UNIVERSIDAD INTERNACIONAL DE LA RIOJA

Universidad Internacional de La Rioja

Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Análisis y Visualización de Datos Masivos

Comparativa U-Net vs. ResNet + PsPNet para el dataset A2D2.

Trabajo Fin de Máster

Tipo de trabajo: Comparativa de Soluciones

Presentado por: Justo Sarmentero, Luis Vicente.

Director/a: Vara Mesa, Juan Manuel.

Resumen

Los avances en el desarrollo y comercialización de sistemas de conducción autónoma han tenido una fuerte aceleración en los últimos años. La complejidad de estos sistemas, la cantidad y diversidad de datos que utilizan y la velocidad de respuesta que necesitan, obligan a buscar soluciones con la máxima velocidad ejecución y el menor coste computacional posible. Dentro de esta búsqueda, la experimentación y comparación de diferentes alternativas para la interpretación del entorno del vehículo es capital para encontrar la solución óptima. Este trabajo se centra en la tarea de la visión artificial para la interpretación del entorno y particularmente en la segmentación semántica de imágenes mediante la utilización de redes neuronales U-Net, ResNet y PsPNet.

El objetivo de este trabajo es contribuir a encontrar la solución más óptima para la tarea de segmentación semántica realizando el entrenamiento, con el dataset A2D2 de Audi AG®, de una red neuronal basada en la arquitectura U-Net y la comparación de su rendimiento con el obtenido por los autores de la publicación del mencionado dataset. Dichos autores, utilizaron una arquitectura compuesta por la combinación de las redes neuronales ResNet y PsPNet.

En este proyecto, primero se abordan los enfoques actuales para la conducción autónoma y cómo encaja la segmentación semántica de imágenes en ellos. Se desarrollan los elementos principales para la ejecución de esta tarea como sensores, dataset, simuladores, herramientas ETL y técnicas de inteligencia artificial profundizando en los distintos tipos de redes neuronales. Se proporciona un método completo mediante la utilización de librerías de Python, TensorFlow y Keras para la extracción, tratamiento y carga del dataset A2D2, la construcción del modelo de red neuronal basado en U-Net, su entrenamiento con dicho dataset, y la evaluación del mismo bajo las métricas más comunes en segmentación semántica.

Como resultado de este trabajo, además del análisis en profundidad del dataset A2D2 y los algoritmos en Python comentados al detalle, se obtiene el rendimiento del modelo U-Net para los índices Dice e IoU, así como el tiempo de entrenamiento y evaluación. Estos resultados se comparan con los publicados por los autores del dataset A2D2 de Audi AG® .

Esta comparación permite concluir que la arquitectura U-Net no presenta una mejora en el rendimiento con respecto a los resultados obtenidos por los mencionados autores. No obstante, dado que los valores de tiempo de entrenamiento y tiempo de valuación no han sido publicados por dichos autores no es posible compararlos con los obtenidos en este trabajo, por lo que quedan disponibles para posibles líneas de trabajo futuras. De igual modo, como futuros trabajos, se propone la parametrización de la red neuronal U-Net de forma diferente para conseguir una mejora del rendimiento.

Palabras clave: Visión Artificial, Segmentación Semántica, Redes Neuronales, U-Net, Python, TensorFlow, Keras.

Abstract

Advances in the development and commercialization of autonomous driving systems have experienced a significant acceleration in recent years. The complexity of these systems, the volume and diversity of data they utilize, and the speed of response they require compel the search for solutions that offer maximal execution speed and minimal computational cost. Within this pursuit, experimentation and comparison of different alternatives for vehicle environment interpretation are crucial in finding the optimal solution. This work focuses on the task of artificial vision for environment interpretation, particularly on semantic image segmentation using U-Net, ResNet, and PsPNet neural networks.

The objective of this study is to contribute to finding the optimal solution for semantic segmentation tasks by training a neural network based on the U-Net architecture with the A2D2 from Audi AG ® dataset. The performance of this model is then compared with the results obtained by the authors of the aforementioned dataset publication. These authors employed a combination of ResNet and PsPNet neural networks for their architecture.

In this project, we first address the current approaches for autonomous driving and how semantic image segmentation fits into them. The main elements for the execution of this task such as sensors, datasets, simulators, ETL tools and artificial intelligence techniques are developed by delving into the different types of neural networks. A complete method is provided using Python and TensorFlow libraries for the extraction, processing and loading of the A2D2 dataset as well as for the construction of the neural network model based on U-Net, its training with this dataset, and its evaluation under the most common metrics in semantic segmentation.

As a result of this work, along with an in-depth analysis of the A2D2 dataset and extensively commented Python algorithms, the performance of the U-Net model is obtained in terms of Dice and IoU indices, as well as training and evaluation times. These results are compared with those published by the authors of the A2D2 dataset from Audi AG® .

This comparison leads to the conclusion that the U-Net architecture does not exhibit improved performance compared to the results achieved by the aforementioned authors. However, since the training and evaluation time values have not been published by said authors, a comparison with the results of this work is not feasible. Therefore, these aspects remain available for potential future lines of work. Similarly, future work could involve parameterizing the U-Net neural network differently to achieve enhanced performance.

Keywords: Computer Vision, Semantic Segmentation, Neural Networks, U-Net, Python, TensorFlow, Keras.

Índice de contenidos

Índice de tablas

Índice de ilustraciones

Introducción.

1. Introducción

La investigación y el desarrollo de la conducción autónoma ha experimentado un crecimiento significativo tanto en resultados directamente ligados al avance de este campo, como en la construcción de conocimiento y técnicas aplicables a otras áreas (Turienzo, 2022).

La complejidad de la conducción autónoma demanda la división de las tareas que lo hacen posible en módulos individuales tales como la percepción del entorno, planificación de la ruta, control del vehículo, diagnosis del estado del vehículo, percepción de entradas del usuario, …) (Le Mero et al., 2022).

Dentro del ámbito de la percepción del entorno, la técnica de la segmentación de imágenes constituye una de las herramientas más ampliamente utilizadas (Cakir et al., 2022). De forma muy básica, la segmentación consiste en dividir una imagen digital en varias grupos de píxeles denominados segmentos para lo cual se clasifica cada pixel de la imagen en una categoría que depende de lo que queremos identificar en la imagen. La segmentación semántica es una especialización de dicha técnica que se focaliza en la clasificación de los píxeles en categorías de alto nivel significante que, para conducción autónoma, serán vehículos, peatones, carretera, señales, semáforos, carriles, líneas divisorias… etc.

Este Trabajo fin de master se centrará en la aplicación de tecnologías del estado del arte para la extracción tratamiento y carga de datos (ETL), así como la utilización de modelos de redes neuronales para la segmentación semántica utilizando sets de datos procedentes de entornos reales, en síntesis:

- Entrenaremos una red neuronal con arquitectura U-Net (Ronneberger et al., 2015) con un conjunto de datos procedentes de entornos reales usando librerías de Keras (Chollet & others, 2015) y TensorFlow (Martín~Abadi et al., 2015).

- Compararemos los resultados del entrenamiento con la arquitectura U-Net con las referencias existentes en la literatura y extraeremos conclusiones sobre la idoneidad de usar la red neuronal U-Net para la segmentación semántica en comparación con la solución existente en la literatura.

En este trabajo se utilizará el conjunto de datos A2D2 procedente de Audi AG® (Geyer et al., 2020). Este conjunto de datos tiene un tamaño de 2.3 Tb aproximadamente y consta de más de 40,000 fotogramas etiquetados para segmentación semántica (2D) y nube de puntos (3D). Asociados a los ficheros de nubes de puntos, encontraremos 12,000 fotogramas etiquetados para marcos de frontera 3D (conocido por su equivalente en inglés boundary boxes) que permiten la localización identificación de objetos. Además, contiene 390.000 fotogramas no etiquetados con sus correspondientes datos de sensores procedentes de varios recorridos en tres ciudades distintas (Gaimershein, Ingolstadt y Munich). Toda la información procede de interacciones con vehículos reales en entornos reales y se distribuye bajo licencia CC BY-ND 4.0[1]. La documentación de este dataset ofrece resultados con respecto a la métrica IOU (Taha & Hanbury, 2015) (conocida en inglés como insertion over the union) para entrenamientos realizados en varios escenarios. Describiremos este último punto en el capítulo 2.

1.1. Motivación

La conducción autónoma emplea de forma exhaustiva, entre otras, 2 tecnologías incluidas en el currículo académico de este master: El big data y el deep learning.

La cantidad y complejidad de los entrenamientos y pruebas a los que se deben someter los agentes inteligentes utilizados para el desarrollo de la conducción autónoma conllevan la necesidad de tratar con cantidades

[1] https://creativecommons.org/licenses/by-nd/4.0/

ingentes de datos que deben extraer, transformar y cargar (ETL) con alta velocidad (Díez Ramírez, 2018). Estos datos serán después utilizados por el agente inteligente para tomar las decisiones que conlleven la consecución del objetivo. Los datos proceden habitualmente de dos fuentes diferenciadas:

- Salida de señales de sensores de distintos tipos. Este tipo de datos, además de su alto volumen, llevan implícita una alta velocidad de generación.
- Captura de imágenes que añade a las 2 características anteriores (volumen y velocidad) la existencia de estructuras de datos diversas.

Se hace patente que el campo de la conducción autónoma cumple con las premisas para la aplicación de tecnologías big data dado el volumen y variedad de los datos, así como la velocidad necesaria para la extracción transformación y carga de los mismos.

La técnica de deep reinforcement learning es ampliamente utilizada para el entrenamiento de agentes inteligentes para la conducción autónoma (Roy Amante Salvador, 2019). Deep reinforcement learning (o aprendizaje profundo por refuerzo) es una técnica de machine learning en la que un agente aprende a realizar una tarea a través de repetidas interacciones de prueba y error con un entorno dinámico. Este enfoque de aprendizaje permite que el agente tome una serie de decisiones que amplían al máximo su métrica de recompensa, sin intervención humana y sin estar programado explícitamente para completar la tarea (Sallab et al., 2017).

El trabajo computacional para el entrenamiento por deep reinforcement learning es dependiente de la interpretación del entorno por parte del agente. Es posible mejorar esta interpretación mediante la aplicación de técnicas de machine learning para entrenar modelos que sirvan como encoder de las entradas del entorno y permitan centrar el entrenamiento del agente en las reacciones necesarias para conseguir el objetivo. De no

utilizar un modelo entrenado para la interpretación del entorno, la fase del aprendizaje del agente partiría de un estado inicial aleatorio en este aspecto, lo que condicionaría negativamente los resultados y el rendimiento del mismo (Salvador et al., 2020).

El uso de las imágenes procedentes de las cámaras de los vehículos es uno de los elementos principales que permiten la percepción del entorno proporcionando información crucial sobre el dominio de la conducción como zonas de conducción despejadas y obstáculos circundantes (Cakir et al., 2022). Por este motivo, la optimización de esta tarea tanto en precisión de resultados como en consumo de recursos es relevante para permitir al agente inteligente focalizar sus recursos en la técnica de deep reinforcement learning comentada anteriormente. La técnica de segmentación semántica permite esta optimización en la interpretación del entorno (Cakir et al., 2022).

Existen en la literatura numerosos casos de entrenamiento de redes neuronales para segmentación semántica en diversos ámbitos, medicina, vigilancia, conducción autónoma... Para la conducción autónoma encontramos entrenamientos realizados tanto con datos procedentes de entornos reales como de simuladores. Estos últimos son utilizados por la evidente barrera de coste que constituye la extracción de entornos reales, así como paliar las limitaciones asociadas a la reproducibilidad de la técnica de deep reinforcement learning (al no ser posible replicar cada escenario posible). En el capítulo 2 se muestra un análisis detallado de los set de datos disponibles para ambos casos.

La variedad de arquitectura de redes utilizadas para la segmentación semántica está también presente en la literatura. Si bien el denominador común de la mayoría de ellas son las redes neuronales convolucionales (Long et al., 2014), se han desarrollado variaciones de las mismas para optimizarlas en set de datos concretos o aplicaciones específicas. Por ejemplo, (Chen et al., 2018) usa la arquitectura DeepLav 3+ para mejorar

los resultados en los sets de datos para PASCAL VOC 2012 (Everingham et al., 2022) y Cityscapes (Marius Cordts, 2016), mientras que (Ronneberger et al., 2015) usa la arquitectura U-Net para interpretación de imágenes biomédicas.

Entre las múltiples posibilidades que se derivan de la existencia de la citada diversidad de arquitecturas cabe destacar la posibilidad de utilizar arquitecturas pre-entrenadas con un set de datos para predecir resultados sobre otro set de datos diferente. Es lo que conocemos como transfer learning (Zhuang et al., 2019). Siguiendo la misma lógica, es posible también probar arquitecturas que funcionan bien en un ámbito (como la biomedicina) para otro distinto (como la conducción autónoma) buscando mejoras en coste computacional o velocidad de predicción (Hussain et al., 2022)

En este proyecto fin de master ensayaremos el escenario de entrenar una arquitectura de red neuronal creada para el ámbito de la biomedicina como es la U-Net con el set de datos A2D2 de Audi AG® que procede de un entorno real.

La comparación de los resultados de la métrica Mean IOU nos permitirá concluir si la arquitectura U-Net tiene un mejor rendimiento que la arquitectura con las que fue entrenado el dataset A2D2 por sus autores.

1.2. Planteamiento del trabajo

El propósito de este trabajo es contribuir a encontrar la solución más óptima para la tarea de segmentación semántica realizando el entrenamiento, con el dataset A2D2, de una red neuronal basada en la arquitectura U-Net y la comparación de su rendimiento con el obtenido por los autores de la publicación del mencionado dataset. Dichos autores, utilizaron una arquitectura compuesta por la combinación de las redes neuronales ResNet y PsPNet.

Durante el estudio del material disponible (dataset, Redes Neuronales previamente configuradas para trabajos similares, librerías de Python, etc.) se ha observado que existen numerosas fuentes cuyos enlaces están rotos, así como código usado con librerías de versiones antiguas. Es de esperar que todo el contenido existente haya de ser adaptado y actualizado al estado actual de las herramientas.

Tras revisar el estado del arte en lo que se refiere a conducción autónoma, técnicas de machine learning, sensores para captación del entorno y tecnologías big data disponibles, focalizaremos la atención en elaborar un código Python que permita el entrenamiento de la red neuronal U-Net. Una vez conseguido, adecuaremos el dataset A2D2 para hacerlo compatible con la dicha arquitectura. Finalmente entrenaremos la red neuronal con el dataset completo y extraeremos conclusiones comparando el rendimiento con las referencias existentes en bibliografía.

Debido al coste del entrenamiento en la nube, se opta por realizar el entrenamiento on-premise.

1.3. Estructura de la memoria

La memoria de este trabajo fin de master se desarrolla de la siguiente forma:

- En el capítulo de Introducción:

Se describe la motivación, el planteamiento de este trabajo fin de master, el objetivo del mismo y los diferentes hitos a cumplir para alcanzarlo. El plan de trabajo del mismo queda relejado en un diagrama basado en el método Gantt (Gantt, 1910) que sirve como referencia temporal para la consecución de dichos ítems. Se describe la metodología adoptada, así como una breve referencia a como se respeta el régimen general de protección de datos (Unión Europea, 2016).

- En el capítulo de estado del arte:

Se repasan las tecnologías, conceptos y métodos que están permitiendo el avance de la conducción autónoma.

Se visitan los diferentes sensores que permiten la captura de información del entorno y se evalúan el estado de las diferentes tecnologías de extracción, transformación y carga para datos masivos que pudieran ser aplicables para la construcción del archivo de entrenamiento del modelo.

Se describe el ecosistema actual de tecnologías que permiten la captura tratamiento y carga de datos masivos, así como los dataset y simuladores existentes para el entrenamiento de modelos de conducción autónoma.

Se visitan en profundidad las diferentes técnicas de machine learning que están siendo aplicadas tanto para el entrenamiento de modelos para la interpretación del entorno mediante segmentación semántica de imágenes.

Se obviaremos la metodología y mecatrónica necesarias para el entrenamiento y prueba del agente inteligente para la conducción autónoma dejando estos puntos fuera del ámbito de aplicación de este trabajo fin de master

- En el capítulo de desarrollo de la contribución:

Se describe en detalle el dataset A2D2 de Audi AG®. Se muestra la solución adoptada para la extracción transformación y carga de los datos. Se describe la ejecución del entrenamiento del modelo de interpretación del entorno mediante segmentación semántica con la utilización de la red neuronal U-Net. Se detalla cómo se hace cada fase, mediante la explicación de código Python utilizado y se obtienen resultados que permiten la comparación del rendimiento del modelo entrenado con las referencias existentes en la literatura.

- En el capítulo de conclusiones :

Se explican los aprendizajes obtenidos, se arroja luz sobre la potencial mejora de rendimiento con la utilización de la arquitectura U-Net y se

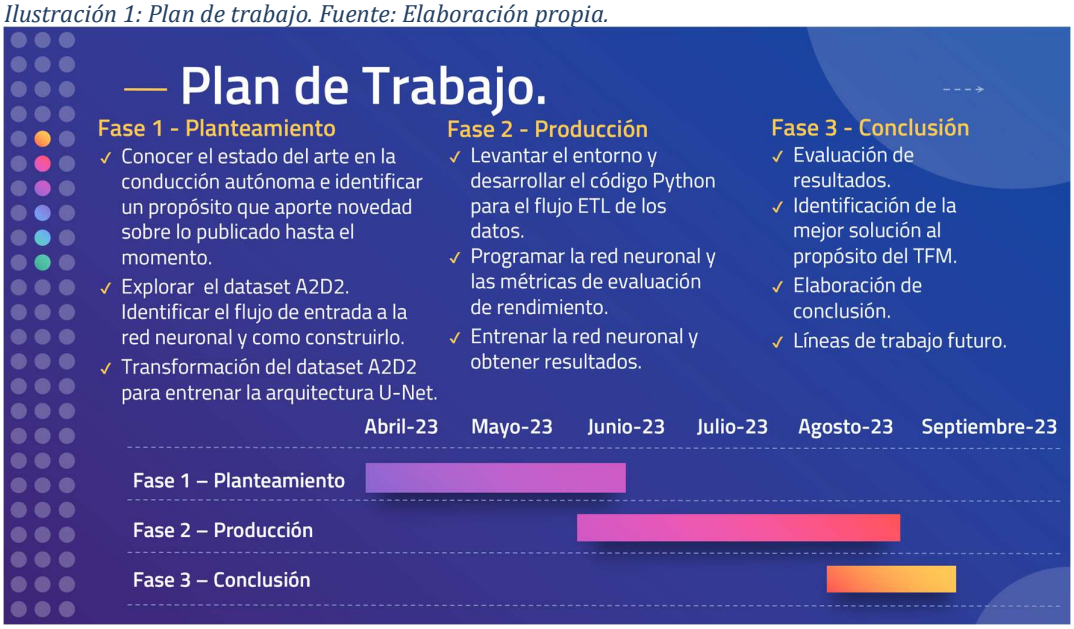
señalan las potenciales líneas de trabajo futuro que pudieran dar continuidad a este trabajo fin de master.

1.4. Objetivos

El Objetivo del presente trabajo fin de master es averiguar si tiene sentido la utilización de una arquitectura creada para el campo biomédico como U-Net para la segmentación semántica de imágenes para la conducción autónoma. Para ello compararemos el rendimiento de dicha arquitectura para el dataset A2D2 de Audi AG® capturado de entornos reales y lo compararemos con la experimentación realizada con dicho dataset con las arquitecturas definidas en su propia documentación que son: ResNet101 + PSP Net.

La consecución del objetivo principal será el resultado de la concatenación de las siguientes tareas que serán llevadas cabo en el orden de tiempo identificado en el diagrama de la Ilustración 1.

Ilustración 1: Plan de trabajo. Fuente: Elaboración propia.



Se plantea conocer el estado del arte en la disciplina de la conducción autónoma de vehículos como el primer objetivo específico a cumplir centrándose en lo que se refiere al entendimiento de las tecnologías

existentes para la captación del entorno, interpretación del mismo, así como la extracción, tratamiento y carga de los datos.

El segundo objetivo es la exploración detallada del dataset A2D2 de Audi AG® , la compresión de su estructura y el formato de sus imágenes. Con esta información se identifican las operaciones de transformación a realizar para hacer este dataset compatible con la arquitectura U-Net.

Las operaciones a realizar para adaptar el dataset constituyen el tercer objetivo, se preparará un programa en lenguaje Python con el que, procesaremos el dataset de forma independiente antes del entrenamiento.

Una vez completadas las fases anteriores se llega a etapa de producción en la que en primer lugar se levanta el entorno de programación. Este objetivo incluye la instalación de las librerías necesarias para la captura procesamiento y carga de datos, así como el entrenamiento de la red neuronal U-Net utilizando la unidad gráfica de procesamiento (GPU).

Una vez se tiene el entorno configurado se programan los protocolos de extracción transformación y carga, así como el modelo de red neuronal junto con las métricas que para la evaluación del rendimiento del modelo. Esta etapa concluye en el momento que se alcanza el objetivo de entrenar la red neuronal, guardar los modelos y obtener resultados.

El objetivo de la etapa de conclusión es el que aglutina el impacto de este proyecto fin de master. Comparando los resultados obtenidos en los distintos escenarios se identifica la mejor arquitectura para la segmentación semántica de imágenes para conducción autónomo entra la propuesta (U-Net) y la usada como referencia en la documentación del dataset A2D2 (que es la formada por ResNet + PsPNet).

1.5. Metodología de trabajo

Con el objetivo en mente de averiguar si tiene sentido la utilización de una arquitectura U-Net para la segmentación semántica de imágenes para la conducción autónoma, el contenido de este capítulo pretende ayudar a

que el trabajo realizado sea replicable y permitir a otros investigadores realizar un estudio utilizando el mimo enfoque metodológico y comparar los potenciales hallazgos con los existentes en este trabajo.

En este trabajo se adopta un enfoque inductivo con el que, mediante exploración, se concluye a partir de los resultados obtenidos de las métricas de rendimiento de la red neuronal U-Net. En el capítulo 3 describimos lo realizado con el detalle suficiente para que puedan ser replicado y utilizado como base o continuación para estudios futuros.

En arras de garantizar la factibilidad del estudio en cuanto a coste tiempo se ha utilizado un ordenador portátil cuyas características se describirán en el capítulo 3. La consecuente limitación de potencia computacional se hace presente si comparamos este recurso con los disponibles en la nube en proveedores como Microsoft®, Google® o Amazon®.

Dado que el objetivo es comparar rendimiento de 2 redes neuronales con respecto al mismo dataset, el potencial sesgo existente en los datos, por ejemplo, en cuanto a los escenarios escogidos, no impactará en las conclusiones dado que los sets de datos son idénticos. El sesgo en los datos no es por tanto una limitación en este trabajo fin de master.

En síntesis, este trabajo fin de master utiliza una metodología inductiva para extraer conclusiones sobre las diferencias de rendimiento de la red neuronal U-Net frente a la red neuronal utilizada en la referencia del dataset A2D2 de Audi AG® . La principal limitación radica en la potencia computacional consecuencia de la utilización de un ordenador portátil para hacer el trabajo fin de master factible en términos de coste.

1.6. Reglamento general de protección de datos (RGPD)

El artículo de 4.1 del reglamento general de protección de datos define datos personales como toda información sobre una persona física identificada o identificable (el interesado). Se considerará persona física identificable toda persona cuya identidad pueda determinarse, directa o

indirectamente, en particular mediante un identificador, como por ejemplo un nombre, un número de identificación, datos de localización, un identificador en línea o uno o varios elementos propios de la identidad física, fisiológica, genética, psíquica, económica, cultural o social de dicha persona. (Unión Europea, 2016)

El dataset A2D2 de Audi AG® contiene imágenes captadas en entornos reales que incluyen número de placas de matrícula que junto con las marcas de tiempo y datos geoespaciales incluidos en ellas podrían dar lugar a la identificación de individuos físicos. La propia documentación del dataset identifica este hecho. Como solución al mismo, en el propio dataset, se aplicó a todas las imágenes una distorsión de las placas de matrícula volviéndolas ilegibles. Se elimina así el cuasi-identificador de las imágenes lo que anula la posible identificabilidad de los sujetos (Geyer et al., 2020). La utilización del dataset A2D2 de Audi AG® no conlleva, por tanto, la aplicación del reglamento general de protección de datos.

Estado del Arte

2. Estado del arte

La evolución de la tecnología de sensores, hardware, software, big data, e inteligencia artificial han hecho posible, en entornos reales, la coexistencia de vehículos con conducción autónoma interactuando con vehículos conducidos por personas. Prueba de ello es el primer viaje comercial realizado por un vehículo completamente autónomo para transportar a un particular al aeropuerto Sky Harbor en Phoenix (US) el primero de noviembre de 2021 por la empresa Waymo® (The Waymo Team, 2022)

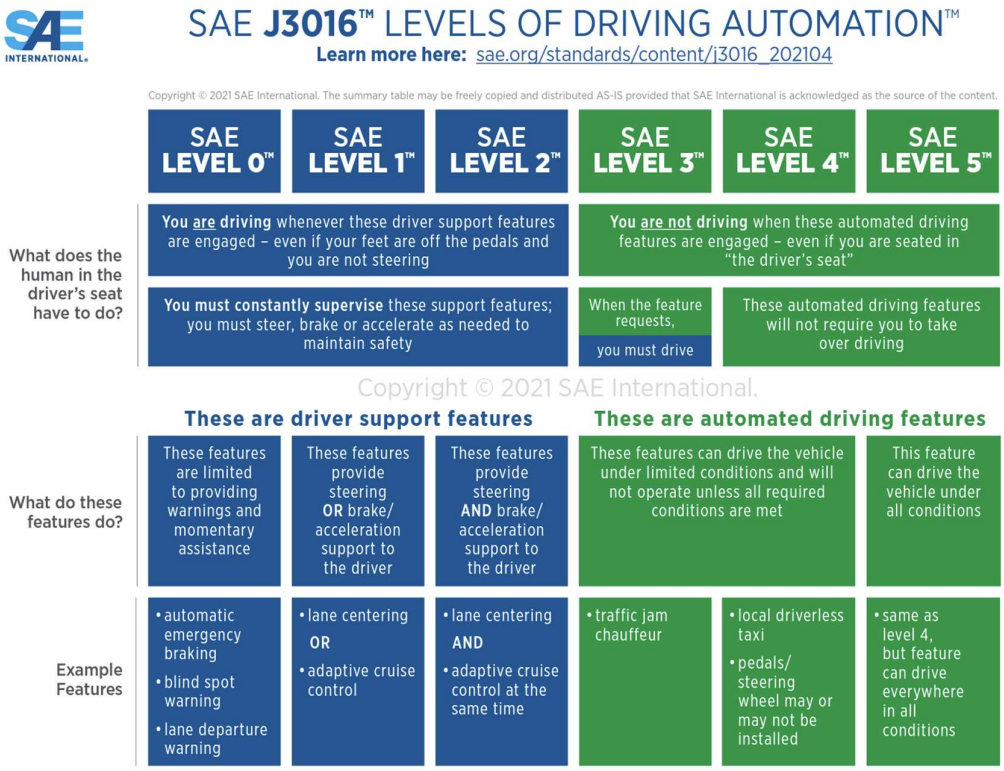
El estándar internacional SAE J3016 (SAE INTERNATIONAL, 2021) ,que define los diferentes niveles de autonomía de vehículos, data de 2014. Desde entonces ha sufrido varias modificaciones a medida que se entendía mejor la evolución y el potencial de la conducción autónoma. Este hecho da idea de la intuición, que ya en 2014, se tenía sobre el potencial de esta tecnología y de los posibles conflictos que podrían surgir de aplicarse sin una referencia clara. La evolución de este estándar y la rigurosidad de su aplicación (Kusano et al., 2022) refleja la seriedad y relevancia con la que se está tratando el desarrollo de esta tecnología. El estado actual de SAE J3016 se resume en la Ilustración 2.

Para tener una idea de la magnitud de la que estamos hablando, la consultora Boston Consulting Group anticipó en 2015 que el campo de la conducción autónoma alcanzará la cifra de 77.000 millones de dólares en 2035 (X. Mosquet, 2015) mientras que las predicciones de Brookings Institution (D. M. West, 2016) predice que en 2040 el 25% de los vehículos serán autónomos e IHS (I. Markit, 2014) estima que en 2050 casi la totalidad de los vehículos tendrán esta característica incorporada.

Además, existe un ecosistema sinérgico en el que los campos de la captura de datos por sensores, tecnologías big data, y técnicas de inteligencia artificial interactúan aprovechando sus capacidades e innovaciones. Se generan aplicaciones para diversas áreas como la

robótica y el pilotaje de drones y aeronaves que, si bien no son transversalmente aplicables de forma directa, pueden adaptarse, ya que todas ellas comparten el mismo paradigma a resolver: Conseguir que un agente inteligente interprete un entorno físico real y reaccione al mismo para cumplir un objetivo asignado. Un ejemplo de ello es la evolución del simulador AirSim (Shah et al., 2017). Este simulador empezó como un entorno de alta fidelidad para coches, con el objetivo de aplicarlo a la conducción autónoma. El repositorio del simulador de coches AirSim será archivado en 2024 (Microsoft®, 2023a) para enfocar los esfuerzos del equipo de desarrollo en un simulador dedicado al pilotaje autónomo de aeronaves (Microsoft®, 2023b) que verá la luz en el año 2024.

Ilustración 2: Niveles de automatización de la conducción. Fuente: (SAE INTERNATIONAL, 2021).



2.1. Enfoques para los sistemas de conducción autónoma

El estado del arte actual para la conducción autónoma de vehículos utiliza un enfoque modular para el diseño del sistema que permitan la autonomía de conducción del vehículo (Le Mero et al., 2022). Siguiendo este enfoque,

existe un módulo encargado de cada tarea individual de la conducción (percepción del entorno, planificación de la ruta, control del vehículo, diagnosis del estado del vehículo, percepción de entadas del usuario, …). En Europa, con la aplicación de este enfoque, se logró la homologación por parde de la Autoridad Alemana de Transporte por Carretera (KBA) de nivel 3 (SAE J3016) (SAE INTERNATIONAL, 2021) de conducción autónoma por parte de Mercedes Benz en diciembre 2021. Los modelos S y EQS fueron los primeros vehículos equipados con esta utilidad en mayo de 2022 y es posible su uso desde entonces en carreteras alemanas exclusivamente.

En paralelo al enfoque modular, durante los últimos años ha cobrado relevancia el enfoque denominado end-to-end también conocido como comportamiento por acto reflejo. En este enfoque, existe un único modulo en el diseño del sistema de conducción autónoma el cual recibe datos en crudo de sensores que mapean el entorno. Esos datos son interpretados para obtener salidas en forma de señales de actuación para los diferentes sistemas mecatrónicos que permiten la conducción física del vehículo (dirección, frenos, acelerador, etc.…) (Le Mero et al., 2022)

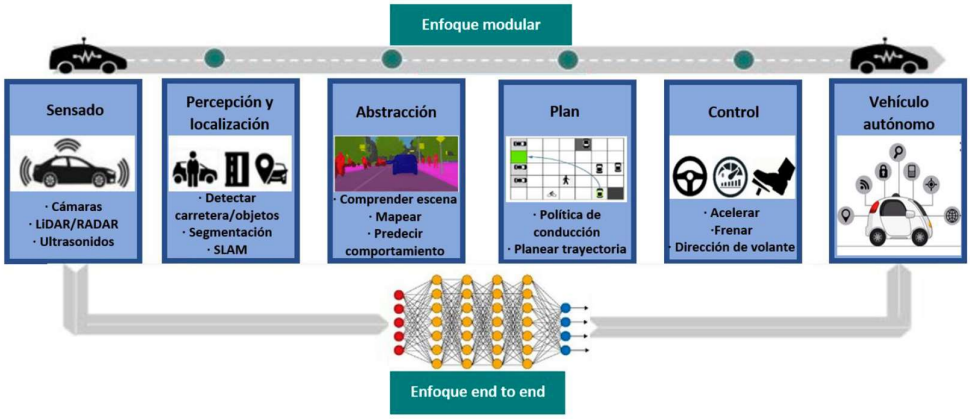
Como se describe en (Le Mero et al., 2022) existe un enfoque intermedio entre los anteriormente explicados que fue expuesto por primera vez en (C. Chen, et al. 2015). Este modelo es denominado como de percepción directa. En este enfoque, existe un módulo que realiza la percepción del entorno y la planificación (que no ejecución) de las tareas de conducción bajo un entorno de acto reflejo. La salida de este módulo es utilizada para tomar decisiones en base a simples esquemas de control.

Mientras que el enfoque modular permite una transparente verificabilidad permitiendo que la salida de cada módulo individual pueda ser verificada y contrastada con el objetivo deseado, los sistemas utilizados en el enfoque end-to-end, carecen de toda posibilidad de verificación que no sea la acción ejecutada frente al entorno recibido, lo cual le confiere una

naturaleza de caja negra poco atractiva desde el punto de vista técnico, ingenieril o científico.

Por otro lado, el enfoque modular es intensivo en coste computacional y por concepto, captura y procesa información que pudiera ser irrelevante para la tarea de conducción, haciendo este enfoque ineficiente desde el punto de vista de coste. Los modelos utilizados en el enfoque end-to-end, son basados en algoritmos de machine learning como el deep reinforcement learning que, por definición, aprenden para ser computacionalmente eficientes. Además, el pre-entrenamiento de redes neuronales para interpretar el entorno (como la segmentación semántica de imágenes) permite concentrar el aprendizaje del agente con el algoritmo de deep reinforcement learning. Estos hechos en combinación con la complejidad técnica de la conducción autónoma hacen del enfoque end-to-end el mejor balanceado en relación coste rendimiento para el paradigma de la conducción autónoma (Salvador et al., 2020).

Ilustración 3: Comparación enfoque modular vs enfoque end-to-end. Fuente: (Salvador et al., 2020)



2.2. Base Teórica

Los pilares que han permitido el desarrollo de la conducción autónoma en los últimos años son, entre otros: 1) los sensores de captación del entorno, 2) la construcción y publicación de dataset con datos recogidos de estos sensores durante sesiones de conducción, 3) las tecnologías de ETL para big data, 4) los simuladores para entrenamiento de agentes artificiales, y

5) las técnicas de inteligencia artificial que son usadas tanto para el entrenamiento de estos agentes inteligentes como para la segmentación semántica de imágenes. A continuación, se presente el estado del arte de los referidos elementos.

2.2.1. Sensores de captación del entorno

La captura de información del entorno físico demanda la utilización de sensores como LIDAR (light detection and ranging) cámaras y otros que, ensamblados en los vehículos y mediante sesiones de conducción, permiten construir conjuntos de datos correspondientes a los escenarios mapeados (Díez Ramírez, 2018). Estos mismos sensores son los utilizados posteriormente para la captura del entorno por el vehículo autónomo durante en el recorrido real.

A continuación, se describen de forma somera los diferentes tipos de sensores utilizados en conducción autónoma.

- Lidar:

Mediante un sistema de láseres y espejos el sensor lidar consigue generar un mapeado 3D del entorno de vehículo. La diferencia de tiempo entre la emisión del Laser desde el sensor y su retorno al mismo tras rebotar en objeto del entorno es usado para construir el mapa 3D.

La mayor ventaja de estos sensores es que no se ven influenciados por las condiciones meteorológicas del ambiente ni las condiciones lumínicas ya que la longitud de onda y la frecuencia del haz laser lo permite. Por otro lado, su coste y volumen son desventajas para su utilización en serie (Díez Ramírez, 2018).

Los sensores LIDAR utilizados para construir el dataset A2D2 de Audi AG® son del modelo Velodyne VLP-16 (Geyer et al., 2020)

Ilustración 4: Velodyne LIDAR data sheet. Fuente: (www.Velodynelidar.com,2023)

- Radar:

La diferencia de tiempo entre la emisión de ondas de radio desde el sensor y su recepción posterior tras rebotar en el objeto permite saber la distancia a objetos y la velocidad de los mismos en la dirección focalizada por el radar. Estos sensores permiten obtener información en 2D (posicionamiento) de objetos a larga distancia, pero no permite saber la altura de los mismos.

- Sonar:

Estos sensores emiten ultrasonidos y son capaces de detectar presencias de objetos a distancias cortas. El tiempo entre la emisión y el rebote permite detectar objetos cercanos y calcular su distancia.

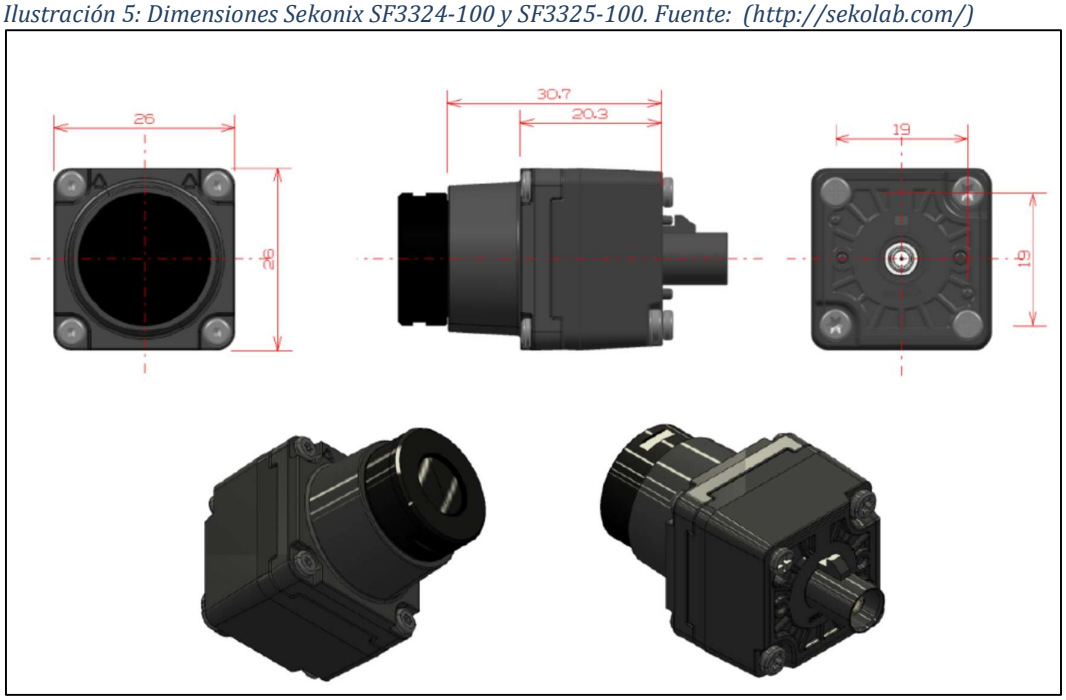
- Cámaras:

Las cámaras apoyadas por sistemas de visión artificial mediate algoritmos de redes neuronales son capaces de reconocer objetos, distancias, posiciones y trayectorias de líneas, límites de carriles y otros, haciendo de

estos sensores los más óptimos para la tarea de conducción autónoma en cuanto a su coste y aprovechamiento.

Las cámaras utilizadas para construir el set de Datos A2D2 de Audi AG® son del modelo Sekonix SF3325-100 para la cámara frontal y SF33224-100 para las demás (Geyer et al., 2020). Estas cámaras equipan un sensor RCCB acrónimo de Red Clear Blue que mejora la sensibilidad del frente a la iluminación exterior gracias a que utiliza el canal verde para interpretar la luminosidad del entorno y compensa el balance de color con una matriz de corrección de color para suplir la ausencia del verde y construir imágenes RGB (Imatest, 2023)

Ilustración 5: Dimensiones Sekonix SF3324-100 y SF3325-100. Fuente: (http://sekolab.com/)



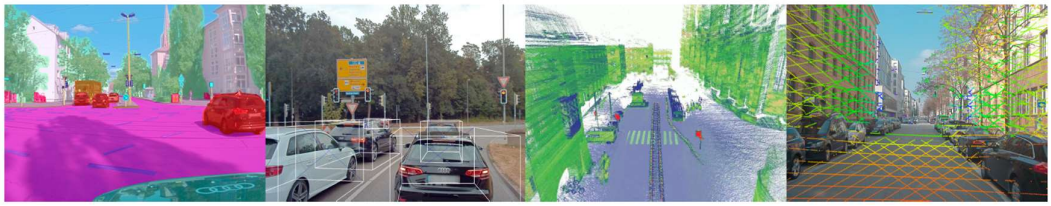
2.2.2. Dataset de entornos reales para la conducción autónoma

El conjunto de tipos de sensores descritos anteriormente se instala en vehículos y, tras la correspondiente calibración, se comienzan recorridos para capturar de forma síncrona la salida de todos ellos y, en algunos casos la salida del BUS de sensores del propio vehículo del vehículo. Una vez completados los recorridos se obtiene un set de datos que se utiliza

posteriormente para el entrenamiento de agente para la conducción autónoma.

En la actualidad existen numerosos sets de datos recogidos de entornos reales, cada uno con sus características propias dependiendo de las condiciones del ambiente en el que fueron recogidos, así como de los sensores utilizados. Estos datasets son utilizados para interpretar el entorno traduciéndolo en conceptos computacionalmente manejables como la segmentación semántica, las cajas de delimitación (boundary boxes), las nubes de puntos o imágenes, etc… (Geyer et al., 2020). La Ilustración 6, presenta un ejemplo de estos conceptos para facilitar su entendimiento.

Ilustración 6: Visualización de datos del dataset A2D2. Desde la Izquierda: Segmentación semántica, Cajas de limitación, nubes de puntos densa del sistema SLAM y nube de puntos solapado sobre una imagen. Fuente (Geyer et al., 2020)



Existen varios sets de datos de acceso público entre los cuales destaca el dataset de Audi AG® A2D2 por su completitud, explicabilidad y precisión (Geyer et al., 2020). A continuación, recogemos algunos de los dataset más importantes publicados por orden cronológico.

El Dataset KITTI, (Andreas Geiger, 2013) fue el primero en ser publicado y tubo un catálogo de comparaciones asociadas (Andreas Geiger, 2012). EL vehículo de captura de datos fue equipado con cuatro cámaras de video (2 a color y 2 en escala de grises), un Scanner LIDAR y un sistema de navegación. Gracias a él fue posible progresar en la selección de objetos 2D y 3D, seguimiento y mapeado simultaneo (SLAM) y otras tareas. En (Jens Behley, 2019) este dataset fue actualizado introduciendo la segmentación semántica en sus fotogramas y su catálogo de comparaciones asociadas añadiendo al dataset 43.000 nuevos mapeados

etiquetados. Como líneas de trabajo futuro, se destacó la importancia de utilizar sensores multimodales para la configuración de la captura de datos. El guante de esta propuesta fue recogido posteriormente por varios dataset como el A2D2 de Audi AG® .

El dataset Cityscapes (Marius Cordts, 2016) presenta la segmentación semántica de imágenes para 5000 fotogramas tomados en 50 ciudades alemanas que fueron etiquetados en 30 clases. Para ello, se utilizaron cámaras a color estéreo-pareadas para capturar las escenas que fueron etiquetadas tanto a nivel de pixeles como de instancias.

El dataset Mapillary Vistas (Gerhard Neuhold, 2017), proporcionó segmentación semántica para escenas de entornos rurales, urbanos y senderos. Contenía 25.000 imágenes anotadas para entornos urbanos de localizaciones alrededor del mundo. El dataset es heterogéneo puesto que las imágenes proceden de las capturas de teléfonos móviles, tables y modelos variados de cámaras.

El dataset Oxford Robocar (Maddern et al., 2017) fue capturado con un vehículo autónomo que recorrió une ruta en Oxford, Inglaterra 2 veces al día durante un año. Contiene sobre 1000Km de conducción grabada y casi 20 millones de imágenes. Los datos fueron capturados mediante 6 cámaras, sensores LIDAR y sistemas de navegación GPS. Siendo la variedad de escenarios reducida ya que la ruta no cambió a lo largo del año, si lo hicieron las condiciones meteorológicas y de luminosidad incluyendo situaciones extremas como fuertes lluvias, sol directo, nieve etc.

El dataset Drive360 (Hecker et al., 2018) incluye 60 horas de conducción en Suiza capturando datos de 8 cámaras distintas que son capaces de proporcionar una visión de 360º el entorno. El dataset se complementa con datos sobre para metros del vehículo como el ángulo de la dirección, velocidad del vehículo y planificación de la ruta.

El dataset Berkley Deep Drive (Fisher Yu, 2018) conocido como (BDD-100) contiene 100.000 imágenes con etiquetado de cajas de limitación, líneas y señales de tráfico, así como identificación de color de semáforos. Contiene además 10.000 imágenes con segmentación semántica a nivel de pixel. Este datase fue el primero en poner énfasis en las cajas de limitación por ello la gran cantidad de imágenes etiquetadas con esta característica.

El dataset ApolloScape (Peng Wang, 2019) contiene más de 140.000 fotogramas de video de varias localizaciones en china bajo diferentes condiciones meteorológicas. Proporciona segmentación semántica en 2D a nivel de pixel y segmentación semántica en 3D a nivel de punto para 28 clases de objetos. También contiene etiquetado en 2D para marcas lineales esenciales para la interpretación del entorno en tareas de conducción autónoma.

El dataset nuScenes (Caesar et al., 2019) consta de imágenes de cámaras, nubes de puntos de sensores LIDAR, datos de sensores RADAR, junto con anotaciones de cajas de limitación 3D combinado escenas de día y noche siempre en condiciones de meteorología de buena luminosidad. El formato de este dataset (nuScene) ha sido utilizado posteriormente en la construcción de otros dataset como The Lyft Level 5 Av (Kesten et al., 2019) que contiene datos de cámaras y sensores LIDAR focalizándose en el la detección de cajas de limitación 3D.

A medida que el interés sobre los dataset multimodales creció encontramos el dataset Waymo Open Dataset (Sun et al., 2019) en el que fueron publicadas 12 millones de anotaciones de cajas de límite 3D sobre nubes de puntos procedentes de sensores LIDAR y 1.2 millones de anotaciones de cajas de límite 2D procedentes de cámaras. Todo ello extraído de 1000 secuencias de 20 segundos procedentes de escenarios urbanos y de poblaciones de extrarradio bajo condiciones diversas de meteorología y luminosidad.

El 14 de abril de 2020, Audi AG® publica el dataset A2D2 (Geyer et al., 2020) y lo hace bajo licencia CCBY-ND 4.0 que es la menos restrictiva de todas permitiendo su uso incluso para fines comerciales. En la exposición del trabajo se describirá en detalle este dataset. A modo de resumen, el dataset tiene un tamaño de 2.3 Tb aproximadamente y consta de más de 40,000 fotogramas etiquetados para segmentación semántica (2D) y nube de puntos (3D) entre los cuales más de 12,000 fotogramas también incluyen etiquetados para cajas de limitación 3D. Además, contiene 390.000 fotogramas no etiquetados con sus correspondientes datos de sensores procedentes de varios recorridos en tres ciudades distintas (Gaimershein, Ingolstadt y Múnich). Para la captura de estos datos se utilizaron 6 cámaras (descritas anteriormente) y 5 sensores LIDAR además del bus de datos del vehículo para recoger información detallada y extensiva sobre 22 parámetros.

Se muestra en la Tabla 1 la comparación resumida de estos dataset mencionados anteriormente.

Tabla 1: Comparación de dataset. Fuente: (Geyer et al., 2020) *y elaboración propia.*

Dataset	***Cámaras***	***LIDAR***	***Datos BUS***	***zona***	***Horas***	***Clima***	***Objetos***	***Actual***
KITTI	4 (0.7 MP)	1 (64 canales)	GPS+IMU	urbano	día	soleado nublado	píxeles 3D, puntos 3D	2015
Oxford Robocar	1(1.3 MP) +3 (1.4 MP)	2 (7 canales) + 1 (16 canales)	GPS+IMU	Urbano	día y noche	Varios climas	3D	2017
Drive360	8(12 MP)	(N/A)	GPS+IMU	Urbano	día	Varios climas	3D	2018
Apollo Scape	2 (9.2MP)	2 (N/A)	GPS+IMU	urbano, varias ciudades	día	varios climas	3D	2018
nuScenes	6 (1.4MP)	1 (32 canales)	-	urbano, dos ciudades	día	varios climas	3D	2019
Lyft Level 5	7 (2.1MP)?	1 + 2 aux. (40 canales)	velocidad, velocidad angular	urbano	día	varios climas	3D, 2D	2019
Waymo OD	3 (2.5MP) + 2 (1.7MP)	1 + 4 aux. (64 canales)	GPS,IMU, freno, ángulo de dirección, acelerador, edometría, velocidad, inclinación, balanceo	urbano	día, noche	varios climas	3D, píxel	2019
A2D2	6 (2.3MP)	5 (16 canales)	GPS, IMU, velocidad, velocidad angular	urbano, autopistas, carreteras rurales, tres ciudades	día, noche	varios climas	3D	2020

2.2.3. Simuladores utilizados para la conducción autónoma

El entrenamiento de los sistemas inteligentes en entornos reales se encuentra con dificultades relevantes derivadas de la seguridad de los individuos ajenos a las pruebas, así como el coste que conllevan (Pan et al., 2017). Por este motivo la utilización de simuladores está muy extendida ya que permiten entrenar y probar los agentes en entornos de bajo coste que, además, evitan los problemas de seguridad derivados de actuar en entornos reales.

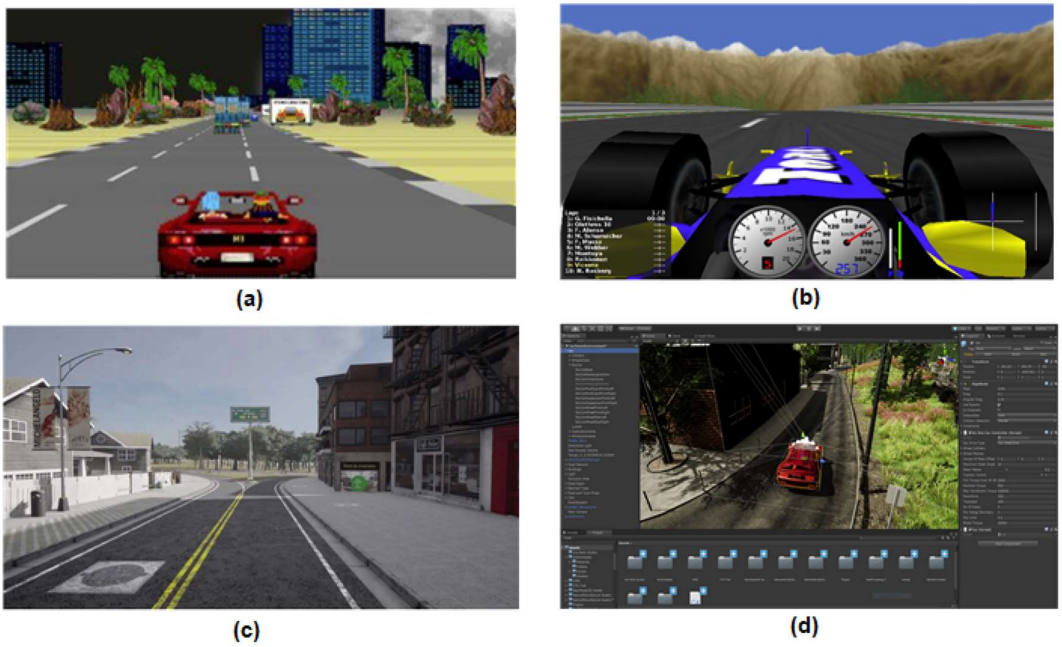
La utilización de simuladores, conlleva la posterior validación de los resultados obtenidos en entornos reales de forma que no es una técnica que reemplace los entornos reales si no que permite su mejor aprovechamiento, así como una mayor seguridad durante la ejecución de las pruebas ya que parte de agentes ya entrenados.

Existen dos categorías principales de simuladores, los realistas y los irreales. En el segundo tipo encontramos Javascript racer utilizado en (Yu et al., 2016) o el juego de carreras TORCS (The Open Racing Car Simulator) que ha ido utilizado con asiduidad como herramienta de investigación para la conducción autónoma como por ejemplo (Zhiqing Huang, 2019) o también (Sallab et al., 2017)

La utilidad de los simuladores para el entrenamiento de agentes para la conducción autónoma provocó el desarrollo de simuladores realistas. Los más importantes son CARLA (Dosovitskiy et al., 2017) y AirSim (Shah et al., 2017). CARLA es un simulador de código abierto para entornos urbanos de alta y baja densidad y rurales mientras que AirSim está construido sobre el motor Unreal® y ofrece variedad de entornos y condiciones siendo posible su utilización para también para drones. Los resultados de agentes de conducción autónoma, así como la idoneidad del simulador pueden verse en (Xiaodan Liang et al., 2018) mientras que AirSim ha sido usado satisfactoriamente en trabajos como (Xinle Liang et al., 2019) y (Roy Amante Salvador, 2019)

A título informativo, se muestran en la Ilustración 7 escenarios de los cuatro simuladores mencionados anteriormente.

Ilustración 7: Simuladores utilizados para trabajos de conducción autónoma. (a) Javascript racer (https://game-central.in/), (b) TORCS (http://www3.uji.es/), (c) CARLA (https://carla.org/), (d) AirSim. Fuente: (https://www.microsoft.com/en-us/research/blog/microsoft-airsim-now-available-on-unity/).



Los simuladores han permitido aumentar de forma exponencial la cantidad de situaciones y escenarios disponibles para entrenar los agentes inteligentes, pero aun así no es realista reproducir todas las combinaciones posibles en un escenario de conducción real con lo que aun siendo una gran contribución no permiten resolver por sí mismos el reto de la conducción autónoma (Le Mero et al., 2022). En cuanto a la transposición de los agentes entrenados con simuladores y técnicas de deep learning a entornos reales, se están haciendo avances como se recoge en (Müller et al., 2018) y (Yang et al., 2018).

2.2.4. Descripción de las tecnologías de ETL para big data

La cantidad y complejidad de los entrenamientos y pruebas a los que se deben someter los agentes inteligentes conllevan la necesidad de tratar con cantidades ingentes de datos cuyas estructuras son diferentes dependiendo de la fuente de captura de la que procedan. Esto datos se

deben extraer, transformar y cargar (ETL) a alta velocidad. Se hace patente que el campo de la conducción autónoma cumple con las premisas para la aplicación de tecnologías Big Data dado el **volumen** y **variedad** de los datos, así como la **velocidad** necesaria para la captura transformación y carga de los mismos.

En la actualidad existen múltiples herramientas para el ETL de datos masivos. Todas ellas tienen su origen en el sistema de archivos distribuido GFS (Google File System) presentado en 2003 (Ghemawat et al., 2003). En ella se explicó en primicia la idea de utilizar entornos de ordenadores convencionales conectados entre sí, formando clústeres de ordenadores, para poder almacenar archivos de tamaño mayor de un disco duro. El concepto y la base del sistema de archivos HDFS que permite distribuir los datos entre distintos nodos de un clúster de ordenadores gestionando la distribución y redundancia de forma transparente. Esto permite al desarrollador abstraerse de las actividades relacionadas con la gestión de archivos de tamaño mayor a un disco duro. Posteriormente, en 2004, Google publicó el modelo de programación MapReduce (Dean & Ghemawat, 2004) que permite procesar en paralelo archivos almacenados en sistemas GFS, así como una biblioteca de programación en código abierto para implementarlo. Al igual que HDFS permite abstraer al desarrollador del almacenamiento de dato en entornos distribuidos, MapReduce permite una simplificación de todos los detalles de hardware, redes y comunicaciones necesarios para el procesamiento de los datos de archivos almacenados en entornos distribuidos. Tomando map reduce como base, se construyó un ecosistema de aplicaciones de código abierto que se conoce como el ecosistema Hadoop.

En 2010, Matei Zaharia (Zaharia et al., 2010) publica una nueva tecnología de código abierto llamada Apache Spark que ha reemplazado a MapReduce prácticamente en su totalidad como motor de ejecución en las aplicaciones de procesamiento de datos.

Hoy en 2023, la tecnología Apache Spark ha dado lugar a herramientas de Apache que constituyen el estándar tecnológico del big data. Estas herramientas son:

HDFS (Hadoop Distributed File System): sistema de archivos distribuido inspirado en el GFS de Google, que permite distribuir los datos entre distintos nodos de un clúster, gestionando la distribución y la redundancia de forma transparente para el desarrollador que vaya a hacer uso de esos datos.

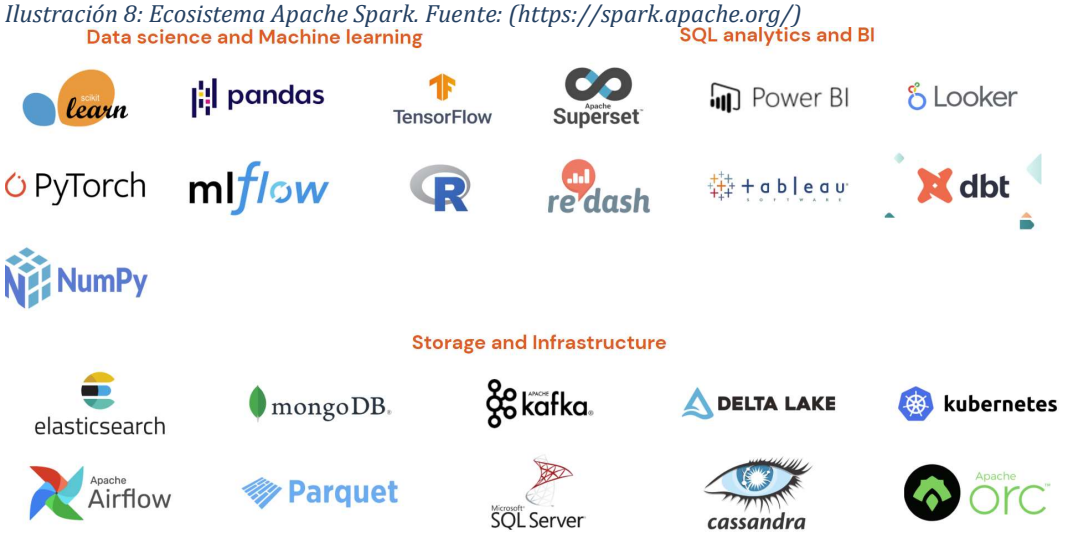
Apache Hive: herramienta para acceder mediante sintaxis SQL a datos estructurados que están almacenados en un sistema de archivos distribuido, como HDFS u otros similares. Las consultas SQL son traducidas automáticamente a trabajos de procesamiento distribuido según el motor que se haya configurado, que puede ser Apache Spark.

Apache Spark: motor de procesamiento distribuido y bibliotecas de programación distribuida de propósito general, que opera siempre en la memoria principal (RAM) de los nodos del clúster.

Apache Kafka: plataforma para manejo de eventos en tiempo real, que consiste en una cola de mensajes distribuida y masivamente escalable sobre un clúster de ordenadores. Estos mensajes pueden ser consumidos por uno o varios procesos externos (por ejemplo, trabajos de Spark).

Apache Spark se integra con prácticamente todos los marcos de trabajo creándose un ecosistema que permite procesar datos masivos con todas las herramientas existentes para el big data.

Ilustración 8: Ecosistema Apache Spark. Fuente: (https://spark.apache.org/)



2.2.5. Técnicas de Inteligencia artificial para conducción autónoma

El otro pilar que ha permitido evolucionar la conducción autónoma en los últimos años han sido las nuevas técnicas de inteligencia artificial, particularmente las relacionadas con deep machine learning y deep reinforcement learning.

Específicamente para la tarea de conducción, se utiliza la técnica de aprendizaje denominada aprendizaje por imitación o clonación del comportamiento (Torabi et al., 2018). El concepto de este tipo de aprendizaje se basa en el entrenamiento imitando las acciones de conducción de un experto distinguiéndose dos categorías la clonación del comportamiento y el aprendizaje por refuerzo inverso (reinforcement learning).

En la clonación del comportamiento el sistema recibirá datos con imágenes y parámetros del vehículo asociadas a acciones del experto. El modelo de aprendizaje profundo, se configura como clasificador o regresor con el objetivo de aprender a reconocer los patrones de asociación de las entradas al sistema con las acciones del experto. La principal ventaja del aprendizaje por imitación es que permite imitar al experto sin necesidad de actuar con el entorno ya que los resultados del entrenamiento de la red

neuronal pueden ser evaluables a través de la función de pérdida. Sin embargo, estas técnicas están limitadas por la complejidad de la conducción autónoma y el propio concepto de aprendizaje por imitación que hace imposible replicar todas las situaciones y ambientes a las que un vehículo de conducción autónoma podría enfrentarse. Esta limitación se hace patente cuando utilizamos los resultados del entrenamiento de los modelos en ambientes y situaciones que difieren significativamente de los contenidos en los dataset. Además, como ya se ha mencionado, un dataset finito, por muy extenso y completo que sea, no puede contener todas las posibles combinaciones de escenarios y ambientes que puedan producirse en la conducción de vehículos. Este hándicap es crítico a la hora de generalizar la aplicación de estos modelos.

Los algoritmos más interesantes dentro de las técnicas de aprendizaje por imitación aplicable a la conducción autónoma son las redes neuronales convolucionales (CNN) (O'Shea & Nash, 2015). Estas CNN se utilizan tomando como entrada imágenes de carreteras capturadas por las cámaras instaladas en el vehículo y como etiqueta la acción que realiza el experto en el entorno capturado por la imagen. Estas etiquetas de acciones son generalmente la captura de los sensores del BUS del vehículo para los accionadores del conductor como son la posición del acelerador, la actuación sobre los frenos, o la posición del volante. Se puede ver esta aplicación en el estudio (Wu et al., 2016).

Las CNN permite identificar las características de la carretera, así como objetos en ella como obstáculos, otros vehículos y peatones. También se utilizan las CNN como extractores de características de imágenes para después utilizarlas como base para un entrenamiento de reinforcement learning. Con este enfoque, permitimos que ese entrenamiento por reinforcement learning se realice más rápido y más seguro (en caso de realizarse en entorno real).

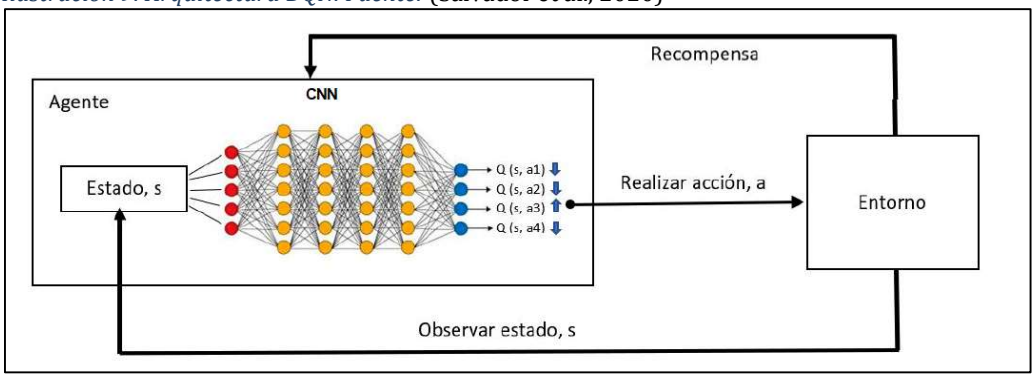
Además de las CNN, las denominadas redes neuronales recurrentes (RNN) también son utilizadas para el aprendizaje por clonación del comportamiento. Este tipo de redes neuronales procesan un elemento de la secuencia de entrada cada vez manteniendo en sus unidades ocultas un vector estado que contiene implícitamente información sobre la historia de todos los elementos de la secuencia pasados por la red. La arquitectura de RNN denominada long-short term (LSTM) es la mejor para el caso de redes profundas como las necesarias para la tarea de conducción autónoma y que al mismo tiempo tienen series secuenciales muy largas gracias a sus tres puertas que controlan el estado de la entrada la salida y la memoria (Salvador et al., 2020). Las LSTM se utilizan en conducción autónoma para para la predicción eficiente de trayectorias de vehículos y peatones como se muestra en (Kim et al., 2017). La predicción de trayectorias permite aumentar el grado de seguridad en entonos dinámicos o de alta densidad de agentes interactuando con el vehículo.

De forma paralela a la clonación del comportamiento, han sido utilizadas técnicas de Deep learning alternativas como el Reinforcement Learning. En esta técnica, el agente aprende patrones de actuación gracias a la experiencia de interacción con el entorno en la que, dependiendo de las acciones que ejecute, recibe recompensas positivas o negativas. Al contrario que el aprendizaje por imitación, el agente no recibe como entrada datos etiquetados de los que aprender, si no que el aprendizaje se basa en una función de recompensa que indica al agente si sus acciones son adecuadas o no. Es pues un concepto basado en prueba y error en el que el agente adapta sus acciones y estrategias en base a la recompensa de la función.

La unión de las dos técnicas explicadas anteriormente deep learning y reinforcement learning ha dado lugar a una técnica llamada deep reinforcement learning o también conocida por deep Q learning en la que los algoritmos deep Q networks están teniendo buenos resultados en las tareas de conducción autónoma como se refleja en (Mnih et al., 2015) y

en (Okuyama et al., 2018). Está técnica unifica los buenos resultados de las redes neuronales convolucionales CNN para extraer características de imágenes (que permiten clasificar e identificar objetos) y lo combina con el enfoque de prueba y error del reinforcement learning. El algoritmo deep Q networks (DQN) es el principal exponente de esta técnica. La salida del algoritmo no son clases sino valores Q, que se asignan a las acciones que el agente ha aprendido en función del entorno identificado por la entrada. El objetivo del algoritmo DQN es encontrar los pesos óptimos de la red neuronal que permita maximizar el valor Q en cada par estado-acción que haya recibido una recompensa positiva (Salvador et al., 2020). El concepto de este algoritmo se puede observar en la Ilustración 9. Como se expone en (Okuyama et al., 2018), se han obtenido buenos resultados aplicando el algoritmo DQN por lo que está siendo muy utilizado en la tarea de la conducción autónoma.

Ilustración 9: Arquitectura DQN. Fuente: (Salvador et al., 2020)



Como ha sido señalado en (Salvador et al., 2020), los modelos utilizados en el enfoque end-to-end para la conducción autónoma, son basados en algoritmos de machine learning como el Deep Reinforcement Learning. Este tipo de algoritmos, por definición, aprenden para ser computacionalmente eficientes. El pre-entrenamiento de redes neuronales para interpretar el entorno (como la segmentación semántica de imágenes) permite concentrar el aprendizaje del agente usando el algoritmo de Deep Reinforcement Learning y optimizar la eficiencia computacional.

Como se ha avanzado, la segmentación de imágenes es una parte importante de los sistemas de percepción visual en los vehículos autónomos. Consiste en la clasificación de las diferentes pixeles de la imagen en segmentos con nivel de significancia. La tarea de segmentación semántica se clasifica en tres categorías principales (Cakir et al., 2022).

La segmentación semántica busca asignar una etiqueta con alto nivel de significancia a cada uno de los pixeles que constituyen la imagen. Mediante esta técnica es posible conocer que elementos de significación relevante existen en la imagen como pudieran ser, en el contexto de la conducción autónoma, personas, coches, camiones, barreras, señales, líneas de limitación de carriles y/o arcenes.

La segmentación de instancias va un paso más allá identificando cada instancia de los objetos presentes en la imagen y delimitando su contorno en un boundary box (caja de delimitación). Esta técnica permitiría la distinción de cada vehículo individual en una imagen en la que se presentan superpuestos.

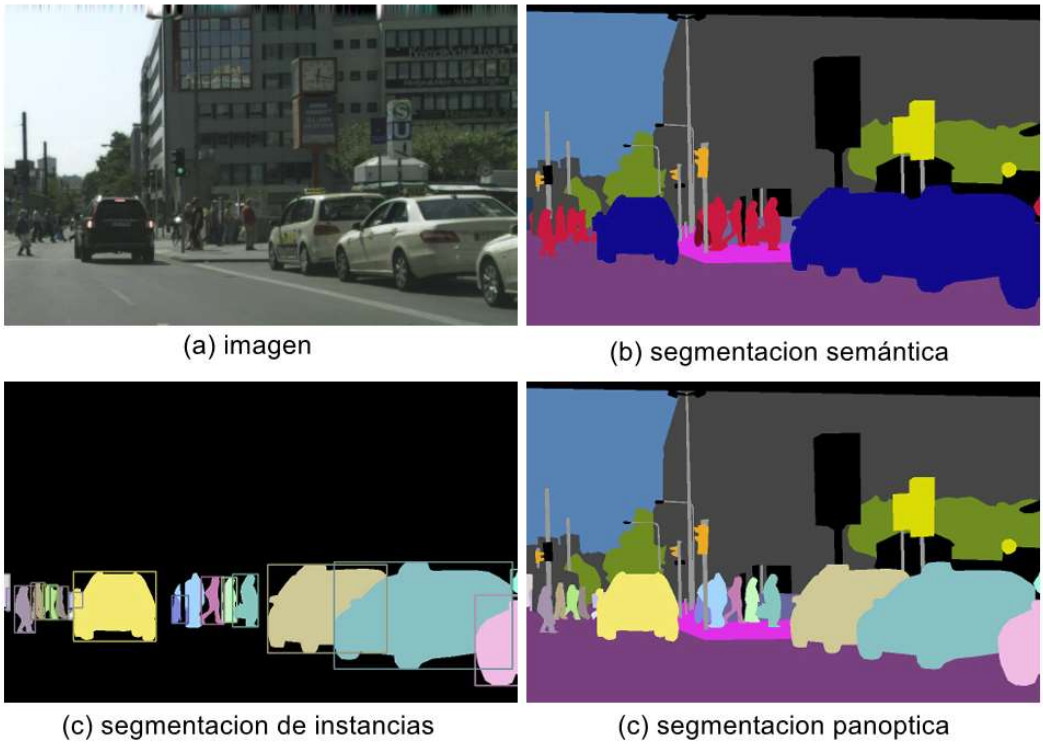
La segmentación panóptica unifica la segmentación semántica y la segmentación de instancias para generar una segmentación coherente de la escena completa incluida en la imagen aportando un nivel más de detalle y riqueza en el significado de la misma (Kirillov et al., 2018).

Para una mejor comprensión se muestra un ejemplo de estas técnicas en la Ilustración 10.

El objetivo último de una segmentación de imágenes radica en conseguir un alto grado de precisión en la distinción de los diferentes segmentos e instancias a la par que permita el procesamiento en tiempo real de imágenes debido a que las imágenes se toman en vivo con las cámaras del vehículo. Es evidente que cuanto más rápida sea la red neuronal encargada de ejecutar la segmentación semántica, más imágenes por segundo (FPS) será capaz de procesar. La evaluación de la precisión en la inferencia requiere, sin embargo, de métricas específicas se explica más

adelante. La red neuronal que sea capaz de alcanzar un valor FPS compatible con el procesamiento en tiempo real por parte del sistema de conducción autónoma con la máxima precisión en la inferencia será la más idónea para la tarea (Cakir et al., 2022)

Ilustración 10: Segmentación semántica y segmentación de instancias. Fuente: (Kirillov et al., 2018)



Se señala en (Minaee et al., 2020) son numerosos y muy variados los algoritmos que han sido desarrollados para la segmentación semántica los primeros métodos fueron basados en umbrales (Otsu, 1979) o minimización de funciones continuas (Najman & Schmitt, 1994). En los últimos años la irrupción de los modelos de aprendizaje profundo ha supuesto un cambio de paradigma en este campo consiguiendo mejoras notables en el rendimiento.

Los modelos de aprendizaje profundo se apoyan en arquitecturas de redes neuronales. Estas arquitecturas son combinadas y conectadas entre sí con el objeto de optimizar los valores de las funciones de perdida en cada iteración del dataset durante etapa de aprendizaje (Minaee et al., 2020).

2.2.6. Arquitecturas de redes neuronales utilizadas para segmentación semántica

Se exponen a continuación las arquitecturas de redes neuronales más importantes usadas para la segmentación semántica.

- Redes neuronales convolucionales (CNNs) (Lecun et al., 1998).

Este tipo de redes neuronales son las más usadas en modelos de deep learning para visión artificial ya que son las que arrojan un mejor resultado en este tipo de tareas (Minaee et al., 2020). Las CNNs constituyen la arquitectura de los modelos de aprendizaje profundo U-Net y ResNet que se utilizan en este trabajo.

Inicialmente propuestas por (Fukushima, 1980) en su seminario Neocogniron, fue (Waibel et al., 1989) quien introdujo las CNNs con pesos compartidos que, entrenadas bajo el algoritmo de backpropagation, fueron capaces de reconocer fonemas. (Lecun et al., 1998) desarrollo una arquitectura CNN para el reconocimiento de documentos. Como puede apreciarse, hablamos de hitos relevantes en los campos del reconocimiento de la voz humana y la digitalización de textos.

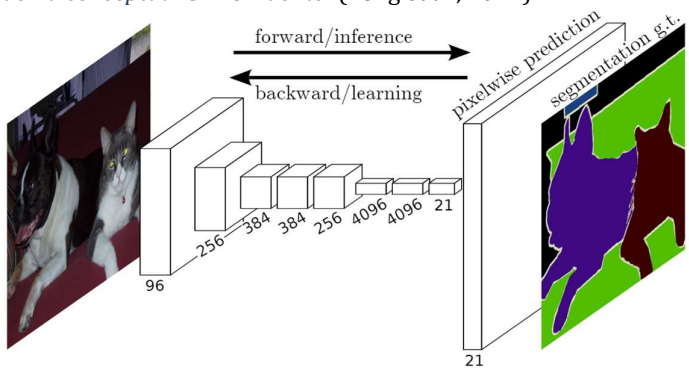
Las redes neuronales convolucionales consisten en tres tipos de capas:

  - La capa convolucional donde un filtro de pesos (conocido por su equivalente en inglés kernel) se pasa por áreas de pixeles de la imagen para extraer las características de dichas áreas. El resultado de estas capas es un mapa de características
  - Las capas no lineales, que aplican una función de activación a los mapas de características anteriores para activar, o no, el modelado de los mismos por la red.
  - Las capas de agrupación (pooling layers), reemplazan la información de cada elemento del mapa de características por alguna medida estadística (media, máximo,…) que representará el elemento del mapa de características para

reducir la resolución y sintetizar, paso a paso, paso la información contenida en el fotograma.

Cada unidad de capas CNN está conectada con las capas anteriores deforma que dada unidad recibe los pesos de la unidad anterior. Apilando capas para formar una pirámide multidimensional cuyas capas más altas aprenden características procedentes de capas que acumulan información a cada nivel. (Long et al., 2014) nos muestra sintéticamente en la Ilustración 11 como una red neuronal convolucional reduce la resolución de la imagen a cambio de incrementar la dimensionalidad del mapa de caracteres en una tarea de segmentación semántica.

Ilustración 11: Esquema conceptual CNNs. Fuente: (Long et al., 2014)



Algunas de los modelos más conocido que utilizan CNNs son: AlexNet (Krizhevsky et al., 2012), VGGNet (Simonyan & Zisserman, 2014), ResNet (He et al., 2015), GoogLeNet (Szegedy et al., 2014), MobileNet (Howard et al., 2017) , U-Net (Ronneberger et al., 2015), DenseNet (Howard et al., 2017).

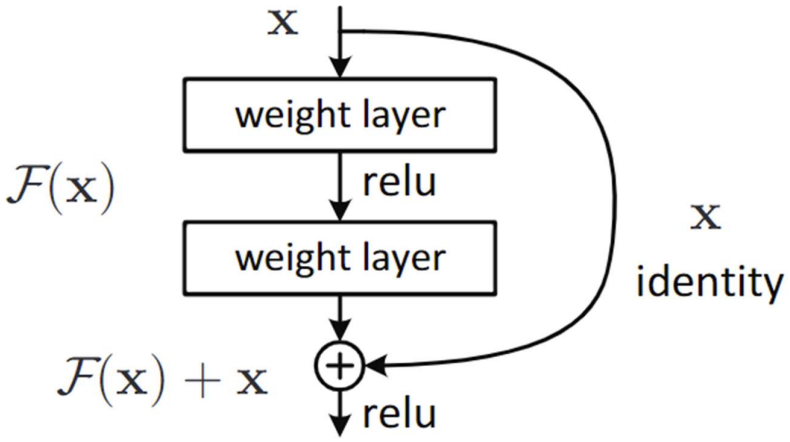
Se profundiza ahora en la arquitectura ResNet utilizada como codificador para la experimentación del dataset A2D2 de Audi AG® utilizado en este trabajo fin de master.

La idea clave detrás de ResNet radica en resolver el problema del desvanecimiento del gradiente, que ocurre cuando las redes neuronales son demasiado profundas y los gradientes se vuelven demasiado pequeños para actualizar los pesos correctamente durante el

entrenamiento. Para mitigar este problema. He et al., en 2015 introdujeron conexiones residuales en la arquitectura. Estas conexiones permiten que la información fluya directamente desde una capa a otra sin ser alterada, lo que significa que las capas más profundas tienen acceso tanto a las características de las capas anteriores como a las nuevas características que están extrayendo.

La unidad básica de construcción en ResNet es el bloque residual. Este bloque consta de dos capas convolucionales seguidas de una conexión residual que suma la entrada original a la salida de estas capas. Esta estructura permite el entrenamiento de redes con cientos o incluso miles de capas sin sufrir una degradación del rendimiento (He et al., 2015). El esquema de esta unidad básica de construcción se muestra en la Ilustración 12.

Ilustración 12: Unidad elemental de construcción de una red residual ResNet. Fuente:(He et al., 2015)

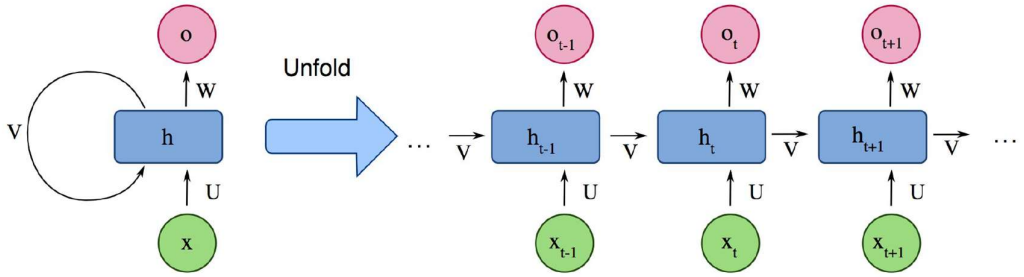


ResNet también viene en diferentes variantes, como ResNet-18, ResNet-34, ResNet-50, ResNet-101 y ResNet-152, que se diferencian en la cantidad de capas y la complejidad de la arquitectura.

- Redes neuronales recurrentes (RNNs) y redes neuronales recurrentes con memoria a corto y medio plazo (LSTM) (Hochreiter & Schmidhuber, 1997)

Las redes neuronales recurrentes se usan principalmente para el proceso de datos secuenciales como conversaciones, textos, videos u series de tiempo. Son tareas en las que los datos de una determinada marca de posición y/o tiempo dependen de los datos previamente procesados. A cada marca de tiempo, la red neuronal recolecta la entrada de la marca actual y el estado (hidden state) del paso anterior. La salida es el valor objetivo y un nuevo estado para la marca actual que será usado en por la siguiente marca de tiempo (Rumelhart et al., 1986). Es lo que llamábamos anteriormente vector de estado que se va construyendo a medida que se para de una marca de tiempo a la siguiente. Se muestra en la Ilustración 13 una representación esquemática de lo descrito.

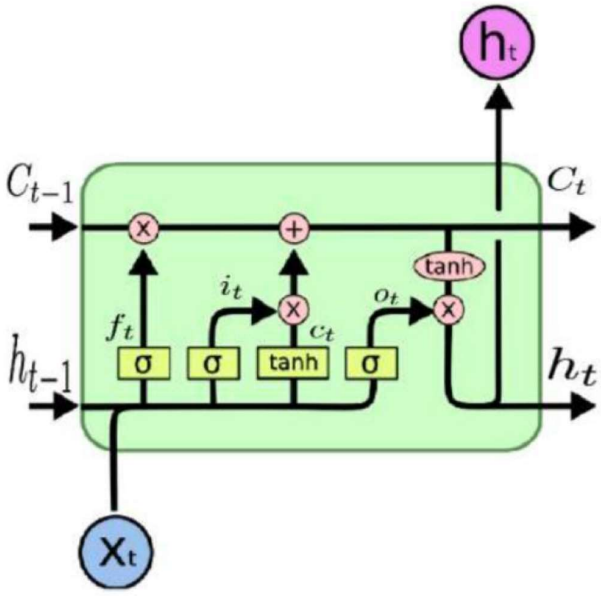
Ilustración 13: Arquitectura de un red neuronal recurrente simple. Fuente:(Minaee et al., 2020)



Las redes neuronales recurrentes, entre otras limitaciones, presentan un problema con el recuerdo de los estados en las secuencias largas de tiempo puesto que el vector de estado solo realimenta el estado de la etapa anterior que, aunque teóricamente debería contener el estado de las marcas de tiempo anteriores, la ponderación de las etapas más lejanas en el vector de estado disminuye a medida que se incorporan los estados de las etapas nuevas. Existe un tipo de redes neuronales recurrentes, denominado redes de memoria a corto y largo plazo (long sort term memory, LSTM) que están diseñadas para mitigar estas limitaciones (Hochreiter & Schmidhuber, 1997). La arquitectura LSTM, que se muestra en la Ilustración 14, incluye tres puertas (puerta de entrada, puerta de salida y puerta de olvido), que regulan el flujo de información hacia dentro

y hacia afuera de la celda de memoria, la cual almacena valores en intervalos arbitrarios de tiempo.

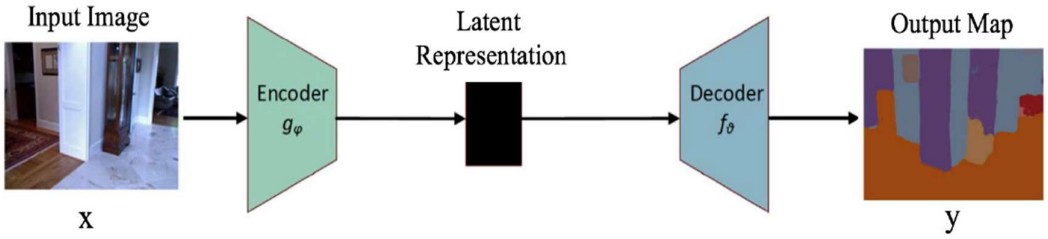
Ilustración 14: Arquitectura de una LSTM. Fuente: (Minaee et al., 2020)



- Codificación – Decodificación. (Badrinarayanan et al., 2017)

Este tipo de arquitecturas aprenden a mapear mapas de puntos de un dominio de entrada (imagen) a un dominio de salida (imagen segmentada) mediante una red neuronal en 2 etapas. La etapa de codificación comprime el domino de entrada en un espacio latente (latent space) que está formado por un vector de características que es capaz de capturar la información semántica de la imagen. La etapa de decodificación predice la imagen segmentada a partir de la información contenida en el espacio latente. Se muestra una representación esquemática de la arquitectura de codificador-decodificador en la Ilustración 15.

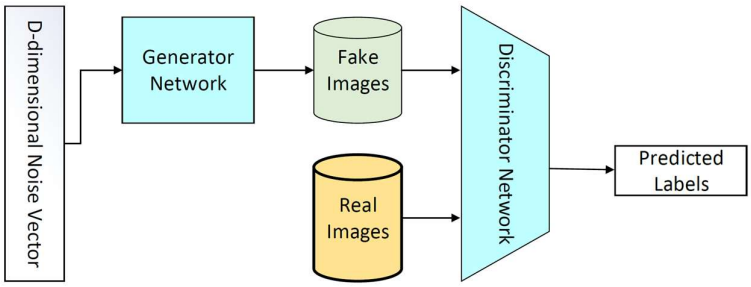
Ilustración 15: Arquitectura de un codificador-decodificador. Fuente: (Minaee et al., 2020)



- Redes generativas antagónicas (GANs). (Goodfellow et al., 2014)

Las redes generativas antagónicas consisten en 2 redes neuronales que trabajan para conseguir objetivos contrapuestos. Mientras que la red generativa aprende a mapear desde el ruido para conseguir la imagen más parecida a la muestra, la red discriminativa intenta distinguir la imagen de muestra de las generadas por la red generativa. Se considera las GAN como un juego de minimax[2] entre la red generativa y discriminativa. La red discriminativa intenta minimizar su error de clasificación al distinguir las imágenes generadas de la real, lo que significa maximizar la función de perdida. La red generativa intenta maximizar el error de clasificación lo que implica minimizar la función de pérdida. La Ilustración 16 muestra el esquema conceptual de una red generativa antagónica.

Ilustración 16: Arquitectura de una red neuronal generativa antagónica. Fuente: (Minaee et al., 2020)



2.2.7. Modelos de aprendizaje utilizados para segmentación semántica

Con las arquitecturas de redes neuronales descritas han sido propuestos cientos de modelos de aprendizaje profundo para la segmentación de imágenes. Con el objeto de ofrecer una perspectiva de los trabajos realizados se citan los diferentes tipos de modelos, en función de las

[2] Aunque existen evidencias de que Charles Babbage ya había trabajado antes sobre una idea similar, fue el matemático francés Émile Borel el primero en ofrecer en 1921 un tratamiento riguroso a los juegos competitivos y en estudiar las estrategias aplicables a los juegos de suma cero. Sin embargo suele atribuirse a John von Neumann el principal mérito de la concepción del principio minimax, ya que fue él quien, en su artículo de 1928 «Zur Theorie der Gesellschaftsspiele» («Sobre la teoría de los juegos de sociedad») publicado en la revista Mathematische Annalen, puso las bases de la moderna teoría de juegos y probó el teorema fundamental del minimax, por el que se demuestra que para juegos de suma cero con información perfecta entre dos competidores existe una única solución óptima.

arquitecturas de redes neuronales utilizadas, para centrar el foco en los modelos del tipo codificador-decodificador y de tipo pirámide, Entre estos modelos se encuentran el modelo U-Net, el modelo PsPNet. Estos 2 modelos son los referidos en el presente trabajo fin de master.

- Modelos de redes convolucionales conectadas o Fully Convolutional Networks (FCN)

Este tipo de modelos usa únicamente redes convolucionales para producir una imagen segmentada del mismo tamaño que la imagen original (Long et al., 2014). Este tipo de modelos son efectivos y populares, pero presentan la limitación de no ser lo suficientemente rápidos para poder predecir segmentaciones de imágenes en tiempo real además de no considerar la información global de la imagen de forma eficiente (Minaee et al., 2020).

- Modelos convolucionales con modelos gráficos.

Las redes FCN ignora el contexto de la escena que podría ser potencialmente útil. Para integrar más contexto, muchos enfoques incorporan modelos probabilísticos gráficos. (Chen et al., 2014) mostraron como las capas finales de las redes convolucionales no son lo suficientemente precisas para la segmentación y, como solución, combinaros la respuesta de las capas finales de las CNNs con un modelo gráfico probabilístico como el CRF (Conditional Random Fields). Los resultados de su solución fueron mejores que los modelos FCN a la hora de determinar los bordes de los distintos segmentos de la imagen.

- Redes convolucionales regionales (R-CNN)

Una de las extensiones de este tipo de redes, faster R-CNN (Ren et al., 2015) detecta objetos mediante la propuesta de áreas de objetos candidatas (regiones de interés, cuyo acrónimo en inglés es RoI) las cuales son computadas por una capa de agrupación (RoIPool) para inferir

la caja de límites que contendrá el objeto a localizar. Este tipo de redes se utilizan para la segmentación de instancias. (Minaee et al., 2020)

- Redes convolucionales con espaciamiento y la familia DeepLab

Este tipo de redes añade un parámetro a las CNN llamada factor de espaciamiento (dilating rate). Simplificando al máximo su significado, el factor de desplazamiento sería el número de pixel que se intercalan entre los pixeles de los cuales extraeremos los pesos. Por ejemplo, para un filtro (kernel) 3x3 con un factor de espaciamiento de 2 se mapearía una área de 5x5 extrayendo únicamente 9 parámetros. Esta metodología permite ampliar el área de inspección sin incrementar el coste computacional minimizando la pérdida de información de la imagen. Este tipo de redes son especialmente populares en el campo de la segmentación de imágenes en tiempo real (Minaee et al., 2020). Una de las más importantes es la familia DeepLab (Chen et al., 2016).

- Modelos basados en redes neuronales recurrentes.

La naturaleza de las redes neuronales recurrentes (RNN) permiten ligar cada pixel con sus vecinos y procesados secuencialmente para modelar el contexto global de la imagen y mejorar la segmentación semántica. (Visin, Ciccone, et al., 2015) propuso el modelo ReSeg basado en RNN para segmentación semántica basado en un modelo anterior (ReNet) (Visin, Kastner, et al., 2015) utilizado para clasificación de imágenes. Por su naturaleza secuencial, los modelos de segmentación semántica basados en RNN son más lentos que los basados en CNNs ya que la computación secuencial no puede ser fácilmente paralelizada (Minaee et al., 2020).

Otros modelo utilizados para la segmentación semántica serían los modelos basados en mecanismos de atención (Chen et al., 2015), modelos basados en redes generativas antagónicas (GANs) (Luc et al., 2016).

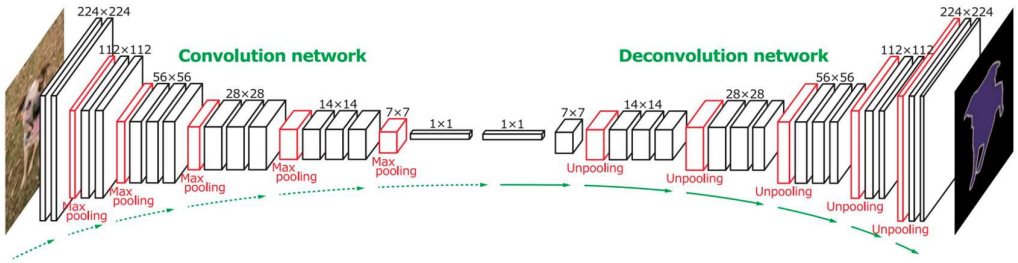
- Modelos codificador - decodificador.

Nos centramos ahora en los modelos de tipo codificador-decodificador al que pertenece el modelo U-NET que se usa en este trabajo fin de master.

Este tipo de modelos son ampliamente usados para la segmentación de imágenes en general a demás ha surgido una especialización de los mismos orientada aplicaciones médicas.

El primer aporte a la tarea de segmentación semántica utilizando un modelo de codificación/decodificación fue realizado por (Noh et al., 2015). Su modelo consistía en dos partes, un codificador compuesto por capas convolucionales heredadas del modelo VGG 16 (Simonyan & Zisserman, 2014) y una etapa de decodificación, que toma el vector de características resultante del decodificador como entrada, para generar un mapa de probabilidades para las clases a asignar a cada pixel. La etapa de decodificación la componen capas de-convolucionales y de des-agrupación que identifican las etiquetas de clase y predicen los segmentos que formarán la imagen semánticamente segmentada. La figura Ilustración 17 nos muestra el concepto de este modelo.

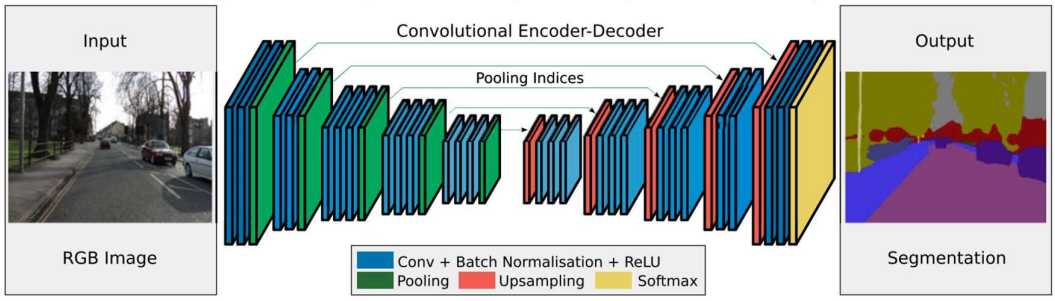
Ilustración 17: Segmentación semántica mediante convolución-de-convolución. Fuente: (Simonyan & Zisserman, 2014)



En (Badrinarayanan et al., 2015) se aportó una novedosa variación anterior modelo mediante la utilización de los índices de agrupación computados en la etapa de codificación para realizar un re-escalado no lineal en la etapa de decodificación. Se muestra el concepto de este modelo en la Ilustración 18. Esta novedad elimina la necesidad de aprender para realizar el re-escalado en la etapa de decodificación. El

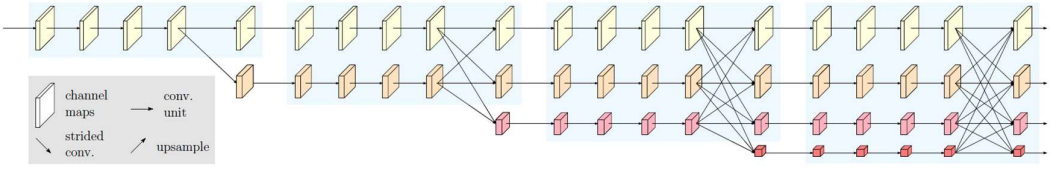
modelo resultante fue bautizado como SegNet y la principal ventaja es que el número de parámetros es significativamente menor que en otros modelos de la misma categoría (Minaee et al., 2020). Los autores del modelo SegNet realizaron una mejora del mismo denominada SegNet bayesiana en la que mejoraban la incertidumbre del modelo anterior en la predicción de los segmentos de la imagen (Kendall et al., 2015).

Ilustración 18: Concepto Modelo SegNet. Fuente: (Badrinarayanan et al., 2015)



Una de las principales limitaciones de los modelos de codificación-decodificación es la pérdida de información granular de la escena. Esto es debido a la pérdida de resolución durante la etapa de codificación ,como hemos comentado en la descripción de las CNNs. Para mitigar esta limitación han surgido modelos como el HRNet (K. Sun et al., 2019) el cual en lugar de re-escalar las imágenes tras la codificación, mantiene la escala original durante la codificación conectando los flujos de alta a baja resolución en paralelo e intercambiando repetidamente la información a través de las resoluciones en cada etapa. Se muestra el concepto de este modelo en la Ilustración 19.

Ilustración 19: Concepto modelo HRNet. Fuente: (K. Sun et al., 2019)



La aplicación para la segmentación de imágenes biomédicas de este tipo de modelos ha sido una de las más importantes dando lugar a modelos especializados para este ámbito que, debido a su éxito han sido usados

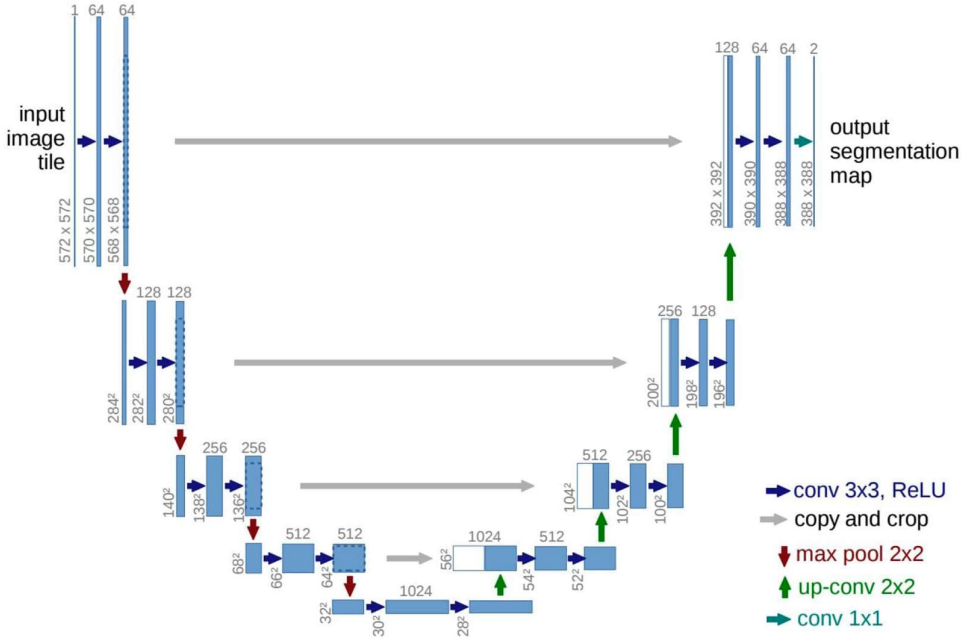
en otros campos. Es el caso de los modelos U-Net (Ronneberger et al., 2015) y V-Net (Milletari et al., 2016).

(Ronneberger et al., 2015) propuso el modelo U-Net para la segmentación de imágenes microscópicas. El modelo U-Net consta de una etapa de compresión en la que se captura el contexto de la imagen y una etapa de descompresión simétrica a la anterior que permite una localización precisa de las etiquetas de los píxeles. La etapa de compresión utiliza una arquitectura de redes convolucionales conectadas FCN que extrae las características a los mapas utilizando capas de convoluciones con filtros 3x3, seguidas de una función de activación ReLu para comprimir, en cada grupo, las dimensiones de la imagen aumentando la profundidad del mapa de características mediante una capa de agrupación 2x2 (pooling). La etapa de descompresión utiliza capas de-conovolucionales reduciendo el mapa de características a la mitad a medida que duplica las dimensiones de la imagen gracias a una capa de des-agrupación 2x2 (up-pooling).

Los mapas de características de las etapas de compresión son copiados a la correspondiente etapa de descompresión para no perder la información de partida. Finalmente, una capa de convolución 1x1 procesa el mapa de características para dar un mapa de segmentación que une cada pixel con la categoría correspondiente. Se muestra el esquema de este modelo en la Ilustración 20. En el capítulo 3 profundizaremos más en este modelo y en la aplicación del mismo para el dataset A2D2 de Audi AG® .

Se encuentran ejemplos de uso de este modelo fuera del ámbito biomédico como en (Zhang et al., 2017) en el que el modelo se usa para la extracción de carreteras de imágenes áreas. Es esta diversificación de la utilización de este modelo la que inspira este trabajo fin de master.

Ilustración 20: Esquema del modelo U-NET. Fuente: (Ronneberger et al., 2015)

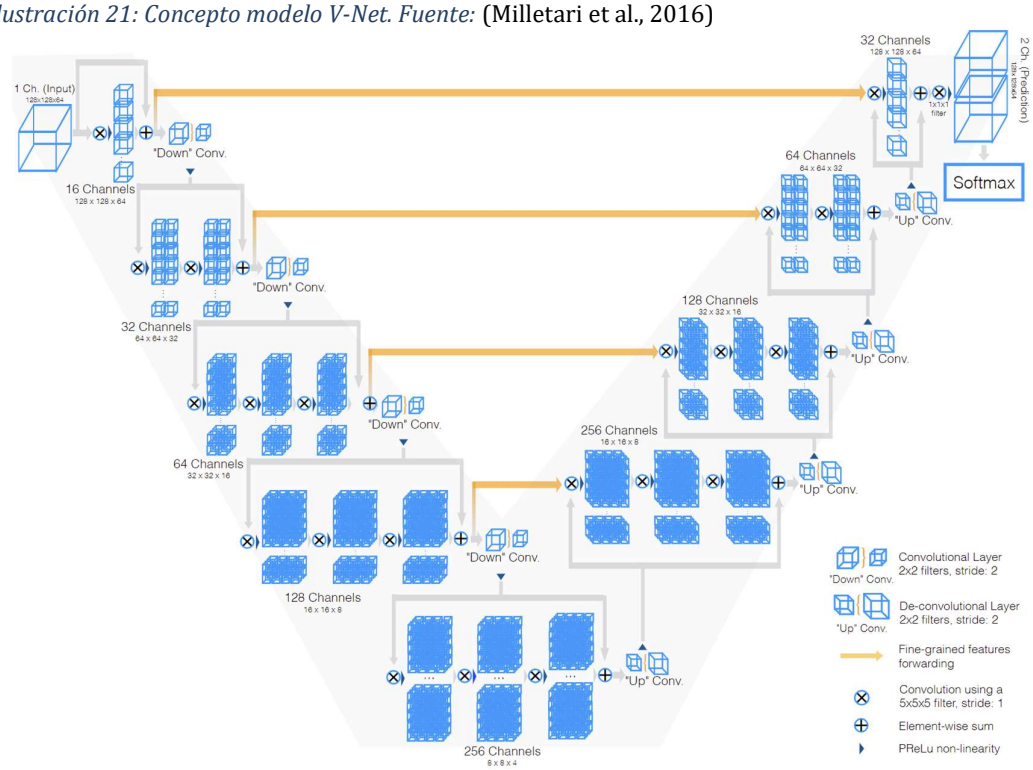


El modelo V-Net (Milletari et al., 2016) está especializado en la segmentación de imágenes médicas tridimensionales. La principal aportación de este modelo es la utilización de una variación del coeficiente Dice (Zijdenbos et al., 1994) como función de pérdida. Esta variación permite al modelo manejar situaciones en las que existe un gran desequilibrio entre el número de voxels (equivalente al pixel para una fotografía 3D) entre el primer plano de la imagen 3D y el fondo de la misma. Se muestra la similitud conceptual entre el modelo U-Net y V-Net en la Ilustración 21.

- Modelos piramidales.

El modelo PsPNet, al que este trabajo fin de master se refiere, pertenece al tipo de modelos piramidales de caracterización (FPN). Este modelo se utiliza como decodificador en la experimentación realizada en el dataset A2D2 de Audi AG® utilizado en este trabajo fin de master.

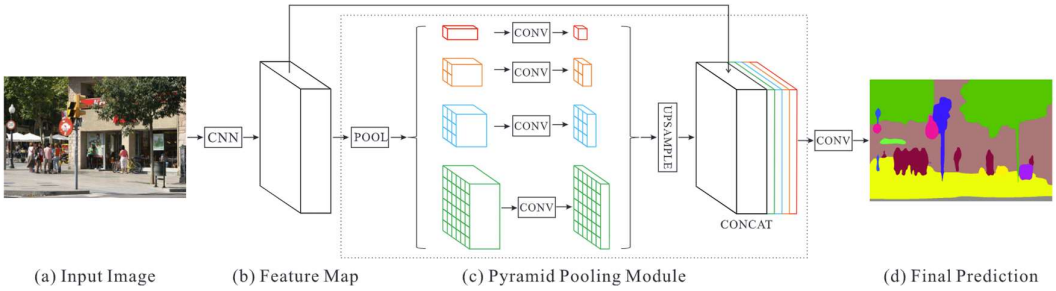
Ilustración 21: Concepto modelo V-Net. Fuente: (Milletari et al., 2016)



El primer modelo de caracterización piramidal (Lin et al., 2016) fue desarrollado para la detección de objetos, aunque también ha sido aplicado para la segmentación de imágenes. El modelo Pyramid Scene Parsing Network (PsPNet) fue desarrollado por (Zhao et al., 2016) como un modelo multiescalar para un mejor aprendizaje del contexto global de una escena. Se muestra esquemáticamente su funcionamiento en la Ilustración 22. El modelo utiliza una red residual profunda (ResNet) como extractor de características combinada con una red convolucional con espaciamiento para obtener el mapa de características global de la imagen (b) en la Ilustración 22. El mapa de características es alimentado en un módulo piramidal de agrupamiento para distinguir patrones a distintas escalas. Estos patrones son agrupados en 4 escalas coincidiendo con un nivel de la pirámide y procesados por una capa convolucional con filtro 1x1 para reducir la dimensionalidad. Las salidas de los niveles piramidales son re-escaladas y combinadas con el mapa de características inicial para capturar conjuntamente el contexto de la información global y local (c) en

la Ilustración 22. Finalmente, una capa convolucional genera las predicciones de las etiquetas para cada pixel.

Ilustración 22: Descripción conceptual PsPNet. Fuente: (Zhao et al., 2016)



2.2.8. Métricas utilizadas en segmentación semántica

Una vez visitados las principales redes neuronales y los modelos de aprendizaje profundo más usados para la tarea de segmentación semántica se recopilan las principales métricas que se utilizan para la evaluación del rendimiento de dichos modelos.

Idealmente un modelo debería evaluarse bajo diferentes puntos de vista. La velocidad de inferencia, los requerimientos de memoria y, por supuesto la precisión en la predicción. En la práctica la gran mayoría de los modelos se focalizan en las métricas de precisión para evaluar el rendimiento.

Las métricas más populares se enumeran a continuación:

- Precisión de pixel o Pixel accuracy (PA):

Ratio de los pixeles clasificados correctamente entre el número total de pixeles. Para K + 1 clases, siendo K el número de clases etiquetadas más el fondo, el pixel accuracy se formularía como se describe en Ecuación 1 siendo pij el número de pixeles de la clase i predichos como pertenecientes a la clase j.

Ecuación 1: Pixel accuracy

$$PA = \frac{\sum_{i=o}^{K} p_{ii}}{\sum_{i=0}^{K} \sum_{j=0}^{K} p_{ij}}$$

- Media del precisión de pixel o mean pixel accuracy (MPA):

Esta métrica, se computa por cada clase de pixel y después se calcula la media aritmética. Se formula como se ve en la Ecuación 2.

Ecuación 2: Media de la precisión de pixel

$$MPA = \frac{1}{K+1}\sum_{i=o}^{K}\frac{p_{ii}}{\sum_{j=0}^{K} p_{ij}}$$

Las métricas anteriores siguen el concepto de tradicional de precisión, y pueden utilizarse para evaluar objetivamente el rendimiento de tareas de clasificación. Sin embargo, para tareas de segmentación semántica, no proporcionan una representación exacta del rendimiento en esta tarea según se explica a continuación.

Dado que segmentación consiste en asignar una etiqueta de clase a cada píxel de una imagen, requiere una delineación precisa de los límites y una localización exacta de los objetos. El rendimiento del modelo viene calificado por la capacidad del mismo de conseguir estos 2 objetivos. En muchos escenarios del mundo real, la clase de fondo supera ampliamente a la de primer plano. En consecuencia, un modelo que prediga sólo la clase mayoritaria (fondo) alcanzaría una gran precisión, aunque no captara los detalles y complejidades de los objetos en primer plano. Este hecho pone de manifiesto la limitación de las métricas de precisión clásicas en la segmentación semántica debido a su sensibilidad al desequilibrio de clases. Para mitigar esta limitación has sido definidas otras métricas que tienen en cuenta este fenómeno.

- Coeficiente Dice (Zijdenbos et al., 1994):

Este coeficiente es muy usado en aplicaciones médicas. Tiene en cuenta tanto la intersección como la unión de las regiones predichas y las de la verdadera muestra, proporcionando una medida de solapamiento o similitud entre ellas. Al considerar la concordancia entre las segmentaciones predichas y las verdaderas, el coeficiente Dice ofrece una evaluación más matizada del rendimiento. Se calcula mediante la siguiente fórmula descrita en la Ecuación 3.

Ecuación 3: Coeficiente Dice

$$\boldsymbol{Dice(A,B)} = \frac{\boldsymbol{2x|A \cap B|}}{\boldsymbol{|A| + |B|}}$$

En términos sencillos, para calcular el coeficiente Dice, se cuenta cuántos elementos hay en ambos conjuntos A y B (la intersección) y se multiplica ese número por 2. Se suma el número de elementos del conjunto A y el número de elementos del conjunto B y se divide el resultado del paso 1 por el resultado del paso 2. El número resultante oscila entre 0 y 1: 0 significa que los conjuntos son completamente diferentes y 1 que son completamente similares. Normalmente, como se presentan varias clases que predecir, se calcula el coeficiente para cada clase y se ofrece la media como resultado.

- Coeficiente IoU

El coeficiente intersection over union (IoU), también conocido como índice Jaccard, es una métrica utilizada comúnmente para evaluar la precisión de la detección o segmentación de objetos en tareas de visión por computadora, especialmente para aplicaciones en el área de la medicina (Taha & Hanbury, 2015). Con un concepto de conjuntos de píxeles similar al explicado para el coeficiente Dice, El coeficiente IoU se calcula comparando el área de intersección entre dos regiones (por ejemplo, la región predicha y la región de referencia) dividida por el área de unión de esas regiones. Matemáticamente, se puede expresar como se indica en la Ecuación 4.

Ecuación 4: Coeficiente IoU.

$$\boldsymbol{IoU(A,B)} = \frac{\boldsymbol{|A \cap B|}}{\boldsymbol{|A| \cap |B|}}$$

El resultado del coeficiente IoU varía entre 0 y 1, donde 0 indica una falta de superposición entre las regiones y 1 indica una coincidencia perfecta entre las regiones. Como el coeficiente Dice, se utiliza calculando el valor por cada clase y arrojando la media aritmética como resultado.

Desarrollo específico de la contribución

3. Desarrollo específico de la contribución

En este capítulo del trabajo, profundizamos en lo que se ha hecho para conseguir el objetivo anunciado. Para su desarrollo se ha utilizado un ordenador portátil modelo Inspiron 16 7610 de la marca Dell® equipado con una CPU del fabricante Intel® modelo i7-11800H de onceava generación con 8 núcleos capaces de alcanzar los 2.30GHz de velocidad de procesamiento. La memoria RAM del equipo es de 16Gb y se utiliza una GPU del fabricante NVIDIA® modelo GForce RTX 3060 con 6GB de memoria dedicada y un a velocidad de proceso máxima de 1280MHz. El sistema operativo el Windows 11® en la versión Home.

3.1. Descripción del dataset A2D2 de Audi AG®

El 14 de abril de 2020, Audi AG® publica el dataset A2D2 (Geyer et al., 2020) y lo hace bajo licencia CCBY-ND 4.0 que es la menos restrictiva de todas permitiendo su uso incluso para fines comerciales. A modo de resumen el dataset tiene un tamaño de 2.3 Tb aproximadamente y consta de más de 40,000 fotogramas etiquetados para segmentación semántica (2D) y nube de puntos (3D) entre los cuales más de 12,000 fotogramas también incluyen etiquetas para cajas de limitación (boundary boxes) 3D. Además, contiene 390.000 fotogramas no etiquetados con sus correspondientes datos de sensores procedentes de varios recorridos en tres ciudades distintas (Gaimershein, Ingolstadt y Munich). Para la captura de estos datos se utilizaron 6 cámaras y 5 sensores LIDAR además del bus de datos del vehículo para recoger información detallada y extensiva sobre 22 parámetros.

Los datos fueron recogidos entre el 7 de agosto y el 4 de diciembre de 2018 en autopistas, carreteras y zonas urbanas del sur de Alemania. Dadas las fechas mencionadas las condiciones climáticas son las correspondientes a las estaciones de verano, otoño e invierno y siempre en condiciones diurnas.

3.1.1. Sistema de sensores

Los datos fueron recolectados utilizando un vehículo en el que se instalaron 6 cámaras y 5 LiDARs cuyos identificadores y características se describen en la Tabla 2 ,en la Tabla 3 y en la Tabla 4.

Tabla 2: Localización sensores en vehículo. Fuente:(Geyer et al., 2020).

SENSOR	LOCALIZACIÓN	TIPO DE SENSOR
CÁMARA	Front-center	Sekonix SF3325-100
CÁMARA	Front-left	Sekonix SF3324-100
CÁMARA	Front-right	Sekonix SF3324-100
CÁMARA	Side-left	Sekonix SF3324-100
CÁMARA	Side-right	Sekonix SF3324-100
CÁMARA	Rear-center	Sekonix SF3324-100
LIDAR	Front-center	Velodyne VLP-16
LIDAR	Front-left	Velodyne VLP-16
LIDAR	Front-right	Velodyne VLP-16
LIDAR	Rear-left	Velodyne VLP-16
LIDAR	Rear-right	Velodyne VLP-16

Tabla 3: Especificaciones de las cámaras. Fuente:(Geyer et al., 2020).

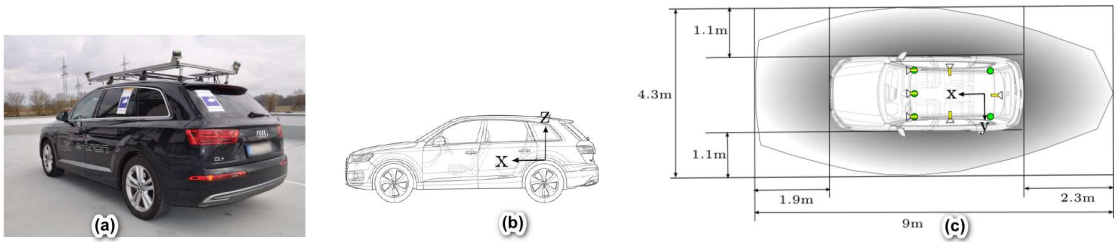
	SF3225-100	SF3224-100
CAMPO DE VISIÓN HORIZONTAL	60º	120º
CAMPO DE VISIÓN VERTICAL	38º	73º
CAMPO DE VISIÓN DIAGONAL	70º	146º
SENSOR	Onsemi AR0231	Onsemi AR0231
RESOLUCIÓN	1928 x 1208 (2MP)	1928 x 1208 (2MP)
FILTRO DE COLOR	RCCB	RCCB

Tabla 4: Especificaciones de LiDAR. Fuente:(Geyer et al., 2020).

	VLP-16
CAMPO DE VISION AZIMUTAL	360º
CAMPO DE VISION VERTICAL	30º (+15º a -15º)
CANALES	16
RESOLUCION VERTICAL	2º
FRECUENCIA	5-20Hz (10Hz utilizado para A2D2)
RANGO	Hasta 100m
TASA	Hasta 300,000 puntos/segundo

Los sensores se dispusieron en el vehículo según se muestra en la Ilustración 23 donde se muestra como el punto de referencia del sistema de coordenadas globales parte de la parte superior del tren trasero.

Ilustración 23: Posicionamiento de los sensores: (a) Vehículo y adaptación. (b) Referencia global de posicionamiento (c) Cámaras en amarillo, LiDAR en Verde. Puntos ciegos de los LiDAR (sombreados). Fuente: (Geyer et al., 2020).



Con esta disposición se busca tener una perspectiva de 360º alrededor del vehículo, minimizando los puntos ciegos de las cámaras y sensores y maximizando los solapamientos entre los fotogramas, y las nubes de puntos obtenidas por los sensores LiDAR. Nótese que existe una zona de puntos ciegos para los sensores LiDAR que coincide con las proximidades del vehículo. La Ilustración 24 muestra visualmente el resultado de este planteamiento una vez ajustadas las posiciones de los sensores. Se consiguió un solapamiento (fuera de los puntos ciegos de los LiDAR) de más del 90%.

Ilustración 24: Mapeo de puntos LiDAR solapados con imágenes de las cámaras. Fuente: (Geyer et al., 2020).

Además de estos sensores, el vehículo fue equipado con un sistema de grabación de los datos del bus que permitió recoger las variables de conducción recogidas en la Tabla 5.

Todos estos datos hacen del dataset A2D2 uno de los más completos y útiles para los sistemas de conducción autónoma ya que engloba la información del entorno en 2D y 3D, información de la significación de las instancias, información del comportamiento del vehículo e inputs del experto todo ello interrelacionado gracias a marcas de tiempo comunes.

Tabla 5: Variables recogidas por el bus del vehículo. Fuente: Elaboración propia.

ATRIBUTO EN ARCHIVO	DESCRIPCIÓN
ACCELERATION_X	Aceleración en el eje X
ACCELERATION_Y	Aceleración en el eje Y
ACCELERATION_Z	Aceleración en el eje Z
ACCELERATOR_PEDAL	Pedal del acelerador
ACCELERATOR_PEDAL_GRADIENT_SIGN	Signo del gradiente del pedal del acelerador
ANGULAR_VELOCITY_OMEGA_X	Velocidad angular en el eje X
ANGULAR_VELOCITY_OMEGA_Y	Velocidad angular en el eje Y
ANGULAR_VELOCITY_OMEGA_Z	Velocidad angular en el eje Z
BRAKE_PRESSURE	Presión de freno
DISTANCE_PULSE_FRONT_LEFT	Distancia del pulso frontal izquierdo
DISTANCE_PULSE_FRONT_RIGHT	Distancia del pulso frontal derecho
DISTANCE_PULSE_REAR_LEFT	Distancia del pulso trasero izquierdo
DISTANCE_PULSE_REAR_RIGHT	Distancia del pulso trasero derecho
LATITUDE_DEGREE	Grado de latitud
LATITUDE_DIRECTION	Dirección de latitud
LONGITUDE_DEGREE	Grado de longitud
LONGITUDE_DIRECTION	Dirección de longitud
PITCH_ANGLE	Ángulo de inclinación
ROLL_ANGLE	Ángulo de balanceo
STEERING_ANGLE_CALCULATED	Ángulo de dirección calculado
STEERING_ANGLE_CALCULATED_SIGN	Signo del ángulo de dirección calculado
VEHICLE_SPEED	Velocidad del vehículo

3.1.2. Estructura del dataset

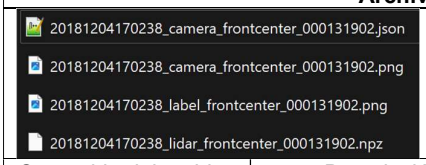
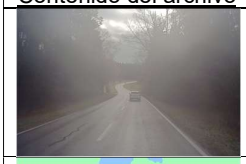
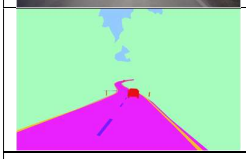
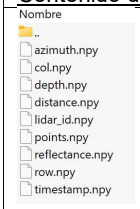
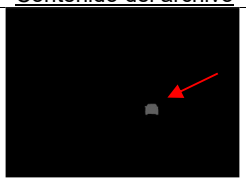
En el dataset encontramos 2 tipos de datos, los datos etiquetados y los no etiquetados. Los datos etiquetados, son aquellos cuyas imágenes y datos de los sensores LiDAR, tienen una correspondencia con otra imagen a la cual se ha aplicado una segmentación semántica y/o una segmentación de instancias, para identificar cada pixel con la clase correcta. Esto datos son los que usaremos para el entrenamiento de la red neuronal U-Net. Los datos no etiquetados que están formados por imágenes y datos de sensores que carecen de la correspondencia antes descrita.

Los datos etiquetados se distribuyen en los archivos camera_lidar_semantic.tar, camera_lidar_semantic_instance.tar, camera_lidar_semantic_bboxes y camera_lidar_semantic_bus.tar. Su estructura se explica en la Tabla 6, la Tabla 9 y la Tabla 10.

Los datos contenidos en el archivo camera_lidar_semantic.tar contienen 31448 imágenes procedentes de 23 secuencias de captura de datos etiquetadas en 38 categorías para la segmentación semántica. Se añaden a estos datos las correspondientes nubes de puntos capturadas por los sensores LiDAR. Constituyendo un total de 174.2 Gb. Cada fotograma

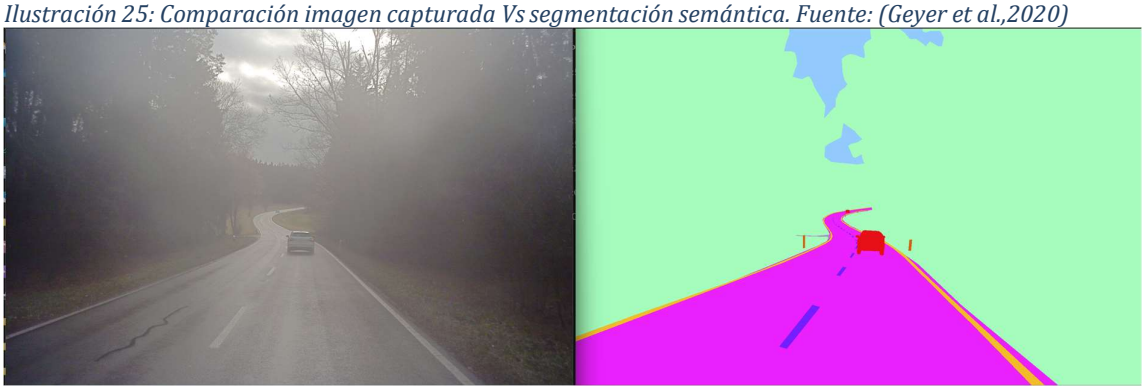
está apoyado por 4 archivos cuyo contenido se explica en la Tabla 6. Este archivo está focalizado en la segmentación semántica de imágenes.

Tabla 6: Archivos camera_lidar_semantic.tar y camera_lidar_semantic_instace.tar. Fuente: Elaboración propia.

<table>
<tr><th colspan="4">Archivo: camera_lidar_semantic.tar</th></tr>
<tr><td colspan="2"> 
</td><td colspan="2">Ejemplo de conjunto de archivos que apoyan cada fotograma del dataset camera_lidar_semantic.tar</td></tr>
<tr><td>Contenido del archivo</td><td>Descripción</td><td>Contenido del archivo</td><td>Descripción</td></tr>
<tr><td>  </td><td>Archivo del fotograma RGB en formato .png.</td><td> 
</td><td>Archivo .JSON con la marca de tiempo del fotograma y el identificativo de los LiDAR</td></tr>
<tr><td>  </td><td>Fotograma de etiquetas tras la segmentación semántica en formato .png</td><td>  </td><td>Archivo .tzn con los puntos 3D procedente del censo LiDAR. (visualización realizada con librería open3d (Zhou et al., 2018)</td></tr>
<tr><th colspan="4">Archivo: camera_lidar_semantic_instace.tar</th></tr>
<tr><td>Contenido del archivo</td><td>Descripción</td><td>Contenido del archivo</td><td>Descripción</td></tr>
<tr><td> 
</td><td>El archivo .tzn incluye archivos en formato .npy (formato binario de almacenamiento de arrays de numpy) con toda la información de la nube de puntos capturada por el sensor LiDAR.</td><td>  </td><td>Archivo con etiquetado de objetos dinámicos en formato .npg. Corresponde a un marca de tiempo muy próxima al fotograma de ejemplo. Se puede distinguir la silueta de un vehículo que coincide con la del vehículo del fotograma de ejemplo.</td></tr>
</table>

Las etiquetas usadas para la segmentación semántica siguen el formato de color hexadecimal de forma que la etiqueta de la imagen es otra imagen en la que el valor de cada pixel de la imagen inicial ha sido reemplazado por el correspondiente código de color hexadecimal asociado a la clase. Se muestra en la Ilustración 25.

Ilustración 25: Comparación imagen capturada Vs segmentación semántica. Fuente: (Geyer et al.,2020)



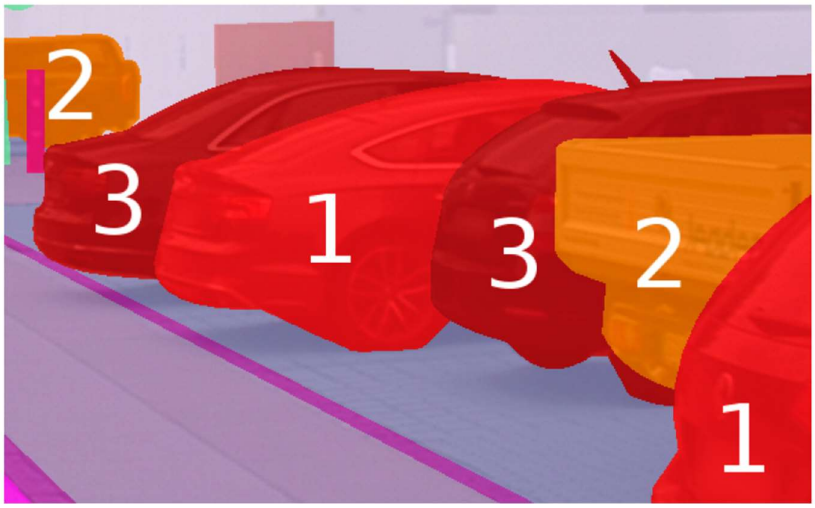
La Tabla 7 muestra una lista parcial de las categorías del dataset usadas para la segmentación semántica.

Tabla 7: Lista parcial categorías segmentación semántica A2D2. Fuente: Elaboración propia.

Lista parcial de categorías del Dataset A2D2 de Audi AG®	
"#ff0000: "Car 1"	"#400040": "Irrelevant signs"
"#b65906": "Bicycle 1"	"#b97a57": "Road blocks"
"#cc99ff": "Pedestrian 1"	"#000064": "Tractor"
"#ff8000": "Truck 1"	"#d23273": "Zebra crossing"
"#00ff00": "Small vehicles 1"	"#ff0080": "Obstacles / trash"
"#0080ff": "Traffic signal 1"	"#fff68f": "Poles"
"#00ffff": "Traffic sign 1"	"#960096": "RD restricted area"
"#ffff00": "Utility vehicle 1"	"#ccff99": "Animals"
"#e96400": "Sidebars"	"#eea2ad": "Grid structure"
"#6e6e00": "Speed bumper"	"#212cb1": "Signal corpus"
"#808000": "Curbstone"	"#b432b4" "Drivable cobblestone"
#ffc125": "Solid line"	"#ff46b9": "Electronic traffic"
"#400040": "Irrelevant signs"	"#eee9bf": "Slow drive area"
"#b97a57": "Road blocks"	"#8000ff": "Dashed line"
"#000064": "Tractor"	"#ff00ff": "RD normal street"
"#8b636c": "Non-drivable street"	"#87ceff": "Sky"

Si en el mismo fotograma existen múltiples instancias de la misma clase, (peatones, ciclistas, coches, camiones…) solapados en el mismo plano, estás instancias están diferenciadas en las etiquetas utilizando subclases como coche1, coche2 etc. Esto sólo aplica en el caso de que las instancias sean adyacentes o estén solapadas. En la Ilustración 26 se muestra un ejemplo de este escenario. Comenzando desde abajo a la derecha de la figura, los vehículos 1, 2 y 3 están etiquetados con diferentes subclases puesto que comparten borde. El siguiente vehículo a la derecha es etiquetado con la subclase 1 puesto que no comparte borde en el vehículo etiquetado con subclase 1 anteriormente, en la Ilustración 26 se muestra una ejemplo explicativo de este mecanismo. El dataset contiene un total de 55 clases de las cuales 15 identifican elementos solapados, por lo que se identifican 40 objetos diferentes.

Ilustración 26: Ejemplo de instancias adyacentes de la misma clase. Fuente:(Geyer et al., 2020).



Los datos contenidos en el archivo camera_lidar_semantic_instance.tar contine contiene 31410 imágenes procedentes de las mismas 23 secuencias de captura de datos del archivo camera_lidar_semantic_.tar etiquetadas en 7 categorías para la identificación de objetos dinámicos. Constituyendo un total de 479Mb. Cada fotograma está apoyado por un único archivo que incluye las etiquetas por cada pixel del objeto dinámico contenido en la el fotograma de la misma marca de tiempo asociada al fotograma del archivo camera_lidar_semantic_.tar. Se puede ver el contenido en la Tabla 6. El número de imágenes del archivo camera_lidar_semantic_instance.tar (31410) no es el mismo que el dataset anterior, esto es debido a que las imágenes sin objetos dinámicos han sido suprimidas. El motivo de la supresión es que la relevancia de la identificación de objetos dinámicos es mayor que los estáticos para el escenario de la conducción autónoma.

Las etiquetas usadas para la identificación de elementos dinámicos han sido integradas dentro de la codificación unit16 de escala de grises que está constituida por 16 dígitos de los cuales los primeros 6 incluirán la información del índice de la clase, comenzando por 1 ya que 0 está reservado para el fondo, mientras que los 10 siguientes darán la identificación del número de instancia (objeto dinámico) incluido en la imagen. Por ejemplo:

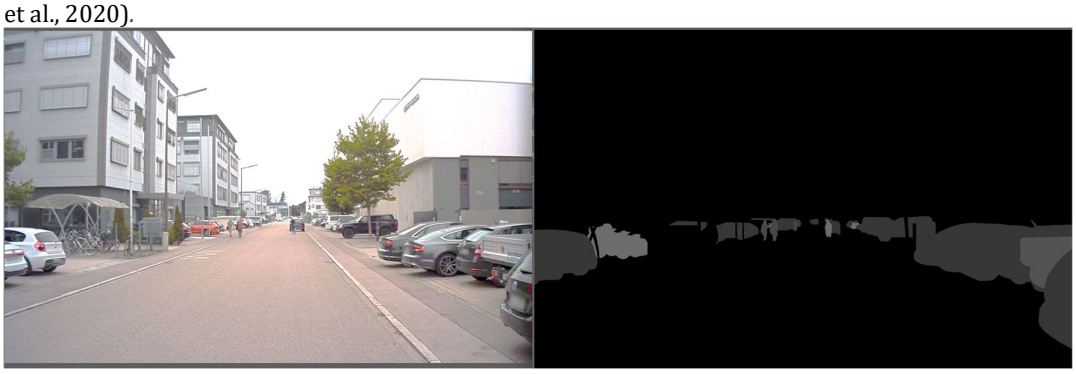
- El coche número 5 de una fotografía se correspondería el valor 000001 para los primeros 6 dígitos ya que la clase coche es la número 1 mientras que los últimos 10 dígitos tendrían el valor 0000000100 identificando el número 5 de los coches incluidos en la fotografía.
- El tractor número 5 de una fotografía se correspondería el valor 000111 para los primeros 6 dígitos ya que la clase coche es la número 7 mientras que los últimos 10 dígitos tendrían el valor 0000000010 identificando el número 3 de los tractores incluidos en la fotografía.

La Tabla 8, muestra las categorías de las clases utilizadas para identificar los objetos dinámicos

Tabla 8: Categorías utilizadas para la categorización de elementos dinámicos A2D2. Fuente: Elaboración propia

Categorías usadas para la identificación de objetos dinámicos Dataset A2D2 de Audi AG®
cars: 1
pedestrians: 2
trucks: 3
smallVehicle: 4
utilityVehicle: 5
bicycle: 6
tractor: 7

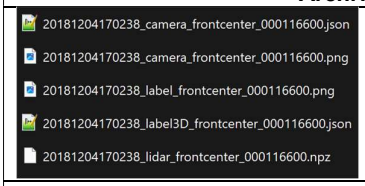
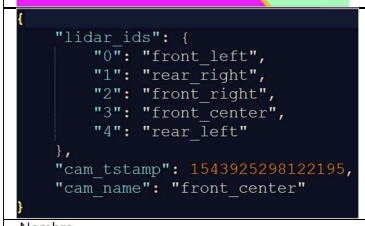
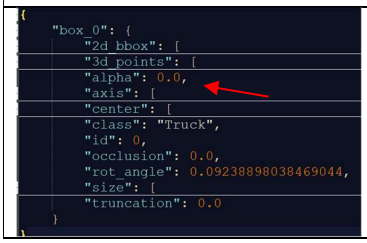
Ilustración 27: Comparación imagen capturada Vs etiquetado de objetos dinámicos. Fuente:(Geyer et al., 2020).



Los datos contenidos en el archivo camera_lidar_semantic_bboxes contine contiene 12497 imágenes procedentes de 18 secuencias de captura de datos ya contenidas en el archivo camera_lidar_semantic_.tar.

Se añaden a estos datos las correspondientes nubes de puntos capturadas por los sensores LiDAR y las etiquetas 3D de las cajas de localización de los objetos. Constituyendo un total de 50.6 Gb. Cada fotograma está apoyado por 5 archivos cuyo contenido se explica en la Tabla 9. Este archivo está focalizado en la localización de objetos dinámicos.

Tabla 9: Archivo camera_lidar_semantic_bboxes.tar. Fuente: Elaboración propia.

Archivo: camera_lidar_semantic_bboxes.tar	
	Ejemplo de conjunto de archivos que apoyan cada fotograma del dataset camera_lidar_semantic_bboxes.tar
Contenido del archivo	Descripción
	Archivo del fotograma RGB en formato .png.
	Fotograma de etiquetas tras la segmentación semántica en formato .png
	Archivo .JSON con la marca de tiempo del fotograma y el identificativo de los LiDAR
	Archivo .tzn con los puntos 3D procedente del LiDAR. (visualización realizada con librería open3d) (Zhou et al., 2018) . El archivo .tzn incluye archivos en formato .npy (formato binario de almacenamiento de arrays de numpy) con toda la información de la nube de puntos capturada por el sensor LiDAR.
	Archivo .JSON con las etiquetas 3D que definen la localización del objeto en 2D y en 3D junto con la clase del objeto dinámico y otra información relevante.

Los datos contenidos en el archivo camera_lidar_semantic_bus.tar contine 21 archivos JSON correspondiente a 21 secuencias de captura de datos ya contenidas del archivo camera_lidar_semantic_.tar. Cada archivo .JSON contiene la información capturada por el bus del vehículo procedente de los sensores del mismo. Constituyen un total de 1.6Gb. Los datos de cada secuencia capturada se almacenan dentro de un archivo .JSON con estructura fija que incluye las variables descritas en la Tabla 5. Cada variable contiene sus propias marcas de tiempo, unidades y valores de todos los datos recogidos dentro de la marcad de tiempo del fotograma ya que hay varias lecturas de los sensores del vehículo por parte del bus por cada fotograma capturado por el sistema. Se muestra la descripción de estos archivos .JSON en la Tabla 10. Este archivo está enfocado en el registrar el estado y comportamiento del vehículo conducido por un experto para las escenas en las que han sido recogidos el resto de datos.

Tabla 10: Archivo contenido en el archivo camera_lidar_semantic_bus. Fuente: Elaboración propia

Descripción de la estructura del archivo .JSON de los catos capturados por el bus del vehículo.	Detalle de los valores y marcas de tiempo de la variable (“acceleration_x”)

A continuación, en la Ilustración 28 y en la Ilustración 29 se muestra los resultados del análisis exploratorio de los datos focalizado en entender la distribución de la las etiquetas de las imágenes segmentadas.

Ilustración 28: Análisis exploratorio de los datos etiquetados. Fuente:(Geyer et al., 2020).

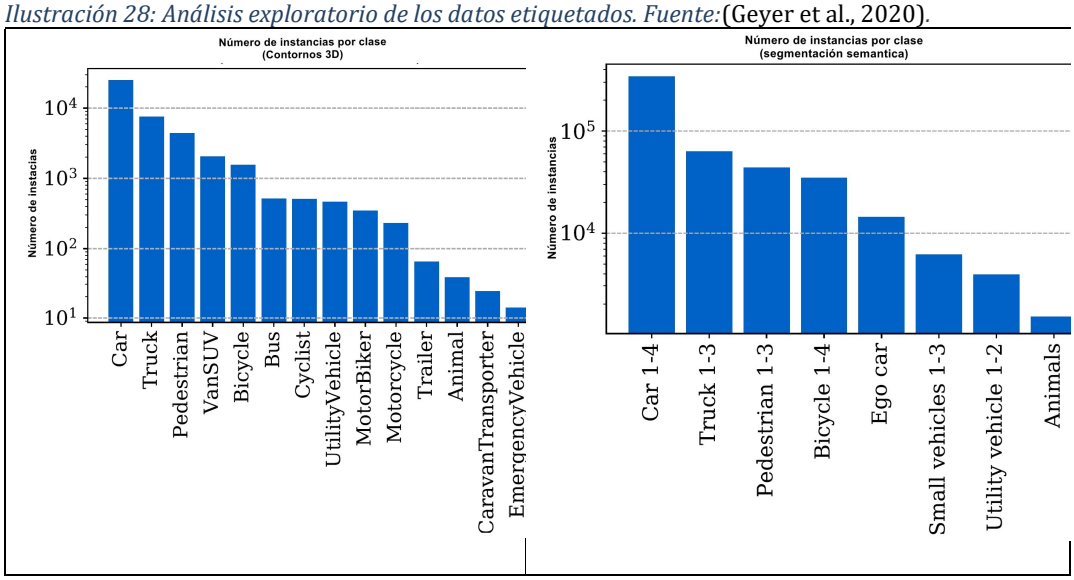
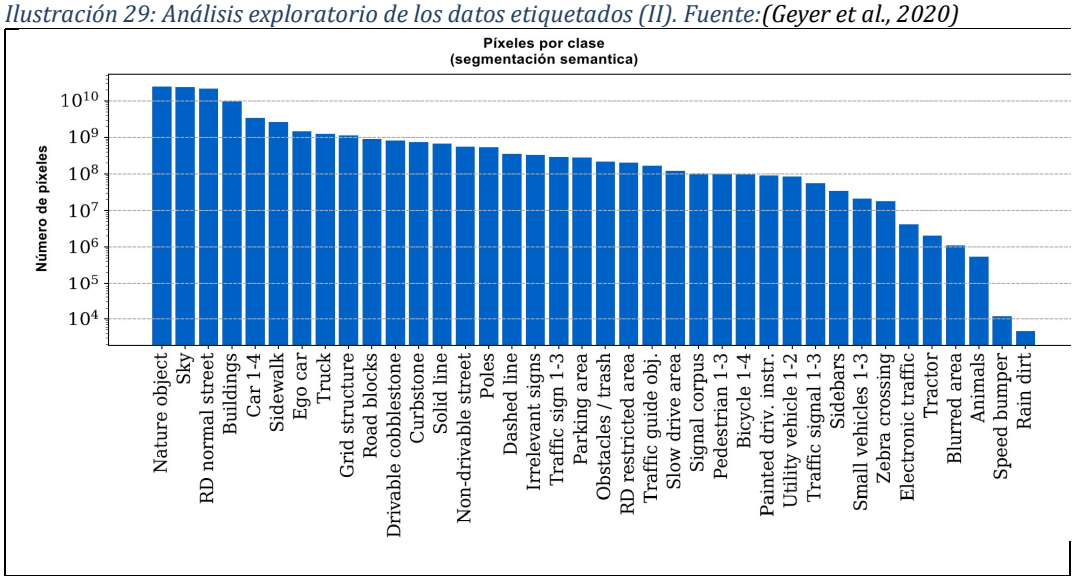


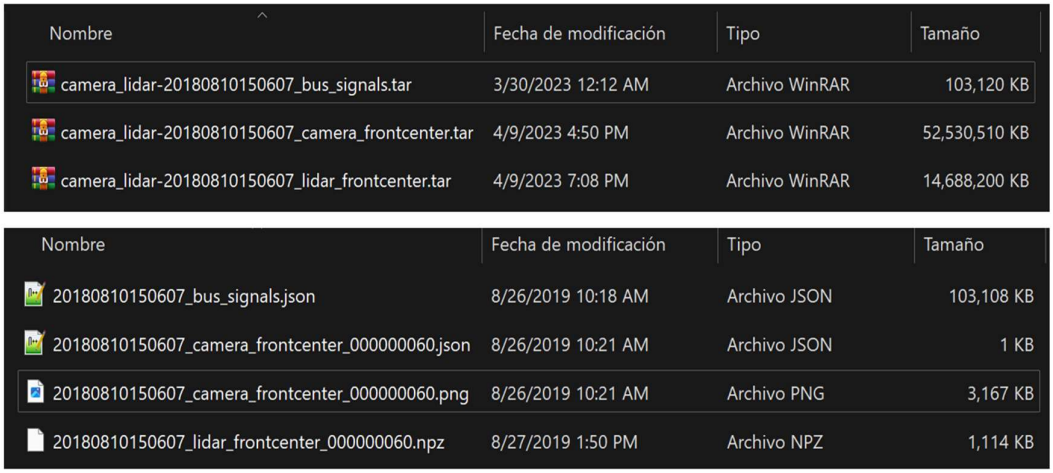
Ilustración 29: Análisis exploratorio de los datos etiquetados (II). Fuente:(Geyer et al., 2020)



Los datos no etiquetados se distribuyen en 3 datasets correspondientes a los datos de las escenas capturadas en 3 ciudades de Alemania Gaimersheim, Ingolstadt y Munich. Cada uno de estos datasets contiene 3 archivos los cuales contienen datos de las cámaras, los sensores LiDAR y el bus. Para el caso de las cámaras y sensores LiDAR sólo hay capturas

de los sensores frontales centrales. En total se dispone de 392.556 RGB fotogramas en formato .png complementados por un archivo .JSON de información y un archivo .npz que contiene la nube de puntos del sensor LiDAR. Como en los datos etiquetados, el archivo de bus es único y contiene los datos de toda la secuencia estructurados dentro de él. La Ilustración 30 muestra los archivos descritos para el dataset de la ciudad de Gaimersheim.

Ilustración 30: Descripción del dataset de la ciudad de Gaimersheim. Fuente: (Geyer et al., 2020).



3.1.3. Codificación y nomenclatura de las rutas, directorios y archivos

Las 23 secuencias etiquetadas siguen una codificación de fecha y hora que después se usa para identificar las carpetas que contienen la información: aaaammdd_hhmmss. Está codificación se aplica en todos los archivos del dataset de forma que se identifican las secuencias por el nombre de la carpeta en todos los casos. Una vez dentro de la carpeta de la secuencia, estas se subdividen dependiendo de las características del archivo de datos (cámara, LiDAR, etiqueta, 3D, instancia…) para subdividirse de nuevo en función del sensor de procedencia y finalmente identificando el archivo por su número de identificación.

A modo de ejemplo se expone la estructura de los archivos de segmentación semántica. Un fotograma número de identificación 2003 de

una escena grabada el 8 de agosto de 2018 a las 16:04:26 desde la cámara frontal central tendrá los siguientes archivos con la siguiente ruta:

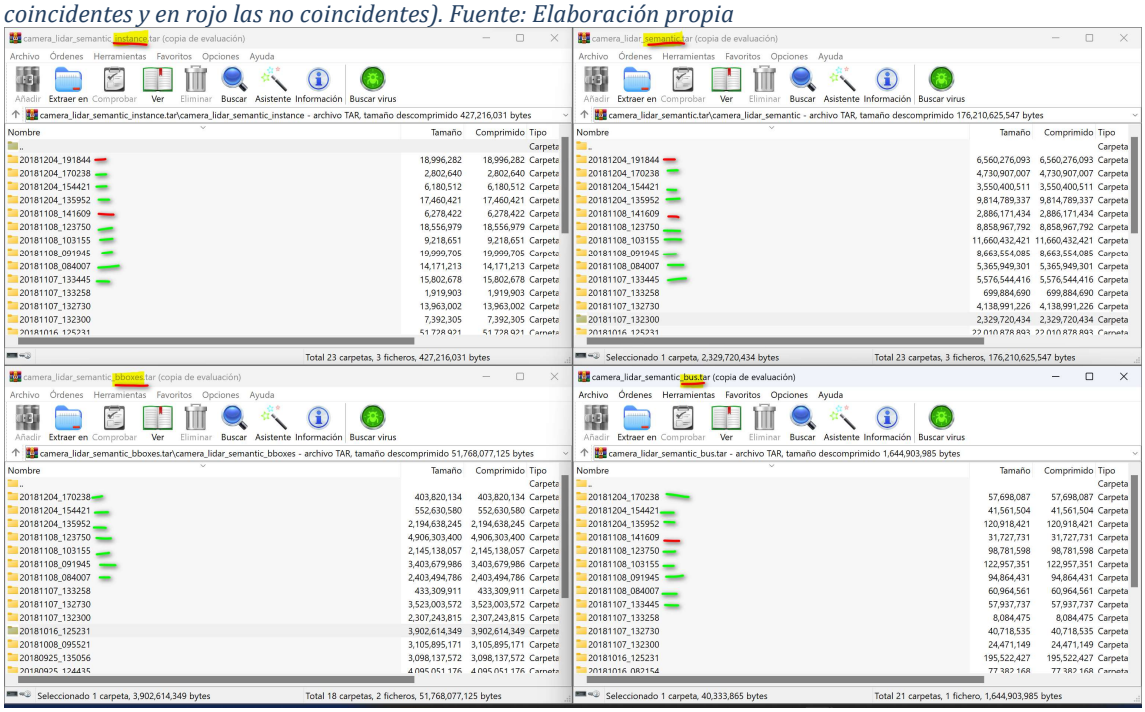
- Imagen RGB:
  20180808_160426/camera/cam_front_center/
  20180808160426_camera_frontcenter_000002003.png
- Información de imagen:
  20180808_160426/camera/cam_front_center/
  20180808160426_camera_frontcenter_000002003.json
- Nube de puntos 3D LiDAR:
  20180808_160426/lidar/cam_front_center
  /20180808160426_lidar_frontcenter_000002003.npz
- Imagen etiquetada por segmentación semántica:
  20180808_160426/label/cam_front_center/
  20180808160426_label_frontcenter_000002003.png

Los datos capturados por el bus del vehículo siguen la misma lógica, pero sin la estructura del sensor de captura ya que el bus es único. Este sería el ejemplo que muestra la estructura de la ruta:

- YYYYMMDD_hhmmss/bus/YYYYMMDDhhmmss_bus_signal.json

Se identifica que no todas las secuencias incluyen todos los datos (Imagen, Mapa de puntos LiDAR, Objetos dinámicos). Analizando en detalle, vemos que las secuencias tomadas el 16 de octubre de 2018 a las 09:50:36 (codificación de carpeta 20181016_095036) y 4 de diciembre de 2018 a las 19:08:44 (codificación de carpeta 20181016_190844) carece de archivos LiDAR, datos de BUS y localización de objetos, mientras que las secuencias con codificación de carpeta 20181016_082154, 20181107_133445 y 20181108_141609 carecen de archivos LiDAR, pero si el resto de información. El resto de las carpetas (18) contienen el conjunto de datos completo. En la Ilustración 31 se muestran las escenas contenidas en cada uno de los archivos del dataset de datos etiquetados.

Ilustración 31: Mapeo de las escenas contenidas en datos etiquetados. (Marcadas en verde las coincidentes y en rojo las no coincidentes). Fuente: Elaboración propia



Los archivos incluidos en los datasets no etiquetados siguen una estructura y codificación idéntica a los etiquetados.

3.2. Extracción, tratamiento y carga de datos (ETL)

Con respecto a la extracción de los datos, la forma de proceder es inmediata ya que los archivos en los que se suministra el dataset, como se ha descrito en el apartado anterior, están estructurados con rutas de carpetas que permiten la interrelación de las imágenes y datos de sensores con sus correspondientes etiquetas de tiempo y/o segmentación semántica. Descomprimiendo los archivos se obtienen los datos ya estructurados para ser tratados y cargados en nuestro modelo. Se utiliza el contenido correspondiente a la cámara frontal del dataset camera_lidar_semantic que contiene 31448 imágenes y sus correspondientes etiquetas con las correspondencia de cada pixel con las clases descritas en el capítulo 3.1.2.

El modelo U-Net presenta unos requisitos que las imágenes del dataset han de cumplir para poder utilizarlo lo que obliga a realizar una serie de transformaciones.

El primer requisito procede del origen de la arquitectura U-Net la cual fue ideada para segmentar células de muestras biológicas microscópicas del fondo de la muestra. Para ello, la salida del modelo es una capa convolucional 1x1 que mapea el mapa final de características de 64 componentes para asignarlos a cada una de las clases aprendidas en la etapa de entrenamiento. Para que este planteamiento funcione, los píxeles de las imágenes etiquetadas han de estarlo en el mismo rango que la última cada (64). En este trabajo se utiliza la red U-Net para clasificar múltiples clases (hasta 55) y las etiquetas de los pixeles están en formato RGB[3] por lo que la etiqueta de cada pixel será una de las 55 tuplas de 3 dígitos entre 1 y 256. Este formato de etiquetas no es compatible para entrenar la red U-Net por lo que se debe re-etiquetar los pixeles de las imágenes segmentadas para darles una tupla con valores iguales en los 3 dígitos valor entre 1 y 55 ambos inclusive. El tipo de formato será [etiqueta_clase, etiqueta_clase, etiqueta_clase].

Para ello, se crea un programa en Python que, mediante un diccionario con la correspondencia entre la etiqueta de la clase original y la correspondiente entre 1 y 55, reemplaza las tuplas pixel a pixel. Esta transformación se realiza mediante la librería cv2 de OpenCV (Bradski, 2000) para leer los archivos de las imágenes y transformarlos en arrays y la librería numpy (Harris et al., 2020) para reemplazar los pixeles aplicando un bucle for.

El diccionario se ha elaborado partiendo de la lista original de clases, cuya correspondencia está en formato de color hexadecimal, y teniendo en

3 El código RGB (rojo, verde, azul), ha sido desarrollado por la Comisión Internacional de Iluminación (Commission Internationale de l'Eclairage, CIE), El modelo RGB propone que cada componente de color se codifique en un byte, que corresponde a 256 intensidades de rojo, 256 intensidades de verde y 256 intensidades de azul.

cuenta que la carga se realiza con la librería cv2 a formato RGB. Para ello se han construido equivalencias entre el formato hexadecimal y RGB para después construir la correspondencia de la etiqueta RGB con la etiqueta entre 1 y 55.

Esta transformación se aplica a las imágenes etiquetadas en escala original (1208x1920) y lleva unos 20 segundos por imagen con el equipo utilizado en este trabajo fin de master. Siendo el tamaño del dataset camera_lidar_semantic de 31448 imágenes etiquetadas el tiempo de proceso es de 175 horas. Se opta por hacer esta transformación de los datos fuera del entrenamiento del dataset dado la magnitud temporal del mismo. La Tabla 11 muestra las 10 primeras entradas del diccionario creado a partir de las clases originales y la clases a reemplazar, mientras que en la Tabla 12 muestra un ejemplo del resultado de la ejecución del programa en Python con un volumen de 4291 etiquetas que duró 18 horas, 58 minutos y 33 segundos.

Tabla 11: Diccionario para reemplazar etiquetas. Fuente: Elaboración propia.

mapeo_RGB_class: {(255, 0, 0): (1, 1, 1), (200, 0, 0): (2, 2, 2), (150, 0, 0): (3, 3, 3), (128, 0, 0): (4, 4, 4), (182, 89, 6): (5, 5, 5), (150, 50, 4): (6, 6, 6), (90, 30, 1): (7, 7, 7), (90, 30, 30): (8, 8, 8), (204, 153, 255): (9, 9, 9), (189, 73, 155): (10, 10, 10),…}

Tabla 12: Comparación valor de pixeles antes y después de la transformación. Fuente: Elaboración propia.

Correspondencia valor pixel imagen Vs valor pixel etiqueta antes del procesado
-Número de carpetas del dataset: 2 -Número de imágenes: 4291 -Número de imágenes etiquetadas: 4291 --- Pixels imagen 0: 0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [157 167 175] 1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [156 166 174] Pixels label 0: 0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [135 206 255] 1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [135 206 255] --- Pixels imagen 1: 0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [152 167 172] 1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [147 162 167] Pixels label 1: 0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [135 206 255] 1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [135 206 255] --- Pixels imagen 2: 0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [152 167 170] 1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [152 167 170] Pixels label 2: 0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [135 206 255] 1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [135 206 255] ---

Correspondencia pixel imagen Vs Pixel etiqueta después del procesado

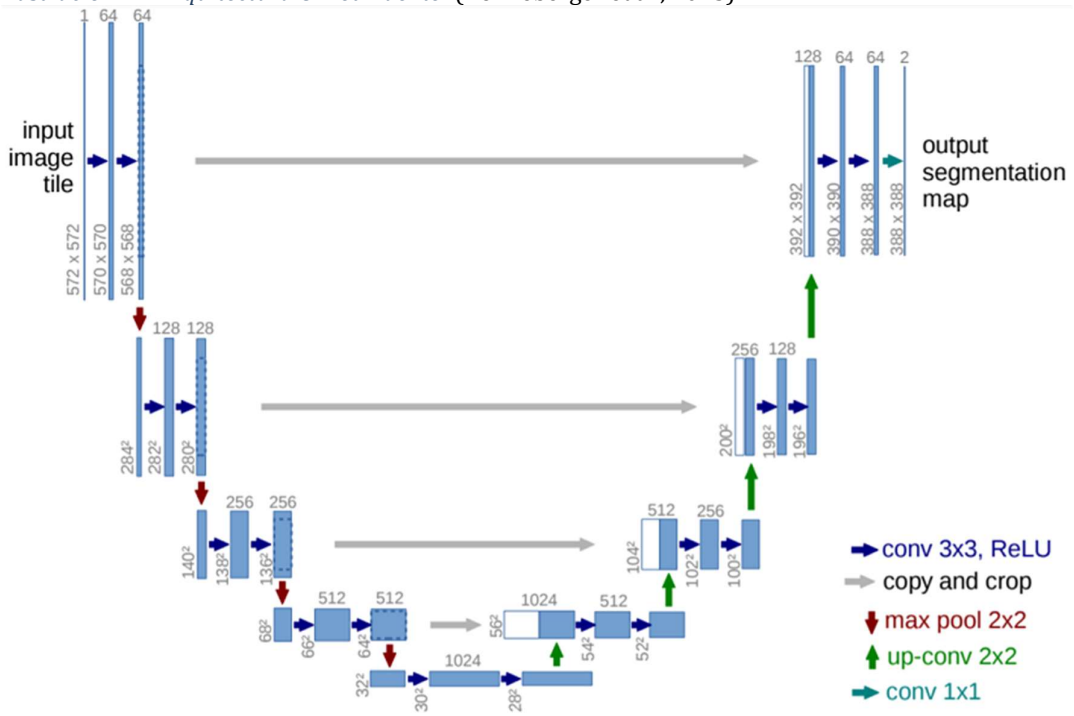
```
-Número de carpetas del dataset:  2
-Número de imágenes:  4291
-Número de imágenes etiquetadas:  4291
--------------------------------------------------------------------------
Pixels imagen 0:
0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [157 167 175]
1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [156 166 174]
Pixels label 0:
0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [52 52 52]
1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [52 52 52]
--------------------------------------------------------------------------
Pixels imagen 1:
0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [152 167 172]
1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [147 162 167]
Pixels label 1:
0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [52 52 52]
1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [52 52 52]
--------------------------------------------------------------------------
Pixels imagen 2:
0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [152 167 170]
1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [152 167 170]
Pixels label 2:
0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [52 52 52]
1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [52 52 52]
--------------------------------------------------------------------------
```

El segundo requisito deriva de la propia arquitectura del modelo U-Net, en el que existen 4 pasos de compresión seguidos de otros 4 pasos de descompresión. En todos estos pasos, se aplica una capa de agrupación/des agrupación (max pooling/ up pooling) 2x2. En la Ilustración 32 se muestra un ejemplo explicativo de la arquitectura del modelo U-Net. Cada rectángulo azul corresponde con un mapa de características multicanal. El número de canales está anotado en la parte superior del rectángulo. El tamaño x-y se muestra en la esquina inferior izquierda de cada rectángulo. Los rectángulos blancos representan mapas de características copiados y las flechas indican las diferentes operaciones entre los mapas de características.

En términos prácticos, esto significa que las dimensiones de nuestras imágenes (ancho y alto) medidas en número de pixeles han de ser divisibles por 2 al menos 4 veces. Las dimensiones de las imágenes del dataset A2D2 de Audi AG® son 1208x1920 por lo que, durante la etapa de procesado y antes del entrenamiento del modelo, se realiza un recorte de 8 pixeles de ancho y 320 pixeles de alto para obtener una imagen de 1200x1600. Para realizar esta operación se utiliza el método de

TensorFlow tf.image.crop_to_boundary box (TensorFlow Github, 2023) que recorta una imagen dentro de una ventana defina. Seguidamente, y para mejorar el tiempo de computación se aplica un escalado hasta llegar a 162x256 pixeles. El método utilizado en el escalado es nearest para evitar distorsión. La operación tf.image.resize de TensorFlow permite la ejecución de este escalado. Las transformaciones descritas en este segundo tipo no suponen un tiempo computacional elevado por lo que se prefiere aprovechar las ventajas de la integración de esta transformación en la secuencia de entrenamiento usando la librerías de TensorFlow mencionadas.

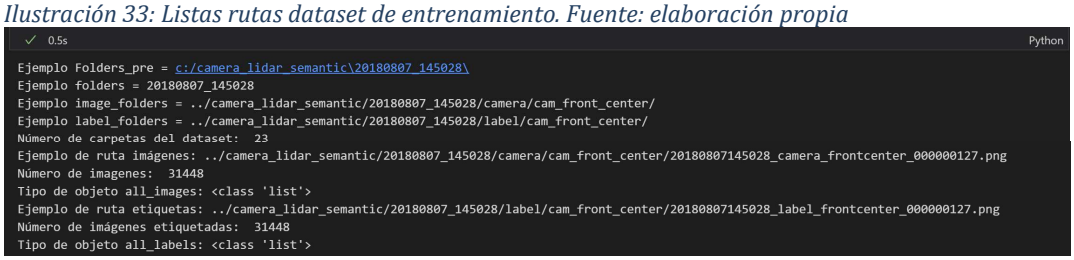
Ilustración 32: Arquitectura U-Net. Fuente: (Ronneberger et al., 2015)



En cuanto a la carga de archivos para el modelo, se debe indicar las rutas de carga de cada una de las imágenes y sus correspondientes etiquetas para que la librería cv2 pueda transfórmalas en arrays. Para ello se utiliza las librerías os que permite ejecutar instrucciones del sistema operativo y el módulo glob que permite mapear directorios a la par que capturar los nombres de los archivos y carpetas contenidos en los mismos (Van Rossum & Drake, 2009). Mediante bucles for se construyen las diferentes

rutas de las imágenes y las etiquetas para almacenarlas en sendos objetos lista que luego se usan para cargar las imágenes. La Ilustración 33 muestra las 23 escenas almacenadas en 23 carpetas (folders) con un total de 31448 imágenes con sus correspondientes 31448 etiquetas que serán usadas para el entrenamiento de las red U-Net.

Ilustración 33: Listas rutas dataset de entrenamiento. Fuente: elaboración propia



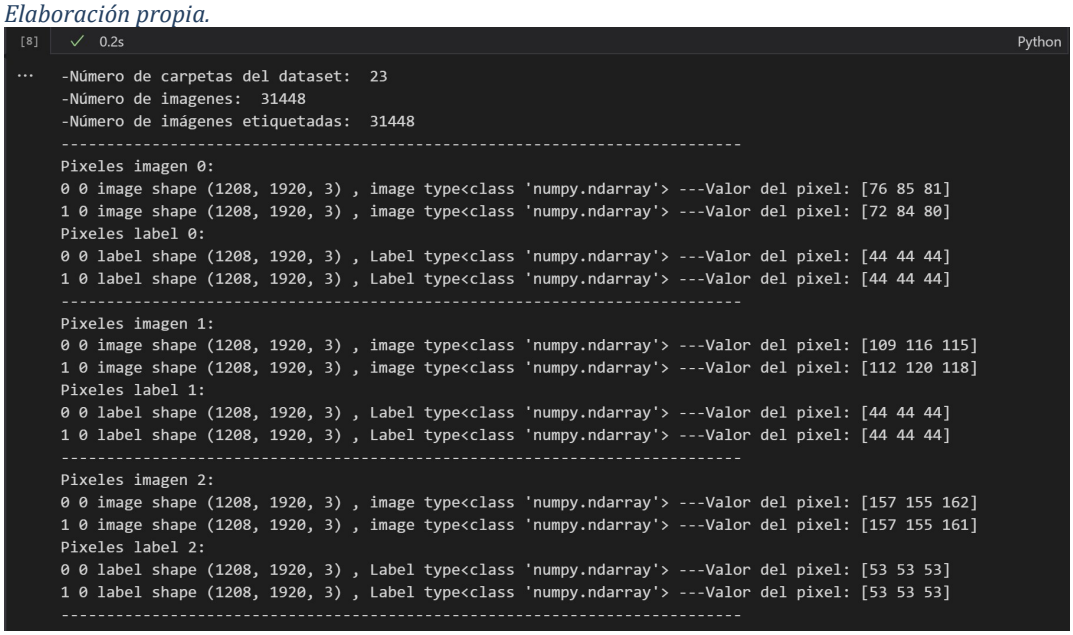
Una vez las rutas están construidas, se utilizan para cargar las imágenes mediante una función que utiliza la librería cv2 que, transforma ficheros en formato imagen en arrays que contienen el valor de los pixeles y que se manejan con la librería numpy . La librería cv2, por defecto, carga las imágenes en formato de color BRG por lo que hay que combinar el método .imread con el método .cvtcolor añadiendo el parámetro .COLOR_BRG2RGB para que la carga de la imagen en el array sea en formato RGB. Seleccionando aleatoriamente 3 imágenes del dataset, se confirman las dimensiones de las imágenes, las etiquetas, así como el espacio de color y la asignación de la correspondiente etiqueta dentro del rango de 1 a 55. Se puede ver este paso en la Ilustración 34.

Usando el método de TensorFlow tf.data.Dataset.list_files y tf.data.Dataset.zip se convierten las rutas en un tensor para poder alimentar el modelo. El resultado de la carga de datos para 2 imágenes y sus correspondientes etiquetas se muestra en la Ilustración 35.

Es momento ahora de ejecutar el recorte y re-escalado de las imágenes lo cual se aprovecha para transformar el valor de los pixeles de las etiquetas del formato [etiqueta_clase, etiqueta_clase, etiqueta_clase] a valores únicos seleccionado el máximo de ellos que, en la práctica equivale la etiqueta de la clase, ya que todos los valores de la tupla son iguales. La

Ilustración 36 muestra el resultado de la transformación que ha recortado y redimensionado cada imagen y cada etiqueta a 192 x 256 pixeles con 3 dimensiones para las imágenes y 1 dimensión (la correspondiente a la etiqueta de la clase).

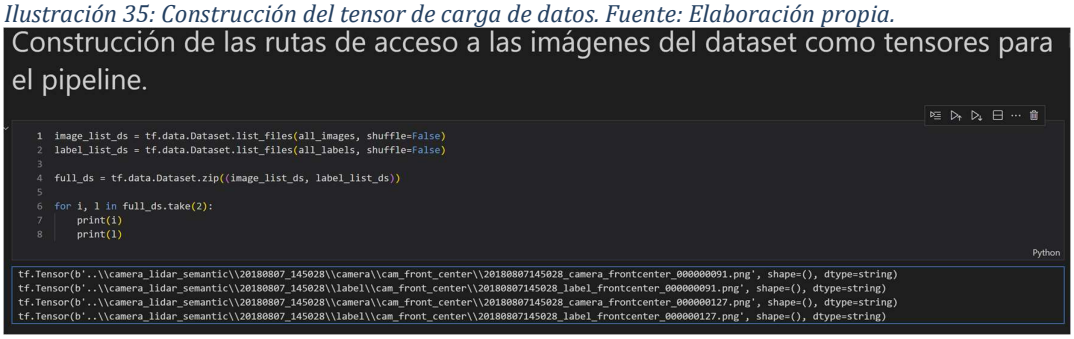
Ilustración 34: Extracto de valor de pixeles y etiquetas usados para el entrenamiento. Fuente: Elaboración propia.



```
[8]  ✓ 0.2s                                                                    Python

...  -Número de carpetas del dataset:  23
     -Número de imagenes:  31448
     -Número de imágenes etiquetadas:  31448
     ------------------------------------------------------------------------
     Pixeles imagen 0:
     0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [76 85 81]
     1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [72 84 80]
     Pixeles label 0:
     0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [44 44 44]
     1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [44 44 44]
     ------------------------------------------------------------------------
     Pixeles imagen 1:
     0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [109 116 115]
     1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [112 120 118]
     Pixeles label 1:
     0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [44 44 44]
     1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [44 44 44]
     ------------------------------------------------------------------------
     Pixeles imagen 2:
     0 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [157 155 162]
     1 0 image shape (1208, 1920, 3) , image type<class 'numpy.ndarray'> ---Valor del pixel: [157 155 161]
     Pixeles label 2:
     0 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [53 53 53]
     1 0 label shape (1208, 1920, 3) , Label type<class 'numpy.ndarray'> ---Valor del pixel: [53 53 53]
     ------------------------------------------------------------------------
```

Ilustración 35: Construcción del tensor de carga de datos. Fuente: Elaboración propia.



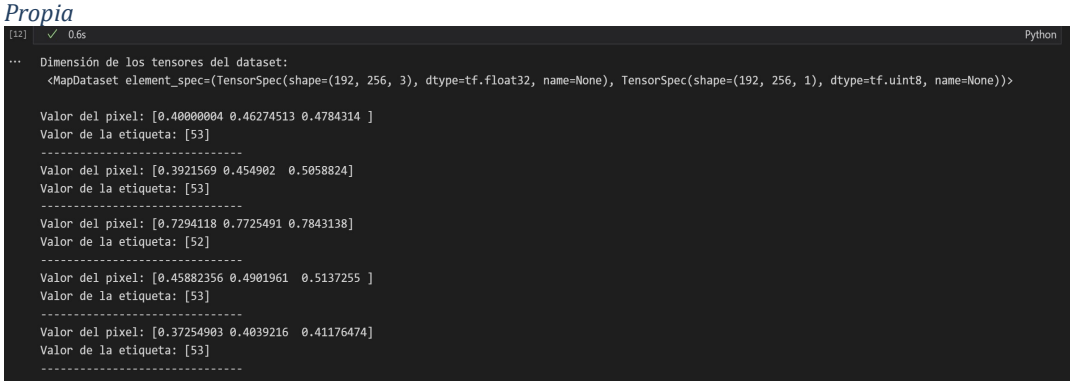
Construcción de las rutas de acceso a las imágenes del dataset como tensores para el pipeline.

```
1  image_list_ds = tf.data.Dataset.list_files(all_images, shuffle=False)
2  label_list_ds = tf.data.Dataset.list_files(all_labels, shuffle=False)
3
4  full_ds = tf.data.Dataset.zip((image_list_ds, label_list_ds))
5
6  for i, l in full_ds.take(2):
7      print(i)
8      print(l)
```

Python

```
tf.Tensor(b'..\\camera_lidar_semantic\\20180807_145028\\camera\\cam_front_center\\20180807145028_camera_frontcenter_000000091.png', shape=(), dtype=string)
tf.Tensor(b'..\\camera_lidar_semantic\\20180807_145028\\label\\cam_front_center\\20180807145028_label_frontcenter_000000091.png', shape=(), dtype=string)
tf.Tensor(b'..\\camera_lidar_semantic\\20180807_145028\\camera\\cam_front_center\\20180807145028_camera_frontcenter_000000127.png', shape=(), dtype=string)
tf.Tensor(b'..\\camera_lidar_semantic\\20180807_145028\\label\\cam_front_center\\20180807145028_label_frontcenter_000000127.png', shape=(), dtype=string)
```

Ilustración 36: Dimensiones de imágenes y valores de pixeles tras transformación. Fuente: Elaboración Propia



```
[12]  ✓ 0.6s                                                                   Python

...  Dimensión de los tensores del dataset:
      <MapDataset element_spec=(TensorSpec(shape=(192, 256, 3), dtype=tf.float32, name=None), TensorSpec(shape=(192, 256, 1), dtype=tf.uint8, name=None))>

     Valor del pixel: [0.40000004 0.46274513 0.4784314 ]
     Valor de la etiqueta: [53]
     ------------------------------
     Valor del pixel: [0.3921569 0.454902  0.5058824]
     Valor de la etiqueta: [53]
     ------------------------------
     Valor del pixel: [0.7294118 0.7725491 0.7843138]
     Valor de la etiqueta: [52]
     ------------------------------
     Valor del pixel: [0.45882356 0.4901961  0.5137255 ]
     Valor de la etiqueta: [53]
     ------------------------------
     Valor del pixel: [0.37254903 0.4039216  0.41176474]
     Valor de la etiqueta: [53]
     ------------------------------
```

A modo de comprobación se realiza un conteo de los valores únicos de las etiquetas para confirmar que después de la transformación, los valores de las etiquetas están dentro del rango esperado. La comprobación arroja 53 valores únicos siendo las clases faltantes, en el conjunto de 55 clases iniciales, la etiquetada como Traffic signal 3 (que tiene el valor hexadecimal de #3c1c64 y etiqueta 20) y la etiquetada como Ego car (el coche en el que se montan los sensores de captura, que tiene el valor hexadecimal de #48d1cc y la etiqueta 47). Para este conteo se utiliza de nuevo la librería numpy usando el método numpy.unique. La salida del código de comprobación se muestra en la Ilustración 37.

Ilustración 37: Comprobación clases tras transformación. Fuente: Elaboración propia.

Se visualizan algunos ejemplos de las imágenes que utilizaremos para el entrenamiento, así como sus etiquetas utilizando de la librería pyplot de matplotlib (Hunter, 2007) y usando el método imshow() que muestra datos como una imagen 2D coloreada en función del valor del dato. En la Ilustración 38 se encuentra la visualización mencionada en la que se aprecia como cada uno de los objetos de la imagen inicial se corresponde con un color diferente dependiendo de la significancia del mismo en términos de interpretación del entorno para la tarea de segmentación semántica.

Ilustración 38: Ejemplos imagen vs etiqueta utilizados para el entrenamiento. Fuente: Elaboración propia.

3.3. Segmentación de los datos y construcción del modelo de red U-Net

Para el entrenamiento de la red neuronal U-Net se dividen los datos en tres grupos. EL primero tiene el 70% del dataset y lo se usa para el entrenamiento. El segundo su usa para la validación durante el entrenamiento y contiene el 20% del dataset. Por último, se utiliza un 10% del dataset como datos de test para la evaluación del rendimiento del modelo con datos nunca usados en el entrenamiento. La mezcla de las imágenes ligadas a sus correspondientes etiquetas se realiza utilizando el método shuffle() que reasigna el orden de los elementos dentro de una secuencia de datos (Van Rossum & Drake, 2009). Se asigna al método suffle() un tamaño igual a la dimensión del dataset para asegurar la homogeneidad del mezclado.

Se construye, a continuación, el modelo de Red neuronal U-Net aplicando ciertas modificaciones a este modelo por lo que se recuerda brevemente esta arquitectura para explicar las modificaciones realizadas.

La arquitectura U-Net clásica (Ronneberger et al., 2015), como se aprecia en la Ilustración 32, consiste en una codificación (parte izquierda) y una decodificación (parte derecha). La codificación consiste en la aplicación repetitiva de 2 capas convolucionales 3x3 sin relleno (padding) seguidas de una capa con función de activación ReLU. Tras este grupo se produce una operación de max pooling tamaño 2x2 con paso 2 para reducir las resolución a la mitad en cada eje del fotograma tomando los valores máximos de los datos de entrada en cada paso (downsampling) a la vez que duplicamos el número de canales. Cada paso en la etapa de decodificación se aumenta la resolución al doble (upsampling) a la par que dividimos el número de canales a la mitad. Para ello se utiliza una capa de-convolucional 2x2. Es necesaria una concatenación entre el nivel de codificación y el de descodificación usando el mapa de características de la tapa de codificación. Para terminar se aplican 2 capas convolucionales

3x3 seguidas por una función de activación ReLU cada una. Al final se utiliza una capa convolucional 1x1 para mapear los 64 componentes del mapa de características al número de clases deseado.

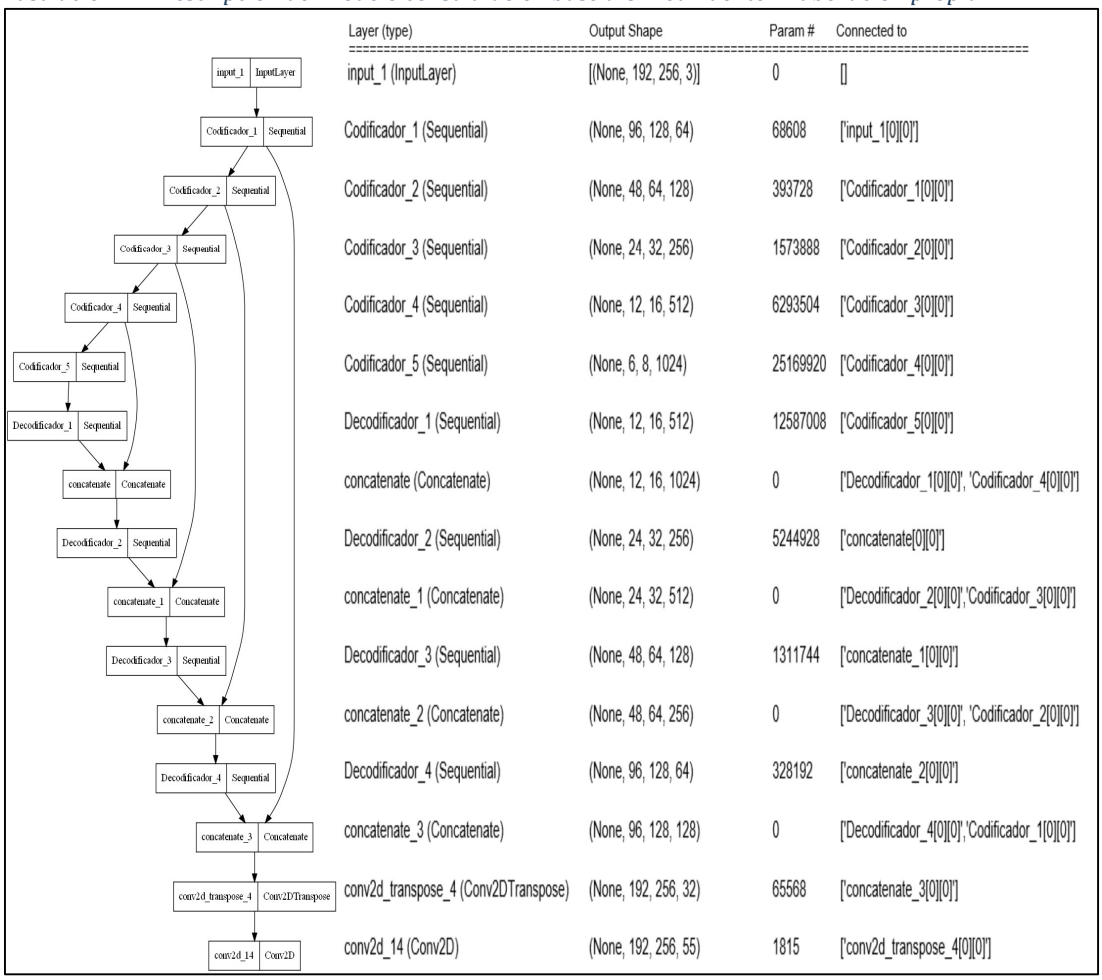
A continuación, se describen las modificaciones aplicadas al modelo clásico de U-Net.

- El tamaño de los filtros de las capas convolucionales se dimensiona en 2x2 en lugar de 3x3.
- Se utiliza relleno con ceros (padding)
- Se establece un paso de 2,2 para la red convolucional de la que entran los datos de la etapa anterior. Con esto conseguimos reducir la resolución del mapa de características a la mitad sin necesidad de aplicar maxpooling.
- A la salida de las etapas de codificación se añade una capa BatchNormalization para mantener la media de los valores de salida cercana a 0 y la desviación típica cercana a 1.
- Entre cada de decodificación, se añade una capa Batchnomalization y al final de cada etapa de decodificación una capa de dropout con una frecuencia del 50% la cual aleatoriamente anula los inputs a 0 con la frecuencia definida. Esto ayuda a prevenir el overfitting.
- Para todas las capas de codificación descodificación (excepto la última de decodificación) se establece una inicialización aleatoria sin semilla para los pesos con media 0 y desviación típica 0.02. Así mismo se excluya la utilización de BIAS en todas ellas.
- Para la última capa se utiliza un inicializador he_normal() que inicializa los pesos a una distribución normal centrada en 0 con deviación estándar igual a la raíz cuadrada de 2 dividido por el número de entradas del tensor. Los bias no están anulados en esta última capa.

Para la construcción del modelo la utilizan las capas existentes en la biblioteca keras de TensorFlow (Tensorflow Github, 2023) como son:

keras.Sequencial(), keras.layers.Conv2D, keras.BatchNormalization(), keras.layers.relu(), keras.Conv2DTranspose(), keras.Dropout(), keras.layers.Input(), keras.layers.Concatenate(). Con funciones de Python (Van Rossum & Drake, 2009) se definen las ramas de compresión y descompresión y con bucles for, se unen las diferentes capas para crear la arquitectura. El método keras.Model() permite construir el modelo. El modelo resultante contiene 53.038.903 de parámetros de los cuales solo 7680 son se entrenan. En la Ilustración 39 se muestra, tanto de forma esquemática como explicita el modelo construido.

Ilustración 39: Descripción del modelo construido en base a U-Net. Fuente: Elaboración propia



Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 192, 256, 3)]	0	[]
Codificador_1 (Sequential)	(None, 96, 128, 64)	68608	['input_1[0][0]']
Codificador_2 (Sequential)	(None, 48, 64, 128)	393728	['Codificador_1[0][0]']
Codificador_3 (Sequential)	(None, 24, 32, 256)	1573888	['Codificador_2[0][0]']
Codificador_4 (Sequential)	(None, 12, 16, 512)	6293504	['Codificador_3[0][0]']
Codificador_5 (Sequential)	(None, 6, 8, 1024)	25169920	['Codificador_4[0][0]']
Decodificador_1 (Sequential)	(None, 12, 16, 512)	12587008	['Codificador_5[0][0]']
concatenate (Concatenate)	(None, 12, 16, 1024)	0	['Decodificador_1[0][0]', 'Codificador_4[0][0]']
Decodificador_2 (Sequential)	(None, 24, 32, 256)	5244928	['concatenate[0][0]']
concatenate_1 (Concatenate)	(None, 24, 32, 512)	0	['Decodificador_2[0][0]','Codificador_3[0][0]']
Decodificador_3 (Sequential)	(None, 48, 64, 128)	1311744	['concatenate_1[0][0]']
concatenate_2 (Concatenate)	(None, 48, 64, 256)	0	['Decodificador_3[0][0]', 'Codificador_2[0][0]']
Decodificador_4 (Sequential)	(None, 96, 128, 64)	328192	['concatenate_2[0][0]']
concatenate_3 (Concatenate)	(None, 96, 128, 128)	0	['Decodificador_4[0][0]','Codificador_1[0][0]']
conv2d_transpose_4 (Conv2DTranspose)	(None, 192, 256, 32)	65568	['concatenate_3[0][0]']
conv2d_14 (Conv2D)	(None, 192, 256, 55)	1815	['conv2d_transpose_4[0][0]']

3.4. Definición de métricas de evaluación

Se utiliza la métrica de precisión (accuracy) ya que es una métrica muy extendida para la evaluación de modelos de clasificación, sin embargo,

para tareas de segmentación semántica, no proporciona una representación exacta del rendimiento en esta tarea en muchos casos como ya se explicó en el capítulo 2.2.8 y que brevemente se resume de nuevo a continuación.

Dado que segmentación consiste en asignar una etiqueta de clase a cada píxel de una imagen, requiere una delineación precisa de los límites y una localización exacta de los objetos. El rendimiento del modelo viene calificado por la capacidad del mismo de conseguir estos 2 objetivos.

En muchos escenarios del mundo real, la clase de fondo supera ampliamente a la de primer plano. En consecuencia, un modelo que prediga sólo la clase mayoritaria (fondo) alcanzaría una gran precisión, aunque no captara los detalles y complejidades de los objetos en primer plano. Este hecho pone de manifiesto la limitación de la precisión en la segmentación semántica debido a la sensibilidad de la métrica de precisión al desequilibrio de clases.

Es por esta razón por lo que, además de la métrica accuracy se utiliza el coeficiente Dice (Reuben R Shamir, 2018) y el coeficiente IoU también conocido como Jaccard index (Taha & Hanbury, 2015). Estos dos coeficiente son los más ampliamente usados para la evaluación del rendimiento de la tarea de segmentación semántica.

A fecha de realización este trabajo la biblioteca de TensorFlow tf.keras.metrics no cuenta con la métrica del coeficiente Dice (Tensorflow Github, 2023) así que se ha definido, mediante código, tanto el coeficiente Dice como el coeficiente IoU. Para conseguirlo, primero se convierten las etiquetas de las imágenes en una representación de un solo punto utilizando la técnica de one hot encoding mediante el método tf.one_hot(). Después, se obtienen los logits de las predicciones utilizando tf.argmax() para el coeficiente Dice y las convertiremos también en one-hot(). Para el coeficiente IoU usamos, el método tf.keras.activation.softmax() escogiendo aquellas cuyo valor resultante está por encima de 0.5.

En TensorFlow, el término logits se refiere a la salida de una capa de red neuronal antes de aplicar una función de activación (Martín~Abadi et al., 2015). Los logits son los valores numéricos sin procesar que representan las puntuaciones o probabilidades asociadas con las clases de salida en un modelo de clasificación (en este caso las capas convolucionales y de-convolucionales).

Los logits se utilizan comúnmente en problemas de clasificación multiclase, donde cada clase tiene asignada una puntuación o probabilidad. Por lo general, los logits no están normalizados y pueden tomar cualquier valor real. Estos valores se utilizan como entrada para una función de activación, como la función softmax, que convierte los logits en probabilidades que suman 1 y representan la distribución de probabilidad sobre las clases de salida.

La ventaja de trabajar con logits en lugar de probabilidades directamente es que los logits son numéricamente más estables durante el proceso de optimización, ya que evitan problemas de cálculo asociados con los valores de probabilidad extremos cercanos a 0 o 1.

Para entender cómo calcular los coeficientes Dice e IoU a partir de tensores de TensorFlow, primero se ha de entender con qué dimensiones estamos tratando.

En este caso, la forma de las etiquetas reales (y_true) y las predichas (y_pred), después de aplicar 'one-hotting', se ven como [None, 192, 256, 13]. La primera dimensión representa el lote, la segunda y la tercera - dimensiones de la imagen, y la cuarta representa la cantidad de clases distintas en la etiqueta. Por lo tanto, para calcular el número de elementos de cada clase, tenemos que reducir la suma de todos los ejes excepto el cuarto, el de las etiquetas. Esto se hace mediante el método tf.reduce_sum().

El coeficiente Dice tiene en cuenta tanto la intersección como la unión de las regiones predichas y las de la verdad sobre el terreno, proporcionando

una medida de solapamiento o similitud entre ellas. Al considerar la concordancia entre las segmentaciones predichas y las verdaderas, el coeficiente Dice ofrece una evaluación más matizada del rendimiento. El coeficiente de similitud de Dice se calcula mediante la formula descrita en la Ecuación 3 que se menciona de nuevo a continuación.

Ecuación 5: Coeficiente Dice

$$\boldsymbol{Dice(A,B)} = \frac{\boldsymbol{2x|A \cap B|}}{\boldsymbol{|A| + |B|}}$$

El número resultante oscila entre 0 y 1: 0 significa que los conjuntos son completamente diferentes y 1 que son completamente similares. Como en este caso tenemos varias clases que predecir, se calcula el coeficiente para cada clase y halla la media entre ellas.

Para obtener la intersección (numerador de la formula), se multiplica por elementos los tensores y_true e y_pred. A continuación, se cuenta los elementos de intersección para cada clase utilizando el método tf.reduce_sum(). Para el denominador de la fórmula, se cuentan los elementos de cada clase en los tensores y_true e y_pred por separado utilizando tf.reduce_sum(). Finalmente, se calcula el coeficiente Dice añadiendo un número pequeño tanto al numerador como al denominador (para evitar la división por cero), y se calcula la media usando el método tf.reduce_mean().

El coeficiente IoU se calcula comparando el área de intersección entre dos regiones (la región predicha y la región de referencia) dividida por el área de unión de esas regiones. Matemáticamente, se expresa como indica la Ecuación 4 que se visualiza de nuevo a continuación.

Ecuación 6: Coeficiente IoU.

$$\boldsymbol{IoU(A,B)} = \frac{\boldsymbol{|A \cap B|}}{\boldsymbol{|A| \cap |B|}}$$

El resultado del coeficiente IoU varía entre 0 y 1, donde 0 indica una falta de superposición entre las regiones y 1 indica una coincidencia perfecta entre las regiones.

Para los cálculos de numerador y denominador se utiliza la misma estrategia que para el coeficiente Dice, pero para el cálculo de la media utilizamos el método tf.experimental.numpy.nanmean() que permite calcular la media del coeficiente para cada pareja real-predicción descartando los valores nan.

3.5. Compilación del modelo y flujo de datos

Para la compilación del modelo construido, se usa el método utilizamos el método model.compile() con los siguientes parámetros.

- Optimizador Adam con tasa de aprendizaje 0.001
- Función de pérdida: SparseCategoricalCrossentropy() con entrada de logits
- Métricas: Accuracy, Índice Dice e Índice IoU
- Batch size: 16. Este parámetro se refiere al número de ejemplos de entrenamiento que se procesan simultáneamente durante una iteración de entrenamiento. Debido a la memoria del sistema utilizado se fija este valor en 16 ya que un valor por encima de este da como resultado un error de saturación de recursos.

En la configuración del flujo de datos se trata de optimizar el máximo el rendimiento mediante la utilización de los datos previamente almacenados en búfer de esta forma se podrá obtener datos del disco sin que la entada o salida se convierta en un cuello de botella. Para ellos se usa el método Dataset.cache() que mantiene las imágenes en la memoria después de que se cargan fuera del disco durante la primera epoch. El conjunto de datos es demasiado grande para caber en la memoria, así que este método crea un caché en disco de alto rendimiento. Con la utilización adicional del método Dataset.prefetch() se superpone el preprocesamiento de datos y la ejecución del modelo durante el entrenamiento para utilizar todas las capacidades de procesado posibles.

3.6. Definición de callbacks y entrenamiento del modelo

En TensorFlow, los callbacks son objetos que permiten personalizar y controlar el comportamiento del entrenamiento de un modelo de machine learning durante su proceso de entrenamiento. Los callbacks son utilizados para realizar acciones específicas en momentos predeterminados durante el entrenamiento, como guardar checkpoints del modelo, detener el entrenamiento tempranamente si se cumplen ciertas condiciones, y más. En este caso se usa tf.keras.callbacks.EarlyStopping() para parar el entrenamiento es en caso de que no haya mejora de la función de pérdida de los datos de validación en 5 epoch consecutivos. Además, se usa tf.keras.callbacks.ModelCheckpoint() para guardar el estado del modelo salvándolo sólo en el caso de que la función de pérdida de los datos de validación es mejor que en el epoch anterior.

El entrenamiento del modelo se realiza mediante el método model.fit con una cantidad de epoch programados de 100. Su resultado se muestra en la Ilustración 42. En esta ilustración, que se ve como del epoch 84 al 88 no hubo mejora de val_loss por lo que se interrumpió el entrenamiento al existir 5 epoch consecutivos sin mejora de esta métrica como se programó en el callback correspondiente. El tiempo de entrenamiento fue de 15 horas, 20 minutos y 8 segundos. Se puede ver este resumen de resultados en la Ilustración 40.

Ilustración 40: Resumen entrenamiento red U-Net con dataset A2D2. Fuente: Elaboración propia.

```
Tiempo de entrenamiento: 15.0 Horas 20.0 Minutos 8 Segundos
Número de Epoch programados para el entrenamiento = 100
Número de Epoch ejecutados = 88
Mejor Epoch:  88
```

3.7. Evaluación del modelo U-Net entrenado con el dataset A2D2 de Audi AG®

Para la evaluación del modelo se escogerá aquel epoch en el que la función de pérdida de los datos de validación alcance el valor mínimo. Para ello nos valemos del método tf.keras.callbacks.ModelCheckpoint() que, como vimos en el punto anterior guarda el modelo en el caso que el valor

de la función de perdida de los datos de validación sea mejor que el epoch anterior. Para ello, una vez finalizado el entrenamiento, cargamos el modelo almacenado en la ruta definida en el callback mediante el método tf.keras,models.load_model(). Para le evaluación de los datos de test utilizamos el model.evaluate() pasándole como parámetro el dataset de test que contiene el 10% de los datos del dataset que no han sido vistos durante el entrenamiento.

Los resultados del entrenamiento se resumen en la Ilustración 42 y los de la evaluación se muestran en la Ilustración 41 en la que se observa como los valores de las métricas utilizadas para los datos de validación del entrenamiento del modelo (val_accuracy, val_DiceMetric, val_IoU) no difieren significativamente de los valores obtenidos en la evaluación de los datos de test. (test_accuracy, test_DiceMetric, test_IoU). Se infiere en base a estas observaciones que el modelo no está sobreentrenado, no hay overfitting.

Ilustración 41: Resultados evaluación U-Net entrenado con A2D2. Fuente: Elaboración propia.

```
···   197/197 [==============================] - 630s 192ms/step - loss: 0.5343 - accuracy: 0.8407 - DiceMetric: 0.4903 - IoU: 0.1667

      Estadísticas del mejor epoch:
      loss - 0.009 accuracy - 0.996 DiceMetric - 0.604 IoU - 0.541 val_loss - 0.017 val_accuracy - 0.995 val_DiceMetric - 0.597 val_IoU - 0.509
      ------------------------------------------
      Estadísticas evaluation de test:
      test_loss: - 0.019   test_accuracy - 0.991   test_DiceMetric - 0.592   test_IoU - 0.501
```

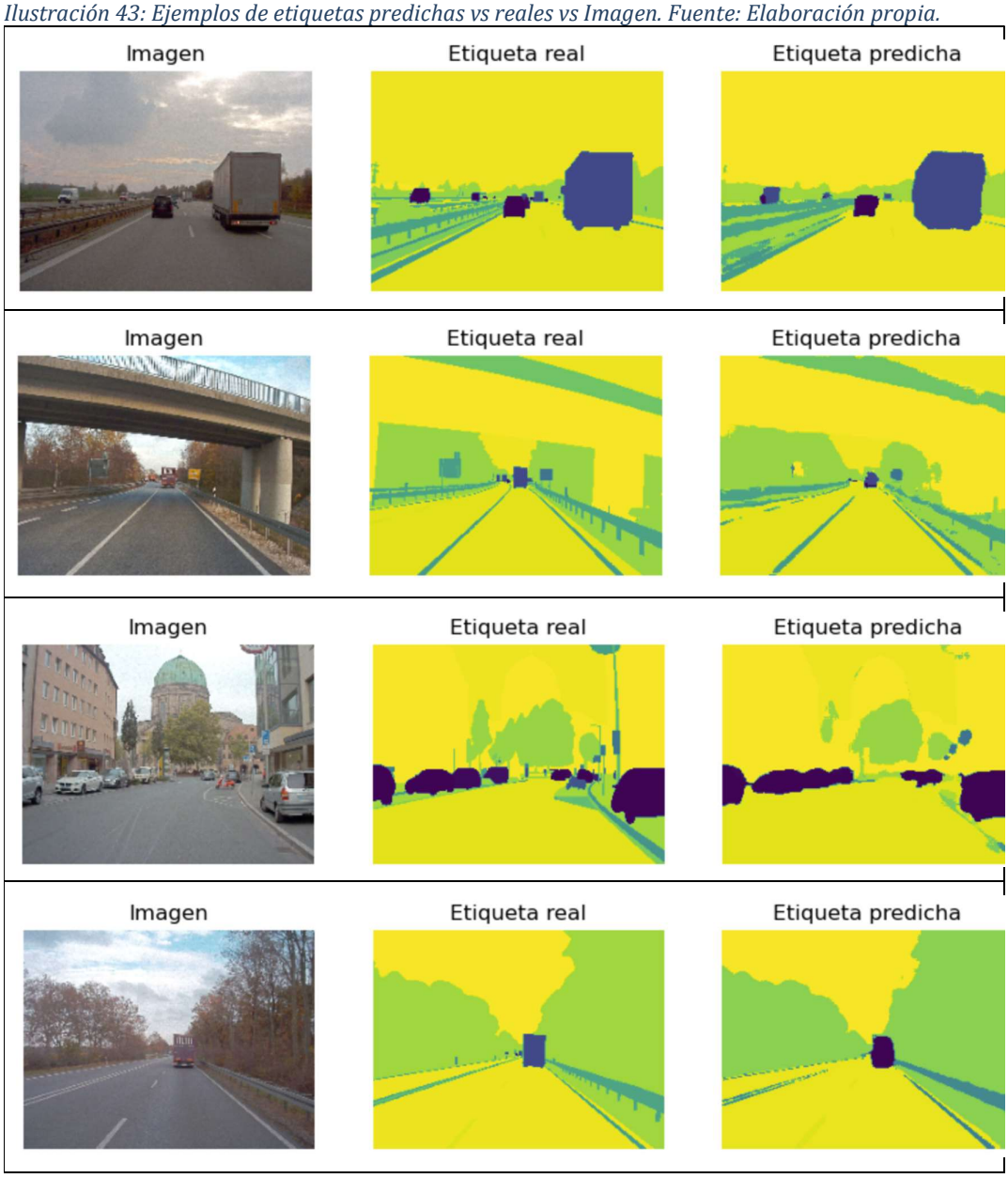
El tiempo de evaluación es de 630 segundos para 3146 imágenes con lo que se obtiene un tiempo de evaluación de 192 milisegundos por imagen. Se pueden visualizar algunos ejemplos de la evaluación del modelo con los datos de test comparando su etiqueta original con la etiqueta predicha por el modelo tal y como se muestra en la Ilustración 43.

Ilustración 42: Resultados entrenamiento U-Net con dataset A2D2. Fuente: elaboración propia

```
Tiempo de entrenamiento: 15.0 Horas 20.0 Minutos 8 Segundos
Número de Epoch programados para el entrenamiento = 100
Número de Epoch ejecutados = 88
Mejor Epoch:  88
Estadísticas del mejor epoch:
loss - 0.009 accuracy - 0.996 DiceMetric - 0.604 IoU - 0.541 val_loss - 0.017 val_accuracy - 0.995 val_DiceMetric - 0.597 val_IoU - 0.509

 Listado de las estadísticas de todos los Epoch
 Epoch 01: loss - 0.708 accuracy - 0.811 DiceMetric - 0.418 IoU - 0.091 val_loss - 0.548 val_accuracy - 0.847 val_DiceMetric - 0.453 val_IoU - 0.130
 Epoch 02: loss - 0.452 accuracy - 0.873 DiceMetric - 0.476 IoU - 0.149 val_loss - 0.412 val_accuracy - 0.882 val_DiceMetric - 0.508 val_IoU - 0.194
 Epoch 03: loss - 0.370 accuracy - 0.894 DiceMetric - 0.514 IoU - 0.196 val_loss - 0.367 val_accuracy - 0.894 val_DiceMetric - 0.537 val_IoU - 0.233
 Epoch 04: loss - 0.325 accuracy - 0.906 DiceMetric - 0.539 IoU - 0.229 val_loss - 0.314 val_accuracy - 0.909 val_DiceMetric - 0.555 val_IoU - 0.260
 Epoch 05: loss - 0.151 accuracy - 0.910 DiceMetric - 0.544 IoU - 0.259 val_loss - 0.310 val_accuracy - 0.914 val_DiceMetric - 0.555 val_IoU - 0.263
 Epoch 06: loss - 0.126 accuracy - 0.925 DiceMetric - 0.544 IoU - 0.276 val_loss - 0.257 val_accuracy - 0.929 val_DiceMetric - 0.556 val_IoU - 0.279
 Epoch 07: loss - 0.108 accuracy - 0.936 DiceMetric - 0.545 IoU - 0.289 val_loss - 0.220 val_accuracy - 0.939 val_DiceMetric - 0.556 val_IoU - 0.292
 Epoch 08: loss - 0.094 accuracy - 0.944 DiceMetric - 0.546 IoU - 0.299 val_loss - 0.192 val_accuracy - 0.947 val_DiceMetric - 0.557 val_IoU - 0.301
 Epoch 09: loss - 0.084 accuracy - 0.950 DiceMetric - 0.547 IoU - 0.307 val_loss - 0.170 val_accuracy - 0.953 val_DiceMetric - 0.557 val_IoU - 0.309
 Epoch 10: loss - 0.075 accuracy - 0.956 DiceMetric - 0.547 IoU - 0.314 val_loss - 0.153 val_accuracy - 0.958 val_DiceMetric - 0.557 val_IoU - 0.315
 Epoch 11: loss - 0.069 accuracy - 0.960 DiceMetric - 0.548 IoU - 0.320 val_loss - 0.139 val_accuracy - 0.961 val_DiceMetric - 0.558 val_IoU - 0.321
 Epoch 12: loss - 0.063 accuracy - 0.963 DiceMetric - 0.549 IoU - 0.325 val_loss - 0.127 val_accuracy - 0.965 val_DiceMetric - 0.558 val_IoU - 0.325
 Epoch 13: loss - 0.058 accuracy - 0.966 DiceMetric - 0.549 IoU - 0.330 val_loss - 0.117 val_accuracy - 0.967 val_DiceMetric - 0.559 val_IoU - 0.330
 Epoch 14: loss - 0.054 accuracy - 0.969 DiceMetric - 0.550 IoU - 0.334 val_loss - 0.108 val_accuracy - 0.970 val_DiceMetric - 0.559 val_IoU - 0.334
 Epoch 15: loss - 0.050 accuracy - 0.971 DiceMetric - 0.551 IoU - 0.338 val_loss - 0.101 val_accuracy - 0.972 val_DiceMetric - 0.560 val_IoU - 0.337
 Epoch 16: loss - 0.047 accuracy - 0.973 DiceMetric - 0.551 IoU - 0.342 val_loss - 0.095 val_accuracy - 0.974 val_DiceMetric - 0.560 val_IoU - 0.341
 Epoch 17: loss - 0.044 accuracy - 0.974 DiceMetric - 0.552 IoU - 0.345 val_loss - 0.089 val_accuracy - 0.975 val_DiceMetric - 0.561 val_IoU - 0.344
 Epoch 18: loss - 0.042 accuracy - 0.976 DiceMetric - 0.553 IoU - 0.348 val_loss - 0.084 val_accuracy - 0.977 val_DiceMetric - 0.561 val_IoU - 0.347
 Epoch 19: loss - 0.040 accuracy - 0.977 DiceMetric - 0.553 IoU - 0.351 val_loss - 0.079 val_accuracy - 0.978 val_DiceMetric - 0.562 val_IoU - 0.349
 Epoch 20: loss - 0.038 accuracy - 0.978 DiceMetric - 0.554 IoU - 0.354 val_loss - 0.075 val_accuracy - 0.979 val_DiceMetric - 0.562 val_IoU - 0.352
 Epoch 21: loss - 0.036 accuracy - 0.979 DiceMetric - 0.555 IoU - 0.357 val_loss - 0.072 val_accuracy - 0.980 val_DiceMetric - 0.563 val_IoU - 0.355
 Epoch 22: loss - 0.034 accuracy - 0.980 DiceMetric - 0.555 IoU - 0.360 val_loss - 0.068 val_accuracy - 0.981 val_DiceMetric - 0.563 val_IoU - 0.357
 Epoch 23: loss - 0.033 accuracy - 0.981 DiceMetric - 0.556 IoU - 0.363 val_loss - 0.065 val_accuracy - 0.982 val_DiceMetric - 0.564 val_IoU - 0.360
 Epoch 24: loss - 0.031 accuracy - 0.982 DiceMetric - 0.557 IoU - 0.365 val_loss - 0.063 val_accuracy - 0.982 val_DiceMetric - 0.564 val_IoU - 0.362
 Epoch 25: loss - 0.030 accuracy - 0.983 DiceMetric - 0.557 IoU - 0.368 val_loss - 0.060 val_accuracy - 0.983 val_DiceMetric - 0.565 val_IoU - 0.364
 Epoch 26: loss - 0.029 accuracy - 0.984 DiceMetric - 0.558 IoU - 0.370 val_loss - 0.058 val_accuracy - 0.984 val_DiceMetric - 0.565 val_IoU - 0.366
 Epoch 27: loss - 0.028 accuracy - 0.984 DiceMetric - 0.559 IoU - 0.373 val_loss - 0.055 val_accuracy - 0.984 val_DiceMetric - 0.566 val_IoU - 0.369
 Epoch 28: loss - 0.027 accuracy - 0.985 DiceMetric - 0.560 IoU - 0.375 val_loss - 0.053 val_accuracy - 0.985 val_DiceMetric - 0.566 val_IoU - 0.371
 Epoch 29: loss - 0.026 accuracy - 0.985 DiceMetric - 0.560 IoU - 0.378 val_loss - 0.052 val_accuracy - 0.986 val_DiceMetric - 0.567 val_IoU - 0.373
 Epoch 30: loss - 0.025 accuracy - 0.986 DiceMetric - 0.561 IoU - 0.380 val_loss - 0.050 val_accuracy - 0.986 val_DiceMetric - 0.567 val_IoU - 0.375
 Epoch 31: loss - 0.024 accuracy - 0.986 DiceMetric - 0.562 IoU - 0.382 val_loss - 0.048 val_accuracy - 0.986 val_DiceMetric - 0.568 val_IoU - 0.377
 Epoch 32: loss - 0.024 accuracy - 0.987 DiceMetric - 0.562 IoU - 0.385 val_loss - 0.047 val_accuracy - 0.987 val_DiceMetric - 0.568 val_IoU - 0.379
 Epoch 33: loss - 0.023 accuracy - 0.987 DiceMetric - 0.563 IoU - 0.387 val_loss - 0.045 val_accuracy - 0.987 val_DiceMetric - 0.569 val_IoU - 0.381
 Epoch 34: loss - 0.022 accuracy - 0.988 DiceMetric - 0.564 IoU - 0.390 val_loss - 0.044 val_accuracy - 0.988 val_DiceMetric - 0.569 val_IoU - 0.384
 Epoch 35: loss - 0.022 accuracy - 0.988 DiceMetric - 0.564 IoU - 0.392 val_loss - 0.043 val_accuracy - 0.988 val_DiceMetric - 0.570 val_IoU - 0.386
 Epoch 36: loss - 0.021 accuracy - 0.988 DiceMetric - 0.565 IoU - 0.394 val_loss - 0.041 val_accuracy - 0.988 val_DiceMetric - 0.570 val_IoU - 0.388
 Epoch 37: loss - 0.020 accuracy - 0.989 DiceMetric - 0.566 IoU - 0.397 val_loss - 0.040 val_accuracy - 0.989 val_DiceMetric - 0.571 val_IoU - 0.390
 Epoch 38: loss - 0.020 accuracy - 0.989 DiceMetric - 0.567 IoU - 0.399 val_loss - 0.039 val_accuracy - 0.989 val_DiceMetric - 0.571 val_IoU - 0.392
 Epoch 39: loss - 0.019 accuracy - 0.989 DiceMetric - 0.567 IoU - 0.401 val_loss - 0.038 val_accuracy - 0.989 val_DiceMetric - 0.572 val_IoU - 0.394
 Epoch 40: loss - 0.019 accuracy - 0.990 DiceMetric - 0.568 IoU - 0.404 val_loss - 0.037 val_accuracy - 0.990 val_DiceMetric - 0.572 val_IoU - 0.396
 Epoch 41: loss - 0.018 accuracy - 0.990 DiceMetric - 0.569 IoU - 0.406 val_loss - 0.036 val_accuracy - 0.990 val_DiceMetric - 0.573 val_IoU - 0.398
 Epoch 42: loss - 0.018 accuracy - 0.990 DiceMetric - 0.569 IoU - 0.408 val_loss - 0.035 val_accuracy - 0.990 val_DiceMetric - 0.573 val_IoU - 0.400
 Epoch 43: loss - 0.018 accuracy - 0.990 DiceMetric - 0.570 IoU - 0.411 val_loss - 0.035 val_accuracy - 0.990 val_DiceMetric - 0.574 val_IoU - 0.402
 Epoch 44: loss - 0.017 accuracy - 0.991 DiceMetric - 0.571 IoU - 0.413 val_loss - 0.034 val_accuracy - 0.990 val_DiceMetric - 0.574 val_IoU - 0.404
 Epoch 45: loss - 0.017 accuracy - 0.991 DiceMetric - 0.572 IoU - 0.416 val_loss - 0.033 val_accuracy - 0.991 val_DiceMetric - 0.575 val_IoU - 0.406
 Epoch 46: loss - 0.016 accuracy - 0.991 DiceMetric - 0.572 IoU - 0.418 val_loss - 0.032 val_accuracy - 0.991 val_DiceMetric - 0.575 val_IoU - 0.408
 Epoch 47: loss - 0.016 accuracy - 0.991 DiceMetric - 0.573 IoU - 0.420 val_loss - 0.032 val_accuracy - 0.991 val_DiceMetric - 0.576 val_IoU - 0.410
 Epoch 48: loss - 0.016 accuracy - 0.992 DiceMetric - 0.574 IoU - 0.423 val_loss - 0.031 val_accuracy - 0.991 val_DiceMetric - 0.576 val_IoU - 0.412
 Epoch 49: loss - 0.015 accuracy - 0.992 DiceMetric - 0.574 IoU - 0.425 val_loss - 0.030 val_accuracy - 0.991 val_DiceMetric - 0.577 val_IoU - 0.414
 Epoch 50: loss - 0.015 accuracy - 0.992 DiceMetric - 0.575 IoU - 0.428 val_loss - 0.030 val_accuracy - 0.992 val_DiceMetric - 0.577 val_IoU - 0.416
 Epoch 51: loss - 0.015 accuracy - 0.992 DiceMetric - 0.576 IoU - 0.430 val_loss - 0.029 val_accuracy - 0.992 val_DiceMetric - 0.578 val_IoU - 0.419
 Epoch 52: loss - 0.015 accuracy - 0.992 DiceMetric - 0.577 IoU - 0.433 val_loss - 0.028 val_accuracy - 0.992 val_DiceMetric - 0.578 val_IoU - 0.421
 Epoch 53: loss - 0.014 accuracy - 0.992 DiceMetric - 0.577 IoU - 0.435 val_loss - 0.028 val_accuracy - 0.992 val_DiceMetric - 0.579 val_IoU - 0.423
 Epoch 54: loss - 0.014 accuracy - 0.993 DiceMetric - 0.578 IoU - 0.438 val_loss - 0.027 val_accuracy - 0.992 val_DiceMetric - 0.579 val_IoU - 0.425
 Epoch 55: loss - 0.014 accuracy - 0.993 DiceMetric - 0.579 IoU - 0.440 val_loss - 0.027 val_accuracy - 0.992 val_DiceMetric - 0.580 val_IoU - 0.427
 Epoch 56: loss - 0.013 accuracy - 0.993 DiceMetric - 0.580 IoU - 0.443 val_loss - 0.026 val_accuracy - 0.993 val_DiceMetric - 0.580 val_IoU - 0.429
 Epoch 57: loss - 0.013 accuracy - 0.993 DiceMetric - 0.580 IoU - 0.446 val_loss - 0.026 val_accuracy - 0.993 val_DiceMetric - 0.581 val_IoU - 0.432
 Epoch 58: loss - 0.013 accuracy - 0.993 DiceMetric - 0.581 IoU - 0.448 val_loss - 0.025 val_accuracy - 0.993 val_DiceMetric - 0.581 val_IoU - 0.434
 Epoch 59: loss - 0.013 accuracy - 0.993 DiceMetric - 0.582 IoU - 0.451 val_loss - 0.025 val_accuracy - 0.993 val_DiceMetric - 0.582 val_IoU - 0.436
 Epoch 60: loss - 0.013 accuracy - 0.993 DiceMetric - 0.583 IoU - 0.454 val_loss - 0.025 val_accuracy - 0.993 val_DiceMetric - 0.582 val_IoU - 0.438
 Epoch 61: loss - 0.012 accuracy - 0.994 DiceMetric - 0.583 IoU - 0.456 val_loss - 0.024 val_accuracy - 0.993 val_DiceMetric - 0.583 val_IoU - 0.440
 Epoch 62: loss - 0.012 accuracy - 0.994 DiceMetric - 0.584 IoU - 0.459 val_loss - 0.024 val_accuracy - 0.993 val_DiceMetric - 0.583 val_IoU - 0.443
 Epoch 63: loss - 0.012 accuracy - 0.994 DiceMetric - 0.585 IoU - 0.462 val_loss - 0.023 val_accuracy - 0.993 val_DiceMetric - 0.584 val_IoU - 0.445
 Epoch 64: loss - 0.012 accuracy - 0.994 DiceMetric - 0.586 IoU - 0.464 val_loss - 0.023 val_accuracy - 0.993 val_DiceMetric - 0.585 val_IoU - 0.447
 Epoch 65: loss - 0.012 accuracy - 0.994 DiceMetric - 0.586 IoU - 0.467 val_loss - 0.023 val_accuracy - 0.994 val_DiceMetric - 0.585 val_IoU - 0.450
 Epoch 66: loss - 0.011 accuracy - 0.994 DiceMetric - 0.587 IoU - 0.470 val_loss - 0.022 val_accuracy - 0.994 val_DiceMetric - 0.586 val_IoU - 0.452
 Epoch 67: loss - 0.011 accuracy - 0.994 DiceMetric - 0.588 IoU - 0.473 val_loss - 0.022 val_accuracy - 0.994 val_DiceMetric - 0.586 val_IoU - 0.454
 Epoch 68: loss - 0.011 accuracy - 0.994 DiceMetric - 0.589 IoU - 0.476 val_loss - 0.022 val_accuracy - 0.994 val_DiceMetric - 0.587 val_IoU - 0.457
 Epoch 69: loss - 0.011 accuracy - 0.994 DiceMetric - 0.589 IoU - 0.479 val_loss - 0.021 val_accuracy - 0.994 val_DiceMetric - 0.587 val_IoU - 0.459
 Epoch 70: loss - 0.011 accuracy - 0.995 DiceMetric - 0.590 IoU - 0.482 val_loss - 0.021 val_accuracy - 0.994 val_DiceMetric - 0.588 val_IoU - 0.461
 Epoch 71: loss - 0.011 accuracy - 0.995 DiceMetric - 0.591 IoU - 0.485 val_loss - 0.021 val_accuracy - 0.994 val_DiceMetric - 0.588 val_IoU - 0.464
 Epoch 72: loss - 0.010 accuracy - 0.995 DiceMetric - 0.592 IoU - 0.488 val_loss - 0.020 val_accuracy - 0.994 val_DiceMetric - 0.589 val_IoU - 0.466
 Epoch 73: loss - 0.010 accuracy - 0.995 DiceMetric - 0.592 IoU - 0.491 val_loss - 0.020 val_accuracy - 0.994 val_DiceMetric - 0.589 val_IoU - 0.469
 Epoch 74: loss - 0.010 accuracy - 0.995 DiceMetric - 0.593 IoU - 0.494 val_loss - 0.020 val_accuracy - 0.994 val_DiceMetric - 0.590 val_IoU - 0.471
 Epoch 75: loss - 0.010 accuracy - 0.995 DiceMetric - 0.594 IoU - 0.497 val_loss - 0.020 val_accuracy - 0.994 val_DiceMetric - 0.590 val_IoU - 0.474
 Epoch 76: loss - 0.010 accuracy - 0.995 DiceMetric - 0.595 IoU - 0.500 val_loss - 0.019 val_accuracy - 0.995 val_DiceMetric - 0.591 val_IoU - 0.476
 Epoch 77: loss - 0.010 accuracy - 0.995 DiceMetric - 0.596 IoU - 0.503 val_loss - 0.019 val_accuracy - 0.995 val_DiceMetric - 0.591 val_IoU - 0.479
 Epoch 78: loss - 0.010 accuracy - 0.995 DiceMetric - 0.596 IoU - 0.506 val_loss - 0.019 val_accuracy - 0.995 val_DiceMetric - 0.592 val_IoU - 0.481
 Epoch 79: loss - 0.010 accuracy - 0.995 DiceMetric - 0.597 IoU - 0.510 val_loss - 0.019 val_accuracy - 0.995 val_DiceMetric - 0.593 val_IoU - 0.484
 Epoch 80: loss - 0.009 accuracy - 0.995 DiceMetric - 0.598 IoU - 0.513 val_loss - 0.018 val_accuracy - 0.995 val_DiceMetric - 0.593 val_IoU - 0.487
 Epoch 81: loss - 0.009 accuracy - 0.995 DiceMetric - 0.599 IoU - 0.516 val_loss - 0.018 val_accuracy - 0.995 val_DiceMetric - 0.594 val_IoU - 0.489
 Epoch 82: loss - 0.009 accuracy - 0.995 DiceMetric - 0.599 IoU - 0.520 val_loss - 0.018 val_accuracy - 0.995 val_DiceMetric - 0.594 val_IoU - 0.492
 Epoch 83: loss - 0.009 accuracy - 0.996 DiceMetric - 0.600 IoU - 0.523 val_loss - 0.018 val_accuracy - 0.995 val_DiceMetric - 0.595 val_IoU - 0.495
 Epoch 84: loss - 0.009 accuracy - 0.996 DiceMetric - 0.601 IoU - 0.526 val_loss - 0.017 val_accuracy - 0.995 val_DiceMetric - 0.595 val_IoU - 0.498
 Epoch 85: loss - 0.009 accuracy - 0.996 DiceMetric - 0.602 IoU - 0.530 val_loss - 0.017 val_accuracy - 0.995 val_DiceMetric - 0.596 val_IoU - 0.500
 Epoch 86: loss - 0.009 accuracy - 0.996 DiceMetric - 0.603 IoU - 0.533 val_loss - 0.017 val_accuracy - 0.995 val_DiceMetric - 0.596 val_IoU - 0.503
 Epoch 87: loss - 0.009 accuracy - 0.996 DiceMetric - 0.603 IoU - 0.537 val_loss - 0.017 val_accuracy - 0.995 val_DiceMetric - 0.597 val_IoU - 0.506
 Epoch 88: loss - 0.009 accuracy - 0.996 DiceMetric - 0.604 IoU - 0.541 val_loss - 0.017 val_accuracy - 0.995 val_DiceMetric - 0.597 val_IoU - 0.509
```

Ilustración 43: Ejemplos de etiquetas predichas vs reales vs Imagen. Fuente: Elaboración propia.



3.8. Comparación de resultados con ResNet + PsPNet

Se expone a continuación, en la Tabla 13, las condiciones de la experimentación llevada a cabo por los autores del dataset A2D2 de Audi AG® (Geyer et al., 2020) con el fin de contrastarlos con los utilizados en este trabajo fin de master y poder comparar posteriormente sus resultados.

Tabla 13: Comparación de condiciones entrenamiento. Fuente: Elaboración propia.

Concepto	Dataset A2D2	Este TFM
Arquitectura del modelo	Codificación: ResNet Decodificación: PsPNet	U-Net con las modificaciones descritas en el punto 3.3
Cantidad imágenes etiquetadas	40030	31448 (se usaron sólo las imágenes de la cámara frontal)
Resolución de imágenes	1920x1208	1920x1208 recortadas y escaladas a 192 x 256.
Segmentación de datos	Entrenamiento: 28.015 (70%) Validación: 4118 (10%) Test: 7897 (20%)	Entrenamiento: 22013 (70%) Validación: 6289 (20%) Test: 3146 (10%)
Número de clases	19, clases iniciales reducido por agrupación de similares.	53.
Data Augmentation	Recorte, variación de brillo, contraste, simetría horizontal	No aplicado
Métricas	IoU	Accuracy, Dice, IoU
EPOCH	Desconocido	100 programados 88 ejecutados
Tasa de aprendizaje	0.01	0.001

Se muestra en la Tabla 14 la comparación del rendimiento entre el modelo entrenado en el dataset A2D2 de Audi AG® y el entrenado en este trabajo fin de master. Los valores de la métrica Media IoU para el dataset A2D2 de Audi AG® han sido extraídos de (Geyer et al., 2020) mientras que el resto procede de los resultados del entrenamiento y evaluación llevado a cabo en este trabajo fin de master utilizando la arquitectura U-Net.

Tabla 14: Comparación de resultados A2D2 Vs este TFM. Fuente: Elaboración propia.

		Métrica de comparación			
Fuente	Arquitectura	Media IoU	Media Dice	Tiempo entrenamiento	Tiempo evaluación
Dataset A2D2 Audi AG®	ResNet-101 + PsPNet	71.01%	No facilitado	No facilitado	No facilitado
Este TFM	U-Net	50.10%	59.2%	15h, 20min, 8seg	192 ms/img

Conclusiones y trabajo futuro.

4. Conclusiones y trabajo futuro

En este trabajo fin de master se desarrollan las herramientas metodología para la extracción, transformación y carga de un dataset de 31448 imágenes con sus correspondientes etiquetas, procedente del dataset A2D2 de Audi AG® , para hacerlo compatible con la arquitectura de red neuronal U-Net. Así mismo se desarrolla la metodología para la construcción, entrenamiento y evaluación de dicha arquitectura mediante la utilización de librerías de TensorFlow, Keras y Python.

Se comparan, considerando las diferencias existentes en los entrenamientos ejecutados, los resultados obtenidos del entrenamiento de la red neuronal U-Net con los aportados por los autores del dataset A2D2 en la métrica de la media del coeficiente IoU (insertion Over the union). Se concluye que los resultados del modelo utilizado por los autores de dataset A2D2 son mejores en los obtenidos en este trabajo para la métrica IoU.

Durante el desarrollo de este trabajo fin de master se intentó contactar con los autores del pape del dataset A2D2 de Audi AG® (Geyer et al., 2020) con el fin de recabar información sobre las condiciones en las que se desarrolló el entrenamiento de los 3 escenarios, así como el tiempo de entrenamiento y evaluación de los mismos. Desafortunadamente, a fecha de depósito de este trabajo fin de master únicamente se recibió una respuesta de fuera de oficina el día 12 de julio de 2023 por parte de uno de ellos, es por este motivo por los que sólo pueden arrojarse conclusiones con respecto a la información existente en el la publicación de dicho dataset.

Como líneas de trabajo futuro, se evidencia la necesidad de comparar el tiempo de entrenamiento y evaluación de los modelos ensayados en el dataset A2D2 de Audi AG® con el desarrollado en este trabajo fin de master para obtener una conclusión sobre a la eficiencia de modelo de red

neuronal U-Net frente a los modelos ResNet + PsPNet ensayados por los autores.

Así mismo, se propone la variación de la parametrización de la red neuronal U-Net de forma diferente para intentar conseguir una mejora del rendimiento. Se podrían variar tanto el tasa de aprendizaje como la inicialización de los pesos o las capas drop-out tanto en su coeficiente de aplicación o la activación de las mismas en cada paso de la codificación.

Bibliografía

5. Bibliografía

Andreas Geiger. (2012). Are we ready for autonomous driving? The KITTI vision benchmark suite. *Andreas Geiger. Are we ready for autonomous driving? Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Andreas Geiger, P. L. C. S. R. U. (2013). Vision meets robotics: The KITTI dataset. *The International Journal of Robotics Research*, *32*(11).

Badrinarayanan, V., Kendall, A. & Cipolla, R. (2015). *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*. http://arxiv.org/abs/1511.00561

Badrinarayanan, V., Kendall, A. & Cipolla, R. (2017). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(12), 2481-2495. https://doi.org/10.1109/TPAMI.2016.2644615

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

C. Chen, A. S. A. K. J. X. (2015). Deepdriving: Learning affordance for direct perception in autonomous driving. *Proceedings of the IEEE International Conference on Computer Vision,* 2272-2730.

Caesar, H., Bankiti, V., Lang, A. H., Vora, S., Liong, V. E., Xu, Q., Krishnan, A., Pan, Y., Baldan, G. & Beijbom, O. (2019). *nuScenes: A multimodal dataset for autonomous driving*. http://arxiv.org/abs/1903.11027

Cakir, S., Gauß, M., Häppeler, K., Ounajjar, Y., Heinle, F. & Marchthaler, R. (2022). *Semantic Segmentation for Autonomous Driving: Model*

Evaluation, Dataset Generation, Perspective Comparison, and Real-Time Capability. http://arxiv.org/abs/2207.12939

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K. & Yuille, A. L. (2014). *Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs*. http://arxiv.org/abs/1412.7062

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K. & Yuille, A. L. (2016). *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs*. http://arxiv.org/abs/1606.00915

Chen, L.-C., Yang, Y., Wang, J., Xu, W. & Yuille, A. L. (2015). *Attention to Scale: Scale-aware Semantic Image Segmentation*. http://arxiv.org/abs/1511.03339

Chen, L.-C., Zhu, Y., Papandreou, G., Schroff, F. & Adam, H. (2018). *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*. http://arxiv.org/abs/1802.02611

Chollet, F. & others. (2015). *Keras*. GitHub. https://github.com/fchollet/keras

D. M. West. (2016). *Securing the future of driverless cars*.

Dean, J. & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*.

Díez Ramírez, A. (2018). *Conducción autónoma: Estudio del estado del arte, impacto sobre la movilidad y desarrollo de simulador de tráfico*.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. & Koltun, V. (2017). *CARLA: An Open Urban Driving Simulator*. http://arxiv.org/abs/1711.03938

Everingham, M., Van~Gool, L., Williams, C. K. I., Winn, J. & Zisserman, A. (2022). *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*.

Fisher Yu, W. X. Y. C. F. L. M. L. V. M. T. D. (2018). BDD100K: A diverse driving video database with scalable annotation tooling. En *preprint arXiv*.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*, 193-202.

Gantt, H. L. (1910). *Work, wages, and profits; their influence on the cost of living*.

Gerhard Neuhold, T. O. S. R. B. P. K. (2017). The Mapillary Vistas dataset for semantic understanding of street scenes. *Proceeding In International Conference on Computer Vision (ICCV)*.

Geyer, J., Kassahun, Y., Mahmudi, M., Ricou, X., Durgesh, R., Chung, A. S., Hauswald, L., Pham, V. H., Mühlegg, M., Dorn, S., Fernandez, T., Jänicke, M., Mirashi, S., Savani, C., Sturm, M., Vorobiov, O., Oelker, M., Garreis, S. & Schuberth, P. (2020). *A2D2: Audi Autonomous Driving Dataset*. http://arxiv.org/abs/2004.06320

Ghemawat, S., Gobioff, H. & Leung Google, S.-T. (2003). *The Google File System*.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. & Bengio, Y. (2014). *Generative Adversarial Networks*.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*, 357-362. https://doi.org/10.1038/s41586-020-2649-2

He, K., Zhang, X., Ren, S. & Sun, J. (2015). *Deep Residual Learning for Image Recognition*. http://arxiv.org/abs/1512.03385

Hecker, S., Dai, D. & Van Gool, L. (2018). *End-to-End Learning of Driving Models with Surround-View Cameras and Route Planners*. http://arxiv.org/abs/1803.10158

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. & Adam, H. (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. http://arxiv.org/abs/1704.04861

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90-95. https://doi.org/10.1109/MCSE.2007.55

Hussain, S. M. F., Hamza, S. M. & Samad, A. (2022). Image Segmentation for Autonomous Driving Using U-Net Inception. *2022 7th International Conference on Signal and Image Processing (ICSIP)*, 426-429. https://doi.org/10.1109/ICSIP55141.2022.9885809

I. Markit. (2014). Self-driving cars moving into the industry's driver's seat. *IHS Online Newsroom*.

Imatest. (2023). *Solution Sensors*. https://www.imatest.com/solutions/sensor/.

Jens Behley, M. G. A. M. J. Q. S. B. C. S. J. G. (2019). SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. *Proc. of the IEEE/CVF International Conf. on Computer Vision (ICCV)*.

Kendall, A., Badrinarayanan, V. & Cipolla, R. (2015). *Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding*. http://arxiv.org/abs/1511.02680

Kesten, R., Usman, M., Houston, J., Pandya, T., Nadhamuni, K., Ferreira, A., Yuan, M., Low, B., Jain, A., Ondruska, P., Omari, S., Shah, S., Kulkarni, A., Kazakova, A., Tao, C., Platinsky, L., Jiang, W. & Shet, V. (2019). *Lyft Level 5 AV Dataset 2019*.

Kim, B., Kang, C. M., Lee, S. H., Chae, H., Kim, J., Chung, C. C. & Choi, J. W. (2017). *Probabilistic Vehicle Trajectory Prediction over Occupancy Grid Map via Recurrent Neural Network*. http://arxiv.org/abs/1704.07049

Kirillov, A., He, K., Girshick, R., Rother, C. & Dollár, P. (2018). *Panoptic Segmentation*. http://arxiv.org/abs/1801.00868

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*, 84-90.

Kusano, K. D., Beatty, K., Schnelle, S., Favarò, F., Crary, C. & Victor, T. (2022). *Collision Avoidance Testing of the Waymo Automated Driving System*.

Le Mero, L., Yi, D., Dianati, M. & Mouzakitis, A. (2022). A Survey on Imitation Learning Techniques for End-to-End Autonomous Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, *23*(9), 14128-14147. https://doi.org/10.1109/TITS.2022.3144867

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324. https://doi.org/10.1109/5.726791

Liang, Xiaodan, Wang, T., Yang, L. & Xing, E. (2018). *CIRL: Controllable Imitative Reinforcement Learning for Vision-based Self-driving*. http://arxiv.org/abs/1807.03776

Liang, Xinle, Liu, Y., Chen, T., Liu, M. & Yang, Q. (2019). *Federated Transfer Reinforcement Learning for Autonomous Driving*. http://arxiv.org/abs/1910.06001

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B. & Belongie, S. (2016). *Feature Pyramid Networks for Object Detection*. http://arxiv.org/abs/1612.03144

Long, J., Shelhamer, E. & Darrell, T. (2014). *Fully Convolutional Networks for Semantic Segmentation*. http://arxiv.org/abs/1411.4038

Luc, P., Couprie, C., Chintala, S. & Verbeek, J. (2016). *Semantic Segmentation using Adversarial Networks*. http://arxiv.org/abs/1611.08408

Maddern, W., Pascoe, G., Linegar, C. & Newman, P. (2017). *Year, 1000km: The Oxford RobotCar Dataset*. http://robotcar-dataset.robots.ox.ac.uk

Marius Cordts, M. O. S. R. T. R. M. E. R. B. U. F. S. R. (2016). The cityscapes dataset for semantic urban scene understanding. *Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semProceedings of the IEEE conference on computer vision and pattern recognition*.

Martín~Abadi, Ashish~Agarwal, Paul~Barham, Eugene~Brevdo, Zhifeng~Chen, Craig~Citro, Greg~S.~Corrado, Andy~Davis, Jeffrey~Dean, Matthieu~Devin, Sanjay~Ghemawat, Ian~Goodfellow, Andrew~Harp, Geoffrey~Irving, Michael~Isard, Jia, Y., Rafal~Jozefowicz, Lukasz~Kaiser, Manjunath~Kudlur, … Xiaoqiang~Zheng. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. https://www.tensorflow.org/

Microsoft®. (2023a, 12. julio). *AirSim announcement: This repository will be archived in the coming year.* https://microsoft.github.io/AirSim/.

Microsoft®. (2023b, 12. julio). *Project AirSim for aerial autonomy*. https://www.microsoft.com/en-us/ai/autonomous-systems-project-airsim?activetab=pivot1%3aprimaryr3.

Milletari, F., Navab, N. & Ahmadi, S.-A. (2016). *V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation*. http://arxiv.org/abs/1606.04797

Minaee, S., Boykov, Y., Porikli, F., Plaza, A., Kehtarnavaz, N. & Terzopoulos, D. (2020). *Image Segmentation Using Deep Learning: A Survey*. http://arxiv.org/abs/2001.05566

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533. https://doi.org/10.1038/nature14236

Müller, M., Dosovitskiy, A., Ghanem, B. & Koltun, V. (2018). *Driving Policy Transfer via Modularity and Abstraction*. http://arxiv.org/abs/1804.09364

Najman, L. & Schmitt, M. (1994). Watershed of a continuous function. *Signal Processing*, *38*(1), 99-112. https://doi.org/https://doi.org/10.1016/0165-1684(94)90059-0

Noh, H., Hong, S. & Han, B. (2015). *Learning Deconvolution Network for Semantic Segmentation*. http://arxiv.org/abs/1505.04366

Okuyama, T., Gonsalves, T. & Upadhyay, J. (2018). *Autonomous Driving System based on Deep Q Learnig*. https://doi.org/10.1109/ICoIAS.2018.8494053

O'Shea, K. & Nash, R. (2015). *An Introduction to Convolutional Neural Networks*. http://arxiv.org/abs/1511.08458

Otsu, N. (1979). A Threshold Selection Method from Gray-Level Histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, *9*(1), 62-66. https://doi.org/10.1109/TSMC.1979.4310076

Pan, X., You, Y., Wang, Z. & Lu, C. (2017). *Virtual to Real Reinforcement Learning for Autonomous Driving*. http://arxiv.org/abs/1704.03952

Peng Wang, X. H. X. C. D. Z. Q. G. R. Y. (2019). The apolloscape open dataset for autonomous driving and its application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Ren, S., He, K., Girshick, R. & Sun, J. (2015). *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. http://arxiv.org/abs/1506.01497

Reuben R Shamir, Y. D. J. K. G. S. and N. H. (2018). *Continuous Dice Coefficient: a Method for Evaluating Probabilistic Segmentations*. https://doi.org/10.1101/306977

Ronneberger, O., Fischer, P. & Brox, T. (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation*. http://arxiv.org/abs/1505.04597

Roy Amante Salvador, M. I. Saludares. (2019). *Autonomous Driving via Deep Reinforcement Learning*.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533-536. https://doi.org/10.1038/323533a0

SAE INTERNATIONAL. (2021). *SURFACE VEHICLE RECOMMENDED PRACTICE Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*.

Sallab, A. El, Abdou, M., Perot, E. & Yogamani, S. (2017). *Deep Reinforcement Learning framework for Autonomous Driving*. https://doi.org/10.2352/ISSN.2470-1173.2017.19.AVM-023

Salvador, B., Directora, R., Cristina, A. & Arnal, M. (2020). *Machine learning for autonomous driving*.

Shah, S., Dey, D., Lovett, C. & Kapoor, A. (2017). *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*. http://arxiv.org/abs/1705.05065

Simonyan, K. & Zisserman, A. (2014). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. http://arxiv.org/abs/1409.1556

Sun, K., Zhao, Y., Jiang, B., Cheng, T., Xiao, B., Liu, D., Mu, Y., Wang, X., Liu, W. & Wang, J. (2019). *High-Resolution Representations for Labeling Pixels and Regions*. http://arxiv.org/abs/1904.04514

Sun, P., Kretzschmar, H., Dotiwalla, X., Chouard, A., Patnaik, V., Tsui, P., Guo, J., Zhou, Y., Chai, Y., Caine, B., Vasudevan, V., Han, W., Ngiam, J., Zhao, H., Timofeev, A., Ettinger, S., Krivokon, M., Gao, A., Joshi, A., … Anguelov, D. (2019). *Scalability in Perception for Autonomous Driving: Waymo Open Dataset*. http://arxiv.org/abs/1912.04838

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. & Rabinovich, A. (2014). *Going Deeper with Convolutions*. http://arxiv.org/abs/1409.4842

Taha, A. A. & Hanbury, A. (2015). Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool. *BMC Medical Imaging*, *15*(1), 29. https://doi.org/10.1186/s12880-015-0068-x

Tensorflow Github. (2023. febrero). *Tensorflow Github issues #59694*.

The Waymo Team. (2022, 1. noviembre). *Making air travel more convenient at Phoenix Sky Harbor Airport*. https://waymo.com/blog.

Torabi, F., Warnell, G. & Stone, P. (2018). *Behavioral Cloning from Observation*. http://arxiv.org/abs/1805.01954

Turienzo, J. , L. J. F. & C. P. (2022). El impacto del vehículo autónomo, conectado y compartido: de la industria automotriz tradicional a la cadena de valor de la nueva movilidad. *Dimensión Empresarial, 20(1), 1-21 DOI: 10.15665/dem.v20i1.2775*, *20*(1), 1-21.

Unión Europea. (2016). *REGLAMENTO (UE) 2016/ 679 DEL PARLAMENTO EUROPEO Y DEL CONSEJO - de 27 de abril de 2016 - relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos y por el que se deroga la Directiva 95/ 46/ CE (Reglamento general de protección de datos)*.

Van Rossum, G. & Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace.

Visin, F., Ciccone, M., Romero, A., Kastner, K., Cho, K., Bengio, Y., Matteucci, M. & Courville, A. (2015). *ReSeg: A Recurrent Neural Network-based Model for Semantic Segmentation*. http://arxiv.org/abs/1511.07053

Visin, F., Kastner, K., Cho, K., Matteucci, M., Courville, A. & Bengio, Y. (2015). *ReNet: A Recurrent Neural Network Based Alternative to Convolutional Networks*. http://arxiv.org/abs/1505.00393

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K. & Lang, K. J. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *37*(3), 328-339. https://doi.org/10.1109/29.21701

Wu, B., Wan, A., Iandola, F., Jin, P. H. & Keutzer, K. (2016). *SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving*. http://arxiv.org/abs/1612.01051

X. Mosquet, T. D. N. L. N. R. A. M.-P. R. A. F. S. (2015). Revolution in the driver's seat: The road to autonomous vehicles,. En *Boston Consulting Group,* (Vol. 11).

Yang, L., Liang, X., Wang, T. & Xing, E. (2018). *Real-to-Virtual Domain Unification for End-to-End Autonomous Driving*. http://arxiv.org/abs/1801.03458

Yu, A., Palefsky-Smith, R. & Bedi, R. (2016). *Deep Reinforcement Learning for Simulated Autonomous Vehicle Control*.

Zaharia, M., Chowdhury, M., Franklin, M. J. & Shenker, S. (2010). *Spark: Cluster Computing with Working Sets*.

Zhang, Z., Liu, Q. & Wang, Y. (2017). *Road Extraction by Deep Residual U-Net*. https://doi.org/10.1109/LGRS.2018.2802944

Zhao, H., Shi, J., Qi, X., Wang, X. & Jia, J. (2016). *Pyramid Scene Parsing Network*. http://arxiv.org/abs/1612.01105

Zhiqing Huang, J. Z. R. T. Y. Z. (2019). Zhiqing HuangEnd-to-end autonomous driving decision based on deep reinforcement learning. *Proceding In 2019 5th International Conference on Control, Automation and Robotics (ICCAR)*.

Zhou, Q.-Y., Park, J. & Koltun, V. (2018). *Open3D: A Modern Library for 3D Data Processing*. http://arxiv.org/abs/1801.09847

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H. & He, Q. (2019). *A Comprehensive Survey on Transfer Learning*. http://arxiv.org/abs/1911.02685

Zijdenbos, A. P., Dawant, B. M., Margolin, R. A. & Palmer, A. C. (1994). Morphometric analysis of white matter lesions in MR images: method and validation. *IEEE Transactions on Medical Imaging*, *13*(4), 716-724. https://doi.org/10.1109/42.363096

Anexo 1: Código Python para la reemplazar el valor del pixel en las etiquetas del dataset A2D2 de Audi AG®

Anexo 1: Código Python para reemplazar el valor del pixel en las etiquetas del dataset A2D2 de Audi AG®

Código ejecutado para 2 carpetas con 4251 imágenes.

- Importación de las librerías necesarias.

```
import numpy as np
import os
import glob
import cv2
import json
from time import time
```

- Construcción de las listas de las rutas de las imágenes y las etiquetas.

```
#Construir las rutas a las carpetas de imágenes y etiquetas
Dataset_path = 'c:/camera_lidar_semantic/' # Defino la ruta del Dataset
folders_pre = glob.glob(Dataset_path + "/*/", recursive = False) # Listo las carpetas
que contiene el Dataset.
print('Ejemplo Folders_pre =',folders_pre[0])

# Listado de los identificadores de las carpetas.
folders =[]
for i in range(len(folders_pre)):
    folder_old = folders_pre[i].split('\\')
    folder_new = folder_old[1]
    folders.append(folder_new)
print('Ejemplo folders =', folders[0])

# Identifico y divido las carpetas que contienen las imágenes y las etiquetas.
image_folders = [f'../camera_lidar_semantic/{f}/camera/cam_front_center/' for f in folders]
label_folders = [f'../camera_lidar_semantic/{f}/label/cam_front_center/' for f in folders]
print('Ejemplo image_folders =',image_folders[0])
print('Ejemplo label_folders =',label_folders[0])

# Identifico las imágenes en cada carpeta y uno sus nombres con la ruta
all_labels = [os.path.join(folder, image) for folder in label_folders for image in os.listdir(folder)]
all_images_pre = [glob.glob(os.path.join(folder,'*.png')) for folder in image_folders]
all_images_flatten = [item for l in all_images_pre for item in l] # aplana la lista anterior

# Reemplazo '\\' por '/' para tener el formato correcto para alimentar las imágenes
all_images =[]
for i in range(len(all_images_flatten)):
    file_old = all_images_flatten[i].split('\\')
    file_new = file_old[0]+'/'+file_old[1]
    all_images.append(file_new)

print('Número de carpetas del dataset: ',len(folders))
print('Ejemplo de ruta imágenes:', all_images[1])
print('Número de imágenes: ', len(all_images))
print('Ejemplo de ruta etiquetas:', all_labels[1])
print('Número de  etiquetas: ',len(all_labels))
```

- Verificamos el tamaño de las imágenes y las etiquetas cargando de forma aleatoria 10 de ellas. Vemos que el tamaño es 1208x1920 con 3 canales por pixel tanto en las imágenes como en las etiquetas.

```
#Función de carga de imágenes
def cargar_imagen(im_path):
    """
    Carga las imágenes de la carpeta y la transforma en espacio de color RGB desde el espacio de color BGR.
    Argumentos:
        im_path (str): La ruta de la imagen a cargar.
    Retorno:
        numpy.ndarray: La imagen cargada en espacio de color RGB.
    """
    return cv2.cvtColor(cv2.imread(im_path), cv2.COLOR_BGR2RGB)
```

```
np.random.seed(123)
for img in np.random.choice(all_images, 10, replace=False):
    image = cargar_imagen(img)
    print("Dimensión de las imágenes: {}".format(image.shape))
print('-----------------------------------------------')
for label in np.random.choice(all_labels, 10, replace=False):
    label = cargar_imagen(label)
    print("Dimensión de las etiquetas: {}".format(label.shape))
```

- Exploramos los valores de los pixeles de las imágenes y etiquetas para las mismas parejas y en los mismos puntos(en el medio de la imagen). Vemos que para distintos valores de pixel de imagen tiene asignado el mismo valor de pixel en la etiqueta coincidiendo con el etiquetado.

```
print('-Número de carpetas del dataset: ',len(folders))
print('-Número de imágenes: ', len(all_images))
print('-Número de imágenes etiquetadas: ',len(all_labels))
print('-------------------------------------------------------------------------------')


for i in range(3):
    print ('Pixels imagen {}:'.format(i))
    image = cargar_imagen(all_images[i])
    for x in range(2):
        for y in range(1):
            print(x,y ,"image shape {} , image type{}".format(image.shape, type(image)),"---Valor del p
ixel:",image[x+600,y+960,])


    print ('Pixels label {}:'.format(i))


    label = cargar_imagen(all_labels[i])
    for x in range(2):
        for y in range(1):
            print(x,y ,"label shape {} , Label type{}".format(label.shape, type(image)),"---Valor del p
ixel:",label[x+600,y+960,])
    print('-------------------------------------------------------------------------------'
```

- Tratamiento de las etiquetas de clases:

Vemos como los pixeles de las etiquetas vienen definidos en formato RGB, sin embargo, la arquitectura U-NET que vamos a utilizar demanda que el valor del pixel identifique la clase de la segmentación a utilizar. Dado que realizaremos el procesado de las imágenes y las etiquetas con 3 canales, repetimos el número de clase en cada uno de los canes para conservar la dimensión del pixel.

Las etiquetas de las clases vienen definidas en formato hexadecimal en un fichero .JSON "class_list.json", y los pixeles de cada imagen se transforman en RGB tras leer la imagen con la función cargar_imagen(im_path) . Por este motivo hay que establecer la relación entre la etiqueta de clase indicada en el fichero .JSON y el valor RGB del pixel.

```
#Importación del archivo de clases
class_labels_ColorHex = json.load(open("C:/camera_lidar_semantic/class_list.json", "r"))
print('class_labels_ColorHex',class_labels_ColorHex)
print('class_labels_ColorHex len:',len(class_labels_ColorHex))
print('----------------------------------------')
#Función para cambiar las claves de un diccionario a otras con correspondencia 1:1
def cambiar_claves(class_labels_origen, class_labels_correspondencia):
    '''Función que cambia las claves del diccionario de etiquetas de origen a etiquetas con correspondencia a
signada
       ARGS: Diccionario original
             Diccionario con la correspondencia clave hexadecimal Vs Clave RGB'''
    class_labels_resultado = {}
    for clave, valor in class_labels_origen.items():
        nueva_clave = str(class_labels_correspondencia.get(clave)) # ha de convertirse a string ya para que
 el diccionario pueda indexar.
        class_labels_resultado[nueva_clave] = valor
    return  class_labels_resultado
#Codificación de etiquetas en formato RGB
def hex_to_rgb(hex_code):
    '''Función que transforma el formato Hexadecimal del color a RGB
       ARGS: Código hexadecimal'''
    hex_code = hex_code.strip("#")
    r = int(hex_code[0:2], 16)
    g = int(hex_code[2:4], 16)
    b = int(hex_code[4:6], 16)
    return [r, g, b]
#Lista de etiquetas RGB
RGB_labels = []
#for key, value in class_labels_ColorHex.items():
#    RGB_labels = hex_to_rgb(key)
#    RGB_labels.append()
#print('RGB_labels: ',RGB_labels)
    #Construcción del diccionario de correspondencia Hexadecimal - RGB
ColorHex_RGB = {}
```

```
#-----Continuación-----

for key,value in class_labels_ColorHex.items():
    ColorHex_RGB[key] = hex_to_rgb(key)
print('ColorHex_RGB:',ColorHex_RGB)
print('ColorHex_RGB len:', len(ColorHex_RGB))
print('----------------------------------------')
    #Construcción del diccionario etiquetas RGB cambiando las claves del original
class_labels_RGB = cambiar_claves(class_labels_ColorHex, ColorHex_RGB)
print('class_labels_RGB:',class_labels_RGB )
print('class_labels_RGB len:',len(class_labels_RGB) )
print('----------------------------------------')

#Construcción del mapeo clase RGB  a número de clase
#lista de valores de las clases en RGB
clases=[]


for i in range(1,56): #Cada clase será una lista de 3 valores iguales del número 1 al 55.
    element = [i,i,i]
    clases.append(element)
print('clases: ', clases)
print('len(clases): ',len(clases))


#Lista de arays conteniendo los valores RGB de las clases del fichero JSON
RGB_labels = list(ColorHex_RGB.values())
print('RGB_labels: ', RGB_labels)
print('RGB labels len: ', len(RGB_labels))
print('----------------------------------------')

#Diccionario de correspondencias convirtiendo las listas de RGBs y Clases a tuplas
mapeo_RGB_class = dict(zip(map(tuple,RGB_labels), map(tuple,clases)))


print('mapeo_RGB_class:',mapeo_RGB_class)
print('mapeo_RGB_class len:',len(mapeo_RGB_class) )
print('----------------------------------------')
```

- Definición de la función de cambio de los pixeles originales de la imagen por los valores de las clases.

```
def replace_pixels(path, replacement_dict):
    label = cv2.cvtColor(cv2.imread(path), cv2.COLOR_BGR2RGB)
    # Crear una copia de la imagen para no modificar la original
    # Obtener las dimensiones de la imagen
    height, width, _ = label.shape
    # Recorrer cada pixel de la imagen


for i in range(height):
        for j in range(width):
            # Obtener el valor RGB del pixel
            pixel = tuple(label[i,j])
            # Verificar si el valor del pixel está en el diccionario de reemplazo
            if pixel in replacement_dict:
                # Reemplazar el valor del pixel por el valor correspondiente en el diccionario
                label[i,j] = replacement_dict[pixel]
            else:
                # Si no hay correspondencia, reemplazar por [0,0,0]
                label[i,j] = [0,0,0]
    return label
```

- Ejecución del reemplazo de pixeles en la imagen de etiqueta.

```
TIME_INIT = TIME()
FOR I IN RANGE(LEN(ALL_LABELS)):
    NEW_LABEL = REPLACE_PIXELS(ALL_LABELS[I], MAPEO_RGB_CLASS)
    CV2.IMWRITE(ALL_LABELS[I], NEW_LABEL)
TIME_FIN = TIME()
TIME_PROCESO = (TIME_FIN-TIME_INIT)
HORAS = TIME_PROCESO  // 3600
MINUTOS = (TIME_PROCESO  % 3600) // 60
SEGUNDOS = INT((TIME_PROCESO  % 3600) % 60)
PRINT('TIEMPO DE PROCESADO:', HORAS, 'HORAS', MINUTOS, 'MINUTOS', SEGUNDOS, 'SEGUNDOS')
```

- Repetimos la exploración del data set para verificar el cambio de los pixeles de las etiquetas por sus correspondientes clases.

```
PRINT('-TIEMPO DE PROCESADO:', HORAS, 'HORAS', MINUTOS, 'MINUTOS', SEGUNDOS, 'SEGUNDOS')
PRINT('-NÚMERO DE IMÁGENES: ', LEN(ALL_IMAGES))
PRINT('-NÚMERO DE IMÁGENES ETIQUETADAS: ',LEN(ALL_LABELS))
PRINT('------------------------------------------------------------------------------')
FOR I IN RANGE(3):
    PRINT ('PIXELS IMAGEN {}:'.FORMAT(I))
    IMAGE = CARGAR_IMAGEN(ALL_IMAGES[I])
    FOR X IN RANGE(2):
        FOR Y IN RANGE(1):
            PRINT(X,Y ,"IMAGE SHAPE {} , IMAGE TYPE{}".FORMAT(IMAGE.SHAPE, TYPE(IMAGE)),"---VALOR DEL P
IXEL:",IMAGE[X+600,Y+960,])
    PRINT ('PIXELS LABEL {}:'.FORMAT(I))
    LABEL = CARGAR_IMAGEN(ALL_LABELS[I])
    FOR X IN RANGE(2):
        FOR Y IN RANGE(1):
            PRINT(X,Y ,"LABEL SHAPE {} , LABEL TYPE{}".FORMAT(LABEL.SHAPE, TYPE(IMAGE)),"---VALOR DEL P
IXEL:",LABEL[X+600,Y+960,])
    PRINT('------------------------------------------------------------------------------')
```

Anexo 2: Código Python para el entrenamiento de una red neuronal U-Net con dataset A2D2 de Audi AG®

Anexo 2: Código Python para el entrenamiento de la red neuronal U-Net con el dataset A2D2 de Audi AG®

El presente cuaderno de muestra un método para entrenar una red neuronal de arquitectura U-Net para la segmentación semántica del Dataset A2D2 publicado por Audi AG® bajo licencia CCBY-ND 4.0.

- Importamos las librerías necesarias.

```
IMPORT TENSORFLOW AS TF
IMPORT NUMPY AS NP
FROM TENSORFLOW.KERAS.LAYERS IMPORT INPUT
FROM TENSORFLOW.KERAS.LAYERS IMPORT CONV2D
FROM TENSORFLOW.KERAS.LAYERS IMPORT MAXPOOLING2D
FROM TENSORFLOW.KERAS.LAYERS IMPORT DROPOUT
FROM TENSORFLOW.KERAS.LAYERS IMPORT CONV2DTRANSPOSE
FROM TENSORFLOW.KERAS.LAYERS IMPORT CONCATENATE
FROM TENSORFLOW.PYTHON.CLIENT IMPORT DEVICE_LIB
IMPORT OS
IMPORT GLOB
IMPORT CV2
IMPORT JSON
FROM TIME IMPORT TIME
IMPORT MATPLOTLIB.PYPLOT AS PLT
IMPORT MATPLOTLIB.PYLAB AS PT
%MATPLOTLIB INLINE
IMPORT ABSL.LOGGING
IMPORT WARNINGS
```

- Identificamos la GPU así como la versión de TensorFlow con la que procesaremos el data set.

```
DEF GET_AVAILABLE_GPUS():
    LOCAL_DEVICE_PROTOS = DEVICE_LIB.LIST_LOCAL_DEVICES()
    RETURN [X.PHYSICAL_DEVICE_DESC FOR X IN LOCAL_DEVICE_PROTOS IF X.DEVICE_TYPE == 'GPU']

PRINT("GPU DE TRABAJO: ", GET_AVAILABLE_GPUS())
PRINT('TENSORFLOW VERSION:',TF.__VERSION__)
```

- Definición de las variables globales del Dataset:

Las imágenes y las etiquetas del Dataset tienen extensión .png por lo que la especificación del espacio de color para los píxeles el BGR. Con lo que el número de canales es 3.

El número de clases del data set es 55 según se define en el fichero de clases clases.json.

Las dimensiones de las imágenes es 1920x1208. Es relevante para la arquitectura U-NET que las dimensiones sean divisibles por 2 debido a

que tanto las capas max pooling utilizadas para la codificación como la capas convolucionales usadas para la descodificación son 2x2. El tamaño de entrada seleccionado para el entrenamiento de la red neuronal es 192x256 cuyos valores permiten dividir por 2 hasta 6 veces. Para poder alimentar este tamaño deberemos recortar, a cada imagen de 1920x1208, 8 pixeles de alto y 320 de ancho, para dejarla en 1600x1200 después escalar al tamaño final de 192x256 (escalado 1:6.25). Esto lo llevaremos a cabo en la etapa de pre-procesamiento.

```
INPUT_CHANNELS = 3  #RGB => 3 CANALES DE ENTRADA
OUTPUT_CHANNELS = 55 #LA CANTIDAD DEL CLASES DE LAS ETIQUETAS ES 55 PARA EL DATASET A2D2
IMG_SHAPE = [192,256] #TAMAÑO DE LAS IMÁGENES DE A2D2 = 1208X1900 => HAY QUE PREPROCESAR.
```

- Construcción de las listas de las rutas de las imágenes y las etiquetas.

```
#CONSTRUIR LAS RUTAS A LAS CARPETAS DE IMÁGENES Y ETIQUETAS
DATASET_PATH = 'C:/CAMERA_LIDAR_SEMANTIC/' # DEFINO LA RUTA DEL DATASET
FOLDERS_PRE = GLOB.GLOB(DATASET_PATH + "/*/", RECURSIVE = FALSE) # LISTO LAS CARPETAS DEL DATASET.
PRINT('EJEMPLO FOLDERS_PRE =',FOLDERS_PRE[0])

# LISTADO DE LOS IDENTIFICADORES DE LAS CARPETAS.
FOLDERS =[]
FOR I IN RANGE(LEN(FOLDERS_PRE)):
    #PRINT(FOLDERS_PRE[I])
    FOLDER_OLD = FOLDERS_PRE[I].SPLIT('\\')
    #PRINT(FOLDER_OLD)
    FOLDER_NEW = FOLDER_OLD[1]
    #PRINT(FOLDER_NEW)
    FOLDERS.APPEND(FOLDER_NEW)
PRINT('EJEMPLO FOLDERS =', FOLDERS[0])

# IDENTIFICO Y DIVIDO LAS CARPETAS QUE CONTIENEN LAS IMÁGENES Y LAS ETIQUETAS.
IMAGE_FOLDERS = [F'../CAMERA_LIDAR_SEMANTIC/{F}/CAMERA/CAM_FRONT_CENTER/' FOR F IN FOLDERS]
LABEL_FOLDERS = [F'../CAMERA_LIDAR_SEMANTIC/{F}/LABEL/CAM_FRONT_CENTER/' FOR F IN FOLDERS]
PRINT('EJEMPLO IMAGE_FOLDERS =',IMAGE_FOLDERS[0])
PRINT('EJEMPLO LABEL_FOLDERS =',LABEL_FOLDERS[0])

# IDENTIFICO LAS IMÁGENES EN CADA CARPETA Y UNO SUS NOMBRES CON LA RUTA
ALL_LABELS = [OS.PATH.JOIN(FOLDER, IMAGE) FOR FOLDER IN LABEL_FOLDERS FOR IMAGE IN OS.LISTDIR(FOLDER)]
ALL_IMAGES_PRE = [GLOB.GLOB(OS.PATH.JOIN(FOLDER,'*.PNG')) FOR FOLDER IN IMAGE_FOLDERS]
ALL_IMAGES_FLATTEN = [ITEM FOR L IN ALL_IMAGES_PRE FOR ITEM IN L] # APLANA LA LISTA ANTERIOR
# REEMPLAZO '\\' POR '/' PARA TENER EL FORMATO CORRECTO PARA ALIMENTAR LAS IMÁGENES
ALL_IMAGES =[]

FOR I IN RANGE(LEN(ALL_IMAGES_FLATTEN)):
    #PRINT(ALL_IMAGES_FLATTEN[I])
    FILE_OLD = ALL_IMAGES_FLATTEN[I].SPLIT('\\')
    #PRINT(FILE_OLD)
    FILE_NEW = FILE_OLD[0]+'/'+FILE_OLD[1]
    #PRINT(FILE_NEW)
    ALL_IMAGES.APPEND(FILE_NEW)
```

```
# -----CONTINUACIÓN-----

PRINT('NÚMERO DE CARPETAS DEL DATASET: ',LEN(FOLDERS))
PRINT('EJEMPLO DE RUTA IMÁGENES:', ALL_IMAGES[1])
PRINT('NÚMERO DE IMÁGENES: ', LEN(ALL_IMAGES))
PRINT('TIPO DE OBJETO ALL_IMAGES:',TYPE(ALL_IMAGES))
PRINT('EJEMPLO DE RUTA ETIQUETAS:', ALL_LABELS[1])
PRINT('TIPO DE OBJETO ALL_LABELS:',TYPE(ALL_LABELS))
PRINT('NÚMERO DE IMÁGENES ETIQUETADAS: ',LEN(ALL_LABELS))
```

- Exploración del dataset

Verificamos el tamaño de las imágenes y las etiquetas cargando de forma aleatoria 10 de ellas. Vemos que el tamaño es 1208x1920 con 3 canales por pixel tanto en las imágenes como en las etiquetas.

```
#FUNCIÓN DE CARGA DE IMÁGENES
DEF CARGAR_IMAGEN(IM_PATH):
    """
    FUNCIÓN PARA CARGA LAS IMÁGENES DE LA CARPETA Y LA TRANSFORMA EN ESPACIO RGB DESDE EL ESPACIO  BGR.
    ARGUMENTOS:
                IM_PATH (STR): THE RUTA DE LA IMAGEN A CARGAR.
    RETORNO:
                NUMPY.NDARRAY: LA IMAGEN CARGADA EN ESPACIO DE COLOR RGB.
    """
    RETURN CV2.CVTCOLOR(CV2.IMREAD(IM_PATH), CV2.COLOR_BGR2RGB)

NP.RANDOM.SEED(123)


FOR IMG IN NP.RANDOM.CHOICE(ALL_IMAGES, 10, REPLACE=FALSE):
    IMAGE = CARGAR_IMAGEN(IMG)
    PRINT("DIMENSIÓN DE LAS IMÁGENES: {}".FORMAT(IMAGE.SHAPE))
PRINT('-------------------------------------------------')
FOR LABEL IN NP.RANDOM.CHOICE(ALL_LABELS, 10, REPLACE=FALSE):
    LABEL = CARGAR_IMAGEN(LABEL)
    PRINT("DIMENSIÓN DE LAS ETIQUETAS: {}".FORMAT(LABEL.SHAPE))
```

Exploramos los valores de los pixeles de las imágenes y etiquetas para las mismas parejas y en los mismos puntos (centro de la imagen). Vemos que para distintos valores de pixel de imagen tiene asignado el mismo valor de pixel en la etiqueta coincidiendo con el número de clase.

```
PRINT('-NÚMERO DE CARPETAS DEL DATASET: ',LEN(FOLDERS))
PRINT('-NÚMERO DE IMÁGENES: ', LEN(ALL_IMAGES))
PRINT('-NÚMERO DE IMÁGENES ETIQUETADAS: ',LEN(ALL_LABELS))
PRINT('-------------------------------------------------------------------------------')
FOR I IN RANGE(3):
    PRINT ('PIXELES IMAGEN {}:'.FORMAT(I))
    IMAGE = CARGAR_IMAGEN(ALL_IMAGES[I])
    FOR X IN RANGE(2):
        FOR Y IN RANGE(1):
            PRINT(X,Y ,"IMAGE SHAPE {} , IMAGE TYPE{}".FORMAT(IMAGE.SHAPE, TYPE(IMAGE)),"---VALOR DEL
            PIXEL:",IMAGE[X+600,Y+960,])


    PRINT ('PIXELES LABEL {}:'.FORMAT(I))
    LABEL = CARGAR_IMAGEN(ALL_LABELS[I])
```

```
# -----CONTINUACIÓN-----

    FOR X IN RANGE(2):
        FOR Y IN RANGE(1):
            PRINT(X,Y ,"LABEL SHAPE {} , LABEL TYPE{}".FORMAT(LABEL.SHAPE, TYPE(IMAGE)),"---VALOR DEL
            PIXEL:",LABEL[X+600,Y+960,])


    PRINT('------------------------------------------------------------------------------')
```

- Construcción de las rutas de acceso a las imágenes del dataset como tensores para el pipeline.

```
IMAGE_LIST_DS = TF.DATA.DATASET.LIST_FILES(ALL_IMAGES, SHUFFLE=FALSE)
LABEL_LIST_DS = TF.DATA.DATASET.LIST_FILES(ALL_LABELS, SHUFFLE=FALSE)

FULL_DS = TF.DATA.DATASET.ZIP((IMAGE_LIST_DS, LABEL_LIST_DS))

FOR I, L IN FULL_DS.TAKE(2):
    PRINT(I)
    PRINT(L)
```

- Pre-Proceso de los datos:

Cargamos las imágenes la función de carga de TensorFlow y los normalizamos a formato float.32 que deja los pixeles en un rango de [0-1]. Para las etiquetas escogemos el máximo valor de los 3 canales que, como todos son iguales e identificó la clase, nos devuelve el valor de la clase en el valor del pixel en un intervalo de [1,55].

Recortamos 8 pixeles de alto y 320 de ancho empezando por el pixel 160 para que la imagen permanezca centrada. Re escalamos al tamaño definido en las variables globales 162x256. Importante aplicar el método 'nearest' para evitar distorsión.

```
DEF PREPROCESS(IMAGE, LABEL):
    '''FUNCIÓN DE RECORTE Y RE ESCALADO DE IMAGEN
       RECORTA 8 PIXELES DE LA PARTE SUPERIOR
       (NO RELEVANTE PARA IMÁGENES DE CONDUCCIÓN AUTÓNOMA YA QUE SERÁ SIEMPRE UN ÁREA FUERA DE LA CARRETERA)
       REDUCE A 240X385 (DIVIDE POR 5 AMBOS EJES) PARA QUE LA IMAGEN SE PUEDA DIVIDIR POR 2 VARIAS VECES
       ARGS: IMAGE = EL ARCHIVO DE IMAGEN
             LABEL = EL CORRESPONDIENTE ARCHIVO DE LABEL'''


    INPUT_IMAGE = TF.IMAGE.CROP_TO_BOUNDARY_BOX(IMAGE, 0, 160, 1200, 1600) # RECORTO 8 PIXELES DE ALTO
 Y 320 DE ANCHO.

    INPUT_IMAGE = TF.IMAGE.RESIZE(INPUT_IMAGE, IMG_SHAPE, METHOD='NEAREST') # APLICO RE ESCALADO CON
MÉTODO 'NEAREST'
    INPUT_LABEL = TF.IMAGE.CROP_TO_BOUNDARY_BOX(LABEL, 0, 160, 1200, 1600) # RECORTO 8 PIXELES DE ALTO
 Y 320 DE ANCHO.
    INPUT_LABEL = TF.IMAGE.RESIZE(INPUT_LABEL, IMG_SHAPE, METHOD='NEAREST') # APLICO RE ESCALADO CON
MÉTODO 'NEAREST'
    RETURN INPUT_IMAGE, INPUT_LABEL
```

```
# -----CONTINUACIÓN-----

DEF PROCESS_PATH(IMAGE_PATH, LABEL_PATH):
    '''FUNCIÓN QUE LEE LA IMAGEN, LA DECODIFICA Y TRANSFORMA LOS VALORES DE LOS PIXELES: IMAGEN EN RGB Y
FLOAT32, LABEL: UNTI8.
       ARGS: RUTA DE IMAGEN
             RUTA DE LA ETIQUETA CORRESPONDIENTE'''
    IMG = TF.IO.READ_FILE(IMAGE_PATH)          # LEE LA IMAGEN
    IMG = TF.IMAGE.DECODE_PNG(IMG, CHANNELS=3)  # LA DECODIFICA A RGB Y UINT8 (VALOR POR DEFECTO DE
 TF.IMAGE.DECODE)
    IMG = TF.IMAGE.CONVERT_IMAGE_DTYPE(IMG, TF.FLOAT32) # TRANSFORMA CADA VALOR DE PIXEL A FLOAT.32 Y
NORMALIZA ENTRE 0 Y 1 TOMANDO COMO BASE EL MAYOR VALOR DE ENTRADA QUE EN RGB ES 255


    LABEL = TF.IO.READ_FILE(LABEL_PATH)           # LEE LA ETIQUETA
    LABEL = TF.IMAGE.DECODE_PNG(LABEL, CHANNELS=3) # DECODIFICA A RGB Y UNIT8 (VALOR POR DEFECTO DE
TF.IMAGE.DECODE)
    LABEL = TF.MATH.REDUCE_MAX(LABEL, AXIS=-1, KEEPDIMS=TRUE) # TOMA COMO VALOR DE LA CLASE EL MÁXIMO DE L
OS VALORES DE LA DIMENSIÓN -1 (LOS VALORES SON IGUALES EN ESTA DIMENSIÓN CON LO QUE EN LA PRÁCTICA REDUCE EL
VALOR DEL PIXEL AL VALOR DE LA CLASE)


    IMG, LABEL = PREPROCESS(IMG, LABEL)


    RETURN IMG, LABEL
```

- Mapeado del dataset con las funciones de preproceso.

```
IMAGE_DS = FULL_DS.MAP(PROCESS_PATH)
```

Visualización del dataset tras el mapeado. Importante verificar las dimensiones del tensor de imágenes. Estas dimensiones han de ser (192, 256, 3) y, para las etiquetas, (192, 256, 1). Así mismo confirmamos que el valor de la etiqueta es el valor de la clase para un pixel tomando como muestra 5 imágenes contenidas en el tensor que generado.

```
NP.SET_PRINTOPTIONS(LINEWIDTH=155)
PRINT('DIMENSIÓN DE LOS TENSORES DEL DATASET:\N', IMAGE_DS , '\N')


FOR IMG, LABEL IN IMAGE_DS.TAKE(5).AS_NUMPY_ITERATOR():
     PRINT('VALOR DEL PIXEL:', IMG[0,0])
     PRINT('VALOR DE LA ETIQUETA:', LABEL[0,0])
     PRINT('-------------------------------')
```

Verificamos los valores de todos los pixeles del dataset para confirmar que todos están contenidos entre [1,55] y contamos los valores únicos para ver el número de clases contenidas realmente en el dataset.

```
DS_SIZE = TF.DATA.EXPERIMENTAL.CARDINALITY(IMAGE_DS).NUMPY() #OBTIENE EL NÚMERO DE FILAS   DEL DATASET DE
IMÁGENES = CARDINALIDAD DEL DATASET.(CADA FILA TIENE UNA IMAGEN Y SU LABEL)
PRINT('TAMAÑO DE PAREJAS DEL DATASET (IMÁGENES Y ETIQUETAS):',DS_SIZE)


CLASES_CONTEO = SET() #CREO UN CONJUNTO VACÍO PARA ALMACENAR LOS VALORES ÚNICOS DE LAS CLASES
```

```
# -----CONTINUACIÓN-----

FOR IMG, LABEL IN IMAGE_DS.TAKE(DS_SIZE):
    '''FUNCIÓN QUE MAPEA EL TENSOR DEL DATASET BUSCA LOS VALORES ÚNICOS Y LOS ALMACENA EN  UN CONJUNTO
       ARGS: IMG: POSICIÓN DE LA IMAGEN A MAPEAR EN EL TENSOR DE IMÁGENES
             LABEL: POSICIÓN DE LA ETIQUETA A MAPEAR EN EL TENSOR ETIQUETAS'''
    CLASES_CONTEO.UPDATE(NP.UNIQUE(LABEL.NUMPY()))


PRINT('CLASES CONTENIDAS EN EL DATASET:', CLASES_CONTEO)
PRINT('NÚMERO DE CLASES CONTENIDAS EN EL DATASET:',LEN(CLASES_CONTEO))
```

- Visualizamos el resultado

```
DEF DISPLAY_IMAGES(IMAGES, TITLES=NONE, PLOT_TITLE=NONE, SIZE=(15, 10), GRAYSCALE=FALSE):
    """
    FUNCIÓN QUE PERMITE MOSTRAR LAS IMÁGENES USANDO MATPLOTLIB.PYPLOT.
    ARGS:
         IMAGES (LISTA): LA LISTA DE IMÁGENES A MOSTRAR.
         TITLES (LIST): LA LISTA DE TÍTULOS DE LAS IMÁGENES A MOSTRAR.
         PLOT_TITLE (STR): TÍTULO PARA TODA LA SECUENCIA DE IMÁGENES A MOSTRAR.
         GRAYSCALE (BOLL): SI LAS ETIQUETAS VIENEN EN ESCALA DE GRISES (TRUE) O NO (FALSE)
    """
    IF TITLES IS NONE: #ASIGNAMOS LA NUMERACIÓN DE IMÁGENES EN EL CASO QUE NO HAYA TÍTULOS PARA LAS MISMAS.
        TITLES = [F"IMÁGENES {I + 1}" FOR I IN RANGE(LEN(IMAGES))]
    IF PLOT_TITLE IS NONE: #ASIGNAMOS 'IMÁGENES” PARA EL TÍTULO DE LA SECUENCIA .
        PLOT_TITLE = "IMÁGENES"
    FIG, AXS = PLT.SUBPLOTS(1, LEN(IMAGES), FIGSIZE=SIZE)
    FOR I, (IMG, TITLE) IN ENUMERATE(ZIP(IMAGES, TITLES)):      #ITERO LA TUPLA DE CADA ELEMENTO.
        AXS[I].IMSHOW(IMG, CMAP='GRAY' IF GRAYSCALE ELSE NONE)  #MUESTRO CADA IMAGEN CON MAPA DE COLOR
(ESCALA DE GRISES SI NO ESTÁ DEFINIDO.
        AXS[I].AXIS('OFF')      #SIN EJES EN LAS IMÁGENES
        AXS[I].SET_TITLE(TITLE) #TITULO ES EL ALIMENTADO POR LA LISTA DE TÍTULOS
    FIG.SUPTITLE(PLOT_TITLE IF PLOT_TITLE IS NOT FALSE ELSE NONE, FONTSIZE=20) # EL SUBTÍTULO DE CADA
FIGURA SERÁ EL MISMO QUE EL TÍTULO SI NO SE DEFINE.
    PLT.SHOW()

FOR IMG, LABEL IN IMAGE_DS.TAKE(5):
    DISPLAY_IMAGES([IMG, LABEL], TITLES=["IMAGEN", "ETIQUETA"], PLOT_TITLE=FALSE, SIZE=(10, 10),GRAYS
CALE=FALSE)
```

- División de dataset en datos para entrenamiento, verificación y test.

Establecemos los parámetros de división del dataset: 70% entrenamiento, 20% validación, 10% test. Mezclamos el data set con un tamaño de buffer igual al tamaño del dataset y a continuación establecemos la división.

```
TRAIN_SIZE = .7 # 70% PARA TRAINING
VAL_SIZE = .2   # 20% PARA VALIDACIÓN
TEST_SIZE = .1  # 10% PARA TEST
DS_SIZE = TF.DATA.EXPERIMENTAL.CARDINALITY(IMAGE_DS).NUMPY() #OBTIENE EL NÚMERO DE FILAS   DEL DATASET DE
IMÁGENES = CARDINALIDAD DEL DATASET.(CADA FILA TIENE UNA IMAGEN Y SU LABEL)
PRINT('TAMAÑO DEL DATASET (IMÁGENES Y ETIQUETAS):',DS_SIZE)
TRAIN_SIZE = INT(DS_SIZE * TRAIN_SIZE)
VAL_SIZE = INT(DS_SIZE * VAL_SIZE)
TEST_SIZE = INT(DS_SIZE * TEST_SIZE)
PRINT("""TAMAÑO ESPERADO ENTRENAMIENTO: {}
TAMAÑO ESPERADO VALIDACIÓN: {}
TAMAÑO ESPERADO TEST: {}""".FORMAT(TRAIN_SIZE, VAL_SIZE, TEST_SIZE))
BUFFER_SIZE =  DS_SIZE #ESTABLECEMOS EL BUFFER_SIZE AL MISMO TAMAÑO DEL DATASET PARA TENER UNA HOMOGENIZA
CIÓN ALEATORIA.
```

```
# -----CONTINUACIÓN-----

IMAGE_DS = IMAGE_DS.SHUFFLE(BUFFER_SIZE) #MEZCLAMOS EL DATASET. CON EL MÉTODO .SHUFFLE, LA CORRESPONDEN
CIA IMAGEN-ETIQUETA SE CONSERVA.
TRAIN_DS = IMAGE_DS.TAKE(TRAIN_SIZE)        # EL DATASET DE TRAINING TOMA EL 70% DE LAS PRIMERAS IMÁGENES
VAL_DS = IMAGE_DS.SKIP(TRAIN_SIZE).TAKE(VAL_SIZE)   # EL DATASET DE VALIDACIÓN TOMA EL 20% SIGUIENTE

TEST_DS = IMAGE_DS.SKIP(TRAIN_SIZE).SKIP(VAL_SIZE)  # EL DATASET DE TEST TOMA EL 10 % RESTANTE.
PRINT("""TAMAÑO REAL ENTRENAMIENTO: {}
TAMAÑO REAL VALIDACIÓN: {}
TAMAÑO REAL TEST: {}""".FORMAT(TF.DATA.EXPERIMENTAL.CARDINALITY(TRAIN_DS).NUMPY(),
                      TF.DATA.EXPERIMENTAL.CARDINALITY(VAL_DS).NUMPY(),
                      TF.DATA.EXPERIMENTAL.CARDINALITY(TEST_DS).NUMPY()))
```

- Definición del modelo.

Para el modelo utilizaremos una arquitectura basada en la red neuronal denominada U-Net. La arquitectura U-Net, Consiste en una codificación (parte izquierda) y una decodificación (parte derecha). Se realizan modificaciones con respecto a la definición de la arquitectura contenida en la publicación de sus autores (Ronneberger et al., 2015).

```
DEF DOWNSAMPLE(FILTERS, SIZE, STRIDES=(2, 2), APPLY_NORM=TRUE, NAME=NONE):
    ''' FUNCIÓN QUE DEFINE LAS CARACTERÍSTICAS DE LAS CAPAS A UTILIZAR EN LA CODIFICACION.
        SECUENCIA DE CAPAS: CONV2D => LEAKYRELU => CONV2D => BATCHNORM => LEAKYRELU
        INCLUYE INICIALIZADOR TF.RANDOM_NORMAL_INITIALIZER(0., 0.02)
        ARGS: -FILTERS: NÚMERO DE FILTROS A UTILIZAR EN LA RED CONVOLUCIONAL
              -SIZE: TAMAÑO DEL FILTRO A USAR EN LA RED CONVOLUCIONAL
              -STRIDES: PASO DEL FILTRO (DEFINIDA POR DEFECTO A 2 EN X , 2 EN Y PARA LA
                       CAPA QUE RECIBE LOS DATOS DE LA CAPA ANTERIOR)
              -APPLY_NORM: SI SE APLICA LA CAPA DE NORMALIZACIÓN. POR DEFECTO = TRUE
              -NAME: NOMBRE DE LA ETAPA. POR DEFECTO NINGUNO.
    '''
    INITIALIZER = TF.RANDOM_NORMAL_INITIALIZER(0., 0.02) # INICIALIZADOR DE LOS PESOS
    RESULT = TF.KERAS.SEQUENTIAL(NAME=NAME) # MÉTODO SECUENCIAL PARA AÑADIR LAS CAPAS
    RESULT.ADD(
      TF.KERAS.LAYERS.CONV2D( # CAPA CONVOLUCIONAL 2D
          FILTERS, SIZE,  # NÚMERO DE FILTROS Y TAMAÑO SEGÚN ARGUMENTO
          PADDING='SAME', # RELLENO CON 0
          KERNEL_INITIALIZER=INITIALIZER, # INICIALIZADOR DE LOS PESOS
          USE_BIAS=FALSE, # NO UTILIZACIÓN DE BIAS
          STRIDES=STRIDES # PASO SEGÚN ARGUMENTO
      ))
    RESULT.ADD(TF.KERAS.LAYERS.LEAKYRELU()) # CAPA CON FUNCIÓN DE ACTIVACIÓN RELU
    RESULT.ADD(
      TF.KERAS.LAYERS.CONV2D( # CAPA CONVOLUCIONAL 2D, EL PASO POR DEFECTO ES (1,1) => NO SE REDUCE LA
RESOLUCIÓN DEL MAPA DE CARACTERÍSTICAS.
          FILTERS, SIZE, # NÚMERO DE FILTROS Y TAMAÑO SEGÚN ARGUMENTO
          PADDING='SAME', # RELLENO CON 0
          KERNEL_INITIALIZER=INITIALIZER, # INICIALIZADOR DE LOS PESOS
          USE_BIAS=FALSE, # NO UTILIZACIÓN DE BIAS
      ))
    IF APPLY_NORM:
        RESULT.ADD(TF.KERAS.LAYERS.BATCHNORMALIZATION()) # CAPA DE NORMALIZACIÓN


    RESULT.ADD(TF.KERAS.LAYERS.LEAKYRELU()) # CAPA CON FUNCIÓN DE ACTIVACIÓN RELU
    RETURN RESULT
```

```
# -----CONTINUACIÓN-----

def upsample(filters, size, apply_dropout=False, name=None):
    '''Función que define las características de las capas a utilizar en la CODIFICACION.
       Secuencia de capas: Conv2DTranspose => Batchnorm => ReLU => Conv2D => Dropout => Batchnorm  =>
Relu
       Incluye inicializador tf.random_normal_initializer(0., 0.02)
       Args: -filters: Número de filtros a utilizar en la red convolucional
             -size: Tamaño del filtro a usar en la red convolucional
             -apply_dropout: Si se aplica la capa de dropout. Por defecto = False
             -name: Nombre para cada una de las etapas. Por defecto = None
    '''
    initializer = tf.random_normal_initializer(0., 0.02) # Inicializador de los pesos

    result = tf.keras.Sequential(name=name) # Método secuencial para añadir las capas

    result.add(
      tf.keras.layers.Conv2DTranspose( # Capa convolucional 2D traspuesta (de-convolucional)
                                      filters, size,  # Número de filtros y tamaño según
                                                      argumento
                                      strides=(2, 2), # Paso 2 en horizontal y 2 vertical
                                      padding='same', # Relleno con 0
                                      kernel_initializer=initializer, # Inicializador de los pesos
                                      use_bias=False)) # No utilización de BIAS
    result.add(tf.keras.layers.BatchNormalization()) # Capa de normalización

    result.add(tf.keras.layers.ReLU()) # Capa con función de activación ReLU

    result.add(
      tf.keras.layers.Conv2D( # Capa convolucional 2D por defecto el paso es (1,1),
                              # no reduce resolución del mapa de características
                             filters, size, # Número de filtros y tamaño según argumento
                             padding='same', # Relleno con 0
                             kernel_initializer=initializer, # Inicializador de los pesos
                             use_bias=False,)) # No utilización de BIAS

    if apply_dropout:
        result.add(tf.keras.layers.Dropout(0.5)) # Capa de dropout configurada al 50%

    result.add(tf.keras.layers.BatchNormalization()) # Capa de normalización

    result.add(tf.keras.layers.ReLU()) # Capa con función de activación ReLU

    return result
```

Ensamblamos el modelo usando las funciones anteriores. Localizamos las secuencias de codificación y decodificación en una lista y ensamblamos el modelo iterando sobre esta lista.

```
filtros_iniciales = 64 # Fijo el número de filtros que inicial la arquitectura U-Net.
#Definición de las capas de cada etapa de CODIFICACION: tamaño del filtro 4x4, Sin capa de normalización
 en la primera etapa.
down_stack = [
    downsample(filtros_iniciales    , 4, apply_norm=False, name='Codificador_1'),
    downsample(filtros_iniciales * 2, 4, name='Codificador_2'),
    downsample(filtros_iniciales * 4, 4, name='Codificador_3'),
    downsample(filtros_iniciales * 8, 4, name='Codificador_4'),
    downsample(filtros_iniciales * 16, 4, name='Codificador_5'),]
```

```
# -----CONTINUACIÓN-----

#Definición de las capas de cada etapa de DECODIFICACION: tamaño del filtro 4x4, con  Capa dropout en todas ellas.
up_stack = [
    upsample(filtros_iniciales * 8, 4, apply_dropout=True,name='Decodificador_1'),
    upsample(filtros_iniciales * 4, 4, apply_dropout=True,name='Decodificador_2'),
    upsample(filtros_iniciales * 2, 4, apply_dropout=True,name='Decodificador_3'),
    upsample(filtros_iniciales    , 4, apply_dropout=True,name='Decodificador_4'),]


#Capas complementarias a codificación/decodificación
initializer = tf.random_normal_initializer(0., 0.02) # inicializador de los pesos.


input = tf.keras.layers.Input(shape=img_shape + [input_channels]) #Capa de entrada
skips = []
x = input
#print('x=input=>',x)###


for down in down_stack: # Añade las etapas del lado de compresión en la lista "skips"

    print(x) ###
    x = down(x)
    print(x) ###
    skips.append(x)
    print(skips) ###


skips = reversed(skips[:-1]) # Invierto los skips para poder concatenar.


#print('reverse_skips =>',skips)
#print('x_prior_concatenate',x)
#print('x_concatenate', x)


for up, skip in zip(up_stack, skips): #Concatena cada capa de codificación con su decodificación
    x = up(x)
    x = tf.keras.layers.Concatenate(axis=3)([x, skip]) # Concatenación de los mapas de características


last = Conv2DTranspose( # Capa de convolucional que devuelve el mapa de
                        # características a la resolución original
                        32, # 32 filtros
                        4,  # Tamaño del filtro 4x4
                        strides=2, # Paso 2 en horizontal y 2 en vertical,
                        activation='relu', # Función de activación ReLU (puesto que no hay
                                           # capa ReLU después)
                        padding='same', # Relleno con 0
                        kernel_initializer='he_normal')(x) # Inicializador  he_normal que inicializa los pesos a una normal centrada en 0 con  stddev =  sqrt(2 / número de unidades de entrada del tensor).Los Bias  no están anulados en estas capas.


# última capa convolucional de 1 dimensión que distribuya las salidas (pixeles) en función de cada clase.
last = Conv2D(output_channels, # Número de filtros = número de clases
              1, # Tamaño del filtro = 1 un valor de salida por pixel
              strides=1, # Paso = 1 cada pixel se mapea
              padding='same')(last) # Relleno con 0 y Marcador de última capa


x = last

model = tf.keras.Model(inputs=input, outputs=x) # Construcción del modelo

model.summary() # Muestra el modelo
```

- Se visualiza el modelo de forma esquemática

```
tf.keras.utils.plot_model(model)
```

- Utilización del coeficiente Dice como métrica.

```
def DiceMetric(y_true, y_pred, smooth=1e-5): # Función que define la métrica del coeficiente Dice
    classes = tf.shape(y_pred)[-1] # Calculamos el número de clases como el valor de la 4 dimensión de la
forma de la predicción.


    y_true = tf.cast(y_true, tf.int32)[..., -1] # Extraemos los valores de la etiqueta. Aseguramos forma
to int32
    y_true = tf.cast(tf.one_hot(y_true, classes), tf.float32) # Aplicamos one-hot encoding. Aseguramos
formato float32
    y_pred = tf.cast(tf.argmax(y_pred, axis=-1), tf.int32) # Extraemos los valores de la
salida de la red neuronal (logits). Aseguramos formato int32
    y_pred = tf.one_hot(y_pred, classes) # Aplicamos codificación one-hot
# ----continuación----.
    intersection = tf.reduce_sum(y_true * y_pred, axis=[1, 2, 0]) # Reducimos las dimensiones sumando los
 elementos resultantes de la multiplicación de cada dimensión del tensor para calcular la intersección del
real y la predicción.
    intersection = 2 * tf.cast(intersection, tf.float32) + smooth # Multiplicamos x2 las intersecciones de
 los mismos. Aseguramos formato float32


#Para el denominador Sumamos por reducción y aseguramos formato float32.

    denominator = tf.cast(
      tf.reduce_sum(y_true, axis=[1, 2, 0]) + tf.reduce_sum(y_pred, axis=[1, 2, 0]),
      tf.float32
    ) + smooth # Para evitar dividir por 0


    dice = intersection / denominator #Cálculo del coeficiente Dice


    return tf.reduce_mean(dice, axis=-1) # Calculamos la media de cada pareja real-predicción
```

- Utilización del coeficiente IoU (intersection over the union) como métrica.

El coeficiente (IoU), también conocido como Jaccard index, es una métrica utilizada comúnmente para evaluar la precisión de la detección o segmentación de objetos en tareas de visión por computadora.

```
def IoU(y_true, y_pred): # Función que defina la métrica del coeficiente  Insertion Over de Union IoU
    classes = y_pred.shape[-1] # Calculamos el número de clases como el valor de la 4  dimensión de la
forma de la predicción.
    y_true = tf.cast(y_true, tf.int32)[..., -1] # Extraemos los valores de la etiqueta.  Aseguramos
formato int32
    y_true = tf.one_hot(y_true, classes) # Aplicamos one-hot encoding. Aseguramos formato  float32
    y_pred = tf.keras.activations.softmax(y_pred, axis=-1)
    y_pred = tf.cast(y_pred > 0.5, tf.float32)
```

```
# -----CONTINUACIÓN-----

    intersection = tf.reduce_sum(y_true * y_pred, axis=[1, 2, 0]) # Reducimos las dimensiones sumando los
 elementos resultantes de la multiplicación de cada dimensión del tensor para calcular la intersección del rea
l y la predicción.
    # La unión entre la real y la predicción la calculamos como la suma de las dimensiones reducidas de ambos
 tensores menos la intersección de los mismos.
    union = tf.reduce_sum(y_true, axis=[1, 2, 0]) + tf.reduce_sum(y_pred, axis=[1, 2, 0]) - intersect
ion
    iou = intersection / union # Calculamos el cociente según la fórmula.


    return tf.experimental.numpy.nanmean(iou, axis=-1) # Calculamos la media de cada pareja real-predicción
 descartando los valores "nan" (división por 0)
```

- Compilar el modelo:

```
BATCH_SIZE = 16     # Definición del batch de entrenamiento (se refiere al número de ejemplos de entrenam
iento que se procesan simultáneamente durante una iteración de entrenamiento)


model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3), #Optimizador Adam con aprendizaje 0.001
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), #Función de pérdida:
 SparseCategoricalCrossentropy con entrada de Logits
    metrics=['accuracy', DiceMetric, IoU] # Métricas: Accuracy, Índice Dice e Índice IoU)
)


#dataset.cache mantiene las imágenes en la memoria después de que se cargan fuera del disco durante la primer
a epochs. Esto asegura que el dataset no se convierta en un cuello de botella mientras entrena su modelo. El
conjunto de datos será demasiado grande para caber en la memoria, así que este método para creará un caché en
disco de alto rendimiento. Dataset.prefetch superpone el preprocesamiento de datos y la ejecución del modelo
durante el entrenamiento
train_ds_b = train_ds.cache().batch(BATCH_SIZE).prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

val_ds_b = val_ds.cache().batch(BATCH_SIZE).prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

- Definición de callbaks y entrenamiento del modelo.

```
EPOCHS = 100  # Definición del número de epoch a aplicar en el entrenamiento (número de veces que se pasar
á el dataset completo.


es_callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5) # Definición del callback
early stop monitorizando el val_loss. Parará cuando el val_loss no mejore en 5 epoch consecutivos.


checkpoint_callback = tf.keras.callbacks.ModelCheckpoint( # Definición del callback check point para     sa
lvar el modelo
    "./unet_A2D2_ep",  # Carpetas en las que se salvará el modelo.
    monitor="val_loss", # Métrica a monitorizar = val_loss
    save_best_only=True, # Sólo salva el modelo si es mejor que el anterior epoch
    save_weights_only=False, # Salva el modelo completo
    save_freq="epoch", # El modelo se salva en cada epoch
    verbose= 1 # Cada vez que el callback se ejecute se lanza un mensaje.
)
```

```
# -----CONTINUACIÓN-----

TIME_INIT = TIME() # TRAZA DE TIEMPO PARA CALCULAR EL TIEMPO DE ENTRENAMIENTO


ABSL.LOGGING.SET_VERBOSITY(ABSL.LOGGING.ERROR) # CONFIGURAMOS PARA QUE NO ARROJEN WARNINGS POR ABSL.
WARNINGS.FILTERWARNINGS("IGNORE") # CONFIGURAMOS PARA QUE NO SE MANIFIESTEN LOS WARNINGS.
MODEL_HISTORY = MODEL.FIT( # MÉTODO FIT PARA ENTRENAR EL MODELO.
    TRAIN_DS_B,           # SET DE DATOS DE TRAINING
    VALIDATION_DATA = VAL_DS_B, # SET DE DATOS DE VALIDACIÓN
    EPOCHS=EPOCHS,  # EPOCH A UTILIZAR
    CALLBACKS=[ES_CALLBACK, CHECKPOINT_CALLBACK], # CALLBAKS A UTILIZAR.
    VERBOSE = 2 # EL MÉTODO DEVUELVE UNA LÍNEA DE RESULTADO DE MÉTRICAS POR CADA EPOCH)

TIME_FIN = TIME() # TRAZA DE TIEMPO PARA CALCULAR EL TIEMPO DE ENTRENAMIENTO
```

- Resultados del entrenamiento

```
TIME_ENTRENAMIENTO = (TIME_FIN-TIME_INIT) # CALCULAMOS EL TIEMPO DE ENTRENAMIENTO Y LO ARROJAMOS EN HORAS/M
INUTOS/SEGUNDOS
HORAS = TIME_ENTRENAMIENTO  // 3600
MINUTOS = (TIME_ENTRENAMIENTO  % 3600) // 60
SEGUNDOS = INT((TIME_ENTRENAMIENTO  % 3600) % 60)
PRINT('TIEMPO DE ENTRENAMIENTO:', HORAS, 'HORAS', MINUTOS, 'MINUTOS', SEGUNDOS, 'SEGUNDOS')


MEJOR_EPOCH =NP.ARGMIN(MODEL_HISTORY.HISTORY["VAL_LOSS"]) #LOCALIZAMOS EL MEJOR EPOCH EN LA HISTORY DEL
ENTRENAMIENTO COMO AQUEL CON MENOR VAL_LOSS


PRINT('MEJOR EPOCH: ',MEJOR_EPOCH + 1) # DADO QUE EL ÍNDICE EL ARRAY DE NUMPY EMPIEZA POR 0 EL NÚMERO DE E
POCH SERÁ ES EL DEL ÍNDICE +1


_STR = "ESTADÍSTICAS DEL MEJOR EPOCH:\N" # VISUALIZO LAS ESTADÍSTICAS DEL MEJOR EPOCH.


FOR METRIC_NAME, METRIC_HISTORY IN MODEL_HISTORY.HISTORY.ITEMS():
    _STR += F"{METRIC_NAME} - {METRIC_HISTORY[MEJOR_EPOCH]:.3F}   "


PRINT(_STR)


_STR2="\N LISTADO DE LAS ESTADÍSTICAS DE TODOS LOS EPOCH" #VISUALIZO LAS ESTADÍSTICAS DE TODOS LOS EPOCH
FOR I IN RANGE(EPOCHS):
    FOR METRIC_NAME, METRIC_HISTORY IN MODEL_HISTORY.HISTORY.ITEMS():
            IF METRIC_NAME == "LOSS":
                _STR2 += F" \N EPOCH {I+1}: "
            _STR2 += F"{METRIC_NAME} - {METRIC_HISTORY[I]:.3F} "


PRINT(_STR2)
```

- Evaluación del modelo

```
PRINT('MEJOR EPOCH: ',MEJOR_EPOCH+1)


PATH_MEJOR_EPOCH ='./UNET_A2D2_EP' # TOMO EL MODELO DEL MEJOR EPOCH


MODEL = TF.KERAS.MODELS.LOAD_MODEL(STR(PATH_MEJOR_EPOCH), # CARGO EL MODELO DEL MEJOR EPOCH
                                   CUSTOM_OBJECTS={"DICEMETRIC": DICEMETRIC, "IOU": IOU}, # HE DE PASAR
LE LAS MÉTRICAS DE MEDIDA QUE HEMOS AÑADIDO AL MODELO
                                   COMPILE=TRUE) # CONFIGURO QUE COMPILE EL MODELO CUANDO LO CARGA
MODEL.SUMMARY() # MUESTRA EL MODELO
```

```
TEST_DS_B = TEST_DS.BATCH(16).PREFETCH(BUFFER_SIZE=TF.DATA.EXPERIMENTAL.AUTOTUNE) # CONSTRUCCIÓN DEL
FLUJO DE DATOS DE TEST PARA LA EVALUACIÓN.
EVALUACION= MODEL.EVALUATE(TEST_DS_B) # EVALUACIÓN DEL MODELO


PRINT('\N',_STR) # VISUALIZO LOS RESULTADOS DEL ENTRENAMIENTO PARA COMPARAR
PRINT('-----------------------------------------')


METRICA_TEST = ['TEST_LOSS:', 'TEST_ACCURACY', 'TEST_DICEMETRIC', 'TEST_IOU']
_STR_TEST = "ESTADÍSTICAS EVALUACION DE TEST:\N"# DEFINO LA LISTA DE MÉTRICAS A EVALUAR


FOR I IN RANGE(LEN(METRICA_TEST)): # ITERO LA LISTA DE MÉTRICAS EN LOS RESULTADOS DE LA EVALUACIÓN
      _STR_TEST += F"{METRICA_TEST[I]} - {EVALUACION[I]:.3F}   "


PRINT(_STR_TEST) # VISUALIZO LOS RESULTADOS DE LA EVALUACIÓN
```

- Ejecución de Predicciones:

```
DEF CREAR_LABEL_PRED(PRED):
    """
    FUNCIÓN QUE CREA UNA IMAGEN ETIQUETA A PARTIR DE UN TENSOR DE TENSORFLOW.
    ARGS:
         PRED (NUMPY.NDARRAY O TENSORFLOW.TENSOR): UNA TENSOR DE PREDICCIONES DE 3 0 4
                                                   DIMENSIONES.
    RETURNS:
         NUMPY.NDARRAY: UNA ETIQUETA EN FORMATO NP.NDARRAY CON LAS MISMAS DIMENSIONES QUE EL
                    TENSOR DE PREDICCIONES.
    """
    LABEL = TF.ARGMAX(PRED, AXIS=-1) # TOMAMOS EL VALOR DE CADA PIXEL DE LA ETIQUETA COMO
 EL MÁXIMO DE LA ÚLTIMA DIMENSIÓN DEL TENSOR.
    LABEL = NP.ARRAY(LABEL[..., TF.NEWAXIS]) #CONSTRUIMOS EL NP.NDARRAY CON ESTOS VALORES
 MÁXIMOS DE CADA PIXEL.
    RETURN LABEL

DEF EXTRAE_PREDICCIONES(MODEL, IMAGES):
    """
    FUNCIÓN QUE GENERA PREDICCIONES A PARTIR DE UNA LISTA DE IMÁGENES USANDO UN MODELO.
    ARGS:
       MODEL (TENSORFLOW.KERAS.MODEL): MODELO UTILIZADO PARA PREDECIR LAS IMÁGENES.
         IMAGES (LISTA DE CADENAS): LISTA DE IMÁGENES PARA GENERAR LAS PREDICCIONES.
    RETURNS:
         LISTA EN FORMATO NUMPY.NDARRAY: LISTA DE LAS PREDICCIONES CORRESPONDIENTES  A LAS
        IMÁGENES.
    """
    PRED = MODEL.PREDICT(IMAGES) # LANZAMOS EL MODELO PARA PREDECIR LAS IMÁGENES QUE SON
DEVUELTAS COMO UN TENSOR DE TENSORFLOW
    PRED = [CREAR_LABEL_PRED(I) FOR I IN PRED] #CONVERTIMOS LAS PREDICCIONES (TENSORES) EN
ETIQUETAS CON LA FUNCIÓN CREAR_LABEL_PRED
    RETURN PRED
```

```
TEST_IMAGENES = []
TEST_TRUE_LABELS = []


FOR IM, LABEL IN TEST_DS.TAKE(5): # CREAMOS LA LISTA DE IMÁGENES A PREDECIR Y SUS
CORRESPONDIENTES ETIQUETAS DE TEST DEL DATASET TEXT_DS
    TEST_IMAGENES.APPEND(IM.NUMPY())
    TEST_TRUE_LABELS.APPEND(LABEL.NUMPY())


PREDICCIONES = EXTRAE_PREDICCIONES(MODEL, NP.ARRAY(TEST_IMAGENES)) # LANZAMOS LA FUNCIÓN
PARA EXTRAER LAS PREDICCIONES CON TODAS LAS IMÁGENES DENTRO DE UN ARRAY DE NUMPY.
```

```
FOR IM, TRUE, PRED IN ZIP(TEST_IMAGENES, TEST_TRUE_LABELS, PREDICCIONES):
    DISPLAY_IMAGES([IM, TRUE, PRED], SIZE=(10, 4), PLOT_TITLE="PREDICCIONES DE EJEMPLO", TITLES=("IMAGE
N", "ETIQUETA REAL", "ETIQUETA PREDICHA"))
```

- Salvando el modelo y los pesos.

```
# SALVAREMOS EL MODELO PARA DESPUÉS PODER ENTRENARLO CON MÁS DATOS O BIEN UTILIZARLO PARA PREDECIR CONFIGURO
QUE COMPILE EL MODELO CUANDO LO CARGA. SALVAMOS EL MODELO INCLUYENDO SU ARQUITECTURA Y CONFIGURACIÓN, LOS
VALORES DE LOS PESOS, LA INFORMACIÓN DE COMPILACIÓN DEL MODELO, Y EL ESTADO DEL OPTIMIZADOR. ESTO QUE PERMITI
RÁ EMPEZAR EL TRAINING DESDE DONDE LO DEJAMOS.

FILEPATH = "C:\\CAMERA_LIDAR_SEMANTIC\MODEL_SAVED" # DEFINIMOS LA RUTA DONDE SALVAREMOS EL MODELO.
MODEL.SAVE(FILEPATH) # SALVAMOS EL MODELO
```