ESCUELA DE INGENIERÍA DE FUENLABRADA

GRADO EN INGENIERÍA DE SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

Segmentación de Nubes De Puntos LiDAR para Conducción Autónoma en Entornos No Estructurados usando Aprendizaje Profundo

Autor: Félix Martínez Alonso
Tutora: Inmaculada Mora Jiménez
Cotutor: José María Cañas Plaza

Curso académico 2024/2025

Este trabajo ha sido parte del proyecto GAIA PLEC2023-010303 financiado por MICIU/AEI/10.13039/501100011033.

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a mis tutores, Inmaculada Mora Jiménez y José María Cañas Plaza, por su orientación, disponibilidad y apoyo constante durante el desarrollo de este trabajo. Su experiencia y dedicación han sido clave para afrontar cada etapa del proyecto y alcanzar los objetivos planteados. También deseo agradecer a David Pascual Hernández, cuyo conocimiento y ayuda en aspectos específicos del trabajo han contribuido al resultado final.

A mis compañeros, amigos y especialmente a mi familia, por su apoyo, comprensión y ánimo incondicional. En particular, a mis padres, por su confianza absoluta, por todos los esfuerzos que han realizado a lo largo de estos años y por acompañarme en todo momento.

Resumen

El presente Trabajo de Fin de Grado se desarrolla en el marco del proyecto GAIA (PLEC2023-010303), cuyo objetivo principal es incrementar el conocimiento de la situación de los entornos forestales mediante el uso de plataformas autónomas, análisis inteligente de datos y técnicas de percepción avanzada orientadas a la prevención, gestión y restauración de áreas afectadas por incendios. En este contexto, el trabajo aborda la problemática de la segmentación semántica de nubes de puntos LiDAR capturadas en exteriores no estructurados, un reto técnico que condiciona la capacidad de los sistemas de percepción para interpretar con precisión el entorno tridimensional en aplicaciones de robótica móvil y conducción autónoma. La heterogeneidad y la variabilidad de estos entornos complican la segmentación fiable de elementos naturales y artificiales, incrementando la necesidad de metodologías robustas y escalables.

Se ha llevado a cabo el estudio, la implementación propia y la evaluación de arquitecturas neuronales de referencia existentes, como PointNet y PointNet++, reproducidas en PyTorch respetando sus planteamientos conceptuales originales. PointNet se caracteriza por procesar la nube de puntos completa mediante una agregación simétrica de características, sin aprovechar explícitamente las relaciones locales entre puntos vecinos. Por su parte, PointNet++ amplía este enfoque mediante un esquema jerárquico que subdivide la nube en regiones locales a diferentes escalas, permitiendo capturar la estructura geométrica con más detalle. Estas implementaciones se adaptaron a los conjuntos de datos empleados, con el objetivo de validar su rendimiento y establecer una base comparativa. Las redes se entrenaron principalmente sobre la base de datos GOOSE, compuesta por escenas reales de exteriores rurales etiquetadas, y se realizó una evaluación cruzada con el conjunto RELLIS-3D para valorar la capacidad de generalización frente a escenarios y configuraciones distintas. Adicionalmente, se desarrolló una variante modificada de PointNet++ que integra la información de intensidad de remisión del LiDAR como canal extra de entrada en el bloque de clasificación final, con el propósito de analizar el impacto de esta fusión en la precisión de la segmentación.

De forma complementaria, se desarrolló una aplicación interactiva de visualización tridimensional que facilita la exploración de las nubes de puntos de las bases de datos y las capturadas en simulaciones con CARLA, permitiendo un análisis cualitativo de la distribución y geometría de los datos.

Como resultado de la evaluación de los modelos entrenados, PointNet++ mostró un rendimiento claramente superior a PointNet en términos de mIoU (58,71 % frente a 32,50 %) y mRecall (70,90 % frente a 48,32 %), confirmando la ventaja de los enfoques jerárquicos para capturar la estructura local de la escena. La incorporación de la remisión en la versión modificada PointNet++* permitió incrementar todavía más las figuras de mérito hasta un mIoU del 60,38 % y un mRecall del 72,82 %, demostrando que la fusión de atributos adicionales puede reducir la confusión entre clases con geometría similar. Estas conclusiones evidencian que las arquitecturas jerárquicas constituyen una solución viable y eficaz en entornos no estructurados, si bien persisten limitaciones en la segmentación de elementos pequeños o menos frecuentes. Este trabajo sienta una base metodológica sólida para continuar investigando arquitecturas más avanzadas, técnicas de aprendizaje multimodal e integraciones con otros sensores que optimicen la percepción tridimensional en aplicaciones de interés social y medioambiental como las planteadas en el proyecto GAIA.

Índice general

Índice de figuras

Índice de cuadros

Lista de acrónimos

2D Dos Dimensiones

3D Tres Dimensiones

AMCW Amplitude Modulated Continuous Wave

CARLA CAR Learning to Act

CNN Convolutional Neural Network

FMCW Frequency Modulated Continuous Wave

FOV Field of View

GAIA Gestión integral para la prevención, extinción y reforestación frente a incendios forestales

GOOSE German Outdoor and Offroad Dataset

GPS Global Positioning System

GUI Graphical User Interface

IMU Inertial Measurement Unit

IoU Intersection over Union

KNN K-Nearest Neighbors

KPConv Kernel Point Convolution

LiDAR Light Detection and Ranging

MEMS Micro Electro Mechanical Systems

MLP Multi-layer Perceptron

RandLA-Net Random Sampling and Local Aggregation Network

RGB Red Green Blue

RNAs Redes Neuronales Artificiales

ToF Time of Flight

Capítulo 1

Introducción y Objetivos

Este primer capítulo tiene como objetivo contextualizar el trabajo realizado, exponiendo las bases tecnológicas sobre las que se apoya y la motivación que justifica su desarrollo. Se introduce el problema general de la segmentación semántica de nubes de puntos captadas por sensores *Light Detection and Ranging* (LiDAR), con especial atención a los retos que plantean los entornos exteriores no estructurados. Además, se enmarca el estudio dentro del proyecto de gestión integral para la prevención, extinción y reforestación frente a incendios forestales (GAIA) [1], en cuyo contexto la percepción tridimensional puede aportar valor. Finalmente, se detallan los objetivos específicos del trabajo y la organización general de la memoria.

1.1 Contexto y motivación

En los últimos años, el sensor LiDAR se ha consolidado como una de las fuentes de información principales para la percepción en robótica móvil, mapeo topográfico, conducción autónoma y monitorización de entornos. Estos sensores generan nubes de puntos en tres dimensiones (3D) emitiendo pulsos de luz láser y midiendo el tiempo de retorno tras reflejarse en las superficies, lo que permite capturar con gran precisión la geometría del entorno. Como se ilustra en la Figura 1.1, estas nubes representan con gran fidelidad el entorno escaneado por el sensor. Sin embargo, puede que la información geométrica por sí sola no baste y sea necesario extraer conocimiento semántico a partir de los datos crudos para que los sistemas autónomos comprendan mejor su entorno y puedan tomar decisiones de navegación más informadas. En este contexto, la segmentación semántica de nubes de puntos permite asignar una clase a cada punto, transformando datos sin procesar en una representación estructurada y comprensible [2].

La robótica de servicios ha avanzado notablemente en la última década, abarcando aplicaciones en ámbitos como la asistencia personal, la agricultura, la logística o la inspección industrial [4]. Uno de los casos más representativos es la conducción autónoma, donde los vehículos deben percibir y entender su entorno en tiempo real [5]. Esta conducción puede desarrollarse en entornos urbanos, interurbanos o no estructurados, cada uno con desafíos específicos. En todos ellos, los sensores más utilizados son los sistemas de visión y los sensores LiDAR, que aportan información visual y geométrica complementaria. La combinación de estos datos permite tareas complejas como la segmentación semántica, el seguimiento de objetos y la planificación de trayectorias en escenarios dinámicos y a

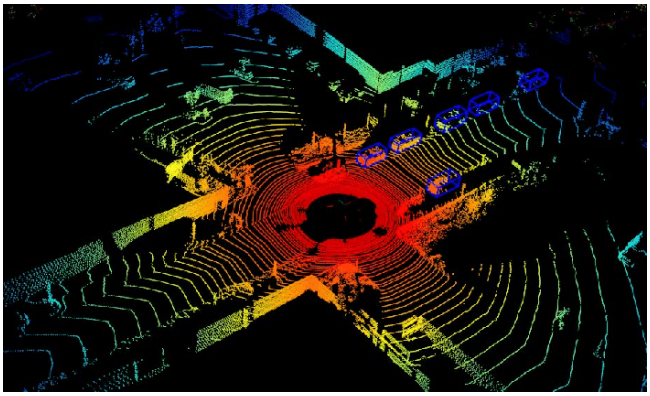


Figura 1.1: Nube de puntos obtenida del sensor Velodyne LIDAR utilizada en una aplicación de detección de vehículos. Obtenida de [3].

menudo impredecibles. En este contexto, el aprendizaje profundo ha supuesto un avance crucial, al permitir modelar con alta precisión patrones complejos a partir de grandes volúmenes de datos sensoriales. Empresas como Tesla, Waymo o Cruise lideran el desarrollo de estas tecnologías, impulsando avances en percepción, segmentación y navegación autónoma.

En el marco del proyecto GAIA[1], este trabajo se orienta al estudio y aplicación de varias arquitecturas neuronales existentes para la segmentación semántica de nubes de puntos LiDAR en entornos exteriores no estructurados. Si bien es posible utilizar y generar *datasets* simulados, por ejemplo mediante entornos de simulación avanzados como CARLA [6], para entrenar y evaluar modelos de segmentación, estos presentan limitaciones importantes. Las nubes de puntos simuladas no son completamente análogas a los datos adquiridos por una configuración LiDAR real, ya que intervienen numerosos factores que introducen ruido, oclusiones y geometrías no ideales. Aunque existen técnicas de procesamiento para aproximar las escenas sintéticas a características más cercanas a las capturas reales, en este trabajo se ha optado por emplear exclusivamente datos reales. Se utilizan bases de datos internacionales a gran escala que representan entornos naturales reales y diversos, como *German Outdoor and Offroad Dataset* (GOOSE) [7] y RELLIS-3D [8], que incluyen anotaciones punto a punto y abarcan múltiples clases de interés. El análisis se centra tanto en figuras de mérito cuantitativas como en observaciones cualitativas de la segmentación obtenida, con el fin de aportar conocimiento útil al desarrollo de sistemas de percepción más fiables, adaptables y eficientes para vehículos autónomos en contextos no estructurados.

A diferencia de los entornos urbanos, caracterizados por geometrías regulares y estructuras previsibles, los entornos exteriores no estructurados, como zonas rurales o naturales, presentan una gran variabilidad espacial y una organización geométrica poco definida [9]. La presencia de vegetación, terreno irregular u objetos deformables añade complejidad al análisis, dificultando la interpretación

[1] https://roboticslaburjc.github.io/projects/gaia

Figura 1.2: Ejemplo visual de entornos exteriores no estructurados. Obtenida de [10].

directa de los datos. Esto exige el desarrollo de modelos robustos capaces de generalizar ante escenas heterogéneas, ruido y geometrías ambiguas. En la Figura 1.2 se presentan algunas imágenes ilustrativas de entornos no estructurados.

1.2 Segmentación Semántica

Como se ha introducido, la segmentación semántica aplicada a nubes de puntos LiDAR persigue asignar una etiqueta a cada punto de la nube, diferenciando entre los distintos tipos de superficie presentes en una escena. En la Figura 1.3 se muestra una nube de puntos tomada en un entorno estructurado y segmentada semánticamente en diferentes categorías, diferenciadas por color en la figura, tales como vehículos, construcciones o calzada. Antes del auge del aprendizaje profundo, esta tarea se abordaba mediante métodos tradicionales como la agrupación basada en geometría, clasificadores supervisados o modelos probabilísticos. Aunque funcionales en escenarios simples o controlados, estas técnicas presentan grandes limitaciones en entornos complejos y no estructurados. La aparición de métodos basados en redes neuronales artificiales (RNAs) ha supuesto un avance significativo al permitir construir límites de decisión complejos sin depender de una ingeniería manual de características [11]. Una de sus principales ventajas es su capacidad para aprender automáticamente qué información es relevante a partir de los propios datos, algo especialmente importante en las nubes de puntos, que carecen de una estructura regular como la de las imágenes, y donde la densidad y distribución de los puntos puede variar significativamente.

Antes de describir las principales arquitecturas, es importante distinguir entre varias tareas fundamentales en la interpretación de nubes de puntos. La segmentación semántica asigna una etiqueta de clase a cada punto, sin distinguir entre instancias distintas del mismo tipo de objeto. En cambio, la segmentación de instancias proporciona no solo la clase, sino también la identidad individual de

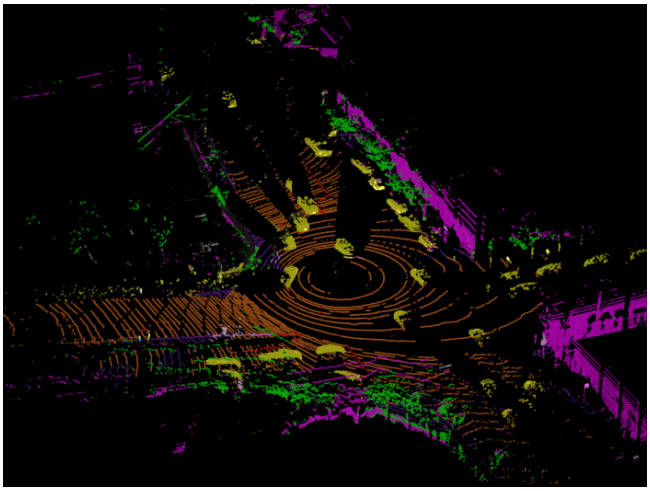


Figura 1.3: Segmentación semántica de una nube de puntos en entorno urbano, con diferenciación de clases mediante codificación por colores únicos. Obtenida de [12].

cada objeto presente, permitiendo diferenciar entre varias instancias de la misma categoría. La detección de objetos busca identificar y delimitar objetos mediante volúmenes delimitadores, como cajas 3D, sin etiquetar cada punto de forma densa. Finalmente, la clasificación de nubes completas asigna una única etiqueta a toda la nube, como podría ser el tipo de estructura escaneada o la escena representada. Aunque relacionadas, cada una de estas tareas tiene objetivos distintos y exige enfoques de procesamiento específicos.

Diversas arquitecturas han sido propuestas específicamente para la segmentación semántica de nubes de puntos, tarea en la que se centra este trabajo. Entre ellas, PointNet [13] supuso un avance fundamental al ser la primera en procesar directamente las coordenadas tridimensionales mediante funciones simétricas que preservan la invariancia al orden de los datos. A partir de esta base, PointNet++ [14] introdujo una estructura jerárquica capaz de capturar relaciones locales mediante agrupaciones espaciales progresivas, lo que mejoró notablemente el rendimiento en escenas con geometría irregular y gran diversidad estructural. Estas dos arquitecturas no solo marcaron un punto de inflexión en la transición desde métodos clásicos hacia redes neuronales especializadas, sino que también han servido como base para muchas propuestas posteriores más complejas. En este trabajo, ambas arquitecturas serán utilizadas como referencia principal para el estudio y comprensión de los fundamentos de la segmentación semántica basada en aprendizaje profundo, así como para su posterior integración y aplicación práctica en entornos exteriores no estructurados.

Entre las contribuciones más recientes, destacan arquitecturas como *Kernel Point Convolution* (KPConv) [15] y *Random Sampling and Local Aggregation Network* (RandLA-Net) [16], que definen operaciones convolucionales sobre vecindarios locales o emplean mecanismos de atención eficientes para mejorar la segmentación sin comprometer el rendimiento computacional. La Figura 1.4 muestra

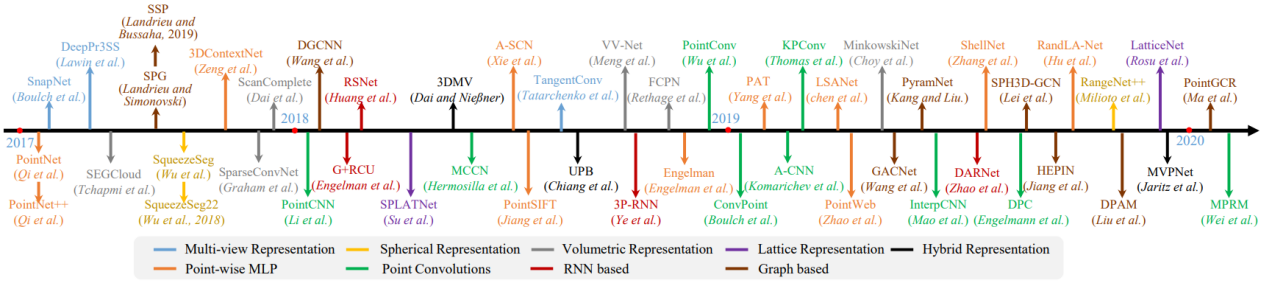


Figura 1.4: Línea temporal de los métodos más relevantes en segmentación semántica 3D basados en aprendizaje profundo. Obtenida de [11].

una línea temporal con algunas de las principales arquitecturas propuestas en los últimos años, donde se puede observar cómo PointNet y PointNet++ ocupan un lugar destacado como punto de partida de la evolución moderna en este campo. Gracias a estas aportaciones, el aprendizaje profundo se ha consolidado como la opción más eficaz para abordar la segmentación semántica de nubes de puntos en entornos reales.

1.3 Objetivos

Este trabajo tiene como finalidad estudiar, implementar desde cero y analizar el comportamiento de las arquitecturas PointNet y PointNet++ aplicadas a la segmentación semántica de nubes de puntos LiDAR en entornos exteriores no estructurados. A partir del uso de bases de datos reales como GOOSE y RELLIS-3D, se busca comprender en profundidad estas arquitecturas fundamentales del estado del arte y evaluar su rendimiento en condiciones complejas. Además, se pretende investigar cómo ciertas modificaciones estructurales pueden influir en la calidad de la segmentación, en línea con los retos planteados por el proyecto GAIA. Los objetivos especificos son:

O1 Desarrollar una herramienta de visualización que permita explorar secuencias LiDAR tanto desde archivos pertenecientes a las bases de datos RELLIS-3D y GOOSE, como desde entornos simulados en CARLA [6], en tiempo real.

O2 Analizar las capacidades y limitaciones de las arquitecturas PointNet y PointNet++ en tareas de segmentación semántica de nubes de puntos, especialmente en escenarios exteriores no estructurados.

O3 Explorar posibles modificaciones en la arquitectura de base que permitan mejorar el rendimiento del modelo en términos de precisión y generalización sobre datos reales.

O4 Determinar la viabilidad práctica de estas arquitecturas en aplicaciones relacionadas con la percepción automática en entornos forestales, dentro del marco del proyecto GAIA.

1.4 Metodología

La organización del trabajo ha seguido la metodología ágil Scrum, adaptada a las condiciones del desarrollo y a la disponibilidad temporal. A continuación, se resumen los principales aspectos metodológicos:

- **Planificación iterativa por sprints:** el trabajo se dividió en bloques temporales semanales, con tareas específicas asignadas a cada sprint. Esto permitió un desarrollo progresivo y flexible.
- **Seguimiento con tutores:** se mantuvieron reuniones semanales con los dos tutores académicos para revisar el avance, resolver dudas técnicas y redefinir tareas si era necesario.
- **Control de versiones:** todo el código y la experimentación fue gestionado mediante un repositorio GitHub[2], facilitando la trazabilidad.
- **Documentación continua:** se llevó un blog técnico interno para anotar avances, decisiones de diseño, problemas encontrados y soluciones adoptadas a lo largo del desarrollo[3].
- **Presentación del trabajo en el congreso WAF2025:** se ha preparado un artículo con los resultados obtenidos como contribución al *International Workshop of Physical Agents* (WAF) 2025[4].

La metodología técnica desarrollada en este trabajo se ha estructurado en las fases descritas a continuación. En la Figura 1.5 se muestra el diagrama de Gantt con los principales hitos del trabajo. A continuación, se enumeran más en profundidad las fases que ha tenido el trabajo:

- **Análisis del estado del arte:** estudio detallado de los sensores LiDAR, las características y formatos de las nubes de puntos y de los enfoques actuales para su segmentación semántica, con énfasis en métodos basados en aprendizaje profundo y sus principales líneas de evolución.
- **Desarrollo de una herramienta de visualización:** implementación de un programa interactivo que permite explorar secuencias LiDAR y visualizar datos en tiempo real generados por el simulador CARLA.

[2] https://github.com/RoboticsLabURJC/2024-tfg-felix-martinez
[3] https://roboticslaburjc.github.io/2024-tfg-felix-martinez/
[4] https://mc3.upct.es/waf25/

- **Análisis de los conjuntos de datos:** exploración de los escenarios con la herramienta desarrollada, las clases y las características geométricas y semánticas presentes en los datos seleccionados para el entrenamiento y la validación de los modelos.
- **Estudio de arquitecturas:** revisión en profundidad del funcionamiento interno de PointNet y PointNet++, así como de sus ventajas y limitaciones en tareas de segmentación semántica.
- **Implementación de modelos:** desarrollo e integración de las arquitecturas PointNet y PointNet++ adaptadas a los requisitos específicos del proyecto y los formatos de los datos, estableciendo una preparación adecuada de los datos.
- **Realización de experimentos con GOOSE:** entrenamiento, validación y evaluación de los modelos sobre el *dataset* GOOSE.
- **Evaluación cuantitativa y cualitativa:** análisis de los resultados mediante figuras de mérito (por ejemplo, precisión global y por clase) y comparación visual de las predicciones con las nubes de puntos originales.
- **Modificación de PointNet++:** adaptación de la arquitectura para incorporar la característica de intensidad de remisión como entrada adicional en las etapas finales de procesamiento.
- **Entrenamiento y evaluación del modelo modificado:** repetición del proceso experimental con la red extendida, comparando su rendimiento respecto a las versiones originales.
- **Pruebas de generalización con RELLIS-3D:** aplicación de los modelos entrenados sobre nubes de puntos del dataset RELLIS-3D para evaluar su comportamiento frente a otras configuraciones LiDAR y entornos no estructurados.

1.5 Organización de la Memoria

El presente documento se organiza en los siguientes capítulos.

- **Capítulo 1. Introducción y Objetivos**: Este capítulo introductorio se compone de cuatro secciones. En la primera, se expone el contexto general del trabajo, destacando la motivación y la problemática a abordar, especialmente en relación con los entornos no estructurados y el uso de sensores LiDAR. En la segunda sección se introduce el concepto de segmentación semántica aplicado a nubes de puntos, incluyendo una visión general de las principales arquitecturas neuronales utilizadas en el estado del arte. A continuación, en la tercera sección, se presentan los

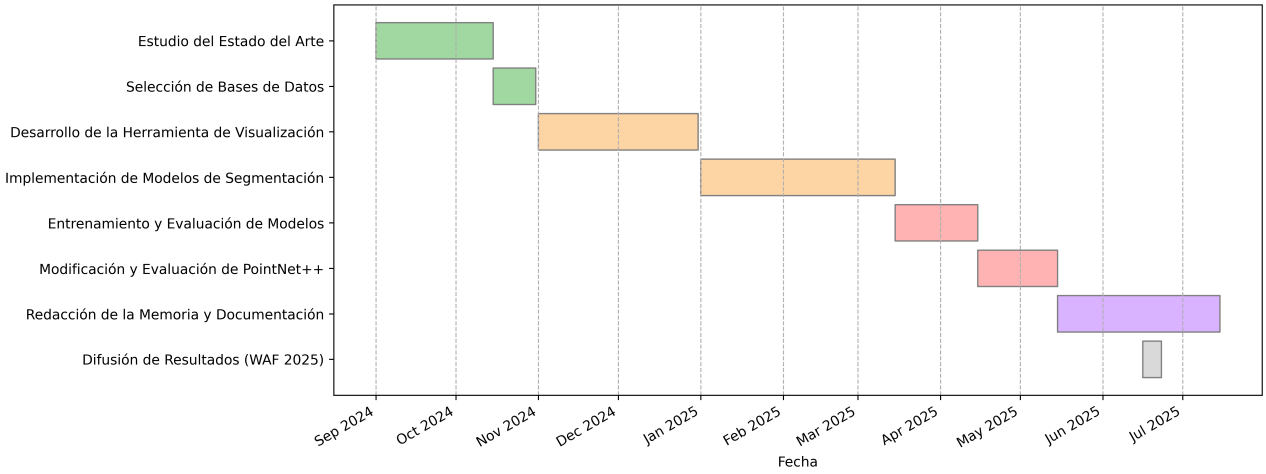


Figura 1.5: Diagrama de Gantt del trabajo con sus principales hitos.

objetivos que guían el desarrollo del proyecto. Por último, en esta cuarta sección, se describe brevemente la organización del resto de la memoria.

- **Capítulo 2. Estado del Arte**: Este capítulo presenta una revisión de los conceptos y tecnologías fundamentales que sustentan el desarrollo del trabajo. En primer lugar, se describen los sensores LiDAR como principal fuente de datos tridimensionales, incluyendo su principio de funcionamiento, tipología, características técnicas y limitaciones más relevantes. A continuación, se introducen las nubes de puntos como representación básica de la información espacial, detallando sus propiedades y los principales formatos de almacenamiento. Posteriormente, se revisan las estrategias de representación y estructuración de nubes de puntos, con especial atención a los métodos de voxelización, reconstrucción de mallas y estructuras de grafos. Finalmente, se expone el estado del arte en segmentación semántica 3D mediante aprendizaje profundo, contextualizando la evolución de las arquitecturas y justificando la selección de PointNet y PointNet++ como objeto de estudio principal.

- **Capítulo 3. Bases de Datos en Percepción Tridimensional**: Este capítulo describe los principales conjuntos de datos empleados en la experimentación de las arquitecturas de segmentación semántica de nubes de puntos. En primer lugar, se expone el contexto general de las bases de datos orientadas a percepción 3D, destacando su importancia como recurso para entrenar y validar arquitecturas de aprendizaje profundo. A continuación, se presentan en detalle los conjuntos GOOSE y RELLIS-3D, seleccionados por su relevancia en entornos naturales y no estructurados. Para cada uno, se explican sus objetivos, la tipología de escenarios capturados, la configuración de sensores utilizada, la estructura de organización de los datos y el formato de las anotaciones semánticas a nivel de punto.

- **Capítulo 4. Desarrollo de la Herramienta de Visualización LiDAR**: Este capítulo presenta la herramienta software Lidar-Visualizer, desarrollada como parte de este trabajo. Se describe su arquitectura, estructurada en torno a un lanzador con interfaz gráfica que invoca dos módulos de visualización independientes: uno para la conexión en tiempo real con el simulador CARLA y otro para la reproducción interactiva de secuencias desde archivos locales. Se detallan las funcionalidades más importantes de cada módulo, explicando la gestión de datos, la interacción con el usuario y el renderizado dual con Open3D y Pygame, todo ello ilustrado con los fragmentos de código más representativos de la aplicación.

- **Capítulo 5. Desarrollo Experimental y Resultados**: Este capítulo describe de manera detallada la metodología experimental seguida para entrenar y evaluar diferentes arquitecturas de segmentación de nubes de puntos. Se expone el proceso de preparación de los datos, que incluye la definición de particiones de entrenamiento, validación y prueba, así como el preprocesado y el mapeo de etiquetas a categorías semánticas agregadas. Asimismo, se presentan los componentes principales del entrenamiento, como las funciones de pérdida ponderadas, el optimizador adaptativo, la programación dinámica de la tasa de aprendizaje y las figuras de mérito empleadas, junto con el procedimiento de búsqueda de hiperparámetros que permitió identificar configuraciones óptimas. Finalmente, se analizan los resultados obtenidos con las versiones implementadas de PointNet, PointNet++ y la variante propuesta PointNet++*, comparando figuras de mérito cuantitativas y visualizaciones cualitativas, y se discute su capacidad de generalización a través de una validación cruzada sobre el conjunto de datos RELLIS-3D.

- **Capítulo 6. Conclusiones y Trabajo Futuro**: Este capítulo recoge los principales hallazgos alcanzados a lo largo del proyecto, sintetizando los resultados obtenidos con las distintas arquitecturas de segmentación y valorando el grado de cumplimiento de los objetivos propuestos. Asimismo, se detallan las competencias técnicas y habilidades desarrolladas durante el trabajo, tanto en el ámbito del aprendizaje profundo aplicado a datos 3D como en el diseño de herramientas de visualización. Finalmente, se proponen diversas líneas de trabajo futuro que contemplan la exploración de arquitecturas más avanzadas, la integración de nuevas fuentes de información y el uso de entornos sintéticos para reforzar la capacidad de generalización de los modelos.

Capítulo 2

Estado del Arte

Este capítulo presenta una revisión de los conceptos y tecnologías fundamentales sobre los que se apoya este trabajo. Se describen los sensores LiDAR como fuente principal de información tridimensional, así como la información que proporcionan estos sensores, las nubes de puntos. Además, se abordan las técnicas de procesamiento de nubes de puntos más reconocidas y las arquitecturas fundamentales que manejan esta información en bruto contextualizando su importancia en el avance del estado del arte actual, *PointNet* y *PointNet*++, que constituyen el núcleo del trabajo desarrollado.

2.1 Sensores LiDAR

En el contexto del desarrollo de sistemas autónomos y de percepción inteligente, los sensores LiDAR han emergido como una tecnología clave para la adquisición de información espacial precisa en 3D [17]. Su capacidad para proporcionar información geométrica detallada, independientemente de las condiciones de iluminación, ha favorecido su adopción en múltiples áreas como la robótica, la conducción autónoma, el mapeo ambiental y en la segmentación de escenas complejas mediante aprendizaje automático. A diferencia de otros sensores como cámaras RGB o estereoscópicas, los sensores LiDAR ofrecen una representación directa y densa del entorno a través de la consecución de nubes de puntos. En los últimos años, el avance tanto en *hardware* como en algoritmos de procesamiento ha impulsado un cuerpo creciente de investigaciones orientadas a aprovechar las ventajas del LiDAR para tareas de percepción avanzada [17]. El objetivo de esta sección es presentar los diferentes tipos de sensores disponibles y analizar sus principales características, con el fin de comprender cómo influyen en la calidad de los datos capturados.

2.1.1 Funcionamiento

El principio de funcionamiento de los sensores LiDAR se basa en la técnica *Time of Flight* (ToF), mediante la cual se determina la distancia a un objeto midiendo el tiempo que tarda un pulso láser en emitirse, reflejarse en una superficie y volver al receptor [18]. El sensor cuenta con un emisor láser que proyecta pulsos de luz infrarroja en direcciones controladas, y un fotodetector que registra el eco de retorno. Como se ilustra en la Figura 2.1, se emite un pulso hacia el entorno y mide el retardo

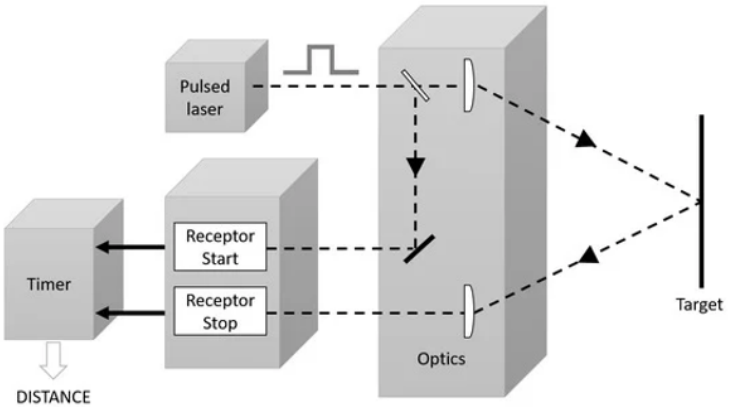


Figura 2.1: Esquema del funcionamiento de un sensor LiDAR mediante técnica ToF. Obtenida de [18].

temporal del eco reflejado, lo que permite estimar la distancia d a través de la siguiente expresión [18]:

$$d = \frac{c \cdot t}{2}$$

donde c es la velocidad de la luz en el vacío y el factor $\frac{1}{2}$ se introduce para tener en cuenta que el tiempo medido corresponde al trayecto de ida y vuelta del pulso.

En sistemas avanzados, el láser se acopla con elementos móviles o direccionales, como espejos rotatorios o sistemas MEMS (*Micro Electro Mechanical Systems*), que permiten escanear el entorno y generar una nube de puntos tridimensional [18]. Además, los sensores LiDAR embarcados suelen incorporar una unidad de medición inercial (IMU, del inglés *Inertial Measurement Unit*) y receptores *Global Positioning System* (GPS), que permiten referenciar espacialmente cada punto capturado y compensar el movimiento del sensor durante la exploración [18]. Cada punto registrado en la nube contiene como mínimo las coordenadas tridimensionales (x, y, z), y opcionalmente información adicional como la intensidad de retorno, es decir, la potencia relativa del pulso reflejado. Esta característica depende de la reflectancia y rugosidad del material impactado, y puede utilizarse como atributo adicional en tareas de segmentación semántica o clasificación mediante técnicas de aprendizaje automático [18].

2.1.2 Tipología

Los sensores LiDAR pueden clasificarse en distintas categorías según su arquitectura de funcionamiento, su método de escaneo y la técnica de emisión láser utilizada [17, 19]. Esta clasificación permite entender las diferencias fundamentales entre los dispositivos disponibles y seleccionar el más adecuado en función del entorno operativo y el tipo de datos requeridos. En particular, en tareas como

Figura 2.2: Sensor Ouster Multi-Layer 3D-LiDAR OS0 Rev 7 128.

la segmentación semántica tridimensional de entornos no estructurados, la elección del sensor incide directamente en la densidad de puntos, la calidad geométrica y la robustez de la representación espacial del entorno.

Una distinción principal se establece entre los sensores mecánicos rotatorios y los de estado sólido [17, 19]. Los sensores LiDAR rotatorios emplean un conjunto de emisores y receptores láser montados sobre una plataforma giratoria que permite una cobertura angular completa, típicamente de 360° en el plano horizontal. Estos sensores, como el *Velodyne HDL-64E* o el *VLP-16*, generan nubes de puntos densas con alta resolución y son ampliamente utilizados en plataformas móviles, robótica de exterior y en la mayoría de bases de datos como SemanticKITTI [20], GOOSE o RELLIS-3D. Por otro lado, los sensores LiDAR de estado sólido prescinden de partes móviles, lo que incrementa su robustez y reduce su tamaño, coste y consumo energético [19]. Existen distintas implementaciones dentro de esta categoría, entre las que destacan los sensores *flash*, que iluminan toda la escena de forma simultánea, y los MEMS, que dirigen electrónicamente el haz mediante pequeños espejos oscilantes [19]. Estas soluciones son especialmente prometedoras en aplicaciones embarcadas, donde el volumen, la fiabilidad mecánica y la integración con otros sistemas son prioritarios, aunque su alcance y resolución son en general inferiores a los de los sensores rotatorios. En la Figura 2.3 se muestra un sensor LiDAR físico, en concreto el Ouster Multi-Layer 3D-LiDAR OS0 Rev 7 con 128 canales, un dispositivo de última generación capaz de capturar nubes de puntos 3D de alta resolución con un campo de visión horizontal de 360° y un rango de detección optimizado para entornos cercanos y medianos, lo que lo convierte en una solución avanzada para aplicaciones de percepción en robótica móvil y vehículos autónomos.

Adicionalmente, los sensores pueden clasificarse por el tipo de señal emitida. En los sistemas pulsados, el sensor emite pulsos individuales de alta potencia y mide el tiempo de vuelo de cada uno, lo que permite alcanzar mayores distancias de detección y una mejor respuesta frente a superficies poco

reflectantes [17]. En contraste, los sensores basados en modulación continua, como los LiDAR *Amplitude Modulated Continuous Wave* (AMCW) o los denominados *Frequency Modulated Continuous Wave* (FMCW) permiten medir la distancia mediante el desfase o el desplazamiento en frecuencia de la señal recibida [17]. Aunque estos últimos ofrecen ciertas ventajas en términos de seguridad ocular y resolución a distancias cortas, su uso aún es limitado en aplicaciones exteriores de largo alcance debido a restricciones técnicas [19].

2.1.3 Características Relevantes

Los sensores LiDAR presentan un conjunto de características técnicas que determinan su rendimiento, especialmente en tareas de percepción tridimensional. Estos parámetros definen la densidad de la nube de puntos, la precisión geométrica de la escena reconstruida y la aplicabilidad de los datos en contextos reales. Uno de los aspectos más determinantes es el campo de visión (*Field of View*, FOV), que indica el ángulo que el sensor puede cubrir tanto en el eje horizontal como en el vertical. En sensores rotatorios, como los utilizados en la mayoría de los sistemas móviles, el FOV horizontal suele ser de 360°, mientras que el vertical depende del número de canales láser, oscilando generalmente entre los 10° y los 40° [18, 21]. Este rango angular influye directamente en la capacidad del sensor para capturar una representación completa del entorno en una sola rotación (véase Figura 2.3). Vinculado a esto se encuentra la resolución angular, que hace referencia a la separación entre dos mediciones consecutivas [21]. Cuanto menor es esta separación, mayor es la densidad de puntos generados, lo que permite representar con mayor detalle objetos pequeños o superficies irregulares. Esta propiedad es especialmente crítica en escenarios naturales, donde no existen formas geométricas regulares y es necesario captar variaciones sutiles del terreno o de la vegetación. La resolución angular depende directamente del motor y del codificador óptico que incorpora el sensor, por ejemplo, *encoders* de disco metálico de alta resolución (4 pulgadas) pueden alcanzar precisiones angulares de hasta 0.072°, lo que permite generar nubes de puntos de gran detalle incluso en escenas complejas [21]. A ello se suma el rango máximo de detección, que determina la distancia límite a la que el sensor puede registrar un retorno útil. Este valor depende de varios factores, como la potencia del emisor, la sensibilidad del receptor, la reflectividad del objeto escaneado y las condiciones atmosféricas [18].

Otro aspecto clave es la tasa de muestreo o frecuencia de escaneo, que define el número de puntos capturados por segundo. Esta tasa puede variar desde decenas de miles hasta varios millones de puntos por segundo, y afecta directamente a la densidad espacial y temporal de los datos. Una mayor frecuencia permite obtener una representación continua y detallada incluso en plataformas móviles en movimiento, lo cual es fundamental en sistemas de navegación y percepción en tiempo real. Además, muchos sensores LiDAR no se limitan a registrar la posición tridimensional de los puntos, sino que también incorporan atributos adicionales [21]. Entre ellos, destaca la intensidad de remisión, que representa la cantidad de energía reflejada por la superficie impactada [18, 21]. Este valor depende de

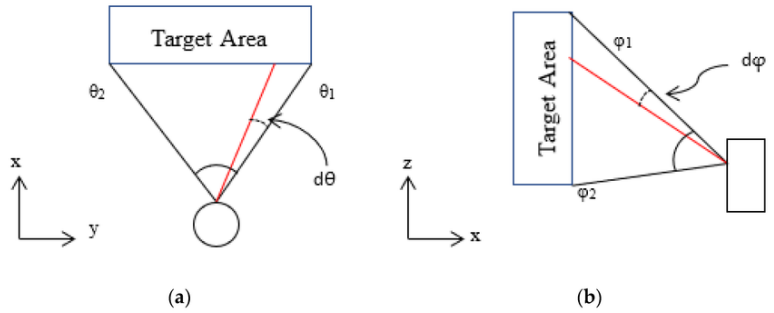


Figura 2.3: Representación esquemática del campo de visión (FOV) de un sensor LiDAR. (a) Vista en planta, muestra la cobertura horizontal definida por los ángulos θ_1 y θ_2. (b) Vista lateral, muestra la cobertura vertical definida por los ángulos φ_1 y φ_2. Obtenida de [21].

la reflectancia, textura y material del objeto, y puede ser utilizado como entrada adicional en modelos de aprendizaje automático para mejorar la segmentación y clasificación, ya que aporta información complementaria a la puramente geométrica.

2.1.4 Ventajas

Los sensores LiDAR incorporan diversas propiedades técnicas que les otorgan ventajas significativas frente a otras tecnologías de adquisición de datos espaciales. Entre las más relevantes se encuentran [17]:

- **Alta precisión geométrica.** Los sistemas LiDAR permiten medir distancias con gran exactitud mediante la técnica ToF, alcanzando errores del orden de pocos centímetros incluso a largas distancias. Esta precisión facilita la reconstrucción detallada de la geometría del entorno escaneado.
- **Independencia de la iluminación ambiental.** Al tratarse de un sistema activo, el sensor LiDAR emite su propia señal, lo que le permite operar de forma efectiva tanto en condiciones de alta luminosidad como en completa oscuridad. La calidad de los datos no se ve afectada por la variabilidad de la iluminación externa.
- **Alta densidad de puntos.** Gracias a su elevada frecuencia de escaneo y al uso de múltiples canales láser, los sensores LiDAR pueden generar millones de puntos por segundo, lo que proporciona una representación espacial continua y detallada. La densidad resultante mejora la resolución de detalles finos en superficies y estructuras.
- **Captura de múltiples retornos.** Algunos dispositivos LiDAR son capaces de registrar varios retornos por cada pulso láser emitido, lo que permite detectar diferentes capas de objetos o

estructuras presentes en una misma línea de visión, como ramas, suelos o superficies superpuestas.

- **Atributos adicionales por punto.** Además de las coordenadas espaciales, muchos sensores incorporan información como la intensidad del retorno, número de retorno, tiempo de adquisición o canal láser. Estos atributos enriquecen los datos y pueden ser utilizados como variables complementarias en distintos procesos de análisis.

2.1.5 Limitaciones

A pesar de sus múltiples ventajas, los sensores LiDAR presentan ciertas limitaciones técnicas y operativas que deben ser consideradas. Entre las principales se encuentran [17]:

- **Coste elevado.** Los sistemas LiDAR con altas prestaciones, como múltiples canales y gran resolución angular, presentan un coste significativamente mayor que otros sensores de percepción espacial. Este factor puede limitar su adopción en ciertas aplicaciones que requieren soluciones de bajo coste.
- **Sensibilidad a condiciones atmosféricas adversas.** La precisión del sensor puede verse afectada por fenómenos como niebla, lluvia, nieve, humo o partículas en suspensión, que atenúan el haz láser o introducen ruido en las mediciones. Estas condiciones reducen la fiabilidad de los datos adquiridos.
- **Ausencia de información visual o espectral.** Los sensores LiDAR registran únicamente información geométrica y radiométrica simple, como la intensidad del retorno, pero no capturan color ni textura superficial. Esto limita la posibilidad de distinguir entre materiales u objetos similares sin sensores complementarios.
- **Generación masiva de datos y requisitos de procesamiento.** Los sensores LiDAR generan grandes volúmenes de datos en forma de nubes de puntos, lo que exige sistemas de almacenamiento y procesamiento de alto rendimiento, especialmente en tareas en tiempo real o con altas frecuencias de escaneo.
- **Campo de visión vertical limitado.** En muchos sensores rotatorios, el ángulo de escaneo vertical está restringido por el número de canales láser disponibles. Esta limitación reduce la cobertura en el eje vertical y puede dificultar la detección de elementos situados fuera del rango angular efectivo.

2.2 Nubes de Puntos

Las nubes de puntos constituyen una representación tridimensional formada por colecciones de muestras discretas que describen la geometría de superficies u objetos mediante coordenadas cartesianas (x, y, z) [22]. Cada punto puede incorporar atributos adicionales, como intensidad de retorno, color, tiempo de adquisición o número de retorno, que enriquecen la descripción del entorno y permiten diferenciar materiales o propiedades reflectantes.

Se trata de un tipo de dato de alta dimensionalidad y estructura inherentemente irregular, ya que los puntos no siguen un orden espacial uniforme ni definen conectividad topológica entre ellos [11]. La densidad local de las muestras puede variar de forma notable dentro de una misma captura debido a factores como la distancia al sensor, el ángulo de incidencia del haz y la reflectividad de las superficies. Además, la ausencia de correspondencias explícitas entre puntos individuales implica que cada muestra se considera independiente de las demás, por lo que la nube de puntos se modela como una lista no estructurada de registros geométricos y atributos asociados. Estas características hacen que la nube de puntos sea un dato extremadamente detallado y flexible, pero también heterogéneo y complejo de manejar en tareas que requieren coherencia espacial o relaciones locales.

Para su almacenamiento e intercambio se utilizan diversos formatos reconocidos, entre ellos `PLY` [23], `LAS` y `PCD`, que permiten definir cabeceras con la estructura de los campos y almacenar metadatos y anotaciones semánticas a nivel de punto. También son habituales formatos binarios más compactos, como archivos `.bin`, en los que los datos se guardan en secuencia binaria sin cabecera, o contenedores propietarios que almacenan registros en un *layout* fijo (por ejemplo, coordenadas e intensidad en flotante de 32 bits) [22]. La flexibilidad de estos esquemas ha facilitado la adopción de las nubes de puntos como recurso fundamental en el desarrollo de sistemas de percepción 3D en robótica, donde la representación detallada del entorno es esencial para tareas de segmentación, clasificación y reconstrucción [11].

2.2.1 Representación y Estructuración de Nubes de Puntos

Las nubes de puntos tridimensionales, como las capturadas por sensores LiDAR, presentan características que dificultan su procesamiento directo mediante técnicas tradicionales de visión por computador. A diferencia de las imágenes, las nubes de puntos no se organizan en una estructura regular, sino que constituyen un conjunto de muestras distribuidas en el espacio euclídeo tridimensional sin orden explícito, sin densidad uniforme ni conectividad predefinida [11]. Esta irregularidad, combinada con la dispersión inherente del formato y la ausencia de correspondencias topológicas, plantea retos significativos para el desarrollo de métodos de representación y preprocesamiento adecuados. Diversas estrategias han sido propuestas con el fin de adaptar estas representaciones a técnicas convencionales

de análisis:

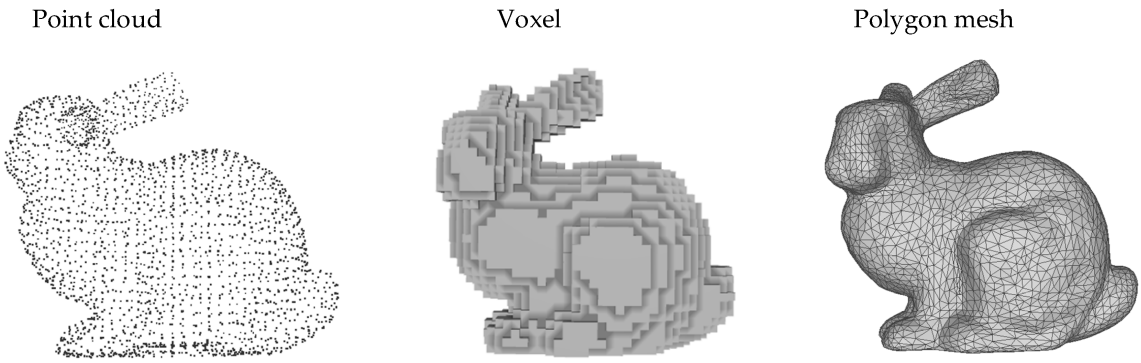


Figura 2.4: Representación de un modelo 3D en nube de puntos, vóxeles y mallas poligonales. Obtenida de [24].

- **Voxelización**: consiste en discretizar el espacio tridimensional en una cuadrícula regular de vóxeles, donde cada celda almacena información de ocupación o características agregadas de los puntos contenidos. Este enfoque facilita operaciones de convolución 3D, pero introduce cuantización espacial, pérdida de detalle geométrico y un coste computacional elevado cuando se requiere alta resolución [11, 25].

- **Reconstrucción de mallas**: transforma la nube en una superficie poligonal aproximada mediante algoritmos como triangulación de Delaunay o reconstrucción de Poisson. Este método genera una representación continua y facilita cálculos geométricos, aunque depende de supuestos sobre la conectividad de los datos y puede producir errores de interpolación en regiones con baja densidad [11].

- **Estructuras de grafos y agrupamiento**: se basa en construir relaciones locales mediante estructuras de vecindad, como grafos de k-vecinos más próximos (*k*-NN), *octrees* o árboles KD. Estas representaciones permiten modelar dependencias espaciales y facilitan operaciones locales, si bien implican un preprocesamiento costoso y presentan desafíos de escalabilidad en grandes volúmenes de datos [11].

La Figura 2.4 muestra una comparación visual de un mismo objeto 3D representado en tres formatos distintos. A la izquierda se muestra la nube de puntos, donde solo se representan muestras puntuales discretas de la superficie. En el centro se observa la representación voxelizada, que aproxima el volumen del objeto mediante pequeños cubos. A la derecha se presenta la malla poligonal, que define la geometría mediante una red de triángulos conectados, generando una superficie continua y más detallada. Cada una de estas técnicas conlleva compromisos entre fidelidad geométrica, eficiencia

computacional y robustez frente a variaciones de densidad y ruido. La selección del método de preprocesamiento adecuado depende del tipo de tarea, de los requisitos de precisión y de las restricciones computacionales.

2.3 Aprendizaje Profundo en Segmentación Semántica de Nubes de Puntos

Numerosos modelos de aprendizaje profundo han sido propuestos para la segmentación semántica 3D. Los enfoques clásicos procesan directamente los datos LiDAR como nubes de puntos no estructuradas, sin suposiciones adicionales o discretización espacial previa. Este es el caso de PointNet [26] y PointNet++ [14]. Estas arquitecturas establecieron una sólida base para trabajos posteriores que exploraron mejoras como la novedosa estrategia de muestreo de RandLA-Net [16] y la convolución basada en *kernel-points* de KPConv [15].

Otras soluciones se han centrado en abordar el problema de la escasez de puntos mediante diferentes métodos de voxelización. Por ejemplo, MinkUNet [27], voxeliza las nubes de puntos y aplica capas convolucionales 3D siguiendo una arquitectura similar a UNet. Por su parte, Cylinder3D [28] realiza la voxelización en coordenadas cilíndricas y SPVCNN [29] adopta un enfoque híbrido. Soluciones alternativas transforman las nubes de puntos en estructuras tipo grafo [30] o imágenes de rango [31] antes de su procesamiento. Más recientemente, arquitecturas basadas en Transformers, como PointTransformer [32] y SphereFormer [33], aprovechan los mecanismos de atención para mejorar aún más el modelado del contexto.

El procesamiento de nubes de puntos 3D mediante redes neuronales profundas presenta desafíos únicos en comparación con datos estructurados como las imágenes. Las nubes de puntos son, por definición, conjuntos desordenados de puntos en el espacio, lo que implica que su representación debe ser invariante a cualquier permutación en el orden de los puntos. Además, el número de puntos en una nube puede variar significativamente, y los modelos deben ser robustos ante transformaciones geométricas como la rotación y la traslación.

El presente trabajo se centra en PointNet y PointNet++ como objeto principal de estudio por varias razones. Por un lado, estas arquitecturas representan la base conceptual sobre la que se han construido muchos de los métodos de segmentación semántica más avanzados en la actualidad, por lo que su análisis proporciona una comprensión fundamental de los principios que sustentan el estado del arte. Además, PointNet y PointNet++ son capaces de procesar directamente nubes de puntos brutas sin requerir conversiones intermedias a formatos regulares, lo que permite preservar de forma más precisa la información geométrica original. Finalmente, su eficiencia computacional y su relativa

simplicidad conceptual las convierten en una opción adecuada para establecer un primer contacto con este tipo de modelos y realizar experimentos controlados que faciliten la interpretación de los resultados.

2.3.1 Arquitectura PointNet

PointNet [26] fue una arquitectura innovadora, siendo una de las primeras en procesar directamente nubes de puntos sin convertirlas a formatos intermedios regulares como vóxeles o imágenes 2D. La idea clave de PointNet es aprender una codificación espacial para cada punto individualmente y luego agregar todas las características de puntos individuales en una representación global.

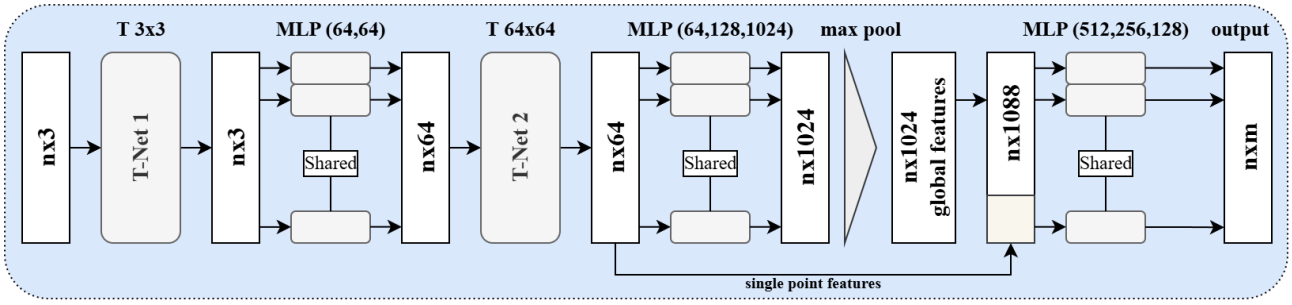


Figura 2.5: Diagrama de bloques de la arquitectura PointNet.

Para lograr la invarianza a la permutación, PointNet utiliza una estrategia simple y potente. La arquitectura, visible en la Figura 2.5, se basa en los tres bloques funcionales que se explican a continuación:

- **Perceptrones Multicapa Compartidos:** Cada punto en la nube se procesa de forma independiente e idéntica a través de una serie de perceptrones multicapa (*Multilayer Perceptrons*, MLPs) compartidos. Esto asegura que la red aprenda a extraer el mismo tipo de características para cada punto, independientemente de su posición en el orden de entrada.
- **Función Simétrica:** Se aplica una función simétrica, específicamente la operación de *max pooling*, a lo largo de la dimensión de características de todos los puntos. Esta función agrega las características aprendidas de todos los puntos en un único vector de características global que describe la forma completa. Esta agregación es inherentemente invariante al orden de los puntos.
- **Redes de Transformación (T-Nets):** Para asegurar la invarianza a las transformaciones geométricas, PointNet incorpora pequeñas redes neuronales llamadas T-Nets. Estas redes predicen una matriz de transformación afín que se aplica a los puntos de entrada y a sus características, estandarizando los datos antes de la extracción de características.

Para la segmentación semántica, donde se requiere una etiqueta para cada punto, PointNet concatena el vector de características global con las características por punto aprendidas antes de la capa de *max pooling*. Esta combinación proporciona tanto el contexto global como la información local para cada punto, permitiendo que la red realice una predicción final por punto.

2.3.2 Arquitectura PointNet++

Aunque PointNet fue revolucionaria, no capturaba explícitamente la estructura geométrica local inducida por el espacio métrico de los puntos. Trataba cada punto de forma independiente antes de la agregación global, perdiendo detalles finos en regiones locales. PointNet++ [14] fue propuesto para abordar esta limitación, introduciendo un enfoque de aprendizaje de características jerárquico que captura características a múltiples escalas.

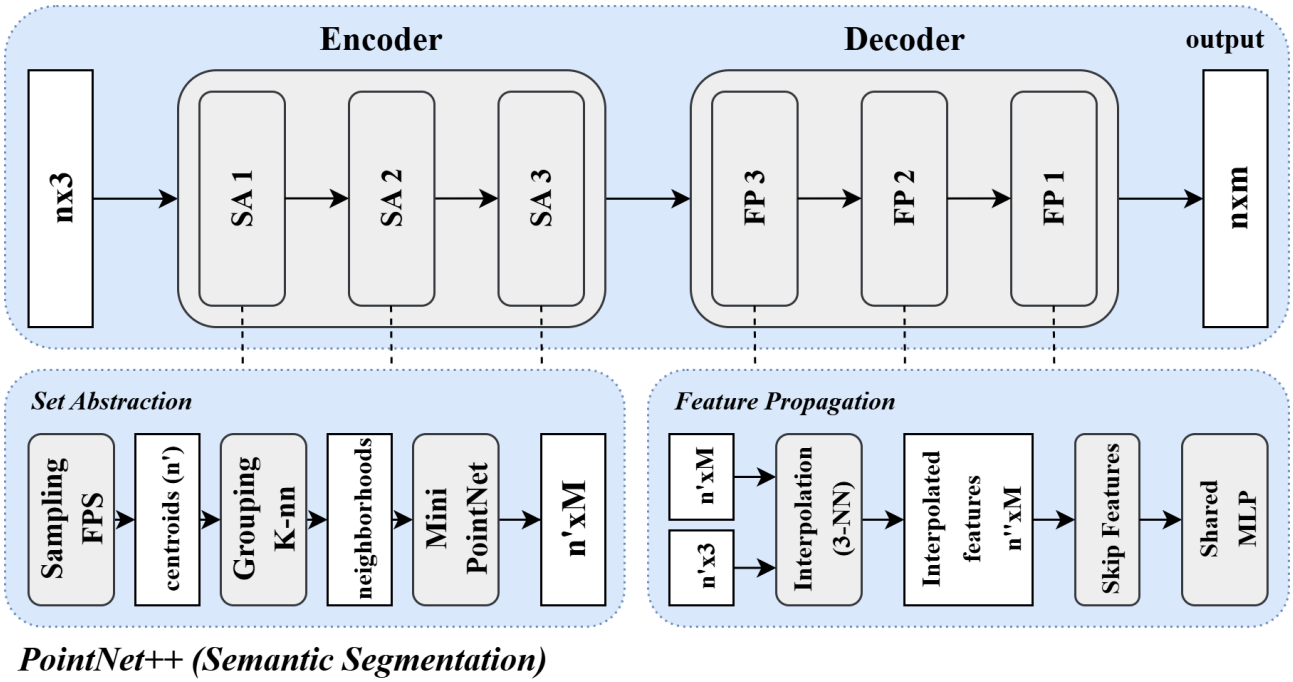


Figura 2.6: Diagrama de bloques de la arquitectura PointNet++.

La arquitectura de PointNet++ (Figura 2.6) se construye sobre una serie de módulos de *Set Abstraction (SA)*, que progresivamente abstraen una región cada vez más grande de la nube de puntos en un vector de características de mayor dimensionalidad. Cada módulo SA consta de tres capas clave:

- **Capa de Muestreo:** Se selecciona un subconjunto de puntos del conjunto de entrada, definiendo los centroides de las regiones locales. Se utiliza el algoritmo de Muestreo del Punto más Lejano (*Farthest Point Sampling*, FPS) para asegurar que los centroides cubran toda la nube de puntos, proporcionando un campo receptivo mejor que el muestreo aleatorio.

- **Capa de Agrupamiento:** Para cada centroide se define una región local encontrando todos los puntos dentro de un cierto radio (o k-vecinos más cercanos). Estos puntos forman una vecindad local.
- **Capa PointNet:** Se utiliza un mini-PointNet para procesar cada región local, extrayendo un vector de características de nivel superior que resume el patrón geométrico dentro de esa vecindad.

Al apilar estos módulos SA, PointNet++ crea una jerarquía de codificación. En cada nivel, el número de puntos se reduce, pero la dimensionalidad de su representación de características se incrementa. Este proceso es análogo a las capas convolucionales y de *pooling* en una CNN, que reducen la resolución espacial mientras aumentan el número de canales de características. Para tareas de predicción densa como la segmentación semántica, las características deben propagarse de vuelta a la nube de puntos original y de resolución completa. PointNet++ logra esto a través de una arquitectura jerárquica de propagación de características que funciona como un decodificador. La propagación de características se realiza nivel por nivel, utilizando interpolación y conexiones residuales (*skip connections*):

- **Interpolación:** En la etapa del decodificador, las características aprendidas en niveles superiores (donde hay menos puntos y la información es más global) se transfieren de vuelta a niveles inferiores y más densos. Esta transferencia se realiza interpolando las características de los puntos vecinos más cercanos, asignando mayores pesos a los puntos más próximos (interpolación ponderada por distancia inversa).
- **Conexiones Residuales (Skip Connections):** Tras la interpolación, estas características se combinan con las almacenadas desde el codificador en el mismo nivel jerárquico. Esto permite que la red recupere detalles locales finos que pueden haberse perdido durante el muestreo jerárquico, al tiempo que aprovecha el contexto global capturado en niveles más gruesos. Como resultado, el modelo puede tomar decisiones por punto utilizando tanto información contextual amplia como información local precisa.
- **MLP Compartido:** Después de esta concatenación, las características combinadas se pasan a través de un PointNet unitario (MLPs compartidos) para actualizar las características por punto. Este proceso se repite hasta que las características se han propagado a todos los puntos originales, de nivel en nivel.

Esta estructura jerárquica con sus fases de codificación y decodificación permite al modelo aprender características robustas y detalladas a múltiples escalas, lo que conduce a un rendimiento superior en tareas complejas de comprensión de escenas.

Capítulo 3

Bases de Datos en Percepción Tridimensional

La disponibilidad en la comunidad científica de bases de datos tridimensionales etiquetadas ha sido un factor clave en el desarrollo y validación de algoritmos de segmentación, clasificación y detección en nubes de puntos. Estos conjuntos de datos proporcionan secuencias o capturas individuales adquiridas mediante sensores LiDAR, generalmente acompañadas de anotaciones semánticas a nivel de punto, necesarias para entrenar y evaluar modelos de aprendizaje profundo supervisado. Las bases de datos orientadas a percepción tridimensional varían en cuanto los escenarios en las que han sido tomadas, su resolución, densidad de puntos, número de clases, tipo de anotación y estructura interna. Además, el tipo de sensor utilizado y la configuración de captura condicionan las características geométricas y cualitativas de los datos disponibles.

Una parte considerable de los recursos disponibles se centra en entornos urbanos o semiurbanos, como es el caso de SemanticKITTI [34], nuScenes [35] o Toronto3D [36]. No obstante, también han surgido conjuntos de datos orientados a escenas naturales o entornos no estructurados, que presentan mayor variabilidad geométrica y densidades no uniformes. Entre estos, destacan recursos recientes como GOOSE y RELLIS-3D, que se caracterizan por ofrecer datos con etiquetado denso, buena resolución espacial y una estructura compatible con arquitecturas de segmentación modernas. A continuación, se describen en detalle estos dos conjuntos de datos, con el objetivo de presentar sus propiedades técnicas, formato de anotación y estructura interna, así como los sensores utilizados para su adquisición.

3.1 Conjunto de datos GOOSE

El conjunto de datos GOOSE [7], publicado en 2023, es un recurso multimodal orientado a la percepción en entornos exteriores. Su estructura incluye información sincronizada procedente de múltiples sensores embarcados, como cámaras RGB, unidades IMU, receptores GPS y un sensor LiDAR tridimensional. Esta combinación de modalidades permite capturar una representación rica del entorno, abarcando tanto aspectos visuales como geométricos y posicionales.

En el contexto de este trabajo, se utilizarán exclusivamente los datos provenientes del sensor LiDAR, los cuales proporcionan nubes de puntos anotadas a nivel de punto con etiquetas semánticas densas. Estas nubes han sido capturadas mediante una plataforma móvil recorriendo entornos na-

turales, y están orientadas a facilitar el entrenamiento y la evaluación de modelos de segmentación semántica 3D. En las siguientes subsecciones se describen las características clave del *dataset* GOOSE: los tipos de entornos registrados, el sensor empleado, la estructura de organización de los datos, la estrategia de etiquetado semántico y su relevancia para arquitecturas de aprendizaje profundo basadas en nubes de puntos. La información detallada se fundamenta en la documentación oficial del conjunto de datos [7].

3.1.1 Tipología de Entornos y Objetivo

GOOSE fue desarrollado con el propósito de facilitar la investigación en tareas de percepción tridimensional mediante datos capturados en espacios abiertos. GOOSE se compone exclusivamente de secuencias adquiridas en entornos rurales y naturales, como caminos de tierra, áreas con vegetación densa, márgenes forestales y zonas mixtas con obstáculos dispersos. Estas escenas presentan una estructura geométrica compleja, sin simetrías definidas, lo que plantea un desafío representativo para algoritmos de segmentación semántica punto a punto. Las trayectorias de adquisición recorren distintos escenarios y condiciones del terreno, incluyendo curvas, desniveles, vegetación con oclusiones parciales y superficies irregulares. La Figura 3.1 ilustra muestras procedentes del conjunto de datos, donde cada fila corresponde a una escena diferente. La primera columna contiene las imágenes RGB capturadas por el sistema de adquisición, que muestran la vista frontal del entorno. La segunda columna presenta la capa de segmentación semántica en imagen, generada como *ground truth*, con los colores que indican la categoría asignada a cada píxel. Por último, las representaciones de la tercera y cuarta columna muestran las nubes de puntos 3D con las anotaciones semánticas por punto en distintas vistas, en la que cada color refleja la clase a la que pertenece cada medición LiDAR.

El objetivo principal del conjunto de datos es proporcionar información realista, capturada con una resolución suficiente para permitir el entrenamiento efectivo de modelos basados en aprendizaje profundo, tanto en 2D como en 3D. La diversidad de escenarios incluidos permite evaluar la capacidad de generalización de los modelos sobre datos capturados en diferentes condiciones, incrementando su aplicabilidad en tareas de percepción tridimensional autónoma.

3.1.2 Configuración

El sistema de captura empleado en el conjunto de datos GOOSE incorpora una configuración específica de sensores LiDAR montados sobre el vehículo de investigación *MuCAR-3*, un Volkswagen Touareg. Estos sensores son los responsables de adquirir las nubes de puntos tridimensionales que constituyen el núcleo del recurso. En concreto, el sistema incluye un total de tres sensores LiDAR rotatorios de alto rendimiento, todos sincronizados para garantizar una adquisición coherente en el tiempo y el espacio. Los parámetros técnicos de los sensores LiDAR utilizados en la plataforma de

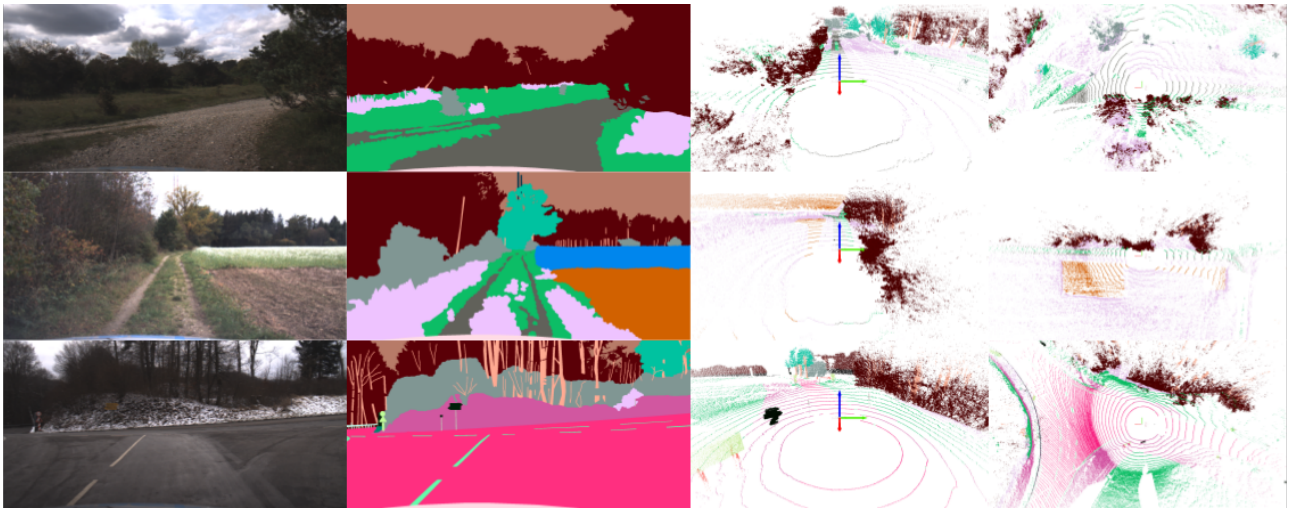


Figura 3.1: Ejemplos ilustrativos del conjunto de datos GOOSE. De izquierda a derecha: imagen RGB de la escena, máscara de segmentación semántica 2D, nubes de puntos LiDAR con etiquetado semántico. Imágenes reproducidas a partir de GOOSE [7].

adquisición GOOSE se resumen en el Cuadro 3.1.

La unidad principal es un *Velodyne Alpha Prime*, un sensor LiDAR de última generación con 128 canales láser y una frecuencia de escaneo de 10 Hz. Este dispositivo proporciona un FOV vertical de 40° y horizontal de 360°, lo que le permite capturar una representación completa del entorno en cada rotación. Se encuentra montado en el centro del sistema, en una posición elevada que le permite captar tanto la geometría frontal como superior con una alta resolución espacial. Destaca también por su alcance máximo de hasta 245 metros, lo que lo hace especialmente eficaz en entornos exteriores amplios, así como por su resolución angular vertical de 0.11° y horizontal variable entre 0.1° y 0.4°, permitiendo detectar detalles finos y estructuras pequeñas con gran fidelidad. Complementando esta unidad principal, se incorporan dos sensores *Ouster OS0*, cada uno con 32 canales láser y también operando a una frecuencia de 10 Hz. Al igual que el *Velodyne Alpha Prime*, ofrecen un FOV horizontal de 360°, pero amplían considerablemente el FOV vertical hasta los 90.8°, mejorando la percepción en zonas cercanas y con ángulos elevados. Estas unidades están montadas lateralmente, con una ligera inclinación hacia el exterior, con el objetivo de incrementar la cobertura lateral y minimizar puntos ciegos. Aunque su rango máximo es inferior (75 metros), son especialmente útiles para captar elementos del entorno próximo con un mayor campo vertical. En cuanto a resolución angular, alcanzan valores en torno a 0.35° en vertical y 0.175° en horizontal para una configuración típica de 1024 puntos por revolución, lo que garantiza una densidad de muestreo adecuada para tareas de segmentación y reconstrucción 3D a nivel local [37, 38]

Los tres sensores se encuentran fijados a la estructura del techo del vehículo, lo que asegura la estabilidad del sistema durante el movimiento. Todos los sensores están sincronizados temporalmente, lo que permite una fusión precisa de las distintas nubes de puntos generadas por cada unidad. Esta

Figura 3.2: Vehículo de adquisición MuCAR-3. Obtenida de https://goose-dataset.de/docs/mucar3/.

combinación de sensores LiDAR de distinto rango y ángulo permite capturar datos tridimensionales con una elevada cobertura espacial, tanto en alcance como en resolución vertical, maximizando la riqueza geométrica de los escenarios registrados en GOOSE.

Cuadro 3.1: Parámetros técnicos de los sensores LiDAR utilizados en GOOSE.

Sensor	N.º de canales	Frecuencia (Hz)	FOV vertical	FOV horizontal	Rango máximo (m)	Res. Ang. V	Res. Ang. H
Velodyne Alpha Prime	128	10	40°	360°	245	0.11°	0.1–0.4°
Ouster OS0	32	10	90.8°	360°	75	0.35°	0.175°

3.1.3 Estructura de los Datos y Anotaciones Semánticas

Los datos del *dataset* GOOSE se organizan en secuencias, donde cada escaneo LiDAR se almacena en un fichero binario (‘.bin’). Cada punto de la nube se representa con sus coordenadas tridimensionales (x, y, z) y la intensidad del retorno. De manera análoga a otros conjuntos de datos como SemanticKITTI o RELLIS-3D, las etiquetas semánticas se proporcionan en un fichero ‘.label’ correspondiente. El proceso de anotación se llevó a cabo mediante una combinación de pre-etiquetado automático y una posterior corrección y refinamiento manual para garantizar una alta fidelidad.

El conjunto de datos define un total de 64 clases semánticas diferentes que describen objetos y superficies presentes en entornos rurales, incluyendo terrenos, vegetación, construcciones, elementos de infraestructura, vehículos y peatones. Para facilitar la evaluación y la comparación con otros trabajos, estas 64 clases se agrupan en 13 categorías principales propuestas por los autores, que con-

solidan etiquetas detalladas en conjuntos más generales. Por ejemplo, múltiples tipos de vegetación baja y alta se unifican bajo categorías agregadas como *Drivable Vegetation* o *Non Drivable Vegetation*, mientras que las distintas estructuras construidas se agrupan en categorías como *Building* u *Object*. Esta organización jerárquica permite obtener métricas de segmentación tanto a nivel fino como a nivel de categoría, y ayuda a mitigar parcialmente el desequilibrio extremo que presentan algunas clases minoritarias.

3.2 Conjunto de datos RELLIS-3D

El conjunto de datos RELLIS-3D [8], presentado en 2020, es una base de datos multimodal diseñada específicamente para la percepción autónoma en entornos exteriores no estructurados. Su importancia radica en la alta calidad y diversidad de sus datos, que incluyen nubes de puntos LiDAR, imágenes RGB y datos de pose, capturados en escenarios complejos que combinan elementos naturales y artificiales. Estos entornos están caracterizados por su alta variabilidad espacial y espectral, incluyendo vegetación densa, terrenos irregulares, superficies con diferente reflectividad, y objetos dispares, lo que representa un desafío considerable para los métodos tradicionales de segmentación y reconocimiento basados en aprendizaje automático. Además, RELLIS-3D proporciona anotaciones semánticas detalladas para cada punto, permitiendo evaluar el rendimiento de algoritmos avanzados de segmentación semántica.

3.2.1 Tipología de Entornos y Objetivo

RELLIS-3D fue concebido para llenar un vacío importante en la investigación autónoma: la falta de datos etiquetados y estructurados para entornos *off-road*, que presentan desafíos particulares frente a los escenarios urbanos o estructurados. Las grabaciones se realizaron en el campus RELLIS de la Universidad de Texas A&M, un área diseñada para pruebas en vehículos autónomos todoterreno, que incluye una gran variedad de terrenos no estructurados. Estos abarcan caminos de tierra y grava, áreas cubiertas por césped, bosques con vegetación densa, arroyos y zonas fangosas o pantanosas. La heterogeneidad de estos ambientes produce complejidades importantes para los sistemas de percepción debido a la diversidad de formas, texturas, reflectancias y condiciones de iluminación. Además, la presencia de elementos artificiales dispersos como estructuras metálicas, vehículos estacionados o señales añade un grado adicional de dificultad. En la Figura 3.3 se muestran ejemplos representativos del conjunto de datos. Cada columna corresponde a una escena distinta capturada en entornos no estructurados. En la primera fila se presentan las imágenes RGB originales obtenidas por la cámara, que reflejan la apariencia visual de cada escenario. La segunda fila muestra la capa de segmentación semántica en imagen *ground truth*, donde cada región está coloreada en función de su clase. Final-

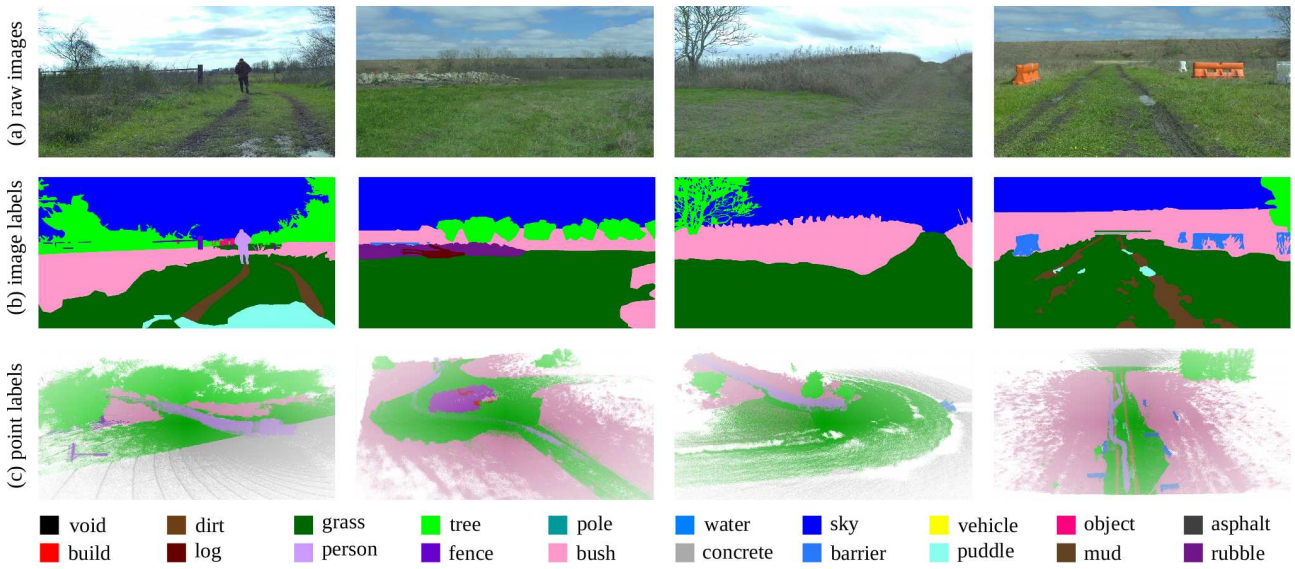


Figura 3.3: Ejemplos ilustrativos de los entornos y las capas de segmentación en RELLIS-3D: (a) imagen RGB de la escena capturada con cámara frontal, (b) máscara de segmentación semántica 2D y (c) nube de puntos LiDAR proyectada con etiquetas de clase. Obtenida de [8].

mente, la tercera fila representa la segmentación semántica 3D, donde cada punto de la nube LiDAR se anota con su etiqueta de clase correspondiente, permitiendo una asociación espacial precisa entre imagen y geometría.

El objetivo principal de RELLIS-3D es servir como banco de pruebas realista para desarrollar algoritmos que puedan manejar eficazmente la diversidad de entornos no estructurados, mejorando la robustez y generalización de los modelos para aplicaciones autónomas en escenarios reales y complejos fuera de carretera.

3.2.2 Configuración

La plataforma de adquisición de RELLIS-3D es un vehículo terrestre no tripulado (*Unmanned Ground Vehicle*, UGV) del modelo Clearpath Robotics Warthog [39], un robot anfibio de gran tamaño diseñado específicamente para operar en terrenos difíciles. Sobre esta robusta plataforma se monta el conjunto de sensores, asegurando la estabilidad y la capacidad de atravesar los complejos escenarios del campus RELLIS. La Figura 3.4 muestra la configuración de la plataforma de adquisición del *dataset*.

Para la captura de datos tridimensionales, el sistema emplea dos sensores LiDAR rotatorios complementarios montados en posiciones elevadas y centradas en el chasis del UGV, lo que proporciona una vista de 360° sin obstrucciones. El primero es un Ouster OS1-64 [40], una unidad de 64 canales que opera a una frecuencia de 10 Hz y genera una nube densa con hasta 2048 puntos por línea horizon-

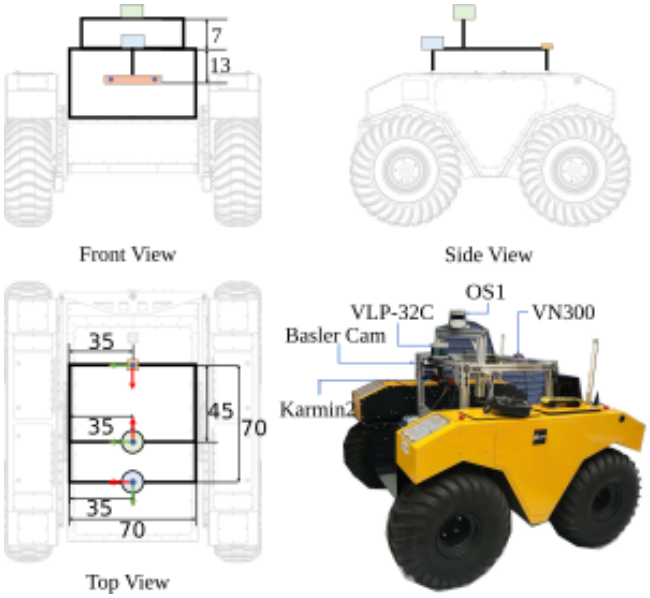


Figura 3.4: Plataforma de adquisición Warthog. Obtenida de
https://github.com/unmannedlab/RELLIS-3D?tab=readme-ov-file.

tal. Este sensor dispone de un campo de visión vertical de 45°, lo que facilita la captura detallada del entorno próximo y objetos elevados. El segundo sensor es un Velodyne Ultra Puck (VLP-32C) [41], con 32 canales, 10 Hz de frecuencia de rotación y un campo de visión vertical de 40°, que complementa al Ouster al proporcionar datos con diferente patrón angular y densidad. La combinación de ambos LiDAR permite obtener una representación tridimensional más rica y robusta, compensando parcialmente las limitaciones individuales de cada sensor.

Una de las características más destacadas del sistema es la alta densidad de puntos obtenida por la fusión de ambos dispositivos, que permite capturar de forma simultánea tanto el terreno inmediatamente cercano al vehículo (charcos, barro, vegetación baja) como elementos de mayor altura (árboles, postes o edificios). El alcance efectivo de los sensores es de hasta 120 metros en el caso del Ouster OS1-64 y aproximadamente 200 metros en el Velodyne Ultra Puck, con precisiones que oscilan entre $\pm$1,5 cm y $\pm$5 cm según la distancia y el ángulo de incidencia. Esta combinación de una plataforma todoterreno y un sistema LiDAR multimodal de alta resolución es clave para obtener los datos ricos y desafiantes que caracterizan a RELLIS-3D [8].

Cuadro 3.2: Parámetros técnicos de los sensores LiDAR utilizados en RELLIS-3D.

Sensor	**N.º de canales**	**Frecuencia (Hz)**	**FOV vertical**	**FOV horizontal**	**Rango máximo (m)**	**Res. Ang. V**	**Res. Ang. H**
Ouster OS1-64	64	10	45°	360°	120	∼0.7°	0.18°
Velodyne VLP-32C	32	10	40°	360°	200	∼1.33°	variable

3.2.3 Estructura de los Datos y Anotaciones Semánticas

Los datos del conjunto RELLIS-3D están organizados en cinco secuencias principales que comprenden un total de 13556 escaneos LiDAR en formato binario (‘.bin’). De estos escaneos, aproximadamente 6235 cuentan con anotaciones semánticas densas y de alta calidad, lo que permite una evaluación rigurosa de modelos de segmentación semántica. Cada archivo de escaneo contiene una nube de puntos donde cada punto está representado por sus coordenadas tridimensionales (x, y, z) junto con un valor de intensidad que indica la reflectancia del láser LiDAR en ese punto particular.

Las anotaciones asociadas se almacenan en archivos ‘.label’ que corresponden a cada escaneo y codifican una taxonomía de 20 clases semánticas distintas. Estas clases incluyen categorías naturales como vegetación, suelo, agua, y diferentes tipos de terrenos, así como elementos artificiales como vehículos, estructuras metálicas, postes o señales. La definición de estas clases está diseñada para capturar la complejidad y diversidad del entorno *off-road*. Esta riqueza en etiquetas semánticas convierte a RELLIS-3D en un recurso fundamental muy útil para el avance del estado del arte en segmentación semántica de nubes de puntos en entornos no estructurados [8].

Capítulo 4

Herramienta de Visualización LiDAR

Este capítulo se explica la herramienta software Lidar-Visualizer, desarrollada como parte de este Trabajo de Fin de Grado. El código fuente de la aplicación se encuentra disponible en el repositorio del proyecto[1]. La herramienta consiste en un programa para la inspección y análisis de datos del sensor LiDAR, tanto en tiempo real desde el simulador CARLA [6] como a partir de archivos. La principal motivación detrás de su desarrollo es la necesidad de contar con un sistema ágil y eficiente que permita a los investigadores y desarrolladores del proyecto GAIA depurar la configuración de sensores en el simulador o simplemente analizar secuencias de datos LiDAR. La aplicación se ha diseñado con un enfoque modular, permitiendo una fácil extensión y adaptación a futuras necesidades. Las funcionalidades clave desarrolladas para esta aplicación son:

- **Visualización en tiempo real desde CARLA.** La aplicación se conecta directamente a una instancia en ejecución del simulador CARLA, crea un sensor LiDAR virtual y muestra los datos recibidos al instante.
- **Reproducción de *datasets* en ficheros locales.** La aplicación permite cargar y reproducir secuencias de nubes de puntos almacenadas localmente en formato binario (`.bin`) o formato (`.ply`), facilitando el análisis de los conjuntos de datos.
- **Control interactivo de la vista.** La ventana de visualización 3D, gestionada por Open3D, permite al usuario manipular la cámara (rotación, zoom, desplazamiento) para inspeccionar la nube de puntos desde cualquier ángulo en el estacio tridimensional.
- **División de arquitectura.** La funcionalidad se ha separado en dos *scripts* independientes y especializados, `carla_viz.py` y `file_viz.py`, lo que permite una clara distinción entre la lógica de visualización en tiempo real y la de reproducción de archivos.

La Figura 4.2 muestra capturas de la visualización LiDAR de tres fuentes de datos coloreadas con un mapa de color que mapea el rango de los valores de intensidad de remisión, donde los valores morados representan menos intensidad y los valores rosas más intensidad. En primer lugar, se encuentra el LiDAR simulado en tiempo real desde el simulador CARLA, en el centro se sitúa una nube de puntos de GOOSE y por último una nube de puntos de RELLIS-3D.

[1] https://github.com/RoboticsLabURJC/2024-tfg-felix-martinez/tree/main/Lidar-Visualizer

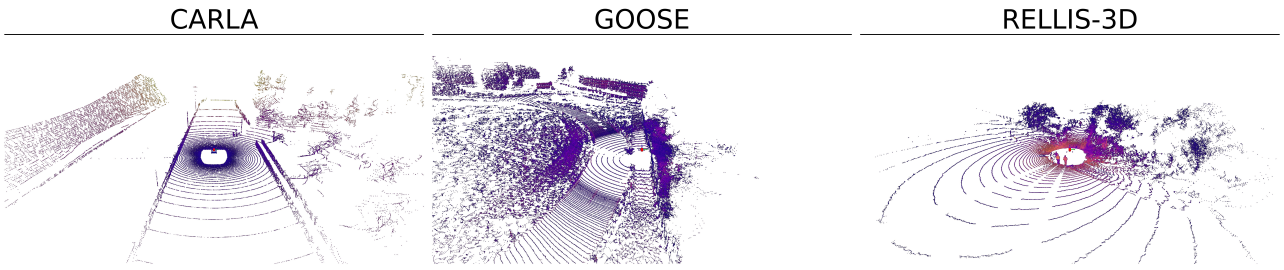


Figura 4.1: Visualización de nubes de puntos LiDAR provenientes de distintas fuentes (CARLA, GOOSE y RELLIS-3D) según su intensidad de remisión a través de la aplicación Lidar-Visualizer.

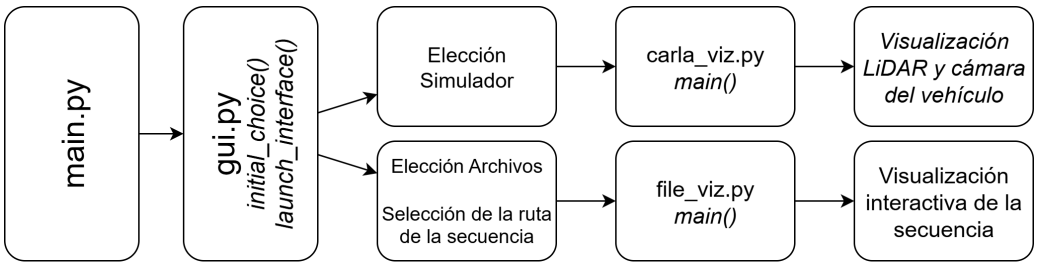


Figura 4.2: Diagrama de bloques de la arquitectura de Lidar-Visualizer.

Adicionalmente se grabaron dos vídeos ilustrativos de los dos modos de funcionamiento de la aplicación: visualización de secuencias LiDAR en local[2] y visualización de secuencias LiDAR en tiempo real con CARLA[3].

4.1 Arquitectura de la Aplicación

La arquitectura de la aplicación se estructura en torno a un conjunto de *scripts* de Python, cada uno desarrollado con una función bien definida. La Figura 4.2 muestra el diagrama de bloques de la arquitectura. La aplicación se inicia desde un punto de entrada principal que lanza una interfaz gráfica de usuario (GUI, del inglés *Graphical User Interface*), la cual actúa como un lanzador para los dos modos de visualización disponibles. Esta separación de responsabilidades en distintos archivos define la modularidad y el flujo de ejecución del programa. Los componentes principales de esta arquitectura son:

- `main.py`**:** Actúa como el punto de entrada principal de toda la aplicación. Su única función es iniciar la interfaz gráfica de usuario, pasando el control del programa al módulo de la GUI.

[2] https://youtu.be/QRldm8U4Yv4?si=n5dB2ay_yMgL6Bd8
[3] https://youtu.be/9qKonLBC3Ws?si=XYJQJwVBvuovdS4U

- `gui.py`**:** Este *script* implementa la interfaz gráfica de usuario utilizando la librería CustomTkinter. Proporciona al usuario una ventana sencilla con dos botones principales, cada uno diseñado para lanzar uno de los modos de visualización. Este módulo no realiza ninguna tarea de visualización LiDAR, sino que actúa como un lanzador que ejecuta los scripts `carla_viz.py` o `file_viz.py` como procesos separados, según la elección del usuario.

- `carla_viz.py`**:** Contiene toda la lógica para la visualización en tiempo real desde el simulador CARLA. Sus responsabilidades incluyen la conexión con el simulador, la creación y configuración de un sensor LiDAR y una cámara RGB, la gestión de la recepción de datos de forma asíncrona, el control manual del vehículo y el renderizado continuo de la nube de puntos con Open3D y de la imagen de la cámara con Pygame.

- `file_viz.py`**:** Este *script* está especializado en la reproducción de secuencias LiDAR desde archivos locales. Se encarga de leer los datos de nubes de puntos desde ficheros en formato `.bin` o `.ply`, iterar sobre ellos y ofrecer una visualización altamente interactiva con múltiples opciones de control para el usuario.

4.2 Visualización en Tiempo Real

El *script* `carla_viz.py` genera dos ventanas de visualización: una para la nube de puntos 3D (Open3D) y otra para la vista de una cámara RGB (Pygame). Una funcionalidad específica de este *script* es el procesamiento de los datos del sensor simulado. Para ello, se utiliza un diseño asíncrono donde la función `lidar_callback`, contenida en la Figura 4.3, se registra para ser invocada cada vez que el sensor produce datos. Esta función procesa los puntos e implementa la lógica para cambiar entre una vista relativa al vehículo y una vista absoluta en el mapa del mundo. Para la vista absoluta, se calcula la matriz de rotación del vehículo a partir de sus ángulos de Euler, y se utiliza su transpuesta para transformar los puntos desde el sistema de coordenadas del vehículo al sistema global. A continuación, se aplica un desplazamiento que sitúa cada punto en sus coordenadas globales. Además, para una mejor visualización, se mapea la intensidad de cada punto a un color.

Otra funcionalidad relevante es la gestión de la interacción del usuario, que se maneja de forma dual. Por un lado, Open3D, mediante `register_key_callback`, se usa para cambiar la perspectiva de la cámara en la visualización del LiDAR simulado. Por otro lado, Pygame gestiona los eventos de teclado en su propia ventana para alternar entre el modo de piloto automático y el control manual del vehículo. La función `vehicle_control` (Figura 4.4) centraliza esta lógica, leyendo el estado de las teclas y aplicando el control correspondiente al vehículo en modo manual. Esta función se invoca en cada iteración del bucle principal.

```python
def lidar_callback(lidar_data, point_cloud, vehicle_transform):
    global initial_vehicle_location

    data = np.copy(np.frombuffer(lidar_data.raw_data, dtype=np.float32))
    data = np.reshape(data, (int(data.shape[0] / 4), 4))
    data[:, 0] = -data[:, 0]  # Reflejar eje X

    if absolute_view:
        vehicle_location = np.array([
            -vehicle_transform.location.x,
             vehicle_transform.location.y,
             vehicle_transform.location.z
        ])
        pitch = vehicle_transform.rotation.pitch
        yaw = vehicle_transform.rotation.yaw
        roll = vehicle_transform.rotation.roll
        rotation_matrix = euler_to_rotation_matrix(pitch, yaw, roll).T

        points_in_world = []
        for point in data[:, :3]:
            rotated_point = rotation_matrix @ point
            global_point = rotated_point + vehicle_location
            points_in_world.append(global_point)
        points_to_display = np.array(points_in_world)

    else:
        points_to_display = data[:, :3]

    # Mapear intensidad a color usando la paleta VIRIDIS
    intensity = data[:, -1]
    int_color = np.c_[
        np.interp(intensity, VID_RANGE, VIRIDIS[:, 0]),
        np.interp(intensity, VID_RANGE, VIRIDIS[:, 1]),
        np.interp(intensity, VID_RANGE, VIRIDIS[:, 2])
    ]

    # Actualizar la nube de puntos en Open3D
    point_cloud.points = o3d.utility.Vector3dVector(points_to_display)
    point_cloud.colors = o3d.utility.Vector3dVector(int_color)

    # Imprimir informaci n de depuraci n
    print(f"Procesados {len(points_to_display)} puntos en modo {'absoluto' if absolute_view else 'relativo'}")

```

Figura 4.3: Fragmento de código en Python que implementa el procesamiento y visualización de datos LiDAR. Procesamiento de datos LiDAR y transformación de coordenadas (función `lidar_callback`, archivo `carla_viz.py`).

```python
def vehicle_control(vehicle):
    global manual_mode
    control = carla.VehicleControl()

    # ... gestion de eventos de Pygame para cambiar de modo ...
    if event.type == pygame.KEYDOWN and event.key == pygame.K_r:
        manual_mode = not manual_mode
        vehicle.set_autopilot(not manual_mode)
        # ...

    keys = pygame.key.get_pressed()
    if manual_mode:
        control.throttle = 1.0 if keys[pygame.K_w] else 0.0
        control.brake = 1.0 if keys[pygame.K_s] else 0.0
        control.steer = -0.3 if keys[pygame.K_a] else 0.3 if keys[pygame.K_d
    ] else 0.0
        vehicle.apply_control(control)

# En el bucle principal de main():
while True:
    vehicle_control(vehicle)
    # ... actualizacion de Open3D y tick de CARLA ...

```

Figura 4.4: Fragmento de código en Python que implementa la gestión de la interacción del usuario con Pygame (función `vehicle_control`, archivo `carla_viz.py`).

4.3 Visualización de secuencias LiDAR en local

El *script* `file_viz.py` está diseñado como una herramienta de análisis de datos interactiva. Una característica central de este es su sistema de control por teclado. Se utiliza un objeto de Open3D para registrar una gran cantidad de atajos. Cada atajo invoca a una función específica que modifica el estado de la visualización en tiempo real, como se muestra en la Figura 4.5. Esto permite al usuario cambiar la vista de cámara (`toggle_camera`), alternar entre mapas de color (`toggle_colormap`), pausar la reproducción, ajustar los FPS, o incluso cambiar entre un modo de reproducción automático y uno manual para avanzar y retroceder fotograma a fotograma.

Adicionalmente, se implementó un método de reproducción automática. En lugar de un bucle `for` con pausas `time.sleep()`, se utiliza un enfoque con `vis.register_animation_callback`. Esta función de Open3D registra la función `update_frame` (véase la Figura 4.6) para que sea llamada en cada ciclo de renderizado de la ventana. Dentro de `update_frame`, se comprueba si ha transcurrido el tiempo suficiente para cargar el siguiente fotograma según el número de FPS actuales. Este diseño proporciona una animación fluida y controlable, que no se ve afectada por el tiempo que tarden otras operaciones, y permite que la ventana siga siendo interactiva durante la reproducción.

```python
def vis_sequences(path, ...):
    # ... inicializacion ...
    vis = o3d.visualization.VisualizerWithKeyCallback()
    vis.create_window(...)

    # ... definicion de funciones callback (toggle_camera, toggle_colormap, etc.)

    # Registro de los atajos de teclado a las funciones
    vis.register_key_callback(ord("V"), toggle_camera)
    vis.register_key_callback(ord("C"), toggle_colormap)
    vis.register_key_callback(ord("B"), toggle_background)
    vis.register_key_callback(32, toggle_pause) # Barra espaciadora
    vis.register_key_callback(265, increase_fps) # Flecha arriba
    vis.register_key_callback(264, decrease_fps) # Flecha abajo
    vis.register_key_callback(ord("M"), toggle_mode)
    vis.register_key_callback(262, lambda vis: next_frame() if not is_auto_mode[0] else None)
    vis.register_key_callback(263, lambda vis: prev_frame() if not is_auto_mode[0] else None)
    # ...
    vis.run()

```

Figura 4.5: Fragmento de código en Python que implementa la configuración de la interactividad mediante *callbacks* de teclado (función `vis_sequences`, archivo: `file_viz.py`).

```python
def vis_sequences(path, ...):
    # ...
    last_update_time = [time.time()]
    FPS = [initial_fps]

    def update_frame(vis):
        if is_paused[0] or not is_auto_mode[0]:
            return
        current_time = time.time()
        if current_time - last_update_time[0] >= 1 / FPS[0]:
            next_frame() # Carga y muestra el siguiente frame
            last_update_time[0] = current_time

    vis.register_animation_callback(update_frame)
    vis.run()

```

Figura 4.6: Fragmento de código en Python que implementa la reproducción automática fluida (función `update_frame` dentro de `vis_sequences`, archivo `file_viz.py`).

Capítulo 5

Desarrollo Experimental y Resultados

En este capítulo se describe la metodología experimental empleada para entrenar, validar y evaluar diferentes arquitecturas de segmentación de nubes de puntos seleccionadas. Se expone el proceso de preparación de los datos, incluyendo la división en conjuntos y el mapeo de clases aplicados para los experimentos, así como la configuración de los procedimientos de entrenamiento y la búsqueda de hiperparámetros que permitieron optimizar el rendimiento de los modelos.

Asimismo, se presentan los resultados obtenidos con las implementaciones desarrolladas de PointNet y PointNet++, así como con una modificación propuesta sobre esta última, que se ha denominado PointNet++*, y que incorpora la información de la intensidad de remisión captada por el LiDAR en las capas finales de la red *(backbone)*. Finalmente, se presentan y analizan las figuras de mérito seleccionadas y las visualizaciones cualitativas que permiten valorar la capacidad de generalización del modelo sobre un conjunto de datos distinto, concretamente RELLIS-3D.

5.1 Preparación de Datos

Para la preparación de los datos se implementó un *DataLoader*, adaptado a las arquitecturas PointNet y PointNet++. Ambos comparten la misma lógica de carga y preprocesado, diseñada para manejar grandes volúmenes de nubes de puntos, realizar el submuestreo y remapear las etiquetas a las categorías definidas. El código que implementa estas funciones se muestra de forma detallada en la Figura 5.1. La carga de datos se basa en la lectura directa de archivos binarios `.bin`, que contienen las coordenadas tridimensionales de los puntos, y sus correspondientes archivos de etiquetas `.label`. Para cada muestra se aplican los siguientes pasos:

- **Filtrado por radio:** se descartan puntos cuya distancia al origen excede un umbral de 25 metros. Este preprocesado reduce la dispersión espacial y permite concentrar el entrenamiento en regiones de mayor densidad informativa y en una zona de interes para el problema.
- **Muestreo aleatorio:** si el número de puntos filtrados es suficiente, se seleccionan por defecto de forma aleatoria 4096 puntos sin reemplazo, es decir se extraen los índices originales de los puntos para extraer posteriormente sus etiquetas. Este procedimiento garantiza la variabilidad entre iteraciones durante el entrenamiento y un tamaño de entrada constante.

- **Remapeo de etiquetas:** se definió un esquema de agrupación de clases con coherencia semánticas que unifica las etiquetas originales del dataset GOOSE en nueve categorías semánticas principales: *Construction*, *Object*, *Road*, *Sign*, *Terrain*, *Drivable Vegetation*, *Non Drivable Vegetation*, *Vehicle* y *Void*. Para ello se generó un diccionario de correspondencia (`label_to_category`) que asocia cada etiqueta numérica con su clase agregada.
- **Semilla de muestreo:** ambos *DataLoaders* incorporan la posibilidad de fijar una semilla (`seed=42`) durante la validación y pruebas, asegurando la reproducibilidad en la selección de puntos. En el caso del entrenamiento, la semilla no se establece, favoreciendo la aleatoriedad entre épocas.

En conjunto, este procedimiento permite generar lotes (*batches*) de nubes de puntos consistentes, mantener una densidad uniforme y proporcionar un flujo de datos eficiente para el entrenamiento y validación de las arquitecturas. El número de puntos seleccionado por defecto es 4096, criterio que se ha utilizado como base experimental. No obstante, este parámetro se modifica posteriormente durante la búsqueda de hiperparámetros, explicado en la Sección 5.2, con el objetivo de optimizar el rendimiento del modelo.

```python
def map_labels(labels: np.ndarray) -> np.ndarray:
    return np.array([label_to_category.get(label, 8) for label in labels], dtype=np.uint8)

def load_bin_file(bin_path: str, num_points: int = 4096, radius: float = 25.0, seed: int = None):
    points = np.fromfile(bin_path, dtype=np.float32).reshape(-1, 4)[:, :3]
    distances = np.linalg.norm(points, axis=1)
    mask = distances <= radius
    points = points[mask]
    num_available = points.shape[0]
    if num_available >= num_points:
        if seed is not None:
            np.random.seed(seed)
        indices = np.random.choice(num_available, num_points, replace=False)
        return points[indices], np.where(mask)[0][indices]
    return None, None

def load_label_file(label_path: str, indices: np.ndarray) -> np.ndarray:
    labels = np.fromfile(label_path, dtype=np.uint32) & 0xFFFF
    return map_labels(labels[indices])
```

Figura 5.1: Fragmento de código en Python utilizado en ambos *DataLoaders* para la preparación de datos.

Además, dado que en el momento de realizar los experimentos el conjunto de *test* oficial de GOOSE no disponía de etiquetas públicas, se decidió reorganizar la división de los datos para poder evaluar los modelos. Concretamente, se reservó de forma proporcional un número de nubes de puntos de cada

escenario contenido en el conjunto de entrenamiento original, conformando así un nuevo conjunto de validación. Por su parte, el conjunto de validación proporcionado por GOOSE se empleó como conjunto de *test* final. Este procedimiento garantizó que todas las escenas quedaran representadas en las particiones y permitió establecer finalmente una distribución de los datos del 80 % para entrenamiento, 10 % para validación y 10 % para *test*.

5.2 Configuración de Entrenamiento y Búsqueda de Hiperparámetros

En esta sección se describen los principales componentes del entrenamiento, incluyendo las funciones de activación, la función de pérdida, el optimizador y otros parámetros relevantes que determinan el comportamiento del proceso de aprendizaje. Además, se explica la importancia de la búsqueda de hiperparámetros y la metodología utilizada para seleccionar las configuraciones más adecuadas. La sección se organiza en dos partes: la descripción de los componentes del entrenamiento y la búsqueda de hiperparámetros.

5.2.1 Componentes del Entrenamiento

El entrenamiento se configuró empleando un conjunto de componentes ampliamente utilizados, tales como optimizadores adaptativos, funciones de activación no lineales en las capas intermedias, funciones de pérdida ponderadas por clase y planificadores (*schedulers*) de la tasa de aprendizaje.

Uno de los factores que más condicionan el rendimiento en este tipo de tareas es el marcado desequilibrio en la frecuencia de aparición de las distintas categorías, que tiende a favorecer las clases mayoritarias y a dificultar la correcta identificación de aquellas que aparecen de forma minoritaria. En este caso particular, la Figura 5.2 ilustra la distribución de frecuencias de las clases presentes en el conjunto de entrenamiento de GOOSE, en la que se observa claramente este desequilibrio entre categorías. En particular, la clase *Non Drivable Vegetation* agrupa más del 50 % de los puntos anotados, mientras que otras categorías apenas superan el 1 % o incluso están ausentes en la mayoría de las muestras. Este patrón de distribución incrementa de forma significativa la dificultad de la tarea de segmentación semántica mediante aprendizaje automático y motivó la elección de una función de pérdida con ponderación por clase. Adicionalmente, debido a la escasez de ejemplos, las categorías *Sky* y *Water* se reasignaron a la clase *Void*, mientras que las clases *Animal* y *Human* se agruparon en la categoría *Object* con el fin de simplificar la clasificación y reducir el impacto del desequilibrio extremo. A continuación, se presentan los principales componentes seleccionados:

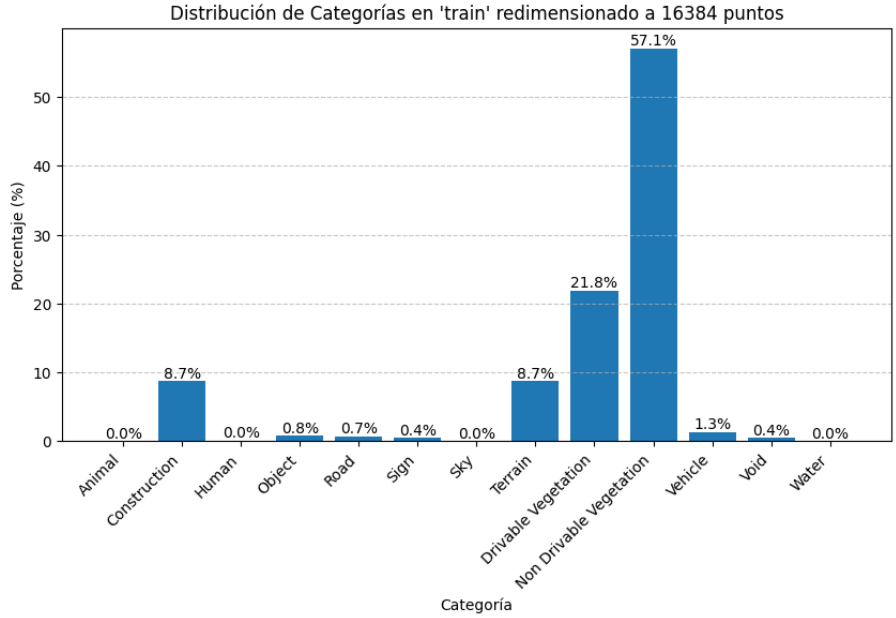


Figura 5.2: Distribución de categorías en el conjunto de entrenamiento de GOOSE, redimensionado a 16384 puntos por nube.

- **Función de pérdida:** se empleó NLLLoss (*Negative Log Likelihood Loss*) [42], que mide la discrepancia entre la distribución de probabilidad predicha por el modelo y la distribución real de las etiquetas. Esta función requiere que la salida de la red esté normalizada mediante `log-softmax` y permite introducir un vector de pesos que penaliza con mayor intensidad los errores en las clases minoritarias, contribuyendo a reducir el sesgo hacia las categorías mayoritarias.

- **Optimización:** se utilizó el optimizador Adam [43], un método basado en estimaciones adaptativas de momentos de primer y segundo orden, ampliamente reconocido por su estabilidad y eficiencia en entornos de alta variabilidad de gradientes. Adam permite ajustar de manera dinámica la tasa de aprendizaje de cada parámetro, facilitando la convergencia incluso cuando las distribuciones de los datos presentan fuertes desequilibrios.

- **Programación de la tasa de aprendizaje:** se incorporó un planificador dinámico (*scheduler*) mediante *ReduceLROnPlateau*. Este mecanismo monitoriza el valor de la función de pérdida en validación y reduce el *learning rate* si no se observa mejora durante un número definido de épocas consecutivas. Esta estrategia permite un ajuste progresivo de la magnitud de las actualizaciones y evita oscilaciones en la fase final del entrenamiento.

- **Funciones de activación:** las activaciones intermedias de la red se implementaron con la función ReLU (*Rectified Linear Unit*), que introduce no linealidad y ayuda a mitigar el problema del desvanecimiento de gradientes [44]. La Figura 5.3 muestra la representación gráfica de la

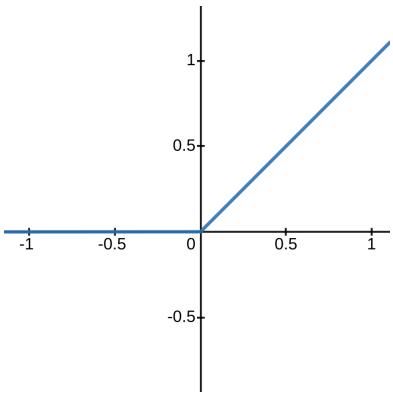


Figura 5.3: Representación gráfica de la función de activación ReLU.

función ReLU, donde los valores entrantes negativos o iguales a cero se mantienen en cero y los valores positivos se transmiten sin modificación. Esto permite preservar la propagación de gradientes en capas profundas y facilita una convergencia más rápida durante el entrenamiento.

- **Figuras de mérito:** la evaluación del desempeño de los modelos se realizó mediante la tasa de acierto global (*accuracy*), la IoU (*Intersection over Union*) y el *Recall*. La tasa de acierto global refleja el porcentaje de puntos correctamente clasificados sobre el total, mientras que la IoU mide el solapamiento espacial entre la segmentación predicha y el etiquetado real. Por su parte, el *Recall* indica la capacidad del modelo para identificar correctamente las instancias positivas. Estas figuras de mérito se calcularon mediante las expresiones [45, 46, 47]:

$$\text{Accuracy} = \frac{\text{TP}+\text{TN}}{\text{TP}+\text{FP}+\text{TN}+\text{FN}} \quad ; \quad \text{IoU} = \frac{\text{TP}}{\text{TP}+\text{FP}+\text{FN}} \quad ; \quad \text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}$$

donde, para cada clase, TP (verdaderos positivos) representa el número de puntos correctamente clasificados, FP (falsos positivos) indica los puntos que el modelo identificó erróneamente como pertenecientes a esa clase, FN (falsos negativos) son los puntos reales de la clase que el modelo no detectó y TN (verdaderos negativos) corresponde a los puntos correctamente identificados como no pertenecientes a la clase. La mIoU y el mRecall corresponden al promedio de estas medidas calculadas de manera individual sobre cada clase.

5.2.2 Búsqueda de Hiperparámetros

La búsqueda de hiperparámetros se planteó con el objetivo de identificar las combinaciones que ofrecieran un mejor compromiso entre precisión global, estabilidad durante la convergencia y capacidad de generalización. Para ello, se exploraron diferentes configuraciones de los principales hiperparámetros del entrenamiento, definidos en los siguientes rangos:

- **Tasa de aprendizaje inicial (*learning rate*):** se exploraron valores en el rango [0.001 , 0.01].
- **Tamaño del lote (*batch*):** se evaluaron configuraciones con 8, 16 y 32 muestras por lote.
- **Número de puntos por muestra:** se experimentó con 1024, 2048, 4096, 8192 y 16384 puntos, con el objetivo de valorar el impacto en la estabilidad del entrenamiento y el detalle espacial de las predicciones.
- **Factor de reducción de la tasa de aprendizaje:** se consideraron los factores de 0.5 y 0.8 en el *scheduler* `ReduceLROnPlateau`.

La combinación de hiperparámetros se evaluaron en el conjunto de validación de GOOSE en términos de precisión global, precisión por clase (*Recall*) y el valor medio del índice IoU (*Intersection over Union*). Del proceso de exploración, se identificó que la configuración más equilibrada correspondía a una tasa de aprendizaje inicial de 0.005, un tamaño de lote de 16 nubes de puntos, un número de puntos por nube de 16384 y un factor de reducción de la tasa de aprendizaje de 0.5. Esta combinación ofreció un equilibrio entre convergencia estable y maximización de las figuras de mérito, evitando el sobreajuste en las clases mayoritarias y manteniendo un nivel aceptable de precisión en las categorías menos representadas. En la siguiente sección se detalla el proceso de entrenamiento con esta configuración.

5.3 Entrenamiento de PointNet y PointNet++

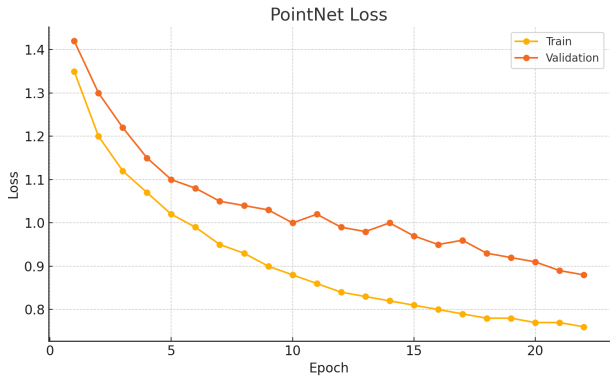
Para el entrenamiento de ambos modelos se empleó la configuración descrita en la Sección 5.2, utilizando como función de pérdida la `NLLLoss` ponderada con un vector de pesos calculado a partir de la distribución de frecuencias de las clases en el conjunto de entrenamiento. Estos pesos se determinaron como la inversa proporcional de la frecuencia relativa de cada clase y posteriormente se normalizaron para que su suma fuese igual a 1, resultando en el siguiente vector de ponderaciones:

$$\alpha = [0{,}028,\ 0{,}242,\ 0{,}295,\ 0{,}392,\ 0{,}023,\ 0{,}010,\ 0{,}004,\ 0{,}140,\ 0{,}428].$$

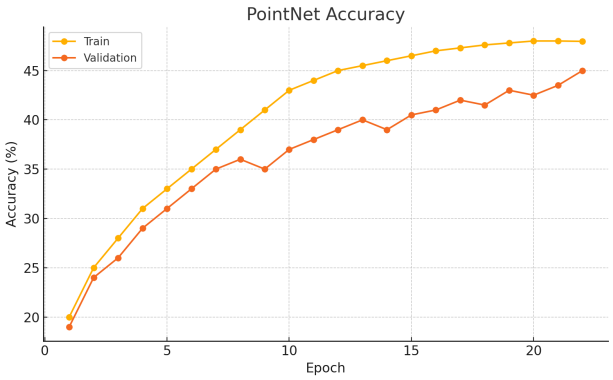
donde las clases menos predominantes, como *Sign* y *Object*, resultaron asociadas a ponderaciones considerablemente más altas en comparación con las categorías mayoritarias, penalizando con mayor intensidad los errores cometidos sobre ellas y contrarrestando parcialmente el desequilibrio extremo del *dataset*.

Se estableció un criterio de parada anticipada que interrumpía el entrenamiento si no se observaba mejora en la métrica de validación durante 5 épocas consecutivas, motivo por el cual las curvas pueden aparentar no alcanzar un mínimo global absoluto.

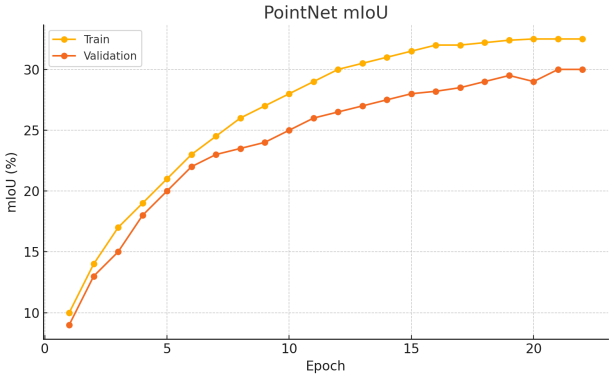
La Figura 5.4 muestra por separado la evolución de la función de pérdida, la precisión y el mIoU durante el entrenamiento de PointNet. Puede observarse que la función de pérdida disminuye de forma progresiva llegando a su mínimo en la época 22. La precisión en validación se estabiliza en torno al 45 %, mientras que el mIoU alcanza aproximadamente el 30 %, valores deficientes para una interpretación correcta y útil del entorno. La diferencia mantenida entre las curvas de entrenamiento y validación indica una ligera tendencia al sobreajuste, esperable dada la baja capacidad de PointNet para modelar relaciones espaciales locales.



(a) Evolución de la función de pérdida.



(b) Evolución de la tasa de acierto (*accuracy*).

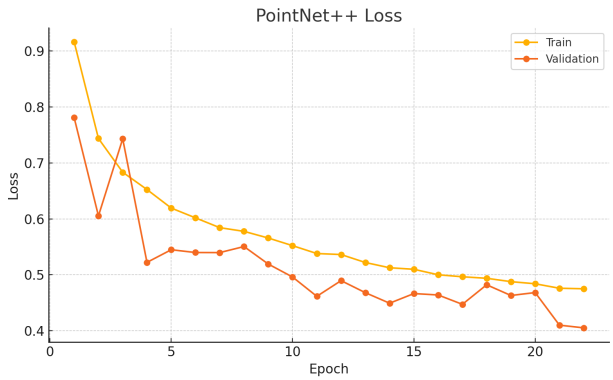


(c) Evolución del mIoU.

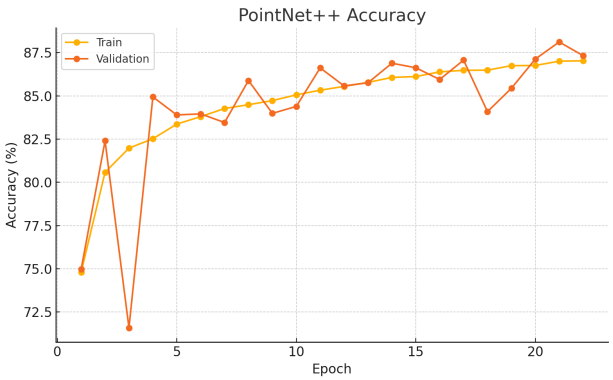
Figura 5.4: Evolución de los indicadores durante el entrenamiento de PointNet: (a) función de pérdida, (b) tasa de acierto (*accuracy*) y (c) mIoU en los conjuntos de entrenamiento y validación.

La Figura 5.5 corresponde al entrenamiento de PointNet++. En este caso, la pérdida desciende de forma más pronunciada y alcanza valores mínimos inferiores a los observados en PointNet. La precisión en validación supera el 85 % y el mIoU se aproxima al 50 %, evidenciando una mejora significativa en la capacidad del modelo para segmentar correctamente las clases. La menor diferencia entre las curvas de entrenamiento y validación sugiere una generalización más robusta, atribuible a la arquitectura jerárquica de agrupación local característica de PointNet++. Como en el caso anterior, el entrenamiento se interrumpió automáticamente al no registrarse mejora durante 5 épocas consecuti-

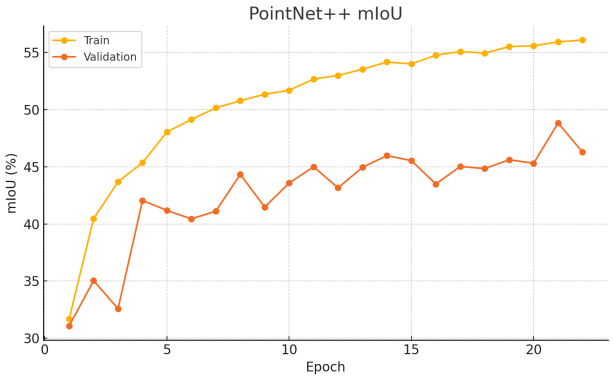
vas.



(a) Evolución de la función de pérdida.



(b) Evolución de la tasa de acierto (*accuracy*).



(c) Evolución del mIoU.

Figura 5.5: Evolución de los indicadores durante el entrenamiento de PointNet++: (a) función de pérdida, (b) tasa de acierto (*accuracy*) y (c) mIoU en los conjuntos de entrenamiento y validación.

5.4 PointNet++*: Modificación Propuesta

La arquitectura estándar de PointNet++ opera principalmente sobre las coordenadas espaciales (x, y, z) de la nube de puntos. No obstante, los sensores LiDAR proporcionan información adicional potencialmente de gran interés, como la intensidad de remisión. Este parámetro, que cuantifica la fuerza de retorno del pulso láser, puede aportar indicios relevantes sobre las propiedades del material de una superficie, facilitando la diferenciación entre objetos que presentan geometrías similares pero están compuestos por materiales distintos.

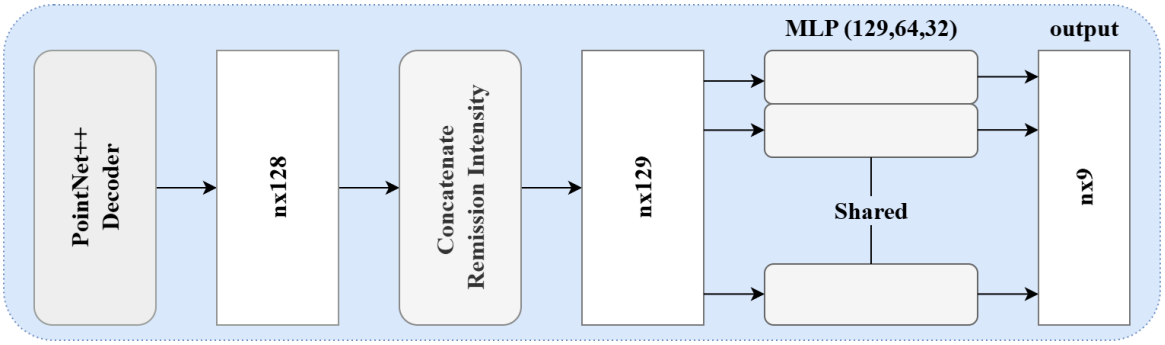


Figura 5.6: Modificación propuesta sobre el *backbone* de PointNet++.

Para aprovechar esta información, se ha propuesto una modificación sencilla pero eficaz del *backbone* de PointNet++, que se denomina PointNet++*. Tal como se ilustra en la Figura 5.6, esta modificación se aplica en la etapa final de la red, una vez que el decodificador ha propagado las características jerárquicas aprendidas de nuevo hasta los puntos originales.

Es importante destacar que los valores de las características generadas por el decodificador y los valores brutos de intensidad presentan escalas muy diferentes. La Figura 5.7 muestra la distribución de los valores por canal generados por el decodificador antes de la fusión con la intensidad de remisión. Se observa que la mayoría de los canales presentan valores comprendidos aproximadamente entre 0 y 3, con una variabilidad moderada entre dimensiones y sin presencia de valores extremos. Este rango contrasta con el de la intensidad de remisión original, que puede alcanzar valores muy superiores. Por este motivo, la normalización de la característica de intensidad mediante Z-score resulta imprescindible para evitar que su magnitud domine el espacio de características cuando se concatena con las salidas del decodificador. De no aplicarse esta normalización, la intensidad actuaría como un factor desproporcionado en el aprendizaje, reduciendo la capacidad de la red para explotar la información geométrica y comprometiendo la estabilidad del entrenamiento. La media y la desviación estándar requeridas para esta operación se calcularon exclusivamente sobre el conjunto de entrenamiento de GOOSE, con el fin de mantener la coherencia experimental.

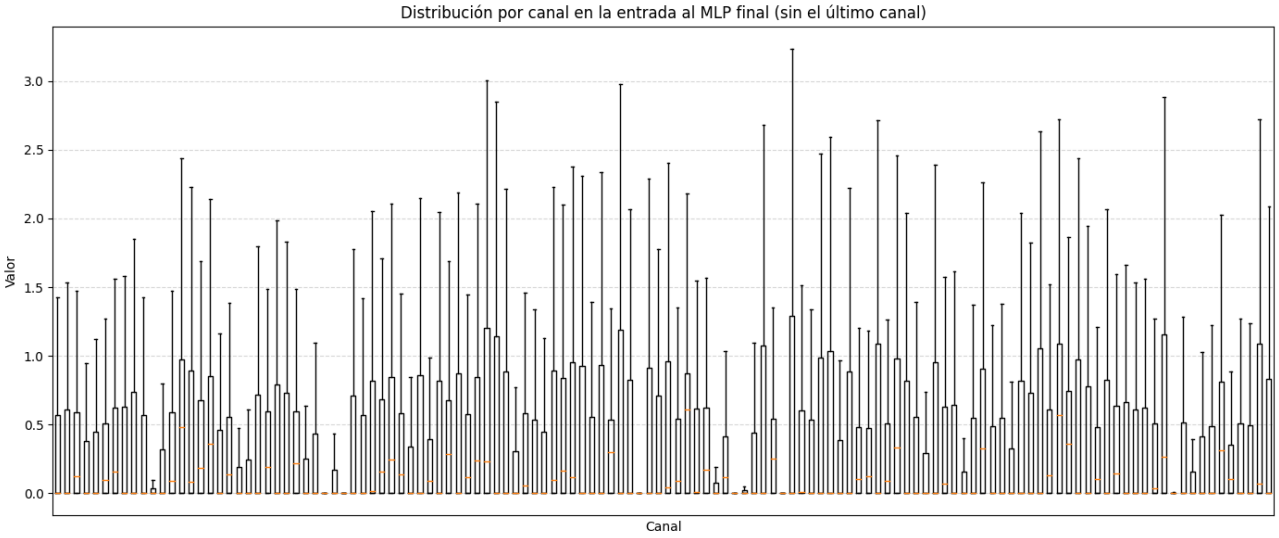
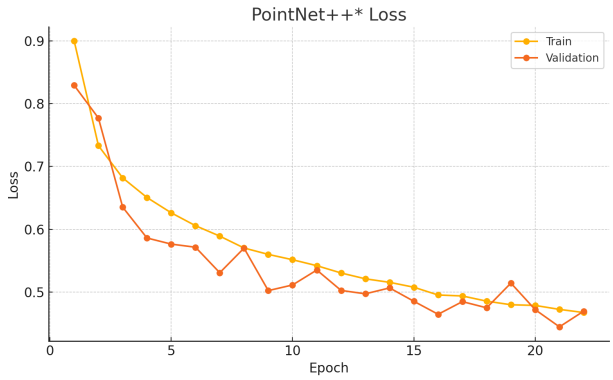


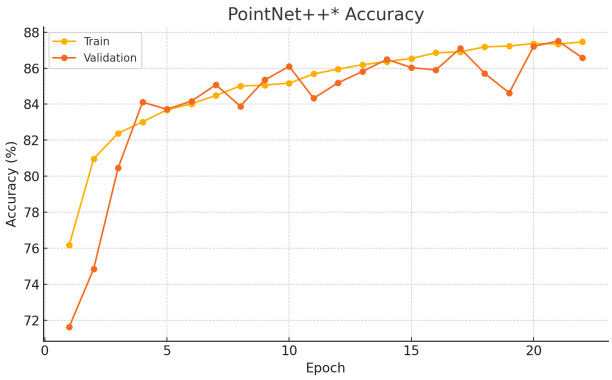
Figura 5.7: Distribución de los valores generados por cada canal del decodificador jerárquico de PointNet++ antes de la concatenación de la intensidad de remisión. Cada canal corresponde a una dimensión del tensor de características propagadas asociado a cada punto.

Una vez normalizada, la intensidad de remisión de cada punto se concatena con su tensor de características correspondiente, ampliando así el espacio de representación hasta obtener un tensor de tamaño $N \times 129$. Este tensor combinado, que incorpora tanto información geométrica como material en una escala comparable, se procesa a través de un MLP compartido final para realizar la clasificación punto a punto. Este enfoque pretende que la red aprenda primero jerarquías espaciales complejas y, posteriormente, utilice la información de intensidad como un decisor adicional capaz de afinar las predicciones.

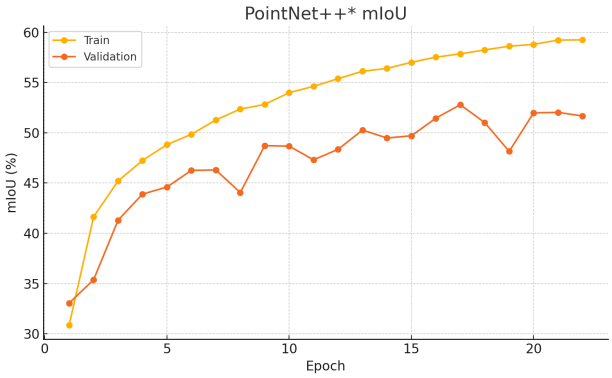
En la Figura 5.8 se muestra por separado la evolución de la función de pérdida, la tasa de acierto global (accuracy) y el mIoU durante el entrenamiento de la arquitectura PointNet++*. Se observa que la función de pérdida disminuye de forma continua y estable a lo largo de las épocas, alcanzando valores finales inferiores a 0.5. La tasa de acierto global en validación experimenta un aumento rápido durante las primeras épocas, superando el 85 % y manteniéndose posteriormente en un rango muy estable sobre el conjunto de entrenamiento. En cuanto al mIoU, se aprecia una mejora progresiva, que alcanza aproximadamente un 52 % en validación al final del proceso. El rendimiento obtenido en este experimento, comparado con el de PointNet++ (véase la Figura 5.5), muestra una mejora apreciable en la capacidad de generalización, tanto en términos de tasa de acierto global como de mIoU. Esta mejora se refleja en una menor diferencia entre las métricas alcanzadas durante el entrenamiento y las observadas en validación, lo que indica que la incorporación de la intensidad de remisión contribuye a reducir el sobreajuste presente en la versión original del modelo.



(a) Evolución de la función de pérdida.



(b) Evolución de la tasa de acierto (*accuracy*).



(c) Evolución del mIoU.

Figura 5.8: Evolución de los indicadores durante el entrenamiento de PointNet++* con la incorporación de la intensidad de remisión normalizada: (a) función de pérdida, (b) tasa de acierto (*accuracy*) y (c) mIoU en los conjuntos de entrenamiento y validación.

5.5 Resultados

En esta sección se presenta un análisis cuantitativo y cualitativo detallado del rendimiento de los modelos propuestos. En los Cuadros 5.1 y 5.2 se presentan los resultados cuantitativos en términos de IoU y Recall, respectivamente. Para complementar estas figuras de mérito, se incluyen las matrices de confusión, ilustradas en las Figuras 5.10, 5.11 y 5.12, que permiten analizar fortalezas y limitaciones en la predicción por clase.

Cuadro 5.1: Valores de IoU % por clase y modelo, evaluados sobre nubes de puntos no utilizadas en entrenamiento. La media de IoU (mIoU %) se muestra en la última columna.

Modelo	**Construction**	**Object**	**Road**	**Sign**	**Terrain**	**DVeg.**	**NDVeg.**	**Vehicle**	**Void**	**mIoU**
PointNet	23.62	11.08	45.62	28.39	31.74	33.46	28.37	10.64	79.60	**32.50**
PointNet++	69.90	23.16	27.66	40.15	59.01	66.69	86.45	72.63	82.78	**58.71**
PointNet++*	68.32	25.24	31.30	42.77	59.26	70.15	88.91	74.67	82.79	**60.38**

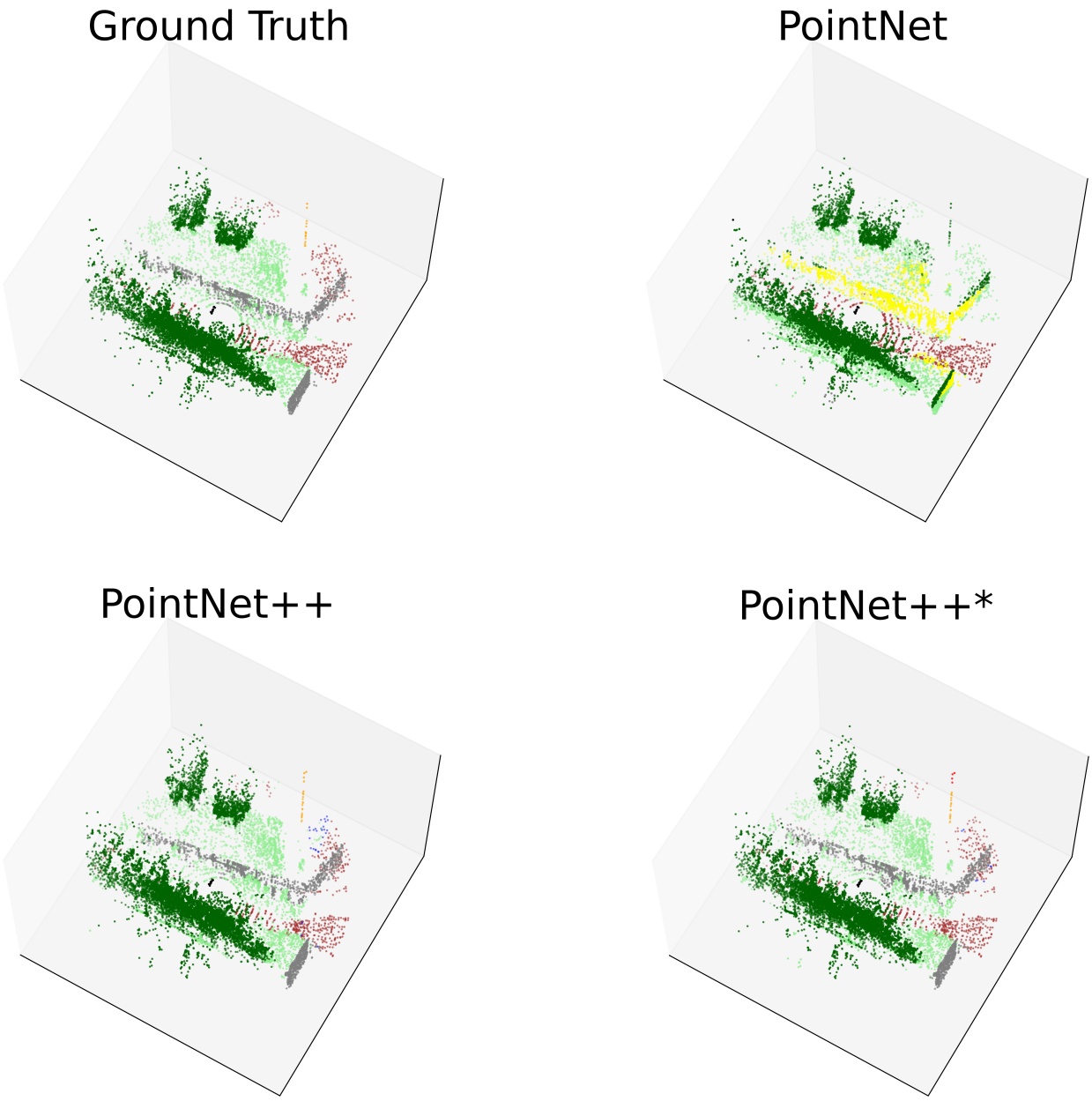


Figura 5.9: Comparativa visual de segmentación de una nube de puntos de GOOSE no empleada durante el entrenamiento: *ground truth* y resultados con diferentes modelos.

Cuadro 5.2: Valores de Recall % por clase y modelo, evaluados sobre nubes de puntos no utilizadas en entrenamiento. La media de Recall (mRecall %) se muestra en la última columna.

Modelo	**Construction**	**Object**	**Road**	**Sign**	**Terrain**	**DVeg.**	**NDVeg.**	**Vehicle**	**Void**	**mRecall**
PointNet	47.36	18.25	70.39	43.18	45.78	51.63	47.46	23.03	87.79	**48.32**
PointNet++	78.83	40.14	34.05	60.65	73.73	80.63	93.87	78.13	98.07	**70.90**
PointNet++*	86.02	32.10	59.88	59.39	68.95	87.06	93.35	84.91	83.74	**72.82**

- **PointNet**: Los resultados indican dificultades para diferenciar clases complejas, llegando solo a un 32.5 % de mIoU. Esta limitación se aprecia visualmente en la Figura 5.9, donde la segmentación que realiza PointNet contiene errores significativos, como la clasificación incorrecta de grandes regiones de construcción como vehículo. La matriz de confusión correspondiente, contenida en la Figura 5.10, refleja estos problemas, mostrando una dispersión considerable que evidencia la dificultad del modelo para capturar características locales distintivas necesarias para discriminar objetos geométricamente similares.

- **PointNet++**: Con el objetivo de superar las limitaciones de PointNet, se implementó el modelo jerárquico PointNet++. Este enfoque logra una mejora considerable, elevando la mIoU al 58.71 %. Los resultados cualitativos muestran una interpretación mucho más coherente y precisa del entorno, con una correcta identificación de zonas de terreno, construcciones y vegetación. La matriz de confusión, contenida en la Figura 5.11, presenta valores notablemente más altos en las celdas correspondientes a clasificaciones correctas, lo que confirma que el aprendizaje jerárquico de características facilita la captura de patrones geométricos locales. No obstante, persisten ciertas confusiones; por ejemplo, la clase `Object` se clasifica erróneamente como `Non Drivable Vegetation` en un 24.33 % de los casos.

- **PointNet++***: La variante propuesta PointNet++*, que incorpora la información de intensidad de remisión, permite afinar todavía más las predicciones y alcanza la mIoU más alta, con un 60.38 %. A nivel visual, la segmentación es similar a la obtenida con PointNet++, pero se aprecian mejoras en detalles al predecir mejor los objetos de alta reflectividad y algunos detalles en el terreno, como se ve en la Figura 5.9 en la parte superior derecha de la escena. En la correspondiente matriz de confusión que se muestra en la Figura 5.12, se observa un incremento notable del *Recall* en algunas clases, como la clase `Vehicle` alcanzando un 84.91 % o *Drivable Vegetation* con un 87.06 %. Estos resultados indican que la intensidad de remisión aporta características discriminativas adicionales que ayudan al modelo a resolver ambigüedades entre clases con perfiles geométricos semejantes.

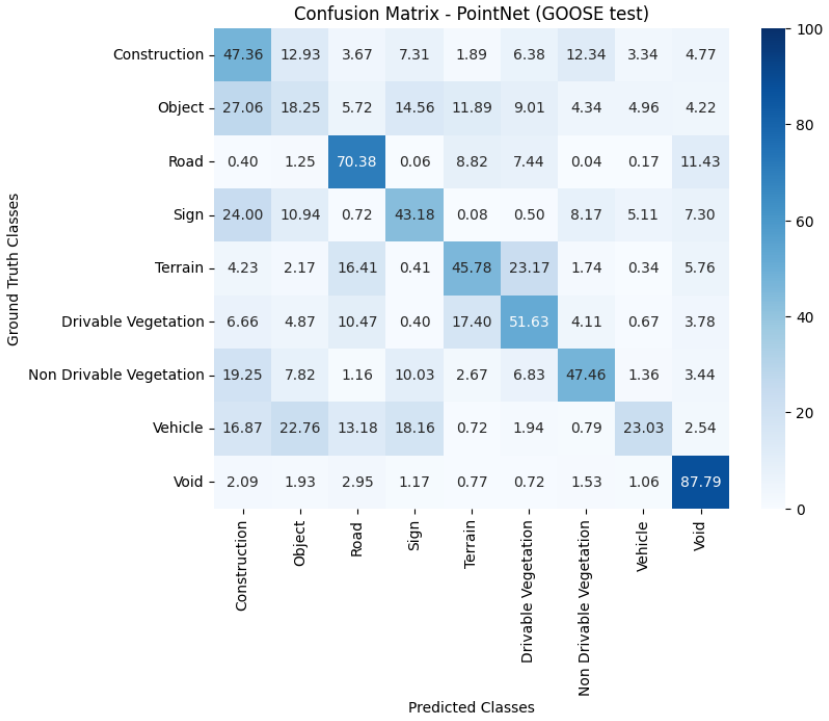


Figura 5.10: Matriz de confusión de PointNet obtenida del conjunto de *test* de GOOSE.

5.6 Evaluación Cruzada

Con el objetivo de evaluar la capacidad de generalización de los modelos, se realizó un experimento de evaluación cruzada empleando el conjunto de *test* procedente del *dataset* RELLIS-3D. Los modelos fueron previamente entrenados exclusivamente sobre muestras del *dataset* GOOSE, sin incluir datos de RELLIS-3D en ninguna etapa de entrenamiento o validación. El propósito de esta evaluación era determinar si el modelo es capaz de transferir el aprendizaje adquirido en un entorno diferente y mantener un rendimiento adecuado sobre escenarios con características similares, pero también con diferencias significativas en la distribución de las clases, la estructura de las nubes de puntos, las condiciones de captura y las configuraciones de los sensores empleados.

En las Figuras 5.13 y 5.14 se presentan ejemplos de la segmentación predicha por las dos arquitecturas sobre la misma muestra de RELLIS-3D. Ambos modelos segmentan adecuadamente las zonas correspondientes a vegetación transitable y no transitable, así como el terreno, aunque se observan ciertas inconsistencias en las regiones correspondientes a objetos verticales y vehículos, atribuibles principalmente a diferencias en la taxonomía de etiquetas entre GOOSE y RELLIS-3D, así como a la ausencia de datos de estos dominios en el proceso de entrenamiento.

Este experimento pone de manifiesto que, si bien existe una capacidad de generalización parcial hacia otro *dataset* capturado en entornos rurales con sensores LiDAR, es necesario contemplar técni-

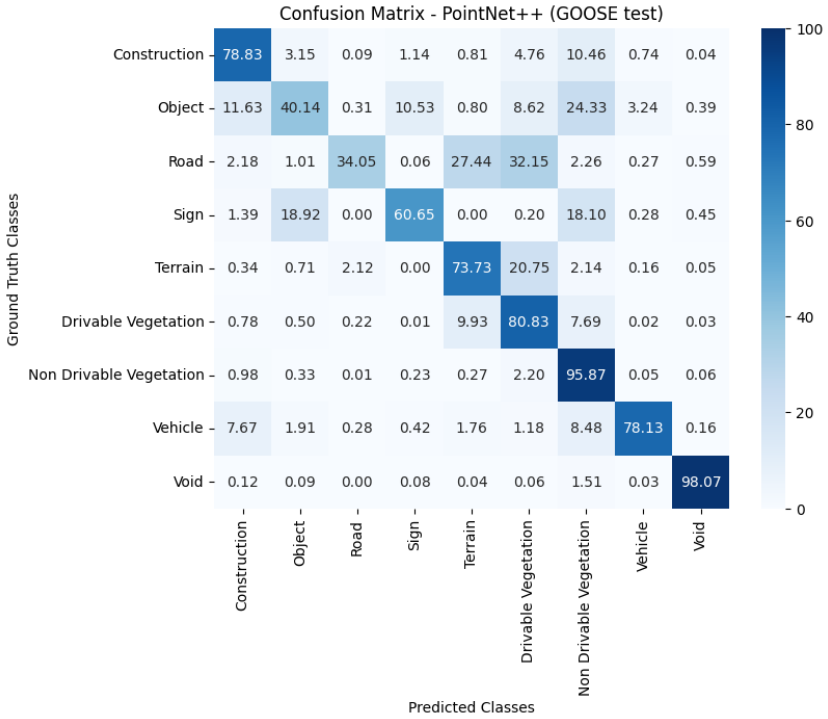


Figura 5.11: Matriz de confusión de PointNet++ obtenida del conjunto de *test* de GOOSE.

cas adicionales o entrenamiento cruzado para mejorar la robustez de los modelos frente a variaciones en la distribución de las clases y las características geométricas de los escenarios.

Cabe destacar que no se ha podido llevar a cabo una evaluación cuantitativa de los resultados obtenidos en este experimento debido a que existe un error en el etiquetado del conjunto de datos RELLIS-3D. Concretamente, una parte significativa de los puntos aparecen etiquetados como clase 0 (*Void*). Esta asignación incorrecta impide disponer de referencias fiables para comparar de manera objetiva las predicciones del modelo con las etiquetas de referencia. Por este motivo, la evaluación se ha limitado únicamente a un análisis cualitativo mediante la visualización de las nubes de puntos segmentadas.

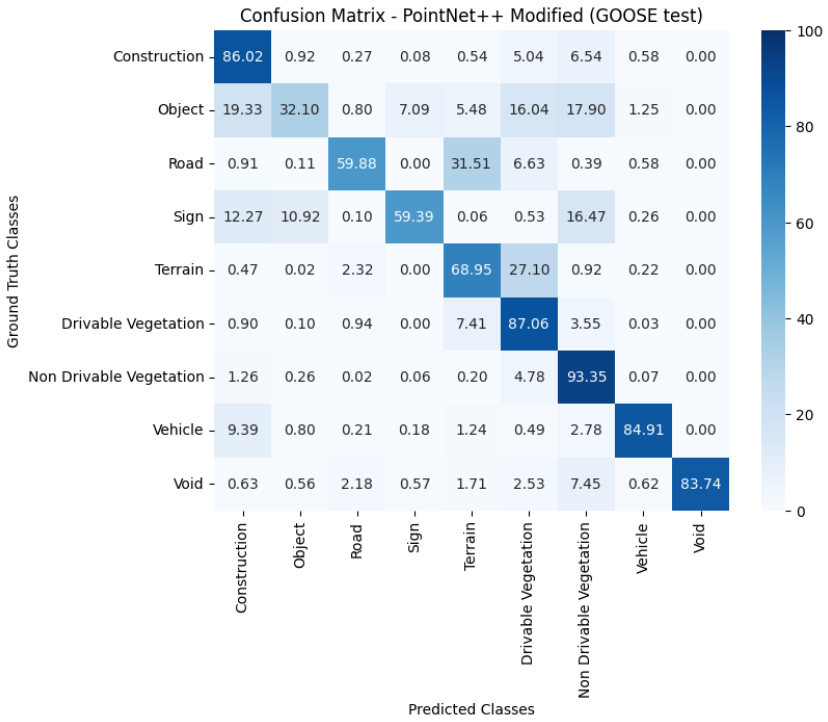


Figura 5.12: Matriz de confusión de PointNet++* obtenida del conjunto de *test* de GOOSE.

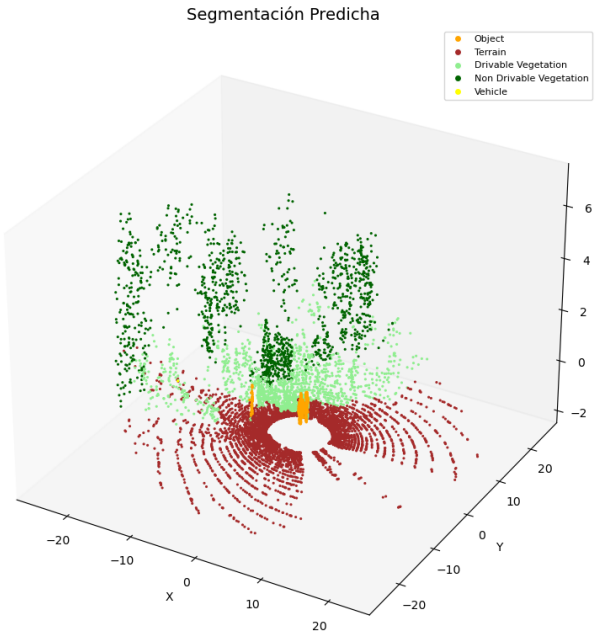


Figura 5.13: Segmentación predicha sobre una nube de puntos de RELLIS-3D utilizando PointNet++ entrenado con GOOSE.

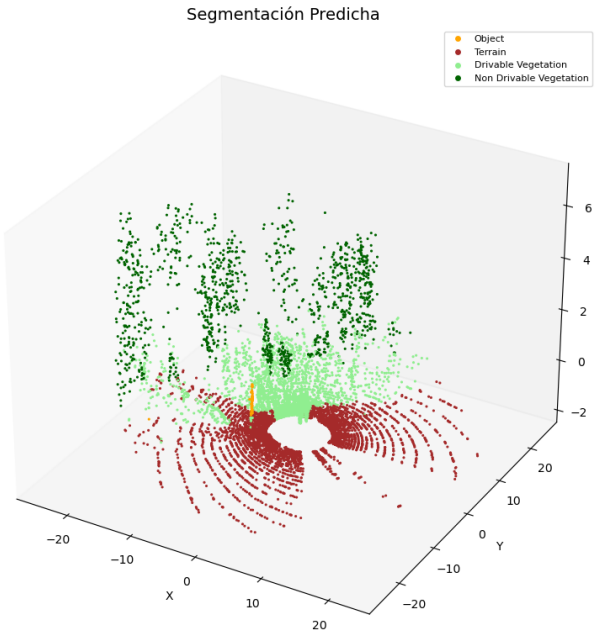


Figura 5.14: Segmentación predicha sobre una nube de puntos de RELLIS-3D utilizando PointNet++* entrenado con GOOSE.

Capítulo 6

Conclusiones y Trabajo Futuro

Este proyecto ha abordado la aplicación de arquitecturas de aprendizaje profundo para la segmentación semántica de nubes de puntos LiDAR en entornos no estructurados. Los experimentos realizados han permitido validar los modelos empleados, proponer mejoras y consolidar un conjunto amplio de competencias técnicas y prácticas. A continuación, se resumen los principales resultados, las competencias y habilidades adquiridas, y las líneas de trabajo futuro.

6.1 Principales Conclusiones

Los resultados obtenidos permiten valorar de forma positiva el grado de cumplimiento de los objetivos planteados al inicio del trabajo:

- En relación con el **Objetivo O1**, se desarrolló una herramienta de visualización interactiva, *LiDAR-Visualizer*, que facilita la exploración de secuencias LiDAR tanto a partir de archivos de las bases de datos GOOSE y RELLIS-3D como de entornos simulados generados con CARLA, permitiendo un análisis detallado en tiempo real de los escenarios y de las características de las nubes de puntos.

- Respecto al **Objetivo O2**, los experimentos confirman que las arquitecturas jerárquicas superan de manera clara a los enfoques globales en la segmentación semántica de escenas complejas. Aunque PointNet fue una propuesta pionera, mostró limitaciones importantes a la hora de capturar las estructuras locales detalladas necesarias para una segmentación precisa en entornos no estructurados, como reflejan sus valores más bajos de mIoU (32.50 %) y mRecall (48.32 %). La arquitectura PointNet++ mejoró de forma significativa el rendimiento al incorporar la captura de características geométricas multiescala, alcanzando un mIoU del 58.71 % y un mRecall del 70.90 %, lo que cumple el objetivo de analizar comparativamente sus capacidades y limitaciones.

- En relación con el **Objetivo O3**, se exploraron modificaciones estructurales mediante la incorporación de la información de remission en la variante propuesta, PointNet++*. Esta estrategia contribuyó a reducir la confusión entre clases y a incrementar las métricas de evaluación hasta

un mIoU del 60.38 % y un mRecall del 72.82 %, demostrando que la fusión de información adicional puede mejorar la precisión general de las predicciones.

- Por último, en cuanto al **Objetivo O4**, la calidad de los resultados obtenidos y la estabilidad que muestran ante variaciones moderadas en las condiciones de entrada permiten afirmar que estas arquitecturas presentan una viabilidad práctica considerable en aplicaciones de percepción automática en entornos exteriores no estructurados, como los contemplados en el proyecto GAIA. No obstante, persisten retos relacionados con la generalización a escenarios con alta variabilidad y la mejora de la integración multimodal.

En conjunto, este trabajo ha permitido demostrar que los modelos jerárquicos como PointNet++ resultan adecuados para la segmentación semántica en exteriores, y que la integración de características complementarias, como la intensidad de remisión, constituye una vía prometedora de optimización. Estas conclusiones sientan las bases para futuras investigaciones centradas en arquitecturas más avanzadas y estrategias de fusión de información más sofisticadas.

6.2 Competencias y Habilidades Adquiridas

Durante el desarrollo del Trabajo de Fin de Grado se han puesto en práctica diversas competencias recogidas en el plan de estudios del Grado en Ingeniería en Sistemas Audiovisuales y Multimedia. Aunque en su redacción original muchas de ellas se enuncian en el contexto de la ingeniería de telecomunicación, su aplicación se extiende igualmente a actividades de desarrollo, investigación e implementación de sistemas basados en aprendizaje automático. Dichas competencias pueden consultarse de forma detallada en la página oficial del grado[1]. A continuación, se enumeran de forma literal aquellas competencias directamente relacionadas con la realización del trabajo:

6.2.1 Competencias Generales

- **B.1**: Capacidad para la resolución de los problemas matemáticos que puedan plantearse en la ingeniería. Aptitud para aplicar los conocimientos sobre: álgebra lineal; geometría; geometría diferencial; cálculo diferencial e integral; ecuaciones diferenciales y en derivadas parciales; métodos numéricos; algorítmica numérica; estadística y optimización.
- **B.2**: Conocimientos básicos sobre el uso y programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación en ingeniería.

[1] https://www.urjc.es/universidad/calidad/637-ingenieria-en-sistemas-audiovisuales-y-multimedia#competencias

6.2.2 Competencias Específicas

- **C.1**: Capacidad para aprender de manera autónoma nuevos conocimientos y técnicas adecuados para la concepción, el desarrollo o la explotación de sistemas y servicios de telecomunicación.
- **C.2**: Capacidad de utilizar aplicaciones de comunicación e informáticas (ofimáticas, bases de datos, cálculo avanzado, gestión de proyectos, visualización, etc.) para apoyar el desarrollo y explotación de redes, servicios y aplicaciones de telecomunicación y electrónica.
- **C.3**: Capacidad para utilizar herramientas informáticas de búsqueda de recursos bibliográficos o de información relacionada con las telecomunicaciones y la electrónica.
- **C.7**: Conocimiento y utilización de los fundamentos de la programación en redes, sistemas y servicios de telecomunicación.
- **FC.1**: Capacidad de comunicarse en forma efectiva en el idioma de uso profesional pertinente.

6.2.3 Habilidades Complementarias Adquiridas

Además, el desarrollo de este trabajo ha permitido adquirir habilidades complementarias que son de utilidad tanto para la práctica profesional en el ámbito de la inteligencia artificial como para la realización de futuras actividades de investigación aplicada.

- Profundización en aprendizaje profundo, su funcionamiento interno y las variantes más utilizadas para procesamiento de datos 3D.
- Capacidad para planificar, desarrollar y documentar proyectos completos de *deep learning*, desde la preparación de datos hasta la validación sistemática de resultados.
- Dominio avanzado de programación en Python, incluyendo el diseño de *pipelines* modulares, la gestión de dependencias y la ejecución de experimentos.
- Conocimiento profundo de arquitecturas específicas como PointNet y PointNet++, así como de librerías como TensorFlow y especialmente PyTorch, lo que resulta esencial para la investigación aplicada en inteligencia artificial.
- Capacidad crítica para diseñar estudios comparativos, valorar resultados de manera rigurosa y presentar conclusiones basadas en evidencia experimental.
- Mayor autonomía en la integración de múltiples fuentes de datos, el diseño de visualizaciones personalizadas y la optimización de recursos computacionales.

- Desarrollo de competencias transversales como la gestión del tiempo, la toma de decisiones técnicas fundamentadas y la adaptación a nuevas herramientas y entornos de desarrollo.

6.3 Líneas de Trabajo Futuro

A partir de este trabajo, se identifican varias líneas de investigación y desarrollo que pueden servir de base para continuar y profundizar en el estudio realizado. En primer lugar, se propone profundizar en la modificación de la arquitectura con el objetivo de aprovechar de manera más eficaz la información de la intensidad de remisión, explorando estrategias de fusión avanzadas que permitan mejorar la discriminación entre clases.

Asimismo, resulta de interés investigar modelos de última generación, como las arquitecturas basadas en *transformers*, que actualmente están mostrando un alto rendimiento en tareas de segmentación 3D y podrían aportar capacidades adicionales de contextualización espacial.

Por último, se plantea experimentar con bases de datos sintéticas con el fin de ampliar la diversidad de los escenarios de entrenamiento y evaluar la capacidad de generalización de los modelos en entornos con variabilidad controlada.

Anexo

A.1 Fundamentos del Aprendizaje Automático

El campo del aprendizaje automático (Machine Learning, ML) ha experimentado una transformación fundamental en las últimas décadas, consolidándose como una disciplina central para el avance tecnológico en una vasta gama de aplicaciones. Dentro de este paradigma, las Redes Neuronales Artificiales (RNAs) han emergido como la arquitectura más influyente y versátil, impulsando la mayoría de los sistemas de inteligencia artificial contemporáneos. Esta sección se dedicará a establecer el contexto del estado del arte en los principios fundamentales que sustentan el ML, con un énfasis particular en la teoría, la evolución y el funcionamiento intrínseco de las RNAs. Se explorarán sus componentes esenciales y los mecanismos de aprendizaje que les permiten adaptarse a los datos.

A.1.1 Redes Neuronales Artificiales

Las RNAs son modelos computacionales cuya concepción se inspira, de forma simplificada, en la estructura y el funcionamiento del cerebro biológico [48]. Su capacidad inherente para reconocer patrones complejos y aproximar funciones no lineales ha sido fundamental para su éxito en diversos dominios. El punto de partida histórico para las RNAs se encuentra en el concepto del perceptrón simple, introducido por Frank Rosenblatt [49]. Este modelo opera calculando una suma ponderada de sus entradas, a la que luego aplica una función para decidir su salida. La salida y de un perceptrón puede expresarse como:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

donde x_i son las entradas, w_i son los pesos correspondientes, b es el sesgo (bias) y f es la función de activación.

Este modelo funcionaba como un clasificador binario capaz de aprender a separar datos linealmente separables. Sin embargo, sus limitaciones se hicieron evidentes al no poder resolver problemas fundamentalmente no lineales, como el famoso problema XOR, tal como demostraron Minsky y Papert [50]. La superación de esta barrera llegó con el desarrollo de los perceptrones multicapa (MLPs), que incorporaron una o más capas ocultas entre la capa de entrada y la de salida. La inclusión de estas capas intermedias, combinada con el uso de funciones de activación no lineales, dotó a las MLPs de la capacidad universal de aproximar cualquier función continua, marcando un hito crucial en el desarrollo de las RNAs.

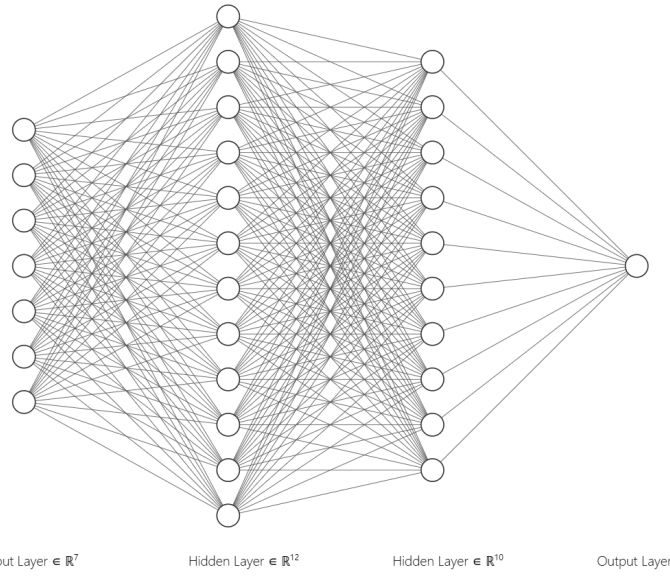


Figura A.1: Ejemplo ilustrativo de una red neuronal artificial con dos capas ocultas.

La arquitectura de una RNA se construye a partir de unidades básicas interconectadas, denominadas neuronas o nodos. Cada neurona recibe una o varias entradas, a las que aplica pesos y un sesgo, para luego generar una salida que es transformada por una función de activación. Estas neuronas se organizan en capas distintivas. La capa de entrada es responsable de recibir y representar la información inicial de los datos. Posteriormente, una o más capas ocultas procesan esta información de forma jerárquica y abstracta, extrayendo características de complejidad creciente. Es el número de estas capas ocultas lo que define la profundidad de la red. Finalmente, la capa de salida produce el resultado final de la red, cuya interpretación varía según la tarea, ya sea una clasificación, una regresión o alguna otra salida estructurada. Durante el proceso de aprendizaje, la red ajusta continuamente los pesos y sesgos, que son los parámetros fundamentales que determinan las relaciones aprendidas entre las entradas y las salidas.

A.1.2 Funciones de Activación

Las funciones de activación son elementos críticos que se aplican a la salida de cada neurona antes de pasarla a la siguiente capa. Su propósito fundamental es introducir una no linealidad en la red, lo cual es indispensable para que las RNAs puedan aprender patrones complejos y no meras combinaciones lineales de sus entradas. Sin estas funciones, una red con múltiples capas se comportaría como un simple modelo de regresión lineal. A continuación se muestran algunas de las funciones más utilizadas:

- **Sigmoide**: La función sigmoide comprime cualquier valor de entrada entre 0 y 1. Se define

como:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{1}$$

- **Tangente Hiperbólica (Tanh)**: Similar a la sigmoide, pero con un rango de salida entre -1 y 1. Se define como:

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2}$$

- **Rectified Linear Unit (ReLU)**: Esta función marcó un avance significativo al mitigar el problema del gradiente desvanecido, convirtiéndose en la función de activación preferida en la mayoría de las arquitecturas modernas por su eficiencia computacional [51]. Se expresa como:

$$f(x) = \text{máx}(0,x) \tag{3}$$

  Existen variantes de ReLU, como Leaky ReLU o ELU, que buscan resolver alguna limitación asociada (dying ReLU problem).

- **Softmax**: Para la capa de salida en problemas de clasificación donde hay múltiples categorías, la función Softmax es comúnmente empleada. Esta función convierte las salidas numéricas de la red en probabilidades que suman uno, facilitando la interpretación de la clasificación. Para un vector de entradas $z = [z_1, z_2, \ldots, z_K]$, la función Softmax se define como:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{4}$$

A.1.3 Funciones de Pérdida

Las funciones de pérdida, también conocidas como funciones de coste o error, son fundamentales para cuantificar qué tan bien (o mal) está rindiendo la red [52]. Miden la discrepancia entre las predicciones de la red y los valores reales (*ground truth*) de los datos de entrenamiento. El objetivo principal durante el entrenamiento de una RNA es minimizar el valor de esta función de pérdida.

- Error Cuadrático Medio (MSE): Utilizado predominantemente en tareas de regresión (donde se predice un valor numérico continuo) para medir el promedio de los cuadrados de las diferencias entre los valores predichos y los reales. Se define como:

$$L_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{5}$$

  donde N es el número de muestras, y_i es el valor real y $\hat{y}_i$ es la predicción del modelo.

- Entropía Cruzada (Cross-Entropy): Es la función de pérdida preferida para tareas de clasificación (donde se predice una categoría). Para una clasificación binaria (dos categorías), con y_i siendo la etiqueta real (0 o 1) e $\hat{y}_i$ la probabilidad predicha para la clase 1:

$$L_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{6}$$

  Para clasificación multiclase (más de dos categorías), la entropía cruzada es:

$$L_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) \tag{7}$$

  donde y_{ic} es 1 si la muestra i pertenece a la clase c, e $\hat{y}_{ic}$ es la probabilidad predicha para la clase c.

A.1.4 Mecanismos de Aprendizaje Automático

El proceso fundamental por el cual una RNA aprende y mejora su rendimiento es ajustando repetidamente sus pesos y sesgos, un procedimiento guiado por la minimización de la función de pérdida. El método principal para este ajuste es el Descenso de Gradiente y sus diversas variantes. La idea es sencilla: la red modifica sus parámetros (pesos w y sesgos b) en la dirección que reduce más rápidamente el valor de la función de pérdida. La regla general de actualización para un parámetro θ (ya sea un peso o un sesgo) es:

$$\theta_{\text{nueva}} = \theta_{\text{antigua}} - \eta\frac{\partial L}{\partial \theta} \tag{8}$$

donde η es la tasa de aprendizaje, que controla el tamaño del paso en cada ajuste. El Descenso de Gradiente Estocástico (SGD) es una de las implementaciones más comunes, donde los pesos se actualizan usando solo un pequeño subconjunto de datos (*mini-batch*) en lugar de todo el conjunto, lo que agiliza el proceso. Variantes más avanzadas, como Adam [43], ajustan la tasa de aprendizaje de forma dinámica para cada parámetro de la red, mejorando la velocidad y robustez del entrenamiento.

El algoritmo clave que permite calcular eficientemente cómo cada peso y sesgo contribuye al error total de la red es la Retropropagación (*Backpropagation*) [53]. Este proceso implica dos fases. Primero, los datos se propagan hacia delante a través de la red para obtener una predicción y calcular la pérdida. Luego, el error se propaga hacia atrás, desde la salida hacia las capas iniciales calculando los gradientes necesarios para actualizar todos los pesos y sesgos de la red.

Un desafío importante en el entrenamiento de modelos de ML es asegurar que la red no solo aprenda bien los datos con los que se entrena, sino que también generalice correctamente a datos

nuevos, que no ha visto antes. Esto lleva a considerar los fenómenos de sobreajuste (*overfitting*) y subajuste (*underfitting*). El sobreajuste ocurre cuando el modelo se adapta demasiado a los datos de entrenamiento, memorizando incluso el ruido", lo que le impide rendir bien con datos diferentes. El subajuste, en cambio, significa que el modelo es demasiado simple y no ha aprendido lo suficiente de los datos. Para evitar el sobreajuste, se utilizan diversas técnicas de regularización. La regularización L1 y L2 (o Weight Decay) añade una penalización a la función de pérdida para desincentivar que los pesos de la red sean muy grandes, lo que ayuda a crear modelos más simples. La función de pérdida con regularización L2 (Ridge) es:

$$L_{\text{total}} = L_{\text{original}} + \lambda \sum w^2 \tag{9}$$

y con L1 (Lasso) es:

$$L_{\text{total}} = L_{\text{original}} + \lambda \sum |w| \tag{10}$$

donde λ es un parámetro que controla la fuerza de la regularización. Dropout [54] es una técnica muy efectiva que desactiva aleatoriamente un porcentaje de neuronas durante el entrenamiento, forzando a la red a no depender de ninguna neurona en particular y a aprender representaciones más robustas y distribuidas. Finalmente, la Normalización por Lotes (Batch Normalization) [55] es una técnica que estabiliza las entradas de las capas intermedias, lo que acelera el entrenamiento y mejora la capacidad de generalización de la red al reducir el çambio de covariante interno", que se refiere a cómo la distribución de las entradas de cada capa cambia a medida que los pesos de las capas anteriores se actualizan.

A.2 Implementación de la arquitectura PointNet

A.2.1 Módulo T-Net

```python
class TNet(nn.Module):
    def __init__(self, dim, num_points=16384):
        super(TNet, self).__init__()
        self.dim = dim
        self.conv1 = nn.Conv1d(dim, 64, kernel_size=1)
        self.conv2 = nn.Conv1d(64, 128, kernel_size=1)
        self.conv3 = nn.Conv1d(128, 1024, kernel_size=1)
        self.fc1 = nn.Linear(1024, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, dim ** 2)
        self.bn1 = nn.BatchNorm1d(64)
        self.bn2 = nn.BatchNorm1d(128)
        self.bn3 = nn.BatchNorm1d(1024)
        self.bn4 = nn.BatchNorm1d(512)
        self.bn5 = nn.BatchNorm1d(256)
        self.maxpool = nn.MaxPool1d(kernel_size=num_points)

    def forward(self, x):
        batch_size = x.shape[0]
        x = self.bn1(F.relu(self.conv1(x)))
        x = self.bn2(F.relu(self.conv2(x)))
        x = self.bn3(F.relu(self.conv3(x)))
        x = self.maxpool(x).view(batch_size, -1)
        x = self.bn4(F.relu(self.fc1(x)))
        x = self.bn5(F.relu(self.fc2(x)))
        x = self.fc3(x)
        identity = torch.eye(self.dim, requires_grad=True).repeat(batch_size
    , 1, 1)
        if x.is_cuda:
            identity = identity.cuda()
        x = x.view(-1, self.dim, self.dim) + identity
        return x

```

Figura A.2: Clase `TNet`: módulo de red neuronal utilizado para estimar matrices de transformación aprendidas que alinean dinámicamente las características de entrada e intermedias.

A.2.2 Cabecera de clasificación

```python
class PointNet(nn.Module):
    def __init__(self, num_points=16384, global_feat_dim=1024, use_local_features=True):
        super(PointNet, self).__init__()
        self.num_points = num_points
        self.global_feat_dim = global_feat_dim
        self.use_local_features = use_local_features
        self.input_tnet = TNet(dim=3, num_points=num_points)
        self.feature_tnet = TNet(dim=64, num_points=num_points)
        self.conv1 = nn.Conv1d(3, 64, kernel_size=1)
        self.conv2 = nn.Conv1d(64, 64, kernel_size=1)
        self.conv3 = nn.Conv1d(64, 64, kernel_size=1)
        self.conv4 = nn.Conv1d(64, 128, kernel_size=1)
        self.conv5 = nn.Conv1d(128, self.global_feat_dim, kernel_size=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.bn2 = nn.BatchNorm1d(64)
        self.bn3 = nn.BatchNorm1d(64)
        self.bn4 = nn.BatchNorm1d(128)
        self.bn5 = nn.BatchNorm1d(self.global_feat_dim)
        self.maxpool = nn.MaxPool1d(kernel_size=num_points, return_indices=True)

    def forward(self, x):
        batch_size = x.shape[0]
        transform_input = self.input_tnet(x)
        x = torch.bmm(x.transpose(2, 1), transform_input).transpose(2, 1)
        x = self.bn1(F.relu(self.conv1(x)))
        x = self.bn2(F.relu(self.conv2(x)))
        transform_feat = self.feature_tnet(x)
        x = torch.bmm(x.transpose(2, 1), transform_feat).transpose(2, 1)
        local_features = x.clone()
        x = self.bn3(F.relu(self.conv3(x)))
        x = self.bn4(F.relu(self.conv4(x)))
        x = self.bn5(F.relu(self.conv5(x)))
        global_features, critical_indices = self.maxpool(x)
        global_features = global_features.view(batch_size, -1)
        if self.use_local_features:
            combined_features = torch.cat(
                (local_features, global_features.unsqueeze(-1).repeat(1, 1, self.num_points)),
                dim=1
            )
            return combined_features, critical_indices, transform_feat
        else:
            return global_features, critical_indices, transform_feat

```

Figura A.3: Clase `PointNet`: arquitectura que extrae características globales y locales mediante transformaciones aprendidas y operaciones convolucionales.

A.2.3 Cabecera de segmentación

```python
class PointNetSeg(nn.Module):
    def __init__(self, num_points=4096, global_feat_dim=1024, num_classes=2)
    :
        super(PointNetSeg, self).__init__()
        self.num_points = num_points
        self.num_classes = num_classes
        self.backbone = PointNet(
            num_points=num_points,
            global_feat_dim=global_feat_dim,
            use_local_features=True
        )
        input_feat_dim = global_feat_dim + 64
        self.conv1 = nn.Conv1d(input_feat_dim, 512, kernel_size=1)
        self.conv2 = nn.Conv1d(512, 256, kernel_size=1)
        self.conv3 = nn.Conv1d(256, 128, kernel_size=1)
        self.conv4 = nn.Conv1d(128, num_classes, kernel_size=1)
        self.bn1 = nn.BatchNorm1d(512)
        self.bn2 = nn.BatchNorm1d(256)
        self.bn3 = nn.BatchNorm1d(128)

    def forward(self, x):
        x, critical_indices, transform_feat = self.backbone(x)
        x = self.bn1(F.relu(self.conv1(x)))
        x = self.bn2(F.relu(self.conv2(x)))
        x = self.bn3(F.relu(self.conv3(x)))
        x = self.conv4(x)
        x = x.transpose(2, 1)
        return x, critical_indices, transform_feat

```

Figura A.4: Clase `PointNetSeg`: cabeza de segmentación que proyecta las características combinadas a etiquetas por punto.

A.3 Implementación de la arquitectura PointNet++

A.3.1 Farthest Point Sampling

```python
def farthest_point_sample(xyz, npoint):
    B, N, C = xyz.shape
    centroids = torch.zeros(B, npoint, dtype=torch.long).to(xyz.device)
    distance = torch.ones(B, N).to(xyz.device) * 1e10
    farthest = torch.randint(0, N, (B,), dtype=torch.long).to(xyz.device)
    batch_indices = torch.arange(B, dtype=torch.long).to(xyz.device)
    for i in range(npoint):
        centroids[:, i] = farthest
        centroid = xyz[batch_indices, farthest, :].unsqueeze(1)
        dist = torch.sum((xyz - centroid) ** 2, -1)
        mask = dist < distance
        distance[mask] = dist[mask]
        farthest = torch.max(distance, -1)[1]
    return centroids
```

Figura A.5: Función `farthest_point_sample`: selección de centroides.

A.3.2 Cálculo de distancias

```python
def square_distance(src, dst):
    B, N, _ = src.shape
    _, M, _ = dst.shape
    dist = -2 * torch.matmul(src, dst.permute(0, 2, 1))
    dist += torch.sum(src ** 2, -1).view(B, N, 1)
    dist += torch.sum(dst ** 2, -1).view(B, 1, M)
    return dist
```

Figura A.6: Función `square_distance`: cálculo de distancias euclídeas al cuadrado.

A.3.3 Búsqueda de vecinos con Query Ball

```python
def query_ball_point(radius, nsample, xyz, new_xyz):
    B, N, _ = xyz.shape
    _, S, _ = new_xyz.shape
    group_idx = torch.arange(N, device=xyz.device).view(1,1,N).repeat(B,S,1)
    sqrdists = square_distance(new_xyz, xyz)
    group_idx[sqrdists > radius**2] = N
    group_idx = group_idx.sort(dim=-1)[0][:,:,:nsample]
    group_first = group_idx[:,:,0].view(B,S,1).repeat(1,1,nsample)
    group_idx[group_idx == N] = group_first[group_idx == N]
    return group_idx
```

Figura A.7: Función `query_ball_point`: búsqueda de vecinos cercanos por radio.

A.3.4 Indexación de puntos seleccionados

```python
def index_points(points, idx):
    B = points.shape[0]
    if idx.dim() == 2:
        return points[torch.arange(B).view(-1,1).to(points.device), idx]
    elif idx.dim() == 3:
        batch_indices = torch.arange(B, device=points.device).view(B,1,1)
        batch_indices = batch_indices.expand(-1, idx.shape[1], idx.shape[2])
        return points[batch_indices, idx]
    else:
        raise ValueError("idx debe tener 2 o 3 dimensiones")
```

Figura A.8: Función `index_points`: indexación de puntos en la nube.

A.3.5 Agrupamiento local de puntos

```
def sample_and_group(npoint, radius, nsample, xyz, points):
    B, N, C = xyz.shape
    fps_idx = farthest_point_sample(xyz, npoint)
    new_xyz = index_points(xyz, fps_idx)
    idx = query_ball_point(radius, nsample, xyz, new_xyz)
    grouped_xyz = index_points(xyz, idx)
    grouped_xyz_norm = grouped_xyz - new_xyz.unsqueeze(2)
    if points is not None:
        grouped_points = index_points(points, idx)
        new_points = torch.cat([grouped_xyz_norm, grouped_points], dim=-1)
    else:
        new_points = grouped_xyz_norm
    return new_xyz, new_points
```

Figura A.9: Función `sample_and_group`: construcción de agrupaciones locales.

A.3.6 Set Abstraction Layer and Feature Propagation Layer

```python
class PointNetSetAbstraction(nn.Module):
    def __init__(self, npoint, radius, nsample, in_channel, mlp, group_all):
        super(PointNetSetAbstraction, self).__init__()
        self.npoint = npoint
        self.radius = radius
        self.nsample = nsample
        self.group_all = group_all

        self.mlp_convs = nn.ModuleList()
        self.mlp_bns = nn.ModuleList()

        last_channel = in_channel
        for out_channel in mlp:
            self.mlp_convs.append(nn.Conv2d(last_channel, out_channel, 1))
            self.mlp_bns.append(nn.BatchNorm2d(out_channel))
            last_channel = out_channel

    def forward(self, xyz, points):
        B, C, N = xyz.shape
        xyz = xyz.permute(0, 2, 1)
        if points is not None:
            points = points.permute(0, 2, 1)

        if self.group_all:
            new_xyz, new_points = sample_and_group_all(xyz, points)
        else:
            new_xyz, new_points = sample_and_group(
                self.npoint, self.radius, self.nsample, xyz, points
            )
        new_points = new_points.permute(0, 3, 2, 1)

        for i, conv in enumerate(self.mlp_convs):
            bn = self.mlp_bns[i]
            new_points = F.relu(bn(conv(new_points)))

        new_points = torch.max(new_points, 2)[0]
        new_xyz = new_xyz.permute(0, 2, 1)
        return new_xyz, new_points
```

Figura A.10: Clase `PointNetSetAbstraction`: Bloque codificador de características jerárquicas.

```python
class PointNetFeaturePropagation(nn.Module):
    def __init__(self, in_channel, mlp):
        super(PointNetFeaturePropagation, self).__init__()
        self.mlp_convs = nn.ModuleList()
        self.mlp_bns = nn.ModuleList()
        last_channel = in_channel
        for out_channel in mlp:
            self.mlp_convs.append(nn.Conv1d(last_channel, out_channel, 1))
            self.mlp_bns.append(nn.BatchNorm1d(out_channel))
            last_channel = out_channel

    def forward(self, xyz1, xyz2, points1, points2):
        B, C, N = xyz1.shape
        _, _, S = xyz2.shape

        if S == 1:
            interpolated_points = points2.repeat(1, 1, N)
        else:
            dists = square_distance(
                xyz1.permute(0, 2, 1),
                xyz2.permute(0, 2, 1)
            )
            dists, idx = dists.sort(dim=-1)
            dists, idx = dists[:, :, :3], idx[:, :, :3]

            dist_recip = 1.0 / (dists + 1e-8)
            norm = torch.sum(dist_recip, dim=2, keepdim=True)
            weight = dist_recip / norm

            interpolated_points = torch.sum(
                index_points(points2.permute(0, 2, 1), idx) * weight.
    unsqueeze(-1),
                dim=2
            )
            interpolated_points = interpolated_points.permute(0, 2, 1)

        if points1 is not None:
            new_points = torch.cat([points1, interpolated_points], dim=1)
        else:
            new_points = interpolated_points

        for i, conv in enumerate(self.mlp_convs):
            bn = self.mlp_bns[i]
            new_points = F.relu(bn(conv(new_points)))

        return new_points
```

Figura A.11: Clase `PointNetFeaturePropagation`: Bloque decodificador de características jerárquicas.

A.3.7 Arquitectura Global de PointNet++

```
class PointNet2SemSeg(nn.Module):
    def __init__(self, num_classes=3, normal_channel=False):

        ...

    def forward(self, x, remission):
        """
        x: (B, 3, N) -> Coordenadas
        remission: (B, 1, N) -> Remission normalizada y alineada con cada punto
        """
        B, _, N = x.shape
        l0_xyz = x
        l0_points = None  # No usamos caracter sticas adicionales al inicio

        # Codificador
        l1_xyz, l1_points = self.sa1(l0_xyz, l0_points)
        l2_xyz, l2_points = self.sa2(l1_xyz, l1_points)
        l3_xyz, l3_points = self.sa3(l2_xyz, l2_points)

        # Decodificador
        l2_points = self.fp3(l2_xyz, l3_xyz, l2_points, l3_points)
        l1_points = self.fp2(l1_xyz, l2_xyz, l1_points, l2_points)
        l0_points = self.fp1(l0_xyz, l1_xyz, l0_points, l1_points)

        # MLP de clasificaci n punto a punto (segmentaci n sem ntica de la nube)
        x = F.relu(self.bn1(self.conv1(l0_points)))
        x = self.drop1(x)
        x = self.conv2(x)  # (B, num_classes, N)
        x = F.log_softmax(x, dim=1)

        return x
```

Figura A.12: Clase `PointNet2SemSeg`: Definición de la arquitectura de PointNet++.

A.4 Implementación de PointNet++*: Modificación propuesta

```python
class PointNet2SemSeg(nn.Module):
    def __init__(self, num_classes=3, normal_channel=False):

        ...

    def forward(self, x, remission):
        """
        x: (B, 3, N) -> Coordenadas
        remission: (B, 1, N) -> Remission normalizada y alineada con cada punto
        """
        B, _, N = x.shape
        l0_xyz = x
        l0_points = None  # No usamos caracter sticas adicionales al inicio

        # Codificador
        l1_xyz, l1_points = self.sa1(l0_xyz, l0_points)
        l2_xyz, l2_points = self.sa2(l1_xyz, l1_points)
        l3_xyz, l3_points = self.sa3(l2_xyz, l2_points)

        # Decodificador
        l2_points = self.fp3(l2_xyz, l3_xyz, l2_points, l3_points)
        l1_points = self.fp2(l1_xyz, l2_xyz, l1_points, l2_points)
        l0_points = self.fp1(l0_xyz, l1_xyz, l0_points, l1_points)

        # Concatenar remission antes del MLP
        l0_points = torch.cat([l0_points, remission], dim=1)

        # MLP profundo para clasificaci n por punto
        x = F.relu(self.bn1(self.conv1(l0_points)))
        x = self.drop1(x)
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.drop2(x)
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.drop3(x)
        x = self.conv4(x)
        x = F.log_softmax(x, dim=1)

        return x
```

Figura A.13: Clase `PointNet2SemSeg`: Definición de la arquitectura de PointNet++*, donde se modifica el *backbone* a la salida del decodificador de características.

Bibliografía

[1] M. de Ciencia e Innovación, "Proyecto gaia: Gestión integral para la prevención, extinción y reforestación debido a incendios forestales," Proyecto financiado por el Ministerio de Ciencia e Innovación en el marco de la convocatoria AEI 2023, 2023, código de proyecto: PLEC2023-010303. [Online]. Available: https://roboticslaburjc.github.io/projects/gaia

[2] B. Schwarz and N. El-Sheimy, "Mapping the world in 3d using lidar," *Photogrammetric Engineering & Remote Sensing*, vol. 76, no. 6, pp. 638–647, 2010.

[3] Z. Wang, B. Dai, and H. Fu, "A fast approach for vehicle-like region proposal based on 3d lidar data," 08 2015, pp. 508–512.

[4] C. Müller, B. Graf, and K. Pfeiffer, Eds., *World Robotics 2022 – Service Robots*. Frankfurt am Main, Germany: IFR Statistical Department, VDMA Services GmbH, 2022.

[5] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, p. 362–386, Nov. 2019. [Online]. Available: http://dx.doi.org/10.1002/rob.21918

[6] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[7] P. Mortimer, R. Hagmanns, M. Granero, T. Luettel, J. Petereit, and H.-J. Wuensche, "The goose dataset for perception in unstructured environments," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2024. [Online]. Available: https://arxiv.org/abs/2310.16788

[8] P. Jiang, P. Osteen, M. Wigness, and S. Saripalli, "Rellis-3d dataset: Data, benchmarks and analysis," 2020.

[9] A. Santo, E. Heredia, C. Viegas, D. Valiente, and A. Gil, "Ground segmentation for lidar point clouds in structured and unstructured environments using a hybrid neural–geometric approach," *Technologies*, vol. 13, no. 4, 2025. [Online]. Available: https://www.mdpi.com/2227-7080/13/4/162

[10] S. R. Price, H. B. Land, S. S. Carley, S. R. Price, S. J. Price, and J. R. Fairley, "Expanding ground vehicle autonomy into unstructured, off-road environments: Dataset challenges," *Applied Sciences*, vol. 14, no. 18, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/18/8410

[11] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3d point clouds: A survey," *CoRR*, vol. abs/1912.12033, 2019. [Online]. Available: http://arxiv.org/abs/1912.12033

[12] MathWorks. (2024) Automate ground truth labeling for lidar point cloud semantic segmentation using lidar labeler. Último acceso: 4 de junio de 2025. [Online]. Available: https://www.mathworks.com/help/lidar/ug/automate-lidar-labeling-for-semantic-segmentation.html

[13] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *arXiv preprint arXiv:1612.00593*, 2016.

[14] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *arXiv preprint arXiv:1706.02413*, 2017.

[15] H. Thomas, C. R. Qi, J.-E. Deschaud, B. Marcotegui, F. Goulette, and L. J. Guibas, "Kpconv: Flexible and deformable convolution for point clouds," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 6411–6420.

[16] Q. Hu, B. Yang, L. Xie, S. Rosa, Y. Guo, Z. Wang, N. Trigoni, and A. Markham, "Randla-net: Efficient semantic segmentation of large-scale point clouds," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 11 108–11 117.

[17] S. Royo and M. Ballesta-Garcia, "An overview of lidar imaging systems for autonomous vehicles," *Applied Sciences*, vol. 9, no. 19, p. 4093, 2019.

[18] ——, "An overview of lidar imaging systems for autonomous vehicles," *Applied Sciences*, vol. 9, no. 19, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/19/4093

[19] T. Autonomous, "Types of lidar: Solid-state vs mechanical vs flash vs mems," https://www.thinkautonomous.ai/blog/types-of-lidar/, Jan. 2023.

[20] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall, "Semantickitti: A dataset for semantic scene understanding of lidar sequences," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 9297–9307.

[21] R. Shrestha, G.-L. Kwon, and Y.-J. Lee, "A survey on lidar scanning mechanisms," *Electronics*, vol. 9, no. 5, p. 741, 2020.

[22] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," *IEEE International Conference on Robotics and Automation*, pp. 1–4, 2011.

[23] G. Turk, "The ply polygon file format," in *Technical Report*, 1994, http://paulbourke.net/dataformats/ply/.

[24] M. Gao, N. Ruan, J. Shi, and W. Zhou, "Deep neural network for 3d shape classification based on mesh feature," *Sensors*, vol. 22, no. 18, p. 7040, 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/18/7040

[25] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015, pp. 922–928.

[26] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *arXiv preprint arXiv:1612.00593*, 2016.

[27] C. Choy, J. Gwak, and S. Savarese, "4d spatio-temporal convnets: Minkowski convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 3075–3084.

[28] X. Zhu, H. Zhou, T. Wang, F. Hong, Y. Ma, W. Li, H. Li, and D. Lin, "Cylindrical and asymmetrical 3d convolution networks for lidar segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 9939–9948.

[29] H. Tang, Z. Liu, S. Zhao, Y. Lin, J. Lin, H. Wang, and S. Han, "Searching efficient 3d architectures with sparse point-voxel convolution," in *European Conference on Computer Vision*. Springer, 2020, pp. 685–702.

[30] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *ACM Transactions on Graphics*, vol. 38, no. 5, pp. 1–12, 2019.

[31] A. Jhaldiyal and N. Chaudhary, "Semantic segmentation of 3d lidar data using deep learning: a review of projection-based methods," *Applied Intelligence*, vol. 53, no. 6, pp. 6844–6855, 2023.

[32] H. Zhao, L. Jiang, J. Jia, P. H. Torr, and V. Koltun, "Point transformer," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 16 259–16 268.

[33] X. Lai, Y. Chen, F. Lu, J. Liu, and J. Jia, "Spherical transformer for lidar-based 3d recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 17 545–17 555.

[34] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, J. Gall, and C. Stachniss, "Towards 3D LiDAR-based semantic scene understanding of 3D point cloud sequences: The SemanticKITTI Dataset," *The International Journal on Robotics Research*, vol. 40, no. 8-9, pp. 959–967, 2021.

[35] 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," *Proceedings of*

the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 11 621–11 631, 2020.

[36] W. Tan, N. Qin, L. Ma, Y. Li, J. Du, G. Cai, K. Yang, and J. Li, "Toronto-3D: A large-scale mobile lidar dataset for semantic segmentation of urban roadways," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 202–203.

[37] Velodyne Lidar, Inc., "Alpha prime datasheet," https://velodynelidar.com/products/alpha-prime/, 2020, accedido en mayo de 2025.

[38] Ouster, Inc., "Os0-32 data sheet rev 7," https://ouster.com/products/os0-lidar-sensor/, 2023, accedido en mayo de 2025.

[39] C. Robotics, "Warthog unmanned ground vehicle robot," 2025, Último acceso: 24 de junio de 2025. [Online]. Available: https://clearpathrobotics.com/warthog-unmanned-ground-vehicle-robot/

[40] Ouster, Inc., "Os1 lidar sensor datasheet," https://data.ouster.io/downloads/datasheets/datasheet-rev7-v3p1-os1.pdf, 2023, revision 7, Version 3.1.

[41] Velodyne LiDAR, Inc., "Ultra puck vlp-32c datasheet," https://www.mapix.com/wp-content/uploads/2018/07/63-9378_Rev-D_ULTRA-Puck_VLP-32C_Datasheet_Web.pdf, 2018, revision D.

[42] D. Zhu, H. Yao, B. Jiang, and P. Yu, "Negative log likelihood ratio loss for deep neural network classification," 2018. [Online]. Available: https://arxiv.org/abs/1804.10690

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[44] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010.

[45] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[46] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[47] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 7, pp. 3523–3542, 2021.

[48] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[49] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[50] M. Minsky and S. Papert, *Perceptrons: An introduction to computational geometry*. MIT press, 1969.

[51] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814, 2010.

[52] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016.

[53] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[54] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[55] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.