

Universidad de Alcalá Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Algoritmos de detección de objetos 3D basados en LiDAR:
comparación entre técnicas PCL clásicas y Deep Learning

ESCUELA POLITECNICA

Autor: Javier de la Peña García

Tutores: Luis Miguel Bergasa Pascual y Carlos Gómez Huélamo

2021

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

**Algoritmos de detección de objetos 3D basados en LiDAR:
comparación entre técnicas PCL clásicas y Deep Learning**

Autor: Javier de la Peña García

Directores: Luis Miguel Bergasa Pascual y Carlos Gómez Huélamo

Tribunal:

Presidente: Felipe Espinosa Zapata

Vocal 1º: Fernando Naranjo Vega

Vocal 2º: Rafael Barea Navarro

Calificación:

Fecha:

A nuestros alumnos pasados, presentes y futuros...

“Empieza haciendo lo necesario, luego haz lo posible y de pronto empezarás a hacer lo imposible.”
Francisco de Asís

Agradecimientos

A todos los que la presente vieran y entendieren.

Inicio de las Leyes Orgánicas. Juan Carlos I

Este trabajo es el fruto de muchas horas de trabajo, tanto de los autores últimos de los ficheros de la distribución como de todos los que en mayor o menor medida han participado en él a lo largo de su proceso de gestación.

Mención especial merece Manuel Ocaña, el autor de la primera versión de las plantillas de proyectos fin de carrera y tesis doctorales usadas en el Departamento de Electrónica de la Universidad de Alcalá, con contribuciones de Jesús Nuevo, Pedro Revenga, Fernando Herráñez y Noelia Hernández.

En la versión actual, la mayor parte de las definiciones de estilos de partida proceden de la tesis doctoral de Roberto Barra-Chicote, con lo que gracias muy especiales para él.

También damos las gracias a Manuel Villaverde, David Casillas, Jesús Pablo Martínez, José Francisco Velasco Cerpa que nos han proporcionado secciones completas y ejemplos puntuales de sus proyectos fin de carrera.

Finalmente, hay incontables contribuyentes a esta plantilla, la mayoría encontrados gracias a la magia del buscador de Google. Hemos intentado referenciar los más importantes en los fuentes de la plantilla, aunque seguro que hemos omitido alguno. Desde aquí les damos las gracias a todos ellos por compartir su saber con el mundo.

Resumen

Este documento ha sido generado con una plantilla para memorias de trabajos fin de carrera, fin de máster, fin de grado y tesis doctorales. Está especialmente pensado para su uso en la Universidad de Alcalá, pero debería ser fácilmente extensible y adaptable a otros casos de uso. En su contenido se incluyen las instrucciones generales para usarlo, así como algunos ejemplos de elementos que pueden ser de utilidad. Si tenéis problemas, sugerencias o comentarios sobre el mismo, dirigidlas por favor a Javier de la Peña García <j.peña@edu.uah.es>.

Palabras clave: Plantillas de trabajos fin de carrera/máster/grado y tesis doctorales, L^AT_EX, soporte de español e inglés, generación automática.

Abstract

This document has been generated with a template for Bsc and Msc Thesis (trabajos fin de carrera, fin de máster, fin de grado) and PhD. Thesis, specially thought for its use in Universidad de Alcalá, although it should be easily extended and adapted for other use cases. In its content we include general instructions of use, and some example of elements than can be useful. If you have problemas, suggestions or comments on the template, please forward them to Javier de la Peña García <j.pena@edu.uah.es>.

Keywords: Bsc., Msc. and PhD. Thesis template, L^AT_EX, English/Spanish support, automatic generation.

Resumen extendido

Con un máximo de cuatro o cinco páginas. Se supone que sólo está definido como obligatorio para los TFGs y PFCs de UAH.

Índice general

Resumen	ix
Abstract	xi
Resumen extendido	xiii
Índice general	xv
Índice de figuras	xix
Índice de tablas	xxi
Índice de listados de código fuente	xxiii
Índice de algoritmos	xxv
Lista de acrónimos	xxviii
1 Introducción	1
1.1 Sistemas de conducción autónomos	1
1.2 Sistemas de percepción	2
1.2.1 Principales sensores para la percepción en vehículos autónomos	3
1.2.1.1 Cámara	3
1.2.1.2 Radar	4
1.2.1.3 LiDAR	4
1.2.2 Sistemas de detección	5
1.2.3 Sistemas de seguimiento	6
1.2.4 Fusión sensorial	7
1.3 Deep Learning	8
2 Propuesta de trabajo	9

3 Sistemas clásicos de percepción con LiDAR	11
3.1 Voxelización	11
3.2 RANSAC-3D	12
3.3 KD-tree	14
3.4 Filtrado previo y posterior a la detección	18
4 Sistemas de percepción con LiDAR basados en Deep Learning	19
4.1 Principales datasets	19
4.1.1 KITTI	19
4.1.1.1 Análisis de la estructura del GT y las PCLs de KITTI	21
4.1.2 Waymo	23
4.1.3 nuScenes	25
4.1.3.1 NuScenes devkit	27
4.1.4 Comparativa entre los diferentes datasets	28
4.2 Estado del arte en detección utilizando LiDAR	29
4.2.1 SECOND	29
4.2.2 PointPillars	31
4.2.3 PointRCNN	32
4.2.4 PV-RCNN	33
4.2.5 CBGS	35
4.3 OpenPCDet	36
5 Desarrollo realizado	39
5.1 Estado del proyecto T4AC	39
5.1.1 ROS	39
5.1.2 Docker	39
5.1.3 Estructura del proyecto	39
5.2 Implementación en CARLA	39
5.2.1 CARLA	39
5.2.2 Funcionamiento del LiDAR en CARLA	39
5.2.3 Implementación del sistema clásico utilizando LiDAR	39
5.2.4 Implementación del sistema basado en Deep Learning utilizando LiDAR	39
5.3 Fusión sensorial	39
5.4 Vehículo del proyecto T4AC	39
5.5 Implementación sobre el vehículo T4AC	39

6 AD DevKit	41
6.1 Estado del arte en evaluación de vehículos autónomos	41
6.2 Obtención del ground truth	41
6.3 Evaluación de los modelos	41
7 Resultados obtenidos	43
7.1 Análisis cuantitativo en Kitti	43
7.2 Análisis cuantitativo en nuScenes	43
7.3 Análisis cualitativo del modelo clásico en CARLA	43
7.4 Análisis cualitativo de CBGS en CARLA	43
7.4.1 Comparativa con PointPillars en CARLA	43
7.5 Análisis cuantitativo de CBGS en CARLA	43
7.6 Análisis cualitativo de CBGS sobre el vehículo T4AC	43
7.6.1 Comparativa con PointPillars sobre el vehículo T4AC	43
8 Conclusiones	45
8.1 Modelos estudiados	45
8.2 Comparativas adicionales	45
8.2.1 Ajuste de modelos basados en Kitti a nuScenes	45
8.2.2 Número de PCL de entrada en modelos evaluados sobre nuScenes	45
8.2.3 Tamaño del voxel en modelos basados en redes neuronales	45
8.3 Futuros trabajos	45
Bibliografía	47
A Herramientas y recursos	51

Índice de figuras

1.1	Arquitectura de un sistema de conducción autonóma.	1
1.2	Niveles de autonomía.	2
1.3	Cámara utilizada en vehículos autónomos.	3
1.4	Uso de cámara con lluvia.	3
1.5	Radar utilizado en vehículos autónomos.	4
1.6	LiDAR utilizado en vehículos autónomos.	5
1.7	Detecciones 2D utilizando cámara.	5
1.8	Detecciones 3D utilizando LiDAR.	6
1.9	Tracking como vista de pájaro sobre una nube de puntos.	7
1.10	Fusión sensorial utilizando cámara y LiDAR.	7
1.11	Convolutional Neural Network.	8
3.1	Entorno 3D voxelizado.	11
3.2	Aplicación de RANSAC para detección de outliers.	12
3.3	Aplicación de RANSAC-3D.	14
3.4	Árbol binario ordenado.	15
3.5	Espacio bidimensional dividido por un KD-tree.	16
3.6	Estructura de un KD-tree de dos dimensiones.	16
4.1	Estructura del dataset de KITTI.	20
4.2	Vehículo utilizado en KITTI.	21
4.3	Medidas del vehículo utilizado en KITTI.	21
4.4	Estructura del archivo tracklet_labels.xml.	22
4.5	Visualización de una nube de puntos de KITTI junto con su ground truth.	23
4.6	Vehículo utilizado en Waymo.	23
4.7	Nube de puntos con las diferentes clases del dataset de Waymo.	24
4.8	Vehículo utilizado en el dataset de nuScenes.	25
4.9	Estructura del dataset nuScenes.	26
4.10	Esquema de nuScenes.	27

4.11 Esquema de nuImages	27
4.12 Transformación mundo a imagen de la nube de puntos segmentada en el devkit de nuScenes. .	28
4.13 Arquitectura propuesta del detector SECOND.	30
4.14 Algoritmo de convolución disperso propuesto en SECOND.	30
4.15 Arquitectura propuesta del detector PointPillars.	31
4.16 Arquitectura propuesta del detector PointRCNN.	32
4.17 Agrupación de regiones de la nube de puntos en el modelo PointRCNN.	33
4.18 Arquitectura propuesta del detector PV-RCNN.	34
4.19 Módulo Predicted Keypoint Weighting del modelo PV-RCNN.	34
4.20 Arquitectura propuesta del detector CBGS.	35
4.21 Diseño de OpenPCDet.	36

Índice de tablas

4.1	Comparativa entre los principales datasets.	29
7.1	Rendimiento medio de CBGS PointPillars Multihead en nuScenes.	43
7.2	Análisis por clase de CBGS PointPillars Multihead en nuScenes.	44
7.3	Análisis por clase de CBGS PointPillars Multihead en CARLA.	44

Índice de listados de código fuente

4.1 Obtención de una nube de puntos segmentada sobre una imagen utilizando nuScenes devkit 28

Índice de algoritmos

3.1	Algoritmo RANSAC	13
3.2	Búsqueda en árbol binario ordenado	15
3.3	Inserción en KD-tree	16
3.4	Cluster por distancia en KD-tree	17

Lista de acrónimos

AD DevKit	Autonomous Driving Development Kit.
ADAS	Advanced Driver Assistance System.
ADS	Automated Driving System.
AHRS	Attitude and Heading Reference System.
BEV	Bird's Eye View.
CNN	Convolutional Neural Network.
DL	Deep Learning.
EKF	Extended Kalman Filter.
FC	Fully Connected.
FCN	Fully Connected Network.
FPS	Frames per Second.
GNSS	Global Navigation Satellite System.
GPS	Global Positioning System.
IMU	Inertial Measurement Unit.
IoU	Intersection over Union.
KF	Kalman Filter.
KIT	Karlsruhe Institute of Technology.
KNN	K Nearest Neighbors.
LiDAR	Light Detection and Ranging.
NMS	Non Maximum Suppression.
Radar	Radio Detection and Ranging.
RANSAC	Random Sampling and Consensus.

ReLU	Rectified Linear Unit.
RoI	Region of Interest.
RPN	Region Proposal Network.
RTK	Real Time Kinematic.
SLAM	Simultaneous Localization and Mapping.
Sonar	Sound Navigation and Ranging.
SOTA	State of the Art.
SSD	Single Shot Detector.
TTIC	Toyota Technological Institute at Chicago.
UKF	Unscented Kalman Filter.
VFE	Voxel Feature Encoding.
XML	Extensible Markup Language.
YOLO	You Only Look Once.

Capítulo 1

Introducción

No te conformes con el mundo que has heredado. Nunca se ha resuelto un desafío sin personas que pensasen diferente.

Tim Cook

Para el pleno entendimiento del trabajo desarrollado se explica el funcionamiento de los sistemas de conducción autónoma, para centrarse en la parte de percepción del vehículo y comprender que gracias al Deep Learning se consigue el estado del arte en este campo.

1.1 Sistemas de conducción autónomos

En los últimos años gracias a una mejora en los sensores, en la capacidad de computo principalmente por la aceleración por hardware, la visión por computador, el Deep Learning (DL) 1.3 y el desarrollo de técnicas de comunicación, ha propiciado que nos encontremos en una carrera por la creación de sistemas de conducción autónomos. Empresas de sectores de la automoción y la tecnológicas como ArgoAI, Audi, Baidu, Cruise, Mercedes-Benz, Tesla, Uber o Waymo entre otras, invierten enormes cantidades de dinero para el desarrollo de estas tecnologías [1].

Para la obtención de sistemas de conducción autónoma es necesario tener un buen entendimiento del entorno y hacer uso de un buen control en tiempo real, para ello son utilizados sensores que puedan aportar información al vehículo como son cámaras, LiDAR, Radar, IMU, GPS o hasta Sonar.

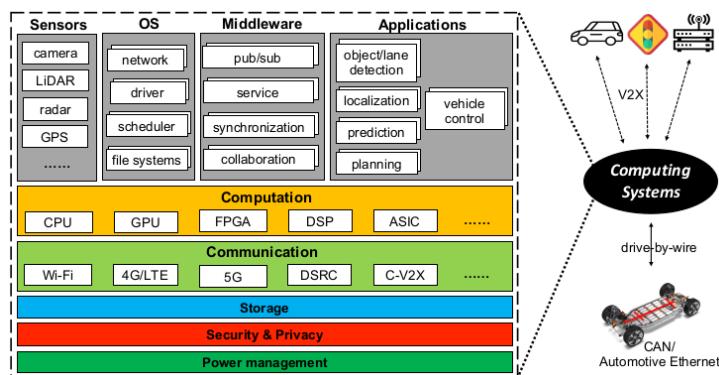


Figura 1.1: Arquitectura de un sistema de conducción autónoma.

En adición a los sensores también es necesario la utilización de sistemas de localización tanto global como local, mapeado del entorno, toma de decisiones y control del vehículo.

La evolución continua de estos sistemas trata de ofrecer un mayor nivel de seguridad al volante con Advanced Driver Assistance System (ADAS), para que en un futuro puedan ser remplazados por Automated Driving System (ADS).

Para analizar el avance de estos sistemas y para poder compararlos, se ha dividido según su nivel de autonomía, por lo que se tiene desde un nivel 0 a un nivel 5. El nivel 0 indica que el coche no tiene ningún tipo de autonomía, en el nivel 1 el vehículo sigue siendo controlado por el conductor pero ciertas características de ayuda a la conducción son añadidas, en el siguiente nivel el vehículo es capaz de acelerar, frenar y hasta dirigir el vehículo pero con el conductor siempre atento, en el nivel 3 el conductor es necesario pero no es requerimiento la atención al entorno, pero debe de estar listo para tomar en control en todo momento, el nivel 4 permite un nivel de autonomía donde el vehículo no requiere de atención pero únicamente en ciertos escenarios y el último nivel es el que habilita la conducción autónoma completa [2].

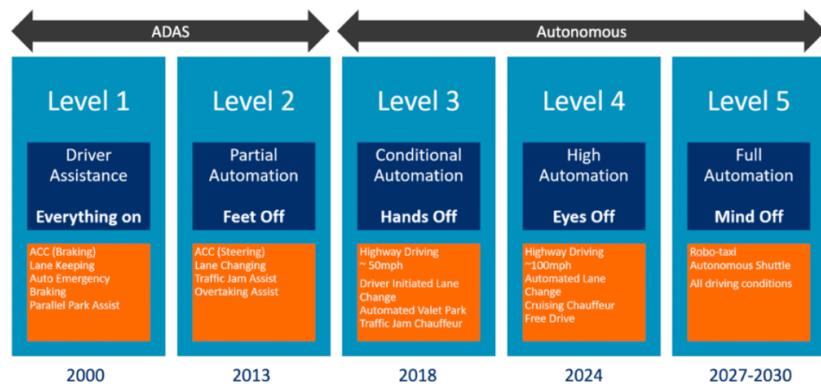


Figura 1.2: Niveles de autonomía.

El desarrollo de este tipo de sistemas no es una tarea sencilla, múltiples empresas involucradas en el desarrollo de vehículos autónomos pretendían tener vehículos en el nivel 4 de autonomía en poco tiempo, pero como se ha visto esto no es posible, actualmente nos encontramos en el mercado con sistemas que se encuentran entre el nivel 2 y el 3, por lo que aún queda un largo camino antes de llegar a conducción autónoma completa.

1.2 Sistemas de percepción

Los sistemas de **ADS/ADAS** requieren de un entendimiento del entorno para poder funcionar correctamente, para ello es necesario añadir diversos sensores a los largo del vehículo que nos permitan obtener la mayor información del exterior posible. A partir de este conocimiento es posible la toma de decisiones y la planificación, por lo que en este apartado se va a explicar de que manera se puede configurar un sistema de percepción, cuales son los principales sensores y que información se puede obtener de cada uno de ellos.

1.2.1 Principales sensores para la percepción en vehículos autónomos

Para la creación un sistema de percepción robusto es necesario el uso de diversos tipos de sensores que ofrezcan una información de relevancia de manera diferente al resto, por ello se utilizan sensores como cámaras, [Radar](#), [LiDAR](#), sensores de ultrasonidos, [GPS](#), [GNSS](#), [IMU](#) etc.

Estos ofrecen información de localización, velocidad, distancia de objetos en el entorno, e incluso información del propio vehículo, como su propia localización, la velocidad lineal y angular que este tiene.

También es necesario tener en cuenta que no todos los sensores funcionan de la misma manera en distintos escenario por lo que en situaciones donde un sensor es incapaz de obtener buenos datos otro sensor puede suplir esta carencia, por lo que la redundancia de sensores aporta otro nivel de seguridad al vehículo ya no solo un nivel mayor de detección del entorno.

1.2.1.1 Cámara

Uno de los sensores más utilizados es la cámara, este es el más extendido debido a la gran riqueza de información que ofrece del entorno. Actualmente se pueden encontrar cámaras que generen imágenes a una gran resolución y a una alta tasa de [FPS](#) por un precio bastante asequible, este es uno de los sensores más baratos.



Figura 1.3: Cámara utilizada en vehículos autónomos.

El problema de este sensor recae en el computo que es necesario para obtener información a partir de las imágenes, ya que estas no son más que píxeles en escala de grises o con un sistema de colores como el RGB. Por ello no solo es necesario tener en cuenta el coste del sensor, sino que también hay que aumentar la capacidad de computo del ordenador de abordo para que pueda analizar en tiempo real las imágenes.

Por último es necesario conocer las limitaciones de la cámara, esta funciona de forma correcta en situaciones de buena luminosidad y sin reflejos, por lo que en situaciones con lluvia [1.4](#), niebla, durante la noche y otros escenarios climatológicos adversos, no es capaz de obtener toda la información que esta obtendría en situaciones más favorables, lo cual hace que otros sensores sean usados en estas condiciones adversas para lidiar con estas limitaciones.



Figura 1.4: Uso de cámara con lluvia.

Aún conociendo las desventajas de estos sensores la gran mayoría de enfoques incluyen cámaras para la obtención de los objetos del entorno tanto en 2D como en 3D, pudiendo utilizar para lo segundo un sistema de cámaras estéreo que obtienen también información de la profundidad.

1.2.1.2 Radar

Los [Radar](#) son utilizados en múltiples aplicaciones como la previsión meteorológica, la astronomía, las comunicaciones, la navegación oceánica y la conducción autónoma entre otras.

Este sensor emite ondas de radio, las cuales son reflejadas devuelta a este, lo cual da una información de donde se hayan los objetos en el espacio tridimensional, lo que implica la distancia a estos junto con los dos ángulos necesarios, además gracias al efecto Doppler se puede inferir la velocidad de los objetos a partir de un fenómeno que hace variar la frecuencia de la onda enviada si hay algún tipo de movimiento local relativo respecto del propio [Radar](#) [3, 4].



Figura 1.5: Radar utilizado en vehículos autónomos.

Como se ha visto este sensor al contrario que la cámara obtiene directamente información utilizable para el entendimiento del entorno de forma directa, el problema radica en la escasa cantidad de datos que provee. Aunque se obtenga información de localización 3D y de velocidad, la cantidad de la nube de puntos producida es muy pequeña, por lo que es necesario de otros sensores para obtener una información completa del entorno.

Por otra parte, una de las principales ventajas del [Radar](#) es que se puede utilizar en cualquier situación meteorológica, únicamente podría verse afectado por lluvias muy intensas. Por lo que es un sensor muy completo y una gran adición para obtener información adicional de posición 3D y velocidad a un precio inferior a un [LiDAR](#) y por ello es adoptado por gran cantidad de sistemas [ADAS](#) en conjunto con sistemas de cámaras 360 alrededor del vehículo.

1.2.1.3 LiDAR

De forma similar al [Radar](#), los sistemas [LiDAR](#) basan su funcionamiento en el escaneo del entorno a partir de el envío de láseres y el cálculo del tiempo desde su envío hasta su retorno. Con esta información de distancia y el ángulo de inclinación del haz que envió esa señal, se construye una nube de puntos que consta de valores x, y, z de posición y otro valor que es el coeficiente de reflectividad del rayo de luz con el objeto incidido [3].

Actualmente los [LiDAR](#) más utilizados son de 64 canales, lo cual indica que se tienen 64 láseres funcionando al mismo tiempo lo que da una gran resolución del entorno, además la nube de puntos generada es de hasta 120 metros alrededor del vehículo lo cual permite detectar objetos a una distancia

considerable y saber de manera casi perfecta su distancia en un entorno tridimensional gracias a los alrededor de 2.000.000 de puntos que se generan por segundo del entorno [5].



Figura 1.6: LiDAR utilizado en vehículos autónomos.

Los sistemas **LiDAR** tienen la ventaja de ser un sensor que aporta mucha información del entorno, pero tienen el mismo problema que las cámaras, en condiciones de lluvia, nieve, granizo o niebla, la efectividad de este sensor decae aunque se trata de minimizar ajustando la longitud de onda del láser utilizado [6].

Aún siendo un sensor muy útil, que puede aumentar el nivel de redundancia del sistema además de la seguridad, múltiples compañías como Tesla tratan de evitar su uso utilizando únicamente cámaras y **Radar**, esto es debido a que un **LiDAR** suele costar entre 8.000 y 100.000 dólares si se requiere de una resolución similar al estado del arte entre 16 y 128 haces [1].

1.2.2 Sistemas de detección

Unicamente con un sistema de sensores no es posible la comprensión del entorno, también es necesario de un procesamiento de los datos, mientras que la cámara no da ninguna información de forma directa, el **LiDAR** y el **Radar** son capaces de obtener la posición de obstáculos alrededor del vehículo, y además el **Radar** es capaz de inferir la velocidad de los objetos sin necesidad de un seguimiento.

Principalmente en los sistemas de detección para conducción autónoma se trata de obtener las posiciones de los diferentes objetos de interés del entorno. Estas detecciones pueden ser tanto en 2D como en 3D, pero el problema radica en como obtener un rectángulo u ortoedro que identifique donde se encuentran dicho objetos del entorno.

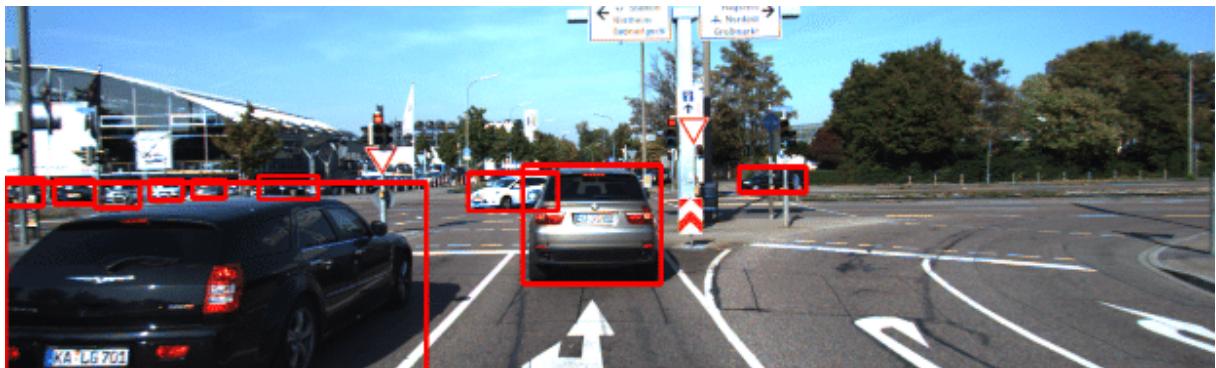


Figura 1.7: Detecciones 2D utilizando cámara.

Una de las formas de detección del entorno es a partir de cámaras, esto puede ser conseguido con un sistema de una cámara o de un sistema multicámara que abarque los 360 grados alrededor del coche, con esto instalado en el coche se puede generar rectángulos sobre las imágenes de los objetos del entorno como coches, peatones, bicicletas, motocicletas... Con esto se obtendría un listado de objetos detectados en 2D, lo cual es obtenible con un modelo basado en redes neuronales como [YOLO](#) [7] que es capaz de hacerlo en tiempo real, y que funciona tal y como se ve en [1.7](#).

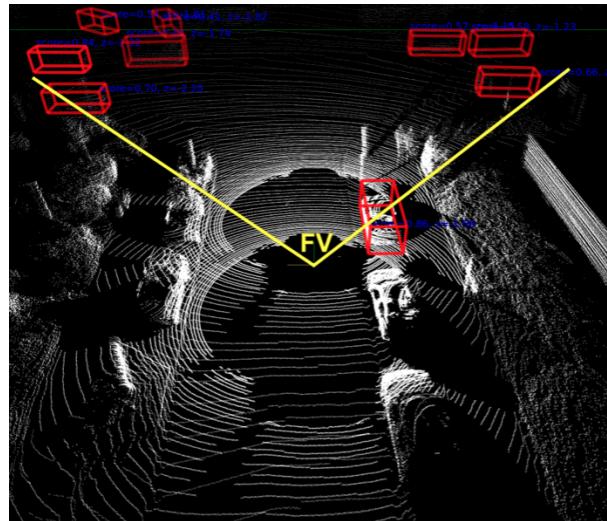


Figura 1.8: Detecciones 3D utilizando LiDAR.

Unicamente con detecciones 2D no se puede obtener la distancia a los vehículos, como mucho una aproximación a partir del tamaño de la bounding box 2D, aparte de la dirección en la que se encuentran respecto del coche. Por lo que termina trabajando con detecciones 3D, las cuales pueden tener como paso intermedio una detección 2D o ser obtenidas directamente con un sistema estéreo de cámaras o a partir de las nubes de nubes del [Radar](#) o del [LiDAR](#).

Las técnicas que realizan detecciones en un sistema tridimensional tienen un coste computacional mayor al trabajar con una dimensión añadida, por lo que es necesaria la utilización de técnicas que permitan realizar estas detecciones en tiempo real, como es el caso de PointPillars [8], modelo que se explicará más adelante en [4.2.2](#), y que como se ve en [1.8](#) es capaz de realizar las detecciones en un entorno tridimensional utilizando únicamente el [LiDAR](#).

1.2.3 Sistemas de seguimiento

Mientras que en los sistemas de detección se suele utilizar un único estado discretizado del entorno percibido por los sensores, el seguimiento o tracking utiliza múltiples estados para el reconocimiento de los objetos en múltiples escenas, con su posterior asociación y el cálculo de la trayectoria de estos.

En este campo se pueden tomar diversos acercamientos al problema, utilizando técnicas clásicas como Kalman Filter (KF), Extended Kalman Filter (EKF) o Unscented Kalman Filter (UKF), modelos neuronales como en [9] o modelos end-to-end que incorporan detección y tracking en un mismo modelo neuronal como es el caso de PointTrackNet [10] o modelos que realizan tracking de forma implícita como CBGS [11] que calculan la velocidad de los objetos sin devolver identificadores de estos [4.2.5](#).

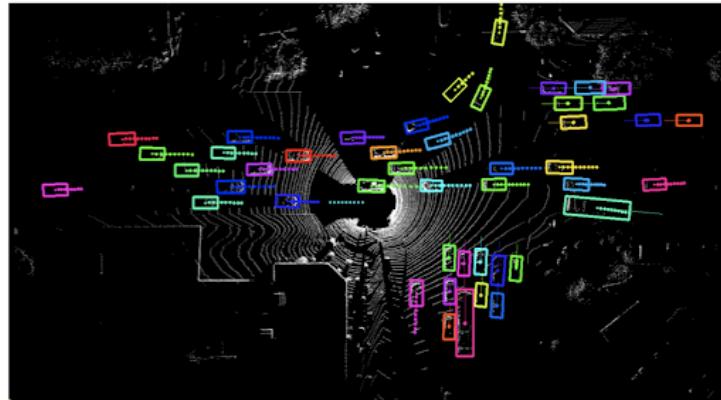


Figura 1.9: Tracking como vista de pájaro sobre una nube de puntos.

Los sistemas de detección suelen trabajar a partir de modelos en dos dimensiones que trabajan en vista de pájaro lo cual elimina lidiar con movimientos verticales, en estos se detecta la posición actual, velocidad lineal, velocidad angular y se puede definir el movimiento todo lo complejo que se desee. Tras esto también se predice los futuros estados de los objetos, manteniendo un identificador asociado a estos a lo largo de los frames analizados.

1.2.4 Fusión sensorial

Los sistemas de fusión sensorial son necesario para aumentar la precisión en los sistemas de detección y tracking, como se ha visto en 1.2.1, los sensores ofrecen información diferente al resto, por lo que el aumento de precisión es producido por el aumento en la cantidad de datos proveniente de todos los sensores. La complejidad de estas técnicas radica en el uso eficiente de todos los sensores.

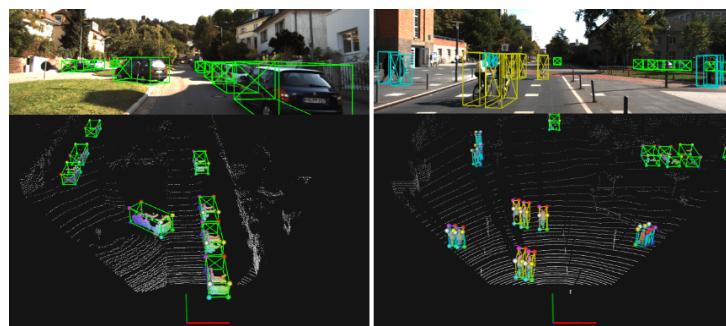


Figura 1.10: Fusión sensorial utilizando cámara y LiDAR.

Para la realización de un sistema de fusión sensorial se pueden tomar diversos acercamientos, en una early fusion se tratan todos los datos en crudo de los sensores para obtener un mejor sistema de percepción, si se trabaja con sensores tratados pero no con las detecciones finales se trata de una middle fusion, en el caso de utilizar las detecciones finales estaríamos ante una late fusion.

Entre las técnicas más utilizadas para sistemas de fusión sensorial encontramos los KF, EKF y UKF [12], pero también se comienzan a utilizar técnicas más complejas que radican en el uso de redes neuronales para la fusión.

1.3 Deep Learning

Gracias al aumento en capacidad de computo, la mejora en el procesamiento de grandes volúmenes de datos y el desarrollo de nuevos algoritmos que aprendan con estos datos, se ha producido un estallido en el uso de técnicas basadas en redes neuronales. Desde 1958 con la creación del Perceptrón se conoce de este tipo de técnicas, pero a principios del siglo XXI es cuando realmente se ve su potencial.

Las redes neuronales profundas o [DL](#) son aquellas redes neuronales con múltiples capas intermedias que permiten la extracción del conocimiento. Estas son utilizadas en ámbitos como el análisis de datos tabulares, modelos del lenguaje o sistemas de visión que es donde se centrará este estudio.

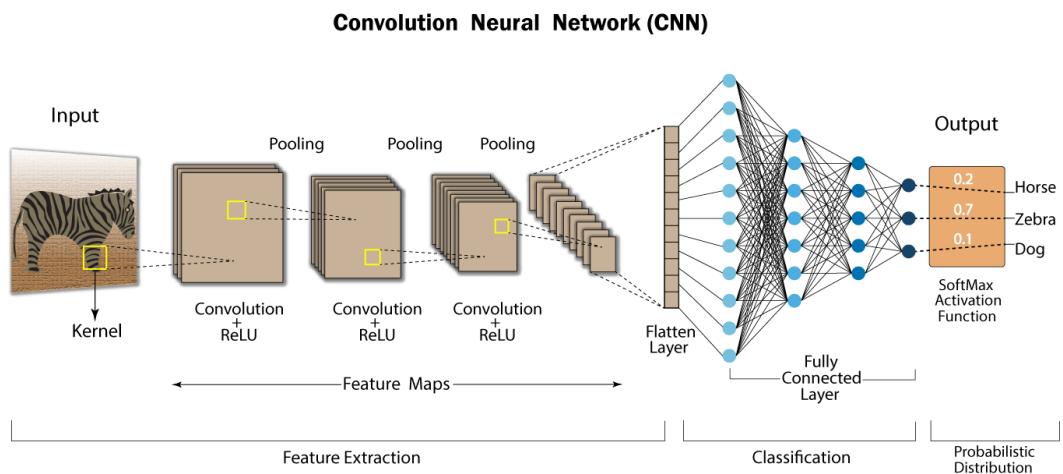


Figura 1.11: Convolutional Neural Network.

En el campo de la visión artificial las Redes Neuronales Convolucionales o Convolutional Neural Network (CNN) han conseguido una mejora importante en la precisión de los modelos. Estas redes son utilizadas principalmente para su uso con cámaras ya que su estructura tridimensional funciona de forma muy buena con las [CNN](#). Un modelo típico para uso con imágenes sería [1.11](#).

Para los sistemas de detección con [LiDAR](#) basados [Deep Learning](#) son utilizadas principalmente las [CNN](#) junto con capas Fully Connected (FC), ya que son las que obtienen un mejor rendimiento como se verá en el capítulo [4](#).

Capítulo 2

Propuesta de trabajo

La educación científica de los jóvenes es al menos tan importante, quizá incluso más, que la propia investigación.

Glenn Theodore Seaborg

Este trabajo se encuentra dentro del proyecto Tech4AgeCars perteneciente al grupo RobeSafe. En este se pretende realizar un estudio de diferentes técnicas de detección utilizando **LiDAR**, tanto con métodos clásicos como basados en **Deep Learning** para el análisis en datasets reales, en el simulador CARLA y en el coche del grupo RobeSafe.

En la arquitectura del proyecto se tiene implementado un procesamiento basado en PointPillars para una ejecución en tiempo real sobre una plataforma NVIDIA Jetson AGX Xavier [13], por lo que se pretende remplazar esta subtarea de detección utilizando **LiDAR**, dentro de la capa de percepción.

Con la implementación de la detección realizada, se pretende realizar una fusión sensorial entre cámara y **LiDAR** junto con un compañero del grupo RobeSafe que se encuentra realizando un TFG de detección 3D utilizando cámara [14].

Tras una fusión y sin la posibilidad de analizar el sistema de percepción a crear, se trabajará por último en un proyecto junto con el Karlsruhe Institute of Technology (KIT) llamado Autonomous Driving Development Kit (AD DevKit) que tratará de analizar un **Automated Driving System**, en concreto se trabajará en el apartado de percepción del kit de desarrollo, todo ello esperando que este proyecto no solo sea útil para los compañeros del proyecto sino para cualquier desarrollador que utilice el simulador CARLA.

Capítulo 3

Sistemas clásicos de percepción con LiDAR

El placer más noble es el júbilo de comprender.

Leonardo da Vinci

Mientras que se tienen múltiples tipos de técnicas de percepción tanto clásicas como basadas en DL, el uso de técnicas clásicas utilizando únicamente LiDAR no abundan, por lo que se presentan las técnicas estudiadas e implementadas en el simulador CARLA que permiten la detección de los objetos del entorno.

3.1 Voxelización

Las nubes de puntos generadas por el LiDAR pueden ser de hasta 1.300.000 puntos por segundo en un LiDAR de 64 haces [5] lo que implicaría el análisis de una gran cantidad de datos en tiempo real lo que puede no ser muy viable ya que se tiene una capacidad de computo limitada en un vehículo.

Para ello se utiliza la voxelización, esta no solo es utilizada en sistemas de percepción, sino que también es utilizada en imágenes volumétricas de ámbito médico, para la representación del terreno o en el pipeline gráfico de un ordenador. Esta técnica trata de reducir la cantidad de datos en memoria a la vez que reduce el computo al reducir la resolución de la escena. Por lo que se puede entender como un proceso de discretización del entorno.

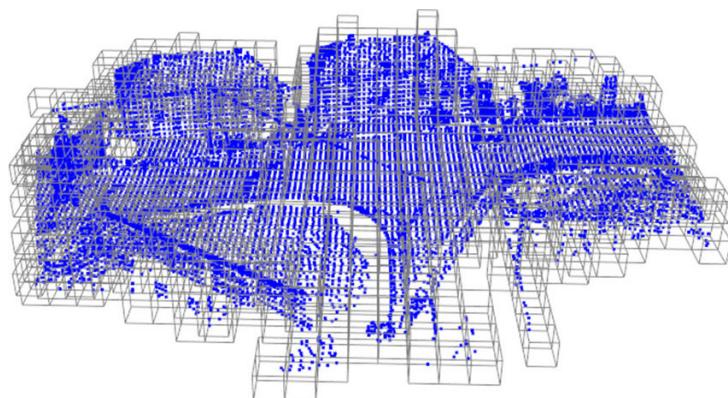


Figura 3.1: Entorno 3D voxelizado.

Trabajando con nubes de puntos, la voxelización sigue los siguientes pasos:

1. Definición del tamaño del vóxel, lo que sería un vector tridimensional.
2. A partir del tamaño del vóxel se divide la escena en un conjunto de ortoedros u vóxeles.
3. Si se encuentra un punto del Light Detection and Ranging (LiDAR) dentro de un vóxel este de activa

Esta técnica como se verá en el capítulo 4, también se utiliza en diversos modelos basados en DL, esto se hace para trabajar de forma similar a lo que sería la estructura de una imagen que se encuentra compuesta por píxeles en vez de por vóxeles.

3.2 RANSAC-3D

Para la detección de los objetos del entorno no es necesaria la información de los puntos que inciden en el suelo, por lo que una de las técnicas utilizadas para la selección del plano perteneciente al suelo es Random Sampling and Consensus (RANSAC)-3D.

El algoritmo **RANSAC** [15] tiene una funcionalidad similar a la regresión lineal, ambos algoritmos a partir de un conjunto de datos hayan la relación lineal entre dos características. La creación de este algoritmo tenía como finalidad el ajuste de datos experimentales, el uso en el análisis de escenas y generación automática de mapas.

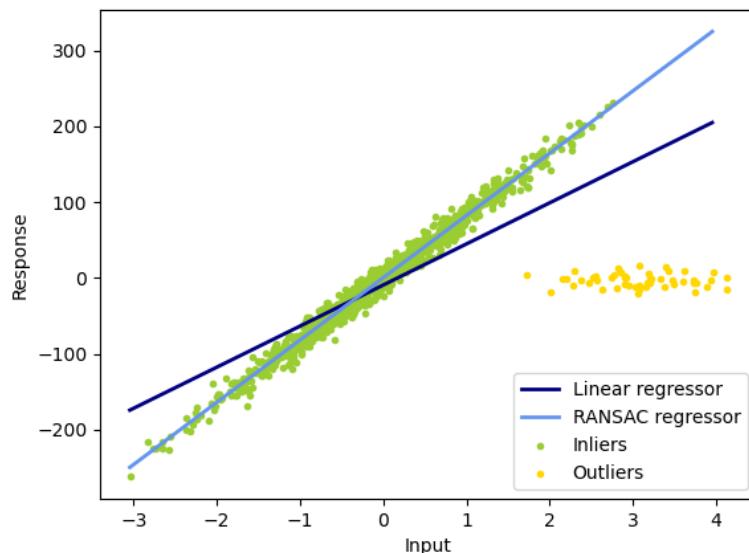


Figura 3.2: Aplicación de RANSAC para detección de outliers.

La idea principal del algoritmo **RANSAC** es la generación de rectas a partir de 2 o más puntos para aceptar como la mejor recta aquella que contenga más puntos entre un límite seleccionado, y esto es repetido un número arbitrario de veces. Esta recta será la que contenga los puntos asumidos como normales o inliers, y el resto de puntos son asumidos como anómalos o outliers. El algoritmo completo sería el siguiente 3.1.

Input

data Conjunto de observaciones
 model Modelo que explica las observaciones
 n Mínimo número de puntos necesarios para estimar un modelo
 k Número de iteraciones del algoritmo
 t Valor límite que indica que puntos se encuentran bien estimados
 d Número de puntos cercanos que asegura que el modelo sea válido

Output

bestFit Parámetros del modelo que ajustan de mejor manera a los datos
 $iterations \leftarrow 0$
 $bestFit \leftarrow \text{null}$
 $bestError \leftarrow \infty$
while $iterations < k$ **do**
 | $maybeInliers \leftarrow n$ puntos seleccionados aleatoriamente
 | $maybeModel \leftarrow$ modelo que se ajusta a $maybeInliers$
 | $alsoInliers \leftarrow$ set vacío
 | **for** cada punto que no se encuentre en $maybeInliers$ **do**
 || **if** error de ajustar el punto a $maybeModel < t$ **then**
 || | añadir punto a $alsoInliers$
 || **end**
 | **end**
 | **if** número de puntos en $alsoInliers > d$ **then**
 | | $betterModel \leftarrow$ parámetros del modelo sobre el que han sido ajustados los puntos de
 | | $maybeInliers$ y $alsoInliers$
 | | $thisErr \leftarrow$ medida de como de bien han sido ajustado los puntos;
 | | **if** $thisErr < bestErr$ **then**
 | | | $bestFit \leftarrow betterModel$
 | | | $bestErr \leftarrow thisErr$
 | | **end**
 | **end**
 | $iterations \leftarrow iterations + 1$;
end

Algoritmo 3.1: Algoritmo RANSAC

En el caso de las nubes de puntos que devuelve el [LiDAR](#), se trabaja en un entorno tridimensional, por lo que no funciona de la misma manera dicho algoritmo, se utiliza una variación, [RANSAC](#)-3D como se ve en [3.3](#), que en vez de trabajar con datos en 2D se trabajan en 3D por lo que en vez de ajustar un modelo lineal se ajusta como un plano, por lo que como mínimo se necesitan tres puntos para generar un posible modelo ya que es el mínimo número de puntos para generar un plano, el resto funciona de forma similar definiendo el límite de distancia, iteraciones...

Como se explicó, los resultados suelen ser similares a una regresión lineal en un entorno bidimensional, pero en este caso no sería del todo cierto, ya que el plano que abarca más puntos suele ser en la mayoría de los casos el correspondiente al suelo. Esto implica una modificación de la regresión lineal a las tres dimensiones, lo sería una regresión ajustada a un plano, esta generaría en la mayoría de las situaciones un plano que se encontraría por encima del suelo, ya que se trataría de minimizar una métrica de error al plano (distancia euclídea, manhattan, minkowski, hamming...), por lo que los objetos de la escena conseguirían levantar el plano para minimizar el error de este a los puntos correspondientes a los objetos.

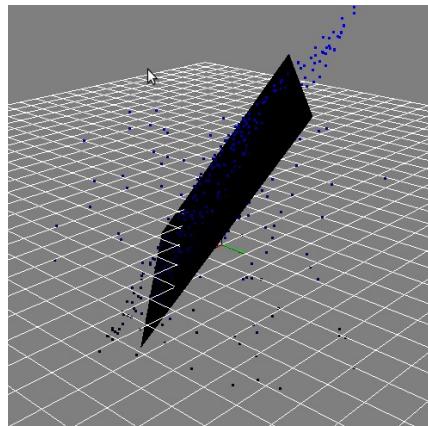


Figura 3.3: Aplicación de RANSAC-3D.

Al tener el resto de puntos por encima del suelo, nos encontraría ante una situación en la que el algoritmo **RANSAC**-3D ajustaría como inliers los puntos correspondientes al suelo y el resto se detectarían como outliers, por lo que es este el algoritmo seleccionado al ajustarse de la mejor manera a la tarea necesitada.

3.3 KD-tree

Tras la eliminación del suelo en la nube de puntos podemos encontrarnos con que los diferentes objetos del entorno se encuentran separados, ya que el suelo era el elemento unificador de la mayoría de puntos de la escena. Teniendo esto, es necesaria de una técnica que sea capaz de agrupar los puntos más cercanos de para que se agrupen por distancia, ya que si se hace comprobando cada punto con el resto se obtendría una complejidad de $O(n^2)$.

Teniendo en cuenta el coste computacional de los algoritmos de clustering y al trabajar con tantos puntos, alrededor de 1.000.000 por segundo y sabiendo que un **LiDAR** suele trabajar a 10 Hz, es muy recomendable aplicar una voxelización si no se aplicó previamente en la eliminación de los puntos incidentes en el suelo.

Para el clustering, se podría utilizar el algoritmo K Nearest Neighbors (KNN), pero esto produciría clústeres no válidos al encontrarse objetos con pocos véxeles o con demasiados lo que produciría clústeres incompletos y otros mal formados sin no se tiene en cuenta una distancia máxima entre véxeles.

El KD-tree [16] es una estructura de datos que con un eficiente uso de memoria, es capaz de hacer búsquedas en un entorno K dimensional con una complejidad media de $O(\log n)$, esto lo convierte en una gran estructura para trabajar con datos en un entorno tridimensional, como es el caso de las nubes de puntos o de véxeles. Un KD-tree tiene una estructura similar a un árbol binario, la eficiencia de la estructura radica en la ordenación del mismo, donde en cada altura del árbol se ordena según una dimensión iterativamente.

Antes de analizar en profundidad la estructura KD-tree, es necesario comprender los árboles binarios, tanto su uso, como su utilidad. Los árboles binarios son una estructura de datos donde cada nodo tiene otros dos nodos hijos, referidos como hijo izquierdo e hijo derecho. La utilidad de la estructura radica en la forma en la que se pueden guardar los datos, mientras que para buscar un valor en una lista, es necesario iterar por todos ellos o hasta que se encuentre con una complejidad máxima de $O(n)$, un KD-tree tiene una complejidad máxima es de $O(\log_2 n)$.

Input

tree Árbol binario ordenado
key Clave del nodo buscado

Output

node Nodo buscado

```

node ← null;
currentNode ← nodo raíz de tree
while currentNode ≠ null do
    if clave de currentNode = key then
        | node ← currentNode
        | break;
    end
    if clave de currentNode < key then
        | currentNode ← hijo derecho de currentNode
    else
        | currentNode ← hijo izquierdo de currentNode
    end
end

```

Algoritmo 3.2: Búsqueda en árbol binario ordenado

En el caso del árbol de la figura 3.4 para buscar el número 7:

1. Se empieza por el nodo con valor 8
2. Al ser $7 < 8$ se pasa al hijo de la izquierda
3. Como $7 > 3$ se salta al hijo de la derecha
4. Teniendo el nodo con valor 6, siendo menor que 7 se coge el hijo de la derecha
5. Por último se llegó al nodo con valor 7 requerido

Teniendo 9 nodos solo ha sido necesario analizar 4 nodos que la peor situación con este árbol.

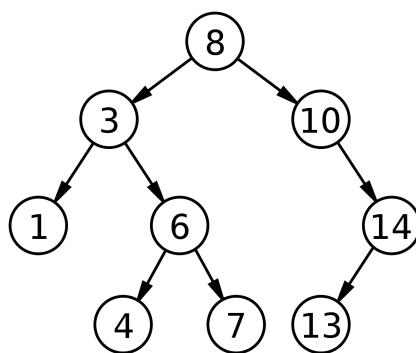


Figura 3.4: Árbol binario ordenado.

Al contrario que los árboles binarios, un KD-tree es capaz de un número K de dimensiones, por lo que hay una diferencia principal que es la rotación entre la dimensión sobre la que se ordena en cada altura del árbol. Esto produce que la forma de inserción 3.3 y búsqueda sea modificada.

Input

tree KD-tree
 node Nodo a introducir
 k Número de dimensiones del árbol

Output

tree KD-tree con el nodo introducido

currentNode \leftarrow nodo raíz de *tree*

depth \leftarrow 0

while *currentNode* \neq null **do**

x \leftarrow *depth* mod *k*

if valor de *currentNode* en la dimensión *x* $<$ valor de *node* en la dimensión *x* **then**
 | *currentNode* \leftarrow hijo derecho de *currentNode*

else

 | *currentNode* \leftarrow hijo izquierdo de *currentNode*

end

depth \leftarrow *depth*+ 1

end

currentNode \leftarrow *node*

Algoritmo 3.3: Inserción en KD-tree

Lo que produce esta forma de guardar los datos en el árbol, es que según se aumenta la profundidad en el árbol, la región de los nodos hijos es cada vez menor, lo que permite una más sencilla agrupación y estudio de los datos por regiones en un entorno K dimensional. Como se ve en la figura 3.5 el espacio bidimensional va siendo dividido por regiones, esto es gracias a que cada nodo divide en dos el espacio sobre el que se encuentran sus hijos, lo cual es una perfecta manera de agrupar los puntos en clústeres utilizando esta estructura, tal y como se detalla en el algoritmo 3.4

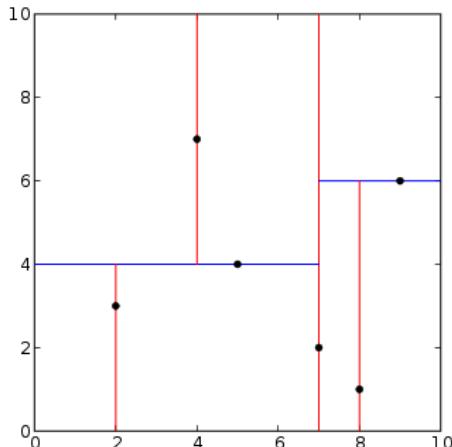


Figura 3.5: Espacio bidimensional dividido por un KD-tree.

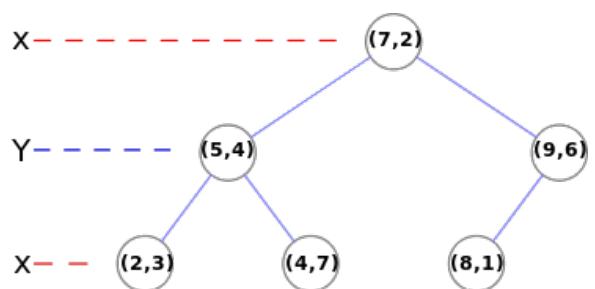


Figura 3.6: Estructura de un KD-tree de dos dimensiones.

Input

points Nube de puntos del LiDAR
 id_node Id del punto sobre el que se va a comenzar el clúster
 node Nodo sobre el que buscar un clúster
 processed Vector de booleanos de tamaño igual al número de puntos
 tree KD-tree
 distance Distancia máxima a los puntos del clúster
 k Número de dimensiones del árbol

Output

cluster Conjunto de los puntos perteneciente al clúster

```

Function proximity (points, id_node, node, cluster, processed, tree, distance, k):
  processed[id_node] ← true
  añadir a cluster points[id_node]
  indexList ← search (points[id_node], tree, distance, k) for index en indexList do
    if processed[index] = false then
      | proximity (points, index, cluster, processed, tree, distance, k)
    end
  end

Function search (node, tree, distance, k):
  indexList ← lista vacía
  searchNodes (node, tree, 0, distance, indexList, k)
  return indexList

Function searchNodes (node, tree, distance, depth, indexList, k):
  if tree ≠ null then
    if distancia entre el nodo raíz de tree y node < distance then
      | añadir índice del nodo raíz de tree a indexList
    end
  end
  x ← depth mod k
  if valor de node en la dimensión x – distance < valor del nodo raíz de tree en la dimensión
  x then
    | searchNodes (node, árbol izquierdo de tree, depth+1, distance, indexList, k)
  end
  if valor de node en la dimensión x + distance > valor del nodo raíz de tree en la dimensión
  x then
    | searchNodes (node, árbol derecho de tree, depth+1, distance, indexList, k)
  end

```

Algoritmo 3.4: Cluster por distancia en KD-tree

Gracias a la estructura KD-tree, se puede reducir la cantidad de nodos o puntos analizados, ya que cada punto no tiene que ser estudiado con el resto sino que solo se estudian los puntos que están una región cercana dentro del radio máximo de distancia definido. Lo que produce una complejidad de $O(n * \log n)$ para la construcción de la estructura más la complejidad $O(n * \log n)$ de la función de clustering por distancia, por lo que en total se tendría una mejora de complejidad de $O(n^2)$ a $O(n * \log n)$.

La eficiencia de esta estructura en ciertas tareas, ha producido que a pesar de ser una técnica del año 1975, se siga estudiando para su utilización junto a KNN [17], aumentar su rendimiento con datos preordenados [18] o la paralelización de su construcción y técnicas como KNN [19].

3.4 Filtrado previo y posterior a la detección

Tras la obtención de las detecciones por parte de los diversos algoritmos clásicos podemos encontrarnos ante diferentes problemas con dichas detecciones.

Estas pueden generar clústeres con pocos o demasiados puntos, lo que puede resultar en clústeres incorrectos. Aquellos con pocos puntos pueden identificar objetos lejanos u objetos que no son necesarios para el entendimiento de la escena, por otra parte, aquellas detecciones con muchos puntos pueden identificar camiones, vehículos de construcción o simplemente objetos muy cercanos, pero también es muy normal que las construcciones sean detectadas por lo que hay que filtrar tanto por un número máximo como mínimo de puntos para obtener mejores detecciones.

Otra práctica para el filtrado, es el ajuste a unos tamaños prefijados en todas las dimensiones, lo cual elimine aquellos objetos que no son similares a los vehículos que se desean detectar.

Estas técnicas de filtrado no solo se pueden utilizar tras la obtención de las detecciones, sino que la nube de puntos obtenida del [LiDAR](#) es posible filtrarla, para que así solo se trabaje con una Region of Interest (RoI), ya que a partir de cierta distancia las detecciones no van a ser muy precisas, para ello se puede filtrar por distancia al vehículo. Además, un filtrado que permita trabajar únicamente con la parte delantera y trasera del coche, aporta una reducción en el computo de los algoritmos, a la vez que se reducen las falsas detecciones.

Capítulo 4

Sistemas de percepción con LiDAR basados en Deep Learning

Si no conozco una cosa, la investigaré.

Louis Pasteur

Los sistemas de percepción pertenecientes al estado del arte o State of the Art (SOTA) se encuentran basados en [DL](#), esto no es diferente en los sistemas de percepción basados en [LiDAR](#), por lo que en este capítulo se presentan los datasets disponibles para el entrenamiento y evaluación de los modelos, las diferentes arquitecturas [SOTA](#) para detección con [LiDAR](#) y la herramienta utilizada para la evaluación, entrenamiento y pruebas realizadas sobre los modelos.

4.1 Principales datasets

Para el desarrollo de un modelo de percepción basado en [DL](#), es siempre necesario un conjunto de datos anotados o dataset sobre el que un modelo pueda aprender a partir de estos. En los últimos años muchas compañías han lanzado datasets para poder entrenar y validar sus modelos, además de que se muchos de estos datasets han sido publicados Open-Source para fomentar el desarrollo de nuevas técnicas.

Entre los datasets para conducción autónoma encontramos: A2D2 Dataset [20], Argoverse Dataset [21], CityScapes Dataset [22], KITTI Vision Benchmark Suite [23], Level 5 Open Data [24], nuScenes Dataset [25] o Waymo Open Dataset [26] entre otros.

En este apartado se van a presentar tres de los datasets más importante en la industria del automóvil, como son KITTI, nuScenes y Waymo dataset.

4.1.1 KITTI

La suite de evaluación de KITTI [23] es un sistema de evaluación para vehículos autónomos, donde se ha desarrollado una plataforma de referencia para tareas de visión estereoscópica, flujo óptico, odometría visual/[SLAM](#) y detección de objetos 3D.

Este dataset fue presentado en 2012 en conjunto por el [KIT](#) y el [TTIC](#), iniciando lo que años después despertaría un interés en la creación de datasets Open-Source para sistemas [ADAS/ADS](#). Al ser el

primer dataset con reconocimiento internacional, fijó las bases de lo que sería el futuro de los sistemas de evaluación, además que ha sido desde su salida uno de los sistemas de evaluación más estandarizados en el desarrollo de técnicas de conducción autónoma, como se puede ver en sus más de 7.000 citas.

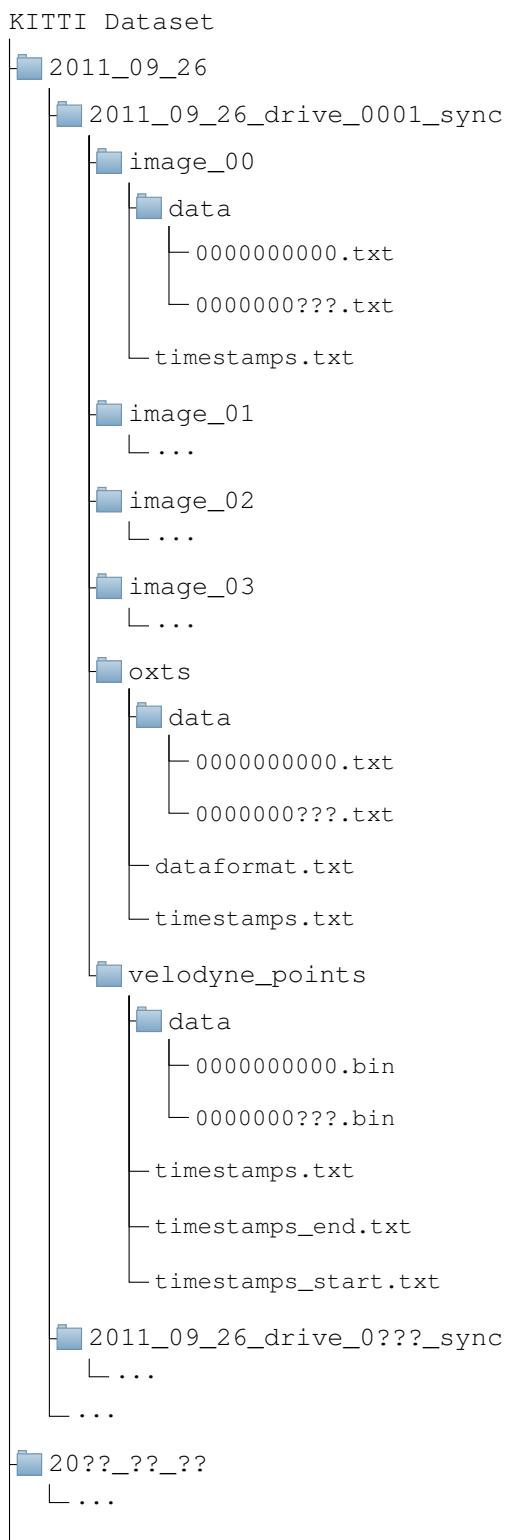


Figura 4.1: Estructura del dataset de KITTI.

El sistema de percepción del vehículo utilizado para la grabación de los datos se encuentra compuesto de:

- 2 sistemas de cámaras estéreo de una resolución de 1240 x 376 píxeles.
- 1 LiDAR Velodyne HDL-64E capaz de generar más de un millón de puntos por segundo gracias a sus 64 haces láser.
- 1 sistema de localización SOTA compuesto por GPS, GNSS, IMU y corrección de señales RTK.

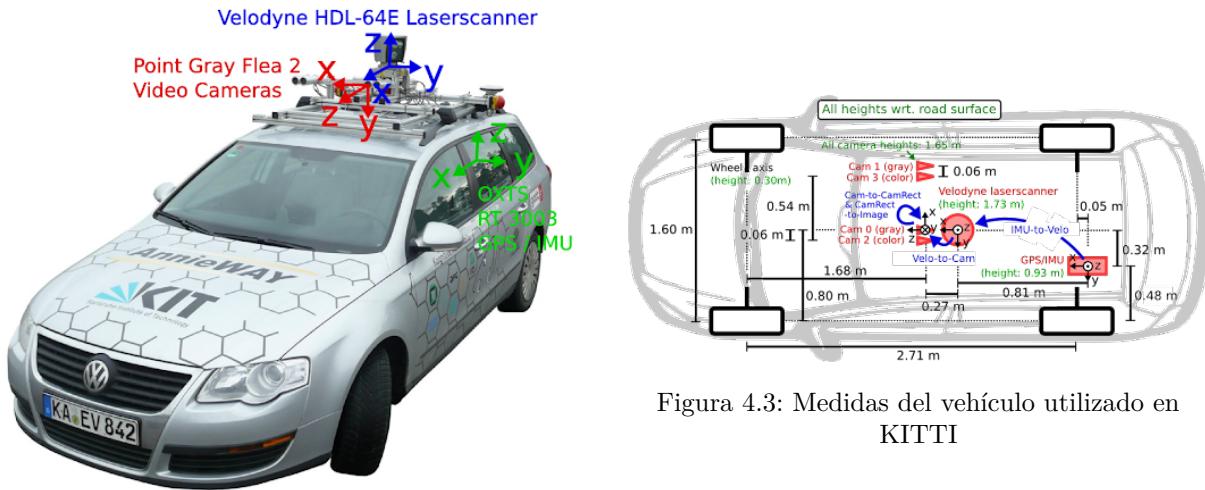


Figura 4.2: Vehículo utilizado en KITTI.

La estructura del dataset es tal y como se muestra en la figura 4.1. Dentro de este los datos se dividen según el día de grabación y la escena, y dentro de cada escena se tienen las imágenes de la cámara estéreo monocromática (image_00, image_01) y RGB (image_02, image_03), información de todo el sistema de localización del vehículo (oxts) y las nubes de puntos provenientes del LiDAR (velodyne_points).

En 2020 tras el éxito que ha sido este dataset se anuncia y se abre al uso KITTI-360 [27] la evolución del dataset KITTI, el cual trae mejoras añadiendo dos cámaras de ojo de pez para una obtención de visión 2D en todos los ángulos del vehículo, y una unidad láser adicional SICK. Además se incluyen en el dataset bounding boxes 3D de todo el entorno, anotaciones de instancia a partir de la información del LiDAR y anotaciones de confianza sobre el vídeo tomado por las cámaras, entre otras.

4.1.1.1 Análisis de la estructura del GT y las PCLs de KITTI

Al estar basado este trabajo en la detección utilizando LiDAR, se estudia la forma en la que se guardan los datos de las nubes de puntos junto con la representación del ground truth, para ello se analiza tanto la carpeta de velodyne_points como el archivo tracklets.xml.

Para el trabajo con bounding boxes 3D, es posible utilizar únicamente el kit de detecciones 3D que contiene un conjunto de txt con la información de los objetos por cada barrido. La información de estos archivos contiene la información de: tipo, truncado, ocluido, bounding box 2D, dimensiones y localización, toda esta información se encuentra de la misma manera en el archivo tracklets.xml, pero la información de rotación se encuentra dividida en un ángulo alpha y otro rotation_y, en vez de rotación por cada uno de los tres ejes, de estos se estudiará más en profundidad en el capítulo 6.

Se utiliza el archivo tracklets.xml ya define el ground truth de la escena pero no por cada barrido del **LiDAR**, sino que se guarda por cada escenario, lo que se traduce en un estructura se puede utilizar tanto para detección como para seguimiento de los objetos, todo ello como un archivo con formato **XML**.

Se ha decidido crear un programa que lea la nube de puntos indicada y que marque en un entorno 3D donde se encuentran los objetos de la escena. Esto podría ser realizado mediante el devkit que KITTI ofrece, pero este se encuentra únicamente de forma oficial en Matlab, por lo que como los modelos de **DL** a utilizar se encuentran utilizando Pytorch, se deberían de analizar los datos utilizando Python.

La figura 4.4 muestra la estructura de los principales atributos del archivo tracklet_labels.xml analizado, en este encontramos el número de los diferentes objetos del escenario, tras esto se analiza cada objeto a través de los diferentes barridos, mostrando primero las características de los objetos que se mantienen en el tiempo, como las dimensiones o el tipo de objeto y tras esto se guarda la información de posición, rotación, etc.

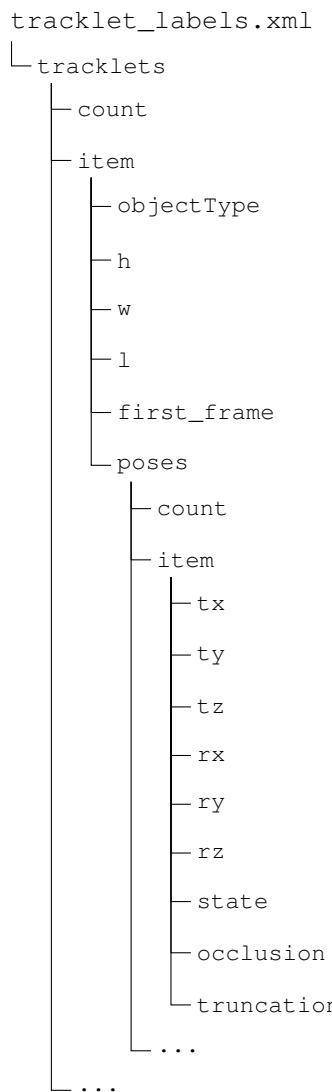


Figura 4.4: Estructura del archivo tracklet_labels.xml.

Tras dicho análisis, utilizando la librería MayaVi para la visualización del entorno 3D, NumPy para las transformaciones en el espacio y Beautiful Soup para la lectura del **XML**, se visualizan múltiples escenas del dataset.

Cabe destacar que como KITTI está realizado para que la parte de percepción se trabaje principalmente con la cámaras frontales del vehículo, aún teniendo información por los laterales y la parte trasera gracias al LiDAR, esta no se tiene en cuenta en el ground truth, razón por la que en la imagen 4.5 los vehículos de la parte trasera no se muestra.

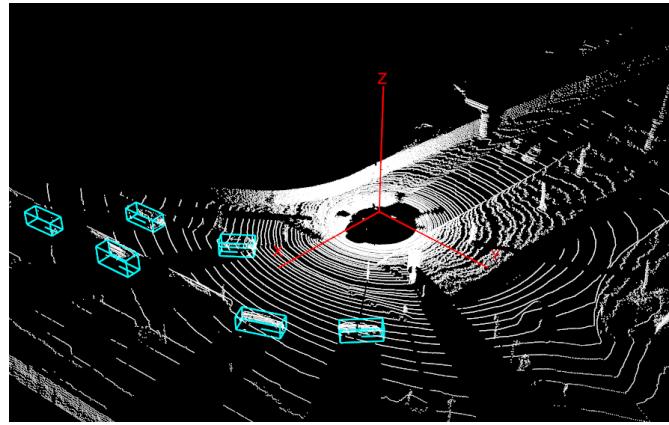


Figura 4.5: Visualización de una nube de puntos de KITTI junto con su ground truth.

Todo el código utilizado para la visualización de los objetos junto con unos escenarios de ejemplo son accesibles en: https://github.com/Javier-DlaP/Display_kitti_pcl_annotations

4.1.2 Waymo

Waymo Open Dataset [26] es el dataset liberado de forma Open-Source por parte de Waymo y Google para la aceleración del desarrollo de tecnologías de conducción autónoma.

La propuesta de este dataset es la oferta de un gran número de anotaciones de alta calidad tanto 2D como 3D, que además contienen información de seguimiento. Se han utilizado múltiples ciudades para sus escenarios grabados, como son: San Francisco, Mountain View, Los Angeles, Detroit, Seattle y Phoenix. Además se trabaja con multitud de entornos y condiciones ambientales como son: construcciones, atardeceres, noches, días lluviosos, etc.

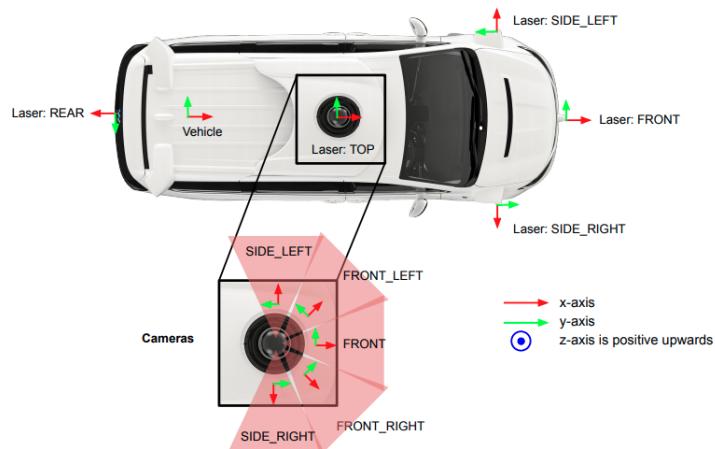


Figura 4.6: Vehículo utilizado en Waymo.

Actualmente el dataset ofrece 1.950 escenas [28] que fueron aumentadas de las 1150 escenas en la salida de la primera revisión del paper, de 20 segundos cada una, con información recogida a 10 Hz proveniente de los sensores, lo que implican 390.000 frames. Toda esta información es recabada de:

- 1 **LiDAR** de medio alcance.
- 4 **LiDAR** de corto alcance.
- 5 cámaras alrededor del vehículo.

Como se ve en la figura 4.6, el acercamiento de la compañía entorno a la construcción del vehículo no utiliza **Radar**, depende únicamente de cámaras y **LiDAR** para el apartado de percepción. Por lo que no se puede trabajar con **Radar** en el dataset, además de que no se tiene información de ningún sistema de localización, ya que dicho dataset se encuentra especializado en tareas de percepción y seguimiento de los objetos de la escena.

Es importante tener en cuenta que en este dataset, la nube de puntos procedente del **LiDAR** no utiliza un sistema de coordenadas cartesiano, sino un sistema de coordenadas esférico, donde las coordenadas (x, y, z) son reemplazadas por (distancia, azimuth, inclinación).

$$\text{distancia} = \sqrt{x^2 + y^2 + z^2}$$

$$\text{azimuth} = \text{atan2}(y, x)$$

$$\text{inclinación} = \text{atan2}(z, \sqrt{x^2 + y^2})$$

Mientras que en otros datasets se tienen multitud de clases, muchas de ellas indistinguibles unas de otras, Waymo utiliza únicamente 4 tipos de objetos diferentes. En las 11,8 millones de bounding boxes 2D se encuentran vehículos, peatones y ciclistas, mientras que en las 12,6 millones de bounding boxes 3D se añaden además la detección y seguimiento de señales.

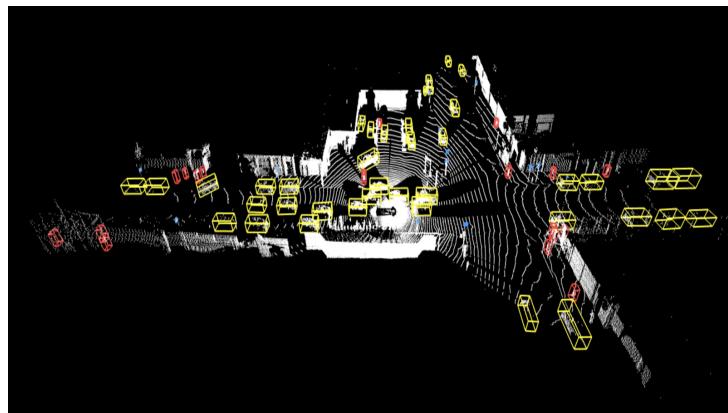


Figura 4.7: Nube de puntos con las diferentes clases del dataset de Waymo.

En la figura 4.7 encontramos las diferentes clases a detectar y seguir en el dataset, teniendo en amarillo los coches, rojo los peatones, azul las señales y rosa los ciclistas.

Waymo ofrece un dataset muy completo si solo se desea trabajar con tareas de percepción, compitiendo cara a cara con los datasets más importantes como son KITTI, nuScenes y Argoverse, teniendo además uno de los sistemas de evaluación para tareas de percepción orientadas a vehículos autónomos, más grandes y con más anotaciones que se pueden encontrar de forma Open-Source.

4.1.3 nuScenes

NuScenes Dataset [25] se presenta como una mejora a al dataset KITTI lanzado en 2012. Esta mejora no solo radica en la calidad de los datos sino en el número de diferentes situaciones disponibles, de la misma manera que Waymo, ofreciendo situaciones nocturnas y días lluviosos.

En comparación con otros datasets, se incluye un set más completo de sensores como son:

- Sistema de 6 cámaras 360° con una resolución de 1600 x 900.
- LiDAR de 32 haces con una frecuencia de 20 Hz, capaz de generar hasta 1,4 millones de puntos por segundo.
- Sistema de 5 Radar con una distancia máxima de 250 metros y una frecuencia de 13 Hz.
- Sistema de localización compuesto por GPS, IMU, AHRS y un sistema de posicionamiento RTK.

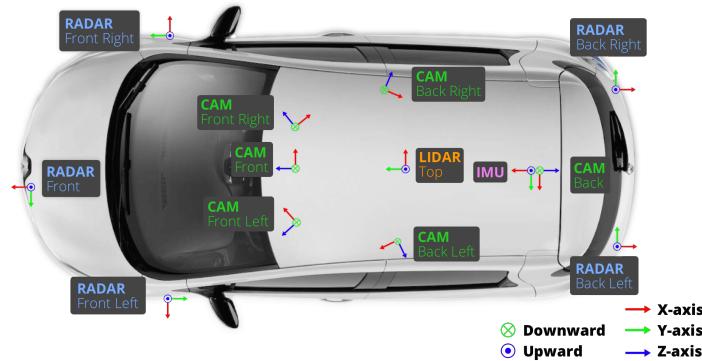


Figura 4.8: Vehículo utilizado en el dataset de nuScenes.

NuScenes tiene 23 clases diferentes a detectar, entre ellas se encuentran como en el resto de datasets: coches, ciclistas, peatones, etc. Pero en este se encuentran además animales, diferenciación por el tipo de los peatones, como serían policías o personas en silla de ruedas, además de barreras entre otras. Esto permite agrupar las clases para detectar únicamente las más simples o servir como un sistema de evaluación con toda la información necesaria para la implementación en un sistema ADS.

El sistema de evaluación de nuScenes permite evaluación de múltiples sistemas a diferentes niveles. Las diferentes tareas a evaluar son:

- Detección de objetos 3D (10 tipos de objetos diferentes) utilizando cámara, LiDAR, Radar y la información de los mapas.
- Seguimiento de objetos 3D (7 tipos de objetos diferentes) utilizando cámara, LiDAR, Radar y la información de los mapas.
- Predicción del movimiento y de la posición de los objetos.
- Segmentación de la nube de puntos del LiDAR a nivel de punto.

El dataset completo de nuScenes ocupa medio terabyte, esto es debido a la cantidad de información que este tiene, la cual es encuentra dividida en: mapas, información de los sensores en los frames que se tienen anotaciones, información de los sensores sin anotaciones y una carpeta v1.0-trainval que contiene todos los archivos json que relacionan todo el dataset además del ground truth 4.9.

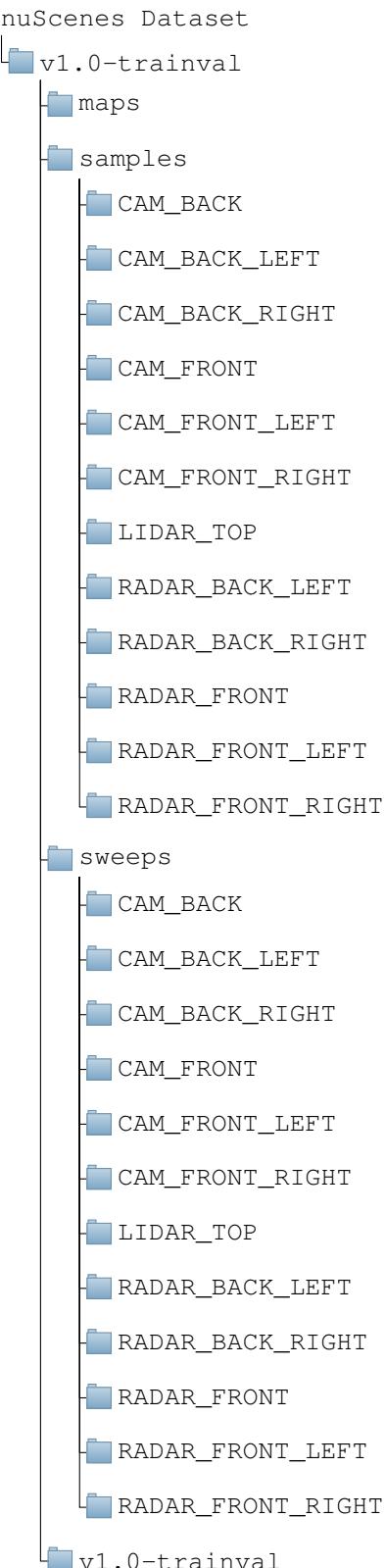


Figura 4.9: Estructura del dataset nuScenes.

Dicha estructura es para el uso de todos los sensores del dataset y los mapas, pero en el caso de que se requiera únicamente de las cámaras, nuScenes ofrece nuImages, este es el dataset de nuScenes reducido, que además cambia la estructura interna de este.

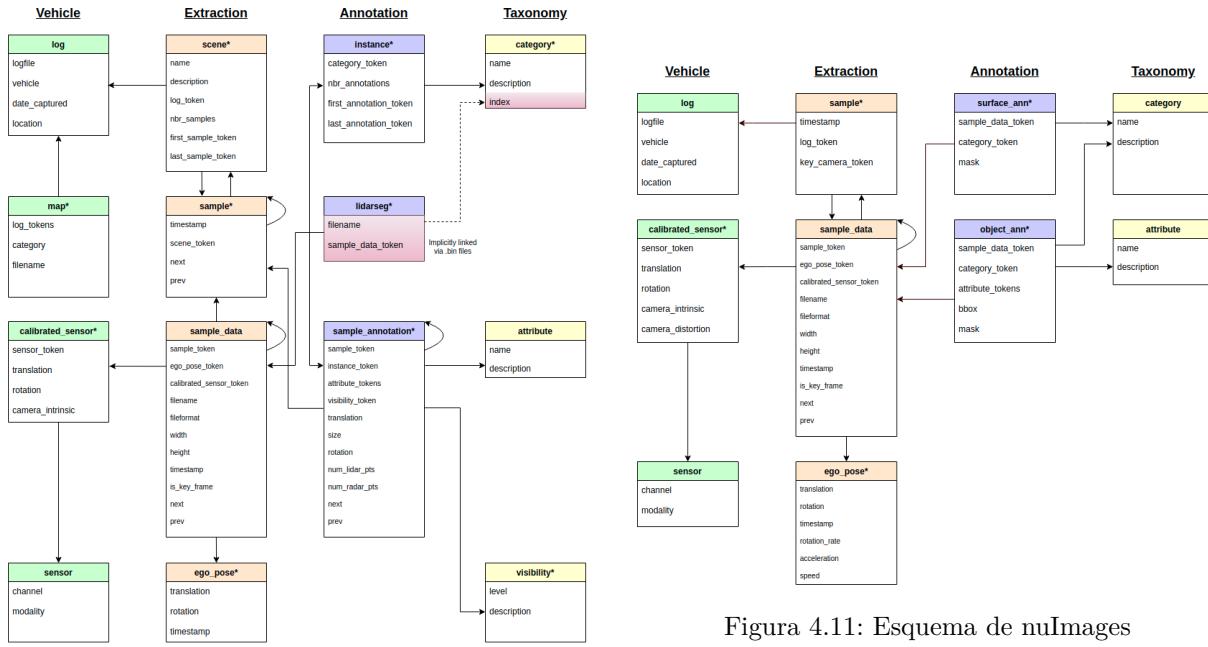


Figura 4.10: Esquema de nuScenes.

Figura 4.11: Esquema de nuImages

Las estructuras de estas versiones del dataset, no son sencillas de trabajar con los datos directamente, por lo que nuScenes ha desarrollado un devkit con el que sea más sencillo de trabajar con ambas estructuras tal y como se explicará en el siguiente apartado.

4.1.3.1 NuScenes devkit

Promover el uso de un dataset es importante en la medida que sino se tienen características nuevas en este o es difícil de utilizar, este no destacará y desaparecerá entre la cantidad de datasets que se encuentran hoy en día. Para ello, nuScenes desarrolla y lanza de forma Open-Source nuScenes devkit.

NuScenes devkit, ofrece un uso muy simplificado del dataset de nuScenes y nuImages. Ambos datasets incorporan a partir de json una estructura similar a la de una base de datos relacional en la que a partir de claves primarias, externas e índices se consigue tener un conjunto de datos normalizado, para así minimizar al máximo la redundancia de datos.

Al ofrecer una gran cantidad de datos no solo de los sensores sino de la escenas, configuraciones de calibración, visibilidad, categorías, etc. Se ha estructurado en diferentes archivos json para no tener que repetir datos por cada imagen o nube de puntos. El devkit ha sido programado para su uso en Python 3 y los tutoriales para aprender a utilizar dicho devkit se encuentran en Jupyter Notebooks, aunque también se recomienda su uso en Google Colab.

Para el aprendizaje del devkit de nuScenes se descarga la versión mini del dataset, el cual tiene la misma estructura que el dataset completo. Junto con Jupyter se estudia todo el dataset siguiendo los diferentes tutoriales que ofrece. Tras trabajar con la herramienta y acostumbrarse al uso de los tokens que relacionan los diferentes archivos json, es muy sencillo obtener toda la información requerida.

A partir del devkit es muy sencillo realizar tareas como filtrado de clases, agrupaciones, transformaciones mundo a cámara, seguimiento de objetos individualizado, etc. De esta manera se ahorra mucho tiempo en la construcción de complejas funciones ya que se encuentran insertadas en el kit de desarrollo, además de que permite ver de forma analítica el comportamiento del dataset.

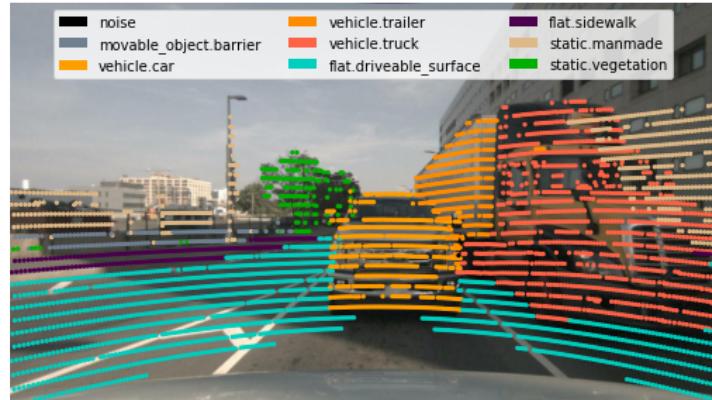


Figura 4.12: Transformación mundo a imagen de la nube de puntos segmentada en el devkit de nuScenes.

Con código tan simple como el siguiente 4.1 es muy fácil obtener la imagen de una cámara con la nube de puntos segmentada tras realizar una transformación de mundo a cámara tal y como se ve en la figura 4.12. Dichas transformaciones como se verá en el capítulo 6 no es algo inmediato, por lo es necesario cierto conocimiento del funcionamiento de las cámaras y de transformaciones geométricas.

Listado 4.1: Obtención de una nube de puntos segmentada sobre una imagen utilizando nuScenes devkit

```
from nuscenes import NuScenes

nusc = NuScenes(version='v1.0-mini',
                 dataroot='/home/javier/nuScenes-dataset-mini-v1.0',
                 verbose=True)
my_sample = nusc.sample[87]

nusc.render_pointcloud_in_image(my_sample['token'],
                                pointsensor_channel='LIDAR_TOP',
                                camera_channel='CAM_BACK',
                                render_intensity=False,
                                show_lidarseg=True,
                                filter_lidarseg_labels=[22, 23],
                                show_lidarseg_legend=True)
```

4.1.4 Comparativa entre los diferentes datasets

Tras el estudio y análisis de los datasets de KITTI, Waymo y nuScenes se obtiene un conocimiento de las arquitecturas hardware utilizadas en el estado de arte del campo de los datasets de conducción autónoma, además de los sistemas de evaluación utilizados.

Mientras que KITTI fue el primero en desarrollo en un dataset para conducción autónoma, hasta hace muy poco estaba casi estandarizado en la comparativa de modelos aplicables a conducción autónoma. Waymo y nuScenes aparecieron años más tarde con una apuesta clara en el uso de más sensores, además del estudio de los 360° del entorno y no solo de la parte frontal del vehículo como realiza KITTI.

En la parte del hardware, se ha visto que el acercamiento por parte de KITTI de tener 4 cámaras

es un enfoque equivocado para un coche real, ya que se necesita tener un rango de visión mayor en el sistema de cámaras. Por parte del uso del [LiDAR](#), se encuentra con que KITTI solo se usa la parte frontal de la nube de puntos, mientras que Waymo y nuScenes utilizan toda la nube de puntos para generar las detecciones. Entre estos dos datasets encontramos diferencias en la cantidad de [LiDAR](#) utilizados, ya que mientras nuScenes utiliza solo uno, Waymo utiliza cinco, lo cual obtiene mucha más información del entorno y una nube de puntos más densa, pero aumenta en mayor medida el precio de un prototipo con estas características. El [Radar](#) es un sensor desaparecido tanto en KITTI como en Waymo pero que se encuentra en nuScenes ofreciendo una nube de puntos de 360° que es capaz de inferir la velocidad de los objetos pero con una menor cantidad de puntos que una proveniente del [LiDAR](#).

Dataset	Año	Situaciones	Horas	Cámaras	LiDAR	RADAR	Clases	Devkit
KITTI	2012	22	1,5	4	1	0	8	Sí
Waymo	2019	1000	5,5	5	5	0	4	No
nuScenes	2019	1150	5,5	6	1	5	23	Sí

Tabla 4.1: Comparativa entre los principales datasets.

En relación a la cantidad de situaciones y de datos Waymo y nuScenes se encuentran en un estado similar con una gran cantidad de datos del entorno, mientras que KITTI se queda más atrás debido a la longevidad del dataset.

Unicamente KITTI y nuScenes cuentan con un devkit, este es utilizado para uso más simplificado del dataset, aunque en el caso de nuScenes es casi requerido su uso. El problema de KITTI en este aspecto es el uso de Matlab para el uso del devkit ya que la mayoría de la comunidad investigadora no utiliza este lenguaje para la creación de los modelos de detección, aunque se pueden encontrar de forma no oficial, repositorios Open-Source que ofrecen variantes del devkit de KITTI en otros lenguajes.

En conclusión, KITTI ha sido una gran base para la generación de la siguiente generación de datasets, aunque para sistemas de percepción algo más complejos se puede quedar corto, lo cual se ha ido mejorando con el tiempo y por esto mismo se está desarrollando KITTI-360. Por otra parte, Waymo y nuScenes ofrecen un dataset más completo, aunque nuScenes ofrece información de un sensor más y contiene la información de los mapas. Debido a esto se ha decidido estudiar más en profundidad y trabajar con los dataset de KITTI y nuScenes.

4.2 Estado del arte en detección utilizando LiDAR

Las técnicas basadas en [DL](#) que hacen uso de [CNN](#) llevan unos años siendo [SOTA](#) en el campo de la detección utilizando únicamente [LiDAR](#). En este apartado se van a estudiar los principales modelos de detección en este campo que hacen uso de estas técnicas.

4.2.1 SECOND

SECOND (Sparsely Embedded CONvolutional Detector) [29] se publica en 2018 para superar los modelos de detección 3D, utilizando únicamente [LiDAR](#). Para ello propone una arquitectura basada en tres componentes principales: extractor de características a nivel de voxel, [CNN](#) dispersa y Region Proposal Network (RPN). Precedido todo ello de una fase de preprocesamiento de la nube de puntos y con una obtención a posteriori de las salidas del modelo.

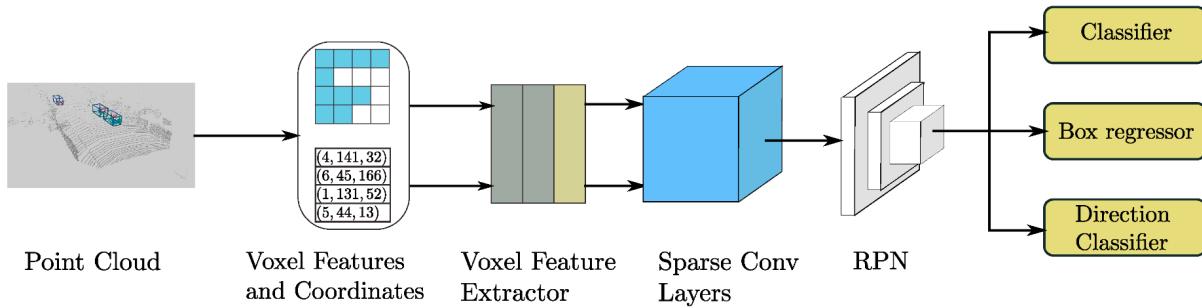


Figura 4.13: Arquitectura propuesta del detector SECOND.

Las diferentes fases del modelo que son utilizadas son las siguientes:

1. Preprocesamiento de la nube de puntos

Se comienza con la voxelización de la nube de puntos haciendo uso de una tabla hash, para su rápido acceso en memoria. Al tener que definir un tamaño de voxel fijo en función de lo que se desee detectar, se ajusta el tamaño de la **RoI**, ya que como se trabajar con KITTI solo se utiliza la parte frontal del **LiDAR**. Para la detección de coches y otros vehículos se usa $[-3, 1] \times [-40, 40] \times [0, 70.4]$ m y para el modelo más pequeño se utiliza $[-3, 1] \times [-32, 32] \times [0, 52.8]$ m para no tener que computar tantos véxeles como el modelo completo. Todas las versiones del modelo utilizan un tamaño de voxel fijo de $[0.4, 0.2, 0.2]$ m y por cada uno de ellos se guarda un máximo de 35 puntos.

2. Extractor de características a nivel de voxel

Se utiliza una capa Voxel Feature Encoding (VFE) tal como se presenta en el detector VoxelNet [30] para extraer las características. Para ello se utiliza una Fully Connected Network (FCN) compuesta de capas **FC**, una capa que aplica batch normalization y una Rectified Linear Unit (ReLU) de salida para extraer las características a nivel de punto y se aplica max pooling para la obtención de las características a nivel de voxel.

3. Extractor convolucional disperso

El uso de Sparse Convolutional Networks ofrece una mejora en rendimiento de al no computar una salida si no hay una entrada dada. Al trabajar con una nube de puntos voxelizada se consigue un mejoramiento en el rendimiento al no utilizar aquellos véxeles que no contienen ningún punto. Para poder aplicar este tipo de redes es necesario hallar que índices del kernel van a ser utilizados, y para ello se necesita un algoritmo que contenga las reglas para indicar que parte del kernel es utilizado.

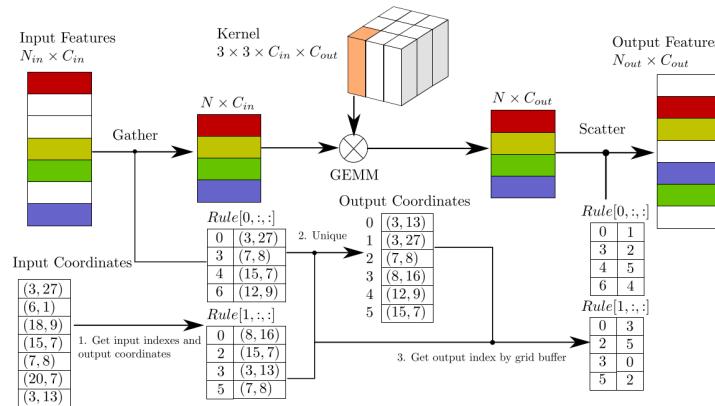


Figura 4.14: Algoritmo de convolución disperso propuesto en SECOND.

Para ello se construye una tabla de matrices de reglas para guardar los indices utilizados. Con esto en cuenta, hace falta el algoritmo que genere las reglas, SparseConvNet [31] es el modelo original que ofrece la implementación del Submanifold Convolution, una técnica dentro del campo de las Sparse Convolutional Networks que realiza la generación de reglas en CPU. En este paper se implementa un generador de reglas en GPU aprovechando la aceleración por hardware y el procesamiento paralelo para conseguir una generación de reglas en la mitad de tiempo que SparseConvNet. Este extractor es utilizado para convertir la información 3D en un formato similar a una imagen Bird's Eye View (BEV).

4. Region Proposal Network

Las RPN fueron presentadas junto con el detector Fast R-CNN [32] y son utilizadas en este modelo para que a partir del mapa de características extraído de la fase anterior, obtenga las predicciones del modelo.

5. Obtención de las detecciones

En la salida se utilizan unos tamaños fijados para las diferentes clases que son ajustados a las detecciones, a partir del centro del objeto detectado. Por cada objeto es fijado un one-hot vector para el ajuste de las cajas y otro para el ajuste de la dirección. Tras esto se aplica un límite de precisión por clase para minimizar los falsos positivos.

Este método tras su publicación consiguió convertirse en SOTA en el benchmark KITTI en su evaluador de detección utilizando LiDAR, además de realizar esto en tiempo real, con tiempos de computo de 20 Hz en su modelo completo y 40 Hz en su modelo reducido.

4.2.2 PointPillars

PointPillars se publica meses después de SECOND y no solo consigue una mejora en la precisión para la detección de objetos 3D sino que consigue esto a una velocidad de inferencia de hasta 62 Hz o hasta 105 Hz con su modelo reducido. Esto no solo permite la utilización con nubes de puntos mayores sin disminuir apenas el rendimiento sino que permite la integración de este modelo en sistemas embebidos.

Para la obtención de esta velocidad de inferencia, PointPillars elimina el uso de las capas convolucionales 3D, convirtiendo las nubes de puntos en imágenes BEV de la escena. Los principales componentes de la red que consiguen este funcionamiento son los siguientes:

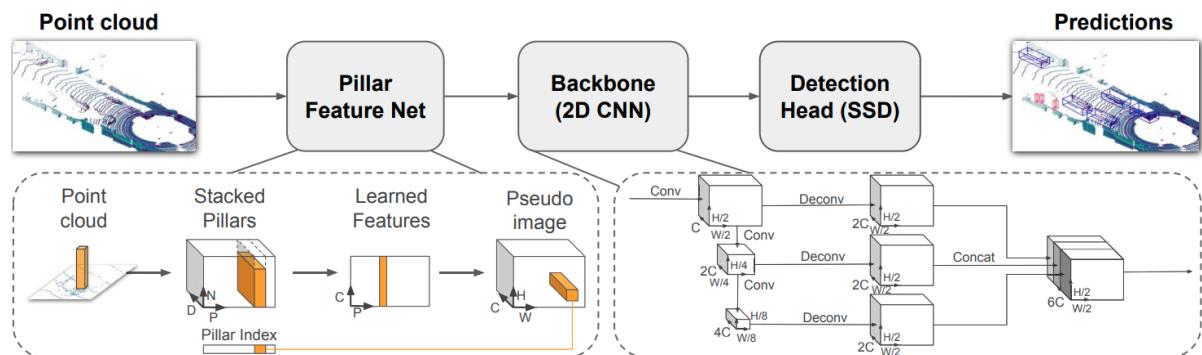


Figura 4.15: Arquitectura propuesta del detector PointPillars.

1. Nube de puntos a pseudo imágenes

Se comienza voxelizando la nube de puntos formando diferentes pilares, esto quiere decir que cada vóxel tiene un tamaño x e y fijo pero en z se tiene un tamaño infinito. Por cada punto en los vóxeles se les añade junto a las tres dimensiones con la información de posición: el grado de reflectividad, la distancia aritmética a la media de los puntos del vóxel en las tres dimensiones y el desplazamiento en x e y al centro del vóxel, por lo que se tiene un espacio de nueve dimensiones por pilar. Para limitar el cómputo, se define un tamaño máximo por pilar de [9, número de pilares no vacíos, número de puntos por pilar], se esta manera se fija un máximo de números de puntos por pilar y se aplica zero padding si el pilar se encuentra con muy poca información.

Tras esto se aplica una capa linear con batch normalization y una ReLU seguido de una operación de máximo por cada uno de los puntos del pilar para obtener un tensor de tamaño [X, número de pilares no vacíos], lo cual es pasado a una pseudo imagen pasando el número de pilares no vacíos a la altura y anchura de la imagen en función de la posición de dichos pilares. Con lo que se consigue un estructura similar a la de una imagen BEV.

2. Backbone

Se utiliza un backbone similar al de VoxelNet [30], este backbone tiene dos subcapas, una que aumenta el número de características a nivel espacial y otra que aumenta y relaciona las características de los pilares.

3. Cabeza detectora

En la salida se utiliza una Single Shot Detector (SSD) para obtener las salidas de las detecciones 3D.

Este enfoque de uso de CNN sobre nubes de puntos de la misma manera que se analizaría para imágenes, se ha visto en este paper, que es muy útil para acelerar la inferencia del modelo, además que se consigue un muy buen rendimiento para tareas de detección 3D y detección BEV 2D.

4.2.3 PointRCNN

PointRCNN [33] propone un modelo basado en dos fases para la detección de objetos 3D. Dichas fases consisten en una primera fase de generación de las detecciones 3D y otra de refinamiento de las bounding boxes 3D.

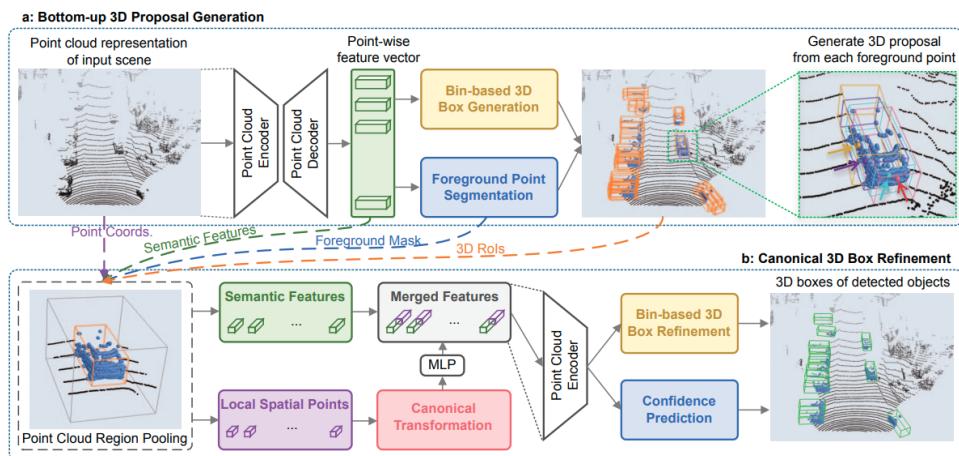


Figura 4.16: Arquitectura propuesta del detector PointRCNN.

El funcionamiento de las dos fases del modelo es el siguiente:

1. Generación de propuestas de detección 3D vía segmentación de la nube de puntos

La primera fase del modelo se basa en el backbone de PointNet++ [34] para extraer las características a nivel de punto de la nube de puntos utilizando un agrupación multiescala. Con dicho modelo junto con un método propio basado en la discretización 2D en BEV (bin-based) se hayan las múltiples detecciones 3D, pero se reducen por cada objeto aplicando Non Maximum Suppression (NMS) basado en el Intersection over Union (IoU) como BEV con un límite de 0,8 y solo las 100 mejores detecciones son mantenidas para la siguiente fase de ajuste de las detecciones 3D.

2. Refinamiento de las bounding boxes 3D

Tras la obtención de las bounding boxes 3D, se trata de mejorar el centro y la orientación de estas. Por cada una de dichas detecciones se aumenta su tamaño por un valor constante incluyendo además una mascara que diferencia aquellos puntos de la detección original al espacio aumentado. Cada una de dichas detecciones aumentadas pasan a utilizar el sistema de coordenadas propio de cada detección.

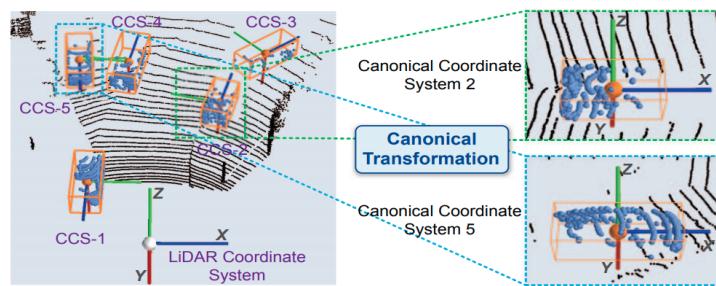


Figura 4.17: Agrupación de regiones de la nube de puntos en el modelo PointRCNN.

A partir de los puntos de cada agrupación de las características extraídas de la primera fase y de un parámetro que agrega información de la distancia al sensor para aquellas agrupaciones con menos puntos calculado como $\sqrt{x^2 + y^2 + z^2}$, se introducen en múltiples capas FC para obtener las características locales. Tras esto se vuelve a utilizar el modelo bin-based utilizado en la primera fase con lo que se obtienen las bounding boxes 3D aplicando nuevamente NMS sobre un IoU en BEV con un límite de 0,01 para eliminar solapamientos.

Al encontrarse basado este método en dos subredes, sufre de un tiempo de computo mayor, lo que se traduce en una velocidad de 12 Hz [35], pero suficiente para aplicarse en tiempo real al obtener típicamente las nubes de puntos a 10 Hz.

4.2.4 PV-RCNN

PV-RCNN [36] unifica los beneficios de las dos principales técnicas de detección de objetos utilizando LiDAR, como son: el uso de técnicas de voxelización junto con Sparse Convolutional Networks y el uso de backbones como el de PointNet [37] o métodos similares. Para ello se construye un modelo basado en tres pasos:

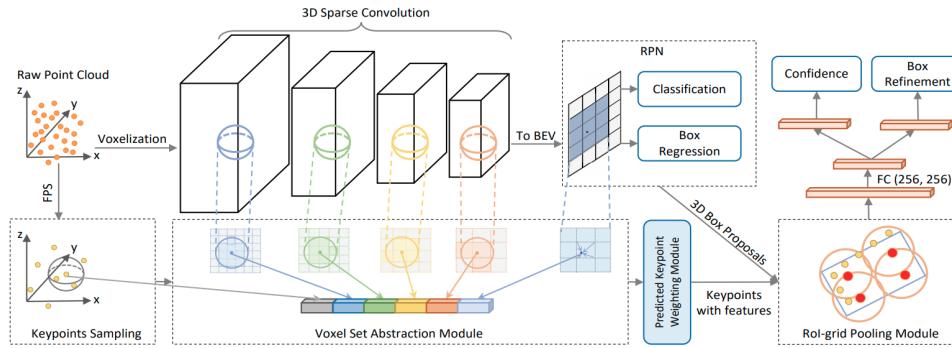


Figura 4.18: Arquitectura propuesta del detector PV-RCNN.

1. 3D Voxel CNN para la extracción de características y generación de detecciones

Se utiliza una Voxel CNN con 3D sparse convolution junto con una RPN, la cual es una elección popular en el SOTA gracias a su eficiencia, por lo que es el backbone utilizado utilizado en este modelo. Este método obtiene de forma interna características semánticas de los véxeles además de conseguir las detecciones de los objetos a partir de los tamaños prefijados o anchors.

2. Codificación de escenas de véxel a puntos clave

Primero se agregan las características de los véxeles en un conjunto de puntos clave que sirven de conexión entre la fase anterior y el refinamiento de las detecciones. Para la obtención de los puntos clave se adopta el algoritmo Furthest Point Sampling y se aplica sobre la nube de puntos, lo cual elige puntos de forma uniformemente distribuida sobre los véxeles no vacíos. Tras esto se propone el módulo Voxel Set Abstraction para codificar características semánticas de la primera fase en los puntos clave. Basándose en la premisa de que los puntos más cercanos tienen que contribuir más a la propuesta de detección se propone el módulo Predicted Keypoint Weighting.

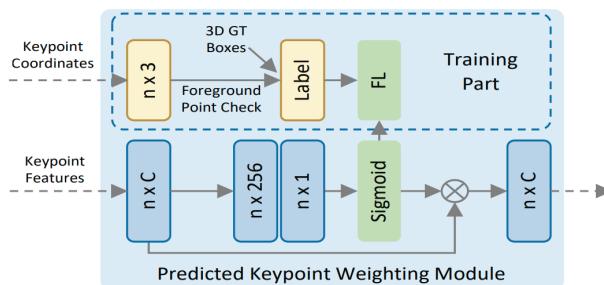


Figura 4.19: Módulo Predicted Keypoint Weighting del modelo PV-RCNN.

En dicho módulo se introducen los puntos clave codificados previamente para dar un peso a cada uno en función de su importancia para las detecciones.

3. Abstracción de características de RoI de puntos clave a cuadrícula para el refinamiento de la detección

Para el refinamiento de las detecciones se utiliza un método de abstracción de características RoI de puntos clave a rejilla. Dadas las detecciones junto con los puntos codificados y sus pesos, se agregan entorno a diferentes puntos de rejilla con un radio determinado. Tras esto se obtienen las características de las rejillas y a partir de estas, con una pequeña red de dos capas se ajusta la detección.

De la misma manera que ocurría en PointRCNN, PV-RCNN tiene una velocidad de inferencia baja de 10 Hz [35] debido a la fase de refinamiento y el procesamiento adicional de las características del backbone.

4.2.5 CBGS

Class-balanced Grouping and Sampling (CBGS) [11] propone un modelo compuesto principalmente de 4 partes: módulo de entrada, extractor de características 3D, RPN y una red con múltiples cabeceras en función del grupo. Con esto se consigue un funcionamiento en detección 3D, predicción de la velocidad y de la clase, todo ello sobre las 10 clases diferentes que utiliza nuScenes.

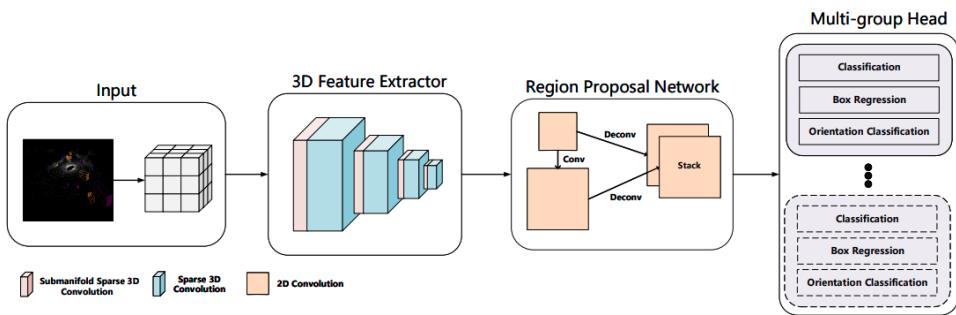


Figura 4.20: Arquitectura propuesta del detector CBGS.

Parte del funcionamiento del modelo proviene de la fase de entrenamiento, en la que se aplica una técnica de muestreo que trata de mejorar la media de uso de las diferentes clases para reducir la irregularidad del dataset de nuScenes, lo que aumenta el conjunto de entrenamiento de 28.130 a 128.100 muestras.

Para realizar la inferencia en el modelo se siguen los siguientes pasos:

1. Entrada de la red

Debido al método de validación de las detecciones en nuScenes es necesaria la inferencia de la velocidad por lo que siguiendo el método oficial del dataset, se utilizan diez barridos de LiDAR para la obtención del dato de la velocidad. En la entrada del modelo es necesario entonces alguna medida de tiempo, por lo que cada punto se encuentra codificado como $[x, y, z, intensity, \Delta t]$, siendo Δt la diferencia de tiempo entre el barrido inicial y el barrido del que proviene el punto. Tras esto se voxeliza la nube de puntos con un tamaño de vértice de $[0.1, 0.1, 0.2]$ m para la reducción del tiempo de computo. Dentro de cada vértice se mantiene únicamente la media de todos los puntos de dicho vértice.

2. Backbone utilizado

En la red principal es utilizado un modelo basado en sparse 3D convolution y un extractor de características. Tras esto se aplica un RPN, lo que termina en una red similar a la de VoxelNet [30] para extraer más características pero con capas convolucionales 2D.

3. Class-balance Grouping

Al tener una aparición de las diferentes clases muy irregular, con casi un 45 % de los objetos siendo coches, es muy difícil para objetos con formas diferentes extraer las diferencias entre ellos. Por ejemplo una bicicleta y una motocicleta tienen una forma muy similar en lo que respecta a la

nube de puntos, al igual que ocurren con un camión y un camión de construcción. Para solucionar este problema se propone la agrupación de clases en función de su similitud, teniendo en cuenta el tamaño de los objetos y un balanceo del tamaño de los grupos en función del número de clases. Lo cual acaba generando 6 grupos diferentes: (Coche), (Camión, Camión de construcción), (Bus, Tráiler), (Barrera), (Motocicleta, Bicicleta), (Peatón, Cono de tráfico). Por lo que tras la elección del grupo, se obtiene su cabecera y se hayan la posición, tamaño, rotación y velocidad.

Este método consigue por tanto una detección de objetos 3D además de un pseudo tracking para inferir la velocidad de los objetos, todo ello trabajando con 10 clases diferentes y obteniendo una velocidad de 9 Hz.

4.3 OpenPCDet

OpenPCDet [38] es un proyecto Open-Source para realizar detecciones de objetos 3D utilizando LiDAR. Esta herramienta ha sido utilizada para la evaluación, entrenamiento e implementación de diversos modelos debido a la diversidad de modelos y de datasets que son soportados.

OpenPCDet es un repositorio en Pytorch [39] que soporta múltiples modelos SOTA en detección de objetos 3D, con un código altamente refactorizado para las arquitecturas basadas en una o dos etapas. Este repositorio de código abierto es activamente actualizado con nuevos datasets soportados y modelos. Basado en OpenPCDet se ha conseguido ganar el Waymo Open Dataset en detección 3D, seguimiento 3D y adaptación del entorno utilizando únicamente las nubes de puntos del LiDAR.

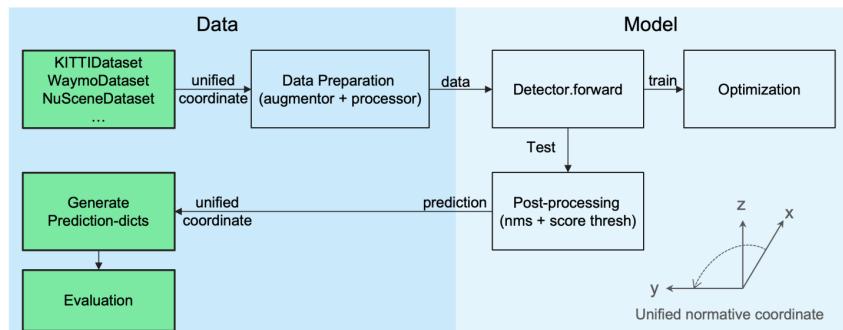


Figura 4.21: Diseño de OpenPCDet.

El patrón de diseño de OpenPCDet se encuentra basado en: la separación de datos a modelo unificando el sistema de coordenadas, definición única de las bounding boxes 3D [$x, y, z, dx, dy, dz, heading$], definición de la estructura de los modelos clara y flexible permitiendo fácilmente el soporte de múltiples modelos. Esta herramienta incluye diversos modelos en diferentes datasets, estos son:

- KITTI [23]
 - PointPillars [8]
 - SECOND [29]
 - PointRCNN [33]
 - Part-A² [40]
 - PV-RCNN [36]
 - Voxel R-CNN [41]

- CaDDN [42]
- nuScenes [25]
 - CBGS [11]
- Waymo [26]
 - SECOND [29]
 - Part-A² [40]
 - PV-RCNN [36]

Para el uso principal de la herramienta se utilizan los ficheros *train.py* y *test.py*, con estos a partir de diferentes flags es posible entrenar y evaluar los diferentes modelos que se deseen, sin necesidad de limitarse a los modelos que se tienen implementados, sino que dentro de *OpenPCDet/tools/cfgs* es posible modificar los ya existentes o también es posible crear nuevos modelos a partir de objetos en Python que utilicen toda la flexibilidad que OpenPCDet ofrece para reutilizar módulos de otros modelos.

Todo esto convierte a OpenPCDet en una gran herramienta para el desarrollo de modelos que sean basados en el **SOTA**, como herramienta de evaluación o para la implementación de modelos ya creados.

Capítulo 5

Desarrollo realizado

La persistencia es muy importante. No debes renunciar al menos que te veas obligado a renunciar.

Elon Musk

5.1 Estado del proyecto T4AC

5.1.1 ROS

5.1.2 Docker

5.1.3 Estructura del proyecto

5.2 Implementación en CARLA

5.2.1 CARLA

5.2.2 Funcionamiento del LiDAR en CARLA

5.2.3 Implementación del sistema clásico utilizando LiDAR

5.2.4 Implementación del sistema basado en Deep Learning utilizando LiDAR

5.3 Fusión sensorial

5.4 Vehículo del proyecto T4AC

5.5 Implementación sobre el vehículo T4AC

Capítulo 6

AD DevKit

La inspiración existe, pero tiene que encontrarte trabajando.

Picasso

6.1 Estado del arte en evaluación de vehículos autónomos

6.2 Obtención del ground truth

6.3 Evaluación de los modelos

Capítulo 7

Resultados obtenidos

Ninguna investigación humana puede ser llamada ciencia real si no puede demostrarse matemáticamente.

Leonardo da Vinci

7.1 Análisis cuantitativo en Kitti

7.2 Análisis cuantitativo en nuScenes

7.3 Análisis cualitativo del modelo clásico en CARLA

7.4 Análisis cualitativo de CBGS en CARLA

7.4.1 Comparativa con PointPillars en CARLA

7.5 Análisis cuantitativo de CBGS en CARLA

7.6 Análisis cualitativo de CBGS sobre el vehículo T4AC

7.6.1 Comparativa con PointPillars sobre el vehículo T4AC

Métrica	Resultado
mAP	0.4474
mATE	0.3379
mASE	0.2598
mAOE	0.3156
mAVE	0.2886
mAAE	0.2025
NDS	0.5832

Tabla 7.1: Rendimiento medio de CBGS PointPillars Multihead en nuScenes.

Tipo de objeto	AP	ATE	ASE	AOE	AVE	AAE
car	0.812	0.189	0.154	0.123	0.664	0.269
truck	0.500	0.354	0.189	0.093	0.415	0.277
bus	0.634	0.367	0.183	0.048	0.869	0.380
trailer	0.352	0.606	0.208	0.396	0.281	0.183
construction_vehicle	0.121	0.761	0.453	0.785	0.123	0.332
pedestrian	0.723	0.167	0.277	0.394	0.440	0.270
motorcycle	0.300	0.229	0.243	0.454	0.988	0.324
bicycle	0.064	0.189	0.273	0.506	0.494	0.093
traffic_cone	0.472	0.182	0.328	nan	nan	nan
barrier	0.499	0.341	0.288	0.071	nan	nan

Tabla 7.2: Análisis por clase de CBGS PointPillars Multihead en nuScenes.

Tipo de objeto	AP	IoU	AVE
Unknown	0.0	0.0	0.0
Unknown_Small	0.0	0.0	0.0
Unknown_Medium	0.0	0.0	0.0
Unknown_Big	0.0	0.0	0.0
Pedestrian	0.0	0.0	0.0
Bike	0.0	0.0	0.0
Car	0.731	0.491	0.715
Truck	0.0	0.0	0.0
Motorcycle	0.0	0.0	0.0
Other_Vehicle	0.0	0.0	0.0
Barrier	0.0	0.0	0.0
Sign	0.0	0.0	0.0

Tabla 7.3: Análisis por clase de CBGS PointPillars Multihead en CARLA.

Capítulo 8

Conclusiones

La verdadera felicidad radica en la finalización del trabajo utilizando tu propio cerebro y habilidades.

Soichiro Honda

8.1 Modelos estudiados

8.2 Comparativas adicionales

8.2.1 Ajuste de modelos basados en Kitti a nuScenes

8.2.2 Número de PCL de entrada en modelos evaluados sobre nuScenes

8.2.3 Tamaño del voxel en modelos basados en redes neuronales

8.3 Futuros trabajos

Bibliografía

- [1] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi, “Computing systems for autonomous driving: State of the art and challenges,” *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6469–6486, 2021.
- [2] “Automated vehicles for safety,” Tech. Rep., 2018. [Online]. Available: <https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety>
- [3] “How self-driving cars work: Sensor systems,” Tech. Rep., 2021. [Online]. Available: <https://www.udacity.com/blog/2021/03/how-self-driving-cars-work-sensor-systems.html>
- [4] Y. Zhang, J. Ran, X. Chen, K. Fang, and H. Chen, “Observation of the inverse, zero and normal doppler effect in configurable transmission lines,” in *2015 IEEE 4th Asia-Pacific Conference on Antennas and Propagation (APCAP)*, 2015, pp. 229–230.
- [5] “Hdl-64e, high definition real-time 3d lidar,” Velodyne Lidar. [Online]. Available: <https://velodynelidar.com/products/hdl-64e/>
- [6] A. M. Wallace, A. Halimi, and G. S. Buller, “Full waveform lidar for adverse weather conditions,” *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7064–7077, 2020.
- [7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.
- [8] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, “Pointpillars: Fast encoders for object detection from point clouds,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 12689–12697.
- [9] F. Lotfi, V. Ajallooeian, and H. D. Taghirad, “Robust object tracking based on recurrent neural networks,” in *2018 6th RSI International Conference on Robotics and Mechatronics (IcRoM)*, 2018, pp. 507–511.
- [10] S. Wang, Y. Sun, C. Liu, and M. Liu, “Pointtracknet: An end-to-end network for 3-d object detection and tracking from point clouds,” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3206–3212, 2020.
- [11] B. Zhu, Z. Jiang, X. Zhou, Z. Li, and G. Yu, “Class-balanced grouping and sampling for point cloud 3d object detection,” *CoRR*, vol. abs/1908.09492, 2019. [Online]. Available: <http://arxiv.org/abs/1908.09492>

- [12] T.-L. Kim, J.-S. Lee, and T.-H. Park, "Fusing lidar, radar, and camera using extended kalman filter for estimating the forward position of vehicles," in *2019 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, 2019, pp. 374–379.
- [13] J. del Egido Sierra, "Detección del entorno 360° de un vehículo autónomo mediante lidar aplicando técnicas deep learning," Trabajo de Fin de Master, UAH Politécnica, 2020.
- [14] M. Antunes, "Sistema de visión estereo aplicado a la detección y seguimiento de objetos en conducción autónoma," Trabajo de Fin de Grado, UAH Politécnica, 2021.
- [15] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981. [Online]. Available: <https://doi.org/10.1145/358669.358692>
- [16] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: <https://doi.org/10.1145/361002.361007>
- [17] W. Hou, D. Li, C. Xu, H. Zhang, and T. Li, "An advanced k nearest neighbor classification algorithm based on kd-tree," in *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, 2018, pp. 902–905.
- [18] Y. Cao, X. Zhang, B. Duan, W. Zhao, and H. Wang, "An improved method to build the kd tree based on presorted results," in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, 2020, pp. 71–75.
- [19] L. Hu, S. Nooshabadi, and M. Ahmadi, "Massively parallel kd-tree construction and nearest neighbor search algorithms," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015, pp. 2752–2755.
- [20] J. Geyer, Y. Kassahun, M. Mahmudi, X. Ricou, R. Durgesh, A. S. Chung, L. Hauswald, V. H. Pham, M. Mühllegg, S. Dorn, T. Fernandez, M. Jänicke, S. Mirashi, C. Savani, M. Sturm, O. Vorobiov, M. Oelker, S. Garreis, and P. Schuberth, "A2d2: Audi autonomous driving dataset," 2020.
- [21] M.-F. Chang, J. Lambert, P. Sangkloy, J. Singh, S. Bak, A. Hartnett, D. Wang, P. Carr, S. Lucey, D. Ramanan, and J. Hays, "Argoverse: 3d tracking and forecasting with rich maps," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.
- [22] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [23] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354–3361.
- [24] J. Houston, G. Zuidhof, L. Bergamini, Y. Ye, L. Chen, A. Jain, S. Omari, V. Iglovikov, and P. Ondruska, "One thousand and one hours: Self-driving motion prediction dataset," 2020.
- [25] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

- [26] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, V. Vasudevan, W. Han, J. Ngiam, H. Zhao, A. Timofeev, S. Ettinger, M. Krivokon, A. Gao, A. Joshi, Y. Zhang, J. Shlens, Z. Chen, and D. Anguelov, “Scalability in perception for autonomous driving: Waymo open dataset,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.
- [27] J. Xie, M. Kiefel, M.-T. Sun, and A. Geiger, “Semantic instance annotation of street scenes by 3d to 2d label transfer,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [28] “Waymo open dataset,” Waymo. [Online]. Available: <https://waymo.com/open>
- [29] Y. Yan, Y. Mao, and B. Li, “Second: Sparsely embedded convolutional detection,” *Sensors*, vol. 18, no. 10, 2018. [Online]. Available: <https://www.mdpi.com/1424-8220/18/10/3337>
- [30] Y. Zhou and O. Tuzel, “Voxelnet: End-to-end learning for point cloud based 3d object detection,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4490–4499.
- [31] “Sparseconvnet,” Facebook. [Online]. Available: <https://github.com/facebookresearch/SparseConvNet>
- [32] R. Girshick, “Fast r-cnn,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.
- [33] S. Shi, X. Wang, and H. Li, “Pointrcnn: 3d object proposal generation and detection from point cloud,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 770–779.
- [34] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, “Pointnet++: Deep hierarchical feature learning on point sets in a metric space,” 2017.
- [35] Z. Liang, M. Zhang, Z. Zhang, X. Zhao, and S. Pu, “Rangercnn: Towards fast and accurate 3d object detection with range image representation,” 2021.
- [36] S. Shi, C. Guo, L. Jiang, Z. Wang, J. Shi, X. Wang, and H. Li, “Pv-rcnn: Point-voxel feature set abstraction for 3d object detection,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 10 526–10 535.
- [37] R. Q. Charles, H. Su, M. Kaichun, and L. J. Guibas, “Pointnet: Deep learning on point sets for 3d classification and segmentation,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 77–85.
- [38] O. D. Team, “Openpcdet: An open-source toolbox for 3d object detection from point clouds,” <https://github.com/open-mmlab/OpenPCDet>, 2020.
- [39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [40] S. Shi, Z. Wang, J. Shi, X. Wang, and H. Li, “From points to parts: 3d object detection from point cloud with part-aware and part-aggregation network,” 2020.

- [41] J. Deng, S. Shi, P. Li, W. Zhou, Y. Zhang, and H. Li, “Voxel r-cnn: Towards high performance voxel-based 3d object detection,” 2021.
- [42] C. Reading, A. Harakeh, J. Chae, and S. L. Waslander, “Categorical depth distribution network for monocular 3d object detection,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 8555–8564.
- [43] “Información sobre gnu/linux en wikipedia,” <http://es.wikipedia.org/wiki/GNU/Linux> [Último acceso 1/noviembre/2013].
- [44] “Página de la aplicación emacs,” <http://savannah.gnu.org/projects/emacs/> [Último acceso 1/noviembre/2013].
- [45] “Página de la aplicación kdevelop,” <http://www.kdevelop.org> [Último acceso 1/noviembre/2013].
- [46] L. Lamport, *LaTeX: A Document Preparation System, 2nd edition.* Addison Wesley Professional, 1994.
- [47] “Página de la aplicación octave,” <http://www.octave.org> [Último acceso 1/noviembre/2013].
- [48] “Página de la aplicación cvs,” <http://savannah.nongnu.org/projects/cvs/> [Último acceso 1/noviembre/2013].
- [49] “Página de la aplicación gcc,” <http://savannah.gnu.org/projects/gcc/> [Último acceso 1/noviembre/2013].
- [50] “Página de la aplicación make,” <http://savannah.gnu.org/projects/make/> [Último acceso 1/noviembre/2013].

Apéndice A

Herramientas y recursos

Las herramientas necesarias para la elaboración del proyecto han sido:

- PC compatible
- Sistema operativo GNU/Linux [43]
- Entorno de desarrollo Emacs [44]
- Entorno de desarrollo KDevelop [45]
- Procesador de textos L^AT_EX[46]
- Lenguaje de procesamiento matemático Octave [47]
- Control de versiones CVS [48]
- Compilador C/C++ gcc [49]
- Gestor de compilaciones make [50]

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR

