

**UNIVERSIDAD MAYOR DE SAN ANDRÉS
FACULTAD DE CIENCIAS PURAS Y NATURALES
CARRERA DE INFORMÁTICA**



TESIS DE GRADO

**MODELO DE CONDUCCIÓN AUTÓNOMA BASADO EN
APRENDIZAJE PROFUNDO Y ALGORITMOS DE VISIÓN
COMPUTACIONAL**

**PARA OPTAR AL TÍTULO DE LICENCIATURA EN INFORMÁTICA
MENCIÓN: CIENCIAS DE LA COMPUTACIÓN**

**POSTULANTE: RAFAEL VILLCA POGGIAN
TUTOR METODOLÓGICO: M.Sc. ALDO VALDEZ ALVARADO
ASESOR: Lic. BRIGIDA ALEXANDRA CARVAJAL BLANCO**

LA PAZ - BOLIVIA

2021



UNIVERSIDAD MAYOR DE SAN ANDRÉS
FACULTAD DE CIENCIAS PURAS Y NATURALES
CARRERA DE INFORMÁTICA



LA CARRERA DE INFORMÁTICA DE LA FACULTAD DE CIENCIAS PURAS Y NATURALES PERTENECIENTE A LA UNIVERSIDAD MAYOR DE SAN ANDRÉS AUTORIZA EL USO DE LA INFORMACIÓN CONTENIDA EN ESTE DOCUMENTO SI LOS PROPÓSITOS SON ESTRICAMENTE ACADÉMICOS.

LICENCIA DE USO

El usuario está autorizado a:

- a) visualizar el documento mediante el uso de un ordenador o dispositivo móvil.
- b) copiar, almacenar o imprimir si ha de ser de uso exclusivamente personal y privado.
- c) copiar textualmente parte(s) de su contenido mencionando la fuente y/o haciendo la referencia correspondiente respetando normas de redacción e investigación.

El usuario no puede publicar, distribuir o realizar emisión o exhibición alguna de este material, sin la autorización correspondiente.

TODOS LOS DERECHOS RESERVADOS. EL USO NO AUTORIZADO DE LOS CONTENIDOS PUBLICADOS EN ESTE SITIO DERIVARA EN EL INICIO DE ACCIONES LEGALES CONTEMPLADOS EN LA LEY DE DERECHOS DE AUTOR.

DEDICATORIA

Dedico la presente tesis a mi familia por todo el apoyo, amor y paciencia durante todos estos años de estudio.

AGRADECIMIENTOS

A mi familia por el apoyo durante todos los años cursando la carrera.

A mi tutor M.Sc. Aldo Valdez Alvarado por la guía durante el desarrollo de la presente tesis,
y a mi asesora Lic. Brigida Carvajal Blanco por los consejos, sugerencias y apoyo.

RESUMEN

La creciente popularidad de vehículos de distintas marcas con funcionalidades autónomas, ocasionó la creación de simuladores y conjuntos de datos para el entrenamiento de modelos de conducción mediante técnicas de visión artificial y aprendizaje profundo.

El problema a abordar en las soluciones que se desarrollan es crear implementaciones eficientes para realizar inferencias rápidas y reaccionar a las distintas situaciones ambientales mediante cámaras.

Se propone un modelo compuesto por redes neuronales y algoritmos de visión computacional que se complementen, y así obtener una conducción autónoma básica y eficiente computacionalmente.

Palabras clave: Aprendizaje profundo, Visión computacional, Redes neuronales, Carla simulator

ABSTRACT

The increasing popularity of different car manufacturers with autonomous features, made possible the creation of simulators and data sets to train self driving models using Computer Vision techniques and Deep Learning.

The main focus of these solutions is to develop efficient implementations to perform fast inference and react to different environmental situations through cameras.

In this work a model composed of deep neural networks and computer vision algorithms which work together is proposed, in order to obtain an efficient and basic self driving capability

Keywords: Deep learning, Computer vision, Neural networks, Carla simulator

Índice general

DEDICATORIA	III
AGRADECIMIENTOS	IV
RESUMEN	V
ABSTRACT	VI
ÍNDICE	XI
ÍNDICE DE FIGURAS	XIII
ÍNDICE DE TABLAS	XIV
1 MARCO REFERENCIAL	1
1.1 INTRODUCCIÓN	1
1.2 ANTECEDENTES	2
1.2.1 ESTADO DEL ARTE	2
1.2.1.1 DETECCIÓN DE OBJETOS	2
1.2.1.2 SEGMENTACIÓN SEMÁNTICA	5
1.2.1.3 SISTEMAS DE CONDUCCIÓN AUTÓNOMA	6
1.2.2 TRABAJOS SIMILARES	7
1.3 PLANTEAMIENTO DEL PROBLEMA	8
1.3.1 PROBLEMA CENTRAL	9
1.3.2 PROBLEMAS SECUNDARIOS	10
1.4 DEFINICIÓN DE OBJETIVOS	10
1.4.1 OBJETIVO GENERAL	10
1.4.2 OBJETIVOS ESPECÍFICOS	10
1.5 HIPÓTESIS	11
1.5.1 OPERACIONALIZACIÓN DE VARIABLES	11
1.6 JUSTIFICACIÓN	11
1.6.1 JUSTIFICACIÓN ECONÓMICA	11
1.6.2 JUSTIFICACIÓN SOCIAL	12
1.6.3 JUSTIFICACIÓN CIENTÍFICA	12

1.7 ALCANCES Y LÍMITES	12
1.7.1 ALCANCES	12
1.7.2 LÍMITES	12
1.8 APORTES	13
1.8.1 PRÁCTICO	13
1.8.2 TEÓRICO	13
1.9 METODOLOGÍA	13
2 MARCO TEÓRICO	15
2.1 SISTEMAS DE CONDUCCIÓN AUTÓNOMA	15
2.1.1 TAREAS DE LA CONDUCCIÓN AUTÓNOMA	15
2.1.2 ARQUITECTURA DE LA CONDUCCIÓN AUTÓNOMA	15
2.1.3 CARLA	16
2.1.4 NIVELES DE CONDUCCIÓN AUTÓNOMA	16
2.2 VISIÓN COMPUTACIONAL	17
2.2.1 PROCESAMIENTO DE IMÁGENES	17
2.2.1.1 REPRESENTACIÓN DE IMÁGENES EN UNA COMPUTADORA ..	17
2.2.1.2 CONVOLUCIÓN	18
2.2.1.3 DETECCIÓN DE CONTORNOS	19
2.2.2 TRANSFORMACIONES MORFOLÓGICAS	19
2.2.2.1 DILATACIÓN	19
2.2.2.2 EROSIÓN	20
2.2.2.3 APERTURA	21
2.2.3 K-MEANS	21
2.2.4 FLOOD FILL	23
2.3 APRENDIZAJE AUTOMÁTICO	23
2.3.1 APRENDIZAJE SUPERVISADO	23
2.3.1.1 APRENDIZAJE CORRECTO PROBABLEMENTE APROXIMADO ..	24
2.3.2 REGRESIÓN	25

2.3.2.1	REGRESIÓN LINEAL	25
2.3.2.2	REGRESIÓN GENERALIZADA	27
2.3.2.3	MEDIAS MÓVILES EXPONENCIALES	27
2.3.3	CLASIFICACIÓN	27
2.3.3.1	REGRESIÓN SOFTMAX	27
2.3.4	DESCENSO DEL GRADIENTE ESTOCÁSTICO	29
2.3.5	ADAM	30
2.4	APRENDIZAJE PROFUNDO	31
2.4.1	PERCEPTRÓN MULTICAPA	31
2.4.1.1	FUNCIONES DE ACTIVACIÓN	33
2.4.1.2	RETROPROPAGACIÓN DE LOS ERRORES	33
2.4.2	REDES NEURONALES CONVOLUCIONALES	34
2.4.2.1	STRIDES	34
2.4.2.2	POOLING	35
2.4.2.3	MOBILENET V2	36
2.4.2.4	FAST DEPTH	37
2.4.3	APRENDIZAJE DE REPRESENTACIONES PROFUNDAS	38
2.5	MÉTRICAS DE ERROR	40
2.5.1	ERROR CUADRÁTICO MEDIO	40
2.5.2	ERROR ABSOLUTO MEDIO	40
2.5.3	PRECISIÓN Y EXHAUSTIVIDAD	41
2.5.4	ÍNDICE JACCARD	42
2.6	COMPILEACIÓN EN TIEMPO DE EJECUCIÓN	42
3	MARCO APLICATIVO	43
3.1	COMPRENSIÓN DEL PROYECTO	43
3.2	COMPRENSIÓN DE LOS DATOS	44
3.2.1	ACELERADOR Y DIRECCIÓN	44
3.2.2	PROFUNDIDAD	45

3.2.3	SEGMENTACIÓN SEMÁNTICA	45
3.3	PREPARACIÓN DE DATOS	47
3.3.1	UNIÓN DE DATAFRAMES	47
3.3.2	ETIQUETADO DE INTERSECCIONES	47
3.3.3	CONCATENACIÓN DE ATRIBUTOS	49
3.4	MODELADO	49
3.4.1	RED DE CONDUCCIÓN	50
3.4.2	RED DE PROFUNDIDAD	51
3.4.3	RED DE SEGMENTACIÓN SEMÁNTICA	52
3.4.4	SUAVIZADO DE DIRECCIÓN	52
3.4.5	CAJA DELIMITADORA	53
3.4.6	CUANTIFICACIÓN DIGITAL DEL COLOR	54
3.4.7	MODELO PARA LA CONDUCCIÓN AUTÓNOMA	55
3.5	EVALUACIÓN	56
3.5.1	DRIVENET	56
3.5.2	DEPTHNET	57
3.5.3	SEMSEGNET	58
3.5.4	CENTROIDES K-MEANS	58
3.6	DESPLIEGUE	59
4	RESULTADOS Y ANÁLISIS	61
4.1	RENDIMIENTO DE LOS MÓDULOS	61
4.1.1	ACELERACIÓN Y GIRO	61
4.1.2	ESTIMACIÓN DE PROFUNDIDAD	62
4.1.3	SEGMENTACIÓN SEMÁNTICA	63
4.1.3.1	MATRIZ DE CONFUSIÓN	64
4.1.3.2	PRECISIÓN Y EXHAUSTIVIDAD	65
4.1.3.3	CAJAS DELIMITADORAS	67
4.1.3.4	ÍNDICE JACCARD	67

4.1.4	DETECCIÓN Y CLASIFICACIÓN DE SEMÁFOROS	68
4.2	REPRESENTACIONES APRENDIDAS	69
4.2.1	DISTANCIAS EN LA REGIÓN DE INTERÉS	69
4.2.2	VISUALIZACIÓN DE ZONAS DE INTERÉS	71
4.3	PRUEBAS EN SIMULADOR	71
4.3.1	CONDUCCIÓN	71
4.3.2	FALLOS	73
4.4	PRUEBA DE LA HIPÓTESIS	74
5	CONCLUSIONES Y RECOMENDACIONES	76
5.1	CONCLUSIONES	76
5.2	RECOMENDACIONES	78
6	BIBLIOGRAFÍA	79
ANEXOS		83

Índice de figuras

1	Contornos de objetos	2
2	Características HAAR	3
3	Vectores direccionales del HOG	3
4	Flujo predicción YOLO	4
5	Partición y recuadros de detección SSD	4
6	Segmentación de carretera por MCRF	5
7	Segmentación de una célula del dataset PhC-U373	6
8	Segmentación semántica de distintas clases	6
9	DAVE	7
10	Diagrama CRISP-DM	14
11	Intensidad de los canales de color RGB	18
12	Convolución del filtro K con la imagen I	19
13	Dilatación sobre un conjunto de letras con discontinuidades	20
14	Erosión de una letra	20
15	Apertura aplicada a una huella digital	21
16	Ajuste iterativo de k-means	22
17	Ajuste de regresión lineal	26
18	Ajuste de regresión logística	28
19	Pasos de los parámetros W en cada iteración camino al mínimo error	29
20	Red neuronal con una capa oculta	32
21	Red neuronal convolucional para la clasificación de dígitos manuscritos	34
22	Convolución con $s = 2$	35
23	Pooling con $s = 1$	35
24	Cuello de botella residual	36
25	Arquitectura FastDepth	38
26	Extracción de las características profundas aprendidas	39
27	Algoritmo ScoreCam	39

28	Arquitectura del proyecto	43
29	Diagrama de secuencia de extracción de datos	44
30	Etiquetas de profundidad imagen	45
31	Segmentación semántica con códigos de color	46
32	Etapas del etiquetado de intersecciones	48
33	Entrenamiento de tres redes neuronales	50
34	Arquitectura DriveNet basada en Mobilenet V2	50
35	Arquitectura FastDepth	51
36	SemsegNet, FastDepth para clasificación 2D	52
37	Aplicación de medias exponenciales móviles	52
38	Cuantificación digital del color	54
39	Delimitado del área de interés para la detección de objetos	55
40	Modelo de conducción autónoma	56
41	Errores de entrenamiento y validación de la DriveNet	57
42	Errores de entrenamiento y validación de la DepthNet	57
43	Errores de entrenamiento y validación de la SemsegNet	58
44	Espacios de color y centroides para dos semáforos distintos	59
45	Predicción Media Móvil Exponencial	62
46	Predicción vs valor esperado de profundidad	63
47	Matriz de confusión de segmentación semántica	64
48	Predicción vs valor esperado de segmentación	66
49	Cajas delimitadoras	67
50	Cuantización y Clasificación del Color	69
51	Distancia a de objetos	70
52	Regiones de Interés en la Inferencia de Dirección	71
53	Comparación de Dirección Real y Estimada	72
54	Charco erroneamente Clasificado	73
55	Colisión con Poste	74

Índice de tablas

1	Niveles de Conducción Autónoma	16
2	MobileNet V2	37
4	Correspondencia numérica de clases de segmentación	46
5	Errores en el conjunto de validación de los parámetros seleccionados	61
6	Métricas de error de clasificación	65
7	Índice de Jaccard	68

CAPÍTULO 1. MARCO REFERENCIAL

1.1. INTRODUCCIÓN

Los vehículos autónomos son algo común en el imaginario colectivo, esto debido a sus apariciones en la ciencia ficción y a las noticias de que empresas como Waymo, Uber y muchas más están trabajando para lograr la tan deseada autonomía completa.

Los modelos de Machine Learning usados en la actualidad fueron desarrollados muchas décadas atrás, estos requerían pre procesar demasiado la información de conjuntos de datos pequeños, sin embargo gracias a los avances en la transmisión de la información y al acceso a datos masivos que antes no eran posibles de almacenar, estos requisitos fueron decreciendo mientras más datos de entrenamiento se tenían disponibles, es así como se da un resurgimiento en el interés por los modelos probabilísticos, superando estos a los basados en reglas, utilizados comúnmente hasta los años 90. (Goodfellow, Bengio & Courville, 2016)

En el ámbito de la visión computacional, los primeros modelos eran capaces de detectar objetos simples en imágenes muy pequeñas (Rumelhart, Hinton & Williams, 1986). Por contraparte los modelos modernos basados en Redes Neuronales Convolucionales, son capaces de detectar por lo menos 1000 categorías distintas con una exactitud muy superior a los métodos clásicos siempre y cuando la cantidad de datos sea suficiente. (Krizhevsky, Sutskever & Hinton, 2012)

Aprovechando los avances de la visión computacional y la mejora de mejores sensores, los sistemas de conducción autónoma refinaron su capacidad de percepción, etapa clave para la toma de decisiones, mediante el uso de las Redes Neuronales para la detección, segmentación y clasificación de objetos en las carreteras, con los más avanzados como el autopilot de Tesla detectando situaciones adversas basado en acciones de otros vehículos en el camino a tiempo real. (Karpathy, 2020)

En el presente trabajo se propone un modelo compuesto por redes neuronales convolucionales y algoritmos de visión artificial para resolver las tareas básicas requeridas para la percepción y control de un vehículo autónomo.

1.2. ANTECEDENTES

A continuación se revisa la evolución de los métodos de visión computacional aplicados en la conducción autónoma y los sistemas resultantes aplicados a esta tarea.

1.2.1. ESTADO DEL ARTE

Se revisa el desarrollo de las tecnologías a utilizar desde su creación hasta el momento.

1.2.1.1. DETECCIÓN DE OBJETOS

1963 PERCEPCIÓN DE SÓLIDOS TRIDIMENSIONALES Lawrence Roberts desarrollo un algoritmo para detectar figuras sólidas simples en imágenes, extraer su estructura característica en forma de líneas para poder aplicarles transformaciones de perspectiva en un espacio 3D. (Roberts, 1963)

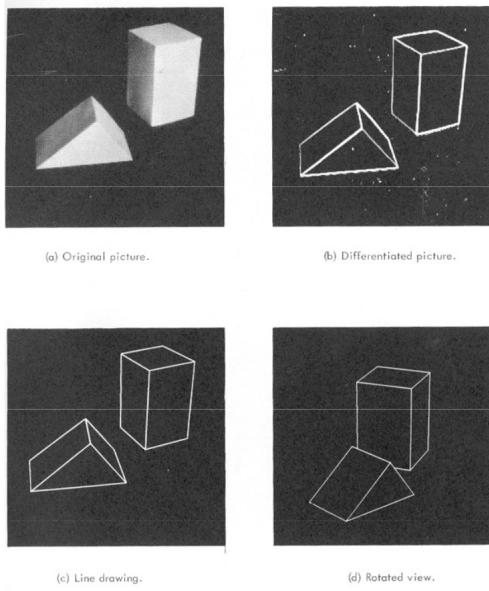


Figura 1: Detección de contornos de objetos
Fuente: (Roberts, 1963)

2001 HAAR FEATURES Paul Viola y Michael Jones desarrollaron el algoritmo Viola-Jones, el cual significó un cambio radical en los métodos de detección de objetos, extrayendo características primitivas con clasificadores débiles uno detrás del otro, y aplicando la técnica

de la imagen integral, fueron capaces de detectar distintos tipos de objetos en imágenes de manera óptima, entrenando el método sobre un conjunto pequeño de imágenes. Este método se usa hasta hoy en día en cámaras celulares para la detección de rostros. (Viola & Jones, 2001)



Figura 2: Características Haar buscando patrones en la imagen
Fuente: (Viola & Jones, 2001)

2005 HISTOGRAMAS DE GRADIENTES ORIENTADOS En 1986 Robert K. McConnel describe un descriptor de características que sería conocido como HOG o histograma de gradientes orientados, en 2005 sería aplicado a la detección de personas, rivalizando con el método de características Haar. (Dalal & Triggs, 2005)

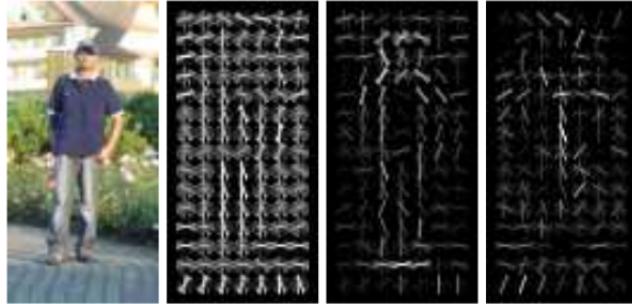


Figura 3: Vectores direccionales del HOG
Fuente: (Dalal & Triggs, 2005)

2016 YOLO La búsqueda de métodos que aceleren el proceso de ventanas deslizantes para localizar objetos, llevó a buscar alternativas que realicen todo el proceso en una sola pasada del método, así nace You Only Look Once, una arquitectura de red neuronal convolucional que barre la imagen en grillas definidas buscando la existencia del centro de un objeto y realizando una supresión de no máximo para eliminar ocurrencias repetidas, siendo uno de los métodos más efectivos en la detección de objetos. (Redmon, Divvala, Girshick & Farhadi, 2016)

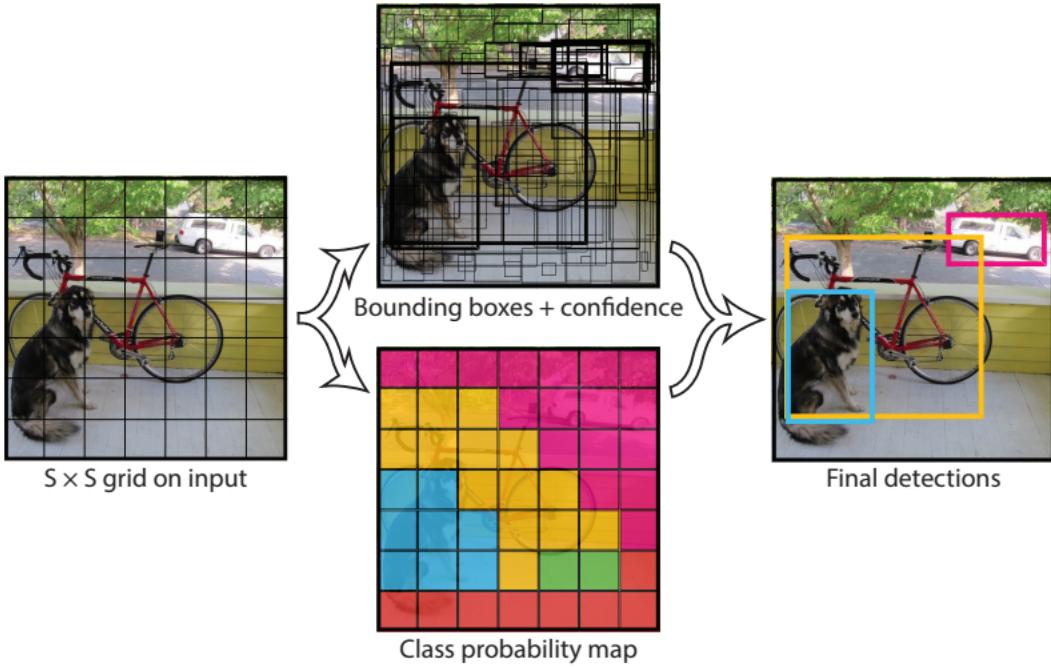


Figura 4: Flujo predicción YOLO
Fuente: (Redmon, Divvala, Girshick & Farhadi, 2016)

2016 SSD Con una arquitectura distinta aplicando la misma idea de la partición de la imagen en una grilla, se propone el método Single Shot MultiBox Detector, el cual es menos preciso en la detección de objetos que su contraparte YOLO, pero más rápido y capaz de detectar objetos más pequeños en la imagen. (Liu et al., 2016)

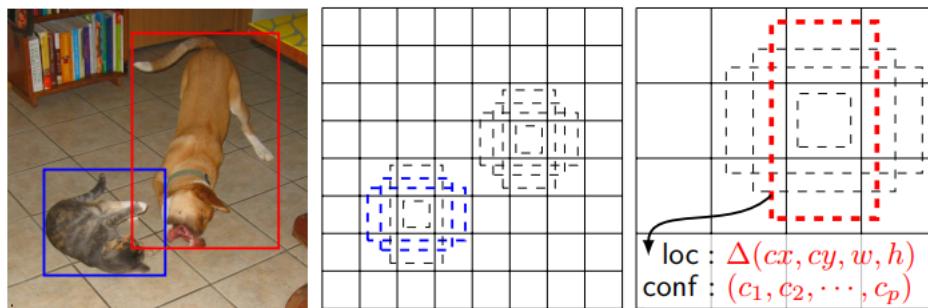


Figura 5: Partición y recuadros de detección SSD
Fuente: (Liu et al., 2016)

2020 RETINANET Buscando solucionar el problema de la detección de objetos pequeños y en gran cantidad en un entorno denso, mediante modificaciones en la función de costo del

clasificador, nace la llamada Retinanet (Lin, Goyal, Girshick, He & Dollár, [2017] 2020)

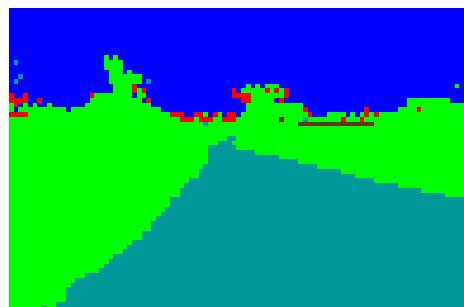
1.2.1.2. SEGMENTACIÓN SEMÁNTICA

SEGMENTACIÓN POR COLOR Y ESCALA DE GRISES Entre los primeros métodos de segmentación semántica están los de segmentación por burbujas de color en espacios de color alternativos al RGB, de forma que toda un área de un color específico con alguna variación permitida era enmascarada como un objeto en la imagen.

2004 CAMPOS ALEATORIOS CONDICIONALES MULTIESCALA La idea de los campos aleatorios condicionales, (MCRF abreviado en inglés) nació para etiquetar y segmentar secuencias de textos, pero fue modificada para lograr segmentar elementos de una imagen, modelando la relación espacial entre los píxeles con una probabilidad condicional que pertenezca a alguna clase dada la clase de los píxeles vecinos, aplicando un campo estocástico bidimensional (generalización de un proceso estocástico). (Xuming He, Zemel & Carreira-Perpinan, 2004)



(a) imagen original



(b) segmentación con MCRF

Figura 6: Segmentación de carretera por MCRF
Fuente: (Xuming He, Zemel & Carreira-Perpinan, 2004)

2015 U-NET Inicialmente pensada para aplicaciones médicas, esta arquitectura de Red Neuronal Convolutacional probó ser efectiva en muchas tareas se segmentación, extendiendo la idea de un auto encoder con saltos entre capas, la topología de las conexiones de la red forman una U, de donde proviene su nombre. (Ronneberger, Fischer & Brox, 2015)

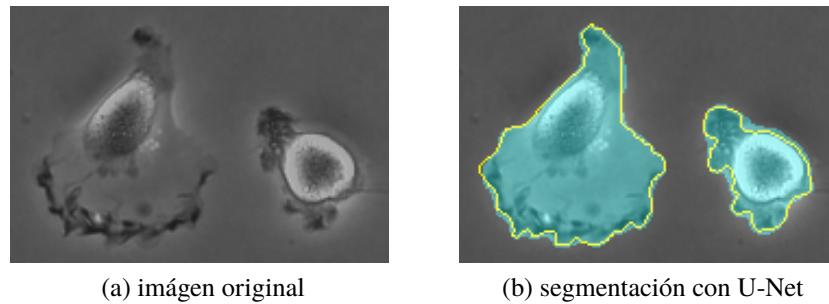


Figura 7: Segmentación de una célula del dataset PhC-U373
Fuente: (Ronneberger, Fischer & Brox, 2015)

2017 MASK-RCNN Siguiendo la propuesta de una red de dos etapas, dividiendo la tarea en dos más sencillas, esta arquitectura primero propone regiones de interés en base a las cuales hace dos tipos de inferencia, una regresión de los puntos delimitando un rectángulo que encierra a cada objeto, y una máscara binaria para cada clase a segmentar. (He, Gkioxari, Dollár & Girshick, 2017) Junto a la U-Net son los dos métodos más usados en la actualidad, incluido en el dominio de vehículos autónomos.



Figura 8: Segmentación semántica de distintas clases
Fuente: (He, Gkioxari, Dollár & Girshick, 2017)

1.2.1.3. SISTEMAS DE CONDUCCIÓN AUTÓNOMA

2004 DAVE Uno de los primeros intentos de programar un sistema de conducción autónoma fue propuesto por un equipo en el que se encontraba Yann LeCun, uno de los precursores en el uso de las redes convolucionales. En este proyecto se usaban dos cámaras, se entrenó una red en imágenes capturadas en un vehículo a control remoto, y se realizaba una predicción del

nivel de aceleración y ángulo de rotación para cada fotograma capturado. (Lecun, Cosatto, Ben, Muller & Flepp, 2004)

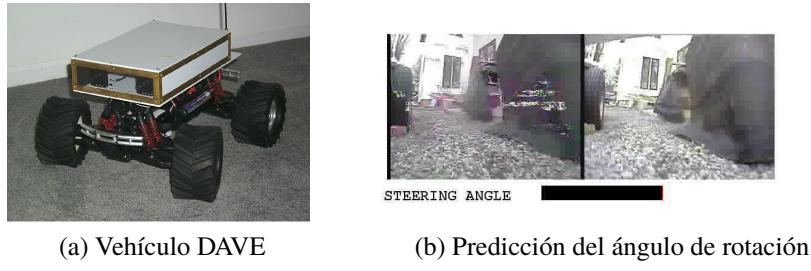


Figura 9: Fuente: Reporte (Lecun, Cosatto, Ben, Muller & Flepp, 2004)

2004 DARPA GRAND CHALLENGE La agencia de investigación en proyectos de defensa avanzados o DARPA abreviado en inglés, organizó una competencia en el desierto Mojave, con una pista preparada en un terreno arenoso. Los vehículos concursantes podían usar distintos tipos de sensores como cámaras, radar, lidar y GPS. (Hooper, 2004)

2010 GOOGLE Despu s de muchos tests con asistencia humana y sin llamar la atenci n, se hacen p blicos los experimentos de Google en la v a publica con veh culos equipados con sensores y un sistema de conducci n aut noma. Este sistema ir a evolucionando hasta la actualidad, cuando despu s de un cambio de nombre del proyecto, ahora bajo la marca Waymo, Ya realizan trayectos totalmente aut nomos. (Markoff, 2010)

2015 TESLA Tesla inició su programa de conducción autónoma oficialmente el año 2015, con un actualización de software para los Tesla Model S equipados con el Hardware necesario para el funcionamiento de su Autopilot. (Nelson, 2015) Posteriormente después de varias actualizaciones de software, y una de hardware, los distintos vehículos sucesivos de Tesla recopilarían los datos necesarios de los trayectos realizados por los usuarios para mejorar la reacción de este sistema (Karpathy, 2020)

1.2.2. TRABAJOS SIMILARES

Título: Aprendizaje fin a fin para la conducción autónoma de vehículos domésticos usando visión artificial y redes neuronales convolucionales.

Autor: Jose Eduardo Laruta Espejo

Año: 2018

Institución: Universidad Mayor de San Andrés

Esta tesis plantea implementar un sistema fin a fin embedido, equipado con una raspberry pi, que recibía fotogramas de una pista para carreras y debía predecir con una sola red neuronal, el ángulo de rotación de los servomotores y la aceleración para cada fotograma.

Título: Deep Vision Pipeline for Self-Driving Cars Based on Machine Learning Methods.

Autor: Mohammed Nabeel Ahmed

Año: 2017

Institución: Ryerson University

Este trabajo propone un flujo de datos para procesar la información en base a cámaras, mediante detección de las líneas de carreteras, clasificación de signos de tráfico y vehículos para realizar control del vehículo usando la predicción de estos.

Título: Modelamiento Semántico del Entorno para la Conducción Autónoma de un Vehículo Terrestre

Autor: Fernando Javier Bernuy Bahamondez

Año: 2017

Institución: Universidad de Chile

La tesis doctoral, propone el uso de segmentación semántica para apoyar técnicas clásicas de visión computacional para la localización espacial de un vehículo en un entorno urbano, logrando ubicarse en posiciones del terreno, estimar su orientación y lugar del recorrido planeado.

1.3. PLANTEAMIENTO DEL PROBLEMA

Si bien la idea de construir vehículos autónomos no es nueva, es en esta época donde finalmente se está logrando salir de etapas prototipo, a incluir estos sistemas en vehículos de venta masiva, por parte de distintos fabricantes, de entre los cuales uno de los más reconocidos es

Tesla Motors, que a diferencia de las alternativas propuestas por Google y Uber, proponen un sistema sin el uso de sensores costosos y poco estéticos para un vehículo comercial como un LIDAR (Granath, 2020), basandose sólamente en sensores de proximidad, radar, GPS y múltiples cámaras, diseñando y entrenando sus modelos de aprendizaje profundo como arquitecturas de redes neuronales y aprendizaje reforzado, con los datos masivos recopilados de los vehículos vendidos que circulan día a día (Karpathy, 2020), buscando que sus vehículos sean capaces de conducirse en todo tipo de situaciones climáticas y de iluminación, ya que los modelos de aprendizaje profundo superan a los algoritmos de visión computacional en estas situaciones (Goodfellow et al., 2016), si se tiene la gran cantidad de datos para entrenarlos correctamente (Krizhevsky et al., 2012).

Un coche que se conduzca solo, respetando todas las leyes de tránsito, con tiempos de reacción instantáneos y sin problemas humanos como el cansancio, en un ecosistema compuesto en su totalidad por vehículos autónomos, reduciría el riesgo de accidentes de tránsito. Existe un historial de accidentes en los que están envueltos autos autónomos, de estos, la mayoría fueron causados por otros conductores humanos que al no respetar las reglas de tránsito impactaron de alguna manera con el vehículo autónomo («List of self-driving car fatalities», 2020). Por esta razón y debido a la forma de vida acelerada en la sociedad actual, automatizaciones que ahorren tiempo en alguna actividad, son productos atractivos en los que se invierte para su investigación, existiendo en el caso de los vehículos autónomos datasets como el Open Waymo dataset, el cual contiene 2TB de datos para que los concursantes implementen sistemas de conducción autónoma y compitan en distintos retos (Waymo, 2019).

A parte de aplicaciones en la movilidad doméstica, existen áreas de investigación en vehículos autónomos para competencias tanto a escala como con vehículos reales, y también se busca aplicar esta tecnología en situaciones de trabajo con maquinaria pesada que conlleva cierto riesgo en la operación de esta.

1.3.1. PROBLEMA CENTRAL

¿De qué manera conseguir un nivel básico de conducción autónoma en vías de doble sentido?

1.3.2. PROBLEMAS SECUNDARIOS

- Los conjuntos de datos disponibles contienen información de distintos sensores y cámaras sin procesamiento previo, debido a esto los datos pesan demasiado y no se tiene información organizada y etiquetada.
- Para lograr una buena autonomía son necesarios componentes de hardware avanzados como un LIDAR que brindan una mejor percepción del espacio, esto ocasiona que el precio de los sistemas de conducción autónoma sea elevado y poco accesible, además de complejizar el problema de implementación.
- Los modelos de aprendizaje profundo generalizan bien pero un modelo de varias capas es más complejo de entrenar, hace necesario recopilar muchos datos y requiere mucho poder de cómputo.
- Al entrenar modelos simples de aprendizaje profundo, las representaciones profundas pueden ser subóptimas, esto ocasiona problemas con cambios en el contenido, perspectiva e iluminación.
- Los algoritmos de visión computacional requieren poca capacidad de cómputo, pero sólo funcionan en condiciones ideales de imágenes para los que se programan, esto da lugar a problemas de generalización de las predicciones.
- Acceder a un vehículo autónomo para realizar pruebas de rendimiento de un modelo es una tarea complicada de lograr, esto hace costoso el desarrollo de la tecnología y la mejora de los modelos disponibles para esta tarea.

1.4. DEFINICIÓN DE OBJETIVOS

1.4.1. OBJETIVO GENERAL

Plantear un modelo para la conducción autónoma que logre una autonomía básica en vías de doble sentido con separación física.

1.4.2. OBJETIVOS ESPECÍFICOS

- Diseñar un componente de aumentación y preprocesamiento de datos para extraer y crear un dataset con el fin de resolver la tarea.

- Reducir la complejidad de implementación del modelo mediante el uso de solamente una cámara.
- Modificar y entrenar redes neuronales con una alta exactitud en las predicciones utilizando menos requisitos de cómputo.
- Analizar las predicciones de los modelos entrenados para comprobar si las representaciones aprendidas son invariantes a los cambios de perspectiva, iluminación y objetos en la imagen.
- Combinar las salidas de algoritmos de visión computacional y modelos de aprendizaje profundo para mejorar la generalización de predicciones.
- Probar el rendimiento del modelo en una simulación, analizando casos de fallas y qué situaciones puede manejar correctamente.

1.5. HIPÓTESIS

El modelo de conducción autónoma mediante el uso de aprendizaje profundo y algoritmos de visión computacional alcanza una autonomía de nivel 2 en vías de doble sentido con separación física.

1.5.1. OPERACIONALIZACIÓN DE VARIABLES

Variable	Tipo
Aprendizaje profundo y algoritmos de visión computacional	Independiente
Modelo de conducción autónoma	Interviniente

Fuente: Elaboración propia

1.6. JUSTIFICACIÓN

1.6.1. JUSTIFICACIÓN ECONÓMICA

La implementación reduce los costos al implementar un sistema de conducción autónoma ya que procura utilizar solamente una cámara para la percepción sin el apoyo de sensores extra, lo que reduce la complejidad de todo el flujo de modelos y datos, centrando todos los esfuerzos en la implementación y entrenamiento de los algoritmos y modelos necesarios para

esta tarea.

1.6.2. JUSTIFICACIÓN SOCIAL

El presente trabajo propone un paso para hacer más accesible la conducción autónoma en vehículos convencionales al ser más sencillo de adaptar y estar disponible el código en github.

1.6.3. JUSTIFICACIÓN CIENTÍFICA

La investigación emplea técnicas de preprocesamiento de datos, procesamiento de imágenes mediante algoritmos de visión computacional y arquitecturas de redes neuronales para predecir las acciones necesarias en la conducción autónoma a partir de los datos obtenidos por la cámara.

1.7. ALCANCES Y LÍMITES

1.7.1. ALCANCES

- Se define una red neuronal para predecir fin a fin la aceleración y dirección del vehículo dado un fotograma de la cámara.
- El modelo contiene un módulo de estimación de distancias para evitar colisiones.
- Se introduce un módulo de segmentación de objetos en la imagen.
- Se implementa un algoritmo para la detección de objetos en base a las predicciones de las redes neuronales
- El modelo contempla un módulo de clasificación de color de semáforos.

1.7.2. LÍMITES

- El modelo no empleará módulos que tengan como entrada otros sensores aparte de una cámara RGB.
- Las predicciones del modelo no serán capaces de permitir cambios de carriles o adelantamientos.
- Las calles sobre las que puede conducirse tienen un carril en cada dirección y están asfaltadas.
- El modelo no es capaz de predecir correctamente en situaciones de lluvia extrema.

- La generalización del modelo está limitada por la complejidad de las redes neuronales a favor de velocidad de cómputo.

1.8. APORTES

1.8.1. PRÁCTICO

En el presente trabajo se presentarán algoritmos de visión computacional los cuales se combinarán con las predicciones de redes neuronales convolucionales en un flujo que inicia por preprocessar la información para luego entrenar los componentes sobre estos datos, permitiendo así reducir la complejidad de la implementación y lograr predicciones para una conducción autónoma básica, probando su rendimiento en una simulación.

1.8.2. TEÓRICO

La presente tesis de investigación, propone un modelo con el fin de dar un paso más en dirección a la conducción autónoma para futuras investigaciones, mediante el uso de aprendizaje profundo con redes neuronales y algoritmos de visión computacional.

1.9. METODOLOGÍA

La presente investigación es de tipo experimental porque se adapta el modelo de conducción autónoma a las vías de doble sentido, la cual es la realidad que se está simulando.

La metodología a usar será “Cross-industry standard process for data mining” (CRISP-DM), la cual está diseñada para proyectos de minería de datos y modelos de analíticas. Esta metodología es empleada principalmente por IBM para sus proyectos de analíticas y está integrada en su producto SPSS (Chapman et al., 2000).

Esta metodología se compone por 6 fases:

- **Comprensión del Negocio:** es la fase de propuestas de objetivos y requisitos del proyecto para convertirlo en una tarea de minería de datos o analíticas.
- **Comprensión de los Datos:** inicia con la recolección de datos y se debe familiarizar con estos, para analizar su calidad y descubrir patrones de utilidad.
- **Preparación de los Datos:** compuesto por todas las tareas de preparación con el fin de

construir el dataset final. Estas tareas se realizaran múltiples veces sin orden específico dependiendo de las necesidades del dataset, y pueden ser limpieza de datos, transformaciones y construcciones de nuevos atributos.

- **Modelado:** se modelan los datos mediante modelos de aprendizaje automático y técnicas de extracción de la información e inferencia.
- **Evaluación:** se evalúan las capacidades de predicción y ajuste de los modelos sobre los datos, con el fin de elegir el mejor modelo o la mejor combinación de estos.
- **Despliegue:** se aplican los modelos en la vida real para las situaciones que fueron diseñados.

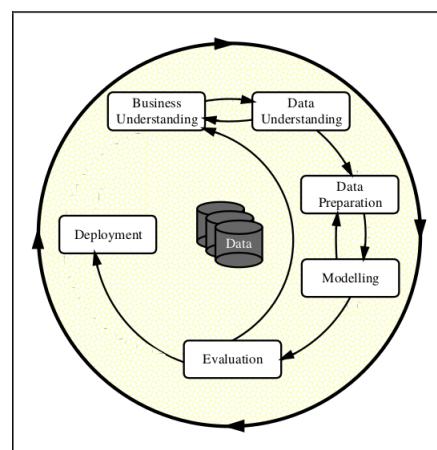


Figura 10: Diagrama CRISP-DM
Fuente: (Wirth, 2000)

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

2.1.1. TAREAS DE LA CONDUCCIÓN AUTÓNOMA

Para lograr la conducción autónoma, se deben dividir tareas modulares, esto con el fin de poder realizar pruebas de cada componente y en caso de fallos, poder detectarlos aisladamente y que no afecten a los demás componentes.

Las tareas son:

- **Control lateral:** Control de la dirección o volante del vehículo.
- **Control longitudinal:** Control de la aceleración y freno.
- **Detección y respuesta de eventos y objetos:** (OEDR por sus siglas en inglés) detección de objetos importantes en la carretera como carriles y vehículos, también objetos fuera de la carretera como peatones o señales de tránsito. Como respuesta debe reaccionar a distintas situaciones peligrosas tanto deteniendo el coche o sacándolo de esa situación.
- **Planeamiento:** A corto plazo son las acciones inmediatas que debe tomar y modificar el camino ante adversidades, y a largo plazo es el mejor camino encontrado desde el origen al destino definido por el usuario.

estas están definidas en el documento de recomendaciones para vehículos autónomos de la Sociedad Internacional de Ingenieros de Automoción (SAE por sus siglas en inglés). (SAE, 2018)

2.1.2. ARQUITECTURA DE LA CONDUCCIÓN AUTÓNOMA

La arquitectura del software para la conducción autónoma se basa en una fase de adquisición de datos para entrenar los modelos predictivos y ajustar los algoritmos, y a partir de ahí una fase de inferencia, la cual tiene cuatro componentes principales:

- **Percepción del ambiente:** Aquí se encuentran todos los modelos y algoritmos que nos

permiten detectar objetos de interés a través de los sensores disponibles.

- **Mapeo del ambiente:** Una vez extraídos los objetos de interés, se procede a ubicarlos en posiciones relativas al vehículo.
- **Planificación de movimiento:** Se procede a armar un plan de acción para el estado del ambiente que se percibe en ese instante.
- **Control:** Una vez se realiza el plan de acción, se ejecuta enviando la información a los controladores de aceleración, freno y dirección.

2.1.3. CARLA

CARLA es un simulador de código libre, desarrollado sobre Unreal Engine 4, para la investigación de la conducción autónoma, al ser pensado para esta tarea, provee una interfaz de comunicación a través de código en C++ y Python mediante su librería. Cuenta con distintos tipos de climas desde el día despejado hasta tardes lluviosas además de distintos modelos de vehículos y pedestres para lograr una simulación realista.

La funcionalidad más importante de carla es poder extraer la imagen de cámaras y sensores virtuales para generar datos de entrenamiento, y poder controlar el vehículo mediante código para realizar pruebas de modelos aprendidos (Dosovitskiy, Ros, Codevilla, Lopez & Koltun, 2017).

2.1.4. NIVELES DE CONDUCCIÓN AUTÓNOMA

Nivel de automatización	Tareas
Nivel 0	No existe ningún tipo de automatización y el conductor humano se encarga de todo
Nivel 1	El sistema puede realizar el control longitudinal (aceleración) o lateral (giro), no ambos.
Nivel 2	El sistema puede realizar el control longitudinal (aceleración) y lateral (giro).
Nivel 3	Puede realizar el control del nivel 2 más cambios de carriles y reacción ante situaciones adversas en ciertos tipos de ambientes para los que se programó, cuando detecta que ya no está en un ambiente conocido se desactiva.
Nivel 4	Puede realizar todo lo descrito en el nivel 3 y aparte cuando detecta algún tipo de falla o sale de un ambiente conocido, lleva al vehículo a un lugar seguro para desactivarse y que el usuario tome el control.
Nivel 5	Puede realizar una conducción autónoma total, invariante a cualquier situación y lugar.

Tabla 1: niveles de conducción autónoma Fuente:(SAE, 2018)

La Sociedad de Ingenieros de Automoción internacional definió 5 niveles de conducción autónomas con sus características.

Se considera como nivel básico de conducción autónoma a las tareas de control descrito en el nivel 2.

2.2. VISIÓN COMPUTACIONAL

2.2.1. PROCESAMIENTO DE IMÁGENES

Es la aplicación de operaciones del procesamiento de señales unidimensionales sobre imágenes, interpretadas como una señal bidimensional.

2.2.1.1. REPRESENTACIÓN DE IMÁGENES EN UNA COMPUTADORA

Una imagen I está compuesta por píxeles (x, y, v) cuyos componentes son una coordenada $(x, y) \in \mathbb{Z}^2$ y un valor $v \in \mathbb{Z}^c$, con c el número de canales de color, siendo $c = 1$ una imagen a escala de grises y $c = 3$ una imagen a colores Rojo, Verde y Azul (RGB por sus siglas en inglés), definida sobre un conjunto Ω .

$$\Omega = \{(x, y) | 1 \leq x \leq N_{columnas} \wedge 1 \leq y \leq N_{filas}\} \subset \mathbb{Z}^2 \quad (2.1)$$

la representación que se observa en la computadora se le llama modelo grid cell, donde cada píxel es un cuadrado pequeño o celda con una intensidad de grises o canales RGB (Klette, 2014) con valores $0 \leq I_{x,y} \leq 255$.

Existen distintos tipos de imágenes con las que se suelte trabajar en el área de visión artificial, las más comunes son las ya citadas escalas de grises y RGB, sin embargo es común trabajar con imágenes binarias, es decir con valores 0 o 255, para la representación de máscaras o regiones de interés, al igual que espacios de color alternativos como HSV.

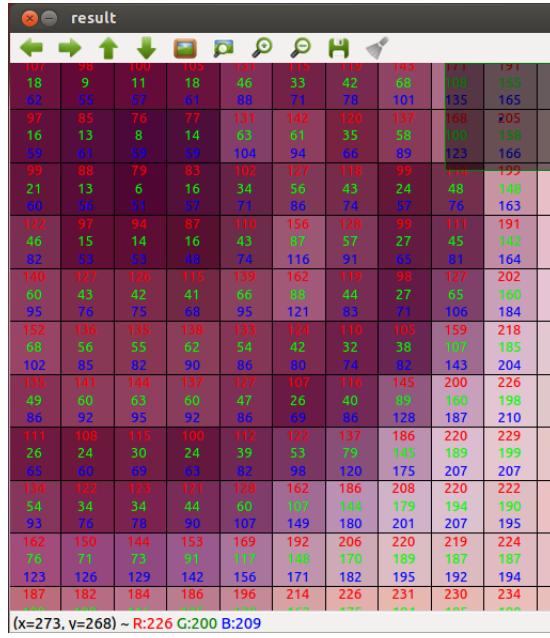


Figura 11: Intensidad de los canales de color RGB
Fuente: (Montabone, 2012)

2.2.1.2. CONVOLUCIÓN

Cuando se desea aplicar alguna transformación de alguna imagen I a alguna imagen I' como difuminados, realce de bordes o extracción de alguna característica, se debe aplicar un operador local lineal entre un filtro que acentúe la característica en la imagen I .

El filtro denotado por W es una matriz cuadrada de dimensión $(2k + 1) \times (2k + 1)$, aplicada en forma de ventana deslizante sobre cada pedazo de la misma dimensión de la imagen original, siendo (x, y) el punto central de cada sección sobre la que se opera, se define la convolución como:

$$I'_{x,y} = \frac{1}{s} \sum_{i=-k}^k \sum_{j=-k}^k w_{i,j} \cdot I_{x+i,y+j} \quad (2.2)$$

y se denota por el operador $*$ como $I' = I * W$ con s un valor de escala (Klette, 2014). Esta operación es equivalente a si se aplasta el filtro y la sección de la imagen en dos vectores, con el segundo vector al revés, y se realiza un producto punto o suma ponderada de sus valores.

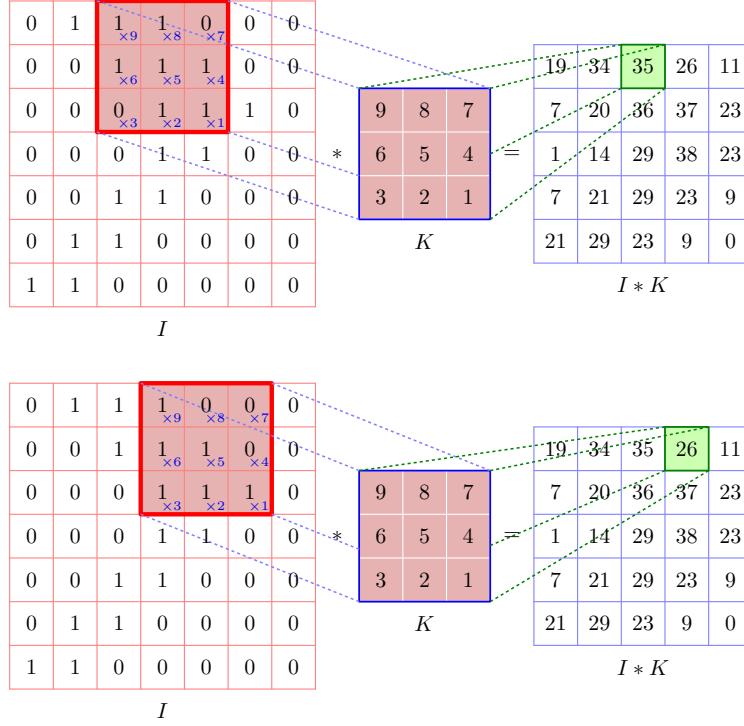


Figura 12: convolución del filtro K con la imagen I

Fuente: elaboración propia

2.2.1.3. DETECCIÓN DE CONTORNOS

2.2.2. TRANSFORMACIONES MORFOLÓGICAS

En tareas de imágenes binarias, el tipo más común de operaciones o filtros aplicados son las transformaciones morfológicas, llamadas así ya que modifican la estructura de los objetos binarios en la imagen, consisten en filtros llamados elementos estructurantes, los cuales son una matriz binaria que contiene un patrón bajo el cual se realiza una comparación o conteo píxel a píxel con recortes de la imagen, similar a una convolución 2D (Szeliski, 2010)

2.2.2.1. DILATACIÓN

Es la operación que consiste en dilatar los píxeles, cuenta cuántos píxeles en el recorte de la imagen que estén en la misma coordenada que los unos del elemento estructurante son distintos de cero, si existe por lo menos un píxel que cumple esta condición entonces el píxel

central de la ventana deslizante se le asignará el valor distinto de cero especificado.

Gráficamente se puede interpretar como engrosar los trazos de algún objeto binario incrementando píxeles en los bordes.

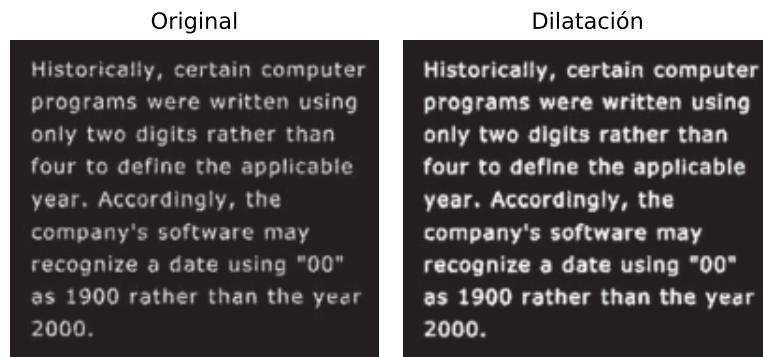


Figura 13: dilatación sobre un conjunto de letras con discontinuidades

Fuente: (Gonzalez & Woods, 2018)

2.2.2.2. EROSIÓN

Es la operación inversa a la dilatación, para esta operación, se realiza el mismo conteo de píxeles distintos de cero que encjen con la estructura del elemento estructurante, pero se asigna un valor positivo al centro solamente si el conteo es igual al número de elementos distintos de cero del filtro.

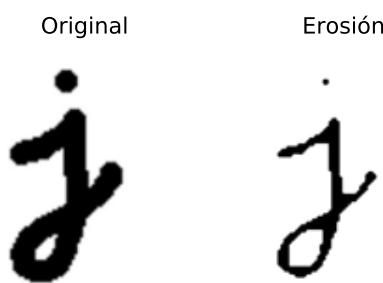


Figura 14: erosión de una letra

Fuente: (Szeliski, 2010)

El resultado gráfico de esta operación es equivalente a reducir los bordes de los objetos haciéndolos más delgados o pequeños.

2.2.2.3. APERTURA

Es una operación que combina erosión seguido de dilatación en un solo paso, se usa para eliminar pequeños artefactos que no pertenecen a los objetos de interés que se buscan extraer, como ruido o falsas detecciones discontinuas.

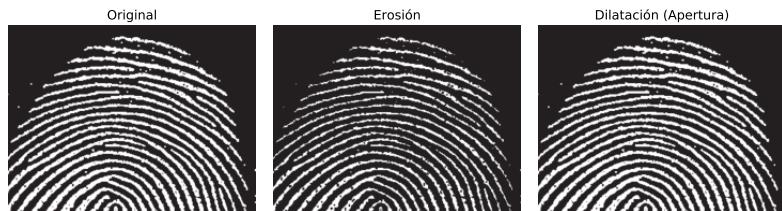


Figura 15: apertura aplicada a una huella digital para eliminar artefactos

Fuente: (Gonzalez & Woods, 2018)

Al aplicar primero una erosión, se eliminan fragmentos discontinuos de menos píxeles que el tamaño de la ventana pero también se reduce el grosor de los objetos de interés, así que se aplica una dilatación, como los artefactos ya desaparecieron para ese momento, simplemente se restauran los objetos que pasaron la fase de erosión.

2.2.3. K-MEANS

En español K-Medias, es un algoritmo nacido en el ámbito del procesamiento de señales y usado ampliamente en minería da datos para particionar un conjunto de n observaciones en k conglomerados o clusters, con el fin de clasificar a qué grupo pertenece cada observación o extraer información relevante de la relación entre los puntos de un mismo subconjunto.

Debido a que se pretende que el algoritmo sea quien extraiga la información de los datos, la única entrada que se le provee es el número de clusters esperados, en base al cual se inicializan k puntos aleatorios representando los centros de cada conglomerado. Al inicializarse aleatoriamente, se deben mejorar iterativamente las coordenadas de los centroides o medias denominados μ_i .

Se define una matriz binaria R de dimensiones $m \times k$ donde m es el número de observaciones, esta matriz codifica en cada fila, representando a cada observación, el cluster al que pertenece, mediante un 1 en la colúmna del cluster i y 0 en las demás. A cada elemento de R

se lo denominará como r_{ij}

Se busca minimizar la distancia entre elementos de cada cluster, es decir la distancia de una observación con su respectivo centroide, esto se hace optimizando la función de costo:

$$J = \sum_{i=1}^m \sum_{j=1}^k r_{ij} \|x_i - \mu_j\|^2 \quad (2.3)$$

La minimización de J se hace mediante dos etapas hasta convergencia. Primero la asignación de centroides a cada punto mediante la regla:

$$r_{ij} = \begin{cases} 1 & \text{si } j = \underset{p}{\operatorname{argmin}} \|x_i - \mu_p\| \\ 0 & \text{e.o.c} \end{cases} \quad (2.4)$$

seguido del cómputo de nuevos centroides mediante el cálculo de la media de los puntos asignados al cluster j en la anterior etapa (Bishop, 2006).

$$\mu_j = \frac{\sum_i r_{ij} x_i}{\sum_i r_{ij}} \quad (2.5)$$

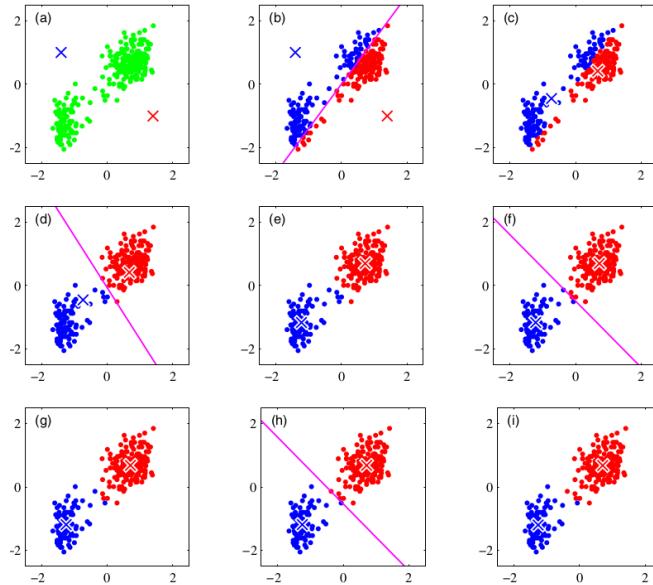


Figura 16: ajuste iterativo de k-means
Fuente: (Bishop, 2006)

2.2.4. FLOOD FILL

Este algoritmo se usa cuando se desea llenar un área delimitada por ciertos valores en una matriz, todas las herramientas de relleno de color en programas de edición de imágenes usan una implementación.

Consiste en recorrer recursivamente la matriz, marcando las posiciones ya visitadas hasta que ya no tenga a dónde ir, esta condición se cumple cuando todas las casillas de alrededor ya están marcadas como visitadas o de algún valor distinto al que se busca (Halim & Halim, 2013).

Algoritmo 1: *Flood Fill*

Requiere: *mat* : Matriz de valores

Requiere: y : coordenada vertical actual

Requiere: x : coordenada horizontal actual

Requiere: d : vector de direcciones

if $mat[y][x] \neq val$ **then**

```
mat[y][x] = marca; // se marca como visitado
```

for $i \rightarrow \text{length } d$ **do**

```
|   flood_fill(mat, y+d[i].y, x+d[i].x)
```

end

end

/* si ya está visitado finaliza la rama de recursión

* /

2.3. APRENDIZAJE AUTOMÁTICO

Del inglés Machine Learning, son métodos basados en la experiencia (datos) que aprenden características de la información disponible para realizar predicciones acertadas (Mohri, Rostamizadeh & Talwalkar, 2018).

2.3.1. APRENDIZAJE SUPERVISADO

Es un tipo de tarea de aprendizaje que consiste en extraer características representativas de un conjunto de datos \mathcal{D} y obtener algún mapeo de cada observación x_i a su correspondiente y_i , donde se conocen los y_i .

2.3.1.1. APRENDIZAJE CORRECTO PROBABLEMENTE APROXIMADO

En una tarea de aprendizaje supervisado se tiene un conjunto de datos \mathcal{D} el cual sigue una distribución conjunta

$$\mathcal{D} \sim P(x, y)$$

con \mathbf{X} el conjunto de observaciones y \mathbf{Y} las etiquetas o valores a predecir. La distribución de los datos es desconocida, por lo que se considera a \mathcal{D} como una muestra aleatoria de tamaño m proveniente de esta distribución.

Con el fin de poder modelar estos datos y aproximarnos a la distribución original para poder realizar inferencias, definimos una función predictiva

$$f : \mathbf{X} \mapsto \hat{\mathbf{Y}}$$

la cual es parte de una familia de funciones o espacio de hipótesis \mathcal{H} definido por un modelo paramétrico, que se ajustan al conjunto de datos con distintos porcentajes de exactitud, dando como salida el valor esperado dado de $y_i \in \mathbf{Y}$ dado $x_i \in \mathbf{X}$.

$$\hat{y}_i = E(y_i|x_i) = f(x_i)$$

La tarea de aprender consiste en elegir la mejor función $f^* \in \mathcal{H}$, que minimice la esperanza de una función de pérdida $\mathcal{L}(\hat{y}_i, y_i)$. Esta función es una métrica definida en \mathbb{R} que nos permite medir cuán lejos está nuestra predicción \hat{y}_i de un valor esperado conocido y_i .

Para obtener la función objetivo se minimiza el riesgo:

$$\begin{aligned} f^* &= \underset{f \in \mathcal{H}}{\operatorname{argmin}} E[\mathcal{L}(f(\mathbf{X}), \mathbf{Y})] \\ &= \int_{\mathbf{X}} \mathcal{L}(f(x_i), y_i) dP(\mathbf{X}, \mathbf{Y}) \end{aligned} \tag{2.6}$$

sin embargo no es posible calcular esta integral ya que no se conoce la distribución, por lo

que se minimiza el riesgo empírico

$$\hat{f} = \operatorname{argmin}_{f \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^m \mathcal{L}(f(x_i), y_i) \quad (2.7)$$

el cual también se conoce como función de costo. (Mohri et al., 2018)

2.3.2. REGRESIÓN

Uno de los tipos más importantes de tareas de aprendizaje automático es la regresión, en que la variable respuesta es continua, es decir $y_i \in \mathbb{R}$.

2.3.2.1. REGRESIÓN LINEAL

Es un modelo que estudia la correlación lineal entre las variables predictoras y las variables predichas, de manera que ajusta el mejor hiperplano (en el caso bidimensional la mejor línea), que modele la correlación lineal de los datos. (Gujarati, 2003)

El modelo de regresión lineal se define como:

$$y_i = \theta \cdot \mathbf{x}_i + \varepsilon_i = \theta_0 + \theta_1 x_{i1} + \dots + \theta_n x_{in} + \varepsilon_i \quad (2.8)$$

dónde x_{ij} es una entrada de la matriz de regresores \mathbf{X} con m observaciones y n características, en la cual el índice i representa el número de observación y j el número de característica para esa observación, θ_j es un parámetro correspondiente a la j -ésima característica el cual representa el grado de incidencia que tiene este atributo sobre la predicción y ε_i : es un término de error que no se puede explicar con las características de nuestros datos o variables regresoras.

De esta manera se puede reescribir el modelo de manera matricial como:

$$\mathbf{y} = \mathbf{X} \cdot \theta + \varepsilon \quad (2.9)$$

con los parámetros

$$\theta = [\theta_0 \quad \theta_1 \quad \dots \quad \theta_n]$$

y con

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1n} \\ 1 & x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & & \ddots & \\ 1 & x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix}$$

con función predictora

$$\hat{\mathbf{Y}} = f(\mathbf{X}) = \mathbf{X} \cdot \theta \quad (2.10)$$

Los parámetros θ que nos permiten encontrar la función \hat{f} se estiman mediante el método de mínimos cuadrados con función de pérdida $\mathcal{L} = (y_i - \hat{y}_i)^2$, y función de costo

$$E(\theta) = \frac{1}{m}(y_i - \hat{y}_i)^2 = \frac{1}{m}(\mathbf{y} - \hat{\mathbf{y}})' \cdot (\mathbf{y} - \hat{\mathbf{y}}) \quad (2.11)$$

dónde el apostrofe representa la operación de trasposición para el caso matricial.

Desarrollando esta fórmula, derivando e igualando a 0 para encontrar el mínimo, se obtiene una fórmula para calcular los parámetros

$$\hat{\theta} = (\mathbf{X}' \cdot \mathbf{X})^{-1} \cdot \mathbf{X}' \cdot \mathbf{y} \quad (2.12)$$

es así como se obtiene la mejor función predictora estimada $\hat{f} = \mathbf{X} \cdot \hat{\theta}$.

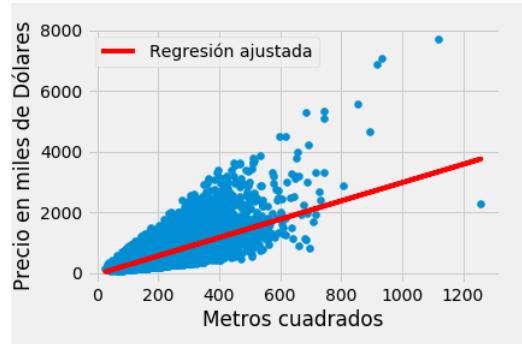


Figura 17: Ajuste de una regresión lineal a un conjunto de datos de precios de casas

Fuente: Elaboración propia

2.3.2.2. REGRESIÓN GENERALIZADA

Si se aplica una función $\mathbb{R}^n \mapsto \mathbb{R}$ antes de obtener la predicción, y se optimiza en base a esta definición, se obtienen variantes de la regresión denominados modelos lineales generalizados.

Un ejemplo de caso categórico es la regresión logística, pero existen distintos otros modelos como la regresión Poisson o incluso la predicción de una regresión con variable de salida en un rango $(-1, 1)$ definida por una tangente hiperbólica.

2.3.2.3. MEDIAS MÓVILES EXPONENCIALES

Es un filtro aplicado a una señal o serie de tiempo, con el fin de promediar los valores en un tiempo t de manera ponderada exponencialmente por los términos anteriores de la serie.

Se aplica comúnmente para encontrar tendencias de los datos y para suavizar oscilaciones dadas por valores atípicos de poca duración en la señal.

Al depender de valores previos de la serie, se calcula de manera recursiva con la fórmula

$$S_t = \begin{cases} y_0, & t = 0 \\ \alpha y_t + (1 - \alpha) S_{t-1}, & t \geq 1 \end{cases} \quad (2.13)$$

definida por un parámetro α que a mayor valor descarta observaciones antiguas más rápido (Washington, 2016).

2.3.3. CLASIFICACIÓN

Cuando la variable respuesta es de tipo categórica, se considera una tarea de clasificación, en que dado un vector de entrada x_i se debe predecir cuál es la categoría a la que pertenece. (Hastie, Tibshirani & Friedman, 2001)

2.3.3.1. REGRESIÓN SOFTMAX

Es un modelo lineal generalizado, el cual extiende la idea de la regresión lineal mediante la función de enlace logit multinomial, también llamada softmax, la cual es una generalización

de la función sigmoide a múltiples clases, cuyo valor de salida es un vector de probabilidades excluyentes para cada fila de la matriz \hat{Y}

$$\text{Softmax}(Z) = \frac{e^Z}{\sum_{j=1}^C e^{Z_j}} \quad (2.14)$$

dónde $Z = \mathbf{X} \cdot \theta$, con θ ahora de dimensiones $(n \times c)$ siendo c el número de clases, de este modo, mientras mayor el valor de $Z_{i,j}$, mayor es la probabilidad al predecir la clase j . (Goodfellow et al., 2016)

Gráficamente, se interpreta este modelo como el mejor hiperplano que separa el espacio en c partes, donde cada parte contiene las observaciones de su correspondiente clase.

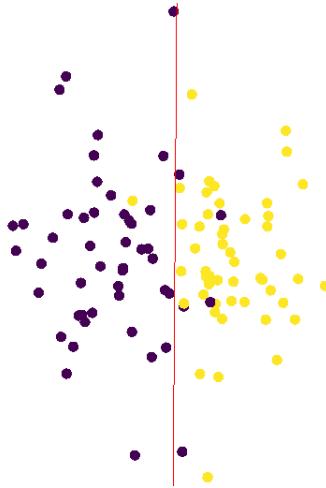


Figura 18: Ajuste de una regresión logística a un conjunto de datos con 2 clases
Fuente: Elaboración propia

Debido a la nolinealidad de la función de enlace, los parámetros ya no se pueden estimar de manera analítica, por lo que se debe usar un método iterativo para resolver el sistema, sin embargo, lo más importante y requisito de todos los métodos, es encontrar la derivada de la función de costo con respecto a los parámetros.

Con el fin de tener una mejor medida del error para las probabilidades de las categorías, se usa la función log softmax

$$\mathcal{L} = -\log(\hat{y}) \quad (2.15)$$

derivando por regla de la cadena $\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial Z} \cdot \frac{\partial Z}{\partial \theta}$, se obtiene

$$\frac{\partial \mathcal{L}}{\partial \theta} = X' \cdot (\hat{y} - y) \quad (2.16)$$

la derivada requisito para cualquier algoritmo de optimización.

2.3.4. DESCENSO DEL GRADIENTE ESTOCÁSTICO

Uno de los métodos más utilizados para resolver el sistema y encontrar los parámetros de modelos no lineales. Consiste en particionar el conjunto de datos en pequeños lotes, de modo que en cada iteración, se evalúa la derivada en cada lote y se mueven los valores de los parámetros en dirección al mínimo de la función (Hastie et al., 2001).

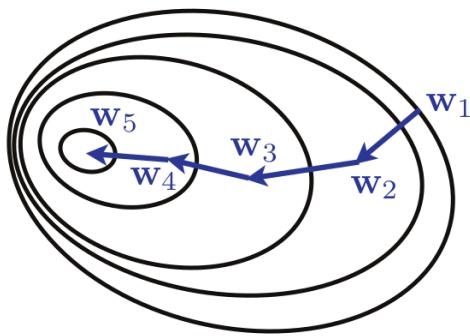


Figura 19: Pasos de los parámetros W en cada iteración camino al mínimo error

Fuente: (Mohri, Rostamizadeh & Talwalkar, 2018)

Algoritmo 2: Descenso del Gradiante estocástico

Requiere: X : Matriz de observaciones
Requiere: Y : Vector de valores a predecir
Requiere: α : Tamaño del paso de aprendizaje
Requiere: b : Tamaño del lote
Requiere: $f(X, \theta)$: Función objetivo a optimizar
 Inicializar aleatoriamente θ
while θ no converge **do**
 | **for** $i \rightarrow \frac{m}{b}$ **do**
 | | $\theta \leftarrow \theta - \alpha \cdot \nabla f(\mathbf{X}_i, \theta)$
 | | **end**
 | **end**
return θ

dónde $\theta \leftarrow \theta - \alpha \cdot \nabla f(\mathbf{X}_i, \theta)$ es el gradiente en cada lote.

2.3.5. ADAM

Es una mejora del descenso del gradiente, la más utilizada debido a la mayor velocidad de convergencia.

Algoritmo 3: Adam

```

Requiere: X : Matriz de observaciones
Requiere: Y : Vector de valores a predecir
Requiere:  $\alpha$  : Tamaño del paso de aprendizaje
Requiere:  $\beta_1, \beta_2 \in [0, 1)$  parámetros de las medias móviles
Requiere: b : Tamaño del lote
Requiere:  $f(\mathbf{X}, \theta)$  : Función objetivo a optimizar
Iniciar aleatoriamente  $\theta$ 
Iniciar en 0  $m$  ;                                // vector de la media
Iniciar en 0  $v$  ;                                // vector de la varianza
 $n \leftarrow 0$ 
while  $\theta$  no converge do
     $n \leftarrow n + 1$ 
    for  $i \rightarrow \frac{n\_observaciones}{b}$  do
         $g \leftarrow \nabla f(\mathbf{X}_i, \theta)$  ;           // gradiente del i-ésimo lote
         $m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot g$  ; // MAE de la media
         $v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot g \odot g$  ; // MAE de la varianza
         $\hat{m} \leftarrow \frac{m}{1 - \beta_1^n}$  ;
         $\hat{v} \leftarrow \frac{v}{1 - \beta_2^n}$ 
         $\theta \leftarrow \theta - \alpha \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$ 
    end
end
return  $\theta$ 

```

El algoritmo aplica dos medias móviles exponenciales, una con parámetro β_1 para estabilizar las oscilaciones calculando la media del gradiente, y otra con β_2 estimar la varianza no central, usando la media de los gradientes al cuadrado debido a la relación.

$$E[X]^2 = E[X^2] - Var[x] \quad (2.17)$$

Dado que los momentos estimados son sensibles a la inicialización, se realiza una corrección de sesgo del estadístico estimado

$$\hat{\bar{X}} = \frac{\bar{X}}{1 - \beta_i^n} \quad (2.18)$$

con n la n -ésima iteración, de esta forma el denominador tiende a 1 conforme pasan las etapas de optimización.

2.4. APRENDIZAJE PROFUNDO

Buscando resolver el problema de la generalización e invarianza a los datos de los que sufren muchos modelos de machine learning o aprendizaje automático, es que nace el Deep Learning o aprendizaje profundo el cual propone apilar múltiples capas con varias neuronas cada una en una red neuronal para lograr representar funciones más complejas y altamente no lineales, a cambio de requerir muchos más datos para su entrenamiento. (Goodfellow et al., 2016)

2.4.1. PERCEPTRÓN MULTICAPA

El perceptrón multicapa, más conocido como red neuronal, extiende la idea de la regresión logística y lineal a un modelo de n -etapas, una especie de concatenación de regresiones con la diferencia que a la salida de las capas se les aplica una función $g(\mathbf{X})$ denominada función de activación, que consiste en alguna transformación no lineal de las variables de salida intermedias con el fin de poder realizar ajustes más complejos a los datos. (Hastie et al., 2001)

Conforme incrementa la cantidad de capas y parámetros de una red neuronal, esta puede aproximar funciones cada vez más complejas, por lo que se le denomina un aproximador universal de funciones, ya que el espacio \mathcal{H} de todas las posibles funciones que puede aproximar, es infinito, es debido a esto que se requieren más observaciones en la muestra de la distribución de datos a modelar. (Goodfellow et al., 2016)

La red neuronal matemáticamente es una composición de n funciones o capas, aplicando una no linealidad $g(\mathbf{X})$ en cada etapa, con una función de enlace $g(\mathbf{X})$ en la capa de salida, la cual comúnmente es una softmax para clasificación o la identidad para regresión, siendo esta última etapa una regresión logística o lineal respectivamente, con variables de entrada procesadas por las anteriores capas de manera que sea linealmente aproximable en un número arbitrario de dimensiones elegidas por el modelo al entrenarse sobre los datos. La función de costo se denota por $J(\theta)$ con θ todos los parámetros del modelo. (Bishop, 2006)

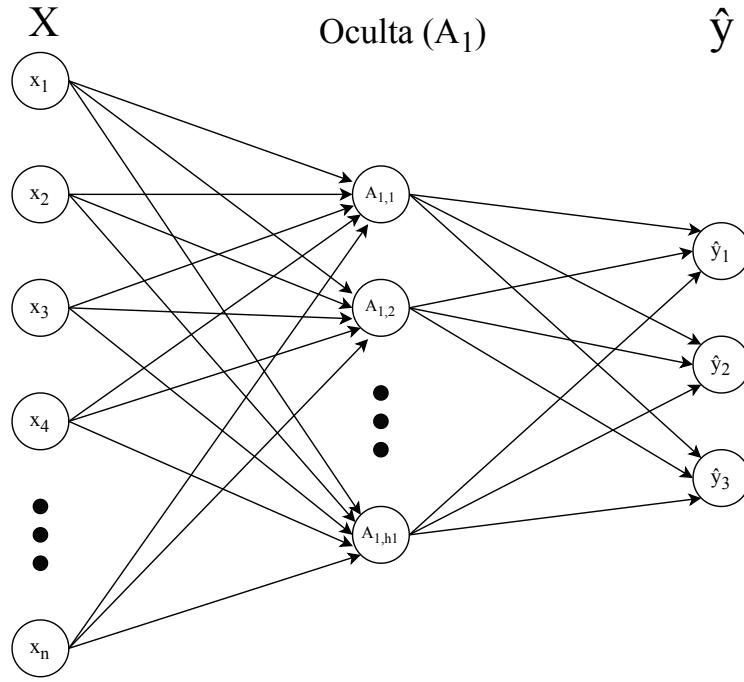


Figura 20: Red neuronal con una capa oculta
Fuente: elaboración propia

Basándonos en la definición de la matriz \mathbf{X} descrita en el modelo de regresión lineal, la columna de unos agregada antes de ajustar el modelo para el parámetro constante, ahora se considerará como un vector de parámetros b_k para las l_k neuronas de la capa k , y los demás parámetros son representados por la matriz W_k .

$$\begin{aligned}
 Z_1 &= \mathbf{X} \cdot W_1 + b_1 \\
 A_1 &= g_1(Z_1) \\
 Z_2 &= A_1 \cdot W_2 + b_2 \\
 A_2 &= g_2(Z_2) \\
 &\dots \\
 Z_n &= A_{n-1} \cdot W_n + b_n \\
 \hat{Y} &= h(Z_n) \\
 J(\theta) &= \sum_i^m \mathcal{L}(\hat{y}_i, y_i)
 \end{aligned} \tag{2.19}$$

2.4.1.1. FUNCIONES DE ACTIVACIÓN

Con el fin de obtener aproximaciones no lineales a los datos, se debe evaluar la salida Z_k de cada capa en una función de activación no lineal $g_k(Z_k)$.

Las funciones de activación más comúnmente usada por ser fácil de computar y diferenciar es la *Rectified Linear Unit* o **RELU** (Goodfellow et al., 2016), la cual está definida por:

$$g(Z_k) = \max(0, Z_k) \quad \forall z_{k,j} / j : 0, 1, \dots, l_k \quad (2.20)$$

cuya derivada con fines de estabilidad numérica es

$$\frac{\partial g(Z_k)}{\partial Z_k} = \begin{cases} 1 & \text{si } z_{k,j} > 0 \\ 0 & \text{en otro caso} \end{cases} \quad (2.21)$$

2.4.1.2. RETROPROPAGACIÓN DE LOS ERRORES

Para ajustar los parámetros o entrenar la red neuronal, al tener más capas por las que pasar para obtener todos los gradientes de los errores, se debe derivar a través de cada una de ellas, a este algoritmo se le llama Backpropagation o Retropropagación, que es simplemente como su nombre dice, propagar los gradientes de reverso a través de la red neuronal.

Primero se obtiene la derivada con respecto da cada uno de los parámetros de la composición de funciones por regla de la cadena

$$\frac{\partial J}{\partial \theta_k} = \frac{\partial J}{\partial Z_n} \cdot \frac{\partial Z_n}{\partial A_{n-1}} \cdot \frac{\partial A_{n-1}}{\partial Z_{n-1}} \cdot \dots \cdot \frac{\partial A_k}{\partial Z_k} \cdot \frac{\partial Z_k}{\partial \theta_k} \quad (2.22)$$

dónde θ_k representa cualquiera de los parámetros de W_k o b_k en la capa k , notese que para obtener la derivada con respecto de los parámetros en la capa k se requiere la derivada en con respecto de las activaciones las capas siguientes, así, cuando se obtiene la derivada con respecto de los parámetros de la capa k ya se calcula para todas las capas siguientes y sólo se debe multiplicar por la derivada de la salida lineal de la capa k con respecto del parámetro θ_k . (Bishop, 2006)

Para el caso de regresión y clasificación con softmax $\frac{\partial J}{\partial Z_n} = (\hat{\mathbf{Y}} - \mathbf{Y})$ de forma vectorial, de manera general la derivada con respecto de algún parámetro está dada por

$$\frac{\partial J}{\partial \theta_k} = \frac{\partial J}{\partial Z_n} \cdot \prod_{i=0}^{n-(k+1)} W_{n-i} \frac{\partial g_{n-(i+1)}(Z_{n-(i+1)})}{\partial Z_{n-(i+1)}} \cdot \frac{\partial Z_k}{\partial \theta_k} \quad (2.23)$$

Similar a la regresión logística, se estiman los parámetros mediante un método de optimización, cuyo requisito sea la derivada con respecto a cada uno de los parámetros.

2.4.2. REDES NEURONALES CONVOLUCIONALES

Es un tipo de arquitectura de redes neuronales diseñada específicamente para tareas sobre imágenes, de manera que las operaciones sobre las observaciones de entrada ya no son composiciones realizando multiplicaciones matriciales, sino que cada neurona se convierte en un filtro o kernel de dimensiones ($k \times k$), de los cuales se tienen varios filtros que se aplican sobre la imagen mediante la convolución.

La idea detrás de este tipo de red es que se apliquen varios filtros en cada capa de la red para extraer características importantes de los objetos que se buscan, estos filtros se aprenden mediante retropropagación y ya no se diseñan a mano (Goodfellow et al., 2016).

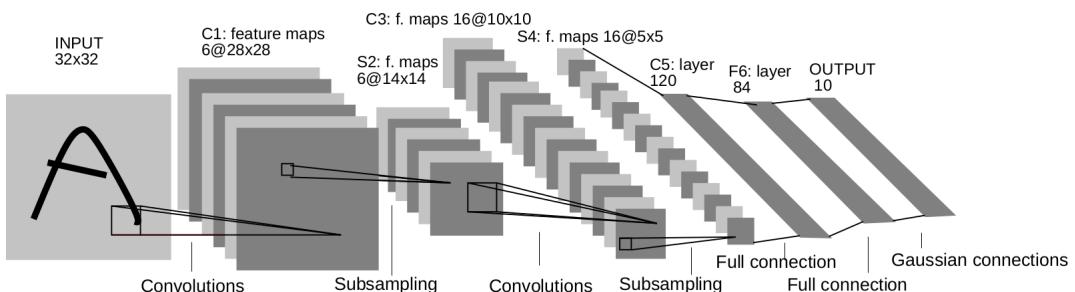


Figura 21: Red neuronal convolucional para la clasificación de dígitos manuscritos
Fuente: (LeCun, Bottou, Bengio & Haffner, 1998)

2.4.2.1. STRIDES

Cuando se desea optimizar la operación sacrificando representabilidad o disminuir la muestra, se puede incrementar el tamaño del salto de la ventana deslizante al convolucionar la imagen con el filtro, a este salto se le llama **stride**

Aplicando esta idea, se obtiene una fórmula general para calcular la dimensión de la matriz resultante al aplicar cada uno de los filtros con un stride s

$$\frac{I_{alto} - k + s}{s} \times \frac{I_{ancho} - k + s}{s}$$

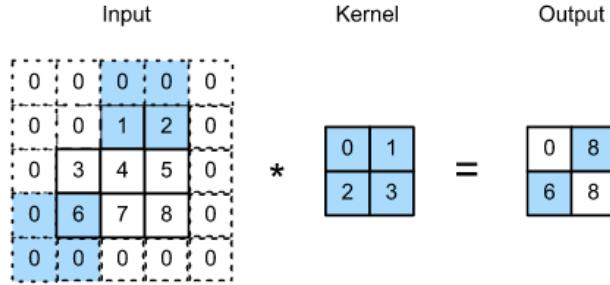


Figura 22: Convolución con $s = 2$
Fuente: (Zhang, Lipton, Li & Smola, 2020)

2.4.2.2. POOLING

Es una operación sobre la entrada bidimensional que con un stride s recorre una ventana deslizante de dimensión $p \times p$ extrayendo información característica de cada sección de la entrada.

Existen dos tipos de pooling más comunes, average pooling que promedia los valores activados en cada sección de la imagen sobre la que pasa la ventana y max pooling, el cual extrae el elemento más representativo, es decir el con mayor valor, de cada sección de la imagen.

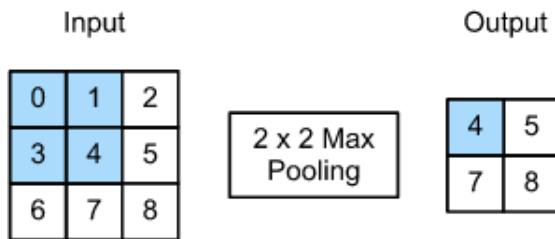


Figura 23: pooling con $s = 1$
Fuente: (Zhang, Lipton, Li & Smola, 2020)

2.4.2.3. MOBILENET V2

Muchas tareas de aprendizaje profundo se despliegan en dispositivos con poco poder de cómputo, las redes neuronales con mejores resultados tienden a tener cada vez más capas y ser más costosas de computar, es por eso que nacen alternativas como la MobileNet, pensada para ejecutarse en dispositivos móviles y embedidos, sacrificando exactitud de predicción por velocidad, aún así obteniendo buenos resultados.

La idea principal, es separar las convoluciones en dos etapas, la primera etapa llamada DepthWise Convolution, consiste en convolucionar la entrada de dimensiones $h_i \times w_i \times c$ con c filtros de dimensiones $k \times k$, para obtener una salida de dimensiones $h_o \times w_o \times c$. La segunda etapa llamada PointWise Convolution recibe la salida de la DepthWise Convolution y le aplica d filtros de dimensiones $1 \times 1 \times c$, para apilar las salidas y obtener finalmente una salida $h_o \times w_o \times d$.

Este mismo resultado se puede obtener mediante una convolución con profundidad por definición, aplicando d filtros de dimensión $k \times k \times c$, sin embargo el número de operaciones es mayor, ya que se realizan $h_i \cdot w_i \cdot c \cdot k^2 \cdot d$ operaciones, comparadas con las $h_i \cdot w_i \cdot c \cdot k^2 + h_o \cdot w_o \cdot c \cdot 1^2 \cdot d$ de la mobilenet. En el caso de que la entrada tenga el mismo alto y ancho que la salida, $h \cdot w \cdot c \cdot (k^2 + d)$ operaciones de las convoluciones separables de la MobileNet.

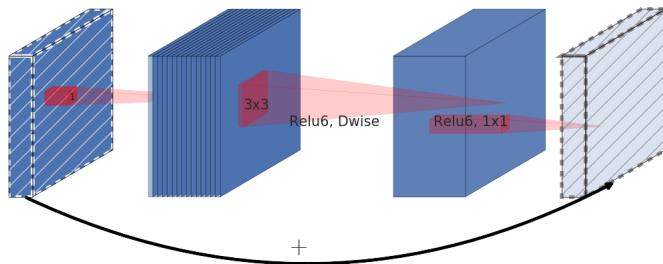


Figura 24: cuello de botella residual (residual bottleneck) Fuente:(Sandler, Howard, Zhu, Zhmoginov & Chen, 2018)

En base esta nueva convolución, se proponen bloques denominados cuellos de botella residuales detallados en la figura 24, que consisten en una capa PointWise con no linealidad Relu truncada con valor máximo 6 llamada Relu6 para obtener $t \cdot k$ filtros (con t llamado el

factor de expansión y k la dimensión del filtro $k \cdot k$), seguido de una capa DepthWise 3×3 con stride s y Relu6, para pasar por otra capa PointWise sin activación no lineal y que devuelve d filtros. El término residual se refiere a que existe una conexión extra que envía directamente la salida de la primera PointWise Convolution a la última para sumarse de manera ponderada, con el fin de prevenir el desvanecimiento de los gradientes durante el entrenamiento (Sandler, Howard, Zhu, Zhmoginov & Chen, 2018).

Usando estos bloques, se construye la arquitectura MobilenetV2 descrita en la tabla 2

Entrada	Operador	Factor t	Canales c	Repeticiones n	Stride s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1×1	-	1280	1	1
$7^2 \times 1280$	average pooling 7×7	-	-	1	-
$1^2 \times 1280$	conv2d 1×1	-	#clases	-	-

Tabla 2: MobileNet V2 Fuente:(Sandler et al., 2018)

2.4.2.4. FAST DEPTH

Una de las áreas abiertas de investigación mediante redes neuronales convolucionales es la de inferencia de profundidad dada una imagen. A diferencia de los métodos de visión estéreo que mediante dos cámaras permite estimar la distancia de objetos al observador, esta tarea pretende hacerlo con una sola cámara, para esto se requiere de una arquitectura Encoder-Decoder, es decir una red codificadora que reciba la imagen RGB como entrada y extraiga las características más importantes de esta en un vector, luego otra red decodificadora, recibe como entrada este vector y devuelve la como salida una nueva imagen dependiendo de la tarea a resolver.

Normalmente para este tipo de problemas se usan redes complejas con alto poder de abstracción, sin embargo para tareas con limitado poder de cómputo se debe simplificar la arquitectura, es así que nace la idea de la red FastDepth una red convolucional de código libre disponible en el repositorio de github de los autores (Wofk, Diana and Ma, Fangchang and Yang, Tien-Ju and Karaman, Sertac and Sze, Vivienne, 2019).

Esta red propone usar una mobilenet v1 como encoder y una nueva red decoder de 5 capas, este decodificador aplica una convolución DepthWise con bordes de relleno para obtener el mismo tamaño de salida, y PointWise para los filtros nuevos, luego de cada convolución se realiza una interpolación por vecinos más cercanos para duplicar el tamaño, hasta llegar a la última capa dónde se aplica solamente una convolucion PointWise.

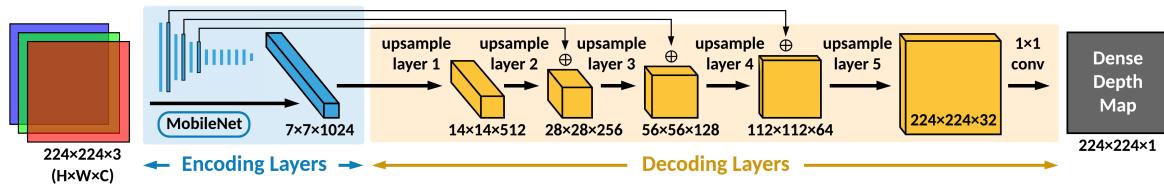


Figura 25: arquitectura FastDepth

Fuente: (Wofk, Diana and Ma, Fangchang and Yang, Tien-Ju and Karaman, Sertac and Sze, Vivienne, 2019)

La salida de esta red es una matriz de 224×224 con las estimaciones de distancia para cada píxel RGB de entrada.

Como especifica la figura 25 se tienen también conexiones residuales, o skip connections, que similar a los bloques de la mobilenet v2, permiten realizar saltos entre las conexiones de las capas mediante una suma ponderada a la capa resultante para así prevenir gradientes ceros durante el entrenamiento, debido a la forma de U de estas conexiones, a esta arquitectura se la denomina U-Net.

2.4.3. APRENDIZAJE DE REPRESENTACIONES PROFUNDAS

En cada etapa de la red neuronal se extraen características representativas de la imagen que ayuden a realizar la predicción de la tarea para la cual se la entrena, conforme pasan más etapas en la red los filtros buscan características más específicas (Goodfellow et al., 2016)

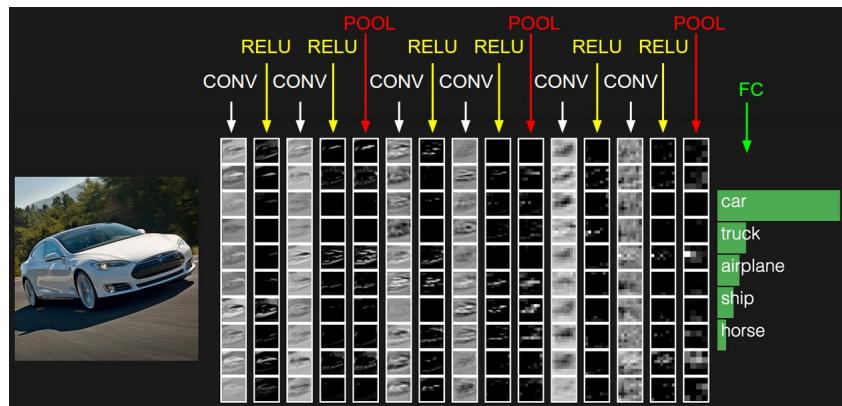


Figura 26: Extracción de las características profundas aprendidas
Fuente: (Stanford, 2020)

de esta manera activando (dando valores altos) a ciertas partes de la imagen que es donde “presta atención” en busca de los objetos que deseé clasificar o en base a los que predecir algún valor numérico, a esto se le llama aprendizaje de representaciones profundas, porque los filtros aprenden información desde bajo a alto nivel que caracterice los objetos en las etiquetas de entrenamiento.

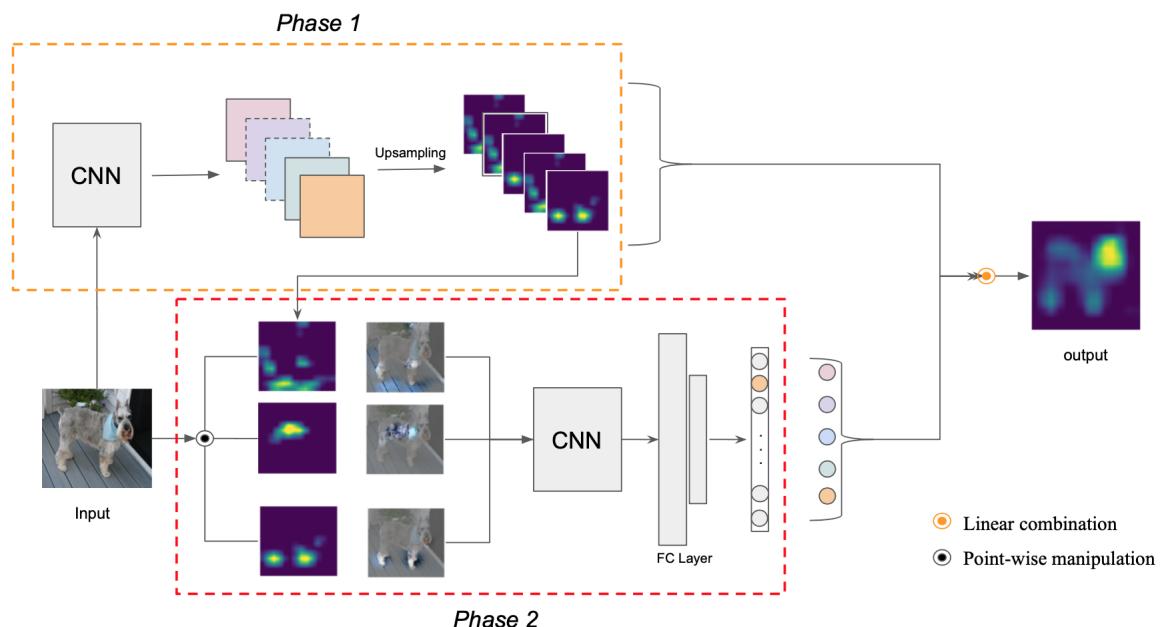


Figura 27: algoritmo ScoreCam
Fuente:(Wang et al., 2020)

Uno de los algoritmos utilizados para esta tarea es ScoreCam, el cual calcula un mapa de prominencia dadas las capas activadas al evaluar una imagen por toda la red descrita en la figura 27. Para lograrlo primero pasa la imagen por la red e intercepta la salida de la última capa convolucional, iterando por sus filtros. En cada iteración para el filtro i -ésimo, realiza un cambio de tamaño mediante interpolación para que el filtro tenga las mismas dimensiones de la imagen original, luego normaliza sus valores activados y los multiplica por la imagen original para re ingresar a la red, se captura la salida de la clase deseada y se acumula ponderando por la clase los valores del filtro por iteración, para finalmente realizar un cambio de tamaño a la variable acumulada y combinar con la imagen original para poder visualizar las áreas de interés de la imagen para la red.

2.5. MÉTRICAS DE ERROR

2.5.1. ERROR CUADRÁTICO MEDIO

Al calcular la diferencia entre observaciones, es útil ponderar las distancias más grandes por sobre las distancias pequeñas, así se define el error cuadrático medio como:

$$MSE = \frac{1}{n} \sum_{i=1}^n (\hat{y} - y)^2 \quad (2.24)$$

Ya que esta métrica se usa al comparar estimaciones y sus valores reales esperados, se denota por \hat{y} a la predicción y y al valor real (Hastie et al., 2001).

2.5.2. ERROR ABSOLUTO MEDIO

Debido a que el error cuadrático medio pondera de manera distinta cada diferencia, si bien es útil para minimizar errores de ajuste de modelos, es difícil de interpretar al reportar resultados, para esto se utiliza el error absoluto medio, también conocido como la media de la norma ℓ_1 , definido como:

$$MAE = \frac{1}{n} \sum_{i=1}^n |\hat{y} - y| \quad (2.25)$$

Esta función también se utiliza para ajustar modelos, sin embargo debido a la singularidad

en el punto 0, no es diferenciable en su dominio, por lo que dependiendo de la implementación se define una derivada en ese punto (Hastie et al., 2001).

2.5.3. PRECISIÓN Y EXHAUSTIVIDAD

Cuando en tareas de clasificación se tienen conjuntos de datos desbalanceados, un cálculo de probabilidad de exactitud no es representativo del rendimiento real del modelo, por lo que se aplican la precisión y exhaustividad.

La precisión (P) se define como el número de positivos reales (T_p), dividido entre el total de verdaderos y falsos positivos F_p .

$$P = \frac{T_p}{T_p + F_p} \quad (2.26)$$

Una alta precisión indica que el umbral de clasificación es más estricto, por lo que pocos valores positivos se clasifican como tal, sin embargo los clasificados como positivo tienen una probabilidad alta de ser predicciones correctas.

La exhaustividad (R) se define como el número de positivos reales, dividido entre la suma de verdaderos casos positivos y falsos negativos (F_n).

$$R = \frac{T_p}{T_p + F_n} \quad (2.27)$$

Una alta exhaustividad o recall, significa que el umbral de clasificación es más permisivo, sin embargo, debido a esto muchas de las predicciones positivas, son en realidad negativas, teniendo así un alto índice de falsos positivos.

Una forma de representar estas dos métricas como una sola es mediante el valor F , definido como:

$$F = 2 \frac{P \cdot R}{P + R} \quad (2.28)$$

al ponderar de manera equitativa la precisión y exhaustividad, un alto valor indica que ambas son altas por igual (Bishop, 2006).

2.5.4. ÍNDICE JACCARD

El índice Jaccard, conocido también como intersección sobre unión, es un estadístico usado para medir la similaridad entre conjuntos finitos mediante una "superposición" de valores comunes.

Se define mediante la fórmula:

$$IoU = \frac{|y \cap \hat{y}|}{|y \cup \hat{y}|} \quad (2.29)$$

En tareas de segmentación semántica, este índice calcula la proporción de píxeles en común entre la máscara real y la estimada, y todos los píxeles que componen la combinación de ambas máscaras (Goodfellow et al., 2016).

2.6. COMPILACIÓN EN TIEMPO DE EJECUCIÓN

Debido a que Python es un lenguaje lento por ser interpretado, en ocasiones donde se requiera código que se ejecute rápidamente sin cambiar de lenguaje, se recurre a la compilación justo a tiempo o Just in Time Compilation, la cual permite compilar una función con los argumentos recibidos la primera vez que se utiliza.

Numba es una librería de código abierto que dota de esta funcionalidad a Python, compilando a código máquina a través de LLVM funciones que cumplan las restricciones para poder inferir correctamente el tipo de dato, está pensado para que de manera directa sólo sea necesario agregar el decorador `@njit` antes de alguna función para que sea compilada la primera vez que se use (Lam, Pitrou & Seibert, 2015).

CAPÍTULO 3. MARCO APLICATIVO

3.1. COMPRENSIÓN DEL PROYECTO

En el presente trabajo se pretende lograr la conducción autónoma básica de un vehículo, al requerir ejecutar pruebas y análisis de resultados, debido a las limitaciones de no contar con el acceso a un vehículo con sensores en la vida real, se hace uso de entorno virtual en un simulador, el elegido para este trabajo es CARLA.

Como primer paso se deben definir los requerimientos, ya que la base de las predicciones son modelos estadísticos basados en datos se tiene la siguiente lista:

- Módulo de extracción de datos.
- Módulo de procesamiento de datos.
- Definir las arquitecturas de redes neuronales a utilizar para cada una de las tareas.
- Módulo de entrenamiento y evaluación.
- Visualización y análisis de las predicciones.

Una vez se tengan los datos a disposición, se tiene como objetivo entrenar las redes neuronales en tres tareas de inferencia:

- Aceleración y giro.
- Profundidad.
- Segmentación semántica.

Utilizando las predicciones de la segmentación para detectar semáforos mediante los contornos, para analizar el color, al igual que las predicciones de distancia para evitar colisiones con otros vehículos.

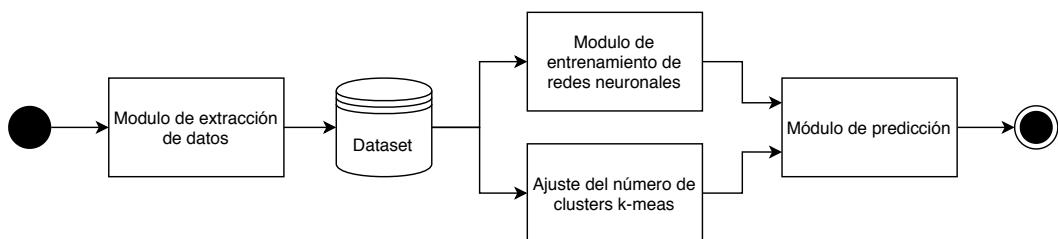


Figura 28: arquitectura del proyecto

Así la estructura del proyecto se resume en los componentes descritos en la figura 28

3.2. COMPRENSIÓN DE LOS DATOS

El simulador cuenta con un sistema integrado para el control de los vehículos llamado Traffic Manager, este hace uso de toda la información disponible sobre el mapa y los objetos en él, siendo estos la topología de las calles, distancia a los obstáculos y coordenadas de los demás vehículos, para mediante código configurar si se quiere ceder el control del vehículo al sistema o controlarlo nosotros.

Se implementa un módulo de extracción (figura 29), que consiste en dejar que el sistema cree un vehículo denominado “jugador”, que será con el que se trabaja, en alguna posición aleatoria del mapa, dejando que el sistema tome control conduciendo el vehículo mediante caminos arbitrarios durante un tiempo definido por 8000 fotogramas, generando datos en forma de imágenes y sus correspondientes etiquetas

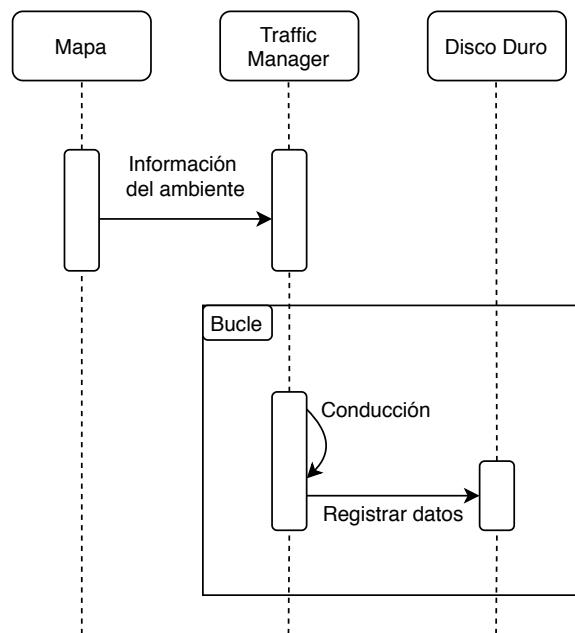


Figura 29: Diagrama de secuencia de extracción de datos

estas son de tres tipos

3.2.1. ACELERADOR Y DIRECCIÓN

Es un par de números, el valor del acelerador en un rango (0, 1), indica cuánto por ciento del acelerador está siendo presionado, la dirección es un valor entre (-1, 1), equivalente a dos

proporciones en una variable, indicando cuánto se debe girar a la izquierda o a la derecha.

3.2.2. PROFUNDIDAD

CARLA cuenta con una cámara virtual que codifica la profundidad o distancia a objetos en un fotograma, estos valores están codificados como números de 24 bits en una imagen RGB que deben ser normalizados de la siguiente manera:

$$\text{distancia} = \frac{R + G \cdot 256 + B \cdot 256 \cdot 256}{256 \cdot 256 \cdot 256 - 1} \cdot 1000$$

para obtener las distancias entre 0 y 1000 metros.

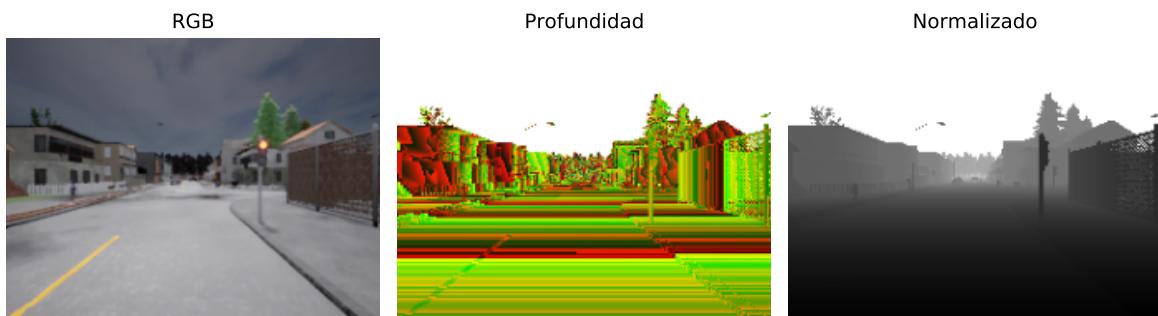


Figura 30: Etiquetas de la profundidad de la imagen

3.2.3. SEGMENTACIÓN SEMÁNTICA

Igualmente CARLA cuenta con una cámara virtual que permite obtener la segmentación semántica de los objetos en un fotograma, estos datos están codificados como valores de píxeles en el canal rojo, cada valor de la máscara indica a qué clase pertenece el píxel correspondiente a esa coordenada de la imagen.

Valor mascara	Clase	Código de color
0	Nada	
1	Edificios	
2	Cercas	
3	Otro	
4	Peatones	

5	Postes	
6	Lineas de carriles	
7	Caminos	
8	Aceras	
9	Vegetación	
10	Vehículos	
11	Paredes	
12	Señales de transito	

Tabla 4: Correspondencia numérica de las clases

para facilitar la visualización se aplica un código de color descrito por la tabla 4, a las clases de píxeles en la imagen.

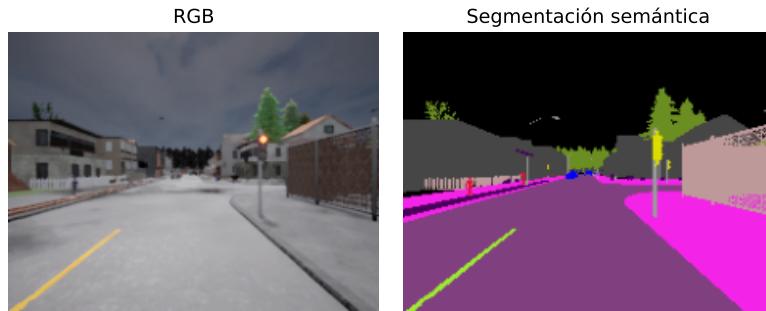


Figura 31: Segmentación semántica con códigos de color

Para facilitar el pre procesado de esta información, por cada simulación se genera una carpeta con tres imágenes dentro, una para imágenes estándar rgb que es lo que se observa a través de la cámara, una carpeta para las máscaras de la segmentación y una para las profundidades, cada archivo dentro de cada carpeta lleva el mismo nombre. Finalmente para las etiquetas del acelerador y dirección, se crea un archivo CSV que contenga la información de manera tabulada, con cada fila conteniendo las columnas

- throttle (acelerador): intensidad del acelerador.
- brake (freno): intensidad de freno.

- steer (aceleración): intensidad de dirección.
- junction (intersección): valor booleano sobre si la imagen pertenece o no a una intersección.

se tienen múltiples CSVs ya que se crean para cada simulación, y la información se obtiene del estado del vehículo y el mapa.

3.3. PREPARACIÓN DE DATOS

Una vez recolectados los datos se debe empezar a procesarlos para agregar atributos de utilidad para los modelos y seleccionar la información que nos interesa, todas las etapas están implementadas en el lenguaje Python.

3.3.1. UNIÓN DE DATAFRAMES

Se debe unir los dataframes en formato CSV en un solo archivo para poder realizar futuras etapas de pre procesamiento.

```
path = '...' # Dirección de los CSVs por simulación
# Cargar una lista de CSVs
dfs = [(file, pd.read_csv(f'{path}/{file}')) for file in listdir_date(path)]

for filename, df in dfs:
    names = [f'{filename.split(".")[0]}/{i}.png' for i in range(len(df))]
    df['filenames'] = names

whole_dataset = pd.concat(list(map(lambda x: x[1], dfs)))

whole_dataset.to_csv(f'{dest}/train_dataset.csv', index=False)
```

Listing 1: unión de dataframes

Se cargan los dataframes en una lista, la cual luego se itera para crear una nueva columna en cada uno, esta columna contiene el nombre de los archivos referenciados en el dataframe bajo la estructura *carpeta/archivo*, para luego unirlos en archivo mediante la función *concat* de la librería de manipulación de datos *pandas*.

3.3.2. ETIQUETADO DE INTERSECCIONES

Luego de un estudio de los datos obtenidos, se decide etiquetar las intersecciones agregando nuevos atributos:

- path left: indica si se tiene un camino a la izquierda disponible.
- path right: indica si se tiene un camino a la derecha disponible.
- path forward: indica si se tiene un camino recto disponible.
- action left: indica si el vehículo tomó el camino por izquierda.
- action right: indica si el vehículo tomó el camino por derecha.
- action forward: indica si el vehículo tomó el camino directo.

Para mantener la consistencia, a las imágenes que no pertenecen a ninguna intersección se les asignó la etiqueta *no action* especificando que no se toma acción alguna.

Debido a las características del mapa, que todas las intersecciones son de tipo T, se debe tener la información de qué caminos se tiene disponibles a tomar, con el fin de ingresar este dato en la etapa de inferencia.

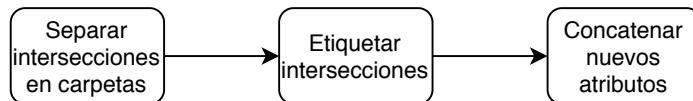


Figura 32: etapas del etiquetado de intersecciones

Este etiquetado se debe realizar de manera manual, por lo que primero se detectan los fotografías que componen las intersecciones, para esto se extraen todas las imágenes marcadas como intersección, se procede a calcular la diferencia de índices entre imágenes, si la diferencia es mayor a 1 entonces inicia una nueva intersección, repitiendo este procedimiento se finaliza la etapa de separación en carpetas (figura 32), los archivos de cada carpeta tienen como nombre el id de simulación de origen, y el nombre de archivo original, para al unificar la información de nuevo sea fácil saber de dónde proviene cada intersección.

Inmediatamente se implementa un script que lea carpeta por carpeta y muestre los fotografías que componen cada intersección como vídeo, para que así el etiquetador pueda ingresar los atributos codificados para el tipo de intersección y la acción tomada, codificado con los controles de dirección estándar en un teclado (wasd).

```

data = defaultdict(list)
for junc in juncs: # Iterando por las carpetas
    data['folder'].append(int(junc))
  
```

```

# Lectura de imágenes
files = sorted(os.listdir(f'junctions/{junc}'),
               key=lambda x: int(x.split('.')[0].split('_')[1]))
for file in files: # Visualización
    img = cv2.imread(f'junctions/{junc}/{file}')
    cv2.imshow('window', img)
    sleep(0.01)
    if cv2.waitKey(25) & 0xFF == ord('q'):
        break
cv2.destroyAllWindows()

print(f'junc: {junc}')
junc_desc = input() # Registro de etiquetas
data['path_left'].append(1 if 'a' in junc_desc else 0)
data['path_right'].append(1 if 'd' in junc_desc else 0)
data['path_forward'].append(1 if 'w' in junc_desc else 0)

junc_action = input()
data['action_left'].append(1 if 'a' in junc_action else 0)
data['action_right'].append(1 if 'd' in junc_action else 0)
data['action_forward'].append(1 if 'w' in junc_action else 0)

df = pd.DataFrame.from_dict(data)
df.to_csv('juncs_final.csv', index=False)

```

Listing 2: interfaz de etiquetado

3.3.3. CONCATENACIÓN DE ATRIBUTOS

En la etapa final del pre procesamiento, se deben concatenar las nuevas columnas de atributos al CSV del conjunto de datos, llenando con valores positivos para la etiqueta *no action* en caso de no pertenecer a una intersección, para esta tarea se implementa un script que cree una lista de valores para cada entrada con valor por defecto 1 y se van asignando ceros para cada elemento de las intersecciones en base a los nombres de los archivos en cada carpeta.

Así finaliza la etapa de pre procesamiento de datos con un CSV indexando todas las imágenes recolectadas con sus correspondientes etiquetas de aceleración, dirección y caminos en intersecciones.

3.4. MODELADO

Una vez se tienen los datos disponibles, se debe definir una arquitectura de entrenamiento de modelos de aprendizaje profundo sobre los datos, las redes neuronales se implementan en

Python usando la librería Pytorch.

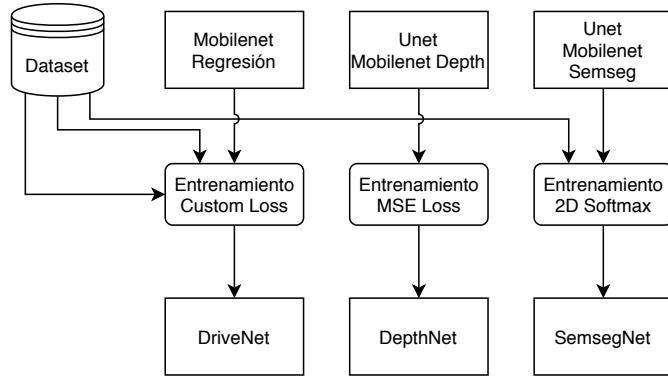


Figura 33: entrenamiento de las tres redes neuronales

Estos módulos están compuestos por tres redes neuronales basadas en la arquitectura MobileNet V2, modificando la implementación estándar y su variante para inferencia de profundidad, un modelo estadístico llamado media exponencial móvil para el suavizado de la dirección, y el algoritmo K means para la cuantificación digital de colores.

3.4.1. RED DE CONDUCCIÓN

Denominada DriveNet, esta red es una modificación de la MobileNet V2 estándar.

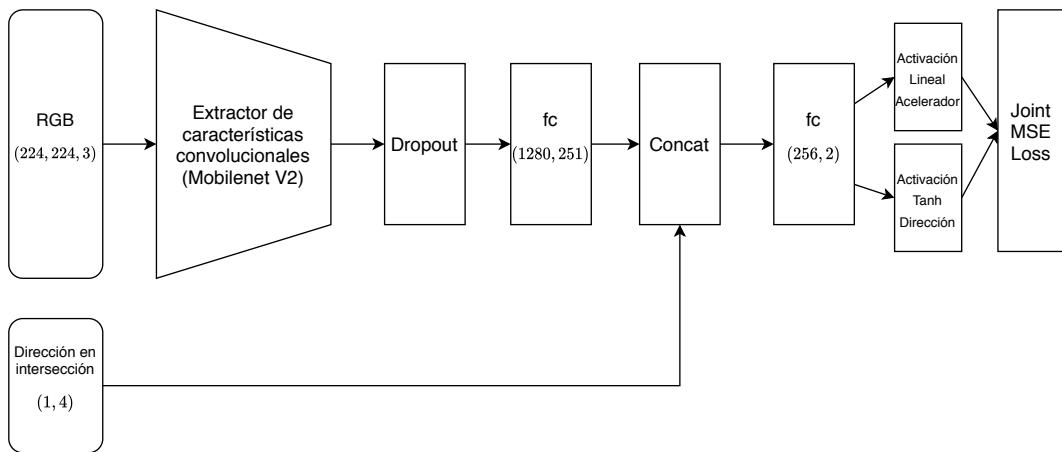


Figura 34: Arquitectura DriveNet basada en MobileNet V2

Se re define la sección del clasificador, en lugar de recibir los 1280 parámetros del extractor de características y predecir el número de clases, se convierte en una etapa de dos capas, una

que recibe 1280 y exporta 251, a la que se le concatenan las 4 nuevas características, para enviar a la siguiente capa que recibirá 256 entradas y devuelve las 2 salidas finales, al adaptar la red a una tarea de regresión, el acelerador es un valor lineal entre 0 y 1, pero la dirección pasará por una tangente hiperbólica ($tanh(x)$) para mapear los valores entre $(-1, 1)$.

```
base_net = mobilenet_v2(pretrained=pretrained)

self.features = base_net.features
self.classifier = nn.Sequential(
    nn.Dropout(0.2),
    nn.Linear(1280, 251),
)
self.concat_fc = nn.Linear(256, num_classes)
```

Listing 3: definición de las capas de predicción

Como función de costo se usa el error cuadrático medio una para la aceleración y otra para la dirección, y la media de ambas como un costo conjunto denominado Joint MSE Loss.

```
x = self.features(x)
x = nn.functional.adaptive_avg_pool2d(x, 1).reshape(x.shape[0], -1)
x = self.classifier(x)
x = torch.cat([x, actions, prev_throttle], dim=1)
x = self.concat_fc(x)
```

Listing 4: propagación hacia adelante y concatenación

3.4.2. RED DE PROFUNDIDAD

Se usa la implementación de la red FastDepth sin modificaciones, basada en una MobilenetV2 con arquitectura de U-Net, los cambios que se dan son en la etapa de entrenamiento, ya que al ingresar las imágenes a la red, se trunca la distancia hasta máximo 30 metros, para simplificar la tarea de inferencia siendo que no es de interés centrarse en objetos a distancias mayores.

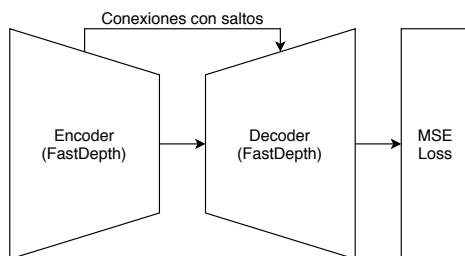


Figura 35: Arquitectura FastDepth

```

img_depth = Image.open(img_depth_path)
img_depth = np.transpose(np.asarray(img_depth, dtype=np.float32), (2, 0, 1))
# Escalando valores entre (0 y 1000)
target = img_depth[0, :, :] + img_depth[1, :, :] * 256 + img_depth[2, :, :] * 256 * 256
# Truncando hasta máximo 30
target = np.clip((target / (256 * 256 * 256 - 1)) * 1000, None, 30)
target = torch.from_numpy(target).float()

```

Listing 5: carga de la imagen para entrenamiento

3.4.3. RED DE SEGMENTACIÓN SEMÁNTICA

Con el fin de simplificar las estructuras de los modelos, se utiliza la misma implementación de la red FastDepth modificando la función de costo por una Softmax 2D, ya que la tarea de segmentación semántica, es un tipo de clasificación píxel a píxel.

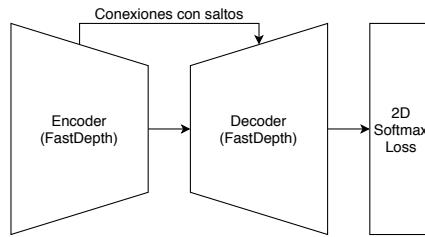


Figura 36: SemsegNet, FastDepth para clasificación 2D

3.4.4. SUAVIZADO DE DIRECCIÓN

Complementando las predicciones de las redes neuronales, se hace uso de una media exponencial móvil con parámetro $\alpha = 0.7$, para suavizar las oscilaciones de la conducción con el fin de reducir “volantazos”, tratando estos valores como una serie de tiempo.

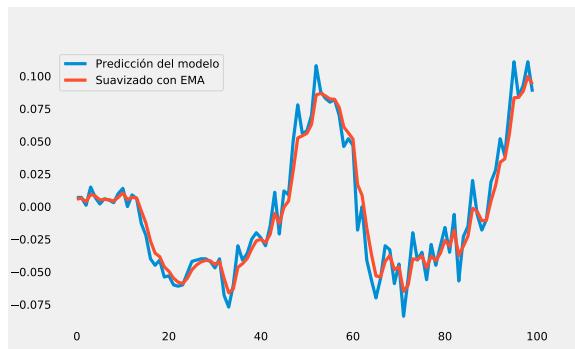


Figura 37: Aplicación de medias exponenciales móviles para suavizar la dirección

Este modelo se aplica en cada iteración de la simulación a las predicciones de giro sólo cuando los valores están dentro del rango $(-0.2, 0.2)$ y fuera de una intersección, ya que valores mayores de giro indican que el vehículo está tomando una curva y no generando una oscilación.

```

alpha = 0.75
ema = None
threshold = 0.2

while True:
    # ... código de control y predicción en cada iteración

    # se obtienen las predicciones de cada red
    out = model_drive(X, action_tensor).cpu()
    depth_map = model_depth(X_D)[0, 0].cpu().numpy()
    segmentation = model_semseg(X).cpu().numpy()[0].argmax(axis=0).astype(np.uint8)

    # Redondeo de valores para reducir precisión y oscilaciones
    # t: throttle (acelerador), s: (steer) dirección
    t, s = round(float(out[0, 0]), 3), round(float(out[0, 1]), 3)
    # Limitación de la velocidad
    t = min(t, 0.5)

    # Si estamos fuera de una intersección
    if isinstance(ema, type(None)): # Si ema está en t=0
        ema = s                      # inicializar
    else:
        ema = alpha*s + (1-alpha)*ema # Dar un paso de EMA

    if abs(s) <= threshold: # Si está en el rango
        s = ema                  # Aplicar el valor del EMA

```

Listing 6: uso de media exponencial móvil en la predicción

3.4.5. CAJA DELIMITADORA

Una vez se tiene la máscara de píxeles que componen los semáforos luego de la limpieza por dilatación, erosión y apertura, se aplica el algoritmo Flood Fill de manera iterativa, partiendo por los píxeles a partir de la coordenada 120 en dirección horizontal, así cada vez que se encuentre un valor positivo de la máscara, el algoritmo lo “pintará” de ceros eliminando el objeto pero devolviendo las coordenadas de los puntos extremos arriba-izquierda y abajo-derecha que definen el rectángulo de mínima área delimitando el objeto.

```

@njit
def ff(mat, i, j, x1, y1, x2, y2, directions):
    if (0 <= i < mat.shape[0]) and (0 <= j < mat.shape[1]) and mat[i, j] != 0:
        mat[i, j] = 0
        x1 = min(x1, j)
        y1 = min(y1, i)
        x2 = max(x2, j)
        y2 = max(y2, i)

        for dx, dy in directions:
            x1, y1, x2, y2 = ff(mat, i+dy, j+dx, x1, y1, x2, y2, directions)

    return x1, y1, x2, y2

```

Listing 7: Flood Fill para la extracción de la caja delimitadora

Debido a que Python es un lenguaje interpretado y lento para tareas pesadas se compila la función mediante numba con el decorador @JIT (Just in Time Compilation).

3.4.6. CUANTIFICACIÓN DIGITAL DEL COLOR

Aplicando K-Means se reduce la cantidad de colores en la imagen, manteniendo los más predominantes, al ingresar al algoritmo un recorte que contiene específicamente un semáforo, los 4 colores más predominantes serán los de la luz emitida, de esta manera, se analiza la existencia de píxeles con valores mayores a cero en ciertos canales.

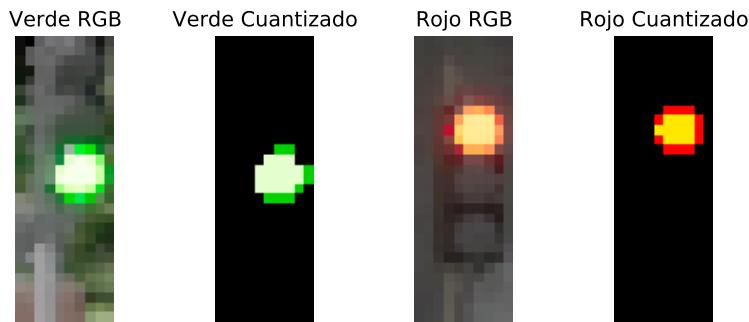


Figura 38: cuantificación digital del color para clasificación de semáforos

En caso que los canales rojo o rojo y verde tengan valores distintos de cero, se tienen rojo o amarillo en cuyo caso el semáforo está en estado de pare, si los valores son positivos en verde y azul, se tiene alguna tonalidad de verde o blanco como se observa en la figura 38

```

cod = ['r', 'g', 'b']
cod_val = 'na'

# En caso de que se detecte un semáforo y se recorte en la variable crop
inp = np.float32(crop.reshape((-1,3)))
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 4
# Label: id del cluster al que pertenece cada pixel de la imagen
# Center: valores de los pixeles de los centros en cada canal
_, label, center = cv2.kmeans(inp, K, None, criteria,10, cv2.KMEANS_RANDOM_CENTERS)
# Toma el id del cluster al que pertenece el pixel y le asigna el valor del centro
res = np.uint8(center[label.flatten()]).reshape((crop.shape)) # Reconstruye la imagen
# Selecciona los valores de pixeles mayor a 200
res = res * (res > 200)
# Suma los pixeles mayores a 200
rgb_sig = [res[:, :, 0].sum(), res[:, :, 1].sum(), res[:, :, 2].sum()]
rc, gc, bc = rgb_sig
# si los canales rojo y verde son mayores a cero
if rc > 0 and gc > 0:
    if bc > 0: # si lo es también azul
        cod_val = 'g' # es verde
    else:
        cod_val = 'r' # sino es rojo
elif rc > 0 or gc > 0 or bc > 0: # si alguno de los 3 es mayor a cero
    cod_val = cod[np.argmax(rgb_sig)] # se toma el color con más ocurrencias

```

Listing 8: kmeans para clasificación del color de semáforos

3.4.7. MODELO PARA LA CONDUCCIÓN AUTÓNOMA

Una vez descritos todos los módulos, se define el modelo y sus interacciones.

La entrada será una imagen en RGB de resolución 240×180 , la cual pasará por las tres redes descritas.



Figura 39: trapecio delimitando el área de interés para la detección de obstáculos

Se usan las predicciones en conjunto de la DepthNet y SemsegNet, pasando previamente por una máscara que elimina todos los píxeles fuera de un trapecio (figura 39) que corresponde al área de interés (con el fin de evitar falsos positivos), para extraer los píxeles correspondientes a posibles obstáculos, y se calcula la moda de las distancias redondeadas, para así decidir si existe peligro de colisión o no.

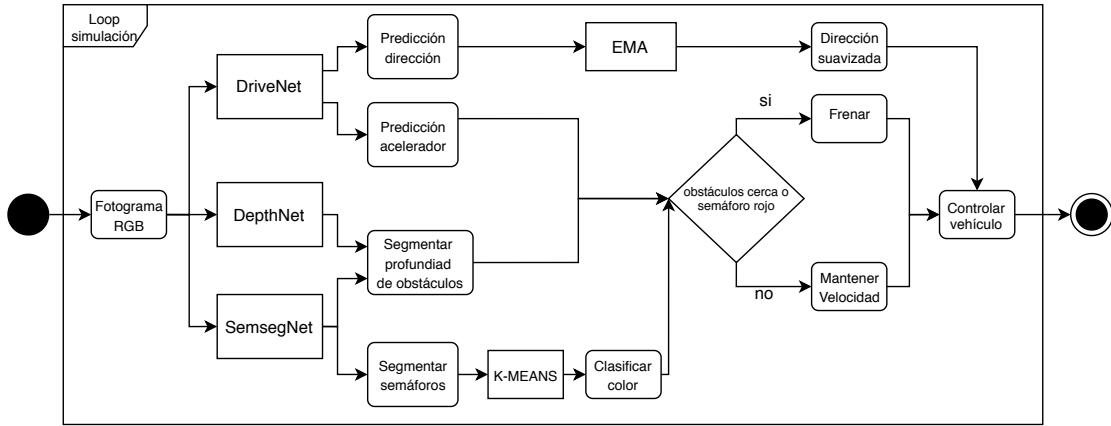


Figura 40: modelo de conducción autónoma

Se realiza un recorte a la zona donde se encuentre algún semáforo en caso de que se lo detecte, para lograr esta detección se aplica dilatación, erosión y apertura con el fin de reducir artefactos en las predicciones de la segmentación semántica, y se cuantiza el color para decidir si detenerse o no.

Combinado con estas predicciones, la DriveNet infiere la dirección, la cual se suaviza mediante la media exponencial móvil, y aceleración, la cual se mantiene en caso de no presentarse obstáculos cerca ni semáforos en rojo.

Los módulos descritos se pueden observar de manera estructurada en la figura 40.

3.5. EVALUACIÓN

Se analizan las curvas de aprendizaje para cada modelo usando Adam como optimizador.

3.5.1. DRIVENET

Para la red de conducción se tiene la curva dada en la figura 41 de un total de 30 iteraciones, con parámetros $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$. De estos resultados se decidió realizar

pruebas entre la iteración 24 y 30, se tomaron los parámetros de la epoch 24 con errores cuadráticos medios de 0.00051 para el conjunto de entrenamiento y 0.00088 para el conjunto de validación.

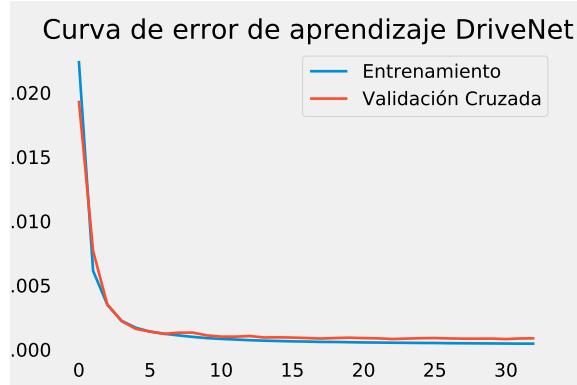


Figura 41: errores de entrenamiento y validación de la DriveNet

3.5.2. DEPTHNET

Para la red de profundidad se tiene la curva dada en la figura 42 de un total de 24 iteraciones, con parámetros $\alpha = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$.

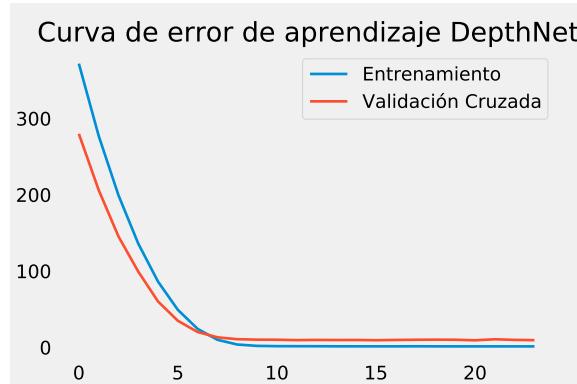


Figura 42: errores de entrenamiento y validación de la DepthNet

De estos resultados se eligieron los parámetros de la iteración 20 con errores cuadráticos medios de 1.30575 para el conjunto de entrenamiento y 9.267 para el conjunto de validación. Es importante notar que no se tomaron los parámetros de alguna iteración que distara poco del error de entrenamiento, sino de la iteración en la que se tenía el menor error de validación durante el entrenamiento.

3.5.3. SEMSEGNET

Para la red de segmentación semántica se tiene la curva dada en la figura 43 de un total de 7 iteraciones, ya que esta era la más costosa de computar, con los mismos parámetros que la DriveNet.

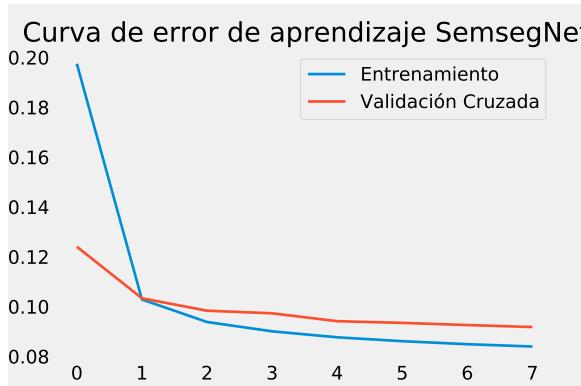


Figura 43: errores de entrenamiento y validación de la SemsegNet

De estos resultados se eligieron los parámetros de la iteración 7 con errores del tipo mean log softmax 2D de 0.08396 para el conjunto de entrenamiento y 0.09178 para el conjunto de validación.

Al igual que para la red de profundidad se eligieron los parámetros con el menor error en el conjunto de validación cruzada.

3.5.4. CENTROIDES K-MEANS

Finalmente se evalúa que la decisión sea correcta para el número de centroides en el algoritmo KMEans, debido a que el espacio de color RGB se puede representar como 3 dimensiones espaciales, se analizó gráficamente el desempeño del algoritmo, se decidió que 4 centroides era la mejor cantidad para la tarea de cuantificación digitalización del color.

Observando la figura 44, se nota que de los cuatro centroides señalizados con figuras distintas de esferas y de color azul, dos de ellos están en el punto de mayor aglomeración que son los píxeles de colores oscuros del borde del semáforo, mientras que los otros dos están en los puntos más intensos del color rojo o verde y en el punto de combinación entre dos colores, blanco para el caso verde-azul y amarillo para rojo-verde.

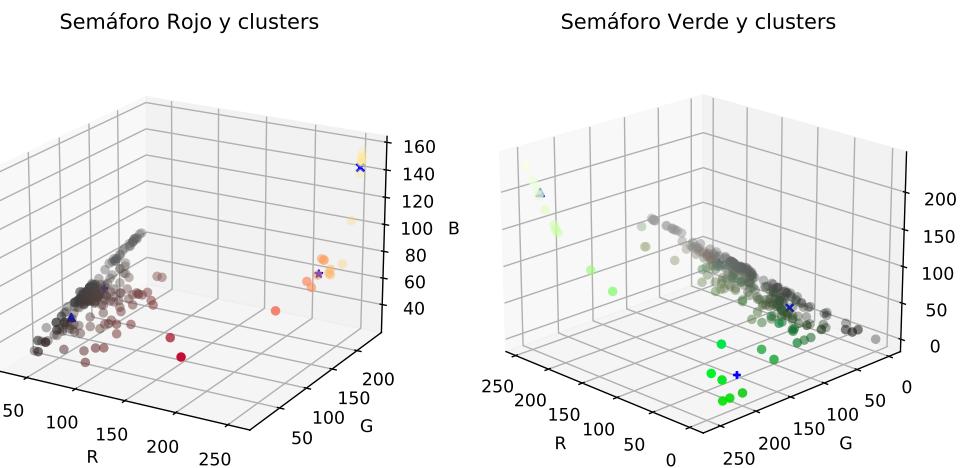


Figura 44: espacios de color y centroides para dos semáforos distintos

3.6. DESPLIEGUE

Para realizar el despliegue del proyecto se implementa un script que se comunique con el simulador para extraer imágenes desde una cámara virtual hasta que el usuario detenga el procedimiento, para cada iteración de la simulación se realizan las predicciones en base al modelo propuesto y se adicionan funciones de visualización de las predicciones de las redes.

Los pasos más importantes son:

1. Definir la pantalla de visualización de imágenes mediante la librería PyGame.
2. Conectar con el simulador e inicializar el vehículo en el mapa con un clima aleatorio.
3. Cargar los parámetros de las redes.
4. Iniciar un bucle de iteraciones de la simulación a 30 fps.
5. Detectar si se está en una intersección mediante el simulador y decidir al azar qué camino tomar.
6. Ingresar los valores de entrada a cada red neuronal y recibir su salida.
7. Corregir las oscilaciones con EMA.
8. Si el vehículo no realiza alguna curva en intersección, se da un impulso mediante un acumulador de giro.
9. Se procesan las predicciones de la segmentación semántica para las clases vehículos y postes.

10. Se calcula la moda de las distancias de objetos cercanos.
11. Se detecta la posición de los semáforos.
12. Se predice un código de color, r para rojo o g para verde.
13. Se usa el código de color para decidir si frenar o no.
14. Se envían las decisiones finales de control al vehículo.

el código del script está listado en el anexo *F.3*.

CAPÍTULO 4. RESULTADOS Y ANÁLISIS

4.1. RENDIMIENTO DE LOS MÓDULOS

Al entrenar las redes neuronales, se siguió el procedimiento estándar de separar el conjunto de datos en entrenamiento y validación cruzada para analizar si el modelo está aprendiendo o memorizando los datos.

Así se seleccionan los parámetros de las iteraciones o epochs con menor error en el conjunto de validación cruzada, detallados en la tabla 5, cuyas gráficas (figuras 41, 42 y 43) fueron descritas en la sección 3.5.

Modelo	Error cuadrático medio	Iteración
DriveNet	0.00088	24
DepthNet	9.267	20
SemsegNet	0.09178	7

Tabla 5: Errores en el conjunto de validación de los parámetros seleccionados

4.1.1. ACELERACIÓN Y GIRO

Debido a que el error cuadrático medio penaliza de manera ponderada los valores más lejanos por sobre los cercanos, razón por la cual es una buena métrica de error a la hora de ajustar modelos, es difícil darle una interpretación comprensible al reportar resultados, es por eso que se calcula el error absoluto medio de las predicciones con los valores reales esperados del conjunto de validación.

Para el caso de la predicción de aceleración, considerando que las redes no predicen el uso de frenos sino que estos se aplican cuando el modelo reacciona a semáforos u objetos cercanos, se obtiene un error absoluto medio de 0.067, es decir que la predicción de aceleración oscila ± 0.067 de su valor real, esto debido a que la aceleración no es muy variable en el conjunto de datos de entrenamiento, por lo que al haber pocas situaciones inesperadas, las predicciones son un número fijo con pocas variaciones en curvas.

De manera similar para la dirección suavizada por la media exponencial móvil, se obtiene un error de ± 0.069 . Este error es mayor al de la aceleración, debido a que es más complicado

para las redes mantener una dirección estable si no tiene información temporal, a pesar del uso de medias móviles exponenciales, esta no puede compensar todos los casos de variación.

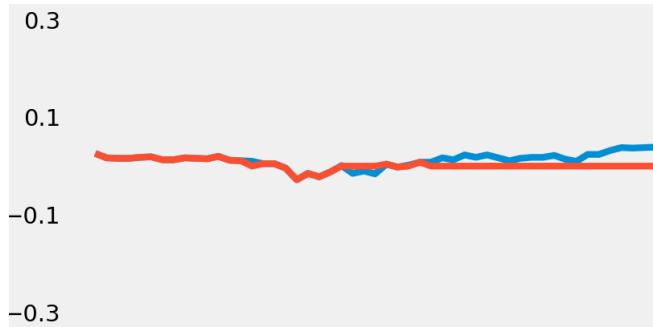


Figura 45: media móvil exponencial (rojo) suavizando los giros originales (azul)

A pesar de las diferencias calculadas, el error es lo suficientemente pequeño para que no existan acelerones inesperados o giros que causen desestabilidad al vehículo, por lo que se consideraran errores aceptables en la práctica.

4.1.2. ESTIMACIÓN DE PROFUNDIDAD

Al ser también una tarea de predicción de valores en \mathbb{R} , se calcula el error absoluto medio entre el mapa de profundidades estimado por la red y el del conjunto de datos de validación. En este caso el error es de ± 1.1931 , el cual es mucho más elevado que los errores de la red de conducción, esto debido a que las posibilidades de error son mayores ya que se predice la distancia pixel a pixel.

La consecuencia de esto se nota mayormente al encontrarse con objetos cercanos, donde se puede predecir erróneamente que está muy cerca, en cuyo caso el vehículo se detiene totalmente ante un falso obstáculo, o el caso más peligroso que se da cuando el objeto está muy cerca y se predice que está una unidad más lejos del límite en el cual el modelo decide detener el auto y se produce una colisión.

Gráficamente en la figura 46 se observa que la predicción es “borrosa” comparado con el mapa de profundidad original

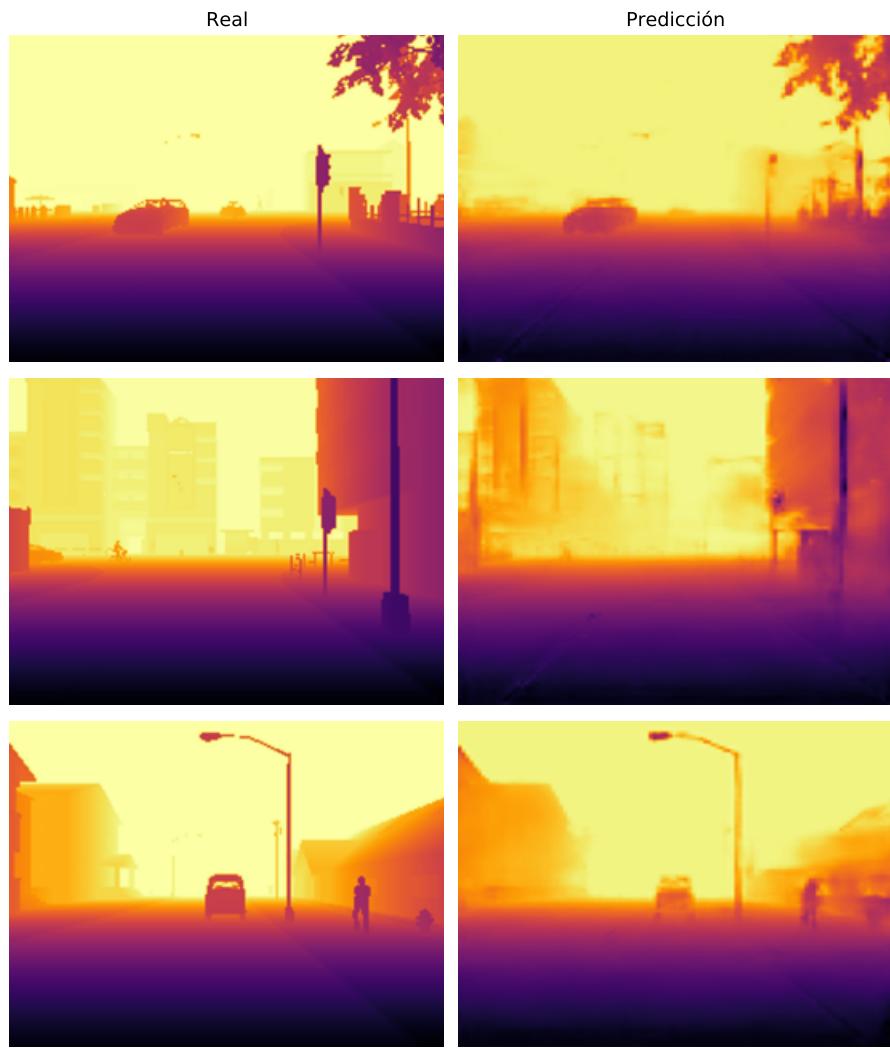


Figura 46: predicción vs valor esperado de profundidad

4.1.3. SEGMENTACIÓN SEMÁNTICA

El módulo de segmentación semántica, considerando la limpieza de predicciones residuales mediante erosión, dilatación y apertura, es uno de los componentes más importantes, ya que la detención ante obstáculos y la detección de objetos depende de las predicciones de esta red, además, al ser una tarea de clasificación, se pueden analizar los resultados a mayor detalle con distintas métricas de error.

4.1.3.1. MATRIZ DE CONFUSIÓN

Se realiza el cálculo de la matriz de confusión, sin embargo, al existir muchas ocurrencias de cada clase, se decide estandarizar los valores a porcentajes por fila para mejor visualización, teniendo así la figura 47, dónde las diagonales indican los valores predichos correctamente, y fuera de la diagonal están las detecciones erróneas.

Matriz de confusión porcentual														
Nada	Edificios	Cercas	Otro	Peatones	Postes	Carreles	Caminos	Aceras	Vegetación	Vehículos	Paredes	Señales		
97.87	0.69	0.06	0.03	0.01	0.11	0.00	0.00	0.12	1.03	0.03	0.05	0.01		
0.66	97.38	0.28	0.18	0.02	0.25	0.00	0.00	0.29	0.62	0.09	0.15	0.08		
0.29	2.32	90.40	0.47	0.10	0.34	0.00	0.48	4.42	0.40	0.72	0.05	0.01		
0.92	8.39	1.10	80.77	0.18	0.34	0.00	0.24	5.57	0.42	0.46	1.59	0.02		
2.00	12.47	3.64	3.30	48.25	0.50	0.00	6.50	6.75	0.14	13.76	2.70	0.00		
10.14	16.13	1.31	1.22	0.03	54.04	0.00	0.52	7.89	6.40	0.52	0.59	1.22		
37.90	0.01	0.00	0.00	0.13	0.00	0.00	61.64	0.02	0.00	0.21	0.00	0.08		
0.01	0.00	0.03	0.01	0.01	0.01	0.00	99.13	0.58	0.00	0.22	0.00	0.00		
0.32	0.46	0.58	0.46	0.04	0.28	0.00	1.41	96.07	0.01	0.25	0.12	0.00		
3.27	2.19	0.21	0.05	0.00	0.23	0.00	0.00	0.08	93.82	0.02	0.08	0.05		
0.19	1.33	0.51	0.19	0.24	0.03	0.00	3.72	0.98	0.16	92.54	0.04	0.07		
1.37	4.90	0.13	0.85	0.22	0.29	0.00	0.09	1.64	0.52	0.60	89.36	0.04		
0.83	24.90	1.02	0.88	0.00	3.22	0.00	0.00	0.00	4.14	0.02	0.03	64.96		

Figura 47: matriz de confusión de segmentación semántica

Se puede observar en la figura 47, que cuatro clases que se utilizan en la conducción, las cuales son peatones, postes, vehículos y señales de tránsito, tienen porcentajes de predicción de verdaderos positivos 48.25 %, 54.04 %, 92.54 % y 64.96 % respectivamente, siendo la detección de peatones y postes la menos exacta, la primera porque los peatones normalmente aparecen en aceras y en los bordes del campo de visión, y los postes debido a que son delgados y por

la baja resolución de predicción es difícil para la red segmentarlos completamente.

4.1.3.2. PRECISIÓN Y EXHAUSTIVIDAD

Una vez se tiene la matriz de confusión, se pueden calcular la exactitud pixel a pixel, precisión, exhaustividad y el valor-F por clases, los cuales se listan en la tabla 6

Clase	Precisión	Exhaustividad	Valor-F
Edificios	0.963	0.9738	0.9684
Cercas	0.8929	0.904	0.8984
Peatones	0.7141	0.4825	0.5759
Postes	0.7398	0.5404	0.6246
Carriles	0.0	0.0	0.0
Caminos	0.9855	0.9913	0.9884
Aceras	0.9517	0.9607	0.9562
Vegetación	0.9218	0.9382	0.9299
Vehículos	0.9402	0.9254	0.9327
Paredes	0.8707	0.8936	0.882
Señales	0.7802	0.6496	0.7089

Tabla 6: métricas de error de clasificación

De la tabla de errores y la matriz de confusión se puede notar primeramente que debido al desbalance de las muestras por clase, las predicciones de líneas de carriles son todas negativas, lo cual da una idea engañosa de alta exactitud, sin embargo el valor F, usado para compensar este sesgo dejan en claro que en realidad la predicción es totalmente errónea. Esto no es un problema para el modelo debido a que no se usa esta clase para las predicciones.

Para las cuatro clases utilizadas tenemos que:

- **Peatones:** un 71.41 % del total de píxeles clasificados correctamente son positivos, 48.25 % de positivos reales son clasificados correctamente lo cual indica una alta cantidad de falsos negativos, con una exactitud balanceada de 57.59 %.
- **Postes:** precisión 73.98 % y exhaustividad del 54.04 %, dando menos falsos negativos con exactitud balanceada de 62.46 %.

- **Vehículos:** precisión del 94.02 %, exhaustividad de 92.54 % y exactitud de 93.27 % siendo esta la clase con los menores errores de predicción.
- **Señales de tránsito:** precisión 78.02 %, exhaustividad 64.96 %, con muchos menos falsos negativos, y 70.89 % de exactitud.

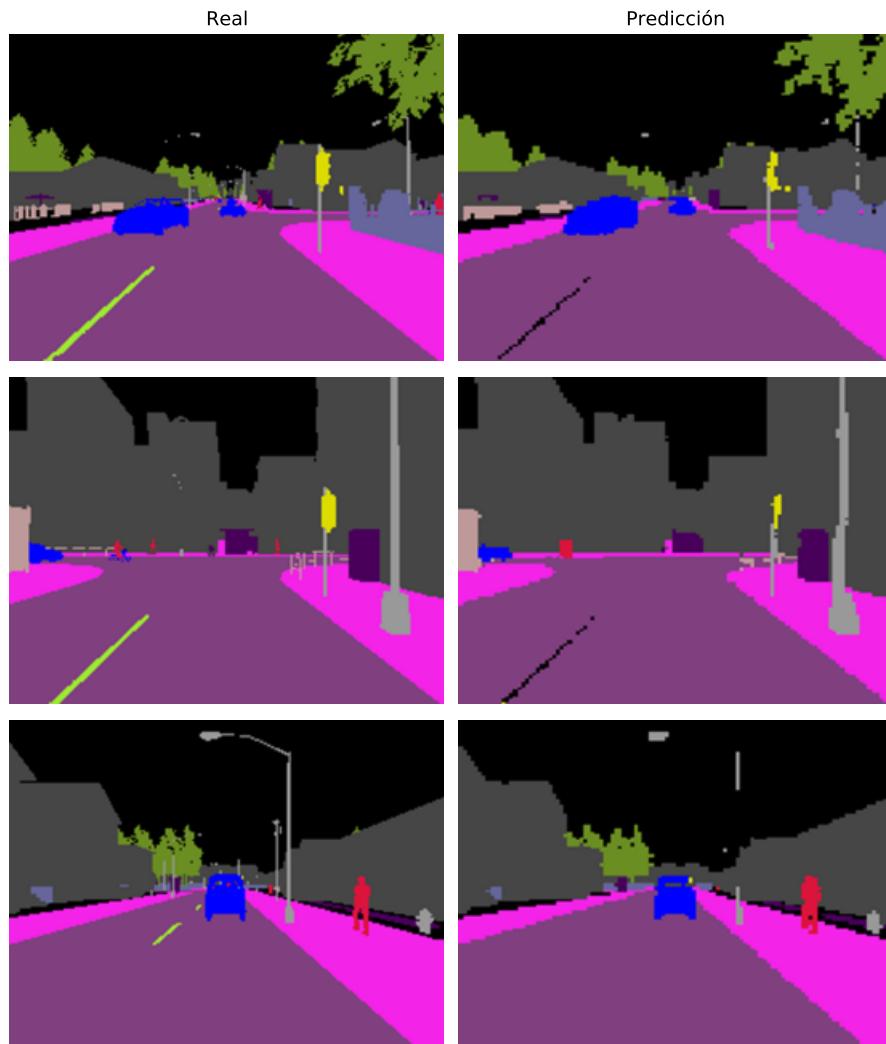


Figura 48: predicción vs valor esperado de segmentación

Por estos resultados se puede afirmar que la detección de vehículos es la más exacta dando lugar a muy pocos casos en que no se los detecte para una parada efectiva. Por otra parte al detectar Peatones, Postes y Señales de tránsito en general, se tiene alta exhaustividad, es

decir que si bien se detecta una cantidad considerable de píxeles de la clase, tiende a detectar menos píxeles de los que realmente componen al objeto (como se observa en la figura 48), sin embargo debido a la alta precisión, estos píxeles detectados tienen una alta confianza de ser de la clase predicha, en otras palabras, el modelo discrimina de manera agresiva, de forma que pocos valores pasan el umbral de clasificación, pero los que pasan tienen más seguridad de ser correctos.

Una de las consecuencias de la alta exhaustividad, que se detallará en las pruebas del simulador, es que pueden no detectarse postes a tiempo dando lugar a colisiones.

4.1.3.3. CAJAS DELIMITADORAS

Finalmente, haciendo uso de las máscaras de segmentación por clase, se pueden visualizar las cajas delimitadoras de los objetos, haciendo uso del flood fill, para comprobar las predicciones gráficamente mediante un código de color para cada clase, celeste para vehículos, naranja para peatones, rosado para postes y rojo o verde para semáforos dependiendo de su estado.

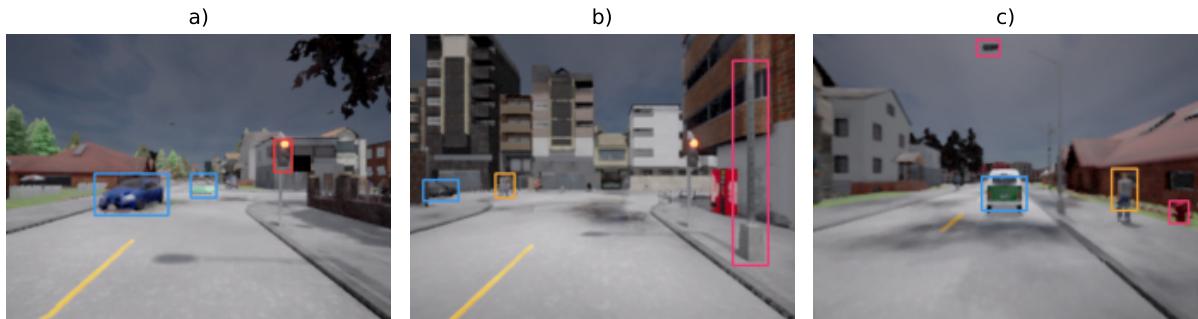


Figura 49: cajas delimitadoras

en el caso de la figura 49, se observa que en la imagen b, no se detectó el semáforo, debido a que el área de la segmentación es muy pequeña para ser considerada una clasificación válida, de manera similar en la imagen c, sólo se detecta la parte de la luz del poste.

4.1.3.4. ÍNDICE JACCARD

Aplicando el índice de Jaccard, conocido como Intersection Over Union (IoU), otra métrica útil en la segmentación semántica, se obtiene la siguiente tabla de porcentajes:

Clase	IoU
Edificios	90.88 %
Cercas	69.54 %
Peatones	45.72 %
Postes	31.93 %
Carriles	03.52 %
Caminos	97.55 %
Aceras	91.01 %
Vegetación	77.76 %
Vehículos	72.46 %
Paredes	57.26 %
Señales	37.25 %

Tabla 7: Índice de Jaccard de la segmentación semántica

si se interpretan los valores como el porcentaje de superposición de la máscara real y la máscara predicha para cada clase, se observa que de las cuatro clases que más nos interesan los postes y señales de tránsito tienen los peores valores debido a la alta discriminación de píxeles de la red neuronal, mientras que los peatones y vehículos tienen mayor porcentaje, útil para evadirlos como obstáculos en la vía.

4.1.4. DETECCIÓN Y CLASIFICACIÓN DE SEMÁFOROS

Debido a que el segmentador no discierne entre semáforos y las demás señales de tránsito, la clasificación de si lo es o no, y su color depende del módulo basado en K-means, evaluando la exactitud de este componente, se obtuvo que clasifica correctamente semáforos verdes en un 98.65 % de entre 1848 pruebas, y rojos en un 99.52 % de exactitud de 2102 imágenes. El 1.35 % de error en el color verde se da cuando se clasifican erroneamente semáforos amarillos que para la tarea se consideran igual a los rojos. Por otra parte el 0.48 % de error en semáforos rojo se debe al porcentaje de falsos negativos, es decir, la proporción de rojos reales clasificados como otro tipo de señal de tránsito. Así la exactitud total de clasificación de los colores es de 99.08 %.

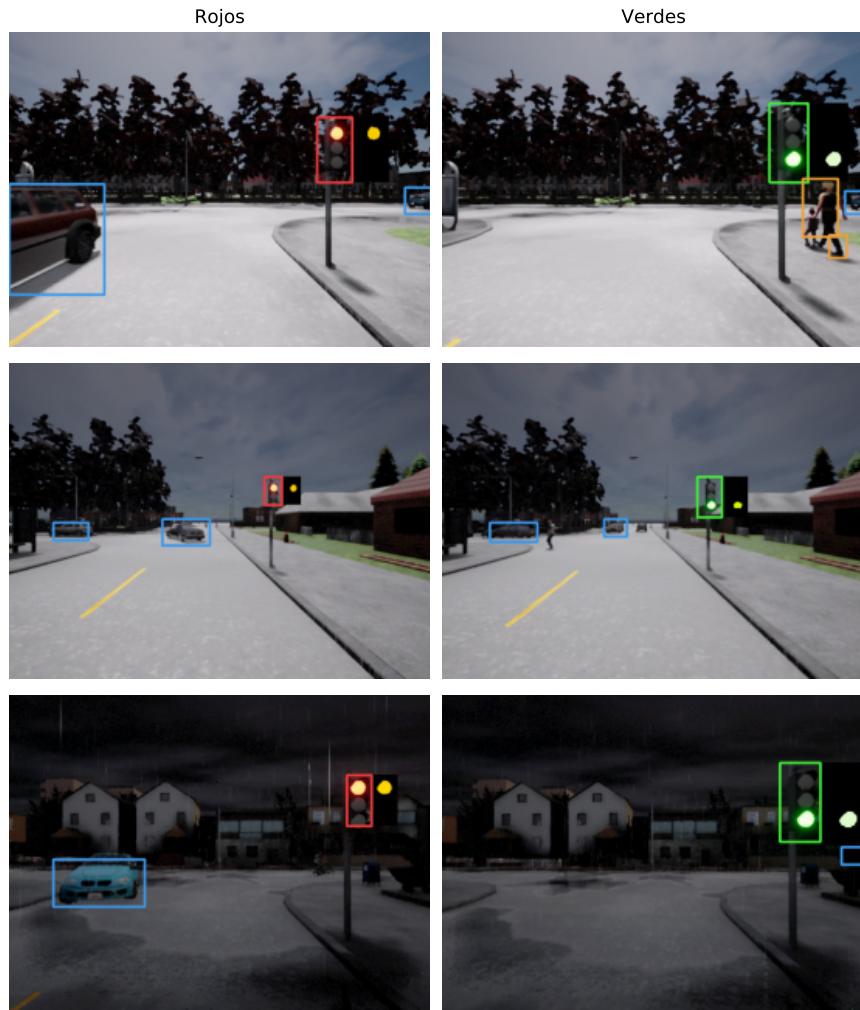


Figura 50: cuantización y clasificación del color

Una vez calculado el error en la clasificación del color se puede visualizar en la figura 50 las predicciones del módulo. El color de la caja delimitadora indica la predicción, y a su lado se muestra la cuantización realizada por K-Means sobre la que se aplican los umbrales de clasificación.

4.2. REPRESENTACIONES APRENDIDAS

4.2.1. DISTANCIAS EN LA REGIÓN DE INTERÉS

Al combinar la máscara de segmentación de posibles obstáculos con el mapa de profundidades se simplifica la tarea de calcular la distancia a estos, además al considerar sólo el trapezo

delimitando el área de circulación denominada región de interés, se reducen las falsas detecciones, como se puede observar en la figura 51



Figura 51: distancia a objetos en el área de interés

en la primera fila se tienen las detecciones generales usando las máscaras de segmentación, en la segunda los mapas de profundidad, y en la tercera la distancia estimada solamente a los posibles obstáculos, esta distancia general se calcula mediante la moda del valor de los píxeles del mapa de profundidades, así en cuanto un objeto está cerca, la mayoría de sus píxeles tendrán un valor bajo, siendo el límite de 4 metros a partir de la cual se empieza a frenar.

4.2.2. VISUALIZACIÓN DE ZONAS DE INTERÉS

Con el fin de comprender gráficamente a qué partes de la imagen la red neuronal de conducción presta atención durante la predicción del giro, se aplica el algoritmo scorecam, del cual se obtiene un mapa de calor que une a la imagen original, y cuyos valores altos indican mayor atención en la región.

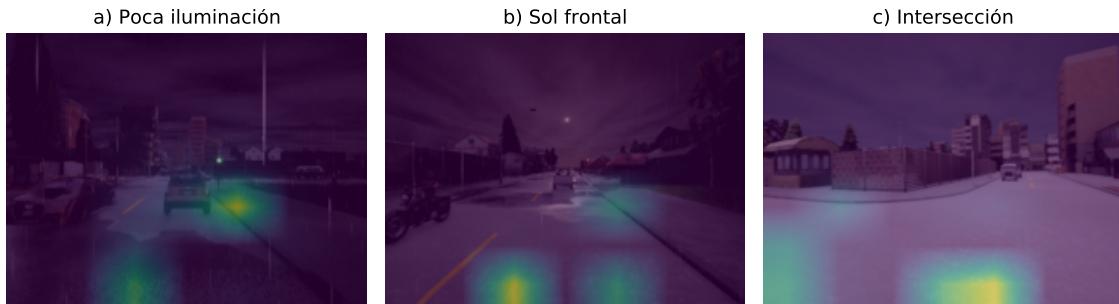


Figura 52: regiones de interés en la inferencia de dirección

En la figura 52 se observan en los casos *a* y *b* que aparte de la atención en el asfalto, debido a que se comparten los filtros para la inferencia de aceleración, se fija en la acera derecha, mientras que en la imagen *c*, al no tener una acera en el lado esperado se centra en la que está al frente, en la nueva calle a incorporarse.

4.3. PRUEBAS EN SIMULADOR

Una vez se tiene la certeza del funcionamiento del modelo mediante el análisis de sus métricas de error, se comprueban los resultados empíricamente con simulaciones en el ambiente virtual de la ciudad de CARLA.

4.3.1. CONDUCCIÓN

Ya se revisaron los resultados de segmentación, estimación de distancias y clasificación de objetos en la imagen, luego de pasar por todas las etapas descritas del modelo, las salidas obtenidas son el valor del giro y la aceleración.



Figura 53: comparación de dirección real y estimada

Comparando las predicciones en distintas situaciones se tienen los resultados de la figura 53, de la cual se puede observar en los casos *a, b, c, h, i*, los cuales consisten en giros durante intersecciones, que el modelo es mucho más agresivo al tomar las curvas, aplicando un giro mayor de la dirección. En el caso *f* que es una intersección en la cual se decide seguir recto, predice correctamente no girar. En la situación de la imagen *g*, se tiene una casi colisión con otro vehículo, en cuyo caso la predicción es 0, al igual que la aceleración, y finalmente en las imágenes *d* y *e* no existe mucha variación del 0 ya que se encuentra en una recta dentro de su carril.

4.3.2. FALLOS

A pesar del buen rendimiento del modelo en las situaciones de prueba, siempre existe un margen de error contemplado en las métricas evaluadas, así se presentan situaciones puntuales donde el vehículo se detiene abruptamente al confundir charcos de agua en la vía con vehículos cercanos (figura 54), o los más graves cuando se sale del camino y colisiona con objetos en las aceras 55.

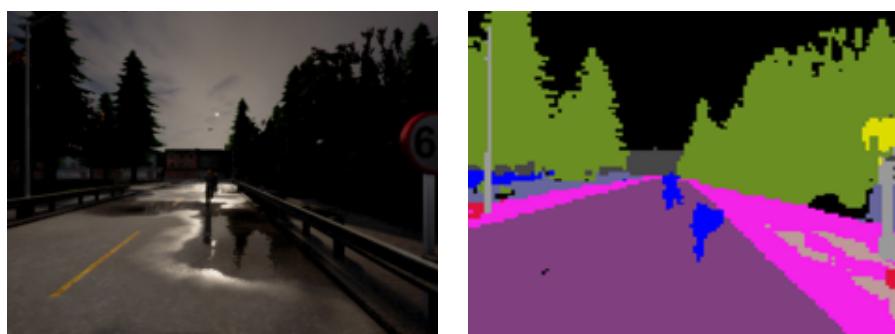
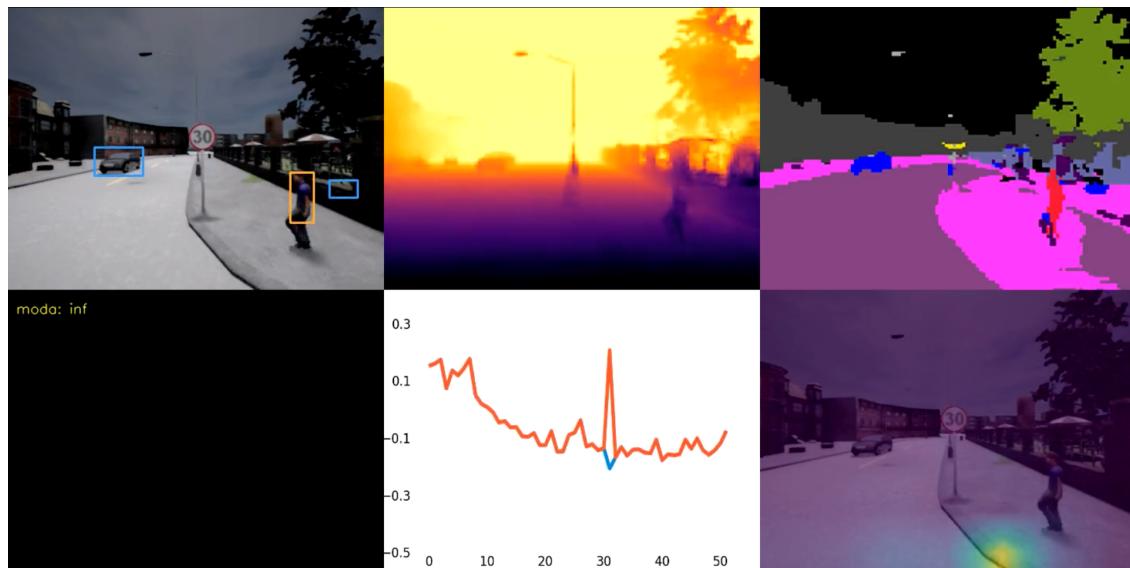


Figura 54: Charco erroneamente Clasificado

En la figura 55 se nota que la segmentación falla al no detectar el poste como obstáculo lo cual deriva en que no se calcule la distancia a este y se produzca la colisión.



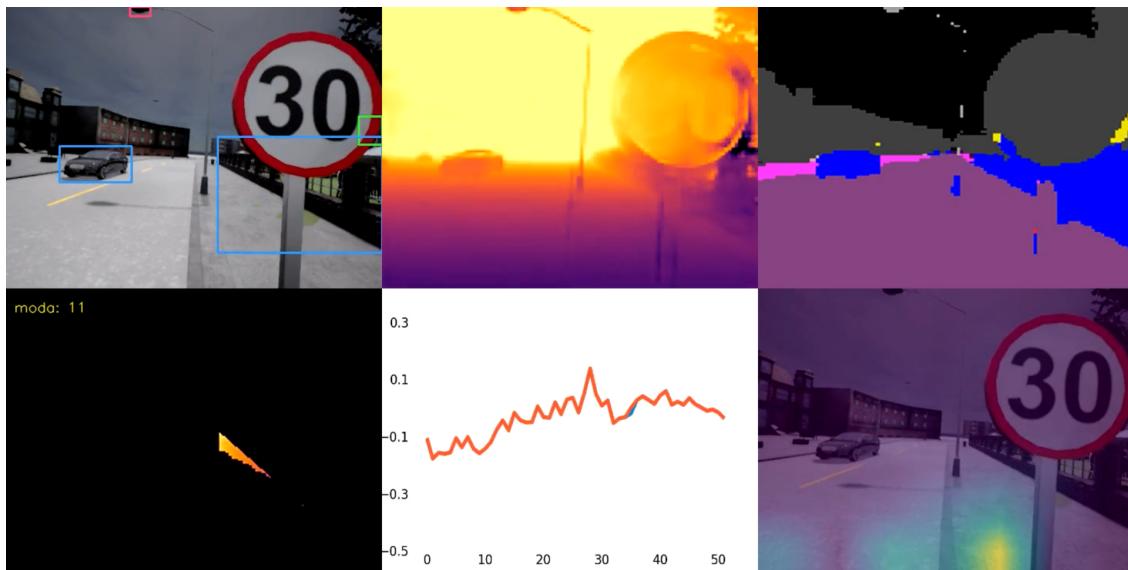


Figura 55: colisión con poste

En total se realizaron 17 simulaciones de prueba de las cuales en 3 se cometieron errores, los cuales fueron confundir un charco como vehículo, frenando totalmente y no reanudando la marcha, el de salirse del camino sin detectar un poste, colisionando así contra él, y el de invadir un carril en contra ruta frenando para no colisionar y congestionando el tráfico ya que no puede dar marcha atrás.

4.4. PRUEBA DE LA HIPÓTESIS

Para la prueba de la hipótesis “*El modelo de conducción autónoma mediante el uso de aprendizaje profundo y algoritmos de visión computacional alcanza una autonomía de nivel 2 en vías de doble sentido con separación física.*” se siguió los siguientes pasos.

1. Se evaluaron los porcentajes de exactitud de predicción de las máscaras de segmentación y estimación de distancias a partir de las salidas de la DepthNet y SemsegNet (modelos de aprendizaje profundo).
2. Se calculó el error de detección y clasificación del color de semáforos, aplicando Flood Fill para la detección de la caja delimitadora y K-Means para la clasificación del color (algoritmos de visión computacional).
3. Se evaluó el error de la predicción de aceleración y frenado, en base a las predicciones

conjuntas de las tres redes neuronales (DepthNet, SemsegNet y DriveNet) combinadas con la clasificación de los semáforos.

4. Se obtuvo el error de estimación de dirección tanto a la izquierda como derecha en base a la predicción de la DriveNet.

En base a los cálculos obtenidos se puede afirmar que:

- Con un porcentaje de exactitud de clasificación para la detección de obstáculos y semáforos de un 71.41 %, el error del control longitudinal es de ± 0.067 , considerando valores negativos como frenado y positivos como aceleración se obtiene un 6.7 % de error en el control longitudinal.
- El error de la dirección es de ± 0.069 lo que da lugar a un 6.9 % de error en el control lateral.
- De las 17 pruebas realizadas un 82 % son satisfactorias (3 pruebas con fallos).

Por lo tanto con un 93.2 % de confianza se puede afirmar que se obtiene una conducción autónoma de nivel 2 en el 82 % de las situaciones (17 pruebas).

CAPÍTULO 5. CONCLUSIONES Y RECOMENDACIONES

5.1. CONCLUSIONES

Finalmente se concluye para cada objetivo específico

- *Diseñar un componente de aumentación y preprocesamiento de datos para extraer y crear un dataset con el fin de resolver la tarea.*

Se diseñó un componente para, extraer imágenes y sus etiquetas de simulaciones, aumentar atributos con el fin de eliminar la ambigüedad en las intersecciones mediante una clasificación manual, y organizar las imágenes de cada simulación en su respectiva carpeta, indexado y etiquetado en un archivo csv.

- *Reducir la complejidad de implementación del modelo mediante el uso de solamente una cámara.*

Se redujo la complejidad de la implementación aplicando redes neuronales y algoritmos de visión artificial para inferir información extra a partir de una imagen RGB obtenida de una sola cámara, logrando así prescindir de sensores extra para la percepción del ambiente.

- *Modificar y entrenar redes neuronales con una alta exactitud en las predicciones utilizando menos requisitos de cómputo.*

Se modificaron y entrenaron las redes MobileNet y FastDepth para inferir la aceleración, dirección, profundidad y segmentación semántica, con menores requisitos computacionales al estar diseñadas para ejecutarse en dispositivos embedidos, obteniendo buenos resultados de predicción analizados en la sección 4.1.

- *Analizar las predicciones de los modelos entrenados para comprobar si las representaciones aprendidas son invariantes a los cambios de perspectiva, iluminación y objetos en la imagen.*

Se analizaron las predicciones fotograma por fotograma de simulaciones con diferentes condiciones climáticas, vehículos en distintas perspectivas y objetos de interés en dis-

tintas posiciones, se constató en las secciones 4.2 y 4.3 que los modelos y algoritmos con los parámetros elegidos son invariantes a estos cambios, además que se comprendió que la red de conducción basa su inferencia en la posición de la acera derecha en la imagen.

- *Combinar las salidas de algoritmos de visión computacional y modelos de aprendizaje profundo para mejorar la generalización de predicciones.*

Se combinaron las salidas de las redes neuronales de profundidad y segmentación semántica con las operaciones de dilatación, erosión y apertura, reduciendo así estimaciones residuales, además de la clasificación de semáforos por cuantización de colores para complementar las predicciones de aceleración por la red de conducción, logrando así generalizar las inferencias del acelerador y dirección dada la imagen de entrada.

- *Probar el rendimiento del modelo en una simulación, analizando casos de fallas y qué situaciones puede manejar correctamente.*

Se comprobó mediante simulaciones que en general el vehículo puede controlar la dirección de manera efectiva y frenar ante obstáculos, cometiendo errores en situaciones puntuales, como ser el fallo de clasificación de charcos de agua como vehículos causando su detención y la no detección de postes, ocasionando colisiones.

Así se cumple el objetivo de lograr la autonomía básica en vías de doble sentido con separación física, ya que realiza el control longitudinal y lateral, al igual que frena al detectar obstáculos cerca, pero requiere la intervención de una persona que esté alerta en caso de fallas en la prevención de colisión y durante salidas del camino.

En base a las pruebas realizadas en el simulador y las métricas obtenidas, no se puede rechazar la hipótesis, por lo tanto se acepta el modelo de conducción autónoma mediante el uso de aprendizaje profundo y algoritmos de visión computacional alcanza una autonomía de nivel 2 en vías de doble sentido con separación física.

5.2. RECOMENDACIONES

Una vez se tienen las conclusiones del proyecto conociendo sus límites y alcances, se plantean recomendaciones para continuar con la línea de investigación

- Diseñar una red sola red capaz de realizar la tarea de las tres redes presentadas en este trabajo, teniendo así múltiples salidas, para esto se requiere una arquitectura más compleja y redefinir la función de costo a optimizar.
- Extender el conjunto de datos de entrenamiento a otros mapas con distintos tipos de vías que cuenten con múltiples carriles e intersecciones de más de tres caminos, aumentando también la resolución de las imágenes.
- Incrementar el uso a más cámaras para lograr una percepción completa del ambiente, reduciendo así los puntos muertos y mejorando la seguridad al tener más información para la toma de decisiones.
- Estudiar la aplicabilidad de un modelo de percepción similar, adaptado para pedestres y los objetos que se puede encontrar en la vía pública, con el fin de ofrecer un sentido de la vista artificial a personas con discapacidades visuales.

CAPÍTULO 6. BIBLIOGRAFÍA

- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag.
- Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C. R. H. & Wirth, R. (2000). CRISP-DM 1.0: Step-by-step data mining guide.
- Dalal, N. & Triggs, B. (2005). Histograms of oriented gradients for human detection. En *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* (Vol. 1, 886-893 vol. 1).
- Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. & Koltun, V. (2017). CARLA: An Open Urban Driving Simulator. En *Proceedings of the 1st Annual Conference on Robot Learning* (pp. 1-16).
- Gonzalez, R. C. & Woods, R. E. (2018). *Digital Image Processing* (4th). USA: Addison-Wesley Longman Publishing Co., Inc.
- Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press.
- Granath, E. (2020). LiDAR systems: costs, integration, and major manufacturers.
- Gujarati, D. (2003). *Basic Econometrics*. Economic series. McGraw Hill.
- Halim, S. & Halim, F. (2013). *Competitive Programming 3: The New Lower Bound of Programming Contests*. Lulu.com.
- Hastie, T., Tibshirani, R. & Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc.
- He, K., Gkioxari, G., Dollár, P. & Girshick, R. (2017). Mask R-CNN. En *2017 IEEE International Conference on Computer Vision (ICCV)* (pp. 2980-2988).
- Hooper, J. (2004). From Darpa Grand Challenge 2004DARPA's Debacle in the Desert. *Populär Science*.

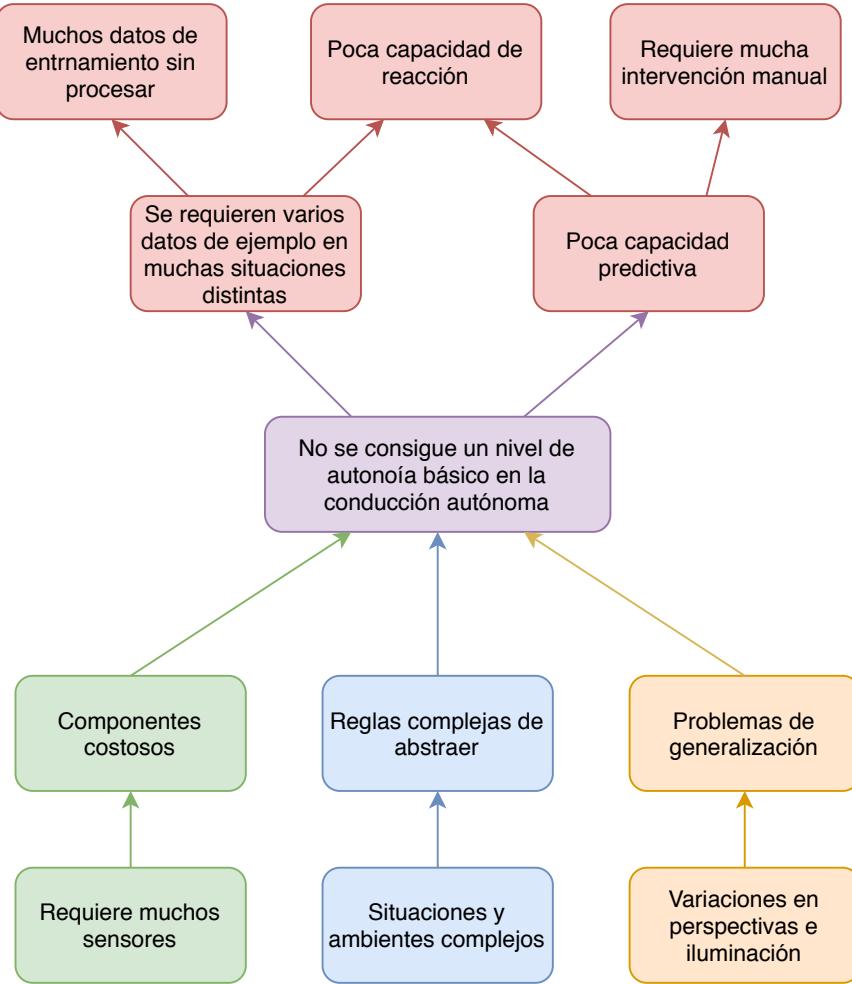
- Karpathy, A. (2020). *AI for Full-Self Driving*. SCALEDML CONFERENCE.
- Klette, R. (2014). *Concise Computer Vision: An Introduction into Theory and Algorithms*. Springer Publishing Company, Incorporated.
- Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. En *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (pp. 1097-1105). NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc.
- Lam, S. K., Pitrou, A. & Seibert, S. (2015). Numba: A LLVM-Based Python JIT Compiler. En *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery.
- Lecun, Y., Cosatto, E., Ben, J., Muller, U. & Flepp, B. (2004). *DAVE: Autonomous Off-Road Vehicle Control Using End-to-End Learning* (inf. téc. N.º DARPA-IPTO Final Report). Courant Institute/CBLL. <http://www.cs.nyu.edu/~yann/research/dave/index.html>.
- LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. En *Proceedings of the IEEE* (Vol. 86, 11, pp. 2278-2324).
- Lin, T., Goyal, P., Girshick, R., He, K. & Dollár, P. ([2017] 2020). Focal Loss for Dense Object Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(2), 318-327.
- List of self-driving car fatalities. (2020). Wikimedia Foundation.
- Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y. & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. En B. Leibe, J. Matas, N. Sebe & M. Welling (Eds.), *Computer Vision – ECCV 2016* (pp. 21-37). Cham: Springer International Publishing.
- Markoff, J. (2010). Google Cars Drive Themselves, in Traffic. *The New York Times*.
- Mohri, M., Rostamizadeh, A. & Talwalkar, A. (2018). *Foundations of Machine Learning* (2nd). The MIT Press.

- Montabone, S. (2012). Using OpenCV in Sage.
- Nelson, G. (2015). Tesla beams down 'autopilot' mode to Model S. *Automotive News*.
- Redmon, J., Divvala, S. K., Girshick, R. B. & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779-788.
- Roberts, L. G. (1963). *Machine perception of three-dimensional solids* (Tesis doctoral, Massachusetts Institute of Technology).
- Ronneberger, O., Fischer, P. & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. (Vol. 9351, pp. 234-241).
- Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536.
- SAE. (2018). *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*.
- Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. & Chen, L. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. En *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4510-4520).
- Stanford. (2020). CS231n Lecture Notes. Stanford.
- Szeliski, R. (2010). *Computer Vision: Algorithms and Applications* (1st). Berlin, Heidelberg: Springer-Verlag.
- Viola, P. & Jones, M. (2001). Rapid Object Detection using a Boosted Cascade of Simple Features. (Vol. 1, pp. I-511).
- Wang, H., Wang, Z., Du, M., Yang, F., Zhang, Z., Ding, S., ... Hu, X. (2020). Score-CAM: Score-Weighted Visual Explanations for Convolutional Neural Networks. En *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

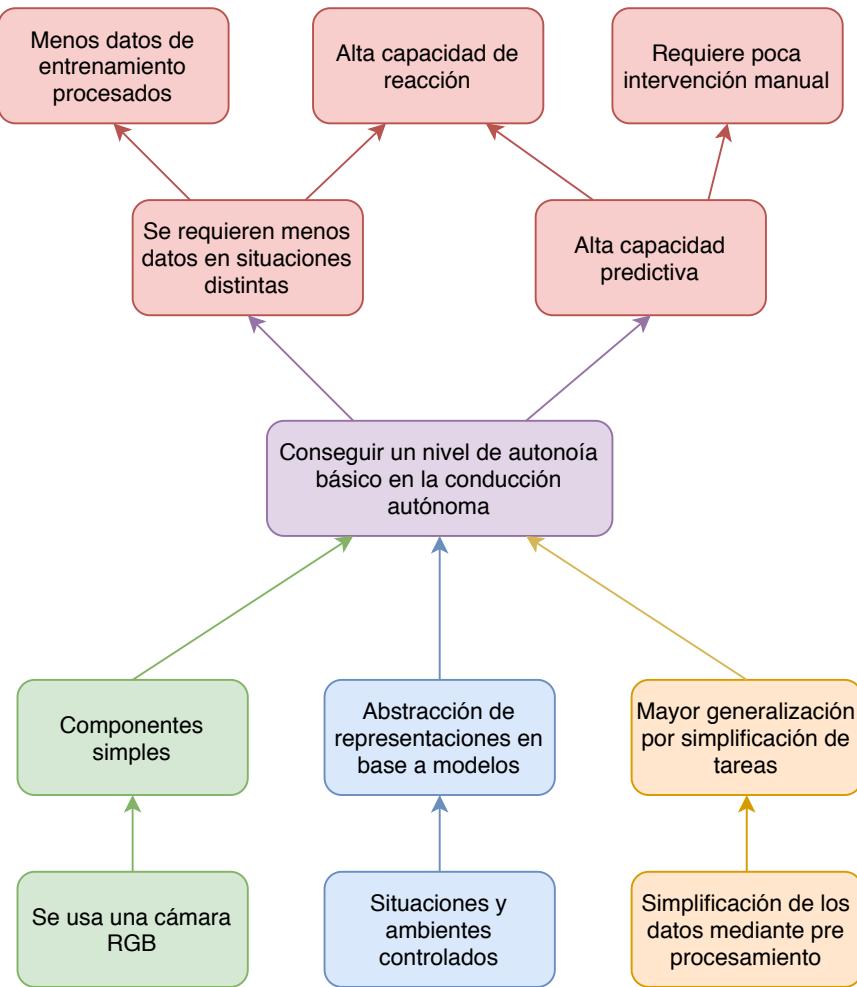
- Washington, T. (2016). *Closure for Finance*. Packt Publishing.
- Waymo. (2019). <https://waymo.com/open/>: Waymo Open Dataset.
- Wirth, R. (2000). CRISP-DM: Towards a standard process model for data mining. En *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining* (pp. 29-39).
- Wofk, Diana and Ma, Fangchang and Yang, Tien-Ju and Karaman, Sertac and Sze, Vivienne. (2019). FastDepth: Fast Monocular Depth Estimation on Embedded Systems. En *IEEE International Conference on Robotics and Automation (ICRA)*.
- Xuming He, Zemel, R. S. & Carreira-Perpinan, M. A. (2004). Multiscale conditional random fields for image labeling. En *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004*. (Vol. 2, pp. II-II).
- Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. (2020). *Dive into Deep Learning*. <https://d2l.ai>.

ANEXOS

ANEXO A - ÁRBOL DE PROBLEMAS



ANEXO B - ÁRBOL DE OBJETIVOS



ANEXO C - MARCO LÓGICO

Nivel	Resumen narrativo	Indicadores	Medios de verificación	Supuestos
Fin	Contribuir con el desarrollo de la conducción autónoma	Investigaciones derivadas que mejoren las propuestas	Tesis derivadas y comentarios sobre los resultados obtenidos	Es posible obtener resultados que apoyen en el avance de la conducción autónoma
Propósito	Plantear un modelo para la conducción autónoma que logre una autonomía básica en pistas y carreteras.	Rendimiento de los algoritmos y redes neuronales	Pruebas en un simulador	Los algoritmos y redes neuronales profundas permiten obtener buenas predicciones
Componentes	<p>a) Diseñar un componente de aumento y preprocesamiento de datos para extraer y crear un dataset con el fin de resolver la tarea.</p> <p>b) Reducir la complejidad de implementación del modelo mediante el uso de solamente una cámara.</p> <p>c) Modificar y entrenar redes neuronales con una alta exactitud en las predicciones utilizando menos requisitos de cómputo.</p> <p>d) Analizar las predicciones de los modelos entrenados para comprobar si las representaciones aprendidas son invariantes a los cambios de perspectiva, iluminación y objetos en la imagen.</p> <p>e) Combinar las salidas de algoritmos de visión computacional y modelos de aprendizaje profundo para mejorar la generalización de predicciones.</p> <p>f) Probar el rendimiento del modelo en una simulación, analizando casos de fallas y qué situaciones puede manejar correctamente.</p>	<p>a) los datos son útiles para el entrenamiento</p> <p>b) disminución de la complejidad</p> <p>c) % de exactitud en las métricas de predicciones</p> <p>d) invariancia en perspectiva e iluminación</p> <p>e) generalización en distintas situaciones de entrada</p> <p>f) % de fallos que causen accidentes en las simulaciones</p>	Métricas del error para medir el rendimiento de los modelos Visualización de las representaciones aprendidas	a) datos con información importante para la tarea b) es posible realizar la tarea con imágenes de una cámara c) las modificaciones permiten entrenar redes que funcionen en la tarea de predicción d) es posible aprender representaciones invariantes e) los algoritmos de visión computacional funcionan bien junto con modelos de aprendizaje profundo f) el modelo funciona bien en el ambiente virtual realista y comete pocos fallos
Actividades	<p>a) procesamiento de datos disponibles, y creación de un dataset.</p> <p>b) basar todos los modelos y algoritmos en imágenes de una cámara RGB</p> <p>c) adaptar redes neuronales que se saben que funcionan para esta tarea.</p> <p>d) recopilar un conjunto de datos procesados, entrenar distintas arquitecturas de redes neuronales hasta obtener buenos resultados.</p> <p>e) visualizar las áreas de atención del modelo, visualizar los filtros aprendidos, probar en casos extremos</p>	Obtención de resultados funcionales	Conducción básica en un simulador	El modelo permitirá lograr una conducción autónoma básica

ANEXO D - PREPARACIÓN DE DATOS

D.1 - EXTRACCIÓN DE IMÁGENES

```
import pickle
import torch
import torchvision.transforms as transforms
from torchvision.models import mobilenet_v2
import torch.nn as nn
import glob
import os
import sys

sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
    sys.version_info.major,
    sys.version_info.minor,
    'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
sys.path.append('../carla')
sys.path.append('path/to/DriveNet')
sys.path.append('path/to/DepthNet')
sys.path.append('path/to/SemsegNet')

from sync_mode import CarlaSyncMode # Clase incluida en los ejemplos de Carla
import carla

import pygame
import numpy as np
import re
import cv2

def should_quit():
    # función incluida con la API de Carla
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            return True
        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_ESCAPE:
                return True
    return False

def img_to_array(image):
    array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
    array = np.reshape(array, (image.height, image.width, 4))
    array = array[:, :, :3]
    return array

def show_window(surface, array, pos=(0,0)):
    if len(array.shape) > 2:
        array = array[:, :, ::-1]
    image_surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
```

```

surface.blit(image_surface, pos)

def create_camera(cam_type, vehicle, pos, h, w, lib, world):
    cam = lib.find(f'sensor.camera.{cam_type}')
    cam.set_attribute('image_size_x', str(w))
    cam.set_attribute('image_size_y', str(h))
    camera = world.spawn_actor(
        cam,
        pos,
        attach_to=vehicle,
        attachment_type=carla.AttachmentType.Rigid)
    return camera

def find_weather_presets():
    # función incluida con la API de Carla
    rgx = re.compile('.+?(?:(?<=[a-z])(?=|[A-Z])|(?<=[A-Z])(?=|[A-Z][a-z])|$)')

def name(x): return ' '.join(m.group(0) for m in rgx.finditer(x))

presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]

if __name__ == '__main__':
    try:
        actor_list = []
        base_path = 'path/to/dataset'
        # Número de simulación
        n_sim = 1

        # Crear las carpetas para las imágenes de la simulación
        os.makedirs(f'{base_path}/Images/{n_sim}/rgb/')
        os.makedirs(f'{base_path}/Images/{n_sim}/depth/')
        os.makedirs(f'{base_path}/Images/{n_sim}/mask/')

        pygame.init()
        w, h = 240, 180
        display = pygame.display.set_mode((w, h),
                                         pygame.HWSURFACE | pygame.DOUBLEBUF)
        clock = pygame.time.Clock()

        data = {'throttle': [], 'brake': [], 'steer': [], 'junction': []}
        try:
            client = carla.Client('localhost', 2000)
            client.set_timeout(2.0)

            world = client.get_world()
            player = spawn_player(world)
            actor_list.append(player)

            weather_index = 0

```

```

weather_presets = find_weather_presets()
preset = weather_presets[weather_index]
world.set_weather(preset[0])

blueprint_library = world.get_blueprint_library()

cam_pos = carla.Transform(carla.Location(x=1.6, z=1.7))

camera_rgb = create_camera(cam_type='rgb',
                           vehicle=player,
                           pos=cam_pos,
                           h=h, w=w,
                           lib=blueprint_library,
                           world=world)
actor_list.append(camera_rgb)

camera_semseg = create_camera(cam_type='semantic_segmentation',
                               vehicle=vehicle,
                               pos=cam_pos,
                               h=h, w=w,
                               lib=blueprint_library,
                               world=world)
actor_list.append(camera_semseg)

camera_depth = create_camera(cam_type='depth',
                             vehicle=vehicle,
                             pos=cam_pos,
                             h=h, w=w,
                             lib=blueprint_library,
                             world=world)
actor_list.append(camera_depth)

# Ceder el control del vehículo al Traffic Manager
player.set_autopilot(True)

with CarlaSyncMode(world, camera_rgb, camera_semseg, camera_depth,
                   fps=20) as sync_mode:
    frame = 0
    while True:
        if should_quit():
            return
        clock.tick()

# Obtener los fotogramas sincronos de las cámaras
snapshot, image_rgb, image_semseg, image_depth = sync_mode.tick(timeout=2.0)

# Convertir la imagen a un 2D array
rgb_arr = img_to_array(image_rgb)
# Visualizar la cámara RGB
show_window(display, rgb_arr)
pygame.display.flip()

```

```

c = player.get_control()

if c.throttle != 0:
    location = player.get_location()
    wp = world.get_map().get_waypoint(location)

    # Convertir las otras cámaras en arreglos BGR
    depth_arr = img_to_array(image_depth)
    mask_arr = img_to_array(image_semseg)

    # Guardar las imágenes como archivos
    cv2.imwrite(f'{base_path}/Images/{n_sim}/rgb/{frame}.png', rgb_arr)
    cv2.imwrite(f'{base_path}/Images/{n_sim}/mask/{frame}.png', mask_arr)

    # Crear una entrada de datos en el diccionario
    data['throttle'].append(min(c.throttle, 0.4))
    data['brake'].append(c.brake)
    data['steer'].append(c.steer)
    data['junction'].append(wp.is_junction)

frame += 1

if frame == 8000:
    # Si se llega a los 8000 fotogramas, terminar simulación
    return
if frame % 1000 == 0:
    print(f'Frame: {frame}')

finally:
    print('destroying actors.')
    for actor in actor_list:
        actor.destroy()
    world.destroy()
    pygame.quit()

    # Convertir el diccionario en un dataframe
    df = pd.DataFrame.from_dict(data)
    # Exportar el dataframe como csv para la simulación
    df.to_csv(f'{base_path}/Dfs/{n_sim}.csv', index=False)
    print('done.')

except KeyboardInterrupt:
    print('\nFin')

```

D.2 - UNIFICACIÓN DE LOS DATAFRAMES

```

import pandas as pd
import os
from pathlib import Path

# Listar los CSVs en orden de creación

```

```

def listdir_date(dirpath):
    return map(lambda p: str(p).split('/')[-1],
               sorted(Path(dirpath).iterdir(), key=os.path.getmtime))

if __name__ == '__main__':
    path = 'path/to/dfs'

    # Cargar los CSVs ordenados por fecha como una tupla(nombre, objeto)
    dfs = [(file, pd.read_csv(f'{path}/{file}')) for file in listdir_date(path)]

    for filename, df in dfs:
        # Insertar columna de los nombres de archivos y número de simulación
        # correspondiente a cada csv
        names = [f'{filename.split(".")[0]}/{i}.png' for i in range(len(df))]
        df['filenames'] = names

    # Concatenar CSVs en uno solo
    whole_dataset = pd.concat(list(map(lambda x: x[1], dfs)))

    # Guardar como un nuevo archivo
    whole_dataset.to_csv('path/to/train_dataset.csv', index=False)

```

D.3 - PREPARACIÓN DE INTERSECCIONES

```

import pandas as pd
import os
from shutil import copy
from tqdm import tqdm

if __name__ == '__main__':
    df = pd.read_csv('path/to/train_dataset.csv')

    # Crear columnas para la carpeta y el archivo
    df['folder'] = [f.split('/')[0] for f in df['filenames'].tolist()]
    df['file'] = [f.split('/')[1].split('.')[0] for f in df['filenames'].tolist()]

    # Se elimina la columna que contiene ambos en uno
    df = df.drop(['filenames'], axis=1)

    # Seleccionar las filas de intersecciones
    df_junc = df.query('junction == True')

    # Extraer la lista de indices
    idxs = df_junc.index.tolist()

    # Se buscan las intersecciones
    diffs = [0]*(len(idxs)-1)
    junc_idxs = []
    temp_idxs = []
    for i in range(len(idxs)-1):

```

```

# Si la diferencia de índices es grande
# son intersecciones distintas
diff = idxs[i+1] - idxs[i]

temp_idxs.append(idxs[i])
if diff != 1 and (idxs[i+1] != 7999 and idxs[i] != 0):
    junc_idxs.append(temp_idxs)
    temp_idxs = []

# Ubicación de las carpetas para cada intersección
base_path = 'path/to/junctions'
# Ubicación de las imágenes originales
origin_path = 'path/to/Images'

for i, junc_img_idxs in tqdm(enumerate(junc_idxs)):
    # Crear carpeta por intersección
    os.makedirs(f'{base_path}/{i}')
    for idx in junc_img_idxs:
        row = df.iloc[idx]
        folder, file = row['filenames'].split('/')

        # Copiar la imagen a su respectiva carpeta
        copy(f'{origin_path}/{folder}/rgb/{file}', f'{base_path}/{i}/{folder}_{file}')

```

D.4 - CREACIÓN DEL DATAFRAME FINAL

```

import pandas as pd
import os

if __name__ == '__main__':
    # CSV del conjunto de datos
    dataset = pd.read_csv('path/to/train_dataset.csv')
    # CSV de intersecciones etiquetadas
    junctions = pd.read_csv('path/to/juncs.csv')

    # Separar la columna filenames en una para carpetas y otra para nombre de archivos
    dataset['folder'] = [f.split('/')[0] for f in dataset['filenames'].tolist()]
    dataset['file'] = [f.split('/')[1].split('.')[0] for f in dataset['filenames'].tolist()]
    dataset = dataset.drop(['filenames'], axis=1)

    # Crear una lista vacía para cada etiqueta llena de ceros
    # (valores negativos en clasificación binaria)
    path_left = [0]*len(dataset)
    path_right = [0]*len(dataset)
    path_forward = [0]*len(dataset)

    action_left = [0]*len(dataset)
    action_right = [0]*len(dataset)
    action_forward = [0]*len(dataset)
    # Por defecto la "No acción" está marcada.

```

```

no_action = [1]*len(dataset)

# Ubicación de las imágenes de intersecciones
junc_path = 'path/to/junctions'

# Ordenar carpetas de intersección por el número de menor a mayor
juncs = sorted(os.listdir(junc_path), key=lambda x: int(x))

for junc in juncs:
    # Ordenar los archivos numéricamente
    files = sorted(os.listdir(f'{junc_path}/{junc}'),
                   key=lambda x: int(x.split('.')[0].split('_')[1]))

    # Indexar la fila correspondiente a la intersección
    row = junctions.iloc[int(junc)]
    for file in files:
        folder, img = list(map(int, file.split('.')[0].split('_')))
        idx = 8000*folder + img

        # Asignar los valores etiquetados
        path_left[idx] = row['path_left']
        path_right[idx] = row['path_right']
        path_forward[idx] = row['path_forward']

        action_left[idx] = row['action_left']
        action_right[idx] = row['action_right']
        action_forward[idx] = row['action_forward']
        # Como es una intersección se desmarca la "No acción"
        no_action[idx] = 0

```

ANEXO E - ENTRENAMIENTO DE REDES

E.1 - DRIVENET

```

import os
import time
import torch
import torch.nn as nn
import torch.optim
import torch.utils.data
import torchvision.transforms as transforms
from custom_mobilenet import CustomMobileNet

from PIL import Image
import numpy as np
import pandas as pd
from tqdm import tqdm

class DriveDataset(torch.utils.data.Dataset):
    def __init__(self, root_dir, transform=None):

```

```

temp = root_dir.split('/')
self.root_dir = '/'.join(temp[:-1])
self.transform = transform

self.data = pd.read_csv(root_dir)

def __len__(self):
    return len(self.data)

def __getitem__(self, idx):
    if torch.is_tensor(idx):
        idx = idx.tolist()

    row = self.data.iloc[idx]

    # Cargar imagen RGB
    folder, file = row['filenames'].split('/')
    img_rgb_path = os.path.join(
        self.root_dir, 'Images', folder, 'rgb', file)
    img_rgb = Image.open(img_rgb_path)

    if self.transform:
        img_rgb = self.transform(img_rgb)

    # leer etiquetas de la i-ésima muestra
    throttle = row['throttle']
    steering = row['steer']
    action_left = row['action_left']
    action_right = row['action_right']
    action_forward = row['action_forward']
    no_action = row['no_action']

    # Crear muestra
    sample = (img_rgb,
              # Tensor de parámetros de acción.
              torch.tensor([
                  float(action_left), float(action_right),
                  float(action_forward), float(no_action)
              ]),
              # Tensor de etiquetas de aceleración y giro.
              torch.tensor([
                  float(throttle), float(steering)
              ]))

    return sample

if __name__ == '__main__':
    model = CustomMobileNet(pretrained=True)

    model.cuda()

```

```

train_dir = 'path/to/train_dataset_final.csv'
val_dir = 'path/to/val_dataset_final.csv'

train_loader = torch.utils.data.DataLoader(
    dataset=DriveDataset(train_dir, transforms.Compose([
        transforms.Resize((224, 224), interpolation=Image.BICUBIC),
        transforms.ToTensor(),
        lambda T: T[:3]
    ])),
    batch_size=64,
    shuffle=True,
    num_workers=12,
    pin_memory=True
)

val_loader = torch.utils.data.DataLoader(
    dataset=DriveDataset(val_dir, transforms.Compose([
        transforms.Resize((224, 224), interpolation=Image.BICUBIC),
        transforms.ToTensor()
    ])),
    batch_size=64,
    shuffle=False,
    num_workers=12,
    pin_memory=True
)

criterion = nn.MSELoss().cuda()
optimizer = torch.optim.Adam(model.parameters())

losses = []

# Iterar para entrenar la red
for epoch in range(50):
    start = time.time()

    model.train()
    train_loss = 0
    # Definir barra de progreso interactiva
    train_progress = tqdm(enumerate(train_loader),
                          desc="train",
                          total=len(train_loader))

    # Iterar por cada minibatch de 64 muestras
    for i, (X, actions, y) in train_progress:
        # Copiar los datos a la GPU
        X = X.cuda(non_blocking=True)
        actions = actions.cuda(non_blocking=True)
        y = y.cuda(non_blocking=True)
        y_hat = model(X, actions)

        # Calcular el error cuadrático medio para la aceleración
        loss1 = criterion(y_hat[:, 0], y[:, 0])

```

```

# Calcular el error cuadrático medio para la dirección
loss2 = criterion(torch.tanh(y_hat[:, 1]), y[:, 1])
# Combinar ambos errores
loss = (loss1 + loss2)/2

# Paso de optimización
optimizer.zero_grad()
loss.backward()
optimizer.step()

train_loss += float(loss.detach())
train_progress.set_postfix(loss=(train_loss/(i+1)))

model.eval()

val_loss = 0
with torch.no_grad():
    model.eval()
    val_progress = tqdm(enumerate(val_loader),
                        desc="val",
                        total=len(val_loader))
    for i, (X, actions, y) in val_progress:
        X = X.cuda(non_blocking=True)
        actions = actions.cuda(non_blocking=True)
        y = y.cuda(non_blocking=True)
        y_hat = model(X, actions)

        loss1 = criterion(y_hat[:, 0], y[:, 0])
        loss2 = criterion(y_hat[:, 1], y[:, 1])
        loss = (loss1 + loss2) / 2

        val_loss += float(loss)
        val_progress.set_postfix(loss=(val_loss/(i+1)))

end = time.time()

t_loss = train_loss / len(train_loader)
v_loss = val_loss / len(val_loader)
print('epoch:', epoch, 'L:', t_loss, v_loss, 'Time:', end-start)

torch.save(
{
    'epoch': epoch,
    'arch': 'mobilenet_custom',
    'state_dict': model.state_dict()
},
f'weights/mob_drive_{epoch}.pth.tar')
losses.append([epoch, t_loss, v_loss])
np.save('hist_drive', np.array(losses))

```

E.2 - DEPTHNET

```
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim
import torch.utils.data
import torchvision.transforms as transforms
from models import CustomMobilenet

from PIL import Image, ImageOps
import os
import numpy as np
import pandas as pd
import time

from tqdm import tqdm

import matplotlib.pyplot as plt

# Clase encargada de cargar los datos
class DriveDepthDataset(torch.utils.data.Dataset):
    def __init__(self, root_dir, transform=None, stills=True):
        temp = root_dir.split('/')
        self.root_dir = '/'.join(temp[:-1])
        self.transform = transform

        self.data = pd.read_csv(root_dir)
        # Si se desean mantener las imágenes donde el vehículo está quieto
        if stills:
            self.data = self.data['n_id'].tolist()
        else:
            self.data = self.data.query('throttle != 0.0')['n_id'].tolist()

    def __len__(self):
        return len(self.data)

    # Sobrecarga del operador [] para indexado
    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        # Decidir si espejar la imagen aleatoriamente
        h_flip = np.random.random() < 0.5

        # Cargar imagen RGB
        img_rgb_path = os.path.join(self.root_dir, 'rgb', f'{self.data[idx]}.png')
        img_rgb = Image.open(img_rgb_path)
        if h_flip:
            img_rgb = ImageOps.mirror(img_rgb)
        if self.transform:
```

```

    img_rgb = self.transform(img_rgb)

    # Cargar imagen de profundidad
    img_depth_path = os.path.join(self.root_dir, 'depth', f'{self.data[idx]}.png')
    img_depth = Image.open(img_depth_path)
    if h_flip:
        img_depth = ImageOps.mirror(img_depth)

    # Convertir la imagen de 24 bits a un mapa de profundidades [0, 1000]
    img_depth = np.transpose(np.asarray(img_depth, dtype=np.float32), (2, 0, 1))
    target = img_depth[0, :, :] + img_depth[1, :, :] * 256 + img_depth[2, :, :] * 256 * 256

    # Truncar las distancias hasta 30 metros como máximo
    target = np.clip((target / (256 * 256 * 256 - 1)) * 1000, None, 30)
    target = torch.from_numpy(target).float()

    # Crear tupla de la muestra
    sample = (img_rgb, target.view(1, target.shape[0], target.shape[1]))

    return sample

if __name__ == '__main__':
    # Valores de normalización de la imagen
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                    std=[0.229, 0.224, 0.225])
    # Instanciar la red de profundidades
    model = CustomMobileNetDepth((180, 240), pretrained=False)
    # Enviar parámetros a la GPU
    model.cuda()

    train_dir = 'path/to/train_data.csv'
    val_dir = 'path/to/val_data.csv'

    # Instanciar el DataLoader para el conjunto de entrenamiento
    train_loader = torch.utils.data.DataLoader(
        dataset=DriveDepthDataset(train_dir, transforms.Compose([
            transforms.ToTensor(),
            lambda T: T[:3],
            normalize
        ])),
        batch_size=64,
        shuffle=True,
        num_workers=12,
        pin_memory=True
    )

    # Instanciar el DataLoader para el conjunto de validación
    val_loader = torch.utils.data.DataLoader(
        dataset=DriveDepthDataset(val_dir, transforms.Compose([
            transforms.ToTensor(),
            normalize
        ])),

```

```

batch_size=64,
shuffle=True,
num_workers=12,
pin_memory=True
)

# Definir función de costo a optimizar
criterion = nn.MSELoss().cuda()
# Definir optimizador
optimizer = torch.optim.Adam(model.parameters())

# Historial de errores
losses = []

# Entrenar por un número de iteraciones o hasta detener el proceso
for epoch in range(50):
    # Guardar tiempo de inicio de iteración
    start = time.time()

    # Modelo en modo entrenamiento
    model.train()
    train_loss = 0
    # Iterar por los bloques de datos de 64 en 64
    for i, (X, y) in tqdm(enumerate(train_loader)):
        X = X.cuda(non_blocking=True)
        y = y.cuda(non_blocking=True)
        # Realizar la predicción
        y_hat = model(X)

        # Calcular el error
        loss = criterion(y_hat, y)
        # Acumular el error
        train_loss += float(loss.detach())

    # Derivar y actualizar los parámetros
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Evaluar la red en el conjunto de validación cruzada
    val_loss = 0
    with torch.no_grad():
        model.eval()
        for i, (X, y) in enumerate(val_loader):
            X = X.cuda(non_blocking=True)
            y = y.cuda(non_blocking=True)
            y_hat = model(X)

            loss = criterion(y_hat, y)
            val_loss += float(loss)

    # Guardar el tiempo de finalización de la iteración

```

```

end = time.time()

# Calcular el error medio de la iteración
t_loss = train_loss / len(train_loader)
v_loss = val_loss / len(val_loader)
print('epoch:', epoch, 'L:', t_loss, v_loss, 'Time:', end - start)

# Guardar los parámetros de la i-ésima iteración
torch.save(
{
    'epoch': epoch,
    'arch': 'mobilenet_depth',
    'state_dict': model.state_dict()
},
f'weights/c_mob_{epoch}.pth.tar')

# Guardar los errores
losses.append([epoch, t_loss, v_loss])
np.save('hist', np.array(losses))

```

E.3 - SEMSEGNET

```

import torch
import torch.nn as nn
import torch.optim
import torch.utils.data
import torchvision.transforms as transforms
from semseg_model import CustomMobileNetSemseg

from PIL import Image, ImageOps
import os
import numpy as np
import pandas as pd
import time

from tqdm import tqdm

class DriveSemsegDataset(torch.utils.data.Dataset):
    def __init__(self, root_dir, transform=None, stills=True, train=True):
        temp = root_dir.split('/')
        self.root_dir = '/'.join(temp[:-1])
        self.transform = transform

        self.data = pd.read_csv(root_dir)
        if stills:
            self.data = self.data['filenames'].tolist()
        else:
            self.data = self.data.query('throttle != 0.0')
            self.data = self.data['filenames'].tolist()

```

```

def __len__(self):
    return len(self.data)

def __getitem__(self, idx):
    if torch.is_tensor(idx):
        idx = idx.tolist()

    h_flip = np.random.random() < 0.5

    # Imagen RGB
    folder, file = self.data[idx].split('/')
    img_rgb_path = os.path.join(self.root_dir, 'Images', folder, 'rgb', file)
    img_rgb = Image.open(img_rgb_path)
    if h_flip:
        img_rgb = ImageOps.mirror(img_rgb)
    if self.transform:
        img_rgb = self.transform(img_rgb)

    # Máscara de segmentación
    img_semseg_path = os.path.join(self.root_dir, 'Images', folder, 'mask', file)
    img_semseg = Image.open(img_semseg_path)
    if h_flip:
        img_semseg = ImageOps.mirror(img_semseg)
    # Se extrae el canal R de la imagen en el que está codificada la máscara
    img_semseg = np.asarray(img_semseg)[:, :, 0].copy()
    sample = (img_rgb, torch.from_numpy(img_semseg))

    return sample

if __name__ == '__main__':
    # Se fija una semilla para hacer el experimento reproducible
    np.random.seed(42)

    model = CustomMobileNetSemseg((180, 240), pretrained=False)
    model = model.cuda()

    train_dir = 'path/to/train_dataset.csv'
    val_dir = 'path/to/val_dataset.csv'

    train_loader = torch.utils.data.DataLoader(
        dataset=DriveSemsegDataset(train_dir, transforms.Compose([
            transforms.ToTensor(),
            lambda T: T[:3],
        ]), stills=False),
        batch_size=64,
        shuffle=True,
        num_workers=12,
        pin_memory=True
    )

    val_loader = torch.utils.data.DataLoader(

```

```

dataset=DriveSemsegDataset(val_dir, transforms.Compose([
    transforms.ToTensor(),
]), stills=False, train=False),
batch_size=64,
shuffle=True,
num_workers=12,
pin_memory=True
)

# Entropía cruzada como criterio de erro (Log loss)
criterion = nn.CrossEntropyLoss().cuda()
# Optimizador Adam con un learning rate de 0.01
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, amsgrad=True)

losses = []

train_len = len(train_loader)
val_len = len(val_loader)
for epoch in range(50):
    start = time.time()

    model.train()
    train_loss = 0

    train_progress = tqdm(enumerate(train_loader), desc="train", total=train_len)
    for i, (X, y) in train_progress:
        X = X.cuda(non_blocking=True)
        y = y.cuda(non_blocking=True)
        y_hat = model(X)

        # Se calcula el error softmax 2D
        loss = criterion(y_hat, y.long())

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        loss_val = loss.detach()
        train_loss += float(loss_val)

    train_progress.set_postfix(loss=(train_loss/(i+1)))

    torch.save({
        'epoch': epoch,
        'arch': 'mobilenet_depth',
        'state_dict': model.state_dict()
    },
    f'weights/s_mob_{epoch}.pth.tar')

    val_loss = 0
    with torch.no_grad():
        model.eval()

```

```

val_progress = tqdm(enumerate(val_loader), desc="val", total=val_len)
for i, (X, y) in val_progress:
    X = X.cuda(non_blocking=True)
    y = y.cuda(non_blocking=True)
    y_hat = model(X)

    loss = criterion(y_hat, y.long())

    val_loss += float(loss)

    val_progress.set_postfix(loss=(val_loss/(i+1))) # Loss info

end = time.time()

t_loss = train_loss / len(train_loader)
v_loss = val_loss / len(val_loader)
print('epoch:', epoch, 'L:', t_loss, v_loss, 'Time:', end - start)

losses.append([epoch, t_loss, v_loss])
np.save('hist', np.array(losses))

```

ANEXO F - INFERENCIA

F.1 - COLA DE PRIORIDAD

```

from heapq import *
# Tipos de datos para anotar cada variable
from typing import List, Tuple, Any, Dict, Union, Optional

class PriorityQueue:
    def __init__(self):
        self.h = []

    def empty(self) -> bool:
        return False if self.h else True

    def push(self, x: Any) -> None:
        heappush(self.h, x)

    def pop(self) -> Any:
        return heappop(self.h)

    def top(self) -> Any:
        return self.h[0] if self.h else None

    def __len__(self) -> int:
        return len(self.h)

    def __str__(self) -> str:

```

```

res = ''
aux = PriorityQueue()

while not self.empty():
    e = self.pop()
    res += str(e) + '\n'
    aux.push(e)

self.h = aux.h
return res.strip()

```

F.2 - CONSTANTES DE IMPLEMENTACIÓN

```

junction_data = {
    (1, 1, 230): ['l', 'f'],
    (18, -1, 195): ['l', 'f'],
    (13, -1, 20): ['l', 'f'],
    (14, -1, 160): ['l', 'f'],
    (0, -1, 230): ['r', 'f'],
    (19, 1, 195): ['r', 'f'],
    (14, 1, 20): ['r', 'f'],
    (15, 1, 160): ['r', 'f'],
    (4, 1, 230): ['l', 'r'],
    (4, -1, 125): ['r', 'f'],
    (8, -1, 125): ['l', 'r'],
    (5, 1, 125): ['l', 'f'],
    (7, 1, 195): ['l', 'r'],
    (10, 1, 20): ['l', 'r'],
    (6, -1, 160): ['l', 'r'],
    (6, 1, 265): ['l', 'f'],
    (5, -1, 265): ['r', 'f'],
    (9, -1, 265): ['l', 'r'],
    (9, 1, 90): ['l', 'r'],
    (10, -1, 90): ['r', 'f'],
    (11, 1, 90): ['l', 'f'],
    (11, -1, 55): ['l', 'r'],
    (8, 1, 55): ['l', 'f'],
    (7, -1, 55): ['r', 'f']
}

contours = np.array([
    [0, 139], [0, 0],
    [240, 0], [240, 139],
    [191, 139], [135, 90],
    [110, 90], [51, 139],
    [0, 139], [0, 180],
    [240, 180], [240, 139], [0, 139]
])

directions = np.asarray([

```

```

(-1, -1),
(0, -1),
(1, -1),
(-1, 0),
(1, 0),
(-1, 1),
(0, 1),
(1, 1)
])

weather_idxs = [0, 1, 2, 3, 4, 9, 10, 11, 12, 13, 14]

```

F.3 - IMPLEMENTACIÓN DEL MODELO

```

import pickle
import torch
import torchvision.transforms as transforms
from torchvision.models import mobilenet_v2
import torch.nn as nn
import glob
import os
import sys

sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
    sys.version_info.major,
    sys.version_info.minor,
    'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
sys.path.append('../carla')
sys.path.append('path/to/DriveNet')
sys.path.append('path/to/DepthNet')
sys.path.append('path/to/SemsegNet')

from sync_mode import CarlaSyncMode
from custom_mobilenet import CustomMobileNet, CustomMobileNetExt
from models import CustomMobilenetDepth
from semseg_model import CustomMobilenetSemseg
from constants import junction_data, contours, directions, weather_idxs

import carla

from collections import Counter
import pygame
import numpy as np
import queue
import re
import math
import weakref
import collections
import cv2

```

```

from PIL import Image
from priority_queue import PriorityQueue

from numba import njit

def should_quit():
    # función incluida con la API de Carla
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            return True
        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_ESCAPE:
                return True
    return False

def find_weather_presets():
    # función incluida con la API de Carla
    rgx = re.compile('.+?(?:(?=<[a-z])(?==[A-Z])|(?<=[A-Z])(?==[A-Z][a-z])|$)')

    def name(x): return ' '.join(m.group(0) for m in rgx.finditer(x))

    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]

def img_to_array(image):
    array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
    array = np.reshape(array, (image.height, image.width, 4))
    array = array[:, :, :3]
    return array

def show_window(surface, array, pos=(0,0)):
    if len(array.shape) > 2:
        array = array[:, :, ::-1]
    image_surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
    surface.blit(image_surface, pos)

def create_camera(cam_type, vehicle, pos, h, w, lib, world):
    cam = lib.find(f'sensor.camera.{cam_type}')
    cam.set_attribute('image_size_x', str(w))
    cam.set_attribute('image_size_y', str(h))
    camera = world.spawn_actor(
        cam,
        pos,
        attach_to=vehicle,
        attachment_type=carla.AttachmentType.Rigid)
    return camera

def set_random_weather(world):

```

```

weather_presets = find_weather_presets()
weather_index = np.random.choice(weather_idxs)
preset = weather_presets[weather_index]
world.set_weather(preset[0])
def spawn_player(world):
    world_map = world.get_map()
    player = None

    while player is None:
        spawn_points = world_map.get_spawn_points()
        spawn_point = np.random.choice(spawn_points) if spawn_points else carla.Transform()
        player = world.try_spawn_actor(world.get_blueprint_library().filter('model3')[0],
                                       spawn_point)
    return player

def load_network(model_class, path, shape=None, pretrained=None):
    if isinstance(shape, type(None)):
        model = model_class()
    else:
        model = model_class((180, 240), pretrained=False)

    epoch, arch, state_dict = torch.load(path).values()
    model.load_state_dict(state_dict)
    model = model.cuda()
    model.eval()
    return model

def take_action(world, junction_data, road_id, lane_id, action, position):
    wp = world.get_map().get_waypoint(position)
    if wp.is_junction:
        junc = wp.get_junction()
        k = (road_id, lane_id, junc.id)
        # Si llega a una intersección por lado desconocido
        if not k in junction_data.keys():
            action = [0, 0, 0, 1]
        else:
            choice = np.random.choice(junction_data[k])
            if action == [0, 0, 0, 1]:
                if choice == 'l':
                    action = [1, 0, 0, 0]
                elif choice == 'r':
                    action = [0, 1, 0, 0]
                elif choice == 'f':
                    action = [0, 0, 1, 0]
            else:
                road_id, lane_id = wp.road_id, wp.lane_id
                action = [0, 0, 0, 1]
    return road_id, lane_id, action

def steer_correction(action, mutable_params, threshold):
    if action == [1, 0, 0, 0]:
        action_str = 'left'

```

```

# Si no está en un rango de giro mínimo
if not mutable_params['s'] < -threshold:
    action_str = 'force left'
    if not mutable_params['s'] < -threshold:
        mutable_params['s'] = max(-threshold - mutable_params['ac'], -0.8)
        mutable_params['ac'] += 0.02
    else:
        mutable_params['ac'] = 0
    ema_str = ''
elif action == [0, 1, 0, 0]:
    action_str = 'right'
    # similar al caso de arriba pero al otro lado
    if not mutable_params['s'] > threshold:
        action_str = 'force right'
        mutable_params['s'] = min(threshold + mutable_params['ac'], 0.8)
        mutable_params['ac'] += 0.02
    else:
        mutable_params['ac'] = 0
    ema_str = ''
elif action == [0, 0, 1, 0]:
    action_str = 'forward'
    if not abs(mutable_params['s']) <= threshold:
        mutable_params['s'] = -mutable_params['s']
    mutable_params['ac'] = 0
    ema_str = ''
elif action == [0, 0, 0, 1]:
    if isinstance(mutable_params['ema'], type(None)): # inicializar EMA
        mutable_params['ema'] = mutable_params['s']
    else:
        mutable_params['ema'] = mutable_params['alpha']*mutable_params['s']+\
            (1-mutable_params['alpha'])*mutable_params['ema']
    if abs(mutable_params['s']) <= threshold:
        s = mutable_params['ema']
    else:
        ema_str = ''
    mutable_params['ac'] = 0
    action_str = 'no action'

return action_str, ema_str # dbg

def get_class_semseg(segmentation, class_id):
    seg = ((segmentation == class_id)*255).astype(np.uint8)
    seg = cv2.morphologyEx(seg, cv2.MORPH_OPEN, np.ones((5, 5)))
    seg = cv2.dilate(seg, np.ones((5, 5)), iterations = 2)
    seg = cv2.erode(seg, np.ones((5, 5)), iterations = 1)
    return seg

def extract_objects_depth(depth_map, contours, veh, poles, ped):
    objects_depth = np.round(depth_map).astype(np.uint8)
    depth_vals_veh = cv2.bitwise_and(objects_depth, veh)
    depth_vals_pol = cv2.bitwise_and(objects_depth, poles)
    objects_depth = cv2.bitwise_or(depth_vals_veh, depth_vals_pol)

```

```

cv2.fillPoly(objects_depth, pts=[contours], color=0)
return objects_depth

def bounding_box(min_area, signals, directions, x_min_thresh=0):
    pq = PriorityQueue()
    if signals.sum() > 0: # Comprobar si contiene semáforos
        bbox = find_contours(signals, directions, x_min_thresh)
        for x1, y1, x2, y2 in bbox:
            area = abs(x1 - x2)*abs(y1 - y2)
            if area > min_area and x1 > x_min_thresh:
                pq.push((-area, x1, y1, x2, y2))
    valid = True if not pq.empty() and abs(pq.top()[0]) > min_area else False
    if valid:
        _, x1, y1, x2, y2 = pq.top()
        x, y = round(x1), round(y1)
        w, h = round(abs(x1 - x2)), round(abs(y1 - y2))
        return x, y, w, h
    return None

def cluster(crop, color_code):
    cod_val = 'na'
    inp = np.float32(crop.reshape((-1,3)))
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
    K = 4
    _, label, center = cv2.kmeans(inp, K, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
    res = np.uint8(center[label.flatten()]).reshape((crop.shape))
    res = res * (res > 200)
    rgb_sig = [res[:, :, 0].sum(), res[:, :, 1].sum(), res[:, :, 2].sum()]
    rc, gc, bc = rgb_sig
    if rc > 0 and gc > 0:
        if bc > 0:
            cod_val = 'g'
        else:
            cod_val = color_code[np.argmax(rgb_sig)]
    elif rc > 0 or gc > 0 or bc > 0:
        cod_val = color_code[np.argmax(rgb_sig)]
    return cod_val, res, rc, gc, bc

@njit
def ff(mat, i, j, x1, y1, x2, y2, directions):
    if (0 <= i < mat.shape[0]) and (0 <= j < mat.shape[1]) and mat[i, j] != 0:
        mat[i, j] = 0
        x1 = min(x1, j)
        y1 = min(y1, i)
        x2 = max(x2, j)
        y2 = max(y2, i)

        for dx, dy in directions:
            x1, y1, x2, y2 = ff(mat, i+dy, j+dx, x1, y1, x2, y2, directions)
    return x1, y1, x2, y2

@njit

```

```

def find_contours(img, directions, x_min_thresh=0, pad=1, copy=True):
    if copy:
        local_img = img.copy()
    else:
        local_img = img
    boxes = []

    for i in range(0, local_img.shape[0]):
        for j in range(x_min_thresh, local_img.shape[1]):
            if local_img[i, j] != 0:
                x1, y1, x2, y2 = ff(local_img, i, j,
                                      np.inf, np.inf, -np.inf, -np.inf, directions)
                boxes.append((int(max(x1-pad, 0)),
                              int(max(y1-pad, 0)),
                              int(min(x2+pad, local_img.shape[1])),
                              int(min(y2+pad, local_img.shape[0]))))

    # retorna en formato x1, y1, x2, y2
    return boxes

if __name__ == '__main__':
    try:
        actor_list = []

        # 1. Definir la pantalla de visualización de imágenes mediante la librería PyGame.
        w, h = 240, 180
        pygame.init()
        display = pygame.display.set_mode((2*w, 2*h),
                                         pygame.HWSURFACE | pygame.DOUBLEBUF)
        clock = pygame.time.Clock()

        try:
            # 2. Conectar con el simulador e inicializar el vehículo en el mapa.
            client = carla.Client('localhost', 2000)
            client.set_timeout(2.0)

            world = client.get_world()
            player = spawn_player(world)
            actor_list.append(player)

            set_random_weather(world)

            blueprint_library = world.get_blueprint_library()
            font = cv2.FONT_HERSHEY_SIMPLEX

            cam_pos = carla.Transform(carla.Location(x=1.6, z=1.7))
            camera_rgb = create_camera(cam_type='rgb',
                                       vehicle=player,
                                       pos=cam_pos,
                                       h=h, w=w,
                                       lib=blueprint_library,
                                       world=world)

```

```

actor_list.append(camera_rgb)

# 3. Cargar los parámetros de las redes.
model_depth = load_network(CustomMobilenetDepth,
                           'path/to/weights',
                           (180, 240), False)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
depth_preprocess = transforms.Compose([
    transforms.ToTensor(),
    lambda T: T[:3],
    normalize
])

model_drive = load_network(CustomMobileNet, 'path/to/weights')

drive_preprocess = transforms.Compose([
    transforms.Resize((224, 224), interpolation=Image.BICUBIC),
    transforms.ToTensor(),
])
model_semseg = load_network(CustomMobilenetSemseg, 'path/to/weights',
                            (180, 240), False)

with CarlaSyncMode(world, camera_rgb, fps=30) as sync_mode:
    road_id, lane_id = 0, 0
    action = [0, 0, 0, 1]

    mutable_params = {'ema': None, 'alpha': 0.75, 'ac': 0,
                      's': None, 't': None, 'b': 0}

    threshold = 0.2
    min_area = 95
    x_min_thresh = 120
    color_code = ['r', 'g', 'b']

# 4. Iniciar un bucle de iteraciones de la simulación a 30 fps.
while True:
    if should_quit():
        return
    clock.tick()

    # Avanzar un tick de la simulación
snapshot, image_rgb = sync_mode.tick(timeout=2.0)

    # Convertir a un arreglo BGR
rgb_arr = img_to_array(image_rgb)

    # Convertir a RGB
X_img = Image.fromarray(cv2.cvtColor(rgb_arr, cv2.COLOR_BGR2RGB))

```

```

X_drive = drive_preprocess(X_img).view(1, 3, 224, 224).cuda()
X_depth = depth_preprocess(X_img).view(1, 3, 180, 240).cuda()

position = player.get_location()
# 5. Verifica con el simulador si está en una intersección y decide el camino
road_id, lane_id, action = take_action(world, junction_data, road_id,
                                         lane_id, action, position)
action_tensor = torch.tensor(action).view(1, -1).cuda()

# 6. Ingresar los valores de entrada a cada red neuronal y recibir su salida.
with torch.no_grad():
    pred_drive = model_drive(X_drive, action_tensor).cpu()
    depth_map = model_depth(X_depth)[0, 0].cpu().numpy()
    segmentation = model_semseg(X_drive).cpu().numpy()[0] \
        .argmax(axis=0).astype(np.uint8)

    mutable_params['t'] = round(float(pred_drive[0, 0]), 3)
    mutable_params['s'] = round(float(pred_drive[0, 1]), 3)
    mutable_params['t'] = min(mutable_params['t'], 0.5)

# 7. Corregir las oscilaciones con EMA.
# 8. Si el vehículo gira lo suficiente, se da un impulso mediante un acumulador.
    action_str, ema_str = steer_correction(action, mutable_params, threshold)

# 9. Se procesan las predicciones de la segmentación semántica para obstáculos.
walkers = get_class_semseg(segmentation, 4)
vehicles = get_class_semseg(segmentation, 10)
poles = get_class_semseg(segmentation, 5)

v_boxes = find_contours(vehicles, directions, pad=1, copy=True)
w_boxes = find_contours(walkers, directions, pad=1, copy=True)
p_boxes = find_contours(poles, directions, pad=1, copy=True)

objects_depth = extract_objects_depth(depth_map, contours,
                                       vehicles, poles, walkers)

# 10. Se calcula la moda de las distancias de objetos cercanos.
if np.sum(objects_depth) != 0:
    c = sorted(Counter(list(objects_depth.ravel())).items(),
               key=lambda x: -x[1])
    if c[0][0] == 0:
        moda = c[1][0]
        count = c[1][1]
    else:
        moda = c[0][0]
        count = c[0][1]
    else:
        moda = np.inf
        count = np.inf

    if moda <= 4 and 40 <= count < np.inf:

```

```

        mutable_params['s'], mutable_params['t'] = 0, 0
        mutable_params['b'] += 0.10
        mutable_params['b'] = round(min(max(0, mutable_params['b']), 1), 3)
    else:
        mutable_params['b'] = 0

# 11. Se detecta la posición de los semáforos.
signals = get_class_semseg(segmentation, 12)
box = bounding_box(min_area, signals, directions, x_min_thresh)

# 12. Se predice un código de color, r para rojo o g para verde.
code_val = 'na'
signal_img = np.ones((180, 60, 3))
rc, gc, bc = 0, 0, 0
k_img = None
if isinstance(box, tuple):
    x, y, w, h = box
    crop = cv2.resize(cv2.cvtColor(rgb_arr[y:y+h, x:x+w, :], cv2.COLOR_BGR2RGB), None, fx=3, fy=3)
    # K-MEANS
    code_val, k_img, rc, gc, bc = cluster(crop, color_code)

# 13. Se usa el código de color para decidir si frenar o no.
if code_val == 'r':
    mutable_params['b'] = 1
    mutable_params['s'] = 0
    mutable_params['t'] = 0

# 14. Se envían las decisiones finales de control al vehículo.
player.apply_control(carla.VehicleControl(throttle= mutable_params['t'],
                                             steer= mutable_params['s'],
                                             brake= mutable_params['b']))
line1 = f"t: {round(mutable_params['t'], 2)} \
          b: {round(mutable_params['b'], 2)} \
          s: {round(mutable_params['s'], 2)} \
          color: {code_val}"
line2 = f"moda: {moda}, r: {int(rc>0)} g: {int(gc>0)} b: {int(bc>0)}"
line3 = f"act: {action_str} {action}"
rgb_arr = cv2.resize(rgb_arr, None, fx=2, fy=2,
                     interpolation=cv2.INTER_CUBIC)
pos1 = (10, 30)
pos2 = (10, 60)
black, green = (0, 0, 0), (156, 237, 58)
cv2.putText(rgb_arr, line1, pos1, font, 0.6, black, 3, cv2.LINE_AA)
cv2.putText(rgb_arr, line1, pos1, font, 0.6, green, 1, cv2.LINE_AA)
cv2.putText(rgb_arr, line2, pos2, font, 0.6, black, 3, cv2.LINE_AA)
cv2.putText(rgb_arr, line2, pos2, font, 0.6, green, 1, cv2.LINE_AA)
cv2.putText(rgb_arr, line3, pos3, font, 0.6, black, 3, cv2.LINE_AA)
cv2.putText(rgb_arr, line3, pos3, font, 0.6, green, 1, cv2.LINE_AA)

# visualización de los rectángulos delimitando los objetos
if not isinstance(box, type(None)):

```

```

color = (58, 58, 237) if code_val == 'r' else (58, 237, 67)
cv2.rectangle(rgb_arr, (box[0]*2, box[1]*2),
              ((box[0]+box[2])*2, (box[1]+box[3])*2), color, 2)

for x1, y1, x2, y2 in v_boxes:
    cv2.rectangle(rgb_arr, (x1*2, y1*2), (x2*2, y2*2), (237, 152, 58), 2)
for x1, y1, x2, y2 in p_boxes:
    cv2.rectangle(rgb_arr, (x1*2, y1*2), (x2*2, y2*2), (115, 58, 237), 2)
for x1, y1, x2, y2 in w_boxes:
    cv2.rectangle(rgb_arr, (x1*2, y1*2), (x2*2, y2*2), (58, 161, 237), 2)

show_window(display, rgb_arr)

# visualizar el semáforo segmentado
if not isinstance(k_img, type(None)):
    k_img = cv2.cvtColor(k_img, cv2.COLOR_RGB2BGR)
    k_img = cv2.resize(k_img, (31, 55), interpolation=cv2.INTER_CUBIC)
    show_window(display, k_img, (rgb_arr.shape[1]-k_img.shape[1], 0))

pygame.display.flip()
finally:
    for actor in actor_list:
        actor.destroy()
    pygame.quit()
except KeyboardInterrupt:
    print('\nFin')

```
