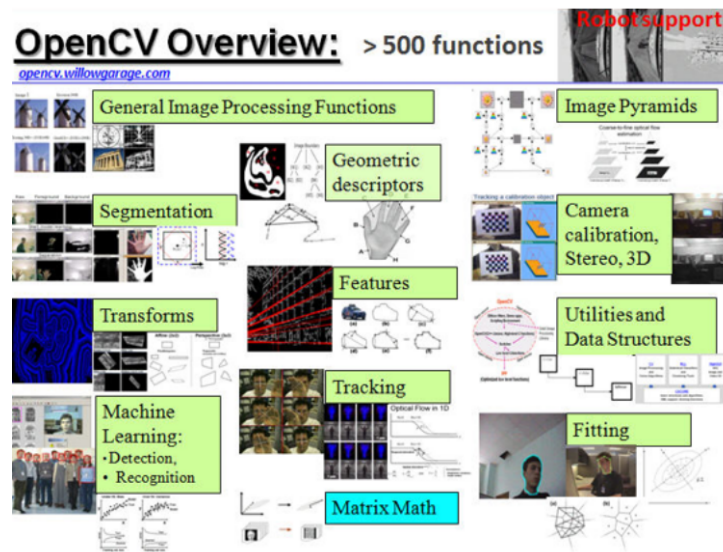


Figura 3.13: Funciones de OpenCV

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. Interfaces gráficas con 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 de la aplicación de control visual basado en *deeplearning* creado en este proyecto está escrita usando PyQt. Las clases de PyQt5 se dividen en ciertos módulos, tales como QtCore, QtGui, QtWidgets, QtXml, QSql, etc.

3.4.7. Middleware neuronal Keras

Keras ¹² es un *middleware* de alto nivel para redes neuronales, escrito en Python y capaz de correr sobre las plataformas TensorFlow, CNTK, o Theano. 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 entorno 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

¹²<https://keras.io/>

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 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.

3.4.7.1. Modelos en Keras

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 en Keras

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 una. Existen diferentes tipos de capas en Keras, pero solamente veremos las empleadas en el proyecto.

- *Convolutional layer*: Es la capa principal de una red CNN, como vimos en la Sección 1.3.3, 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 rellenará 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 1.3.3, 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*, *GlobalMaxPooling2D*, 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:

$$g(x) = \max(0, x) \tag{3.5}$$

En la Figura 3.14 se muestra la función de activación *ReLU* en el intervalo $[-10, 10]$.

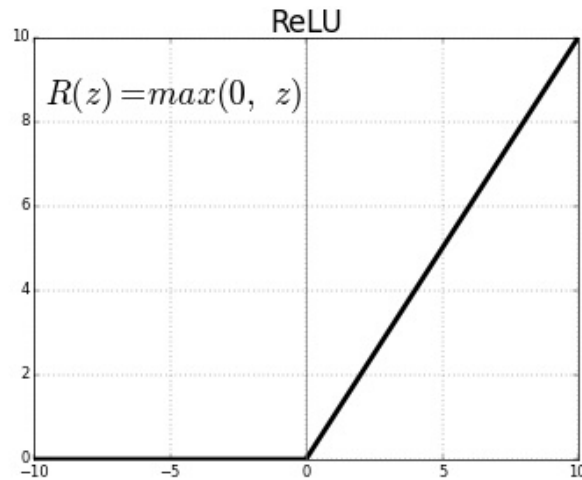


Figura 3.14: Función de activación *ReLU*

- **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 en la ecuación (3.6), 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.
- *ConvLSTM2D layer*: Es similar a una capa Long Short-Term Memory (LSTM), pero las transformaciones de entrada y las transformaciones recurrentes son convolucionales. Esta capa tiene algunos argumentos esenciales:
 - *filters*: Número de filtros.
 - *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.
 - *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 rellenará con ceros para dar lugar a una salida que conserve las dimensiones de la entrada. El valor por defecto es *valid*.
 - *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* en Keras

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 sobrescriba 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

Hierarchical 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 ¹³ para Python.

¹³www.h5py.org

Capítulo 4

Infraestructura desarrollada

El propósito de este proyecto es que un coche autónomo sea capaz de conducir en diferentes circuitos mediante distintas redes neuronales que aprenden control visual. El coche dispondrá de una cámara que le proporciona información de su entorno, permitiéndole tomar decisiones.

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 describirá en la Sección 4.3. Con el fin de generar este conjunto de datos se ha programado un piloto autónomo basado en visión (Sección 4.2) que es capaz de dar vueltas alrededor del circuito sin intervención humana.

Además se definirán los circuitos de carreras empleados tanto para el entrenamiento de las redes como para el *test*.

4.1. Circuitos de carreras en Gazebo

El objetivo de este proyecto es que un Fórmula 1 sea capaz de conducir de forma autónoma por un circuito de carreras, por lo que tendremos que crear diferentes entornos (circuitos) donde se moverá. Para facilitar el algoritmo del piloto autónomo programado y del piloto autónomo basado en redes neuronales, los circuitos tienen una línea roja pintada en el suelo.

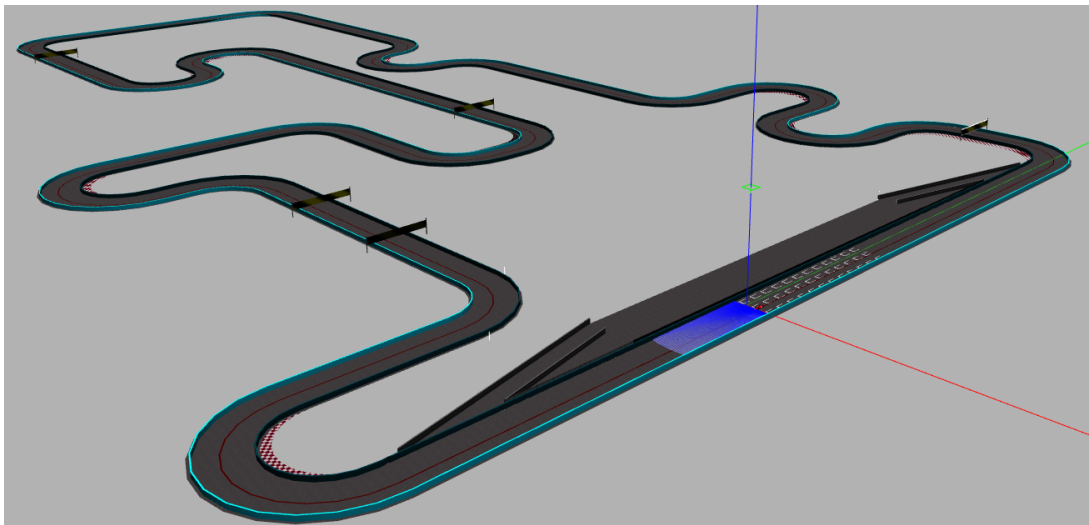


Figura 4.1: Modelo pistaSimple

Los modelos de circuitos fueron creados con una herramienta de modelado 3D (Blender, SketchUp, etc). La mayoría de los circuitos son de grandes dimensiones. En estos circuitos no veremos muchas gradas alrededor, ni público ni otros elementos habituales de los circuitos reales, sino que se ha simplificado para que sea rápida 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 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 modelo se puede observar en la Figura 4.1.

El segundo modelo de circuito se denomina *monacoLine* y simula el circuito de carreras 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 4.2.

El tercer modelo simula el circuito de carretas Nürburgring (Alemania) acortado, lla-

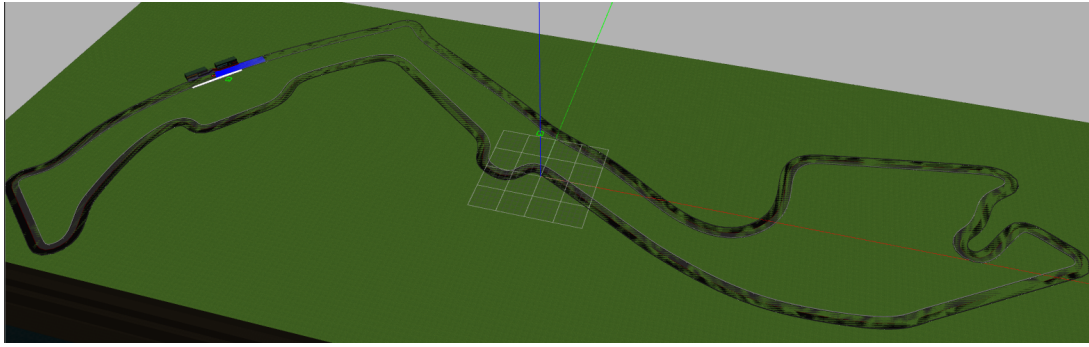


Figura 4.2: Modelo monacoLine

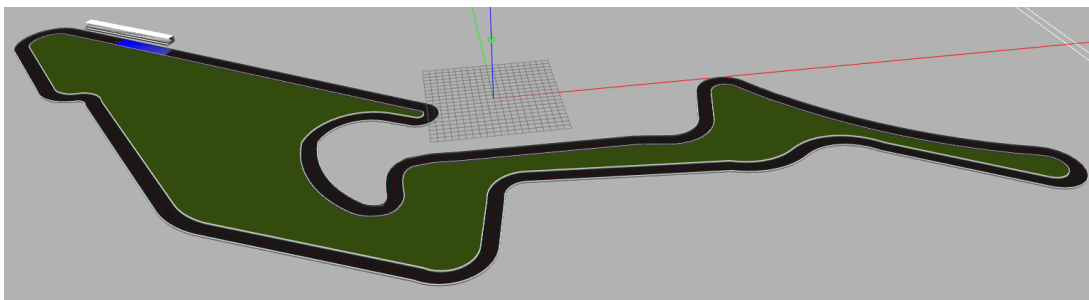


Figura 4.3: Modelo nurburgrinLine

mado *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 4.3.

El cuarto modelo de circuito, llamado *curveGP*, simula un circuito que no tiene ninguna recta, solamente tiene curvas, tanto curvas leves como abruptas. Este circuito es de gran tamaño, y no tiene ningún elemento de adorno. Se puede ver en la Figura 4.4.

El quinto modelo de circuito se denomina *pista_simple*. Este circuito es el más corto de todos, no tiene ningún elemento de adorno. Se puede ver en la Figura 4.5.

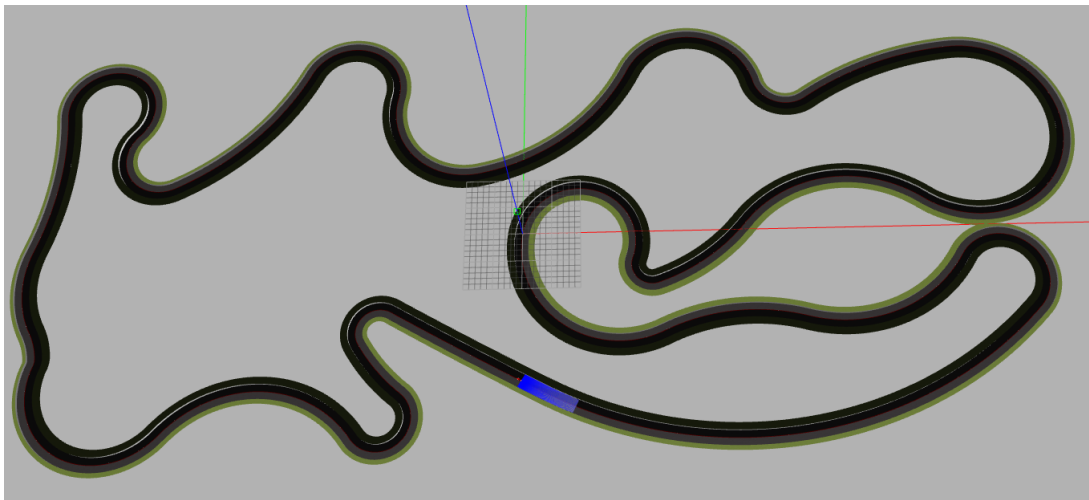


Figura 4.4: Modelo curveGP

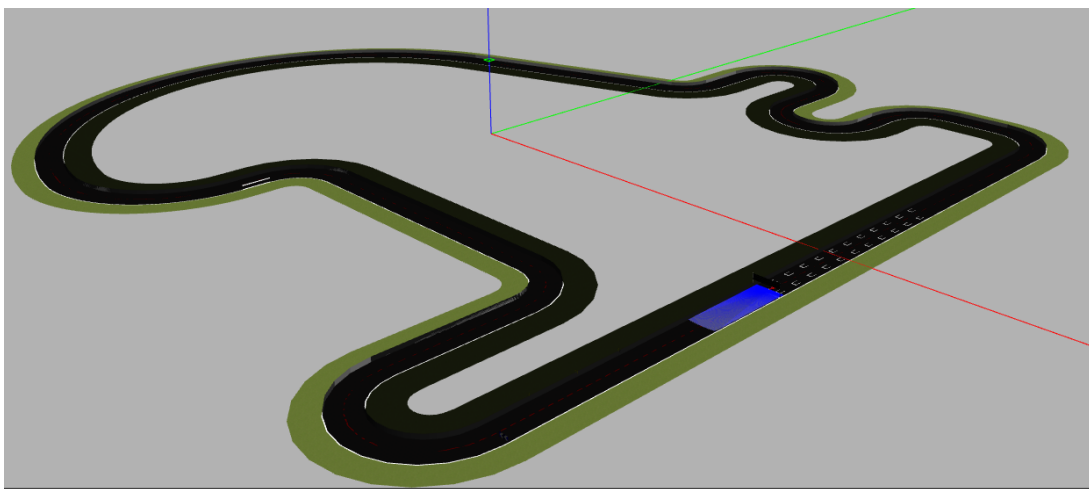


Figura 4.5: Modelo pista_simple

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 la posición del modelo del coche también, 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>
    <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), 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"/>
  </include>

```

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

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*.

4.2. Piloto autónomo programado

Para entrenar las redes neuronales es necesario disponer de una serie de datos supervisados de conducción autónoma. Con el fin de grabar este conjunto de datos se ha programado un piloto autónomo basado en visión que es capaz de dar vueltas alrededor del circuito de forma autónoma y generar ese conjunto de datos supervisados.

El piloto autónomo creado es 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 programado tendrá una parte perceptiva y una parte de control. En la parte perceptiva el vehículo extrae 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 implementa un control reactivo PD, que es capaz de corregir la velocidad de giro para mantener al coche por encima de la línea.

Lo primero que hace el piloto es acceder a las imágenes de la cámara. Es conveniente detectar la línea roja que está en el centro de la carretera, es decir, será necesario realizar una umbralización de esta línea. Para ello, primero transformamos las imágenes al espacio

de color HSV, ya que HSV es más robusto que RGB ante cambios de iluminación. Después, 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.

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 4.6. Para solventar este problema se ha aplicado un cierre morfológico (primero hace una dilatación y después una erosión). Esta operación hace que los píxeles que aparecían en negro en la línea filtrada ahora pertenezcan a la línea filtrada, es decir, que aparezcan en blanco. Se puede observar en la Figura 4.7 cómo después de aplicar el cierre morfológico no quedan huecos en la parte segmentada al filtrar la línea roja.

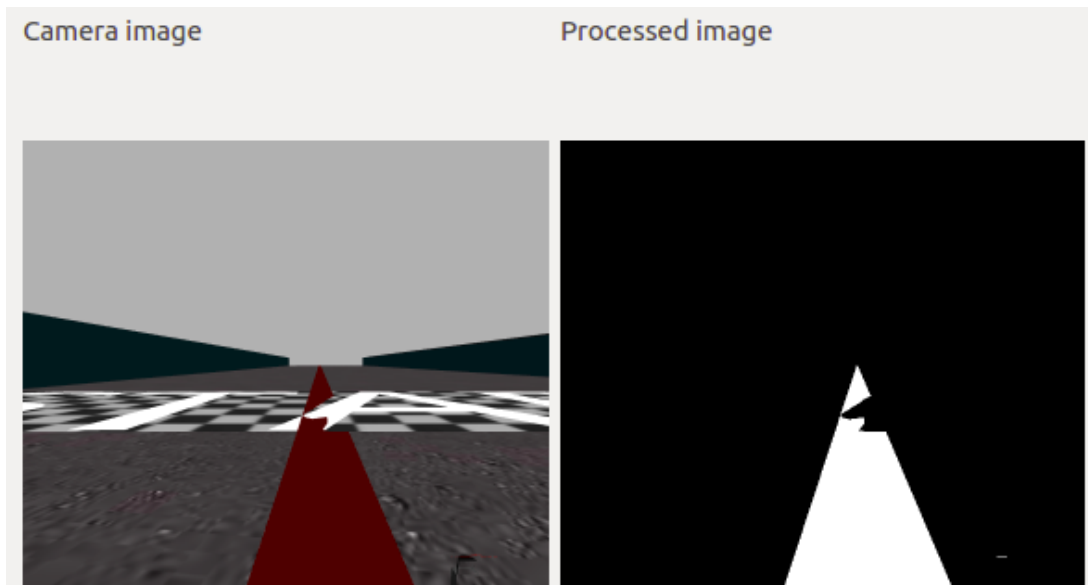


Figura 4.6: Filtrado de color

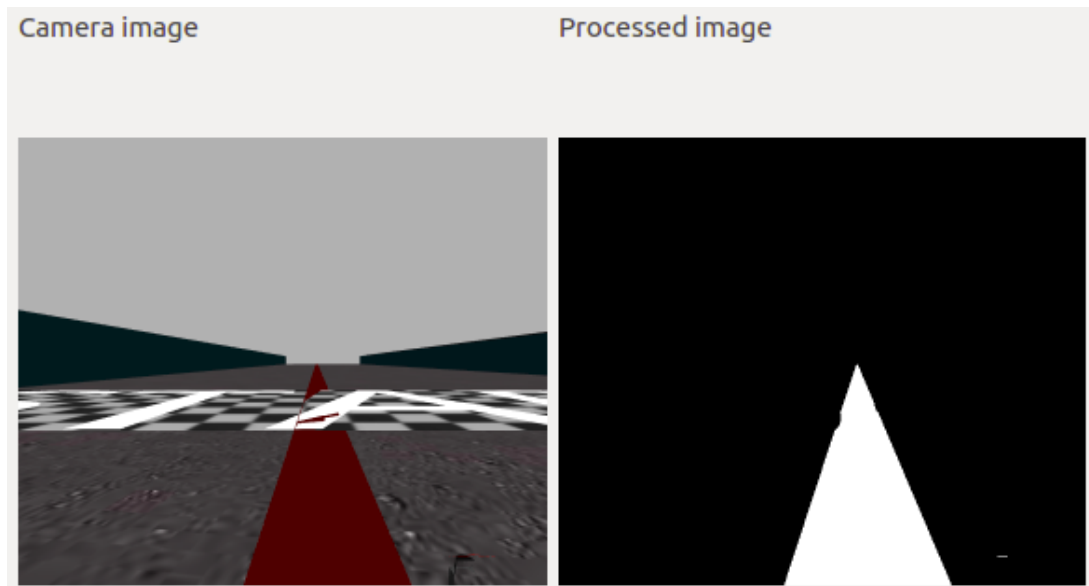


Figura 4.7: Filtrado de color con cierre

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, 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 analizemos 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 $y1 = 260$, $y2 = 310$, e $y3 = 350$. 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 $y1$ (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 y_2 (310) e y_3 (350) es posible que se pierdan al llegar a una curva grande. Si no encontramos la línea roja en la posición y_1 es debido a que nos hemos salido de la línea roja. En esta situación el coche deberá ir hacia atrás y girar hacia el centro 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.

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 y_1 (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, 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 es si el centro situado en la fila situada más abajo ($y_3 = 350$) 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 (y_1) y el centro situado en la fila de en medio (y_2), se darán diferentes situaciones en las que se ajustará un PD diferente.

En el piloto programado 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:

$$\text{Correccion} = k_p \text{ error} + k_d (\text{error} - \text{errorAnterior})$$

Tabla 4.1: Resultados del Pilotaje autónomo explícito

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

Los valores de las constantes k_p y k_d 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 constantes 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 y_3 (350) y de y_1 (260). Lo que se hace exactamente es calcular la recta que pasa por el centro de y_1 e y_3 , y después se mira la posición de x que se encuentra en esta recta para la fila y_2 (310). De este modo conociendo el centro de la fila y_2 y el punto x que se encuentra en la recta calculada, podemos saber si estamos en curva o recta.

El piloto programado es eficaz y consigue completar el recorrido en todos los circuitos creados. Se ha ejecutado en los diferentes circuitos, tanto en sentido horario como en sentido anti-horario. Se han 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.

4.3. Creación de conjunto de datos para entrenamiento neuronal

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” [43], 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.

Ha sido necesario crear un conjunto de datos para conducción autónoma en el simulador Gazebo. Esta base de datos se ha creado a partir del piloto autónomo programado que se ha explicado en la Sección 4.2. El conjunto de datos creado es un *dataset* “case-ro” donde se han almacenado las imágenes de la cámara en cada instante, así como los datos de velocidad generados por el piloto programado. Los datos de velocidades se han guardado en un archivo *.json*. En este fichero se han guardado diferentes datos adicionales 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 autónomo programado que conducía de forma autónoma únicamente por el circuito *pistaSimple* en un mismo sentido. Este conjunto tenía un problema y es que el vehículo únicamente 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 se ha creado una gráfica de estadísticas (Figura 4.8) del conjunto de entrenamiento (círculos rojos) contra los datos de una conducción fallida en *test* (cruces azules). En cada imagen se han analizado dos filas 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.

Se puede observar que hay bastantes casos conocidos para el coche, por ejemplo situaciones donde se encuentra en recta el coche. Pero también 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 en las que no sabe qué hacer.

La gráfica 4.8 nos hace pensar que es necesario entrenar el modelo con un conjunto de

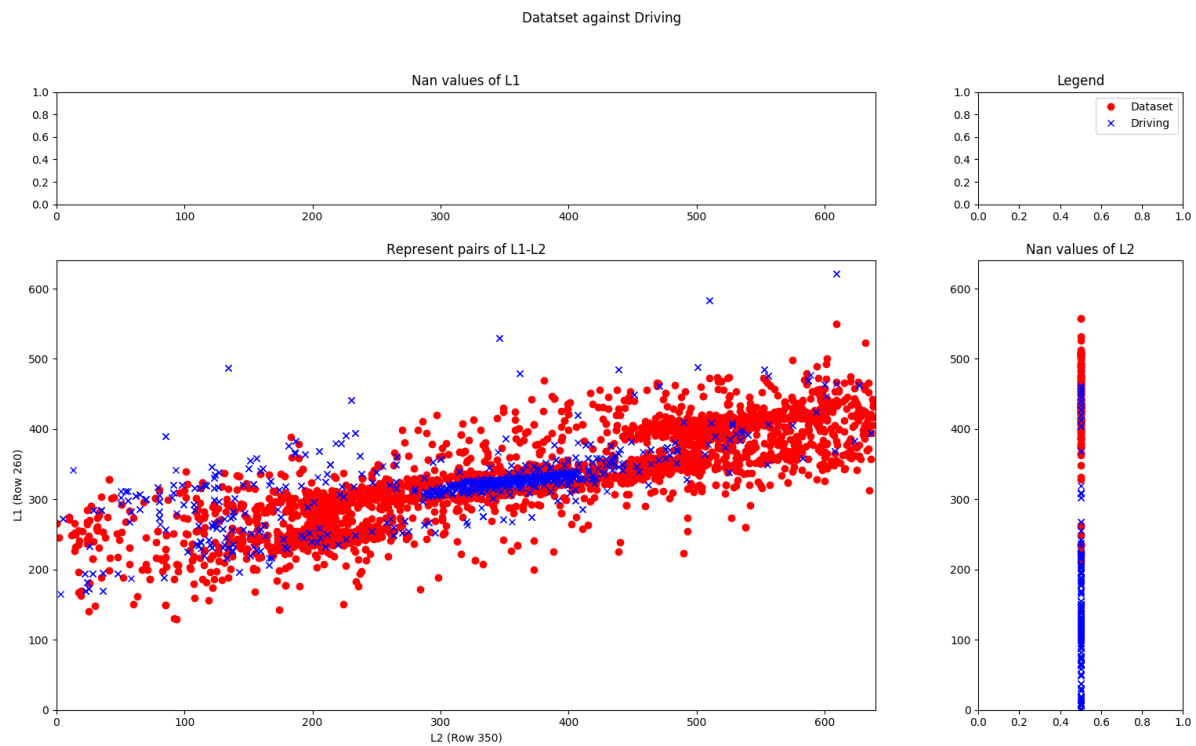


Figura 4.8: Representación pares L1-L2 (*Dataset1* contra conducción)

imágenes más representativo de las situaciones en las que el coche se puede encontrar. Por este motivo se creó un nuevo conjunto de datos. El nuevo conjunto de datos (denominado *Dataset*) se ha grabado en tres circuitos diferentes: *pistaSimple*, *monacoLine*, *nurburgrin-Line*. El piloto programado ha dado varias vueltas a los 3 circuitos en ambos sentidos para grabar este conjunto. Este *dataset* consta de 17341 pares de imágenes-acciones. Se han dividido los datos obteniendo 9710 pares de entrenamiento, 2428 pares de validación, y 5203 pares para *test*.

La misma gráfica estadística que se ha empleado para evaluar el dataset inicial “fallido” se ha utilizado con el fin de evaluar el nuevo. En la Figura 4.9 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.

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 también un *dataset complementario* al anterior de un circuito que solamente posee curvas (*curveGP*). Este conjunto extendido

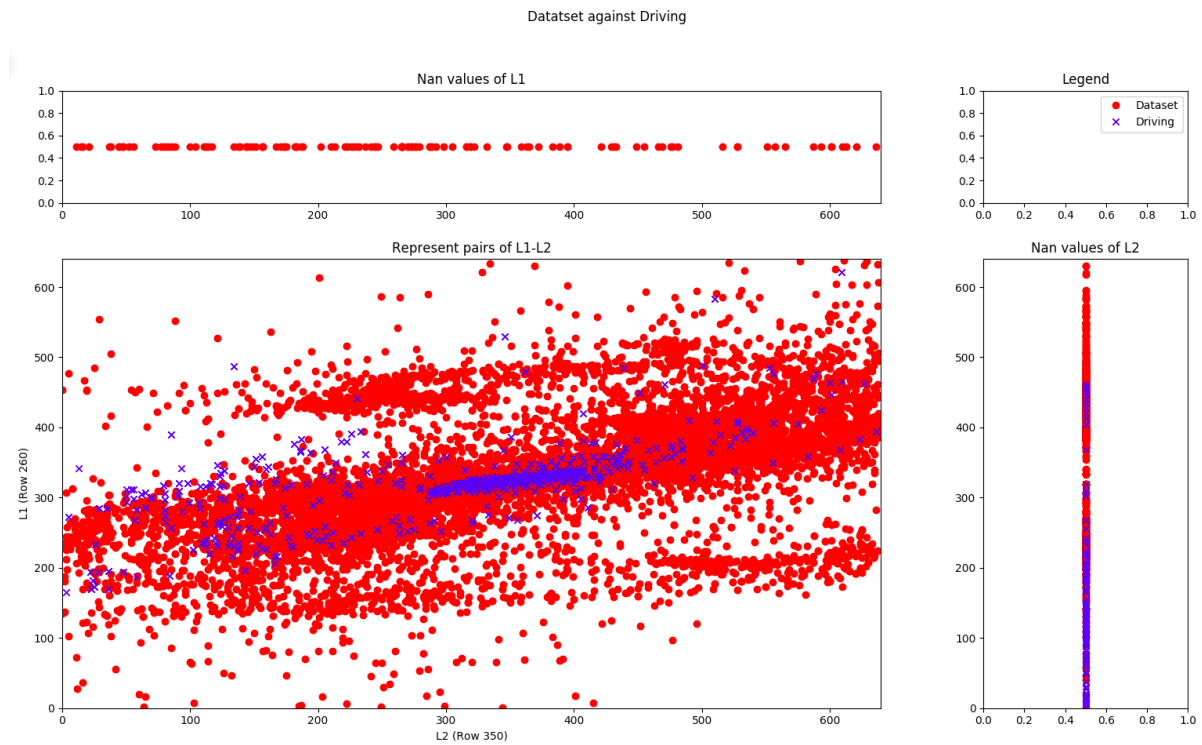


Figura 4.9: Representación pares L1-L2 (nuevo *Dataset* contra conducción)

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* normal. En la Figura 4.10 se puede ver la gráfica estadística de esta extensión del conjunto de datos (*Dataset_Curves*).

La gráfica 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 4.11 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 4.12 se representa el análisis de los pares L1-L2 para 5 clases de v (“negative”, “slow”, “moderate”, “fast”, “very_fast”). En estas imágenes se observa cómo los ejemplos quedan más o menos agrupados por clases entorno a un rango de valores de L1 y L2.

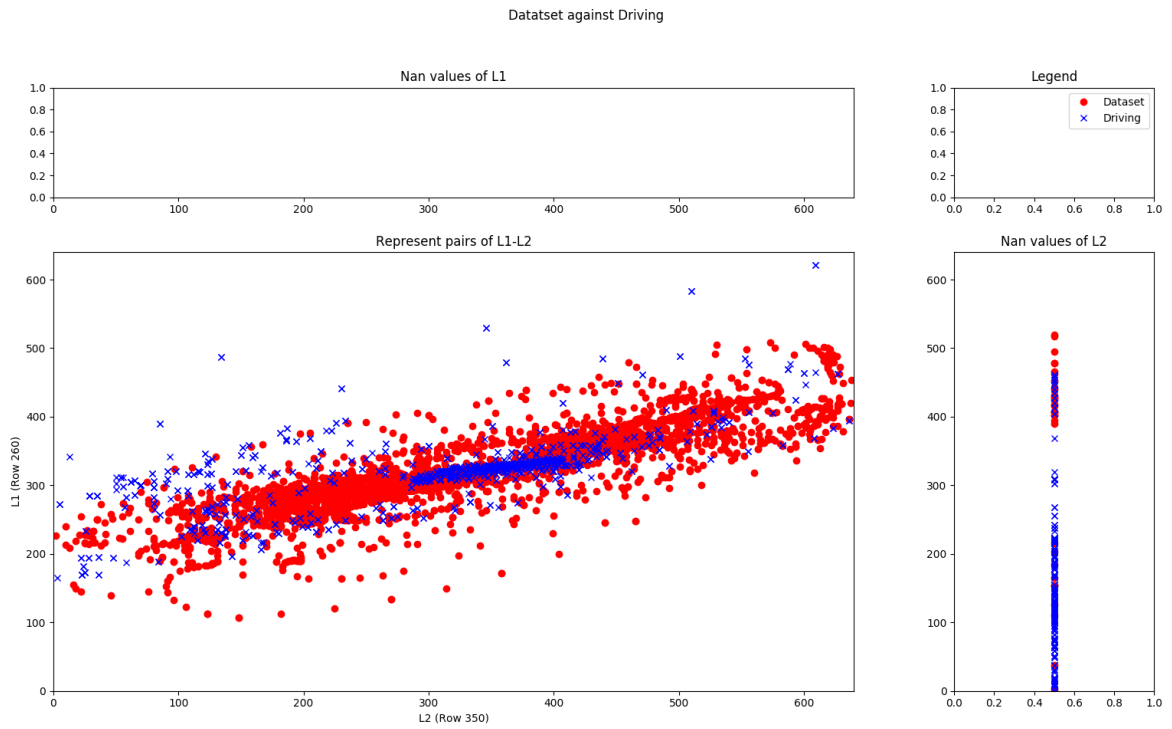


Figura 4.10: Representación pares L1-L2 (*Dataset_Curves* contra conducción)

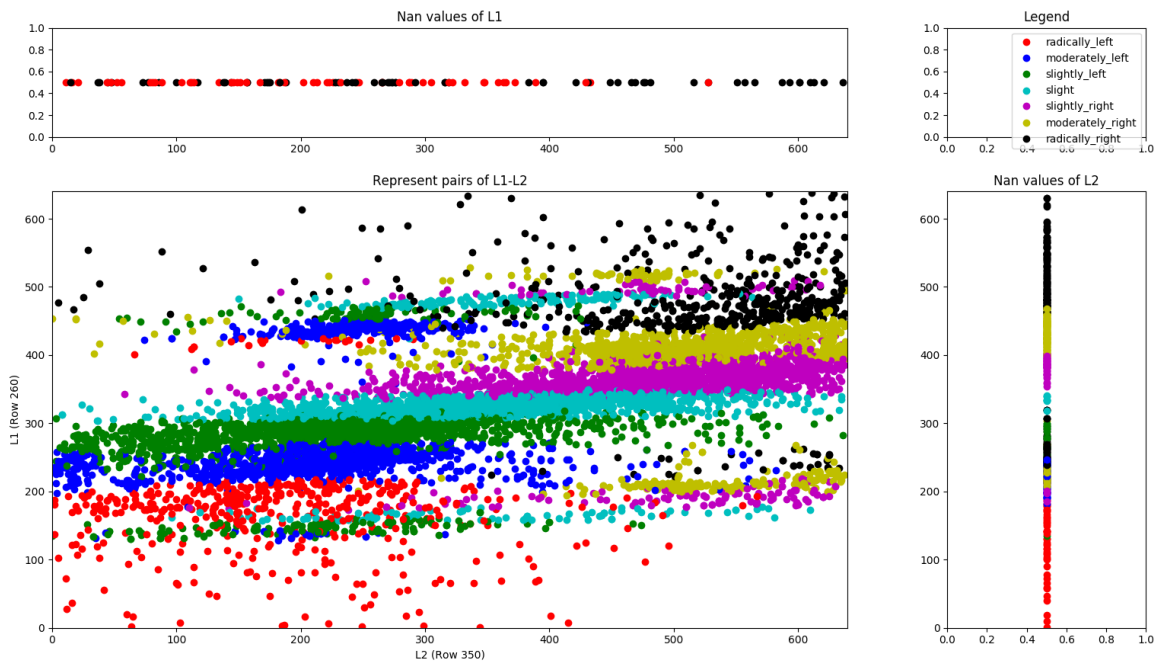


Figura 4.11: Análisis de pares L1-L2 (*Dataset*) para w

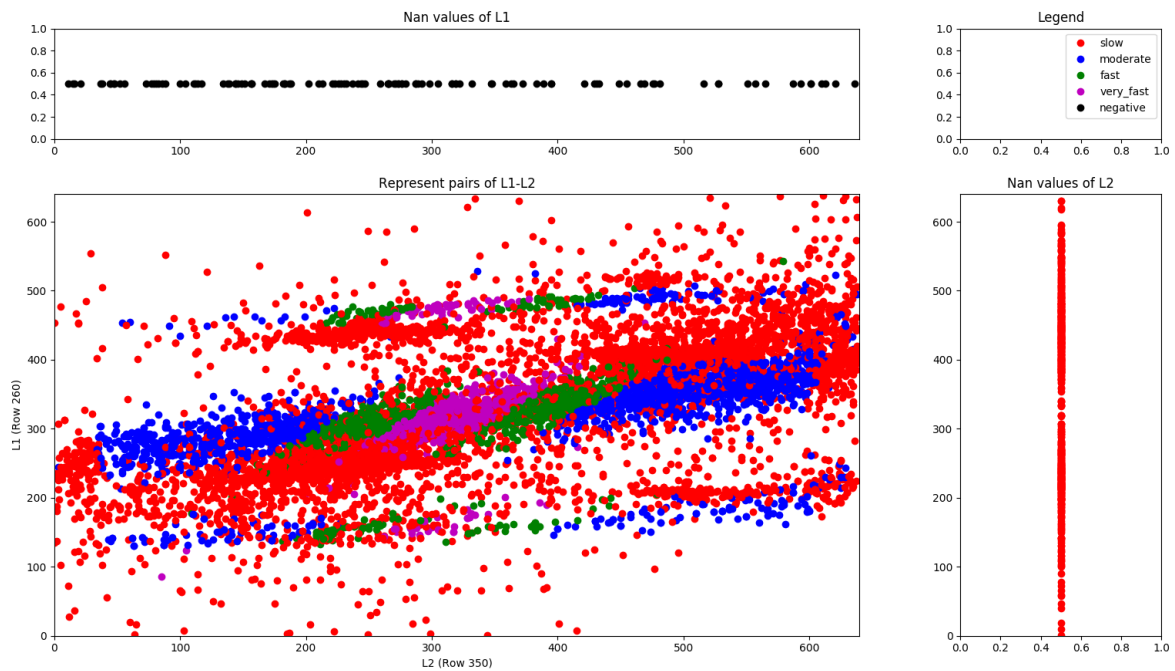


Figura 4.12: Análisis de pares L1-L2 (*Dataset*) para v

En el entrenamiento de las redes neuronales convolucionales se ha empleado el conjunto 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 el circuito *pista_simple* en ambos sentidos que las redes no han empleado para entrenamiento.

4.4. Piloto autónomo basado en redes neuronales

Se ha programado una aplicación de control visual con la infraestructura necesaria 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 enviar a los motores las órdenes estimadas por las redes (bien de clasificación o de regresión). Lo deja todo atado para que el usuario sólo

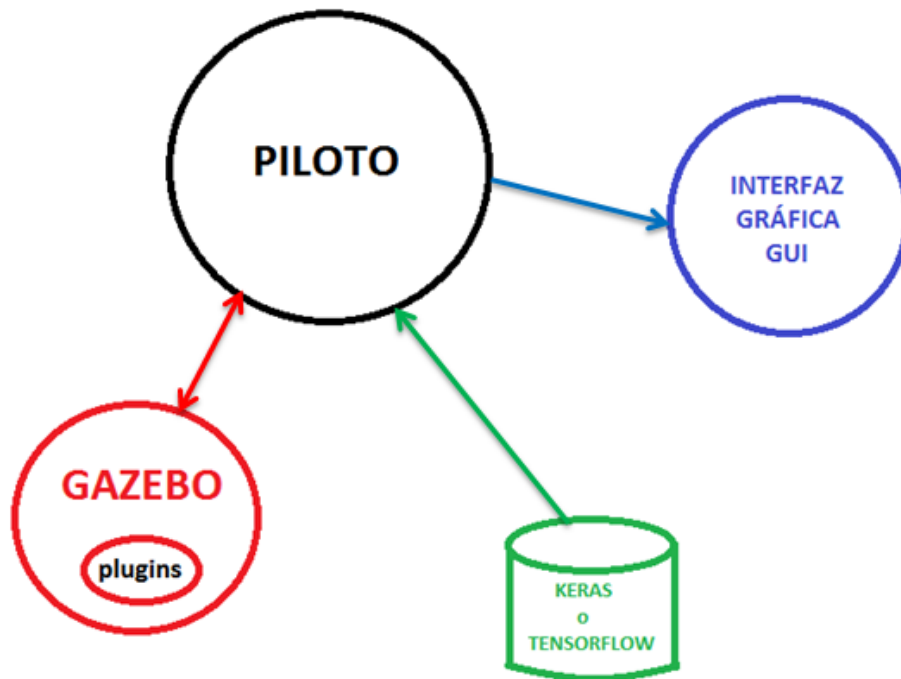


Figura 4.13: Estructura de la aplicación de control visual basada en redes neuronales

tenga que incluir su red y retoque un fichero donde se proporciona al vehículo las órdenes de velocidad predichas por la red. En la Figura 4.13 se puede observar la estructura que tiene esta aplicación.

Esta aplicación 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.

En el archivo YML ¹ 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

¹YAML Ain't Markup Language format

Podemos ver que los motores emplean el Puerto 9999, mientras que la cámara emplea el Puerto 8995. Además, podemos observar 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 *.yaml*, 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 un fichero podremos emplear las velocidades predichas por la red e indicar las órdenes de velocidad al vehículo.

Se ha dividido la aplicación piloto en diferentes partes e 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 importante, ya que tenemos que mostrar la imagen que ve el coche en tiempo real. El intervalo de tiempo debe ser pequeño, se actualizará cada 50 ms.
- Hilo de la red neuronal: Es el encargado de inferir valores a partir de 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 llamada. Este

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

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.

4.4.1. Interfaz gráfica

La interfaz 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 4.14) 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.

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 muestran 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. La velocidad angular del vehículo se controla moviendo el *joystick* en sentido horizontal.

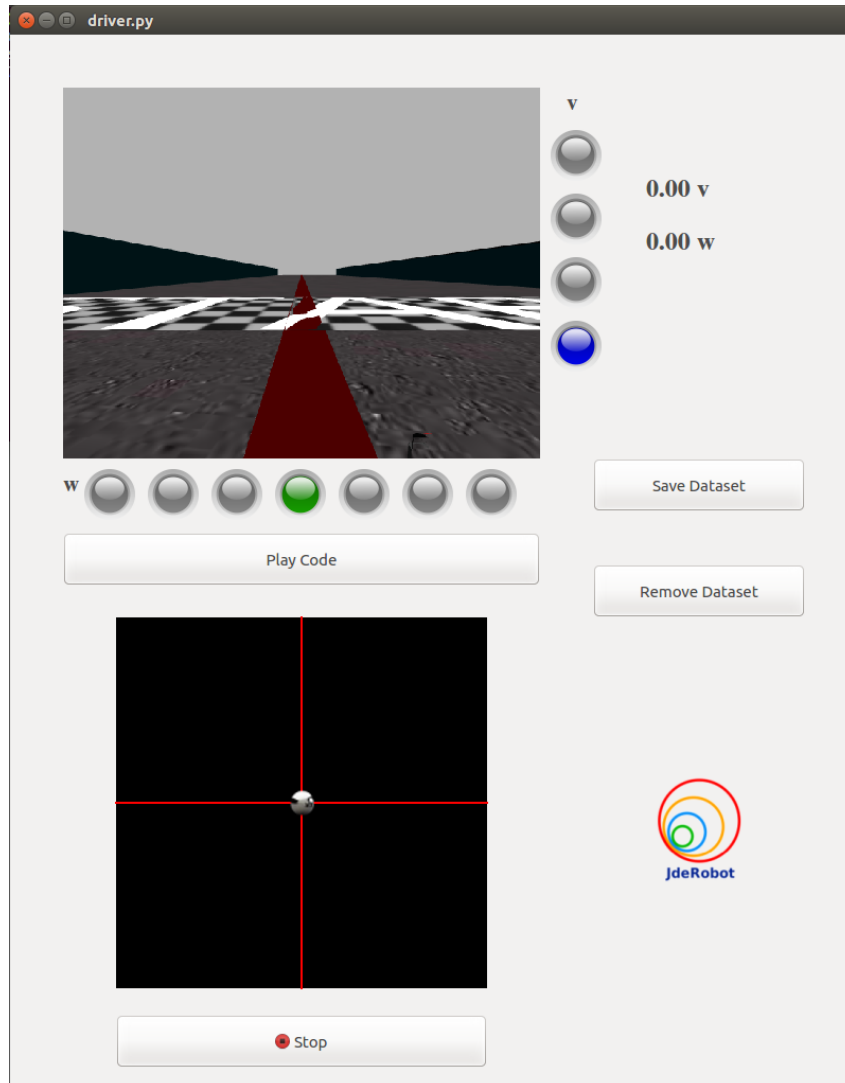


Figura 4.14: Interfaz gráfica (GUI)

En la aplicación gráfica también hay cuatro botones importantes. 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 donde le damos órdenes a los motores del coche. 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 4.3, 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.

Capítulo 5

Redes de clasificación

Las redes neuronales de extremo a extremo se han empleado ampliamente en problemas de clasificación, es decir, en problemas en los que el objetivo es determinar la clase a la que pertenece un elemento. En este proyecto se emplea este tipo de red con el fin de calcular las acciones de dirección y velocidad de tracción de un vehículo.

Con esa finalidad se han cuantificado tanto las medidas de velocidad lineal como las medidas del ángulo de dirección en valores discretos, que representan las etiquetas de las clases. Se ha estudiado la influencia de las especificaciones de cuantización de clase en la conducción del vehículo. Las especificaciones son tanto el número de clases, como el rango de valores de estas clases.

Las diferentes clasificaciones empleadas en función de los valores de los ángulos y las velocidades lineales se pueden ver en la creación del dataset (Sección 4.3), donde se especifican los rangos de valores que toma cada clase en cada clasificación. En los experimentos realizados se verán las combinaciones empleadas de cierto número de clases para la velocidad de tracción (v) y para la velocidad de rotación (w). Se han probado combinaciones de 4 clases de v con 7 u 9 clases de w (con mayor o menor rango de w), y la combinación de 5 clases de v con 7 clases de w .

En la conducción, la red predecirá una determinada clase tanto para v como para w , pero esta clase se debe traducir a órdenes de velocidad que se envía al coche. Debido a que las clases que predecimos se encuentran en un determinado rango lo que se hace es mandar como orden de velocidad a los motores del coche el resultado de la media

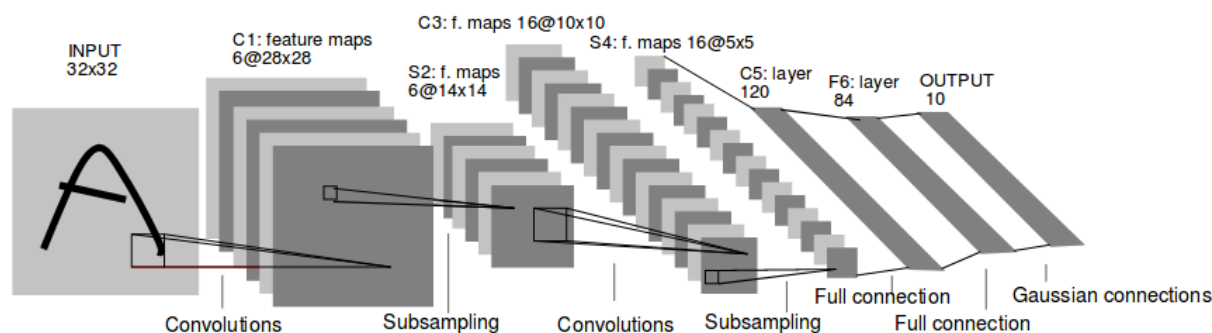


Figura 5.1: Arquitectura LeNet-5.

del mínimo y el máximo valor de ese rango, o un valor próximo a dicha media ajustado experimentalmente. Si por ejemplo, la red predice la clase “moderately_right” de w , que para una clasificación de 7 clases tiene un rango de valores entre -0.5 y -1, entonces le indicaremos al vehículo que tome una velocidad de rotación igual a -0.75.

5.1. Arquitecturas de red

En esta sección se explicarán las arquitecturas de red de clasificación que se han estudiado y parte de las experiencias recabadas con ellas. Las redes neuronales empleadas para clasificación son todas CNN.

5.1.1. LeNet-5

En el momento inicial se empleó la arquitectura de red LeNet-5 [44] (Figura 5.1), propuesta por Yann LeCun, Leon Bottou, Yosuha Bengio y Patrick Haffner para el reconocimiento de caracteres.

LeNet-5 es una red muy simple que consta únicamente de 7 capas. Tres de estas capas son capas convolucionales (C1, C3 y C5), las cuales emplean un filtro de tamaño 5×5 y un *stride* de 1. Entre las capas convolucionales se aplica una capa de submuestreo (*pooling*), es decir, en total dos capas de *pooling* (S2 y S4) con un tamaño de filtro de 2×2 . La capa 6 es una *fully-connected* (F6), seguida por la capa de salida.

Al comienzo del proyecto se empleó LeNet-5 para entrenar las redes de clasificación, pero pronto se vió que era un modelo demasiado simple y no era suficiente para que el coche aprendiera a conducir de forma autónoma.

5.1.2. SmallerVGGNet

La arquitectura de red finalmente empleada [45] (Figura 5.2) es una versión reducida del modelo VGGNet, que fue propuesto por Simonyan y Zisserman en el artículo “Very Deep Convolutional Networks for Large Scale Image Recognition” [46]. La arquitectura SmallerVGGNet empleada está diseñada para problemas multiclase.

En esta red inicialmente tenemos un bloque compuesto por una capa convolucional de 32 filtros de tamaño 3x3 y activación *ReLU*, seguida de una capa de normalización del lote (*BatchNormalization*), una capa de sumuestreo (*pooling*), y una capa de *dropout* del 25%. A continuación, hay dos bloques compuestos por una capa convolucional seguida de una capa *BatchNormalization*, una capa convolucional, una capa *BatchNormalization*, una capa de *pooling*, y una capa de *dropout* del 25%. En el primero de estos dos bloques en las capas convolucionales se emplean 64 filtros de tamaño 3x3 y activación *ReLU*; mientras que en el segundo bloque en las capas convolucionales se usan 128 filtros de tamaño 3x3 y activación *ReLU*. Al final de la red tenemos un bloque de capas *Fully connected*, donde en la última de estas capas se utiliza una función de activación sigmoide para la clasificación de múltiples etiquetas.

Esta arquitectura es bastante más sofisticada que la arquitectura LeNet-5, permitiendo de esta forma a la red aprender situaciones más complejas. Este modelo se ha empleado en los diferentes experimentos realizados que veremos en la siguiente sección.

5.2. Experimentos

Se han realizado numerosas pruebas y experimentos con estas redes para caracterizar la influencia de los datos de entrenamiento, el uso de imágenes completas o recortadas, el número de clases y su combinación, etc.

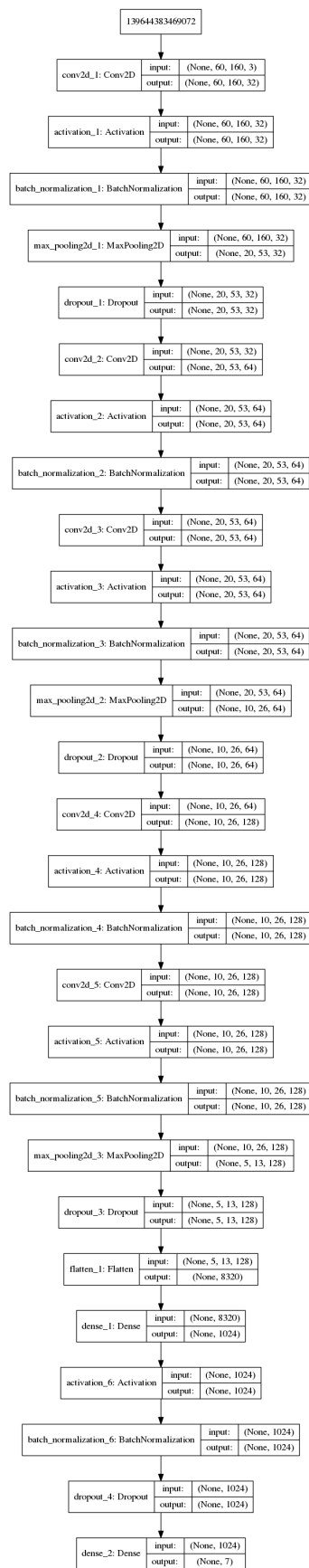


Figura 5.2: Arquitectura SmallerVGGNet.

5.2.1. Métricas de evaluación

Con el fin de evaluar objetivamente cada red entrenada se calculan ciertas métricas de evaluación en el conjunto de *test*. En las redes de clasificación las métricas neuronales que se han evaluado son: *Accuracy*, *Accuracy Top 2*, *Precision*, *Recall*, y *F1-Score*.

En los problemas de clasificación, *Accuracy* es el número de predicciones correctas realizadas por el modelo sobre todo tipo de predicciones realizadas. Se puede calcular con la siguiente fórmula:

$$\frac{1}{N} \sum_{n=1}^N \delta\{\hat{p}_n = p_n\}$$

Donde \hat{p}_n es la etiqueta que la red predice en la clasificación, p_n son las etiquetas reales, y N es el número de muestras. Por último, la función $\delta\{x\}$ se define como:

$$\delta\{\text{condición}\} = \begin{cases} 1 & \text{si condición} \\ 0 & \text{resto} \end{cases}$$

El término *Accuracy Top 2* lo calculamos como la métrica *Accuracy*, pero en este caso si la clase predicha es una de las clases adyacentes o la clase real se considera que es correcta la predicción.

La métrica *Precision* se define como la relación entre los positivos verdaderos (TP) y el número total de positivos predichos por un modelo (TP y FP). En la clasificación multiclase se debe tener en cuenta que existen varias clases y se emplea la siguiente fórmula:

$$Precision = \frac{TP_X}{TP_X + FP_X}$$

Donde TP_X es el número de verdaderos positivos para la clase X, es decir, el número de aciertos correspondiente para dicha clase. Mientras que FP_X es el número de falsos positivos para la clase X, es decir, el número de veces que se ha predicho dicha clase sin ser así.

El parámetro *Recall* es la relación entre los positivos verdaderos (TP) y el número total de positivos que se producen. En los problemas multiclase se puede definir como:

$$Recall = \frac{TP_X}{TP_X + FN_X}$$

Donde FN_X son los falsos negativos para la clase X, es decir, el número de veces que se predijo erróneamente otra clase habiéndose producido X.

La métrica *F1-Score* es una puntuación única que representa tanto a Precision (P) como a Recall (R). Se puede calcular mediante la siguiente fórmula:

$$F1 - Score = \frac{2 * Precision * Recall}{Precision + Recall}$$

Adicionalmente, las métricas operativas más importantes que usaremos para evaluar el rendimiento de las redes son el porcentaje del circuito recorrido por el piloto autónomo basado en las redes, así como el tiempo por vuelta al circuito. Estas métricas nos darán una idea real del funcionamiento de las redes en ejecución.

Estas métricas calculadas en el conjunto de *test* nos dan una idea de cómo de bien ha ido el entrenamiento. Cada una de las medidas se calculará para cada una de las redes entrenadas para v y w.

En la Tabla 5.1 se pueden ver los resultados de las métricas promedio para las redes de velocidad de rotación (w) con imágenes recortadas. En este caso vemos tres redes de 7 clases de w en función de cómo entrenamos según los datos, y tres redes con 9 clases para w.

Tabla 5.1: Métricas de test de redes de clasificación (w, imagen recortada)

Red	Accuracy	Accuracy top 2	Precision	Recall	F1-Score
7w sesgada	94 %	99 %	95 %	95 %	95 %
7w balanceada	93 %	99 %	94 %	94 %	94 %
7w desbalanceada	95 %	99 %	95 %	95 %	95 %
9w sesgada	93 %	99 %	94 %	94 %	94 %
9w balanceada	93 %	99 %	94 %	94 %	94 %
9w desbalanceada	95 %	99 %	96 %	96 %	96 %

En la Tabla 5.2 se pueden observar los resultados de las métricas para las redes de velocidad de tracción (v) con imágenes recortadas. En esta tabla tenemos 3 redes para 4 clases de v y 3 redes para 5 clases de v.

Tabla 5.2: Métricas de test de redes de clasificación (v, imagen recortada)

Red	Accuracy	Accuracy top 2	Precision	Recall	F1-Score
4v sesgada	95 %	98 %	95 %	95 %	95 %
4v balanceada	92 %	96 %	94 %	93 %	93 %
4v desbalanceada	95 %	97 %	95 %	95 %	95 %
5v sesgada	93 %	96 %	95 %	93 %	94 %
5v balanceada	92 %	95 %	93 %	92 %	93 %
5v desbalanceada	93 %	96 %	95 %	94 %	94 %

En estas tablas se pueden ver los resultados en el conjunto de prueba en la mayoría de casos superan el 90 %, pero esto no implica que la conducción vaya a tener éxito, ya que el resultado de las métricas es un promedio de todas las clases. Por ejemplo en una clase nos puede dar un resultado de *Accuracy* del 100 % mientras que en otra clase nos da un resultado mucho menor, pero al hacer un promedio nos hace intuir que los resultados serán buenos. Aún así dan una idea de cómo ajustar los parámetros durante el entrenamiento.

En las pruebas realizadas se han obtenido los mejores resultados operativos en cuanto a porcentaje recorrido y tiempo por vuelta de circuito empleando la red sesgada con 5

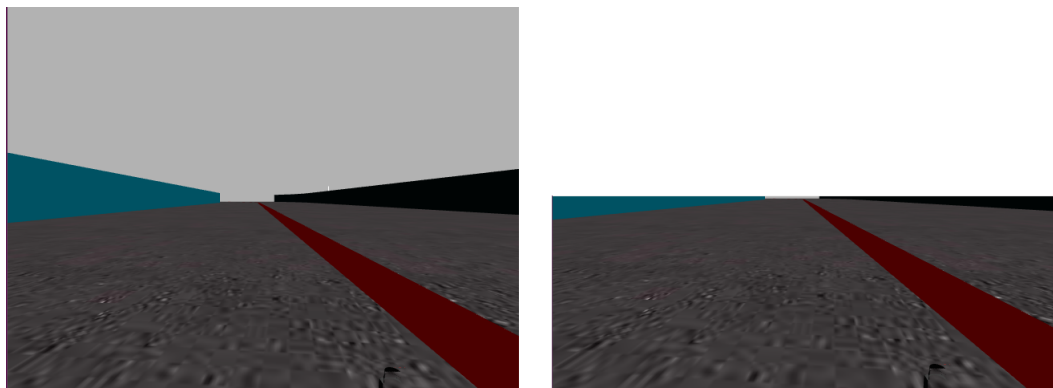


Figura 5.3: Imagen completa (izquierda) e imagen recortada (derecha)

clases de v y 7 de w . Sin embargo, como vemos las Tablas 5.1 y 5.2, hay alguna otra red que obtiene mejores resultados en las métricas de evaluación en cuanto a accuracy, recall, precision y F1-Score se refiere. Se comprueba con ello que aunque obtengamos buenos resultados en las métricas neuronales para cada una de las clases, es posible que en algunos casos con estas redes el vehículo choque contra la valla, y en cambio en otros casos donde se obtengan peores resultados neuronales el vehículo será capaz de completar el circuito. Esto se debe a que cuando estamos pilotando el coche, puede ser que si predecimos mal un valor no implique mucha desviación del coche de la línea roja. Pero, sin embargo, si la red en un instante dado predice 3 o 4 valores seguidos mal, el coche se irá desviando cada vez más y no será capaz de rectificar para volver a la recta. Esto indica que el entrenamiento de redes neuronales para conducción autónoma es algo complejo y se necesita mucha experimentación antes de lograr un buen resultado.

5.2.2. Imágenes de distintas dimensiones

Las imágenes capturadas por la cámara del vehículo tienen unas dimensiones de 640 x 480 píxeles. Algunas pruebas emplean las imágenes completas (Figura 5.3) y se entrenan las redes de clasificación con las mismas en formato BGR. Antes de entrenar las redes con estas imágenes, se reducen las dimensiones de las mismas por un factor de escala de 1/4 en horizontal y 1/4 en vertical en total para aliviar la carga del entrenamiento. Por lo que las imágenes a la entrada de la red tienen unas dimensiones de (160, 120, 3).

Además, se han realizado pruebas empleando un recorte de imagen (*image cropping*),

que consiste en extraer una zona concreta de la imagen donde se considera que se almacena la parte relevante de información. La imagen recortada (Figura 5.3) contiene información acerca de la carretera, eliminando de esta forma la parte del cielo de la imagen. Esta imagen tiene unas dimensiones de 240 x 640 píxeles, aunque antes de entrenar la red se reducen las dimensiones 1/4 en horizontal y 1/4 en vertical, siendo las dimensiones de la imagen de (160, 60, 3) al entrar a la red.

Se han llevado a cabo diferentes pruebas en función de las dimensiones de las imágenes (completa y recortada). En la Tabla 5.3 se recogen los resultados de la comparativa. Esta tabla compara los resultados de la combinación de redes 5v+7w (5 clases para v y 7 para w) sesgadas (Sección 5.2.4), ya que la red 5v+7w sesgada (imagen recortada) es la que mejor resultados obtiene, siendo capaz de recorrer todos los circuitos en los dos sentidos.

Tabla 5.3: Resultados de conducción con redes de clasificación (imagen completa e imagen recortada)

Circuitos	Programado	5v+7w sesgada recortada		5v+7w sesgada completa	
	Tiempo	%	Tiempo	%	Tiempo
pistaSimple (h)	1' 35"	100 %	1' 41"	35 %	
pistaSimple (ah)	1' 33"	100 %	1' 39"	100 %	1' 49"
monacoLine (h)	1' 15"	100 %	1' 20"	100 %	1' 24"
monacoLine (ah)	1' 15"	100 %	1' 18"	100 %	1' 29"
nurburgrinLine (h)	1' 02"	100 %	1' 03"	100 %	1' 10"
nurburgrinLine (ah)	1' 02"	100 %	1' 05"	100 %	1' 07"
curveGP (h)	2' 13"	100 %	2' 06"	95 %	
curveGP (ah)	2' 09"	100 %	2' 11"	7 %	
pista_simple (h)	1' 00"	100 %	1' 02"	8 %	
pista_simple (ah)	59"	100 %	1' 03"	12 %	

Los resultados muestran que es mejor emplear una imagen de entrada recortada a una imagen completa. Esto se debe a que la red posiblemente se distraiga con la parte superior de la imagen (el cielo) y no consiga centrar su atención en la parte más importante de la imagen (la carretera).

5.2.3. Número de clases

Para pilotar al coche es necesario entrenar una red para la velocidad lineal (v) y una red para la velocidad de rotación, siendo ambas empleadas durante la conducción. El número de clases, así como el rango de valores de velocidad de cada clase influirá en el rendimiento de la conducción.

En el proyecto se han estudiado varias combinaciones de clases de v y clases de w , sabiendo que los datos de velocidad de rotación se encuentran en un rango de $(-2.9269; 3.1138)$ en rad/s y los datos de velocidad de tracción se encuentran en el rango $(-0.6; 13)$ en m/s . En un primer momento se ha estudiado la combinación de 7 clases para w y 4 para v . Como segundo experimento se han empleado 9 clases para w y 4 clases para v . Finalmente, se ha experimentado con 7 clases de velocidad de rotación y 5 clases de velocidad de tracción.

En la clasificación de 7 clases de velocidad de rotación (w), las clases se dividen según los siguientes rangos:

```
"radically_left": w >= 1
"moderately_left": 0.5 <= w < 1
"slightly_left": 0.1 <= w <= 0.5
"slight": - 0.1 < w < 0.1
"slightly_right": - 0.5 < w <= -0.1
"moderately_right": - 1 < w <= -0.5
"radically_right": w <= -1
```

En la clasificación de 9 clases de velocidad de rotación, las clases se dividen según los siguientes rangos:

```
"radically_left": w >= 2
"strongly_left": 1 <= w < 2
"moderately_left": 0.5 <= w < 1
"slightly_left": 0.1 <= w <= 0.5
"slight": - 0.1 < w < 0.1
"slightly_right": - 0.5 < w <= -0.1
```

```
"moderately_right": - 1 < w <= -0.5  
"strongly_right": -2 < w <= -1  
"radically_right": w <= -2
```

En la clasificación de 4 clases de velocidad de tracción (v), las clases se dividen en función a los siguientes rangos:

```
"slow": v <= 7  
"moderate": 7 < v <= 9  
"fast": 9 < v <= 11  
"very_fast": v > 11
```

En la clasificación de 5 clases de velocidad de tracción, las clases se dividen en función a los siguientes rangos:

```
"negative": v <= 0  
"slow": 0 < v <= 7  
"moderate": 7 < v <= 9  
"fast": 9 < v <= 11  
"very_fast": v > 11
```

Al comienzo de los experimentos se contempló la combinación del empleo de 4 clases de velocidad lineal y 7 clases de velocidad de rotación. Pronto se comprobó que las redes entrenadas con estos números de clases no eran capaces de completar todos los circuitos. Esto se debe a que dos circuitos poseen curvas abruptas, donde o bien el vehículo deberá aplicar una velocidad de rotación mayor o una velocidad de tracción menor.

Para solventar este problema se optó por realizar una combinación de 4 clases de velocidad lineal y 9 clases de velocidad de rotación. En los resultados del pilotaje basado en estas redes se observó que el vehículo tampoco era capaz de recorrer todos los circuitos en ambos sentidos. Llegamos a la conclusión de que aunque empleemos un mayor número de clases para la velocidad de rotación, no es suficiente para un buen rendimiento en la conducción.

La conclusión obtenida de los resultados anteriores es que este fallo se produce debido a que en los datos hay casos donde la velocidad es negativa, es decir, el vehículo da marcha atrás para poder girar sin chocar con la valla. Por este motivo se ha estudiado la combinación de 7 clases de velocidad de rotación y 5 clases de velocidad de tracción, donde se contemplan valores negativos.

Los resultados de las redes 4v+7w, 4v+9w, y 5v+7w, con redes sesgadas e imágenes recortadas como entrada, se pueden ver en la Tabla 5.4. Se ha empleado este tipo de redes (sesgadas) porque son las que ofrecen mejores prestaciones, como se explicará más adelante.

Tabla 5.4: Resultados de conducción con redes de clasificación modificando la combinación del número de clases (imagen recortada)

Circuitos	Programado	4v+7w sesgada		4v+9w sesgada		5v+7w sesgada	
	Tiempo	%	Tiempo	%	Tiempo	%	Tiempo
pistaSimple (h)	1' 35"	100 %	1' 38"	100 %	1' 42"	100 %	1' 41"
pistaSimple (ah)	1' 33"	100 %	1' 38"	100 %	1' 39"	100 %	1' 39"
monacoLine (h)	1' 15"	5 %		5 %		100 %	1' 20"
monacoLine (ah)	1' 15"	5 %		12 %		100 %	1' 18"
nurburgrinLine (h)	1' 02"	8 %		8 %		100 %	1' 03"
nurburgrinLine (ah)	1' 02"	90 %		80 %		100 %	1' 05"
curveGP (h)	2' 13"	100 %	2' 19"	100 %	2' 17"	100 %	2' 06"
curveGP (ah)	2' 09"	100 %	2' 12"	100 %	2' 13"	100 %	2' 11"
pista_simple (h)	1' 00"	100 %	1' 04"	100 %	1' 04"	100 %	1' 02"
pista_simple (ah)	59"	100 %	1' 04"	100 %	1' 02"	100 %	1' 03"

Se ha llegado a la conclusión de que tanto el número de clases como el rango de valores que abarca cada una de las clases tienen una gran influencia en el rendimiento del problema planteado. Se establece que con 7 clases de velocidad de rotación es suficiente para lograr una buena conducción, y que no es necesario emplear 9 clases para conseguir el objetivo. Sin embargo, en el caso de las clases de velocidad lineal, no es suficiente con emplear 4 clases, hay que añadir una quinta clase para que el coche sea capaz de tomar

la velocidad necesaria en cada caso.

5.2.4. Influencia de los datos de entrenamiento

Los resultados de las redes neuronales no solamente tienen que ver con la arquitectura de red empleada o con la forma de entrenar, sino que el conjunto de entrenamiento tiene una gran influencia sobre el mismo. En el entrenamiento se ha empleado la base de datos propia *Dataset* (Sección 4.3).

Uno de los problemas de los datos en las redes de clasificación es que normalmente no se dispone del mismo número de datos por cada clase. Este problema implica que en algunas ocasiones hay muchos datos de una determinada clase y muy pocos de otra, haciendo que la red aprenda las situaciones donde hay muchos datos y no aprenda las clases donde tenemos menos datos.

En el conjunto de datos empleado justo sucede este inconveniente, ya que hay algunas clases de las que disponemos de muchos datos (como en la clase *slight* para w) y de otras de muy pocos (como en la clase *radically_left*), desequilibrando de esta forma el aprendizaje de la red.

En este conjunto de datos disponemos de un total de 17341 pares de imágenes-actuaciones, de los cuales se emplean para entrenamiento únicamente 12138 pares. Si empleamos el conjunto de datos entero para ver de cuántos datos disponemos en función de las clases nos encontramos con:

- En la clasificación de 7 clases de velocidad de rotación (w), el número de datos por cada clase es:

```
Numero de datos de "radically_left": 825
Numero de datos de "moderately_left": 3054
Numero de datos de "slightly_left": 2882
Numero de datos de "slight": 4030
Numero de datos de "slightly_right": 2606
```

Numero de datos de "moderately_right": 2907

Numero de datos de "radically_right": 1037

- En la clasificación de 9 clases de velocidad de rotación, el número de datos por cada clase es:

Numero de datos de "radically_left": 80

Numero de datos de "strongly_left": 745

Numero de datos de "moderately_left": 2054

Numero de datos de "slightly_left": 2882

Numero de datos de "slight": 4030

Numero de datos de "slightly_right": 2606

Numero de datos de "moderately_right": 2907

Numero de datos de "strongly_right": 953

Numero de datos de "radically_right": 84

- En la clasificación de 4 clases de velocidad de tracción (v), el número de datos por cada clase es:

Numero de datos de "slow": 9885

Numero de datos de "moderate": 3251

Numero de datos de "fast": 2535

Numero de datos de "very_fast": 1670

- En la clasificación de 5 clases de velocidad de tracción, el número de datos por cada clase es:

Numero de datos de "negative": 197

Numero de datos de "slow": 9688

Numero de datos de "moderate": 3251

Numero de datos de "fast": 2535

Numero de datos de "very_fast": 1670

El conjunto de entrenamiento (12138 pares de datos) sigue más o menos las mismas proporciones que hay en los datos completos. Como se puede ver, tanto en el conjunto de datos completo como en el conjunto de entrenamiento existe un desbalanceo de los datos, lo que influirá en el entrenamiento. Por este motivo, se han realizado 3 experimentos:

- El primer experimento consiste en entrenar la red con el conjunto de entrenamiento sin ninguna modificación. A las redes neuronales entrenadas de esta forma se las llamará desbalanceadas.
- El segundo experimento consiste en crear una nueva base de datos de entrenamiento balanceada, es decir, donde exista el mismo número de ejemplos por cada clase. Para lograr este objetivo se parte de la clase con el menor número de ejemplos, y se selecciona el mismo número de datos aleatoriamente para cada una de las otras clases. Por ejemplo, en la base de datos balanceada de 4 clases de velocidad lineal tendremos 1162 ejemplos por cada clase. A las redes entrenadas con estos conjuntos las llamaremos balanceadas.
- El tercer experimento consiste en entrenar las redes con el conjunto de entrenamiento por completo, pero al entrenar se emplea el parámetro *class_weight* de Keras que nos permite dar pesos a cada clase. Este parámetro es un diccionario donde se indican las etiquetas (clases) y los pesos que le damos a cada etiqueta. De esta forma aunque dispongamos de menos datos para alguna de las clases le daremos más peso durante el entrenamiento. A las redes entrenadas de esta manera las llamaremos sesgadas.

En la Tabla 5.5 se muestran los resultados para una combinación de redes sesgadas, otra combinación de redes balanceadas y otra de redes desbalanceadas (según el entrenamiento realizado) para el caso 5v+7w (imagen recortada). En esta tabla se puede observar que el entrenamiento de las redes sesgadas es mejor que el entrenamiento de redes desbalanceadas o balanceadas.

Tabla 5.5: Resultados de conducción con redes de clasificación (estudio de la influencia de los datos de entrenamiento)

Circuitos	Programado	5v+7w sesgada		5v+7w balanceada		5v+7w desbalanceada	
	Tiempo	%	Tiempo	%	Tiempo	%	Tiempo
pistaSimple (h)	1' 35"	100 %	1' 41"	75 %		100 %	1' 42"
pistaSimple (ah)	1' 33"	100 %	1' 39"	100 %	1' 39"	100 %	1' 43"
monacoLine (h)	1' 15"	100 %	1' 20"	70 %		85 %	
monacoLine (ah)	1' 15"	100 %	1' 18"	8 %		100 %	1' 20"
nurburgrinLine (h)	1' 02"	100 %	1' 03"	100 %	1' 03"	100 %	1' 05"
nurburgrinLine (ah)	1' 02"	100 %	1' 05"	80 %		80 %	
curveGP (h)	2' 13"	100 %	2' 06"	97 %		100 %	2' 15"
curveGP (ah)	2' 09"	100 %	2' 11"	100 %	2' 05"	100 %	2' 15"
pista_simple (h)	1' 00"	100 %	1' 02"	100 %	1' 02"	100 %	1' 01"
pista_simple (ah)	59"	100 %	1' 03"	100 %	1' 03"	100 %	1' 04"

Se concluye que son muy importantes los datos de entrenamiento para conseguir una red eficaz. En los resultados se ha podido observar que el mejor entrenamiento es con las redes sesgadas, es decir, donde le damos pesos a las clases. De esta forma podemos darle más importancia a las clases pertenecientes a curvas más abruptas que a las rectas. Así nuestro vehículo aprende mejor las relaciones visuales con las clases.

Por otra parte con las redes entrenadas con el conjunto de datos desbalanceado es normal que no consigamos el mejor resultado debido a que tenemos muchos más datos para algunas clases que para otras. Sin embargo, aunque podría parecer que las redes entrenadas con el *dataset* balanceado podrían dar el mejor resultado, no sucede así. La razón de este problema es que aunque poseamos el mismo número de datos para cada una de las clases, no tenemos un gran número de imágenes por cada clase, lo que hace que el vehículo no sea capaz de aprender a conducir. Es decir, es necesario tener un amplio conjunto de entrenamiento que nos permita aprender las relaciones que deseamos.

Un dato a tener en cuenta es que si nos fijamos en la columna “Programado” se pueden ver los tiempos realizados por el piloto programado, mientras que si nos fijamos en la columna de tiempo de la red 5v+7w sesgada se ven los tiempos logrados con esta red. Los tiempos obtenidos del pilotaje mediante esta red no se encuentran muy lejanos a los

resultados del piloto programado. Esto permite concluir que esta red aprende de forma correcta a conducir de forma autónoma. El resultado de la conducción del vehículo de forma satisfactoria se puede ver en las Figuras 5.4 y 5.5. Una ejecución típica de $5v+7w$ sesgada (imagen recortada) se puede ver en este vídeo ¹. En las Figuras 5.6 y 5.7 se pueden ver dos fotogramas de este vídeo.

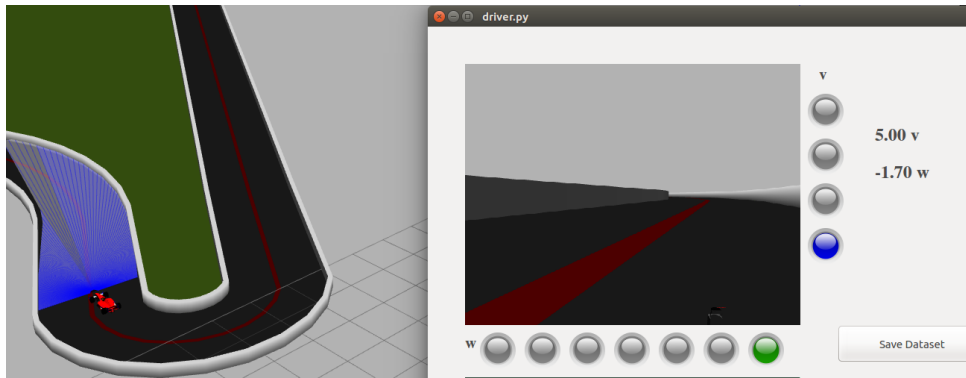


Figura 5.4: Pilotaje del coche en el circuito nurburgrinLine

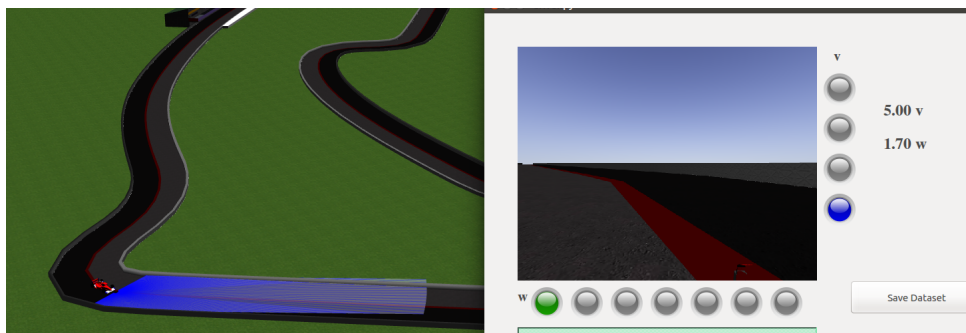


Figura 5.5: Pilotaje del coche en el circuito monacoLine

¹<https://www.youtube.com/watch?v=3Wk6J5kirRY>

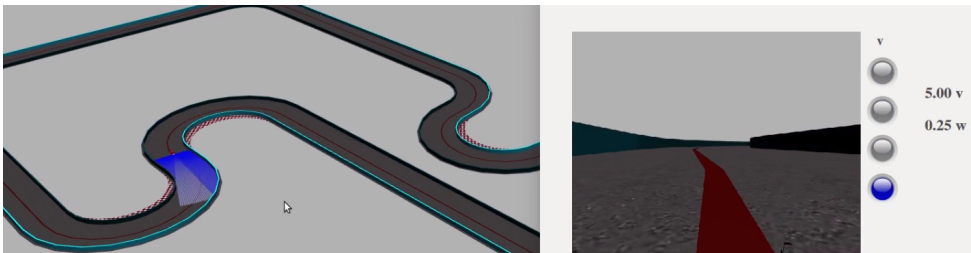


Figura 5.6: Pilotaje del coche en el circuito pistaSimple

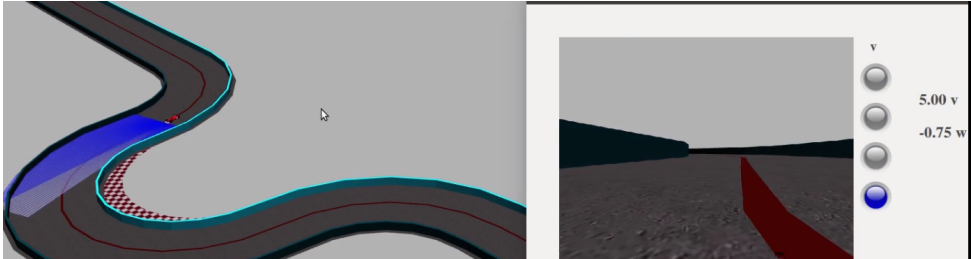


Figura 5.7: Pilotaje del coche en el circuito pistaSimple

Capítulo 6

Redes de regresión

Las redes neuronales son ampliamente empleadas en problemas de regresión. El objetivo de los problemas de regresión es predecir el valor real, gradual, continuo de una variable numérica (variable dependiente) en base a los valores de una o varias variables independientes.

En las redes neuronales, la regresión puede ayudar a modelar la relación entre una variable dependiente (que se está tratando de predecir) y una o más variables independientes (la entrada del modelo). El análisis de regresión puede mostrar si existe una relación significativa entre las variables independientes y la variable dependiente. La ecuación de regresión lineal más simple sigue la siguiente fórmula:

$$y = \beta_1 + \beta_2 X_2 + \dots + \beta_k X_k + \epsilon$$

donde las variables son:

- y : el valor que el modelo de regresión pretende predecir (variable dependiente).
- X_1, X_2, \dots, X_k : uno o más valores que el modelo toma como entrada (variables independientes), usándolos para predecir las variables dependientes.
- $\beta_1, \beta_2, \dots, \beta_k$: ponderaciones (coeficientes) que definen la importancia de cada una de las variables para predecir la variable dependiente.
- ϵ : es el error, es decir, la distancia entre el valor predicho por el modelo y la variable dependiente real y . Los métodos estadísticos pueden usarse para estimar y reducir el error.

Las técnicas de regresión son utilizadas en gran medida para resolver tareas donde el objetivo es predecir valores continuos. Este problema es el que se plantea en este Capítulo, ya que debemos ser capaces de predecir un valor de velocidad continua (de avance o de giro del coche) para una entrada visual dada. En este proyecto se emplean redes de regresión con el fin de predecir desde las imágenes las acciones adecuadas de dirección y velocidad de tracción de un vehículo autónomo.

6.1. Arquitecturas de red

En esta sección se explicarán con detalle las arquitecturas de redes de regresión estudiadas. Las redes neuronales empleadas para regresión son CNN y RNN.

6.1.1. PilotNet

La primera arquitectura de red de regresión estudiada es PilotNet, propuesta por Nvidia en los artículos “End to end learning for self-driving cars” [24] y “Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car” [14]. Es una red neuronal convolucional (CNN) que mapea píxeles en crudo de una sola cámara frontal a comandos de dirección.

La red PilotNet (Figura 6.1) consta de 9 capas, que incluyen una capa de normalización, 5 capas convolucionales y 3 capas *fully-connected*. Las capas convolucionales se diseñaron para realizar la extracción de características. Las dos primeras capas convolucionales emplean un *stride* de tamaño 2x2 y un kernel de tamaño 5x5, donde la primera usa 24 filtros y la segunda 36. Mientras que las 3 últimas capas utilizan un *non-stride* y un kernel de dimensiones 3x3, donde la primera de estas utiliza 48 filtros y la última 64. Las 3 capas *fully-connected* fueron diseñadas para funcionar como un controlador de la dirección. El modelo de red aprende automáticamente las representaciones internas, como la detección de características útiles de la carretera.

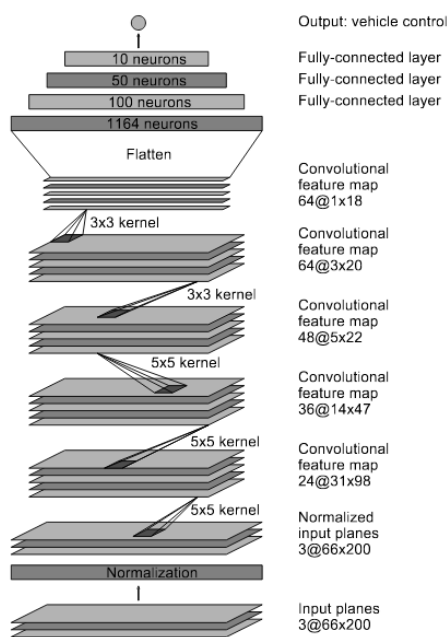


Figura 6.1: Arquitectura PilotNet.

6.1.2. TinyPilotNet

La segunda arquitectura de red empleada se llama TinyPilotNet, que fue propuesta en el artículo “Self-driving a Car in Simulation Through a CNN” [27]. Esta red se deriva de la arquitectura PilotNet [24] [14] y es una reducción de la misma.

La arquitectura TinyPilotnet (Figura 6.2) está formada por dos capas convolucionales que emplean 8 filtros de kernel 3x3, seguidas por 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.

6.1.3. LSTM-TinyPilotNet

La tercera arquitectura estudiada es conocida como LSTM-TinyPilotNet y fue propuesta en el artículo “Self-driving a Car in Simulation Through a CNN” [27]. Esta arquitectura se basa en TintPilotNet (Figura 6.2) con el fin de mejorar el rendimiento de la misma.

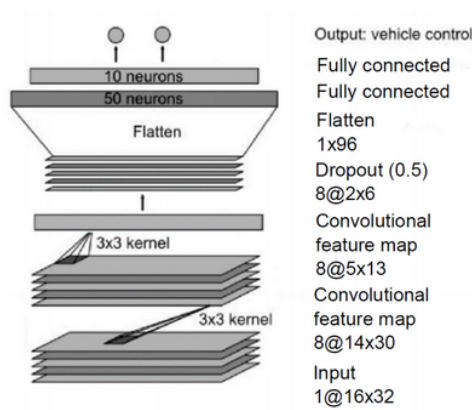


Figura 6.2: Arquitectura TinyPilotNet.

La arquitectura LSTM-TinyPilotNet (Figura 6.3) intenta introducir un efecto de memoria en la red con el fin de tener en cuenta los instantes anteriores y no únicamente los datos de un único instante. Para lograr este efecto se añaden capas ConvLSTM2D a la salida de la red TinyPilotNet. Este tipo de capas mezcla el efecto de las capas LSTM con un efecto convolucional.

La red LSTM-TinyPilotNet está compuesta por 3 capas convolucionales que emplean filtros (8, 16 y 32 filtros) de kernel 3x3, combinadas con capas *maxpooling*. Tras estas capas convolucionales se añade una capa LSTM convolucional (*ConvLSTM2D*) para aportar el efecto de memoria mencionado anteriormente. Finalmente, se añade una capa convolucional con un filtro y kernel de tamaño 3x3, seguida de una capa *fully-connected*.

6.1.4. DeepestLSTM-TinyPilotNet

La cuarta arquitectura empleada se llama DeepestLSTM-TinyPilotNet, propuesta en el artículo “Self-driving a Car in Simulation Through a CNN” [27]. Esta arquitectura se basa en la red LSTM-TinyPilotNet (Figura 6.3). Esta red tiene mayor profundidad, ya que busca aumentar el número de parámetros configurables de la red para conseguir unos resultados mejores en el aprendizaje de los datos.

Esta red (Figura 6.4) está formada principalmente por 3 capas convolucionales que utilizan 8 filtros de kernel 3x3, combinadas con capas *maxpooling*. Estas capas son segui-

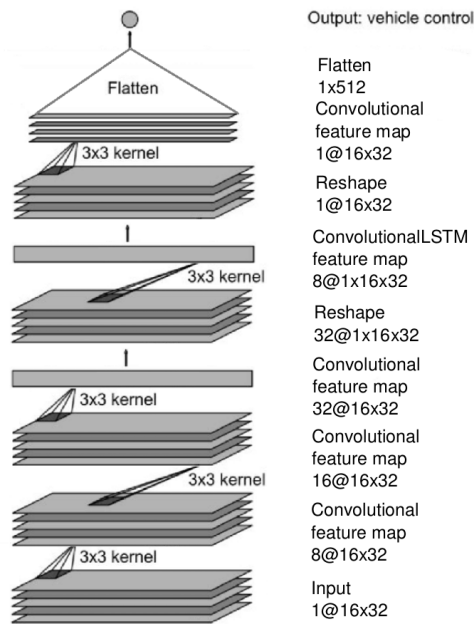


Figura 6.3: Arquitectura LSTM-TinyPilotNet.

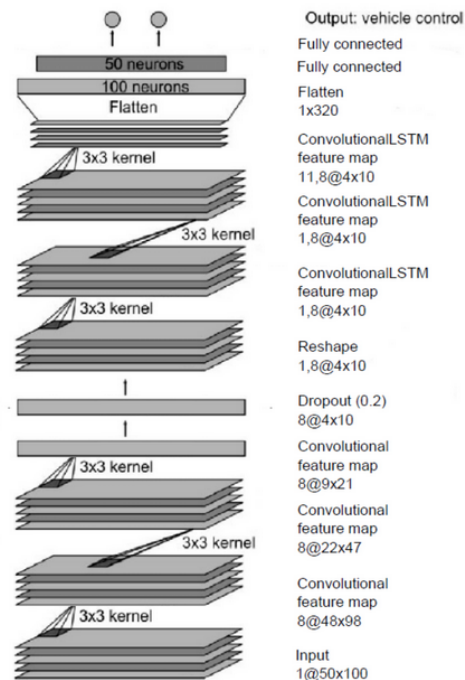


Figura 6.4: Arquitectura DeepestLSTM-TinyPilotNet.

das por 3 capas *ConvLSTM2D* que emplean 8 filtros con un kernel de 5x5 cada una. Estas capas aportan un efecto de memoria, y son seguidas por 2 capas *fully-connected*.

En la arquitectura empleada se ha modificado el número de filtros de las capas *ConvLSTM2D*, empleando en nuestro caso 16 filtros en las primeras 2 capas y 12 en la última. Todos estos filtros son de kernel 3x3 en nuestro caso.

6.2. Experimentos

A continuación se explicarán todos los experimentos realizados durante el entrenamiento de redes de regresión, tanto los relacionados con las dimensiones como los tipos de las imágenes, el aumentado de los datos, etc.

6.2.1. Métricas de evaluación

Es necesario evaluar los resultados logrados tras el entrenamiento de las redes de regresión. Para ello se emplean diferentes métricas de evaluación que cuantifican el rendimiento de la red en el conjunto de *test*.

Las métricas de evaluación se calculan comparando los resultados que predice la red con los resultados de *Ground Truth* (valores reales que toma el piloto programado explícitamente). En las redes de regresión las métricas neuronales evaluadas han sido: *Mean Squared Error (MSE)* y *Mean Absolute Error (MAE)*.

La métrica *Mean Absolute Error (MAE)* es el promedio de la diferencia entre los valores reales y los valores predichos por la red. Esta medida nos da una idea de cuán lejos están las predicciones de los valores reales. Sin embargo, no nos aporta ninguna información acerca de la dirección del error, es decir, si estamos prediciendo un valor por debajo de los datos o prediciendo por encima de los datos. Matemáticamente se expresa como:

$$MAE = \frac{1}{N} \sum_{j=1}^N |y_j - \hat{y}_j| \quad (6.1)$$

Donde N es el número de ejemplos, y_j es el valor real del ejemplo, e \hat{y}_j es el valor predicho por la red para dicho ejemplo.

La métrica Mean Squared Error (MSE) es bastante similar a Mean Absolute Error (MAE), aunque existe una diferencia y es que el MSE toma el promedio del cuadrado de la diferencia entre los valores reales y los valores predichos por la red. La ventaja de MSE es que es más fácil calcular el gradiente que con MAE. A medida que tomamos el cuadrado del error, el efecto de los errores más grande se vuelve más pronunciado que el error más pequeño, por lo que el modelo ahora puede centrarse más en los errores más grandes. Matemáticamente se expresa como:

$$MSE = \frac{1}{N} \sum_{j=1}^N (y_j - \hat{y}_j)^2 \quad (6.2)$$

Donde N es el número de ejemplos, y_j es el valor real del ejemplo, e \hat{y}_j es el valor predicho por la red para dicho ejemplo.

Las métricas operativas más importantes que usaremos para evaluar el rendimiento de las redes son el porcentaje recorrido por el piloto autónomo basado en las redes, así como el tiempo por vuelta al circuito. Estas métricas nos darán una idea real del funcionamiento de las redes en ejecución.

Estas métricas de evaluación que calculamos en el conjunto de *test* dan una idea de cómo de bueno ha sido el entrenamiento, ya que se supone que queremos minimizar el error para que la conducción sea perfecta. Cada una de las medidas se calculará para cada una de las redes entrenadas para v y w .

En las Tablas 6.2.1 y 6.2 se recopilan las métricas promedio para las redes de velocidad lineal (v) y velocidad de rotación (w) para algunas arquitecturas de red como ejemplo. Se

muestran los resultados de las arquitecturas PilotNet, TinyPilotNet, LSTM-TinyPilotNet y DeepestLSTM-TinyPilotNet que han sido entrenadas con imágenes BGR completas (sin recortar). Se analizan estas redes con estos parámetros de entrada porque PilotNet, TinyPilotNet y DeepestLSTM-TinyPilotNet son las que han obtenido unos mejores resultados en las métricas porcentaje recorrido y tiempo por vuelta al circuito.

Tabla 6.1: Métricas de test de redes de regresión (v, imagen completa)

Red	Mean squared error	Mean absolute error
PilotNet	0.205252	0.142394
TinyPilotNet	0.459055	0.296804
LSTM-Tinypilotnet	0.309642	0.414752
DeepestLSTM-Tinypilotnet	0.541856	0.391695

Tabla 6.2: Métricas de test de redes de regresión (w, imagen completa)

Red	Mean squared error	Mean absolute error
PilotNet	0.000316	0.007184
TinyPilotNet	0.001366	0.022281
LSTM-Tinypilotnet	0.000607	0.018625
DeepestLSTM-Tinypilotnet	0.000743	0.018208

En estas tablas se puede observar que la red PilotNet es la que tiene un menor MAE y MSE, por lo que estos resultados nos dan una idea de que es la red que mejor entrenamiento ha tenido, donde deberíamos obtener mejores resultados.

En las tablas anteriores se puede ver que tanto el valor de MSE como el valor de MAE son mayores para las velocidades lineales que para las velocidades de rotación. Esto se debe a que los datos de velocidad de rotación se encuentran en un rango de (-2.9269; 3.1138) en rad/s y los datos de velocidad de tracción se encuentran en el rango (-0.6; 13) m/s . Esto quiere decir que como la velocidad lineal está en un rango de valores mayor, es más probable que el error acumulado sea mayor.

En estas tablas se puede ver también que los resultados neuronales en el conjunto de prueba en algunos casos parece que tienen un error bajo y se corresponde con el resultado de rendimiento operativo logrado (Sección 6.2.2). Este es el caso de las redes PilotNet y TinyPilotNet. Aunque en algunas ocasiones no sucede así. Las métricas generalmente dan una idea de cómo ajustar los parámetros durante el entrenamiento, ya que nuestro objetivo será obtener un error de 0.

Se ha comprobado que los resultados de las métricas neuronales de evaluación buenos pueden dar una idea de que tenemos un buen entrenamiento. Sin embargo, es posible que en algunos casos redes con buenos resultados en las métricas de evaluación, no sean capaces de evitar que el coche choque en algunas situaciones; mientras que redes con peores resultados neuronales pueden lograr una conducción efectiva. Este por ejemplo, es el caso de LSTM-TinyPilotNet, que aunque logre mejores resultados en las métricas neuronales que DeepestLSTM-TinyPilotNet no es capaz de recorrer todos los circuitos, mientras que DeepestLSTM-TinyPilotNet sí. La razón es la misma que se explicó para redes de clasificación en la Sección 5.2.1.

6.2.2. Comparativa entre redes de regresión

Es necesario tener en cuenta algunos aspectos del entrenamiento realizado por las arquitecturas de red mencionadas en la Sección 6.1. Estas redes han sido entrenadas con las imágenes de la cámara frontal centrada del vehículo. Además, en los artículos que propusieron dichas arquitecturas, los datos de entrenamiento se aumentan con imágenes de las cámaras izquierda y derecha del vehículo que simulan el coche en diferentes posiciones fuera del centro y fuera de la orientación. Para las imágenes aumentadas, el comando de control de objetivo se ajusta adecuadamente a uno que llevará el vehículo de vuelta al centro del carril.

En el caso del entrenamiento realizado durante el proyecto, las imágenes de entrada a la red son las imágenes proporcionadas por la cámara del vehículo, situada en la parte frontal *izquierda* del vehículo, que es la empleada por el piloto autónomo programado explícitamente al grabar el conjunto de datos. Es decir, no poseemos una cámara frontal centrada.

El entrenamiento de las redes LSTM-TinyPilotNet y DeepestLSTM-TinyPilotNet no se realiza aleatorizando las imágenes de entrada, ya que el objetivo es que estas redes sean capaces de relacionar los datos actuales con los datos de instantes anteriores. Es decir, se produce un efecto de memoria que hace que los valores de velocidad que predice la red estén influenciados por la secuencia de imágenes.

Además, es necesario mencionar que los datos están algo desbalanceados, como sucedía en las redes de clasificación. Esto se debe a que tenemos más ejemplos de conducción de rectas que de curvas, así mismo más datos de curvas leves que de curvas muy pronunciadas. Por este motivo, antes de realizar el entrenamiento se realiza un preprocesado de los datos donde los valores más atípicos se vuelven a introducir a la red un par de veces. De esta forma se consigue que el vehículo aprenda de ciertas situaciones difíciles de las cuales no podría aprender al tener un número reducido de datos. Algunos de estos datos de los que tenemos menor representación en el conjunto de datos son velocidades lineales negativas o velocidades de rotación con ángulos elevados.

Las redes PilotNet y TinyPilotNet han sido entrenadas únicamente con el conjunto de datos propio *Dataset*; mientras que las redes LSTM-TinyPilotNet y DeepestLSTM-TinyPilotNet han sido entrenadas con los conjuntos de datos propios *Dataset* y *Dataset_Curve*. Se ha añadido este último conjunto de entrenamiento a las redes neuronales recurrentes puesto que necesitamos añadir más información de curvas, que es donde poseemos menos imágenes, y necesitamos introducir las imágenes de forma continua para que sean capaces de conseguir el efecto de memoria que hemos mencionado anteriormente.

Las arquitecturas de red mencionadas en la Sección 6.1 se han empleado para entrenar tanto la velocidad de tracción (v) del vehículo como la velocidad de rotación (w). Los resultados de los entrenamientos y las conclusiones se muestran en esta sección. Primero se expondrán los resultados obtenidos con las cuatro arquitecturas de red mencionadas en la Sección 6.1. Estas redes emplean como entrada la imagen instantánea proporcionada por la cámara del vehículo.

En la Tabla 6.3 se muestran los resultados de las redes neuronales convolucionales

(PilotNet y TinyPilotNet) durante el pilotaje del coche.

Tabla 6.3: Resultados de conducción con redes neuronales de regresión (imagen completa)

Circuitos	Programado	PilotNet		TinyPilotNet	
	Tiempo	%	Tiempo	%	Tiempo
pistaSimple (h)	1' 35"	100 %	1' 41"	100 %	1' 39"
pistaSimple (ah)	1' 33"	100 %	1' 39"	100 %	1' 38"
monacoLine (h)	1' 15"	100 %	1' 21"	100 %	1' 19"
monacoLine (ah)	1' 15"	100 %	1' 23"	100 %	1' 20"
nurburgrinLine (h)	1' 02"	100 %	1' 03"	100 %	1' 05"
nurburgrinLine (ah)	1' 02"	100 %	1' 06"	100 %	1' 06"
curveGP (h)	2' 13"	100 %	2' 20"	100 %	2' 11"
curveGP (ah)	2' 09"	100 %	2' 16"	100 %	2' 06"
pista_simple (h)	1' 00"	100 %	1' 07"	100 %	1' 02"
pista_simple (ah)	59"	100 %	1' 09"	100 %	1' 02'

En esta tabla se puede observar que tanto la red PilotNet como la red TinyPilotNet son capaces de completar el circuito entero en todos los entornos empleados. Si comparamos los tiempos de la columna “Programado” con los tiempos de PilotNet y TinyPilotNet vemos que no están muy alejados de los resultados del piloto autónomo programado explícitamente. Es decir, estas redes aprenden a conducir adecuadamente de forma autónoma. En este caso los resultados obtenidos en las métricas neuronales de evaluación de estas redes eran buenos y se corresponde con el resultado de rendimiento operativo que hemos conseguido.

El resultado de la conducción del vehículo de la red PilotNet (imagen completa) se puede ver en la Figura 6.5 y el resultado de TinyPilotnet se puede ver en la Figura 6.6. Una ejecución típica de PilotNet se puede ver en este vídeo ¹, mientras que una ejecución típica de TinyPilotNet se puede ver en el vídeo ².

¹<https://www.youtube.com/watch?v=WXDACKjgwi4>

²<https://www.youtube.com/watch?v=Mv0fUMADLqE>

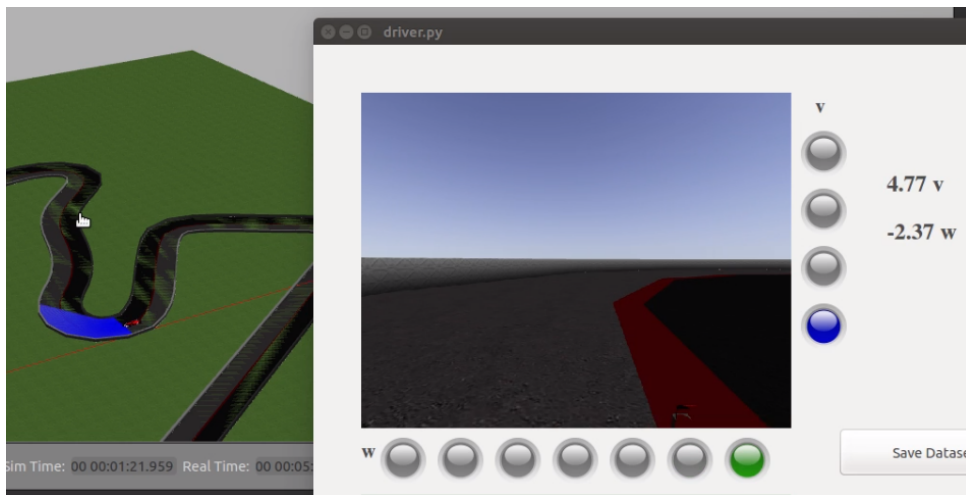


Figura 6.5: Pilotaje del coche en el circuito monacoLine

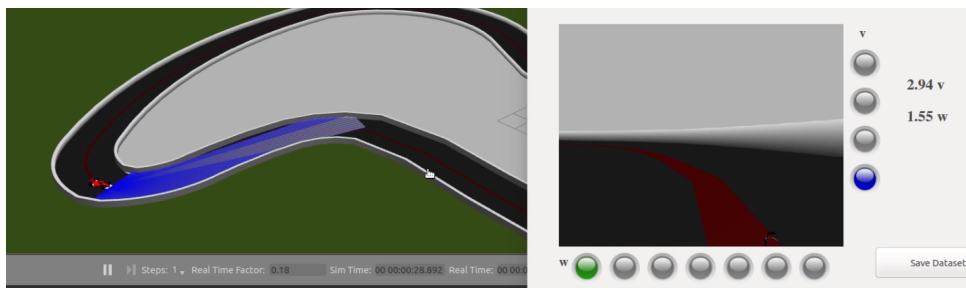


Figura 6.6: Pilotaje del coche en el circuito nurburgringLine

En la Tabla 6.4 se puede observar el resultado del uso de redes neuronales recurrentes con imágenes instantáneas de entrada. Ejemplos de estas redes son LSTM-TinyPilotNet, y DeepestLSTM-TinyPilotNet.

En esta tabla se puede observar que la red DeepestLSTM-TinyPilotNet es capaz de completar todos los circuitos. Si nos fijamos en los resultados de los tiempos logrados por esta red y los comparamos con los que consigue el piloto programado no son muy dispares, aunque la conducción del vehículo mediante la red neuronal tarde un pelín más. Por lo que se puede decir que la conducción del vehículo mediante esta red es eficaz. El resultado de la conducción del vehículo se puede ver en las Figuras 6.7 y 6.8. Una ejecución típica de DeepestLSTM-TinyPilotNet se puede ver en este vídeo ³.

³<https://www.youtube.com/watch?v=-tFzQp0984w>

Tabla 6.4: Resultados de conducción con redes neuronales recurrentes de regresión (imagen completa)

Circuitos	Programado		LSTM-Tinypilotnet		DeepestLSTM-Tinypilotnet	
	Tiempo	%	Tiempo	%	Tiempo	
pistaSimple (h)	1' 35"	100 %	1' 39"	100 %	1' 38"	
pistaSimple (ah)	1' 33"	100 %	1' 40"	100 %	1' 39"	
monacoLine (h)	1' 15"	50 %		100 %	1' 22"	
monacoLine (ah)	1' 15"	12 %		100 %	1' 21"	
nurburgrinLine (h)	1' 02"	20 %		100 %	1' 05"	
nurburgrinLine (ah)	1' 02"	80 %		100 %	1' 08"	
curveGP (h)	2' 13"	100 %	2' 20"	100 %	2' 19"	
curveGP (ah)	2' 09"	100 %	2' 25"	100 %	2' 18"	
pista_simple (h)	1' 00"	100 %	1' 11'	100 %	1' 09"	
pista_simple (ah)	59"	100 %	1' 09"	100 %	1' 08"	

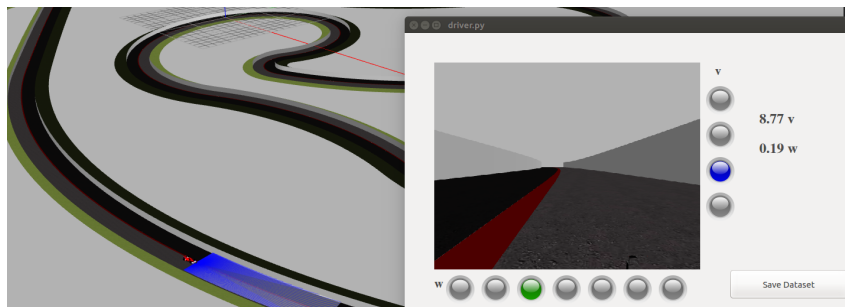


Figura 6.7: Pilotaje del coche en el circuito curveGP

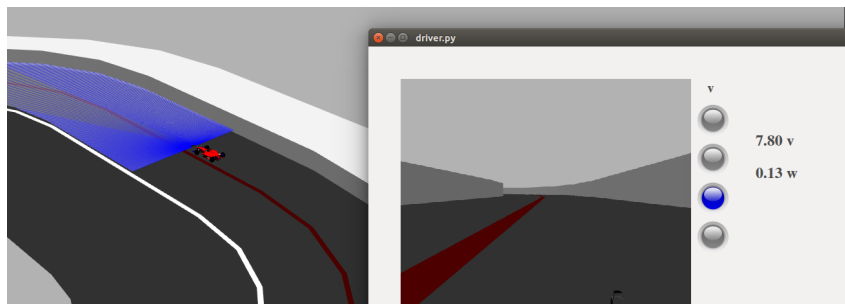


Figura 6.8: Pilotaje del coche en el circuito pista_simple

La red DeepestLSTM-TinyPilotNet logra mejores resultados que la red LSTM-TinyPilotNet. Esto se debe a que introducir mayor profundidad de red mejora la conducción, haciendo que la red sea capaz de aprender información temporal que ayuda a que la conducción sea más suave. De esta forma vemos que el coche tiende a volver a la línea roja constantemente, aunque en algunas situaciones sea complicado.

En primer lugar, se ha llegado a la conclusión de que los datos de entrenamiento tienen una gran influencia en el rendimiento del problema planteado, ya que si no poseemos de datos de todas las posibles situaciones en las que se puede encontrar el vehículo o suficientes datos, la red no podrá aprender y el coche se enfrentará a situaciones desconocidas, por lo que no sabrá qué hacer. Por este motivo, si no disponemos de suficientes imágenes de curvas complejas frente a rectas tendremos un conjunto de entrenamiento desbalanceado y será necesario realizar algún procesado de datos para balancear los mismos o bien crear un nuevo conjunto adicional que incorpore situaciones complejas. De esta forma, las redes serán capaces de aprender cualquier situación en la que se encuentren.

En segundo lugar, hemos visto en las tablas de resultados que en general con redes más profundas logramos unos mejores resultados en la conducción, como con PilotNet o DeepestLSTM-TinyPilotNet. Además, en el pilotaje realizado con la red DeepestLSTM-TinyPilotNet se observa que el pilotaje es más suave, ya que en casi todo momento tiende a volver a la línea roja en cuanto puede el vehículo. Esto se debe al efecto de memoria introducido por la red, que permite que el vehículo tenga un mayor conocimiento acerca de la situación. En el pilotaje del vehículo interesará que la conducción sea más suave, evitando vaivenes en la conducción. Por eso, el mejor resultado es el obtenido con DeepestLSTM-TinyPilotNet, aunque no sea el que logre un menor tiempo por vuelta de circuito.

6.2.3. Aumentado de los datos

Los datos de entrenamiento son una parte fundamental del aprendizaje por parte de las redes neuronales. En algunas ocasiones es difícil extraer una gran cantidad de datos, por este motivo se suele hacer un aumentado de datos. El aumentado de datos consiste en ampliar la información de la red a partir del propio conjunto de datos existente, men-

diante transformaciones controladas.

En el entrenamiento se ha realizado un preprocesado para aumentar los datos de los que disponemos, y de esta forma tener los mismos datos de velocidad de rotación a la izquierda que a la derecha. Es decir, lo que se ha hecho exactamente es emplear la operación *cv2.flip* de OpenCV, que consiste en hacer un espejo de nuestra imagen. De esta forma para una imagen en la que giramos a la izquierda (velocidad de rotación 0.5), ahora tendríamos de la imagen correspondiente pero a la derecha (velocidad de rotación -0.5).

En la Figura 6.9 se puede ver una imagen obtenida por la cámara del coche, mientras que en la Figura 6.10 se puede observar la misma imagen tras aplicar la operación *flip* de OpenCV.

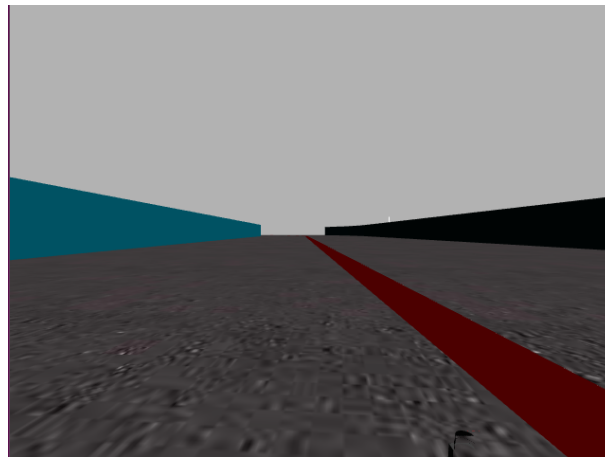


Figura 6.9: Imagen de la cámara

6.2.4. Imágenes de distintas dimensiones

Se ha explorado el empleo de imágenes completas o imágenes recortadas como se hizo también en las redes de clasificación. A continuación, se expondrá una comparativa entre los resultados obtenidos por las redes empleando imágenes de entrada completas e imágenes de entrada recortadas. Los resultados obtenidos al emplear imágenes de entrada completas se pueden observar en las Tablas 6.3 y 6.4 (Sección 6.2.2); mientras que los resultados obtenidos al emplear imágenes recortadas como entrada se exponen a conti-

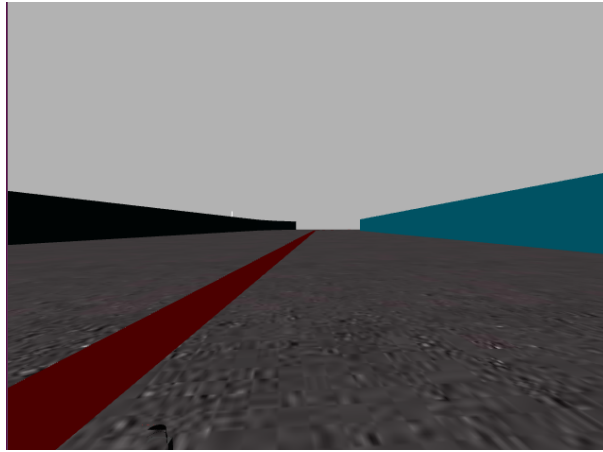


Figura 6.10: Imagen tras realizar la operación *flip*

nuación.

En la Tabla 6.5 se muestran los resultados de las redes neuronales convolucionales entrenadas con imágenes recortadas. Estos son los casos de las redes PilotNet y TinyPilotNet. En esta tabla se puede observar que la red PilotNet es capaz de completar el circuito entero en todos los entornos empleados. Un dato a tener en cuenta es que si nos fijamos en la columna “Programado” se pueden ver los tiempos realizados por el piloto autónomo programado explícitamente, mientras que si nos fijamos en la columna de tiempo de PilotNet se ven los tiempos logrados con esta red. Los tiempos obtenidos del pilotaje mediante esta red no se encuentran muy lejanos a los resultados del piloto programado. Esto permite concluir que esta red aprende de forma correcta a conducir de forma autónoma. En este caso los resultados obtenidos en las métricas neuronales de evaluación de esta red (Sección 6.2.1) eran buenos y se corresponde con el resultado de rendimiento operativo logrado. El resultado de la conducción del vehículo se puede ver en la Figura 6.11. Una ejecución típica de PilotNet se puede ver en este vídeo ⁴.

⁴https://www.youtube.com/watch?v=_pwZHgp8IG4

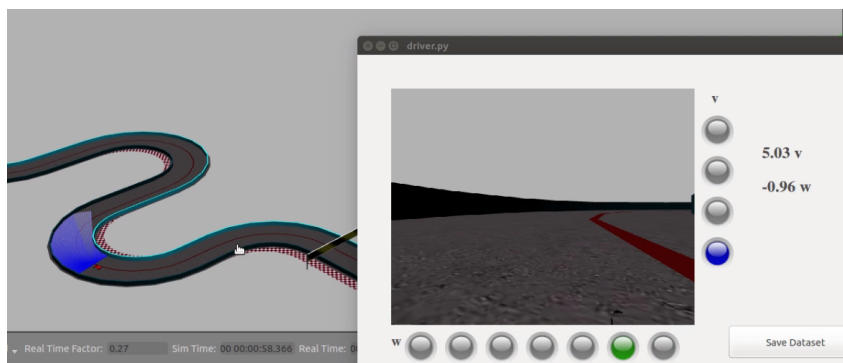


Figura 6.11: Pilotaje del coche en el circuito pistaSimple

Tabla 6.5: Resultados de conducción con redes neuronales de regresión (imagen recortada)

Circuitos	Programado	PilotNet		TinyPilotNet	
	Tiempo	%	Tiempo	%	Tiempo
pistaSimple (h)	1' 35"	100 %	1' 37"	100 %	1' 41"
pistaSimple (ah)	1' 33"	100 %	1' 38"	100 %	1' 41"
monacoLine (h)	1' 15"	100 %	1' 20"	100 %	1' 19"
monacoLine (ah)	1' 15"	100 %	1' 19"	100 %	1' 18"
nurburgrinLine (h)	1' 02"	100 %	1' 04"	100 %	1' 04"
nurburgrinLine (ah)	1' 02"	100 %	1' 06"	100 %	1' 05"
curveGP (h)	2' 13"	100 %	2' 16"	25 %	
curveGP (ah)	2' 09"	100 %	2' 12"	75 %	
pista_simple (h)	1' 00"	100 %	1' 04"	100 %	59"
pista_simple (ah)	59"	100 %	1' 05"	100 %	1' 00'

En la Tabla 6.6 se muestran los resultados de las redes neuronales recurrentes entrenadas con imágenes recortadas. Estas redes intentan introducir un efecto de memoria en la red con el fin de tener en cuenta los instantes anteriores y no únicamente los datos en un único instante. Para lograr este efecto añaden capas ConvLSTM2D. Ejemplos de este tipo de redes son las redes LSTM-TinyPilotNet, y DeepestLSTM-TinyPilotNet.

En esta tabla se puede observar que la red DeepestLSTM-TinyPilotNet (imagen recortada) casi logra completar todos los circuitos, únicamente choca en uno en una curva complicada. Este hecho refleja que es complicado lograr combinar una red de v y una red

de w que logren un buen rendimiento conjuntamente. La red DeepestLSTM-TinyPilotNet logra conseguir mejores resultados que la red LSTM-TinyPilotNet. Esto se debe a que introducir mayor profundidad de red mejora la conducción.

Tabla 6.6: Resultados de conducción con redes neuronales recurrentes de regresión (imagen recortada)

Circuitos	Programado		LSTM-Tinypilotnet		DeepestLSTM-Tinypilotnet	
	Tiempo	%	Tiempo	%	Tiempo	%
pistaSimple (h)	1' 35"	100 %	1' 40"	100 %	1' 36"	
pistaSimple (ah)	1' 33"	100 %	1' 38"	100 %	1' 37"	
monacoLine (h)	1' 15"	50 %		100 %	1' 21"	
monacoLine (ah)	1' 15"	35 %		100 %	1' 19"	
nurburgrinLine (h)	1' 02"	40 %		100 %	1' 04"	
nurburgrinLine (ah)	1' 02"	50 %		80 %		
curveGP (h)	2' 13"	100 %	2' 17"	100 %	2' 17"	
curveGP (ah)	2' 09"	100 %	2' 04"	100 %	2' 19"	
pista_simple (h)	1' 00"	100 %	1' 07"	100 %	1' 05"	
pista_simple (ah)	59"	100 %	1' 03"	100 %	1' 08"	

En este proyecto se ha logrado que un vehículo sea capaz de conducir de forma autónoma mediante redes de regresión. Este es el caso de las redes PilotNet (imagen recortada e imagen completa), TinyPilotNet (imagen completa) y DeepestLSTM-TinyPilotNet (imagen completa). Los resultados de los tiempos empleados para recorrer cada uno de los circuitos no están muy alejados de los resultados logrados por el piloto programado explícitamente.

Gracias a los diferentes experimentos realizados y los resultados obtenidos, se pueden sacar algunas conclusiones acerca del entrenamiento de estas redes y del efecto que tiene emplear una u otra imagen.

Los resultados al comparar la Tabla 6.3 con la 6.5 y la 6.4 con la 6.6 muestran que en las redes de regresión es mejor emplear una imagen de entrada completa que una imagen

recortada. Esto se debe a que en el caso de la imagen recortada puede ser que la red necesite información acerca de la valla que no aparece en dicha imagen, mientras que en la imagen completa sí. La información de la valla puede ser necesaria para saber si el vehículo se está acercando demasiado a la valla y debe disminuir la velocidad o incluso dar marcha atrás. Además, en estos casos la velocidad de rotación debería ser mayor para girar más y volver a la línea roja.

6.2.5. Tipo de imagen de entrada

La imagen de entrada de la red PilotNet, en el artículo “End to end learning for self-driving cars” [24], se divide en planos YUV y se pasa a la red. En este proyecto la red PilotNet ha sido entrenada en el espacio de color BGR en vez de en YUV.

En el artículo “Self-driving a Car in Simulation Through a CNN” [27], las redes TinyPilotNet, LSTM-TinyPilotNet, y DeepestLSTM-TinyPilotNet fueron entrenadas con imágenes con un único canal formado por el canal de saturación del espacio de color HSV. En nuestro caso las redes se han entrenado con imágenes en el espacio de color BGR.

Con el fin de introducir temporalidad en una red de extremo a extremo (CNN) se concatenan varias imágenes de entrada separadas por un margen 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 es la única variable que se modifica, es decir, no se modifica la red, que en nuestro caso será PilotNet. Por este motivo, las imágenes se concatenan en la dimensionalidad de profundidad, es decir, en el canal, y no en una nueva dimensión. De esta forma no hay que modificar la dimensionalidad de la red. Por ejemplo, si se apilan dos imágenes seguidas en el espacio de color RGB de tamaño $(65, 160, 3)$ su tamaño se modificaría a $(65, 160, 6)$. En este proyecto se ha estudiado este uso de imagen apilada, pero en vez de concatenar varias imágenes seguidas como sucede en el artículo “From Pixels to Actions: Learning to Drive a Car with Deep Neural Networks” [29], se han apilado 2 imágenes separadas por un margen de 10 imágenes. A las redes entrenadas con este concepto de imagen apilada las hemos llamado PilotNet (*stacked*).

Además, se han realizado otros experimentos con el fin de introducir temporalidad

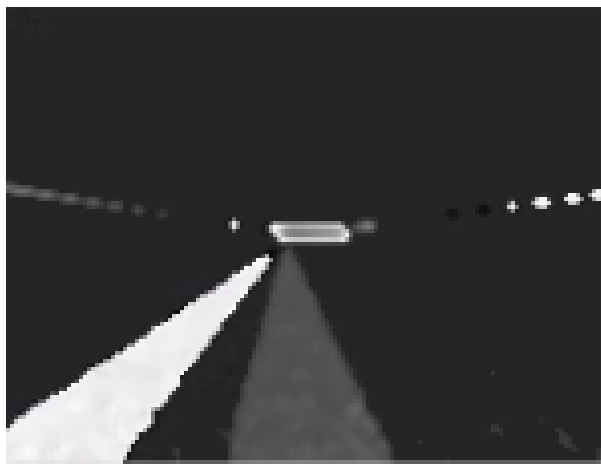


Figura 6.12: Imagen diferencia

con una única imagen y una red extremo a extremo, como PilotNet en nuestro caso. Para ello se ha creado una imagen diferencia ($\text{img}(t) - \text{img}(t-1)$) que aporte información temporal. Esta imagen dará información de los cambios que han sucedido entre un instante t y un instante $t-10$, ya que hemos empleado una diferencia de 10 fotogramas. La imagen diferencia se ha creado en escala de grises y entre un rango de -128 a 128 con el fin de introducir información en la imagen acerca de si estamos en una curva hacia la izquierda o hacia la derecha. Además, en esta imagen se ha realizado un filtrado para eliminar ruido no deseado en la misma. La imagen diferencia creada se puede ver en la Figura 6.12. La red entrenada con este formato de imagen la llamaremos *Temporal (dif)*.

Aprovechando la imagen diferencia creada, se ha experimentado también con el concepto de imagen apilada. En este caso, en vez de apilar una imagen y otra imagen separada 10 fotogramas, se apilará la imagen actual y la imagen diferencia con el fin de ver si esta imagen aportará más información de temporalidad. Cuando apilamos una imagen BGR con la imagen diferencia en la dimensionalidad de profundidad (canal), las dimensiones ya no son $(65, 160, 6)$ como cuando apilábamos dos imágenes BGR, sino que ahora las dimensiones serán $(65, 160, 4)$. Las redes entrenadas con este tipo de imágenes apiladas se conocerán como PilotNet (*stacked, dif*).

En la Tabla 6.7 se muestran los resultados operativos de las redes neuronales convolucionales entrenadas con imágenes completas donde intentamos introducir temporalidad en

dichas imágenes. Ejemplos de este tipo de redes son las redes PilotNet (*stacked*), PilotNet (*stacked, dif*), y Temporal (*dif*).

Tabla 6.7: Resultados de conducción con redes neuronales de regresión introduciendo temporalidad (imagen completa)

Circuitos	Programado	PilotNet (<i>stacked</i>)		PilotNet (<i>stacked, dif</i>)		Temporal (<i>dif</i>)	
	Tiempo	%	Tiempo	%	Tiempo	%	Tiempo
pistaSimple (h)	1' 35"	100 %	1' 40"	100 %	1' 43"	35 %	
pistaSimple (ah)	1' 33"	100 %	1' 46"	10 %		10 %	
monacoLine (h)	1' 15"	50 %		5 %		3 %	
monacoLine (ah)	1' 15"	7 %		5 %		3 %	
nurburgrinLine (h)	1' 02"	50 %		8 %		8 %	
nurburgrinLine (ah)	1' 02"	80 %		50 %		3 %	
curveGP (h)	2' 13"	25 %		25 %		12 %	
curveGP (ah)	2' 09"	100 %	2' 07"	75 %		3 %	
pista_simple (h)	1' 00"	100 %	1' 11'	100 %	1' 03"	25 %	
pista_simple (ah)	59"	100 %	1' 08"	100 %	1' 02"	15 %	

En esta tabla se puede observar que en ninguna de las tres redes conseguimos completar todos los circuitos, aunque se puede ver que las redes PilotNet (*stacked*) y (*stacked, dif*) consiguen completar varios circuitos, mientras que la red Temporal (*dif*) no consigue completar ninguno. Esto se debe a que apilar dos imágenes proporciona más información que únicamente emplear la imagen diferencia. También a que es bastante complejo crear una imagen que proporcione información a la red acerca de cómo ha variado la situación del vehículo o no, ya que si estábamos en recta y seguimos en recta la imagen diferencia prácticamente tendrá valor 0 en todos sus píxeles, lo que no aportará mucha información al vehículo acerca de su situación. Sin embargo, si la situación ha cambiado mucho en un periodo de tiempo muy corto esta imagen poseerá mucha información acerca de la nueva situación. Además, otra complejidad es saber qué instante de tiempo debemos tomar para lograr una conducción efectiva, es decir, en los experimentos se ha tomado la diferencia entre dos imágenes separadas por 10 *frames* (mejor resultado obtenido en los experimentos), pero es bastante complicado saber qué margen debemos tomar entre las dos imágenes para aportar información a la red en todas las situaciones.

Otra conclusión que se puede sacar de esta tabla es la dificultad de lograr una imagen apilada que proporcione un buen rendimiento en la conducción. El motivo es la dificultad de saber cuál es la imagen apilada perfecta, es decir, es posible que la red necesite 2, 3, 4, ... N imágenes, y que estas imágenes se encuentren seguidas en espacio del tiempo o separadas por un número de fotogramas. La dificultad de conocer estos parámetros se debe a que no tenemos siempre la misma situación (recta, curva leve o curva pronunciada), y lograr una combinación de imágenes apiladas que consiga proporcionar información a la red sobre todas las situaciones será una tarea difícil. Inicialmente las pruebas se hicieron apilando 3 imágenes separadas por 2 fotogramas, es decir, que para la imagen en el momento t concatenamos la imagen t , $t-3$ y $t-6$. Pero esta combinación de imágenes daba peor resultado que emplear 2 imágenes apiladas separadas por 10 fotogramas, que es el caso de los resultados de la tabla.

Además, se han obtenido las métricas neuronales promedio (Sección 6.2.1) para las redes PilotNet (*stacked*), PilotNet (*stacked dif*), y Temporal (*dif*). Estas métricas se han calculado para las redes de velocidad lineal y velocidad de rotación con imágenes completas. Los resultados se muestran en las Tabas 6.8 y 6.9.

Tabla 6.8: Métricas de test de redes de regresión (v, imagen completa)

Red	Mean squared error	Mean absolute error
PilotNet (stacked)	1.084940	0.469273
PilotNet (stacked, dif)	1.889947	0.468693
Temporal (dif)	0.442861	0.316337

Tabla 6.9: Métricas de test de redes de regresión (w, imagen completa)

Red	Mean squared error	Mean absolute error
PilotNet (stacked)	0.004471	0.034673
PilotNet (stacked, dif)	0.005292	0.031134
Temporal (dif)	0.002155	0.029771

En estas tablas se puede ver que los resultados en el conjunto de prueba en algunos casos parece que tienen un error bajo, pero la conducción no tiene éxito, ya que el resulta-

do de las métricas es un promedio de todos los valores. Es decir, las métricas neuronales no reflejan el comportamiento exacto de la conducción del vehículo.

Los resultados dan una idea de la complejidad que tiene proporcionar información temporal a una red neuronal en una imagen, ya sea diferencia o apilada. El motivo es que no tenemos una idea de cómo interpreta esta información la red de forma interna, es decir, no sabemos qué partes de la imagen temporal considera relevantes y cuáles no. No obstante, se puede esperar que si se lograra conseguir introducir información temporal a las redes extremo a extremo la conducción sería más suave, es decir, el coche iría casi en todos los instantes de tiempo por encima de la línea roja, ya que sería capaz de decir “estoy entrando en una curva y necesito bajar la velocidad” o “estoy saliendo de una curva y puedo aumentar poco a poco la velocidad”.

Capítulo 7

Conclusiones

En este capítulo se exponen las conclusiones finales obtenidas, así como los posibles trabajos futuros, tras analizar las diferentes redes neuronales empleadas (redes de regresión y clasificación) para conducción autónoma basada en visión, y algunos experimentos realizados con las mismas.

Además, se detallarán los cuatro subobjetivos logrados: (1) creación de una aplicación de pilotaje neuronal, (2) programación de un piloto autónomo basado en visión y generación de bases de datos supervisadas, (3) estudio de redes neuronales convolucionales de clasificación para conducción autónoma, (4) estudio de redes neuronales convolucionales y recurrentes de regresión para conducción autónoma.

7.1. Conclusiones

En este trabajo se ha alcanzado satisfactoriamente el objetivo global de explorar diferentes redes neuronales de clasificación y regresión para la conducción de un vehículo mediante control visual, así como realizar diferentes experimentos con las mismas. El objetivo era lograr que el coche fuera capaz de conducir en diferentes entornos en el simulador Gazebo y se ha conseguido.

Primero, se ha diseñado y programado una aplicación ROS, llamada Piloto, que nos ha permitido la conducción autónoma neuronal del coche de carreras y evaluar el rendimiento operativo de las redes en el control del vehículo. Esta aplicación realiza las conexiones de

los sensores y actuadores del vehículo, y además, permite cargar y emplear diferentes redes neuronales. La aplicación consta de una interfaz gráfica que proporciona datos durante el pilotaje del coche.

Segundo, se ha programado un piloto autónomo basado en visión y control PD con el que se han generado varias bases de datos supervisadas con las que realizar los entrenamientos neuronales. La primera base de datos, llamada *Dataset*, posee datos de rectas y de curvas más o menos abruptas. El segundo conjunto de datos, *Dataset_Curves*, solamente posee curvas.

En tercer lugar, se ha experimentado con redes de clasificación, evaluando la influencia de la cuantización de las clases, es decir, tanto el número de clases como el rango de valores que abarca cada una de estas clases. Estudiando el efecto de estas cuantizaciones se ha conseguido que la red de clasificación $5v+7w$ sesgada (imagen recortada) fuera capaz de recorrer todos los circuitos de manera satisfactoria. Se han sacado algunas conclusiones acerca del entrenamiento de estas redes y de la cuantización de las clases:

- El número de las clases y el rango de los valores que abarca cada clase tiene una gran influencia en el rendimiento de la conducción. El uso de 7 clases de velocidad de rotación y 5 clases de velocidad lineal es suficiente para lograr una buena conducción. En caso de tener menos clases el rendimiento sería deficiente, además de que es necesario ajustar el rango de valores de velocidad que abarca cada clase adecuadamente.
- El uso de una imagen recortada en este tipo de redes mejora el rendimiento, ya que la red no se distrae con información de la imagen que no es importante para la conducción.
- El entrenamiento más adecuado se realiza con redes sesgadas, es decir, ponderando las clases para dar más influencia a aquellas clases donde tenemos menor número de datos o clases que se corresponden con situaciones más difíciles para el coche.

El cuarto subobjetivo era la experimentación con redes de regresión, evaluando el uso de diferentes arquitecturas, así como la influencia que tiene la imagen de entrada empleada. Experimentando con estos aspectos se ha logrado que las siguientes redes de regresión

completen todos los circuitos: PilotNet (imagen recortada e imagen completa), TinyPilotNet (imagen completa) y DeepestLSTM-TinyPilotNet (imagen completa). Se han sacado algunas conclusiones acerca del entrenamiento de las redes con las que se ha experimentado, así como de la influencia del tipo de la imagen de entrada:

- El uso de una imagen completa mejora el rendimiento de las redes de regresión, ya que estas imágenes disponen de información de las vallas que hay alrededor del circuito y que pueden dar idea de si el coche se está aproximando a ellas o no, por lo que debería disminuir la velocidad y girar en el sentido adecuado. En las imágenes recortadas no disponemos de esta información, aunque la red PilotNet sí consigue lograr un resultado efectivo con este tipo de imágenes.
- El empleo de redes más profundas mejora el resultado de la conducción. Este es el caso de PilotNet y DeepestLSTM-TinyPilotNet frente a TinyPilotNet y LSTM-TinyPilotNet.
- Al introducir capas *ConvLSTM2D* en las redes se observa un pilotaje más suave. Es el caso de la red DeepestLSTM-TinyPilotNet, donde vemos que el coche en casi todo momento intenta volver a situarse encima de la línea roja. Esto se debe al efecto de memoria introducido por la red, que permite que el vehículo tenga un mayor conocimiento acerca de la situación.
- Es complejo conseguir una imagen apilada perfecta para la conducción, ya que es necesario alcanzar un número de imágenes a apilar adecuado, así como un margen temporal adecuado entre las mismas.
- Proporcionar una imagen diferencia con información relevante acerca de cómo ha variado la situación del vehículo o no (sigues en recta, entras en una curva, etc) es bastante complejo.
- Es difícil proporcionar información temporal a una red neuronal en una imagen, ya sea la imagen diferencia o la imagen apilada. El motivo es que no tenemos una idea de cómo interpreta esta información la red de forma interna, es decir, no sabemos qué partes de la imagen temporal considera relevantes y cuáles no.

Además, a lo largo del proyecto se han extraído algunas conclusiones que sirven tanto para las redes de clasificación como para las redes de regresión empleadas para la conducción autónoma. Algunas de estas son:

- Los datos de entrenamiento que empleemos influirán ampliamente en la calidad de los resultados de la red. Por este motivo es necesario disponer de un amplio conjunto de entrenamiento que permita aprender de todas las situaciones a las que se pueda enfrentar el vehículo. En ocasiones es difícil disponer de muchos datos de situaciones difíciles, por eso será necesario realizar algún procesamiento de los datos que nos permita balancear un poco los mismos. De esta forma las redes no aprenderán únicamente a conducir en situaciones simples.
- Buenos resultados en las métricas neuronales de evaluación no implican una buena conducción. Es posible que el empleo de redes con buenas métricas neuronales dé como resultado que el vehículo choque contra la valla, y en cambio en otros casos donde se obtengan peores resultados neuronales el vehículo sea capaz de conducir.
- El entrenamiento de las redes neuronales de conducción autónoma es complejo y necesita mucha experimentación. Esto se debe a que debemos emplear una red de velocidad lineal y una red de velocidad angular para el pilotaje, y en algunas ocasiones esta relación puede fallar ya sea por parte de una red u otra. Además, en el pilotaje si predecimos un valor mal puede ser que no afecte mucho a la calidad de la conducción, pero si predecimos varios valores erróneamente de forma continua el coche se desviará de la línea roja chocando con la valla.
- Tanto en redes de clasificación como en redes de regresión se ha logrado que alguna red sea capaz de efectuar una conducción autónoma en todos los circuitos de prueba. Los tiempos de conducción conseguidos por todas las redes satisfactorias no se encuentran muy lejanos a los tiempos del piloto programado explícitamente. Esto nos da una idea de que las redes están aprendiendo adecuadamente desde los datos que poseen. Debido a esto se puede establecer que si el piloto programado en una situación difícil no actúa de forma exacta, la red tampoco predecirá un valor perfecto de conducción. Esta situación se puede dar en el proyecto, ya que el conjunto de datos está basado en un piloto programado que realiza una conducción autónoma basándose en visión, pero no es necesariamente el pilotaje ideal.

- Entrenar dos redes (velocidad de rotación y velocidad lineal) que conjuntamente sean efectivas en todos los entornos de los que se dispone es difícil, y por ello se necesita bastante tiempo de experimentación.

7.2. Trabajos futuros

A continuación, se detallarán posibles líneas concretas en las que se puede mejorar cada una de las redes.

- Probar la conducción autónoma basada en redes neuronales en un robot real. De esta forma se podría ver el rendimiento de cada red y comprobar cuál es la más efectiva en una situación real.
- Grabar una base de datos adicional de situaciones bastante complejas o situaciones que desconoce el vehículo. Algunos ejemplos podrían ser el vehículo ha perdido de vista la línea roja o se encuentra en una curva bastante abrupta que no ha visto con anterioridad.
- Realizar un estudio más amplio de la cuantización de las clases (número de clases y rango) permitiendo ajustar de forma más precisa el número de clases y el rango a los valores de los que disponemos en el conjunto de datos. Una posibilidad sería incluir un mayor número de clases para los ángulos de giro (de mayor valor) que se dan en las situaciones complejas (pérdida de carretera o curvas muy abruptas), ajustando de esta forma más la conducción a este tipo de situaciones.
- Una dificultad encontrada en el proyecto ha sido cómo incorporar información temporal con imágenes apiladas. Una posible mejora es realizar un estudio más amplio que nos lleve a ajustar de manera más adecuada el número de imágenes que apilamos y el margen entre las mismas.
- Realizar un estudio más amplio de la imagen diferencia que nos permita introducir información temporal en una única imagen. El objetivo es buscar qué margen de tiempo entre las dos imágenes de las que se realiza la diferencia es el adecuado. Además, una posible mejora consiste en el estudio de la posibilidad de incluir algún

preprocesado de imagen que nos permita resaltar la información que nos interesa en las imágenes.

- Crear un algoritmo que intente explicar lo que las redes aprenden y cómo toman sus decisiones. Este método fue desarrollado en el artículo “Explaining how a deep neural network trained with end-to-end learning steers a car” [14] [26], donde se intenta conocer cómo toma sus decisiones PilotNet. En este artículo se crea un método que determina qué elementos en la imagen influyen más en las decisiones de las velocidades de la red. A las secciones de la imagen con esta propiedad las denominan objetos salientes. La idea sería emplear este conocimiento de los objetos salientes de la imagen para ajustar parámetros de las redes y mejorar así la conducción. Además, esta idea nos podría ayudar a crear una imagen diferencia adecuada, ya que seríamos capaces de saber cuál es la información relevante para las redes.

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://andrealzcano.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. Alvin: 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.smmmt.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://socs.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_md1/lic/IEL/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://gazebo.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, Praseon 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.nnet.categorical_crossentropy, 2017. [Accedido 13 de Junio de 2019].

- [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] Jordi Torres. *Deep Learning, Introducción práctica con Keras*. WATCH THIS SPACE, 2018. <https://torres.ai/deep-learning-inteligencia-artificial-keras>.
- [44] Yann LeCun. Gradient-based learning applied to document recognition. 1998.
- [45] Adrian Rosebrock. Multi-label classification with Keras, *PyImageSearch*. <https://www.pyimagesearch.com/2018/05/07/multi-label-classification-with-keras/>, 2018. [Accedido 27 de Junio de 2019].
- [46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2015.
- [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-camposdelaiabzjh>, 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 Móviles 2009/10: 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=2ahUKEwj3_TF9tziAhUuxYUKHdmADZQQFjAFegQIAxAC&url=http%3A%2F%2Fwww.roboticaeducativa.org%2Fmod%2Fresource%2Fview.php%3Fid%3D2051&usg=AOvVaw1WxJXLKEuuu7WpK08VivRb, 2010. [Accedido 30 de Mayo de 2019].
- [52] Abdelmalik Moujahid Pedro Larra naga, I naki Inza. Tema 8. Redes Neuronales. <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/monograis/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, Pontificia 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 Bungal. Deep Learning básico con Keras (Parte 1). <https://enmilocalfunciona.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-python-deep-learning-library-keras/1, 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].

- [97] Packt. ROS Architecture and Concepts. <https://hub.packtpub.com/ros-architecture-and-concepts/>, 2016. [Accedido 15 de Junio de 2019].
- [98] 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].
- [99] 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].
- [100] 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].
- [101] JdeRobot. Página del repositorio dl-objectdetector. <https://github.com/JdeRobot/dl-objectdetector>, 2018. [Accedido 18 de Junio de 2019].
- [102] 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].
- [103] Juan Ignacio Bagnato. Qué es overfitting y underfitting y cómo solucionarlo, *Aprende Machine Learning*. <https://www.aprendemachinlearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/>, 2017. [Accedido 23 de Junio de 2019].
- [104] 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].
- [105] 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].
- [106] 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].
- [107] 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].
- [108] 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].
- [109] 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].
- [110] srirangatarun. Video Frame Prediction with Keras, *Machine Learning Concepts*. <https://srirangatarun.wordpress.com/2018/07/09/video-frame-prediction-with-keras/>, 2018. [Accedido 25 de Junio de 2019].
- [111] Adrian Rosebrock. Keras and Convolutional Neural Networks (CNNs), *PyImageSearch*. <https://www.pyimagesearch.com/2018/04/16/keras-and-convolutional-neural-networks-cnns/>, 2018. [Accedido 25 de Junio de 2019].
- [112] Muhammad Rizwan. LeNet-5 – A Classic CNN Architecture , *engMRK*. <https://engmrk.com/lenet-5-a-classic-cnn-architecture/>, 2018. [Accedido 26 de Junio de 2019].
- [113] Max Pechyonkin. Key Deep Learning Architectures: LeNet-5. <https://medium.com/@pechyonkin/key-deep-learning-architectures-lenet-5-6fc3c59e6f4>, 2018. [Accedido 26 de Junio de 2019].
- [114] Aditya Mishra. Metrics to Evaluate your Machine Learning Algorithm, *Towards Data Science*. <https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234>, 2018. [Accedido 27 de Junio de 2019].

- [115] Mohammed Sunasra. Performance Metrics for Classification problems in Machine Learning, *Medium*. <https://medium.com/thalus-ai/performance-metrics-for-classification-problems-in-machine-learning-part-i-b085d> 2017. [Accedido 28 de Junio de 2019].
- [116] Juan Miguel Marín Diazaraque. Introducción a las redes neuronales aplicadas. <http://halweb.uc3m.es/esp/Personal/personas/jmmarin/esp/DM/tema3dm.pdf>. [Accedido 28 de Junio de 2019].
- [117] Stéphane Lathuilière, Pablo Mesejo, Xavier Alameda-Pineda, and Radu Horaud. A comprehensive analysis of deep regression. *IEEE transactions on pattern analysis and machine intelligence*, 2019.
- [118] James McCaffrey. Ejecución de pruebas: regresión basada en redes neuronales. <https://msdn.microsoft.com/es-es/magazine/mt683800.aspx>, 2016. [Accedido 29 de Junio de 2019].