

UNIVERSIDAD MAYOR DE SAN ANDRÉS
FACULTAD DE INGENIERÍA
CARRERA DE INGENIERÍA ELECTRÓNICA



PROYECTO DE GRADO

**“Aprendizaje fin a fin para la conducción autónoma de vehículos domésticos
usando visión artificial y redes neuronales convolucionales”**

POSTULANTE: JOSE EDUARDO LARUTA ESPEJO

TUTOR: JAVIER SANABRIA GARCIA

D.A.M.: GONZALO SAMUEL CABAS MORALES

LA PAZ, DICIEMBRE 2018

*Dedicado a mis padres Edwin y Lourdes,
pilares fundamentales de mi formación y principal inspiración
en mi búsqueda de superación personal y profesional.*

Agradecimientos

Agradezco infinitamente a mis padres Edwin y Lourdes, por su incansable e incondicional apoyo y paciencia en mi desarrollo personal y moral.

A mi asesor Javier Sanabria, por su valiosa y desinteresada guía, enseñanzas y consejos en el ámbito académico, ético y profesional en el desarrollo de este proyecto y a lo largo de toda la carrera.

A mis profesores, por haberme transmitido amablemente su experiencia y conocimiento en las distintas asignaturas en toda la carrera, garantizándome una formación ingenieril integral.

A mis compañeros y amigos con los que pude compartir experiencias y conocimientos, que han Enriquecido mi desarrollo profesional dentro de la carrera y mi desarrollo personal fuera de la misma.

Resumen

Los vehículos autónomos han pasado de ser un tema de ciencia ficción a convertirse en una realidad cada vez más cercana. Si bien existe un recorrido muy largo para llegar a implementar sistemas completamente autónomos en las calles, los recientes avances en la tecnología junto y creciente interés económico de grandes empresas, universidades y centros de investigación en el mundo han hecho posible la inclusión exitosa de diversos niveles de autonomía en vehículos, con fines de uso doméstico e industrial. El presente proyecto se centra en el desarrollo de un sistema de conducción autónoma basado en visión artificial, para la generación de comandos de control para la conducción autónoma de un vehículo doméstico. Se ha logrado desarrollar un sistema de aprendizaje “fin a fin” basado en una red neuronal convolucional, que consta de un modelo de predicción que genera comandos de control a partir de un estímulo visual proveniente de una cámara monocular. El sistema de aprendizaje está implementado sobre una plataforma de cómputo de bajo costo y bajo consumo de energía basado en un microcontrolador ARM Cortex M y una SBC Raspberry Pi encargados del control de bajo nivel en tiempo real; la adquisición de datos de entrenamiento, el entrenamiento de la red neuronal y un sistema de control e inferencia autónomo implementados en los lenguajes de programación Python y C++ usando ROS y Tensorflow. Finalmente, se ha validado el entrenamiento de la red neuronal convolucional en conjunto con todo el sistema de conducción autónoma en base a pruebas estadísticas de rendimiento y un análisis de representaciones internas de la red a estímulos visuales de diversa naturaleza.

Índice general

Agradecimientos	II
Resumen	III
Lista de figuras	VIII
Lista de tablas	X
1. Introducción	1
1.1. Antecedentes	1
1.1.1. Sistemas de Conducción Autónoma	1
1.1.1.1. Arquitectura general de un sistema de conducción autónoma	3
1.1.1.2. Sistemas de aprendizaje fin a fin	4
1.2. Justificación del Proyecto	4
1.2.1. Justificación académica	4
1.2.2. Justificación tecnológica	5
1.2.3. Justificación técnica	5
1.3. Análisis de la problemática y planteamiento del problema	5
1.3.1. Análisis de la problemática	5
1.3.2. Planteamiento del problema	6
1.4. Objetivos	8
1.4.1. Objetivo General	8
1.4.2. Objetivos Específicos	8
1.5. Alcance	9
1.6. Límites	9
2. Marco Teórico	10
2.1. Sistemas de Conducción Autónoma	10
2.1.1. Niveles de Autonomía SAE	12
2.1.2. Tecnologías requeridas	12
2.1.2.1. Estado del vehículo	13
2.1.2.2. Estado del entorno	13
2.1.3. Arquitectura de un sistema de conducción autónoma	14

2.1.3.1.	Subsistema de control y actuación	14
2.1.3.2.	Subsistema de adquisición de datos y entrenamiento	15
2.1.3.3.	Subsistema de inferencia y conducción autónoma	15
2.2.	Visión Artificial	15
2.2.1.	Caracterización de una imagen digital	17
2.3.	Redes Neuronales Artificiales	18
2.3.1.	Aprendizaje Automático	18
2.3.1.1.	Aprendizaje supervisado	19
2.3.1.2.	Aprendizaje no supervisado	19
2.3.2.	Aprendizaje Profundo	21
2.3.2.1.	Redes neuronales feedforward	21
2.3.2.2.	Funcion de costo	24
2.3.2.3.	Entrenamiento usando gradientes y retropropagación	25
2.3.2.4.	Funciones de activación	28
2.3.2.5.	Métricas y puntajes para tareas de regresión	29
2.3.2.6.	Diseño de Arquitectura e hiperparámetros de una red neuronal	31
2.3.3.	Redes Neuronales Convolucionales	32
2.3.4.	Operación de convolución	32
2.3.5.	Operación de max pooling	33
2.3.5.1.	Procesamiento de imágenes con redes neuronales convolucionales	34
2.3.5.2.	Aprendizaje de representaciones internas	34
2.3.6.	Sistemas de Aprendizaje Fin a Fin	36
2.4.	Robots móviles y locomoción con ruedas	37
2.4.1.	Modelo de locomoción de Ackerman	38
2.4.1.1.	Cinemática directa	38
2.4.2.	Modelo de la bicicleta o triciclo	39
2.4.2.1.	Cinemática directa	39
3.	Ingeniería del proyecto	41
3.1.	Arquitectura del sistema	41
3.1.1.	Visión general	41
3.1.2.	Esquema del sistema	42
3.1.3.	Subsistema de control y actuación	42
3.1.4.	Subsistema de adquisición de datos y entrenamiento	43
3.1.5.	Subsistema de inferencia y control autónomo	44
3.2.	Herramientas de software	46
3.2.1.	Robot Operating System - ROS	46
3.2.1.1.	Infraestructura de comunicación	46
3.2.1.2.	Nodos de ROS	47
3.2.1.3.	Características específicas para robótica	47
3.2.1.4.	Herramientas adicionales	48
3.2.1.5.	Criterios de selección	49

3.2.2. Tensorflow	51
3.2.3. Keras	53
3.2.4. ARM Mbed	54
3.3. Herramientas de hardware	55
3.3.1. Plataforma de tiempo real	55
3.3.2. Computadora de Abordo - OBC	59
3.3.3. Estación de trabajo	60
3.3.4. Sensores	62
3.3.4.1. Cámara	62
3.3.4.2. Sensor de proximidad	64
3.4. Subsistema de Control y actuación	65
3.4.1. Módulo de potencia en tiempo real	66
3.4.1.1. Control de actuadores mediante ROS	68
3.4.2. Módulo de la computadora de abordo	69
3.4.2.1. Esquema de comunicación en la OBC	70
3.4.2.2. Nodo /serial_node	71
3.4.2.3. Nodo /laser	72
3.4.2.4. Nodo /camera	72
3.5. Subsistema de Adquisición de Datos y Entrenamiento	72
3.5.1. Módulo de adquisición de datos y operación manual	72
3.5.1.1. Sesiones de entrenamiento	73
3.5.2. Módulo de aumentación de datos	75
3.5.3. Módulo de generación de datos de entrenamiento	77
3.5.4. Módulo de Entrenamiento	77
3.6. Subsistema de Inferencia y control autónomo	79
3.6.1. Módulo de inferencia con una red neuronal convolucional	79
3.6.2. Módulo de detección de obstáculos	81
3.6.3. Módulo del piloto automático	82
3.7. Diseño de la arquitectura de la red neuronal	84
3.7.1. Requerimientos	84
3.7.2. Unidades y profundidad	84
3.7.2.1. Capas Convolucionales y Pooling	84
3.7.2.2. Capas densamente conectadas	85
3.7.3. Funciones de Activación	86
3.7.4. Función de Costo	87
3.7.5. Optimizador	87
3.7.6. Arquitecturas implementadas	88
3.7.6.1. Red neuronal tradicional	88
3.7.6.2. Red neuronal convolucional	88
3.8. Proceso de Entrenamiento de la	
Red neuronal Convolutacional	90
3.8.1. Caracterización del conjunto de datos	90

3.8.1.1. Balanceo del conjunto de datos	91
3.8.2. Separación de conjuntos de entrenamiento, pruebas y validación	94
3.8.2.1. Conjunto de entrenamiento	94
3.8.2.2. Conjunto de validación	95
3.8.2.3. Conjunto de prueba	96
4. Análisis y discusión de resultados	97
4.1. Proceso de entrenamiento	97
4.1.1. Análisis de las curvas de entrenamiento	97
4.1.1.1. Error de entrenamiento	97
4.1.1.2. Error de validación	98
4.1.2. Puntajes y errores en el conjunto de prueba	100
4.2. Pruebas	101
4.2.1. Análisis del rendimiento en pruebas de campo	101
4.2.2. Análisis de representaciones internas	102
4.2.2.1. Giro a la izquierda	103
4.2.2.2. Giro a la derecha	103
4.2.2.3. Conducir hacia adelante	103
5. Conclusiones y recomendaciones	107
5.1. Conclusiones	107
5.2. Recomendaciones	109
Bibliografía	111
Apéndice	114
A. Función sigmoide	115
B. Código Fuente del Proyecto	117
B.1. Control de Actuadores en Tiempo Real	117
B.2. Script de entrenamiento	118
B.3. Sincronización de Mensajes	121
B.4. Aumentación de datos	124
B.5. Generación de datos	126
B.6. Nodo de Inferencia	128
B.7. Nodo de detección de obstáculos	130
B.8. Piloto Automático	132
B.9. Control teleoperado con Joystick	135
B.10. Nodo del sensor de proximidad	137
C. Hojas de datos	140

Índice de figuras

1.1.	Stanley	2
1.2.	Niveles de automatización SAE	2
1.3.	Vehículo de Waymo	3
1.4.	Arquitectura de un sistema de conducción autónoma	4
1.5.	Inferencia y control autónomo tradicional	6
2.1.	Vehículos del <i>Grand Challenge</i>	11
2.2.	Visualización de los datos provenientes de un LIDAR	14
2.3.	Algunas aplicaciones de la visión artificial	16
2.4.	Obtención de una imagen con una cámara digital	17
2.5.	Representación de una imagen digital	18
2.6.	Regresión lineal	20
2.7.	Clustering	21
2.8.	Ilustración de un modelo de aprendizaje profundo	22
2.9.	Diagrama de una red neuronal de dos capas	24
2.10.	Visualización de la función de error	26
2.11.	Gráfico de la función tangente hiperbólica	29
2.12.	Gráfico de la función ReLU	30
2.13.	Gráfico de la operación max pooling	34
2.14.	Visualización de la operación de convolución sobre una imagen digital	35
2.15.	Kernels convolucionales de tamaño	36
2.16.	Centro instantáneo de curvatura	38
2.17.	Modelo cinemático de Ackerman	39
2.18.	Modelo cinemático del triciclo	40
3.1.	Esquema en diagrama de bloques del sistema	42
3.2.	Subsistema de control y actuación	43
3.3.	Subsistema de Adquisición de datos y Entrenamiento	44
3.4.	Subsistema de Inferencia y Control Autónomo	45
3.5.	Diagrama de comunicación de nodos	46
3.6.	Esquema de la comunicación entre nodos	47
3.7.	Interfaz de visualización de ROS rviz	48
3.8.	Ejemplo de un grafo de cómputo utilizado en Tensorflow	51

3.9. Placa de desarrollo Nucleof303k8 de ST Microelectronics	57
3.10. Placa de desarrollo Raspberry Pi 3 model B+	60
3.11. Raspberry Pi Camera Module V2	63
3.12. Conexión de la Camera Module V2	63
3.13. Sensor VL53L0X de ST Microelectronics	64
3.14. Ilustración del principio ToF	65
3.15. Breakout Board para el sensor VL53L0X	66
3.16. Prototipo del vehículo	67
3.17. Fotografía del servomotor de dirección	67
3.18. Fotografía del motor DC de aceleración	68
3.19. Visualización del nodo serial	69
3.20. Fotografía de la OBC	70
3.21. Fotografía del prototipo	70
3.22. Interacción de los nodos en la OBC	71
3.23. Fotografía del mando inalámbrico de Xbox 360	73
3.24. Esquema de nodos para la adquisición de datos	73
3.25. Imágenes capturadas por la cámara en una sesión de entrenamiento	75
3.26. Esquema del módulo de inferencia	80
3.27. Esquema del módulo de detección de obstáculos	82
3.28. Esquema de los nodos del sistema en modo autónomo	82
3.29. Visualización de una capa convolucional	85
3.30. Visualización de una capa de max pooling	86
3.31. Visualización de una capa densamente conectada	87
3.32. Arquitectura de la red neuronal densamente conectada	90
3.33. Arquitectura de la red neuronal convolucional	92
3.34. Overfitting en modelos de aprendizaje	93
3.35. Distribución inicial del dataset	93
3.36. Distribución final del dataset	94
3.37. Ejemplos de aumentación de datos	95
3.38. Separación del conjunto de datos	96
4.1. Error de entrenamiento	98
4.2. Error de validación	99
4.3. Ejemplos de predicción de la red convolucional	101
4.4. Filtros de la primera capa convolucional	102
4.5. Filtros de la penúltima capa convolucional	102
4.6. Activaciones para una imagen con giro a la izquierda	104
4.7. Activaciones para una imagen con giro a la derecha	105
4.8. Activaciones para una imagen con dirección hacia adelante	106
A.1. Gráfico de la función sigmoide	115

Índice de tablas

2.1. Niveles de automatización según SAE.	13
3.1. Tabla comparativa de librerías y plataformas para sistemas robóticos.	50
3.2. Tabla comparativa de plataformas de desarrollo embebido.	55
3.3. Comparación de características de microcontroladores disponibles.	58
3.4. Características técnicas del microcontrolador STM32f303k8.	58
3.5. Tabla comparativa de SBC's disponibles en el mercado.	59
3.6. Características de la placa Raspberry Pi 3 model B+.	61
3.7. Características de la estación de trabajo seleccionada.	61
3.8. Características de la Raspberry Pi Camera Module V2.	62
3.9. Características del sensor VL53L0X.	65
3.10. Nodos de interfaz con los actuadores y sensores	71
3.11. Columnas de la tabla del conjunto de datos generado	74
3.12. Transformaciones realizadas en la aumentación de datos.	76
3.13. Funciones de activación usadas	86
3.14. Arquitectura de la red neuronal densamente conectada.	89
3.15. Arquitectura de la red neuronal convolucional.	91
3.16. Cantidad de muestras en los conjuntos de datos.	95
4.1. Métricas para la evaluación del modelo.	100
4.2. Evaluación de puntajes sobre el conjunto de prueba.	100
4.3. Muestra de predicciones de la red convolucional.	100

Capítulo 1

Introducción

1.1. Antecedentes

El primer intento de desarrollo de un sistema de conducción autónomo “fin a fin” fue llevado a cabo por la Agencia de Proyectos de Investigación Avanzada en Defensa de los Estados Unidos (DARPA) con un proyecto conocido como el Vehículo Autónomo de DARPA o DAVE [1] en el cual un vehículo radio controlado a escala tenía la tarea de conducir a través de un entorno escabroso. El vehículo DAVE fue entrenado a partir de cientos de horas de conducción humana en entornos similares pero no idénticos. Los datos de entrenamiento incluyeron imágenes de dos cámaras de video y comandos de control generados por un operador humano.

Paralelamente a este esfuerzo realizado por el DARPA y debido a la limitada capacidad computacional de la época, los avances en las distintas tareas que componen la conducción autónoma se han enfocado en el tratamiento de las señales y datos provenientes de los sensores con algoritmos de procesamiento básicos llegando a crearse implementaciones efectivas basadas en un flujo de trabajo descrito a continuación.

1.1.1. Sistemas de Conducción Autónoma

Un sistema de conducción autónoma es una combinación de varios componentes o subsistemas donde las tareas de percepción, toma de decisiones y operación de un vehículo son desarrolladas por un sistema electrónico en lugar de un conductor humano.

El primer hito en el desarrollo de un sistema completamente autónomo vino con la organización del DARPA “Grand Challenge” en el cual equipos de varias universidades, institutos de investigación y empresas tuvieron que enfrentar el difícil reto de desarrollar un sistema capaz de controlar un vehículo doméstico a través de una carretera ripiada en medio del desierto de Arizona. Dentro las 2 versiones del Darpa Grand Challenge destacaron los proyectos de universidades como Stanforad con el robot Stanley [2] que fue el primer vehículo en recorrer mas de 170 kilómetros en una carretera ripiada de manera completamente autónoma.

El éxito de los proyectos que participaron en el grand challenge sentó un gran precedente



Figura 1.1: Stanley, el vehículo autónomo de Stanford que ganó la competencia DARPA Grand Challenge en 2005. Fuente: stanford.edu

en el desarrollo de lo que ahora se conoce como *Self Driving Car* o vehículo autónomo. De hecho, muchos de los equipos participantes de este concurso se constituyen en la actualidad como existosas empresas de desarrollo o coadyuvan en iniciativas privadas de gigantes de la tecnología como Google, Uber o Nissan.

Sin embargo, debido al creciente interés tanto en investigación como económico en los sistemas de conducción autónoma, la Sociedad de Ingenieros en Automoción (SAE por sus siglas en inglés) ha elaborado un estándar donde se detallan distintos aspectos concernientes. La regulación define varios niveles de autonomía en vehículos terrestres, aéreos y acuáticos yendo desde un control completamente manual, normalmente observado en vehículos completamente mecánicos, pasando por asistencias al control hasta llegar a un vehículo completamente autónomo en todas sus tareas

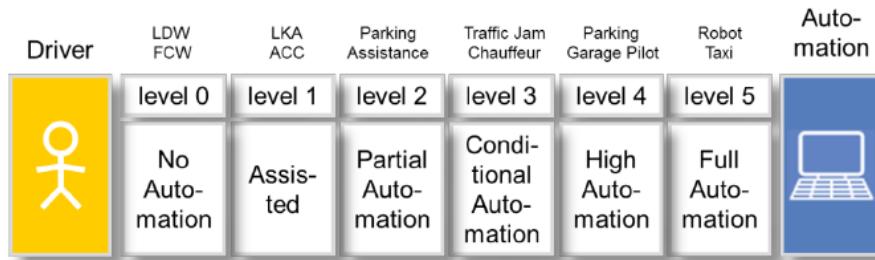


Figura 1.2: Niveles de automatización en la conducción según SAE. Fuente: researchgate

La creación de estándares y regulación ha tenido como consecuencia que, en la actualidad, existan varias iniciativas en el desarrollo de los *self Driving Cars*, siendo una de las más importantes la empresa Waymo, dependiente de Google a través de su empresa Pública Alphabet. Waymo, ha aprovechado el uso de tecnologías emergentes de sensado como el LIDAR para mejorar el mapeo y la navegación a través de algoritmos de fusión de sensores. Aparte de Alphabet, existen diversas iniciativas privadas en el desarrollo de vehículos autónomos

con fines comerciales como los Self Driving Cars de Uber, Toyota, BMW, Ford, entre otros.



Figura 1.3: El vehículo autónomo de Waymo. waymo.com

Una de las tareas más importantes dentro de un *self driving car* es la detección y mantenimiento del carril del vehículo. Fabricantes de vehículos automotores han incluido con éxito sistemas de asistencia al conductor para la mantención del carril usando cámaras digitales y visión artificial para poder detectar la posición del automóvil con respecto al carril. Estos sistemas se consideran fundamentales en sistemas de conducción autónoma. Durante las últimas dos décadas, se han desarrollado distintos tipos de sistemas y enfoques para resolver el problema de la mantención de carril.

1.1.1.1. Arquitectura general de un sistema de conducción autónoma

En general, la arquitectura de un sistema de conducción autónoma se puede entender como la integración de varios módulos o subsistemas funcionales que operan en coordinación tal como se puede observar en la Figura(1.4).

Normalmente, este tipo de sistemas cuenta con una etapa de adquisición de datos y entrenamiento que servirá para alimentar una base de conocimiento o reglas en las que se basará el módulo de inferencia y control autónomo. Asimismo, tanto el subsistema de adquisición de datos y entrenamiento como el subsistema de inferencia y control autónomo interactúan directamente con el subsistema de control y actuación del vehículo.

Los mayores esfuerzos se han enfocado principalmente al desarrollo del subsistema de inferencia y control autónomo ya que es el que define el rendimiento de un sistema de conducción autónoma en sí. Este subsistema, a su vez, puede ser analizado como un conjunto de varios módulos que interactúan entre sí de manera secuencial, como se puede apreciar en la Figura(1.5).

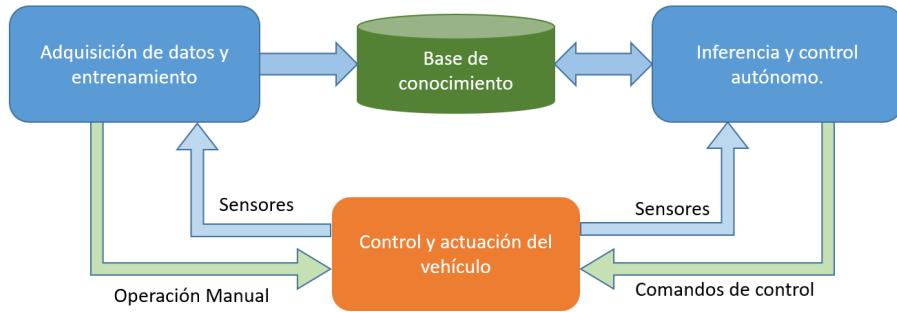


Figura 1.4: Arquitectura de un sistema de conducción autónoma. Fuente: Elaboración propia.

1.1.1.2. Sistemas de aprendizaje fin a fin

Tradicionalmente, los sistemas de aprendizaje requieren de múltiples etapas de procesamiento que interactúan entre sí, como se muestra en la Figura(1.5). Por su parte, los sistemas de aprendizaje fin a fin intentan condensar estas etapas de procesamiento y reemplazarlas usualmente con una sola red neuronal. Estos sistemas han demostrado ser altamente efectivos en contraste a los enfoques tradicionales, principalmente porque abstraen y resumen el diseño de las etapas intermedias de un sistema de aprendizaje tradicional con una sola etapa. La desventaja de los sistemas de aprendizaje fin a fin radica en la necesidad de grandes cantidades de datos de entrenamiento en comparación con los enfoques tradicionales, sin embargo, gracias a la gran disponibilidad de datos de entrenamiento y la accesibilidad de instrumentos y herramientas de adquisición de datos, esta desventaja no representa una dificultad de gran magnitud en el desarrollo de sistemas fin a fin.

Los sistemas de aprendizaje fin a fin se han explorado de manera exitosa en los últimos años, esto debido a la creciente disponibilidad de sistemas de cómputo de alta concurrencia, en especial las Unidades de Procesamiento Gráfico de propósito general o GPGPU por sus siglas en inglés. Esta disponibilidad ha logrado que se puedan entrenar redes neuronales completas en una estación de trabajo que no consume demasiada energía. Una de las empresas pioneras en GPGPU es Nvidia con su herramienta CUDA, que ha permitido el desarrollo de algoritmos de entrenamiento e inferencia para redes neuronales de manera sencilla. Es precisamente Nvidia que ha demostrado que los sistemas de aprendizaje fin a fin pueden tener éxito con el desarrollo de un prototipo y arquitectura de vehículo autónomo [3].

1.2. Justificación del Proyecto

1.2.1. Justificación académica

Desde el punto de vista académico, el proyecto se justifica en el entendido del uso de técnicas y procedimientos de ingeniería para el análisis y diseño de un sistema de aprendizaje

“fin a fin” usando redes neuronales y una plataforma para el entrenamiento y despliegue del mismo. Tales técnicas y procedimientos incluyen la definición de la arquitectura de la red neuronal, el entrenamiento y el análisis del rendimiento de la misma. Así como también el dimensionamiento de los componentes de cómputo embebido para el prototipo y la implementación de los sistemas de control electrónico de bajo nivel para el mismo. Tales técnicas y procedimientos se corresponden de manera integral con los conocimientos adquiridos a lo largo de la carrera de Ingeniería Electrónica en sus distintas asignaturas.

1.2.2. Justificación tecnológica

Dada la creciente relevancia de los sistemas autónomos en la actualidad, el proyecto se justifica desde el punto de vista tecnológico dado que se presenta la aplicación de nuevas herramientas y plataformas de software para el desarrollo de sistemas de autónomos, visión por computador y redes neuronales convolucionales, que representan áreas vigentes en la investigación tecnológica hoy en día.

El sistema desarrollado se constituirá a su vez en una plataforma de desarrollo sobre el cual se podrá extender su funcionalidad y mejorar sus resultados usando herramientas de software de fácil acceso y aprendizaje presentando la posibilidad de continuar y extender la investigación en sistemas de conducción autónoma, robótica móvil, visión por computador y aprendizaje profundo.

1.2.3. Justificación técnica

Desde el punto de vista de las técnicas aplicadas, el proyecto se justifica dado que se pretende presentar una técnica alternativa al enfoque tradicional en el desarrollo de sistemas de aprendizaje, presentando el desarrollo de un sistema de aprendizaje fin a fin, que facilitará su análisis, diseño, entrenamiento y puesta en marcha en futuros proyectos de investigación y aplicaciones en distintas áreas de la ingeniería.

La propuesta de la nueva técnica de aprendizaje fin a fin representa un avance en relación al desarrollo de sistemas tradicionales por su impacto en el requerimiento de recursos y de conocimiento específico requerido.

1.3. Análisis de la problemática y planteamiento del problema

1.3.1. Análisis de la problemática

Se han estudiado diferentes enfoques para lograr solucionar la tarea de conducción autónoma para vehículos domésticos usando sistemas de aprendizaje. Normalmente, la salida del sistema se expresa como una serie de comandos de control de aceleración y dirección del volante del vehículo. Estos comandos se pueden obtener de diversas maneras dependiendo el nivel de robustez y abstracción que el sistema requiere. Muchos sistemas se basan en la fusión

de distintos tipos de sensores y fuentes de información como ser mapas satelitales, GPS, sensores láser y cámaras. La combinación de esta información es procesada y fusionada mediante distintos algoritmos de filtrado tales como el filtro de kalman. La característica de este tipo de sistema es que se puede expresar como una serie de etapas de procesamiento mediante el cual la información fluye y se transforma, cada una de las etapas es diseñada e implementada en base a conocimiento específico y con requerimientos y limitaciones específicas de la tarea que realiza tal como se puede apreciar en la Figura(1.5).

Si bien el enfoque anteriormente mencionado ha logrado conseguir importantes avances y resultados muy prometedores, involucra un gran esfuerzo a la hora de diseñar cada una de las etapas independientemente para luego hacer que funcionen todas juntas y cumplan la tarea asignada. Este proceso usualmente requiere de un equipo de expertos que sea capaz de realizar las tareas de diseño de las etapas o módulos del sistema y el de la integración de los módulos en un solo sistema funcional. Este enfoque, pese a que ha demostrado ser una forma efectiva de trabajo para diversos problemas, tiene la desventaja de requerir muchos recursos y tiempo para poder lograr un sistema funcional.

1.3.2. Planteamiento del problema

De acuerdo a lo establecido anteriormente, se puede considerar a la etapa de inferencia y control autónomo de un sistema de conducción autónoma como un sistema de procesamiento de información que consta de varias etapas secuenciales que transforman la información de acuerdo a parámetros previamente establecidos. Se debe tener en cuenta varios aspectos concernientes tanto al diseño como a la implementación de dichos tipos de sistemas.

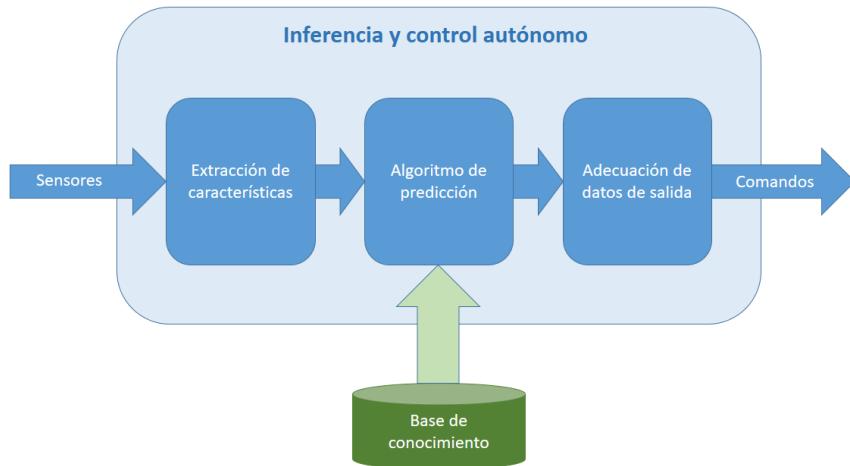


Figura 1.5: Componentes del subsistema de inferencia y control autónomo tradicional. Fuente: Elaboración propia.

En el área de visión por computadora para tareas de conducción autónoma, normalmente

se sigue el siguiente flujo en el desarrollo un sistema o prototipo:

1. **Extracción de características.** Esta etapa incluye el preprocesamiento y transformación de la imagen en un conjunto de características de distinta índole. Estas características se suelen llamar también descriptores y sirven para describir los aspectos más relevantes de la imagen para la tarea final, por ejemplo, la detección de bordes. La extracción de características también se usa para reducir la dimensionalidad inicial de la imagen a una más tratable y amigable con la capacidad de procesamiento computacional disponible. Las características o descriptores a usarse se definen manualmente por medio de conocimiento experto y se afinan de la misma manera.
2. **Algoritmo de predicción.** Esta etapa incluye típicamente un algoritmo de aprendizaje previamente entrenado con un conjunto de datos adecuado, permite realizar distintas tareas de alto nivel sobre los descriptores obtenidos de la imagen. Estas descripciones de alto nivel incluyen normalmente tareas de detección, clasificación o regresión. Los algoritmos de aprendizaje incluyen típicamente algoritmos básicos, tales como árboles de decisión, regresión lineal o máquinas de soporte vectorial ya que deben realizar la tarea de predicción en un conjunto de dimensionalidad relativamente baja (los descriptores).
3. **Adecuación de los datos de salida.** La información extraída de la anterior etapa debe procesarse para poder ser traducida a comandos de control que actúen directamente con las etapas de bajo nivel del vehículo, es decir la etapa de actuación y potencia. En esta etapa se suele incluir algún algoritmo de control realimentado para el control de motores así como también algoritmos de fusión de distintas fuentes de información para obtener finalmente una señal de comando para los actuadores.

Como se ha podido observar, el flujo de trabajo en un sistema de conducción autónomo se compone de varias etapas secuenciales que se deben realizar con conocimiento y experiencia específica en cada una de las mismas.

Por su parte, otra de las dificultades con este acercamiento, al reto de la conducción autónoma es el de la reducida flexibilidad del sistema. En otras palabras, si se quisiera modificar el sistema para agregar requerimientos o expandir la funcionalidad del mismo, se debe realizar una modificación a la etapa específica y evaluar el impacto de las modificaciones en todo el sistema en su conjunto. Esto dificulta de manera sustancial la reutilización de diversos componentes en sistemas similares.

Finalmente, la exagerada complejidad y conocimientos requeridos para poder implementar un sistema experimental de esta naturaleza hace prácticamente imposible su desarrollo por equipos de investigación pequeños o investigadores individuales. Dada la importancia y la potencialidad de los sistemas de conducción autónoma es esencial reducir esta dificultad de implementación y experimentación.

En conclusión, el desarrollo de un sistema de conducción autónoma presenta tres principales dificultades a la hora de ser abordado:

1. Conocimiento experto de cada una de las etapas involucradas en el sistema.

2. Poca flexibilidad en el diseño y la implementación del sistema una vez establecido y probado.
3. El tiempo y recursos necesarios para poder diseñar e implementar un sistema de tal naturaleza lo hace privativo para equipos de investigación pequeños o con poco presupuesto.

1.4. Objetivos

1.4.1. Objetivo General

Diseñar un sistema de aprendizaje “fin a fin” capaz de generar de comandos de control de vehículos domésticos basado en visión artificial y redes neuronales convolucionales para facilitar el diseño e implementación de sistemas de conducción autónoma.

1.4.2. Objetivos Específicos

Para alcanzar el objetivo general será necesario:

- Estudiar los aspectos concernientes al desarrollo de sistemas de conducción autónoma y sistemas de aprendizaje.
- Analizar los requerimientos de un sistema de conducción autónoma capaz identificar y mantener su carril mientras se conduce.
- Diseñar la arquitectura de un sistema de conducción autónoma en base a los requerimientos previamente establecidos.
- Diseñar el subsistema de control y actuación para la conducción autónoma de un vehículo con características similares a las de un vehículo doméstico real.
- Diseñar el subsistema de adquisición de datos y entrenamiento para tareas de conducción autónoma.
- Diseñar el subsistema de inferencia y control autónomo basado en el uso de redes neuronales convolucionales.
- Analizar los resultados del entrenamiento e implementación del subsistema de inferencia y control autónomo.
- Realizar pruebas de rendimiento y análisis comparativos en el sistema implementado.

1.5. Alcance

El presente proyecto de grado cubre los siguientes aspectos dentro de su alcance:

- El enfoque del estudio de sistemas de conducción autónomos de vehículos terrestres con el modelo de Ackermann.
- Investigación de arquitecturas y plataformas de software para el diseño y despliegue de robots móviles y tareas de conducción autónoma.
- El desarrollo del sistema se contempla en el marco de un proyecto académico y, por tanto, será implementado usando herramientas de software comúnmente utilizadas en investigación de sistemas autónomos.

El alcance detallado previamente estará acotado a su vez por una serie de supuestos.

1.6. Límites

El sistema, por su parte, contará con ciertas restricciones detalladas a continuación:

- La tarea de conducción autónoma estará enfocada exclusivamente al seguimiento y mantención del carril basado en imágenes provenientes de una cámara sin considerar el reconocimiento e interpretación de otro tipo de información como señales de tránsito, cruces e intersecciones o la presencia de peatones, ciclistas, animales y otros objetos en la ruta.
- El prototipo a escala servirá solamente para un análisis superficial de la dinámica de un vehículo automotor doméstico tomando como punto de inicio modelos matemáticos simplificados y limitaciones de rangos de trabajo dentro de dichos modelos.
- El diseño de la arquitectura de la red neuronal estará orientado a tareas de aprendizaje supervisado y aproximación de funciones y limitado por la capacidad de procesamiento disponible en el momento de la realización del presente proyecto.

Capítulo 2

Marco Teórico

2.1. Sistemas de Conducción Autónoma

Un sistema de conducción autónoma es una combinación de varios componentes o subsistemas donde las tareas de percepción, toma de decisiones y operación de un vehículo son desarrolladas por un sistema electrónico en lugar de un conductor humano. Usualmente, un sistema de conducción autónoma incluye varios subsistemas de automatización que operan de manera conjunta y coordinada para poder tomar el control total o parcial del vehículo.

En algunas ocasiones, la automía del control se implementa de manera condicional, es decir, que el sistema toma el control del vehículo para ciertas situaciones pero no todo el tiempo como por ejemplo sistemas de estabilización de frenos o prevención de impactos. Este tipo de sistemas se ha ido desarrollando e implementando en vehículos comerciales de manera paulatina pero todavía no existe un vehículo completamente autónomo circulando por las calles o carreteras. Notese que los términos autonomía y automatización se usan de manera intercambiable en este contexto.

Se han desarrollado tres enfoques concernientes al desarrollo de vehículos inteligentes de acuerdo al nivel de autonomía que presentan estos sistemas:

1. **Enfoques centrados en el conductor.** Se diseñan en base a la idea de tener a un humano en el lazo de control supervisando las funciones del vehículo
2. **Enfoques centrados en una red.** Se diseñan con la idea de que los vehículos inteligentes puedan compartir información entre sí en una infraestructura de red.
3. **Enfoques centrados en el vehículo.** Tienen el objetivo de desarrollar vehículos inteligentes completamente autónomos, sin la necesidad de la intervención de un humano en el lazo de control.

El interés en el desarrollo de vehículos completamente autónomos ha despertado el interés tanto de investigadores en todo el mundo así como también de instituciones privadas y gubernamentales que han invertido esfuerzos en fomentar éste desarrollo. Una de las primeras instituciones en materializar una iniciativa en el desarrollo de vehículos autónomos fue la

Agencia de Proyectos de Investigación Avanzados (DARPA, por sus siglas en inglés) con el lanzamiento del concurso *Grand Challenge* en el año 2003 que era una carrera de vehículos autónomos que debían viajar por más de 200 Km en terrenos escabrosos. Este concurso sin precedentes atrajo la atención de un número de instituciones de investigación de alto nivel, lo que permitió que el desarrollo de este tipo de vehículos diera un gran paso adelante.

En 2005, una nueva versión del DARPA *Grand Challenge* requirió que los vehículos condujeran por una carretera en un desierto. En esta oportunidad, cinco vehículos completaron el trayecto de 211 Km en el desierto, con el vehículo de la Universidad de Stanford *Stanley* como ganador, habiendo recorrido todo el trayecto en 6 horas con 54 minutos. En la Figura(2.1), se pueden apreciar fotografías de los vehículos con mejor desempeño en la competencia.

El siguiente hito en el desarrollo de vehículos autónomos, fue el desarrollo del concurso organizado por DARPA *Urban Challenge* en el año 2007, en el cual, los vehículos debían cumplir con ciertas tareas de navegación pero esta vez en un entorno urbano simulado, con avenidas, intersecciones, señalización y otros vehículos circulando simultáneamente. El recorrido comprendía aproximadamente 90 Km y representaba un reto por las complejas tareas de decisiones y comportamientos en un entorno de tráfico similar al que se puede encontrar cualquier conductor humano conduciendo en una ciudad. En esta ocasión, el ganador de la competencia fue el equipo Tartan Racing, un esfuerzo conjunto entre la Carnegie Mellon University y General Motors Corporation con el vehículo *Boss*, una versión modificada de un Chevrolet Tahoe.



Figura 2.1: Vehículos del *Grand Challenge* de 2005: (a) Stanley (1er lugar), (b) Sandstorm (2do lugar). Vehículos del *Urban Challenge* de 2007: (c) Boss (1er lugar) (d) Junior (2do lugar). Fuente: [4]

2.1.1. Niveles de Autonomía SAE

Debido al creciente interés e inversión en el desarrollo de sistemas de conducción autónoma se ha establecido una manera de categorizar los niveles de automatización de la conducción por parte de la Sociedad de Ingenieros en Automoción (SAE, por sus siglas en inglés) en la que se definen seis niveles de automatización en vehículos terrestres, acuáticos y aéreos.

Nivel 0: Sin automatización El conductor está en completo control de todas las funciones del vehículo en todo momento, no existe intervención de ningún sistema automatizado en el control. Sistemas de alerta de colisión o pérdida de carril entran en esta categoría.

Nivel 1: Conducción asistida El conductor tiene el control del vehículo, pero el sistema puede modificar la aceleración o dirección del mismo. Los sistemas de control de velocidad de crucero caen en esta categoría.

Nivel 2: Automatización parcial El conductor debe poder ser capaz de tomar el control del vehículo si ciertas se necesitan ciertas correcciones, pero ya no está en control de la aceleración y dirección del vehículo directamente. Es importante resaltar que desde los niveles 0 al 2 el conductor no puede estar distraído en ningún momento de la conducción. Los sistemas de parqueo automático representan un buen ejemplo de sistemas de Nivel 2.

Nivel 3: Automatización condicional El sistema automatizado tiene el control del vehículo, tanto de la aceleración, dirección así como también del monitoreo del entorno bajo condiciones específicas. El conductor debe estar preparado para intervenir cuando el sistema así lo requiera, por tanto, se permiten distracciones ocasionales. Uno de los sistemas recientemente implementados que cae en esta categoría es el sistema *autopilot* de los vehículos de Tesla Motors.

Nivel 4: Automatización elevada El sistema está en completo control del vehículo y la presencia humana ya no es necesaria, sin embargo, la operación autónoma del vehículo está limitada a condiciones específicas. Si las actuales condiciones del entorno sobrepasan las fronteras de rendimiento definidas, el vehículo puede desplegar un protocolo o secuencia de emergencia. Actualmente el desarrollo de vehículos autónomos o *self driving cars* se enfoca en este nivel.

Nivel 5: Automatización completa El sistema está en completo control del vehículo y la presencia humana no es necesaria en absoluto. El sistema es capaz de proveer las mismas características que en el Nivel 4, pero en esta ocasión puede operar al vehículo en todas las condiciones. En este nivel, el conductor pasa a ser un pasajero en el vehículo. Actualmente, no existen sistemas que operen en este nivel.

La relación entre la responsabilidad del sistema y el conductor en los distintos niveles se puede apreciar en la Tabla 2.1:

2.1.2. Tecnologías requeridas

Las tecnologías básicas para sensado y actuación para vehículos autónomos ya existen en el mercado. El reto clave es el de integrar dichas tecnologías con nuevos desarrollos en un sistema estable. Para poder desarrollar sistemas de conducción autónoma confiables y seguros se necesita lo siguiente:

Nivel SAE	Denominación	Ejecución de aceleración y dirección	Monitoreo del entorno	Responsable en condiciones difíciles	Modos de conducción
0	Sin Automatización	Humano			Ninguno
1	Conducción asistida	Humano y sistema	Humano	Humano	
2	Automatización parcial				Algunos Modos
3	Automatización condicional	Sistema			
4	Automatización elevada		Sistema	Sistema	Varios Modos
5	Automatización completa				Todos los Modos

Tabla 2.1: Niveles de automatización según SAE. Fuente: SAE

- Una forma de medir o estimar el estado del vehículo.
- Una forma de medir o estimar el estado del entorno en el que el vehículo circula.
- Acceso a mapas e información satelital.
- Una infraestructura de comunicación distribuida con otros vehículos.

2.1.2.1. Estado del vehículo

La localización del vehículo es fundamental para tareas de conducción autónoma y debe ser conocida si se desea que siga una trayectoria definida. Para controlar un vehículo de forma que sea capaz de evadir obstáculos o mantener un carril, se necesita conocimiento del estado cinemático y dinámico del mismo. Diversas fuentes de información y datos sobre el estado del vehículo pueden ser incorporados como encoders, o unidades iniciales en combinación con medidas de navegación absolutas como GPS. Adicionalmente, la posición del vehículo puede ser conocida con referencia a marcadores locales. Aparte de la localización del vehículo, sistemas de medición de parámetros internos y control de funciones como aceleración, dirección, estado del motor, y otros, pueden ser incorporados.

El desarrollo de modelos adecuados para el control del vehículo también representa una tarea importante pues es en base a estos modelos sobre los cuales se desarrollarán los algoritmos que lo controlen.

2.1.2.2. Estado del entorno

Otro aspecto crítico en el desarrollo de un sistema autónomo es el sensado del estado del entorno y cómo el mismo evoluciona en el tiempo. La tarea más difícil para un vehículo es poder generar un entendimiento de lo que pasa en la carretera, esto incluye poder detectar marcadores clave, otros vehículos, peatones, la carretera o avenida por la que esta circulando actualmente y otros obstáculos como árboles.

La información se obtiene de diversos tipos de sensores montados en el mismo vehículo, de los cuales los más comunes son los sensores visuales o cámaras que pueden ser posicionadas en distintas orientaciones para poder observar todo lo que ocurre alrededor del vehículo. Los datos de las cámaras son procesados con el objetivo de extraer información valiosa como la

ubicación del carril, detección de peatones y otros obstáculos. Otro tipo de sensores utilizados comúnmente son sensores de proximidad, que permiten obtener medidas de distancia a los obstáculos más cercanos ofreciendo así información al vehículo sobre su posición relativa a los mismos.

Un tipo especial de sensores muy comunes en vehículos autónomos son los LIDAR¹, que generan una nube de puntos con información acerca de la distancia a los obstáculos más cercanos en dos y tres dimensiones.

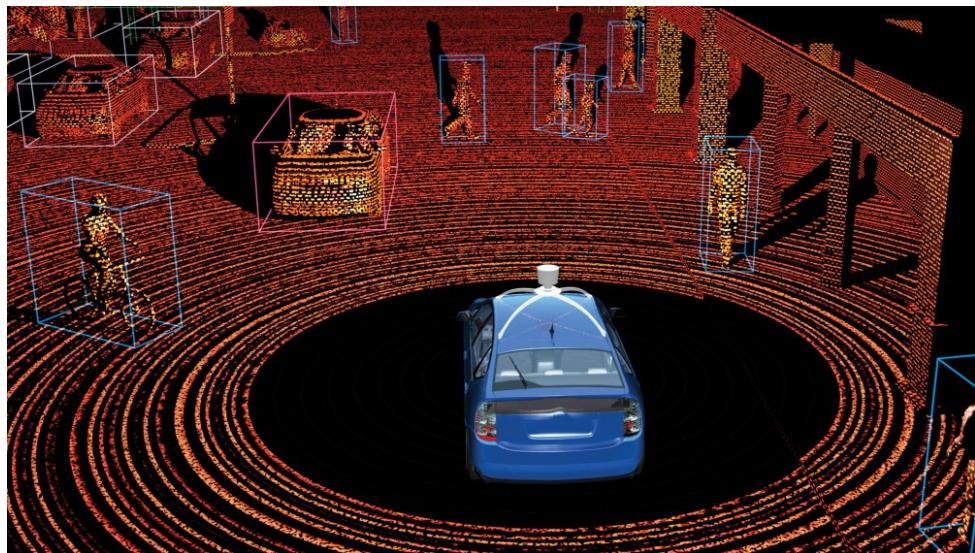


Figura 2.2: Visualización de los datos provenientes de un LIDAR para tareas de conducción autónoma. Fuente: [5]

2.1.3. Arquitectura de un sistema de conducción autónoma

En base a los distintos aspectos detallados anteriormente (Figura(1.4)), en el marco de los objetivos del presente proyecto, se puede describir a la tarea de conducción autónoma usando una arquitectura compuesta por tres elementos o subsistemas fundamentales: el subsistema de control y actuación del vehículo, el subsistema de adquisición de datos y entrenamiento y el subsistema de inferencia y control.

2.1.3.1. Subsistema de control y actuación.

Este subsistema representa la parte física y de bajo nivel del vehículo, el vehículo en sí, sumado a los sistemas de control y sensado incorporados. Este subsistema cuenta con las interfaces necesarias para controlar el vehículo mediante comandos de control de aceleración y dirección y con los medios para extraer información del estado interno del vehículo, así

¹Acrónimo del inglés *Laser Imaging Detection and Ranging*

como también del entorno con cámaras o sensores de proximidad. Por sí solo, este subsistema no es capaz de realizar ninguna tarea de conducción autónoma.

2.1.3.2. Subsistema de adquisición de datos y entrenamiento.

Este subsistema se compone de un conjunto de herramientas y utilidades para la adquisición de conjuntos de datos de entrenamiento y validación, así como también un módulo de entrenamiento de la red neuronal convolucional para la tarea de la generación de comandos de control. En la parte de adquisición de datos, provee de las herramientas necesarias para poder extraer información relevante de sesiones de conducción controlada por un operador humano de forma que se tengan los datos necesarios para que el vehículo pueda aprender la forma de conducir en base a una referencia de conducción humana.

2.1.3.3. Subsistema de inferencia y conducción autónoma.

El subsistema de inferencia y conducción autónoma tiene la tarea de obtener y ejecutar las predicciones del modelo predictivo entrenado en el Subsistema de Adquisición de Datos y Entrenamiento. Este subsistema tomará como entradas los datos de los sensores a bordo del vehículo para generar comandos de control acordes con el entorno percibido. Este subsistema usualmente está implementado en forma de un programa de “piloto automático” capaz de conducir el prototipo de manera autónoma de acuerdo a las limitaciones definidas en el diseño.

2.2. Visión Artificial

La visión es un sentido fundamental en el desarrollo de cualquier persona y, entender el entorno basados en la información que un humano puede ver es relativamente sencillo. Tareas como reconocer la forma y color de un objeto cercano o contar las personas presentes en una fotografía de grupo son tareas triviales para un ser humano. La visión artificial es un campo de las ciencias de la computación que intenta reproducir las capacidades de una persona para entender imágenes. Este campo aglutina técnicas matemáticas para recuperar la forma y apariencia en tres dimensiones de objetos en una imagen. Se trata del desarrollo de distintas técnicas y algoritmos que intentan recuperar la información más importante de una imagen digital, en otras palabras, que una computadora pueda entender una imagen. En la actualidad, las técnicas de visión artificial son usadas en diversas aplicaciones de la vida real de las cuales podemos citar algunas:

- **Reconocimiento óptico de caracteres (OCR):** Lectura de dígitos manuscritos de códigos postales (Figura(2.3a)) y reconocimiento automático de placas de vehículos.
- **Inspección automática:** Inspección de partes para el control de calidad en líneas de producción industriales(Figura(2.3b)).
- **Ventas:** Reconocimiento de objetos para pagos en cajas automatizadas(Figura(2.3c)).

- **Reconstrucción de modelos en 3D (fotogrametría):** Reconstrucción automática de modelos en tres dimensiones a partir de fotografías aéreas.
- **Imagenología médica:** Registro de imágenes pre y post operatorias para estudios y diagnósticos especializados(Figura(2.3d)).
- **Seguridad automotriz:** Detección de obstáculos inesperados como peatones o ciclistas en situaciones donde métodos convencionales de detección de obstáculos no pueden aplicarse (Figura(2.3e)).
- **Seguridad y vigilancia:** Monitoreo de actividad sospechosa y análisis de tráfico en carreteras (Figura(2.3f)).

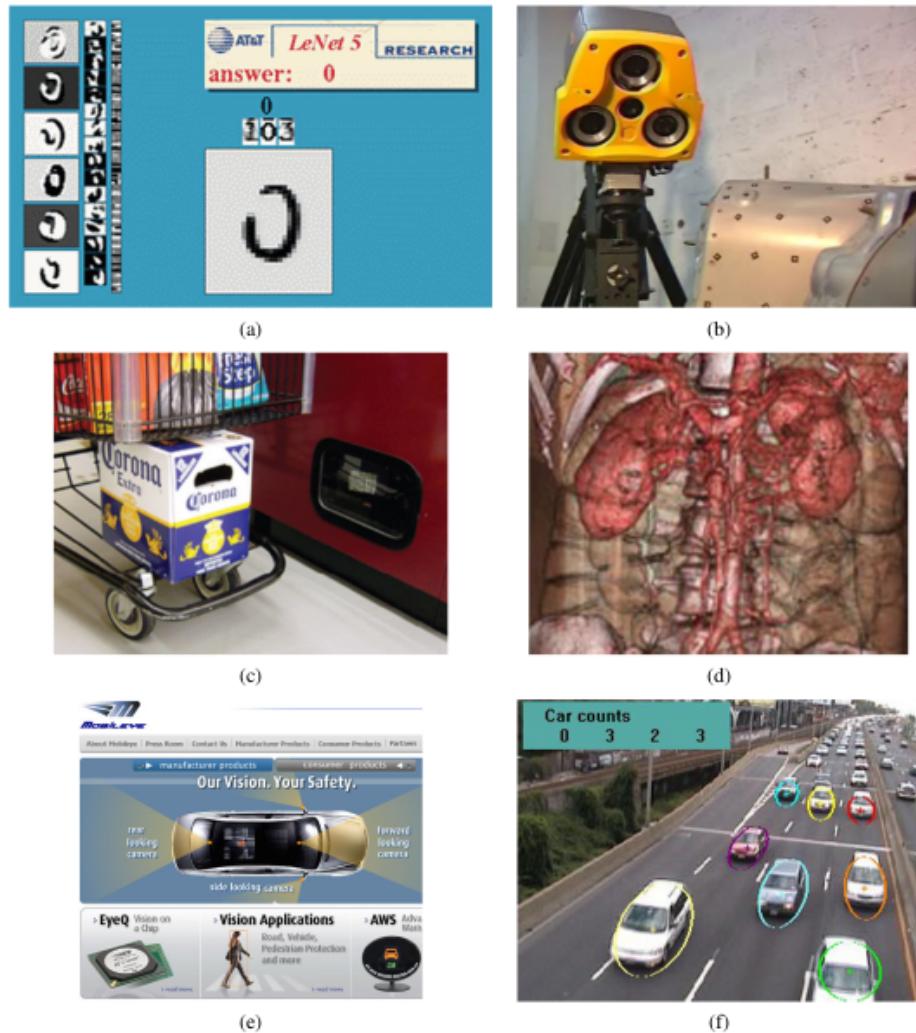


Figura 2.3: Algunas aplicaciones de la visión artificial. Fuente: [6]

2.2.1. Caracterización de una imagen digital

Luego de surgir de una o más fuentes de luz, reflejarse de una o más superficies y pasar a través del lente de una cámara, la luz finalmente llega al sensor óptico. Los fotones incidentes en el sensor son convertidos en valores de intensidad de color rojo, verde y azul (RGB) en un arreglo matricial que es lo que se conoce como una imagen digital. Todas las etapas envueltas en la obtención de una imagen digital se pueden apreciar en la Figura(2.4) donde se diferencian dos subproductos del proceso de conversión: la imagen *raw* y la imagen compresa. Esta diferenciación nace de la necesidad de representar imágenes digitales sin ocupar mucho espacio en la memoria y para poder comprimir una imagen *raw* se realizan varios procesos complementarios.

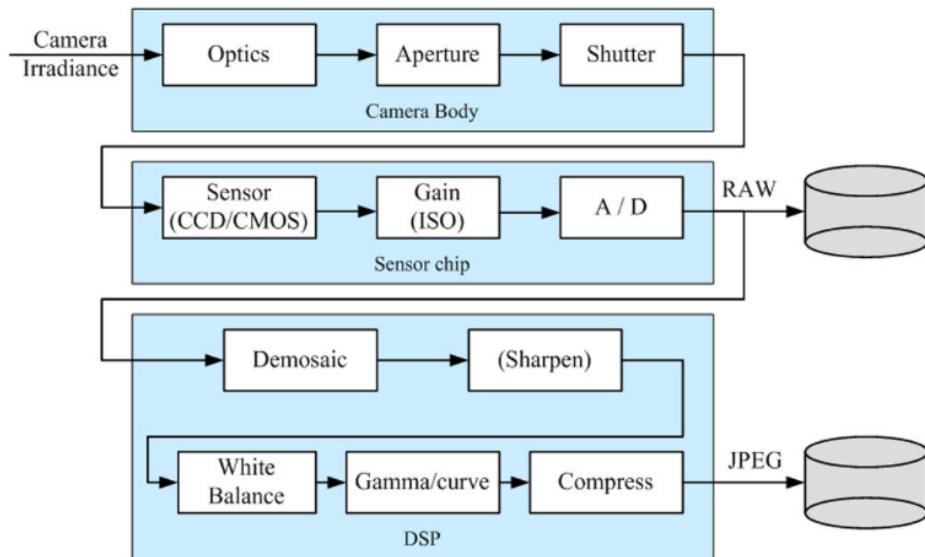


Figura 2.4: Proceso de la obtención de una imagen con una cámara digital. Fuente: [6]

El sensor óptico se compone de un arreglo bidimensional de píxeles sensibles a la luz y a ciertos rangos de longitud de onda de la luz, lo que significa que pueden detectar ciertos colores o características de la luz que incide en ellos. Este arreglo se transforma en una matriz con valores de intensidad para cada color en cada pixel, es así como se puede entender a una imagen digital como un arreglo multidimensional de valores.

La tarea de la visión artificial es especialmente complicada pues se intenta dar sentido o significado a un arreglo bidimensional de píxeles. Existen distintos métodos y algoritmos que intentan convertir esta inmensa cantidad de datos (los píxeles) en información útil (objetos).

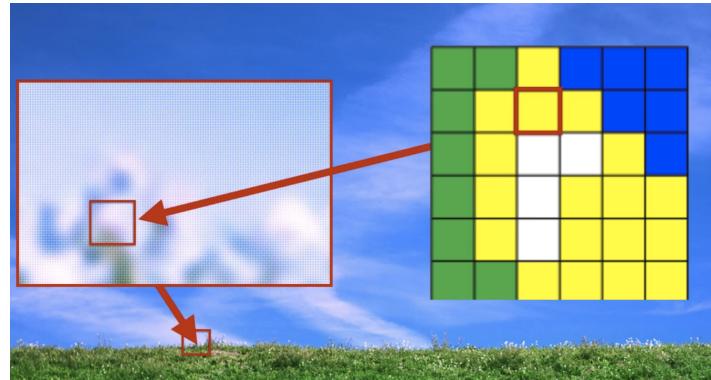


Figura 2.5: Representación de una imagen digital como un arreglo de píxeles. Fuente: [6]

2.3. Redes Neuronales Artificiales

2.3.1. Aprendizaje Automático

El aprendizaje automático es un subcampo de la inteligencia artificial que intenta extraer patrones mediante un proceso de *aprendizaje* a partir de datos [7]. Este proceso de aprendizaje se define de acuerdo a una **tarea específica** T que intenta aprenderse en base a **experiencia pasada** E tomando como referencia una **medida de rendimiento** P . Dentro de esta definición, se puede listar varios ejemplos de tareas de aprendizaje que usualmente se resuelven usando los conceptos del aprendizaje automático o también llamado *machine learning*:

- **Un algoritmo de aprendizaje que pueda jugar ajedrez:**
 - **Tarea T :** Jugar Ajedrez.
 - **Medida de Rendimiento P :** Porcentaje de partidas ganadas contra el oponente.
 - **Experiencia E :** Información de varias partidas de práctica.
- **Un algoritmo de aprendizaje que pueda reconocer dígitos manuscritos:**
 - **Tarea T :** Reconocer y clasificar dígitos manuscritos dentro de una imagen.
 - **Medida de Rendimiento P :** Porcentaje de dígitos correctamente clasificados.
 - **Experiencia E :** Base de datos de imágenes de dígitos con sus etiquetas correspondientes.
- **Un algoritmo de aprendizaje que pueda reconocer la voz:**
 - **Tarea T :** Extraer una secuencia de palabras de una grabación de voz.

- **Medida de Rendimiento P :** Porcentaje de palabras correctamente predichas.
- **Experiencia E :** Grabaciones de voz con una transcripción correspondiente.

Esta definición de aprendizaje es lo suficientemente amplia como para englobar todas las tareas que el campo del aprendizaje automático intenta resolver en la actualidad. Sin embargo, debido a su naturaleza, se pueden clasificar las tareas de aprendizaje en tres grandes categorías que tienen características particulares: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.

La diferencia entre estos tres tipos de problemas surge de la distinta naturaleza de la experiencia E disponible para el entrenamiento. A continuación, se procede a detallar cada uno de ellos.

2.3.1.1. Aprendizaje supervisado

En el caso de las tareas de aprendizaje supervisado, la experiencia constituye un conjunto de datos o *dataset* que contiene ejemplos con *características* y cada ejemplo está asociado con una *etiqueta*. Por ejemplo, un conjunto de datos de flores donde cada registro contiene datos de la flor (características) y la especie a la que pertenece (etiqueta). Dentro de los algoritmos que atacan problemas de aprendizaje supervisado se pueden encontrar 2 grandes categorías.

Clasificación Las tareas de clasificación tienen como característica el hecho de que la etiqueta de cada ejemplo en el conjunto de datos pertenece a una categoría o, en otras palabras, tiene una naturaleza discreta y finita. Por ejemplo, en el caso de la clasificación de las flores mencionada anteriormente, la etiqueta solamente puede pertenecer a un conjunto finito de especies de flores y cada ejemplo pertenece a una de estas especies.

Regresión En las tareas de regresión, las etiquetas pertenecen a un conjunto de números reales o de naturaleza continua. En este caso, las etiquetas no se asocian con categorías sino más bien con otro tipo de variables. Un ejemplo muy conocido es el de la tarea de la predicción del precio de una casa en base a sus características, el precio de una casa no puede categorizarse porque representa un número que puede tener infinitos valores dentro de un rango definido.

En las tareas del aprendizaje supervisado, se puede considerar cada ejemplo como una descripción de una situación (características) en conjunto con una especificación (etiqueta), cada uno de los ejemplos dentro el conjunto de datos son eventos independientes y se pueden analizar por separado. En este sentido la tarea del algoritmo es generalizar la respuesta para casos no presentes en el conjunto inicial de datos.

2.3.1.2. Aprendizaje no supervisado

En las tareas del aprendizaje no supervisado la experiencia contenida en el conjunto de datos tiene la característica de no poseer ninguna etiqueta, por tanto, usualmente se intenta buscar una estructura escondida dentro el conjunto de datos o, dicho de otra manera, se

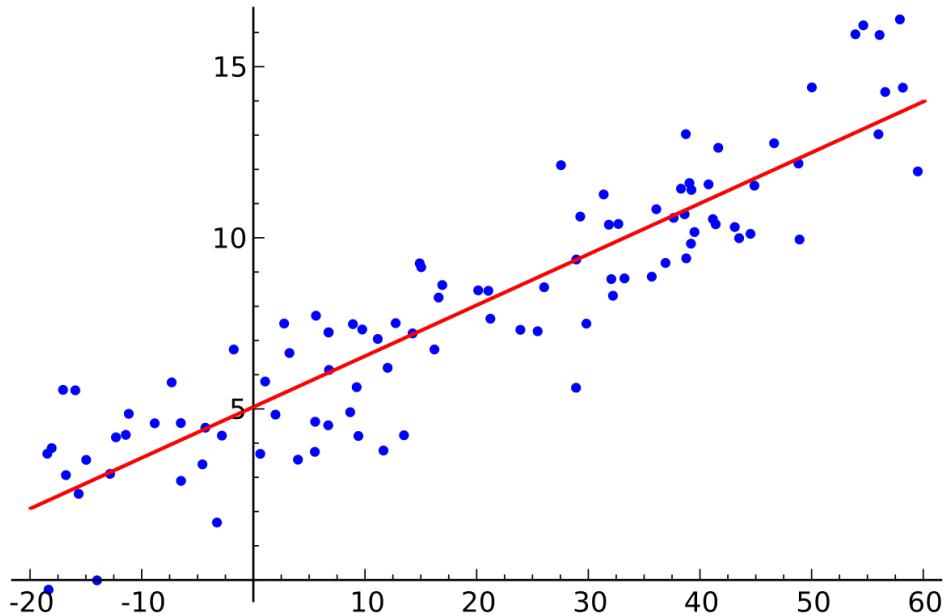


Figura 2.6: Regresión lineal, una tarea de aprendizaje supervisado. Fuente: [4]

buscan patrones que puedan presentarse en dichos datos. Estos patrones pueden aprovecharse para extraer información relevante de la naturaleza de datos de muy alta dimensionalidad, información que normalmente no es trivial de encontrar o visualizar por una persona. Entre algunas de las tareas más comunes dentro del aprendizaje no supervisado, se pueden listar:

Clustering Refiere a la tarea de separar y agrupar los datos en un número finito de conjuntos o *clusters*. Los *clusters* normalmente denotan una estructura oculta dentro de los datos y proporcionan información acerca de la similaridad entre ejemplos del conjunto de datos.

Reducción de dimensionalidad Uno de los problemas con las bases de datos y conjuntos de datos disponibles es que poseen una dimensionalidad bastante alta haciendo prácticamente imposible para un humano poder visualizar o encontrar patrones e información útil en los mismos. Este problema se suele tratar con algoritmos de reducción de dimensionalidad, en la que se encuentra una representación estimada de los datos pero con menos dimensiones. Uno de los algoritmos más conocidos y usados en esta categoría es el análisis de componente principal o PCA, por sus siglas en inglés, en el que se encuentra una representación de los datos en una menor dimensión usando proyecciones ortogonales.

Estimación de probabilidad Muchos conjuntos de datos son obtenidos de distintas fuentes y a lo largo de varios intervalos de tiempo, en este entendido, es muy útil conocer o aproximar la distribución de probabilidad de los datos para luego poder realizar predicciones o tratarlos con algún modelo en específico.

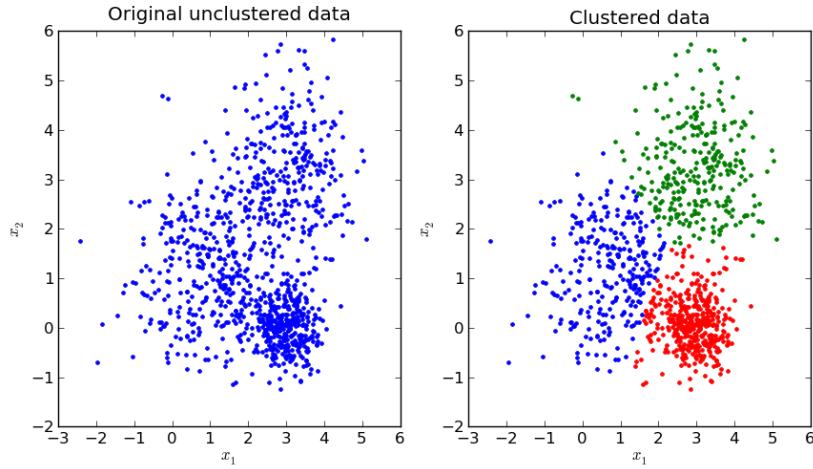


Figura 2.7: Clustering, una tarea de aprendizaje no supervisado. Fuente: [8]

2.3.2. Aprendizaje Profundo

Dentro del campo de la inteligencia artificial y el aprendizaje automático se han implementado diversos tipos de algoritmos con éxito en los tipos de tareas de aprendizaje mencionados anteriormente. La base teórica y los detalles de implementación de estos algoritmos son muy variados, sin embargo, las redes neuronales artificiales han experimentado un incremento en el interés en la investigación y en las aplicaciones muy importante. Tal es el éxito de las mismas que se ha creado un subcampo exclusivo llamado aprendizaje profundo o *deep learning*. El aprendizaje profundo es un campo de la inteligencia artificial que se encarga de estudiar exclusivamente a las redes neuronales artificiales, sus componentes, arquitectura y aplicaciones. El impresionante rendimiento de estos algoritmos reside principalmente en el concepto de la representación que generan a partir de los datos que se procesan.

El aprendizaje profundo resuelve el problema del aprendizaje de representaciones al introducir representaciones que se expresan en términos de otras representaciones más simples. Además, permite a una computadora construir conceptos complejos a partir de conceptos más simples. Un ejemplo de la generación de estos conceptos o representaciones se puede apreciar en la Figura(2.8).

A continuación, se procede a definir los conceptos más importantes de redes neuronales artificiales con los cuales se podrá plantear la solución al problema de la conducción autónoma usando visión artificial.

2.3.2.1. Redes neuronales feedforward

Las redes neuronales feedforward o también llamadas perceptrón multicapa, son la base fundamental de los modelos de aprendizaje profundo. El objetivo de una red neuronal feedforward es el de aproximar una función f^* . Por ejemplo, para una tarea de clasificación, $y = f^*(\mathbf{x})$ mapea una entrada \mathbf{x} a una categoría y . Una red neuronal feedforward define

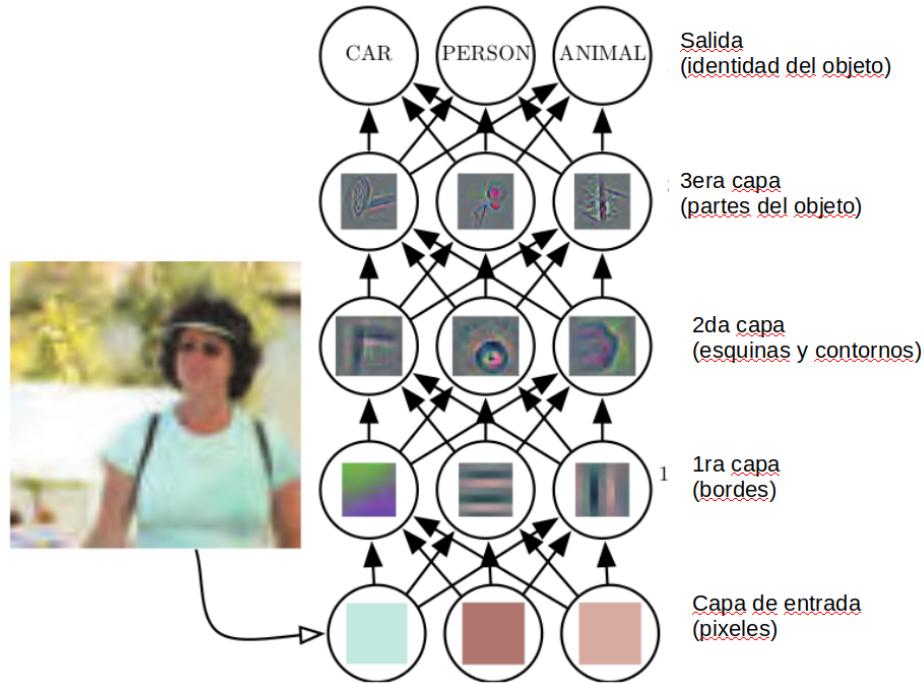


Figura 2.8: Ilustración de un modelo de aprendizaje profundo. Las representaciones se generan en las capas ocultas y corresponden con características de distintos niveles de complejidad. Fuente: [9]

un mapeo $\mathbf{y} = f(\mathbf{x}, \mathbf{W})$ y aprende el valor de los parámetros \mathbf{W} que resulten en la mejor aproximación[9].

Este tipo de modelos son denominados feedforward debido a que la información fluye a por la función siendo evaluada desde \mathbf{x} , a través de distintos cálculos intermedios definidos por f , hasta llegar a la salida \mathbf{y} . No existen conexiones de retroalimentación en las que la salidas del modelo se inyecten de nuevo a sí mismo. Las redes neuronales que poseen este tipo de conexiones de retroalimentación son denominadas redes neuronales recurrentes.

Para definir una red neuronal feedforward se puede comenzar definiendo un modelo basado en una combinación lineal en conjunto con una función no lineal que toma la siguiente forma:

$$y(\mathbf{x}, \mathbf{W}) = f \left(\sum_{j=1}^M w_j x_j \right) \quad (2.1)$$

donde $f()$ es una función de activación no lineal. Esto lleva al modelo básico de una red neuronal que puede ser descrita como una serie de transformaciones. Primero, se construyen M combinaciones lineales de las variables de entrada x_1, \dots, x_D donde D es la dimensión del vector de entrada \mathbf{x} :

$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \quad (2.2)$$

donde $j = 1, \dots, M$, y el superíndice (1) indican que los correspondientes parámetros se encuentran en la primera capa de la red. Los parámetros $w_{ji}^{(1)}$ se suelen conocer también con el nombre de *pesos* y los parámetros $w_{j0}^{(1)}$ con el nombre de *sesgos* o *biases*. Las cantidades a_j se conocen como *activaciones*, y cada una de ellas es luego transformada usando una función no lineal y derivable conocida como la *función de activación* $h()$ para luego obtener:

$$z_j = h(a_j) \quad (2.3)$$

Estas cantidades corresponden con la salida de la capa y también se suelen referir por el nombre de *unidades ocultas*. Las funciones no lineales $h()$ pueden escogerse dependiendo a diversos criterios de rendimiento o de comportamiento. Siguiendo a la Ecuación(2.1), las unidades ocultas se pueden volver a procesar con una combinación lineal y función de activación en una segunda capa:

$$a_k = \sum_{j=1}^M w_{kj}^{(2)} z_j + w_{k0}^{(2)} \quad (2.4)$$

donde $k = 1, \dots, K$ y K corresponden con el número de salidas de la segunda capa. Finalmente, si se considera a esta capa como la capa de salida, podemos transformar las activaciones de la segunda capa con una función de activación. Normalmente, para una tarea de regresión, la función de activación es una función identidad, es decir $y_k = a_k$. Para una tarea de clasificación binaria, en cambio, la función de activación es una función sigmoidal:

$$y_k = \sigma(a_k) \quad (2.5)$$

donde

$$\sigma(a) = \frac{1}{1 + e^{-a}} \quad (2.6)$$

Finalmente, se pueden combinar las etapas en una función general de la red que, para una salida sigmoidal, toma la siguiente forma:

$$y_k(\mathbf{x}, \mathbf{W}) = \sigma \left(\sum_{j=1}^M w_{kj}^{(2)} h \left(\sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \quad (2.7)$$

De esta manera, se define una red neuronal de dos capas a partir de la combinación lineal de las entradas y las unidades ocultas con los parámetros o pesos de la red transformados por funciones de activación no lineal. La arquitectura de la red definida en la Ecuación(2.7) se puede visualizar en la Figura(2.9) donde se observa claramente las relaciones que se han

definido anteriormente en forma gráfica y la naturaleza del flujo en una sola dirección (feed-forward) de los datos desde la entrada hasta la salida. En este caso, la red neuronal analizada es una red neuronal con una capa oculta.

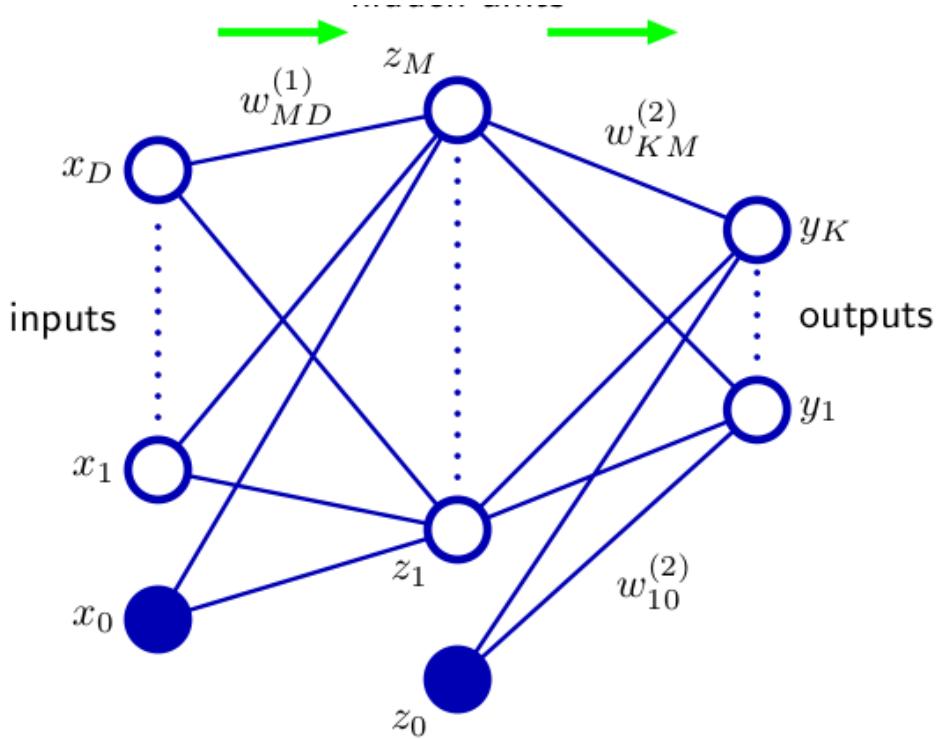


Figura 2.9: Diagrama de la red neuronal de dos capas correspondiente a la Ecuación(2.7).
Fuente: [10]

2.3.2.2. Función de costo

La función de costo es una función que permite definir el rendimiento de una red neuronal con respecto a las predicciones esperadas a la salida. Un aspecto importante en el diseño de una red neuronal es la elección adecuada de la función de costo.

La función de costo o función de error, se define de manera similar al caso del ajuste de curvas. Es decir, se desea minimizar una suma de errores cuadráticos. Dado un conjunto de entrenamiento compuesto por un conjunto de vectores de entrada $\{\mathbf{x}_n\}$, donde $n = 1, \dots, N$, junto con un conjunto de vectores objetivo $\{\mathbf{t}_n\}$ el objetivo es minimizar la función:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 \quad (2.8)$$

donde el valor de \mathbf{w} que minimice la función de error $E(\mathbf{w})$ será considerado como el mejor conjunto de parámetros sobre el cual el modelo puede generalizar.

2.3.2.3. Entrenamiento usando gradientes y retropropagación

Una vez definidos con claridad los componentes básicos de una red neuronal feedforward y cómo es el proceso del flujo de la información desde la entrada \mathbf{x} hasta la salida \mathbf{y} , solamente queda especificar el procedimiento necesario para ajustar los parámetros o pesos de la red \mathbf{W} . Para este cometido, el método más usado es el de la retropropagación o *backpropagation*, popularizado a partir del paper de Rumelhart [11] y ampliamente usado en la actualidad en la mayoría de las redes neuronales. La retropropagación tiene el objetivo de encontrar los gradientes, en específico, se desea encontrar el gradiente de la función de costo o error con respecto a los parámetros $\nabla_{\mathbf{W}} E(\mathbf{W})$. Si se toma en cuenta que una red neuronal feedforward puede tener varias capas ocultas, la gradiente de la función de costo está en función de los parámetros y funciones de activación de cada capa, por tanto, se necesita usar la regla de la cadena para poder encontrar estos gradientes intermedios.

Sea $y = g(x)$ y $z = f(g(x)) = f(y)$ dos funciones reales con argumento real, la regla de la cadena define lo siguiente:

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} \quad (2.9)$$

Se puede generalizar la anterior expresión para casos fuera de una variable escalar donde $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, g mapea de \mathbb{R}^m a \mathbb{R}^n y f mapea de \mathbb{R}^n a \mathbb{R} , si $\mathbf{y} = g(\mathbf{x})$ y $z = g(\mathbf{y})$, entonces:

$$\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \quad (2.10)$$

en notación vectorial, sería equivalente a lo siguiente:

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \nabla_{\mathbf{y}} z \quad (2.11)$$

donde $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ es el jacobiano $n \times m$ de g . Lo importante aquí es recordar que en una función multivariable, el gradiente proporciona la dirección hacia donde la función crece más rápidamente, esta intuición es fundamental para poder actualizar los pesos de la red en una etapa posterior.

El concepto fundamental de la retropropagación es encontrar estos gradientes de manera secuencial, partiendo desde la capa de salida hasta llegar a la capa de entrada.

Actualización de los parámetros de la red La importancia de hallar los gradientes de la red con respecto a los pesos reside en que son justamente los gradientes los que proporcionan la información de la evolución de los pesos con respecto de la función de error y en qué dirección se encuentra el mínimo. En la Figura(2.10) se puede ver cómo, para una combinación arbitraria de pesos \mathbf{w}_C el gradiente de la función de error ∇E indica la dirección de máximo crecimiento de la superficie

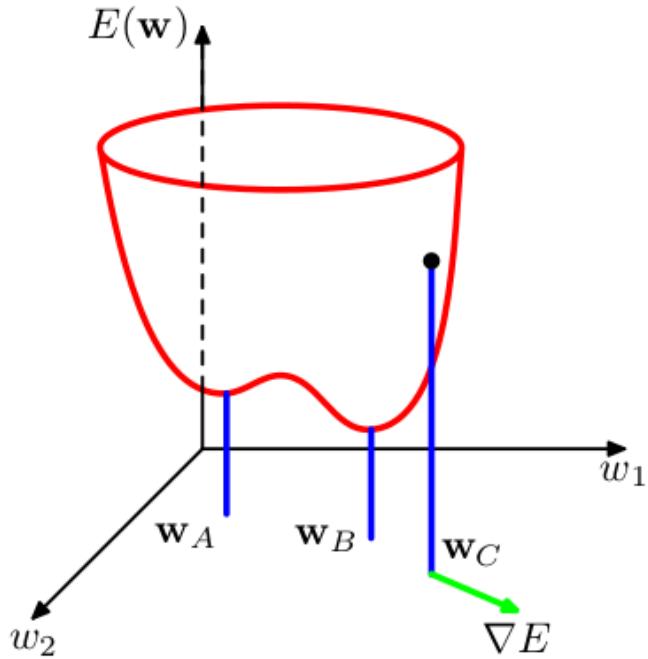


Figura 2.10: Visualización de la función de error $E(\mathbf{w})$ como una superficie. El punto \mathbf{w}_A es un mínimo local y el punto \mathbf{w}_B es el mínimo global. Fuente: [10]

La actualización de los parámetros dados los gradientes de la red se puede realizar de distintas formas, pero una de las más comunes es el algoritmo llamado *Descenso de gradiente*, donde, de forma iterativa, se va actualizando los pesos restando una medida del gradiente de la siguiente manera:

Dicho de otro modo, en el descenso de gradiente, se avanza un pequeño paso en la dirección opuesta al gradiente para avanzar hacia el mínimo:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \alpha \nabla E(\mathbf{w}^{(\tau)}) \quad (2.12)$$

donde el parámetro $\alpha > 0$ se conoce como la razón de aprendizaje o *learning rate*. Después de cada actualización, el gradiente es recalculado para poder encontrar los nuevos pesos y el proceso se repite hasta encontrar el punto donde $\nabla E = 0$, lo que quiere decir que se ha llegado al mínimo. Usualmente, por la naturaleza de la superficie de la función de error y diversos errores en el cálculo en una computadora, el número de iteraciones se limita en base a cierto parámetro de rendimiento.

Tradicionalmente, el algoritmo de descenso de gradiente se evalúa sobre todo el conjunto de datos de entrenamiento, este tipo de descenso de gradiente es llamado *batch gradient descent*, una *pasada* por todo el conjunto de entrenamiento se denomina usualmente una *época*. La principal desventaja es que la actualización de los pesos de la red neuronal solamente se actualiza una sola vez en cada época; esto representa una dificultad en tiempo

de procesamiento y memoria cuando se procesan conjuntos de datos muy grandes. Se han desarrollado versiones alternativas al *batch gradient descent* que mejoran su desempeño y son más tratables con conjuntos de datos muy grandes:

- ***Stochastic Gradient Descent.*** Refiere a que la actualización de los pesos en cada muestra procesada, es decir, que los pesos se actualizan n veces en una época.
- ***Mini-batch Gradient Descent.*** Es una mezcla del descenso de gradiente tradicional y el descenso de gradiente estocástico, en este caso, las muestras se alimentan al algoritmo en *mini-batches* o muestras de tamaño pequeño.

Adam Adam, abreviación de *Adaptive Moment Estimation*, es un algoritmo de optimización alternativo al descenso de gradiente que utiliza promedios tanto de los gradientes como los momentos de segundo orden de los gradientes [12]. Dados los parámetros $\mathbf{w}^{(\tau)}$ y la función de error $E^{(\tau)}$, la actualización de parámetros de acuerdo a Adam se da mediante las siguientes expresiones:

$$\begin{aligned} m_w^{(\tau+1)} &= \beta_1 m_w^{(\tau)} + (1 - \beta_1) \nabla_w E^{(\tau)} \\ v_w^{(\tau+1)} &= \beta_2 m_w^{(\tau)} + (1 - \beta_2) (\nabla_w E^{(\tau)})^2 \\ \hat{m}_w &= \frac{m_w^{(\tau+1)}}{1 - (\beta_1)^{(\tau+1)}} \\ \hat{v}_w &= \frac{v_w^{(\tau+1)}}{1 - (\beta_2)^{(\tau+1)}} \\ w^{(\tau+1)} &= w^{(\tau)} - \alpha \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon} \end{aligned} \quad (2.13)$$

Donde m_w representa a los momentos de primer orden y v_w representa a los momentos de segundo orden. Adam tiene los siguientes hiperparámetros:

- α es la razón de aprendizaje, análogo al caso del descenso de gradiente tradicional.
- β_1 es la razón de decaimiento exponencial para el estimado de los momentos de primer orden.
- β_2 es la razón de decaimiento exponencial para el estimado de los momentos de segundo orden.
- ϵ es un escalar pequeño usado para prevenir una posible división entre cero.

La introducción de los estimados de los momentos de primer y segundo orden de los gradientes permiten que se tome en cuenta el *ímpetu* con el que los gradientes cambian en el proceso de entrenamiento lo cual evita que pueda sobrepasar el mínimo y garantiza su convergencia a mayor velocidad que la de una actualización con un descenso de gradiente tradicional.

2.3.2.4. Funciones de activación

Se ha estudiado con mucho detalle la naturaleza de la función de activación $f()$, introducida en la Ecuación(2.1), analizando fundamentalmente dos aspectos: su contribución a la generación de representaciones internas en las capas ocultas de la red neuronal, así como también su comportamiento de sus gradientes con relación a la etapa de aprendizaje, tal como se ha visto en la Sección(2.3.2.3), donde se ha establecido claramente que el papel de la función de activación y sus gradientes es fundamental para el proceso de retropropagación y ajuste de los pesos de la red. Una guía con algunos criterios puede encontrarse en [13], de donde se rescatan los siguientes aspectos a la hora de escoger una función de activación:

- Que modele de forma no lineal la naturaleza de una representación interna fácil de interpretar.
- Que sea derivable en todo el rango de trabajo.

Tomando en cuenta lo anterior, se han generado diversas tendencias en la aplicación de funciones de activación y a continuación se analizarán las más relevantes para este proyecto, la función sigmoide, se analiza en el Apéndice(A).

Función tangente hiperbólica La función tangente hiperbólica o $\tanh()$ se define mediante la siguiente expresión:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.14)$$

Esta función también cumple con los requisitos de no linealidad y de ser derivable, pero en este caso, el rango de salida de la función $\tanh()$ es desde -1 a 1 . Considerando el concepto de función de probabilidad que ofrece la salida de la función sigmoide, la función tangente hiperbólica introduce el concepto de aceptación o negación de una premisa, donde una premisa aceptada se acerca a 1 y una premisa rechazada se acerca a -1 , el valor de 0 , indica indecisión. La función $\tanh()$ se ha usado junto a su par sigmoide ampliamente en los inicios de las redes neuronales para las activaciones de las capas ocultas, pero, tal como se puede observar en la Figura(2.11), presenta la misma desventaja del desvanecimiento de gradientes, dado que para valores muy grandes, la derivada se aproxima a cero.

En la actualidad, para implementaciones de redes neuronales profundas, el uso de las funciones sigmoide y tangente hiperbólica en las capas ocultas está ampliamente desaconsejado por los problemas anteriormente planteados. En su lugar se ha generado nuevos tipos de funciones de activación que presentan características bastante favorables para el aprendizaje y ajuste de pesos basados en gradientes.

Unidad Lineal Rectificada - ReLU Introducida por primera vez en el año 2010 por Nair y Hinton, las Unidades Lineales Rectificadas o ReLU, se propusieron para mejorar el rendimiento de un tipo especial de red neuronal: las máquinas restringidas de Boltzman (RMB, por sus siglas en inglés) [15], pero pronto, ganarían gran popularidad también en las

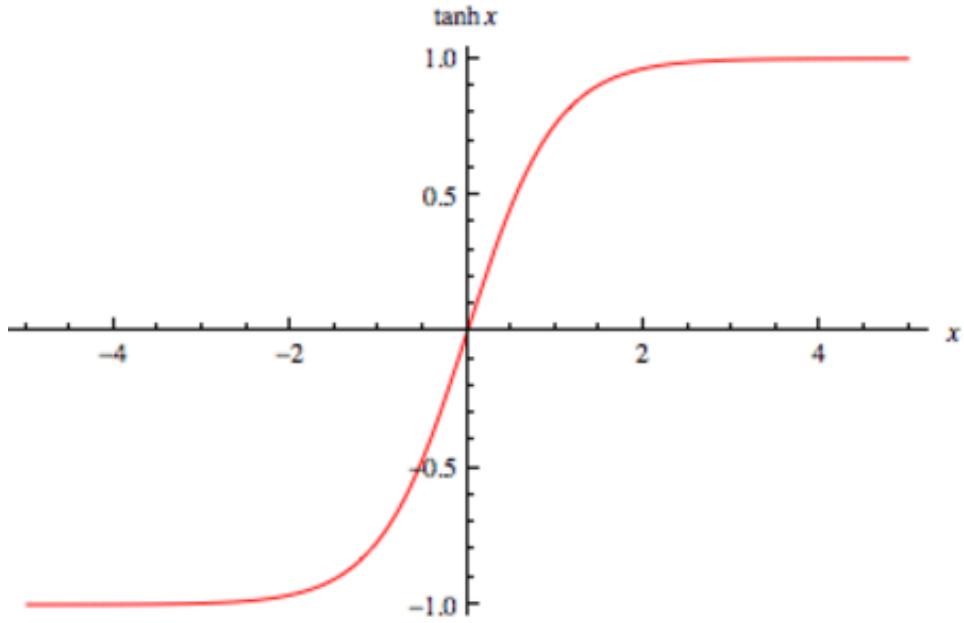


Figura 2.11: Gráfico de la función tangente hiperbólica $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Fuente: [14]

redes neuronales feedforward y en redes neuronales convolucionales como se puede apreciar en [16].

La función ReLU se define de la siguiente manera:

$$g(x) = \max\{0, x\} \quad (2.15)$$

Tal como se puede apreciar, la característica más importante de la función ReLU es su simplicidad pues es muy similar a una función identidad, la diferencia es que ReLU toma el valor de cero para todos los valores de x que sean negativos. Esto resuelve el problema del desvanecimiento de gradientes pues garantiza que el gradiente tenga un valor razonable si la entrada está activa y además sea consistente.

Sin embargo, dado que la función ReLU tiene una gradiente nula para valores negativos es de vital importancia que se garantice la existencia de gradientes positivos, aunque sea pequeños durante la inicialización y las capas previas.

2.3.2.5. Métricas y puntajes para tareas de regresión

Una vez definidos los componentes de una red neuronal y el procedimiento para el entrenamiento, es necesario definir alguna forma de evaluar el rendimiento de la misma. Esta evaluación normalmente se realiza en base al cálculo de puntajes o *scores* para cada tarea en específico. Si se trata de una tarea de regresión, como es el caso del presente proyecto, se tienen distintas métricas para evaluar el rendimiento del modelo.

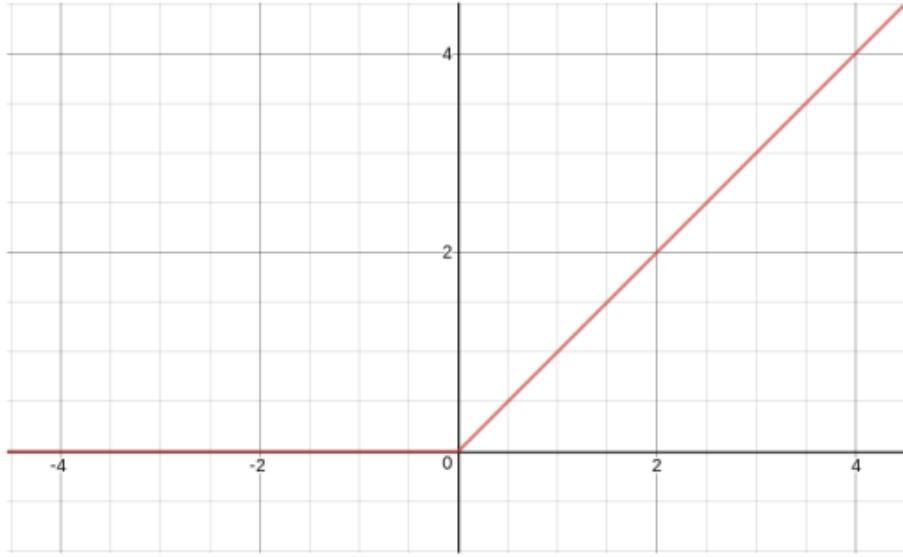


Figura 2.12: Gráfico de la función ReLU. Fuente: [14]

Error cuadrático medio - MSE El error cuadrático medio es una medida correspondiente con el valor esperado del error elevado al cuadrado [17]. Si \hat{y}_i es el valor predicho de la i -ésima muestra y y_i es el valor real correspondiente, entonces el error cuadrático medio (MSE) estimado sobre n muestras se define como:

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2 \quad (2.16)$$

Un valor de MSE igual a 0 indica que el modelo puede predecir perfectamente todas las muestras.

Error absoluto medio - MAE El error absoluto medio es una medida correspondiente con el valor esperado del error absoluto [18]. Si \hat{y}_i es el valor predicho de la i -ésima muestra y y_i es el valor real correspondiente, entonces el error absoluto medio (MAE) estimado sobre n muestras se define como:

$$MAE(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} |y_i - \hat{y}_i| \quad (2.17)$$

Un valor de MAE igual a 0 indica que el modelo puede predecir perfectamente todas las muestras.

Coeficiente de determinación - R^2 El coeficiente de determinación R^2 provee una medida de cuán efectiva será la predicción de futuras muestras por el modelo. El mejor puntaje

possible es 1.0 y puede tomar valores negativos para una predicción especialmente ineficiente [19].

Si \hat{y}_i es el valor predicho de la i -ésima muestra y y_i es el valor real correspondiente, entonces el puntaje R^2 estimado sobre n muestras se define como:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1} (y_i - \bar{y}_i)^2} \quad (2.18)$$

donde $\bar{y} = \frac{1}{n} \sum_{i=0}^{n-1} y_i$.

2.3.2.6. Diseño de Arquitectura e hiperparámetros de una red neuronal

Como se ha podido ver en las secciones anteriores, la característica más importante de una red neuronal es que es capaz de generar representaciones internas a partir de los datos y la retropropagación de los gradientes, lo que ha finalizado la era de la “ingeniería de características” donde se necesitaba conocimiento experto para elegir las representaciones adecuadas y cómo calcularlas, sin embargo, esto ha llevado a que el diseño de las redes se enfoque en otros aspectos como la arquitectura y otros parámetros externos que definen el modelo de la red. Por arquitectura de la red, se entiende a la estructura general de la red neuronal, el número de capas, el número de unidades o neuronas por cada capa y cómo las unidades y capas se conectan entre sí.

La gran parte de las redes neuronales están organizadas en grupos de unidades llamadas capas, asimismo, las capas se ordenan de manera encadenada siendo cada capa una función de la capa que la precede, esto se ha establecido para una red de dos capas en la Ecuación(2.7) en su forma general, pero, se puede expresar el modelo de forma vectorial de la siguiente manera:

$$\mathbf{h}^{(1)} = g^{(1)} (\mathbf{W}^{(1)T} \mathbf{x} + \mathbf{b}^{(1)}) \quad (2.19)$$

$$\mathbf{h}^{(2)} = g^{(2)} (\mathbf{W}^{(2)T} \mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \quad (2.20)$$

En términos de redes neuronales, la cantidad de unidades en cada capa se denomina el ancho o *width* y la cantidad de capas se denomina profundidad o *depth*.

Hiperparámetros Tanto la profundidad como el ancho de la red, son parámetros que se deben escoger en base a ciertos criterios, éstos parámetros son externos a los parámetros o pesos \mathbf{W} de la red neuronal, por lo que se denominan hiperparámetros. Aparte del ancho y profundidad, en el diseño de una red neuronal se consideran las siguientes variables:

- Razón de aprendizaje α .

- Funciones de activación.
- Tamaño del *mini-batch*.
- Número de épocas de entrenamiento.
- Algoritmo optimizador (Ejemplo: Descenso de gradiente).
- Función de costo.

Los hiperparámetros deberán ser escogidos de manera cuidadosa, pero debido a la gran complejidad y poca predictibilidad del comportamiento de una red neuronal profunda normalmente se eligen en conjunto con un proceso de prueba y error, analizando las curvas de aprendizaje y rendimiento en conjuntos de datos de validación y prueba.

2.3.3. Redes Neuronales Convolucionales

Las redes neuronales convolucionales son un tipo especializado de red neuronal que sirven para procesar datos de tipo "grilla" [9]. Algunos ejemplos de datos de tipo grilla que se pueden mencionar son los siguientes:

- **Series de tiempo.** Grilla de una dimensión tomados en intervalos regulares de tiempo.
- **Imágenes digitales.** Grilla de pixeles de dos o más dimensiones (Escala de grises, RGB).

Las también llamadas redes convolucionales, han demostrado un éxito impresionante en diversas aplicaciones prácticas especialmente en el campo de la visión por computador y el procesamiento de texto y lenguaje natural. El término "red neuronal convolucional" proviene del hecho de que en este tipo de redes neuronales se utiliza una operación matemática llamada **convolución**, siendo la convolución una operación lineal especializada para procesar datos de tipo grilla.

En los párrafos posteriores, se procede a describir la operación de convolución en el contexto de redes neuronales, pues, no siempre la definición de la misma corresponde con el concepto de convolución usado en distintos campos de la ciencia y la ingeniería.

2.3.4. Operación de convolución

En su forma más general, la convolución es una operación entre dos funciones reales y su definición se puede introducir usando el concepto de un promedio ponderado. Sea una función $x(t)$ dependiente del tiempo, tanto x como t son números reales; en este caso, la función x puede entenderse como una serie de medidas en un instante de tiempo t . Considérese una segunda función de ponderación $w(\tau)$ donde τ es la antigüedad de una medida. Si se aplica

la función de ponderación en cada instante de tiempo, se puede obtener una nueva función definida por:

$$s(t) = \int x(\tau)w(t - \tau)d\tau \quad (2.21)$$

Esta operación es llamada la *operación de convolución* y es denotada tradicionalmente con un asterisco:

$$s(t) = (x * w)(t) \quad (2.22)$$

En el ejemplo de la ponderación, w debe ser una función de densidad de probabilidad válida, o la salida no podrá ser considerada como un promedio ponderado. Además, w también debe ser 0 para cualquier $t < 0$, esta última característica se denomina comúnmente como el principio de “causalidad”. En general, la convolución está definida para cualquier función en la cual la integral anteriormente declarada esté definida y puede ser usada para otros propósitos aparte de promedios ponderados.

Hablando en términos de una red neuronal convolucional, el primer argumento (en el ejemplo, la función x) es comúnmente referido como la **entrada**, y el segundo argumento (w , en el ejemplo) es referido como el **kernel**. La salida, a su vez, es normalmente referida como el **mapa de características**.

Por su parte, cuando se trata de señales digitales, como los datos en una computadora, el tiempo tiene una naturaleza discreta, es decir, que los datos estarán disponibles en intervalos regulares de tiempo. En este caso, el índice de tiempo t puede tomar solamente valores enteros y, entonces, es válido asumir que tanto x como w están definidos solamente para valores enteros de t . De este modo, se puede definir la convolución discreta:

$$s(t) = (x * w)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)w(t - \tau) \quad (2.23)$$

En el contexto de las aplicaciones de aprendizaje automático o, más específicamente, aprendizaje profundo, la entrada es usualmente un arreglo multidimensional de datos, y el kernel es usualmente un arreglo multidimensional de parámetros que se adaptan en el proceso de aprendizaje.

2.3.5. Operación de max pooling

Usualmente, las capas convolucionales utilizan a su salida una operación de *pooling*, en específico, el *max pooling*. Esta operación permite hacer un submuestreo del mapa de características de la salida de la capa convolucional donde se toman en cuenta solamente los valores máximos. Esta etapa de submuestreo tiene la ventaja de reducir la dimensionalidad de los mapas de características obtenidos y, a su vez, de mantener solamente los valores de activación máximos para las capas convolucionales. Además, la operación permite agregar algo de invarianza a pequeñas variaciones en el mapa de características para mantener siempre las activaciones máximas.

La operación trabaja sobre vecindades de pixeles definidas de acuerdo a la parametrización de cada capa de *max pooling*. Como se puede apreciar en la Figura(2.13)

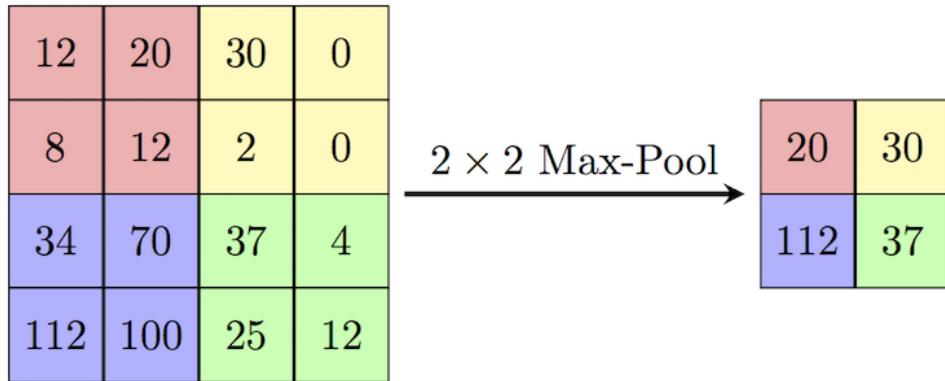


Figura 2.13: Gráfico de la operación max pooling. Fuente: [14]

2.3.5.1. Procesamiento de imágenes con redes neuronales convolucionales

La operación de convolución se usa frecuentemente sobre datos con más de una dimensión. Las imágenes digitales son un perfecto ejemplo de un arreglo multidimensional de datos. Una imagen digital se representa mediante una matriz con filas y columnas, donde cada elemento se denomina pixel y contiene información acerca de la intensidad o luminancia, para una imagen en escala de grises o el nivel de color para distintos canales en una imagen a color. Si se toma el ejemplo de la imagen en escala de grises, se tiene una entrada o imagen bidimensional I con un kernel bidimensional correspondiente K :

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \quad (2.24)$$

Dado que la convolución es commutativa, se puede reescribir la ecuación 2.24 como:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n) \quad (2.25)$$

Frecuentemente, la última fórmula es la más utilizada en librerías de aprendizaje profundo por su sencillez en la implementación en un sistema computacional, esto, dado que existe menos variación en el rango de valores válidos de m y n . En la Figura(2.14), se puede apreciar una visualización de la operación de convolución aplicada a una imagen.

2.3.5.2. Aprendizaje de representaciones internas

Una de las preguntas clave en la visión por computador es el cómo generar una buena y significativa representación interna de una imagen, dado que la mayor parte de la imagen corresponde con pixeles que no aportan mucha información relevante a la tarea asignada. Por ejemplo, si se quisiera detectar un rostro dentro de una imagen, normalmente se suele encontrar una representación que ayude a aislar solamente las porciones de la imagen que pueden contener el rostro, tales como la búsqueda de contornos, bordes y características típicas de un

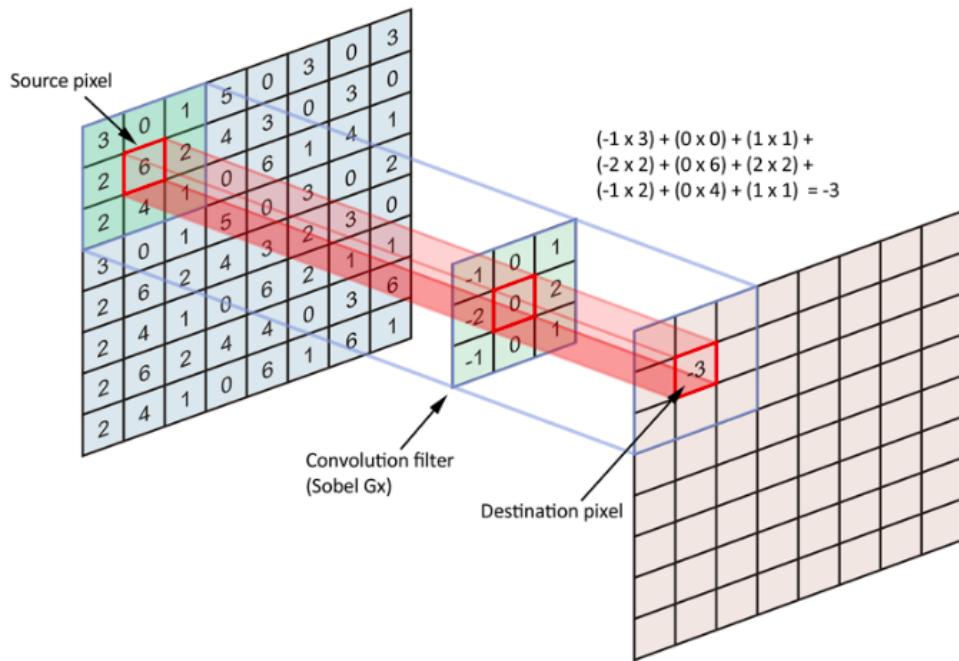


Figura 2.14: Visualización de la operación de convolución sobre una imagen digital. Fuente: [20]

rostro. Antes de la aparición de las redes convolucionales, estas representaciones se hallaban de manera manual y gracias al conocimiento de expertos en el área del procesamiento de imágenes. La definición de características y mapas de características era comúnmente conocida como la *ingeniería de características*, en la cual los expertos creaban descriptores para tareas específicas con una gran inversión de tiempo en la sintonización fina de los mismos.

En contraste con el anterior enfoque, las redes convolucionales generan sus propias representaciones internas de manera automática gracias al aprendizaje de los parámetros de cada uno de los kernels que componen las distintas capas de la red neuronal. En principio, las redes convolucionales se inspiraron en el trabajo de Hubel y Wiesel sobre la corteza visual primaria de un gato[21]. En dicho trabajo, se logró identificar células simples que respondían de manera sobresaliente a distintas orientaciones con campos receptivos locales. Éstas células receptivas simples se pueden corresponder con los kernels de convolución usados en las redes convolucionales por la sencillez y la localidad de su campo de receptividad.

Posteriormente, las redes convolucionales ganaron una gran popularidad debido a su rendimiento en tareas de clasificación de imágenes y detección y reconocimiento de objetos en imágenes. El primer hito de su capacidad para procesar imágenes de manera efectiva fue en concurso de clasificación de imágenes de ImageNet, donde el equipo de Geoffrey Hinton logró superar el mejor resultado en precisión de clasificación por un gran margen usando una arquitectura de red convolucional [16]. En este trabajo, se pudo apreciar con gran detalle las ventajas del enfoque del aprendizaje de representaciones internas en una red convolucional.

Tal como se puede apreciar en la Figura(2.15), en la primera capa convolucional, los kernels

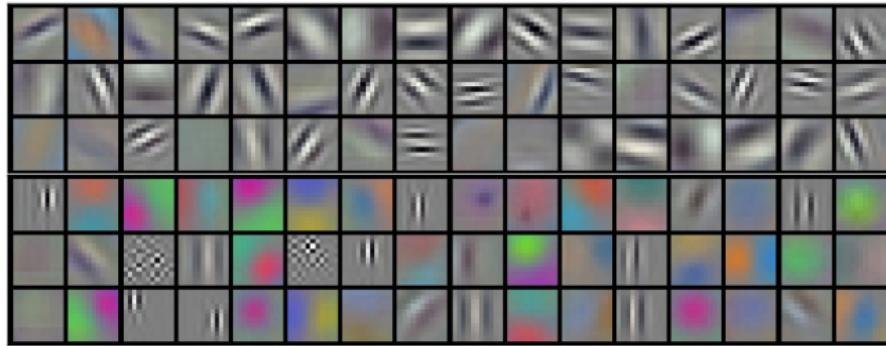


Figura 2.15: Kernels convolucionales de tamaño $11 \times 11 \times 3$ en la primera capa convolucional.
Fuente: [16]

de convolución corresponden con representaciones básicas en una imagen como la búsqueda de bordes en distintas orientaciones, esto va acorde a lo establecido anteriormente en el modelo de la corteza visual de un gato. Puede decirse entonces que las redes convolucionales emulan, en cierto modo, al proceso biológico de visión en animales.

2.3.6. Sistemas de Aprendizaje Fin a Fin

Los sistemas de aprendizaje fin a fin refieren a sistemas complejos que se conciben, modelan y entranan como un conjunto para resolver cierto problema en vez de tratarse de forma separada por módulos. En términos de redes neuronales, los sistemas de aprendizaje fin a fin refieren a redes neuronales complejas, compuestas por distintos componentes y tipos de unidades que usualmente tratan un problema de aprendizaje en su totalidad.

En contraste con el desarrollo de los sistemas de aprendizaje tradicionales, tal como se pudo apreciar en la Sección(1.3.2) donde cada etapa del proceso debía ser cuidadosamente diseñada y evaluada por alguien con la experiencia suficiente en el área, los sistemas fin a fin, dejan la responsabilidad de encontrar las representaciones internas adecuadas al proceso de entrenamiento de la red neuronal en sí. Las representaciones adecuadas son generadas a partir del proceso de aprendizaje y ajuste de pesos en las distintas capas ocultas de la red neuronal. Usualmente, las redes neuronales para aplicaciones de aprendizaje fin a fin se componen de varias etapas de distinta naturaleza, de acuerdo al problema que se intenta resolver.

Los sistemas de aprendizaje fin a fin han demostrado ser una alternativa bastante llamativa debido a que presentan las siguientes ventajas:

- **Diseño simplificado.** El diseño de un sistema fin a fin es mucho menos demandante en cuestión de tiempo y conocimiento experto, la extracción de características no necesita ser una etapa elaborada por un experto pues se generará automáticamente.
- **Entrenamiento centralizado.** Dado que se trata de una sola red neuronal profunda

que se incorporará en el flujo de trabajo, el entrenamiento de la misma se realiza de forma única para todo el sistema.

- **Flexibilidad.** Debido a que el proceso de diseño solamente introduce criterios generales para la tarea deseada, los sistemas fin a fin pueden ser fácilmente adaptados para realizar otras tareas en el futuro. Las iteraciones en el diseño y entrenamiento no requieren de mucho tiempo ni esfuerzo.
- **Modularidad.** La naturaleza de las representaciones internas generadas en las redes neuronales profundas permite que las mismas se puedan reutilizar en otras tareas o arquitecturas.

Gracias a las ventajas anteriormente descritas, los avances en el desarrollo de sistemas fin a fin han tenido particular éxito en las siguientes áreas:

- Reconocimiento de voz [22].
- Control de brazos robóticos [23].
- Conducción autónoma. [3].

En el presente proyecto, se utilizarán los conceptos de aprendizaje profundo, redes convolucionales y aprendizaje fin a fin para diseñar un sistema de conducción autónoma.

2.4. Robots móviles y locomoción con ruedas

Un robot móvil es una máquina electromecánica con ciertos niveles de autonomía definida que son capaces de moverse y navegar por un entorno. Las ruedas aprovechan la fricción y el contacto con el suelo para hacer que el robot pueda moverse. Si se considera un vehículo ideal, donde las ruedas no se deslizan hacia los costados y mantienen contacto con el suelo en todo momento, para que exista un movimiento debe existir un punto alrededor del cual cada rueda sigue una trayectoria circular. Este punto es conocido como el Centro Instantáneo de Rotación o Curvatura (ICC). En la práctica, el ICC se puede identificar de manera sencilla porque es el punto que yace en una línea coincidente con el eje de rotación de cada rueda. El ICC define la curvatura de la trayectoria que sigue el robot en cada momento. En la Figura(2.16), se puede apreciar una configuración compatible con el movimiento sin deslizamiento de las ruedas y una configuración incompatible.

La existencia del ICC es una condición necesaria pero no suficiente para el movimiento de un robot con ruedas, también las velocidades de cada rueda deben ser consistentes con la rotación rígida del vehículo en su conjunto. Un robot localizado en un plano tiene tres grados de libertad: una posición (x, y) y una orientación θ . Este conjunto de parámetros (x, y, θ) es comúnmente denominado la pose del robot en el plano.

Los robots móviles usualmente no poseen un control absoluto sobre los tres parámetros de su pose y deben realizar diversas maniobras para poder alcanzar una pose en particular. Estas restricciones en el control de su movimiento se conocen como restricciones no holonómicas.

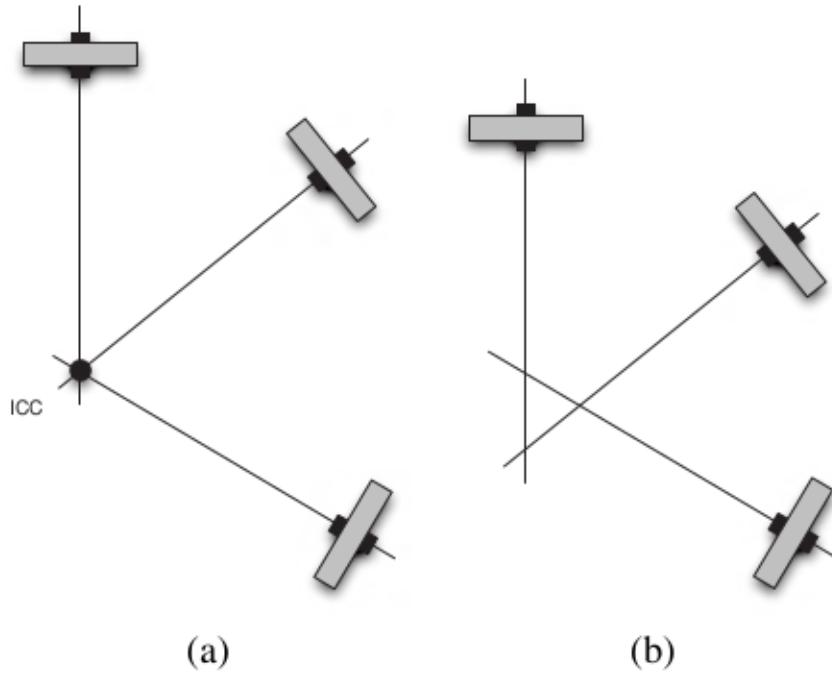


Figura 2.16: Centro instantáneo de curvatura o ICC. (a) Arreglo compatible con el movimiento, (b) arreglo incompatible con el movimiento. Fuente: [16]

2.4.1. Modelo de locomoción de Ackerman

Es el tipo de locomoción encontrada en la mayoría de los automóviles domésticos. En este modelo, las ruedas frontales están direccionaladas y pueden rotar distintos ángulos de tal manera que sus ejes de rotación se intersectan en el ICC. La rueda direccionalada interna debe rotar un ángulo un poco mayor que la externa para que la condición se cumpla Figura(2.17).

La locomoción de Ackerman es la preferida en vehículos de gran tamaño, los cuales están diseñados para operar en carreteras existentes y soportar una carga considerable. Normalmente, el tamaño del vehículo permite incorporar una gran cantidad de sensores y sistemas de control.

2.4.1.1. Cinemática directa

El vehículo rota alrededor de un punto que yace en la linea que pasa por el eje trasero una distancia R del centro del vehículo donde

$$R - l/2 = d \tan(\pi/2 - \alpha_l)$$

Para que las ruedas puedan rodar la segunda rueda de dirección debe ser rotada un ángulo α_r , donde

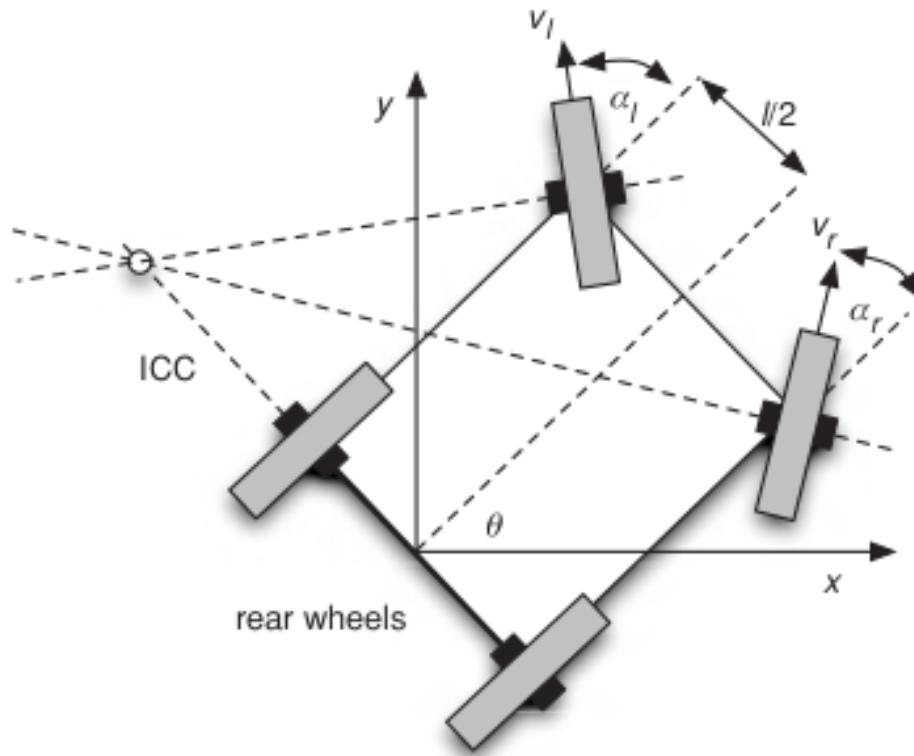


Figura 2.17: Modelo cinemático de Ackerman. Fuente: [24]

$$R + l/2 = d \tan(\pi/2 - \alpha_r)$$

En general, las cuatro ruedas viajan por el suelo a velocidades diferentes y al especificar la velocidad de una sola rueda se puede obtener las velocidades de las restantes. El modelo de locomoción de Ackerman es muy sofisticado y ha sido estudiado ampliamente. Para los objetivos del presente proyecto, se puede simplificar el modelo sin perder ninguna propiedad importante.

2.4.2. Modelo de la bicicleta o triciclo

Los modelos de la bicicleta y el triciclo tienen modelos cinemáticos muy similares. Un triciclo típico tiene tres ruedas: dos ruedas de tracción trasera y una rueda de dirección delantera. El movimiento del robot es controlado por la dirección α y la velocidad v .

2.4.2.1. Cinemática directa

Si la rueda de dirección se orienta en un ángulo α de la dirección hacia adelante, el triciclo rotará con una velocidad angular ω con respecto al punto que yace a una distancia R de la

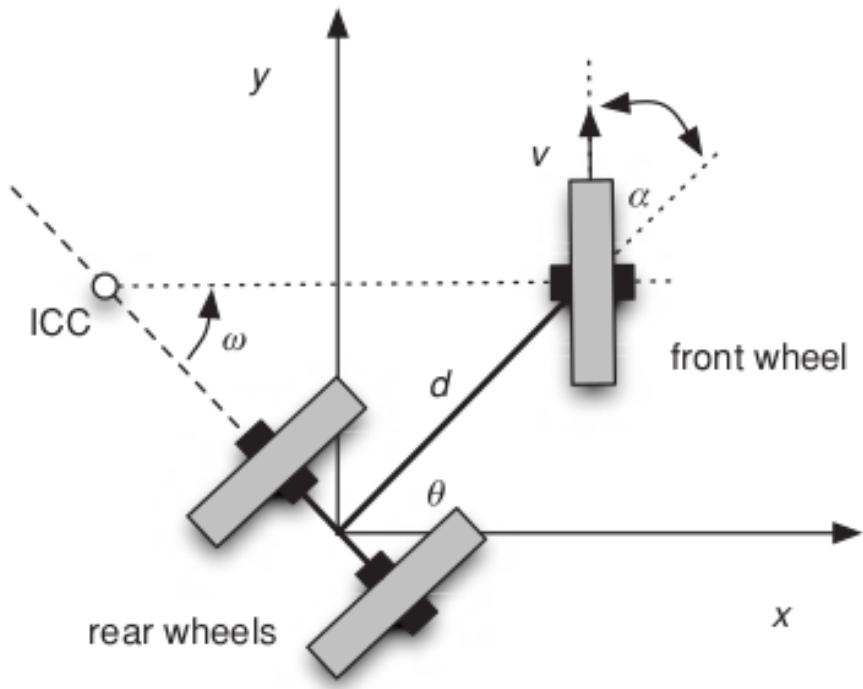


Figura 2.18: Modelo cinemático del triciclo. Fuente: [24]

línea perpendicular que pasa por el eje trasero donde R y ω están dados por las siguientes expresiones:

$$R = d \tan\left(\frac{\pi}{2} - \alpha\right) \quad (2.26)$$

$$\omega = \sqrt{\frac{v}{d^2 + R^2}} \quad (2.27)$$

donde d es la distancia desde la rueda frontal al eje trasero, como se observa en la Figura(2.18).

En el presente proyecto, se considerará el uso del modelo del triciclo definido en la Sección(2.4.2) por razones de simplicidad sin perder generalización.

Capítulo 3

Ingeniería del proyecto

3.1. Arquitectura del sistema

En la presente sección se procede a detallar la arquitectura del sistema de forma general analizando cada uno de los subsistemas, sus funcionalidades, componentes y características. Posteriormente se detallarán los detalles técnicos y de implementación de cada subsistema. Para comenzar, es necesario describir la visión general del sistema, la finalidad y alcance del mismo.

3.1.1. Visión general

Tal como se ha establecido en el Capítulo(1), el objetivo del presente proyecto es el de diseñar un sistema de aprendizaje fin a fin para la tarea de conducción autónoma en vehículos domésticos. Este sistema ha sido diseñado con la finalidad de plantear una alternativa para el desarrollo de sistemas de conducción autónoma en especial en el subsistema de inferencia y control autónomo. No obstante, se ha desarrollado un prototipo completamente funcional de un vehículo autónomo que cumple la tarea de seguir una carretera y detenerse cuando un obstáculo se interpone de manera autónoma.

Es importante destacar que este proyecto también brinda un conjunto de herramientas de software y hardware de manera que se pueda replicar el mismo de forma fácil y con un presupuesto reducido. Si bien el presente proyecto se centra en el desarrollo del sistema de visión artificial para la generación de comandos de dirección usando una red neuronal convolucional, la naturaleza modular de la arquitectura del mismo permite realizar cambios o mejoras en cada uno de los subsistemas. Estos cambios y mejoras se pueden introducir aprovechando la naturaleza modular de los nodos de ROS y la infraestructura de comunicación presente en el sistema pudiendo agregarse más de un sistema de control en el mismo, como por ejemplo, un sistema de reconocimiento de peatones o señales de tránsito.

3.1.2. Esquema del sistema

Se procede a detallar el esquema general del sistema en base a la interacción de tres subsistemas básicos en la Figura(3.1).

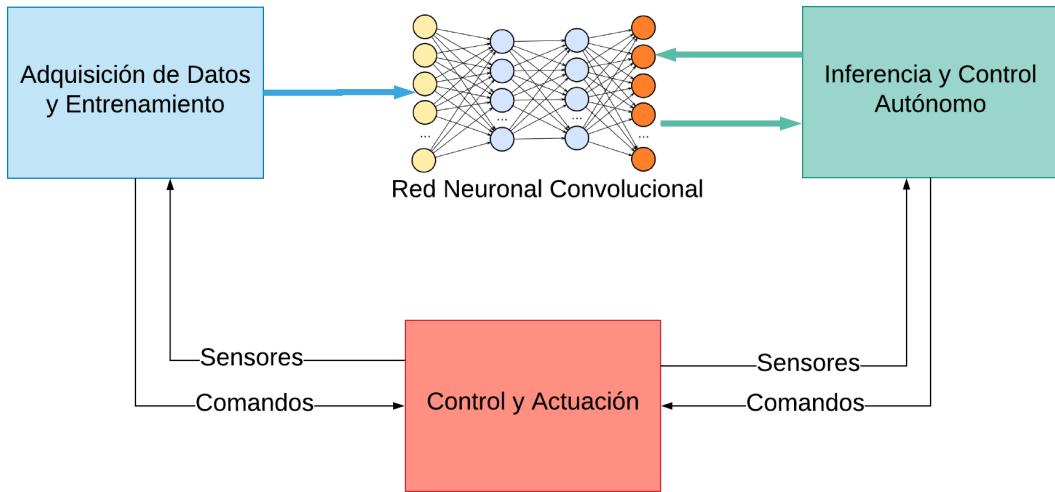


Figura 3.1: Esquema en diagrama de bloques del sistema. Fuente: Elaboración propia.

En el esquema, se puede observar la interacción entre los subsistemas que componen el sistema en su conjunto haciendo un énfasis en la comunicación entre el subsistema de control y actuación y los otros dos subsistemas. El sistema de control y actuación representa la plataforma sobre la cual se van a ejecutar los otros dos subsistemas, el de adquisición de datos y entrenamiento en primer lugar, para el entrenamiento de la red neuronal y seguidamente el de inferencia y control autónomo en el que se usará la información generada por el primero para lograr la tarea de conducción autónoma.

Este esquema es particularmente popular y bien establecido en muchos sistemas de aprendizaje automático en el que se cuenta, por un lado, con una forma de obtener y adecuar los datos para el entrenamiento de la red y, por el otro, se tiene una etapa de inferencia o predicción en la cual se valida la eficacia del modelo. En este caso, la planta o el sistema donde se puede validar la eficacia de la red neuronal es el prototipo con el vehículo.

En las siguientes secciones se detalla la arquitectura de cada subsistema exponiendo sus diagramas de bloques asociados.

3.1.3. Subsistema de control y actuación

El subsistema de control y actuación tiene el objetivo de servir como base física para la implementación de los algoritmos de control. En la Figura(3.2) se puede apreciar sus componentes y la forma en que interactúan entre sí. Este subsistema cuenta con varios módulos funcionales que se encargan del control y actuación del vehículo. Los módulos tienen la ta-

rea de brindar una plataforma para que los algoritmos de control se puedan ejecutar en el vehículo.

Este subsistema cuenta con módulos que tienen tres principales responsabilidades.

- Ejecutar un control de tiempo real en los actuadores disponibles para la locomoción del vehículo a través de un sistema embebido.
- Brindar una plataforma para la adquisición de los datos de los sensores (la cámara y el sensor de proximidad).
- Brindar una plataforma de desarrollo para el control autónomo del vehículo mediante ROS (Sección(3.2.1)).

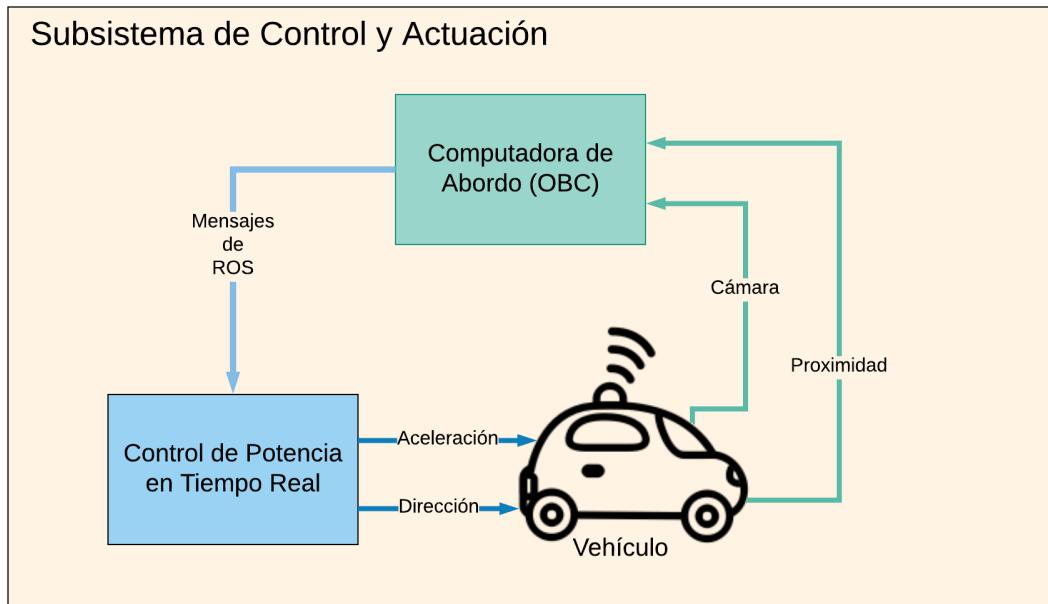


Figura 3.2: Esquema en diagrama de bloques del Subsistema de control y actuación. Fuente: Elaboración propia.

3.1.4. Subsistema de adquisición de datos y entrenamiento

El subsistema de adquisición de datos y entrenamiento tiene la finalidad de proporcionar las herramientas necesarias para dos tareas fundamentales:

- Generar un conjunto de datos o *dataset* para el entrenamiento y validación de la red neuronal.
- Entrenar una red neuronal a partir del conjunto de datos y ciertos parámetros previamente definidos.

Para tal cometido, este subsistema cuenta con varios módulos que interactúan entre sí y brindan distintas funcionalidades. En la Figura(3.3). Es importante destacar que la naturaleza de los módulos de este subsistema hicieron que los mismos se deban ejecutar en una estación de trabajo remota con características especiales para las tareas necesarias. Esta forma de trabajo se ha adoptado para disminuir el tiempo necesario en la tarea del entrenamiento de la red neuronal la cual es una tarea que demanda mucho poder computacional.

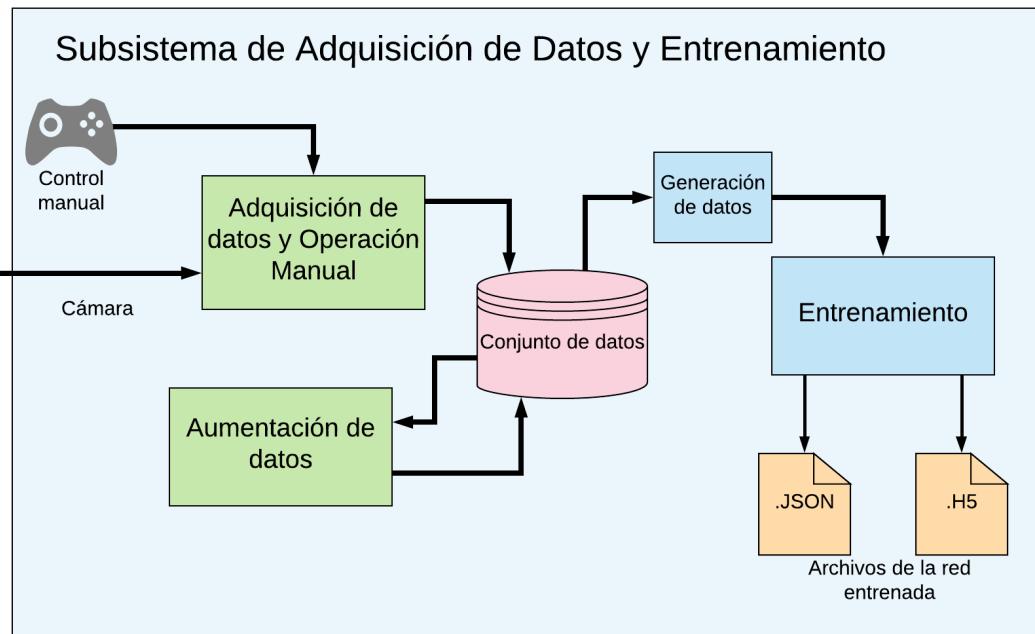


Figura 3.3: Esquema en diagrama de bloques del Subsistema de Adquisición de datos y Entrenamiento. Fuente: Elaboración propia.

El producto de este subsistema es un conjunto de archivos que representan tanto la arquitectura de la red neuronal como los datos de los pesos obtenidos en el proceso de entrenamiento. Dichos archivos serán utilizados en el subsistema de inferencia y control autónomo.

3.1.5. Subsistema de inferencia y control autónomo

En este subsistema se concentra la mayor complejidad del sistema siendo que contiene los módulos correspondientes con la etapa de inferencia de la red neuronal en el bucle de control, así como también un módulo encargado de enviar comandos de control para la aceleración que toman en cuenta la detección de obstáculos que se presenten al frente del vehículo mientras navega por su entorno.

El subsistema debe cumplir las siguientes tareas:

- Cargar y ejecutar el modelo de predicción implementado en la red neuronal entrenada en el subsistema de adquisición de datos y entrenamiento para la generación de comandos

de dirección del vehículo.

- Ejecutar un algoritmo de control para la aceleración basado en la detección de obstáculos presentes frente al vehículo.
- Ejecutar un algoritmo de arbitraje que combine ambos sistemas de control en conjunto con un control manual de respaldo que será referido como el *piloto automático*.

En la Figura(3.4) se puede observar el esquema en diagramas de bloques del subsistema con la interacción entre los módulos que lo componen.

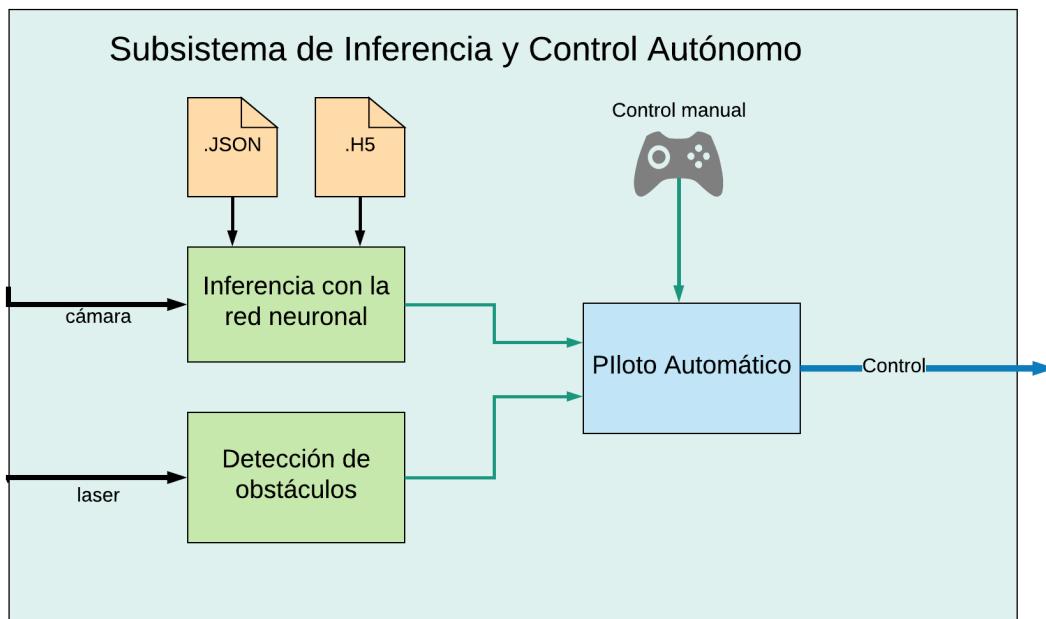


Figura 3.4: Esquema en diagrama de bloques del Subsistema de Inferencia y Control Autónomo. Fuente: Elaboración propia.

Este subsistema depende de los otros dos subsistemas de la siguiente manera:

- El subsistema de adquisición y entrenamiento es el encargado de generar los archivos de la red neuronal que se usan en este subsistema.
- El subsistema de control y actuación es el encargado de generar los datos de los sensores y de proveer una plataforma para el control del vehículo.

El presente proyecto presenta en este subsistema una alternativa a otros métodos tradicionales para sistemas de visión artificial para tareas de control autónomo.

3.2. Herramientas de software

3.2.1. Robot Operating System - ROS

ROS o Sistema Operativo Robótico es un *framework* flexible para desarrollar software para robots. Se compone de una colección de herramientas, librerías y convenciones que tienen el objetivo de simplificar la tarea de crear comportamientos complejos y robustos en plataformas de robótica en general [25].

ROS ha sido construido con el objetivo de hacer accesible el desarrollo de sistemas robóticos mediante el trabajo colaborativo de paquetes y utilidades, su naturaleza modular hace posible que se puedan implementar sistemas pieza por pieza de acuerdo a las necesidades específicas de cada proyecto. Dentro de las facilidades que ROS ofrece, se listan a continuación diversas utilidades que permiten el desarrollo de sistemas con una complejidad elevada.

3.2.1.1. Infraestructura de comunicación

En su núcleo, ROS ofrece una interfaz de intercambio de mensajes que provee comunicación inter-procesos y es comúnmente referida como el *middleware*. El *middleware* de ROS ofrece las siguientes facilidades:

- Intercambio de mensajes mediante publicación/subscripción y tópicos.
- Registro y reproducción de mensajes.
- Llamadas a procedimientos del tipo request/response.
- Sistema de administración distribuido de parámetros.



Figura 3.5: Diagrama de comunicación de nodos usando mensajes. Fuente: [26]

La naturaleza distribuida de ROS y las facilidades que ofrece el *middleware*, hacen que el desarrollo de sistemas robóticos modulares sea una tarea trivial. Aparte de la infraestructura de comunicación, ROS ofrece otras características especialmente diseñadas para el desarrollo de robots.

3.2.1.2. Nodos de ROS

Un nodo en ROS representa a un proceso que se ejecuta de manera independiente y puede aprovechar las facilidades de la infraestructura de comunicación de ROS. Los nodos pueden publicar o suscribirse a uno o varios tópicos.

Para que los nodos puedan comunicarse entre sí de manera distribuida y coordinada, existe un proceso especial llamado el ROS master, que registra a todos los nodos y establece los canales de comunicación entre ellos usando los tópicos y mensajes a los cuales los nodos se suscriben y publican.

Una característica importante de los Nodos de ROS es que se pueden ejecutar de manera distribuida sobre una red local, lo que implica que se pueden desarrollar sistemas robóticos distribuidos en varias computadoras o estaciones de trabajo. Esto es muy útil en casos donde las características de tamaño, peso y consumo de energía de un robot móvil impiden la inclusión de una computadora con las características de memoria y velocidad de procesamiento necesarias para tareas de procesamiento de imágenes o inteligencia artificial, pudiendo ejecutar los nodos con altos requerimientos computacionales en una estación de trabajo remota.

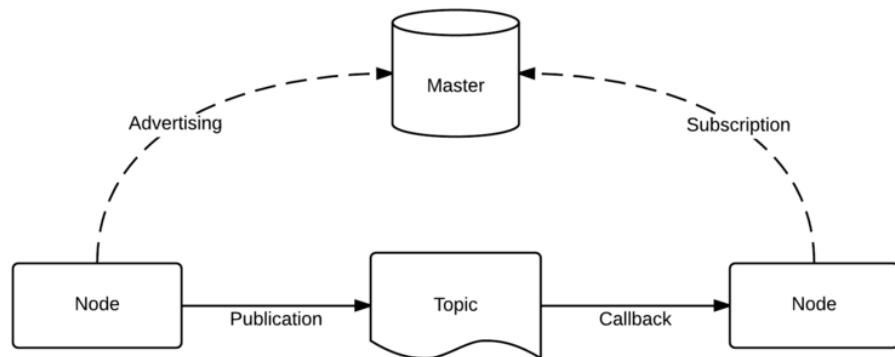


Figura 3.6: Esquema de la comunicación entre nodos. Fuente: [26]

3.2.1.3. Características específicas para robótica

Adicionalmente a los componentes del *middleware*, ROS tiene a disposición librerías y herramientas específicas para el desarrollo rápido de sistemas robóticos. Algunas de las características más importantes se listan a continuación:

- Definiciones de mensajes estándar para robots.
- Lenguaje de descripción de robots URDF¹.

¹URDF: Universal Robot Description Format

- Herramientas de diagnóstico.
- Localización.
- Mapeo.
- Navegación.
- Drivers de sensores y actuadores.

3.2.1.4. Herramientas adicionales

Una de las características más atractivas de ROS es el conjunto de herramientas para desarrollo. Estas herramientas soportan análisis, depuración y visualización del estado del sistema que está siendo desarrollado. Los mecanismos presentes de publicación y suscripción permiten analizar de manera espontánea el flujo de datos en el sistema. Las herramientas de ROS aprovechan esta característica y se presentan como una colección de herramientas gráficas y de línea de comandos que simplifican el desarrollo y depuración de robots.

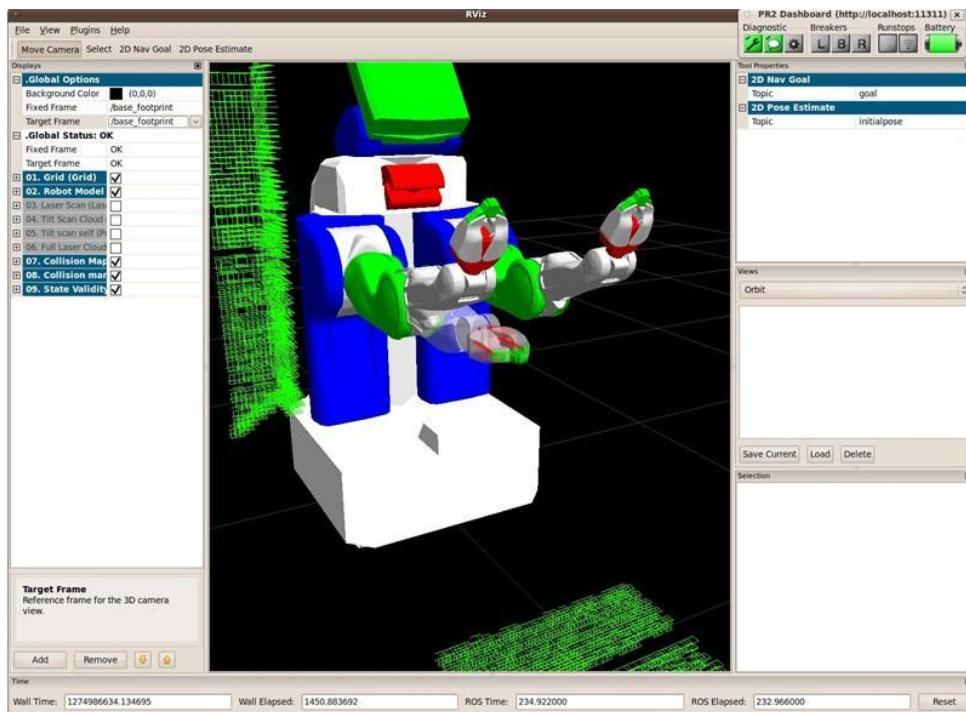


Figura 3.7: Interfaz de visualización de ROS rviz. Fuente: [26]

- **Herramientas de Línea de Comandos.** Permiten el control y depuración de los sistemas de manera remota en una interfaz de línea de comandos. Existen comandos disponibles para ejecutar procesos, analizar tópicos y mensajes, grabar y reproducir sesiones de mensajes y ejecutar servicios.

- **Rviz.** Es una interfaz de visualización de diversas fuentes de datos y modelos de robots. Con la herramienta rviz es posible visualizar diversos tipos de mensajes provenientes de sensores tales como cámaras o sensores láser. También es posible agrupar los distintos tipos de visualizaciones de manera jerárquica en la misma ventana.
- **Rqt.** Rqt es un *framework* para el desarrollo de interfaces gráficas para robots. Con rqt es posible crear interfaces de control o monitoreo de manera gráfica y personalizada usando componentes llamados plugins.

3.2.1.5. Criterios de selección

En el marco del presente proyecto y el tiempo establecido para su desarrollo se ha basado la selección del entorno de trabajo en base a los siguientes criterios:

- **Interfaz de comunicación distribuida.** Es necesario que se puedan desarrollar componentes del sistema de manera independiente y puedan ser ejecutados de la misma manera. ROS ofrece mediante el desarrollo de paquetes y nodos la facilidad de poder ejecutar y comunicar procesos de manera sencilla y distribuida a través del intercambio de mensajes.
- **Implementación de funcionalidades comunes.** También se necesita una plataforma con funcionalidades básicas implementadas y disponibles para su uso, esto con el fin de concentrar el tiempo de desarrollo en las funcionalidades del sistema en su conjunto más que en la plataforma sobre la cual se va a desplegar. Se necesitan herramientas reutilizables para evitar lo que comúnmente se denomina como *reinventar la rueda*.
- **Uso libre y código abierto.** ROS es una plataforma de código abierto, lo que permite utilizarlo de manera libre ya sea para proyectos académicos y comerciales. Además, su naturaleza open source permite también realizar cambios o mejoras en su funcionalidad de manera sencilla. El uso libre es importante dado que en entornos académicos normalmente no se cuenta con la facilidad de adquirir licencias de software privativo. El uso libre también permite el desarrollo por parte de investigadores independientes y estudiantes que no pertenecen a alguna institución que pueda apoyarlos financieramente.
- **Facilidad de uso.** El entorno de trabajo debe tener la facilidad de ser accesible para personas con un conocimiento previo en electrónica y programación. Tanto los lenguajes de programación como las herramientas de desarrollo, compilación y despliegue tienen que estar disponibles y ser fáciles de utilizar.
- **Compatibilidad con herramientas externas.** En el marco del proyecto y la aplicación de los conceptos de visión artificial y aprendizaje profundo. El entorno de trabajo debe ser compatible o poder extender sus funcionalidades con otros entornos dedicados al procesamiento de imágenes y visión artificial como a entornos y librerías para el desarrollo y entrenamiento de redes neuronales.

- **Interfaces con sistemas de bajo nivel y tiempo real.** Es necesario que la plataforma también sea compatible con el desarrollo de sistemas embebidos y de tiempo real para el control de actuadores y sensores que no se pueden conectar a una PC directamente.

Es en este sentido que se ha escogido usar al *framework* ROS como plataforma de desarrollo para los distintos módulos del sistema. Cabe resaltar que ROS no es la única plataforma para desarrollar robots, y algunas alternativas se detallan en la Tabla(3.1) donde se puede analizar las características de cada una.

Nombre	Interfaz de Comunicación Distribuida	Sistema de compilación	Gestión de paquetes	Drivers de bajo nivel	Lenguajes de programación
ROS	SI	SI	SI	SI	C++ Python Java
YARP	SI	NO	NO	SI	C++
ROCK	SI	SI	NO	NO	C++
MRTP	NO	NO	NO	SI	C++
Player	SI	SI	NO	NO	C++
Robotics Library	NO	NO	NO	SI	C++

Tabla 3.1: Tabla comparativa de características entre distintas plataformas y librerías para desarrollo de sistemas robóticos. Fuente: Elaboración propia

ROS se usa de manera extensiva en el desarrollo del presente proyecto para las siguientes tareas:

- En el subsistema de control y actuación como una interfaz común de intercambio de mensajes para el control de los motores presentes en el prototipo, así como también en la recuperación de los datos de los sensores. Estas interfaces están implementadas como nodos de ROS.
- En el subsistema de adquisición de datos y entrenamiento como una herramienta de captura de información del control manual y la cámara, tomando en cuenta las estampas de tiempo y sincronización para cada mensaje de ROS.
- En el subsistema de inferencia y control autónomo como la plataforma sobre la cual se definen los distintos controladores como nodos de ROS y el programa del piloto automático como un árbitro entre los mensajes de los distintos controladores.

- En todo el sistema como la interfaz de comunicación distribuida a través del intercambio de mensajes entre el prototipo y la estación de trabajo remota.

3.2.2. Tensorflow

Tensorflow es una librería para cálculos numéricos que funciona en base a grafos de flujo de datos Figura(3.8). Las operaciones matemáticas se representan como nodos en el grafo y los vértices representan matrices de datos multidimensionales o tensores que fluyen de un nodo a otro [27]. Debido a esta implementación, los grafos pueden ejecutarse de manera distribuida en varias CPU o GPU. Las operaciones matemáticas están disponibles para utilizar en la librería y sus implementaciones están altamente optimizadas, lo que permite aprovechar al máximo el hardware disponible.

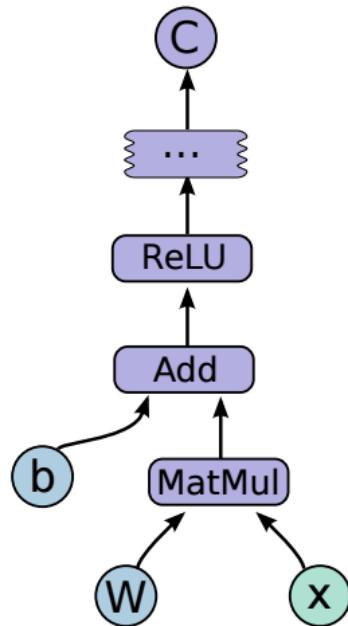


Figura 3.8: Ejemplo de un grafo de cómputo utilizado en Tensorflow. Fuente: [28]

Tensorflow se ha hecho popular por la facilidad con la que se puede implementar la arquitectura de una red neuronal usando grafos de cómputo y por la optimización de los algoritmos usados. Actualmente, Tensorflow representa el estándar en la implementación de redes neuronales profundas tanto en la academia como la industria.

Otra de las características de Tensorflow es que presenta una API en el lenguaje de programación Python, lo que permite el desarrollo de redes neuronales de manera muy sencilla e intuitiva.

En el presente proyecto, se utiliza Tensorflow como librería base para la implementación de la red neuronal tanto en la etapa de entrenamiento como en la etapa de inferencia. El

entrenamiento e inferencia se implementan usando los algoritmos de Tensorflow optimizados para GPU² de la marca Nvidia.

Es importante listar algunos términos que se usarán en el contexto de este proyecto, relacionados exclusivamente con la implementación de la red neuronal convolucional correspondiente con el sistema fin a fin que se implementa.

- **Tensor:** Es una generalización de un vector o una matriz en dimensiones superiores. Internamente, Tensorflow representa tensores como arreglos n-dimensionales de tipos de datos base, como ser Int32 o Float64.
- **Variable:** Refiere a la manera de presentar el estado persistente que se puede manipular por el programa o grafo de cómputo. Una variable contiene internamente un tensor con valores que se pueden modificar mediante operaciones. Las variables en Tensorflow comúnmente se utilizan para representar a los pesos o parámetros de la red neuronal.
- **Grafo:** Un grafo es un objeto de Tensorflow que contiene la información acerca de la estructura del grafo de cómputo que se va a utilizar. Contiene la información de las distintas operaciones y las conexiones entre las mismas por las que fluyen los tensores. La estructura del grafo debe ser declarada antes de su ejecución.
- **Operación:** Una operación representa a un nodo en el grafo, tiene como entrada uno o varios tensores y produce como salida uno o varios tensores. Las operaciones definen los cálculos que se realizan entre tensores como ser una multiplicación de matrices o una operación de convolución, entre otras.

En el siguiente ejemplo, se puede observar la definición de un grafo de cómputo básico en Tensorflow:

```
import tensorflow as tf
#definicion de variables
input1 = tf.Variable(3.0)
input2 = tf.Variable(2.0)
input3 = tf.Variable(5.0)

#definicion de las operaciones y el grafo
intermed = tf.add(input2,input3)
mul = tf.mul(input1,intermed)

#ejecucion de las operaciones
with tf.Session() as sess:
    result = sess.run([mul,intermed])
    print(result)
```

Ejemplo de un programa escrito con Tensorflow

²GPU: Graphics Processing Unit o Unidad de Procesamiento de Gráficos

3.2.3. Keras

Keras es una librería para la definición e implementación de redes neuronales de alto nivel escrita en Python y compatible con diversas plataformas de cómputo tales como Tensorflow, CNTK o Theano [29]. Esta librería ha sido desarrollada con el objetivo de facilitar la experimentación y prototipado rápido de modelos de aprendizaje profundo. Las características de la librería que la convierten en una opción viable en el desarrollo de modelos de aprendizaje profundo son las siguientes:

- Permite el prototipado rápido a través de su facilidad de uso, modularidad y capacidad de ser extendida.
- Soporta la definición de redes neuronales recurrentes y redes neuronales convolucionales. La última categoría es la más importante para el presente proyecto.
- Soporta la ejecución tanto en CPU como en GPU.

Keras se basa en la definición de redes neuronales en base a capas. Existe una clase especial de modelo llamado *Sequential* que representa básicamente una red neuronal feedforward (Sección(2.3.2.1)). En un modelo *Sequential* se define a la red en base a las capas de las que se compone, cada capa puede tener distinta naturaleza y características.

Además de la definición de las capas, Keras también cuenta con implementaciones de algoritmos de optimización y funciones de costo comúnmente utilizadas en trabajos de investigación en la actualidad, lo cual facilita todavía más el desarrollo de modelos de redes neuronales. En el siguiente ejemplo, se puede apreciar la definición de la red neuronal de dos capas definida en la Ecuación(2.7) con 32 unidades en la capa de entrada y 4 unidades en la capa de salida, con una función de costo de entropía cruzada categórica y el algoritmo de optimización de *Stochastic Gradient Descent*:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

modelo = Sequential()
#primera capa
model.add(Dense(32), input_dim=128)
model.add(Activation('sigmoid'))
#segunda capa
model.add(Dense(4), input_dim=128)
model.add(Activation('sigmoid'))
#optimizador y funcion de costo
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy']
)
```

Ejemplo de una red neuronal usando la librería Keras

3.2.4. ARM Mbed

Mbed es una iniciativa llevada adelante por ARM que brinda un conjunto de herramientas de hardware y software para el desarrollo de dispositivos IoT (Internet de las Cosas). Mbed es un ecosistema de desarrollo sobre el cual se pueden desarrollar aplicaciones con microcontroladores con arquitectura ARM provenientes de distintos fabricantes [30]. La característica principal de Mbed es la sencillez de su uso y la amplia gama de librerías disponibles para distintos componentes de hardware como sensores, actuadores o displays. ARM Mbed presenta las siguientes características clave para el desarrollo de sistemas embebidos de manera rápida y favorables al contexto del presente proyecto:

- Variedad de placas de desarrollo de microcontroladores ARM de distintos fabricantes.
- Una interfaz de programación común a todos los microcontroladores y fabricantes para la interfaz con periféricos embebidos.
- Un compilador en línea donde se pueden crear, compilar y desplegar proyectos.
- Variedad de librerías para dispositivos como sensores, módulos de comunicación o actuadores.
- El uso del lenguaje C++ con la especificación completa hace posible el desarrollo orientado a objetos para aprovechar altos niveles de abstracción en la programación.
- La capacidad de generación de símbolos de depuración para su ejecución paso por paso con el fin de identificar *bugs* en tiempo de ejecución.
- La compatibilidad con la librería Rosserial que hace posible poder comunicar al microcontrolador con el *middleware* de ROS de manera directa usando un puerto serial.

Mbed se usará como la plataforma para el desarrollo del control de tiempo real en el subsistema de control y actuación ya que ofrece todas las facilidades en cuanto a librerías y potencia computacional necesarias para esta tarea. En la Tabla(3.2) se puede observar una tabla comparativa entre diversas plataformas de desarrollo embebido consideradas para el presente proyecto.

Una de las plataformas de desarrollo más utilizadas en la actualidad es Arduino, pese a haberse considerado esta plataforma por su disponibilidad, popularidad y gran soporte por la comunidad se ha detectado algunas limitaciones en la misma que hacen que no se la pueda recomendar para desarrollos académicos:

- Si bien el nivel de abstracción facilita la introducción a los microcontroladores para personas sin experiencia, oculta varios aspectos referidos al hardware de los periféricos del microcontrolador que escapan de control. Esta falta de control de bajo nivel puede ocasionar fallos y situaciones en las que no se pueda predecir con seguridad el comportamiento de un sistema. La predictibilidad es una característica fundamental en cualquier sistema de tiempo real.

Plataforma	Arquitectura	Lenguaje	Nivel de abstracción	Enfoque
Arduino	AVR - 8 bits	Wiring (C++)	Alto	Hobby, Arte y Educación
Mbed	ARM - 32 bits	C++	Alto	IoT, Sistemas de tiempo real
Energia	TI MSP430 - 16 bits TI ARM - 32 bits	Wiring (C++)	Alto	Hobby, Sistemas de tiempo real
Freedom E SDK	RISC V - 32 bits	C	Bajo	Sistemas de tiempo real
libOpenCM3	ARM - 32 bits	C	Bajo	Sistemas de tiempo real

Tabla 3.2: Tabla comparativa de plataformas de desarrollo embebido. Fuente: Elaboración propia

- El lenguaje de programación usado Wiring es un subconjunto del lenguaje C++ que carece de varias funcionalidades y no permite el desarrollo de clases con herencia y polimorfismo implementadas de manera adecuada.
- El entorno de desarrollo integrado proporcionado, el Arduino IDE, es un entorno demasiado limitado para desarrollos de proyectos de mediana y gran envergadura.
- La falta de capacidades de depuración, una limitación de la propia arquitectura AVR imposibilita el análisis del comportamiento en tiempo de ejecución del código y la identificación de posibles *bugs* que puedan aparecer. Esta característica es de vital importancia para el desarrollo de sistemas de seguridad crítica.

3.3. Herramientas de hardware

Se han seleccionado diversas herramientas de hardware para la implementación del sistema de conducción autónoma. Dichas herramientas corresponden con la base física electrónica sobre la cual se ejecutarán las tareas de los tres subsistemas. Se procede a detallar las herramientas utilizadas en el diseño del sistema. Cabe señalar que las hojas de datos relevantes a las herramientas de hardware utilizadas en este proyecto se pueden consultar en el Apéndice(C).

3.3.1. Plataforma de tiempo real

La interfaz de más bajo nivel del sistema es el de la interacción con los actuadores de los motores del prototipo. Esta interfaz debe tener la capacidad de poder comunicarse con la OBC y además de poder cumplir ciertos requisitos de ejecución en tiempo real. Estos requisitos de tiempo real hace que tal comportamiento no se pueda implementar en la OBC pues la misma usa un sistema operativo basado en el kernel GNU/Linux, el cual no cuenta

por defecto con capacidades de tiempo real dura. Por tanto, se ha establecido la necesidad de utilizar una plataforma embebida con una arquitectura más sencilla y con un nivel de predictibilidad mucho mayor al de la OBC. En este caso se usará un microcontrolador con arquitectura ARM Cortex M3.

La arquitectura ARM se ha popularizado bastante en los últimos años principalmente por su característica de tener un conjunto reducido de instrucciones y sencillez en la microarquitectura del procesador en comparación con otras arquitecturas comúnmente encontradas en servidores y computadoras personales como son X86 o PowerPC. Esta simplificación en la arquitectura y la reproducción del conjunto de instrucciones ha permitido que los dispositivos basados en ARM puedan reducir dramáticamente el consumo de energía sin degradar demasiado el rendimiento. Es por eso que ARM, en la actualidad se constituye como la principal arquitectura en dispositivos móviles y de bajo consumo con cientos de millones de dispositivos usándola alrededor del mundo.

Sin embargo, el desarrollo e implementación de esta arquitectura se ha dirigido bastante hacia procesadores de aplicación, presentes en dispositivos como teléfonos inteligentes o tablets. Es por eso que ARM ha presentado una familia de procesadores ARM que están orientados exclusivamente al desarrollo de sistemas embebidos con capacidades de tiempo real y ultra bajo consumo de energía. Esta familia es la familia ARM Cortex-M que cuenta con varias características que la hacen ideal para el desarrollo del sistema en tiempo real requerido para la interfaz con los actuadores del presente proyecto.

Por su parte, dado que ARM no fabrica chips sino más bien vende licencias de la arquitectura a distintas marcas fabricantes, existe una multitud de procesadores usando esta arquitectura de distintos fabricantes, entre los cuales se puede mencionar a ST Microelectronics, Texas Instruments, Nordic, entre otros. Para el presente proyecto, se necesita que el microcontrolador pueda cumplir con las siguientes características.

- Al menos un puerto de comunicación serial.
- Periféricos capaces de generar señales PWM o con los recursos necesarios para emular PWM por software.
- Memoria de datos y de programa suficiente para poder incluir la librería de Rosserial en el mismo.
- Capacidades de depuración y ejecución paso por paso para la identificación de fallas.
- Operación en niveles de tensión compatibles con la OBC.

Es por eso que se ha seleccionado la placa de desarrollo de ST Microelectronics Mucleo f303k8 (Figura(3.9)), tomando en cuenta que cumple con todos los requisitos anteriormente establecidos y además cuenta con las siguientes características:

- Circuito grabador-depurador en la placa, STlink V2.

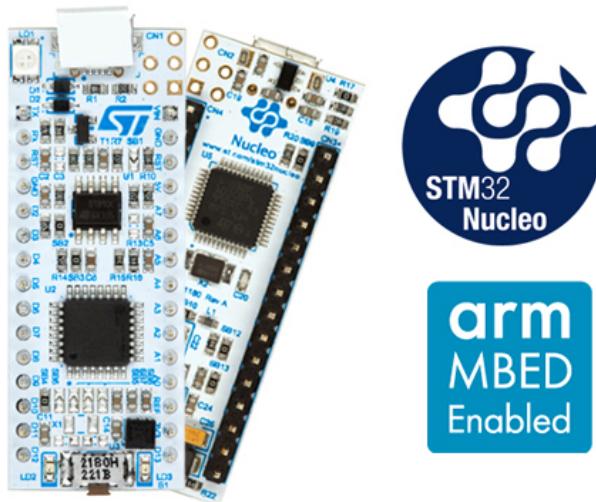


Figura 3.9: Placa de desarrollo Nucleof303k8 de ST Microelectronics. Fuente: [31]

- Capacidades de comunicación seria mediante USB para un puerto de comunicación seria virtual e interfaz de depuración.
- Múltiples fuentes de alimentación.
- Leds indicadores.
- Factor de forma compatible con varios entornos de desarrollo electrónico.

Se han explorado diversas alternativas al uso de la placa de desarrollo Nucleof303k8 para la implementación de este módulo. En la Tabla(3.3) se pueden apreciar las características de varias placas de desarrollo embebido candidatas disponibles en el mercado.

El microcontrolador utilizado, el STM32F303k8 pertenece a la familia de microcontroladores con arquitectura ARM Cortex M4f del fabricante ST Microelectronics. Es parte de la gama de microcontroladores f3 de la familia STM32 que está orientado al procesamiento de señales pues cuenta con un procesador de 32 bits con una unidad de punto flotante que le permite realizar cálculos y operaciones con números decimales de manera eficiente. Las características del microcontrolador seleccionado se listan a continuación:

Como se puede observar en la Tabla(3.4), las características del microcontrolador cumplen con los requisitos establecidos tanto en interfaces de comunicación como en cantidad y variedad de periféricos para una posterior extensión de la funcionalidad. Por otra parte, tal como se puede apreciar en la Tabla(3.3), esta placa es compatible con el entorno de trabajo de programación ARM Mbed, y con la librería Rosserial.

Modelo	Arquitectura	RAM (KB)	ROM (KB)	Frecuencia de Reloj (MHz)	Canales PWM	Entorno de Trabajo
Atmega328p	AVR (8 bits)	2	32	16	6 (8 bits)	Arduino, Atmel Studio
PIC18f2550	PIC (8 bits)	2	32	48	2 (8 bits)	MPlab
STM32f103c8	ARM CortexM3 (32 bits)	20	64	72	12 (16 bits)	STM32 HAL, Arduino
STM32f303k8	ARM CortexM4f (32 bits)	16	64	72	8	STM32 HAL, Mbed

Tabla 3.3: Comparación de características de microcontroladores disponibles. Fuente: Elaboración propia.

Núcleo	ARM Cortex M4 de 32 bits con unidad de punto flotante
Frecuencia de reloj	72 MHz Máximo
Voltaje de operación	2.0 V a 3.6 V (nominal: 3.3 V)
Memoria de datos	16 KB SRAM
Memoria de programa	64 KB FLASH
Timers Disponibles	7
Interfaces de Comunicación	SPI/I2S, I2C, USART, CAN
Periféricos adicionales	GPIO (con interrupciones), ADC, DAC, RTC

Tabla 3.4: Características técnicas del microcontrolador STM32f303k8. Fuente: [31]

Placa	Procesador	RAM	LAN	Conexión Inalámbrica	Soporte y documentación	Precio (\$us)	Soporta ROS
Asus Tinkerboard	4x A17 @ 1.8 GHz	2 GB	GBe	Wifi Bluetooth	Regular	60	NO
Odroid X4u	4x A15 @ 2.0 GHz + 4x A7 @ 1.4 GHz	2 GB	GBe	NO	Bueno	85	SI
Raspberry Pi zero W	1x A8 @ 1GHz	512 MB	NO	Wifi Bluetooth	Muy bueno	20	NO
Beaglebone Black	1x A8 @ 1GHz	512 MB	Fast	NO	Bueno	60	SI
Raspberry Pi 3b+	4x A53 @ 1.4 GHz	1 GB	Fast	Wifi Bluetooth	Muy bueno	40	SI
Rock64	4x A53 @ 1.5 GHz	2 GB	GBe	NO	Regular	35	NO
Beaglebone Blue	1x A8 @ 1GHz	512 MB	no	Wifi Bluetooth	Regular	80	SI

Tabla 3.5: Tabla comparativa de SBC's disponibles en el mercado. Fuente: Elaboración propia.

3.3.2. Computadora de Abordo - OBC

En el caso del hardware necesario para la OBC, se requiere un sistema capaz ejecutar un sistema operativo GNU/Linux completo con el fin de poder correr el software necesario para el control, comunicación y adquisición de imágenes de una cámara, necesarias para el funcionamiento correcto del sistema en su conjunto. Es en este sentido que se ha optado por el uso de una *Single Board Computer* o Computadora de Una Placa, que refiere a placas de desarrollo con todas las características de una computadora de escritorio, es decir: procesador, memoria y periféricos incorporados. Existe gran variedad de SBC en el mercado con distintas características y aplicaciones objetivo. En la Tabla(3.5) se puede apreciar una comparación de varias SBC disponibles en el mercado y sus características relevantes al presente proyecto.

La placa seleccionada como OBC debe cumplir ciertas características específicas para este proyecto:

- Factor de forma: Dimensiones y peso reducidos para poder ser incorporada en el prototipo.
- Consumo de energía: Bajo consumo de energía, es necesario que pueda ser alimentado por baterías disponibles en el mercado.
- Compatibilidad de software: La placa debe poder correr una distribución completa de GNU/Linux reciente, compatible con ROS.
- Compatibilidad de hardware: La placa debe poder conectarse de manera nativa con el microcontrolador elegido para la interfaz de tiempo real. Contar con puertos USB y puertos USART.

- Interfaz con una cámara: Debe contar también con una forma de conectar y adquirir imágenes provenientes de una cámara digital.
- Comunicación inalámbrica: Se necesita la capacidad de poder conectarse a una red LAN mediante Wifi, ya sea con un módulo incorporado o un accesorio externo.
- Precio y disponibilidad: Es importante que el precio no sea demasiado elevado para poder garantizar la replicabilidad del proyecto, así como también la disponibilidad en el mercado.
- Soporte y documentación: Es necesario que la placa cuente con una buena documentación y soporte de la comunidad o fabricante para que el resolver problemas relativos a la placa no tome demasiado tiempo.



Figura 3.10: Placa de desarrollo Raspberry Pi 3 model B+. Fuente: [32]

Al final, se ha seleccionado a la placa Raspberry Pi 3B+ (Figura(3.10)) para la OBC del proyecto pues cuenta con todas las características necesarias para poder ejecutar las herramientas de software requeridas y también cuenta con los periféricos adecuados para comunicarse con la plataforma de tiempo real. La placa en cuestión cuenta con las características listadas en la Tabla(3.6)

3.3.3. Estación de trabajo

La estación de trabajo se usa para realizar el entrenamiento de la red neuronal en el subsistema de adquisición de datos y entrenamiento. Dada la naturaleza de la tarea del

SoC	Broadcom BCM2837B0
CPU	1.4 GHz 64-bit quad core ARM Cortex-A53
RAM	1 GB LPDDR2 SDRAM
Wifi	Dual-band 802.11ac wireless LAN (2.4GHz and 5GHz) y Bluetooth 4.2.
Video	VideoCore IV 3D
USB	2.0, 4 puertos
Interfaces de Comunicación	SPI/I2S, I2C, USART
Interfaces de cámara	USB webcam, CSI

Tabla 3.6: Características de la placa Raspberry Pi 3 model B+. Fuente: [32]

entrenamiento de una red neuronal, esta estación de trabajo presenta algunos requerimientos especiales:

- Sistema Operativo: Se necesita un sistema operativo basado en GNU/Linux compatible con ROS, Tensorflow y Keras.
- GPU: Con la finalidad de acelerar el tiempo de entrenamiento de la red neuronal convolucional, es altamente recomendable contar con una GPU de la marca Nvidia, compatible con Tensorflow.
- Wifi: Es necesaria una conexión Wifi inalámbrica para poder comunicarse con el prototipo tanto en la etapa de adquisición de datos y entrenamiento, como en la de inferencia para el monitoreo remoto.

Considerando los requisitos, las características de la estación de trabajo seleccionada se listan en la Tabla(3.7).

Modelo	MSI GL62 6qd
CPU	Intel Core i7-6700HQ @ 2.6 GHz x 8 núcleos
RAM	16 GB DDR4 @ 2133 MHz
Wifi	Intel Dual-band 802.11ac wireless LAN (2.4GHz and 5GHz) y Bluetooth 4.2.
GPU	Nvidia GTX950m 2GB VRAM
Sistema Operativo	Ubuntu 18.04 Linux 4.15

Tabla 3.7: Características de la estación de trabajo seleccionada. Fuente: Elaboración propia.

La estación de trabajo se utilizará para el entrenamiento de la red neuronal pues cuenta con los requisitos de memoria RAM y una GPU compatible para la paralelización de los algoritmos de entrenamiento de la librería Tensorflow. Estos algoritmos incluyen el cálculo de gradientes de toda la red y el ajuste de los pesos optimizados para ejecutarse de forma paralela dada la naturaleza matricial de las operaciones involucradas.

3.3.4. Sensores

Un aspecto fundamental en el desarrollo de un sistema de conducción autónoma es la elección adecuada de los sensores, que son los dispositivos que recolectan datos acerca del estado del entorno del vehículo. En el presente proyecto, se están utilizando dos tipos de sensores: Una cámara para recuperar las imágenes de la carretera y un sensor de proximidad para detectar obstáculos al frente del vehículo.

3.3.4.1. Cámara

En el caso de la cámara, existen diversas opciones para poder recuperar imágenes del entorno variando desde la calidad de la imagen que recuperan, velocidad de captura, rango dinámico y otras características. Sin embargo, se debe tomar en cuenta algunas limitaciones impuestas por el sistema en su conjunto:

- La capacidad de procesamiento es limitada en la OBC al tratarse de un sistema de bajo consumo de energía y dimensiones reducidas.
- La capacidad de transferencia en la red está limitada a especificaciones de los módulos de comunicación inalámbrica, reduciendo la cantidad de información que puede ser compartida entre la estación de trabajo remota y el prototipo.
- El consumo de energía debe ser limitado al estar el sistema alimentado por baterías.

Considerando las limitaciones planteadas, se puede reducir las opciones de cámaras a utilizar en este proyecto a cámaras web y al módulo de cámara de Raspberry Pi (Figura(3.11)). El último ítem cuenta con algunas características muy interesantes que la hacen una candidata idónea para su uso en el presente proyecto que se pueden observar en la Tabla(3.8).

Sensor	Sony IMX219 8 megapixeles
Resolución	Fotografía: 3280x2464 Video: 1080p
Framerate	1080p@30Hz, 720p@60Hz, 640p@90Hz
Interfaz	CSI
Dimensiones	25mm x 23mm x 9mm
Peso	3.4g

Tabla 3.8: Características de la Raspberry Pi Camera Module V2. Fuente: [33]

Dentro de las razones por la cual se ha escogido a la Raspberry Pi Camera Module V2 se tiene la capacidad de grabar video a 90 cuadros por segundo, una característica importante para poder incrementar el tiempo de muestreo del bucle de control. En segundo lugar, la cámara utiliza una interfaz CSI que puede conectarse directamente a la placa Raspberry Pi tal como se puede apreciar en la Figura(3.12)

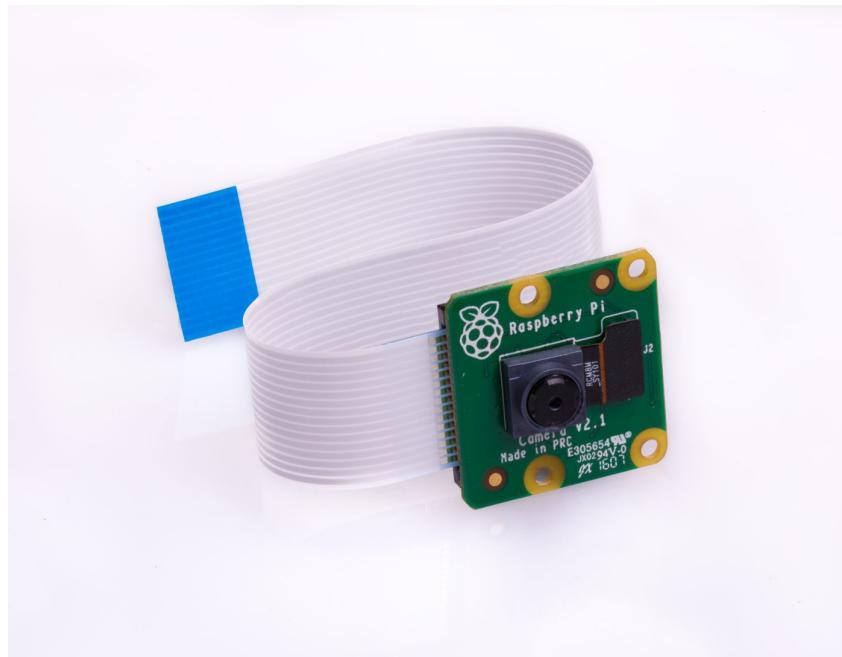


Figura 3.11: Raspberry Pi Camera Module V2. Fuente: [33]



Figura 3.12: Conexión de la Camera Module V2 con una placa Raspberry Pi. Fuente: [33]

La ventaja de poderse conectar con la placa mediante la interfaz CSI es que los *frames* provenientes del sensor se procesan directamente en la GPU de la Raspberry Pi y no así en la GPU, esto gracias a la disponibilidad de los controladores correspondientes para la Camera

Module V2. Esto no es posible cuando se trata de cámaras USB o webcam debido a que los controladores de las mismas procesan los *frames* en la CPU haciendo su procesamiento más lento y ocupando mayor cantidad de ciclos de CPU.

3.3.4.2. Sensor de proximidad

El segundo sensor necesario para la implementación del sistema de control autónomo es un sensor de proximidad que sea capaz de detectar la presencia y distancia de obstáculos que se presenten en frente del vehículo. Este sensor también debe ser compatible con los protocolos de comunicación o periféricos disponibles en la OBC y en el sistema de control de tiempo real.



Figura 3.13: Conexión de la Camera Module V2 . Fuente: [34]

El sensor seleccionado para la tarea es el sensor VL53L0X (Figura(3.13)) de ST Microelectronics. El cual es un sensor óptico de proximidad con láser que usa el principio ToF *Time of Flight* o Tiempo de Vuelo, se basa en el tiempo en el que la luz tarda en reflejarse de la superficie para calcular la distancia dada una velocidad constante de la luz mediante la siguiente fórmula:

$$d = \frac{c}{t_{vuelo}/2} \quad (3.1)$$

donde d es la distancia recorrida y c es la velocidad de la luz. Se puede apreciar una ilustración del principio de funcionamiento del sensor en la Figura(3.14).

Los sensores láser que usan el principio ToF tienen varias ventajas en relación a otro tipo de sensores como los sensores ultrasónicos, o los sensores infrarrojos refractarios. En primer lugar, son inmunes a interferencia sonora o ultrasónica y son más robustos en cuanto a la variedad de superficies sobre las cuales se puede reflejar el haz de luz. El sensor VL53L0X se constituye como una opción favorable para el desarrollo del presente proyecto y por eso se ha elegido. Sus características se detallan en la Tabla(3.9).

El sensor se ha utilizado en una placa de desarrollo o *Breakout Board* que incluye un regulador de tensión y los pines de comunicación correspondientes para su conexión a la



Figura 3.14: Ilustración del principio ToF. Fuente: [34]

Láser	940 nm Clase 1 (seguro a los ojos)
Rango efectivo	30 mm - 1200 mm
Interfaz de comunicación	I2C @ 400 KHz
Tiempo de muestreo mínimo	20 ms
Voltaje de operación	2.6 V - 3.5 V (nominal: 3.3 V)

Tabla 3.9: Características del sensor VL53L0X. Fuente: [35]

SBC. La información proveniente del sensor de proximidad será recuperada y procesada mediante un nodo de ROS de manera que se pueda integrar al sistema en su conjunto.

3.4. Subsistema de Control y actuación

El subsistema de control y actuación representa la base fundamental sobre la cual se desarrolla el resto del proyecto. De acuerdo con la naturaleza del mismo, se pretende implementar una plataforma flexible y modular sobre la cual se pueda controlar el prototipo de múltiples maneras, partiendo desde un modo teleoperado básico donde un operador humano tiene el control de todos los grados de libertad del vehículo, pasando por un modo híbrido hasta un modo autónomo donde los comandos de control son generados por un programa que procesa los datos provenientes de los sensores y otras fuentes externas.

La finalidad del subsistema es brindar una interfaz amigable y flexible para el control de los actuadores del vehículo. Esta interfaz se logra gracias a la infraestructura de ROS mediante el envío de mensajes. Los mensajes utilizados en este subsistema son los mensajes Twist, que se utilizan comúnmente para expresar la velocidad del movimiento de un robot o vehículo en ROS. Se ha escogido el mensaje Twist porque presenta de manera muy conveniente la separación entre la velocidad lineal o la tracción del vehículo y la desviación o la dirección.

La interfaz con el *middleware* de ROS se logra a través de un paquete llamado `rosserial`, que crea un nodo especial que es capaz de enviar mensajes mediante el puerto serial a

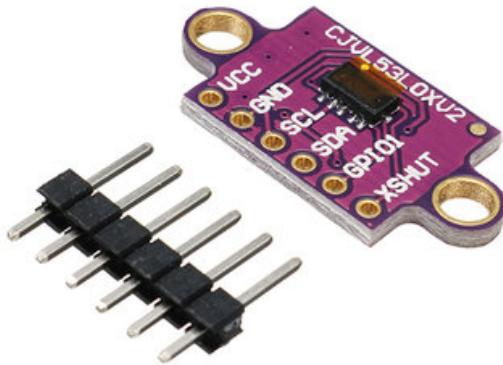


Figura 3.15: Breakout Board para el sensor VL53L0X. Fuente: [34]

sistemas embebidos que no cuentan con una conexión ethernet. De esta manera, el control de los actuadores es transparente a cualquier otro nodo en la red haciendo posible que se pueda controlar el vehículo de distintas maneras, incluso desde nodos ejecutados de manera remota en estaciones de trabajo en la red local.

El código fuente de los programas y nodos concernientes a este subsistema se pueden consultar en el Apéndice(B).

3.4.1. Módulo de potencia en tiempo real

El prototipo cuenta con dos actuadores para la tracción y la dirección respectivamente. En el caso de la tracción, se utiliza un motor de corriente continua de imán permanente en conjunción con un controlador de potencia capaz de manipular al mismo. El control del motor se basa en la combinación de dos pines de dirección y un pin correspondiente con la velocidad que es controlada mediante una señal modulada en ancho de pulso o PWM.

Por su parte, el control de la dirección se realiza mediante el uso de un servomotor dimensionado adecuadamente para la aplicación que controla el ángulo de las ruedas delanteras. El ángulo de las ruedas define el ICC (Sección(2.4)), que a su vez define el radio de giro mínimo en base al máximo rango en el que puede variar el ángulo de las ruedas delanteras. En la Figura(3.16) se puede apreciar el prototipo y la disposición de los actuadores del mismo.

El servomotor que se encarga de la dirección del vehículo se puede apreciar en la Figura(3.17) y el motor DC de aceleración en la Figura(3.18).

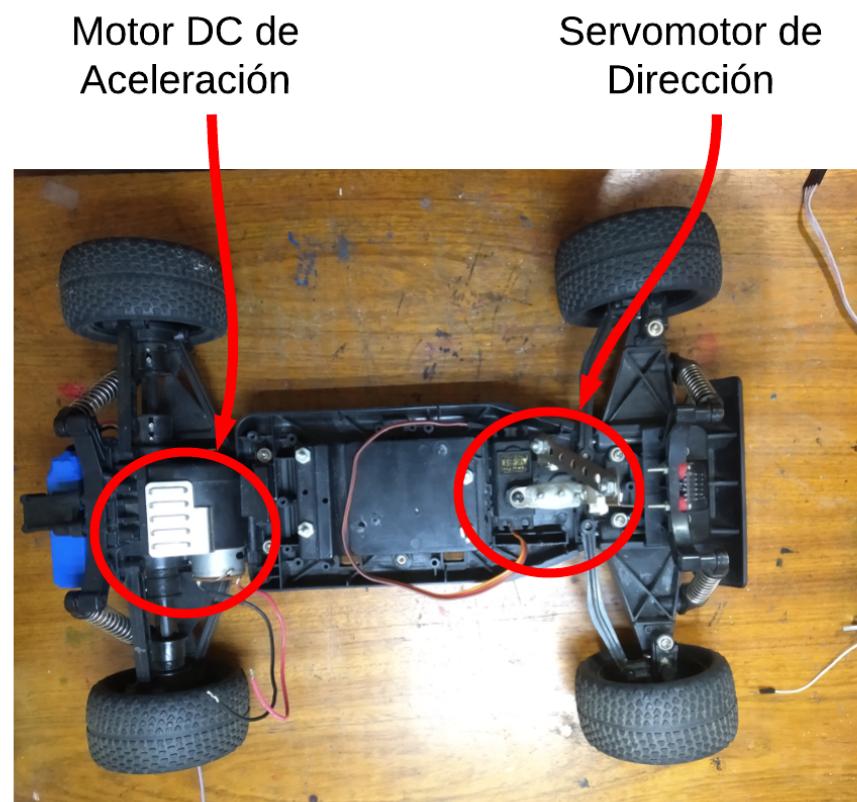


Figura 3.16: Prototipo del vehículo. Fuente: Elaboración propia.

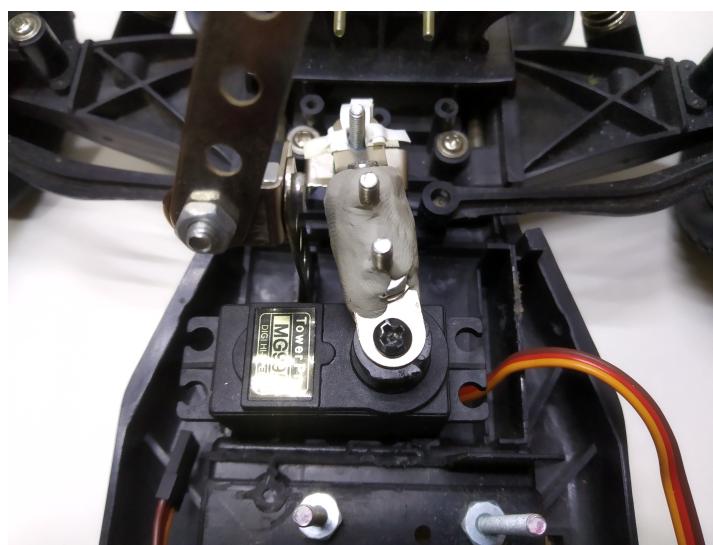


Figura 3.17: Fotografía del servomotor de dirección. Fuente: Elaboración propia.

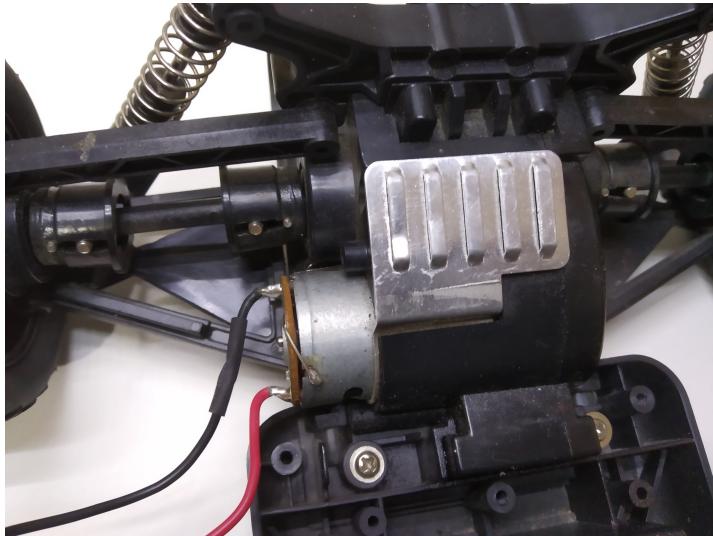


Figura 3.18: Fotografía del motor DC de aceleración. Fuente: Elaboración propia.

3.4.1.1. Control de actuadores mediante ROS

Tal como se pudo especificar en la Sección(3.3.1) el microcontrolador elegido para la tarea tiene la capacidad de usar la librería Rosserial que hace posible crear un nodo de ROS en el microcontrolador usando el puerto serial. En este sentido, el control de motores se realiza de manera asíncrona con un tiempo de muestreo fijo. Se puede apreciar el algoritmo implementado en el nodo de control en el Algoritmo(1).

Algoritmo 1 Algoritmo de control de actuadores

- 1: Conexión con el ros master
 - 2: **Mientras** No hay conexión **hacer:**
 - 3: Intentar conexión
 - 4: **fin Mientras**
 - 5: **Mientras** Conectado con el ros master **hacer:**
 - 6: **Si** Hay mensaje de control pendiente **entonces:**
 - 7: Calcular señal de control
 - 8: Enviar comando a los actuadores
 - 9: **fin Si**
 - 10: **fin Mientras**
-

En este caso, el microcontrolador entra en un bucle que se ejecuta mientras exista conexión con el maestro. Dentro del bucle, espera la llegada de algún mensaje de control para realizar un pequeño cálculo dependiendo a la calibración de los mismos con el fin de enviar una señal de control adecuada para la actuación del vehículo.

En la Figura(3.19) se puede apreciar el diagrama de comunicación entre los nodos de ros correspondientes con el nodo serial y un control manual con un joystick.

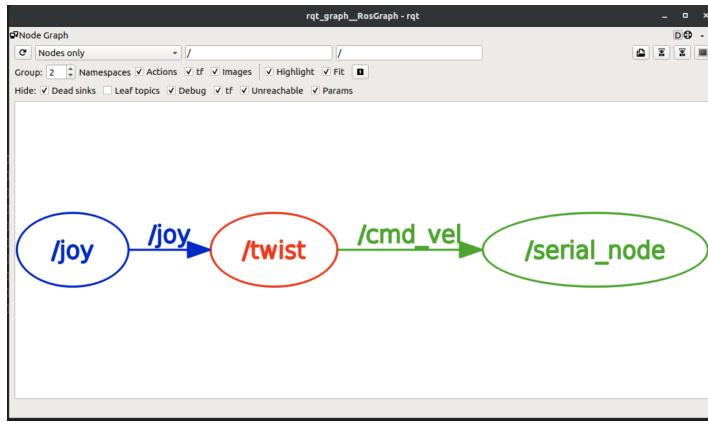


Figura 3.19: Visualización del nodo serial. Fuente: Elaboración propia.

En el caso del control manual, existen tres nodos ejecutándose:

- **/joy** es el nodo que recupera los datos provenientes del joystick, botones presionados y valores de los potenciómetros provenientes de los joysticks analógicos y los triggers. Este nodo publica mensajes de tipo `joy` en el tópico `/joy`.
- **/twist** es el nodo que convierte los datos de entrada del joystick en comandos de control de aceleración y dirección. Se suscribe al tópico `/joy` y publica mensajes de tipo `Twist` en el tópico `/cmd_vel`.
- **/serial_node** es el nodo que se comunica con el microcontrolador mediante el puerto serial. El nodo se suscribe a los mensajes publicados en el tópico `/cmd_vel`.

3.4.2. Módulo de la computadora de a bordo

Mientras que el módulo de control de tiempo real se encarga de traducir los mensajes de control de ROS en señales de control para aplicarse directamente a los actuadores por el microcontrolador a través de un controlador de potencia, la computadora de a bordo u OBC, se encarga de controlar todos los aspectos de *alto nivel* del vehículo. Este módulo se compone de una SBC corriendo un sistema operativo Linux con una distribución de ROS instalada en ella. La OBC será la responsable de la comunicación con todos los nodos que interactúan con el vehículo, tanto de los nodos que generan datos del entorno como las imágenes provenientes de la cámara o sensores; como con nodos que consumen estos datos y los procesan de alguna manera, como el nodo del piloto automático.

En la Figura(3.20) se puede apreciar la OBC y en la Figura(3.21) el prototipo con la OBC incorporada al mismo. Cabe resaltar que por sus características, tanto la OBC, como el módulo de control de tiempo real pueden alimentarse por una batería a bordo del vehículo. A su vez, la comunicación inalámbrica de la OBC permite que el vehículo pueda funcionar sin la necesidad de un solo cable ya sea de alimentación de energía o comunicación.

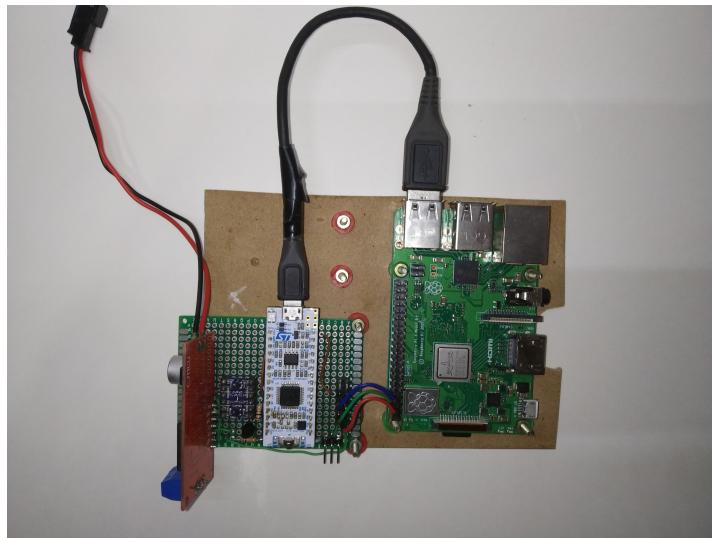


Figura 3.20: Fotografía de la OBC. Fuente: Elaboración propia.

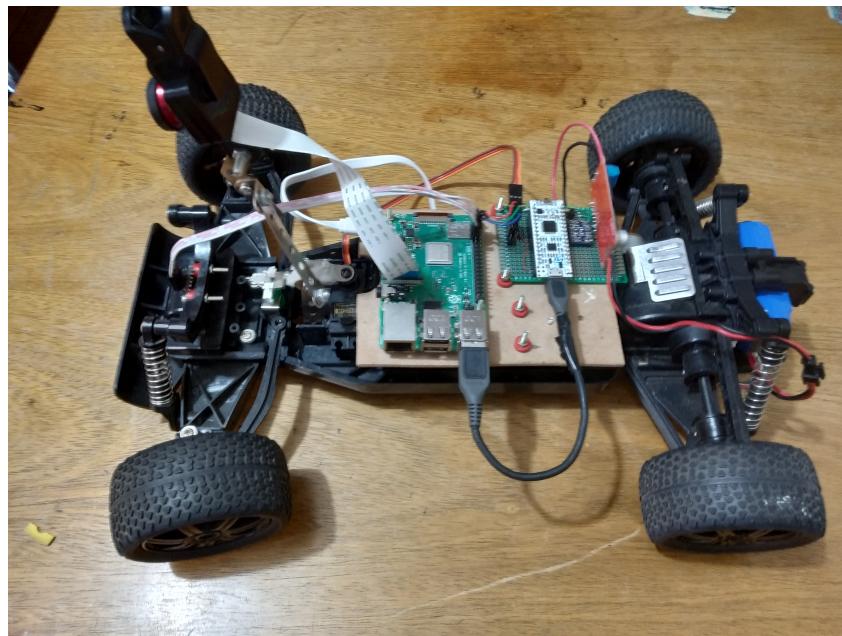


Figura 3.21: Fotografía del prototipo. Fuente: Elaboración propia.

3.4.2.1. Esquema de comunicación en la OBC

La OBC corre el maestro de ROS, que es el que arbitra toda la comunicación en el sistema y, junto con eso, también se encarga de ejecutar varios nodos correspondientes con los sensores y el control. Por su parte, el esquema de comunicación entre dichos nodos, se puede apreciar en la Figura(3.22).

La lista detallada de los nodos que corren en la OBC se puede observar en la Tabla(3.10).

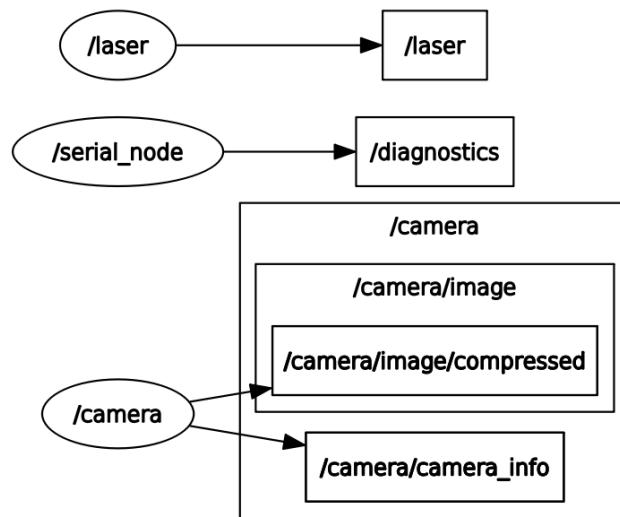


Figura 3.22: Interacción de los nodos en la OBC. Fuente: Elaboración propia.

Nodo	Subscribe		Publica	
	Tópicos	Mensajes	Tópicos	Mensajes
/serial_node	/cmd_vel	geometry_msgs/Twist	/diagnostics	diagnostics_msgs/DiagnosticArray
/laser	-	-	/laser	sensor_msgs/Range
/camera	-	-	/camera/image/compressed /camera/camera_info	sensor_msgs/CompressedImage sensor_msgs/CameraInfo

Tabla 3.10: Nodos de interfaz con los actuadores y sensores. Fuente: Elaboración propia

Se puede rescatar la naturaleza modular del sistema usando el *middleware* de ROS dado que cada nodo se suscribe (si corresponde) a un tópico del cual obtendrá mensajes con la información necesaria, o publica mensajes producto de algún cálculo o proceso de adquisición.

Dado que ROS permite el desarrollo de sistemas distribuidos, existen otros nodos que se ejecutan en la estación de trabajo y se comunican con el maestro, corriendo en la OBC, a través de Wifi mediante una conexión TCP. La implementación de la comunicación en la red está implementada en el *middleware* de ROS y este proyecto no se concentra en los detalles de la misma.

3.4.2.2. Nodo /serial_node

Este nodo se encarga de establecer la conexión con el módulo de potencia en tiempo real a través de un puerto serial. El microcontrolador es capaz de enviar y recibir mensajes como si se tratara de un nodo común de ROS. El nodo serial es parte del paquete `rosserial`[36] y publica información de conexión en un tópico de diagnóstico, este tópico es útil en la depuración y cuando existen problemas en la comunicación serial con el microcontrolador.

3.4.2.3. Nodo /laser

Este nodo se encarga de publicar mensajes del tipo `sensor_msgs/Range` con información sobre el sensor de proximidad que se encuentra en frente del vehículo. Se establece la comunicación mediante el protocolo I2C ³ con el sensor VL53L0X (Sección(3.3.4.2)) y se adecúan los datos para ser publicados en el tópico `/laser`.

3.4.2.4. Nodo /camera

Este nodo realiza la conexión con la cámara conectada a la OBC, la Raspberry Pi Camera Module V2 (Sección(3.3.4.1)) y publica las imágenes en formato compresado JPEG ⁴ para utilizar menos ancho de banda. Las imágenes se publican en el tópico `/camera/image/compressed`. El nodo es parte del paquete `raspicam_node` [37].

3.5. Subsistema de Adquisición de Datos y Entrenamiento

Una vez establecida la plataforma de trabajo en el Subsistema de Control y Actuación se procede a detallar el diseño del Subsistema de Adquisición de Datos y Entrenamiento que se encarga principalmente de brindar una forma adecuada de sincronizar y almacenar los datos necesarios para el entrenamiento de la red neuronal así como también de brindar las herramientas necesarias para el entrenamiento en sí. Este subsistema cuenta con varios módulos funcionales que interactúan entre sí como se pudo apreciar en la Sección(3.1.4).

El código fuente de los programas y nodos concernientes a este subsistema se pueden consultar en el Apéndice(B).

3.5.1. Módulo de adquisición de datos y operación manual

Como se ha podido establecer en la Sección(2.3) referida al aprendizaje automático. Para que un algoritmo de aprendizaje pueda entrenarse de manera efectiva, es necesario contar con un conjunto de datos sobre el cual se realizará el mismo. Este conjunto de datos o *dataset* necesita estar almacenado de manera adecuada con el formato y las características necesarias para un entrenamiento efectivo.

El módulo cuenta con un nodo que se suscribe a los mensajes provenientes tanto de la cámara como del control manual con el joystick (Figura(3.23)). Luego, se realiza una sincronización para que se pueda generar un par entrada - salida adecuado y luego se procede a almacenar el mismo. En la Figura(3.24) se puede apreciar el grafo de comunicación de los nodos concernientes a este módulo.

³I2C: Circuito inter-integrado (I2C, del inglés Inter-Integrated Circuit) es un bus serie de datos para la comunicación entre diferentes partes de un circuito, por ejemplo, entre un controlador y circuitos periféricos integrados.

⁴JPEG: Estándar de compresión y codificación de archivos e imágenes.



Figura 3.23: Fotografía del mando inalámbrico de Xbox 360. Fuente: Elaboración propia.

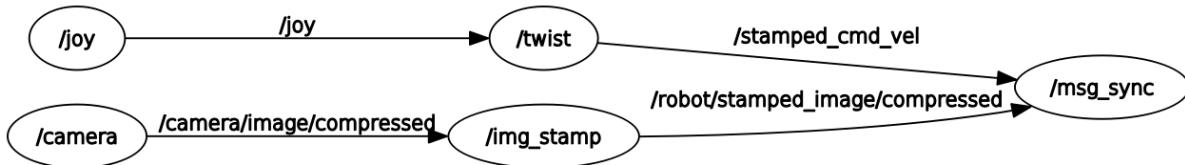


Figura 3.24: Esquema de nodos para la adquisición de datos. Fuente: Elaboración propia.

El nodo `/msg_sync` se encarga de recuperar los mensajes de la cámara y del joystick y sincronizarlos para luego guardar la imagen en un archivo .bmp y su correspondiente información en el archivo `target.csv`. Se puede observar el procedimiento en el Algoritmo(3).

El *dataset* se compone de un conjunto de imágenes en formato PNG junto con un archivo csv que contiene las columnas detalladas en la Tabla(3.11). En la tabla se almacenan tanto los datos de aceleración y dirección como el nombre del archivo de la imagen correspondiente con la misma. Los registros se almacenan en el mismo orden en el que fueron capturados, no obstante, para la tarea de regresión definida en este proyecto, la dependencia temporal entre las muestras no es importante.

3.5.1.1. Sesiones de entrenamiento

La manera de adquirir el conjunto de datos es mediante las denominadas *Sesiones de Entrenamiento o Grabación de Comportamiento*. Estas sesiones corresponden con períodos de tiempo en el cual se recuperan datos de manera constante. La cantidad de registros o muestras recuperadas depende tanto de la resolución temporal de la generación de imágenes

Algoritmo 2 Algoritmo de sincronización de mensajes y almacenamiento de datos.

```

1: Subscripción a /stamped_cmd_vel
2: Subscripción a /robot/stamped_image/compressed
3: Registrar callback
4: Mientras Conectado con el ros master hacer:
5:   Repetir:
6:     Si Llegan mensajes sincronizados entonces:
7:       Decodificar imagen
8:       Redimensionar imagen
9:       Guardar archivo
10:      Generar registro con datos de entrada/salida
11:      Actualizar registro en el archivo target.csv
12:    fin Si
13:  fin Repetir:
14: fin Mientras

```

Columna	Descripción	Tipo de dato
id	Número de identificación único para cada muestra del dataset.	Int32
angular	Valor del comando de dirección proveniente del Joystick 1: completamente a la izquierda -1: completamente a la derecha 0: adelante	Float32
imgpath	Nombre del archivo de la imagen correspondiente con el id actual	String
linear	Valor del comando de aceleración proveniente del Joystick 1: aceleración máxima adelante -1: aceleración máxima atrás 0: detenido	Float32
target	Par de valores correspondientes con los valores angular y linear.	Array

Tabla 3.11: Columnas de la tabla del conjunto de datos generado. Fuente: Elaboración propia.

y comandos de control sincronizados así como también de la duración de la sesión en sí. Cada sesión de entrenamiento puede generar un *dataset* distinto que se puede combinar con otros *datasets* o tomarse de manera independiente.

Es importante tomar en cuenta que mientras más sesiones se realicen y más variadas en

condiciones ambientales sean las mismas, más robusto será el algoritmo entrenado y mejor será su generalización. Sin embargo, esta capacidad de generalización estará limitada por la arquitectura y la cantidad de parámetros disponibles para generar representaciones internas del proceso.

En el presente proyecto, se han realizado sesiones de entrenamiento con un clima parcialmente nublado con buena iluminación ambiental, de manera que se pueda distinguir la pista del piso (Figura(3.25)).



Figura 3.25: Imágenes capturadas por la cámara en una sesión de entrenamiento. Fuente: Elaboración propia.

3.5.2. Módulo de aumentación de datos

Las redes neuronales profundas tienen la impresionante capacidad de aprender representaciones internas útiles de manera automática usando el algoritmo de la retropropagación, pero esta capacidad solamente resalta cuando se cuenta con un conjunto de datos de un tamaño considerable, es decir, que se necesitan muchas muestras para que el entrenamiento de una red neuronal profunda sea exitosa y pueda mostrar resultados favorables.

Por su parte, la recuperación de datos es un proceso costoso en tiempo y recursos, por lo que no se puede invertir demasiado tiempo solamente recuperando muestras. De hecho, se ha establecido que el proceso de la generación de un *dataset* efectivo es el más complicado en un sistema de aprendizaje automático. Por tanto, es importante poder aprovechar al máximo los datos obtenidos en una sesión de entrenamiento.

Es por este motivo que se ha decidido incluir una etapa de aumentación de datos que se compone de una serie de transformaciones de las imágenes originales del *dataset* para poder incrementar la cantidad de muestras sobre la cual se realice el entrenamiento de la red de manera sintética. En la Tabla(3.12) se puede apreciar las transformaciones usadas.

Transformación	Descripción
Rotación	Se rota la imagen un ángulo aleatorio entre -20 y 20 grados.
Desplazamiento Vertical	Se desplaza la imagen de forma vertical un valor aleatorio entre 0 y 20%.
Espejo Horizontal	Se invierte la imagen horizontalmente de forma aleatoria.

Tabla 3.12: Transformaciones realizadas en la aumentación de datos. Fuente: Elaboración propia.

Este módulo no solamente cuenta con ciertas transformaciones disponibles para aumentar los datos existentes en el *dataset* si no también cuenta con la función de poder eliminar registros con un valor de dirección nulo si es que fuera necesario. La utilidad de esta funcionalidad se puede apreciar en la etapa de entrenamiento en la Sección(3.8). El algoritmo para la aumentación de datos se puede apreciar en el Algoritmo(3).

Algoritmo 3 Algoritmo del módulo de aumentación de datos.

- 1: Carga del dataset
 - 2: Configuración de los parámetros de transformación
 - 3: Iniciar generador de pares (*imagen*, *target*)
 - 4: **Para cada** imagen **hacer:**
 - 5: *img_{aux}* = *imagen*
 - 6: **Si** *abs(target)* > 0.09 **entonces:**
 - 7: *hTarget* = *-target*
 - 8: *hImg* = *flipHorizontal(img_{aux})*
 - 9: Guardar (*hImg*, *hTarget*)
 - 10: *nImg* = *shiftVertical(img_{aux})*
 - 11: *nImg* = *rotacion(nImg)*
 - 12: Guardar (*hImg*, *target*)
 - 13: **fin Si**
 - 14: **fin Para**
-

En el caso de la transformación de espejo horizontal, se debe invertir también la salida o *target* de la imagen para que corresponda con el espejo, es decir, que si para una imagen *img* el *target* = 0.67 para su transformación *hImg* = *flipHorizontal(img)* se debe crear un nuevo *hTarget* = *-target* = -0.67.

Para las transformaciones de desplazamiento vertical y rotación, se mantiene el *target* original de la imagen pues no se afecta la naturaleza de la imagen de manera sustancial.

3.5.3. Módulo de generación de datos de entrenamiento

Una de las desventajas de las redes neuronales convolucionales para el procesamiento de imágenes es que las imágenes ocupan bastante espacio en memoria y cargar todo el conjunto de datos de entrenamiento en la memoria RAM es usualmente imposible o demasiado costoso. Es por eso que se necesita una forma de cargar las imágenes en la memoria de forma fraccionada, de manera que los requisitos de memoria sean manejables por la estación de trabajo.

El módulo de generación de datos de entrenamiento se ha desarrollado con la finalidad de poder generar *mini batches* o conjuntos pequeños de datos para el entrenamiento de la red neuronal convolucional. El algoritmo de la generación se detalla en el Algoritmo(4).

Algoritmo 4 Algoritmo del módulo de generación de datos.

Entrada: Tabla de datos, tamaño del mini-batch

- 1: Reordenar el dataset aleatoriamente
 - 2: **Para cada** mini-batch **hacer:**
 - 3: $X = [\dots]$
 - 4: $Y = [\dots]$
 - 5: **Para cada** par(*imagen*, *target*) **hacer:**
 - 6: Cargar *imagen* desde el disco
 - 7: Agregar *imagen* a *X*
 - 8: Agregar *target* a *Y*
 - 9: **fin Para**
 - 10: Enviar (*X*, *Y*)
 - 11: **fin Para**
-

Como se puede observar, para cada mini-batch generado se crean dos listas *X* y *Y* que contendrán los datos de las imágenes y los targets. De esta manera ya no se necesita cargar todo el *dataset* en la memoria ram pues se cargarán solamente las imágenes del disco correspondiente a cada mini-batch.

Este módulo es capaz de generar *mini batches* de manera parametrizada de acuerdo a la necesidad de cada sesión de entrenamiento.

3.5.4. Módulo de Entrenamiento

Una vez se tienen las herramientas para la correcta generación y aumentación de datos del conjunto de datos de entrenamiento se necesita una herramienta para ejecutar el entrenamiento de la red neuronal convolucional en sí. El módulo de entrenamiento se compone de un programa que ejecuta las siguientes tareas:

- Carga del dataset y el modelo de red neuronal.
- Definición de hiperparámetros.
- Ejecución, monitoreo y control del proceso de entrenamiento en línea.
- Salvaguarda de los parámetros del modelo entrenado.
- Generación de reportes del proceso de entrenamiento.

En el Algoritmo(5) se puede apreciar el proceso detallado de entrenamiento de la red de acuerdo a parametrización definida por el usuario.

Algoritmo 5 Algoritmo del módulo de entrenamiento.

Entrada: Conjunto de datos, modelo de la red neuronal

- 1: Generar conjunto de **entrenamiento**
- 2: Generar conjunto de **prueba**
- 3: Generar conjunto de **validación**
- 4: Definir **hiperparámetros**
- 5: Cargar el **modelo** de la red neuronal
- 6: Crear un generador de datos
- 7: Definir $n_{epochas}, n_{minibatches}$
- 8: **Para cada epocha hacer:**
 - 9: Generar minibatches
 - 10: **Para cada** (X, Y) **hacer:**
 - 11: Propagación hacia adelante de $X \rightarrow \dots \rightarrow \hat{Y}$
 - 12: Cálculo del error $E = Y - \hat{Y}$
 - 13: Cálculo de gradientes ∇E (retropropagación)
 - 14: Ajuste de parámetros con ADAM $w \leftarrow w - \alpha \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$
 - 15: Reporte de datos de entrenamiento de minibatch
 - 16: **fin Para**
 - 17: Reporte de datos de entrenamiento de época
- 18: **fin Para**
- 19: Guardar modelo en archivos .json y .h5

Este módulo se ejecuta en la estación de trabajo y hace uso de la GPU disponible para la paralelización de los algoritmos de cálculo de gradientes y optimización con el fin de acelerar el tiempo de entrenamiento. El módulo es capaz de realizar el entrenamiento de distintos modelos de arquitectura de redes neuronales definidas por el usuario en un archivo de código fuente, en otras palabras, se puede utilizar el mismo programa para entrenar múltiples redes neuronales sin realizar grandes cambios en el código fuente. También se puede definir los directorios donde están almacenados los datos de entrenamiento y el destino de los reportes del entrenamiento, esto es útil para cuando se necesita validar el entrenamiento de múltiples

modelos con múltiples conjuntos de datos o para separar las sesiones de entrenamiento de distintos sistemas o prototipos.

Los productos obtenidos por este módulo son dos:

- **Arquitectura de la red neuronal:** Se almacena la información acerca de la arquitectura de la red en un archivo con formato JSON donde se puede encontrar la información acerca de las dimensiones de las capas ocultas, cantidad de unidades por capa y dimensiones de cada unidad en la red.
- **Pesos entrenados de la red neuronal:** Se almacena también los valores de todos los parámetros o pesos de la red neuronal, resultado del entrenamiento. Estos valores están almacenados en un archivo con extensión h5.

Con estos dos archivos será posible ejecutar la tarea de inferencia en etapas posteriores. Además, es importante mencionar que los mismos archivos pueden usarse para re-entrenar la red con un nuevo conjunto de datos para mejorar su rendimiento.

Como se ha podido observar, los módulos que componen el Subsistema de Adquisición de Datos y Entrenamiento se han diseñado con el fin de poderse utilizar de manera individual con otros sistemas o de manera conjunta, enfocando las funcionalidades en el entrenamiento de redes neuronales convolucionales para tareas de procesamiento de imágenes y visión artificial.

3.6. Subsistema de Inferencia y control autónomo

El subsistema de inferencia y control autónomo se ha diseñado tomando en cuenta la modularidad necesaria para poder ser extendible en sus funcionalidades. En este sentido, se aprovecha al máximo las capacidades de comunicación y ejecución distribuida de ROS con el objetivo de que el desarrollo de este subsistema pueda extenderse o reemplazarse de una manera trivial.

El código fuente de los programas y nodos concernientes a este subsistema se pueden consultar en el Apéndice(B).

3.6.1. Módulo de inferencia con una red neuronal convolucional

Este módulo tiene la tarea de ejecutar la tarea de predicción del comando de control de dirección con la red neuronal entrenada por el módulo de entrenamiento del subsistema de adquisición de datos y entrenamiento. Para que la red pueda utilizarse para realizar predicciones es necesario cargar los parámetros de arquitectura y pesos de la red previamente generados en el Subsistema de Adquisición de datos y Entrenamiento. Una vez cargados los datos, la red puede usarse para predecir comandos de control de dirección en base a nuevas imágenes provenientes de la cámara.

El proceso de predicción se detalla en el Algoritmo(6). Es importante destacar que este módulo se ha implementado como un nodo de ROS, de manera tal que puede integrarse al sistema usando tópicos y mensajes. En este caso, el nodo se suscribe al tópico de la cámara

frontal y publica comandos de control en un tópico especial que puede ser consumido por cualquier otro nodo, en este proyecto, los mensajes provenientes del módulo de inferencia son consumidos por el módulo del piloto automático.

Algoritmo 6 Algoritmo del módulo de inferencia.

- 1: Cargar parámetros
 - 2: Cargar la arquitectura de la red neuronal (archivo .json)
 - 3: Cargar los pesos de la red neuronal (archivo .h5)
 - 4: Suscribirse al tópico de las imágenes de la cámara /camera/image/compressed
 - 5: Crear tópico para publicar la salida /neural_ouput
 - 6: Conectar con el ROS master
 - 7: **Mientras** hay conexión **hacer**:
 - 8: **Si** ha llegado un mensaje de la cámara **entonces**:
 - 9: Decodificar imagen
 - 10: Redimensionar imagen
 - 11: Normalizar imagen
 - 12: Obtener predicción de la red neuronal en *angular*
 - 13: Crear mensaje de tipo `Float32()`
 - 14: Publicar *angular* en el tópico /neural_ouput
 - 15: **fin Si**
 - 16: **fin Mientras**
-

El nodo de predicción se suscribe al tópico /camera/image/compressed para obtener mensajes provenientes de la cámara y publica mensajes de tipo std_msgs/Float32 en el tópico /neural_ouput que representa la salida de la red neuronal. Se puede apreciar el diagrama de conexión de los nodos en la Figura(3.26).

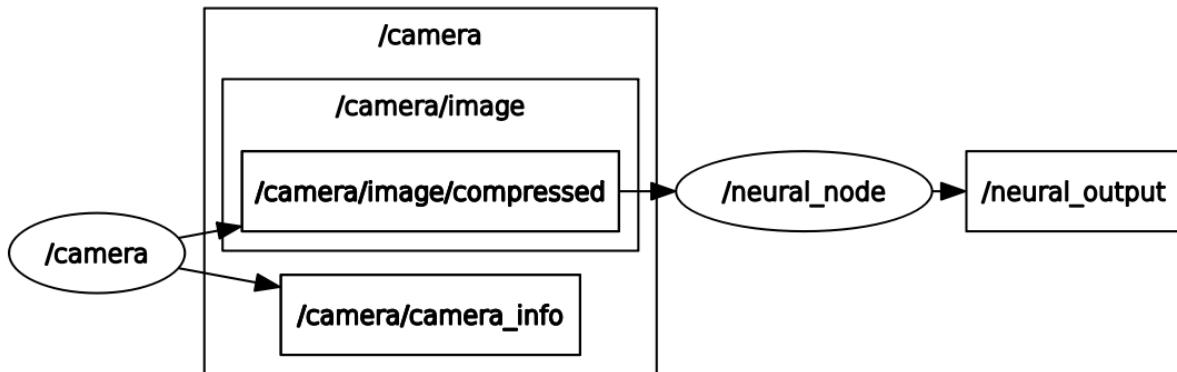


Figura 3.26: Esquema del módulo de inferencia. Fuente: Elaboración propia.

Este módulo tiene especial importancia pues implementa el concepto de aprendizaje **fin a fin** en el sistema de conducción. Como se ha podido observar, la única tarea que realiza

es alimentar la red neuronal con imágenes provenientes de la cámara para obtener comandos de control a la salida, sin realizar ningún procesamiento previo a la imagen, como puede ser la extracción de características o reducción de dimensionalidad.

3.6.2. Módulo de detección de obstáculos

El módulo de detección de obstáculos tiene la finalidad de controlar la aceleración del vehículo basado en la presencia de obstáculos físicos frente al mismo. Se basa en las mediciones de proximidad realizadas por el sensor de proximidad montado en frente del vehículo.

Este módulo se suscribe al nodo que publica los mensajes de distancia del sensor, y publica el valor de la aceleración en otro nodo para ser consumido por el nodo de piloto automático. Cuando no se encuentran obstáculos cerca del vehículo, el control imprime una aceleración constante, si se detecta un obstáculo, el algoritmo reduce la aceleración de manera proporcional a la distancia con el fin de obtener una deceleración suave para detenerse a una distancia segura. Si el obstáculo se encuentra a una distancia menor a la segura, el algoritmo hace que el vehículo retroceda para no colisionar. El algoritmo se detalla en el Algoritmo(7).

Algoritmo 7 Algoritmo del módulo de detección de obstáculos.

Entrada: Distancia segura $distancia_{stop}$, Constante proporcional K_p , Aceleración máxima acc_{max}

- 1: Cargar parámetros
- 2: Suscribirse al tópico de las mediciones del sensor /laser
- 3: Crear tópico para publicar la salida /obstacle_ouput
- 4: Conectar con el ROS master
- 5: **Mientras** hay conexión **hacer:**
- 6: **Si** ha llegado un mensaje del sensor *msg* **entonces:**
- 7: $error = msg.range - distancia_{stop}$
- 8: $aceleracion = K_p \cdot error$
- 9: **Si** $aceleracion > acc_{max}$ **entonces:**
- 10: $aceleracion \Leftarrow acc_{max}$
- 11: **fin Si**
- 12: **Si** $(distancia_{stop} - 0.03) < aceleracion < (distancia_{stop} + 0.03)$ **entonces:**
- 13: $aceleracion \Leftarrow 0$
- 14: **fin Si**
- 15: Crear mensaje de tipo `Float32()`
- 16: Publicar *aceleracion* en el tópico /obstacle_ouput
- 17: **fin Si**
- 18: **fin Mientras**

El módulo se ha implementado como un nodo de ROS, de la misma manera que el nodo de inferencia con la red neuronal, para que sea compatible con toda la infraestructura de comunicación que brinda ROS. El esquema de la interacción del nodo con otros nodos se

puede apreciar en la Figura(3.27). Como se puede observar, el nodo es una unidad de procesamiento independiente y puede ser implementada de distintas maneras. Este módulo puede ser reemplazado por uno más sofisticado o tomando en cuenta más sensores y variables.



Figura 3.27: Esquema del módulo de detección de obstáculos. Fuente: Elaboración propia.

3.6.3. Módulo del piloto automático

Este módulo se encarga de arbitrar los comandos de control provenientes de las distintas fuentes utilizadas en el sistema. Se suscribe a cada tópico correspondiente y redirecciona los mensajes para el control del vehículo mediante mensajes de tipo `geometry_msgs/Twist`.

De la misma manera que los anteriores módulos del subsistema, el módulo del piloto automático está implementado como un nodo de ROS que se suscribe a ciertos tópicos y publica mensajes en el tópico correspondiente con el control del vehículo cerrando el lazo de control. El funcionamiento del nodo se puede analizar en el Algoritmo(8).

Es importante resaltar que, por seguridad, todavía se mantiene activo el control manual con el joystick, esto con el objetivo de poder brindar a el operador una forma de evitar algún accidente debido a fallos en el sistema. Con todo, en la Figura(3.28) se puede apreciar el esquema de conexión de nodos del sistema en su conjunto, operando en modo autónomo.

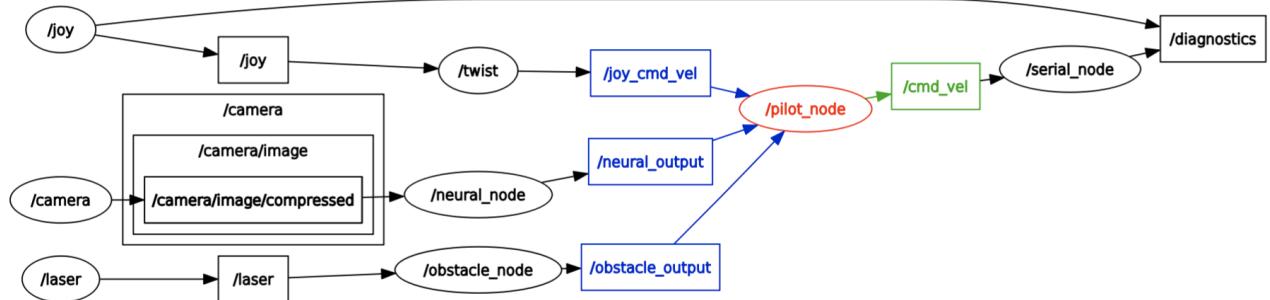


Figura 3.28: Esquema de los nodos del sistema en modo autónomo. Fuente: Elaboración propia.

Una característica importante de este módulo es que la interacción no está limitada a las tres fuentes de datos para el control definidas: la red neuronal, la detección de obstáculos y el joystick, sino que es completamente extensible a otras fuentes de control o decisión. Se pueden agregar múltiples fuentes de información para el control de alto nivel del vehículo tales como

Algoritmo 8 Algoritmo del módulo de detección de obstáculos.

Entrada: Tiempo de muestreo t_s

- 1: Cargar parámetros
- 2: Suscribirse al tópico de la detección de obstáculos /obstacle_ouput
- 3: Suscribirse al tópico de la red neuronal /neural_ouput
- 4: Suscribirse al tópico de los comandos del joystick /joy_cmd_vel
- 5: Crear tópico para publicar la salida /cmd_vel
- 6: Configurar temporizador para el tiempo de muestreo fijo
- 7: Conectar con el ROS master
- 8: **Mientras** hay conexión **hacer:**
 - 9: **Si** ha llegado un mensaje de la detección de obstáculos *aceleracion entonces:*
 - 10: Guardar *aceleracion*
 - 11: **fin Si**
 - 12: **Si** ha llegado un mensaje de la red neuronal *direccion entonces:*
 - 13: Guardar *direccion*
 - 14: **fin Si**
 - 15: **Si** ha llegado un mensaje del joystick *twist_{joy}* **entonces:**
 - 16: Guardar *twist_{joy}*
 - 17: **fin Si**
 - 18: **Si** se ha cumplido el intervalo de tiempo t_s **entonces:**
 - 19: **Si** hay comando de control *twist_{joy}* **entonces:**
 - 20: Publicar *twist_{joy}* en /cmd_vel
 - 21: **Si no**
 - 22: Publicar (*aceleracion, direccion*) en /cmd_vel
 - 23: **fin Si**
 - 24: **fin Si**
 - 25: **fin Mientras**

detección de señales de tránsito, detección de peatones, detección de otros vehículos, sensores de proximidad en todo el vehículo o sensores iniciales. También es importante considerar que el piloto automático es un modo de control de medio nivel pues simplemente implementa el seguimiento de una trayectoria definida por la red neuronal. Se puede incluir y desarrollar sistemas de toma de decisión de alto nivel o módulos de planificación y navegación avanzados. Estas son las ventajas de utilizar un *framework* de trabajo distribuido como ROS.

Otra característica es que el nodo actualiza los comandos de control con un tiempo de muestreo fijo, esto con el fin de garantizar la estabilidad del subsistema de control y actuación.

3.7. Diseño de la arquitectura de la red neuronal

Una vez establecida la plataforma sobre la cual se va a implementar el entrenamiento de la red neuronal así como también la etapa de inferencia para el control autónomo, es necesario definir el procedimiento de diseño de la arquitectura de la red que se encargará de tomar las imágenes provenientes de la cámara para mapearlas en un valor escalar que representa a la desviación de la dirección del vehículo.

Con el propósito de validar la eficacia del uso de una red neuronal convolucional se han generado dos arquitecturas de red neuronal profunda: una tradicional, y una con capas convolucionales.

3.7.1. Requerimientos

Se necesita definir los requerimientos de desempeño para la red neuronal considerando la tarea de generación de comandos de control en base a imágenes de la carretera o entorno.

3.7.2. Unidades y profundidad

Tomando en cuenta el requerimiento de poderse ejecutar de manera fluída en el sistema embebido proporcionado como computadora de abordo (OBC) la red neuronal no debe poseer demasiados parámetros pues la cantidad de memoria es limitada. Es necesario recordar que en el caso de contar con una GPU, los parámetros de la red se cargan en la memoria integrada en la GPU, sin embargo, si el sistema no cuenta con una GPU dedicada, los parámetros se cargan en la memoria RAM del sistema. Para la OBC escogida para el presente proyecto, se tiene disponible una cantidad de memoria RAM de 1 GB, lo cual significa que todo el sistema en su conjunto, incluyendo nodos de sensores, actuación y sistemas de control necesitan compartir la memoria RAM del sistema. Por tanto, en base a estas consideraciones se han generado dos modelos de arquitectura con el objetivo de comparar el rendimiento de ambas.

3.7.2.1. Capas Convolucionales y Pooling

Como se ha visto en la Sección(2.3.3), las redes neuronales convolucionales tienen la capacidad de generar representaciones internas para imágenes a través de filtros o *kernels* de

convolución. Cada capa convolucional tiene como entrada una imagen o mapa de características y como salida se obtiene otro mapa de características que, a medida que se va avanzando en profundidad, constituyen representaciones cada vez más complejas de la imagen de entrada. No se debe olvidar que el propósito de usar una red neuronal convolucional para el procesamiento de imágenes es el de reducir la dimensionalidad de entrada en cantidad de pixeles a otro tipo de representación más abstracta en la salida, en este caso, un comando de control correspondiente con una fotografía de la carretera. En la Figura(3.29) se puede apreciar la naturaleza de una capa convolucional en la red implementada en Tensorflow.

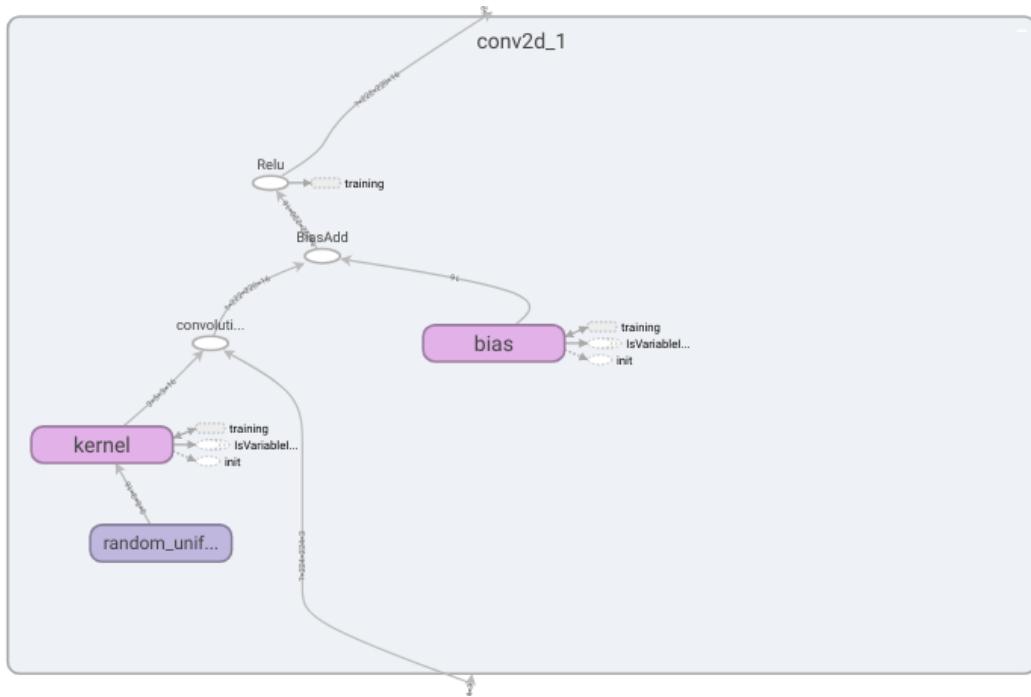


Figura 3.29: Visualización de una capa convolucional. Fuente: Elaboración propia.

Es así que a medida que se avanza en las capas de la red convolucional, la dimensión de los mapas de características van reduciéndose más y más. Un elemento importante para esta reducción se denomina *pooling*, como se ha visto en la Sección(2.3.5), que reduce las dimensiones de la imagen seleccionando los pixeles con valores de activación más altos.

Las capas de *max pooling* no contienen parámetros entrenables y por tanto, no influyen en el proceso de aprendizaje con el algoritmo de la retropropagación. En la Figura(3.30) se puede apreciar una visualización de una capa de max pooling implementada en Tensorflow.

3.7.2.2. Capas densamente conectadas

Las capas convolucionales cumplen un gran trabajo en la generación de representaciones internas abstractas a medida que se incrementa la profundidad de la red, pero pese a eso, siguen siendo representaciones en varias dimensiones. En el caso de la tarea de regresión,

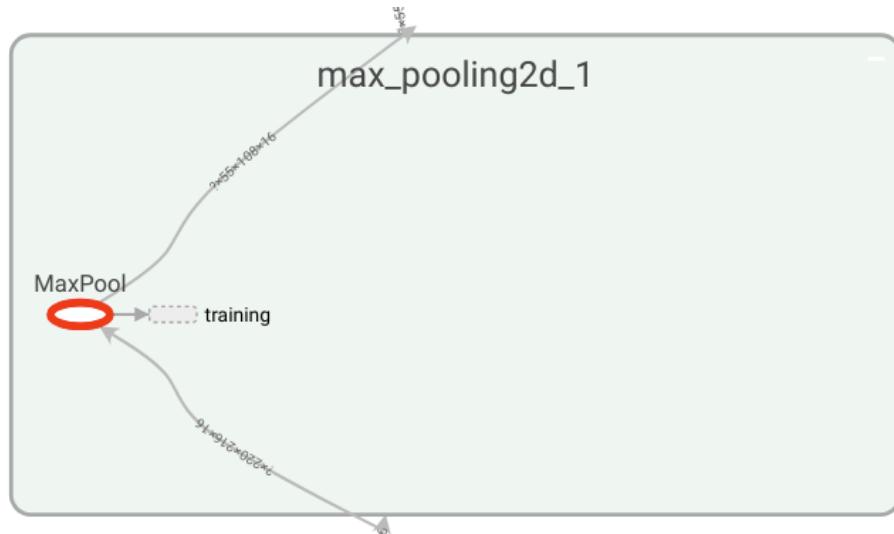


Figura 3.30: Visualización de una capa de max pooling. Fuente: Elaboración propia.

la salida de la red se expresa simplemente como un escalar, es por eso que, en las últimas capas de la red, se suelen agregar una o varias capas densamente conectadas, con unidades tradicionales para generar una representación vectorial de la imagen de la entrada. Esta representación vectorial se suele denominar *embedding* en terminología de redes neuronales.

La salida corresponde simplemente con una combinación lineal del *embedding* seguido de una función de activación no lineal. En la Figura(3.31) se puede apreciar la implementación de una capa densamente conectada en Tensorflow.

3.7.3. Funciones de Activación

Se han escogido distintas funciones de activación para las capas ocultas y la capa de salida en base a los requisitos de rendimiento, naturaleza de las capas de la red e implementaciones previas en la literatura y proyectos relacionados. Las funciones de activación escogidas se listan en la Tabla(3.13).

Función de activación	Objetivo	Criterios de implementación
ReLU(Sección(2.3.2.4))	Rectificar las activaciones de las capas ocultas de la red. Mantener los gradientes.	Se usan para todas las activaciones de las capas ocultas de la red, tanto capas convolucionales como densamente conectadas
Tangente hiperbólica(Sección(2.3.2.4))	Mapear la activación en un rango de -1 a 1.	Se usa para la capa de salida de la red dado que la salida requiere estar en rango -1 a 1

Tabla 3.13: Funciones de activación usadas. Fuente: Elaboración propia.

Las funciones de activación usadas en las capas ocultas corresponden con la función ReLU,

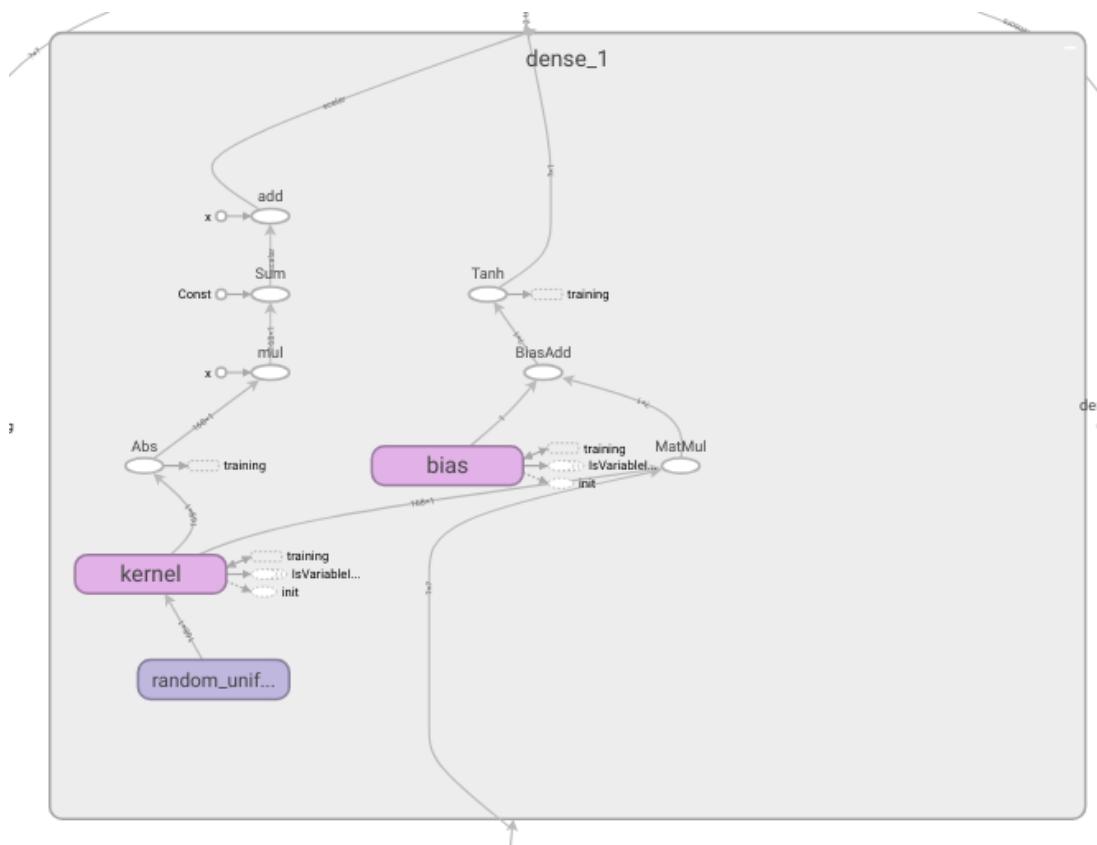


Figura 3.31: Visualización de una capa densamente conectada. Fuente: Elaboración propia.

pues se ha demostrado que son eficaces en la implementación y en la generación de gradientes para el entrenamiento por retropropagación.

En el caso de la capa de salida, se ha utilizado una función de activación tangente hiperbólica, debido a que esta función representa de manera muy fiel la naturaleza del proceso de generación de comandos, pues los comandos de dirección varían entre los valores -1 y 1 .

3.7.4. Función de Costo

La función de costo utilizada es el error cuadrático medio, definido en la Ecuación(2.8). La función de costo escogida permite calcular una medida del error básica y funciona perfectamente en las tareas de regresión.

3.7.5. Optimizador

El algoritmo de optimización es una parte importante en el proceso de entrenamiento de una red neuronal profunda pues en base a este se define el rendimiento del proceso de entrenamiento en si. En otras palabras, se debe escoger un algoritmo de optimización adecuado que garantice la convergencia de la disminución del error así como también que los

tiempos de entrenamiento para cada época sean razonables. En este sentido, se ha escogido el algoritmo de ADAM (Sección(2.3.2.3)) pues garantiza la convergencia de la red usando el concepto de momentos de primer y segundo orden. Por su parte, ADAM es un algoritmo altamente recomendado para tareas de regresión con redes neuronales profundas [12].

3.7.6. Arquitecturas implementadas

Con todo, se ha llegado a la implementación de dos distintas arquitecturas de red neuronal para su entrenamiento en el presente proyecto. Se ha implementado, en primer lugar una red neuronal tradicional, con capas ocultas densamente conectadas y una red neuronal convolucional.

3.7.6.1. Red neuronal tradicional

Con el objetivo de validar la efectividad de una red neuronal convolucional para tareas de visión artificial y procesamiento de imágenes, se ha implementado una red neuronal densamente conectada, o una red tradicional que servirá como punto de referencia para el entrenamiento de la red convolucional posteriormente.

La arquitectura de la red se puede analizar en la Tabla(3.14). Se puede observar que esta no es una red compuesta exclusivamente por capas densamente conectadas, pues se tiene una capa convolucional en la entrada. Esta capa se suele usar para crear un mapa de características inicial sobre el cual se puedan generar representaciones en las neuronas en las capas posteriores. Las demás capas de la red corresponden con capas densamente conectadas como las analizadas en la Sección(2.3.2.1).

Es interesante tomar en cuenta la cantidad de capas de la red y la cantidad de parámetros entrenables en la misma. En este caso, se cuentan con más de 9 millones de parámetros para una red de 10 capas. La cantidad de parámetros está relacionada directamente con la cantidad de memoria RAM usada en el proceso de predicción, es decir, mientras más parámetros tenga la red, más memoria RAM necesita el sistema para poder ejecutar la inferencia con dicha red.

En la Figura(3.32) se puede apreciar la implementación de la red en Tensorflow visualizada mediante la herramienta Tensorboard. Una herramienta de visualización y monitoreo que incluye la librería Tensorflow.

3.7.6.2. Red neuronal convolucional

Por su parte, como se ha establecido en el Capítulo(1), el objetivo de este proyecto es lograr implementar y validar el funcionamiento de una red neuronal convolucional para la tarea de generación de comandos para la conducción autónoma. Es en este sentido que se ha diseñado la arquitectura detallada en la Tabla(3.15) siguiendo las convenciones usadas en diversos proyectos, en especial en la implementación de la red fin a fin de Nvidia [3].

En este caso, la red cuenta con 12 capas, de las cuales 11 son capas convolucionales, y se cuenta con una unidad densamente conectada en la salida para poder obtener un escalar. Se

Capa	Tipo	Unidades (Filtros)	Dimensiones de salida	Parámetros entrenables	Función de Activación
input_1	InputLayer	-	(224,224,3)	0	
conv2d_1	Conv2D	3 (1 x 1)	(224,224,3)	12	ReLU
dense_1	Dense	64	64	9633856	ReLU
dropout_1	Dropout	-	64	0	-
dense_2	Dense	32	32	2080	ReLU
dropout_2	Dropout	-	32	0	-
dense_3	Dense	16	16	528	ReLU
dropout_3	Dropout	-	16	0	-
dense_4	Dense	12	12	204	ReLU
dense_5	Dense	1	1	13	Tanh
		Total Capas		10	
		Total Parámetros		9636693	

Tabla 3.14: Arquitectura de la red neuronal densamente conectada. Fuente: Elaboración propia

puede diferenciar tres macrocapas que cuentan con dos capas convolucionales y una capa de max pooling a la salida, esta arquitectura se ha escogido tomando en cuenta que las capas convolucionales generan representaciones mediante mapas de características en su salida y las capas de max pooling dejan pasar solamente las activaciones máximas para reducir su dimensión. Es importante también destacar la naturaleza de los filtros convolucionales y su antisimetría. Esto se ha diseñado con el fin de reducir la dimensionalidad en el eje vertical más agresivamente que en el eje horizontal por la naturaleza de las imágenes procesadas. Se puede apreciar la visualización de la implementación en Tensorflow en la Figura(3.33).

Una característica importante de destacar es la cantidad de parámetros utilizados en la arquitectura de red convolucional que ha pasado de más de 9 millones de parámetros a 120 mil. Si bien esto representa una reducción sustancial en la memoria RAM requerida para ejecutar el modelo la cantidad de procesamiento se incrementa por la naturaleza de las capas convolucionales, donde se necesita realizar la operación de convolución para cada filtro.

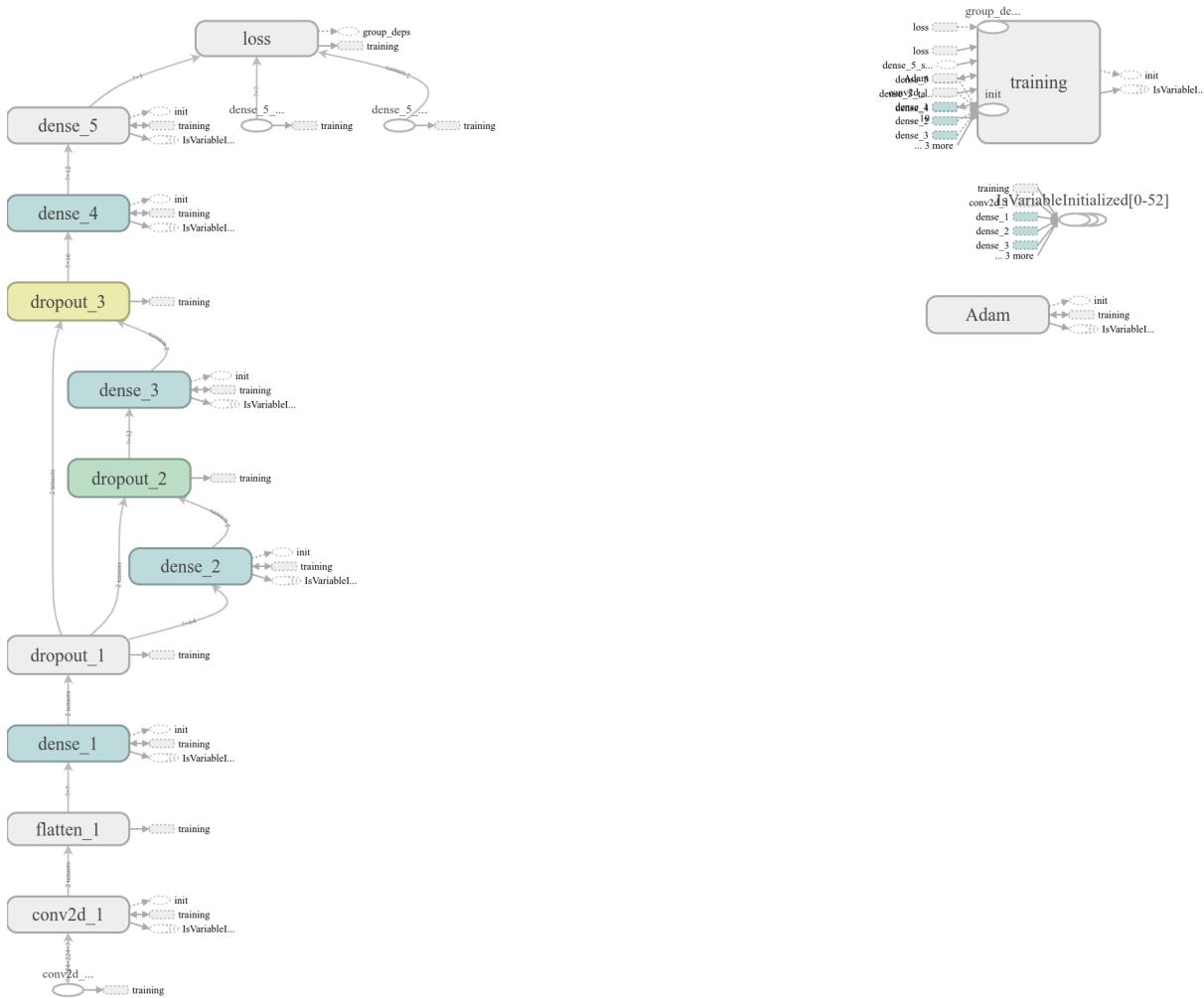


Figura 3.32: Visualización de la arquitectura de la red neuronal densamente conectada. Fuente: Elaboración propia.

3.8. Proceso de Entrenamiento de la Red neuronal Convolucional

3.8.1. Caracterización del conjunto de datos

Antes de entrenar la red neuronal directamente con todos los datos obtenidos en una sesión de entrenamiento es necesario analizar las características y calidad del conjunto de entrenamiento, pues el desempeño final de la red es enteramente dependiente a los datos que se usan en el entrenamiento.

Capa	Tipo	Unidades (Filtros)	Dimensiones de salida	Parámetros entrenables	Función de Activación
input_1	InputLayer	-	(224,224,3)	0	
conv2d_1	Conv2D	16 (3 x 5)	(222,220,16)	736	ReLU
conv2d_2	Conv2D	16 (3 x 5)	(220,216,16)	3856	ReLU
max_pooling2d_1	MaxPooling2D	-	(55,108,16)	-	-
conv2d_3	Conv2D	32 (3 x 5)	(53,104,32)	7712	ReLU
conv2d_4	Conv2D	32 (3 x 5)	(51,100,32)	15392	ReLU
max_pooling2d_2	MaxPooling2D	-	(12,50,32)	-	-
conv2d_5	Conv2D	64 (3 x 5)	(10,46,64)	30784	ReLU
conv2d_6	Conv2D	64 (3 x 5)	(8,42,64)	61504	ReLU
max_pooling2d_3	MaxPooling2D	-	(2,21,64)	-	-
conv2d_7	Conv2D	64 (3 x 5)	(2,21,4)	260	ReLU
dense_1	Dense	1	1	169	Tanh
		Total Capas		12	
		Total Parámetros		120413	

Tabla 3.15: Arquitectura de la red neuronal convolucional. Fuente: Elaboración propia

3.8.1.1. Balanceo del conjunto de datos

Se dice que un conjunto de datos está balanceado cuando la distribución de probabilidad de las muestras que lo componen se acerca lo más posible a la distribución uniforme, de esta manera, se puede asegurar que existen cantidades similares de muestras para todos los casos a los que se ha sometido al sistema en el proceso de las sesiones de aprendizaje.

En el caso de la conducción autónoma para el seguimiento de una carretera, se tiene que gran parte del tiempo en la sesión de entrenamiento el vehículo avanza en línea recta, es decir, con un valor de desviación de 0 o muy cercano a 0. Esto constituye un riesgo muy grande para el entrenamiento pues la red neuronal puede sufrir de *overfitting* donde, al ver demasiadas muestras similares, simplemente empieza a *memorizar* la salida y deja de aprender, lo que causa que su capacidad de generalización sea gravemente afectada. Se puede apreciar el efecto del *overfitting* en la Figura(3.34). Por su parte, si la red carece de la complejidad necesaria para aproximar la función que define la tarea, se puede sufrir de *underfitting* que, de la misma manera que el *overfitting*, tiene un efecto negativo en el rendimiento final de la red neuronal.

El efecto del desbalanceo se puede analizar en el histograma de la Figura(3.35), donde

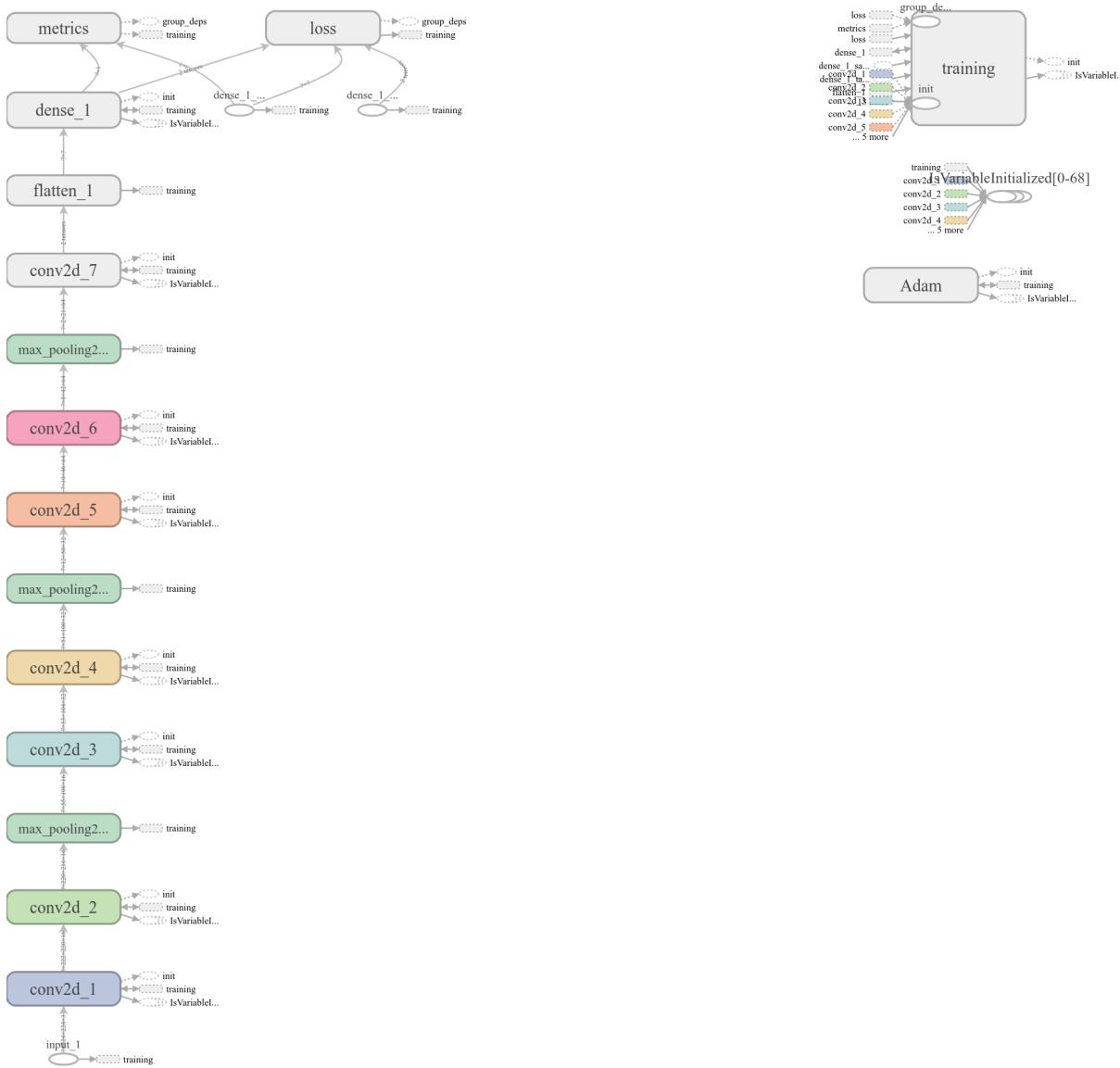


Figura 3.33: Visualización de la arquitectura de la red neuronal convolucional. Fuente: Elaboración propia.

se puede apreciar claramente que existen muchas muestras alrededor del valor de cero. Esto puede ocasionar un sobreentrenamiento en la red en el que la red podría empezar a predecir 0 todo el tiempo y pese a eso obtener un rendimiento relativamente bueno. Otra forma de entender este caso es considerar que los parámetros de la red se estancan en un mínimo local que no cuenta con los requisitos de rendimiento establecidos.

Es por eso que se procede a eliminar la mayor parte de las muestras con el valor de 0 en

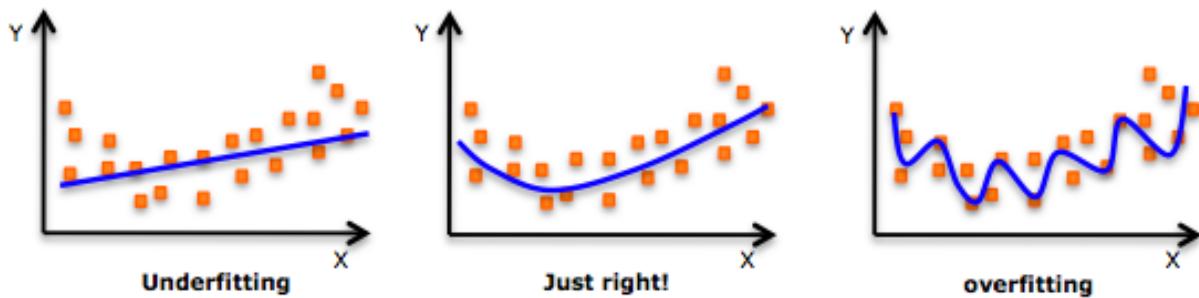


Figura 3.34: Cuando el modelo se aproxima mucho al conjunto de entrenamiento, pierde capacidad de generalización (figura de la derecha). Por su parte, cuando el modelo es demasiado sencillo, no es capaz de representar adecuadamente la naturaleza de los datos y ocurre el underfitting (figura de la izquierda). Se busca un modelo que sea sencillo y sea capaz de aproximar los datos razonablemente (figura central). Fuente: [38].

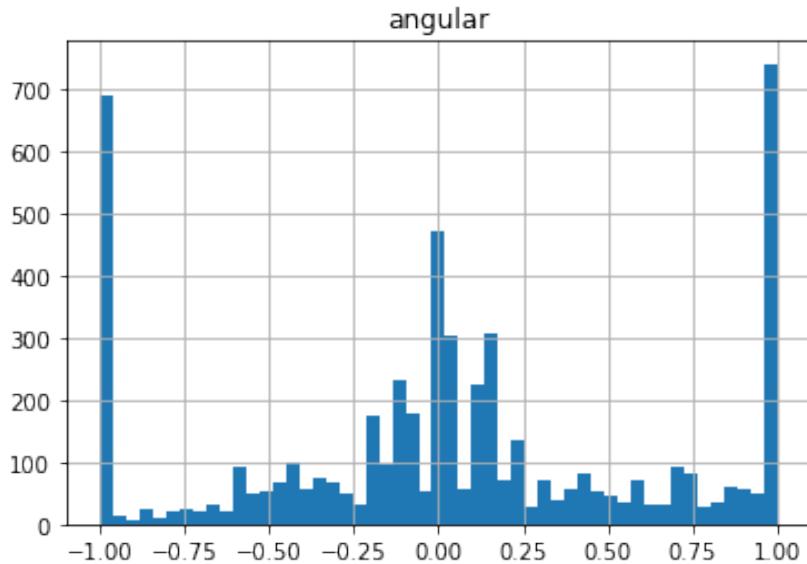


Figura 3.35: Distribución inicial del dataset. Fuente: Elaboración propia.

la salida y muestras con valores muy cercanos a 0. De esta manera, el conjunto de datos se puede balancear de mejor manera teniendo muestras en los extremos, representando casos en los que el carro debe girar con la máxima desviación. En el histograma de la Figura(3.36) se puede apreciar la distribución de las muestras luego de la limpieza de los valores iguales a cero.

Por su parte, en este proceso de balanceo también se ha aplicado las transformaciones definidas en la Sección(3.5.2) para incrementar las muestras que no son cercanas al valor de 0, ya sean positivas o negativas. En la Figura(3.37) se puede observar algunos ejemplos de las imágenes generadas con el proceso de aumentación de datos. Es importante también

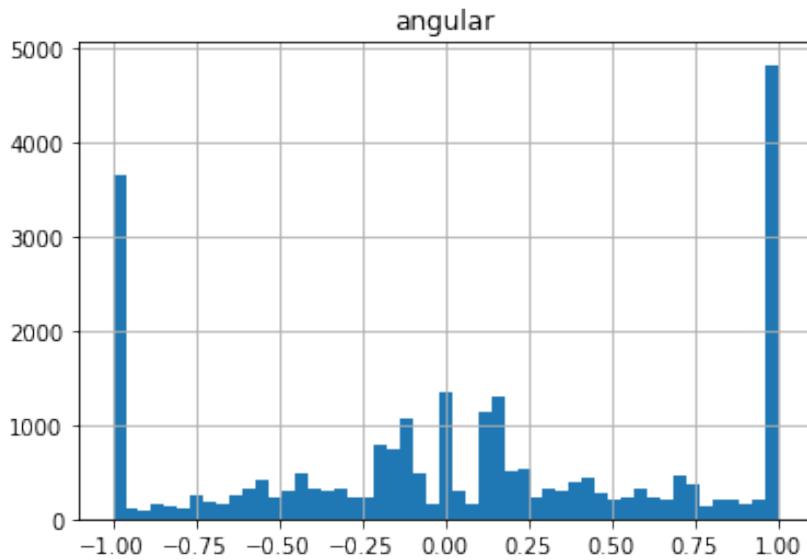


Figura 3.36: Distribución final del dataset. Fuente: Elaboración propia.

mencionar que este proceso incrementa el número de muestras en un 40 % al número de muestras original.

Es importante resaltar la tarea de aumentación de datos, pues, como se pudo observar en la Figura(3.37), a partir de una sola imagen, se han generado tres imágenes adicionales completamente válidas para el entrenamiento. Esto ayuda bastante en el proceso de entrenamiento pues es como si se estuviera expandiendo el conjunto de entrenamiento y sometiendo a la red neuronal a varios casos simultáneamente.

Existen otras formas de aumentar los datos como realizar cambios en el brillo y simular distintos escenarios de iluminación global, en este proyecto, no se han considerado esos casos.

3.8.2. Separación de conjuntos de entrenamiento, pruebas y validación

Tal como se acostumbra para cualquier sistema de aprendizaje automático, el conjunto de datos se ha dividido en tres conjuntos, mostrados en la Figura(3.38) y la cantidad de muestras en el conjunto de datos se puede apreciar en la Tabla(3.16):

3.8.2.1. Conjunto de entrenamiento

Este conjunto de datos se usa para el proceso de entrenamiento de la red neuronal, con los datos de este conjunto se alimenta a la red para el proceso de retropropagación. En cada época, se realiza una pasada completa por el conjunto de entrenamiento. En otras palabras, la red neuronal se entrena en base a lo que *ve* en este conjunto de datos.

Se puede calcular el error de entrenamiento simplemente acumulando el error en cada

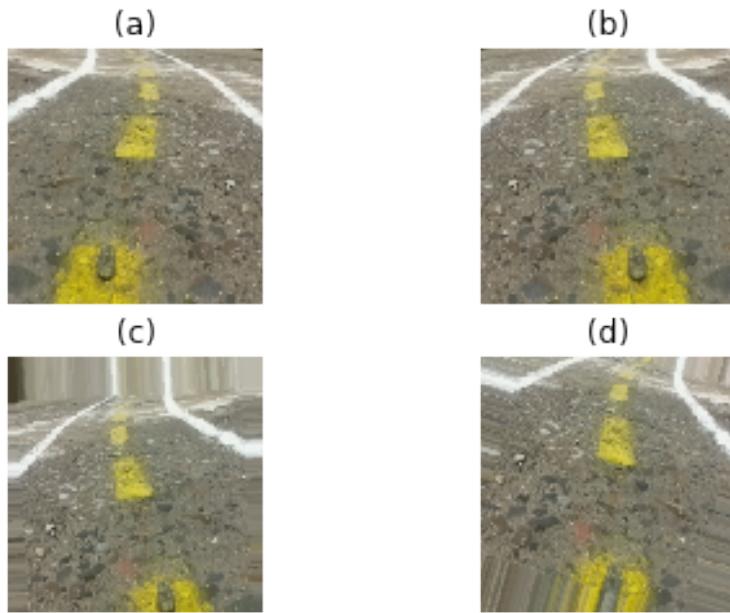


Figura 3.37: Ejemplos del proceso de aumento de datos: (a) imagen original, imagen espejada horizontalmente, (c) imagen desplazada verticalmente, (d) imagen rotada. Fuente: Elaboración propia.

Conjunto de datos	Muestras	Fracción
Entrenamiento	21084	80 %
Prueba	3691	14 %
Validación	1581	6 %
Total	26356	100 %

Tabla 3.16: Cantidad de muestras en los conjuntos de datos. Fuente: Elaboración propia.

mini-batch y comparándolo con los valores reales, sin embargo, el error de entrenamiento no ofrece el panorama completo del rendimiento de la red. Si bien un error de entrenamiento grande indica que el rendimiento de la red es malo, un error de entrenamiento pequeño no siempre indica un buen rendimiento, en este caso, puede tratarse de un sobreentrenamiento.

Del total de muestras en el conjunto de datos se ha tomado el 80 % para el conjunto de entrenamiento, el cual es un valor estándar en tareas de aprendizaje automático.

3.8.2.2. Conjunto de validación

Este conjunto se utiliza para evaluar al sistema durante el entrenamiento, usualmente, corresponde con una pequeña fracción del conjunto de datos, con el cual se analiza el error

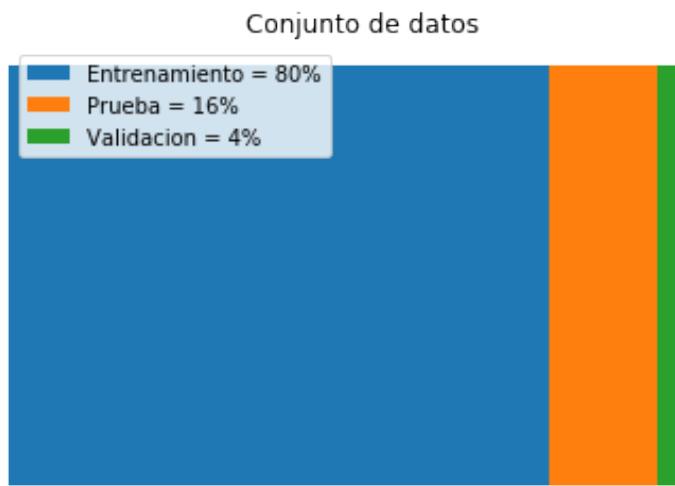


Figura 3.38: Separación del conjunto de datos en conjunto de entrenamiento, prueba y validación. Fuente: Elaboración propia.

del modelo en cada época. Es decir, que el conjunto de validación puede ofrecer una idea de cómo está evolucionando el entrenamiento de la red.

Es importante destacar que el conjunto de validación no se utiliza en el entrenamiento, mas bien representa una medida de evaluación que afecta el entrenamiento de la red de manera indirecta. Por ejemplo, puede existir el caso de que el error de entrenamiento esté disminuyendo conforme pasan las épocas pero el error de validación comienza a incrementar. Este caso puede ser el indicio de un sobreentrenamiento en la red.

La fracción de muestras dedicada al conjunto de validación es del 6 % del total de muestras disponibles.

3.8.2.3. Conjunto de prueba

El conjunto de prueba se usa para evaluar el modelo una vez se ha terminado de entrenar, el error en el conjunto de prueba ofrece el panorama completo del rendimiento de la red. Con el error sobre el conjunto de prueba se puede obtener una medida final del rendimiento de la red.

Normalmente la fracción de muestras para el conjunto de prueba es mayor al del conjunto de validación, en el presente proyecto, se ha utilizado un 14 % del total de muestras del conjunto de datos para la tarea de prueba del modelo.

Capítulo 4

Análisis y discusión de resultados

4.1. Proceso de entrenamiento

El primer paso para poder verificar la validez del sistema de predicción con la red neuronal es el análisis de las curvas de entrenamiento. Se puede extraer información valiosa acerca del rendimiento observando la naturaleza de dichas curvas. Se ha trabajado con el conjunto de datos explorado en la Sección(3.8.1) que cuenta con un total de 26356 muestras entre datos reales y datos aumentados. Se procede a explorar la naturaleza de las curvas de entrenamiento.

4.1.1. Análisis de las curvas de entrenamiento

Las curvas de entrenamiento representan una herramienta bastante valiosa a la hora de evaluar el proceso de entrenamiento de un algoritmo de aprendizaje. A través de las mismas se puede extraer información sobre la convergencia del entrenamiento, la evolución del mismo y también, gracias al proceso de validación cruzada, se puede analizar la presencia de sobreentrenamiento en el sistema.

4.1.1.1. Error de entrenamiento

En la Figura(4.1) se puede observar la evolución del error de entrenamiento para ambas arquitecturas implementadas.

Observando las curvas de error de entrenamiento se pueden realizar las siguientes observaciones con respecto a la red neuronal convolucional:

- **El error de entrenamiento inicial es menor:** Esto indica que la capacidad de ajuste de parámetros es más eficaz en la red neuronal pues en la primera época es capaz de obtener un error de entrenamiento 7 veces menor. A esta propiedad se le llama eficacia a nivel de muestras.
- **El error mínimo es menor:** Esto indica que la red convolucional se aproxima mejor al conjunto de entrenamiento debido a las representaciones generadas por las capas convolucionales internas. En el caso de la red tradicional, se puede observar que el mínimo

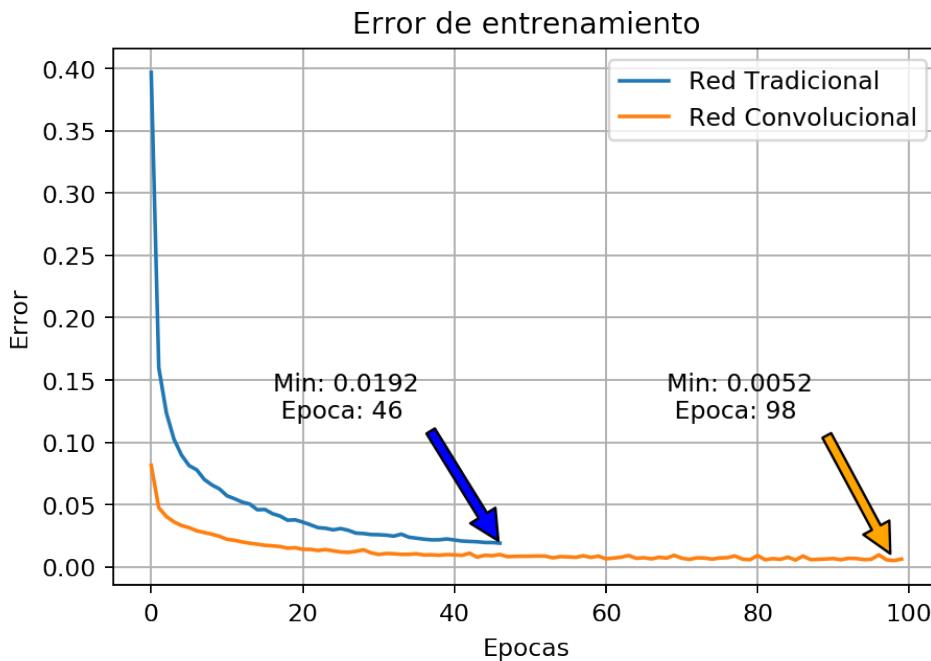


Figura 4.1: Error de entrenamiento. Fuente: Elaboración propia.

se mantiene en un valor casi cuatro veces mayor que en la red convolucional. Esto ocurre por la limitada capacidad de generar representaciones internas de imágenes de una red con capas densamente conectadas pues en las primeras capas se trabaja a nivel de pixeles mientras que en una capa convolucional se procesan mapas de características.

- **Convergencia más rápida:** La convergencia hacia el valor mínimo comienza a notarse alrededor de la época número 20, en contraste con la red tradicional que comienza a converger a partir de la época 35. La convergencia está directamente relacionada con el tiempo de entrenamiento requerido por cada arquitectura.

Considerando los aspectos anteriormente mencionados, se puede observar claramente que el rendimiento en el conjunto de entrenamiento de la red convolucional es superior al de la red tradicional.

4.1.1.2. Error de validación

Los resultados obtenidos en el análisis del error de entrenamiento no son suficientes para determinar si la red neuronal convolucional tiene un mejor rendimiento general que la red tradicional. Para poder obtener un mejor panorama, se puede usar el error de validación, que básicamente corresponde con el cálculo del error de predicción de la red en cada época contra un conjunto de datos nunca antes visto, llamado el conjunto de validación. El error de validación brinda información acerca de la capacidad de generalización de la red. En la Figura(4.2), se puede observar la evolución del error de validación para ambas arquitecturas.

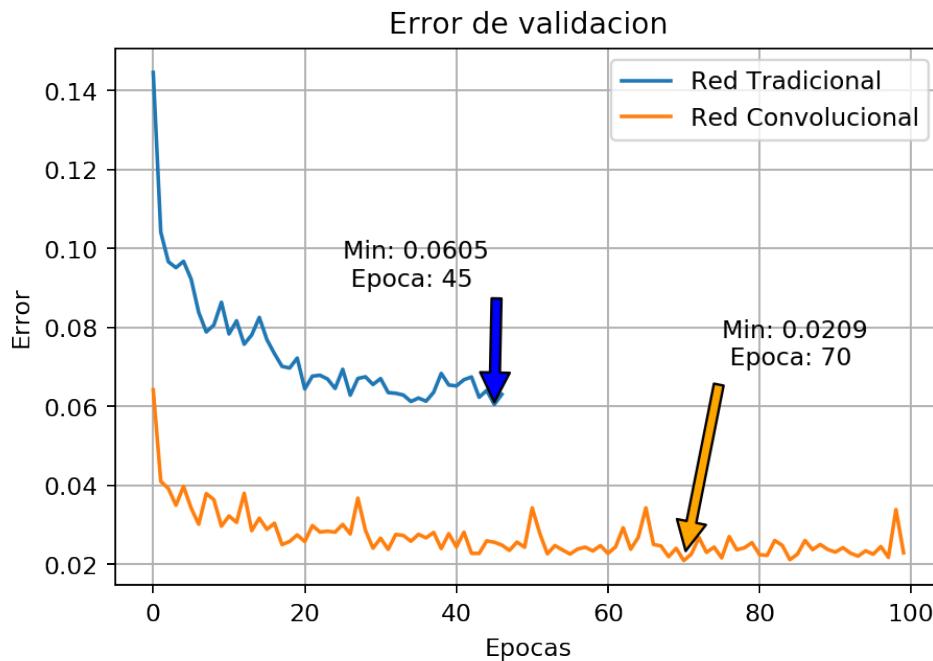


Figura 4.2: Error de validación. Fuente: Elaboración propia.

De manera similar al análisis de las curvas del error de entrenamiento se puede realizar algunas observaciones concernientes a las curvas del error de validación:

- **El error de inicial es menor:** Es un indicativo que la capacidad de generalización de la red convolucional es mucho mayor desde la primera época. Es decir, que con solamente una pasada por el conjunto de entrenamiento, tiene una capacidad mayor de realizar predicciones cercanas a la realidad en casos nunca antes vistos, como son las muestras del conjunto de validación.
- **El error mínimo es menor:** En este caso, el error de validación mínimo indica que la capacidad de generalización de la red convolucional es mayor al de la red tradicional. Si el error de la red convolucional fuera mayor, esto podría indicar sobreentrenamiento de la red. En este caso, se puede observar que se llega al mínimo alrededor de la época número 70.
- **Convergencia:** La convergencia del error de validación indica que ambas redes neuronales no están siendo sobreentrenadas, sin embargo, los valores del error son menores para la red convolucional mostrando que su rendimiento y capacidad de generalización son mayores al de una red tradicional.

4.1.2. Puntajes y errores en el conjunto de prueba

Más allá del análisis de las curvas de entrenamiento, se puede evaluar al modelo entrenado en base a sus puntajes o *scores* definidos para este tipo de tarea. En el presente proyecto, se evaluó a los modelos en base a los puntajes definidos en la Tabla(4.1).

Métrica	Nombre	Definición
MSE	Error Cuadrático Medio	Ecuación(2.16)
MAE	Error Absoluto Medio	Ecuación(2.17)
R^2	Coeficiente de Determinación	Ecuación(2.18)

Tabla 4.1: Métricas para la evaluación del modelo. Fuente: Elaboración propia.

Los resultados de la evaluación del modelo sobre el conjunto de prueba se pueden observar en la Tabla(4.2) donde se nota claramente la superioridad en todas las métricas de la red neuronal convolucional.

Modelo	MSE	MAE	R^2
Tradicional	0.0254	0.0976	0.9278
Convolucional	0.0160	0.0872	0.9484

Tabla 4.2: Evaluación de puntajes sobre el conjunto de prueba. Fuente: Elaboración propia.

Se puede observar en detalle algunas predicciones en la Tabla(4.3), de la cual se debe rescatar principalmente el signo de la predicción que indica la dirección de viraje correcta.

Nótese que la única predicción errónea en dirección de la muestra tiene que ver con un valor nulo, sin embargo, el error de la predicción es muy similar a las otras muestras.

Valor real	Predicción	Diferencia	Predicción de signo correcta
-0.297	-0.243	0.054	✓
1.0	0.921	0.079	✓
0.149	0.22	0.07	✓
-1.	-0.968	0.032	✓
-0.	0.069	0.069	✗

Tabla 4.3: Muestra de predicciones de la red convolucional. Fuente: Elaboración propia.

4.2. Pruebas

Pese a que se ha validado la efectividad de la red neurona convolucional analizando el error en los conjuntos de entrenamiento, validación y prueba, es necesario realizar un análisis cualitativo de los resultados en base a pruebas de campo del modelo. Se procede a analizar el rendimiento y generación de representaciones internas de la red neuronal convolucional dadas nuevas muestras que ingresan al modelo. Se presta especial atención al signo de la predicción.

4.2.1. Análisis del rendimiento en pruebas de campo

Dada la naturaleza de la tarea, es importante analizar de manera detallada las predicciones que está arrojando la red neuronal para casos nuevos. Se ha grabado una nueva sesión de muestras para evaluar el rendimiento del sistema. Los resultados de esta evaluación se pueden observar en la Figura(4.3).

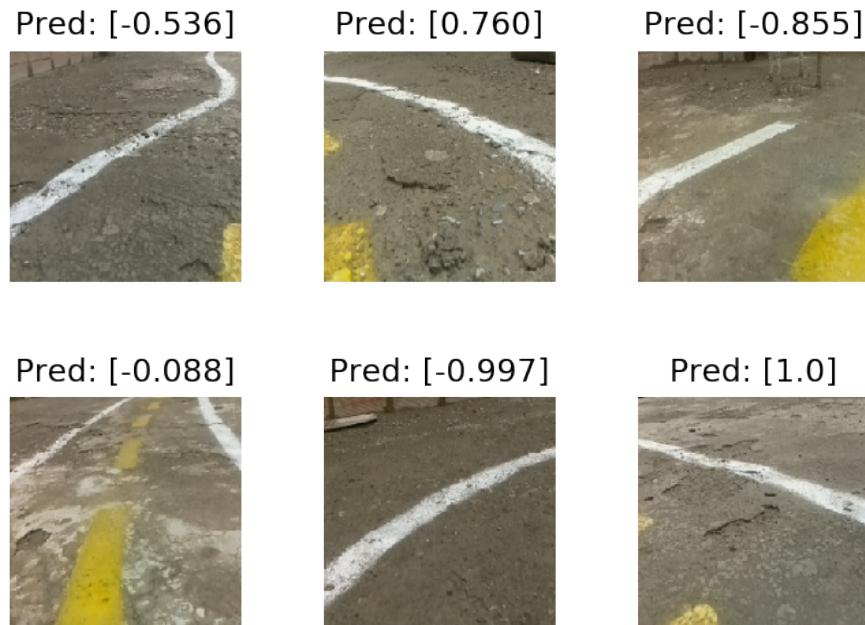


Figura 4.3: Ejemplos de predicción de la red convolucional. Fuente: Elaboración propia.

De la anterior figura, es importante tomar en cuenta que el signo de la predicción indica la dirección en la que se debe mover el vehículo dada la imagen de la carretera. El signo correcto de la predicción es fundamental para el funcionamiento adecuado del sistema. Una predicción con signo errado puede llevar a la inestabilidad del sistema.

También se puede notar que la predicción es correcta pese a que no se puedan observar ambas líneas delimitadoras del carril e incluso, en casos en los que la línea no cruza completamente la imagen.

4.2.2. Análisis de representaciones internas

Ya se ha discutido la importancia de las capas convolucionales para la generación automática de representaciones internas de los datos de entrada. En la Figura(4.4) se puede visualizar los filtros de la primera capa convolucional.

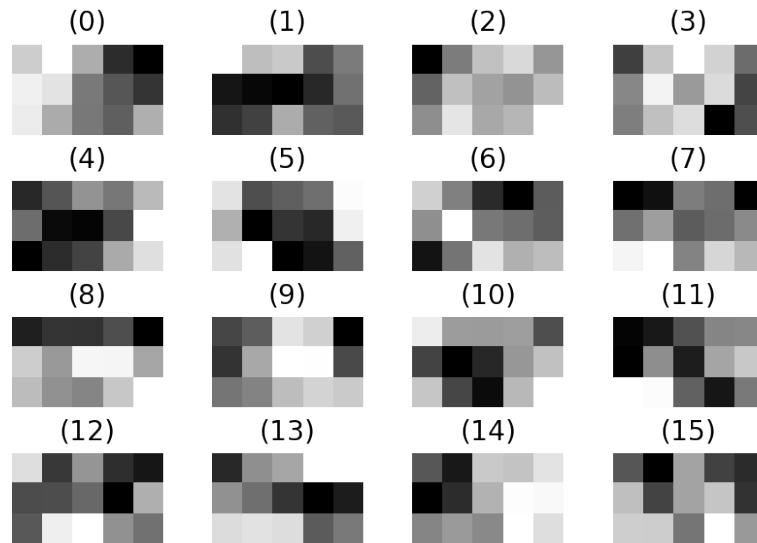


Figura 4.4: Filtros de la primera capa convolucional. Fuente: Elaboración propia.

Es interesante analizar la naturaleza de los filtros generados en el proceso de aprendizaje. Por ejemplo, en el filtro (4.4,0), se puede apreciar claramente que el filtro está dando importancia a bordes con una inclinación hacia la derecha, de manera similar, el filtro (4.4,5) detecta bordes inclinados hacia la izquierda.

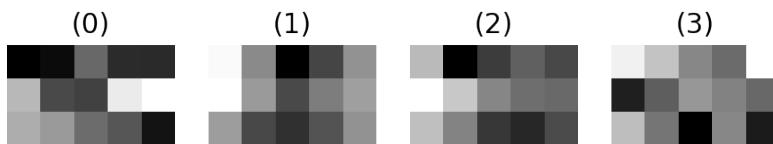


Figura 4.5: Filtros de la penúltima capa convolucional. Fuente: Elaboración propia.

En el caso de los filtros de las capas superiores, se puede observar que las representaciones son mucho más abstractas. Por ejemplo, en el filtro (4.5,0) se da importancia a activaciones que aparezcan a la derecna, en el (4.5,2) a la izquierda y en el (4.5,4) a activaciones que aparecen a ambos lados, cuando el vehículo está centrado en la carretera.

Lo más importante en este análisis es resaltar el hecho de que la red neuronal ha generado estos filtros exclusivamente mediante el proceso de aprendizaje basado en retropropagación

de manera automática. En ningún momento en la etapa de diseño, se ha incorporado algún criterio indicando que las líneas de la carretera tendrían bordes inclinados a la izquierda o derecha o aparecerían a los costados de la imagen. Esto demuestra la impresionante capacidad de las redes neuronales convolucionales para analizar imágenes de manera natural.

Posteriormente, se puede analizar las activaciones que generan en las capas convolucionales algunas imágenes obtenidas en la etapa de pruebas del prototipo. Se analizarán tres casos en los que la red establezca predicciones de viraje a la izquierda, a la derecha y al centro.

4.2.2.1. Giro a la izquierda

En la Figura(4.6) se puede observar el paso de una imagen en la que el vehículo tiene que virar hacia la izquierda por las capas convolucionales de la red neuronal.

Los mapas de características generados para esta imagen denotan activaciones con valor alto en las regiones donde están presentes las líneas de la carretera. A medida que se avanza en las capas convolucionales, se puede observar un máximo en el lado izquierdo de la imagen. Este máximo corresponde con el valor de la predicción que se puede interpretar como un comando de viraje máximo a la izquierda.

4.2.2.2. Giro a la derecha

En la Figura(4.7) se puede observar el paso de una imagen en la que el vehículo tiene que virar hacia la derecha por las capas convolucionales de la red neuronal.

Los mapas de características generados para esta imagen denotan activaciones con valor alto en las regiones donde están presentes las líneas de la carretera. A medida que se avanza en las capas convolucionales, se puede observar un máximo en el lado derecho de la imagen. Este máximo corresponde con el valor de la predicción que se puede interpretar como un comando de viraje a la derecha.

4.2.2.3. Conducir hacia adelante

En el caso en el que el vehículo se encuentre centrado, el comando de control debe aproximarse a cero, pues este valor representa que se tiene que mantener la dirección actual. En la Figura(4.8) se observa cómo el máximo se encuentra cerca al centro de la imagen, lo que significa que el vehículo debe mantener su dirección.

De esta manera se ha podido observar la naturaleza del procesamiento que es capaz de hacer una red neuronal convolucional y la forma en la que las representaciones internas que se generan en el aprendizaje son válidas para la tarea de regresión.

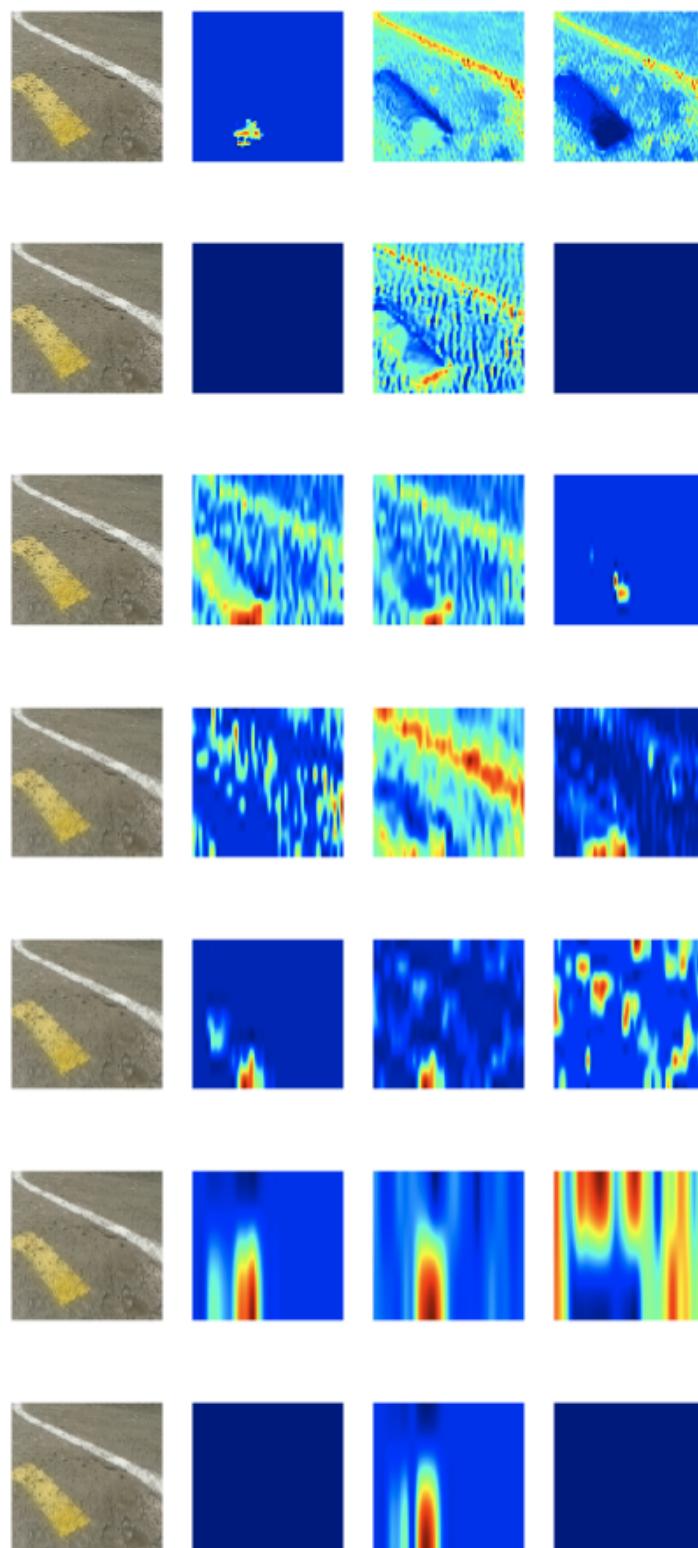


Figura 4.6: Activaciones para una imagen con giro a la izquierda, el valor de la predicción es 1. Fuente: Elaboración propia.

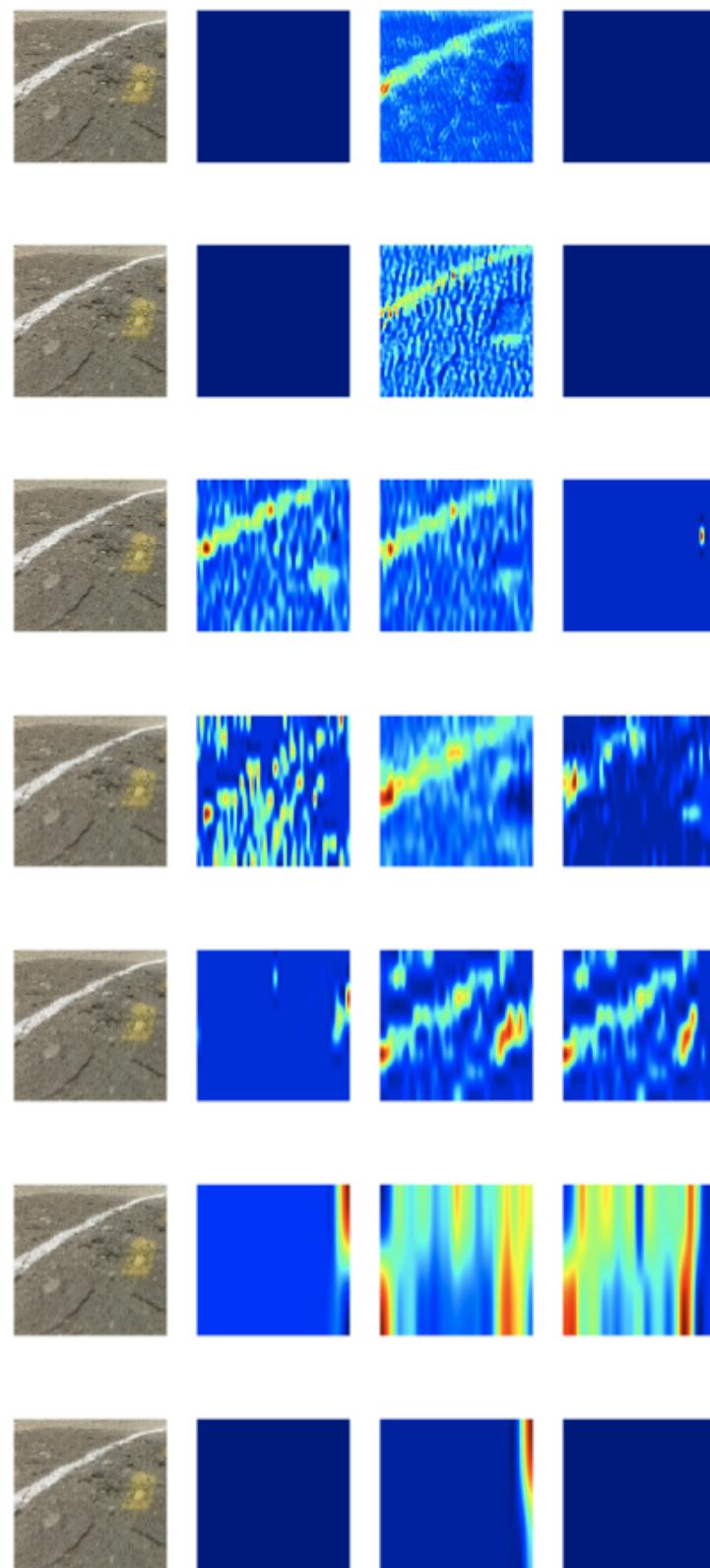


Figura 4.7: Activaciones para una imagen con giro a la derecha, el valor de la predicción es -0.84. Fuente: Elaboración propia.

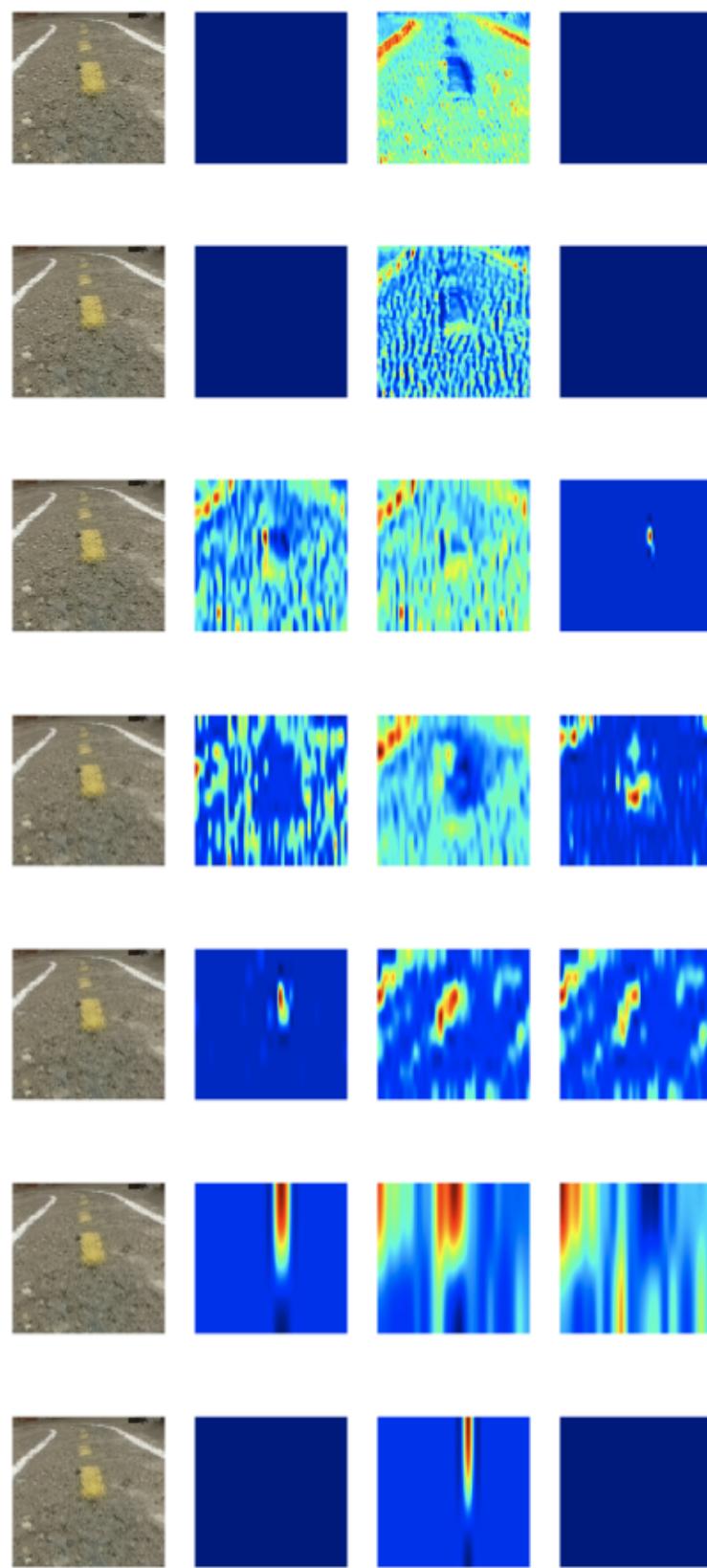


Figura 4.8: Activaciones para una imagen con dirección hacia adelante, el valor de la predicción es -0.1. Fuente: Elaboración propia.

Capítulo 5

Conclusiones y recomendaciones

5.1. Conclusiones

Se detallan las conclusiones correspondientes con cada objetivo específico planteado:

- *Estudiar los aspectos concernientes al desarrollo de sistemas de conducción autónoma y sistemas de aprendizaje.*

Conclusión: Se estudiaron los aspectos relacionados al desarrollo de sistemas de conducción autónoma, en principio, analizando su origen y potencialidad en la Sección(1.1). Para posteriormente estudiar los conceptos básicos de aprendizaje automático, aprendizaje profundo, redes convolucionales y el proceso de entrenamiento de una red neuronal.

- *Analizar los requerimientos de un sistema de conducción autónoma capaz identificar y mantener su carril mientras se conduce.*

Conclusión: Se ha introducido el esquema de la arquitectura de un sistema de conducción autónomo y se han planteado los requisitos y funcionalidades que debe tener el sistema en su conjunto en la Sección(3.1) y de las características de los subsistemas de los que se compone: el subsistema de control y actuación en la Sección(3.1.3), el subsistema de adquisición de datos y entrenamiento en la Sección(3.1.4), el subsistema de inferencia y control autónomo en la Sección(3.1.5).

- *Diseñar la arquitectura de un sistema de conducción autónoma en base a los requerimientos previamente establecidos.*

Conclusión: Se ha planteado la arquitectura general del sistema en base a las características analizadas en el Capítulo(1) y (2) en la Sección(3.1). Considerando todos los alcances y limitaciones planteados por el proyecto en el inicio. Por su parte, también se han definido las herramientas de hardware en la Sección(3.2) y las herramientas de software en la Sección(3.2) que hicieron posible la implementación de todos los subsistemas asociados.

- *Diseñar el subsistema de adquisición de datos y entrenamiento para tareas de conducción autónoma.*

Conclusión: Se ha diseñado el Subsistema de Adquisición de Datos y Entrenamiento en base a la arquitectura planteada en la Sección(3.1.4) en la Sección(3.5) considerando los detalles de implementación, algoritmos y características necesarias para las tareas de adquisición de datos, aumentación de datos y entrenamiento de modelos de redes neuronales explorados en el Capítulo(3). Los detalles del proceso de diseño y entrenamiento de las arquitecturas de redes neuronales planteadas se exploran con detalle en la Sección(3.7) para el diseño de la red neuronal y en la Sección(3.8) para el entrenamiento.

- *Diseñar el subsistema de control y actuación para la conducción autónoma de un vehículo con características similares a las de un vehículo doméstico real.*

Conclusión: Se ha diseñado el Subsistema de Control y Actuación de acuerdo con la arquitectura planteada en la Sección(3.1) en la Sección(3.4). Se ha considerado las características y requerimientos planteados para dicho subsistema en las tareas de control de tiempo real implementado en un microcontrolador, interfaz con los actuadores y sensores sobre la plataforma de comunicación de ROS. Se han expuesto los detalles de implementación tanto a nivel de hardware como de software procurando que el diseño de este subsistema sea modular.

- *Diseñar el subsistema de inferencia y control autónomo basado en el uso de redes neuronales convolucionales.*

Conclusión: Por su parte, se ha diseñado a detalle el Inferencia y Control autónomo en base a todos los fundamentos teóricos de redes neuronales exploradas en la Sección(??). Los detalles de la implementación se pueden encontrar en la Sección(3.6) donde se abunda en la implementación de los módulos que componen este subsistema para las tareas de predicción de dirección con la red neuronal convolucional, detección de obstáculos con un sensor de proximidad y el algoritmo del piloto automático implementados como nodos de ROS para garantizar la modularidad del sistema.

- *Analizar los resultados del entrenamiento e implementación del subsistema de inferencia y control autónomo.*

Conclusión: El análisis de los resultados del entrenamiento de la red neuronal convolucional se ha explorado en la Sección(4.1) pudiendo hacer la comparación de resultados en errores de entrenamiento, validación y prueba de dos arquitecturas: una con una red neuronal tradicional o densamente conectada, y otra con una red neuronal convolucional. En base a los resultados y puntajes en el entrenamiento obtenidos se puede concluir que una red neuronal convolucional es capaz de cumplir la tarea de generar comandos de control para la tarea de conducción autónoma.

Por otro lado, la implementación de la red y los resultados en pruebas de campo se han explorado en la Sección(4.2) en la cual se pudo observar la efectividad de la red neuronal convolucional para generar representaciones internas relevantes y útiles. En base a este

análisis se puede concluir que, efectivamente, una la red convolucional entrenada es capaz de generar representaciones internas relevantes en muestras nunca antes vistas, en otras palabras, que la capacidad de generalización es aceptable.

- *Realizar pruebas de rendimiento y análisis comparativos en el sistema implementado.*

Conclusión: Con el fin de realizar un análisis comparativo entre distintas implementaciones de redes neuronales para la tarea especificada por este proyecto, se ha diseñado y entrenado un par de arquitecturas de red neuronal sobre las cuales se realiza un análisis comparativo de rendimiento en base a indicadores y puntajes estándar en el campo de la estadística en la Sección(4.1.2).

Por su parte, también se ha analizado las predicciones generadas por la red convolucional, para distintos casos, explorando la naturaleza de las representaciones generadas por la misma en la Sección(4.2.2). Por tanto, se puede concluir que la implementación de la red neuronal convolucional planteada se ha realizado exitosamente.

5.2. Recomendaciones

Luego de haber analizado y generado conclusiones relativas a los resultados obtenidos en base a los objetivos planteados en la etapa inicial del presente proyecto, se plantea una serie de recomendaciones:

- Debido a la capacidad de generalización que puede lograrse con una red neuronal convolucional se recomienda generar distintos conjuntos de entrenamiento en condiciones climáticas diversas. Esto incrementará la complejidad de la red convolucional y logrará que las predicciones de la red sean más robustas en cambios de iluminación causados por distintos aspectos climáticos.
- Por su parte, gracias a la naturaleza modular del presente proyecto, se recomienda diseñar, entrenar y validar arquitecturas de redes neuronales distintas o con variaciones a una red neuronal convolucional tradicional. Se recomienda, por ejemplo, implementar una red convolucional recurrente que sea capaz de tomar en cuenta el estado anterior de la misma. Implementar otros algoritmos de predicción y visión artificial también ayudarían al desarrollo de sistemas más robustos.
- Se recomienda también extender la implementación de la red a una tarea de clasificación categórica para comparar el rendimiento entre el entrenamiento y rendimiento obtenidos en el problema planteado como una regresión y una clasificación.
- Debido a la modularidad del sistema planteado en el presente proyecto, se recomienda implementar el mismo sistema fin a fin para otros modelos de vehículos, como pueden ser un robot de tracción diferencial u otro modelo.

- Es también importante poder extender el sistema presentado en este proyecto con tareas de control y detección avanzadas, incluyendo en el flujo de trabajo módulos de planificación de trayectorias, detección de objetos y programación de misiones.
- Dada la gran potencialidad en el área de sistemas de conducción autónoma se recomienda que se pueda crear una línea de investigación dedicada al diseño e implementación de sistemas robóticos inteligentes en la cual se agrupen esfuerzos para el desarrollo de cada uno de los subsistemas que componen un vehículo autónomo.
- Por último, la forma en la que ha sido implementado el presente proyecto permite modificarlo y extenderlo de distintas maneras y a distintos niveles. Se recomienda considerar este proyecto como una plataforma de desarrollo sobre la cual se puedan implementar diversos algoritmos y aplicaciones útiles para el desarrollo de nuestra sociedad.

Bibliografía

- [1] Y. LeCun, E. Cosatto, J. Ben, U. Muller, and B. Flepp, “Dave: Autonomous off-road vehicle control using end-to-end learning,” Technical Report DARPA-IPTO Final Report, Courant Institute/CBLL, <http://www.cs.nyu.edu/yann/research/dave/index.html>, Tech. Rep., 2004.
- [2] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney, “Stanley: The robot that won the DARPA grand challenge,” *Journal of Field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.
- [3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, “End to end learning for self-driving cars,” *arXiv preprint arXiv:1604.07316*, 2016.
- [4] “Linear regression,” Nov 2018. [Online]. Available: https://en.wikipedia.org/wiki/Linear_regression
- [5] O. Cameron, “An introduction to lidar: The key self-driving car sensor,” May 2017. [Online]. Available: <https://news.voyage.auto/an-introduction-to-lidar-the-key-self-driving-car-sensor-a7e405590cff>
- [6] R. Szeliski, *Computer Vision*. Springer-Verlag GmbH, 2011. [Online]. Available: https://www.ebook.de/de/product/9630757/richard_szeliski_computer_vision.html
- [7] T. M. Mitchell, *Machine Learning*. PN, 1990. [Online]. Available: <https://www.amazon.com/Machine-Learning-Tom-M-Mitchell/dp/1259096955?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1259096955>
- [8] N. Viswarupan, “K-means data clustering – towards data science,” Jul 2017. [Online]. Available: <https://towardsdatascience.com/k-means-data-clustering-bce3335d2203>
- [9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.

- [10] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer-Verlag New York Inc., 2006. [Online]. Available: https://www.ebook.de/de/product/5324937/christopher_m_bishop_pattern_recognition_and_machine_learning.html
- [11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, p. 533, 1986.
- [12] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [13] H. N. Mhaskar and C. A. Micchelli, “How to choose an activation function,” in *Advances in Neural Information Processing Systems*, 1994, pp. 319–326.
- [14] T.-y. Wang, “From perceptron to deep learning,” Jan 2016. [Online]. Available: <http://databeauty.com/blog/2018/01/16/From-Perceptron-to-Deep-Learning.html>
- [15] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.
- [16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [17] “Mean squared error,” Sep 2018. [Online]. Available: https://en.wikipedia.org/wiki/Mean_squared_error
- [18] “Mean absolute error,” Sep 2018. [Online]. Available: https://en.wikipedia.org/wiki/Mean_absolute_error
- [19] “Coefficient of determination,” Sep 2018. [Online]. Available: https://en.wikipedia.org/wiki/Coefficient_of_determination
- [20] D. Cornelisse, “An intuitive guide to convolutional neural networks,” Apr 2018. [Online]. Available: <https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050>
- [21] Y. LeCun, K. Kavukcuoglu, C. Farabet *et al.*, “Convolutional networks and applications in vision.” in *ISCAS*, vol. 2010, 2010, pp. 253–256.
- [22] A. Graves and N. Jaitly, “Towards end-to-end speech recognition with recurrent neural networks,” in *International Conference on Machine Learning*, 2014, pp. 1764–1772.
- [23] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end training of deep visuomotor policies,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

- [24] M. D. Gregory (McGill University, *Computational Principles of Mobile Robotics*. CAMBRIDGE UNIVERSITY PRESS, 2010. [Online]. Available: https://www.ebook.de/de/product/9473553/gregory_mcgill_university_montreal_dudek_computational_principles_of_mobile_robotics.html
- [25] “Powering the world’s robots.” [Online]. Available: <http://www.ros.org/>
- [26] “Wiki.” [Online]. Available: <http://wiki.ros.org/>
- [27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: <https://www.tensorflow.org/>
- [28] M. Asjad, “Notes on tensorflow (basics),” Aug 2016. [Online]. Available: <https://medium.com/@asjad/notes-on-tensor-flow-b90ef02b144f>
- [29] F. Chollet *et al.*, “Keras,” <https://keras.io>, 2015.
- [30] “Mbed os documentation.” [Online]. Available: <https://os.mbed.com/docs/v5.10/>
- [31] “Nucleo-f303k8.” [Online]. Available: <https://www.st.com/en/evaluation-tools/nucleo-f303k8.html>
- [32] “Raspberry pi 3 model b.” [Online]. Available: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>
- [33] “Camera module v2.” [Online]. Available: <https://www.raspberrypi.org/products/camera-module-v2/>
- [34] “Vl53l0x - world smallest time-of-flight (tof) ranging sensor.” [Online]. Available: <https://www.st.com/en/imaging-and-photonics-solutions/vl53l0x.html>
- [35] *World’s smallest Time-of-Flight ranging and gesture detection sensor*, ST Microelectronics, 4 2018, rev. 2.
- [36] M. Ferguson, “Rosserial.” [Online]. Available: <http://wiki.ros.org/rosserial>
- [37] R. Agrawal and W. Gramlich, “raspicam_node,” Dec 2018. [Online]. Available: https://github.com/UbiqityRobotics/raspicam_node
- [38] A. Rayon, “El machine learning en la era del big data.” [Online]. Available: <https://blogs.deusto.es/bigdata/tag/overfitting/>

- [39] S. Narayan, "The generalized sigmoid activation function: competitive supervised learning," *Information sciences*, vol. 99, no. 1-2, pp. 69–82, 1997.
- [40] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

Apéndice A

Función sigmoide

Esta función ha sido introducida en la Ecuación(2.6) y representa, históricamente, la función más utilizada en las primeras redes neuronales artificiales porque modela de manera aproximada, una respuesta característica de las neuronas del cerebro [39], pero sobre todo, porque posee las dos características mencionadas anteriormente: naturaleza no lineal y diferenciable. Otra característica llamativa, es que se puede interpretar a la salida como una función de probabilidad, pues sus valores van desde 0 a 1, como se puede apreciar en la Figura(A.1). Usualmente se incluye un parámetro adicional para controlar el *radio de activación* que hace que la función responda con más o menos sensibilidad a su entrada.

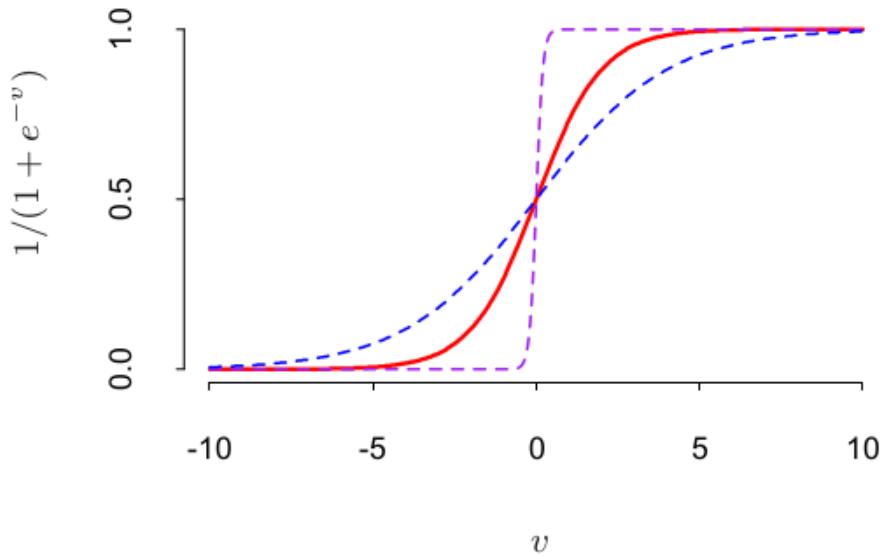


Figura A.1: Gráfico de la función sigmoide $\sigma(v) = 1/(1+e^{(-v)})$ (curva roja), y dos variaciones con un radio de activación de la forma $\sigma(sv)$ con valores $s = 1/2$ (curva azul) y $s = 10$ (curva púrpura) . Fuente: [9]

Pese a las características anteriormente mencionadas, la función sigmoide tiene una gran

desventaja: el llamado *desvanecimiento de gradientes* [40]. Este fenómeno ocurre cuando la activación de una capa oculta tiene valores muy altos o muy negativos, se puede apreciar en la Figura(A.1), que para estos valores, la derivada tiene un valor muy pequeño, aproximándose a cero mientras más grandes sean los valores. Este fenómeno ocasiona que, mientras se realiza la retropropagación de gradientes, dado que el gradiente tiene un valor muy bajo, el ajuste en los pesos sea mínimo, deteniendo así el aprendizaje de la neurona en la que ocurre el fenómeno.

Esta dificultad se presenta especialmente cuando la red neuronal se compone de varias capas ocultas y ha motivado el desarrollo de nuevas funciones de activación que no presenten esta limitación y puedan mantener valores de gradientes adecuados durante el entrenamiento.

Cabe resaltar que, pese a las dificultades con la propagación de los gradientes de la función sigmoide, ésta se suele utilizar bastante en la capa de salida, cuando se trata de clasificación binaria, ya que representa de manera adecuada la noción de probabilidad, lo cual es deseable en este tipo de modelos.

Apéndice B

Código Fuente del Proyecto

B.1. Control de Actuadores en Tiempo Real

main.cpp

```
#include <mbed.h>
#include <Servo.h>
#include <Motor.h>
#include <ros.h>

// ros msgs
#include <geometry_msgs/Twist.h> // sub to Twist message for control
#include <sensor_msgs/Range.h>    // pub to Range message for sensor

DigitalOut led(D13);

Servo steering(D9);           // steering servo
Motor motor(D12, D10, D11); // motor driver

// ROS part
// software utils
Timer t;
Ticker pub_ticker;
// ros part
ros::NodeHandle nh;

// messages
geometry_msgs::Twist control_cmd; // control command is of type twist

// callback prototypes
void ctrlCommandCb(const geometry_msgs::Twist &command);

// subscribers
// ros subscriber for control command
ros::Subscriber<geometry_msgs::Twist> ctrlSub("cmd_vel", ctrlCommandCb);
```

```

// prototypes
// callbacks
void ctrlCommandCb(const geometry_msgs::Twist &command);

int main() {

    motor.period(0.0005);           // frecuencia 2 KHz
    steering = 0;
    // init sensors
    led = 1;
    // ros initialization
    nh.initNode();
    nh.subscribe(ctrlSub);
    //
    led = 0;

    while(1) {
        nh.spinOnce();
        wait_ms(1);
    }
}

// control command callback
void ctrlCommandCb(const geometry_msgs::Twist &command)
{
    led = !led;
    float linear = command.linear.x;
    float angular = command.angular.z / 2 + 0.5; // (command.angular.z * 0.35)
                                                + 0.32;
    steering = angular;
    // testigo = abs(linear * 8);
    motor.speed(linear);
}

```

B.2. Script de entrenamiento

```

train_model.py

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

from keras.models import model_from_json

import numpy as np
import pandas as pd
import bcolz
import threading

from time import time
import os
import sys

```

```
import glob
import shutil

from sklearn.model_selection import train_test_split

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.callbacks import TensorBoard, ModelCheckpoint

from keras.utils import plot_model
import models
from utils import *

class RobocarTrainer(object):

    # data for training
    input_shape=(224,224,3)
    im_shape = (224, 224)

    # train parameters
    batch_size = 32
    n_epochs = 100

    def __init__(self, model_name, model_path, dataset_path, log_path='trainlogs'):
        self.model_name = model_name
        self.model_path = model_path
        self.log_path = log_path
        self.dataset_path = dataset_path

        # creating callbacks
        self.tfBoardCB = TensorBoard('{}/{}_{}/'.format(self.log_path,
                                                       model_name, time()), write_graph=True)

        filepath= model_path + model_name + '_best.h5'

        self.checkpointCB = ModelCheckpoint(filepath, monitor='val_loss',
                                           verbose=1, save_best_only=True, mode='min')

    def LoadDataset(self):

        print('loading_dataset...')
        self.dataset = pd.read_csv(self.dataset_path + 'target.csv')

        self.dataset['imgpath'] = self.dataset.id.apply(file_path_from_db_id,
                                                       args=("{}.bmp", self.dataset_path))

        self.train, self.test = train_test_split(self.dataset, test_size=0.2)
```

```
self.valid, self.test = train_test_split(self.test, test_size=0.7)

self.train_steps = int(self.train.shape[0] / self.batch_size)
self.valid_steps = int(self.valid.shape[0] / self.batch_size)
self.test_steps = int(self.test.shape[0] / self.batch_size)
print('dataset_loaded!')


def Train(self):

    print('loading_model...')
    self.model = models.vanilla(self.input_shape)
    self.model.summary()
    model_json = self.model.to_json()
    with open(self.model_path + self.model_name + '.json', "w") as
        json_file:
            json_file.write(model_json)

    plot_model(self.model, to_file=self.model_name + '.png', show_shapes=
        True)
    print('dataset_size:', self.train.shape[0],
          'train_steps:', self.train_steps,
          'valid_steps:', self.valid_steps,
          'test_steps:', self.test_steps)

    print('hiperparameters:')
    print('batch_size:{}' .format(self.batch_size))

    print('training...')
    self.model.fit_generator(
        generator_from_df(self.train, self.batch_size,
                           self.im_shape, 'angular'),
        steps_per_epoch=self.train_steps,
        epochs=self.n_epochs,
        validation_data=generator_from_df(self.valid,
                                          self.batch_size, self.im_shape, 'angular'),
        validation_steps=self.valid_steps,
        callbacks=[self.tfBoardCB, self.checkpointCB],
        verbose=2
    )
    print('finished_training')
    self.score = self.model.evaluate_generator(
        generator_from_df(self.test, self.batch_size,
                           self.im_shape, 'angular'),
        steps=self.test_steps
    )
    print('loss:', self.score)

def SaveModel(self):
    # serialize model to JSON
    self.model_json = self.model.to_json()
```

```

        with open(self.model_name + '.json', "w") as json_file:
            json_file.write(self.model_json)
        # serialize weights to HDF5
        self.model.save_weights(self.model_name + '.h5')
        print("Saved_model_to_disk")

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Entrenamiento de una red neuronal convolucional')
    parser.add_argument("model_name", help="name of the model to train")
    parser.add_argument("model_path", help="path where the model files will be saved")
    parser.add_argument("dataset_path", help="path where the dataset is saved")

    args = parser.parse_args()

    trainer = RobocarTrainer(args.model_name, args.model_path, args.dataset_path)
    trainer.LoadDataset()
    trainer.Train()
    trainer.SaveModel()

```

B.3. Sincronización de Mensajes

```

msg_sync.py

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

from keras.models import model_from_json

import numpy as np
import pandas as pd
import bcolz
import threading

from time import time
import os
import sys
import glob
import shutil

from sklearn.model_selection import train_test_split

from keras.models import Sequential

```

```
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.callbacks import TensorBoard, ModelCheckpoint

from keras.utils import plot_model
import models
from utils import *

class RobocarTrainer(object):

    # data for training
    input_shape=(224,224,3)
    im_shape = (224, 224)

    # train parameters
    batch_size = 32
    n_epochs = 100

    def __init__(self, model_name, model_path, dataset_path, log_path='trainlogs'):
        self.model_name = model_name
        self.model_path = model_path
        self.log_path = log_path
        self.dataset_path = dataset_path

        # creating callbacks
        self.tfBoardCB = TensorBoard('{}/{}/{}'.format(self.log_path,
                                                       model_name, time()), write_graph=True)

        filepath= model_path + model_name + '_best.h5'

        self.checkpointCB = ModelCheckpoint(filepath, monitor='val_loss',
                                           verbose=1, save_best_only=True, mode='min')

    def LoadDataset(self):

        print('loading_dataset...')
        self.dataset = pd.read_csv(self.dataset_path + 'target.csv')

        self.dataset['imgpath'] = self.dataset.id.apply(
            file_path_from_db_id, args=("%d.bmp", self.dataset_path))

        self.train, self.test = train_test_split(self.dataset, test_size
                                              =0.2)
        self.valid, self.test = train_test_split(self.test, test_size=0.7)

        self.train_steps = int(self.train.shape[0] / self.batch_size)
        self.valid_steps = int(self.valid.shape[0] / self.batch_size)
        self.test_steps = int(self.test.shape[0] / self.batch_size)
```

```
print('dataset_loaded!')  
  
def Train(self):  
  
    print('loading_model...')  
    self.model = models.vanilla(self.input_shape)  
    self.model.summary()  
    model_json = self.model.to_json()  
    with open(self.model_path + self.model_name + '.json', "w") as  
        json_file:  
            json_file.write(model_json)  
  
    plot_model(self.model, to_file=self.model_name + '.png',  
               show_shapes=True)  
    print('dataset_size:', self.train.shape[0],  
          'train_steps:', self.train_steps,  
          'valid_steps:', self.valid_steps,  
          'test_steps:', self.test_steps)  
  
    print('hiperparameters:')    print('batch_size:{}' .format(self.batch_size))  
  
    print('training...')  
    self.model.fit_generator(  
        generator_from_df(self.train, self.  
                           batch_size, self.im_shape, 'angular'),  
        steps_per_epoch=self.train_steps,  
        epochs=self.n_epochs,  
        validation_data=generator_from_df(self.  
                                         valid, self.batch_size, self.im_shape,  
                                         'angular'),  
        validation_steps=self.valid_steps,  
        callbacks=[self.tfBoardCB, self.  
                   checkpointCB],  
        verbose=2  
    )  
    print('finished_training')  
    self.score = self.model.evaluate_generator(  
        generator_from_df(self.test, self.  
                           batch_size, self.im_shape, 'angular'),  
        steps=self.test_steps  
    )  
    print('loss:', self.score)  
  
def SaveModel(self):  
    # serialize model to JSON  
    self.model_json = self.model.to_json()  
    with open(self.model_name + '.json', "w") as json_file:  
        json_file.write(self.model_json)  
    # serialize weights to HDF5
```

```

        self.model.save_weights(self.model_name + '.h5')
        print("Saved_model_to_disk")

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Entrenamiento de una red
        neuronal convolucional')
    parser.add_argument("model_name", help="name_of_the_model_to_train")
    parser.add_argument("model_path", help="path_where_the_model_files_
        will_be_saved")
    parser.add_argument("dataset_path", help="path_where_the_dataset_
        is_saved")

    args = parser.parse_args()

    trainer = RobocarTrainer(args.model_name, args.model_path, args.
        dataset_path)
    trainer.LoadDataset()
    trainer.Train()
    trainer.SaveModel()

```

B.4. Aumentación de datos

```

augmentation.py

from keras.preprocessing.image import load_img, ImageDataGenerator
from keras.preprocessing.image import img_to_array

from pandas.plotting import bootstrap_plot
import numpy as np
import pandas as pd
import bcolz
import threading

import cv2

from utils import *

import os
import sys
import glob
import shutil
import matplotlib.pyplot as plt
get_ipython().magic(u'matplotlib_inline')

dataset1 = pd.read_csv('../datasets/dataset/target.csv')
dataset1['imgpath'] = dataset1.id.apply(file_path_from_db_id)
```

```
datagen = ImageDataGenerator(  
    rotation_range=20,  
    height_shift_range=0.2,  
    shear_range=0.15,  
    zoom_range=0.15,  
    fill_mode='nearest')  
  
test_gen = batch_generator(dataset1, 1, (224,224), 'angular', process=  
    False, shuffle=False)  
img, tg = test_gen.next()  
img = img[0]  
img = (img)*255  
plt.subplot(131)  
plt.imshow(img)  
  
for i, row in dataset1.sample(1).iterrows():  
    img3 = img_to_array(load_img(row['imgpath']))  
  
    plt.subplot(132)  
    plt.imshow(img3*255)  
  
    img2, tg2 = horizontal_flip(img, tg)  
    plt.subplot(133)  
    plt.imshow(img2)  
  
datagen = ImageDataGenerator(  
    rotation_range=20,  
    height_shift_range=0.2,  
    shear_range=0.15,  
    zoom_range=0.15,  
    fill_mode='nearest')  
  
test_gen = batch_generator(dataset1, 1, (224,224), 'angular', process=  
    False, shuffle=False)  
  
offset = dataset1.shape[0] + 1  
id_offset = dataset1['id'].max() + 1  
i = 0  
newData = []  
for idx, row in dataset1.iterrows():  
    # generador auxiliar  
  
    ## extraemos la imagen en una variable auxiliar  
    img = img_to_array(load_img(row['imgpath']))  
    target = row['angular']  
    ## aplicamos la transformacion de acuerdo a ciertas condiciones  
    ## si es cero, o muy cercano a 0 no aplicamos la transformacion  
    if np.abs(target) > 0.09:  
        ## transformacion horizontal  
        hImg, hTarget = horizontal_flip(img, target)
```

```

# guardar imagen y entry en el dataframe
filename = "dataset/" + str(id_offset + i) + ".bmp";
hImg = cv2.cvtColor(hImg, cv2.COLOR_BGR2RGB)
cv2.imwrite(filename, hImg)
newData.append([(id_offset + i),
                row['linear'],
                hTarget,
                filename])
# incrementa indice para siguiente entrada
i += 1

## transformacion aleatoria
choice = np.random.choice([0,1])
if choice == 1:
    rImg = datagen.flow(img.reshape((1,) + img.shape), y=None,
                        batch_size=1).next()
    rImg = rImg[0]
    # guardar imagen y entry en el dataframe
    filename = "dataset/" + str(id_offset + i) + ".bmp";
    rImg = cv2.cvtColor(rImg, cv2.COLOR_BGR2RGB)
    cv2.imwrite(filename, rImg)
    newData.append([(id_offset + i),
                    row['linear'],
                    target,
                    filename])
# incrementa indice para siguiente entrada
i += 1

### guardamos la imagen en el directorio y el nuevo target en el d
columns = ['id', 'linear', 'angular', 'imgpath']
newDataset = pd.DataFrame(newData, columns=columns)

newDataset = pd.DataFrame(newData, columns=columns)

dataset = dataset1.append(newDataset, ignore_index=True)
dataset.to_csv('augmented.csv')

```

B.5. Generación de datos

utils.py

```

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

from keras.models import model_from_json

import numpy as np
import pandas as pd
import bcolz
import threading

```

```
import cv2

import os
import sys
import glob
import shutil

from sklearn.model_selection import train_test_split

import models

def generator_from_df(df, batch_size, target_size, target_column='target',
                      features=None, process=True):
    print('generating_minibatch!')
    nbatches, n_skipped_per_epoch = divmod(df.shape[0], batch_size)
    #print nbatches
    count = 1
    epoch = 0
    # New epoch.
    while 1:
        df = df.sample(frac=1) # shuffle in every epoch
        epoch += 1
        i, j = 0, batch_size
        # Mini-batches within epoch.
        mini_batches_completed = 0
        for _ in range(nbatches):
            sub = df.iloc[i:j]
            try:
                if process == True:
                    X = np.array([(2 * (img_to_array(load_img(f,
                        target_size=target_size)) / 255.0 - 0.5)) for f in
                                  sub.imgpath])
                else:
                    X = np.array([(img_to_array(load_img(f, target_size=
                        target_size)))) for f in sub.imgpath])
            except IOError as err:
                count -= 1
                continue
            Y = sub[target_column].values
            # Simple model, one input, one output.
            mini_batches_completed += 1
            print ".",
            yield X, Y
        i = j
        j += batch_size
        count += 1
        if epoch > n_skipped_per_epoch:
            break
```

B.6. Nodo de Inferencia

neural_node.py

```
#!/usr/bin/env python
from __future__ import print_function
import roslib
import rospkg
roslib.load_manifest('robocar')
import sys
import csv
import numpy as np
import rospy
import cv2
import message_filters
from std_msgs.msg import String, Float32
from sensor_msgs.msg import Image, Range, CompressedImage, Joy
from geometry_msgs.msg import Twist, TwistStamped
from cv_bridge import CvBridge, CvBridgeError

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

from keras.models import model_from_json

import pandas as pd
import threading

import tensorflow as tf
import os
import sys
import glob
import shutil

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.callbacks import TensorBoard, ModelCheckpoint

from keras.utils import plot_model
import models
from models import custom_loss
import imutils

class AutoPilot:
    idx = 0

    dim = (224, 224)

    linear = 0
```

```

angular_joy = 0

def __init__(self, folder):
    rospack = rospkg.RosPack()
    pack_path = rospack.get_path('robocar')
    model_path = pack_path + '/scripts/simple2'

    # get ros params
    self.img_topic = rospy.get_param('img_topic', default='/camera/image/compressed')
    self.output_topic = rospy.get_param('output_topic', default='/neural_output')
    self.model_name = rospy.get_param('model', default=model_path)

    ## cargar la red neuronal en la memoria
    # load json and create model
    json_file = open(self.model_name + '.json', 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    self.model = model_from_json(loaded_model_json)

    # load weights into new model
    self.model.load_weights(self.model_name + "_best.h5")
    print("Loaded_model_from_disk")
    self.model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])
    self.model.summary()
    self.graph = tf.get_default_graph()

    print('creando_subs_y_pubs...')
    # image subscriber for the predictor
    self.image_sub = rospy.Subscriber(self.img_topic, CompressedImage,
                                      self.imCallback, queue_size=1)

    # float32 publisher for output
    self.output_pub = rospy.Publisher(self.output_topic, Float32,
                                      queue_size=1)

#this callback executes when the two subscribers sync
def imCallback(self, img):
    """ este calback lee la imagen de la camara, la preprocesa y obtiene
    una predicción para el comando de control del robot"""

    # lee la imagen y la preprocesa
    np_image = cv2.imdecode(np.fromstring(img.data, np.uint8), cv2.IMREAD_COLOR)
    np_image = cv2.resize(np_image, self.dim, interpolation = cv2.INTER_AREA)
    #print (np_image.shape)
    np_image = (2 * (np_image / 255.0 - 0.5))
    x = np_image.reshape((1,) + np_image.shape) # this is a Numpy array
    with shape (1, h,w, c)

```

```

#print (x.shape)
# obtiene la predicción de la red neuronal
angular = 0.0
with self.graph.as_default():
    angular = self.model.predict(x, batch_size=1, verbose=0)

angular = np.asscalar(angular.flatten())
#print ("predicción: ", angular)
# crea el mensaje para el control del carro y publica
# msg = Twist()

# msg.angular.z = angular
# self.twist_pub.publish(msg)

output_msg = Float32()
output_msg.data = angular
self.output_pub.publish(output_msg)

def main(args):
    rospy.init_node('neural_node', anonymous=True)
    stamper = AutoPilot(None)

    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("shutting down")
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main(sys.argv)

```

B.7. Nodo de detección de obstáculos

obstacle_node.py

```

#!/usr/bin/env python
from __future__ import print_function
import roslib
import rospkg
import rospy
roslib.load_manifest('robocar')

from sensor_msgs.msg import Range
from std_msgs.msg import Float32
import sys
import csv

class ObstacleDetector:
    idx = 0
    # model_name = '/home/pepe/catkin_ws/src/robocar/scripts/simple2'

```

```

dim = (224, 224)

linear = 0
angular_joy = 0
max_acc = 0.3

kp = 4.0

def __init__(self, folder):
    # get ros params
    rospy.loginfo("Obstacle_node_init")
    self.range_topic = rospy.get_param('range_topic', default='/laser')
    self.output_topic = rospy.get_param('output_topic', default='/
        obstacle_output')
    self.stop_distance = rospy.get_param('stop_distance', default
        =0.15)
    # input subscriber for the predictor
    self.range_sub = rospy.Subscriber(self.range_topic, Range, self.
        RangeCallback, queue_size=1)

    # float32 publisher for output
    self.output_pub = rospy.Publisher(self.output_topic, Float32,
        queue_size=1)

#this callback executes when the two subscribers sync
def RangeCallback(self, msg):
    """ este callback recibe el rango del sensor y calcula la salida
    correspondiente
    """
    error = msg.range - self.stop_distance
    error = error * 2 if error < 0 else error

    acceleration = self.kp * error
    acceleration = self.max_acc if acceleration > self.max_acc else
        acceleration
    acceleration = -self.max_acc * 2 if acceleration < -self.max_acc
        else acceleration
    acceleration = 0 if (acceleration < (self.stop_distance + 0.03))
        and (acceleration > (self.stop_distance - 0.01)) else
        acceleration

    out_msg = Float32()
    out_msg.data = acceleration
    self.output_pub.publish(out_msg)

def main(args):
    rospy.init_node('neural_node', anonymous=True)
    stamper = ObstacleDetector(None)
    rospy.spin()

```

```
if __name__ == '__main__':
    main(sys.argv)
```

B.8. Piloto Automático

```
pilot_node.cpp

#include <ros/ros.h>
#include <boost/thread.hpp>
#include <std_msgs/Float32.h>

#include <std_msgs/String.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/TwistStamped.h>
#include <sensor_msgs/Joy.h>

#include <cmath>

class Pilot
{

private:
    // data
    bool manual_;
    // topics from param server
    std::string neural_topic_;
    std::string obstacle_topic_;
    std::string joy_twist_topic_;

    std::string output_topic_;

    int freq_hz_;
    // messages
    geometry_msgs::Twist twist_msg_;           // simple twist message from
                                                // joystick
    geometry_msgs::Twist manual_twist_msg_;      // simple twist
                                                // message from joystick
    // geometry_msgs::TwistStamped stamped; // stamped twist for sync

    // subs and PUBs
    ros::Subscriber neural_sub_;
    ros::Subscriber obstacle_sub_;
    ros::Subscriber joy_twist_sub_;

    ros::Publisher twist_cmd_pub_;

    // subs callback
    // for neural steering
    void NeuralCallback(const std_msgs::Float32::ConstPtr &steering);

    // for obstacle node
```

```

void ObstacleCallback(const std_msgs::Float32::ConstPtr &acceleration)
{
};

// for joy tELoop
void JoyTwistCallback(const geometry_msgs::Twist::ConstPtr &joy_twist)
{
};

// for find object 2d
void ObjectCallback(const std_msgs::Float32::ConstPtr &steering);

public:
Pilot();

ros::Timer timer;

ros::NodeHandle nh_;
void TimerCallback(const ros::TimerEvent &event);
float GetInterval();
};

// constructor
Pilot::Pilot() : manual_(false)
{
    ROS_INFO("Iniciando_nodos");

    nh_.param<int>("freq_hz", freq_hz_, 20);
    if (freq_hz_ <= 0)
    {
        ROS_WARN("Invalid_frequency_value,_default:_20Hz");
        freq_hz_ = 20;
    }

// messages are received in QUEues
nh_.param<std::string>("neural_topic", neural_topic_, "/neural_output"
);
nh_.param<std::string>("obstacle_topic", obstacle_topic_, "/obstacle_output");
nh_.param<std::string>("joy_twist_topic", joy_twist_topic_, "/joy_cmd_vel");

nh_.param<std::string>("output_topic", output_topic_, "/cmd_vel");

// subscribe to nodes
ROS_INFO("Creando_subs_y_pubs");
neural_sub_ = nh_.subscribe<std_msgs::Float32>(neural_topic_, 1, &
Pilot::NeuralCallback, this);
obstacle_sub_ = nh_.subscribe<std_msgs::Float32>(obstacle_topic_, 1, &
Pilot::ObstacleCallback, this);
joy_twist_sub_ = nh_.subscribe<geometry_msgs::Twist>(joy_twist_topic_,
1, &Pilot::JoyTwistCallback, this);

// output topic por pilOT node

```

```
twist_cmd_pub_ = nh_.advertise<geometry_msgs::Twist>(output_topic_, 1)
;

}

float Pilot::GetInterval()
{
    return 1.0 / freq_hz_;
}

///////////////////////////////
// CALLBACKS ///////////////////
///////////////////////////////

void Pilot::NeuralCallback(const std_msgs::Float32::ConstPtr &steering)
{
    // extract the command
    // put in the message
    // double angular = steering;
    twist_msg_.angular.z = steering->data;
}

void Pilot::ObstacleCallback(const std_msgs::Float32::ConstPtr &
    acceleration)
{
    // double LineAr = acceleration;
    twist_msg_.linear.x = acceleration->data;
}

void Pilot::JoyTwistCallback(const geometry_msgs::Twist::ConstPtr &
    joy_twist)
{
    manual_ = true;
    manual_twist_msg_.linear.x = joy_twist->linear.x;
    manual_twist_msg_.angular.z = joy_twist->angular.z;
}

void Pilot::TimerCallback(const ros::TimerEvent &event)
{
    if(abs(manual_twist_msg_.linear.x) > 0.01 || abs(manual_twist_msg_.
        angular.z) > 0.01)
    {
        twist_cmd_pub_.publish(manual_twist_msg_);
        manual_ = false;
    }
    else
    {
        twist_cmd_pub_.publish(twist_msg_);
        manual_ = false;
    }
}
```

```

int main(int argc, char **argv)
{
    ros::init(argc, argv, "pilot_node");
    Pilot teleop_joy;

    teleop_joy.timer = teleop_joy.nh_.createTimer(ros::Duration(teleop_joy
        .GetInterval()),
                                                &Pilot::TimerCallback,
                                                &teleop_joy);
    ros::spin();
}

```

B.9. Control teleoperado con Joystick

```

joy_teleop.cpp

#include <ros/ros.h>
#include <boost/thread.hpp>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/TwistStamped.h>
#include <sensor_msgs/Joy.h>

class TeleopRobocar
{
    public:
    TeleopRobocar();

    ros::Timer timer;

    ros::NodeHandle nh_;
    void timerCallback(const ros::TimerEvent &event);
    float GetInterval();

    private:
    void joyCallback(const sensor_msgs::Joy::ConstPtr &joy);

    int linear_; // axis id
    int brake_;
    int angular_;

    double l_scale_;
    double a_scale_;
    double l_offset_;

    std::string output_topic;

    int freq_hz_;

    ros::Publisher vel_pub_;

```

```

ros::Publisher velStamped_pub_;
ros::Subscriber joy_sub_;

// for callbacks
geometry_msgs::Twist twist;           // simple twist message from
                                         joystick
geometry_msgs::TwistStamped stamped;   // stamped twist for sync
};

TeleopRobocar::TeleopRobocar()
{
    nh_.param<int>("freq_hz", freq_hz_, 20);
    if (freq_hz_ <= 0)
    {
        ROS_WARN("Invalid_frequency_value,_default:_20Hz");
        freq_hz_ = 20;
    }

    nh_.param<int>("axis_linear", linear_, 5);
    nh_.param<int>("axis_brake", brake_, 2);
    nh_.param<double>("scale_linear", l_scale_, -0.5);
    nh_.param<double>("offset_linear", l_offset_, 0.5);
    nh_.param<int>("axis_angular", angular_, 0);
    nh_.param<double>("scale_angular", a_scale_, -1.32);      // -0.34
    nh_.param<std::string>("output_topic", output_topic, "/joy_cmd_vel");

    vel_pub_ = nh_.advertise<geometry_msgs::Twist>(output_topic, 1);
    velStamped_pub_ = nh_.advertise<geometry_msgs::TwistStamped>(
        "/stamped_cmd_vel", 1);

    joy_sub_ = nh_.subscribe<sensor_msgs::Joy>("joy", 1, &TeleopRobocar::
        joyCallback, this);
}

float TeleopRobocar::GetInterval()
{
    return 1.0 / freq_hz_;
}

void TeleopRobocar::joyCallback(const sensor_msgs::Joy::ConstPtr &joy)
{
    twist.angular.z = joy->axes[angular_] * a_scale_;

    double acceleration = l_scale_ * joy->axes[linear_] + l_offset_;
    ROS_DEBUG("%f", acceleration);

    double reverse = l_scale_ * joy->axes[brake_] + l_offset_;
    ROS_DEBUG("%f", reverse);

    twist.linear.x = acceleration - reverse;
}

```

```

ROS_DEBUG("%f", twist.linear.x);

// vel_pub_.publish(twist);
}

void TeleopRobocar::timerCallback(const ros::TimerEvent &event)
{
    stamped.twist = twist;
    stamped.header.stamp = ros::Time::now();

    vel_pub_.publish(twist);
    velStamped_pub_.publish(stamped);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "joy_teleop");
    TeleopRobocar teleop_joy;

    teleop_joy.timer = teleop_joy.nh_.createTimer(ros::Duration(teleop_joy
        .GetInterval()),
                                                &TeleopRobocar::
                                                timerCallback,
                                                &teleop_joy);

    ros::spin();
}

```

B.10. Nodo del sensor de proximidad

laser_node.cpp

```

#include "VL53L0X.h"
#include <ros/ros.h>
#include <sensor_msgs/Range.h> // para el sensor
// libreria para el sensor laser
class LaserNode
{
    public:
        LaserNode();

        ros::Timer timer;

        ros::NodeHandle nh_;
        void timerCallback(const ros::TimerEvent &event);
        float GetInterval();

    private:
        // sensor
        VL53L0X sensor_;

    int freq_hz_;

```

```
double l_scale_;
double a_scale_;
double l_offset_;

ros::Publisher laser_pub_;
// msgs
sensor_msgs::Range laser_msg_;
};

LaserNode::LaserNode()
{
    nh_.param<int>("freq_hz", freq_hz_, 20);
    if (freq_hz_ <= 0)
    {
        ROS_WARN("Invalid_frequency_value,_default:_20Hz");
        freq_hz_ = 20;
    }

    // init publisher
    laser_pub_ = nh_.advertise<sensor_msgs::Range>("laser", 1);

    // laser
    sensor_.initialize();
    sensor_.setTimeout(200);

    // init some fixed data
    laser_msg_.radiation_type = 1;
    laser_msg_.header.frame_id = "laser";
    laser_msg_.field_of_view = 0.1;
    laser_msg_.min_range = 0.0;
    laser_msg_.max_range = 1.20;
}

float LaserNode::GetInterval()
{
    return 1.0 / freq_hz_;
}

void LaserNode::timerCallback(const ros::TimerEvent &event)
{
    uint16_t distance = sensor_.readRangeSingleMillimeters();

    if (!sensor_.timeoutOccurred())
    {
        // create message and publish
        laser_msg_.range = distance / 1000.0;
        laser_msg_.header.stamp = ros::Time::now();
        laser_pub_.publish(laser_msg_);
    }
    else
    {
```


Apéndice C

Hojas de datos